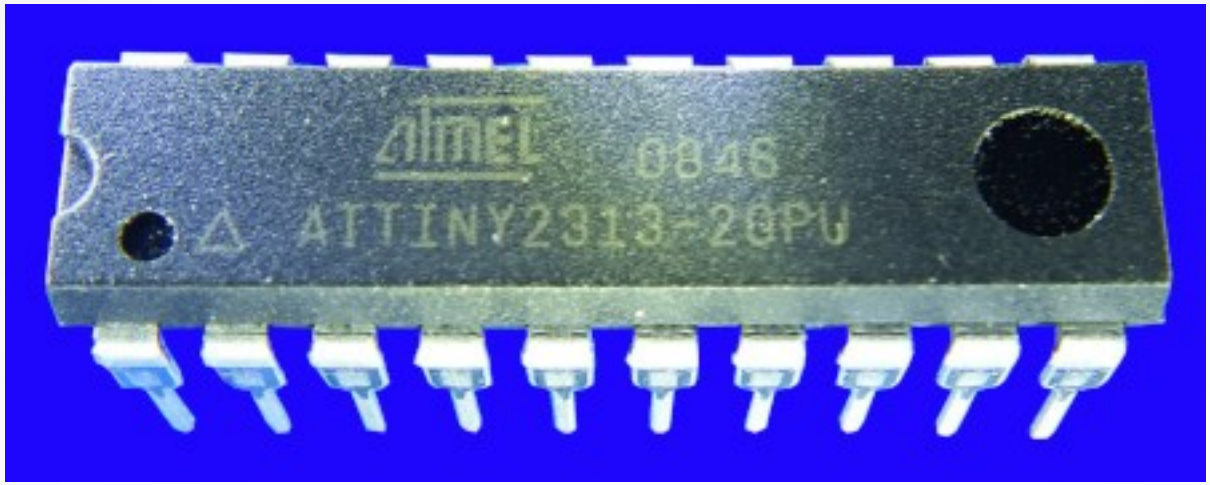


Lecture 4: The second blinker

Hardware, Internals and Programming of AVR Microcontrollers in Assembler

by
Gerhard Schmidt
Kastanienallee 20
D-64289 Darmstadt

The seconds blinker

- From the lecture 3 we know that blinking a LED in one second rhythm requires a 24 bit down counter.
- Blinking would require the following:
  - A LED attached to one of the pins of the ATtiny24 (preferably PB0) via a current-limiting resistor, either to GND (sourcing current) or to the operating voltage (sinking current).
  - The portpin has to be set as output pin.
  - The output has to be set (HIGH). Then a period of 500,000 clock cycles has to be absolved.
  - The output has to be cleared (LOW). Again, a period of 500,000 clock cycles has to be absolved.
  - The program then has to return to the HIGH-setting code.
- That implies the following: the delay period has to be executed two times. This can be resolved in three ways:
  1. We can write identical delay code in between the HIGH and the LOW period. This is the most simple way to solve this.
  2. We can write the code once, store it as macro code and place the macro call twice between the two I/O instructions.
  3. We can write the code once, store it as a subroutine and call this same code twice.
- Method 1 is trivial. The latter two methods are demonstrated here.
- Finally we learn how the assembled code is written to the controller.

The 24-bit loop

- **The source code for the 24-bit down-count loop is of course as follows:**

```
; Defining loop counts for 499,998 cycles
.equ cOuter = 15 ; the outer loop count
.equ cMiddle = 55 ; the middle loop count
.equ cInner = 201 ; the inner loop count
; Defining register names
.def rOuter = R16 ; defining the register for the outer loop
.def rMiddle = R17 ; defining the register for the middle loop
.def rInner = R18 ; defining the register for the inner loop
; Starting the outer loop
      LDI rOuter,cOuter ; Set register rOuter to the cOuter count value
LoopOuter:
      LDI rMiddle,cMiddle ; Set register rMiddle to the cMiddle count value
LoopMiddle:
      LDI rInner,cInner ; Set register rInner to the cInner count value
LoopInner:
      DEC rInner ; Decrease inner loop counter
      BRNE LoopInner ; Jump back to the inner label if not zero
      DEC rMiddle ; Decrease middle loop counter
      BRNE LoopMiddle ; Jump back to the middle label if not zero
      DEC rOuter ; Decrease outer loop counter
      BRNE LoopOuter ; Jump back to the outer label if not zero
; Very long loop done
```

Execution times

- **The clock cycles of the inner loop are again CCi = 3 * cInner.**
- **The clock cycles of the middle loop are:**
  CCm = 1 + (cMiddle – 1) * (CCi + 3) + CCi + 2
  CCm = 1 + cMiddle * CCi + 3 * cMiddle – CCi -3 + CCi +2
  CCm = cMiddle * CCi + 3 * cMiddle
  CCm = 3*cMiddle*cInner + 3 * cMiddle
- **The clock cycles of the outer loop are:**
  CCo = 1 + (cOuter – 1) * (CCm + 3) + CCm + 2
  CCo = 1 + cOuter * CCm + 3 * cOuter – CCm – 3 + CCm + 2
  CCo = cOuter * CCm + 3 * cOuter
  CCo = cOuter * (3 * cMiddle * cInner + 3 * cMiddle) + 3 * cOuter
  CCo = 3 * cOuter * cMiddle * cInner + 3 * cOuter*cMiddle + 3 * cOuter
  CCo = 3 * cOuter * (cMiddle * cInner + cMiddle + 1)
- **An optimal combination to achieve around 499,998 clock cycles is:**
  cOuter = 15 ; cMiddle = 55 ; cInner = 201 ; CCo = 499,995

Delay loop as macro

- **The following makes a macro for the loop code:**

```
.macro delay
; Starting the outer loop
      LDI rOuter,cOuter ; Set register rOuter to the cOuter count value
LoopOuter:
      LDI rMiddle,cMiddle ; Set register rMiddle to the cMiddle count value
LoopMiddle:
      LDI rInner,cInner ; Set register rInner to the cInner count value
LoopInner:
      DEC rInner ; Decrease inner loop counter
      BRNE LoopInner ; Jump back to the inner label if not zero
      DEC rMiddle ; Decrease middle loop counter
      BRNE LoopMiddle ; Jump back to the middle label if not zero
      DEC rOuter ; Decrease outer loop counter
      BRNE LoopOuter ; Jump back to the outer label if not zero
; Very long loop done
.endmacro
```

- **When the assembler sees the directive .macro he grabs the following text until a line saying „.endmacro“ or „.endm“ occurs.**
- **If the macro‘s name („delay“) is found instead of a mnemonic this text is inserted into the source code as if it it were placed there – and assembled.**

Calling the macro

- **Calling the macro is simple:**

```
; Defining loop counts for 499,998 cycles
.equ cOuter = 15 ; the outer loop count
.equ cMiddle = 55 ; the middle loop count
.equ cInner = 201 ; the inner loop count
; Defining register names
.def rOuter = R16 ; defining the register for the outer loop
.def rMiddle = R17 ; defining the register for the middle loop
.def rInner = R18 ; defining the register for the inner loop
; (Place the macro delay here)
      sbi DDRB,PORTB0 ; PB0 as output
LedLoop:
      sbi PORTB,PORTB0 ; Output pin high
      DELAY ; Insert the macro content here
      cbi PORTB,PORTB0 ; Output pin low
      DELAY ; Insert the macro content the second time
      RJMP LedLoop
```

- **Note that the macro code is inserted twice, which consumes flash memory (of which the ATtiny24 has plenty of).**
- **Unfortunately avr_sim cannot set and handle breakpoints inside macros, so you cannot measure execution times separately.**

Where to get cOuter etc. from?

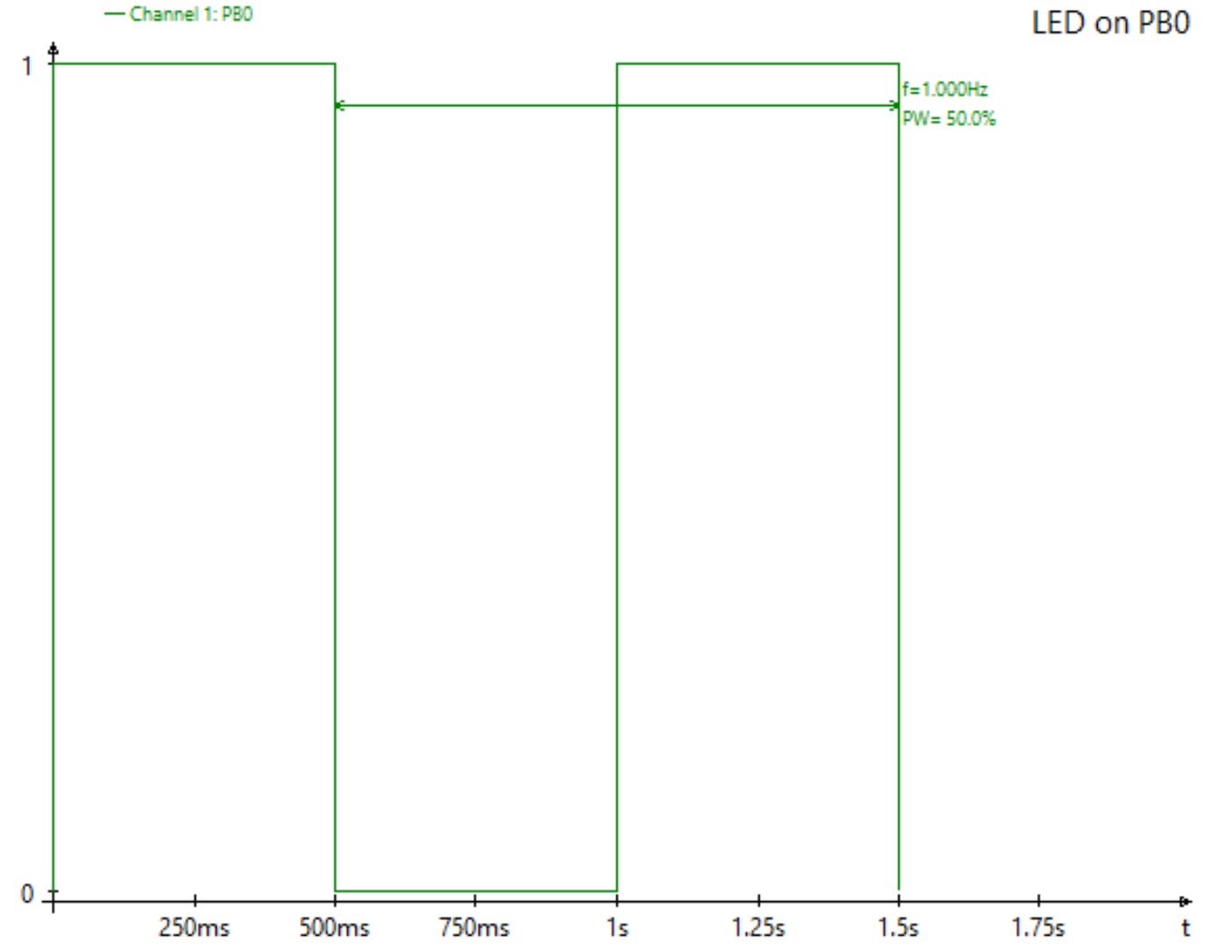
- **Where to get those constants from? I used MS-Excel and some VBA code:**

```
Private Sub CommandButton1_Click()
Dim cOut As Long, cMid As Long, cInn As Long
Dim Should As Long, Value As Long, Delta As Long, DeltaMin As Long
Dim Out As Long, Mid As Long, Inn As Long, Val As Long
Should = Cells(3, 1)
DeltaMin = 9999999
For cOut = 0 To 255
  For cMid = 0 To 255
    For cInn = 0 To 255
      Value = 3 * cOut * cMid * cInn + 3 * cOut * cMid + 3 * cOut
      Delta = Abs(Value - Should)
      If Delta < DeltaMin Then
        Val = Value
        DeltaMin = Delta
        Out = cOut
        Mid = cMid
        Inn = cInn
        End If
      Next cInn
    Next cMid
  Next cOut
Cells(3, 2) = Out
Cells(3, 3) = Mid
Cells(3, 4) = Inn
Cells(3, 5) = DeltaMin
Cells(3, 6) = Val
End Sub
```

- **Note that there are many more combinations that yield approximately the target value.**
- **68 combinations are within a distance of +/- 4 to 500,000.**
- **84 are within a distance of +/- 5.**

The second blinker – as simulated

- **This is what you get from the source code within the simulator:**



- **A rather exact signal frequency of 1.000 Hz - But in reality?**
- **The internal RC oscillator has +/-3% accuracy at 3.3 Volt operating voltage, but +/-10% at 5 Volt.**
- **So your real controller will produce something in between 0.9 and 1.1 Hz by default.**
- **The difference cannot be seen, but rather can be measured.**
- **If you need it more exact: clock your ATtiny24 with a crystal or a crystal oscillator – see the next pages.**

Subroutines

- The second method of using the same source code twice (or more times) and writing it once is to formulate it as subroutine. Calling it is simple:

```
; Calling a subroutine
      RCALL MySub
; Continue here if MySub has finished execution
```

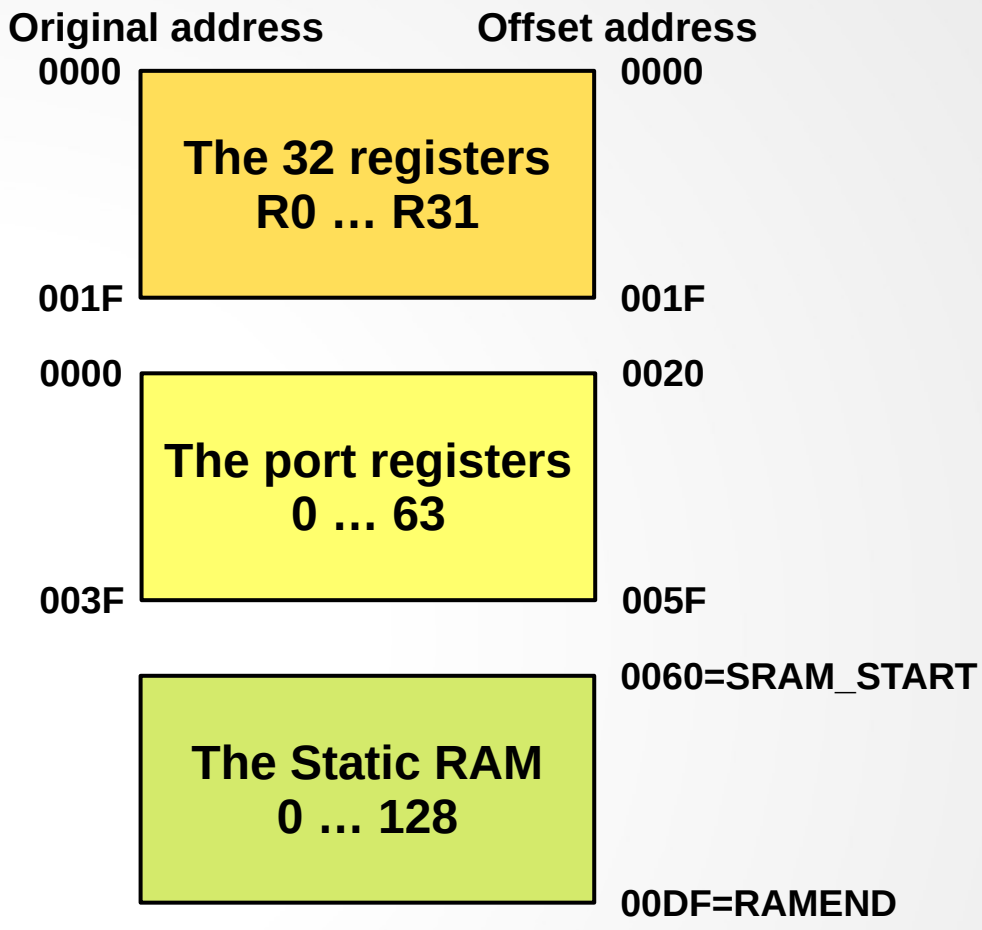
- Subroutines start with a label (where they can be called) and end with RET (for RETurn):

```
; My subroutine
MySub: ; Label for calling the subroutine
      ; The subroutine does something here
      RET ; Control goes back where it was called from
```

- The subroutine has to know where it was called from, it‘s return address. The place where this is stored is called the „Stack“.
- The stack is a memory space at the end of the Static RAM (SRAM) of the controller. The def.inc file provides a constant called „RAMEND“ for this. In the ATtiny24 RAMEND is 0x00DF (decimal 223).

Memory organization of AVRs

- This demonstrates how the memory space in AVR is organized in an ATtiny24.
- Not shown are
  - the flash memory where the binary code is stored, starting at 0000,
  - the EEPROM memory where durable data can be stored, also starting at 0000.
- The other memories are stapled:
  - The 32 registers from 0000 to 001F appear on their original addresses.
  - The 64 port registers, originally addressed from 0000 to 003F, appear at 0020 etc.
  - The static RAM starts at SRAM_START and ends at RAMEND.



- This organization allows to access all memory components of the AVRs at any time and from every binary code location. Memory is completely unprotected, and it is up to the program writer to preserve needed information from being changed in an uncontrolled way.

Initiating the stack

- Initiating the stack should be the first thing on top of the executable code.
- It differs a little bit depending from the size of SRAM memory that the device has:
  - Devices with 256 bytes SRAM or less (like the ATtiny24) have one port register for the stack pointer. It is named SPL (for Stack Pointer Low).

```
; Init stack for devices with small SRAM
      LDI R16, Low(RAMEND) ; Load the LSB of RAMEND to R16
      OUT SPL,R16 ; Write R16 to the LSB of the stack pointer
```

  - Devices with more than 256 bytes STAM have an additional port register named SPH (H for High). It holds the Most Significant Byte (MSB) of the stack pointer.

```
; Init stack for devices with large SRAM
      LDI R16, High(RAMEND) ; Load the MSB of RAMEND to R16
      OUT SPH,R16 ; Write R16 to the MSB of the stack pointer
      LDI R16, Low(RAMEND) ; Load the LSB of RAMEND to R16
      OUT SPL,R16 ; Write R16 to the LSB of the stack pointer
```

- From now on the stack is ready for use, and calling addresses can be stored there. The whole program for 0.5-seceonds-delay-counting looks like this:

Use of the stack for address storage

```
; Init the stack for the ATtiny24 device
      LDI R16, Low(RAMEND) ; Load the LSB of RAMEND to R16
      OUT SPL,R16 ; Write R16 to the LSB of the stack pointer
; Make PB0 output
      SBI DDRB, DDB0 ; Portpin PB0 as output
; The second loop blinks
Second: ; Label for jump back
      SBI PORTB, PORTB0 ; Make port pin PB0 high
      RCALL Delay ; Call the subroutine DELAY
      CBI PORTB, PORTB0 ; Make port pin PB0 low
      RCALL Delay ; Again call the subroutine DELAY
      RJMP Second ; Restart the second loop
; (Continued on the next page)
```

- The difference to the macro code above are the two instructions RCALL. This is a single-word instruction, but consumes three clock cycles.

- It does the following:
  - It throws the LSB of the program counter PC onto the current stack position and decreases the Stack Pointer by one.
  - Then it throws the MSB of the PC onto this next stack position and again decreases the Stack Pointer by one.
  - It loads the relative address of the subroutine „Delay“ onto the PC.
  - The next instruction executed is the first instruction of the subroutine.

The subroutine

```
; The subroutine DELAY
Delay: ; Label for calling the subroutine
; Starting the outer loop
	LDI rOuter,cOuter ; Set register rOuter to the cOuter count value
LoopOuter:
	LDI rMiddle,cMiddle ; Set register rMiddle to the cMiddle count value
LoopMiddle:
	LDI rInner,cInner ; Set register rInner to the cInner count value
LoopInner:
	DEC rInner ; Decrease inner loop counter
	BRNE LoopInner ; Jump back to the inner label if not zero
	DEC rMiddle ; Decrease middle loop counter
	BRNE LoopMiddle ; Jump back to the middle label if not zero
	DEC rOuter ; Decrease outer loop counter
	BRNE LoopOuter ; Jump back to the outer label if not zero
; Very long loop done
	RET ; RETurn to the caller
```

- Even though this looks pretty much like the macro from above, it is working completely different:
  - It ends with a RET instruction.
  - It is only once coded (and not twice like the macro).

Demonstration with the simulator

- Assemble the program and start simulation.
- In the View hardware section click on „SRAM“.
- Click „Step“ twice and see the changed stackpointer entry in the Status window.
- Then click „Step“ twice again and the cursor is now on the RCALL.
- With the next „Step“ three things happen:
  1. The highest two bytes of the SRAM display the calling address, that has been thrown onto the stack.
  2. The stackpointer in the Status window is now by 2 positions below its inital value.
  3. The next executable instruction is LDI rOuter, cOuter in the delay subroutine.
- In order to not having to wait a half second, click „Skip“ until the RET instruction is executed next.
- When RET is executed, the following happens:

  1. The PC is loaded with the address on top of the stack, the next executable instruction is the one that follows the first RCALL instruction.

  2.The stackpointer is at its initial value.

  3.The SRAM content remains unchanged.
- Note that the RET consumes four clock cycles, so don‘t write subroutines just for fun, if your program has to perform very fast.

Conclusions

- **There are always many different modes that a task can be resolved with.**
- **As with all programming the arousing twist is optimization: does the source code what is required (functionality and correctness), is it easy to understand (readability, understandability) and is it nice and aesthetic (elegancy)? (Spaghetti code is always the opposite of the latter two, even it works correct.)**
- **Assembler offers much more optimization opportunities than Higher-Level languages because of its „Anything is allowed, be your own master“ attitude.**
- **The counter-side of this freedom is:**
  1. **You can easily produce spaghetti code, no built-in conventions or safety functions hinder you.**
  2. **You are solely responsible for where you place data and variables to and to protect those. No mechanism warns you, if you overwrite data needed later on.**

Questions and tasks in Lecture 4

Task 4-1: Write a program that counts three registers up until they reach a target value. Use the assembler functions BYTE1, BYTE2 and BYTE3 to derive the three bytes from the constant:

```
; Deriving single bytes from a constant
.equ constant = 1234567 ; = hexadecimal 0x12D687
.equ b1 = BYTE1(constant) ; Yields 0x87
.equ b2 = BYTE2(constant) ; Yields 0xD6
.equ b3 = BYTE3(constant) ; Yields 0x12
```

Use the instruction CPI (ComPare with Immediate) and the Z flag to find out, if the value has been reached. Try if you can derive a formula for the clock cycles and verify with the simulator if your formula is working correct and exact.

(Note: This task is solvable!)

Questions and tasks in Lecture 4 - Continued

Task 4-2: Try to change the code so that the three registers down-count the initial constant value.

Hint: Do not use the instruction DEC (for DECrease) because it does not affect the overflow flag C (for Carry). Use the instructions SUBI (for SUBtract Immediate) and BRCC (BRanch on Carry flag Clear) for conditional jumps.

Try to derive the formula for the number of clock cycles.

Another hint: The formula to derive the consumed clock cycles for that might get a little complicated, so give up if unsuccessful for longer than two hours. Genius approaches are welcome!

Questions and tasks in Lecture 4 - Continued

Task 4-3: If your crystal (next lecture) would have a frequency of 4 MHz, what would be the constants for

a) cOuter, cMiddle and cInner for the method first introduced here,

b) for the up-counting, and

c) for the down-counting.

Try verification with avr_sim, by inputting the clock frequency and disabling the CLKDIV8 fuse (but use a very fast computer for that, place breakpoints, let avr_sim work over night and disable the operating system‘s sleep features!)