[MICROCONTROLLERS & EMBEDDED • ANALOGUE • AUDIO • DIGITAL • TEST & MEASUREMENT]

Exclusive E-Book for Members!

elektor

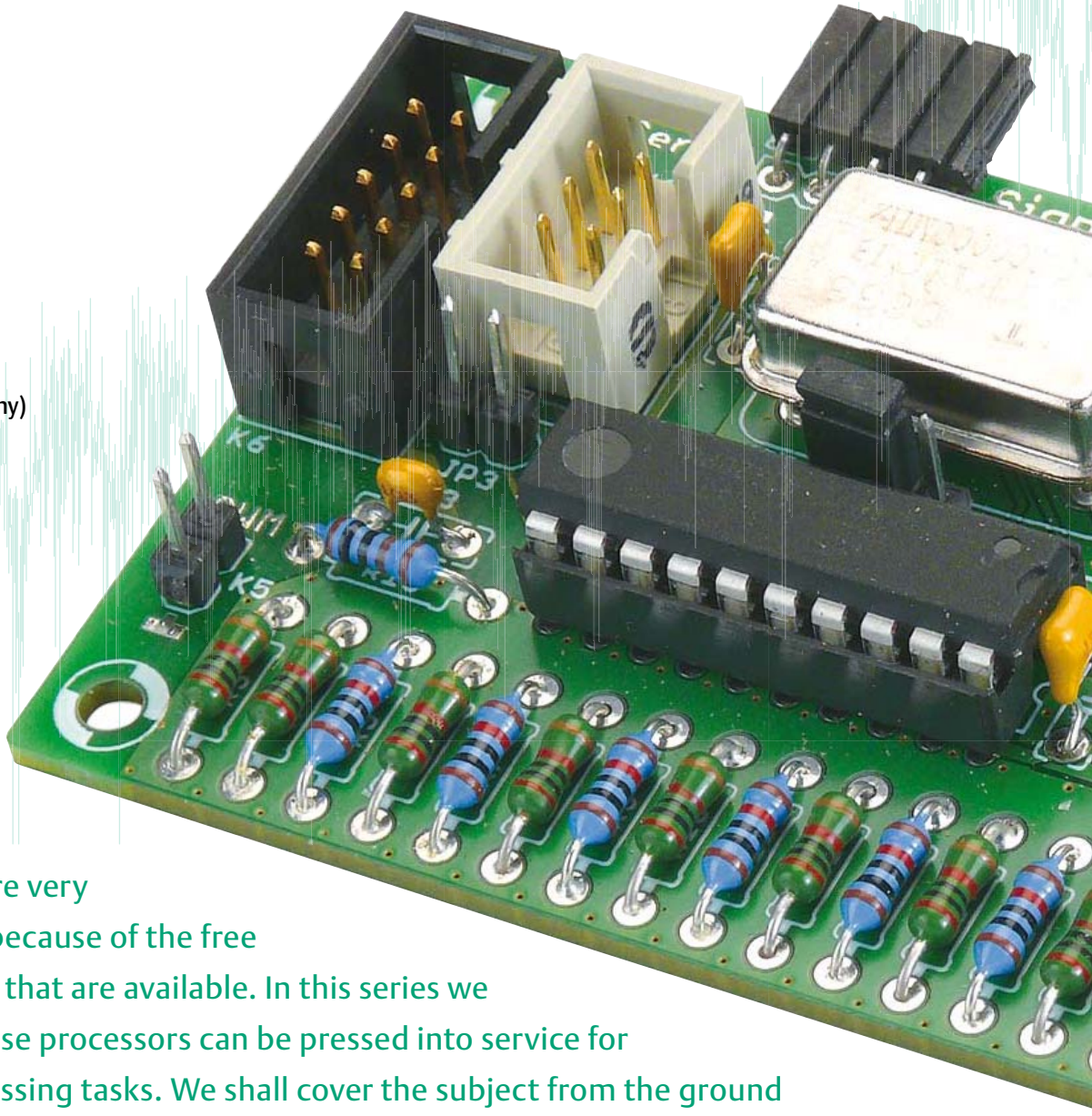
www.elektor.com

AVR Software Defined Radio

Generating precision signals using an ATtiny micro

AVR Software Defined

Generating precision signals using

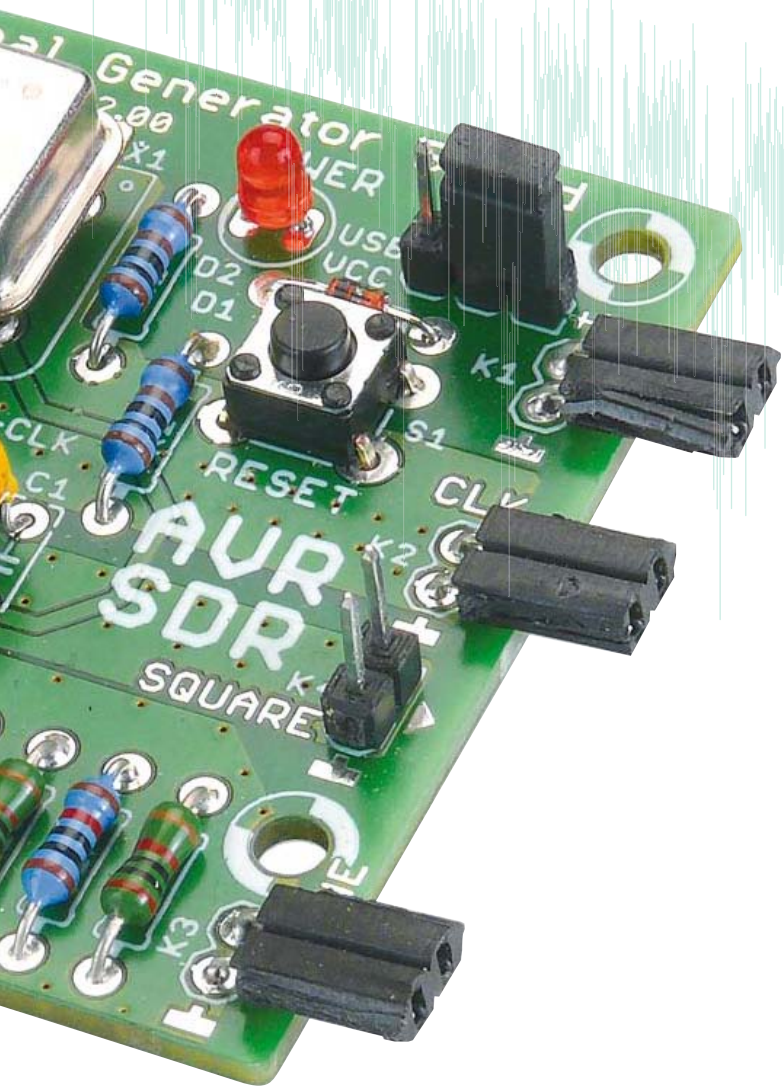


By Martin Ossmann (Germany)

Atmel AVR microcontrollers are very popular, not least because of the free development tools that are available. In this series we shall show how these processors can be pressed into service for digital signal processing tasks. We shall cover the subject from the ground up, making the series suitable for beginners, and in true Elektor style the focus will be on practical experiments. You can build the hardware yourself or you can obtain boards from Elektor, and the software is as ever available for download as source code from our website. Let's generate some signals!

Radio Part 1

g an ATtiny micro



First a quick peek at what is in store in this series. The first board, which includes an ATtiny2313, a 20 MHz oscillator and an *R-2R* DAC, will be used to make a signal generator. The second board will fish signals out of the ether. It contains all the hardware needed to make a digital software-defined radio (SDR), with an RS-232 interface, an LCD panel, and a 20 MHz VCXO (voltage-controlled crystal oscillator), which can be locked to a reference signal. The third board provides an active ferrite antenna. The software for all these projects is written using the WinAVR GCC compiler in AVR Studio and can be downloaded as C source code (plus fuse settings) or as hex files.

The series is built around practical experiments. We can look forward to sine- and squarewave generators, an RMS voltmeter, experiments in FM, AM and PM, FIR and IIR filters, wireless data transmission, reception of the DCF timecode signal, RTTY weather messages, BBC long-wave radio transmissions and much more!

Before we get started, one word of warning: fluorescent energy-saving light bulbs are based on switching regulators which splatter interference all over the long-wave band. So we advise carrying out the more sensitive experiments with the fluorescents off and the mains halogen lights on (or by candlelight!).

Signal generator board

The signal generator board is based on an AVR microcontroller clocked at 20 MHz and an *R-2R* ladder forming a digital to analogue converter to produce the output voltages. This is hardly a novel circuit, but we will show how it can be used in a wide range of applications. In particular we will use it to generate outputs useful for testing other circuits, such as frequency- and phase-modulated signals. Then, for even greater precision, we will connect the signal generator to an external clock source which is in turn locked to a frequency

Elektor Products and Support

- Signal generator kit including printed circuit board and all components: # 100180-71
- BOB-FT232R USB-to-TTL converter, ready built and tested: # 110553-91
- USB AVR programmer, printed circuit board with SMDs fitted, plus all other components: # 080083-71
- Free software download (hex files and source code): file # 100180-11.zip

All products and downloads are available via the web pages for this article: www.elektor.com/100180

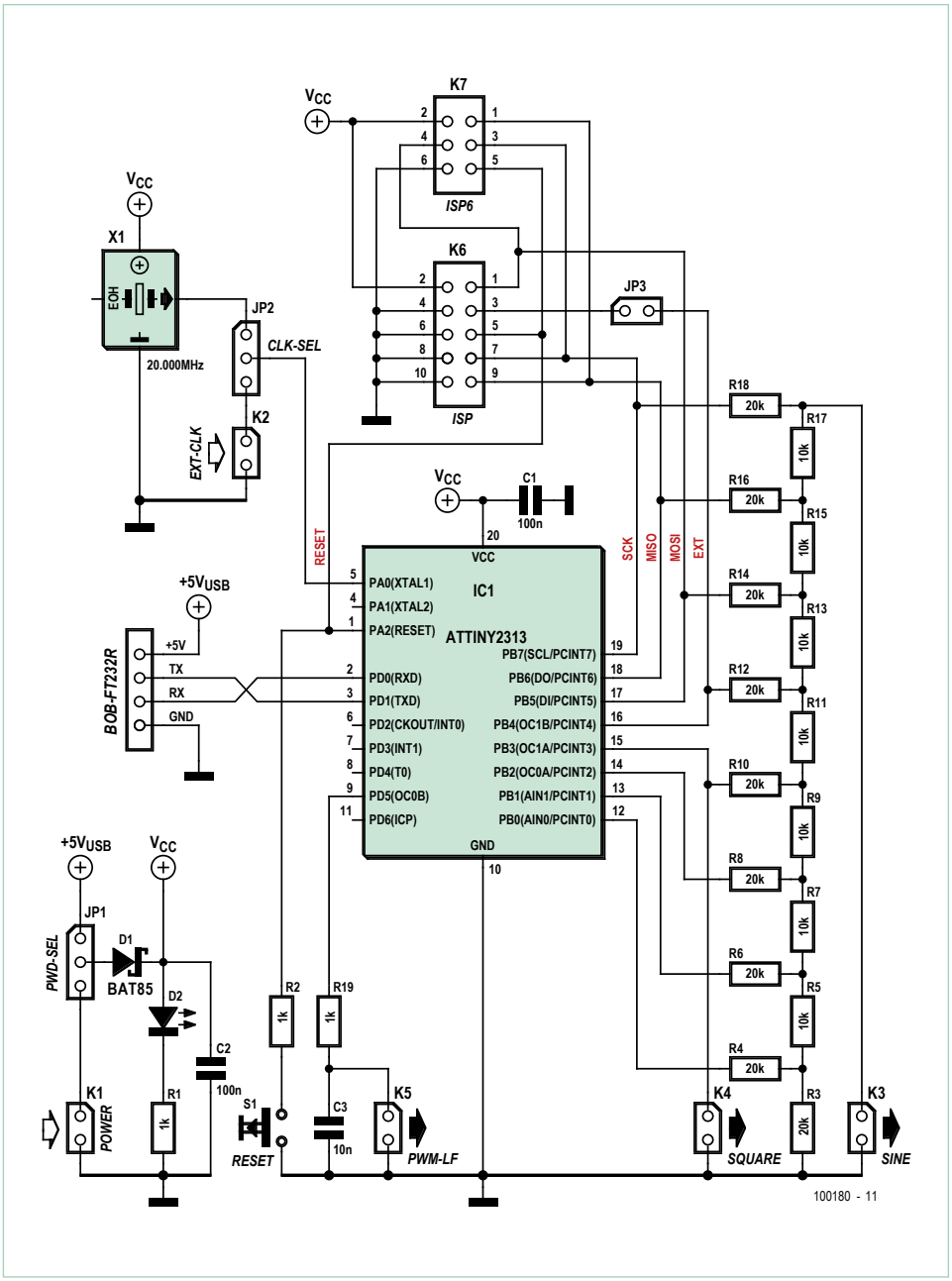


Figure 1. Circuit diagram of the signal generator.

standard such as the German DCF77 signal on 77.5 kHz or French TDF signal on 162 kHz.

The circuit of the signal generator is shown in **Figure 1**. The central component is the ATtiny2313 microcontroller, with the *R-2R* ladder connected to port B forming the digital-to-analogue converter. The analogue output signal appears on K3 (SINE). Note, however, that the output impedance of the circuit is relatively high at 10 kΩ. PWM output OC1A of the microcontroller is also available at K4 (SQUARE). We will use this output to generate square waves with frequencies of up to a few hundred kilohertz, as well as to modulate other signals. Another PWM output, OC0B, is brought out to K5 (PWM-LF) via a low pass filter comprising R19 and C3: this is suitable for generating low-frequency analogue signals.

The processor is clocked at 20 MHz by oscillator X1. It is a good idea to choose a relatively high-precision component here (50 ppm or better). Using a socket makes it easier to try out different types of oscillator or oscillators of different frequencies. Jumper JP2 allows the use of an external clock signal, which should be supplied at K2 (EXT-CLK).
The signal generator software programs allow a certain amount of external configuration using the microcontroller's UART. The relevant pins are brought out to a connector on the board (which is available from *Elektor* in the form of a kit including all the components). The connector is suitable for directly attaching the BOB-FT232R USB-to-serial converter [1]. JP1 allows power to be obtained over the USB connection when the unit is used with a PC: in this case no additional AC power adaptor is needed.
Populating the printed circuit board (**Figure 2**) should present no particular difficulties: all the components are ordinary leaded types. It is worth using a socket for the processor in addition to the clock oscillator. Be sure to observe correct polarity on the programming connectors K6 and K7. Programming can be done using the Elektor AVRprog [2]. It is of course important to get the fuse configuration right: the source code gives this along with the compiler options in each case.

DDS sinewave generator

Our first application is a simple sinewave generator programmed in C. The basic sample clock is produced by one of the timers built in to the microcontroller, arranged to trigger an interrupt. The interrupt routine is responsible for calculating the next sample value of the sinewave (**Figure 3**). Call the *k*th sample $S[k]$. Writing $p[k]$ for the phase of this sample, we have

$$S[k] = \sin(p[k]).$$

Between one sample and the next the phase advances by a constant value d (the 'phase increment'):

$$p[k+1] = p[k] + d.$$

In a perfect sinewave generator these calculations must be carried out exactly, which of course is not practical. Instead, the direct digi-

COMPONENT LIST

Resistors
R1,R2,R19 = 1kΩ
R5,R7,R9,R11,R13,R15,R17 = 10kΩ
R3,R4,R6,R8,R10,R12,R14,R16,R18 = 20kΩ

Capacitors
C1,C2 = 100nF 100V
C3 = 10nF

Semiconductors
D1 = BAT 85 (Schottky diode)
D2 = LED, green
IC1 = ATtiny2313-20PU, programmed

Miscellaneous
S1 = pushbutton
K4,K5 = 2-pin pinheader, lead pitch 0.1'' (2.54mm)
JP3 = 2-pin pinheader, lead pitch 0.1'' (2.54mm) with jumper
JP1,JP2 = 3-pin pinheader, lead pitch 0.1'' (2.54mm) with jumper

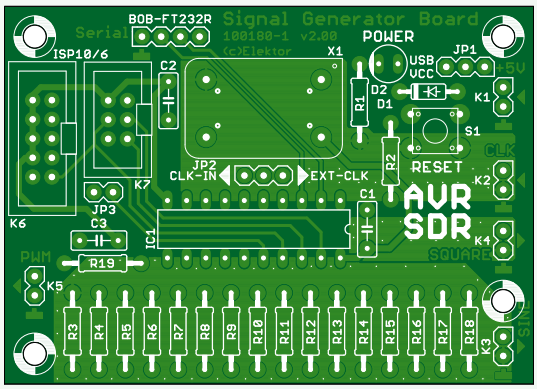


Figure 2. The printed circuit board is available from Elektor as part of a kit including all the components.

K1,K2,K3 = 2-way receptacle, right-angled
BOB = 4-way receptacle, right-angled
K6 = 10-way ISP boxheader
K7 = 6-way ISP boxheader
X1 = 20MHz quartz crystal (with 4 receptacles Harwin type H3153F01)
BOB-FT232R-001 = Elektor 'BOB' USB/TTL converter (ready assembled and tested, # 110553-91)
Printed circuit board

Alternatively
Kit, including board and all parts: # 100180-71.

tal synthesiser (DDS) stores the current phase value DDSp to finite precision as an *m*-bit number in the so-called 'phase accumulator'. One complete period of the sinewave corresponds to this value covering the range of values from 0 to 2^m–1. The same precision is used for storage of the phase increment and for the phase addition operation.

The next step is to convert the phase value into the corresponding sinewave sample. This is done using a look-up table which stores a complete sinewave period. If we were to store a sample for each of the 2^m possible values in the phase accumulator the table would be unmanageably big: instead we use just the top *n* (where *n*<*m*) bits of the phase accumulator to index the table, which now need only contain 2^n samples. The values stored in the table are rounded from their exact values to *r*-bit samples *S*[*k*], where *r* is the number of bits in the digital-to-analogue converter. **Figure 4** illustrates the process. In our first program we use *m*=32 and *n*=8. A 32-bit phase accumulator gives us very precise frequency control over our signals. We use a sine table with 256=2^8 entries and an 8-bit DAC (*r*=8). The program EXP-SinusGeneratorDDS-T1INT-V01.c [3] is configured to produce a fixed frequency output at 1 kHz; the result can be verified on an oscilloscope (**Figure 5**). The interrupt service routine code is shown in **Listing 1**.

Timing

The DDS is clocked at f_{DDSCLK} = 100 kHz. To generate a desired output frequency *f* the required phase increment is calculated using

$$DDSd = 2^n \times f / f_{DDSCLK}$$

and so for *f* = 1 kHz we have

$$DDSd = 2^{32} \times 1\ kHz / 100\ kHz = 42949673$$

and you will find precisely this value in the C source code where DDSd is initialised.

We can see from this formula that the higher the sample rate the higher the frequencies we can generate. Why did we choose 100 kHz? The answer lies in the timing of the interrupt service rou-

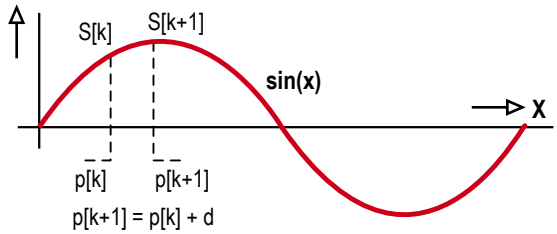


Figure 3. Sampling a sinewave.

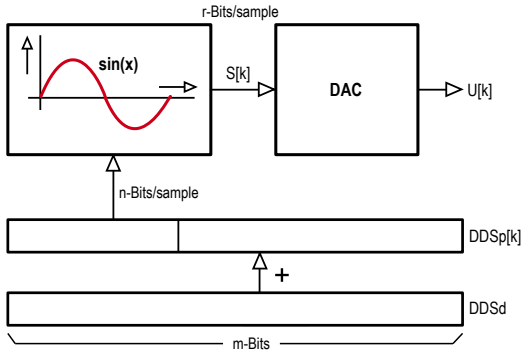


Figure 4. How the DDS sinewave generator works.

Figure 5. Testing the sinewave generator.

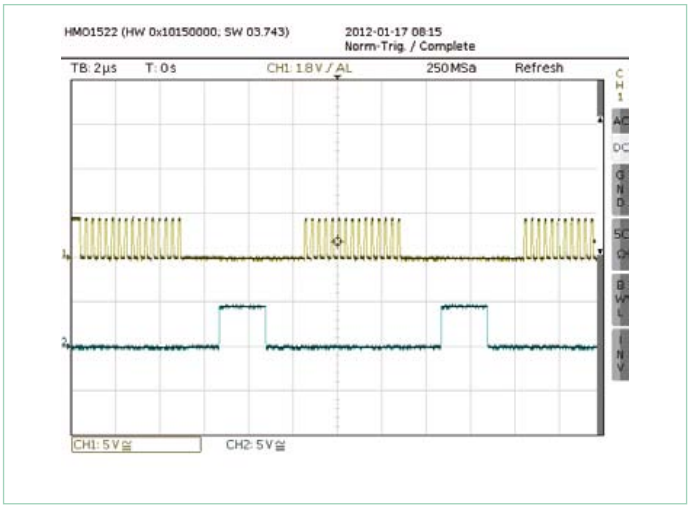


Figure 6. Measuring the execution time of the interrupt service routine.

tine. As you can see from the code snippet above, we have bracketed the calculation with commands to set and clear port pin PD.4. This allows us to observe the execution time of the calculation using an oscilloscope: in this case we see a total time of around 2.2 µs. However, we must be careful as this does not include other contributions to the total time needed to service the interrupt: for example, the time to save and restore processor registers will not be counted. However, with a relatively simple experiment we can determine these times as well.

Simply set up the main program as an infinite loop in which a port pin (say PD.5) is toggled as quickly as possible. We can then observe on the oscilloscope the periods when the toggling stops, which is when the interrupt service routine is active: see **Figure 6**.

In our experiment we measured the total time needed to process an interrupt at about 5.4 µs. The maximum allowable interrupt rate is therefore 180 kHz. Adding a safety margin, we arrive at our figure of 100 kHz.

As the output frequency f approaches f_{DDSCLK} we start to observe undesirable artefacts such as jitter, noise and aliases in the output spectrum. With a sample rate of 100 kHz it is best to keep f below about 10 kHz. Perhaps we can do a bit better if we use assembly code?

A faster DDS sinewave generator

In order to make our sinewave generator capable of higher frequencies we need to rewrite the DDS routine in assembler. With the help of a cunning arrangement of variables in registers we can manage to get the sample rate of the 32-bit DDS as high as 2 MHz. The code **(Listing 2)** uses the T flag to allow it to break out of its loop.

Our project now consists of a mixture of C and assembler code, and we need to store the sinewave table at a fixed address in memory. Configuring the project within WinAVR to achieve this is not a task for the beginner. If you do not plan to make any changes to the code it is probably best to program the ready-compiled hex file into the

Listing 1

```
ISR(TIMER1_OVF_vect) {
  PORTD |=  _BV(4) ;                          // set sample timing flag
  PORTB=pgm_read_byte( SIN8+(DDSp>>24)) ;     // fetch and output sine sample
  DDSp += DDSd ;                              // advance DDS phase DDSp by DDSd
  PORTD &= ~  _BV(4) ;                        // clear sample timing flag
  }
```

Listing 2

```
loop:
    add  DDSphase0,DDSdelta0       // 1     LSB of 32 bit DDS adder
    adc  DDSphase1,DDSdelta1       // 1
    adc  DDSphase2,DDSdelta2       // 1
    adc  ZL       ,DDSdelta3       // 1     MSB is in ZL as pointer
    lpm  R0,Z                      // 3     access sine table
    out  PORTB,R0                  // 1     out to R-2R DAC on PORTB
    brtc loop                      // 2 (1) loop until T flag set by interrupt routine
                                   // 10 cycles in total for one loop
```

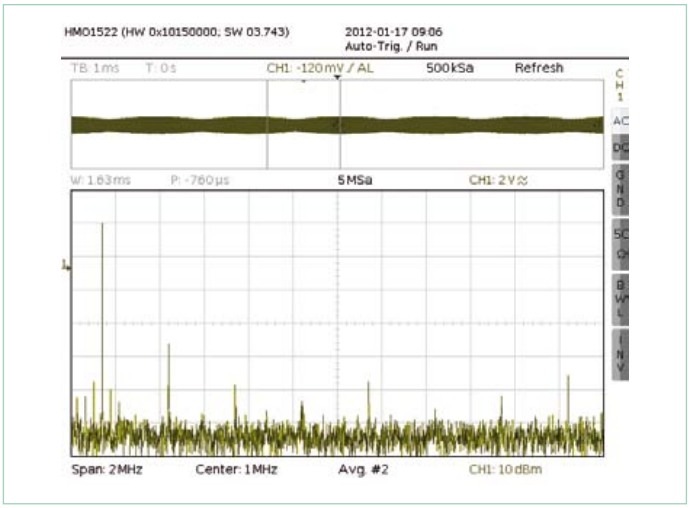


Figure 7. Spectrum of the generated signal.

processor (paying attention to the fuse bit settings). The project is called EXP-SinusGenerator-DDS-ASM-C-V01.
To make the sinewave generator more flexible it includes the ability to be configured over the UART interface (19200 baud, 8N1 data format). Using a terminal program, simply type in the desired frequency followed by CR and LF. The maximum usable signal frequency is about 200 kHz. The theoretical frequency resolution is given by

$$f_{DDSCLK} / 2^n = 2\ \text{MHz} / 2^{32} = 0.00046\ldots\ \text{Hz}.$$

To take advantage of this resolution the software allows you to enter a frequency with up to three digits after the decimal point, for example as '1000.045' (followed by CR and LF). The internal calculations required to turn the entered frequency into a suitable parameter value for the DDS need to be carried out very accurately. To this end the author has written special-purpose arithmetic routines, including one for fixed-point division.

Figure 7 shows the spectrum of the sinewave output signal at frequency f = 125.123 kHz over the range from 0 Hz to 2 MHz. As you can see, there are harmonics present, but all at more than 30 dB below the desired signal. A low level of wideband noise is also visible: this is a by-product of the DDS technique.

Listing 3

```
uint32_t DDS24 ;                          // DDS phase, 24 bits used
volatile uint32_t dDDS24 ;                // delta for DDS
uint16_t TOP1 ;                           // integer part of divider for PWM

ISR(TIMER1_OVF_vect) {
  PORTD |=  _BV(4) ;
  DDS24 += dDDS24 ;                       // advance DDS phase
  if (DDS24 & 0x1000000UL ) {             // check bit 24 for overflow
    ICR1 =TOP1 ;                          // on overflow PWM width = TOP1+1
    }
   else {
    ICR1 =TOP1-1  ;                       // else PWM width = TOP1
    } ;
  DDS24 &=0xffffffUL ;                    // make DDS24 24 bits again
  PORTD &= ~  _BV(4) ;
  }
```

If an ordinary crystal oscillator is used the overall accuracy of the system will be in the region of plus or minus 100 ppm. In that context it hardly makes sense to claim that the generator can produce an output frequency of say 100.00005 kHz. For this reason the generator can accept an external 20 MHz clock signal, and in a later instalment of this series we will see how such a clock can be derived from a transmitted reference. This combination allows the generation of sinewaves with excellent frequency accuracy.

Trimming resonant circuits

Later in this series we will use an AVR microcontroller to receive and process longwave transmissions such as the DCF time code on 77.5 kHz, France Inter on 162 kHz and BBC Radio 4 on 198 kHz. Usually a ferrite antenna will be used, and adjusting such an antenna can be made much simpler using our sinewave generator: connect the circuit up as shown in **Figure 8** and adjust the trimmer capacitor for maximum amplitude.

It is possible to use the phase relationship between the output voltage U_{OUT} and the input voltage U_{IN} to determine whether the resonant frequency of the circuit is higher or lower than that of the input sinewave. If the phase of U_{OUT} leads U_{IN} then the sinewave frequency is lower than the resonant frequency; if U_{OUT} lags U_{IN} the signal frequency is higher than the resonant frequency. When the frequencies agree U_{IN} and U_{OUT} are in phase.

The example circuit shows component values for a resonant frequency of 125 kHz; coil L1 is a small pot core inductor. We will use this circuit later to generate test signals at a frequency of 125 kHz. The trimmer capacitor can be adjusted to bring the resonant frequency of the circuit to exactly 125 kHz: it is possible to use either the signal from the *R-2R* ladder (K3) or the squarewave from the PWM output (K4) to do this.

PWM squarewave with a fractional divider

We will now look at another application of the DDS principle. If we use a timer with a PWM output to generate a squarewave, we are normally limited to producing frequencies that exactly divide the frequency at which the timer is clocked. If N is the division ratio and f_{CLK} the timer's clock frequency then the output frequency will be $f = f_{CLK}/N$. However, if we adjust the division ratio on the fly (say between N and N+1) then we can also produce intermediate frequencies. So for example if we alternate between using a divisor of N and a divisor of N+1 then the overall effective division ratio will be N+0.5. If a division ratio of 10.333... is wanted, then this can be achieved by using a division ratio of 11 with 'probability' 0.333... and a division ratio of 10 the rest of the time.
How can we use this in practice? We need an algorithm that will tell us, given the desired division ratio, when to divide by N and when by N+1.
Again, the m-bit DDS synthesiser comes to the rescue, with a sufficiently large value of m to achieve the desired precision. In this case we make use of the overflow of the phase accumulator. The rate p at which an m-bit phase accumulator overflows is just

$$p = \mathrm{DDSd} / 2^m$$

and so we can control this rate as precisely as we need using the variable DDSd. Whenever the accumulator overflows the timer is instructed to divide by N+1 rather than N on the next cycle.
So, for example, if we wish to generate a 77.5 kHz squarewave from a 20 MHz master clock, the required overall division ratio is

$$20000 / 77.5 = 258.0645161...$$

and so we need to divide by either N=258 or N=259 on each cycle. The 'probability' of selecting N=259 will be p = 0.0645161..., which with a 24-bit DDS means that DDSd = $p \times 2^{24}$ = 1082401. **Listing 3** shows an interrupt service routine embodying this idea.

The squarewave produced by this code does of course suffer from a small amount of short-term jitter, but in the long term the agreement with the ideal frequency is excellent.

The whole routine, including interrupt overheads, has an execution time of about 6 µs, which means that we can use the technique to generate frequencies of up to about 160 kHz. Rewriting the routine in assembler should allow considerably higher frequencies.
For convenience the squarewave generator program can also be controlled using a terminal. The source code is called EXP-SquareGenerator-DDS-T1INT-V01.c in the downloadable zip archive [3].
The fractional divider has many other applications. For example, it can be used to generate a signal with any desired sample rate, or form part of a digital PLL.

FM generator

On its own the squarewave generator is perhaps not particularly exciting. However, since controlling the PWM output does not occupy all the processor's time, we have some computing power left over to change the frequency dynamically to make an FM generator.

The German meteorological service [4] transmits weather information on 147.3 kHz using frequency shift keying (FSK) in radioteletype (RTTY) format. Later in this series we will build a receiver capable of decoding these messages. To adjust and test the receiver it is helpful to have a test signal. Using a fractional divider and the PWM output this is easy: we just use a stream of data bits to control the output frequency.

We will first program our test signal generator to work with a carrier frequency of f = 125 kHz. A circuit along the lines of Figure 8 is used to turn the squarewave output into a sinewave. We have already seen the interrupt service routine for the fractional divider; the new routine, 'SendBit', is responsible for sending a single bit **(Listing 4)**.

First we wait for Timer 0 to run through COUNT2 cycles: in other words, the bit rate is the Timer 0 overflow rate divided by COUNT2. Then, depending on the value of the bit to be sent, the values of deltaDDS24 and TOP1 are set. This is where the frequency modulation occurs. Note that the commands that set these values are enclosed between a cli() and sli() pair. If this is not done it is possible that the interrupt service routine could be invoked when one parameter has been changed but not the other, with potentially unpredictable results. The routines can be found in the program EXP-SQTX-FM-RTTY-V01.c. With further auxiliary routines we can transmit characters using the Baudot [5] code, emulating the meteorological service transmissions.

Figure 9 shows the spectrum of the FM RTTY signal. There are two narrow peaks adjacent to one another, at frequencies of 125 kHz ± 50 Hz. The spectrum is continuous, falling off rapidly beyond ± 1 kHz.

Having built such a transmitter, it is natural to want to test it to check that the modulation is correct. For that reason the next instalment in this series will start to look at the components that comprise a digital receiver.

(100180)

Internet Links

[1] www.elektor.com/110553

[2] www.elektor.com/080083

[3] www.elektor.com/100180

[4 www.hfunderground.com/wiki/RTTY_maritime_weather_transmissions

[5] http://en.wikipedia.org/wiki/Baudot_code

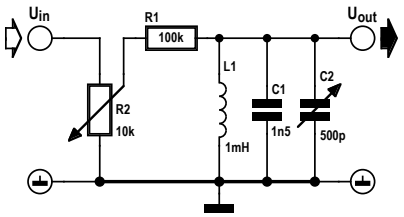


Figure 8. Trimming a resonant circuit.

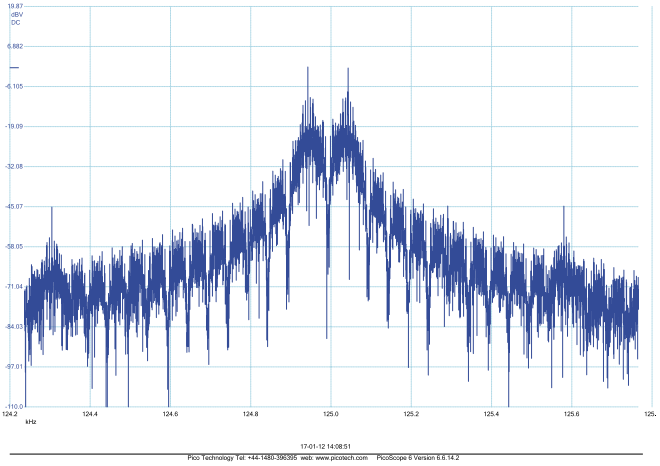


Figure 9. Spectrum of a frequency-modulated signal at 125 kHz ± 50 Hz.

Listing 4

```
void SendBit(uint8_t theBit) {
  uint8_t k ;
  for (k=0 ; k<COUNT2 ; k++){
    while( ( TIFR & (1 << TOV0) )== 0 ) { }
    TIFR |= (1 << TOV0) ;
    }
  if ( theBit==MARK  ) {
    cli() ;
    deltaDDS24=MARK_deltaDDS24 ;
    TOP1=MARK_TOP1 ;
    sei() ;
    }
   else{
    cli() ;
    deltaDDS24=SPACE_deltaDDS24 ;
    TOP1=SPACE_TOP1 ;
    sei() ;
    }
  }
```

AVR Software Defined Radio (2)

Sampling signals

By Martin Ossmann (Germany)

As this series shows, the popular AVR microcontroller can be used for digital signal processing tasks. Here we will use an ATmega88 to sample amplitude- and phase-modulated signals which we can either synthesise ourselves or fish out of the ether. We can even operate at frequencies of above 100 kHz. How does it work? Read on to find out, in theory and in practice.

A carrier wave can be used to send audio or data through the ether by modulating it in amplitude, frequency or phase. In a 'software defined radio' the first thing that happens is that the received signal is sampled; then a processor carries out the necessary calculations to recover the modulating signal. In the case of data transmission, we then decode the signal into a stream of bits. To understand how this all works, we will first look at how an analogue receiver operates.

Reception, the analogue way

The input stage of many modern receivers looks like **Figure 1** (where we have not shown the first stage of 'preselection' filtering). It works as follows: suppose first that we want to receive a signal at U_{in} with a frequency of f_{RX} = 2 kHz. We set the frequency f_{LO} of the local oscillator (or 'LO') also to f_{LO} = 2 kHz. In the upper branch of the circuit (which is called the 'in-phase' or 'I' channel) the input signal is multiplicatively mixed with the cosine wave produced by the LO. This produces a DC component *X*, which passes unchanged through the low-pass filter that follows. A component at 4 kHz is also produced, which is removed

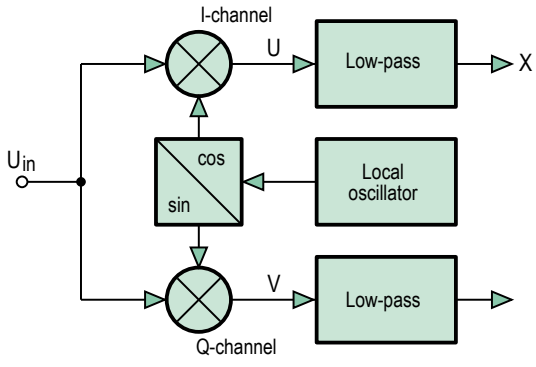


Figure 1. Quadrature mixing.

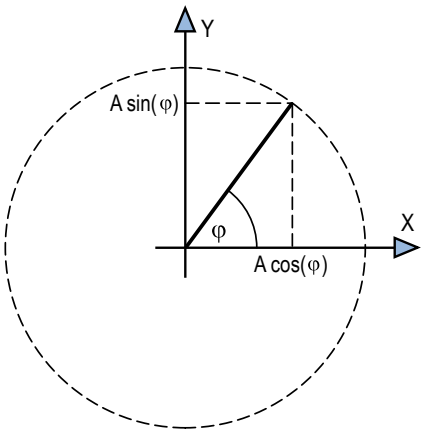


Figure 2. Geometrical interpretation.

by the low-pass filter. The value of *X* depends on the amplitude *A* and phase φ of the input signal, the phase being measured relative to the output of the local oscillator. More precisely, ignoring any gain in the low-pass filter, we have $X = A \cos \varphi$. If the input signal is exactly in phase with the cosine output of the local oscillator, *X* is maximised: this is why this branch is labelled 'in-phase'.

Much the same happens in the lower branch of the diagram. The difference is that the input is mixed with a sine signal (the cosine signal with a 90 ° phase shift). The value of *Y* depends again on the amplitude *A* and phase φ of the input signal, and we have $Y = A \sin \varphi$. *Y* is maximised when the input signal is 90 ° phase shifted with respect to the cosine output of the LO. For this reason, the lower branch is called the 'quadrature' channel.

Figure 2 shows the relationships graphically. The receiver can calculate the amplitude *A* and phase φ of the input signal from the values of *X* and *Y*.

Let's get digital

Now let's consider what happens when all the signals involved are sampled at a sample rate f_S = 8 kHz, exactly four times the frequency of the input signal (see **Figure 3**).

The process of sampling converts the continuous-time input signal into a sequence of numbers. If the input signal U_{in} is a cosine wave with amplitude *A* and frequency 2 kHz (at the top of Figure 3), sampling generates the sequence of values $U_{in} = A, 0, -A, 0, A$ and so on. The values repeat every four samples, because we are sampling at four times the input frequency.

Let us look first at the in-phase channel and sample the cosine output signal of the local oscillator. The sequence of samples is LO_{COS} = 1, 0, –1, 0, 1, ... Again, this repeats every four samples. The mixer multiplies the samples of U_{in} by those of LO_{COS}. The result is the sequence $U = A, 0, A, 0, A, ...$ After the mixer this sequence is passed through a low-pass filter. We can construct a simple low-pass filter by calculating a rolling average of sequences of four consecutive samples of *U*. For simplicity we multiply this result by 2, and the output of the filter is then $X = A, A, A, A, A, ...$; in other words, the output is a constant with value *A*. *X* can be thought of as samples of a DC level of *A*, where *A* is exactly the amplitude of the original input signal.

Now we turn to the quadrature branch. The inputs to this branch's mixer are the sequences $U_{in} = A, 0, -A, 0, A, ...$ and $LO_{SIN} = 0, 1, 0, -1, 0, ...$ The product of these sequences is $V = 0, 0, 0, 0, 0, ...$; in other words, the constant value zero. The result of low-pass filtering *V* will also be zero.

The same argument can be applied when the input signal is a sine wave $U_{in} = A \sin (2\pi \cdot 2\text{ kHz} \cdot t)$, giving the results $X = 0$ and $Y = A$. This shows that our discrete-time I-Q mixer works in just the same way as the classical analogue I-Q mixer described above. We have also seen that if the sample rate is four times the signal frequency, the output signals of the LO only take on the values zero, plus one and minus one. This in turn means that the digital mixer does not need a multiplier: we simply need to add and subtract the relevant input samples in the low-pass filter to calculate the values of *X* and *Y*.

The hardware...

A simple front end circuit (**Figure 4**) was designed to test this idea on an AVR microcontroller. The analogue-to-digital converter inside the ATmega88 is used to sample the input signal U_{in} and digitise it. The firmware then carries out the necessary calculations and the results, *X* and *Y*, are output using PWM on pins OC0A and OC0B. To

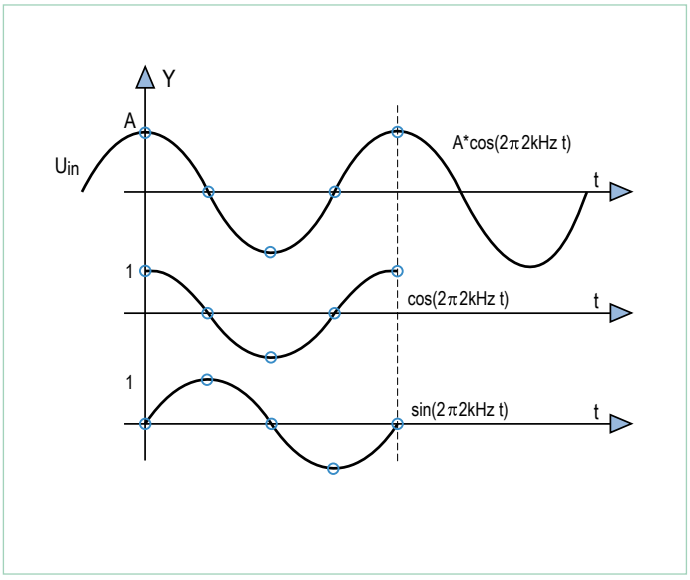


Figure 3. Sampling at four times the signal frequency.

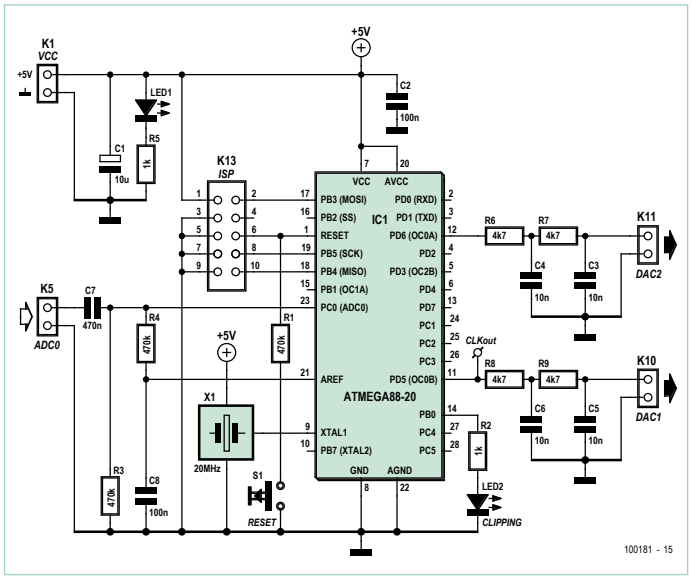


Figure 4. Hardware for a simple front end.

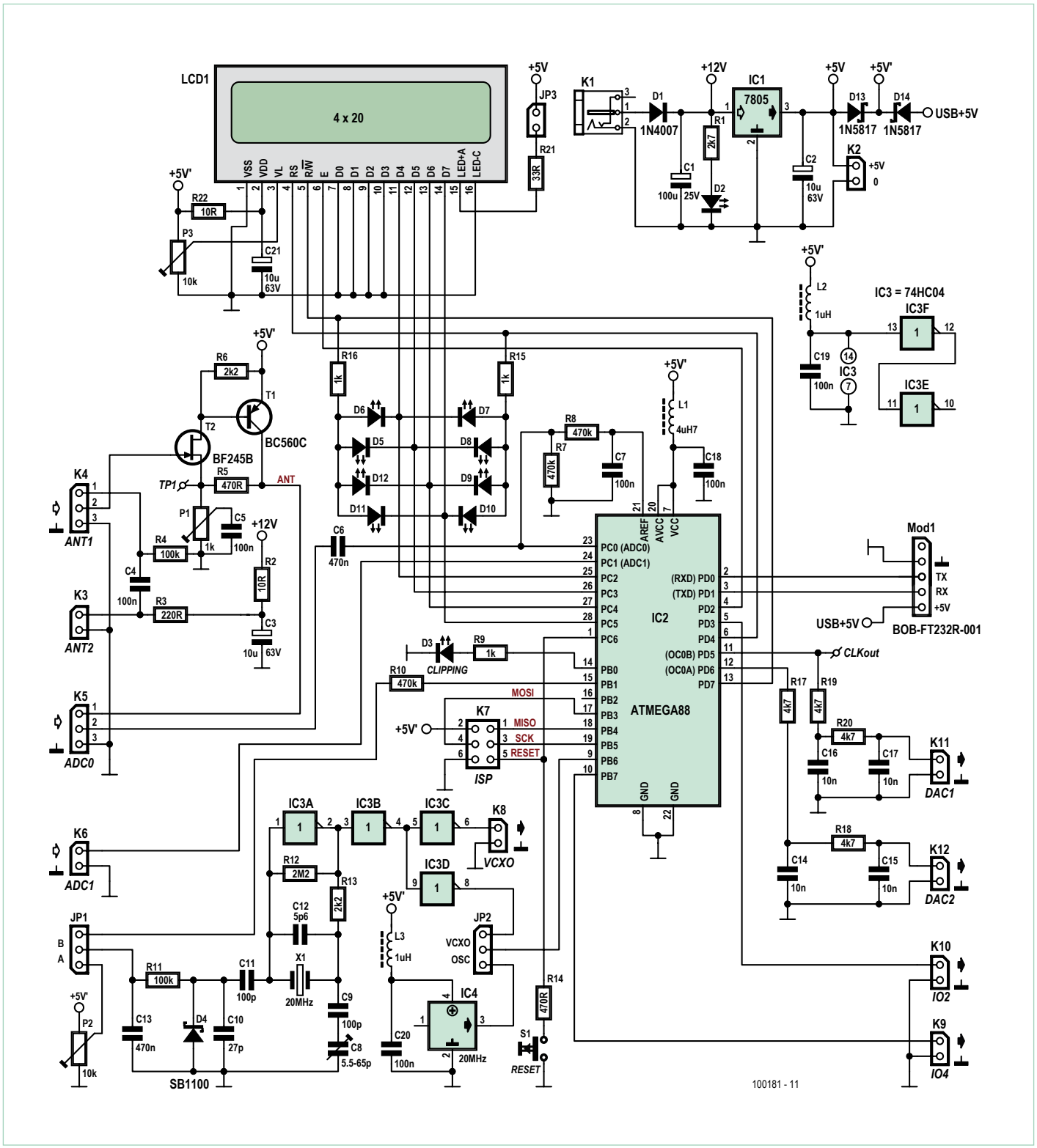


Figure 5. Circuit diagram of our universal receiver board.

COMPONENT LIST

Resistors (5%)
R1 = 2.7kΩ
R2,R22 = 10Ω
R3 = 220Ω
R4,R11 = 100kΩ
R5,R14 = 470Ω
R6,R13 = 2.2kΩ
R7,R8,R10 = 470kΩ
R9,R15,R16 = 1kΩ
R12 = 2.2MΩ
R17,R18,R19,R20 = 4.7kΩ
R21 = 33Ω
P1 = 1kΩ 20%, 0.15W trimpot
P2,P3 =10kΩ 20%, 0.15W trimpot

Capacitors
C1 = 100µF 25V, radial
C2,C3,C21 = 10µF 63V, radial
C4,C5,C7,C18,C19,C20 = 100nF 50V
C6,C13 = 470nF 63V
C8 = 5.5–65pF 150V trimmer
C9,C11 = 100pF 5% 100V
C10 = 27pF 2% 100V
C12 = 5.6pF ±0.25pF 100V
C14,C15,C16,C17 = 10nF 5% 50V

Inductors
L1 = 4.7µH, 190mA, 1.7 Ω
L2,L3 = 1µH, 270mA, 0.8 Ω

Semiconductors
D1 = 1N4007
D2,D3,D5–D12 = LED, red
D4 = SB1100
D13,D14 = 1N5817
T1 = BC560C
T2 = BF245B
IC1 = 7805
IC2 = ATmega88-20PU, programmed
IC3 = 74HC04
IC4 = 20MHz oscillator module

Miscellaneous
X1 = 20MHz quartz crystal, 50ppm
S1 = pushbutton SPST-NO, 6mm footprint
K1 = low voltage adapter socket
K2,K6,K8 = 2-pin pinheader, right angled, lead pitch 0.1 in. (2.54mm)
K3,K9–K12,JP3 = 2-pin pinheader, lead pitch 0.1 in. (2.54mm)

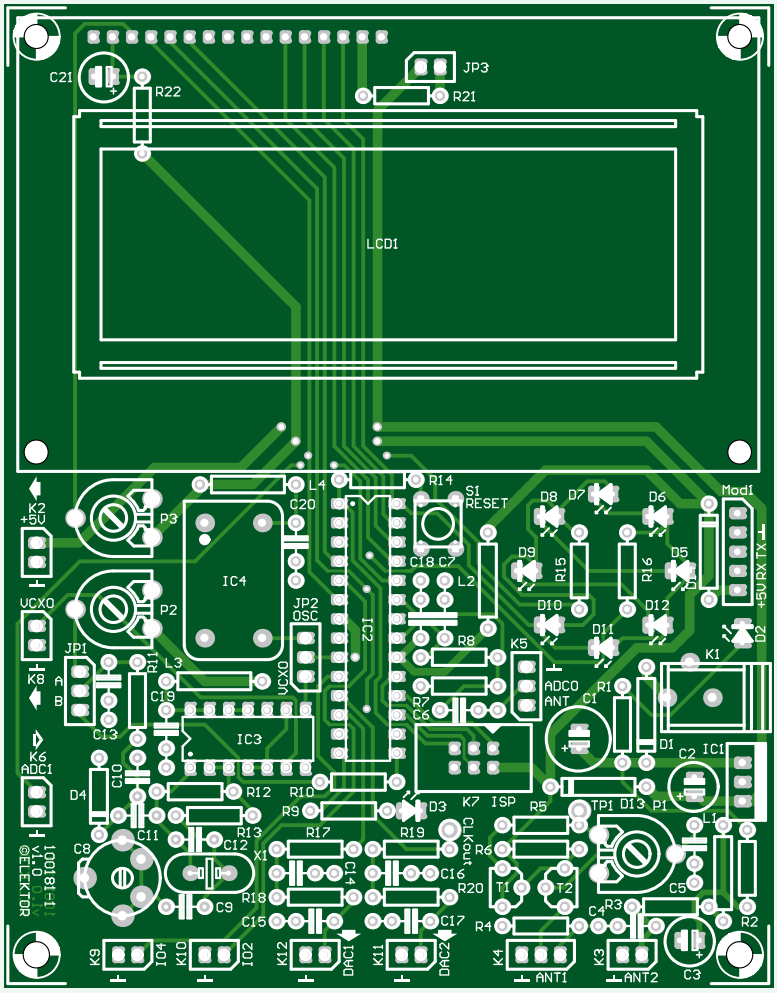


Figure 6. The printed circuit board is available from Elektor as part of a kit including all the components.

K4,K5,JP1,JP2 = 3-pin pinheader, lead pitch 0.1 in. (2.54mm)
K7 = 6-pin pinheader, lead pitch 0.1 in. (2.54mm)
JP1,JP2,JP3 = jumper 0.1 in. (2.54mm)
CLKout,TP1 = PCB solder pin

4 IC pin receptacles (for IC4)
Mod = 5-way pinheader for Elektor BOB-FT232R-001
LCD1 = 4x20 LCD e.g. HC200401C-YF62L-VA
PCB # 100181-1

attenuate the PWM frequency component in these output signals, each is passed though a two-stage RC low-pass filter.
The circuit is straightforward enough to be built on a small piece of prototyping board. Things are made even easier if the Elektor universal receiver board is used: its circuit diagram is shown in **Figure 5**. As was the case for the signal generator described in the first part of this series [1], this is available as a kit including the printed circuit board (**Figure 6**) and all components. This is a good option, as populating the board is not a tricky task. As you can see from the circuit diagram the universal receiver board includes all the components of the simple front end, but also allows for a wide range of additional future possibilities that we will look at later on in this series. For example, it is possible to connect an active ferrite antenna: an example of such an antenna is again available as an *Elektor* kit, and

Listing 1: Calculating the quadrature components

```
U=0 ;
if (sampleTime==0){ U=   ADCv ; }
if (sampleTime==2){ U= - ADCv ; }
U3=U2 ; U2=U1 ; U1=U0 ; U0=U ;
X=U0+U1+U2+U3 ;
OCR0A=128+X/8 ;

V=0 ;
if (sampleTime==1){ V=   ADCv ; }
if (sampleTime==3){ V= - ADCv ; }
V3=V2 ; V2=V1 ; V1=V0 ; V0=V ;
Y=V0+V1+V2+V3 ;
OCR0B=128+Y/8 ;
```

the electronics and printed circuit board will be described in the next instalment in this series.

... and the software...

The software, as always, is available as source code and as a hex file for download from the Elektor web site [2]. For our first experiment the software we use is called `EXP-SimpleFrontend-2kHz-IQout-V01.c`.
The program samples the input signal at a rate of 20 MHz / 5000 = 8 kHz. The signal is then mixed with a 2 kHz signal. A simple low-pass filter then produces the *X* and *Y* outputs, which we can also label as the quadrature components *I* and *Q*.

Listing 1 shows the heart of this routine. The timer variable `sampleTime` always counts cyclically from 0 to 3, and thus represents the current phase of the local oscillator. The variables `U` and `V` are used to hold the values that are obtained by multiplying the input value `ADCv` by the cosine sequence [1, 0, –1, 0] and the sine sequence [0, 1, 0, –1] respectively. The values of `U` and `V` are then fed into a simple low-pass filter that calculates a rolling average over sets of four samples. The results of this calculation, `X` and `Y`, are the in-phase and quadrature components of the signal and are written to the PWM registers OCR0A and OCR0B respectively.

... and, finally, testing

To test the receiver, we feed a sinewave signal from our signal generator into the front end hardware via a 10 kΩ potentiometer to allow adjustable attenuation. We use the SINE OUT (K3) output of the signal generator and the program called `EXP-SinusGenerator-DDS-ASM-C-V01`; the wiper of the potentiometer is connected to input ADC0 on the receiver board.
The analogue outputs DAC1 and DAC2 are connected to an oscilloscope operating in X-Y mode. Then we instruct the signal generator over its RS-232 interface to generate a 2 kHz sinewave [1] and adjust the amplitude of the signal using the potentiometer until the red LED (D3 in Figure 5) does not quite light. The front end is then being driven to its maximum level, just short of clipping. The oscilloscope should show a single point that moves slowly in a circle. In theory the point should be stationary, but because the oscillator controlling the front end is not running at exactly the same frequency as that controlling the signal generator, the point will move.
To see the effect more clearly, adjust the signal generator to produce a frequency of 2005 Hz. Then the point will move in a circle making five revolutions per second. With the signal generator set to 1995 Hz, the point will again move at five revolutions per second, but in the opposite direction. Adjusting the amplitude of the input signal using the potentiometer affects the radius of the circle in which the point moves.
Our 'I-Q demodulator' has mixed the signal in the band around 2 kHz down to a centre frequency of 0 Hz. Signals in sidebands above and below 2 kHz are now distinguished in the direction of rotation of the point on the oscilloscope display. Now frequencies around 2 kHz are of relatively little practical interest: more interesting are frequencies in the longwave bands used for sending data by various transmitters. This requires one further small step, as we shall see in the next section.

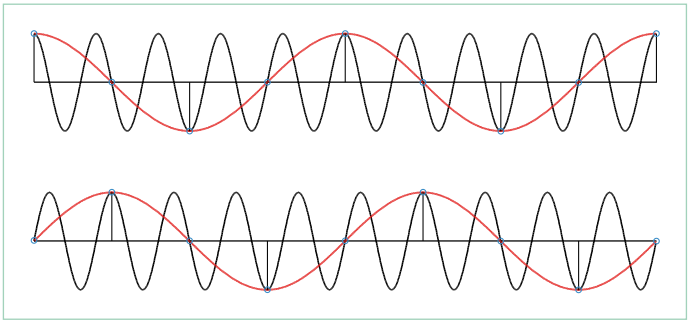


Figure 7. Sampling a 10 kHz signal and a 2 kHz signal (red) at 8 kSa/s.

The road to RF

The Nyquist-Shannon sampling theorem states that a sample rate of at least 2 *f* is required to represent losslessly a signal containing frequency components up to a frequency *f*. Using a lower sample rate than this is called 'undersampling' or 'sub-Nyquist sampling'. Take a look at **Figure 7**. The time interval illustrated is 1 ms long. The upper black curve is a cosine wave, with the corresponding sine wave below, both at 10 kHz. As before, the signal is sampled at 8 kSa/s (kilosamples per second), giving sample values indicated by the small blue circles. With conventional sampling we need at least two samples per period of the 10 kHz signal, but here we have less than one sample per period: hence the signal is undersampled. The sequence of sample values that we obtain is [1, 0, –1, 0, 1, 0, –1, 0, 1] for the upper signal and [0, 1, 0, –1, 0, 1, 0, –1, 0] for the lower one. Superimposed on the figure, in red, are 2 kHz cosine- and sinewaves. For these signals the sample rate of 8 kSa/s satisfies the sampling theorem; but the surprise is that the 2 kHz signal and the 10 kHz signal give rise to the same set of sample values. This means that a 10 kHz signal sampled at 8 kSa/s is indistinguishable from a 2 kHz signal sampled at the same rate. In turn this means that our front end, which uses an 8 kSa/s sample clock, can equally well be used to demodulate signals at 10 kHz.

There is of course a full theoretical analysis of the above phenomenon, known as the Nyquist-Shannon sampling theorem for band-

pass signals. One consequence of this theorem is that a sampled bandpass signal with bandwidth B starting at a multiple of the sample rate f_S can be reconstructed perfectly as long as $B < f_S / 2$. In particular, we can reconstruct a bandpass signal with components stretching from $n \times f_S$ to $n \times f_S + f_S / 2$ for any chosen integer n; or, stated another way, a bandpass signal centred at $n \times f_S + f_S / 4$ with total bandwidth up to $f_S / 2$. In particular, if we have a sample rate of 8 kHz we can demodulate signals equally well around any of the following frequencies: 2 kHz, 2 kHz + 1 × 8 kHz = 10 kHz, 2 kHz + 2 × 8 kHz = 18 kHz, 2 kHz + 3 × 8 kHz = 26 kHz, ..., 2 kHz + 20 × 8 kHz = 162 kHz, and so on. We can easily test this, for example by setting the signal generator frequency to 26005 Hz.

For undersampling like this to be successful the signal being sampled must be band-limited, for example by the insertion of a bandpass filter in front of the AVR. Image frequencies at $n \times f_S - f_S / 4$ are also received: in our case these images are at 8 kHz – 2 kHz = 6 kHz, 2 × 8 kHz – 2 kHz = 14 kHz, ..., 25 × 8 kHz – 2 kHz = 198 kHz, and so on.

So far we have assumed that the signal is perfectly digitised, and of course that is not the case. The sample-and-hold stage in any ADC has a so-called 'aperture', and a result of this is that it is not possible to mix down very high frequencies using the AVR. However, in the next part of this series, we will see how a signal transmitted by the BBC on 648 kHz (in the mediumwave band!) can easily be decoded.

Amplitude and phase

The signals *X* and *Y* give the strength of the in-phase and quadrature components of the input signal. However, we are rather more interested in its amplitude *A* and phase φ, since one of our ultimate aims is to decode amplitude- and phase-modulated signals. If we had fast floating-point arithmetic we could compute the amplitude and phase from the *X* and *Y* coordinates using the following two statements.

```
A = sqrt(X * X + Y * Y);
PHI = atan2(Y, X);
```

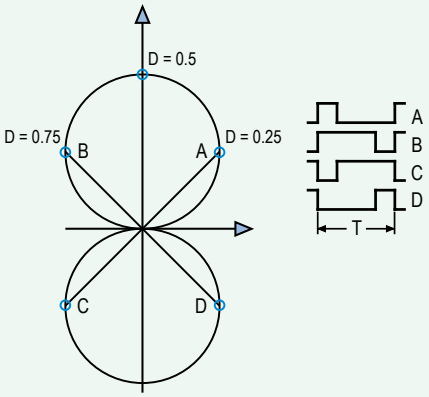
The program `EXP-SimpleFrontend-2kHz-Phase-Ampl-V01.c` calculates phase and amplitude using a rather more efficient method and outputs the results on DAC1 and DAC2. The output voltage level representing amplitude is logarithmically scaled, enabling a direct conversion to dB. **Figure 8** shows how the phase of the signal transmitted by the BBC on 198 kHz changes over time. The signal was received using an active ferrite antenna whose amplified output is fed into the receiver's front end. Antenna input ANT2 on the receiver board can be used for this purpose, with pin 1 of K4 linked to pin 2 of K5 so that the signal appears on input ADC0 of the microcontroller. Again we use a sample rate of 8 kHz. Since 198 kHz = 25 × 8 kHz – 2 kHz, the signal of interest is mixed down to 2 kHz. The BBC transmission includes digital data sent using a phase modulation of ±22.5 degrees at a rate of 25 bit/s. This digital modulation can clearly be observed in the mixed-down signal.

FM, PM and AM — with PWM

Next we want to try to generate some modulated signals ourselves. First install the program `EXP-SQTX-125kHz-PWMa-V01.c` into

BPSK and QPSK modulation schemes

A special case of phase modulation is binary phase shift keying (BPSK), where the bit values zero and one are encoded by signals 180 degrees out of phase with respect to each other. This cannot be done using the PWM method described in the text. However, we can take advantage of the settings available in the AVR's registers to select whether the event TimerValue == CompareValue results in a positive-going or negative-going edge on the output: in other words, we can selectively invert the PWM output. Introducing this possibility has the effect of adding a further circle to the diagram shown in Figure 11, as shown in the figure here.



In particular we now have four signal shapes (represented by the points A, B, C and D) that are spaced at 90 degree phase intervals. This lets us implement QPSK (quadrature phase-shift keying) modulation. If we restrict ourselves to just a pair of diametrically opposite points (for example, A and C) we have the choice of two signals with phases 180 degrees apart: this gives us BPSK modulation. Our PWM generator thus covers all the most important modulation schemes. They are all implemented in the program code and can be selected by invoking the appropriate preprocessor directive.

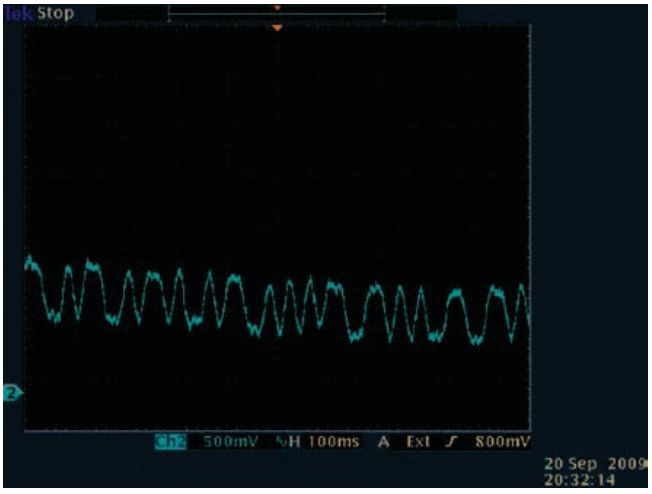


Figure 8. Phase modulation observed on a BBC 198 kHz (Droitwich) transmission.

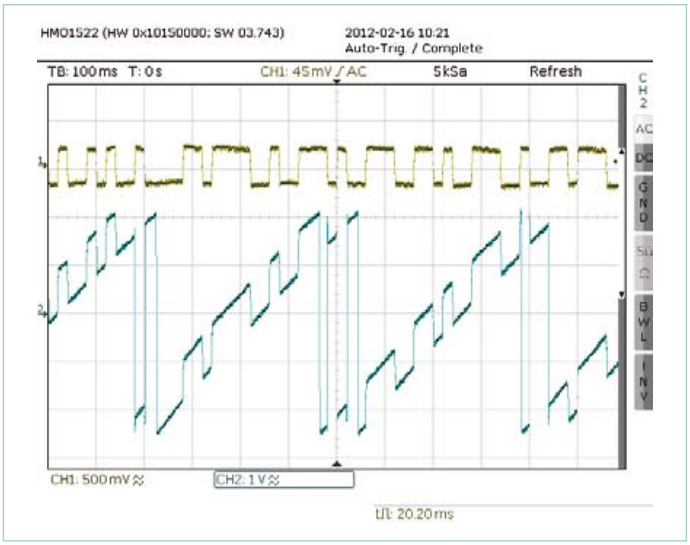


Figure 9. Pulsewidth modulation: amplitude shown in yellow, phase in blue.

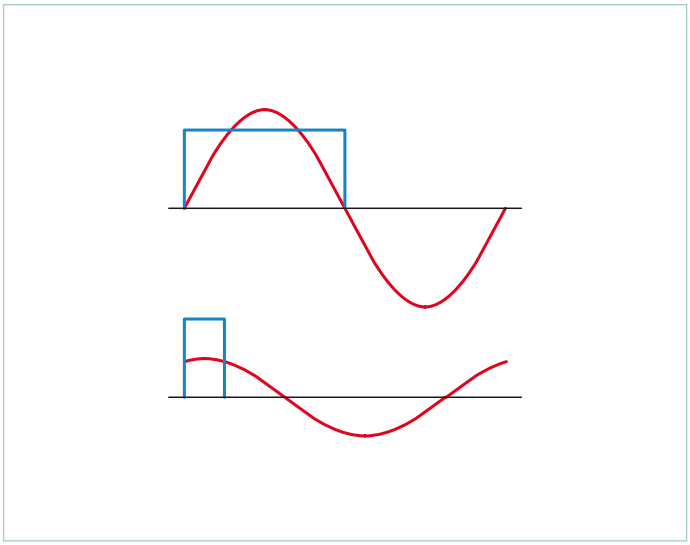


Figure 10. Pulsewidth modulation (D = 0.5 and D = 0.125).

the signal generator's microcontroller. The 125 kHz squarewave output by this code is filtered using the resonant circuit described in part 1 of this series [1] into a sinewave. Feed this signal into the ADC0 input of the receiver front end. This circuit arrangement will be used for the following experiments.

With the program `EXP-SimpleFrontend-125kHz-Phase-Ampl-V01.c` running in the front end the outputs will represent the amplitude and phase of the received 125 kHz signal. Again the amplitude output is logarithmically scaled so that a wider dynamic range can be represented: an output voltage of 4 V corresponds to an input amplitude of 1 V_{PP} and a step of 20 dB in amplitude gives a change of 1 V in the output level. A phase output voltage of 5 V represents a phase angle of exactly 360 °.

In the PWM code mentioned above a range of different modulation schemes and data sequences can be selected using #define preprocessor directives. First we will try straightforward PWM, where the frequency of a squarewave remains constant but its mark-space ratio is modulated. The bit transmission routine simply loads the modulating value into PWM register OCR1A (**Listing 2**).

The period is fixed at 160, which means that the output frequency is 20 MHz / 160 = 125 kHz. The duty cycle is switched between D = 80 / 160 = 0.5 and D = 20 / 160 = 0.125. **Figure 9** is an oscilloscope trace showing the resulting variation in amplitude and phase. It is clear that both of these quantities are modulated, the amplitude varying by 0.4 V (which, as 1 V corresponds to 20 dB, means 8 dB). The phase variation is trickier to analyse. The first point to note is that the phase jumps, in line with the data bit being transmitted, by about 0.92 V. This corresponds to 0.92 / 5 × 360 = 66 degrees. On top of this is superimposed a slow sawtooth signal with a slope (read from the oscilloscope trace) of about 5 V in 0.5 s. Now, 5 V corresponds to a phase angle of 360 degrees, and so the phase angle is making approximately two complete revolutions every second. The reason for this is again that the transmitter and receiver have very slightly different ideas of what '125 kHz' means: in fact, in this case they differ by 2 Hz. We will need to be careful to allow for this effect in future experiments. An error of 2 Hz in 125 kHz corresponds to 16 ppm, which is rather better than the typical ±50 ppm tolerance of a crystal oscillator. It is possible to compensate for this drift using a PLL control loop to drive the oscillator in the receiver.

A more refined kind of modulation

The above is all very well, but we would naturally like to implement pure amplitude modulation. And likewise, when implementing phase modulation, we would like to alter only the phase of the signal and not the amplitude. To that end we need to understand better what is happening in the PWM system.

Figure 10 shows the squarewave signal with the two different mark-space ratios D = 0.5 and D = 0.125, in each case accompanied by the sinewave signal that results after extracting just the fundamental

Listing 2: Modulation using simple PWM

```
void bitSend(uint8_t theBit){
if (theBit) {
  OCR1A  = 80 ;
  }
 else {
  OCR1A = 20 ;
  }
}
```

Listing 3: PWM modified for pure phase modulation

```
void bitSend(uint8_t theBit){
if (theBit) {
  OCR1A  = 80+10 ;
  }
 else {
  OCR1A   = 80-10 ;
  }
}
```

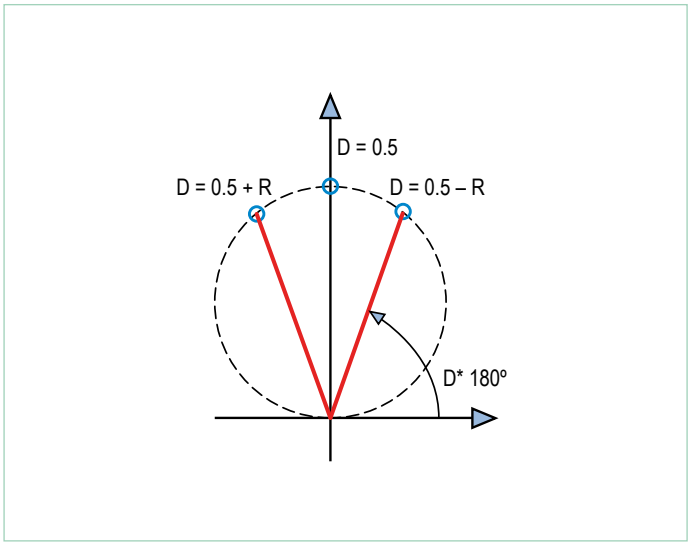


Figure 11. With judicious choice of mark-space ratios it is possible to achieve pure phase modulation.

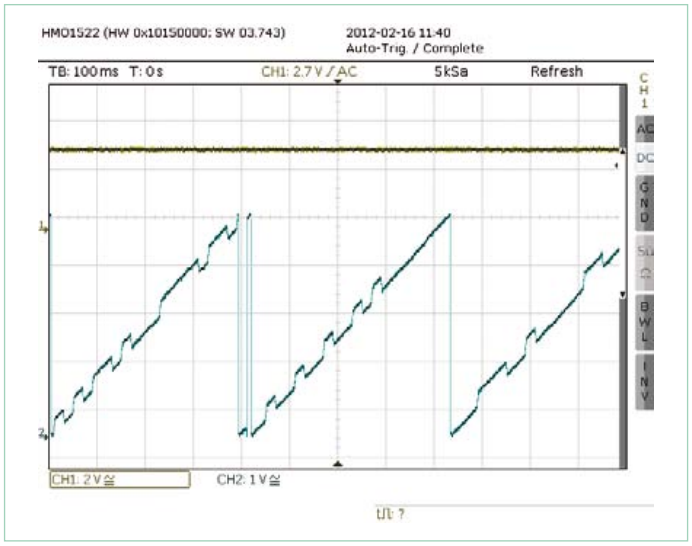


Figure 12. Pure phase modulation: amplitude shown in yellow, phase in blue.

component using the 125 kHz resonant circuit. The filtered sinewave reaches its peak exactly in the middle of the squarewave pulse. The mid-point moves when the mark-space ratio is changed, and so there is a phase shift when the mark-space ratio is changed.

In our example, when we use a mark-space ratio of $D = 0.5$ the phase angle is 0.5 × 180 degrees, or 90 degrees. When using a mark-space ratio $D = 0.125$ the corresponding phase angle is 0.125 × 180 degrees, or 22.5 degrees. The difference between these two phase angles is 90 – 22.5 = 67.5 degrees, which is in close agreement with the measured value from Figure 9 of 66 degrees.
With a little more mathematics we can also calculate the amplitude variation as a function of the mark-space ratio D. The peak-to-peak amplitude $\hat{A}$ of the sinewave is given by the formula

$$\hat{A} = 5\ \text{V} \times (4 / \pi) \sin \pi D$$

and from this we can determine that the amplitude ratio between the cases $D = 0.5$ and $D = 0.125$ should be 0.3826834… = –8.343 dB. Again, this is in good agreement with the value of 8 dB measured from Figure 9.

The relationship given above can also be illustrated graphically. **Figure 11** shows how we can determine the amplitude that we will obtain with a given mark-space ratio D. Construct a line from the origin of the plot inclined at an angle of D × 180 degrees to the point where it intersects the indicated circle. The length of this line then gives the amplitude. As can be seen, the amplitude is maximal when $D = 0.5$, and for any R, mark-space ratios of $D = 0.5 – R$, and $D = 0.5 + R$ give the same amplitudes. We can therefore use a pair of such values to create pure phase modulation without an amplitude component, and this is exactly what our PWM generator does when it is switched to PM mode using a #define directive. The essential part of this code is shown in **Listing 3**.
Figure 12 shows the result. The phase switches constantly between the two values that represent zero and one. Superimposed on this is a small drift due to the frequency error. The amplitude (yellow curve) is constant.

In the next instalment in this series we will look at how we can achieve pure amplitude modulation. However, we will not just be looking at this in theory: we will also implement a DCF test transmitter and receive DCF signals. DCF is a German time standard transmitter.

(100181)

Internet Links

[1] www.elektor.com/100180
[2] www.elektor.com/100181

Elektor Products and Support

- * Signal generator (kit including printed circuit board and all components): # 100180-71
- * Universal receiver (kit including printed circuit board and all components): # 100181-71
- * Active ferrite antenna (kit including printed circuit board and all components): # 100182-71
- * Combined kit (all three of the above): # 100182-72
- * BOB-FT232R USB-to-TTL converter, ready built and tested: # 110553-91
- * USB AVR programmer, printed circuit board with SMD parts fitted, plus all other components: # 080083-71
- * Free software download (hex files and source code)

All products and downloads are available via the web pages accompanying this article: www.elektor.com/100181

AVR Software Defined Radio part 3

AM and FM, plus an active ferrite antenna

By Martin Ossmann (Germany)

As this series shows, the popular AVR microcontroller can be used for digital signal processing tasks. In this instalment we will look at a few experiments involving amplitude and frequency modulation, including a small DCF time code test transmitter. We will also extend the hardware by adding an active ferrite antenna which will allow longwave and mediumwave signals to be received.

In the previous instalment of this series [2] we saw how a simple pulse-width modulated (PWM) signal exhibits variations in both amplitude and phase. This means that our basic signal generator cannot generate a signal that is purely amplitude modulated. However, the PWM generator in the AVR microcontroller has some extra features allowing us to switch it to 'phase correct' PWM mode. In this mode the PWM counter counts alternately upwards and downwards, between zero and a maximum value specified in register ICR1. If this limit is 80, then a complete up-and-down cycle of the counter takes 160 clock cycles: if the clock runs at 20 MHz the basic PWM frequency will be 125 kHz. Each time the counter value passes the compare value set in register OCR1A in either direction the corresponding PWM output bit is alternately set and cleared. Changing the compare value thus alters the mark-space ratio of the output signal, but the output pulse is always centred on the point where the counter hits zero and thus has constant phase. If we filter the squarewave PWM output using a resonant circuit to generate a sinewave, we can calculate the amplitude $\hat{A}$ of the result using the formula $\hat{A} = A \times (4 / \pi) \sin (D \times \pi)$, where D is the mark-space ratio of the squarewave and A its amplitude.

This takes us neatly into our first experiment, which uses the signal generator and the universal receiver board (or the 'simple front-end' described in [2]). The transmitter routine is simple in structure, as illustrated in the **Listing**. The software for the signal generator, in file `EXP-SQTX-125kHz-PWMc-V01.c`, is as usual available for download from the project web pages [3]. At the receiver end we use the program `EXP-SimpleFrontend-125kHz-Phase-Ampl-V01.c`.

If we connect the two outputs of the receiver to an oscilloscope the result will be as shown in **Figure 1**. The value in register OCR1A is being switched between 8 and 40, which corresponds to switching the mark-space ratio of the output between 0.1 and 0.5. The amplitude ratio in this case is $\sin (0.1 \times \pi) / \sin (0.5 \times \pi) = 0.309016\ldots = -10.200$ dB. Since the 'amplitude' output of the receiver has a scale of 1 V per 20 dB, the voltage difference between the two levels is about 0.5 V (see the yellow trace).

The other output of the receiver gives the phase of the received signal. As can be seen (blue trace) this is not affected by the modulation. There is, however, a gentle drift which is a result of the frequency difference between the transmitter and the receiver.

DCF77, part one

With what we have developed so far we can build ourselves a DCF time code test transmitter. The DCF77 time code transmitter is located in Mainflingen, Germany, and has a range of about 1,200 miles. The carrier frequency of 77.5 kHz unfortunately does not

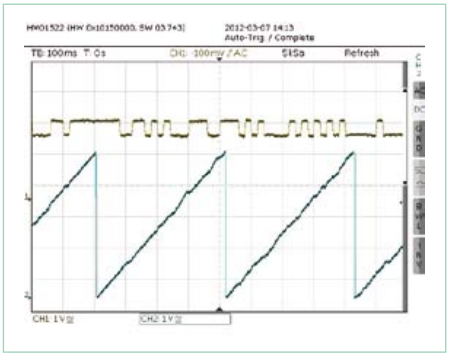


Figure 1. AM modulation: amplitude in yellow, phase in blue.

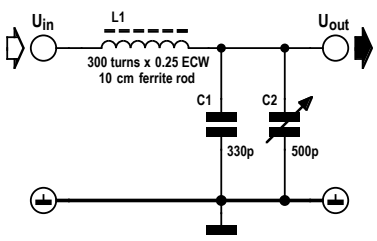


Figure 2. DCF77 transmitter tuned circuit.

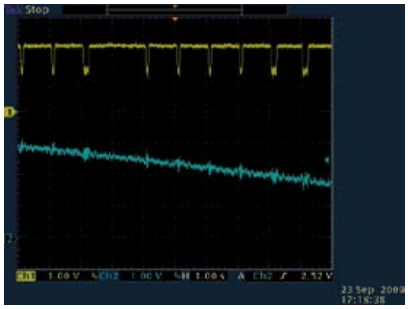
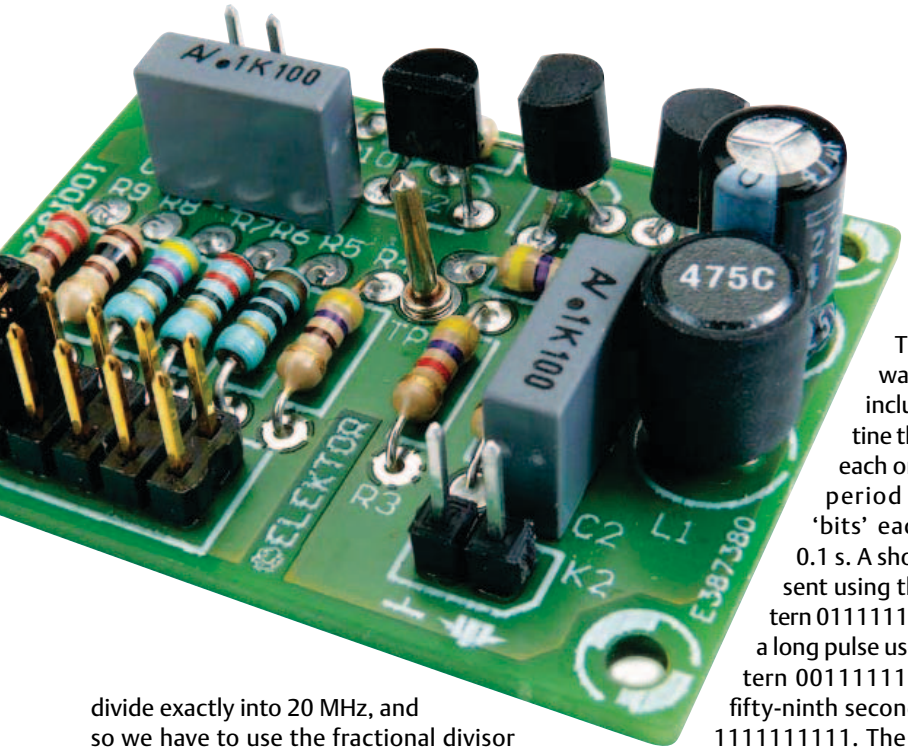


Figure 3. DCF77 reception. The short and long pulses can be seen in the yellow amplitude trace.



divide exactly into 20 MHz, and so we have to use the fractional divisor technique along with a 24-bit DDS accumulator and a timer interrupt as described in the first instalment of this series [1]. In this new code, however, we will generate 'phase correct' PWM, as we do not want any phase modulation on the output.

We feed the PWM output of the signal generator into a resonant circuit (**Figure 2**) consisting of a ferrite antenna and a suitable capacitor. An additional variable capacitor allows us to trim the circuit for maximum output amplitude.

The code running in the ATtiny microcontroller in the signal generator is `DCF_TX_V01.C`, which produces messages compatible with the DCF77 time code. Each message is composed of pulses that start at the beginning of each second, the time information being conveyed by whether the pulse is a short or a long reduction in signal amplitude. No pulse is sent in the fifty-ninth second of each minute. The software includes a routine that divides each one-second period into ten 'bits' each lasting 0.1 s. A short pulse is sent using the bit pattern 0111111111, while a long pulse uses the pattern 0011111111. In the fifty-ninth second we send 1111111111. The complete message is built by concatenating these three templates.

The program initialises the time to 11:41 on 15 August 2008. If the resonant circuit is correctly adjusted it is possible to set the time on DCF-controlled clocks within a radius of a couple of metres. Most such devices correct their time only fairly infrequently, but can usually be prodded into adjusting themselves by briefly removing the battery.

DCF77, part two

We would also like to be able to receive the real DCF time code from Germany. To do this we need the active ferrite antenna that is described later in this article, and which is available as a kit from *Elektor*. The antenna is connected to the ANT2 connection on the receiver board. On the receiver board itself we connect pin 1 of K4 to pin 2 of K5 so that the input signal is taken to the ADC0 input of the ATmega. The software used is `EXP-Simple-DCF77-RX-V01.c`.

We sample the input signal at 10 kSa/s. Since 77.5 kHz is exactly 8 × 10 kHz – 10 kHz / 4, we can do the demodulation using the bandpass sub-Nyquist sampling technique described in the previous instalment of this series. The oscilloscope traces in **Figure 3** show the results. The upper (yellow) trace shows the amplitude, with the short periods when the amplitude is reduced clearly visible. It is also possible to see that both long and short reductions in amplitude are present. Extracting the time information is now just a short step away.

We can also make use of the phase component of the DCF77 signal. In one of our later experiments we will clock the receiver using a voltage-controlled 20 MHz crystal oscillator (VCXO) rather than a fixed-frequency oscillator. If we adjust the frequency of the oscillator so that the phase no longer drifts, the 20 MHz signal will be

Listing: Phase correct PWM

```
void bitSend(uint8_t theBit){
if (theBit)
        {
                OCR1A = 40;
        }
        else
        {
                OCR1A = 8;
        } // 10dB
}
```

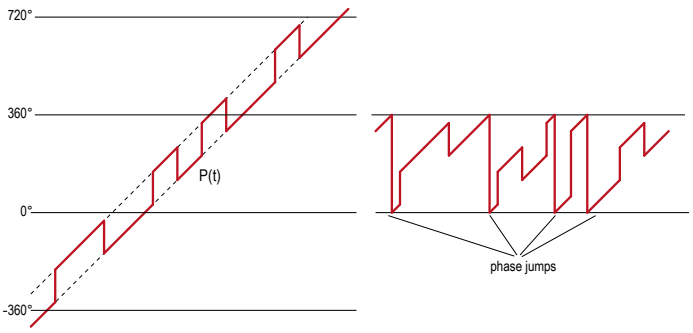


Figure 4. The same phase behaviour shown in two different ways.

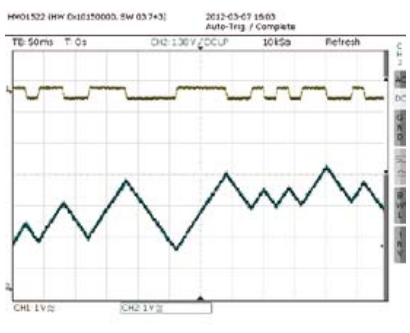


Figure 5. Frequency shift keying (FSK).

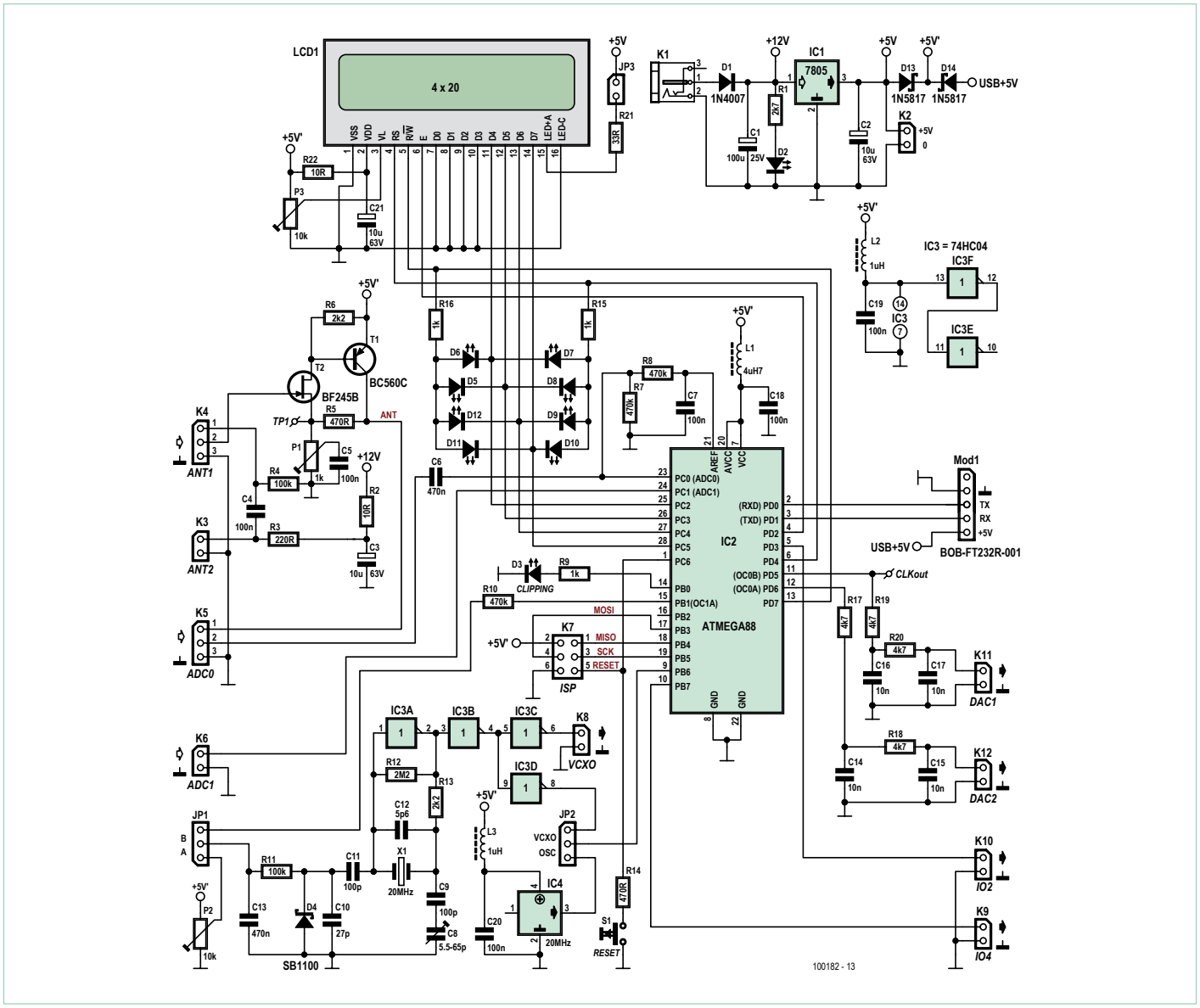


Figure 6. Circuit diagram of the AVR-SDR universal receiver board.

locked to the very precise carrier frequency of the DCF77 transmitter. In Figure 3 it is possible to see the phase changing slowly: by using a phase-locked loop it is possible to automate the adjustment, as we shall see later in this course. It is also possible to lock the receiver to other sources, such as BBC Droitwich transmissions on 198 kHz or France Inter on 162 kHz, both of which also provide very precise frequency references.

Understanding the phase changes

Figure 1 shows the by now familiar sawtooth pattern in the phase angle that results from a frequency offset between transmitter and receiver. The phase changes smoothly, wrapping round between

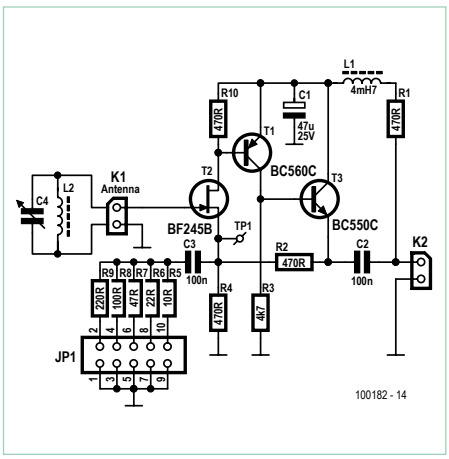


Figure 7. Circuit diagram of the active ferrite antenna.

360 degrees = 5 V and 0 degrees = 0 V. The wrap-around appears sudden on the oscilloscope trace, but the underlying physical behaviour is continuous.

We can often get a clearer picture if, rather than restricting the angle to lie between 0 degrees and 360 degrees, we allow it to go below 0 or beyond 360. In **Figure 4** on the left we can see a representation in this form of a phase-modulated signal with a frequency offset superimposed. The curve is rather easier to interpret than when the phase angle is restricted.

A couple of analogies may help to explain what is happening. First imagine walking in a circle around the north pole: at a certain point, which has no particular significance on the ground, your longitude

will jump between 180 degrees west and 180 degrees east. Now imagine not walking in a circle, but climbing a spiral staircase. After one complete revolution you are not in exactly the same place: you are one floor higher. If, along with the phase angle, we keep track of which 'floor' we are on, we can represent phase differences of more than 360 degrees. This is a technique used in constructing PLLs that have a wide capture range.

Figure 5 shows a nice application of this technique. We use our signal generator as an FSK (frequency shift keying) transmitter, using the software in `EXP-SQTX-FM-RTTY-V01.c`. The output of the signal generator on K4 is taken via a resonant circuit acting as a filter (see the first part of this series [1]) to input ADC0 on the receiver. The code running in the receiver is `EXP-SimpleFrontend-125kHz-extPhase-Freq-V01.c`, which has a phase output scaled so that it can represent a wider range of phases. The scaling is such that 5 V represents 8 × 360 degrees. The 125 kHz carrier generated by the transmitter is shifted by +/– 50 Hz to represent the bits 1 and 0, and the data rate is 50 bits per second. A shift of +/– 50 Hz means that in one bit time, 1/50 s, the transmitted signal will gain or lose one period relative to the reference signal. Each bit thus corresponds to a phase shift of 360 degrees with a direction that depends on the value of the bit being sent. In turn, a phase change of 360 degrees gives a change in the output voltage of 5 V / 8 = 0.625 V, and this occurs over a period of 20 ms. The blue trace in Figure 5 shows this effect clearly.

Demodulating the FSK signal is easy: the instantaneous rate of change of phase corresponds to the current frequency shift and hence to the transmitted bit. The rate of change of phase can be calculated by taking the difference between consecutive phase values: the result is shown in the yellow trace. When the phase angle is increasing the yellow trace is 'high'; when it is decreasing, the trace is 'low': from this is it easy to read off the bits being sent. A software UART can be added to make a complete software defined FSK receiver. In the next instalment of this series we will see how a couple of extra filters can be added to make the receiver more robust.

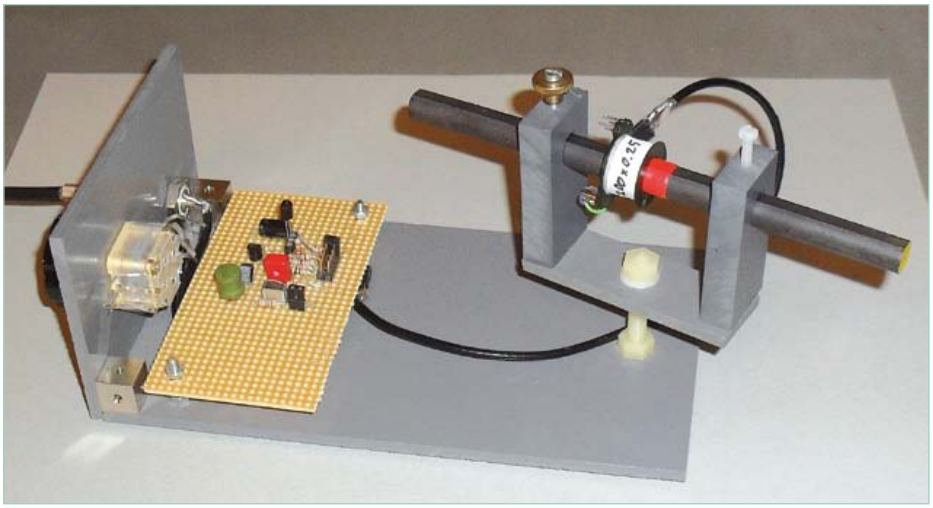


Figure 8. The author's prototype of the active ferrite antenna.

The universal receiver board

Now that we have carried out a few experiments with the simple receiver circuit, it is time to move on to a more advanced receiver board. The universal receiver board was described, including a printed circuit board layout, in the previous instalment of this series. **Figure 6** shows the circuit diagram again to help explain some of the interesting possibilities that it opens up. A four-line LCD panel is provided as a display. Header Mod1 allows a BOB-FT232R USB-to-TTL converter to be added: this lets you communicate between the board and a PC, for example to log received data.

A discrete 20 MHz oscillator is provided as a clock source. The frequency of this oscillator can be adjusted over a narrow range using a control voltage. This voltage is derived either from potentiometer P2 or from the AVR microcontroller itself via PWM output OC1A/PB1 and a lowpass filter comprising R10 and C13. This latter option allows the VCXO to form part of a phase-locked control loop, for example to derive a precision frequency reference from the DCF77 signal. A divided-down version of the clock frequency can be output on pin OC0B. Alternatively, an integrated quartz crystal oscillator module (IC4) can be selected to provide the master clock using jumper JP2.

Ports C and D are used to drive eight LEDs arranged in a circle that can be used as a phase display. These provide a simple means of determining when the PLL is in lock, and give a clear indication when a small frequency offset is present.

Analogue signals are presented to the microcontroller on the ADC0 input to its analogue-to-digital converter. R7 and R8 provide a DC offset voltage on this input equal to half the converter's reference voltage AREF, while C6 provides AC coupling for the input. T1 and T2 form a preamplifier to whose input (K4 pins 2 and 3) a resonant receiver circuit consisting of a ferrite antenna and a tuning capacitor can be directly connected. The output of the preamplifier can be fed to the ADC input by connecting together pins 1 and 2 of K5. Another possibility is to connect a ferrite antenna with phantom power to the preamplifier input: in this case pins 1 and 2 of K4 should be connected together and the ferrite antenna connected to K3.

For some of the experiments we have seen so far (and for some in the future) we have generated two outputs from the receiver and visualised them using an oscilloscope. These outputs are generated using PWM based on Timer 0 and are available on pins OC0A and OC0B. Each of these is equipped with a two-stage RC filter. The resulting voltages are available on K11 and K12.

Active ferrite antenna

To complete the picture we equip our receiver with an active ferrite antenna for the longwave and mediumwave bands. **Figure 7** shows the circuit diagram. Thanks to JFET T2 the input has a very high impedance, and so the tuned circuit forming the antenna has a high Q-factor and selectivity. T1 provides a useful amount of gain and emitter follower T3 gives a low-impedance output. Resistor R2 gives negative feedback for DC and AC, the latter being configurable

COMPONENT LIST Active ferrite antenna

Resistors
(all 5%, 0.25W)
R1,R2,R4,R10 = 470Ω
R3 = 4.7kΩ
R5 = 10Ω
R6 = 22Ω
R7 = 47Ω
R8 = 100Ω
R9 = 220Ω

Capacitors
C1 = 47µF 25V, 20%, radial, 2.5mm, I_{AC} 95mA
C2,C3 = 100nF 63V, 5%, MKT, 5mm or 75mm
C4 = 2x265pF + 2x20pF dual gang quad section tuning capacitor (e.g. [4])

Inductors
L1 = 4.7mH, 81mA, 132Ω, radial, 3mm
L2 = ferrite rod, L = 90mm, D = 10mm (e.g. [4])
3 pcs coil former, 10mm, 5-pin
24.5m (82 ft.) enamelled copper wire, 0.22mm diameter (#31 AWG)

Semiconductors
T1 = BC560C
T2 = BF245B (JFET)
T3 = BC550C

Miscellaneous
K1,K2 = 2-pin pinheader, lead pitch 0.1 in. (2.54mm)
JP1 = 10-pin pinheader, lead pitch 0.1 in. (2.54mm)
TP1 = PCB solder pin, 1.3mm diam.
PCB # 100182-1 [3]

Alternatively: kit, comprising PCB and all parts: # 100182-71 [3]
Alternatively: Combined kit; 3 kits plus BOB FT232 USB/TTL-converter: # 100182-72 [3]

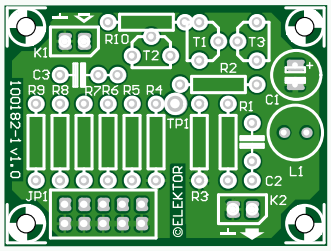


Figure 9. The printed circuit board for the active ferrite antenna is available from Elektor as part of a kit, along with all the components.

Table: Ferrite antenna and tuning capacitor details

AK Modul-Bus dual-gang tuning capacitor 2 × 265 pF, C_{min} = 50.00 pF, C_{max} = 500.00 pF		
AK Modul-Bus ferrite antenna, 90 mm long, AL = 100.00 nH / n^2 (experimentally determined value, depends on the coil geometry and other factors)		
Wind 50, 150 and 200 turns on each of three coil formers to allow total turns counts of 50, 200 and 400.		
Turns	**Inductance**	**Frequency range**
n = 50	L = 0.250 mH	450.2 kHz to 1423.5 kHz
n = 200	L = 4.000 mH	112.5 kHz to 355.9 kHz
n = 400	L = 16.000 mH	56.3 kHz to 177.9 kHz

using JP1. The antenna receives phantom power at approximately 12 V.
Part-to-part variation in T2 can affect the function of the circuit. It is therefore necessary to select T2 so that the voltage at its source is around 2 V. It is also necessary to ensure that the input connections are kept away from the output connections as otherwise feedback can cause the circuit to oscillate, and it is best to use screened cables. **Figure 8** shows the author's prototype.
As in the case of the signal generator and the universal receiver board a complete kit for the active ferrite antenna is available from *Elektor* including all components and a printed circuit board (**Figure 9**). The kit includes a ferrite rod, three coil formers and 24.5 m of enamelled copper wire.

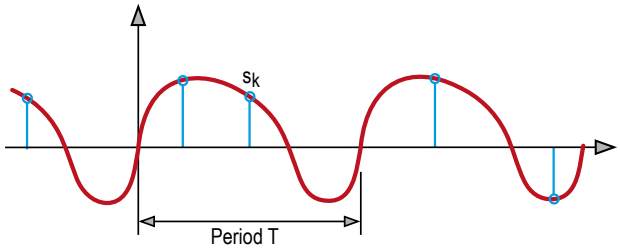


Figure 10. Random sampling.

RMS voltmeter with random sampling

At various points in this course it is useful to be able to measure the RMS value of an alternating voltage. For example, one application is in adjusting a ferrite antenna to a given frequency with the help of the signal generator. As it happens, we can turn our receiver board into an RMS voltmeter!
In principle, to calculate the RMS value S_{RMS} of a periodic signal $s(t)$ we need to take a sufficiently large number N of samples s_k over a period and then compute the 'quadratic mean':

$$S_{RMS} = \sqrt{(1/N \Sigma s_k^2)}$$

Now, we would like to work with signals whose frequencies are as high as 1 MHz, and the ATmega88 cannot sample its input sufficiently fast (the limit is about 10 kSa/s at 10-bit precision).
Instead of taking many samples over a single period, however, we can take readings at random points in time over many periods (see **Figure 10**), in a technique called 'random sampling'. The disadvantage of this approach is that a relatively large number of samples is required to get a sufficiently accurate result. On the other hand, it has the advantage of working equally well with non-periodic noise-like signals.
The ATmega88 random sampling RMS voltmeter code is contained in the file `EXP-RMSmeter-V01.c`. The program automatically switches the A/D converter's reference voltage between 5 V and 1.1 V as needed to obtain the best possible accuracy. The quadratic mean is calculated over a total of 2048 samples, and simultaneously displayed on the receiver board's LCD panel and output on its serial interface. The display is then updated after every 256 new samples. As we describe in the 'Aperture time' text box, this RMS voltmeter is an entirely practical piece of equipment.

Aperture time

The ATmega88 requires a total of 13 ADC clock periods to carry out a conversion, and the maximum ADC clock frequency is 200 kHz. In our case we generate the ADC clock by dividing the 20 MHz CPU clock by 128, which gives a frequency of 156.25 kHz and a conversion rate of about 12 000 conversions per second. According to the sampling theorem this lets us digitise any input signals that do not contain frequency components above 6 kHz. Despite this, the random sampling technique lets us measure the RMS value of signals with considerably higher frequency components. The limiting factor in this is the quality of the sample-and-hold circuit in the microcontroller immediately before the converter itself. In particular, the time period over which the input is sampled (called the 'aperture time') must be as short as possible. Unfortunately the ATmega88 datasheet does not give precise information on this point, and so we need to do some experiments to find out the maximum frequency for which we can get a respectable level of accuracy in our measurement of RMS value.

No sooner said than done: the author set up a test with a 100 mV_{RMS} sinewave signal fed simultaneously into the AVR RMS meter and a Tektronix digital oscilloscope. The table gives the amplitude readings shown for frequencies of up to 2 MHz. Up to 200 kHz our device gives good accuracy; at 500 kHz the error is around 10 %; and at 1 MHz the error is around 30 %.

It seems from these results that it is possible to sample frequencies of up to a few hundred kilohertz reasonably accurately. If we are prepared to allow for the 30 % (about 3 dB) attenuation, then we can work with frequencies of up to 1 MHz. The conclusions are twofold: first, our RMS voltmeter is not bad at all; and second, the aperture time of the AVR microcontroller is rather short, which will come in handy later when we look at digitising signals in the longwave and mediumwave bands using sub-Nyquist sampling.

Frequency	AVR display (mV)	Oscilloscope (mV)
1 kHz	99.0	100.0
2 kHz	100.0	100.0
5 kHz	101.9	101.5
10 kHz	102.0	102.0
20 kHz	102	102.5
50 kHz	102	102.3
100 kHz	101	102.2
200 kHz	98.0	101.7
500 kHz	90.0	101.0
1 MHz	68.0	100.9
2 MHz	42.0	99.0

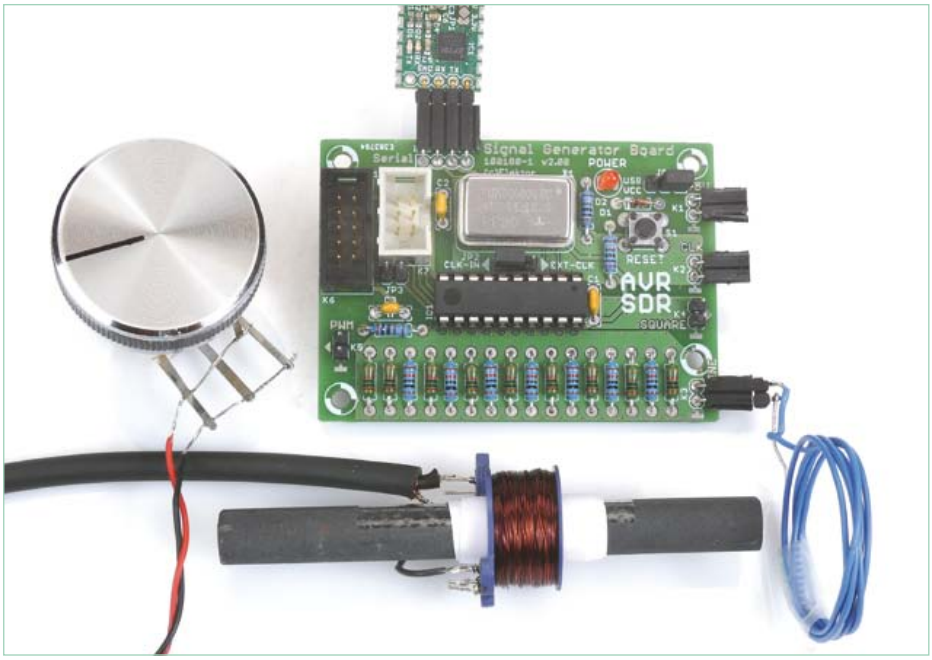


Figure 11. Set-up for adjusting the ferrite antenna tuned circuit.

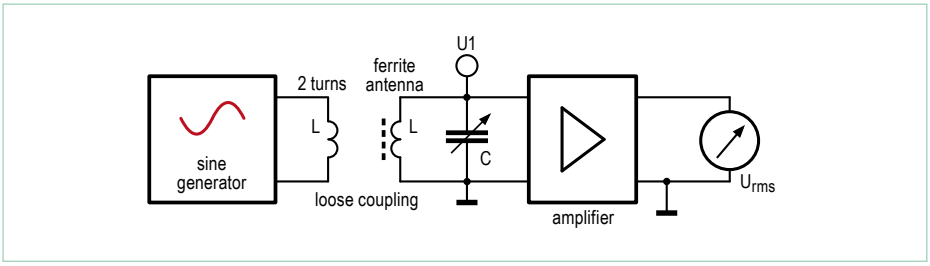


Figure 12. Schematic arrangement of components for adjusting the ferrite antenna tuned circuit.

The ferrite antenna and its adjustment

The carrier frequencies used by transmitters that we will want to receive lie between 50 kHz and 700 kHz. In this range a ferrite antenna is the best choice. We use a ferrite rod 90 mm long and with a diameter of 10 mm (available, for example, from AK Modul-Bus [4], and included in the kit). An RM10 coil former can be used to wind suitable coils. To cover the above frequency range using a dual-gang 2 × 265 pF tuning capacitor we can use three separate coils, with 200, 150 and 50 windings (see **Table**). When hunting for a signal it is necessary both to tune the antenna and adjust its orientation. It makes life easier if the tuning part has been done accurately in advance, and we now have at our disposal all the equipment we need to do this. The signal generator is set up to produce sinewaves (`EXP-SinusGenerator-DDS-ASM-C-V01.c`), which we feed into a small 30 mm diameter coil wound with just a few turns of wire (**Figure 11**), making a 'magnetic' test transmitter.

It is best to set up the resonant circuit in the receiver in exactly the configuration in which it will subsequently be used. For example, connecting an oscilloscope to the circuit will shift its centre frequency significantly.

First connect the active antenna to the receiver board and run the RMS voltmeter 'receiver' software `EXP-RMSmeter-V01.c`. **Figure 12** shows the arrangement schematically. Now set the test transmitter to the desired frequency as described in the first part of this series [1]. Bring the transmitter coil up fairly close to the ferrite rod. Adjust the tuning capacitor until the receiver is at resonance with its output level at a maximum. To tune more precisely it may be necessary to move the transmitter coil further away from the ferrite antenna and adjust again. With the tuned circuit set to the correct frequency it is a lot easier to find the desired station.

In the next instalment we will continue with a look at filters and how to use a PLL to generate a high-precision frequency reference, and we will see how to receive marine weather information on 147.3 kHz.

(100182)

Internet Links

[1] www.elektor.com/100180
[2] www.elektor.com/100181
[3] www.elektor.com/100182
[4] www.ak-modul-bus.de (site in German only)

Elektor products and support

- Signal generator (kit including printed circuit board and all components): # 100180-71
- Universal receiver (kit including printed circuit board and all components): # 100181-71
- Active ferrite antenna (kit including printed circuit board and all components): # 100182-71
- Combined kit (all three of the above plus BOB-FT232R USB-to-TTL converter): # 100182-72
- BOB-FT232R USB-to-TTL converter, ready built and tested: 110553-91
- USB AVR programmer, printed circuit board with SMDs fitted, plus all other components: 080083-71
- Free software download (hex files and source code)

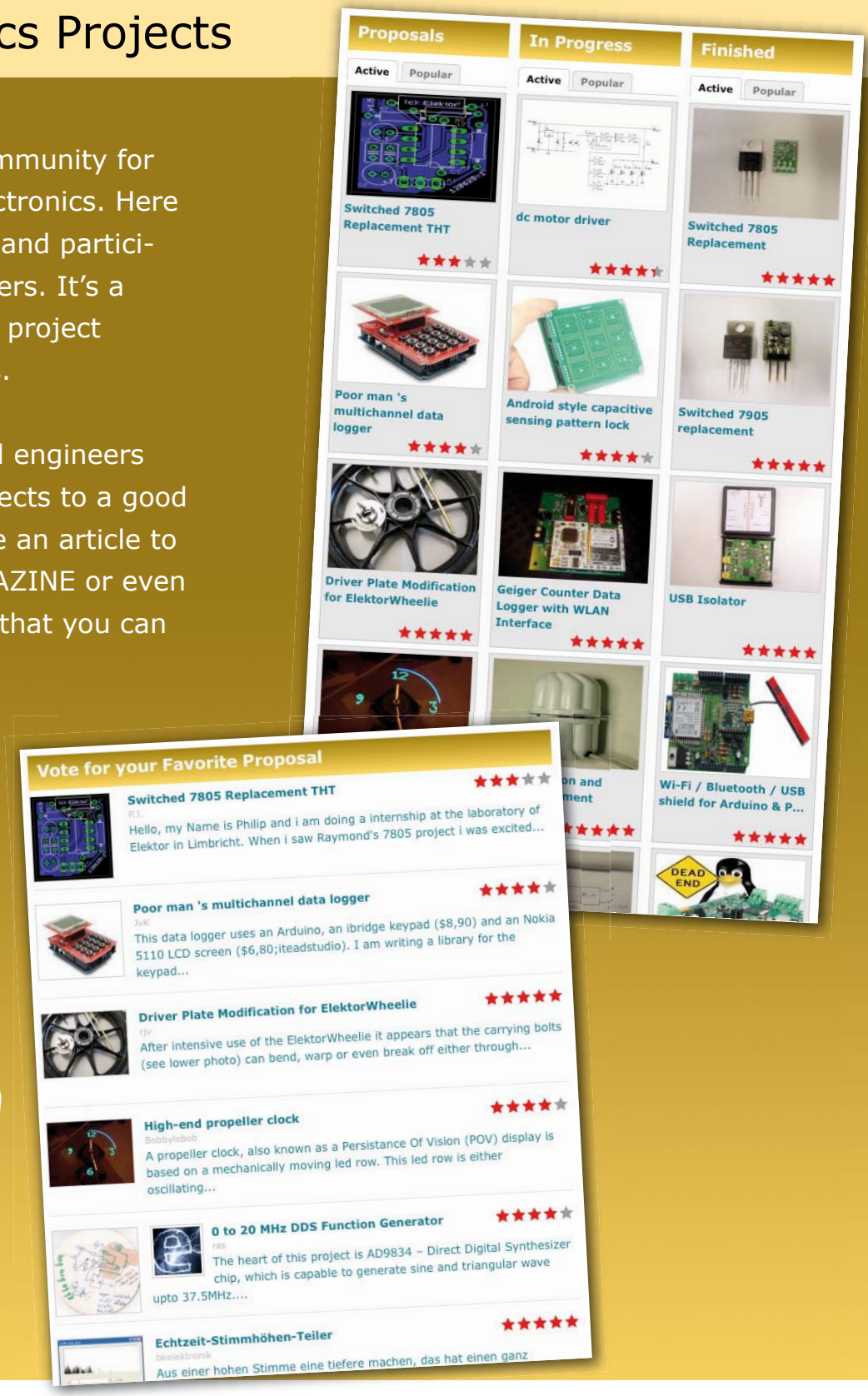
All products and downloads are available via the web page accompanying this article: www.elektor.com/100182

elektor labs

Sharing Electronics Projects

Elektor.LABS is an online community for people passionate about electronics. Here you can share your projects and participate in those created by others. It's a place where you can discuss project development and electronics.

Elektor's team of editors and engineers assist you to bring your projects to a good end. They can help you write an article to be published in Elektor.MAGAZINE or even develop a complete product that you can sell in Elektor.STORE!



GET ELEKTORIZED

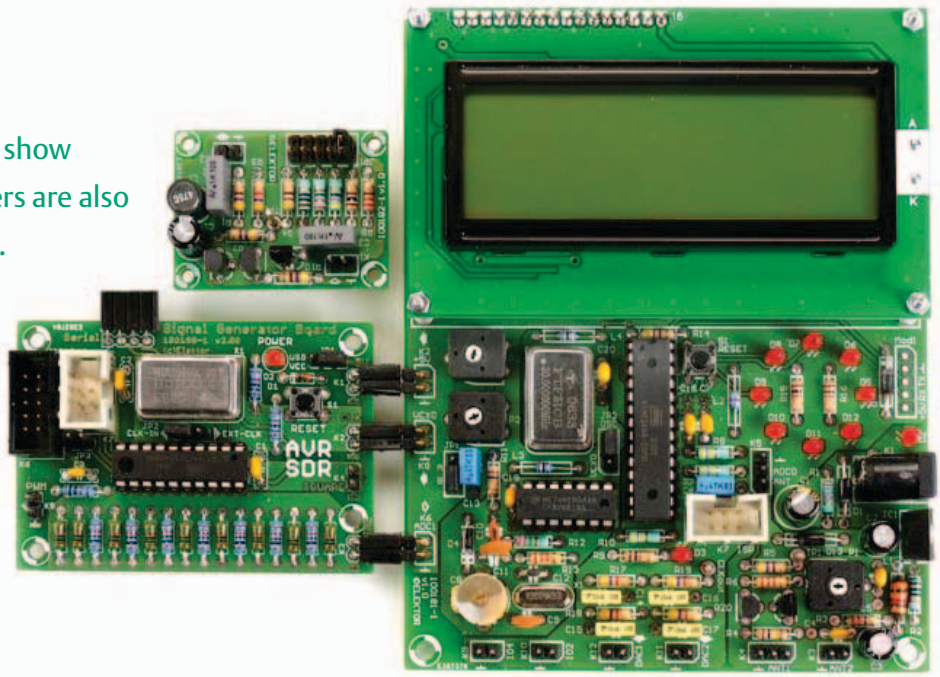
Join or Create a Project at www.elektor-labs.com

AVR Software Defined Radio

Part 4: Digital radio reception: DCF77, weather service and more

By Martin Ossmann (Germany)

The aim of this series of articles is to show that the popular AVR microcontrollers are also suitable for digital signal processing. This time we turn our attention to something practical: using our DIY SDR to receive signals from DCF77 and other time reference transmitters, as well as the German weather service. We also decode the time and weather signals.



In the previous instalment [3] we described a test transmitter and an active ferrite antenna, which we can put to good use in this instalment. Now we dedicate our efforts to expanding the scope of our digital receiver. We have already described the digital I/Q mixer. Now it's time for the filters downstream of the mixer. In order to implement narrow-band reception, we need low-pass filters after the mixer, and more specifically the same filters for the in-phase and quadrature components. The filters located directly after the mixer operate at the same sampling rate as the front end — e.g. 10 kHz for DCF77. The filters need to be simple to allow the AVR to compute them in real time. In the front end we use what are called CIC filters for this purpose. Now it's time for a description of how these simple filters work.

Moving-average and CIC filters

CIC filters are based on the moving-average principle. **Figure 1** shows a filter that calculates the moving average of a set of five samples.

A series of delay stages stores the past samples, which are summed for each average value. When this filter is implemented as an algorithm, the samples are stored in a ring buffer. Each time a new sample is added, it replaces the oldest sample and the buffer pointer is incremented. All samples in the ring buffer are added together each time to calculate the output value (four addition operations). This takes $N - 1$ addition operations for a filter with N samples. A large N requires a lot of computation time.

It can be shown that the filter depicted in **Figure 2** does the same thing as the one in Figure 1. Linear time-invariant filters are fully characterised by their impulse response. The impulse response is the out-

Elektor products and support

- Signal Generator kit with PCB and all components: # 100180-71
- Universal Receiver kit with PCB and all components: # 100181-71
- Active Ferrite Antenna kit with PCB and all components: # 100182-71
- Combo kit with all three component kits plus BOB FT232 USB to TTL converter: # 100182-72
- BOB FT232 USB to TTL converter, fully assembled and tested: # 110553-91
- USB AVR programmer; PCB pre-assembled with SMD components plus all other components: # 080083-71
- Free software download (hex files and source code)

All products and downloads are available on the web page for this article: www.elektor.com/120088

put signal resulting from a single '1' at the input (in the series 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, ...). The series 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, ... is the impulse response of a moving-average filter, due to the fact that the single '1' passes through the five summing nodes.

Now let's look at what happens to a '1' that enters the filter shown in Figure 2. This filter consists of two parts: a delay line with five stages, followed by an integrator (accumulator).

The integrator simply sums all incoming values and outputs the sum. When a single '1' enters the filter, it passes through the delay line. However, it also reaches the integrator via the direct path. The '1' remains stored in the integrator, which outputs a '1' until the '1' from the delay line also reaches the integrator. The latter is multiplied by –1, and after five steps the resulting '–1' neutralises the '1' that arrived previously, with the result that the output from the integrator consists of a string of zeros. This means that the impulse response of this filter is exactly the same string of five ones.

The first section of the filter (the delay line followed by the subtractor) is called a comb filter because its frequency response looks like a comb. This means that our filter consists of a comb filter followed by an integrator. This type of filter needs only two addition operations per time increment, regardless of the number of delay stages (*N*). This implementation considerably simplifies matters for microcontrollers. Here again, the delay line is implemented as a ring buffer.

As the sequence of several linear time-invariant filters may be swapped, the filter in **Figure 3** also has the same effect, but in this case the integrator is located before the comb filter.

Now let's work out whether this filter does in fact have the right impulse response. After a single '1' enters the filter, the integrator constantly outputs a '1'. The question now is how this filter responds when it receives only a series of '1' values. At some point the integrator must overflow, but if you use the right floating-point arithmetic in the implementation, the filter works properly despite this. This behaviour is essential for the usability of this filter version.

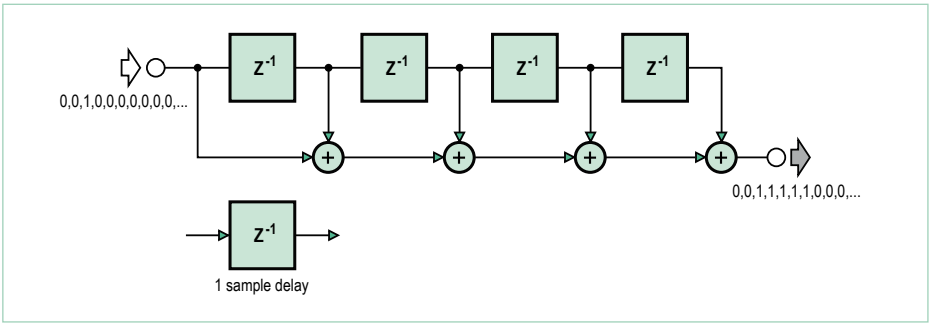


Figure 1. A moving-average filter.

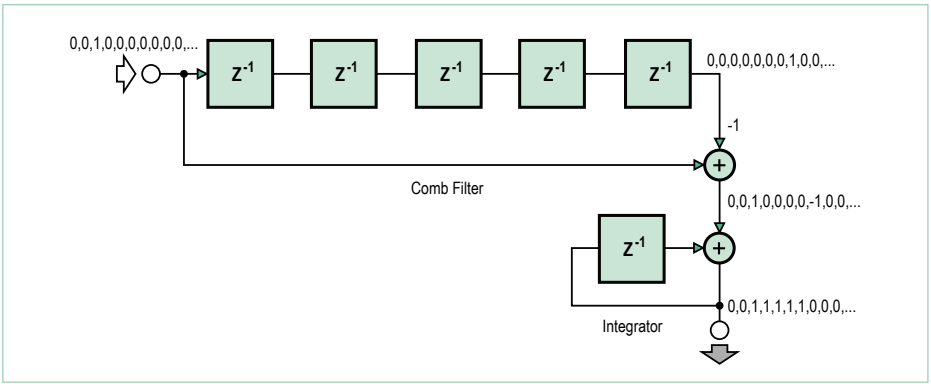


Figure 2. A five-stage delay line and integrator, which is equivalent to the circuit in Figure 1.

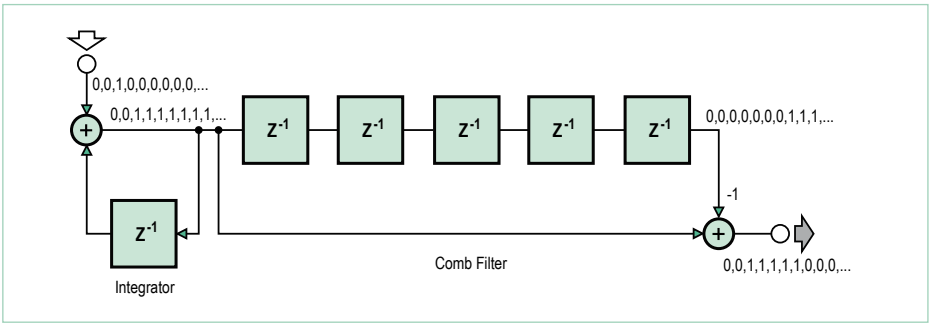


Figure 3. Filter sections in reverse order compared with Figure 2, but with the same effect.

Filters and downsampling

Due to the high sampling rate, the front end does not have much time for complicated filter algorithms. If the signal to be processed is sufficiently narrow-band, the signal can be downsampled (decimated) by a factor *R* after pre-filtering. This means that only every *R*th sample is kept and processed. If you want to use a moving-average filter, you can use the especially efficient CIC structure shown in **Figure 4**.

Here integration is still performed at the

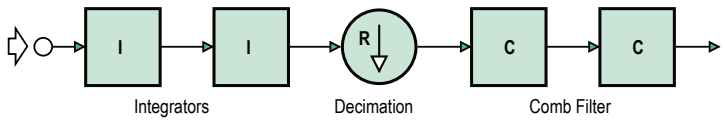


Figure 4. The CIC architecture is especially efficient.

Listing 1: In-phase channel

```
if (sampleTime & 1) {
if (sampleTime & 0b10) {
IIintegrator1 += ADCv ;
}
else {
IIintegrator1 -= ADCv ;
}
IIintegrator2 += IIintegrator1 ;
}

sampleTime++ ;
if (sampleTime==Md) {
sampleTime=0 ;
IntFifoII[IntFifoInPtr]=IIintegrator2 ;
IntFifoInPtr=(IntFifoInPtr+1) & IntFifoMask ;
}
```

Listing 2: FIFO filter

```
IIsample=IntFifoII[IntFifoOutPtr] ;
IntFifoOutPtr=(IntFifoOutPtr+1) & IntFifoMask ;
// do 1.st comb-step of two stage downsampling CIC filter
IIcomb1out=IIcomb1store-IIsample ; IIcomb1store=IIsample ;
// do 2.nd comb-step of two stage downampling CIC filter
IIcomb2out=IIcomb2store-IIcomb1out ; IIcomb2store=IIcomb1out
;
```

Listing 3: Third filter stage

```
// integration step of smoothing CIC filter I-channel
IIintegrator3 += IIcomb2out / P ;
// comb step of smoothing CIC filter
IIcic3out = IIintegrator3 - IIfifo[CICfifoPTR] ;
// FIFO update of smoothing CIC filter
IIfifo[CICfifoPTR] =  IIintegrator3 ;
// advance pointer and bump around
CICfifoPTR++ ; if ( CICfifoPTR==Mc ) { CICfifoPTR=0 ; }
```

fast data rate. The comb filter, which has a length of *R*, 2*R* or *NR* here, does not need to operate at the fast data rate, but only at the date rate reduced by the factor *R* after decimation. This significantly reduces the load on the fast interrupt routine that has to process the ADC values. Here the routine for the in-phase channel takes the form shown in **Listing 1**, where *Md* is the downsampling factor. The front end stuffs the samples into a FIFO buffer, from which they are fetched by the following stage. If the following stage is momentarily too slow to process the samples immediately, one or several samples remain in the buffer until they are fetched. This technique makes it easier to program the subsequent stages. **Listing 2** shows the finished code for this.
The result after double filtering is held in the variable *IIcomb2out*. As you can see, the CIC filter needs only a few addition operations. As the bandwith after the two cascaded filters is still not sufficiently narrow, a third filter is added. This filter does not decimate the signal, but instead leaves the sampling rate unchanged. The associated code shown in **Listing 3**. Here again, only a few operations are necessary, and the number of operations does not depend on the length of the filter.

This concludes our discussion of the filtering of the I and Q signals. **Figure 5** shows the resulting architecture of the front end with the filters. Timer 1 generates the ADC sampling rate clock *Fs*, which has a frequency of (20 MHz)/*Ns*. Downsampling of the input signal with frequency *f* yields a signal with frequency f_{IF}. This frequency must be exactly one-quarter of the sampling frequency, so that it can be mixed with the signal from the local oscillator to obtain the baseband signals. Each of the resulting signals is filtered by a triple low-pass filter. The first two filters, which are also used for decimation, are of order *Md*. The final filter is of order *Mc*. The limit frequencies of the filters decrease with increasing values of *Mc* and *Md*. The X and Y signals are available at the outputs of the filters for further processing.

Receiving frequencies

In order to apply the concept described above, the quotient of *f*/*Fs* must differ from

an integer value by exactly 0.25. As you can see from **Table 1**, this requirement can be fulfilled for a large number of receiving frequencies.

If necessary, this requirement is only needs to be approximately fulfilled. In such case the signal is not mixed down to absolute baseband, but instead to a very low frequency.

CORDIC concept for phase and amplitude

The aim of our first experiment is to receive the DCF77 signal in the same was as described in the previous instalment [3]. For this we use the software EXP-Simple-DCF77-RX-IQ-V01 [4]. DCF77 is a time standard transmitter in Mainflingen, Germany, and operating at 77.5 kHz. It controls millions of clocks in Europe.

The input signal, decomposed into the I and Q components, is available after low-pass filtering. However, in may cases what you need is the amplitude, phase or frequency. This corresponds to a conversion from rectangular coordinates to polar coordinates. With floating-point numbers the phase can be determined using the atan2 function, but floating-point computations take a lot of processing power. For this reason, we propose a method that manages with pure integer computations.

Let's start by looking at about **Figure 6**. Here the objective is to determine the phase of point *P* with coordinates *X0* and *Y0*. The basic idea is to rotate the position of the point in the clockwise direction until the point is located on the positive X axis. This is done in a manner similar to that of an ADC that uses the successive approximation method. First we try a 180° rotation. If the *Y1* coordinate of the resulting point is negative, we have rotated too far. If *Y1* is positive (as in the example shown here), we next rotate by 90°, then by 45°, and so on. We keep a running total of all the rotation angles. If we rotate too far in one of these steps, we reverse the rotation. At the end the point is located on the X axis within a specific margin of error, and the total rotation angle corresponds to the original phase.

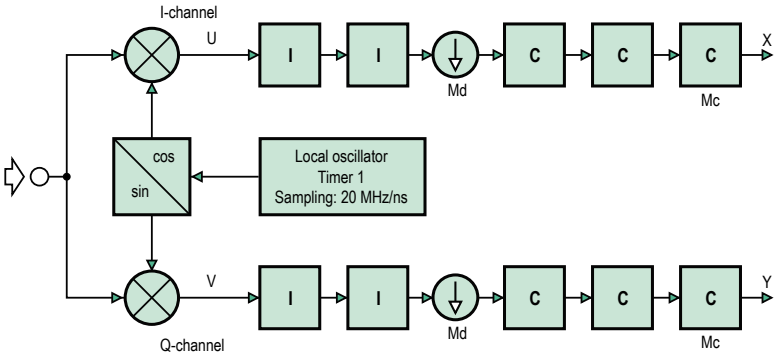


Figure 5. Block diagram of the front end with filters.

Table 1: Receiving frequencies and parameters

Receiving frequency f	Divisor Ns	Fs (20 MHz/Ns)	f/Fs	Name
60.0 kHz	2250	8.888... kHz	6.75	MSF
75.0 kHz	1800	11.111... kHz	6.25	HBG
77.5 kHz	2000	10.0 kHz	7.75	DCF77
125.0 kHz	1800	11.111... kHz	11.25	Test
162.0 kHz	2500	8.0 kHz	20.25	TDF
198.0 kHz	2500	8.0 kHz	24.75	BBC
648.0 kHz	1875	10.666... kHz	60.75	BBC

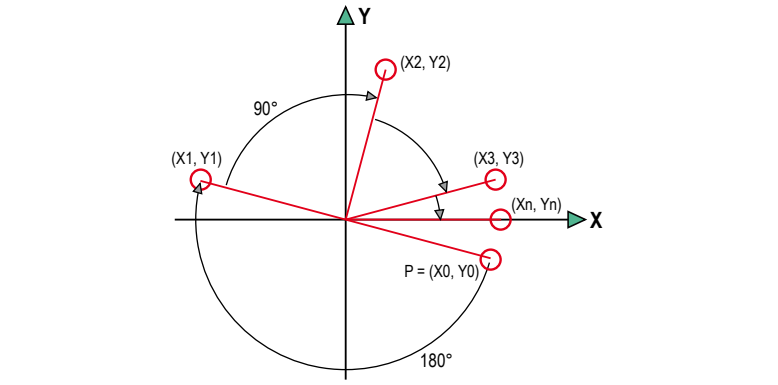


Figure 6. Operating principle of CORDIC rotation.

Listing 4: CORDIC rotation

```
c=iCordicCosTab[k_cordic] ;
s=iCordicSinTab[k_cordic] ;
// rotate (x0,y0) by phi into (x1,y1)
x1=( c*x0+s*y0) / 65536L ;
y1=(-s*x0+c*y0) / 65536L ;
```

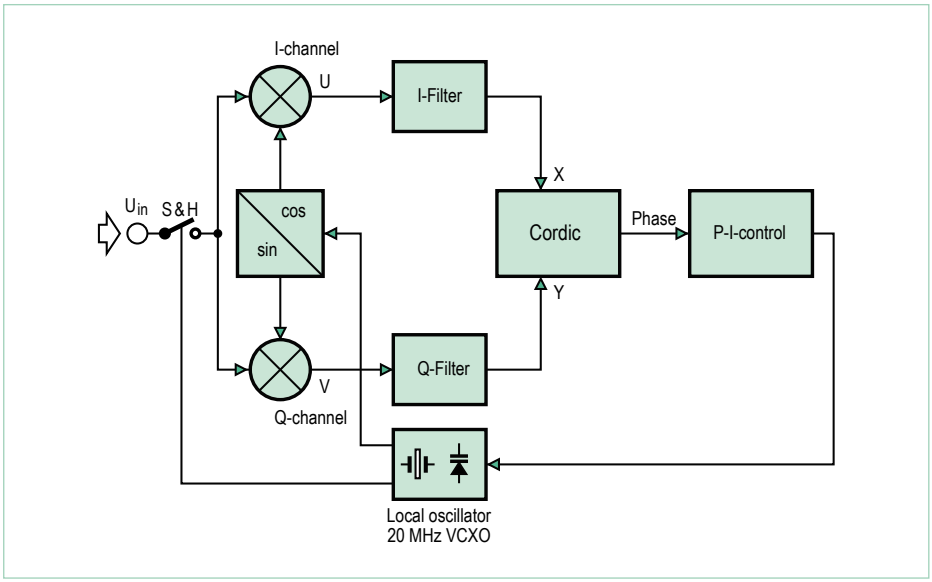


Figure 7. PLL block diagram.

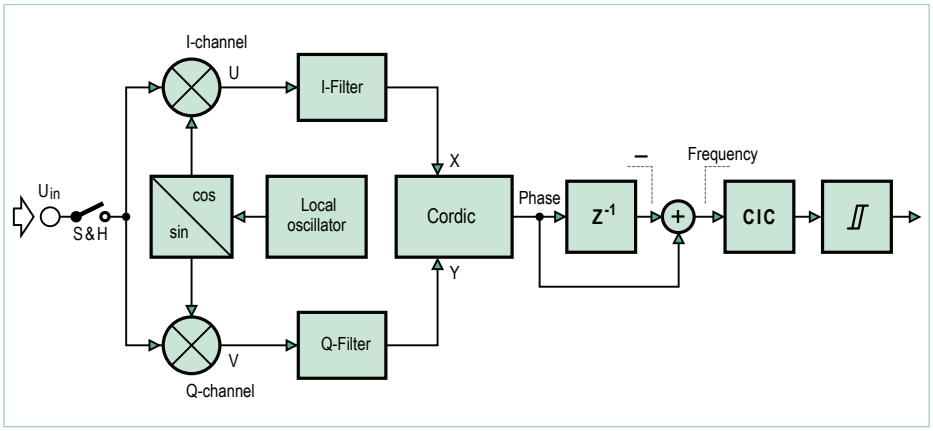


Figure 9. Block diagram of RTTY decoding.

For the implementation, the coefficients of the rotation matrices are multiplied by a factor of 65,536 and stored in a table. For an angle of 22.5 (for example), this gives:
65,536 × cos(22.5°) = 60.547
65,536 × sin(22.5°) = 25,080

The routine for computing the rotation of a point at (*X0*,*Y0*) to a new point at (*X1*,*Y1*) is shown in **Listing 4**.
This method is very similar to an algorithm called the CORDIC algorithm [6] [7]. After the point has been rotated to the X axis, its X coordinate corresponds to the amplitude, which makes it very easy to determine the amplitude. Our CORDIC routine delivers the angle scaled such that a full rotation (360°) corresponds to exactly 256. This means that range of CORDIC values for a phase range of 0 to 360° is the same as the range of values of a single byte.

High frequency accuracy with a PLL

For this we use the software EXP-Simple-DCF77-RX-V01 [4] to display the amplitude and phase.

If you examine the signal received from the DCF77 time reference transmitter, you will see that its phase drifts slowly. The reason for this is that the frequency of our crystal oscillator is not exactly 20 MHz. You can try to calibrate the oscillator by adjusting the trimmer to minimise the phase drift, but this sort of adjustment will naturally not remain constant for a long time. A better option is to put together a phase-locked loop (PLL). This involves applying the phase signal to a control amplifier and feeding the amplifier output to the control input of the VCXO, as shown in **Figure 8**. If everything is correctly dimensioned and aligned, the PLL will lock and the signal from the crystal oscillator will be held tightly in phase with the received signal. You can use this to build your own frequency standard by tuning the radio to receive a signal broadcast as a frequency standard.

References: DCF77, France Inter and BBC

The receiver can be adapted to various transmitters by simply altering a few parameter values. This only requires activating the corresponding options with '#define' statements in the source code. First let's try receiving the DCF77 signal. This signal can be received everywhere in continental Europe. In the western part of Europe, signals from BBC Droitwich (at 198 kHz) and France Inter (at 162 kHz) can also be received. The same approach can be used in other countries with suitable local transmitters.

Putting the PLL into operation

The receiver must be suitably aligned in advance to enable it to lock onto the received signal. Use the following procedure for this purpose. First adjust the active ferrite antenna. To do this, use the signal generator to generate a sinewave signal at the desired frequency. Weakly couple this signal into the antenna by connecting the output signal from the generator through a 1-pF capacitor to the hot end of the tuned circuit. Then adjust the rotary capacitor to maximise the signal level at the output of the active antenna. Of course, the active antenna must be connected to the receiver during this process, to provide it with a source of power. You can use the previously described RMS voltmeter to measure the signal amplitude.

Next, load the program 'EXP-VCXO-PLL-V01', compiled for the right frequency, into the ATmega88. Put jumper JP1 in position A so that the PLL can be adjusted with trimpot P2. Start by setting P2 to its midpoint. If you now orient the receiving antenna, the lit LED in the LED circle should rotate more or less quickly. Use the trimmer capacitor C8 to adjust the VCXO frequency until the LED point remains stationary or rotates only very slowly. If you now adjust P1, the direc-

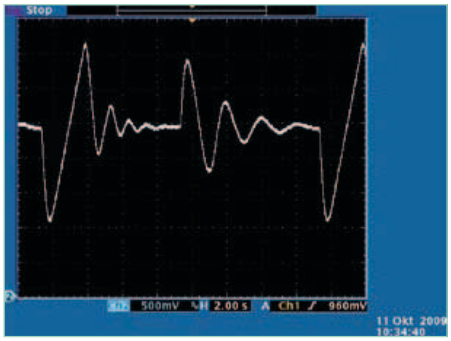


Figure 8. PLL phase response.

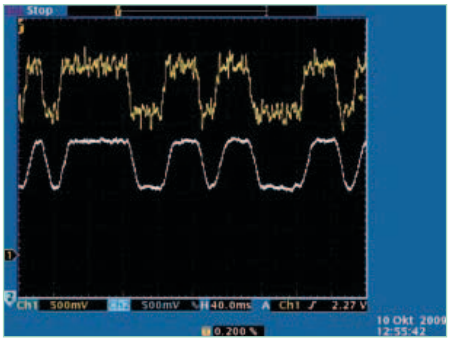


Figure 10. Frequency signal before filtering (yellow) and after filtering (blue).

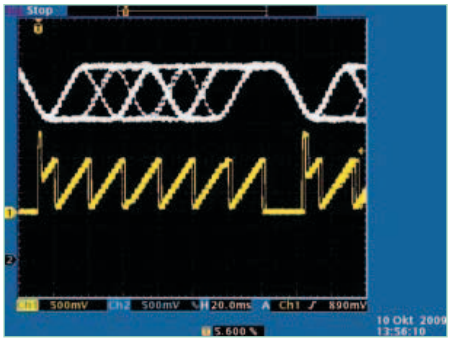


Figure 11. Eye pattern (blue) and clock signal (yellow).

tion and speed of rotation of the LED point should change according to the rotation of P1. If this does not happen, you can try using a different 20 MHz crystal oscillator or slightly modifying the values of C9 and C12. The above procedure ensures that the VCXO can be adjusted over the right range by the 0–5 V control voltage. Now place jumper JP1 in position B to close the control loop. After the circuit is reset, the lit LED should wander back and forth a bit before settling down at a single point. When this happens, the PLL is locked and the 20 MHz oscillator is generating exactly 20 MHz. Incidentally, the two PWM AC outputs of the receiver still deliver the amplitude and phase of the received signal, which can be used as tuning aids. When receiving the DCF77 signal, you can recognise the dip in the amplitude that occurs at a 1-second rate. With the signals from the TDF162 (France Inter) and BBC189 transmitters, you can see the phase modulation.

To check the behaviour of the PLL, we modified the signal generator program to cause it to generate a 125-kHz signal with a step change of plus or minus 1 Hz every couple of seconds. In the oscillograph in **Figure 8** you can see the PLL phase response. The decaying oscillation after each step change shows the PLL locking onto the new frequency.

DDH47: radio teletype at 147.3 kHz

The German Weather Service transmits marine weather reports on 147.3 kHz as radio teletype messages. The messages are transmitted with frequency shift keying (FSK), which produces frequency modulation. After you determine the phase with the CORDIC method, it's relatively easy to determine the frequency by measuring the difference between successive phase values, since the frequency corresponds to the rate of change of the phase. However, this frequency signal is very noisy if the received signal is not very clean. This can be remedied relatively easily by filtering with a CIC low-pass filter. The output signal from the filter is applied to a Schmitt trigger, resulting in the demodulated radio teletype (RTTY) signal.

The block diagram of this arrangement is shown in **Figure 9**. In **Figure 10**, the top trace is the frequency signal immediately after the difference operation, and the bottom trace is the filtered signal — which is a good deal cleaner.

The output signal from the Schmitt trigger is fed to a software UART. The UART performs start-bit detection, serial to parallel conversion, and conversion from Baudot to ASCII code.

Now we tune the ferrite antenna to 147.3 kHz (see [3]) and download the software EXP-DDH47-RTTY-RX-V01 to the SDR microcontroller. The RTTY signal and the bit clock appear at the DAC outputs, and the received signal strength is shown on the LCD.

Figure 11 shows the frequency signal as an eye pattern. Here the oscilloscope trace is triggered by the start bit detection signal, and several curves are shown superimposed. If they eyes in the pattern are free from noise and wide open, signal reception is good.

The received characters are output over the serial port of the ATmega88 and can be viewed in a terminal emulator program running on the PC. A short excerpt of the received data is shown in **Listing 5**.

This series will continue in the next issue with a test transmitter and more details on the decoding of various transmitter signals.

(120088-I)

Web links:

[1] www.elektor.com/100180
[2] www.elektor.com/100181
[3] www.elektor.com/100182
[4] www.elektor.com/120088
[5] www.ak-modul-bus.de
[6] www.dspguru.com/dsp/faqs/cordic
[7] http://en.wikipedia.org/wiki/CORDIC

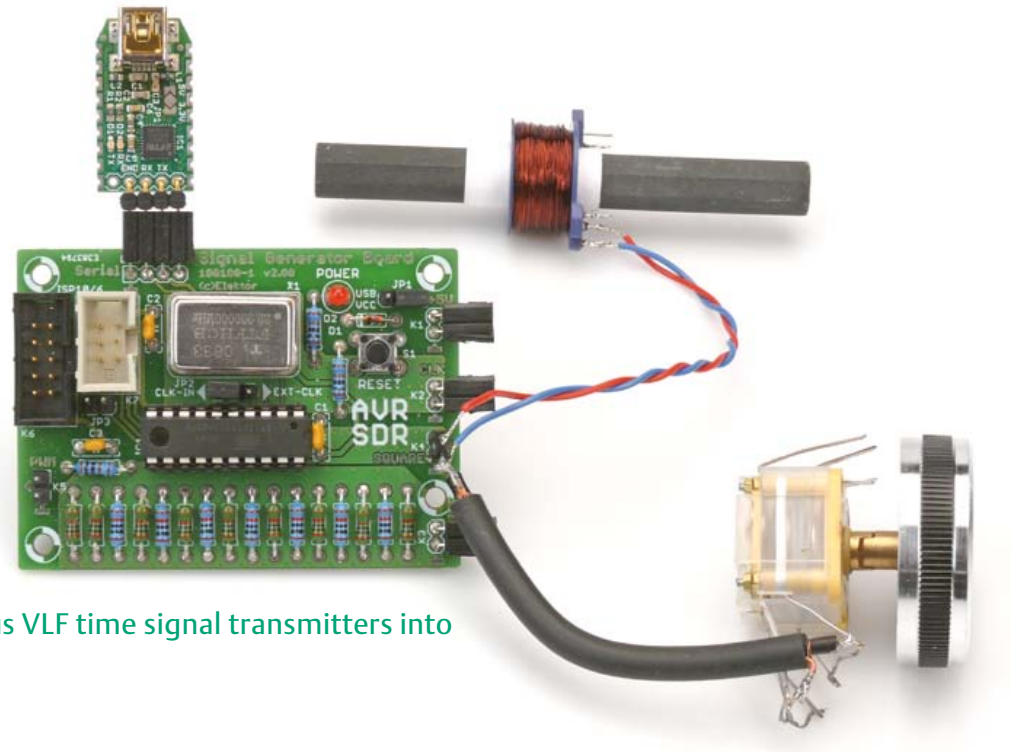
Listing 5: Message received from the German Weather Service

```
CQ CQ CQ DE DDH47 DDH9 DDH8
FREQUENCIES   147.3 KHZ   11039 KHZ   14467.3 KHZ
RYRYRYRYRYRYRYRYRYRYRYRYRYRYRYRYRYRYRYRYRYRYRYRYRYRYRYRYRYRYRY
ZCZC 483
FQEW57 EDZW 100600
SEEWETTERBERICHT FUER WESTEUROPAEISCHE GEWAESSER
HERAUSGEGEBEN VOM SEEWETTERDIENST HAMBURG
10.10.2009, 06 UTC:
WETTERLAGE:
HOCH 1025 ZENTRALE OSTSEE UND OSTPOLEN OSTWANDERND,
```

AVR Software Defined Radio (5)

Part 5: Decoding DCF77, MSF and TDF162 using IIR and matched filters

By Martin Ossmann (Germany)



The aim of this series of articles is to show that the popular AVR microcontrollers are also suitable for digital signal processing. In this instalment we use a variety of decoding methods and filters to convert the signals from various VLF time signal transmitters into digital data.

In the previous instalment [4] we devoted our attention to the reception of digital radio signals. Using the digital receiver described in that instalment, we received signals from time signal transmitters such as German DCF77 and from weather services and decoded the received signals. Now it's time to examine a variety of decoding methods. Along with transmitting and receiving RTTY signals, we discuss the reception and decoding of signals from DCF77 in Germany, MSF in the UK and TDF162 in France. Finally, we describe how to use a matched filter for bit decoding. Sadly the author can't even dream of ever receiving WWVB at his location, but the information supplied in this article should enable Elektor USA readers to develop suitable decoding and demodulation algorithms. WWVB transmits at 60 kHz.

Wireless data transmission at 125 kHz

If the level of the DDH signal at your receiver location (see the previous instalment in this series) is too weak or absent altogether, you can build your own test transmitter. The hardware is exactly the same as for the DCF test transmitter described in the third instalment [3]. A simple ferrite antenna and a variable capacitor form a series-resonant circuit that is fed from the square-wave output of the signal generator. This arrangement is shown in the photo at start of the article.

The software for the transmitter is contained in the *EXP-SQTX-FM-RTTY-V01.c* file. After configuring the reception software in *EXP-125kHz-RTTY-RX-V01.c* for operation at 125 kHz, you can start your signal transmission and reception experiments. The author obtained a range of over 5 m (15 ft) (even through walls) with this simple system. Among other things, this arrangement could form the basis for a wireless remote control system.

With the same software, you can also decode conventional RTTY ('telex') signals in the amateur radio bands. All that is necessary for this is to adjust the filters and the timing to match the transmission parameters. For testing, you can generate RTTY signals with suitable PC software such as MMtty (download from [6]) and a sound card.

Decoding DCF77

An oscillogram of the received and demodulated DCF77 signal was shown in Figure 3 of the third instalment of this series. Now let's see how you can recover the data from the keying pulses. As the amplitude of the received signal can vary widely depending on reception conditions, the first thing you need to do is to define a threshold level (as automatically as possible) for deciding whether the instantaneous amplitude is high or low. For the sake of simplicity, you could set the threshold level at half the average signal level. However, this requires determining the long-term average value. If we wanted to use a CIC filter to average the signal, we would need a very large interim buffer. This is why we use an IIR filter [7] instead.

IIR filtering

The abbreviation IIR stands for 'infinite impulse response'. This long impulse

response (which corresponds to the averaging interval) is obtained by using a recursive filter. The computation formula for a simple IIR filter is:

$y_{k+1} = a\,x_k + (1 - a)\,y_k$

Here y_k is the output sample series, x_k is the input sample series, and a is a number less than 1 but close to 1. The new sample y_{k+1} is the averaged sum of the previous sample and the new input sample.
In this formula the factors a and $(1 - a)$ are normally numbers less than 1. This makes it tempting to use floating-point operations to compute the filter, but that would be much to complicated. In our implementation (see **Listing 1**) the filter coefficients are represented as fractions, with the denominator (65,536) being a power of 2.
Here a = (65536 – 50) / 65536 = 0.999237 ..., and $(1 - a)$ is accordingly 0.000763...
The computations are performed using 32-bit integer numbers. The compiler coverts division by 65,526 into simply selecting the most significant 16 bits, which considerably speeds up the computation. The result is a highly efficient IIR filter. The output of the IIR filter is the signal in binary form, and the time of day can be obtained by evaluating the pulse lengths.
In central Europe and the south-east of the UK, DCF77 reception with decoding requires the simple front-end or the universal receiver, along with the active ferrite antenna tuned to 77.5 kHz. With the *EXP-DCFdecode-RX-V01.c* software loaded in the receiver microcontroller, you obtain a seconds timer and the signal is output by the PWM DAC. The signal strength and time of day are also shown on the LCD module, and the decoded data is output on the RS232 or USB port.

Decoding MSF

There are several time signal and standard frequency transmitters in Europe. One of them is MSF [8] in Rugby, Great Britain, which broadcasts at 60 kHz (coinciding with WWVB in Fort Collins, Colorado, USA). Its signal is relatively weak at the author's place of residence, so a bit more effort is necessary. The amplitude of the MSF carrier wave is keyed at 1-second intervals (see **Figure 1**). The carrier is keyed to a low level for 0.5 s at the start of each minute. At the start of each subsequent second, the carrier is initially keyed low for 0.1 s. The next 0.5 s is the time slot for the A bit. The A bit is high if the carrier amplitude is low in this time slot. The following 0.1 s is the time slot for the B bit. The B bit is also high if the carrier amplitude is low in this interval. The A and B bits are used to transmit time data, additional synchronisation data and error detection codes.

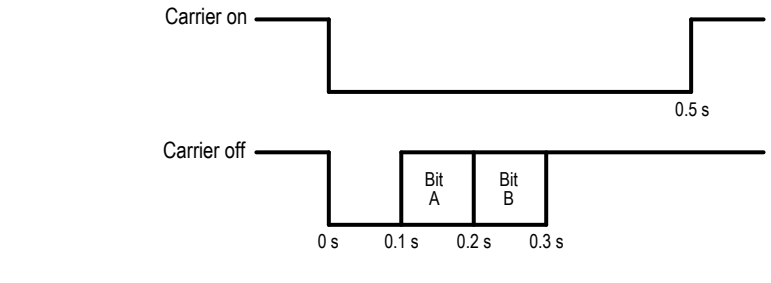


Figure 1. Carrier keying of the Rugby MSF signal.

Listing 1: IIR filter

```
#define ALPHA 50
#define BETA  65536
int32_t Threshold ;

  Threshold=(ALPHA*(int32_t)Amplitude+(BETA-ALPHA)*Threshold)/BETA ;
  if (Amplitude>Threshold) {
    DCFlevel=0 ;
    }
   else {
    DCFlevel=1 ;
    }
```

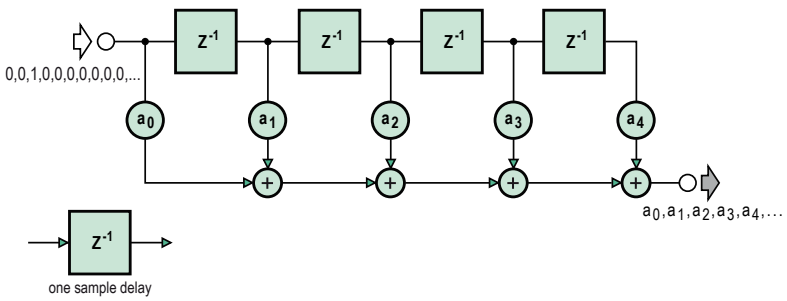


Figure 2. Signal paths in an FIR filter.

Our first attempts to receive the signal yielded a severely distorted signal, which could not be adequately filtered by a simple CIC filter. We therefore needed a better filter with a cutoff frequency at around 10 kHz and fairly strong attenuation. Another requirement for the filter was that it should not affect the waveform too much. These characteristics can be achieved with a FIR filter [9] of sufficient order.

For MSF reception you need the receiver, the active antenna tuned to 60 kHz, and the software *EXP-RX-MSF60decode-V01.c*. With this arrangement, the signal strength and time of day are shown on the LCD module and the date and time are output on the serial port. The structure of the fifth-order FIR filter is shown in **Figure 2**. The input

Listing 2: FIR summation

```
#define FIRLength 60

prog_int16_t FIRCoeffs []={
   112,     3,   -17,
   -52, ............. 112 }  ;

  FIRSum=0 ;
  j=FIRPointer ;
  for (k=0 ; k<FIRLength ; k++ ) {
    FIRcoef=pgm_read_word( FIRCoeffs1 + k ) ;
    FIRSum += (int32_t)FIRcoef*(int32_t)FIRBuffer[j] ;
    j++ ; if (j==FIRLength) { j=0 ; }
    }
  return FIRSum/0x10000L ;
```

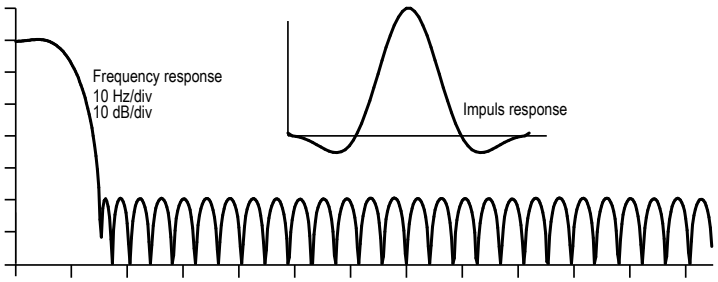


Figure 3. Impulse response and frequency response of the FIR filter.

Figure 4. Demodulated and filtered MSF signal and blue timing trace.

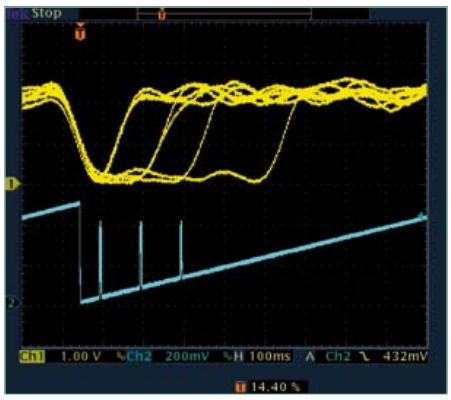


Figure 5. The three spikes show where the signal is evaluated.

samples pass through a buffer, and the output sample is the sum of the samples in the buffer weighted by the coefficients *k*.
All of this is implemented in the C code shown in **Listing 2**. Here again we use fast integer arithmetic followed by division. This routine filters the signal, which is sampled at a rate of 250 samples per second. The computation time (300 µs) is sufficiently short for a filter order up to 60, since 4 ms is available for each sample in this case.
Figure 3 shows the impulse response and frequency response of the filter. The initial cutoff frequency is approximately 8 kHz, and the attenuation above 15 kHz is around 50 dB.

Figure 4 shows the demodulated and filtered MSF signal (yellow trace). The minutes pulse with a duration of 0.5 s can be seen near the middle of the image. A pulse with A = 1 and B = 0 can be seen two seconds before the minutes pulse, and a pulse with A = 0 and B = 1 can be seen after the minutes pulse. This is followed by several pulses with A = 0 and B = 0. A trace with a sawtooth waveform is visible below the yellow trace. This signal can be used to keep track of the timing.

The waveforms are shown in more detail in **Figure 5**. The software has a variable called *SecondTimer*, which recurrently counts from 0 to 250 at a rate of 250 Hz. This clock rate is the same as the sampling rate of the demodulated signal. *SecondTimer* is reset when the minutes pulse is detected, so it always starts counting from the start of a second. The *SecondTimer* count state is output by one of the PWM DACs. The three blue spikes show where the software evaluates the received signal. This is where the values of the A and B bits are determined. After all the bits have been collected, they are evaluated and the time data is output to the LCD module and on the RS232 port. **Listing 3** shows an example of the output data.

Decoding France Inter (TDF)

France Inter (TDF) also broadcasts time data using a phase-modulated carrier. The signal waveforms for the low and high states are shown in **Figure 6**. The data in

Listing 3: Example data from the MSF transmitter

```
<200000000000000000010000000110010000010001110001001011111103>  07:09 08.03.10 MONDAY    P=1 P=1 P=1 P=1
<200000000000000000010000000110010000010001110010000011113103>  07:10 08.03.10 MONDAY    P=1 P=1 P=1 P=1
<200000000000000000010000000110010000010001110010001011111103>  07:11 08.03.10 MONDAY    P=1 P=1 P=1 P=1
```

this signal is coded in the same way as the DCF77 signal. However, we encounter a new problem with the decoding. The time code bit is transmitted at the start of each second, but this is followed by a lot of phase-modulated data that is used for internal purposes. This means that the receiver must first determine the exact position of the seconds pulse. There is a long modulation gap before the seconds pulse for the null second, since the pulse for the 59th second is mission. Once the receiver has detected this gap, it looks for the next peak in the phase waveform (see Figure 6). This peak marks the exact position of the desired seconds pulse. At exactly one-second intervals from this point onward, the receiver checks whether the received signal indicates a 1 or a 0.

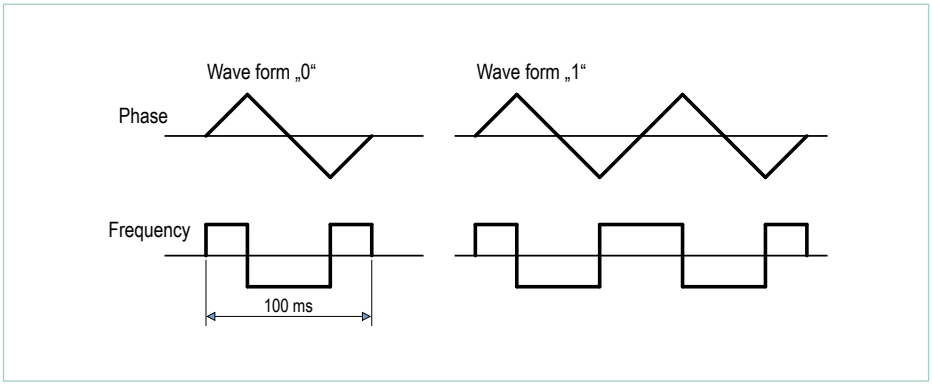


Figure 6. TDF162 signal waveforms.

A matched filter with CIC

If you want to evaluate the phase directly, you don't want to have any phase drift. Consequently, you need to use a PLL to stabilise the phase of the signal. However, this is a bit complicated and control loops are not always stable, so a different approach is better. Instead of evaluating the phase, we evaluate the frequency. The frequency can be obtained from the phase by differentiating successive samples and filtering the resulting samples with a CIC low-pass filter to clean up the waveform. To determine whether a 0 or a 1 has been transmitted, we check whether the waveform corresponding to a 1 has been received at the right time. If it has not, we conclude that a 0 has been received.

This leaves us with the question of how to determine whether the signal waveform we are looking for has actually been received. A matched filter is often used for this purpose in telecom engineering. The impulse response of such a filter exactly matches the desired waveform (although it is mirrored in time). Our 0 waveform consists of a combination of three rectangular pulses. This makes it fairly easy to implement a matched filter in the form of a CIC filter, since the impulse response of such a filter — the response to a single 1 at the input — is naturally a single rectangular pulse, as described in [4].

The filter is shown schematically in **Figure 7**. Here the signal is sampled at a rate of 500 samples per second. The first pulse has a length of 25 ms and therefore corresponds to twelve samples. The second pulse has a length of 50 ms and corresponds to 25 samples. The final pulse, like the first one, has a length of 25 ms. These pulse lengths translate directly into the required delay stages. This filter is ideal for detecting the 0 waveform. As the 1 waveform is similar to the 0 waveform (consisting of two successive 0 waveforms), we can use the same filter and simply check the output 50 ms later. This is exactly the strategy that is used in the receiver.

The oscillograms (**Figures 8** and **9**) show the corresponding waveforms. The yellow trace shows the state of the timer that

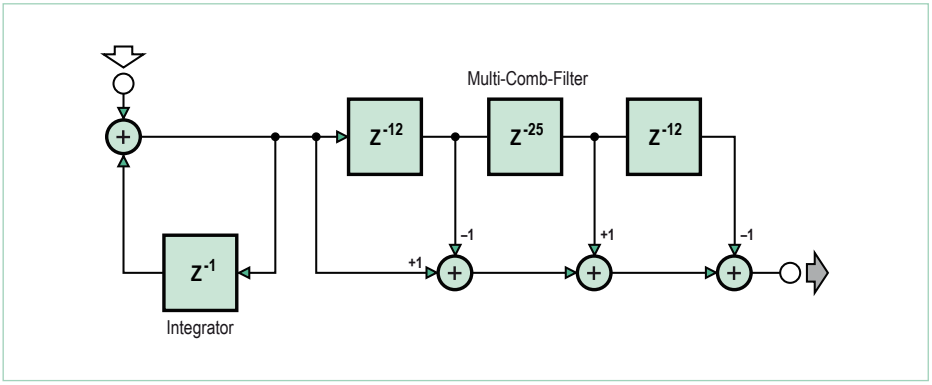


Figure 7. A matched filter implemented as a CIC filter.

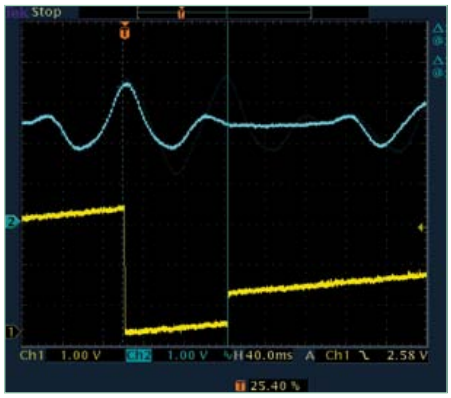


Figure 8. The output signal of the matched filter when a 0 is received.

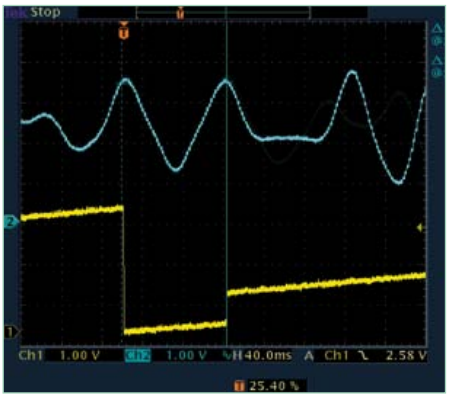


Figure 9. The output signal of the matched filter when a 1 is received.

detects the gap for the 59th second. The input signal is sampled at the points where there is a sudden change in the yellow signal, which is exactly where the peaks occur in the received signal.
The zeros and ones are stored in an array.

After the final bit is received, the data in this array is decoded and displayed on the LCD module. It is also output at the same time on the serial port at 19,200 baud. **Listing 4** shows an example of the received data.

The final instalment of this series will appear in the next edition. In that instalment we will look at bit clock synchronisation and describe the 'early late gate synchroniser', among other things.

Finally, we will decode the signal from the BBC198 Droitwich transmitter and get acquainted with several other decoding methods.

(120089-I)

Listing 4: Example data from TDF162

```
TDF 162
0001110000000000001011001100100010011110101100100000000100000: 08:19 17.02 WEDNESDAY 2010 p000
0000110000000000001010000010100010011110101100100000000100000: 08:20 17.02 WEDNESDAY 2010 p000
0000110000000000001011000010000010011110101100100000000100000: 08:21 17.02 WEDNESDAY 2010 p000
0000110000000000001010100010000010011110101100100000000100000: 08:22 17.02 WEDNESDAY 2010 p000
0001110000000000001011100010100010011110101100100000000100000: 08:23 17.02 WEDNESDAY 2010 p000
```

Internet Links

[1] www.elektor.com/100180
[2] www.elektor.com/100181
[3] www.elektor.com/100182
[4] www.elektor.com/120088
[5] www.elektor.com/120089
[6] http://hamsoft.ca/pages/mmtty.php
[7] www.dspguru.com/dsp/faqs/iir/basics
[8] www.npl.co.uk/science-technology/time-frequency/time/products-and-services/msf-radio-time-signal
[9] www.dspguru.com/dsp/faqs/fir/basics

Subscribe to *audioXpress* magazine!

Do your **electronics speak** to you? Are the words **"audio," "vacuum tubes,"** and **"speaker technology"** music to your ears?

Then you should be reading *audioXpress!*

Recently acquired by The Elektor Group, *audioXpress* has been providing engineers with incredible audio insight, inspiration and design ideas for over a decade. If you're an audio enthusiast who enjoys speaker building and amp design, or if you're interested in learning about tubes, driver testing, and vintage audio, then *audioXpress* is the magazine for you!

What will you find in *audioXpress*?

- In-depth interviews with audio industry luminaries
- Recurring columns by top experts on speaker building, driver testing, and amp construction
- Accessible engineering articles presenting inventive, real-world audio electronics applications and projects
- Thorough and honest reviews about products that will bring your audio experiences to new levels

Choose from print delivery, digital, or a combination of both for maximum accessibility.

Subscribe to *audioXpress* at
www.audioamateur.com
today!

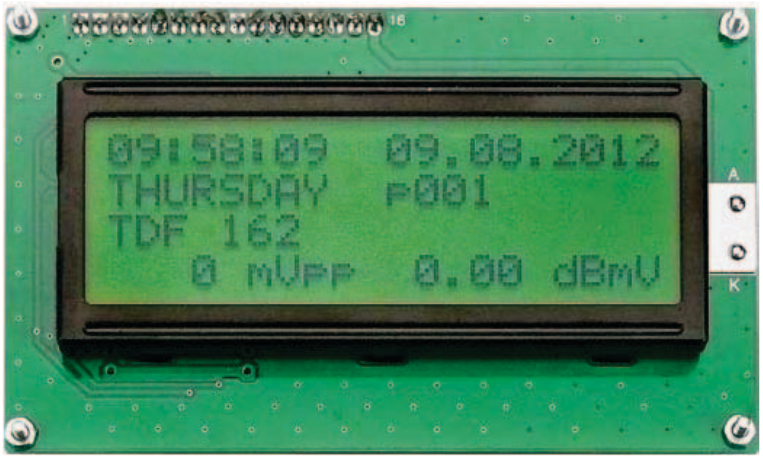
audioXPRESS

AVR Software Defined Radio (6)

Decoding options for BBC 198 kHz data signals

By Martin Ossmann (Germany)

This series has demonstrated that the ever-popular AVR controllers are also useful for signal processing. In this final instalment we take a closer look at a few more decoding methods. As usual we like to back up the theory with some practice: This instalment shows how to extract time-of-day information from BBC AM broadcasts.



In the last instalment of this series [5] we demonstrated the decoding of time-signal transmissions. Using the described digital IIR and matched CIC filter it is possible to receive and decode time signals sent by the German DCF77, the UK MSF and French TDF162 stations. In this instalment we go on to decode the digital information transmitted by the BBC on long wave at 198 KHz (Droitwich) and also the recently discontinued medium wave 648 KHz service. To this aim we look at alternative decoding strategies and demonstrate how they can be implemented.

Bit-clock synchronisation

In the world of communications technology there are many methods by which synchronous data can be sent through the ether. Here a digital data stream runs continuously through the communication system at a defined clock rate. At the receiver the signal often first undergoes demodulation. The resultant signal is then a sequence of data bits which contain the transmitted information.
For this experiment to achieve bit-clock synchronisation with the receive data it will be necessary to use a simple radio front-end (or the receiver board available from the Elektor shop), the receiver software 'EXP-RX-FM-125kHz-RDSlike-BitSync-V01.c', a signal generator with a 125 kHz series resonant circuit and the transmitter software 'EXP-TX-FM-125kHz-RDSlike-V01.c'. This last program generates frequency modulated digital data at 125 kHz with a data rate of 31.25 bit/s. The data stream and bit clock are output from the D/A converter outputs.
At the receiver site the signal is sampled at a frequency of

31.25 Hz x 16 = 500 Hz

to recover the clock. Each bit is therefore sampled 16 times. You could try to reconstruct the 31.25 KHz bit clock from the received

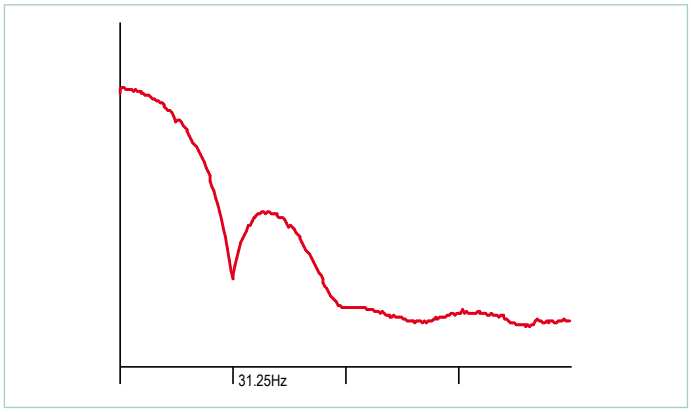


Figure 1. Frequency spectrum of the 31.25 Hz data signal.

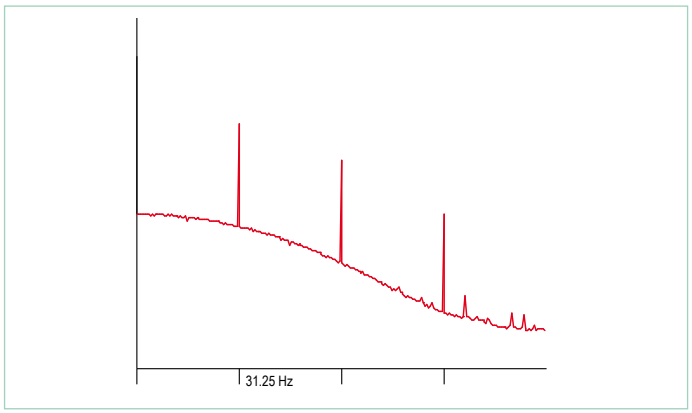


Figure 2. The spectrum of the rectified data signal.

data signal by using a PLL (phase locked loop). Unfortunately it is not quite so simple. The reason becomes clear when you look at the received signal spectrum given in **Figure 1**. There is in fact no signal component at 31.25 Hz for a PLL to lock onto.

Once the data signal has been rectified however, its spectrum (**Figure 2**) now shows a component at 31.25 Hz. A PLL will now be able to lock onto the signal. The principle of the data recovery PLL is outlined in **Figure 3**.

This PLL is in fact similar to the design used in an earlier instalment of this series to recover the carrier frequency (see Figure 7 in the fourth instalment of this series [4]). With reference to the 'EXP-RX-FM-125kHz-RDSlike-BitSync-V01.c' program the bit-clock frequency is controlled in the following manner: The do-Signal-Sample() routine in **Listing 1** is called at 16 times the bit clock rate. In this routine the variable Clk16 is incremented or reset to zero when it reaches 16. The phase of the saw-tooth signal is regulated so that the centre of the ramp (Clk16 = 8) is coincident with the centre of the transmitted bit. The bit value is evaluated and processed in the routine doBitSample(). Phase adjustment is performed in the following manner; in the normal case Clk16 is incremented to 16 and then reset by the digital signal clock. Now depending on the sign of the output from the phase comparator this value can be changed to either 15 or 17 to provide some phase adjustment.

In order to correctly control the clock PLL it is necessary to output the demodulated signal at the output of DAC1. Clk16 with Peak at Clk16 = 8 (i.e. at the sampling instant) is output from DAC2. **Figure 4** shows the reception of BBC Radio 4 at 198 KHz LW (see **Figure 5**).

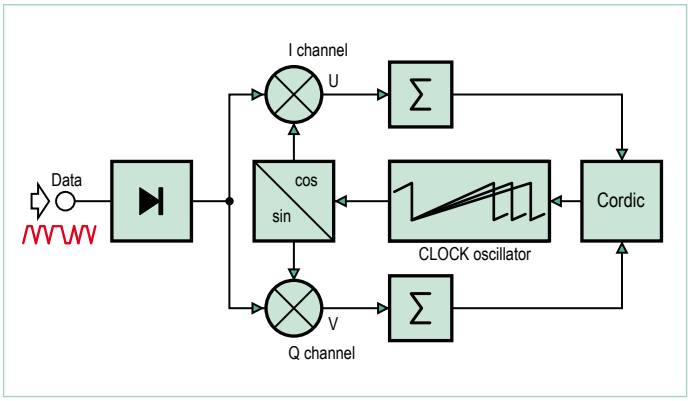


Figure 3. The clock-PLL principle.

Listing 1: Phase adjustment

```
void advanceCLK16() {
  Clk16++ ;
  if ( Clk16 >= ClockPeriod ) { Clk16=0 ; }
  }

void doSignalSample(){
  int16_t Signal ;
  ATOMIC_BLOCK(ATOMIC_FORCEON) {
    Signal=FRQcicout ;
  }
  ClockPLL16(Signal) ;
  doBitSample(Signal) ;
  advanceCLK16() ;
}
```

Early-Late Synchroniser

In addition to the conventional PLL a so-called 'Early-Late' synchroniser can be used to recover the clock signal. The eye-diagram shown in **Figure 6** helps explain how this technique works.

Again it is necessary for the Clk16 signal to synchronise with the bit clock so that the data signal is sampled in the centre of the bit. In order for this to be achieved the signal is sampled at the early time E and the late time L. This can be performed, for example on the third (E) and twelfth (L) clock of the Clk16 clock. When both of these measured values are the same amplitude then we can be confident that E and L are centred about the mid point. When on average one set of the values is greater than the other it indicates that the eye has shifted from its optimal point and phase of Clk16 requires some correction.

The program code corresponding to this process in given in **Listing 2**. Ten samples of the E (early) and L (late) values are summed to produce the value 'earlyLateDelta'. This value is then used to correct the Clk16 period. The code for this synchroniser can be tested with the 'EXP-RX-FM-125kHz-RDSlike-BitSync-V01.c' program.

Decoding the phase modulated BBC198 signal

Now we come to a practical application of bit synchronisation when we program the receiver to recover the phase-modulated data sent out by the BBC as part of its Radio 4 LW broadcast on 198 kHz.

Details of the transmitted signal are given in the BBC publication [7]. Data is sent by modulating the 198 KHz carrier phase by ±22.5°. The data rate achieved is 25 bit/s using Manchester coding (also known as bi-phase coding) with filtering. Using this type of modulation ensures that the average phase deviation always sums to zero

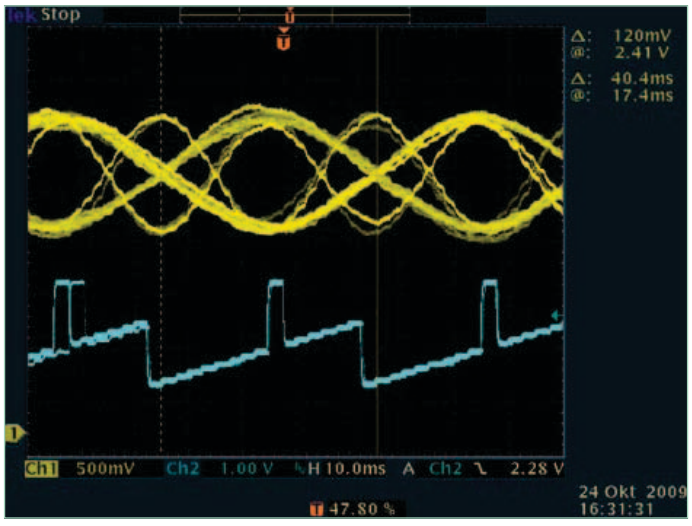


Figure 4. Eye diagram of the BBC198 signal.

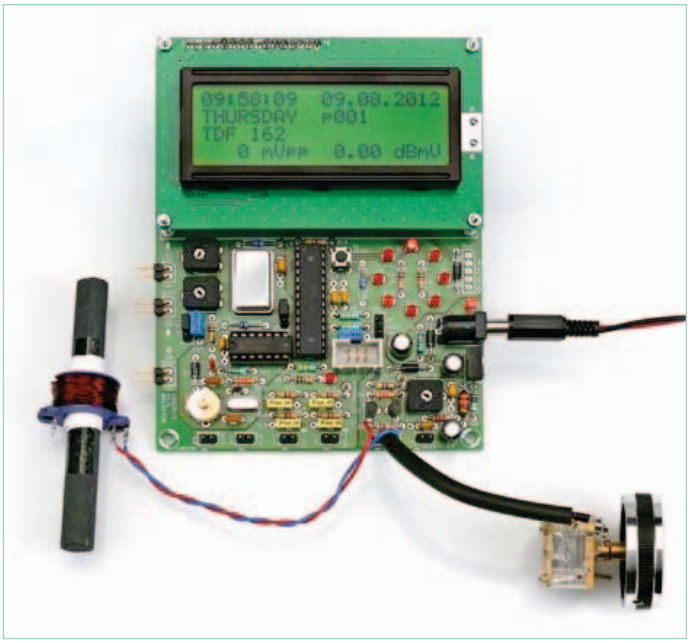


Figure 5. The complete receiver for reception of BBC198.

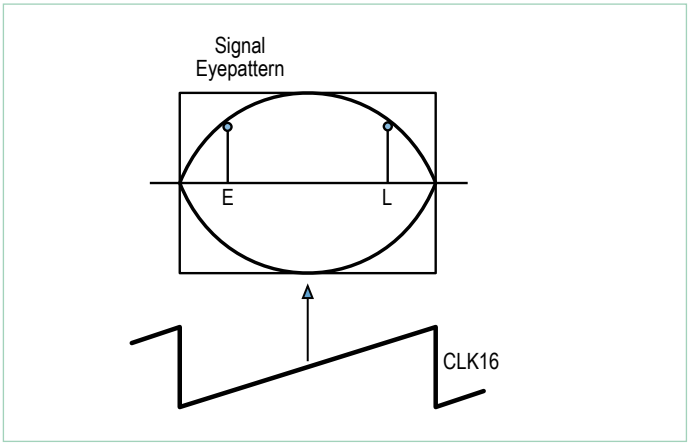


Figure 6. Eye diagram of the early-late synchroniser.

Listing 2: Early-Late Synchroniser

```
void earlyLateSynchronizer(int16_t Signal){
  if ( Clk16==3  ) { earlyValue=abs(Signal) ; }
  if ( Clk16==12 ) { lateValue=abs(Signal) ; }
  if ( Clk16==8  ) {
    earlyLateDelta += lateValue-earlyValue;
    ClkCnt++ ;
    ClockPeriod=16 ;
    if (ClkCnt==10){
      ClockPeriod += earlyLateDelta/16 ;
      earlyLateDelta=0 ;
      ClkCnt=0 ;
    }
  }
}
```

so (as before the modulation was introduced) the carrier can still be used as a highly stable frequency reference.
The A/D converter uses a clock equal to

20 MHz / 2,500 = 8 kHz.

At this sampling rate the 198 kHz carrier is mixed down to

198 kHz – (25 x 8 kHz) = –2kHz.

The 'intermediate frequency' is therefore –2 kHz and is sampled at four samples per period in the same way as in the IQ sampling mixer earlier. The negative sign indicates that the mirrored sideband is used. After separation into the *I* and *Q* components both are first passed through second-order CIC filters and downsampled with an M_d factor = 16. Next in line is another low-pass CIC but this time without downsampling. Instantaneous values of amplitude and phase values are now produced from these filtered *I* and *Q* signals processed using the CORDIC process.
One possibility for signal demodulation is to use a PLL to recover the carrier and then decode the NRZ (Non Return to Zero) signal. Here it refers to the original data signal which has been used to phase modulate a carrier and which will be recovered in the receiver. For decoding however the use of a PLL control loop which interprets a phase modulation as a frequency modulation is not necessary. The conversion is shown in **Figure 7**.
The theoretical NRZ-phase response is represented as a signal response p(t). To reduce the signal bandwidth phase changes occur only slowly. The actual phase response is more like the signal q(t). The sampling rate for the demodulated signal is 400 Hz so each bit is sampled 16 times. Now according to communication theory frequency is the rate of change of phase. The frequency response f(t) is given by the derivative of q(t) with time. When the phase increases linearly with time there is a constant positive frequency. When the phase decreases linearly with time there will be a constant negative frequency.
Using additional low pass filtering the signal g(t) is recovered from f(t) and can now be readily evaluated. The data stream is sampled mid-bit to determine its value. When the value is positive it is interpreted as a '1' and when negative as a '0'. Application of the CORDIC algorithm produces phase samples to give the instantaneous frequency. These are again low-pass filtered with a CIC filter to produce the signal g(t). This signal in turn supplies the bit-clock recovery. This results in the waveform shown in Figure 8 in the second instalment of this series [2].
With more processing the blocks of data are identified and decoded. How this functions will be explained later on. For now **Listing 3** shows a data sample received from the BBC 198 kHz signal.

Block Synchronisation

Once the data bit stream has been recovered from the receive signal the final task is to identify the beginning and end of the data packet. This process is known as block synchronisation.
The majority of radio traffic systems use blocks of data of fixed

length. When the receiver is first switched on it is necessary to determine where the beginning of each block occurs in the data stream. There are several commonly used techniques to identify data start; one method is to use a unique sequence of bits to form a preamble. This sequence is never allowed to occur in the data field. RFID tags of the type EM4102 use this method. This preamble occupies space in the data stream and therefore, to some extent reduces the available channel data bandwidth. Many communication systems use the information contained in the data for error correction field as the block boundary marker.

Error detection

Many data communication systems use a checksum added to the end of each block of data which allows the receiver to detect data corruption and also to correct the error. It can be shown that this additional information can also be used to perform block synchronisation. First we will describe how the checksum is calculated using the 125 KHz test transmitter together the 'EXP-TX-FM-125kHz-RDSlike-V01.c' program. The calculation software is given in **Listing 4** and the CRC hardware generator block diagram is shown in **Figure 8**.

Ten CRC check bits are calculated using a shift register with XOR gated feedback taps. The 16 data bits are clocked through a shift register and control the new bits which end up in the syndrome shift register. The CRC checksum is produced after 16 clock periods corresponding to the 16 data bits. A hex value of 198H is added to the checksum using the XOR function which simplifies the synchronisation process at the receiver site. The 10 check bits are sent after the 16 data bits. This technique is very similar to the standard RDS system described in [8].

When the receiver knows where a new block starts it computes the syndrome (in exactly the same way as the transmitter) of the following 16 data bits and compares it with the 10 CRC bits following the data. In order to find the start of the block it is necessary to check each time a new bit is received whether the previous 26 bits produced an error-free combination of 16 data bits and 10 checksum bits. This test produces an 'OK' at a spacing of 26 bits while receiving a continuous data stream. Occasionally an 'OK' will occur in an incorrect position. The block synchronisation algorithm looks for a pattern with three 'OK' signals spaced by 26 bits. When this is detected it assumes that the block start has been correctly identified.

Figure 9 shows schematically how the check bits are continuously calculated. The syndrome calculator can be looked on as a type of filtering. The feedback shift register is configured in the same way as the one used in the transmitter path. The continuous stream of bits are processed and as each bit emerges from the 10+16 bit shift register it is taken into account in the syndrome shift register. The software need only check if the syndrome shift register contains the value 198H which was the value added during encoding at the transmitter end. When this value is found the software responds with an 'OK'. A counter called 'localTime' is incremented from zero to 25. A value of zero indicates that a complete block together with the check bits have been processed.

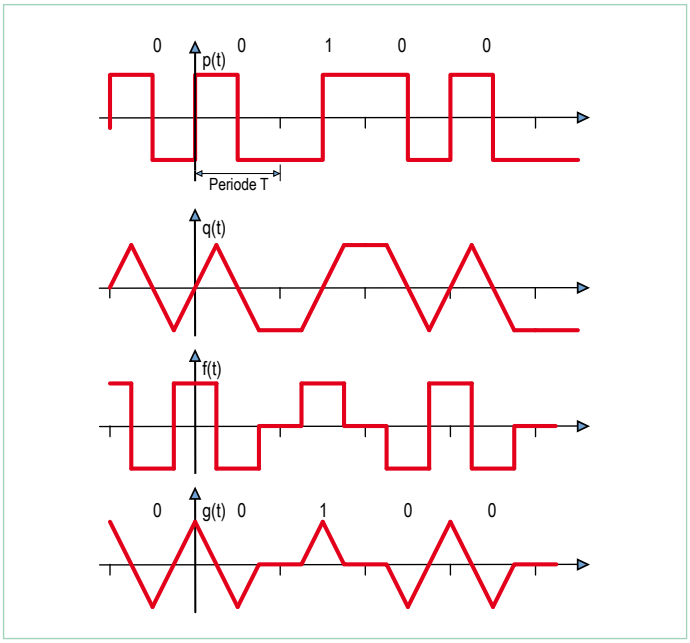


Figure 7. Phase modulation as frequency modulation.

Listing 3: BBC 198 kHz signal: receive Data

```
+E EB11 0391
+0 AAAA AAAA
+E E8F1 0391
+E EB09 0391
+E EB49 0391
+0 AAAA AAAA
+E EB31 0391
+E EB41 0391
+0 5475 2980  time= 18:38 week=07 day=02 tuesday
+E EB29 0391
```

Listing 4: Transmitter CRC calculation

```
SendBits(w,16) ;
Syndrom=0 ;
for (j=0 ; j<16 ; j++) {
  Syndrom=Syndrom<<1 ; // outgoing bit is at mask
0x0400
  if (w & 0x8000) {
    Syndrom ^= 0x0400 ;  // xor into outgoing bit
  }
  w=w<<1 ;
  if ( Syndrom & 0x0400 ) {
    Syndrom ^= 0x1B9 ;
  }
} ;
Syndrom=(Syndrom^ofs)<<6 ;
SendBits(Syndrom,10) ;
```

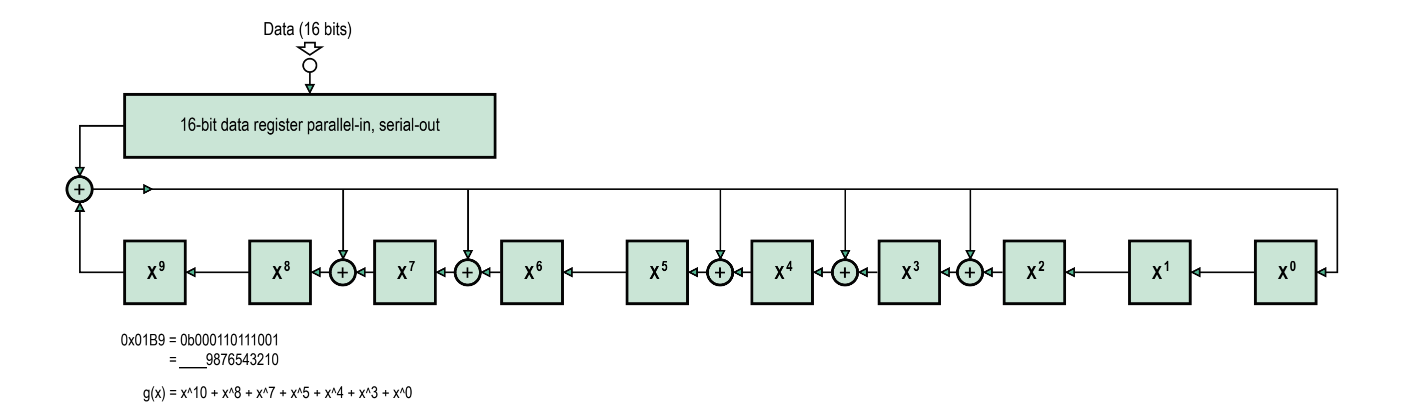


Figure 8. CRC1.

To demonstrate this behaviour, a variant of the receiver was programmed so that the actual time (localTime) is output as a sawtooth waveform from the PWM-DAC output. The second PWM DAC output will be high when the synchronisation circuit detects an 'OK'. The repetitive waveform can be seen in **Figure 10.** Also visible is a position where a false 'OK' is detected. It is may be necessary to wait a little before this random event happens.

After synchronisation when the 'OK' occurs at the time interval zero it indicates that error-free block has been received and the data can be output. With this the data reception process is complete.

This description of synchronisation applies for the BBC long wave transmission on 198 kHz. The data sent by the BBC does not however contain RDS parameters but uses blocks of data containing 47 data bits with a 13-bit checksum. The data is described in detail in the documentation [7] and is relatively easy to extract. In particular, for simplicity the time-of-day information is simply coded in one complete block.

BBC648 AMSS reception and decoding

This last exercise was intended to demonstrate data reception from the European World Service signal which the BBC transmits on 648 KHz. In the mean time due to budget cuts transmission on this frequency has sadly been discontinued. It is however instructional to demonstrate how the AMSS data is decoded in this case. Up until recently the BBC sent low frequency AMSS data as part of its 648 kHz AM transmission (see the associated literature [10]). The front end of our receiver samples at 20 MHz / 1,875 = 10,666... kHz. This results in an intermediate frequency of:

648 kHz – 61 x 10,666... kHz = -2,666... kHz = 0.25 x 10,666... kHz.

Bi-phase signalling is used with a phase modulation of ±20°.

With regard to the modulation and bit-coding method used by the data traffic it is (almost) identical to 198 KHz system described earlier. The difference here is that each block consists of 36 message bits and 11 CRC check bits. Two blocks form a group and a defined number of groups form a DRM-SDC block. A complete description of all the features is available in the DRM specification. Our simple receiver only extracts and displays the name of the transmitter station but is, never the less, a fully functioning data receiver.

Listing 5 shows the receive data. The output '2/7+' indicates that the second group from seven has been received with no detected errors. The data is decoded when all seven groups have been received.

And finally...

This now brings to a close the six-part series on signal processing and 'Software Defined Radio' using AVR microcontrollers. It has powerfully demonstrated just what can be achieved with even relatively modest microcontrollers. Commercial SDR receivers employ

Listing 5: BBC 648 kHz signal: AMSS SDC data

```
[6 1]BBC  WS   [C B]   [6 B]   0/7+ 1/7+ 2/7+ 3/7+ 4/7+ 5/7+ 6/7+ 7/7+!0007 ;
[6 1]BBC  WS   [C B]   [6 B]   0/7+ 1/7+ 2/7+ 3/7+ 4/7+ 5/7+ 6/7+ 7/7+!0007 ;
[6 1]BBC  WS   [C B]   [6 B]   0/7+ 1/7+ 2/7+ 3/7+ 4/7+ 5/7+ 6/7+ 7/7+!0007 ;
[6 1]BBC  WS   [C B]   [6 B]   0/7+ 1/7+ 2/7+ 3/7+ 4/7+ 5/7+ 6/7+ 7/7+!0007 ;
[6 1]BBC  WS   [C B]   [6 B]   0/7+ 1/7+ 2/7+ 3/7+ 4/7+ 5/7+ 6/7+ 7/7+!0007 ;
```

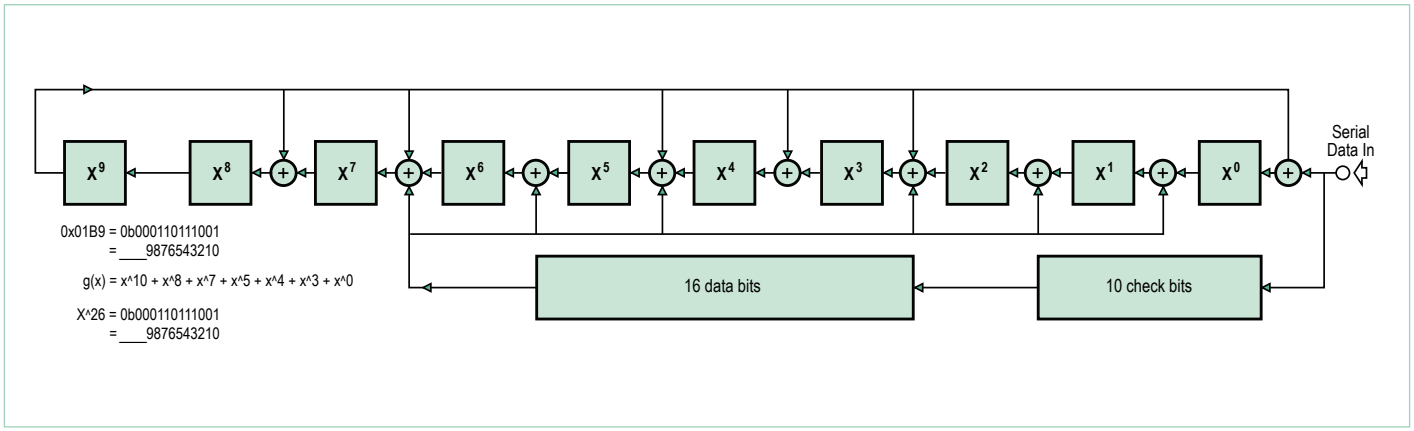


Figure 9. CRC2.

exactly the same techniques as detailed here; the only difference is that they use more powerful DSPs and radio front-ends which of course also makes them correspondingly more expensive!

(120392)

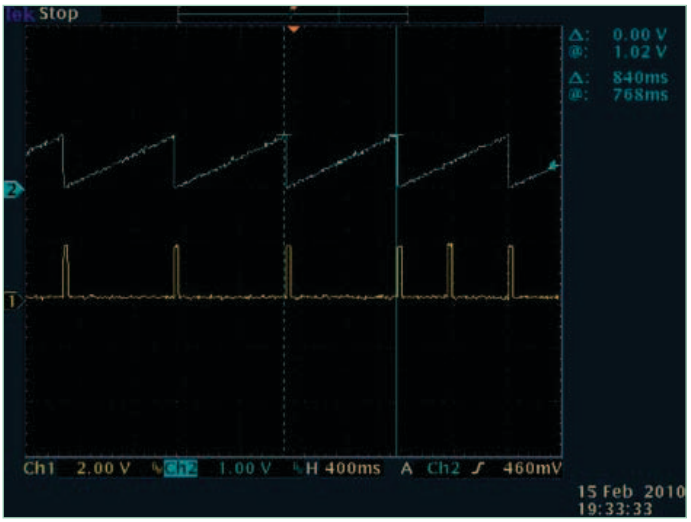


Figure 10. The OK signal.

Internet Links

[1] www.elektor.com/100180
[2] www.elektor.com/100181
[3] www.elektor.com/100182
[4] www.elektor.com/120088
[5] www.elektor.com/120089
[6] www.elektor.com/120392
[7] http://downloads.bbc.co.uk/rd/pubs/reports/1984-19.pdf
[8] ftp://ftp.rds.org.uk/pub/acrobat/rbds1998.pdf
[9] www.drm.org/
[10] www.ebu.ch/fr/technical/trev/trev_305-murphy.pdf
www.broadcastpapers.com/whitepapers/ABUBBCamss2006.pdf?CFID=16508900&CFTOKEN=dac28b1a87e54d77-47F9A337-9A30-F5E3-667FAB2A9EA27223

Elektor products and support

- Signal generator (Kit with PCB and all components, # 100180-71)
- Universal receiver (Kit with PCB and all components, # 100181-71)
- Active ferrite antenna (Kit with PCB and all components, # 100182-71)
- Combi-kit with all three components plus USB/TTL converter BOB FT232: # 100182-72
- USB/TTL converter BOB FT232, fully populated and tested, # 110553-91
- Free software downloads (hex files and source code)

All of these products and downloads are available through the web page: www.elektor.com/120392