escale Semiconductor
Application Note

Document Number: AN3795
Rev. 0, 04/2009

Using the CRC Module on the Flexis AC Family

by: Gordon Borland
MSG Applications Engineering. East Kilbride, Scotland

1 Introduction

There are increasing requirements for in-application tests and checks upon embedded control software, to ensure it will operate without a malfunction that may cause a hazard. Peer reviews and following best programming practice can help to ensure that the fundamental software operation is safe, but it cannot protect an application against a corruption of it's software, for example an EMC event.

The incorporation of a cyclic redundancy check module within the Flexis AC family provides a mechanism for users to verify, that their program data has not been corrupted.

1.1 Abstract

This document is written to show how the cyclic redundancy check module of the Flexis AC family can be used to check in-application software for corruption and verify data integrity.

Contents

1.2 Objective

This application note will explain the fundamental principles behind cyclic redundancy checking, it's implementation in the MC9S08AC128, and how it can be used to meet the embedded control software safety requirements by verifying the software integrity. Example code is also provided.

2 The Flexis AC CRC Module

2.1 Advantages of the Flexis AC CRC Module

Whilst it is possible to implement a cyclic redundancy check using software routines only, the Flexis AC CRC offers several advantages over a software-only solution.

Software CRC solutions will require certification to show that they fulfill the requirements of the protocol being followed. The Flexis AC CRC module already has the compliancy to CRC16-CCITT built in using the $x^{16} + x^{12} + x^5 + 1$ polynomial and it's implementation meets the ITU-T (International Telecommunication Union Telecommunication Standardization Sector) V.41 recommendation.

The compliance to CRC16-CCITT means that the Flexis AC CRC module can detect all single, double, odd, and most multi-bit errors.

A software solution will require several hundred bus cycles (~700) to process a single byte through the CRC algorithm. The Flexis CRC module can shift a byte into the CRC generator in one bus cycle resulting in a greatly increased speed of calculation.

For example measurements on the MC9S08AC128 show that a software solution compliant with CRC16-CCITT running at 20 MHz bus speed takes approximately 6.7 seconds to perform a cyclic redundancy check on 128 K of flash memory. By comparison the same check requires only 170 ms using the CRC module, which is significantly faster.

The increased speed of cyclic redundancy calculation of the CRC module over software means that the Flexis AC CRC module is ideally suited to perform cyclic redundancy checking on data transmissions with minimal additional overhead, when compared to a software solution.

The Flexis AC CRC modules' high-speed CRC calculation means that a self check of the application software at startup can be performed without significantly adding to the initialization time.

Further, the user can take the advantage of the fast cyclic redundancy calculation to implement a self test of the software at regular intervals, whilst the application is running without adversely impacting the performance.

In addition to the advantages offered over a software CRC, the Flexis AC CRC module is capable of handling programmable seed values and optional augmentation. This flexibility allows the Flexis AC CRC module to comply with slightly different ITU-T protocols based upon CRC16-CCITT without any major code changes.

2.2 Cyclic Redundancy Checking

Error detection is important, whenever there is a chance of data getting corrupted. Whether it’s a piece of stored code or a data transmission, you can add a piece of redundant information to validate the data and protect it against corruption.

The simplest example is a parity bit. Typically one parity bit is used for each byte. For odd parity, the number of non-zero bits in a byte are counted. If the number is even the parity bit is set, otherwise it is cleared. If one of the nine bits is flipped, the sum will be even and an error will be detected. Of course, if two bits are flipped, the error will remain undetected.

Cyclic redundancy checking is a much more robust error-checking algorithm, which is commonly used to detect errors either in data transmission or data storage.

Whilst it is not the intention of this document to go into detail regarding the mathematics behind the concept of cyclic redundancy, it is worthwhile to explain a few basic principles. Computation of a cyclic redundancy check is derived from the mathematics of:

- polynomial division
- modulo two arithmetic

In practice it resembles a long division of the binary message string by a generator polynomial string, except that exclusive OR operations replace the subtractions.

2.2.1 Modulo Two Arithmetic

Modulo two arithmetic is simple single-bit binary arithmetic with all “carries or borrows” ignored. Each digit is considered independently. For example, we can add two binary numbers X and Y as follows:

$$
\begin{array}{rl}
 & 10101001\ (X) \\
+ & 00111010\ (Y) \\
\hline
 & 10010011\ (Z)
\end{array}
$$

From this example the modulo two addition is equivalent to an exclusive OR operation. What is less obvious is that modulo two subtraction gives the same results as an addition. From the previous example let’s add X and Z:

$$
\begin{array}{rl}
 & 10101001\ (X) \\
+ & 10010011\ (Z) \\
\hline
 & 00111010\ (Y)
\end{array}
$$

In our previous example we have seen how $X + Y = Z$ therefore $Y = Z – X$, but the example above shows that $Z+X = Y$ also, hence modulo two addition is equivalent to modulo two subtraction, and can be performed using an exclusive OR operation.

2.2.2 Polynomial Division

In integer division dividing A by B will result in a quotient Q, and a remainder R.

A / B = Q with remainder R where R < B. *Eqn. 1*

Polynomial division is similar except that when A and B are polynomials, the remainder is a polynomial, whose degree is less than B.

Thus if the polynomial represented by $A = 2x^3 + 5x^2 + 9x + 5$ and $B = x^2 + 2x + 3$ (degree 2), then

A / B = Q with remainder R *Eqn. 2*

can be represented as:

$$(2x^3 + 5x^2 + 9x + 5) \ / \ (x^2 + 2x + 3) = (2x+1) \text{ with remainder } x + 2 \qquad \textit{Eqn. 3}$$

Where $Q = 2x + 1$ (degree 2) and $R = x + 2$ (degree 1).

If a change is made to polynomial A, then there will be a resultant change in the remainder R.

Let $A = 2x^3 + 5x^2 + 9x + \underline{\mathbf{4}}$

A / B = Q with remainder R *Eqn. 4*

Is now represented as:

$$(2x^3 + 5x^2 + 9x + 4) \ / \ (x^2 + 2x + 3) = (2x+1) \text{ with remainder } \underline{x + 1} \qquad \textit{Eqn. 5}$$

R is now x + 1.

The key point here is that any change to the polynomial A causes a change to the remainder R. This behaviour forms the basis of the cyclic redundancy checking.

If we consider a polynomial, whose coefficients are zeros and ones (modulo two), this polynomial can be easily represented by its coefficients as binary powers of two.

For example

$$1x^7 + 1x^6 + 0x^5 + 1x^4 + 0x^3 + 1x^2 + 1x + 1 = 11010111 \qquad \textit{Eqn. 6}$$

Thus it is apparent that a polynomial can be represented as a series of bits, and conversely any binary string of bits, such as a data message or software code, can be represented as a polynomial.

In terms of cyclic redundancy calculations, the polynomial A would be the binary message string or data and polynomial B would be the generator polynomial. The remainder R would be the cyclic redundancy checksum. If the data changed or became corrupt, then a different remainder would be calculated.

2.2.3 Basic CRC Calculation

Now we can begin to perform a basic cyclic redundancy checksum calculation using polynomial division and modulo two arithmetic. As described previously, this is basically the division of the data polynomial by a generator polynomial. This basic calculation will use a simple piece of data such as 0xD6B5 (in binary 1101011010110101) and use the polynomial $x^3 + x^2 + 1$ as the generator polynomial.

The data polynomial is divided by the generator polynomial $x^3 + x^2 + 1$ (or 1101), using an exclusive OR to replace the subtraction (modulo two arithmetic), and we record the remainder, which is the cyclic redundancy checksum.

```
        1000010000100 (Q)
1101 ) 1101011010110101 (A)
(B)    1101xxxxx
            001101
              1101xxxxx
                 001101
                   1101xx
                      001  (R, the CRC)
```

Thus in this example the remainder (cyclic redundancy checksum) is 001. If any of the bits in the dividend (binary message) were to change, then a different cyclic redundancy checksum would be calculated. Using the previous data, let’s assume that bit one of the data has been corrupted. The data will now read 0xD6B7 instead of 0xD6B5.

```
        1000010000100  (Q)
1101 ) 1101011010110111  (A)
(B)    1101xxxxx
            001101
              1101xxxxx
                 001101
                   1101xx
                      011  (R, the CRC)
```

Thus to determine, whether any corruption of data has occurred, it is a simple matter to repeat the division using the same generator polynomial, and compare the remainder (cyclic redundancy checksum) produced against the one calculated previously. If they are the same, then no corruption of the data has occurred and the integrity of data is intact.

2.2.4 Null Cyclic Redundancy Checksum

It has already been demonstrated, how the integrity of data can be verified by calculating the cyclic redundancy checksum and comparing it to one calculated previously. An alternate method is to add the cyclic redundancy checksum to the data and perform the cyclic redundancy checksum calculation. If the data is not corrupted, then the result will be null (0x00). In the earlier analogy with integer division, dividing A by B will result in a quotient Q, and a remainder R.

For example:

A / B = Q with remainder R *Eqn. 7*

We defined A to be the data polynomial and B the generator polynomial.

If

A / B = Q with remainder R *Eqn. 8*

then

(A – R) / B = Q with no remainder *Eqn. 9*

No remainder = 0x00 = Null CRC result. *Eqn. 10*

Let’s illustrate this using our previous example. Using 0xD6B5 = 1101011010110101 as our data polynomial, and $x^3 + x^2 + 1$ = 1101 as the generator polynomial, we calculate a checksum of 001.

```
        ____1000010000100 (Q)
1101 ) 1101011010110101 (A)
(B)     1101xxxxx
           001101
             1101xxxxx
                001101
                  1101xx
                     001  (R)
```

Now let’s add the checksum to the data polynomial using modulo two arithmetic. Remember that in modulo two arithmetic, addition is the same as subtraction.

```
Data polynomial = 0xD6B5 = 1101011010110101
  + CRC                    _____________001
                           1101011010110100 = 0xD6B4
```

Calculating the cyclic redundancy checksum for the data + checksum we get:

```
        ____1000010000100 (Q)
1101 ) 1101011010110100 (A-R)
(B)     1101xxxxx
           001101
             1101xxxxx
                001101
                  1101xx
                     000  NULL CRC
```

As shown the result is 0x00 or null.

2.2.5 Augmentation

Augmentation is a technique used to produce a null CRC result, whilst preserving both the original data and the CRC checksum.

No augmentation is required with the Flexis AC Family CRC. A null CRC result can be obtained by writing the CRC result to the CRC module after the data. Augmentation is mentioned only to explain the concept and assist users of other cyclic redundancy implementations, using augmentation migrate to the Flexis AC Family CRC.

In a communications system using cyclic redundancy checking, it would be desirable to obtain a null CRC result for each transimission, as the simplified verification will help to speed up the data handling.

The previous section demonstrated, how to produce a null CRC result by adding the cyclic redundancy checksum (0b001) to the data (0xD6B5), and calculating the CRC on the new data (0xD6B4). Whilst this simplifies the verification, it has the unfortunate side effect of changing the data. Any node receiving 0xD6B4 will be able to verify that no corruption has occurred, but will be unable to extract the original data, because the checksum (0b001) is not known. This can be overcome by transmitting the checksum along with the modified data, but any data-handling advantage gained in the verfication process is offset by the additional steps needed to recover the original data.

Augmentation allows the data to be transmitted along with its checksum, and still obtain a null CRC result.

In the previous example to obtain a null CRC result, the data changed from 0xD6B5 to 0xD6B4, when the checksum was added. Augmentation avoids this by shifting the data left or augmenting it with a number of zeros, equivalent to the degree of the generator polynomial. When the CRC result for the shifted data is added, both the original data and the checksum are preserved.

In our previous example, our generator polynomial ($x^3 + x^2 + 1$ or 1101) is of degree 3, so the data (0xD6B5) is shifted to the left by three places or augmented by three zeros.

0xD6B5 = 1101011010110101 becomes 0x6B5A8 = 1101011010110101000.

Note that the original data is still present within the augmented data.

0x6B5A8 = 1101011010110101**000**

Data = D6B5   Augmentation = 000

Calculating the CRC result for the augmented data (0x6B5A8) using our generator polynomial (1101), gives a remainder of 101 (degree 2). If we add this to the augmented data, we get:

0x6B5A8 + 0b101 = 1101011010110101000 + 101

= 1101011010110101**101**

= 0x6B5AD

We know from Section 2.1, “Advantages of the Flexis AC CRC Module” that calculating the cyclic redundancy checksum for 0x6B5AD will result in a null checksum, simplifying the verification. What is less apparent is that the original data is still preserved intact.

0x6B5AD = 1101011010110101**101**

Data = D6B5   CRC = 101

The degree of the remainder or cyclic redundancy checksum is always less than the degree of the generator polynomial. By augmenting the data with a number of zeros equivalent to the degree of the generator polynomial, we ensure that the addition of the checksum does not affect the augmented data.

In any communications system using cyclic redundancy checking, the same generator polynomial will be used by both transmitting and receiving nodes to generate checksums and verify data. As the receiving node knows the degree of the generator polynomial, it is a simple task for it to verify the transmission by calculating the checksum and testing for zero, and then extract the data by discarding the last three bits.

Thus augmentation preserves the data, whilst allowing a null cyclic redundancy checksum for faster verification and data handling.

2.3 Flexis AC CRC Implementation

Although the algorithm for cyclic redundancy calculations looks complicated, it only involves shifting and exclusive OR operations. Using modulo two arithmetic, division is just a shift operation and subtraction is an exclusive OR operation.

Cyclic redundancy calculations can therefore be efficiently implemented in hardware, using a shift register modified with XOR gates. The shift register should have the same number of bits as the degree of the generator polynomial and an XOR gate at each bit, where the generator polynomial coefficient is one.

The Flexis AC family use a generator polynomial of degree 16; $x^{16} + x^{12} + x^5 + 1$. This is often represented by the hexadecimal number 0x1021, where each coefficient in the polynomial is represented by it's equivalent power of two. As the highest coefficient in a polynomial is always one, it is assumed and not recorded in the hexadecimal representation.

The generator polynomial used by the Flexis AC family is commonly known as CRC16-CCITT, and is used in communication protocols such as CDMA, Bluetooth, and V.41 to name a few.

The hardware implementation of the cyclic redundancy check using the generator polynomial is represented in the block diagram shown in Figure 1.

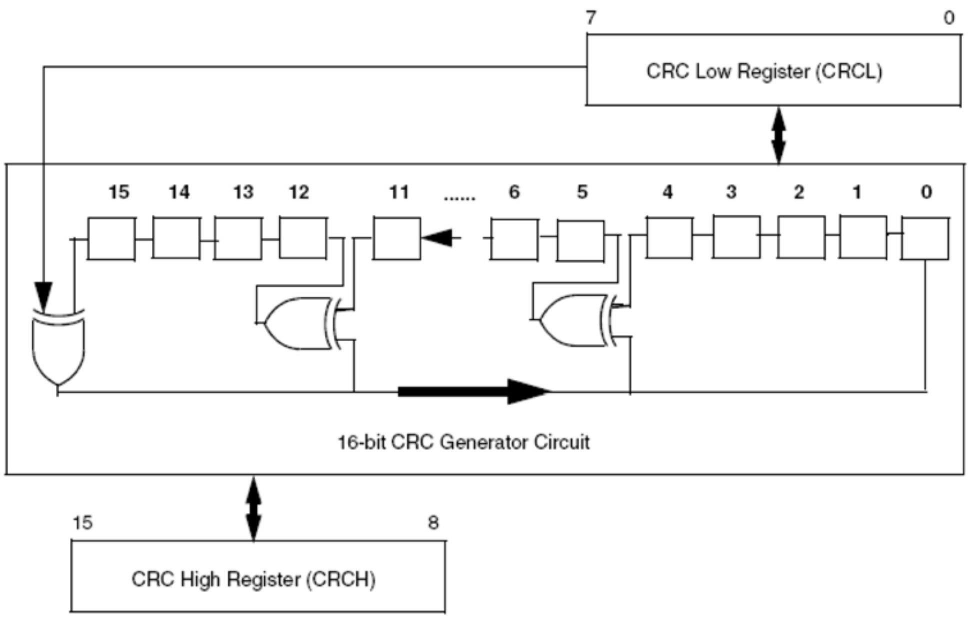

Figure 1. Cyclic Redundancy Check (CRC) Module Block Diagram

It can be seen from the block diagram that the CRC module contains only two registers, CRC High and CRC Low, which combine to create a 16-Bit CRC seed and result register (CRCH:L).

At the start of any CRC calculation, a programmable seed value is written into register CRCH:L. Using a programmable seed value allows the user to pre-configure the shift register.

The seed value is programmable to allow the CRC module to meet the seed value requirements of different communication protocols. The CRC16-CCITT standard uses a seed value of 0xFFFF.

At the end of a CRC calculation, the result is stored in CRCH:L. The procedure to calculate a CRC checksum using the CRC module is very simple.

Firstly to enable the CRC function, a seed value should be written into the CRCH:L. The seed value is placed directly into bits 15:0 of the CRC generator shift register. Writing a byte to CRCH triggers the start of the seed mechanism and loads the value written to CRCH into the upper byte of the CRC generator shift register. The CRC then requires a write to the CRCL to prime the lower byte of the CRC generator shift register, and completes the seed mechanism.

Now that the CRC seed has been loaded, the first byte of the data on which a CRC calculation is to be performed, can be passed to the CRC module. This is achieved by writing the byte of data to CRCL. This will cause the CRC module to begin the cyclic redundancy calculation by shifting the bits in CRCL into the CRC generator shift register, MSB first. This step should be repeated until all of the data bytes have been written to CRCL.

After each byte has been written, a new CRC result can be read from the CRCH:L registers one-bus cycle after the previous write to CRCL.

Once all of the data bytes have been written to CRCL, the result of the CRC calculation or cyclic redundancy checksum is available in CRCH:L.

The process can be restarted by writing a new seed value to CRCH:L, to initiate the seed mechanism and a new CRC calculation will begin.

2.4 How to Use the CRC Module

There are increasing requirements for in-application tests and checks, to ensure that embedded control software will operate reliably and safely. Examples include:

- Checking for bit errors in the program code.
- Ensuring there is no corruption of a specific string of data.

The CRC module offers a means to meet these requirements simply and efficiently, without making excessive demands on the CPU or memory resources.

Four simple steps are all that is required to use the CRC to verify the embedded software.

1. Write the seed value into CRCH:L.
2. Write the data, byte by byte to CRCL, until the end of the data to be checked.
3. Augment the data with zeros. This step is dependent upon the CRC implementation and is optional.
4. Verify the calculated cyclic redundancy checksum against a stored value.

To illustrate this, an example using the MC9S08AC128 has been included in this document (See Section Appendix A, “Cyclic Redundancy Check Software”). The MC9S08AC128 memory map has an extended address memory, which can only be accessed through a paging window, unlike other Flexis AC family MCUs, which can access all of their memory directly.

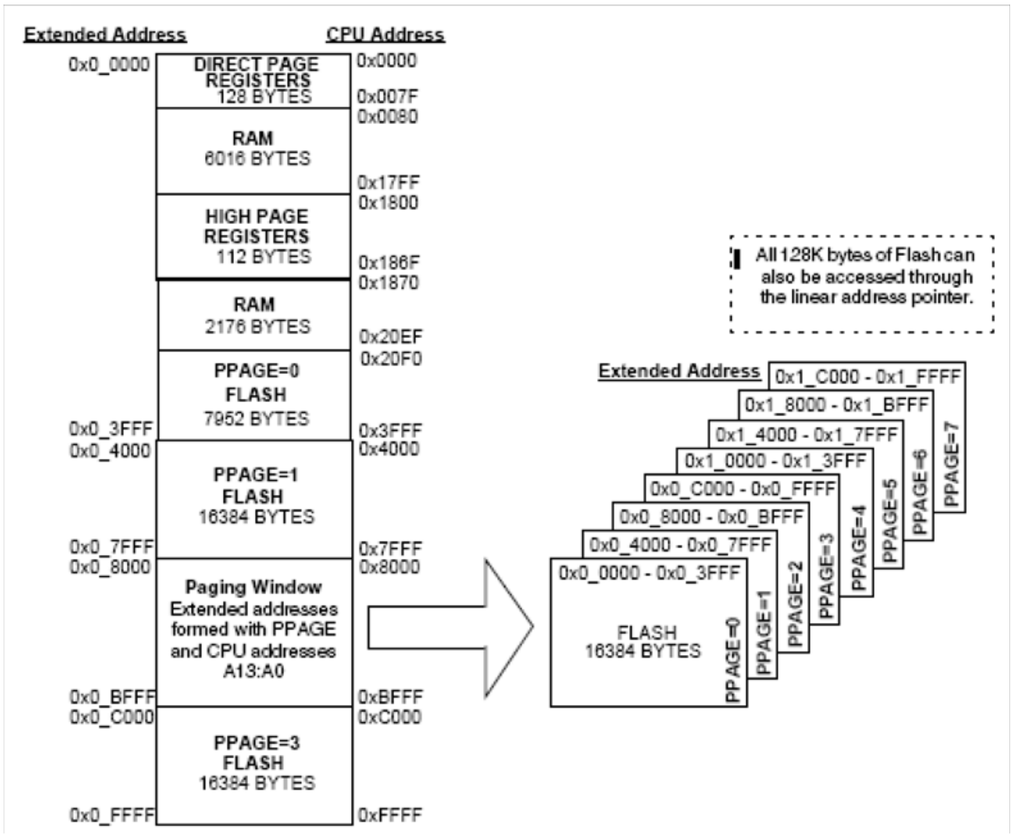

Figure 2. MC9S08AC128 Memory Map

The HCS08 core architecture limits the available CPU addressable space to 64 KB. The addressable space can be extended to 128 KB using a paging window scheme controlled by a memory management unit (MMU). A full description of the memory management unit and the paging window scheme is available in the MC9S08AC128 data sheet.

The procedures and example software provided for the MC9S08AC128 use it’s memory management unit to access the data, though they can be readily adapted for use on other Flexis AC family MCUs.

2.4.1 Seeding the CRC

The example calculates a cyclic redundancy checksum for the flash memory of the AC128. Writing a seed value is easily performed as the function init_crcCCITT shown below.

```
void init_crcCCITT(void)
         {
                  CRCH = 0xFF;       // CRC seeded 0xFFFF
                  CRCL = 0xFF;       // CRC seeded 0xFFFF
}
```

As the CRC module complies with CRC16-CCITT, the seed value used in the example is 0xFFFF, though other values can be used to allow the CRC module to be compatible with different communication protocols, which use CRC16-CCITT. Other common seed values include 0x102D and 0x0000. If greater flexibility is required, the seed value could be passed to the function as a variable.

2.4.2 Writing the Data

The functions crc_flash and update_crcCCITT demonstrate how to write the data, or in this case the embedded software contained in the non-volatile memory of the MCU to the CRC module for inclusion in the checksum calcuation.

A start and stop memory address indicating the range of the memory to be checked is passed to crc_flash. The function then accesses the data contents at each location within the specified memory range, and passes the data to the function update_crcCCITT, which in turn writes the data to the CRC module via register CRCL.

There is no difficulty using the function crc_flash should the non-volatile memory or the embedded software not be contained within a single continuous block. To perform the CRC calculation for all of the desired memory locations, call crc_flash more than once, each time specifying a different memory range, until all of the non-volatile memory or embedded software has been accounted for.

```
void crc_flash (long start_addr, long stop_addr)
{
      /* Put base address of flash into the LAP registers */
      LAP2 = (byte) (start_addr>>16);
      LAP1 = (byte) (start_addr>>8);
      LAP0 = (byte) start_addr;

      /* increment through the memory array passed */
      for ( 1; start_addr <= stop_addr; start_addr++ )
{
         update_crcCCITT ( LBP ); // send a char to the CRC
}
}  /* end of crc_flash function */

void update_crcCCITT (char ch)
{
    CRCL = ch;
}     /* end of update_crcCCITT function */
```

2.4.3 Augmentation with Zeros

It is not necessary to augment the data with zeros when using CRC16-CCITT, and when using the Flexis AC Family CRC module, though some protocols require this. One common variation of CRC-CCITT requires the message to be augmented with zeros and a SEED = 0xFFFF. The Flexis AC Family CRC module will give the same results of this alternative implementation, when SEED = 0x1D0F, and no message augmentation.

If the Flexis AC Family CRC module is being used to verify data transmissions with another CRC system, which uses augmentation, then this step may be necessary to ensure the requirements of the communicating system are met.

Shown below is the function augment_message_for_crc_16, which augments the CRC data with zeros by writing 0x00 to register CRCL twice.

```
void augment_message_for_crc_16()
{
    CRCL=0x00;
    CRCL=0x00;
}
```

2.4.4 Verifying the Calculated Checksum

Once the checksum has been calculated, it is necessary to compare it against a previously stored value.

The function crc_check provides an example of how this can be achieved. A pointer to the memory location of the stored checksum is passed to the function. The calculated value in CRCH:L is then compared against the stored value. If the two checksums are the same, no action is taken and the software is safe to continue executing. If there is a difference, the function sys_error is called, which is used to place the MCU in a safe state.

```
void crc_check(short *blog)
{
  short CRC = ((CRCH<<8)|CRCL);
      if (CRC != (short)*blog)  sys_error();
      /* else return as CRC check is OK */
}
```

When the CRC result has been calculated using a Flexis AC Family CRC module, then a null CRC result can be obtained by writing the CRC result to the CRC module after the data.

2.4.5 Practical Example

Section A.1, "Main.c" contains a complete program, which shows how the routines shown above and the four steps can be combined, and put into a program to check the embedded software before starting executing of the application software.

In the example program, a declaration at the start of the software stores the checksum (0xFBF9) at location 0xFF80 (0x0FF80 in extended addressing). On the MC9S08AC128, this is an unused vector address, which may be used by the software. Similar locations can be used for other members of the Flexis AC family.

The example program seeds the CRC, writes the contents of the flash memory into the CRC, and then compares the CRC output to the checksum stored at location 0xFF80.

The software starts by using init_crcCCITT to seed the CRC with 0xFFFF.

The flash memory contents are written to the CRC module by calling the function crc_flash twice. In the example program, the checksum is stored at location 0xFF80, which is in the middle of the flash memory address area (0x00000 to 0x1FFFF extended addressing).

The stored checksum should not be included in the CRC calculation, as this will affect the calculated result, causing it to differ from the stored checksum. This is because the calculation to determine the value of the stored checksum does not take the contents of the stored checksum memory location into account. By effectively splitting the flash into two blocks, at either side of the stored checksum location, it is possible to avoid including it in the CRC calculation.

Once the flash memory contents have been written to the CRC, the data can be augmented with zeros at this point if required. The example program uses augment_message_for_crc_16 to perfrom this task.

Finally the program calls the function crc_check, to read the calculated checksum and compare it to the stored checksum value. If they match, then the program can continue to execute the application software.

In this example the CRC check of the embedded software is performed at the start of the application program. This will execute each time the application is started or reset. Alternatively, the CRC verification software could be included in a separate function, which is called by a periodic interrupt, allowing for a regular verification of the embedded software throughout the time that the application is running.

2.4.6 Calculating a Stored Checksum Value

If the cyclic redundancy checking is being used to verify non-volatile memory, then a checksum value can be calculated prior to final programming by running the CRC verification software without including the memory location where the checksum will be stored, and noting the calculated value. This value is then stored in the skipped memory location.

It is important not to include the checksum memory location, when verifying the embedded software in an active application. Including the stored checksum memory location in the CRC calculation will affect the calculated result, causing it to differ from the stored checksum. This is because the original calculation to determine the value of the stored checksum did not take the contents of its memory location into account.

This has the advantage that the software used by the application to verify it's embedded software can also be used to determine the value of the stored checksum.

Returning to the example program in Section A.1, "Main.c," the value of the stored checksum can be readily obtained by running the code using a debugger with a breakpoint set at the call to function crc_check.

When the program halts at the breakpoint, the registers CRCH:L will contain the calculated checksum, which can be read out from the debugger memory window. This value should be noted and stored in the checksum memory location as a declaration at the start of the software.

3 Conclusion

Freescale's Flexis AC CRC module represents an evolutionary step in error detection, offering improved performance and enhanced features over a software solution.

The user is advised to read the relevant chapters of the latest reference manual for the particular Flexis AC microcontroller, to ensure that all of the features and operation of the CRC module have been fully captured.

4 References

MC9S08AC128 Reference Manual www.freescale.com

ITU-T V.41 Recommendation: Code-Independent Error Control System, available at http://www.itu.int/publications/index.html.

ITU-T X.25 Recommendation: Interface between Data Terminal Equipment (DTE) and Data Circuit-terminating Equipment (DCE) for terminals operating in the packet mode and connected to public data networks by dedicated circuit, available at http://www.itu.int/publications/index.html.

ITU-T T.30 Recommendation: Procedures for document facsimile transmission in the general switched telephone network, available at http://www.itu.int/publications/index.html

5 Glossary

CRC — cyclic redundancy check

CRC16 — CCITT — A cyclic redundancy-checking algorithm based on the polynomial $x^{16} + x^{12} + x^5 + 1$.

CHECKSUM — A number that has been calculated as a function of some data, which can be used to verify its authenticity.

ITU-T — International Telecommunication Union Telecommunication Standardization Sector

Appendix A Cyclic Redundancy Check Software

The code shown within this appendix is only intended as an example for the CodeWarrior compiler and the MC9S08AC128. It has only been given a minimum level of test. It is provided as seen with no guarantees and promise of support.

A.1 Main.c

```
#include <hidef.h> /* for EnableInterrupts macro */
#include "derivative.h" /* include peripheral declarations */

long  start1_address ;    // start1 address to pass to crc_flash();
long  stop1_address;      // end1 address to pass to crc_flash();
long  start2_address ;    // start2 address to pass to crc_flash();
long  stop2_address;      // end2 address to pass to crc_flash();

/* This 16bit constant resides in the unused vector space outwith the main program */
const short CRC16signature @0xFF80 = (0xFBF9);

/* Error Routine If the CRC fails then the MCU will get locked in here!!!! */
 void sys_error(void)
  {
  for (;;) ;
    /* At this point the MCU system should be placed in a safe state */
     __RESET_WATCHDOG();
  }

void main(void) {



  EnableInterrupts;
  SOPT = 0x53; // Disable Watchdog



/* START OF FLASH CRC CHECK *****************************************************************/

/* Test H/W CRC engine by
    1) Initialise the CRC engine with 0xFFFF
    2) update the CRC engine from start_address to stop_address
    3) check the calculated result compares with expected result
*/
 start1_address = 0x00000;    // Start of Program code given to 16bit pointer
 stop1_address = 0x0FF7F;       // End of Program code give to 16bit pointer

 start2_address = 0x0FF82;    // Start of Program code given to 16bit pointer
 stop2_address = 0x1FFFF;       // End of Program code give to 16bit pointer

 init_crcCCITT();    // 0xFFFF seed CRC engine

 crc_flash (start1_address,stop1_address);    // calculate CRC signature
 crc_flash (start2_address,stop2_address);    // calculate CRC signature

 augment_message_for_crc_16(); // shift 16 0's to comply with CCITT algorithm (this is optional)

 crc_check (&CRC16signature); // check calculated CRC against CRC16signature
/* Send the address of the constant CRC16signature to this function */

/* END OF FLASH CRC CHECK *****************************************************************/

/* The application software can begin from here */

  for(;;) {
    __RESET_WATCHDOG(); /* feeds the dog */
  } /* loop forever */
  /* please make sure that you never leave main */
}
```

A.2 init_crcCCITT

```
void init_crcCCITT(void)
         {
                  CRCH = 0xFF;       // CRC seeded 0xFFFF
                  CRCL = 0xFF;       // CRC seeded 0xFFFF
         }
```

A.3 crc_flash

```
void crc_flash (long start_addr, long stop_addr)
{
      /* Put base address of flash into the LAP registers */
      LAP2 = (byte) (start_addr>>16);
      LAP1 = (byte) (start_addr>>8);
      LAP0 = (byte) start_addr;

      /* increment through the memory array passed */
      for ( 1; start_addr <= stop_addr; start_addr++ )
      {
         update_crcCCITT ( LBP ); // send a char to the CRC
      }
}  /* end of crc_flash function */
```

A.4 update_crcCCITT

```
void update_crcCCITT (char ch)
{
    CRCL = ch;
}     /* end of update_crcCCITT function */
```

A.5 augment_message_for_crc_16()

```
void augment_message_for_crc_16()
{
    CRCL=0x00;
    CRCL=0x00;
}
```

A.6 crc_check

```
void crc_check(short *blog)
{
  short CRC = ((CRCH<<8)|CRCL);
      if (CRC != (short)*blog)  sys_error();
      /* else return as CRC check is OK */
}
```

How to Reach Us:

Home Page:
www.freescale.com

Web Support:
http://www.freescale.com/support

USA/Europe or Locations Not Listed:
Freescale Semiconductor, Inc.
Technical Information Center, EL516
2100 East Elliot Road
Tempe, Arizona 85284
+1-800-521-6274 or +1-480-768-2130
www.freescale.com/support

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
www.freescale.com/support

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064
Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor China Ltd.
Exchange Building 23F
No. 118 Jianguo Road
Chaoyang District
Beijing 100022
China
+86 10 5879 8000
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

Document Number: AN3795
Rev. 0
04/2009