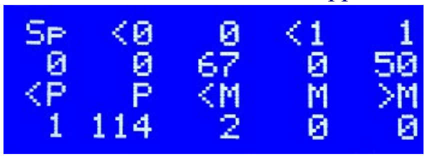


```
Sp  <0   0  <1   1
 0   0  67   0  50
<P   P  <M   M  >M
 1 114   2   0   0
```

AVR applications

Signal analysis with ATtiny24

Hardware, Operation, Use and Software for the signal analysis

Analyzing DCF77 signals with ATtiny24

1. Properties
2. Hardware
3. Operation
4. Software

1 Properties of DCF77 signal measuring device

1.1 Purpose

Analyzes DCF77 receiver modules for their signal duration.

1.2 Functioning

The hardware properties are:

- detects active DCF77 signals (high-to-low or low-to-high) on the DCF77 input pin,
- measures their duration (roughly in milliseconds),
- analyses these durations if they are valid Zeros, Ones, Pauses or Minute changes,
- detects false durations that do not fall under those categories of pulses,
- counts those valid pulses and errors,
- displays those counts on a 4-by-20 LCD attached,
- adds the measured duration to a 32-bit adder,
- on key action: copies the counters and adders to the EEPROM, from which the complete content can be downloaded and converted to a comma-separated-value (CSV) file, to be imported to an OpenOffice spreadsheet for further analysis, and
- counters can be cleared by an attached key.

1.3 Operates with different voltages

1. If your LCD requires 5V and your DCF77 receiver module allows operation with +5V, you can use it un-modified.
2. If your LCD requires 3.3V and your receiver module works at 3.3V, you can also use it unmodified.
3. If your LCD requires 5V and your module only allows operation with up to 3.3V, use the described modified hardware.

1.4 Simple hardware

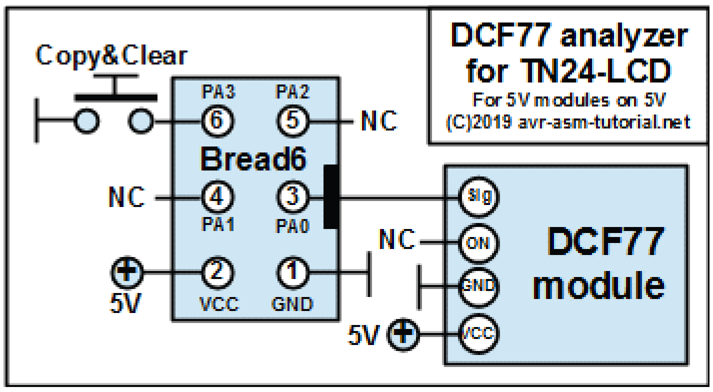
The hardware is based on a ATtiny24-LCD module, as described here. The receiver module and a key (to reset the counter) can be plugged into the Bread6 interface of the module with a 6-pin parallel cable and plug. Only the software described here is needed.

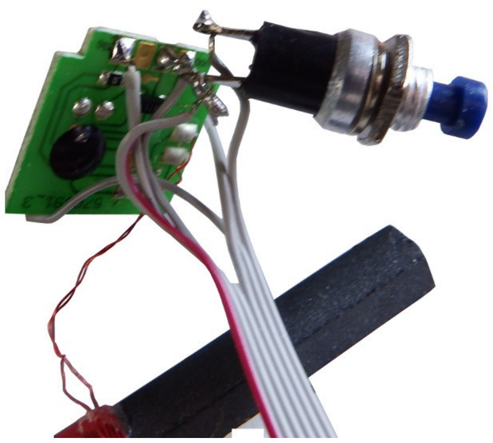
2 Hardware

2.1 5V version

This is the whole schematic for the 5V version. If both, the LCD and the receiver module, work at 3.3V, you can also use this.

Testing of the hardware is simple: just check your connections.

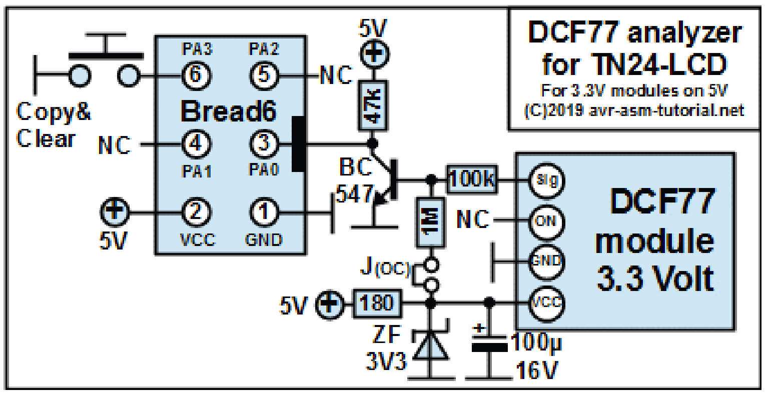




The picture to the left shows such a 5V receiver module, as attached to the ATtiny-LCD-module.

2.2 The 5V/3.3V version

The alternative, if your LCD has 5V and your receiver module allows only 3.3V, is this.



The operating voltage is reduced by a resistor of 180Ω and a Zener diode of 3.3V. Zener noise is suppressed by the 100µF electrolytic capacitor as DCF77 receiver modules are very sensitive for noise on the operating voltage.

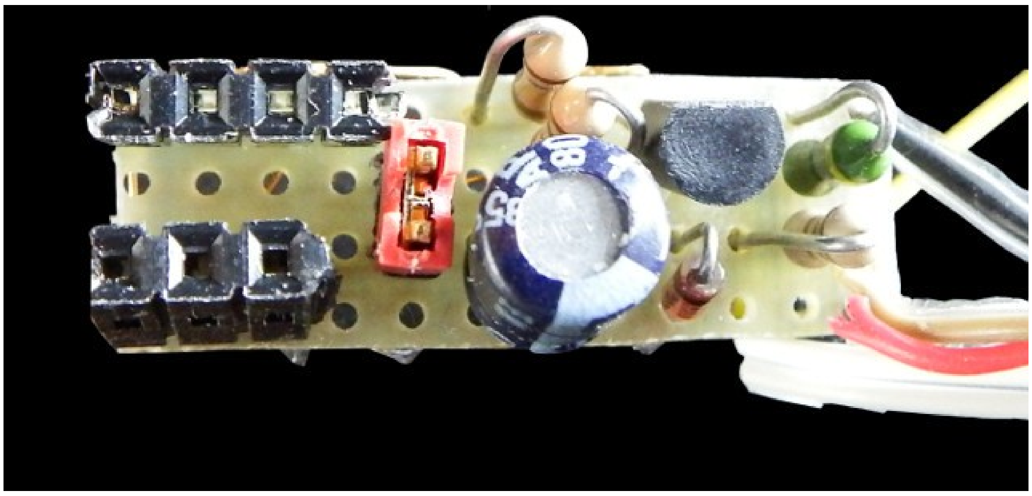
The output pin of the module delivers either a rectangle signal or has an open collector. In the first case the 3.3V voltage swing has to be translated to 5V by use of an NPN transistor (BC547 or equivalent). As receiver modules do not generate lots of current, a high resistor of 100k produces enough base current to drive the 47k collector resistor and produces a stable input voltage for the ATtiny24.

In the second case, if your module has an open collector output only, you'll have to provide enough current on the 3.3V level to drive the base of the transistor high if the open collector is inactive. That is done with the 1M resistor. In that case you'll have to close the jumper. All receiver modules that I have tested operate correct with the 1M, make sure you have that closed in case your module is of the open collector type.

Testing of the hardware is also simple, but not as simple as in the previous version. Of course, your connections shall be ok, too. If the LCD does not count any signals, try closing the jumper. That should resolve most potential errors.

If not, remove the receiver module and the jumper and connect the signal input with GND. The output, the collector of the transistor, shall now show a voltage near to 5V. Then tie the signal input to the +3.3V (on the cathode of the Zener diode, e.g.). Now the output shall be close to zero Volt.

This is such an interface. On the three- and four-pin female plug-ins different receiver modules can be tested. The connection to the ATtiny24-lcd-module are wired, all components are mounted on a small PCB.



3 Operation

3.1 Selecting the tolerance value

The main parameter to select is how accurate the receiver's signal shall be. This is defined by the constant **cDcfTol**. This is the tolerance in percent that the signals can have to be correct.

Avoid too narrow tolerances below 6 percent and too large tolerances beyond 33 percent. This unavoidably results in errors because, e.g., the recognition of a pause has an average duration of 850 ms, but can be as well be 800 (or slightly shorter) or 900 (or slightly longer). If you receive many P errors, that might be a hint for a too small tolerance.

Also, avoid too large tolerances. If you are above 33%, the accepted areas would overlap. When assembling, those overlaps are recognized, you'll get an error message and no hex code is generated.

20% is a good margin, and a good receiver shall operate with that with no errors.

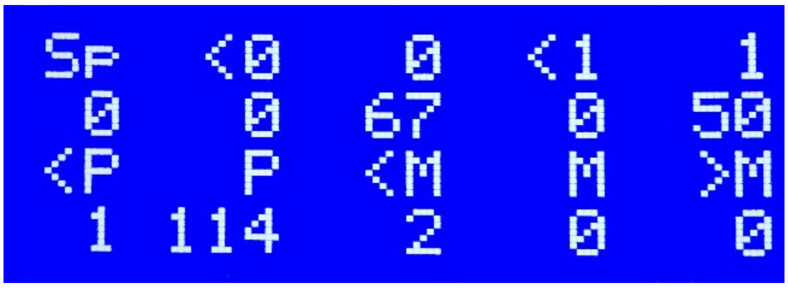
3.2 Display of counts

The software works as follows:

1. Each signal change on the DCF input restarts the TC1 counter, which is clocked by 1 MHz and prescaled by 1,024 (frequency = 976.6 Hz, time per tick = 1.024 ms).
2. The length of each pulse duration is compared with constants that are derived from the settings of the times in the **Adjustable constants** section of the software. There, a logical zero transmitted by DCF77, is set to last 100 ms, a logical one to 200 ms. The pause that follows a logical zero or a logical one, until the beginning of the next second, is set to 850 ms, but can be either 800 or 900 ms depending from

the bit that was sent. Another signal duration is the length of a missing second pulse (the 60th second is omitted by DCF77). That results in a pause between 1,800 and 1,900 ms.

3. The analysis of the pulse duration can have the following results:
    1. Shorter than the **Spurious signal** duration: very short signal (shorter than 10 ms), this count is the first number in line 2 of the LCD below **Sp**,
    2. Longer than the spurious signal duration, but Shorter than the time necessary at least (100 ms minus tolerance) for a zero, this count is below the **<0** in line 2 of the LCD,
    3. A correct zero signal, below **0** in line 2,
    4. The same for ones: either too short for a one **<1** or a correct **1**.
    5. Further results are listed in lines 3 and 4 of the LCD: the pause duration (**<P** and **P**), the minute change duration (**<M** and **M**) and if the signal duration exceeds the maximum (**>M**).
4. If counts reach 10,000 or higher, **9999** is displayed, but the counter is 16 bit wide and can count 65,535 pulses.
5. Use the key to copy all counters to EEPROM, clear all counters and to restart counting.



```
Sp  <0   0  <1   1
 0   0  67   0  50
<P   P  <M   M  >M
 1 114   2   0   0
```

In the example no spurious signals and shorter than zero or ones had happened, but one pause was too short and two minute change signals were too short.

As a pause is always associated with each received one and zero (and follows those), the number of pauses should be equivalent to the sum of both. This is approximately the case in the example.

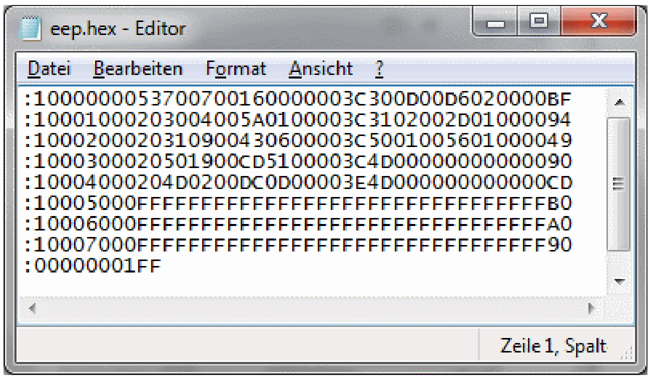
3.3 Downloading and calculating signal averages

The software not only counts the number of signal duration categories and displays this live on the LCD but also adds the real measured signal duration to a 32-bit adder for each category in the SRAM buffer. As this information does not fit onto the small LCD, it is only held in SRAM.

Every time that the key is pressed, the content of this SRAM buffer, consisting of

1. two ASCII characters for the category (the same as displayed on the LCD headers in line 1 and 3),
2. two bytes for the counter (up to 65,535 events),
3. four bytes for the adder (up to a sum of 4.29 millions),

are written to the EEPROM and can be downloaded from there. The generated EEPROM content e.g. looks like this.



```
:1000000053700700160000003C300D00D6020000BF
:10001000203004005A0100003C3102002D01000094
:1000200020310900430600003C5001005601000049
:1000300020501900CD5100003C4D00000000000090
:10004000204D0200DC0D00003E4D000000000000CD
:10005000FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFB0
:10006000FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFA0
:10007000FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF90
:00000001FF
```

That is rather cryptic, and we need a tool to convert this into a more meaningful format. For this I wrote the Pascal program "dcfsigana_eep2csv". Its source code is here and compiles with any Pascal compiler (tested with Free Pascal Compiler, FPC). If you are afraid of installing or not allowed to install FPC (admins sometimes are hypercritical) or if FPC is not available for your operating system (which rarely can be true), you can download the already compiled Windows-64-bit executable from here.

This software eats an Intel hex format file, by default named "eep.hex" (select h from the menu, or then input any other name), tests it by counting the lines and checking whether it only consists of colons as first characters on the lines and of hexadecimal numbers (select a from the menu) and if successfully tested, converts it to a CSV encoded file. Each line of the file is the content of one out of ten of the above SRAM buffer structure. The ASCII characters are enclosed into ‟, the numbers are separated by tab characters (ASCII-9). The CSV file can be imported with OpenOffice (do not try Excel, it cannot handle CSV files conveniently). Make sure that you tick the tabulator as a separator (if not already marked).

Select the decoded raw data, copy it to the clipboard and paste it to the "rawdata" sheet of the OpenOffice spreadsheet here. The results are used on the sheet "measuring" to calculate average pulse durations in milliseconds and to compare those with the ideal duration.

With this tool you'll see how long your receiver module's signals really are, why zeroes, ones, pauses and minutes fit or fit not into the decoder's math. You'll also see how long spurious signals on the input line really last.

With that you should be able to do identify any kind of problem that your module shows.

4 Software

4.1 Downloads

The assembler source code is available for download here and needs the LCD include to assemble correct.

The source can be viewed in the attachment here.

Please note that the previously published lcd.inc, which was first used by me in creating this software, had a serious bug: it cleared the lower part of the output port A, by that initiating key PCINTs constantly. After identifying and correcting the bug, the key interrupts work fine. So make sure you use the latest lcd.inc, as linked above, and not an older version.

4.2 How the software works

4.2.1 Time measurements

Time measurement in performed with TC1. This timer is clocked by the 1 MHz system clock with a prescaler of 1,024. Timer ticks are therefore roughly 1 ms long (more exact: 1.024 ms).

Each interrupt on the DCF77 pin reads the current timer count and clears it thereafter.

TC0 is not used and free for other uses.

4.2.2 DCF duration classification

The times for each DCF77 signal category is given in the software as a constant (names starting with t) in milliseconds. Those are converted into timer values using the controller's clock, its prescaler value and the tolerance for which those times shall be included into the categories.

The times and counter values can be best viewed in the OpenOffice spreadsheet here, but can also be read as symbols (with a c instead of the t) from gavrasm's symbol list, if assembling is done with the -s option active.

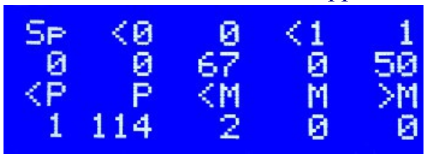
Those constants are all included in a table that is used by the software to identify the signal category.

4.2.3 Reading results from EEPROM

Following any key event,

1. all category strings, counters and adders are copied from SRAM to EEPROM,
2. the routine waits 100 ms until the key has been released (with debouncing),
3. clears all counters and adders and restarts.

The EEPROM can be read and its content can be viewed by assistance of the above described software and can be imported to a spreadsheet, for which the OpenOffice spreadsheet here can be used.



```
Sp  <0   0  <1   1
 0   0  67   0  50
<P   P  <M   M  >M
 1 114   2   0   0
```

AVR applications

Analysis for DCF77 receiver signals **Assembler software for analyzing**

Assembler source code for the DCF77 signal analysis

The original assembler source code is here, it needs the LED include to assemble.

```
;
; *********************************
; * Test an attached DCF77 module *
; * with a tn24_lcd module        *
; * (C)2019 avr-asm-tutorial.net  *
; *********************************
;
.nolist
.include "tn24adef.inc" ; Define device ATtiny24A
.list
;
; **********************************
;    D E B U G   S W I T C H E S
; **********************************
;
.equ Yes = 1 ; Enable switch
.equ No = 0 ; Disable switch
;
; Skip all LCD operations for simulation
.equ Debug_SkipLcd = No
;
; Simulate the DCF duration categories
.equ Debug_DcfCat = No
  .equ Debug_DcfCatD = 200 ; Set the DCF duration
;
; **********************************
;        H A R D W A R E
; **********************************
;
; Device: ATtiny24A, Package: 14-pin-PDIP_SOIC
;
;              _________
;          1 /         |14
;    +5V o--|VCC    GND|--o 0V
; LCD-RS o--|PB0    PA0|--o DCF
; LCD-RW o--|PB1    PA1|--o NC
;  RESET o--|RESET  PA2|--o NC
;  LCD-E o--|PB2    PA3|--o Key
; LCD-D7 o--|PA7    PA4|--o LCD-D4
; LCD-D6 o--|PA6    PA5|--o LCD-D5
;         7 |__________|8
;
; **********************************
; P O R T S   A N D   P I N S
; **********************************
;
.equ pDcfKeyO = PORTA ; Define the output port
.equ pDcfKeyD = DDRA ; Define the direction port
.equ pDcfKeyI = PINA ; Define the input port
.equ bDcfO = PORTA0 ; Define the DCF output port
.equ bDcfI = PINA0 ; Define the DCF input port
.equ bKeyO = PORTA3 ; Define the key pins
.equ bKeyI = PINA3 ; Define the key input port
; (LCD pins are defined in Adjustable Constants section)
;
; **********************************
;  A D J U S T A B L E   C O N S T
; **********************************
;
; Delay between initial LCD message and measuring start
.equ tDelay = 5 ; Number of seconds delay
;
; Set pull up resistor on DCF input
.equ DcfPullUp = No ; Yes or no
;
; Time settings for the DCF77 input signals
.equ cDcfTol = 20 ; Tolerance of all times in %
.equ tDcfSp = 10 ; Spurious signals, in ms
.equ tDcf0 = 100 ; Signal duration for a 0, in ms
.equ tDcf1 = 200 ; Signal duration for a 1, in ms
.equ tDcfP = 850 ; Signal duration for a pause, in ms
.equ tDcfM = 1850 ; Signal duration for a missing second pulse, in ms
.equ tDcfTO = 2500 ; DCF signal missing time-out, in ms
;
; Parameter set of properties/definitions for lcd
;   (Needed by LCD include)
.equ clock = 1000000 ; Clock frequency of controller in Hz
; LCD size:
  .equ LcdLines = 4 ; Number of lines (1, 2, 4)
  .equ LcdCols = 20 ; Number of characters per line (8..24)
; LCD bus interface
  .equ LcdBits = 4 ; Bus size (4 or 8)
  ; If 4 bit bus:
    .equ Lcd4High = 1 ; Bus nibble (1=Upper, 0=Lower)
; LCD wait/busy mode
  .equ LcdWait = 0 ; Access mode (0 with busy, 1 with delay loops)
; LCD data ports
  .equ pLcdDO = PORTA ; Data output port
  .equ pLcdDD = DDRA ; Data direction port
; LCD control ports und pins
  .equ pLcdCEO = PORTB ; Control E output port
  .equ bLcdCEO = PORTB2 ; Control E output portpin
  .equ pLcdCED = DDRB ; Control E direction port
  .equ bLcdCED = DDB2 ; Control E direction portpin
  .equ pLcdCRSO = PORTB ; Control RS output port
  .equ bLcdCRSO = PORTB0 ; Control RS output portpin
  .equ pLcdCRSD = DDRB ; Control RS direction port
  .equ bLcdCRSD = DDB0 ; Control RS direction portpin
; If LcdWait = 0:
  .equ pLcdDI = PINA ; Data input port
  .equ pLcdCRWO = PORTB ; Control RW output port
  .equ bLcdCRWO = PORTB1 ; Control RW output portpin
  .equ pLcdCRWD = DDRB ; Control RW direction port
  .equ bLcdCRWD = DDB1 ; Control RW direction portpin
```

```
; If you need binary to decimal conversion:
  ;.equ LcdDecimal = 1 ; If defined: include those routines
; If you need binary to hexadecimal conversion:
  ;.equ LcdHex = 1 ; If defined: include those routines
; If simulation in the SRAM is desired:
  ;.equ avr_sim = 1 ; 1=Simulate, 0 or undefined=Do not simulate
;
; **********************************
;  F I X  &  D E R I V.  C O N S T
; **********************************
;
; Converting DCF times to TC1 clock cycles with tolerances
.equ ckHz = (clock+500)/1000 ; Clock frequency in kHz
.equ cDcfSp = (tDcfSp*ckHz+512)/1024 ; (0..3)
.equ cDcf0Min = ((tDcf0-cDcfTol*tDcf0/100)*ckHz+512)/1024 ; (4..7)
.equ cDcf0Max1 = ((tDcf0+cDcfTol*tDcf0/100)*ckHz+512)/1024+1 ; (8..11)
.equ cDcf1Min = ((tDcf1-cDcfTol*tDcf1/100)*ckHz+512)/1024 ; (12..15)
.equ cDcf1Max1 = ((tDcf1+cDcfTol*tDcf1/100)*ckHz+512)/1024+1 ; (16..19)
.equ cDcfPMin = ((tDcfP-cDcfTol*tDcfP/100)*ckHz+512)/1024 ; (20..23)
.equ cDcfPMax1 = ((tDcfP+cDcfTol*tDcfP/100)*ckHz+512)/1024+1 ; (24..27)
.equ cDcfMMin = ((tDcfM-cDcfTol*tDcfM/100)*ckHz+512)/1024 ; (28..31)
.equ cDcfMMax1 = ((tDcfM+cDcfTol*tDcfM/100)*ckHz+512)/1024+1 ; (32..35)
.equ cDcfTO = (tDcfTO*ckHz+512)/1024+1 ; (36..39)
;
; Checking of constants
.if cDcfTol == 0
  .error "Tolerance cDcfTol cannot be zero!"
  .endif
.if (cDcf0Min <= cDcfSp)
  .error "Minimum for zero less than spurious, reduce tolerance"
  .endif
.if (cDcf0Max1 >= cDcf1Min)
  .error "Zero maximum overlaps One minimum, reduce tolerance"
  .endif
.if (cDcf1Max1 >= cDcfPMin)
  .error "One maximum overlaps Pause minimum, reduce tolerance"
  .endif
.if (cDcfPMax1 >= cDcfMMin)
  .error "Pause maximum overlaps Minute minimum, reduce tolerance"
  .endif
;
; **********************************
;       R E G I S T E R S
; **********************************
;
.def rBinL = R0 ; Binary LSB
.def rBinH = R1 ; dto., MSB
; free: R2 to R14
.def rSreg = R15 ; Save/Restore status port
.def rmp = R16 ; Define multipurpose register
.def rimp = R17 ; Multipurpose inside ints
.def rFlag = R18 ; Flag register
  .equ bDcf = 0 ; DCF flag
  .equ bKey = 1 ; Key input flag
.def rInput = R19 ; Input register
.def rDcfDurL = R20 ; DCF duration, LSB
.def rDcfDurH = R21 ; dto., MSB
; free: R22 to R25
; used: R27:R26 = X as counter pointer
; used: R29:R28 = Y for decimal conversion
; used: R31:R30 = Z for various purposes
```

```
;
; **********************************
;           S R A M
; **********************************
;
.dseg
.org SRAM_START
sCounters:
.byte 8 ; Spurious count
  ; 2 bytes description
  ; 2 bytes counter
  ; 4 bytes adder
.byte 8 ; Shorter than zero
.byte 8 ; Correct zero
.byte 8 ; Shorter than one
.byte 8 ; Correct one
.byte 8 ; Shorter than pause
.byte 8 ; Correct pause
.byte 8 ; Shorter than minute
.byte 8 ; Correct minute
.byte 8 ; Longer than minute
sCountersEnd:
;
; **********************************
;         C O D E
; **********************************
;
.cseg
.org 000000
;
; **********************************
; R E S E T  &  I N T - V E C T O R S
; **********************************
        rjmp Main ; Reset vector
        reti ; EXT_INT0
        rjmp Pcint0Isr ; PCI0
        reti ; PCI1
        reti ; WATCHDOG
        reti ; ICP1
        reti ; OC1A
        reti ; OC1B
        reti ; OVF1
        reti ; OC0A
        reti ; OC0B
        reti ; OVF0
        reti ; ACI
        reti ; ADCC
        reti ; ERDY
        reti ; USI_STR
        reti ; USI_OVF
;
; **********************************
;  I N T - S E R V I C E   R O U T .
; **********************************
;
; PCINT0 interrupt service routine
PcInt0Isr:
  in rSreg,SREG ; Save SREG
  in rimp,pDcfKeyI ; Read current pins
  eor rimp,rInput ; Check pins that changed
  sbrc rimp,bKeyI ; Check key pin changed
  rjmp PcInt0Isr1 ; No, skip
  sbis pDcfKeyI,bKeyI ; Check key pin is high
  sbr rFlag,1<<bKey ; No, set key low flag
PcInt0Isr1:
  sbrs rimp,bDcfI ; Check if DCF input changed
  rjmp PcInt0Isr2 ; No, skip
  sbr rFlag,1<<bDcf ; Set DCF flag
  in rDcfDurL,TCNT1L ; Read counter, LSB first
  in rDcfDurH,TCNT1H ; dto., MSB next
  clr rimp ; Restart counter
  out TCNT1H,rimp ; MSB first
  out TCNT1L,rimp ; LSB next
PcInt0Isr2:
  in rInput,pDcfKeyI ; Read input port
  out SREG,rSreg ; Restore SREG
  reti
;
; **********************************
;  M A I N   P R O G R A M   I N I T
; **********************************
;
Main:
        ldi rmp,Low(RAMEND)
        out SPL,rmp ; Init LSB stack pointer
  ; Init ports
  sbi pDcfKeyO,bKeyO ; Set pull up on key pin
  .if DcfPullup == Yes
    sbi pDcfKeyO,bDcfO ; Set pull up on DCF pin
        .endif
.if Debug_SkipLcd == No
  ; Init the LCD
  rcall LcdInit ; Init the LCD
  ldi ZH,High(2*Text_Intro) ; Intro text
  ldi ZL,Low(2*Text_Intro)
  rcall LcdText ; Display intro text
  ; Delay the execution a little bit
  ldi rmp,3*tDelay ; tDelay seconds
Main1:
  sbiw ZL,1 ; 2 cycles
  brne Main1 ; 2 cycles for branching, 1 at the end
  dec rmp ; 1 cycle
  brne Main1 ; 2 cycles for branching, 1 at the end
  .endif
  ; Complete time for the loop:
  ;   Inner loop: cycles = 65535 * 4 + 3
  ;   Plus outer loop: cycles = (Inner loop + 3) * (rmp -1) + Innerloop + 2
  ;     = (rmp-1)*(Inner+3) + Inner + 2
  ;     = rmp*Inner + 3*rmp - Inner - 3 + Inner + 2
  ;     = rmp*65535*4 + 3 + 3*rmp - 65535*4 - 12 - 3 + 65535*4 + 3 + 2
  ;     = rmp*262140 + 3*rmp + 3 - 12 - 3 + 3 + 2
  ;     = rmp*(262140+3) - 7
  ;   For one second:
  ;   rmp = (1000000+7)/262143 = 3
  ;
  ; Init TC1 for counting signal durations
  ldi rmp,High(cDcfTO) ; Set time-out for DCF signals, MSB
  out OCR1AH,rmp ; in MSB compare A first
  ldi rmp,Low(cDcfTO) ; dto., LSB
  out OCR1AL,rmp ; in LSB compare next
  clr rmp ; Mode TC1 setting
  out TCCR1A,rmp ; in mode control A
  ldi rmp,(1<<WGM12)|(1<<CS12)|(1<<CS10) ; CTC, prescaler=1024
  out TCCR1B,rmp ; in mode control B
  ; Restart all counters, set standard text, set LCD to standard frame
  rcall ClearOnly ; Restart counting
.if Debug_DcfCat == Yes
  ldi rDcfDurH,High(Debug_DcfCatD)
  ldi rDcfDurL,Low(Debug_DcfCatD)
  rcall DcfSig
  endless: rjmp endless
  .endif
  ; Init ports
;  endless1: rjmp endless1
; Only use this if your module is compatible with
;   the pull-up switched on (mot necessary if you
;   use an NPN driver stage)
;  sbi pDcfKeyO,bDcfO
  ; Init PCINTs on the DCF and Key inputs
  ldi rmp,(1<<bDcfO)|(1<<bKeyO) ; Enable DCF and key ints
  out PCMSK0,rmp
  ldi rmp,1<<PCIE0 ; Enable PCINT0 interrupts
  out GIMSK,rmp ; in General Interrupt Mask
  ; Sleep and interrupts
  ldi rmp,1<<SE ; Sleep mode idle
  out MCUCR,rmp ; in Microcontroller Universal Control Port
  sei ; Enable interrupts
;
; **********************************
;    P R O G R A M   L O O P
; **********************************
;
Loop:
  sleep ; Go to sleep
  nop ; Wake up dummy
  sbrc rFlag,bDcf ; DCF flag clear?
  rcall DcfSig ; Count DCF signal
  sbrc rFlag,bKey ; Key flag clear?
  rcall ClearAll ; No, clear all counters
  rjmp loop
;
; A DCF signal came in
DcfSig:
  cbr rFlag,1<<bDcf ; Clear the flag
  ldi XH,High(sCounters+2) ; Point X to sCounters
  ldi XL,Low(sCounters+2)
  ldi ZH,High(2*DcfDurTab) ; Point Z to duration table
  ldi ZL,Low(2*DcfDurTab)
  lpm rmp,Z+ ; Read table value, LSB
  cp rDcfDurL,rmp ; Compare LSB duration
  lpm rmp,Z+ ; Read table value, MSB
  cpc rDcfDurH,rmp ; dto., MSB with carry
  brcs DcfSigCnt ; Increase count
DcfSig1:
  adiw XL,8 ; Next counter
  cpi XL,Low(sCountersEnd) ; End of valid counters?
  breq DcfSigEnd ; Yes, up-count last
  lpm rmp,Z+ ; Read LSB from table
  cp rDcfDurL,rmp ; Compare LSB
  lpm rmp,Z+ ; Read MSB from table
  cpc rDcfDurH,rmp ; Compare MSB and Carry
  brcs DcfSigCnt ; Shorter than minimum
  adiw XL,8 ; Next counter
  cpi XL,Low(sCountersEnd) ; End of valid counters?
  breq DcfSigEnd ; Yes, up-count last counter
  lpm rmp,Z+ ; Read LSB from table
  cp rDcfDurL,rmp ; Compare LSB
  lpm rmp,Z+ ; Read MSB from table
  cpc rDcfDurH,rmp ; Compare MSB and Carry
  brcs DcfSigCnt ; Shorter than maximum plus one, count
  rjmp DcfSig1 ; Next compare
DcfSigEnd:
;
; Count the signal counts one up
DcfSigCnt:
  ld rmp,X ; Read LSB from SRAM
  inc rmp ; Count LSB one up
  st X+,rmp ; Save LSB
  brne DcfSigCnt1 ; No overflow
  ld rmp,X ; Read MSB from SRAM
  inc rmp ; Incrase MSB
  st X,rmp ; Store MSB
DcfSigCnt1:
  adiw XL,1 ; Point to adder byte 1
  ld rmp,X
  add rmp,rDcfDurL
  st X+,rmp
  ld rmp,X
  adc rmp,rDcfDurH
  st X+,rmp
  ldi rDcfDurL,0
  ld rmp,X
  adc rmp,rDcfDurL
  st X+,rmp
  ld rmp,X
  adc rmp,rDcfDurL
  st X+,rmp
  sbiw XL,6 ; Point back to counter
  ; Position on LCD
  mov ZL,XL ; Copy SRAM position to Z
  mov ZH,XH
  andi ZL,0xFC ; Clear bit 1 and 0
  ldi rmp,Low(sCounters)
  sub ZL,rmp
  ldi rmp,High(sCounters)
  sbc ZH,rmp
  ldi ZH,1 ; Line 2 of the LCD
  lsr ZL ; Divide counter by two
  cpi ZL,20 ; End of line 2?
  brcs DcfSigCnt2
  subi ZH,-2 ; Add two to line number
  subi ZL,20 ; Subtract one line length
DcfSigCnt2:
  rcall LcdPos ; Set LCD position
  ld rBinL,X+ ; Read counter value to binary
  ld rBinH,X
  ldi XH,High(10000) ; Check out of decimal range
  ldi XL,Low(10000)
  cp rBinL,XL
  cpc rBinH,XH
  brcc DcfSigOvf
  set ; Set the T flag
  ldi XH,High(1000) ; Count the thousands
  ldi XL,Low(1000)
  rcall DecDigit ; Calculate and write the decimal digit
  ldi XH,High(100)
  ldi XL,Low(100)
  rcall DecDigit
  ldi XH,High(10)
  ldi XL,Low(10)
  rcall DecDigit
  ldi rmp,'0'
  add rmp,rBinL ; The last decimal digit
  rjmp LcdChar
DcfSigOvf:
  ldi rmp,'9' ; Write 9999 to LCD
.if Debug_SkipLcd == No
  rcall LcdChar
  ldi rmp,'9'
  rcall LcdChar
  ldi rmp,'9'
  rcall LcdChar
  ldi rmp,'9'
  rjmp LcdChar
  .else
  ret
  .endif
;
; Count the decimal digit and write it to LCD
;   Binary in rBinH:rBinL
;   Decimal in X
;   T flag enables leading zero suppression
DecDigit:
  ldi rmp,0xFF
DecDigit1:
  inc rmp
  sub rBinL,XL ; Subtract decimal from binary
  sbc rBinH,XH
  brcc DecDigit1
  add rBinL,XL ; Undo last subtraction
  adc rBinH,XH
  brtc DecDigit3 ; Do not suppress leading zeroes
  tst rmp ; Zero?
  brne DecDigit2 ; Not zero, stop suppression
  ldi rmp,' ' ; Display a blank
.if Debug_SkipLcd == No
  rjmp LcdChar
  .else
  ret
  .endif
DecDigit2:
  clt ; Stop suppression
DecDigit3:
  subi rmp,-'0' ; Add ASCII zero
.if Debug_SkipLcd == No
  rjmp LcdChar
  .else
  ret
  .endif
;
; Table with the DCF77 counter values
DcfDurTab:
  .dw cDcfSp ; Spurious signal, 0..3
  .dw cDcf0Min, cDcf0Max1 ; 4..11
  .dw cDcf1Min, cDcf1Max1 ; 12..19
  .dw cDcfPMin, cDcfPMax1 ; 20..27
  .dw cDcfMMin, cDcfMMax1 ; 28..35
DcfDurTabEnd:
;
; Add strings to SRAM
Str2Sram:
  ldi XH,High(sCounters)
  ldi XL,Low(sCounters)
  ldi ZH,High(2*StrTab)
  ldi ZL,Low(2*StrTab)
Str2Sram1:
  lpm rmp,Z+
  st X+,rmp
  lpm rmp,Z+
  st X+,rmp
  adiw XL,6
  cpi XH,High(sCountersEnd)
  brne Str2Sram1
  cpi XL,Low(sCountersEnd)
  brne Str2Sram1
  ret
;
; Table with the text strings in SRAM
StrTab:
  .db "Sp<0 0<1 1<P P<M M>M"
;
; The clear all key has been pressed
ClearAll:
  cbr rFlag,1<<bKey ; Clear the key flag
  ; Write SRAM content to EEPROM
  clr ZL ; EEPROM address, LSB
  clr ZH ; dto., MSB
  ldi XH,High(sCounters) ; X points to Sram counters, MSB
  ldi XL,Low(sCounters) ; dto., LSB
Copy2Eep:
  ; Wait for EEPROM programming clear
  sbic EECR,EEPE ; Wait for EEPE bit clear
  rjmp Copy2Eep ; Not yest
  ldi rmp,(0<<EEPM1)|(0<<EEPM0) ; Atomic Write mode
  out EECR,rmp  ; to EEPROM control
  out EEARH,ZH ; Set write address, MSB
  out EEARL,ZL ; dto., LSB
  adiw ZL,1 ; Next address
  ld rmp,X+ ; Read Sram byte
  out EEDR,rmp ; to data register
  cli ; Do not interrupt
  sbi EECR, EEMPE ; Write program mode enable
  sbi EECR, EEPE ; Write program enable
  sei ; Interrupts allowed again
  cpi XL,Low(sCountersEnd) ; End of the Sram?
  brne Copy2Eep ; No, continue
Copy2EepEnd:
  sbic EECR,EEPE ; Wait for the last EEPE bit clear
  rjmp Copy2EepEnd ; Not yest
; Clear the SRAM counters
ClearOnly:
  ldi XH,High(sCounters) ; Point X to counters, MSB
  ldi XL,Low(sCounters) ; dto., LSB
  clr rmp ; Zero to all registers
ClearAll1:
  st X+,rmp ; Clear counter
  cpi XL,Low(sCountersEnd) ; All counters cleared?
  brne ClearAll1 ; Still counters to be cleared
  rcall Str2Sram ; Add strings to SRAM
.if Debug_SkipLcd == No
  ldi ZH,0
  ldi ZL,0
  rcall LcdPos
  ldi ZH,High(2*Text_Template) ; Z to text table
  ldi ZL,Low(2*Text_Template)
  rcall LcdText ; Display the complete text
  .endif
WaitKey:
  ; Wait for key release
  ldi ZH,High(16666) ; For 100 ms delay after last key event
  ldi ZL,Low(16666) ;  (See below for the formula)
WaitKey1:
  sbis pDcfKeyI,bKeyI ; Read key input pin, +2 = 2
  rjmp WaitKey ; Key input low, restart toggle counter
  sbiw ZL,1 ; Count toggle counter down, +2 = 4
  brne WaitKey1 ; Wait o, +1/2 = 5/6
  ; Time required:
  ;   clocks = 6 * (Z - 1) + 5 = 6 * Z - 1
  ;   Z = (clocks + 1) / 6
  ;   clocks = 100,000 for 100 ms
  ;   Z = 8334
  clr rFlag ; Clear the flag
  clr rmp
  out TCNT1H,rmp ; Clear the signal counter
  out TCNT1L,rmp
  ret

;
; *****************************************
;      T E X T   T E M P L A T E S
; *****************************************
;
; Initial text,  20 chars per line
Text_Intro:
  .db " DCF77 signal check ",0x0D,0xFF
  .db "  tn24_lcd module   ",0x0D,0xFF
  .db "                    ",0x0D,0xFF
  .db "   (C)2019 DG4FAC   ",0xFE,0xFF
;      01234567890123456789
;
Text_Template:
  .db "  Sp  <0   0  <1   1",0x0D,0xFF
  .db "   0   0   0   0   0",0x0D,0xFF
  .db "  <P   P  <M   M  >M",0x0D,0xFF
  .db "   0   0   0   0   0",0xFE,0xFF
;      01234567890123456789
;
; LCD routines include
.include "lcd.inc"
;
; End of source code
```

Praise, error reports, scolding and spam please via the comment page to me.