UNIVERSITÀ DI TRENTO

Advanced Electronics Laboratory
Bell 202 Modem

Written By:
Francesco Osti

Date: March 2, 2022

Contents

1 Bell 202 Modem

Bell 202 modem is a modulation standard for audio frequency signals. Modulation used is continuous phase frequency shift keying with two symbols: "mark" at 1.2 kHz and "space" at 2.2 kHz. The baud rate used is 1.2 kbit/s. At the upper level data is encoded with the standard HDLC (High-Level Data Link Control). In particular is utilized Non Return to Zero (NRZI) encoding and bit stuffing (5 bits).

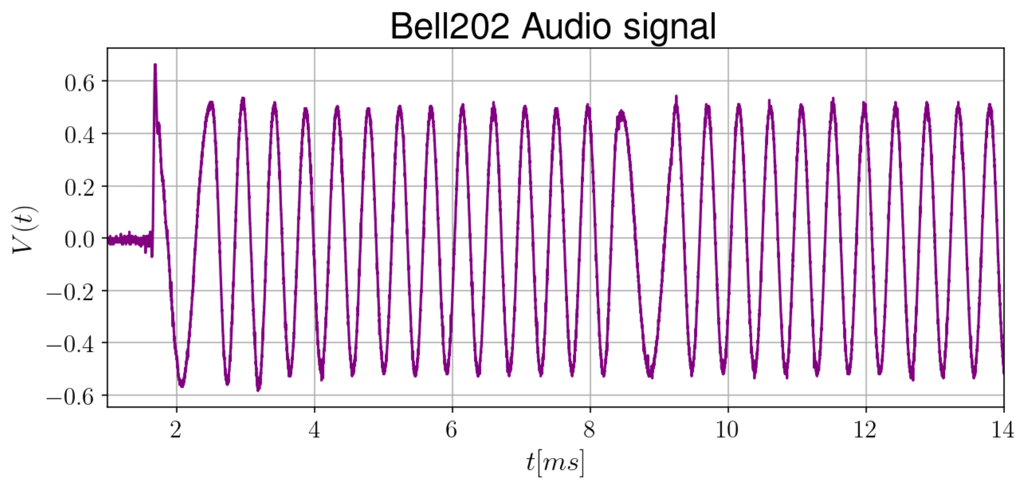


Figure 1.1: General block diagram

The NRZI encode a "0" bit with a change of frequency of the modulated sine signal, while a "1" bit is given if sine doesn't change modulation in a bit lenght time. Since the change of frequency of the sine is used to synchronize the received signal, if no frequency change happen within a given time synchronization can be lost. To solve this problem the bit stuffing technique is used. If 5 consecutive "1" bits are sent an adittional "0" bit is added to restore synchronization. If 6 "1" bit are received the byte is considered as a flag bit and this is used as a synchronization byte at the start of the message. If more than 6 "1" bits are detected the data is considered invalid.

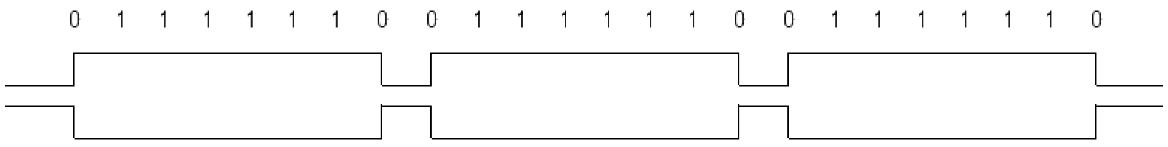


Figure 1.2: NRZI encoding, source [1]

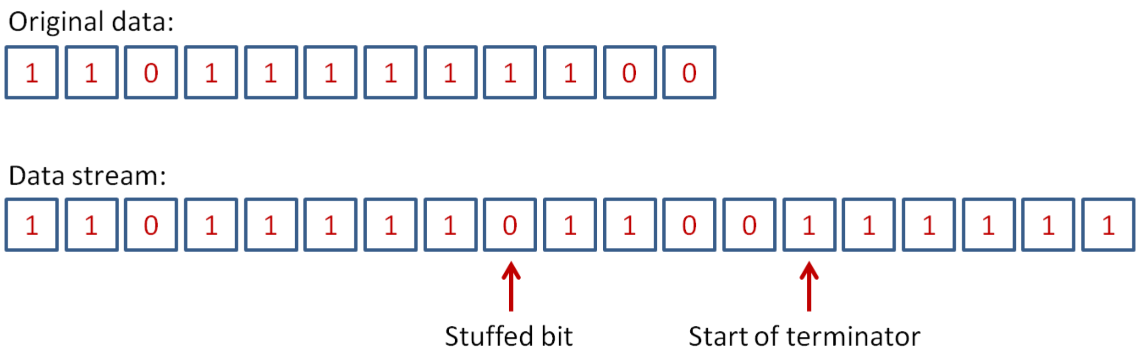


Figure 1.3: bit stuffing, source [2]

2 General scheme

The general scheme of the modem is composed of two parts: The Bell202 to Serial port and the Serial to Bell202. To do so each part is composed of a de/modulator, a de/serializer and a serial port interface. The demodulator takes care of detecting ”mark” or ”space” bit (DATA) and the presence of a signal (DET) from audio, while modulator to produce an audio signal from the digital data. The deserializer takes the data at the output of the demodulator, decode it and bring out 8 bits at a time. The serializer instead takes 8 bits of data, encode them and bring out them in a serial format. At last serial port interface send or receive data from serial port 8 bits at a time.

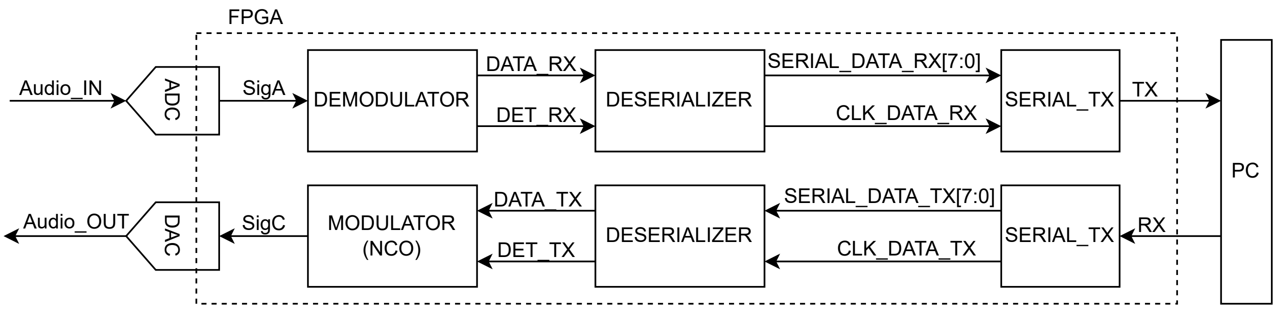


Figure 2.1: Bell 202 sample signal

3 Demodulator

The demodulation of the audio signal is done, after the sampling with the ADC, entirely in the digital form. To do this a etherodine scheme is used. The signal at the input was multiplied with a sine and cosine at 1.2 kHz and 2.2 kHz. Then 4 signals are produced, this are subsequentially filtered with a low pass filter, in this way the signal is down converted. For every signal the modulus is calculated and the in-phase and quadrature components are summed for both 1.2 kHz and 2.2 kHz to get the amplitude of the ”space” and ”mark” symbols. The data (DATA) at the output is obtained by using a comparator (with hysteresis) with at input the two amplitudes. The presence of a signal (DET) is obtained by comparing the sum of the two signals with a threshold.

The block diagram of the demodulator is presented below:

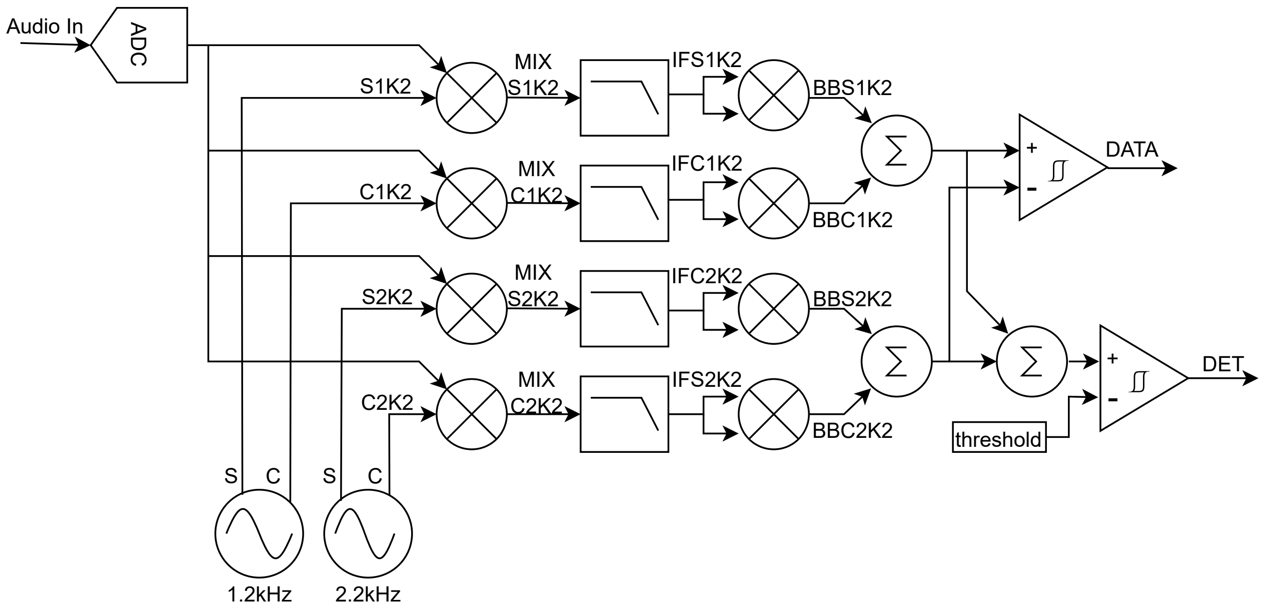
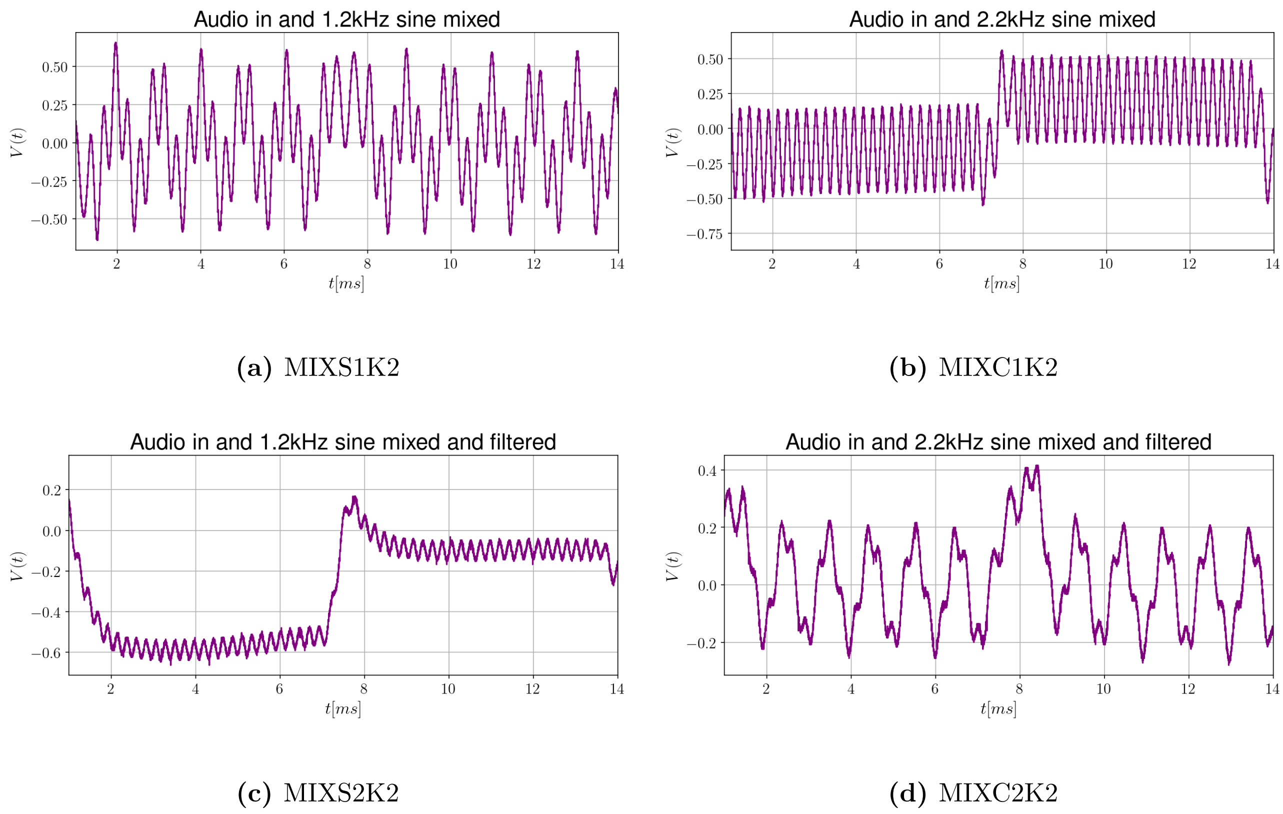


Figure 3.1: General block diagram

Some intermediate signal of the demodulator are presented below, all of them refer to the Bell 202 signal of figure 1.1 in the same time interval:



(a) MIXS1K2

(b) MIXC1K2

(c) MIXS2K2

(d) MIXC2K2

Figure 3.2: Signal produced inside the demodulate module

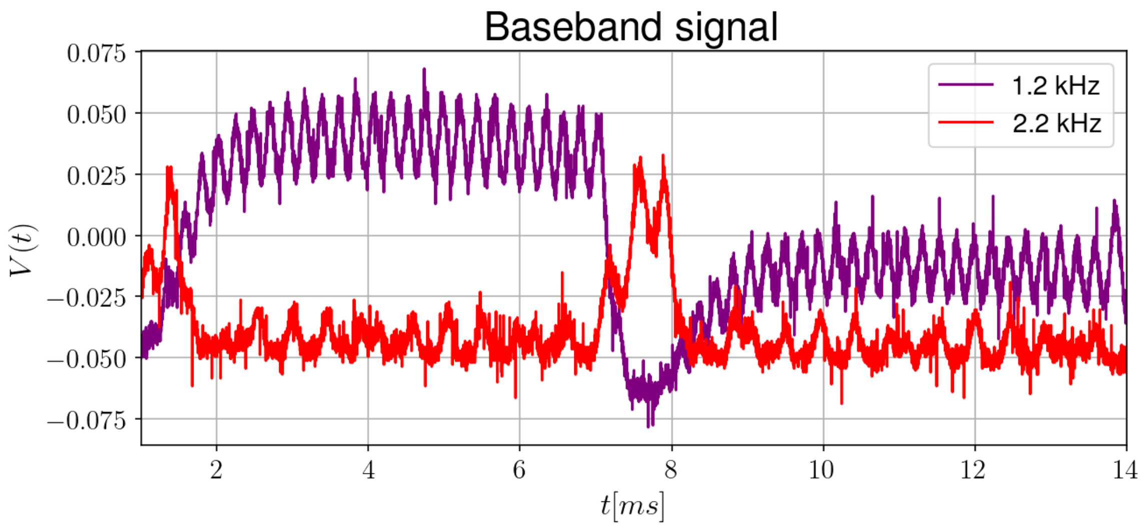


Figure 3.3: BBS1K2 and BBS2K2 signals

The last signal show two demodulated signals. If we compare the last figure with figure 1.1 we can see that "blue" signal (BBS2K2) is high when $2.2kHz$ audio signal is present, instead when $1.2kHz$ audio signal is present the "red" signal (BBS1K2) is high and BBS2K2 is low. From this we can retrive the DATA from the audio signal.

3.1 Oscillator

The oscillator required for the demodulator must produce a sine and cosine signal in quadrature and must be possible to tune frequency. For that reasons the oscillator utilized is the described below and it is taken from [3].
A simple oscillator can be made using the Cordic scheme where a sine and cosine are produced by considering them as the Cartesian's components of a vector that is rotated every cycle with a matrix:

$$\begin{pmatrix} Y_{n+1} \\ X_{n+1} \end{pmatrix} = \begin{pmatrix} +\cos(\omega_0) & -\sin(\omega_0) \\ +\sin(\omega_0) & +\cos(\omega_0) \end{pmatrix} \cdot \begin{pmatrix} Y_n \\ X_n \end{pmatrix} \tag{3.1}$$

This last recursive relation can be used to generate the signals but the output is unstable due to numerical errors and require 4 multiplications. To solve these problems we can rewrite the above equation in the following way:

$$\begin{aligned} X_{n+1} &= KX_n + Y_n \\ Y_{n+1} &= KY_n - (1-K^2)X_n = k(Y_n + KX_n) - X_n = KX_{n+1} - X_n \end{aligned} \tag{3.2}$$

where we can define K as:

$$K = 1 - \epsilon = 1 - k/2^{(n\_bits)} = cos(\omega_0) \tag{3.3}$$

This last recursive relation requires only three multiplications and is also stable against numerical errors. This recursive relation can be represented in the below block diagram:

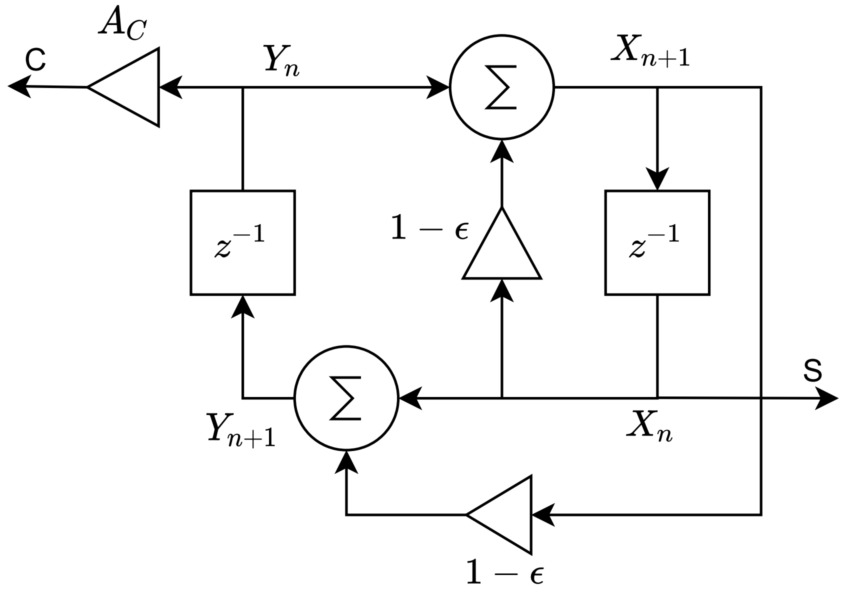


Figure 3.4: Oscillator block diagram

We can find the response of the oscillator by performing Z-transform on equations 3.2 and imposing initial conditions $X_0 = 1$ and $Y_0 = 0$ we get:

$$\begin{aligned} X(z) &= \frac{1 - Kz^{-1}}{1 - 2Kz^{-1} + z^{-2}} \\ Y(z) &= \frac{(K^2 - 1)z^{-1}}{1 - 2Kz^{-1} + z^{-2}} \end{aligned} \tag{3.4}$$

Which are Z-transform of:

$$\begin{aligned} x[n] &= cos(\omega_0 n)u[n] \\ y[n] &= sin(\omega_0)sin(\omega_o n)u[n] \end{aligned} \tag{3.5}$$

Where we have defined $cos(\omega_0) = K$
This show that $x[n] = S$ produces a sinusoidal signal (Sine) and $A_C \cdot y[n] = C$ produces another sinusoidal signal in quadrature with the first. If we choose $A_C = 1/sin(\omega_0)$ the two signals have the same amplitude.

Implementation To improve numerical accuracy we defined ϵ as $k/2^{(n\_bits)}$, this is done in practice by multipling by k and by bit shifting the result by n_bits bits to the left. We choose n_bits = 12 and with the choice of the sampling frequency f_S chosen for the ADC of $f_S = 128.84\,\text{kHz}$ dictates all the requirements to calculate k and A_C according to following equations:

$$\begin{aligned} k &= (1 - \cos(\omega_0)) \cdot 2^{(n\_bits)} \\ Ac &= 1/\sin(\omega_0) \end{aligned} \tag{3.6}$$

The results are presented in the table below:

frequency f	1.2 kHz	2.2 kHz
$\omega_0 = 2\pi f$	7.539	13.823
$\cos(\omega_0/f_S) \cdot 2^{(n\_bits)}$	4088.9	4072.4
$1/\sin(\omega_0/f_S)$	17.09	9.33
$k \approx (1 - cos(\omega_0/f_S)) \cdot 2^{(n\_bits)}$	8	24
$A_C \approx 1/\sin(\omega_0/f_S)$	17	9

Table 1: Numerical values for the parameters

Equation 3.2 can written in Verilog code as follows:

```
assign X1 = X - ((k * X) >>> n_bits) + Y;

always @(negedge CLK) begin
  Y <= X1 - ((k * X1 ) >>> n_bits ) - X;
  X <= X1;
end

assign C = Y * Ac;
assign S = X;
```

Listing 1: Oscillator code

A sample waveform output of the oscillators are plotted below:

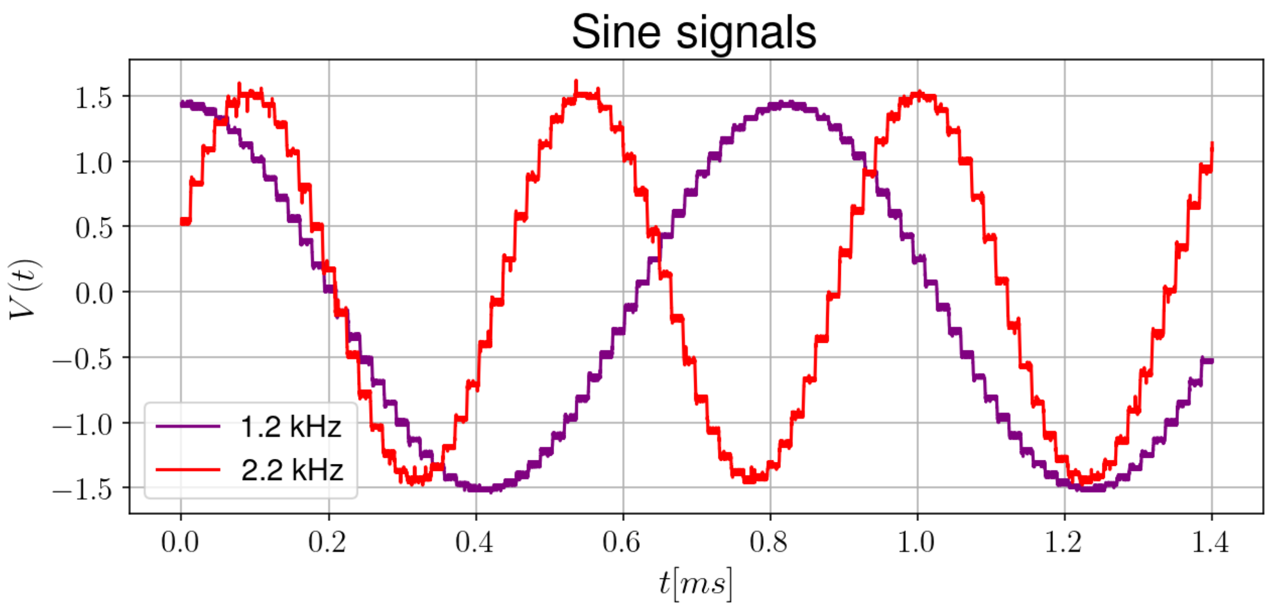


Figure 3.5: Oscillator waveform output

3.2 Low Pass Filter

To implement a low pass filter we start from transfer function of a first order filter in time domain:

$$\tilde{H} = \frac{2}{1 + is\tau} \tag{3.7}$$

By applying simulation theorem and by using bilinear transform $s \rightarrow \frac{2}{T}\frac{z-1}{z+1}$ we get:

$$V(z) = (1-c)\frac{z+1}{z-c} \tag{3.8}$$

Where $c = \frac{2\tau - T}{2\tau + T}$ this last constant can be written as $c = 1 - \epsilon = 1 - 2^{-k}$. Then if we perform inverse z-transform we get a recursive formula for the filter:

$$\begin{aligned} y[n] - cy[n+1] &= (1-c)\left(x[n] + x[n+1]\right) \\ y[n] &= (1-2^{-k})y[n+1] + 2^{-k+1}x[n] \end{aligned} \tag{3.9}$$

where in the last equation we have approximated $(x[n] + x[n+1]) \approx 2x[n]$. Then the last equation can be represented in the block diagram below:

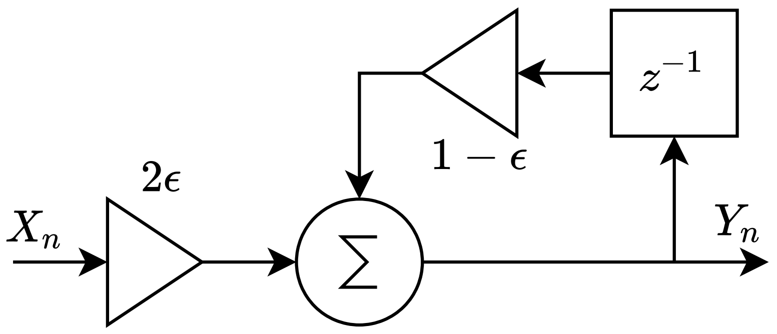


Figure 3.6: Filter block diagram

Implementation From the above equations we can see that cut-off frequency is given by:

$$\begin{aligned} \tau &= \frac{T}{2}\left(2^{k+1} + 1\right) \\ f_{3dB} &= \frac{1}{2\pi\tau} \approx \frac{2^{-k}}{2\pi} f_S \end{aligned} \tag{3.10}$$

Since the sampling frequency is $f_S = 128.84\,\mathrm{kHz}$ and k is chosen to be 6. The cut-off frequency of the filter is: $f_{3dB} = 320Hz$

The block diagram above can be translated in the following Verilog code:

```verilog
always @(posedge CLK) OUT <= OUT - (OUT>>>k) + (IN>>>(k-1));
```

Listing 2: Low pass filter code

4 Modulator

The modulator can be built using the same oscillator of section 3.1 by changing K and A_c according to the digital data input (k_1 and $A_{c,1}$ for 1.2 kHz signal and k_2 and $A_{c,2}$ for 2.2 kHz signal). If we want a constant amplitude at the output C we need also to consider equation 3.5 from which it follows that if we want constant amplitude when we change from 1.2 kHz to 2.2 kHz we must have:

$$y[n+1] = \frac{\sin(\omega_1)}{sin(\omega_2)} y[n] \tag{4.1}$$

and when we change from 2.2 kHz to 1.2 kHz we must have:

$$y[n+1] = \frac{\sin(\omega_2)}{sin(\omega_1)} y[n] \tag{4.2}$$

Implementation Using the same value of k_1, k_2, $A_{c,1}$ and $A_{c,2}$ as the oscillator, the two above values can be approximated as:

$$\frac{\sin(\omega_1)}{sin(\omega_2)} \approx \frac{29}{16} \qquad \frac{\sin(\omega_2)}{sin(\omega_1)} \approx \frac{9}{16} \tag{4.3}$$

The signal at the output is enabled with an input (DET). When DET is high the oscillator is working, when DET is low X and Y are reset to initial value. The output C of figure 3.4 is the sinusoidal output of the modulator.
The code produced is the following:

```
assign X1 = X - (( (DATA ? k2 : k1) * X) >>> n_bits) + Y;

always @(negedge CLK) begin
  if( (DATA != DATA_old)) begin
    DATA_old <= DATA;
    Y <= (DATA ? (Y*29)>>>4 : (Y*9)>>>4); //joint condition
  end else begin
    if(DET) begin
      Y <= X1 - (( (DATA ? k2 : k1) * X1 ) >>> n_bits ) - X;
      X <= X1;
    end else begin
      X <= 17'h0E800;
      Y <= 17'h00000;
    end
  end
end

assign S = Y * (DATA ? Ac2 : Ac1);
```

Listing 3: Modulator code

5 De/serializer

The data from the demodulator or to the modulator, is sent is serial way, this needs to be converted from or to parallel data (8 bits). Moreover we need to consider also the NRZI coding and the bit stuffing. The NRZI is encode a ”0” bit in a change of the DATA and ”1” if no change happen in a determined time. To do so a counter is needed *cnt*. When an edge in DATA is detected, *cnt* is reset to 0 and a ”1” bit is stored is the *DataReg*. If no edge is detected *cnt* value is increased until reaching 153. This value correspond to the length of 1.5 bits of data at $1.2 kbit/s$ of baud rate, using clock frequency of 128.84 kHz. The half a bit delay is used read the data at the half of the bit, this is done to prevent the detection of false edges or missing edges in the DATA signal. Then when *cnt* reaches 153, *cnt* is set to 51 (0.5 bit lenght) and a ”0” bit is stored is the *DataReg* if the bit stuffing conditions are fulfilled. The bit stuffing is implemented using a counter (*stuffedBits*), when a edge is detected (”0” bit) *stuffedBits* counter is reset to 0. Instead when a ”1” bit is detected (no change in DATA) *stuffedBits* counter is increased. When $stuffedBits < 5$ the next ”0” bit is written in *DataReg* else if $stuffedBits = 5$ the next edge is a synchronization one so needs not to be written. If $stuffedBits > 5$ the byte written is a flag bit, the data is then written but is marked as non valid (with the *VALID* register), moreover this flag byte can be used to synchronize the following data. This is done with the *bitCnt* that holds the number of bits received, when the 7th bit is received *CLK_out* is pull high to indicate that data is ready to be read otherwise is *CLK_out* low.
The flowchart diagram of the Deserializer is presented below:

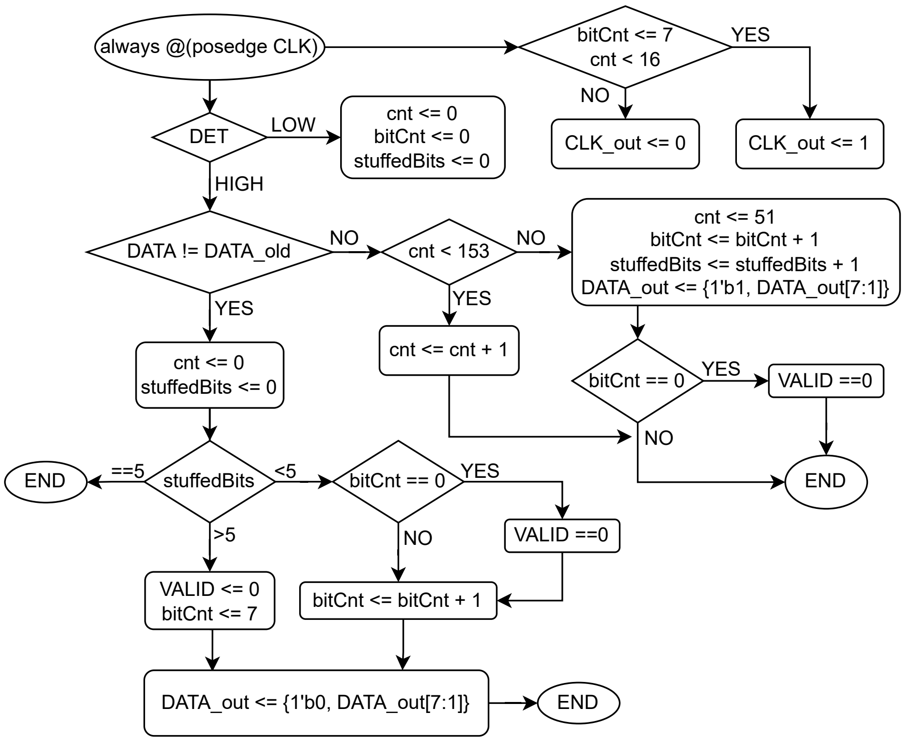


Figure 5.1: Deserializer block diagram

A similar strategy was also used for the serializer a block diagram of that is presented below:

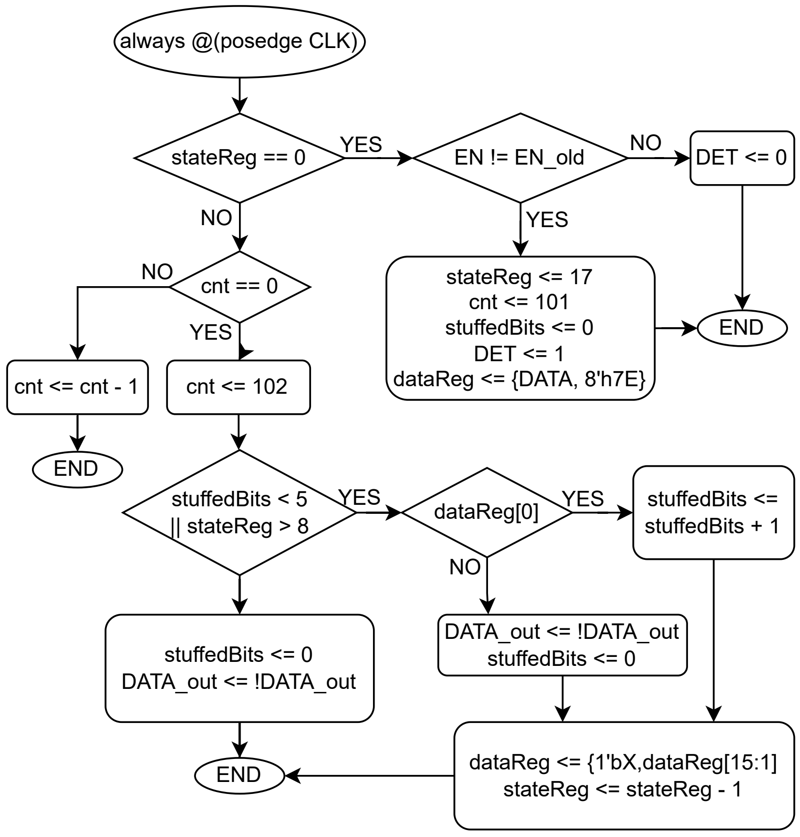


Figure 5.2: Serializer block diagram

In this case a register (*stateReg*) is needed to record the number of bit sent. The writing procedure is started at the rising edge on module input EN, when this happen the following things are done. DATA at module input is stored in *dataReg* register, preceded by a *8'h7F* byte (flag byte) this is done for synchronizing of the receiver. Counter register *cntReg* is set to 102, *stateReg* is set to 17 and DET output is pull high. On the followings clock (CLK) positive edges, *cnt* register is decreased once is $cnt = 0$ then a new bit is written and counter *cntReg* is restored to 102. When a new bit is wrote a bit is pulled out from *dataReg*. If this bit is high the output *DATA_out* is toggled and *stuffedBits* is set to zero. If the *dataReg* bit is low is performed a check of bit stuffing. If *stuffedBits* ≥ 5 bit "1" is added (*DATA_out* is toggled) else a bit "0" sent (o change in *DATA_out* value).

6 SPI comunication

The communication between FPGA and the ADC (LTC1407A), DAC (LTC2624) and preamplifier (LTC 6912-1) uses SPI comuication. SPI is a synchronous serial communication protocol. Data is trasported with the MISO (Master Input Slave Output) or MOSI (Master Output Slave Input) wires while clock is transported over the SCK wire. In the SPI is not present addressing features, so every device connected to the bus must be enable with an additional wire CS (Chip Select). Data length is not fixed by the protocol but depends on the device used, for example in our case we the DAC uses 24 bit, ADC uses 32 bits and preamplifier 8 bits of data. The bits of data are read one by one at every rising edge of SCK while CS is held in the active state.

Implementation The development board that we have used has MOSI output of the FPGA in common with the DAC and the preamplifier, so it was not possible to communicate at both

in the same time. To solve this issue it was necessary to use the CS wire that are one for every device. The chosen option was to write the required gain value at the preamplifier some milliseconds after startup, then set the CS outputs so that communication only happen with the DAC. The communication with the DAC is controlled by the homonym module, while communication with preamplifier is controlled by module GAIN. Since both module controls MOSI output an intermediate module (SPI) was needed to tell which module should control the output. The control of the ADC (reading of the data) is done by the "ADC" module. The clock SCK is in common with the three peripherals and is controlled by the SPI module.
Every module (ADC,DAC and GAIN) uses different code, but we can give a general scheme of working: Data is read or wrote by the FPGA at the falling edge of the SCK line, this was done to allow data to change giving room to the required falling and rising time of the signals. At every falling edge of the SCK data is written or read from the last bit of a shift register, after that data is shifted by one place to make room for the next bit. The number of bit sent or received are stored in a register *stateReg*, this also controls the SC bit to synchronize the FPGA and the peripheral and also to clock the parallel data to or from other modules of the code.

7 Serial port

Comunication with the computer is done with serial port with baud rate of 9600 bit/s. Serial port communication is an asynchronous serial communication protocol. In our case data bits sent are 8 and one start and stop bit are added. The line is held high during no transmission, when transmission start the line is pulled low for a bit length, then the 8 bits are sent. As last a one bit length stop bit (high bit) is set. A serial data sample is presented below:

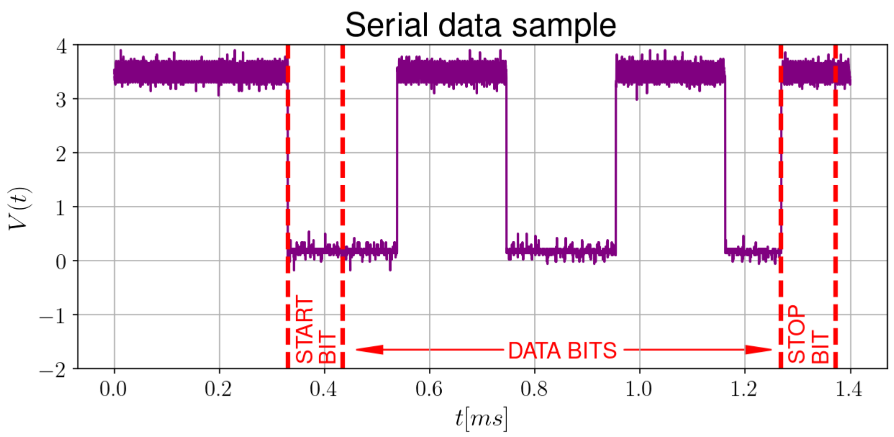
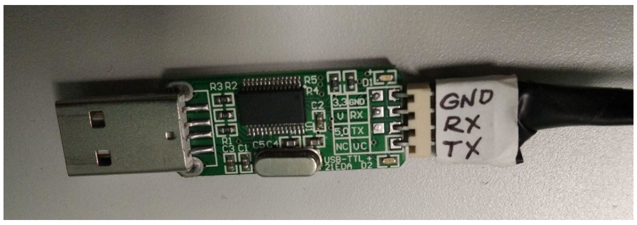


Figure 7.1: Serial data

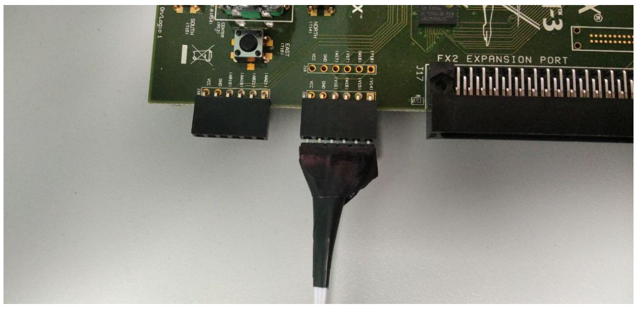
The *SERIAL_RX* and *SERIAL_TX* modules are used to receive and transmit data over serial protocol. The scheme to decode/encode the data is similar to what done for the SPI. A shift register *DataReg* is used to store the received data or to be transmitted. The register *StateReg* record the number of bits sent or received including the START and STOP bit. The data is read or written taking care of the delays required to get the wanted baud rate with the register *cntReg*.

8 Hardware description

To test the code, two hardware conponents was used. A generic USB to serial converter, a possible alternative can be: https://www.adafruit.com/product/5335Adafruit CP2102N Friend - USB to Serial Converter. The converter is connected to the board with three wires: *TX*, *RX* and *GND*.



(a) USB to Serial converter



(b) USB to Serial converter, connection to the board

To record the audio produced from the modulator or to send audio to the demodulator a BNC to 2.5mm jack connector adapter board was built. The board also include a DC decoupling capacitor, a trimmer to attenuate the signal and a two way jumper to select the right or left channel of the jack connector.

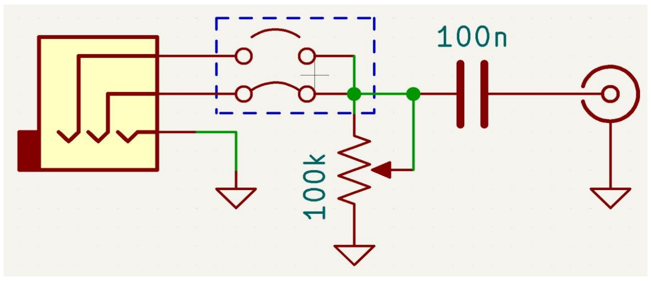
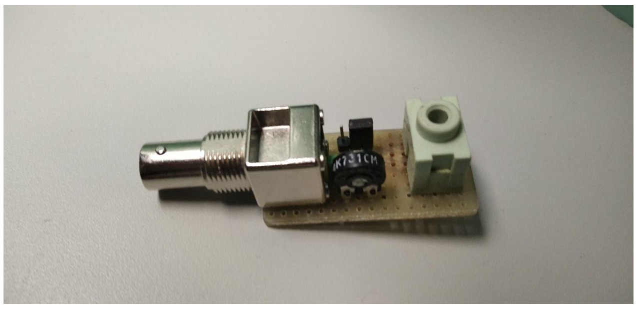
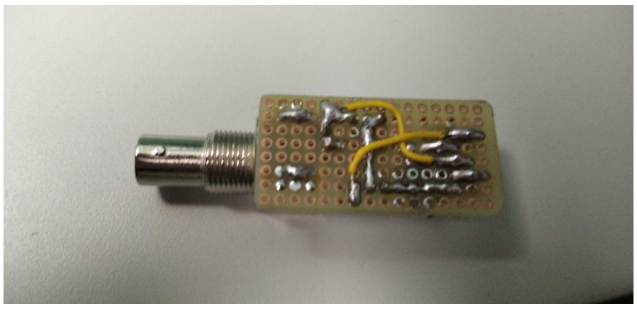


Figure 8.2: adapter board schematic



(a) adapter board front



(b) adapter board back

9 Testing

The modem was tested by sending serial data from the computer using the software Tio: https://github.com/tio/tio that is a simple serial terminal, using the baud rate $9600 bit/s$ and delay between characters of $20ms$.

```
tio -b 9600 -o 20 /dev/ttyUSB0
```

This was needed beacause the Bell202 uses a baud rate of $1200 bit/s$ slower than the Serial port, so we need to wait that the byte to be all written before sending another one.
The first test done was to send characters with Tio trough the serial port. The serial data trough the USB to serial converter is sent to board where is modulated. Then the audio signal exiting form the DAC was bring to the ADC with a coaxial cable (with BNC connectors). The incoming data is then received and demodulated, then sent to the serial port, where the computer (Tio) is listening. If all the steps are correct the the key on the PC pressed should correspond to the character seen on the screen.
The second test performed was similar to the first, but we recorded the signal outgoing from the DAC with the audio input of a PC with the help of the adapter board (figure 8.2). In a second time the audio was sent at the ADC, also in this case with the adapter board, the result of the demodulation was observed on the PC (Tio).
In both cases the Modem was able to modulate and demodulate the signal correctly, showing its correct behaviour.

References

[1] *Wikipedia.* [Online]. Available: https://en.wikipedia.org/wiki/High-Level_Data_Link_Control

[2] *www.embedded.com.* [Online]. Available: https://www.embedded.com/bit-stuffing/

[3] J. Nam, *A Study of Sinusoid Generation Using Recursive Algorithms.* [Online]. Available: https://ccrma.stanford.edu/~juhan/pubs/jnam-emile05.pdf