AVR236: CRC Check of Program Memory

8-bit AVR® RISC Microcontroller

Application Note

Features

- CRC Generation and Checking of Program Memory
- Supports all AVR® Controllers with LPM Instruction
- Compact Code Size, 44 Words (CRC Generation and CRC Checking)
- No RAM Requirement
- Checksum Stored in EEPROM
- Execution Time: 90 ms (AT90S8515 @ 8 MHz)
- 16 Bits Implementation, Easily Modified for 32 Bits
- Supports the CRC-16 Standard, Easily Modified for CRC-CCITT, CRC-32

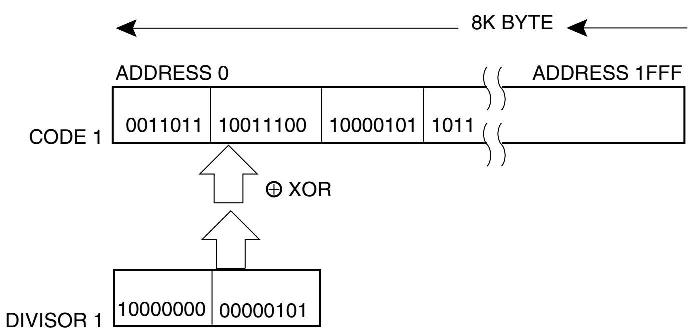
Introduction

This application note describes CRC (Cyclic Redundancy Check) theory and implementation of CRC to detect errors in program memory of the Atmel AVR microcontroller.

CRC is a widely used method of detecting errors in messages transmitted over noisy channels. New standards for secure microcontroller applications has introduced CRC as a method of detecting errors in Program memory of microcontrollers. It is preferable to implement the CRC calculation in compact code with low requirement for data storage memory since it frees up MCU resources for use in the actual application.

The implementation of CRC used in this application note is optimized for minimum code size and register usage.

Figure 1. CRC Checking of Program Memory Using 16-bit Divisor



Rev. 1143B–AVR–05/02

Theory of Operation

Checksums was originally used in communication through noisy channels. A number (the checksum) is computed as a function of the transmitted message. The Receiver uses the same function to compute a checksum, and compares the computed value with the value received from the transmitting side.

In this application note the checksum is constructed as a function of the code, and stored in the internal EEPROM. The microcontroller can later use the same function to calculate the checksum of the code and compare it with the appended checksum.

Example: Checksum calculated by summing the numbers of the code:

```
Code:                       04 29 06
Code with checksum:         04 29 06 39
```

This checksum is simply the sum of the numbers in the code.

If the second byte in the code is corrupted from 29 to 23, the error will be detected when the original checksum is compared with the computed checksum.

```
Original code with checksum:  04 29 06 39
Code with error:              04 23 06 39  -> Wrong !
```

If the first byte in the code is corrupted from 04 to 10 and the second byte is corrupted from 29 to 23, the checksum will not detect the errors.

```
Original code with checksum:  04 29 06 39
Code with error:              10 23 06 39-> Correct !
```

The problem with this checksum is that it is too simple. It may not detect errors on multiple bytes in the code and it may not detect errors in the checksum itself.

This example shows that addition is not sufficient to detect errors. CRC calculations use division instead of addition to calculate the checksum for the code. The principles are similar, but by using division multiple bit errors and burst errors will be detected.

The CRC algorithm treat the Program memory as an enormous binary number, which is divided by another fixed binary number. The remainder of this division is the checksum. The microcontroller will later perform the same division and compare the remainder with the calculated checksum.

Note that the division uses polynomial (modulo-2) arithmetic, which is similar to regular binary arithmetic, except it uses no carry. The addition of the numbers with polynomial arithmetic are simply XOR'ing the data.

Example: Addition in polynomial arithmetic:

$$\begin{array}{r} 1011\ 0110 \\ +\ \underline{1101\ 0011} \\ 0110\ 0101 \end{array}$$

The addition is equal to XOR'ing the two numbers.

Lets define the some properties for the polynomial arithmetic:

M(x) = a k-bit number (the code to be checked).

G(x) = an (n+1) bit number (the divisor or polynom).

R(x) = an n-bit number such that k>n (the remainder or checksum).

$$\frac{M(x)*2^n}{G(x)} = Q(x) + \frac{R(x)}{G(x)} \quad \text{Where Q(x) is the quotient}$$

Q(x) can now be described as:

$$Q(x) = \frac{M(x)*2^n + R(x)}{G(x)} \quad M(x) \times 2^n \text{ equals adding n zeros to the end of the code}$$

$$\frac{\text{If } M(x)*2^n}{G(x)} \text{ is replaced in the last equation}$$

$$\frac{M(x)*2^n+R(x)}{G(x)} = Q(x)+\frac{R(x)}{G(x)}+\frac{R(x)}{G(x)}=Q(x)$$

Which is equal to Q(x) since the divisor and the remainder are the same number, and adding it to itself is the same as XORing it, which results in zero.

Example of CRC Division

The hexadecimal number 6A which is the binary number 0110 1010, is divided with the divisor 1001 (=9 hex). The checksum will be the remainder of the operation 0110 1010 divided with 1001.

First append W zeros to the end of the original message (where W is the width of the divisor).

```
011010100000 / 1001 = 01100   Quotient is Ignored
0000
 1101
 1001
  1000
  1001
   0011
   0000
    0110
    0000
     1100
     1001
      1010
      1001
       0110
       0000
        1100
        1001
        0101 = 5 = Remainder = Checksum
```

The checksum is added to the end of the original code. The resulting code will be 6A5. When this code is checked, the code and the checksum is divided by the divisor. The remainder of this division is zero if no errors has occurred, non-zero otherwise.

Several standards are used today for CRC detection. The characteristics of the divisor vary from 8 to 32 bits, and the ability to detect errors varies with the width of the divisor used. Some commonly used CRC divisors are:

```
CRC-16    = 1 1000 0000 0000 0101= 8005(hex)
CRC-CCITT = 1 0001 0000 0010 0001= 1021 (hex)
CRC-32    = 1 0000 0100 1100 0001 0001 1101 1011 0111 = 04C11DB7 (hex)
```

Observe that in 16 bits divisors, the actual numbers of bits are 17, and in a 32 bits divisor the number of bits are 33. The MSB is always 1.

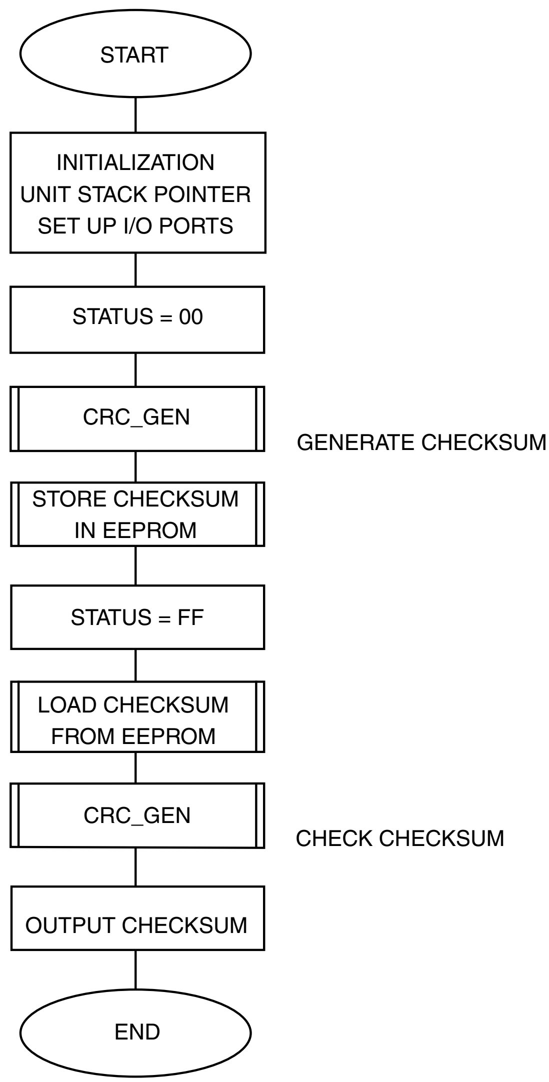
Software Description

Main Program

The main program is supplied to show operation of both the CRC generation and CRC checking. The checksum generated is stored in the internal EEPROM, and read back before the CRC checking is performed.

In most applications, the checksum will be generated by a programmer and placed at the last address of the Program memory.

Figure 2. Flowchart for the Main Program



The main program call the sub routine CRC_gen with Status Register = 0x00 after reset to generate a new checksum for the code. The generated checksum is stored in EEPROM.

To check the CRC checksum the routine CRC_gen is called with Status Register = 0xFF, or any value different from 0x00.

CRC Checksum Generation

The operation is based on the principle of rotating the entire Program memory bit by bit. The MSB is shifted into the Carry Flag. If the Carry Flag is 1 (one), the word is XOR'ed with the divisor. Note that the MSB of the Program memory which is shifted into the Carry Flag also is XOR'ed with the MSB of the divisor. Since they are both 1, the result will always be zero and the division is ignored.

At the end of the Program memory 16 zeros are appended to the code. The checksum is the resulting value of the complete XOR operation.

CRC Checksum Checking

The same principles are applied as for the generation, but the generated checksum is appended to the code, replacing the zeros. The result of the calculation including the appended checksum is zero if no errors has occurred, non-zero otherwise.

If the checksum is included in the Program code, only the checking part of the computation needs to be done in the Program code.

The same routine is used for both CRC generation and the CRC checking. A Global Register Status is loaded with 0x00 at function call to perform CRC generation. If the Status Register is loaded with any value different from 0x00 at function call, the function performs a CRC checksum checking.

The flowchart shows the flow of crc_gen routine which includes both the CRC generation and CRC checking.

The flowcharts in Figure 3 and Figure 4 describes the operation of the crc_gen subroutine.

Figure 3. CRC_gen Subroutine

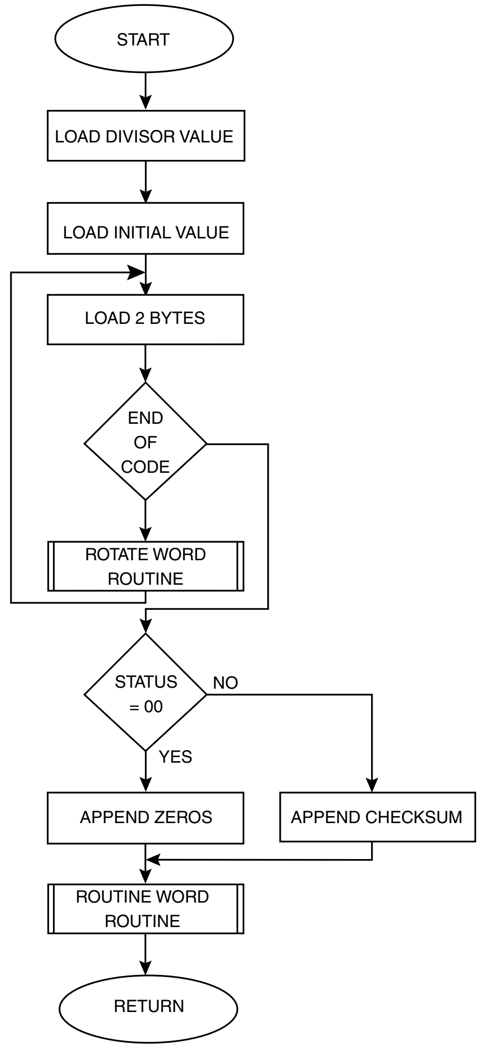
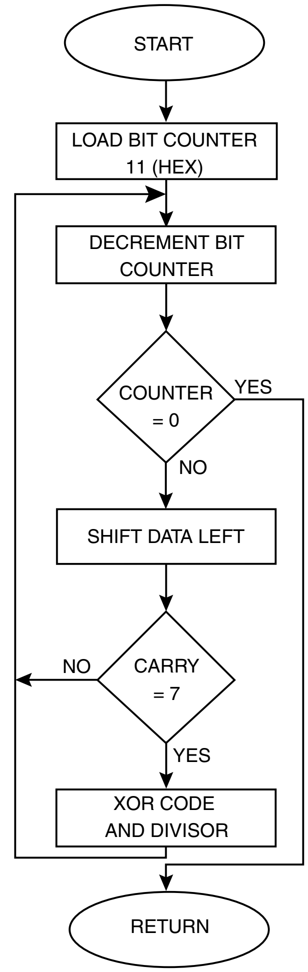


Figure 4. Rotate Subroutine



Modifications

The code example implements a 16-bit checksum for CRC-16 computation. The code is easily modified to support 32-bit checksum by increasing the size of the code buffer from 32 to 64 bits, and increasing the size of the divisor from 16 to 32 bits.

If the checksum is generated by a programmer and placed in the last memory location, only the code for checking the checksum needs to be included in the program. The code in the “end” section of the routine can be removed. Please see comments in the code.

Some CRC-algorithms requires the data register to have an initial value different from 0x00. If other values is used, the initial values can be loaded into the registers, replacing the two first LPM instructions. See comments in code for more information.

If the CRC algorithm is reflected, which means that the LSB of the bytes are shifted in first instead of the MSB, the routine can support this by replacing the LSL (Logical Shift Left) and ROL (Rotate Left) instructions with LSR (Logical Shift Right) and ROR (Rotate Right) instructions.

Other implementations of CRC computation exists with higher speed, most of them use a lookup table to increase the speed of the operation. The RAM requirements for such application makes them suitable for more complex systems.

Resources

Table 1. CPU and Memory Usage

Function	Code Size	Cycles	Register Usage	Interrupt	Description
main	36 words	–	R2, R3, R16, R22, R23, R24, R25	–	Initialization and Example Program
CRC_gen	44 words	700.000 (approx.)	R0, R1, R2, R3, R17, R18, R19, R20, R21, R22, R30, R31	–	Generate and Check CRC Checksum
EEwrite	7 words	13 cycles	R16, R23, R24, R25	–	Write CRC Checksum to EEPROM
EERread	4 words	8 cycles	R16, R23, R24, R25	–	Read CRC Checksum from EEPROM
TOTAL	91 words	–		–	

Table 2. Peripheral Usage

Peripheral	Description
2 Bytes EEPROM	Storing CRC Value
8 I/O Pins	Output Low Byte of CRC to LEDs

References

Fred Halsall

"Data Communication, Computer Networks and Open Systems"
1992 Addison-Wesley Publishers

Ross N. Williams

"The Painless Guide to Error Detection Algorithms"
ftp://ftp.rocksoft.com/papers/crc_v3.txt

Atmel Headquarters

Corporate Headquarters
2325 Orchard Parkway
San Jose, CA 95131
TEL 1(408) 441-0311
FAX 1(408) 487-2600

Europe
Atmel Sarl
Route des Arsenaux 41
Case Postale 80
CH-1705 Fribourg
Switzerland
TEL (41) 26-426-5555
FAX (41) 26-426-5500

Asia
Room 1219
Chinachem Golden Plaza
77 Mody Road Tsimhatsui
East Kowloon
Hong Kong
TEL (852) 2721-9778
FAX (852) 2722-1369

Japan
9F, Tonetsu Shinkawa Bldg.
1-24-8 Shinkawa
Chuo-ku, Tokyo 104-0033
Japan
TEL (81) 3-3523-3551
FAX (81) 3-3523-7581

Atmel Operations

Memory
2325 Orchard Parkway
San Jose, CA 95131
TEL 1(408) 441-0311
FAX 1(408) 436-4314

Microcontrollers
2325 Orchard Parkway
San Jose, CA 95131
TEL 1(408) 441-0311
FAX 1(408) 436-4314

La Chantrerie
BP 70602
44306 Nantes Cedex 3, France
TEL (33) 2-40-18-18-18
FAX (33) 2-40-18-19-60

ASIC/ASSP/Smart Cards
Zone Industrielle
13106 Rousset Cedex, France
TEL (33) 4-42-53-60-00
FAX (33) 4-42-53-60-01

1150 East Cheyenne Mtn. Blvd.
Colorado Springs, CO 80906
TEL 1(719) 576-3300
FAX 1(719) 540-1759

Scottish Enterprise Technology Park
Maxwell Building
East Kilbride G75 0QR, Scotland
TEL (44) 1355-803-000
FAX (44) 1355-242-743

RF/Automotive
Theresienstrasse 2
Postfach 3535
74025 Heilbronn, Germany
TEL (49) 71-31-67-0
FAX (49) 71-31-67-2340

1150 East Cheyenne Mtn. Blvd.
Colorado Springs, CO 80906
TEL 1(719) 576-3300
FAX 1(719) 540-1759

Biometrics/Imaging/Hi-Rel MPU/
High Speed Converters/RF Datacom
Avenue de Rochepleine
BP 123
38521 Saint-Egreve Cedex, France
TEL (33) 4-76-58-30-00
FAX (33) 4-76-58-34-80

e-mail
literature@atmel.com

Web Site
http://www.atmel.com

1143B–AVR–05/02 0M