RTTY: an FSK decoder program for Linux.

Jesús Arias (EB1DIX)

23rd February 2003

Contents

Chapter 1

rtty-2.1 Program Description.

1.1 What is RTTY

1.1.1 The RTTY transmissions.

RTTY is a program that listen to your sound-card audio input and tries to decode FSK modulated signals. These signals can be found in HF bands and are commonly known as Radio-Tele-TYpe (RTTY).

In RTTY signals a given frequency is used to transmit a logic “1” and a different frequency is used for the logic “0”. We call these frequencies MARK & SPACE frequencies respectively. The data format is those found in asynchronous serial communications: Every character starts with a START bit with “0” logic value, then follows the data bits, LSB first, and finally one or two STOP bits with a logic level “1” are added to the character. Inter-character time is filled with a logic “1”. The character is often 5 bits long, and uses an old encoding called BAUDOT, but sometimes a plain ASCII encoding is used. in this case the character is 7 or 8 bits long, and one optional parity bit can be added in order to check the received data integrity. The data rate is slow. Typical values range from 45 to 300 baud.

1.1.2 The program.

The RTTY program, version 2.1 is not a complete “closed box” solution with all the features you think are needed for an easy reception. Instead, the program has been designed with an academic purpose in mind, and allows people to look what’s happening inside during reception.

In addition to RTTY, the program can also decode CW (Morse) signals and AX25 packets. The code for these modes is still in a very experimental phase. Other data formats, like AMTOR, PACTOR, Weather data, etc. are not supported. In fact the program is able to recover those bit-streams, but i don’t know very much about those data formats.

The program gives you a graphical interface that shows several signal traces in a window that resembles an oscilloscope screen. This window helps to tune the incoming signals and also to identify some of its characteristics.

1.1.3 System requirements.

In order to run rtty-2.0 in your computer you will need at least:

- An Intel Pentium or compatible processor. The program requires very little CPU power. In a 166 MHz Pentium it consumes only a 5% CPU time with a sampling rate of 44100 Hz. With lower sampling rates (11025 Hz is the recommended minimum) the CPU use is negligible. This suggest that the program can run in older computers, but this was not tested. Anyway, a hardware FPU is needed.

- A sound-card. Tested sound-cards are SoundBlaster-16 and SiS-7018 AC’97. The first one is an old card and has an stable driver. The SiS-7018 is embedded in a laptop chipset. Its driver is still buggy.

- Linux Operating System. Of course.

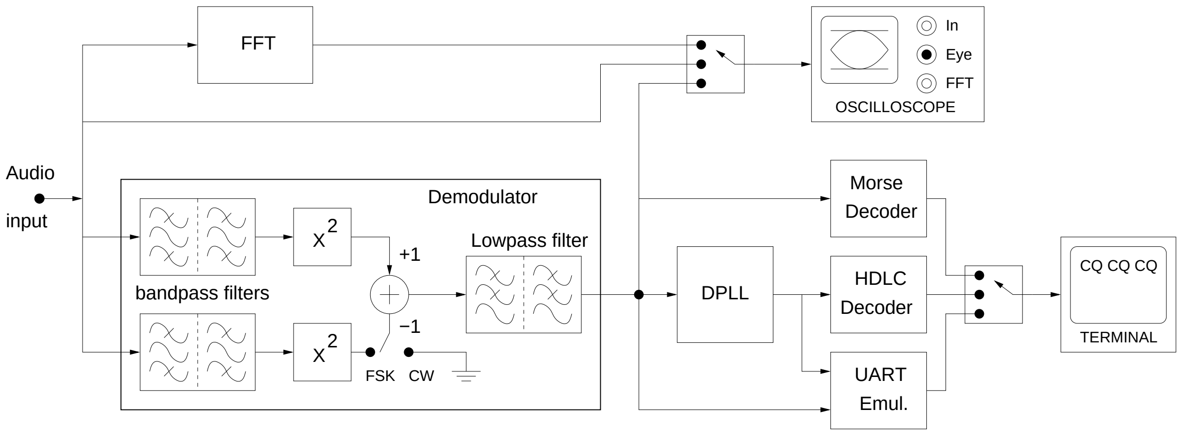


Figure 1.1: Block diagram of the rtty program.

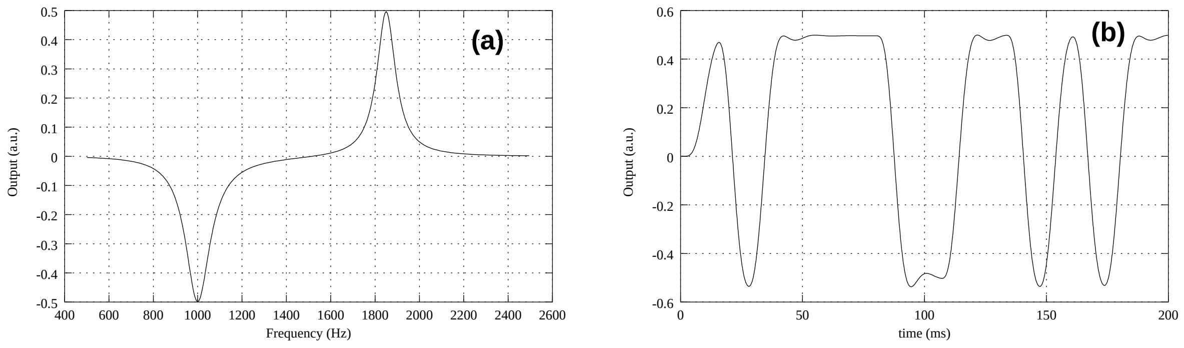


Figure 1.2: Demodulator response for a 1000Hz, 1850Hz, 75 baud configuration. (a) Frequency response, (b) Time response to a pseudo-random bit stream.

- A sound-card driver. This driver can be compiled into the Linux kernel or can be inserted as a module. As previously stated, some drivers are buggy or lack some functionality. The SiS-7018 driver only allows recording at 48 kHz sampling rate, and some “ioctl” calls are not implemented. As a result, the oscilloscope window shows a poor animation.
- An X-window server for the graphical interface.
- An ANSI color terminal for the decoded output.

1.2 How the program works.

1.2.1 General decoder structure.

Figure 1.1 shows the block-diagram of the rtty program. It includes an FSK demodulation block, an UART emulator block, a character terminal and some monitoring blocks.

The demodulator section takes audio samples as the input and generates a “0” or “1” logic level at the output depending on the frequency of the input signal. This section is mainly based on digital filters, and works as follows:

The input signal is filtered through a two bandpass filters. One filter is tuned to the MARK frequency while the other is tuned to the SPACE frequency. As a result, two separated bands are obtained. Then, the “rms” amplitude of each band is computed. This is achieved by squaring the band signals and then low-pass filtering the result. The two “rms” amplitudes are then subtracted and the sign of the result is the logic output value. In the actual implementation the subtraction and the low-pass filter are swapped, and only one low-pass filter is needed. The response of such section is shown in figure 1.2.

The bandpass filters are second order “biquads”, with a Q adjusted to obtain the desired bandwidth. Two identical band-pass filters were connected in series in each branch in order to improve the out-of-band attenuation. The low-pass filter is implemented by two biquadratic sections, yielding an stopband attenuation of 80 dB/dec. The bandwidth of all the filters is proportional to the data rate.

Following the demodulator are the decoder blocks. The DPLL section generates a clock signal synchronized with the bit stream. Then, for RTTY mode, the data bits are assembled in the UART emulator according to settings to obtain characters. These characters are checked against errors and then printed to stdout. If the data length is 5 bits a BAUDOT encoding is assumed, and therefore the characters are translated to ASCII before its printing. For AX25 packets the processing is more complicated. This includes the detection of flag delimiters: the character that signals the beginning and end of packets, the removal of stuffed zeroes and CRC check. When a packet with good CRC arrive its AX25 address field is decoded and printed followed by a hexadecimal dump of the whole packet contents. The CW mode is quite different. Only one frequency band is used, the comparator threshold has to be positive and the data is encoded on the mark length instead of the levels.

1.2.2 The Digital filters.

The filters used in the RTTY program are of the IIR type. IIR means Infinite Impulse Response, which in turn, means that the filter is implemented using some kind of feedback. The general IIR filter algorithm is:

$$y_i = a_0 x_i + a_1 x_{i-1} + \dots + a_n x_{i-n} + b_1 y_{i-1} + \dots + b_n y_{i-m} \tag{1.1}$$

Where x_i is the current input sample and y_i is the current output sample. As it can be seen, the current output sample depends on the previous output samples, so here is the feedback.

The IIR filters are computer efficient, but they are very sensitive to coefficient accuracy and can become unstable. To avoid instability, the order of the filter must be low. Higher order filters are implemented by cascading lower order sections. In this program the filter section is order 2 and it is commonly called “biquad”. The filter behavior is dictated by its own coefficients, and the calculation of these coefficients are the purpose of the digital filter design.

The filter design starts with a continuous-time version of the biquad filter section, the resonant frequency, ω_0, is “prewharped” to account for the further frequency response distortion. Then the Bilinear transform is used to translate the continuous-time filter to a discrete-time version of the filter, where its transfer function is written in terms of the Z transform. Finally the filter coefficients are derived from the discrete-time transfer function.

In the following example a bandpass resonator is designed. The filter parameters are its resonant frequency ω_0, and its quality factor Q. In order to account for the error introduced by the Bilinear transform, the resonant frequency is prewharped, so the used resonant frequency is: $\widehat{\omega}_0 = 2 f_s \tan\left(\frac{\omega_0}{2 f_s}\right)$, where f_s is the sampling frequency. The continuous-time transfer function, written in the Laplace transform terms, is:

$$H(s) = \frac{\frac{1}{Q}\frac{s}{\widehat{\omega}_0}}{\frac{s^2}{\widehat{\omega}_0^2} + \frac{1}{Q}\frac{s}{\widehat{\omega}_0} + 1} \tag{1.2}$$

Then, we use the Bilinear transformation to obtain an approximate discrete-time equivalent transfer function. The Bilinear transform is merely the replacement of “s” by “$2 f_s \frac{1-z^{-1}}{1+z^{-1}}$” in equation 1.2. The new transfer function is then:

$$H(z) = \frac{\frac{2\alpha}{Q}(1 - z^{-2})}{(4\alpha^2 + \frac{2\alpha}{Q} + 1) + (2 - 8\alpha^2) z^{-1} + (4\alpha^2 - \frac{2\alpha}{Q} + 1) z^{-2}} \tag{1.3}$$

where $\alpha = \frac{f_s}{\widehat{\omega}_0}$

Then, the recurrence algorithm is derived taking into account that $H(z) = \frac{Y(z)}{X(z)}$, and that z^{-n} is represented in the time domain by an n clock cycle delay. From equation 1.3 we obtain:

$$y_i = \frac{\frac{2\alpha}{Q}}{4\alpha^2 + \frac{2\alpha}{Q} + 1}(x_i - x_{i-2}) + \frac{8\alpha^2 - 2}{4\alpha^2 + \frac{2\alpha}{Q} + 1} y_{i-1} - \frac{4\alpha^2 - \frac{2\alpha}{Q} + 1}{4\alpha^2 + \frac{2\alpha}{Q} + 1} y_{i-2} \tag{1.4}$$

And finally, by matching equation 1.4 with equation 1.1, the filter coefficients are easily obtained.

These bandpass resonators are used to split the signal into a mark & space bands in the very front-end of the demodulator, but the demodulator also includes a low-pass filter, which is divided into two biquad sections giving a total stopband attenuation of 80 dB/dec. Each section has the following transfer functions:

$$H(s) = \frac{1}{\frac{s^2}{\hat{\omega}_0^2} + \frac{1}{Q}\frac{s}{\hat{\omega}_0} + 1} \tag{1.5}$$

$$H(z) = \frac{1 + 2z^{-1} + z^{-2}}{(4\alpha^2 + \frac{2\alpha}{Q} + 1) + (2 - 8\alpha^2)z^{-1} + (4\alpha^2 - \frac{2\alpha}{Q} + 1)z^{-2}} \tag{1.6}$$

And the recurrence algorithm is:

$$y_i = \frac{1}{4\alpha^2 + \frac{2\alpha}{Q} + 1}(x_i + 2x_{i-1} + x_{i-2}) + \frac{8\alpha^2 - 2}{4\alpha^2 + \frac{2\alpha}{Q} + 1}y_{i-1} - \frac{4\alpha^2 - \frac{2\alpha}{Q} + 1}{4\alpha^2 + \frac{2\alpha}{Q} + 1}y_{i-2} \tag{1.7}$$

1.2.3 Bit recovery.

The DPLL block accounts for the clock recovery from the demodulator output. But currently no clock signal is generated. Instead the DPLL routine returns when the demodulator output shows a valid bit. A new call to the DPLL routine does not return until a new bit has arrived.

The DPLL routine uses the “bit-phase” concept. The bit-phase is an static 16 bit integer variable that can be incremented in a circular fashion. For each demodulator sample the bit-phase is incremented an amount that is related to the data rate and the sampling frequency. When the bit-phase overflows the DPLL routine returns.

In order to get the bit-phase variable synchronized with the incoming bits some feedback is needed. In our scheme the demodulator output must change only when the bit-phase is around 0 or 65536. When bit change is detected after 0 (late change) the bit-phase is advanced 1/32 of its total value, and if the bit change takes place just before 65536 (early change) the bit-phase is delayed.

During the duration of the bit the incoming signal from the demodulator is integrated. At the end of the bit time we returns a logic one if the signal integral is positive or zero if it is negative.

1.2.4 Character recovery (UART emulator).

The asynchronous serial data needs also a word synchronization. We wait for a falling edge in the demodulator output that signals the beginning of a START bit. Then we reset the bit-phase value and we call the DPLL routine. If the returned bit is one we keep waiting for another START bit.

When a valid START bit is detected we call the DPLL routine one time for each following bit. The data bits and parity bit are recovered in this way. One stop bit can also be retrieved calling the DPLL routine, but the last stop bit is read directly from the demodulator after half a bit delay.

The recovered parity and STOP bits are checked against errors and the character is marked as erroneous if this check fails. The character recovery routine waits until all the STOP bits are read as “1” allowing for a more probably good START bit in the following character.

1.2.5 AX25 packet mode (HDLC).

The HDLC mode is a synchronous mode. Therefore, all bits are recovered by calling the DPLL routine. HDLC data uses a NRZI coding. That means that the actual data is encoded in the bit transitions instead of bit levels. A 0 bit is encoded as a level change and a 1 as a no change. One interesting property of this scheme is that an inverted bit stream give the same data as the original, so, the mark and space bands can be swapped without affecting the received data. The NRZI stream is converted to a typical NRZ data by XORing each bit with its inverted predecessor.

The packet starts with a flag delimiter: “01111110” and ends with the same flag delimiter. The packet length is not known until the whole packet is received. In order to avoid unexpected flag characters in the data, the bit stream is “stuffed” with zeroes before transmission: Each time 5 consecutive ones are transmitted, a zero bit is inserted in the bit stream.

The program first waits for a flag character and keeps recording data until another flag character is received. If 5 consecutive ones are received the next zero is removed. If the next bit is one, it can be a flag character, so an additional bit is read. If this bit is zero a flag was received and the packet ended. If this bit is one we have an error for sure and the packet is discarded.

After receiving a packet its CRC is checked. If the CRC test fails the packet is discarded. If the CRC is correct, the packet is printed on stdout. The AX25 address field is decoded to show the IDs of the sender and destination stations. The data payload is dumped in hexadecimal and ASCII.

1.2.6 CW mode (Morse code).

For CW (Morse code) signals only the MARK band-pass filter is used. The resulting signal is compared to a variable threshold level that depends on the average signal amplitude. Then the length of marks is recorded and each mark is translated to a dot or a dash depending on this length and the “keying time” settings. Long spaces are interpreted as symbol delimiters, and “really long” spaces as the “space” character. The received symbols are translated to ASCII via a lookup table and printed.

I must reckon this mode is a little tricky to tune up. Things are still worse because most CW signals are keyed by human operators whose timing accuracy is far from ideal (a lot of phase noise :). Another problem arises from the 100% AM modulation nature of CW signals that can generate nasty transients in the AGC loops of receivers. This problem is worse for strong signals. It is ironic that weak signals are decoded more reliably than strong ones. In spite of all these problems the program is still useful for unexperienced Morse listeners like me:)

Chapter 2

User's Manual.

2.1 Monitoring and Control Form.

Figure 2.1 shows the window that is opened after the program start. The window is divided in two parts. The lower part includes several controls that allows us to select the mode, data format and rate. The data rate can be selected from a list of commonly used data rates, or it can be entered directly by typing it in the input filed. The mark and space frequencies can be adjusted using four counter-buttons. The frequencies can be changed in steps of 10 Hz or 100Hz. The bandwidth of the filters is made equal to the data rate. The dot-length for CW reception can be adjusted manually using a slider, or automatically clicking on the "ATIM" button. The dot-time units are somewhat arbitrary because they depends on the sampling rate of the sound card. For a 12kHz effective sampling rate 1 dot-time unit is 5.3 ms. For example, the "QSO" word will take 44$\times$dot-time $\times$5.3 ms to receive, or 2.3 seconds for dot-time=10.

Not all settings are required for a particular mode. The number of data bits, stop bits and parity only makes sense for RTTY mode. For CW mode the SPACE frequency is also irrelevant.

The upper part of the form is a display that can plot several traces. This display is off after start or after clicking in the "DISP OFF" button. The figure 2.1 shows the power spectrum of the input signal. The vertical axis is logarithmic, and shows 80 dB range. The plot also shows two rectangular bands that corresponds to the selected mark and space frequencies. The space frequency band is red while the mark frequency band is green. The width of these bands is the same as the filter bandwidth. As it can be seen in this plot, a properly tuned receiver generates a signal spectrum that shows two peaks centered around the two filter frequencies. The width of the peaks is also close to the bandwidth of the filters and proportional to the data rate. In CW mode only the mark frequency band is displayed .

Several other different signals can be plotted on the oscilloscope display. Figure 2.2 shows the display snapshots for "INPUT", "DEMOD" and "EYE" selections. When "INPUT" is selected the display trace is composed of raw input samples. The horizontal scale is 1 pixel/sample.

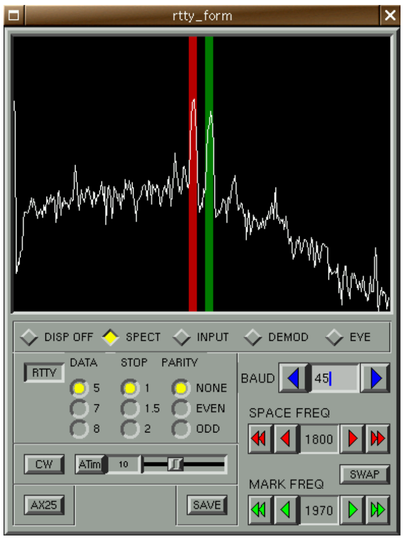
If "DEMOD" is selected, the display shows the signal after demodulation. The total time displayed is 10 times one bit duration, and therefore it depends on the data rate.

Finally, if "EYE" is selected, the display shows several traces, all of them of only one-bit duration. This graph can be useful in determining the correct data rate and signal quality.

2.2 Text-only mode.

The graphical interface can be disabled if the "-x" option is given in the command line arguments. Under these circumstances the program setup can only be entered via configuration file (see section 2.3). This mode can be useful for batch execution or when an X-window terminal is not available.

Note that even in text mode the program needs to link the "libX11.so.6" dynamic library. If you don't want to use the X-window environment nor library at all you must edit the "Makefile" file and recompile the source code.



[DISP OFF] [✓ SPECT] [INPUT] [DEMOD] [EYE]

DATA: [✓ 5] 7 8

STOP: [✓ 1] 1.5 2

PARITY: [✓ NONE] EVEN ODD

BAUD 45

SPACE FREQ 1800

MARK FREQ 1970

10

Figure 2.1: rtty form with power spectrum display on.

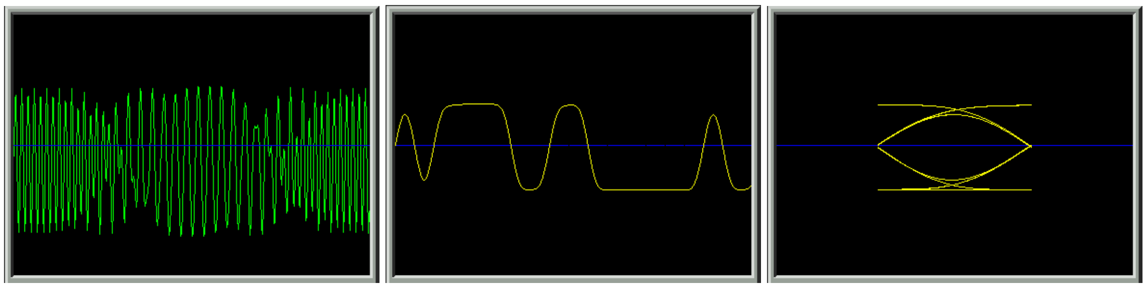


Figure 2.2: Oscilloscope display for input, demodulated signals and eye plot.

2.3 The configuration file.

When the program starts it tries to read the mode, data format, data rate and demodulator tuning from the "./rtty.cfg" file. This file must be located in the current directory. This is a text file like:

```
#-------- Modes: 0=RTTY, 1=CW, 2=HDLC -------
MODE=2
#
SPACEF=1050.000000
MARKF=1200.000000
DR=300
NBIT=8
NSTOP=1.000000
#-------- parity: 0=none, 1=even, 2=odd ------
PARITY=0
CWT=9.854756
```

As it can be seen, each line contains a VARIABLE=VALUE statement. The equal character must be always present and no spaces are allowed in the line. Comment lines start with the grid (#) character.

A file like this, with the current settings, is generated when clicking in the "SAVE" button.