# AVR236: CRC Check of Program Memory

AVR236

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**

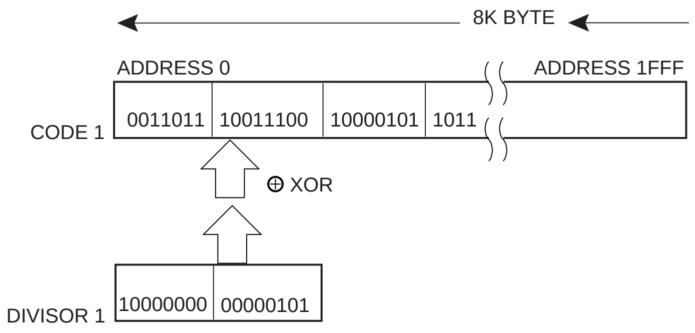
## Introduction

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. 1143A–10/98

## 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:

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

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).

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

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.

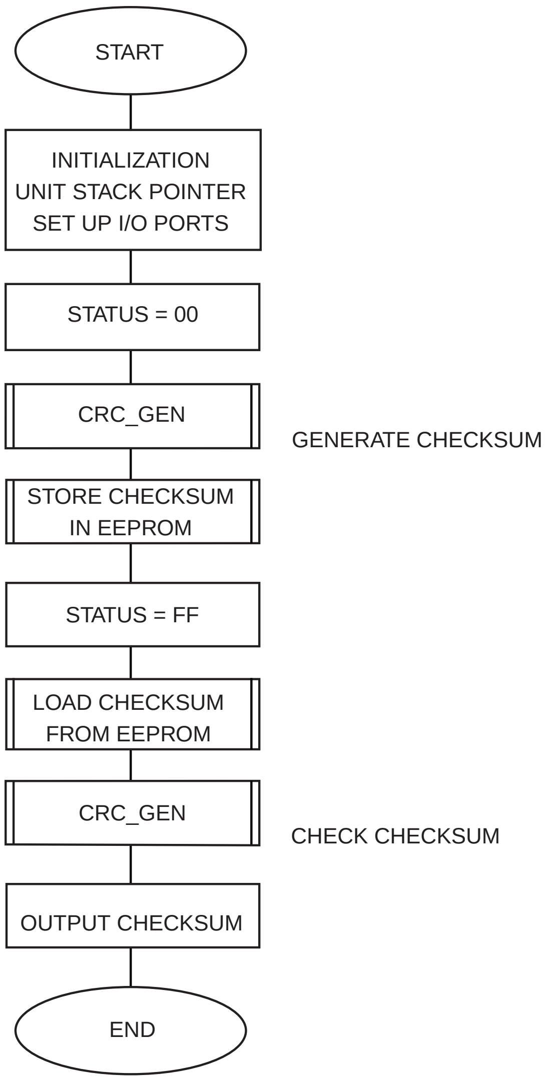
## Software Description

### 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

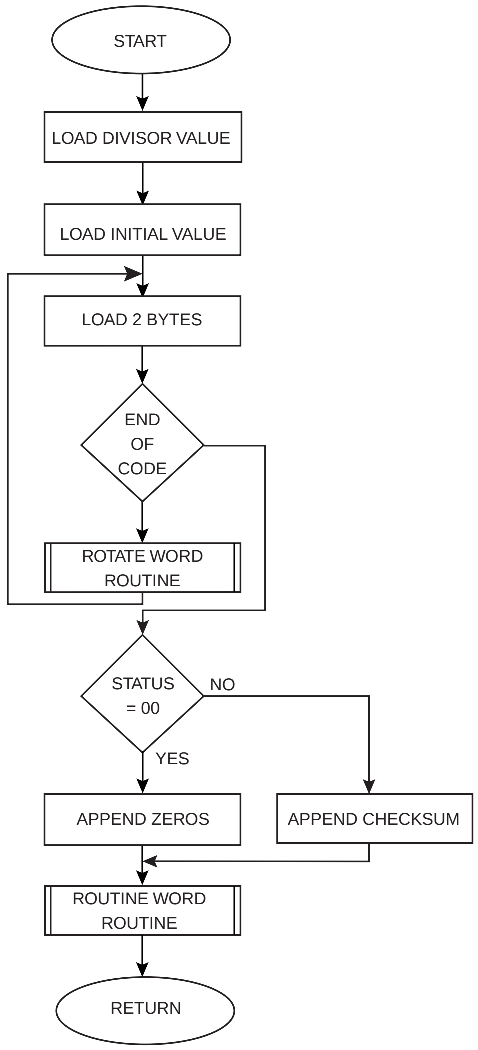
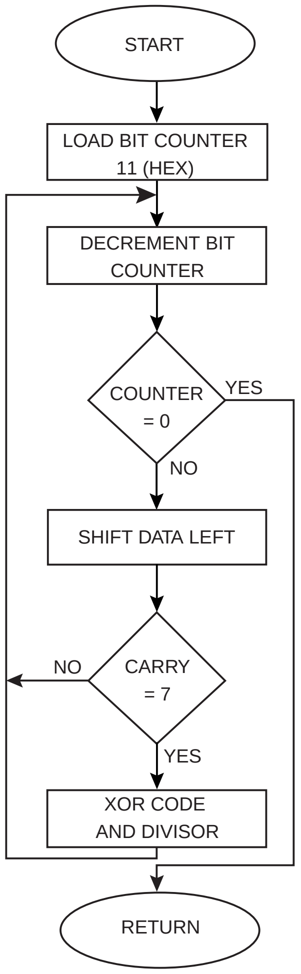


**Figure 4.** Rotate Subroutine



## 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. C-code examples for table driven CRC implementation can be obtained from avr@atmel.com.

## 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 |

## Code Listing

```
;**** A P P L I C A T I O N   N O T E   A V R236 ************************
;*
;* Title:           CRC check of program memory
;* Version:         1.0
;* Last updated:    98.06.15
;* Target:          AT90Sxxxx (All AVR Devices with LPM instr)
;*
;* Support E-mail:  avr@atmel.com
;*
;* NOTE: Always check out the Atmel web site, www.atmel.com for the latest and
;*updated version of this software.
;*
;* DESCRIPTION
;* This application note describes how to perform CRC computation
;* of code memory contents using a simple algorithm.
;* To generate CRC checksum load the register "status" with 00 and call the
;*routine "crc_gen". The resulting checksum is placed in the registers
;* byte2(low byte) and byte3(high byte).
;*
;* To check the CRC checksum load the register "status" with FF and call the
;*routine "crc_gen". The resulting checksum is placed in the registers
;* byte2(low byte) and byte3(high byte). If the checksum is 00 the program code
;*is
;* correct, if the checksum is non-zero an error has been introduced in the
;*program code
;***************************************************************************

.include "8515def.inc"
.device AT90S8515

;***** Constants

.equ    PROGSIZE    = 0x1FFF    ; Size of program memory(bytes)
.equ    CR          = 0x8005    ; CRC divisor value

;***************************************************************************
;*
;*      PROGRAM START - EXECUTION STARTS HERE
;*
;***************************************************************************

        .cseg

        .org $0000
        rjmp RESET      ;Reset handle


;****************************************************************************
;*
```

```
;* "crc_gen" - Generation and checking of CRC checksum
;*
;* This subroutine generates the checksum for the program code.
;* 32 bits are loaded into 4 register, the upper 16 bits are XORed
;* with the divisor value each time a 1 is shifted into the carry flag
;* from the MSB.
;*
;* If the status byte is 0x00,the routine will generate new checksum:
;* After the computing the code 16 zeros are
;* appended to the code and the checksum is calculated.
;*
;* If the status byte is different from 0X00, the routine will check if the
;* current checksum is valid.
;* After the computing the code the original checksum are
;* appended to the code and calculated. The result is zero if no errors occurs
;* The result is placed in registers byte2 and byte3
;*
;* Number of words      :44 + return
;* Number of cycles     :program memory size(word)*175(depending on memory content)
;* Low registers used   :6 (byte0,byte1,byte2,byte3)
;* High registers used  :7 (sizel,sizeh,crdivl,crdivh,count,status,zl,zh)
;*
;***************************************************************************

;***** Subroutine Register Variables

.def    byte0  = r0             ; Lower byte of lower word
.def    byte1  = r1             ; Upper byte of lower word
.def    byte2  = r2             ; Lower byte of upper word
.def    byte3  = r3             ; Upper byte of upper word
.def    crc    = r4                     ; CRC checksum low byte
.def    crch   = r5                     ; CRC checksum high byte
.def    sizel  = r17                    ; Program code size register
.def    sizeh  = r18
.def    crdivl = r19            ; CRC divisor register
.def    crdivh = r20
.def    count  = r21            ; Bit counter
.def    status = r22            ; Status byte: generate(0) or check(1)

crc_gen:ldi  sizel,low(PROGSIZE)  ;Load end of program memory address
  ldi        sizeh,high(PROGSIZE)
  clr        zl                   ;Clear Z pointer
  clr        zh
  ldi        crdivh,high(CR)      ;Load divisor value
  ldi        crdivl,low(CR)
  lpm                             ;Load first memory location
  mov        byte3,byte0          ;Move to highest byte
  adiw       zl,0x01              ;Increment Z pointer
  lpm                             ;Load second memory location
  mov        byte2,byte0
```

```
new_word:cp  zl,sizel            ;Loop starts here
  cpc        zh,sizeh            ;Check for end of code
  brge       end                 ;Jump if end of code
  adiw       zl,0x01
  lpm                            ;Load high byte
  mov        byte1,byte0         ;Move to upper byte
  adiw       zl,0x01             ;Increment Z pointer
  lpm                            ;Load program memory location
  rcall      rot_word            ;Call the rotate routine
  rjmp       new_word

end:         ;ret                ;uncomment this line if checksum is in last flash memory address.
  ldi        count,0x11
  cpi        status,0x00
  brne       check
  clr        byte0               ;Append 16 bits(0x0000) to
  clr        byte1               ;the end of the code for CRC generation
  rjmp       gen
check:mov    byte0,crc           ;Append the original checksum to
  mov        byte1,crch          ;the end of the code for CRC checking
gen:rcall    rot_word            ;Call the rotate routine
  mov        crc,byte2
  mov        crch,byte3
  ret                            ;Return to main prog

rot_word:ldi count,0x11
rot_loop:dec count               ;Decrement bit counter
  breq       stop                ;Break if bit counter = 0
  lsl        byte0               ;Shift zero into lowest bit
  rol        byte1               ;Shift in carry from previous byte
  rol        byte2               ;Preceed shift
  rol        byte3
  brcc       rot_loop            ;Loop if MSB = 0
  eor        byte2,crdivl
  eor        byte3,crdivh        ;XOR high word if MSB = 1
  rjmp       rot_loop
stop:ret

;***************************************************************************
;*
;* EERead_seq
;*
;* This routine reads the
;* EEPROM into the global register variable "temp".
;*
;* Number of words     :4+ return
;* Number of cycles    :8 + return
;* High Registers used :4 (temp,eeadr,eeadrh,eedata)
;*
;***************************************************************************
```

```
.def    temp      = r16
.def    eeadr     = r23
.def    eeadrh    = r24
.def    eedata    = r25

;***** Code

eeread:
  out   EEARH,eeadrh   ;output address high byte
  out   EEARL,eeadr    ;output address low byte
  sbi   EECR,EERE      ;set EEPROM Read strobe
  in    eedata,EEDR    ;get data
  ret

;***************************************************************************
;*
;* EEWrite
;*
;* This subroutine waits until the EEPROM is ready to be programmed, then
;* programs the EEPROM with register variable "EEdwr" at address "EEawr"
;*
;* Number of words       :7 + return
;* Number of cycles      :13 + return (if EEPROM is ready)
;* Low Registers used    :None
;* High Registers used:  :4 (temp,eeadr,eeadr,eedata)
;*
;*********************************************************************

.def    temp      = r16
.def    eeadr     = r23
.def    eeadrh    = r24
.def    eedata    = r25

eewrite:sbic EECR,EEWE    ;If EEWE not clear
  rjmp       EEWrite      ;         Wait more
  out        EEARH,eeadrh ;Output address high byte
  out        EEARL,eeadr  ;Output address low byte
  out        EEDR,eedata  ;Output data
  sbi        EECR,EEMWE
  sbi        EECR,EEWE    ;Set EEPROM Write strobe
  ret

;************************************************************************
;*
;*  Start Of Main Program
;*
.cseg
.def    crc       = r4      ;Low byte of checksum to be returned
.def    crch      = r5      ;High byte of checksum to be returned
.def    temp      = r16
```

```
.def    status    = r22         ;Status byte: generate(0) or check(1)
.def    eeadr     = r23
.def    eeadrh    = r24
.def    eedata    = r25

RESET:ldi   r16,high(RAMEND)   ;Initialize stack pointer
      out   SPH,r16            ;High byte only required if
      ldi   r16,low(RAMEND)    ;RAM is bigger than 256 Bytes
      out   SPL,r16

      ldi   temp,0xff
      out   DDRB,temp          ;Set PORTB as output
      out   PORTB,temp         ;Write 0xFF to PORTB
      clr   status             ;Clear status register,ready for CRC generation
      rcall crc_gen

      ldi   eeadr,0x01         ;Set address low byte for EEPROM write
      ldi   eeadrh,0x00        ;Set address high byte for EEPROM write
      mov   eedata,crc         ;Set CRC low byte in EEPROM data
      rcall eewrite            ;Write EEPROM

      ldi   eeadr,0x02         ;Set address low byte for EEPROM write
      ldi   eeadrh,0x00        ;Set address high byte for EEPROM write
      mov   eedata,crch        ;Set CRC high byte in EEPROM data
      rcall eewrite            ;Write EEPROM

      out   PORTB,crc          ;Output CRC low value to PORTB

mainloop:
      sbic  EECR,EEWE          ;If EEWE not clear
      rjmp  mainloop           ;    Wait more

;********** Insert program code here *************

      ldi   eeadr,0x01         ;Set address low byte for EEPROM read
      ldi   eeadrh,0x00        ;Set address high byte for EEPROM read
      rcall eeread             ;Read EEPROM
      mov   crc,eedata         ;Read CRC low byte from EEPROM

      ldi   eeadr,0x02         ;Set address low byte for EEPROM read
      ldi   eeadrh,0x00        ;Set address high byte for EEPROM read
      rcall eeread             ;Read EEPROM
      mov   crch,eedata        ;Read CRC low byte from EEPROM

      ser   status             ;Set status register, prepare for CRC checking
      rcall crc_gen

loop: out PORTB,crc            ;Output CRC low value to PORTB
      rjmp loop
.exit
```

## References

| | |
|---|---|
| Fred Halsall | “Data Communication, Computer Networks and Open Systems”<br>1992 Addison-Wesley Publishers |
| Ross N. Williams | “The Painless Guide to Error Detection Algorithms”<br>ftp.adelaide.edu.au/pub/rocksoft/crc_v3.txt |