Logic Instructions and Programs

READING

The AVR Microcontroller and Embedded Systems using Assembly and C
by Muhammad Ali Mazidi, Sarmad Naimi, and Sepehr Naimi

Chapter 5: Arithmetic, Logic Instructions, and Programs

Section 5.3: Logic and Compare Instructions

Section 5.4: Rotate and Shift Instructions and Data Serialization

Section 5.5: BCD and ASCII Conversion

1 Sections 5.1 and 5.2 covered in AVR ALU and SREG Lecture
Contents

Reading .................................................................................................................................................. 2
Overview .............................................................................................................................................. 4
Sample Application – Knight Rider ....................................................................................................... 5
Sample Application – Bicycle Light ....................................................................................................... 6
Clearing and Setting Bits ....................................................................................................................... 6
Clearing and Setting a Bit in the AVR Status Register ....................................................................... 9
Testing Bits .......................................................................................................................................... 10
Toggling Bits......................................................................................................................................... 11
Rotating and Shifting Bits .................................................................................................................... 12
Clearing and Setting a Bit in One of the first 32 I/O registers ............................................................... 13
Setting a Bit Pattern ............................................................................................................................ 14
Questions .............................................................................................................................................. 15
Appendix A: Knight Rider Optimized .................................................................................................. 16
Appendix B: Knight Rider Addressing Indirect .................................................................................. 17
### Overview

#### Clearing and Setting a Bit In ...

<table>
<thead>
<tr>
<th>Where</th>
<th>Instruction</th>
<th>Alternative</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>I/O (0 – 31)</td>
<td>cbi, sbi</td>
<td></td>
<td>Use with I/O Ports</td>
</tr>
<tr>
<td>SREG</td>
<td>cl{i,t,h,s,v,n,z,c}</td>
<td>bclr</td>
<td></td>
</tr>
<tr>
<td></td>
<td>se{i,t,h,s,v,n,z,c}</td>
<td>bset</td>
<td></td>
</tr>
</tbody>
</table>

#### Working with General Purpose Register Bits

**Clearing and Setting a Byte**
- clr, ser

**Clearing Bits**
- and, cbr

**Testing Bits**
- and

**Setting Bits**
- or, sbr

**Inserting a Bit Pattern**
- cbr ▶️ sbr

**Complementing (Toggling) Bits**
- eor

**Rotating Bits**
- rol, ror

**Shifting Bits**
- lsl, lsr, asr

**Swapping Nibbles**
- swap

Also consider using sbrc, sbrs, sbic, sbis (see Control Transfer Lecture)
SAMPLE APPLICATION – KNIGHT RIDER

KnightRider:

; See page 5 and 6 - Clearing and Setting Bits
clr r16 // start with r9 bit 6 set - LED 6
sbr r16, 0b10000000
; Q1: How could we have done this using 1 instruction?
ldi r17, (1<<SREG_T) // equivalent to 0b01000000

; See page 7 - Clearing and Setting a Bit in the AVR Status Register
clt // initialize T = 0, scan right

; See page 8 - Testing Bits
loop:
  ldi r19, 0b100000001
  and r19, r16 // test if LED hit is at an edge
  breq contScan // continue scan if z = 0

; See page 9 - Toggling Bits
  in r16, SREG // toggle T bit
  eor r16, r17
  out SREG, r16

; See page 10 - Rotating and Shifting Bits
contScan:
  brts scanLeft // rotate right or left
  lsr r16
  rjmp cont
scanLeft:
  lsl r16
cont:
  mov spiLEDs, r16
  call WriteDisplay
  rcall Delay
  rjmp loop
SAMPLE APPLICATION – BICYCLE LIGHT

A bicycle light has 5 LEDs.

BicycleLight1: A repeating pattern starts with the center LED turned ON. The center LED is then turned OFF, and the LEDs to the left and right of the center LED are turned ON. Each LED continues its scan to the left or right. Once the LEDs reach the end the pattern repeats itself. Using the CSULB shield, write a program to simulate this bicycle light.

BicycleLight2: Same as Bicycle1 except when LEDs reach the edge, they scan back to the center.

BicycleLight1:

```assembly
clr r7    // turn off 7 segment
begin: ldi r16, 0x04    // scan register r16 = 4
        mov r17, r16     // scan register r17 = 4
scan:   mov r8, r16    // do not modify r16
        cbr r17, 0x20    // r17 bit 5 = 1 at end of cycle
        or r8, r17      // combine scan registers
        rcall Delay
        call WriteDisplay
        lsr r16          // scan r16 right
        lsl r17          // scan r17 left
        brne scan       // if r17 <> 0 then continue scan
        rjmp begin      // else start next cycle
```

BicycleLight2:

```assembly
//    |  
// ldi r16, 0x08    // 00010_0000 start just in from edges
// ldi r17, 0x02    // 00010_0010
scan: clr r8    // combine scan registers
        or r8, r16
        or r8, r17
        rcall Delay
        call WriteDisplay
        lsl r17        // scan r17 left
        lsr r16        // scan r16 right
        brcc scan
        rjmp BicycleLight2
```

```assembly
```
CLEARING AND SETTING BITS

To clear a bit set the corresponding mask bit to 0
and source/dest register, mask register

Problem: Convert numeric ASCII value ('0' – '9') to its binary coded decimal (BCD) equivalent (0 – 9).

- What we have: '0' to '9' which equals 30_{16} to 39_{16}
- What we want: 0 to 9 which equals 00_{16} to 09_{16}

Solution: Mask out high-order nibble

```
lds r16, ascii_value
ldi r17, 0x0F
and r16, r17  // or simply andi
sts bcd_value, r16
```

An alternative to the and instruction is the Clear Bits in Register cbr instruction.

cbr source/dest register, mask bits

The cbr instruction clears the specified bits in the source/Destination Register (Rd). It performs the logical AND between the contents of register Rd and the complement of the constant mask (K). The result will be placed in register Rd.

\[ \text{Rd} \leftarrow \text{Rd} \cdot (0xFF - K), \] here is how the previous problem would be solved using the cbr instruction.

```
lds r16, ascii_value
cbr r16, 0xF0
sts bcd_value, r16
```
Clearing and Setting Bits

To set a bit set the corresponding mask bit to 0

or source/dest register, control register

Example: Set to one (1) bits 4 and 2 in some port.

in r16, some_port
ldi r17, 0b00010100
or r16, r17 // or simply ori
out some_port, r16

An alternative to the or instruction is the Set Bits in Register sbr instruction.

sbr source/dest register, mask bits

The sbr instruction sets the specified bits in the source/Destination Register (Rd). It performs the logical ORI between the contents of register Rd and the constant control (K). The result will be placed in register Rd.

Rd ← Rd + K

Here is how the previous problem would be solved using the cbr instruction.

in r16, some_port
sbr r16, 0b00010100
out some_port, r16
CLEARING AND SETTING A BIT IN THE AVR STATUS REGISTER

<table>
<thead>
<tr>
<th>Bit</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x3F (0x5F)</td>
<td>I</td>
<td>T</td>
<td>H</td>
<td>S</td>
<td>V</td>
<td>N</td>
<td>Z</td>
<td>C</td>
</tr>
<tr>
<td>Read/Write</td>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
</tr>
<tr>
<td>Initial Value</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

AVR Instructions for Clearing and Setting SREG bits

cl{i,t,h,s,v,n,z,c} or bclr SREG_{I,T,H,S,V,N,Z,C} // defined in m328Pdef.inc
se{i,t,h,s,v,n,z,c} or bset SREG_{I,T,H,S,V,N,Z,C} // defined in m328Pdef.inc

Examples:

Disable all Interrupts

cli

Set T bit

set

---

**TESTING BITS**

Use the `andi` instruction to test if more than one bit is set

```assembly
andi  source/dest register, mask bits
```

**Example 1:** Branch if bit 7 or bit 0 is set

```assembly
lds  r16, some_bits  // 1000 0000 ← example
andi r16, 0b10000001  // 1000 0001
brbc SREG_Z, bit_set  // 1000 0000 (alt. brne)
```

**Example 2:** Branch if bit 4 and bit 2 are clear

```assembly
lds  r16, some_bits  // 1101 1001 ← example
andi r16, 0b00010100  // 0001 0100
brbs SREG_Z, bits_zero  // 0001 0000 (alt. breq)
```

Consider using one of the “Skip if Bit” instructions if you only need to test one bit.

*Review “Control Transfer” lecture material for details.*

Use the `tst` instructions to test if a register is Zero or Minus.

Tests if a register is zero or negative. Performs a logical AND between a register and itself. The register will remain unchanged.

**Example:** Branch if bear is in the forest

```assembly
rcall inForest  // returns false(r24 = 0) if bear is not in the forest
tst  r24
breq not_in_forest  // branch if r24 = 0
```
**TOGGING BITS**

To toggle (complement) a bit set the corresponding mask bit to 1

\[ \text{eor} \quad \text{source/dest register, mask register} \]

**Example: Toggle bits 5 and 3 of I/O-Port D.**

\[
\begin{align*}
\text{ldi} & \quad r16, \text{PORTD} & \quad \text{// 1101 1001} \quad \text{-example} \\
\text{ldi} & \quad r17, \text{0x28} & \quad \text{// 0010 1000} \\
\text{eor} & \quad r16, r17 & \quad \text{// 1111 0001} \\
\text{out} & \quad \text{PORTD}, r16 & \\
\end{align*}
\]

When toggling an I/O-Port bit, consider writing a one to the corresponding pin.

*Review “AVR Peripherals” lecture material for details.*

**Example: Toggle bits 5 and 3 of I/O-Port D.**

\[
\begin{align*}
\text{sbi} & \quad \text{PIND, PIND5} & \quad \text{// equivalent to sbi 0x09, 5} \\
\text{sbi} & \quad \text{PIND, PIND3} & \\
\end{align*}
\]

When toggling a byte (8 bits), use the Complement instruction.

**Example: Write TurnAround code snip-it (i.e., toggle SRAM variable dir)**

\[
\begin{align*}
\text{lds} & \quad r16, \text{dir} & \quad \text{// 1101 1001} \quad \text{- facing East} \\
\text{com} & \quad r16 & \quad \text{// 0010 0110} \quad \text{- facing West} \\
\text{cbr} & \quad r16, \text{0xFC} & \quad \text{// 1111 1100} \quad \text{clear unused bits (optional)} \\
\text{sts} & \quad \text{dir, r16} & \quad \text{// 0000 0010} \\
\end{align*}
\]

**Question:** How could you have complemented dir without modifying the other 6 bits?
ROTATING AND SHIFTING BITS

Rotate Instructions allow us to rearrange bits without losing information and to sequentially test bit (brcc, brcs). Shift instructions allow us to quickly multiply and/or divide signed and/or unsigned numbers by 2.

**Rotate Left through Carry**

\[ \text{rol} \quad \text{Rd} \]

Shifts all bits in Rd one place to the left. The C Flag is shifted into bit 0 of Rd. Bit 7 is shifted into the C Flag. This operation, combined with LSL, effectively multiplies multi-byte signed and unsigned values by two.

**Rotate Right through Carry**

\[ \text{ror} \quad \text{Rd} \]

Shifts all bits in Rd one place to the right. The C Flag is shifted into bit 7 of Rd. Bit 0 is shifted into the C Flag. This operation, combined with ASR, effectively divides multi-byte signed values by two. Combined with LSR it effectively divides multibyte unsigned values by two. The Carry Flag can be used to round the result.

**Logical Shift Left (Arithmetic Shift Left)**

\[ \text{lsl} \quad \text{Rd} \]

Shifts all bits in Rd one place to the left. Bit 0 is cleared. Bit 7 is loaded into the C Flag of the SREG. This operation effectively multiplies signed and unsigned values by two.

**Logical Shift Right**

\[ \text{lsr} \quad \text{Rd} \]

Shifts all bits in Rd one place to the right. Bit 7 is cleared. Bit 0 is loaded into the C Flag of the SREG. This operation effectively divides an unsigned value by two. The C Flag can be used to round the result.

**Arithmetic Shift Right**

\[ \text{asr} \quad \text{Rd} \]

Shifts all bits in Rd one place to the right. Bit 7 is held constant. Bit 0 is loaded into the C Flag of the SREG. This operation effectively divides a signed value by two without changing its sign. The Carry Flag can be used to round the result.
### Clearing and Setting a Bit in One of the First 32 I/O Registers

Example: Pulse Clock input of Proto-Shield Debounce D Flip-flop (PORTD5). Assume currently at logic 0.

```c
sbi PORTD, 5
cbi PORTD, 5
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Name</th>
<th>Bit 7</th>
<th>Bit 6</th>
<th>Bit 5</th>
<th>Bit 4</th>
<th>Bit 3</th>
<th>Bit 2</th>
<th>Bit 1</th>
<th>Bit 0</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1F (0x3F)</td>
<td>EECR</td>
<td>−</td>
<td>−</td>
<td>EEMP1</td>
<td>EEMP0</td>
<td>EERIE</td>
<td>EEMPE</td>
<td>EEPE</td>
<td>EEIE</td>
<td>21</td>
</tr>
<tr>
<td>0x1E (0x3E)</td>
<td>GPIOR0</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>26</td>
</tr>
<tr>
<td>0x1D (0x3D)</td>
<td>EIMSK</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>72</td>
</tr>
<tr>
<td>0x1C (0x3C)</td>
<td>EIFR</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>72</td>
</tr>
<tr>
<td>0x1B (0x3B)</td>
<td>POIFR</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>72</td>
</tr>
<tr>
<td>0x1A (0x3A)</td>
<td>Reserved</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
</tr>
<tr>
<td>0x19 (0x39)</td>
<td>Reserved</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
</tr>
<tr>
<td>0x18 (0x38)</td>
<td>Reserved</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
</tr>
<tr>
<td>0x17 (0x37)</td>
<td>TIFR2</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>163</td>
</tr>
<tr>
<td>0x16 (0x36)</td>
<td>TIFR1</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>139</td>
</tr>
<tr>
<td>0x15 (0x35)</td>
<td>TIFR0</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
</tr>
<tr>
<td>0x14 (0x34)</td>
<td>Reserved</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
</tr>
<tr>
<td>0x13 (0x33)</td>
<td>Reserved</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
</tr>
<tr>
<td>0x12 (0x32)</td>
<td>Reserved</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
</tr>
<tr>
<td>0x11 (0x31)</td>
<td>Reserved</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
</tr>
<tr>
<td>0x10 (0x30)</td>
<td>Reserved</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
</tr>
<tr>
<td>0x0F (0x2F)</td>
<td>Reserved</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
</tr>
<tr>
<td>0x0E (0x2E)</td>
<td>Reserved</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
</tr>
<tr>
<td>0x0D (0x2D)</td>
<td>Reserved</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
</tr>
<tr>
<td>0x0C (0x2C)</td>
<td>Reserved</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
</tr>
<tr>
<td>0x0B (0x2B)</td>
<td>PORTD</td>
<td>PORTD7</td>
<td>PORTD6</td>
<td>PORTD5</td>
<td>PORTD4</td>
<td>PORTD3</td>
<td>PORTD2</td>
<td>PORTD1</td>
<td>PORTD0</td>
<td>99</td>
</tr>
<tr>
<td>0x0A (0x2A)</td>
<td>DDRD</td>
<td>DDD7</td>
<td>DDD6</td>
<td>DDD5</td>
<td>DDD4</td>
<td>DDD3</td>
<td>DDD2</td>
<td>DDD1</td>
<td>DDD0</td>
<td>99</td>
</tr>
<tr>
<td>0x09 (0x29)</td>
<td>PIND</td>
<td>PIND7</td>
<td>PIND6</td>
<td>PIND5</td>
<td>PIND4</td>
<td>PIND3</td>
<td>PIND2</td>
<td>PIND1</td>
<td>PIND0</td>
<td>93</td>
</tr>
<tr>
<td>0x08 (0x28)</td>
<td>PORTC</td>
<td>−</td>
<td>PORTC5</td>
<td>PORTC4</td>
<td>PORTC3</td>
<td>PORTC2</td>
<td>PORTC1</td>
<td>PORTC0</td>
<td>−</td>
<td>92</td>
</tr>
<tr>
<td>0x07 (0x27)</td>
<td>DDRC</td>
<td>−</td>
<td>DDG6</td>
<td>DDG5</td>
<td>DDG4</td>
<td>DDG3</td>
<td>DDG2</td>
<td>DDG1</td>
<td>DDG0</td>
<td>92</td>
</tr>
<tr>
<td>0x06 (0x26)</td>
<td>PINC</td>
<td>−</td>
<td>PINC6</td>
<td>PINC5</td>
<td>PINC4</td>
<td>PINC3</td>
<td>PINC2</td>
<td>PINC1</td>
<td>PINC0</td>
<td>92</td>
</tr>
<tr>
<td>0x05 (0x25)</td>
<td>PORTB</td>
<td>PORTB7</td>
<td>PORTB6</td>
<td>PORTB5</td>
<td>PORTB4</td>
<td>PORTB3</td>
<td>PORTB2</td>
<td>PORTB1</td>
<td>PORTB0</td>
<td>92</td>
</tr>
<tr>
<td>0x04 (0x24)</td>
<td>DDRB</td>
<td>DDB7</td>
<td>DDB6</td>
<td>DDB5</td>
<td>DDB4</td>
<td>DDB3</td>
<td>DDB2</td>
<td>DDB1</td>
<td>DDB0</td>
<td>92</td>
</tr>
<tr>
<td>0x03 (0x23)</td>
<td>PINB</td>
<td>PINB7</td>
<td>PINB6</td>
<td>PINB5</td>
<td>PINB4</td>
<td>PINB3</td>
<td>PINB2</td>
<td>PINB1</td>
<td>PINB0</td>
<td>92</td>
</tr>
<tr>
<td>0x02 (0x22)</td>
<td>Reserved</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
</tr>
<tr>
<td>0x01 (0x21)</td>
<td>Reserved</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
</tr>
<tr>
<td>0x00 (0x20)</td>
<td>Reserved</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
</tr>
</tbody>
</table>
Use the Clear Bits in Register \texttt{cbr} or functionally equivalent \texttt{andi} instruction in combination with the Set Bits in Register \texttt{sbr} to set a bit pattern in a register.

**Problem:** Convert a binary coded decimal (BCD) (0 – 9) number to its ASCII equivalent value (‘0’ – ‘9’).
- What we have: 0 to 9 which equals X0_{16} to X9_{16}
  The X indicates that we do not know what is contained in this nibble.
- What we want: ‘0’ to ‘9’ which equals 30_{16} to 39_{16}

**Solution:** Set high-order nibble to 3_{16}

\begin{verbatim}
lds r16, bcd_value
andi r16, 0x0F    // clear most significant nibble
sbr r16, 0x30    // set bits 5 and 4
sts ascii_value, r16
\end{verbatim}

**What is Happening**

\begin{verbatim}
bcd_value 1000 0111 0x87   // BCD 7
andi 0000 1111
0000 0111
sbr 0011 0000
0011 0111 0x37   // ASCII ‘7’
\end{verbatim}
**QUESTIONS**

1. What instruction is used to divide a signed number by 2?

2. What instruction is used to multiply an unsigned number by 2?

3. What instruction(s) would be used to convert a word pointer into a byte pointer? A word pointer is a register pair like Z containing the address of a 16-bit data (2 byte) word in an SRAM Table. A byte pointer is a register pair like Z containing the address of an 8-bit data byte in a corresponding SRAM Table. Assuming there is a one-to-one relationship between each word in the first table with a byte in the second table. And remembering that SRAM is always addressed at the Byte level, how would convert a pointer defined for the word table into a pointer defined for the byte table.
APPENDIX A: KNIGHT RIDER OPTIMIZED

.INCLUDE <m328pdef.inc>
   rjmp  reset

.INCLUDE "spi_shield.inc"

reset:
   call InitShield

// initialize knight rider
   ldi r16, 0b10000000 // start with r9 bit 7 set - LED 7
   mov spiLEDS, r16

// initialize roulette
   ldi  r19,0xE0
   ldi  r20,0x1F
   ldi  r16,0x01
   mov spi7SEG,r16

loop:
// night rider routine
   ldi   r16, 0b10000001
   and   r16, spiLEDS  // test if LED hit is at an edge
   breq  contScan    // continue scan if z = 0
   bst   spiLEDS, 0  // if right LED ON, then T = 1
contScan:
   brts  scanLeft    // rotate right or left
   lsr   spiLEDS
   rjmp  cont

scanLeft:
   lsl   spiLEDS

cont:
// roulette routine
   add  spi7SEG, r19
   and  spi7SEG, r20
   rol  spi7SEG
   rcall WriteDisplay
   rcall Delay

// display routine
   rcall WriteDisplay
   rcall Delay
   rjmp  loop
APPENDIX B: KNIGHT RIDER ADDRESSING INDIRECT

begin:
  ldi   r16,  14  // loop 14 times
  ldi  ZH, high(Table<<1) // set base address
  ldi  ZL, low(Table<<1)

scan:
  lpm  r9, Z+  // load constant to LED display register
  rcall WriteDisplay  // display routine
  rcall Delay
  dec   r16
  brne scan  // if r17 <> 0 then continue scan
  rjmp  begin  // else start next cycle

KnightRider: .DB 0x80, 0x40, 0x20, 0x10, 0x08, 0x04, 0x02
            .DB 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40