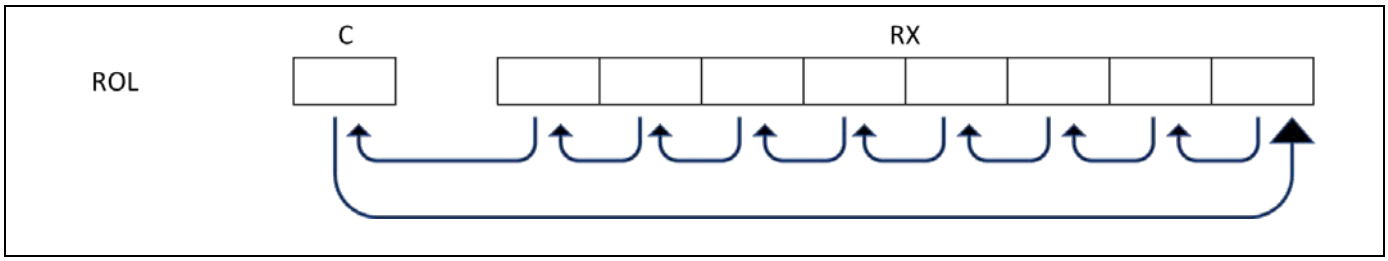
Microcontroller board - the screen: Tips and tricks

This document lists some tips and tricks to guide you on writing a clean and efficient screencode

- A screen needs a certain refresh rate -> this can be achieved by updating the screen in interrupt at a certain frequency
    - One can either write the whole screen in every interrupt
    - Or update only one line per interrupt
- This frequency (refresh rate) should be chosen fast enough not to see any flickering but slow enough not to delay the main loop too much
- There are 7 physical rows, but for the implementation it can sometimes be easier to use 8 rows

1. Send 8 bits to the shiftregister in an efficient way

- Instead of counting the number of 1's and 0's you need to send to the screen
- We could preload a register RX with the 8 bits that need to be sent
- Example: we want to send 0-1-0-1-0-0-1-1 to the screen. We preload RX with 0b01010011
- One can now use the 'ROL – Rotate left through Carry' instruction



- After every ROL instruction the carry ('C') will contain the bit value that needs to be sent to the shiftregister.
- We need to copy the carry-bit (C) to PB3.
    - This cannot be done with an instruction immediately but using an instruction like branch-if-carry-is-set (BRCS) or branch-if-carry-is-cleared (BRCC)

The pseudo code becomes:

```
init:

LDI RY,0x08             ;need to shift 8 times
LDI RX,0b01010011       ;init the value we want to send to the shiftregister

ROL_loop8:
        CBI PORTB,3     ;init PB=0
    ROL RX              ;ROL
    BRCC CarryIs0       ;skip next line if C=0 => PB=0
        SBI PORTB,3     ;if C=1 => PB=1
    CarryIs0:
    CBI PORTB,5
    SBI PORTB,5         ;create rising edge of PB5 to shift

    DEC RY
    BRNE ROL_loop8      ;ROL and shift 8 times
```

- Using this trick, 8-bit data can easily be sent to the screen. The remaining part is making sure the correct data (depending on what you want to show) on the screen is in R16 before the ROL instruction.

2. Send row select

- To enable row1 we should send the following 8 bits to the shiftregister: 0-0-0-0-0-0-0-1 (0x01)
- To enable row2 we should send the following 8 bits to the shiftregister: 0-0-0-0-0-0-1-0 (0x02)
- To enable row3 we should send the following 8 bits to the shiftregister: 0-0-0-0-0-1-0-0 (0x04)
- …
- To enable row7 we should send the following 8 bits to the shiftregister: 0-1-0-0-0-0-0-0 (0x40)
- To enable 'row8' we should send the following 8 bits to the shiftregister: 1-0-0-0-0-0-0-0 (0x80)

- Using the trick mentioned above, one could send the data for the first row by initializing RX with 0x01 and executing 8 times the ROL instruction. The 8 steps are shown below:

	C		RX							
Init row1	X		0	0	0	0	0	0	0	1
ROL1	0		0	0	0	0	0	0	1	X
ROL2	0		0	0	0	0	0	1	X	0
ROL3	0		0	0	0	0	1	X	0	0
ROL4	0		0	0	0	1	X	0	0	0
ROL5	0		0	0	1	X	0	0	0	0
ROL6	0		0	1	X	0	0	0	0	0
ROL7	0		1	X	0	0	0	0	0	0
ROL8	1		X	0	0	0	0	0	0	0
Init row2	X		0	0	0	0	0	0	1	0

8 Bits that will be sent

- Now what happens if we use Rotate Right through Carry (ROR) instead of ROL?
  - If we start from the same starting value (0x01) this means we will turn on row8 first

	C		RX							
Init row8	X		0	0	0	0	0	0	0	1
ROR1	1		X	0	0	0	0	0	0	0
ROR2	0		1	X	0	0	0	0	0	0
ROR3	0		0	1	X	0	0	0	0	0
ROR4	0		0	0	1	X	0	0	0	0
ROR5	0		0	0	0	1	X	0	0	0
ROR6	0		0	0	0	0	1	X	0	0
ROR7	0		0	0	0	0	0	1	X	0
ROR8	0		0	0	0	0	0	0	1	X
Init row7	Y		0	0	0	0	0	0	1	0

8 Bits that will be sent

- One can observe that the value after ROR8 is exactly the value we need when we want to turn on row7 if we make sure the carry was initialised to '0' instead of X. After 8 more ROR's, RX will have the value we need when we want to turn on row6 and so on…

- So if we load only once 0x01 in RX, we can send the rowdata to the shiftregister by executing 8 ROR's and at the end RX is ready for the next row. In this way we will turn on row8->7->6->5->4->3->2->1

    - REMARK! After row1 the value in RX will be 0b00000000 and the carry will hold the '1' to turn on a row -> after sending row1 you need one ROL in order to restart with the correct value being 0b0000_0001 in RX.

The pseudo code becomes:

```
init:
    LDI RX, 0x01        ;init once the rowvalue

Send1Row:

    ; send column data
    ...

    ;send row data
    LDI RY,0x08             ;need to shift 8 times
    CLC                     ;make sure the carry is cleared before the first ROR
    ROR_loop8:
            CBI PORTB,3     ;init PB=0
        ROR RX              ;ROL
        BRCC CarryIs1       ;skip next line if C=0 => PB=0
            SBI PORTB,3     ;if C=1 => PB=1
        CarryIs1:
        CBI PORTB,5
        SBI PORTB,5         ;create rising edge of PB5 to shift

    DEC RY
    BRNE ROL_loop8          ;ROL and shift 8 times

    CBI PORTB,4
    SBI PORTB,4             ;rising edge at PB4 -> copy shift data to latch data
    ;delay
    CBI PORTB,4             ;PB4=0 -> Output enable

    TST RX                  ;Test RX for being 0 -> 8 rows send
    BRNE Send1Row           ;Send next row (7->6->5->4->3->2->1)
    ROL RX                  ;Reset RX to 0b00000001
    RJMP Send1Row           ;Send row8
```

3. Send column data

- There are mainly two methods to define the data that needs to be sent to the screen.
- One of them is using a screenbuffer. This is mostly useful in games where the screen can have an infinite ways to look without predefined patterns.
- The second one is by dividing the screen in different blocks with predefined characters. This is mostly useful in calculators, alarm clocks, piano's, text-writing projects,...

3.1 Using a screenbuffer

- Use the free SRAM memory starting from address 0x0100 to store a copy of your screen
  - 80bits/row * 7rows = 560 bits -> 70 bytes
- Every bit of that memory resembles one LED off the screen. For example address 0x0100 holds the 8 bits from the topleft of your screen (first row, 8 outer left leds), address 0x0101 holds the following 8 bits (first row, 8 next leds),…
  - Maybe other structures are better depending on your project -> think about it
- Every LED will have one bit at a certain location in this memory which we will call the screenbuffer
- Certain functions of your code will write to this memory, your screenfunction (in interrupt) will read this memory and send the data to the screen using the method explained above.
- To write to this memory one can use the following instructions:
  - ST X/Y/Z , Rr
    - Where X/Y/Z is a pointer to the correct address of the screenbuffer -> calculated before
    - Rr holds the value you want to write to that address
    - One can use "*ST X+*" to auto-increment the address after writing or use "*ST -X*" to auto-decrement the address before writing.
- To read from this memory one can use the following instructions:
  - LD Rd, X/Y/Z
    - Where X/Y/Z is a pointer to the correct address of the screenbuffer -> calculated before
    - Rd gets the value you will read from that address
    - One can use *"LD Rd, X+"* to auto-increment the address after reading or use "*LD Rd,- X*" to auto-decrement the address before reading

The pseudo code becomes:

```
;Send data to screenbuffer
    LDI ZL,0x00
    LDI ZH,0x01                     ;init Z to point to address 0x0100
    LDI Ra,0x55                     ;we will write this value to every byte of the screenbuffer

    LDI Rb,70                       ;need to write 70 bytes to fill the whole screenbuffer
WriteByteToScreenbuffer:
    ST Z+,Ra                        ;write value from Ra to address pointed by Z and auto-increase Z pointer
    DEC rb
    BRNE WriteByteToScreenbuffer    ;write 70 bytes

;Read data from screenbuffer for one row
    LDI ZL,0x00
    LDI ZH,0x01                     ;init Z to point to address 0x0100

    LDI Rb,70                       ;need to write 70 bytes to fill the whole screenbuffer
WriteByteToScreenbuffer:
    LD Ra,Z+                        ;write value from address pointed by Z to Ra and auto-increase Z pointer
    DEC rb
    BRNE WriteByteToScreenbuffer    ;write 70 bytes
```

- Depending on which part of the memory/screen one wants to read/write one needs to set the pointer correctly and decide how many bytes one needs to read/write

3.2 Predefined characters

- Every character can be defined in your code memory (flash memory) with the .db directive
  - Put this table at the end of your code
  - Every byte represents one line of a certain character on the screen
  - The first byte will be shown on row1, the second byte on row2,…
  - Also define an 8th byte for row8 being 0x00.

```
CharTable:
.db 0b00001111,0b00001001,0b00001001,0b00001001,0b00001001,0b00001001,0b00001111,0b00000000 ;0
.db 0b00000001,0b00000011,0b00000101,0b00000001,0b00000001,0b00000001,0b00000001,0b00000000 ;1
.db 0b00001111,0b00000001,0b00000001,0b00001111,0b00001000,0b00001000,0b00001111,0b00000000 ;2
```

  - One can use the free SRAM memory starting at address 0x0100 to store which character should be shown on the screen
    - Address 0x0100 will hold a value from the character that needs to be shown on the top-left block, address 0x0101 will hold a value from the character that needs to be shown on the second block,…
    - We need a memory of 16byte (1 byte per block on the screen)
      - We will call this the characterbuffer.
    - 'the values' written into this characterbuffer are the indexes in the character table.
    - 0x00 is to show a '0', … , 0x09 to show a '9', 0x0A to show 'A', … , 0x0F to show 'F', 0x10 to show 'G', … , 0x24 to show a 'Z', 0x25 and following to show other predefined symbols,…

  - To load the correct byte from the table we need:
    - Instruction: LPM Rd, Z
      - Where Z is a pointer to the correct byte in the CharTable
      - Rd will get the value where Z is pointing to.
    - Z can be calculated as follows:
      - Initialize Z to point to the first byte in the table

```
LDI ZH,high(CharTable<<1)   ;load address table of chars into Z
LDI ZL,low(CharTable<<1)    ;load address table of chars into Z
```

      - Increment Z to point to the correct character
        - Every character has 8 bytes to define its pattern
        - If we want to show a '1', we need to increment Z over the 8 bytes of character '0'. So we need to increment Z with 1*8
        - If we want to show a '2', we need to increment Z over the 8 bytes of '0' plus over the 8 bytes of '1'. So we need to increment Z by 2*8
        - If we want to show a certain character which is defined on the Xth line in the table, we need to increment Z by X*8.
        - => Z = Z + X*8
          - Where X is the value loaded from the characterbuffer
      - Increment Z to point to the correct byte of the character
        - Depending which row we are loading to the screen we need to select another byte of the character
        - For row1 we need the first byte, no need to increment Z
        - For row2 we need the second byte, increment Z by 1
        - For rowN we need the (N-1)th byte, increment Z by N-1
        - => Z = Z + X*8 + (rownumber-1)

- After pointing Z to the correct location, the byte can be loaded by using the LPM instruction. Rd now holds the value that needs to be sent to the screen (only the 5 rightmost bits are important as we will send one character to one LED-block of the screen being 5x7 leds
- These 5 bits can be sent to the screen with the tricks mentioned above

The pseudo code becomes:

```
;send column data

; init charbuffer pointer
    LDI YH,high(CHARBUFFER_END) ;last char should be sent first -> load address to Y
    LDI YL,low(CHARBUFFER_END)  ;last char should be sent first -> load address to Y
    LDI Ra,16                   ;number of blocks
;code for 1 block
BlockLoop:
    ;calc offset in table depending on charvalue
    LD Rb,-Y                    ;pre-decrement Y and load char value pointed to by Y (from data memory/SRAM)
    LDI ZH,high(CharTable<<1)   ;load address table of chars into Z
    LDI ZL,low(CharTable<<1)    ;load address table of chars into Z
    ;Calculate offset in chartable and add it to Z
    ...

    ;load column data
    LPM Rb,Z                    ;load data to send to the screen pointed to by Z (from code memory)
    LDI Rc,5                    ;number of shifts for column data per block = number of columns per display

    BlockColLoop:
    ;send 5 bits of loaded byte to the screen
    ...

    DEC Rc                      ;decrement Rc (number of shifts) (flagZ becomes 1 if equals 0)
    BRNE BlockColLoop           ;jump back for next data bit

DEC Ra                          ;decrement Ra (number of blocks) (flagZ becomes 1 if equals 0)
BRNE BlockLoop                  ;jump back for next block
```

!!! All the code examples are purely to illustrate the principles and instructions. This code is incomplete, not tested and will not work if copied as-is. Use these examples to understand the principles and use them as a guide for your own implementations.

With these tips and tricks you should be able to implement your screen in an efficient and clean way. If you have more questions feel free to send us an email or pass by our office.

Good luck!