25. AM radio receiver

The chapter describes the programming of a microcontroller to demodulate a signal from a local radio station. To keep the circuit simple the signal from the local amplitude modulated radio station should be strong, this is the case in the place where the author of this chapter lives. The local radio station transmits at 918 kHz, and a simple LC tuning circuit and antenna suffice to get a signal that can be sampled by the microcontroller. The demodulation is done in real time by the software, and the demodulated signal is sent to an active speaker using a DAC.

25.1. The theory

The block diagram of the AM receiver is depicted in Fig. 25.1. The input signal for the receiver comes from an antenna, but may also come from a suitable amplitude modulated function generator. The input signal gets sampled into *X1*, then comes the block for the removal of a DC component to obtain signal *X2*. Next is a multiplication of the signal *X2* with two sinusoidal signals from a software local oscillator (a DDS generator can be used here); the two output signals from the local oscillator are in quadrature, i.e. their phases differ for 90 degrees. Two low pass filters strip away high frequency components, and the final block calculates the current amplitude of the incoming signal *X1*.

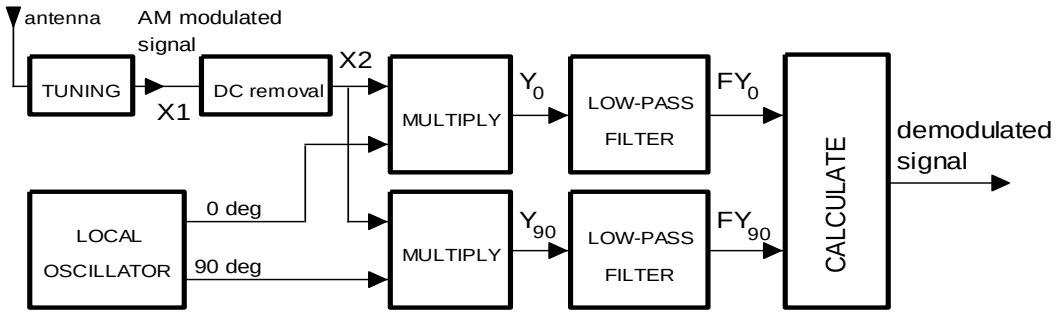

Figure 25.1: The AM receiver: block diagram

Let the input amplitude modulated signal *X1* be:

$$X_1 = A(t)\sin\omega t$$

The frequency ω is the carrier frequency of the received signal, in our case 918 kHz. The amplitude *A(t)* stands for the envelope of the input signal *X1*; since Y1 is amplitude modulated, the envelope depends on time as emphasized by the part in brackets. The envelope represents the signal to be transmitted by the radio, the speech of a narrator, or music. The frequency spectrum of the envelope is limited by the radio transmitter to 4.5 kHz.

The Nyquist criteria states that a signal should be sampled at least twice per period, this would require the sampling frequency of about 2 MHz.

The ADC in the microcontroller can sample at this speed; however, even the microcontroller as fast as the STM32F407 cannot process signals this fast unless we use a trick that is based on an extension of the Nyquist criteria, the trick is called “under-sampling”. The details are explained in [1], here we just present a brief justification of the technique.

Consider a harmonic signal, sampled more than twice per period as the Nyquist criteria requires. An example is given in Figure 25.2; the signal (green) has a frequency of 18 kHz, and the sampling rate is 100 kHz. The green plot is the signal, and the blue dots represent the samples taken by the ADC.

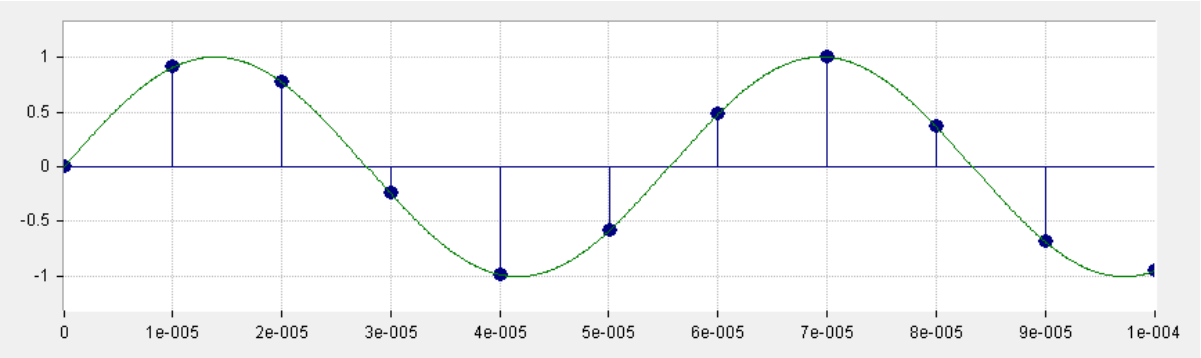

Figure 25.2: Sampling more often than Nyquist criteria requires; horizontal scale in seconds, arbitrary vertical scale

Consider now the same sampling rate and a new input signal with frequency of 318 kHz, which is clearly above the Nyquist requirement. This is shown in Figure 25.3 with the new input signal in red. The sampled values are exactly the same as when sampling the green signal, and the observer is deceived to sample the green signal as in Figure 25.2. There is no way to tell the frequency of the input signal from samples obtained (the blue dots) unless we increase the sample rate.

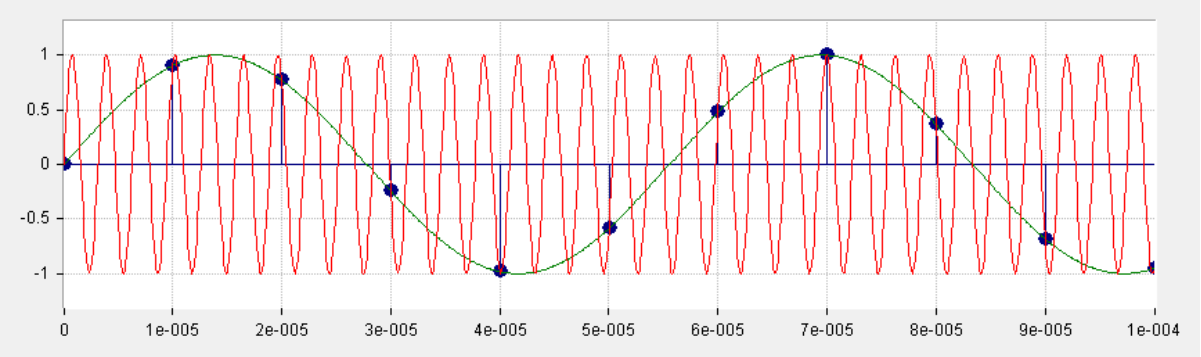

Figure 25.3: Under-sampling: the input signal frequency is 318 kHz

Consider the same sampling rate and an input signal with frequency of 918 kHz, Figure 25.4. Again, the sampled values are the same as when sampling a signal with frequency of 18 kHz!

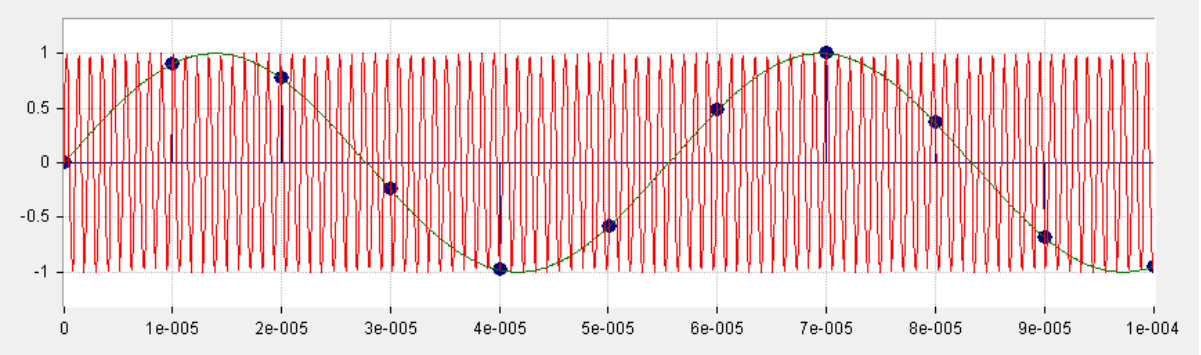

Figure 25.4: Under-sampling: the input signal frequency is 918 kHz

This is why the Nyquist criteria requires to limit the frequency of the input signal to one half of the sampling rate f_s: to be able to determine the frequency of the input signal without any doubts. Also, if there are multiple signals present, and some are above the Nyquist frequency, those appear as having

a frequency in the range from zero to $f_s/2$, and can deceivably change the result of processing. However, if one is not interested in the frequency of the input signal, and one is capable of limiting the range of input frequencies, then one can sample a signal less than twice per period. The signal *X*, which has a frequency f_X anywhere between f_s and $3f_s/2$, will appear as having a frequency of f_x-f_s. Also, when the frequency f_X lies between $2f_s$ and $5f_s/2$, the sampled version will appear to have a frequency of f_x-$2f_s$, Figure 25.5.

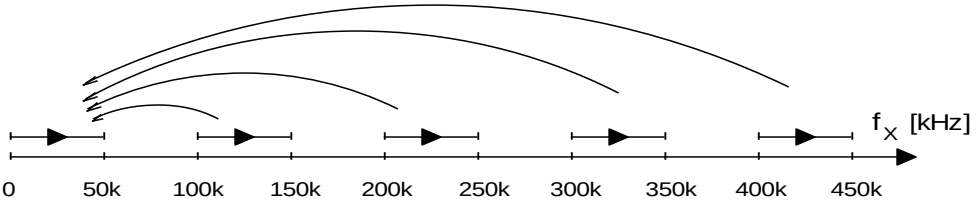

Figure 25.5: Under-sampling regions:
input signal with frequency f_X $(nf_s \le f_X \le (n+0.5)f_s)$ translates as having frequency $f'_X = f_X - nf_s$,
input signal with frequency f_X $((n+0.5)f_s \le f_X \le (n+1)f_s)$ as having a frequency $f'_X = (n+0.5)f_s - f_X$,

We can therefore sample the 918 kHz input signal with a sampling frequency of 100 kHz, and the sampled version of the input signal *X1* will appear to have a frequency of 18 kHz!

There is only one detail still to be considered: the sampling time of the sample & hold (S&H) circuit at the input of the ADC. This circuit is used to make a local copy of the input analog voltage and hold is steady during the conversion. The circuit needs some time to make a local copy, this time is selected by a parameters during the initialization of the ADC. The time to make a copy should be significantly shorter than a period of the input signal, in our case about 1 μs; if the sampling time is long then the circuit is actually averaging the shape of the input signal during the sampling time, and this results in reduced amplitude of the sampled signal. The sampling time of 100 ns is arbitrary the maximum we can afford. In our case the initialization of the ADC uses 'ADC_SampleTime_3Cycles' parameter for function call "ADC_RegularChannelConfig". With the ADC clock set at 30 MHz, one ADC clock cycle takes about 30 ns, and three clock cycles take about 90 ns, which is still sufficient not to reduce the amplitude significantly.

The rest of the data processing is close to the one described in Chapter 23. The sampled input signal *X1* is first stripped of a DC to make *X2*. Since the frequency of the input signal is at 18 kHz, the "DC removal" filter returns almost the same amplitude as the AC signal *X1* has, see the diagram in Figure 24.3.

$$X_2 = A(t)\sin\omega t$$

A local DDS generator provides two signals with frequency$\omega_l = 2\pi \cdot 18\ kHz$, these are phase shifted for 90 degrees. Two multipliers provide signals *Y0* and *Y90*.

$$Y_0 = A(t)\sin\omega t \cdot \sin\omega_l t = \frac{A(t)}{2}\left[\cos(\omega-\omega_l)t - \cos(\omega+\omega_l)t\right]$$

$$Y_{90} = A(t)\sin\omega t \cdot \cos\omega_l t = \frac{A(t)}{2}\left[\sin(\omega-\omega_l)t + \sin(\omega+\omega_l)t\right]$$

Each of these intermediate signals is composed of two parts. Since the frequency of the local oscillator is close to the frequency of the sampled signal, the frequency of the left component $(\omega - \omega_l)$ is almost zero, while the frequency of the right component $(\omega + \omega_l)$ is almost twice the frequency of the local oscillator. Low-pass filters, as the next stage in signal processing, remove both high frequency components.

A brief discussion is in place here. Actually, the left component is multiplied with the envelope signal A(t) which spans up to 4.5 kHz; the product therefore spans up to $4.5\ \mathrm{kHz} + |\omega - \omega_l|$. The low-pass filter should have a corner frequency of at least 4.5 kHz to let the usable components pass. The implemented version of a filter has a corner frequency of 2500 Hz, this affects the frequency spectrum of the received signal, and the music sounds as having a "treble" setting on a regular receiver pushed to a low setting to muffle the high tones. A different set of filter coefficients might be in order here.

The result of the filtering are then *FY0* and *FY90*:

$$FY_0 = \frac{A(t)}{2}\cos(\omega - \omega_l)t \qquad FY_{90} = \frac{A(t)}{2}\sin(\omega - \omega_l)t$$

These two signals contain the information on the amplitude *A(t)* of the signal *X*, and we can calculate the envelope using a simple mathematics:

$$\sqrt{FY_0^2 + FY_{90}^2} = \frac{A(t)}{2} \qquad \rightarrow \qquad A(t) = 2\sqrt{FY_0^2 + FY_{90}^2}$$

Calculating the geometrical sum of signals FY_0 and FY_{90} gives a weighted envelope of the detected signal, this is then sent to an active loudspeaker for listening.

25.2. The implementation of the AM radio receiver

The AM radio receiver project requires an antenna and a tuning circuit, the schematic diagram is given in Figure 25.6. A signal from a function generator can be connected instead, and the generator can be set to generate the AM signal. The output of the tuning circuit is applied to the analog input of the microcontroller, ADC_IN2, where it gets sampled into *X1* and then processed.

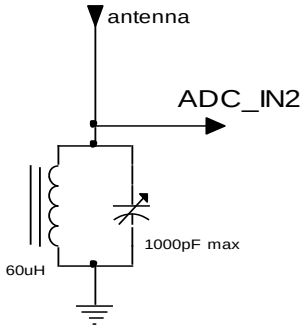

Figure 25.6: The external circuit to the BaseBoard for the AM radio receiver

The program for signal processing is implemented as an interrupt routine, which is driven by timer TIM5, the time interval between successive interrupts is fixed to 10 μs (100 kHz sampling). The program generates signals for local oscillator using the DDS technique, and processes the acquired signals.

The sampling of the input signal is implemented the same way as described in chapters 12 and 13. The timer overflow triggers the ADC to start the conversion, and the end-of-conversion signal from the ADC is used to interrupt the microcontroller and start the interrupt routine ADC_IRQhandler.

The theory behind the AM demodulation requires the calculation of a geometrical sum of two signals. Squaring a variable is fast, but taking a square root of a variable is slow; this takes about 4 μs on the STM32f407 running at full speed. This is just sufficient to perform the complete data processing between two successive samples.

The ADC interrupt function for the AM radio receiver is presented in the listing below.

```
void ADC_IRQHandler(void)  {
  GPIOE->BSRRL = BIT_8;         // about 8 us altogether                                  // 2

  int Spare = ADC1->DR;               // to clear ADC IRQ flag                            // 4
  X1[k] = (ADC2->DR & 0xfff);         // acquire X1                                       // 5
  X2[k] = X1[k] - X1[(k-1) & 63];     // remove DC component                              // 6

  Y0[k]  = X2[k] * Table[  ptrTable >> 4               ];     // multiply with sin        // 7
  Y90[k] = X2[k] * Table[((ptrTable >> 4) + 1024) & 4095];     // multiply with cos        // 8

  float conv = 0;                                              // LP filter (sin)        // 10
```

```
  for (int m = 0; m<5; m++)  conv += a[m] * Y0[(k - m) & 63];                                  // 11
  for (int n = 1; n<5; n++)  conv += b[n] * FY0[(k - n) & 63];                                 // 12
  FY0[k] = conv;                                                                               // 13

  conv = 0;                                                    // LP filter (cos)             // 15
  for (int m = 0; m<5; m++)  conv += a[m] * Y90[(k - m) & 63];                                 // 16
  for (int n = 1; n<5; n++)  conv += b[n] * FY90[(k - n) & 63];                                // 17
  FY90[k] = conv;                                                                              // 18

  float Res = FY0[k] * FY0[k] + FY90[k] * FY90[k];             // add geometrically            // 20
  float ResRes = (float)(sqrt((double)Res));                   // sqrt takes about 4 us !     // 21

  DAC->DHR12R1 = Table[ptrTable >> 4] + 2048;                  // local oscillator out        // 23
  DAC->DHR12R2 = (int)(ResRes / 64 + 128.0);                   // demodulated out             // 24

  ptrTable = (ptrTable + 11796) & 0xffff;                      // next excitation             // 26

  k++; k &= 63;                                                                                // 28
  GPIOE->BSRRH = BIT_8;                                                                        // 29
}
```

The function commences with setting pin 8, port E, high to signal the start of execution, line 2; the same pin is reset when the function ends, line 29. Lines 4 and 5 read the result of conversion from both ADCs and store one of them into circular buffer *X1*, which has 64 elements. The removal of the DC component is performed in line 6, and the result is stored in a circular buffer named *X2*.

The following two lines 7 and 8 implement the multiplication with locally generated signals with frequency of 18 kHz and different phases. The multiplication is followed by filtering for each component separately in lines 10 to 13 and 15 to 18. Both filtered components *FY0* and *FY90* are added geometrically in lines 20 and 21. The resultant envelope is sent to a speaker using a DAC2 in line 24, for testing purposes the signal of local oscillator is available at DAC1.

The frequency of the local oscillator depends on incrementing of the pointer to the table. For frequency of 18 kHz we need to use 11796 to increment the pointer, as implemented in line 26. In practice, any value between 10500 and 13000 will do; values off the correct value only reduce the amplitude of the detected envelope.

The listing of the rest of the program is given below.

```
int main ()  {
  for (ptrTable = 0; ptrTable <= 4095; ptrTable++)                                             // 2
    Table[ptrTable] = (int)(1850.0 * sin((float)ptrTable / 2048.0 * 3.14159265));              // 3

  GPIOEinit ();               ADCinit_T5_CC1_IRQ();          DACinit();                      // 5
  TIM5init_TimeBase_CC1(840);                 // 840 == 10us == 100kHz                       // 6

  while (1) {};                               // endless loop                                // 8
}
```

This listing starts with the inclusion of CMSIS files and the declaration of the variables used; this is not shown in this listing. The function “main()” commences with the initialization of the table for the DDS generator as already explained in chapter 16, and continues to the configuration functions for port E, ADC, DAC and timer TIM5; all configuration functions were already described in previous chapters.

The endless while loop, line 8, is empty; all data processing is performed in the interrupt routine.

[1] Richard G. Lyons: Understanding digital signal processing, second edition, chapter 2