Applications of
AVR single chip controllers AT90S, ATtiny, ATmega and ATxmega

DCF77 synchronized alarm clock with LCD Date and time

Date and time with an AVR in assembler

Sometimes it happens that one has to handle a date or time on an AVR, e.g. if programming an alarm clock. While C programmers have their large and mighty libraries (and do not understand what those do at all), the assembler programmer has to do it by himself. And it is pretty simple.

0 Content

1 To measure seconds as correct as necessary

1.1 Loops for timing

If the AVR has nothing else to do than this (this happens rarely) one can send the AVR into an endless count to come to seconds, minutes, hours, days and years. Fortunately an AVR is a rather silent controller and he will not cry or complain about having to do such nonsense like counting down the seconds in a 28-bit register series from 0x01E13380 down to zero before advancing the year by one (or rather 0x01E28500 if it is a leap year). Or, if he runs at a clock frequency of 1 MHz, to absolve 0x1CAE8C13E000 useless clock cycles using a 48-bit wide down-counter. He will be quit during his job, and will do as the assembler programmer has told him.

For counting down a second at 1 MHz clock frequency a counter is needed that can count-down from 300,000 to zero. For this we need 20 bits, a single byte or a 16-bit counter is insufficient for that. Such a counter in assembler can be constructed as down-counter or as up-counter. A down-counter has first to set a delay loop constant, then to down-count (LSB first, if carry: MSB next, if carry HSB last. Hint: use SUB or SUBI to down-count instead of DEC because DEC does not influence the carry flag when the register is decreased from zero to 0xFF.

That would produce a source code like this:

```
  ldi R16,BYTE1(cDelay) ; Set start values in R18:R17:R16
  ldi R17,BYTE2(cDelay)
  ldi R18,BYTE3(cDelay)
DelayLoop:
  subi R16,1 ; Down-count LSB
  brcc DelayCheck ; No carry: check zero
  subi R17,1 ; Down-count MSB
  brcc DelayCheck ; No carry: check zero
```

```
  subi R18,1 ; Down-count HSB
DelayCheck:
  tst R16 ; Check LSB
  brne DelayLoop
  tst R17 ; Check MSB
  brne DelayLoop
  tst R18 ; Check HSB
  brne DelayLoop
DelayEnd:
  ; One second is over
```

When up-counting the source code is similar:

```
  clr R16 ; Start from zero
  clr R17
  clr R18
DelayLoop:
  subi R16,-1 ; up-count LSB, carry flag reversed!
  brcs DelayCheck
  subi R17,-1 ; Up-count MSB, carry flag reversed!
  brcs DelayCheck
  subi R18,-1 ; Up-count HSB, carry flag reversed!
DelayCheck:
  cpi R16,BYTE1(cDelay)
  brne DelayLoop
  cpi R17,BYTE2(cDelay)
  brne DelayLoop
  cpi R18,BYTE3(cDelay)
  brne DelayLoop
DelayEnd:
  ; One second over
```

Now, which number cDelay should be chosen for a second? For that we need to know how many clock cycles the whole loops require. As each BRCC/BRCS and BREQ/BRNE needs two clock cycles in case it jumps and one clock cycle in case it does not jump, a very sophisticated formula would be needed to calculate the complete number of cycles from a constant cDelay, and vice versa.

The only way out of this is to construct the different branches in a way that they consume the same number of execution cycles. Each line executed is analyzed for its clock cycles and, with the help of NOP instructions and jumps, delayed so that each branch needs the same clock cycles. In case of the down-counting the subtraction phase so would look like this:

```
DelayLoop: ; 0 clock cycles
  subi R16,1 ; + 1 = 1 clock cycle
  brcc Delay1 ; + 1 = 2 for not jumping, + 2 = 3 for jumping
  subi R17,1 ; + 1 = 3 clock cycles
  brcc Delay2 ; + 1 = 4 for not jumping, +2 = 5 for jumping
  subi R18,1 ; +1 = 5 clock cycles
  brcc Delay3 ; +1 = 6 for not jumping, + 2 = 7 for jumping
  rjmp DelayCheck ; + 2 = 8 clock cycles
Delay1: ; 3 clock cycles
  nop ; + 1 = 4 clock cycles
  nop ; + 1 = 5 clock cycles
Delay2: ; 5 clock cycles
  nop ; + 1 = 6 clock cycles
  nop ; + 1 = 7 clock cycles
Delay3: ; 7 clock cycles
```

```
  nop ; + 1 = 8 clock cycles
DelayCheck: ; 8 clock cycles
  ; (Now check if all zero)
```

Now the complete execution between DelayLoop: and DelayCheck: needs 8 clock cycles, where-ever all those branches run along. A predictable execution time for this. Now the same for the DelayCheck: that checks if all registers are zero:

```
DelayCheck: ; 8 clock cycles
  tst R16 ; + 1 = 9 clock cycles
  brne DelayCont1 ; + 1 = 10 for not jumping, + 2 = 11 for jumping
  tst R17 ; + 1 = 11 clock cycles
  brne DelayCont2 ; + 1 = 12 for not jumping, + 2 = 12 for jumping
  tst R18 ; + 1 = 13 for not jumping, + 2 = 14 for jumping
  breq DelayEnd ; + 1 = 14 for not jumping, + 2 = 15 for jumping
  rjmp DelayLoop ; + 2 = 16
DelayCont1: ; 11 clock cycles
  nop ; + 1 = 12 clock cycles
DelayCont2: ; 12 clock cycles
  nop ; + 1 = 13 clock cycles
  nop ; + 1 = 14 clock cycles
  rjmp DelayLoop ; + 2 = 16
DelayEnd: ; 15 clock cycles
  nop ; + 1 = 16 clock cycles
  ; One second is over
```

So the whole number of clock cycles N needed is

$$N = 3 + 16 * cDelay$$

of which the 3 is for the initial phase of setting the registers, and cDelay for one second can be calculated simply like this:

$$cDelay = (N - 3) / 16$$

The result for N = 1,000,000 is 62,499 (rounded down), which corresponds to 3 + 62,499 * 16 = 999,987 cycles. If you need to be more exact (see accuracy discussion below), you can add 13 NOP instructions at the end.

Note that in our example R18 is zero, because 62,499 fits into two registers and we can skip that, but without removing the code (because that would change code execution times and with a shorter loop would go beyond the 16 bit limit).

With those instruments we can construct whatever type of counting loops (three, four, five, etc. bytes) to fit to any time (minutes, hours, days and years) to arrive at an exact timing.

1.2 Timer as clock counter

Get rid of the boring counting by the use of built-in timers. Those timers can count up (and sometimes down, too) and they are very reliable and predictable. They are in any case doing their jobs even if numerous interrupts or other tasks happen in between.

Unfortunately timers and their prescalers work in binary mode, so they like the two's more than the ten's. If our clock rate is at 1 MHz (which is more the ten's domain) a timer which divides by 64 yields 15,625. The next higher binary divider, 128, already has a frequency which is not an integer value but has a fraction following. Not to speak about divider rates of 256, 2,048 or 16,384 or even higher (for which all timers have a prescaler).

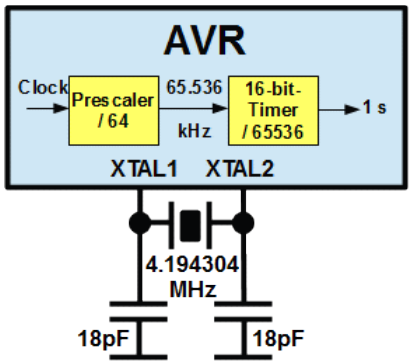
1.2.1 16-bit timer with a crystal

As a clock device needs a crystal anyway to be exact enough, we can select xtal frequencies that better fit to binaries, e.g. 2.048 MHz, 2.097152 MHz, 2.4576 Mhz, 3.072 MHz, 3.2768 MHz or 4.194304 MHz.

This here are the divider rates when using possible prescaler values in 8- and 16-bit timers. We see that the 8-bit timer has rather low divider rates even with a prescaler of 1,024. To come down to one second our xtal would have to be either of the 32,768 Hz type or hand-crafted to our selected frequency. And: the controller would act but would act like a lame duck at those low frequencies.

Prescaler	Timer 8-Bit	Divider	Timer 16-Bit	Divider
1	256	256	65.536	65.536
8		2.048		524.288
64		16.384		**4.194.304**
256		65.536		16.777.216
1024		262.144		67.108.864

More promising is a 16-bit timer with a prescaler of 64. Xtals with a frequency of 4.194,304 MHz are sold on each street corner or electronics store and cost around 25 Cents. With that the seconds timer is already complete, just do the following:



1. attach the Xtal and two ceramic capacitors of 18 pF to GND to an AVR that has an internal Xtal oscillator,
2. change the fuses of the AVR to use an external xtal with medium or high frequency,
3. copy the interrupt vector table of the controller with all *RETI* except for the Reset vector (*RJMP Start*) and the overflow interrupt for TC1 (*RJMP TC1OvfIsr*),
4. write an overflow interrupt service routine "TC1OvfIsr" for the timer with the two instructions *SET* (set the T flag in the status register) and *RETI* (Return from Interrupt),
5. in the main program "Start:" init the stack pointer,
6. then set the 16-bit timer to normal operation and the prescaler to 64,
7. enable TC1 overflow interrupts (TOIE1) and the interrupts generally with *SEI*, and
8. ask in a loop if the seconds flag T is set, if yes clear it with *CLT* and do what you'll want to do in each second.

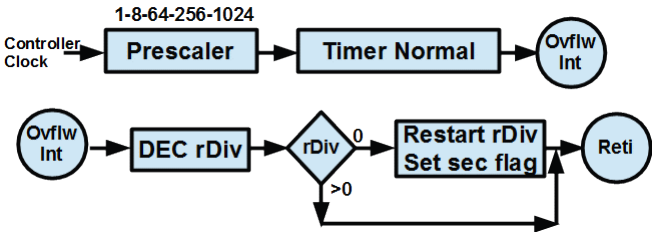
That is all you need for a xtal controlled clock. Anything else that you want to do with that seconds pulse, e.g. advancing time and date counters, displaying on an LCD or on a 7-segment display, can last as long as necessary: the timer works correct if that does not last longer than two seconds. Which is rather long, even with a lame-duck type of 32.768-kHz-oscillator.

- Those who find a 4.1 MHz clock too fast and want to save battery current, or
- who do not find such an Xtal on their street corner,
- who need the 16-bit timer for other more valuable purposes, e.g. to play a melody with it when the clock reaches 07:30 in the morning,

can find other solutions for that.

1.2.2 8-bit timer with a register divider

In any case the selected xtal frequency divided by the prescaler and by the timer used has to be a pure integer value. If the result of the division is not one but an integer value: use an 8-bit (smaller or equal 256) or a 16-bit register (256 to 65,536)



to divide the resulting frequency down to one.

With the above named xtals an 8-bit timer has the following register divider values.

These are the crystals that can be used with an 8-bit timer, the different prescalers and a register divider. Red marked register dividers are 16-bit, green marked ones are 8 bit. Only at 3.072 MHz a 16-bit divider is necessary.

Xtal frequency (Hz)	2048000	2097152	2457600	3072000	3276800	4194304
8-Bit, ./. 256	8000	8192	9600	12000	12800	16384
Presc, ./. 8	1000	1024	1200	1500	1600	2048
Presc, ./. 64	125	128	150		200	256
Presc, ./. 256		32			50	64
Presc, ./. 1024		8				16

In Assembler dividing within the overflow interrupt service routine goes like this:

```
  ; Calculation constants
  .equ clock = 2097152 ; Xtal frequency
  .equ prescaler = 1024
  .equ timertop = 256
  .equ second_divider = clock / timertop / prescaler ; = 8
  ;
  ; Registers
  .def rSreg = R15 ; Save status
  .def rSeconddivider = R17 ; Counter down
  .def rFlag = R18 ; Flag register
  .equ bSek = 0 ; Second flag in flag register
;
Tc0OvflwInt:
  in rSreg,SREG ; Save SREG
  dec rSeconddivider ; Divide by 8
  brne TC0OvflwIntRet ; Not yet zero
  ldi rSecondendivider,second_divider ; Restart divider
  sbr rFlag,1<<bSek ; Set seconds flag: one second is over
Tc0OvflwIntRet:
  out SREG,rSreg ; Restore SREG
  reti
```

1.2.3 Timer in CTC mode

The method does not fit well if you are addicted to a certain clock frequency due to other reasons. Only at 4.0 MHz and an 8-bit timer dividing by 256 the integer 15.625 results. This can be divided in a 16-bit register, such as R25:R24, by use of the instruction *SBIW R24,1*. When the timer reaches zero, it is restarted with 15,625 and the second is over. This does not work on many other frequencies. But for those there is another method available.

Many nice clock frequencies such as 1 or 2 MHz do not work with those methods. In those cases the timer has to be made more ten-friendly by clearing it when he reaches a count of 100 or 250. This can be done in the so-called CTC mode (Clear Timer on Compare). If the counter reaches the value that is stored in its Compare Port Register COMPA, the next count will restart the timer at zero (in 16-bit timers the Input Capture Port Register ICR can also be used for the comparison purpose). The one extra clock cycle means that the compare value has to be one smaller than the desired division rate. If you need the same timer also for other purposes, e.g. as a pulse width generator, the CTC mode setting also alters its resolution.

With 1 MHz clock the prescaler can be set to 64 (which means 15,625 kHz on the timer input), can set the compare value to 124 (dividing with CTC by 125) and dividing with a register divider by 125 to yield one second. Many other frequencies can be counted down like that to an exact second.

Provided that we have

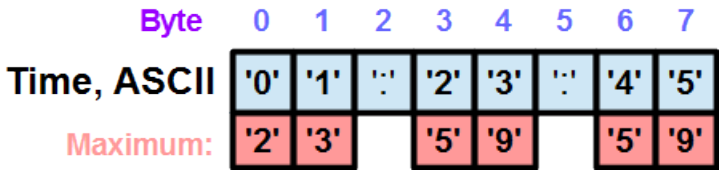
- the counter running,
- its interrupts enabled (TOIE0/TOIE1 with an overflow int resp. OCIE0A/OCIE1A with a compare match A int),
- the main interrupt flag I in SREG is set,
- the interrupt vector is properly set and branches to the service routine, and
- the stack is initiated and works properly.

2 Date and time formats

Dates and times can be handled in four different usual formats. Fortunately all those formats can be converted into each other.

Instead of dividing the day into two halves (AM and PM) we use here the European notation for hours (0 to 23).

2.1 Time in ASCII format



This is the simplest format in respect to displaying the time on an LCD: each digit in one byte, encoded as ASCII character. ASCII has been designed for some US teletype machines, so it has some interesting control codes between characters 0 and 31, such as BEL for wringing a bell on your teletype - ahem, computer. Or the character number 13, which moves the write head to the left side to the margin (carriage return). Or number 10, that moves the printing paper of the teletype printer by one line up (linefeed). As those low level codes of ASCII were already occupied for this kind of pseudo characters the numbers start with code number 48 for zero and reach until 57 for nine. The rest of the ASCII codes is not used for time encoding, only the ':' character makes some sense here.

Writing the numbers as '0' to '9' means that the ASCII representation is meant. Even assemblers understand that:

```
  ldi R16,'0' ; Load ASCII 0
  ldi R17,'7' ; Load ASCII 7
  ldi R18,':' ; Load : character as separator
```

If we want to write eight characters to SRAM we reserve space in SRAM for it:

```
.dseg
TimeAscii:
.bytes 8
```

and write the following code:

```
.cseg
  ldi ZH,HIGH(TimeAscii) ; Point to SRAM address
  ldi ZL,LOW(TimeAscii)
  ldi R16,'0' ; Load ASCII 0
```

```
  st Z+,R16 ; Store in SRAM and increase address
  ldi R16,'1'
  st Z+,R16
  ldi R16,':'
  st Z+,R16
  ldi R16,'2'
  st Z+,R16
  ldi R16,'3'
  st Z+,R16
  ldi R16,':'
  st Z+,R16
  ldi R16,'4'
  st Z+,R16
  ldi R16,'5'
  st Z,R16
```

A different formulation for that would be:

```
  ldi YH,High(TimeAscii) ; MSB of the SRAM address to YH
  ldi YL,Low(TimeAscii) ; LSB to YL
  ldi R16,'0' ; ASCII-Null in R16
  st Y,R16 ; To Hour-Tens
  inc R16
  std Y+1,R16 ; To Hour-Ones
  inc R16
  std Y+3,R16 ; To Minutes-Tens
  inc R16
  std Y+4,R16 ; To Minute-Ones
  inc R16
  std Y+6,R16 ; To Second-Tens
  inc R16
  std Y+7,R16 ; To Second-Ones
  ldi R16,':' ; Separator to R16
  std Y+2,R16 ; To the first separator location
  std Y+5,R16 ; To the second separator location
```

STD (and when reading: LDD) does not change Y but temporarily adds the displacement behind + and writes the byte in R16 to this location. This works with Y and Z, but not with X.

To increase the second-ones by one second. If this leads to the ASCII character behind '9', it has to restart with '0' and has to increase the second-tens. With the constant address in Y that goes like that:

```
  ldi YH,High(TimeAscii) ; MSB of the SRAM address to YH
  ldi YL,Low(Zeit) ; LSB to YL
  ldd R16,Y+7 ; Read second-ones to R16
  inc R16 ; Increase second-ones by one
  std Y+7,R16 ; Write increased second-ones
  cpi R16,'9'+1 ; Compare with ASCII code for next char behind nine
  brcs Done ; If carry set no overflow to next higher digit
  ldi rmp,'0' ; Restart second-ones
  std Y+7,R16 ; Write ASCII-zero to second-ones
  ldd R16,Y+6 ; Read second-tens
  inc R16 ; Increase second-tens
  std Y+6,R16 ; and write back to second-tens
  cpi R16,'6' ; Second-tens at six?
  brcs Done ; No, ready
  ; ... Minutes and hours similarly
Fertig:
```

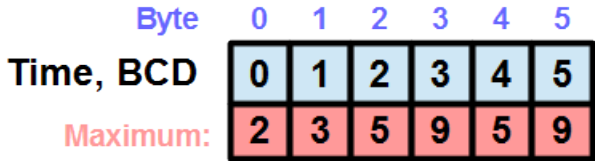
```
; ... Done with the clock increase
```

When increasing the hour, those two criteria come into play:

- if the hour-ones are larger than '9', and
- if the hour-ones are four AND the hour-tens are '2'.

By using the relative addressing with STD and LDD time increasing gets rather simple.

2.2 Time in BCD format



With this second method the ones and tens of seconds, minutes and hours are not stored as ASCII characters but as binary encoded decimal digits (BCD). Those bytes range from binary 0 (0b00000000) to binary 9 (0b00001001) for the ones, from 0 to 5 for the second- and minute-tens and from 0 to 2 for the hour-tens.

The comparison if the ones have exceeded the nine is now done with the instruction *CPI R16,10* instead of *CPI R16,'9'+1*. Restart of the ones is done with *CLR R16* instead of *LDI R16,'0'*. Anything else remains the same, but the ':' makes no sense any more and is inserted when displaying the time but has no own SRAM location.

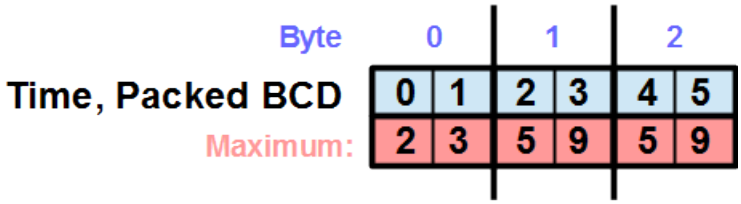
When displaying the BCD codes on the LCD one simply has to add 48 to the BCD. Because there is no *ADDI* instruction on the AVR, there are three different ways around for that:

1. We write the 48 to another register (e.g. *LDI R17,48*) and add this register to the BCD in R16: *ADD R16,R17*.
2. Set the bits bits 4 and 5 in the BCD with either *ORI R16,0x30* or with *SBR R16,0x30* or with *SBR R16,(1<<4)|(1<<5)*.
3. Subtract -48 from the BCD with *SUBI R16,-'0'*. Subtracting a negative number is the same as adding the positive number.

All three methods have the same result. Only the first methods is different because it requires an additional register.

When displaying on the LCD do not forget to insert the separator on the correct location, otherwise the time would look a bit strange.

2.3 Time in packed BCD format



Because a BCD needs only four bits, you can pack two of those into one byte - and save some memory and registers. The ones are fine in the lower four bits (0 to 3), the tens fit into the upper four bits (4 to 7). The package of four bits are called lower and upper "nibble". The complete set of time information now fits into three bytes. The format is called "packed BCD".

If we want to increase such a second, minute or hour we can use *INC R16*, too. But it is more complicated then to detect whether the lower nibble exceeded 9: first we would have to copy the register (*MOV R17,R16*), then we have to clear the upper nibble (*ANDI R17,0x0F*) and then we can compare this with 10 (*CPI R17,10*) to decide if we would have to add (0x10 - 10) to R16, by that clearing the lower nibble and increasing the upper nibble.

The more simple solution for that uses a special hardware feature that each CPU, including the AVR CPUs, has: the half-overflow flag H in the status register. This flag bit signals if

during adding an overflow from the lower to the upper nibble occurred. To use this feature we add 7 to the lower nibble: if the lower nibble was nine before adding, a half overflow would occur setting the H flag. The upper nibble would be increased and would already be fine. In case that the lower nibble was not nine H is clear and we would have to subtract 6 from the result. The conditional branching instructions BRHC and BRHS can be used.

The source code would be:

```
  ; Increase lower nibble of R16
  ldi R17,7 ; Add 7 to packed BCD
  add R16,R17 ; in R16
  brhs DoNotSubtract6
  ldi R17,6 ; Subtract 6 from packed
  sub R16,R17 ; from R16
DoNotSubtract6:
  ; Done
```

To save one register (R17) we can use *SUBI R16,-7* instead, but the H-bit now is also reversed: it is cleared when a half overflow occurred and set if not. So BRHS will have to be changed to BRHC.

Even though it is a little bit more complicated to increase the lower nibble it is much simpler to check if seconds or minutes exceed 59 or the hours exceed 23: just compare the packed BCD with the next higher limit.

```
  cpi R16,0x60 ; Compare seconds with 60
  brne done
  clr R16 ; Restart seconds
  ; ...
  cpi R17,0x60 ; Compare minutes with 60
  brne done
  clr R17 ; Restart minutes
  ; ...
  cpi R18,0x24 ; Compare hours with 24
  brne done
  clr R18 ; Restart hours
  ; ...
```

Instead of two registers to be compared now we have only one to be compared if the hours reached 24. And instead of two registers to be cleared there is only one. A clear advantage of packed BCD over ASCII and BCD.

The whole packed BCD second increase as assembler source code:

```
  ldi ZH,High(sTimePbcd) ; Z points to hours in packed BCD format
  ldi ZL,Low(sTimePbcd)
  ldd R16,Z+2 ; Read the seconds
  subi R16,-7 ; Add seven
  brhc ChkSec ; H clear, tens increased, check seconds for 60
  subi R16,6 ; H set, subtract six
ChkSec:
  std Z+2,R16 ; Write seconds
  cpi R16,0x60 ; 60 seconds?
  brcs Done ; No, completed
  ; One minute is over
  clr R16 ; Restart seconds
  std Z+2,R16 ; Write seconds to SRAM
  ldd R16,Z+1 ; Read minutes
  subi R16,-7 ; Add seven
```

```
  brhc ChkMin ; H clear, tens increased, check minutes for 60
  subi R16,6 ; H set, subtract six
ChkMin:
  std Z+1,R16 ; Write minutes to SRAM
  cpi R16,0x60 ; 60 minutes reached?
  brcs Done ; No, completed
  ; One hour is over
  clr R16 ; Restart minutes
  std Z+1,R16 ; And write to SRAM
  ld R16,Z ; Read hours
  subi R16,-7 ; Add seven
  st Z,R16 ; And write to SRAM
  brhc ChkHour ; H clear, tens increased, check hours for 24
  subi R16,6 ; H set, subtract six
  st Z,R16 ; and write to SRAM
ChkHour:
  cpi R16,0x24 ; 24 hours over?
  brcs Done ; Smaller than 24
  clr R16 ; Restart hours
  st Z,R16 ; Write to SRAM
  ; Increase date here
Done: ; Increase done
```

This is it. Those who do not believe that it works can simulate the source code by setting the SRAM location sTimePbcd to some desired values, such as 0x23:0x59:0x59 and run through the code.

To display the time formatted as packed BCDs on an LCD the upper nibble has to converted to an ASCII character, then the lower nibble. In case of the hours that goes like that:

```
  ld R16,Z ; Z points to hour in SRAM, read hours
  swap R16 ; Exchange nibbles: make upper to lower nibble
  and R16,0x0F ; Isolate lower nibble
  subi R16,-'0' ; Add ASCII zero
  rcall LcdChar ; R16 to LCD
  ld R16,Z ; Read hours again
  andi R16,0x0F ; Isolate lower nibble
  subi R16,-'0' ; Add ASCII zero
  rcall LcdChar ; R16 to LCD
```

The further display of the separator, the minutes, another separator and the seconds is all the same. If you use the instruction *LD R16,Z+* for the second copy from SRAM instruction, you can call the above hour display three times and you are done.

That is all and your time processing is complete in packed BCD.

2.4 Time in binary format

Byte	0	1	2
Time, Binary	1	23	45
Maximum:	23	59	59

Finally the simplest of all time formats: seconds, minutes and hours in binary format. As a maximum of 59 has to be handled all fits into one byte. Increasing by one second is done with *INC R16*, detection of the complete minutes and hours is with *CPI R16,60* resp. *CPI R16,24*. The following example of seconds increase assumes that the time is in three registers.

```
  .def rHour = R4 ; Hours register
  .def rMin = R5 ; Minutes register
```

```
  .def rSec = R6 ; Seconds register
IncSec:
  ldi R16,60 ; Detect end of seconds and minutes
  inc rSec
  cp rSec,R16 ; Seconds smaller than minute end?
  brcs Done ; No
  clr rSec ; Restart seconds
  inc rMin ; Increase minutes
  cp rMin,R16 ; Minutes smaller than hour end?
  brcs Done ; No
  clr rMin ; Restart minuts
  inc rHour ; Increase hours
  ldi R16,24 ; Hours per day
  cp rHour,R16 ; Hours smaller than day end?
  brcs Done ; No
  clr rHour ; Restart hours
Done:
  ; Time increase done
```

With those 14 simple instructions for a 24 hour clock it is a simple thing to program a clock. No reason at all to include a powerful datetime library and blow up those 14 simple instructions by at least 100-fold.

The display of the binary encoded time information now requires a binary-to-decimal-to-ASCII conversion. The following routine converts the binary in R16 to two ASCII digits and displays those on the LCD.

```
Bin2Dec2:
  clr R0 ; Count tens in R0
Bin2Dec2a:
  inc R0 ; Increase counter
  subi R16,10 ; Subtract 10
  brcc Bin2Dec2a ; No carry, repeat counting and subtract
  subi R16,-10-48 ; Undo last subtraction (add 10) and
                  ; convert to ASCII (add 48)
  push R16 ; Is needed later on, pushed to stack
  ldi R16,'0'-1 ; Counter minus one plus ASCII zero
  add R16,R0 ; Add counter value
  rcall LcdChar ; R16 as character to LCD
  pop R16 ; Restore second digit from stack
  rjmp LcdChar ; and write to LCD
```

The routine LcdChar, usually in the LCD include code, displays the character in R16 on the current position on the LCD and advances the cursor to the next position on the LCD. Due to the use of *RCALL* and *PUSH/POP* the stack has to be working.

For testing the routine you can point Z to the SRAM location SRAM_START and instead of the include routine LcdChar: you use *ST Z+,R16* and *RET* to write the converted ASCII characters to SRAM.

The complete display of the time then goes like this:

```
Display:
  mov R16,rStd ; Hours to R16
  rcall Bin2Dec2 ; Call conversion and display routine
  ldi R16,':' ; Separator
  rcall LcdChar ; to LCD
  mov R16,rMin ; Minutes to R16
  rcall Bin2Dec2 ; Call conversion/display
  ldi R16,':' ; Another separator
```

```
rcall LcdChar ; to LCD
mov R16,rSec ; Seconds to R16
rjmp Bin2Dec2 ; Jump to conversion/display
```

With these 20 instructions, of which 10 are for the display of binaries on the LCD, the whole stuff is not too sophisticated.

2.5 The best format

... is of course the binary, but the other three formats have also their pro's. So choose what you want, anything goes.

Top of page	Seconds	Formats	Date and time

3 Time and date

After having handled times we can handle dates as well. It works like times but has some extra rules making it a little bit more complicated:

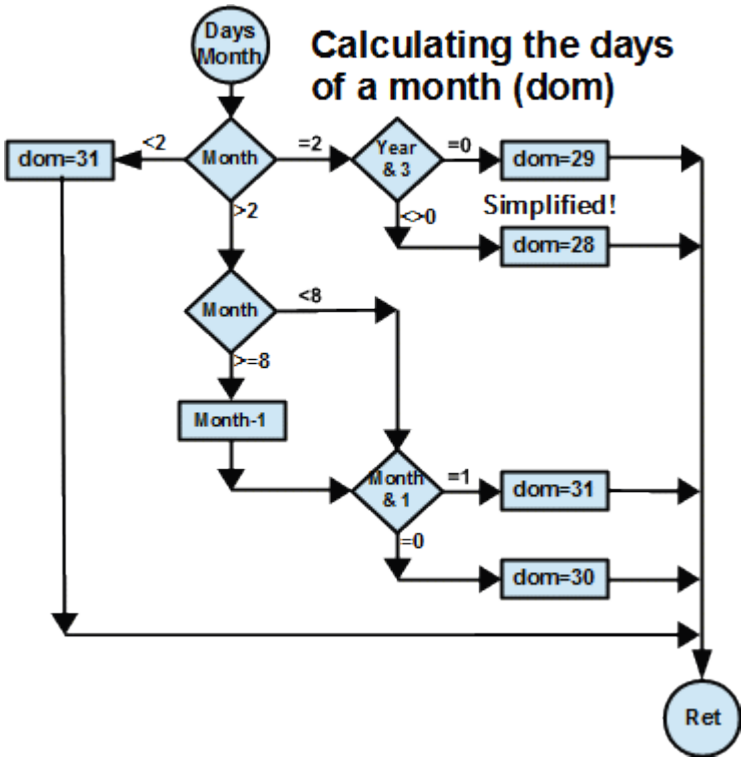
- Pope Gregor the XIIIth has, in the year 1582, issued the bulletin "*Inter gravissimas*", which decided that in contrary to seconds, minutes, hours and years days and years do not start with zero but with a one,
- the fact that the speed of the earth in rouding the sun is not a simple number but has some fractional parts (365.242188 days), that make it necessary to define some extra time to the 365 days. The Egyptians rounded this up a bit (to 365.25 days) and worked with that when they defined their tax paying algorithm. Pope Gregor wanted to be more exact, but did not e.g. define an extra "Earth rumble day" with a special duration of 5 hours, 48 minutes, 45 seconds, 43 milliseconds, 199 microseconds and a few nanoseconds. Instead of this simple solution pope Gregor went into a much more sophisticated algorithm that makes the life of calendar designers and assembler programmers more complicated,
- he decided that the months have not an equal number of days, e.g. ten months from 0 to 9, with 36 days each (day 0 to 35), but defined 12 months instead, even though 365 / 12 = 30.416666 (periodic) and started with 1 instead,
- the decision to choose 12 as the month's basis leaded to further complicated consequences:
    - the number of days per month was not a constant but varied,
    - this number of days did not follow simple rules, e.g. such as 30/31 alternating with the December length varying, but the alternating rule was reversed between July and August and February's length was largely varied,
    - the February's length was defined as being 28, but
        - not every four years if the year can be divided by four without a fractional part, then it is 29 days long,
        - but not every 100 years, when the year can be divided by 100 without a fractional part, then it is 28 days long,
        - but not every 400 years, when the year can be divided by 400 without a fractional part, then it is 29 days long,
- he further decided that one week has seven weekdays and not 10, an ancient "magic" number rather than a decimal's society preferred selection, but
- their numbering remained undefined, still causing confusion: some start with the Monday, others with the Sunday.

When writing dates on the LCD, and when storing those in SRAM, further illogical decisions complicate the life of the assembler programmers. While the time follows the simple rule that the further left is the more significant number (hours:minutes:seconds) dates are confused by other rules:

- The format convention Days:Months:Years reverses priority.
- The format convention Months/Days/Years simply confuses any priority rule.
- The format convention Years:Months:Days would be similar to the time format, but is rarely used.

But whatever you want to use: place them in the same row in SRAM as you want to display them, either forwards or backwards, so your pointer can increase or decrease after having outputted it. Or use displacement pointers.

This all makes programmers confused. To determine the number of days of a month we need this algorithm. Looks complicated but isn't that complicated in assembler:




```
;
; Subroutine calculates the
number of days of a month
;   Month in rMonth, Year in
rYear, result in rDom
;
DaysOfMonth:
  ldi rDom,31 ; 31 days
  cpi rMonth,2 ; February?
  brcs DaysOfMonthRet ; January
  brne DaysOfMonth1 ; Not
February
  ; February
  ldi rDom,28 ; No leap year
  mov R16,rYear ; Leap year?
  andi R16,0x03 ; Last two bits
of year?
  brne DaysOfMonthRet ; Last two
bits not 00
  ldi rDom,29 ; Leap year
  rjmp DaysOfMonthRet
DaysOfMonth1:
  mov R16,rMonth ; Copy month
  cpi R16,8
  brcs DaysOfMonth2 ; March to July
  dec R16 ; Monthes larger or equal August, reverse
DaysOfMonth2:
  ldi rDom,31 ; Months with 31 days
  andi rmp,0x01 ; Uneven?
  brne DaysOfMonthRet
  ldi rDom,30
DaysOfMonthRet:
  ret ; Done
```

Simulation of the DOM routine with the monthes 1 to 12 for a leap year (first line) and a non-leap year (second line) demonstrates that the days for the months (+01 to +0C), as written to the SRAM, are correct.

SRAM

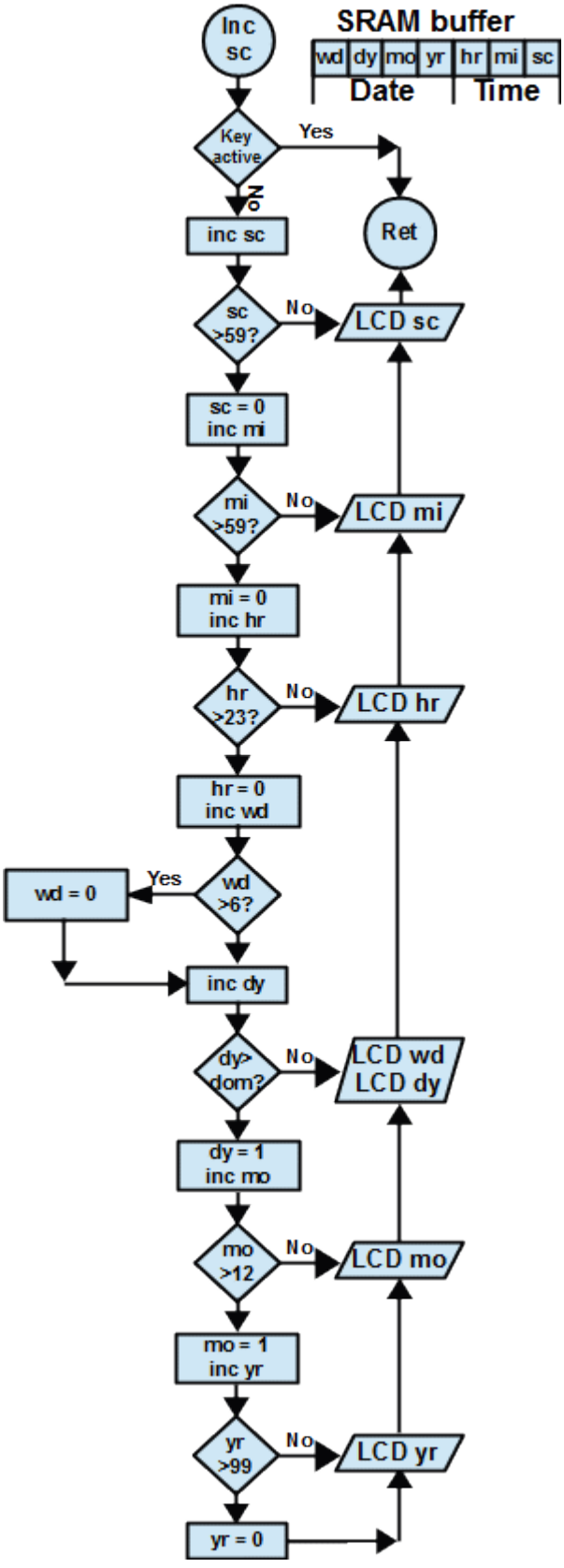
	+00	+01	+02	+03	+04	+05	+06	+07	+08	+09	+0A	+0B	+0C	+0D	+0E	+0F	ASCII text
$0060	4C	1F	1D	1F	1E	1F	1E	1F	1F	1E	1F	1E	1F	FF	FF	FF	L. . . . .
$0070	4E	1F	1C	1F	1E	1F	1E	1F	1F	1E	1F	1E	1F	FF	FF	FF	N. . . . .

Simulation was, of course, performed with avr_sim.

With these tools we can start to design a complete time and date flow diagram for a date/time clock. Rectangles are calculations, rhombuses are display operations and squares rotated by 45 degrees are decisions or conditional branches.

In the algorithm only those display elements are updated that are changed during an increase of that second. As a change of the year requires an update to all displayed elements on the LCD the update algorithm runs from bottom to the top while the time and date algorithm runs from top to bottom.

The first decision is that the date and time on the LCD is controlled by a different routine during a date and time input. Note that in that case the increase in time does not work but waits for a completion of the input procedure.



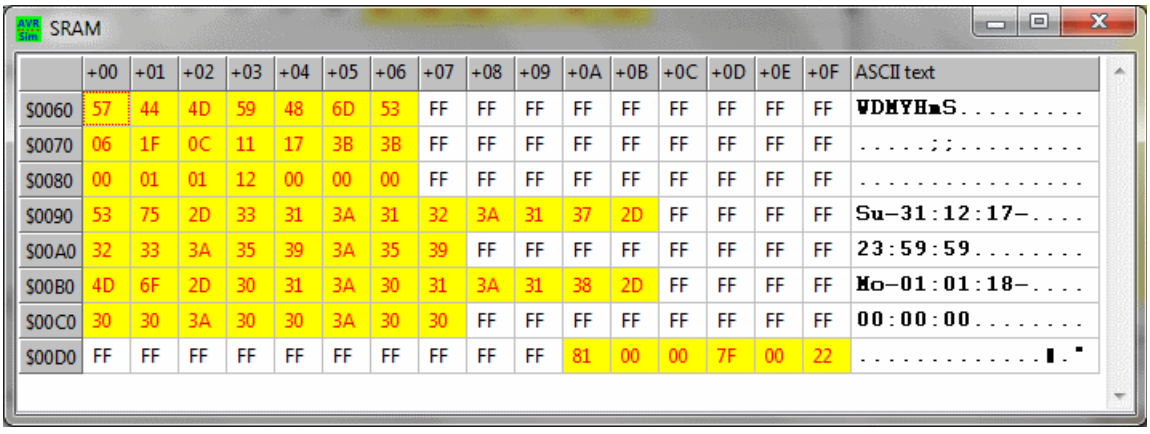
Here the complete increase time/date routine in assembler:

```
;
; Increase time in seconds
;
IncSec:
  ldi ZH,High(DateTimeBuffer)
  ldi ZL,Low(DateTimeBuffer)
  ldd R16,Z+6 ; Seconds
  inc R16
  std Z+6,R16
  cpi R16,60
  brcs DisplaySec
  clr R16
  std Z+6,R16
  ldd R16,Z+5 ; Minutes
  inc R16
  std Z+5,R16
  cpi R16,60
  brcs DisplayMin
  clr R16
  std Z+5,R16
  ldd R16,Z+4 ; Hours
  inc R16
  std Z+4,R16
  cpi R16,24
  brcs DisplayHours
  clr R16
  std Z+4,R16
  ld R16,Z ; Weekdays
  inc R16
  st Z,R16
  cpi R16,7
  brcs IncDay
  clr R16
  st Z,R16
IncDay:
  rcall DaysOfMonth ; Get days of that month in R16
  inc R16
  mov rData,R16
```

```
  ldd R16,Z+1
  inc R16
  std Z+1,R16
  cp R16,R0
  brcs DisplayWeekdays
  ldi R16,1
  std Z+1,R16
  ldd R16,Z+2 ; Monthes
  inc R16
  std Z+2,R16
  cpi R16,13
  brcs DisplayMonthes
  ldi R16,1
  std Z+2,R16
  ldd R16,Z+3 ; Years
  inc R16
  cpi R16,100
  std Z+3,R16
  brcs DisplayYears
  clr R16
  std Z+3,R16
DisplayYears:
  ; ...
DisplayMonthes:
  ; ...
DisplayWeekdays:
  ; ...
DisplayHours:
  ; ...
DisplayMinutes:
  ; ...
DisplaySeconds:
  ; ...
  ret
```

To simulate the work of IncSec: here a few tests. First, the year change on the 31nd of December, 2017, is shown. The line from address

- 0x0070 shows the original data for the Sunday at that date, for 23:23:59, in binary format,
- 0x0080 shows the date/time increased by one second, as binary,
- 0x0090 displays the original date/time in ASCII, and
- 0x00B0 on shows the increased date/time in ASCII.

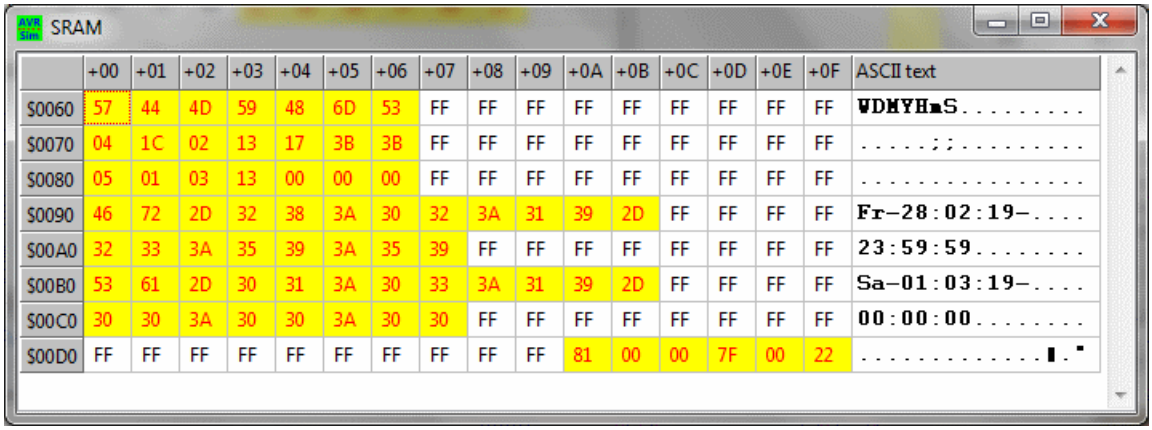


	+00	+01	+02	+03	+04	+05	+06	+07	+08	+09	+0A	+0B	+0C	+0D	+0E	+0F	ASCII text
$0060	57	44	4D	59	48	6D	53	FF	FF	FF	FF	FF	FF	FF	FF	FF	WDMYHmS
$0070	06	1F	0C	11	17	3B	3B	FF	FF	FF	FF	FF	FF	FF	FF	FF	
$0080	00	01	01	12	00	00	00	FF	FF	FF	FF	FF	FF	FF	FF	FF	
$0090	53	75	2D	33	31	3A	31	32	3A	31	37	2D	FF	FF	FF	FF	Su-31:12:17-
$00A0	32	33	3A	35	39	3A	35	39	FF	FF	FF	FF	FF	FF	FF	FF	23:59:59
$00B0	4D	6F	2D	30	31	3A	30	31	3A	31	38	2D	FF	FF	FF	FF	Mo-01:01:18-
$00C0	30	30	3A	30	30	3A	30	30	FF	FF	FF	FF	FF	FF	FF	FF	00:00:00
$00D0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	81	00	00	7F	00	22	

The change in the year works correct.

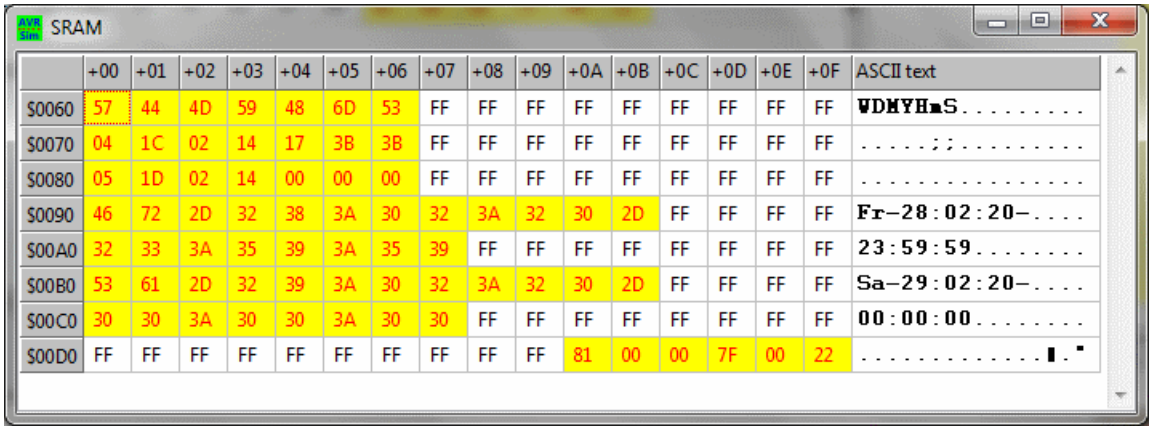
Simulation was again, of course, performed with avr_sim.

This simulates the seconds increase for February 28, 2019, which was not a leap year. Also correct.



	+00	+01	+02	+03	+04	+05	+06	+07	+08	+09	+0A	+0B	+0C	+0D	+0E	+0F	ASCII text
$0060	57	44	4D	59	48	6D	53	FF	FF	FF	FF	FF	FF	FF	FF	FF	WDMYHmS. . . . .
$0070	04	1C	02	13	17	3B	3B	FF	FF	FF	FF	FF	FF	FF	FF	FF	. . . . . ;; . . . . .
$0080	05	01	03	13	00	00	00	FF	FF	FF	FF	FF	FF	FF	FF	FF	. . . . .
$0090	46	72	2D	32	38	3A	30	32	3A	31	39	2D	FF	FF	FF	FF	Fr-28:02:19-. . . .
$00A0	32	33	3A	35	39	3A	35	39	FF	FF	FF	FF	FF	FF	FF	FF	23:59:59. . . . .
$00B0	53	61	2D	30	31	3A	30	33	3A	31	39	2D	FF	FF	FF	FF	Sa-01:03:19-. . . .
$00C0	30	30	3A	30	30	3A	30	30	FF	FF	FF	FF	FF	FF	FF	FF	00:00:00. . . . .
$00D0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	81	00	00	7F	00	22	. . . . . ▮."

And this is the seconds increase on February 28, 2020, which is a leap year. The routine obviously works correct.



	+00	+01	+02	+03	+04	+05	+06	+07	+08	+09	+0A	+0B	+0C	+0D	+0E	+0F	ASCII text
$0060	57	44	4D	59	48	6D	53	FF	FF	FF	FF	FF	FF	FF	FF	FF	WDMYHmS. . . . .
$0070	04	1C	02	14	17	3B	3B	FF	FF	FF	FF	FF	FF	FF	FF	FF	. . . . . ;; . . . . .
$0080	05	1D	02	14	00	00	00	FF	FF	FF	FF	FF	FF	FF	FF	FF	. . . . .
$0090	46	72	2D	32	38	3A	30	32	3A	32	30	2D	FF	FF	FF	FF	Fr-28:02:20-. . . .
$00A0	32	33	3A	35	39	3A	35	39	FF	FF	FF	FF	FF	FF	FF	FF	23:59:59. . . . .
$00B0	53	61	2D	32	39	3A	30	32	3A	32	30	2D	FF	FF	FF	FF	Sa-29:02:20-. . . .
$00C0	30	30	3A	30	30	3A	30	30	FF	FF	FF	FF	FF	FF	FF	FF	00:00:00. . . . .
$00D0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	81	00	00	7F	00	22	. . . . . ▮."

I wish you success in the self-making of date/time routines in assembler.

Top of page	Seconds	Formats	Date and time