Low Power RTTY and PSK31 Decoder for Ham Radio Applications

Ben Bales

ECE400

12-5-2010

Abstract: This paper describes the basics for implementation of RTTY and PSK31 decoders in low power ham radio applications. The decoders within are based on a modified time-delay tanlock loop (TDTL) and code and schematics are provided for operation on a Microchip dsPIC. Performance-wise, in the presence of noise the PSK31 decoder makes it useless and the RTTY decoder is only mediocre, but the code should still serve as a very good start for developing low-milliwatt decoders in the future.

Thanks goes to Dr. Mongi Abidi of the University of Tennessee for providing support for this project, the Spirit of Knoxville Balloon Team for providing inspiration, and Saleh R. Al-Araji, Zahir M, Hussain, and Mahmoud A. Al-Qutayri for writing a very nice book on TDTL operation (Digital Phase Lock Loops, Springer, Dordrecht, The Netherlands. 2006)

Index

1. Introduction

Currently, most digital Ham Radio modes are confined to computer operation. However, as most of these modes are simple and can be implemented on modern, cheap, low power processors. Unfortunately, most commercial communication developments are focused towards more profitable fields, and there is a significant gap in embedded ham radio digital communications hardware.

As of the writing of this report, only one other solution exists for decoding PSK31 and RTTY signals without a computer; unfortunately, it is expensive ($150) and requires many hundred milliwatts to operate. This paper shows the possibility of creating a much cheaper demodulator at a much lower power point.

Largely this project was inspired by the trans-Atlantic Spirit of Knoxville Balloon project. In this, the group wanted a way to transmit message to the balloon. However, while the group had sufficient embedded development experience to handle the incoming information intelligently, they did not possess the necessary digital signal processing expertise to implement their own demodulators. This project was originally posed as a way to give the group a black-box receiver they could add to their boards and interface with their existing software infrastructure. As such, more effort is placed on defining a portable software interface rather than defining a comprehensive hardware platform. This software should be useful for people developing small radios as well, as it will make it possible for them to embed receivers directly in the equipment rather than depending on computers.

2. Digital Modes

To maximize the usefulness of this project to the community, RTTY and PSK31 were chosen as the modes this digital demodulator should support. From a practical perspective, RTTY and PSK31 are the most widely used digital modes currently used. However, implementing these demodulators would also be very useful from a theoretical perspective. RTTY is a very simple binary frequency shift keying (BFSK) modulation scheme, and PSK31 is a very simple binary phase shift keying (BPSK) modulation scheme. FSK and PSK are very common modulation schemes for digital modes, and implementing the simple forms of these is key to implementing the more complex forms. Of note, audio shift keying (ASK) is also used (most significantly in continuous wave (CW) Morse code transmission). However, no ASK demodulator was written for the sake of simplicity. It could certainly be added though.

a. PSK31

PSK31 is a binary phase shift keying mode introduced by Peter Martinez in December 1998 (Martinez). Official documentation is available online at http://aintel.bi.ehu.es/psk31.html.

The mode will be outlined very simply here, though interested persons should reference the above web page for more accurate and comprehensive information (especially with respect to PSK63, a derivative mode using quadrature-PSK modulation).

PSK31 is a binary phase-shift keying mode and as such uses a single sinusoid with alternating 180 degree phase reversals modulated at 31.25hz to encode information. A phase reversal symbolizes a zero and no phase reversal symbolizes a one. To avoid nasty noise, the phase reversals are applied slowly. Traditionally, PSK modulation would look like:

$$y(t) = phasemessage(t)\cos(2\pi F t)$$

In this case, phasemessage(t) is a square wave switching back and forth between -1 and 1 whenever phase reversals are necessary at 31.25hz. However, this creates quite a bit of unnecessary noise, so PSK31 applies the phase reversals like cosine waves:

$$y(t) = \cos\left(\frac{2\pi}{31.25}\int_0^t message(t)\,dt\right)\cos(2\pi F t)$$

In this case, message(t) is the logical inverse of the actual binary symbol encoding (as opposed to phasemessage(t) above, a message(t) equal to 1 would represent a 0 in the symbol encoding and would cause a transition from -1 to 1 or 1 to -1 in phasemessage(t)).

Symbols are encoded in a custom form called Varicode. In this, no two consecutive zeros are allowed in in the symbol. In this way, if two sequential phase reversals (or zeros) are encountered, it can be assume that the character has ended. This modulation scheme lends itself to a variable length alphabet, with space being one bit and the others taking many more.

As an example, symbol[n], phasemessage(t), message(t), and y(t) are plotted in Figure 1 for encoding the letter ‘a’, symbol == ‘1011’, preceded and followed by end of character symbols, ‘00’. For this sample, F is set to 150hz for simplicity (phase reversals are still sent at 31.25hz though).

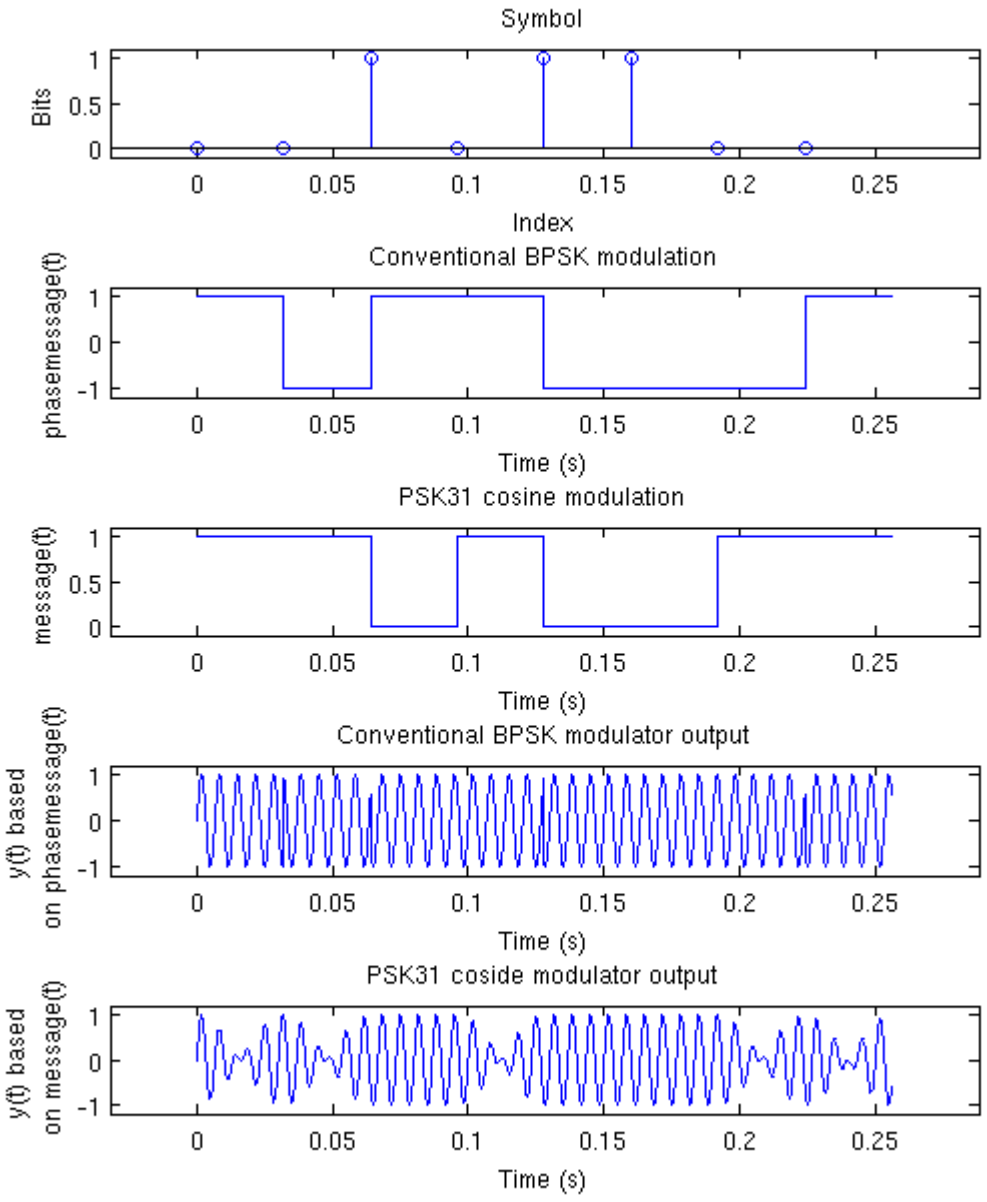

Figure 1: PSK modulation examples

b. RTTY:

RTTY stands for Radio Teletype and is much older than PSK31. Some accounts place Radio Teletype FSK systems in use as far back as 1936 (Bartlett). Naturally, RTTY systems

have evolved and changed with technology, and, as a result, today there is really no set standard. This implementation tries as much as possible to match the MTTY implementation, MTTY being one of the most commonly used computer RTTY modulation programs. Anyone interested in using the demodulator should understand this section fully to avoid further confusion later.

Two pages, one by George W. Henry Jr. (http://www.digigrup.org/ccdd/rtty.htm) (Henry) and one by Donald A. Hill (http://www.aa5au.com/gettingstarted/rtty_diddles_technical.htm) (Hill) were especially useful in development of this code. However, again, there is no standard RTTY, and so these can only be used as vague guides. This section will describe as accurately as possible the RTTY modulation scheme used in this decoder.

RTTY, as implemented by default in MTTY, is a 45.45 baud binary FSK, with the lower frequency, the ‘mark’ representing a ‘1’, and the higher frequency, the ‘space’, representing a ‘0’. Most often, the low frequency is 1275hz and the high frequency is 1445hz. This is often quoted as the mark frequency of 1275hz with a shift of 170hz. Again, this is not standardized, and sometimes the mark is the high frequency and the shift references a shift down. In short, if a control signal ‘message(t)’ is created from the bits in the encoded message, output looks like:

$$y(t)=\sin\left(2\pi\int_0^t\left(\left(Fshift\ message(t)\right)+Fmark\right)dt\right)$$

Every character is followed by start and stop bits. In this decoder, one start bit and two stop bits are used. By default in MTTY, one start bit and 1.42 stop bits are used (1.42 being represented by a tone 1.42 times the length of a single stop bit).

A variation of the Baudot code is used by MTTY for symbol coding. This is a two state, five bit code. Each of the five bit characters can represent one of two things, depending on whether the decoder is in Letter or Figure mode. For instance, ‘10000’ means ‘R’ in Letter mode and ‘4’ in Figure mode. Naturally, this duplicity can result in errors. If the string ‘R4R’ is to be sent, the RTTY transmitter will have to send (and the modulator will have to receive) ‘R’, Set

state to Figures, ‘4’, Set state to Letters, ‘R’. If the decoder misses the first set state, the message will be interpreted as ‘RR4’, or, if the second set state is missed, the message will be interpreted as ‘R44’. It can be very problematic to reception if a false state switch is received or one is missed because the decoder will need to either miss another state switch or receive another bad one to get back on track without some other intervention mechanism. As of now, the decoder includes no logic for automatically correcting itself. Some characters like space, state set commands, and line feed are standard between the two modes so that these duplicity errors will not effect them.

Figure 1 shows an example of modulation of ‘A’ (‘11000’) with one start bit (‘0’) and two stop bits (‘11’). For this sample, Fmark is set to 200hz and Fspace 400hz for simplicity (or Fmark is 200hz and Fshift is 200hz).

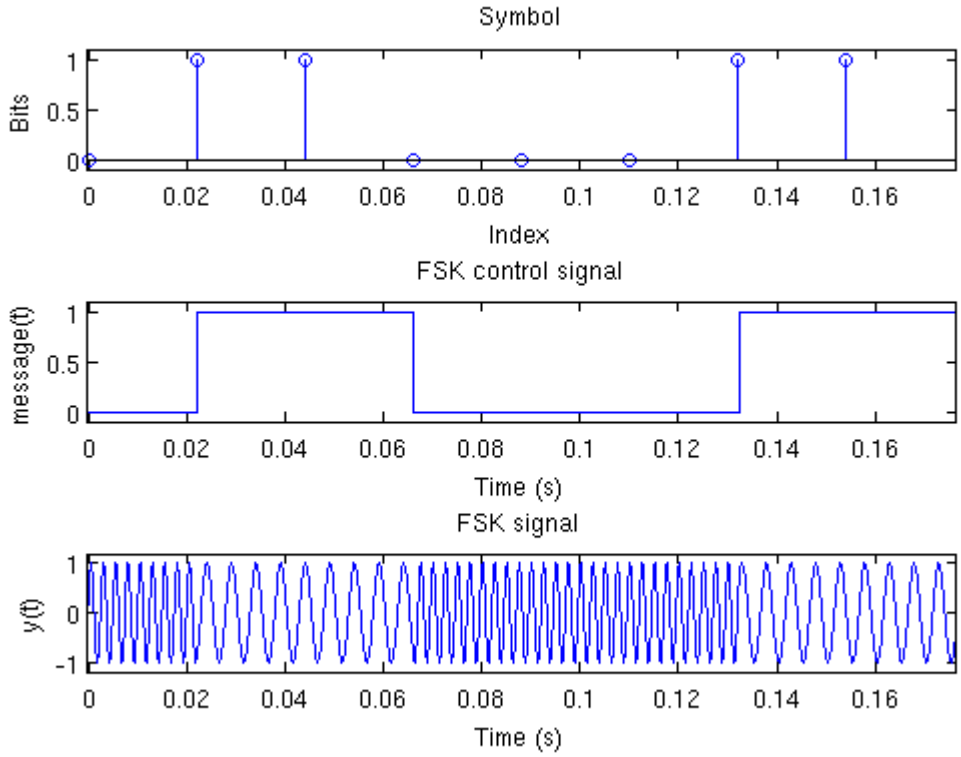

Figure 2: RTTY modulation example

3. Demodulator Design:

This section lays out the theory and implementations of the RTTY and PSK31 demodulators. In both cases, demodulation is handled based on the error signal of a digital phase-lock loop. As this error signal is updated, the RTTY and PSK31 decode schemes use hardware timers to detect incoming symbols, which are accumulated until valid messages are received.

a. Time-delay tanlock loop

All of the information for this section on time-delay tanlock loops was taken from *Digital Phase Lock Loops* by Saleh R. Al-Araji, Zahir M. Hussain, and Mahmoud A. AL-Qutayri (Al-Araji, et al).

For an ideal tanlock loop (TLL) a real signal is sampled and split into two. The lower half is fed immediately into one input of a four quandrant arctangent while the upper half is delayed by 90 degrees and then fed into the other input of the four quandrant arctangent. The error signal produced is used to adjust the sample rate.

In this way, if the phase difference between the signals is anything other than 90 degrees, it can be asserted that either the input signal has incurred a phase shift (due to PSK modulation) or frequency of the input signal has been modified (as in FSK modulation).

FSK signals can be detected by measuring the value of the steady-state phase difference signal. From this value, the frequency of the input signal can be determined.

PSK signals can be detected by integrating the phase difference signal. Across a 180 degree phase shift in a BPSK signal, the integral of the error signal will produce a 180 degree step.

In all reality, while it would be possible to implement the 90 degree phase delay with a Hilbert transform, these are expensive in practice and amount to very high order FIR filters. In that case, the tanlock loop will converge for the entire range of frequencies that the Hilbert transform is defined for. It is possible to imagine that if there is only one frequency on the input, then the Hilbert transform could be reduced to a simple time delay that corresponded to a 90 degree phase shift for that single frequency. It turns out that this simple time delay will allow the loop to converge for a small range of frequencies around that frequency the time delay is designed for. This modification to the TLL is called a time-delay tanlock loop (TDTL) and is useful in this application because it significantly reduces computational overhead of signal detection. PSK31 only uses one frequency (and sends information by applying phase shifts to that signal), and the frequency shift in RTTY is small (170hz) compared to the mark and space frequencies (1275hz and 1445hz), which makes this a valid simplification for this application.

In conclusion, the TDTL looks something like Figure 3.

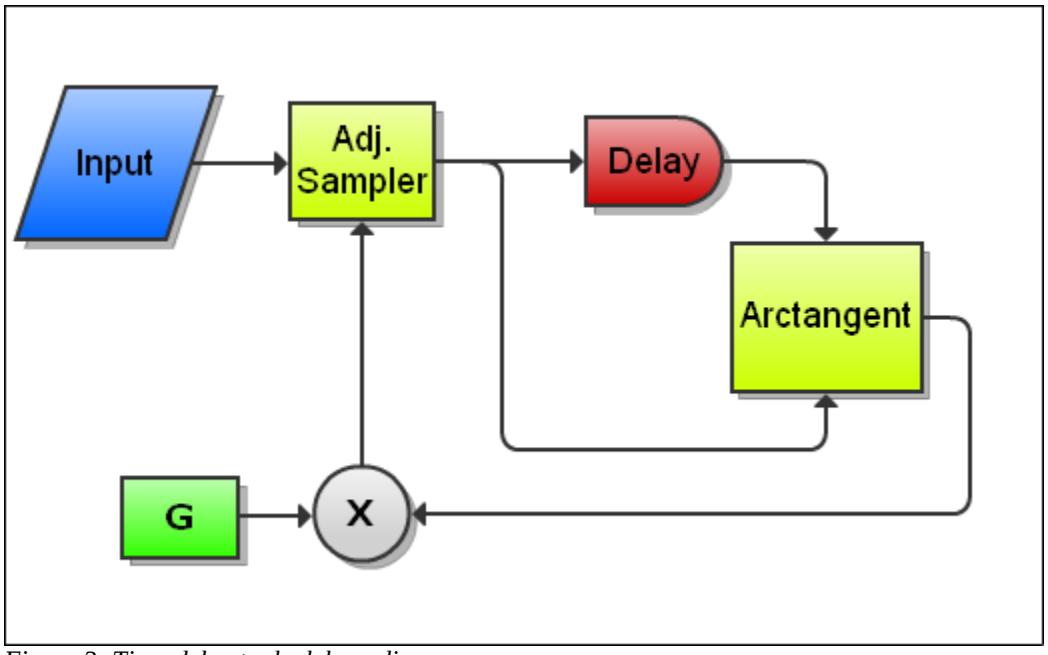

Figure 3: Time-delay tanlock loop diagram

To make the TDTL operate with any sort of efficiency on the target embedded platforms, two custom fixed point types were defined, a custom arctangent approximation was implemented, and the adjustable sampling algorithm was adjusted to fit over a fixed rate audio sampler.

i. Fixed point number system

Most low power processors have very limited computational capacity, and it is not possible to use even low precision floating point numbers. To make up for this, fixed point numbers are frequently used in place of floating point for embedded DSP; however, even then, there is little in terms of standard fixed point formats and processing. Because this code is designed to be cross platform, two new fixed point types are defined in terms of signed integer arithmetic. On the upside, this gives a fast alternative to floating point numbers. On the downside, it is unlikely that the compilers will be able to convert these integer operations into instructions that use custom fixed point processing pipelines if they are available (and so some performance will be lost). On top of this, by using signed types, information about numerical overflow and underflow is lost.

The F15 is the base type and is the faster of the two. It is defined as a 16-bit fixed point number with one sign bit and 15 fractional bits. This means it can represent numbers between -1.0 and 0.99997 with a resolution of about 3.05e-5 (1 / 32768). Addition is done with regular signed integer addition. It is quite possible that overflow will occur and result in inaccurate numbers (0.75 + 0.75 would turn into something around 0.5), but another type is defined to deal with cases where this sort of overflow is unavoidable. Multiplication is done by converting the two 16 bit signed values to 32 bit signed values, passing them through a signed integer multiplier, shifting the number 15

bits to the right, and saving the bottom 16 bits as the resultant. For instance, if a is the signed integer representation of the fixed point number A and b is the signed integer representation of the fixed point number B, then A is understood to be a / 32768 and B is understood to be b / 32768. The operation C = A * B is then equivalent to C = (a / 32768) * (b / 32768). The resultant value will need to be converted to a signed integer format for storage (right now it would be fractional), so the operation c = (a * b) / 32768 is performed, where c is the signed integer representation of the fractional value C (C = c / 32768). The operation (a * b) is simple signed integer multiplication, and the 1 / 32768 multiplication is equivalent to a right shift by 15 bits. There are issues with using this approach for negative numbers (numbers will be rounded in the wrong direction, -3 / 2 is -1, but -3 shifted to the right by 1 is -2), but it is asserted that these problems will not be significant.

Similarly, an F16 type is used when there are overflow dangers with the F15 base type. F16 types are represented with 32 bit signed integer values with one sign bit, 15 integer bits, and 16 fractional bits (minimum -32768 and maximum 32767.99998). Addition is performed with regular 32 bit signed integer addition, and again, overflow can cause problems, but most of the math in this application does not use a significant portion of the integer part of the number, so it is hoped that the chances of overflow are low. Multiplication is performed similarly to that above except the values are converted to 64 bit signed values and the right shift is by 16 bits.

The numerical needs of the algorithm are largely determined by the precision of the analog to digital converter and the hardware multiplication units on the processor. For the reference board, 16 bit multipliers were available, so using a 32bit multiply for the F15 types was logical. However, only 12 bits of precision were available from the ADC

to fill the 16 bit F15 numbers. If the processor had only 8 bit hardware multipliers, it may have been better (for code speed and size purposes) to use a base fraction type of 12 bits so that only 24 bits would need computed in each multiplication. This optimization would do nothing for the 16 bit multipliers, as 32 bits of result would be computed even if only 24 bits were required in the result, but with the 8 bit multipliers some operations can be shaved off.

ii. Modified arctagent

Secondly, full arctangent computations are expensive. To combat this, a custom arctangent based on F15 input and F16 output was created that uses linear interpolation between reference points to compute the output value. Conditionals were used in combination with a single quadrant arctangent to emulate the four quadrant behavior desired.

As a practical note, 64 points spaced evenly from zero to 16 were used as inputs in creation of the interpolation tables. The time-delay tanlock loop generally converges faster and better as the arctagent approximation is improved. Figure 4 compares the error between a full double precision arctangent and the custom arctangent.

As a caveat, this graph does not take into account a low precision division used in the arctangent approximation calculation, so the values in Figure 4 are best case errors.

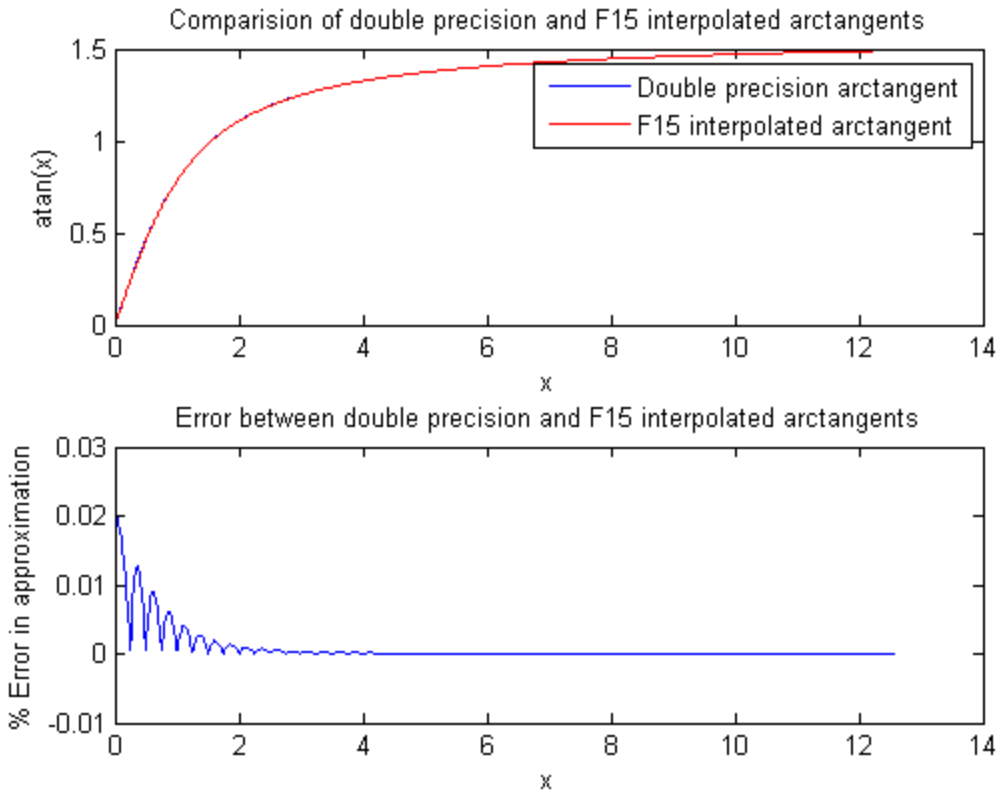

Figure 4: Comparison of double precision and F15 interpolated arctangents

iii. Sample interpolation

Finally, it is often not possible to implement the variable sample rate algorithm and fractional delay on all hardware platforms. To combat this, variable rate sampling is emulated using linear interpolation between neighboring points in a stream of values sampled at a constant rate. It is assumed that this will provide sufficient precision with a high enough sample rate.

The TDTL algorithm was adjusted to use time differences based on F16 numbers with fractional parts. The integer portions of these numbers are used to find the real samples to interpolate between, and interpolations are done using the fractional portions of the numbers.

b. PSK31

As the phase difference signal is updated, the PSK31 demodulator watches the last few values. If the sum of these is greater than some representative value of 180 degrees (it can be quite a bit lower without errors in decode), then the demodulator knows a phase switch has happened.

If this signal occurs far enough from the last received signal, the signal timer is realigned and the phase switch is recorded in the bit buffer as a zero. Under the PSK demodulation scheme, two phase switches in series (or two zeros) indicates that the character has ended. If another phase switch is received in the 31.25hz window, then the last two zeroes are dumped from the bit buffer, and the remaining bits are passed to the print function (which will print nothing if the character is invalid).

If instead of receiving another phase switch in the next 31.25hz window the demodulator detects nothing, a one is recorded in the bit buffer and the demodulator knows it can receive either another one or zero in the next window without ending the character.

The 31.25hz window has some margin for error on it. Practically, the demodulator will wait past the 31.25hz cutoff between symbols for the next phase shift, and it will accept the phase shift if it appears before as well. The decoder is currently set to wait 25% past the end of the cuttoff for phase switches and accept phase switches 25% before the cutoff.

As a graph, the decode logic is shown in Figure 5.

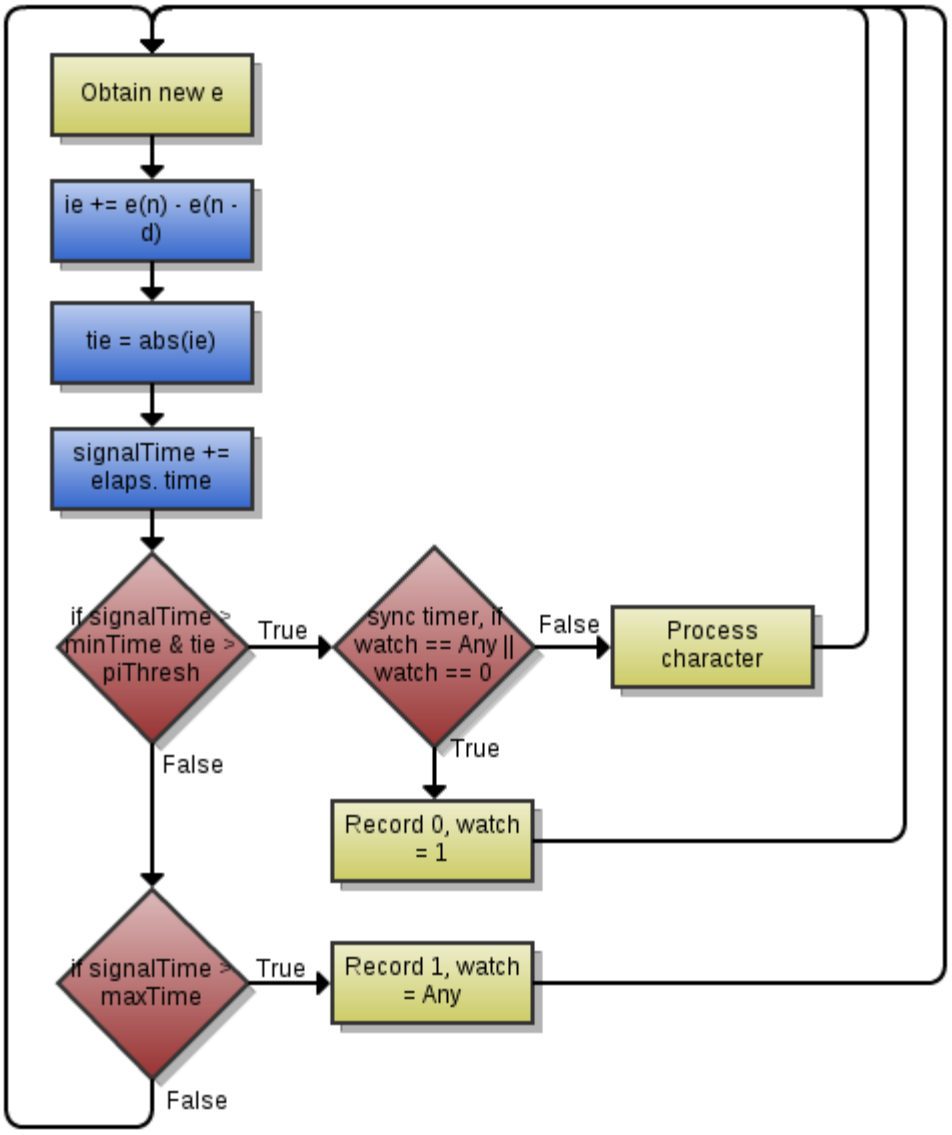

Figure 5: PSK31 decode logic

c. RTTY

RTTY demodulation requires somewhat more code, but in all reality, the ideas are just as simple. Again, as the phase difference signal is updated, the RTTY demodulator classifies it either as representative of a mark or space. The TDTL constants are chosen such that a value less than or equal to zero represents a mark and a value greater than zero represents a space.

If enough time passes (around one fifth of the 70hz single bit time) between transitioning from one state to another, the demodulator counts this time as valid and starts accumulating this time in the respective timer bin. If the transition flips back and forth before this switch is hit then the flips are ignored.

Whenever this timer switch occurs, the other timer bin is processed for valid bits. For instance, if a switch from accumulating space time to mark time occurs, then the switch time is moved from the space bin to the mark bin and the value of the space bin is checked to see just how many spaces were received. As valid spaces are detected, zeroes are inserted into the received bit record, and as valid marks are received, ones are inserted in the received bit record.

If the bit record every gets longer than seven bits, the demodulator knows it may have received a valid character. It checks to see if these bits form a valid symbol. If they do, it strips the one start bit and two stop bits, prints the character, and removes all seven bits from the record. If the bits do not form a valid symbol, the first received bit is thrown away, leaving six bits. The demodulator will check the remaining bits once it receives another seventh bit.

As a graph, the decode logic is shown in Figure 6.

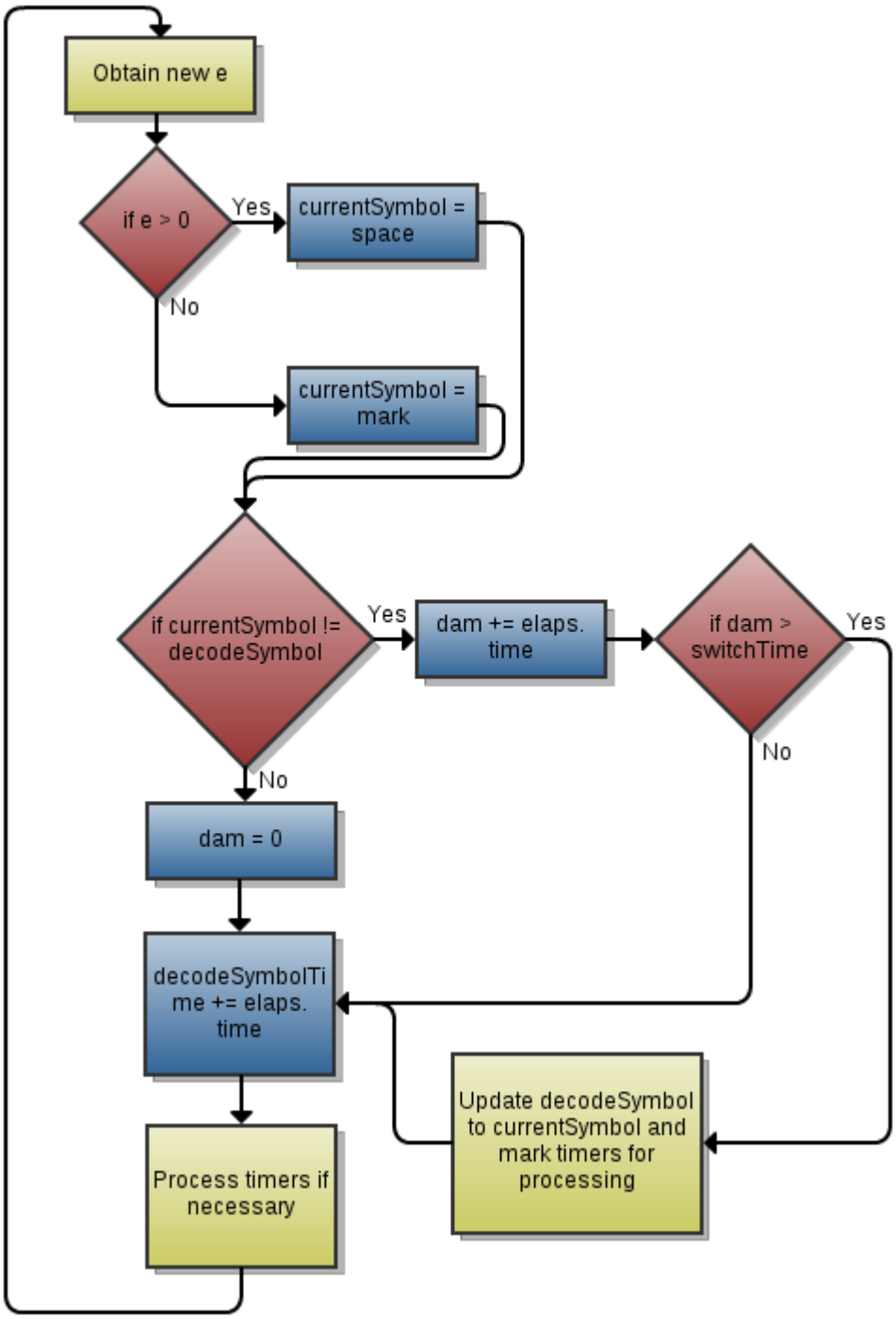

Figure 6: RTTY decode logic

4. Software Design

The software employs an interrupt-based approach to implementing the algorithms. At the simplest level, a processor using this algorithm would need one hardware timer for decode logic and one mechanism for sampling audio at a fixed rate (8192hz used in the reference code).

The application is written entirely in C, as this is the most widely available embedded programming language. Currently, no recursion is used, and so there is no need for a particularly large stack. However, at least a single precision floating point implementation along with a two quadrant arctangent will be necessary for running the intialization routines. In all reality, these functions could be replaced by precomputing and manually coding in the interpolation tables, but it is simple to just run the code live when the floating point libraries are available.

Some attempts were used to confine device specific information to util.h and the various initialization functions. However, this was not possible in all cases, and there may be some device specific hooks laying around in inconvenient places.

Currently the instruction clock of the processor will need to be about 3Mhz or higher with availability of an efficient 16 bit multiplier. If only an 8 bit multiplier is available, the clock will need to be much higher, maybe 10Mhz even. The computational requirements are largely determined by the corresponding frequency of the variable rate TDTL sampler which will be close to the corresponding frequency of the fixed delay (1365hz in the reference code). While the processor will interrupt at the sampling rate to record new values, it will do most of its processing when the TDTL requires an update. In this case, the current PIC implementation uses around 110 clock cycles for the front end filter, 1200-1400 clock cycles for TDTL processing, around 100 clock cycles for the PSK31 demodulator, and 200-400 clock cycles for the RTTY demodulator. Those numbers do not include the time necessary to print a successfully received

character. This is executing the code in debug mode as well. When optimizations are applied, those numbers will drop significantly

As far as memory goes, running both decoders together would probably require a little over 1024 bytes of memory. Lookup tables use the majority of memory. 512 bytes are dedicated to the PSK31 reference tables, 128 bytes to RTTY, and the arctangent tables use 256 bytes. Other than this, filter coefficients, stack variables, and state variables probably take up another couple hundred bytes.

If necessary, the RTTY and arctangent tables could be placed in slow memory (because the RTTY accesses are simple, and the arctangent computations are mainly bound by multiplication performance). However, the PSK31 print function performs a binary search on the PSK31 reference tables, so the slow memory performance hit many not be acceptable there. The organization of the code is fully outlined in Appendix A; however, Figure 7 should paint at least a basic picture of how all the pieces come together:

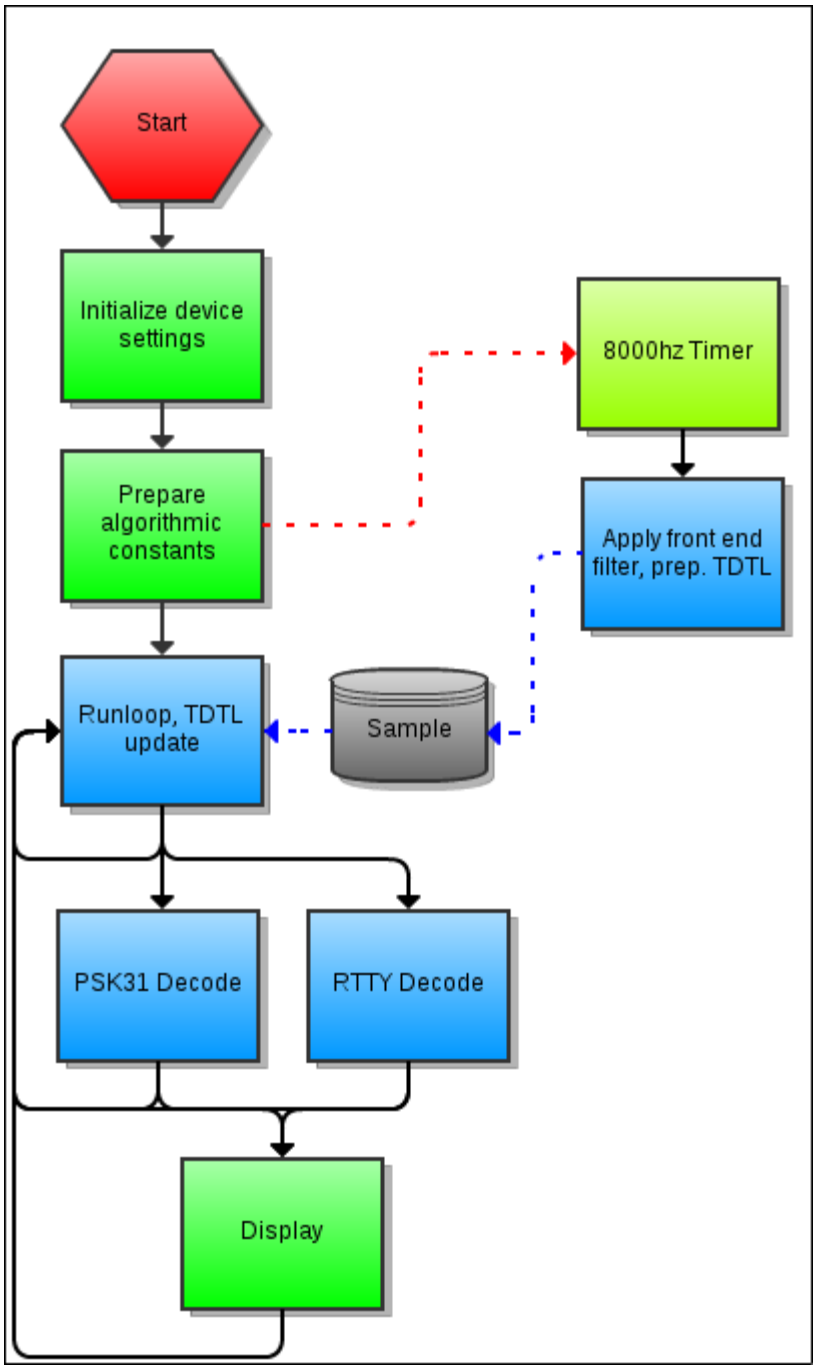

Figure 7: Operational graph of software

4. Reference Board

For development purposes, a reference board was developed with a dsPIC33FJ64GP802 processor. This is very much an non-optimal design and uses far too much power. The processor is oversized with respect to memory, computational capacity, and pin count, and the audio circuitry is sub-optimally designed. On top of this, many applications probably will not require the display. However, the code is currently written to run on this board with this pin layout, and so it should still be useful as a reference. The schematic is shown in Figure 8.

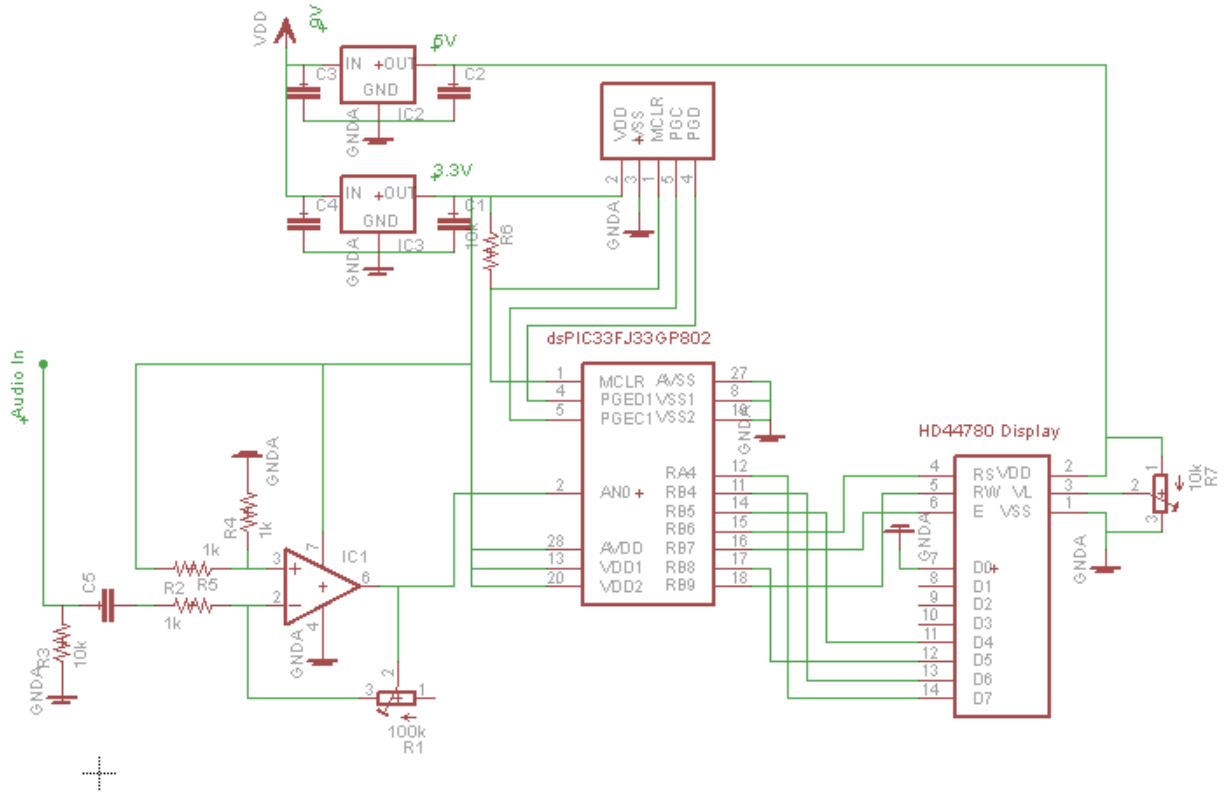

Figure 8: Schematic of reference board

As a side note, the parts on this board cost approximately $15. While a professionally fabricated board would add $5-10 to this cost, the entire project could probably be made to cost less than $15 quite easily.

6. Noise

For noise free signals, the vanilla demodulators work well. However, when noise was introduced weaknesses in the demodulation system became very evident. First of all, because the TDTL is a time-domain centered algorithm, it is sensitive to noise that is far off its center frequency. To combat this, a two pole IIR filter centered at 1365hz was added between the audio sampler and the TDTL input.

While the input filter was quite effective at reducing the effects of noise, some jitter still appeared at the TDTL output. As the TDTL is a non-linear system, no attempt was made to place more filters inside it (in fear of messing up the convergence). However, there is room to filter the outputs before they are fed into the PSK31 and RTTY demodulators.

For PSK31, as the demodulator is currently designed, it is difficult to perform effective filtering. In looking at a graphs of the phase difference signal used for demodulation, the PSK31 signals appear as impulses. Unfortunately, so does noise. Filtering the noise would hurt attenuate the valuable signals just as badly. Figure 9 shows a BPSK signal with a 12dB signal to noise ratio (added with MATLAB's awgn function), and Figure 10 shows a BPSK signal with a 3dB awgn signal to noise ration. The phase-switch spikes can be cleanly seen in Figure 9, but they are much harder to discern from the noise in Figure 10. Because the noise and the phase-switch spikes are both high frequency signals, it is difficult to separate them.

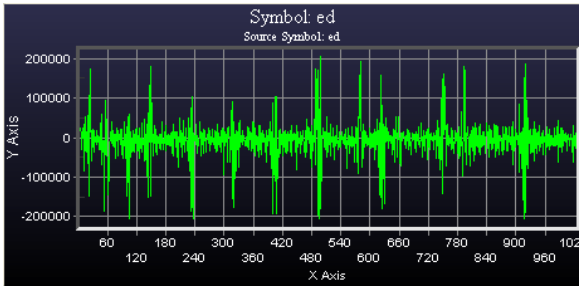

Figure 9: BPSK signal with 12dB signal to noise ratio

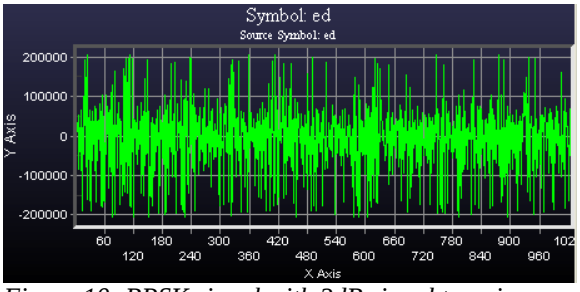

Figure 10: BPSK signal with 3dB signal to noise ratio

The PSK31 decoder was recast in terms of integration (so that the noise could be averaged out). However, this did not improve noise resiliency much at all. The error signal coming from the TDTL is simply not ideal for decoding a noisy PSK signal. For instance, depending on the frequency of the PSK signal, the error signal make not be centered at zero for no phase shift. This can be corrected by creating a zero reference signal with a high pass filter, but there are still transients of this translation that become problematic in the integration that follows. All in all, the TDTL is probably not appropriate for PSK31 demodulation, and another mechanism must be developed.

For RTTY, noise reduction is fairly simple. Because the phase difference signal appears as a square wave, the high frequency noise can be cut out with a simple, well-designed low-pass filter that responds quickly enough to avoid distorting the 45.45 baud square waves.

a. Performance

The following graphs show the front end filter and RTTY lowpass filter in action. Figure 11 shows an input signal with noise added by MATLAB's awgn function to ensure a 3db signal to noise ratio, and Figure 12 shows that signal after it has been passed through the front end filter. Figure 13 shows the output of the TDTL with that input signal, and Figure 14 shows the result of passing this signal through a lowpass filter before feeding it into the RTTY demodulator. Figure 15 and Figure 16 show the TDTL output and filtered TDTL output if there is no front end filter in place.

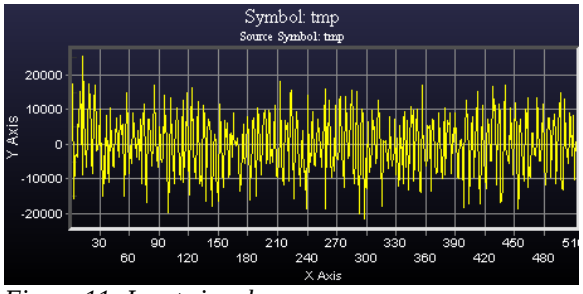

Figure 11: Input signal

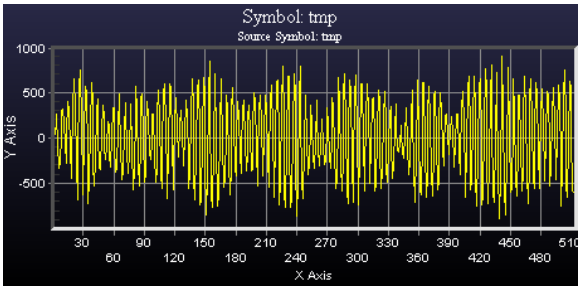

Figure 12: Signal after front end filter

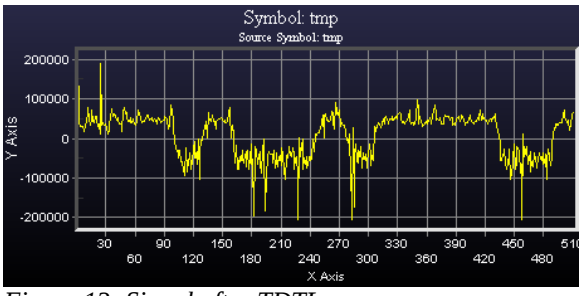

Figure 13: Signal after TDTL

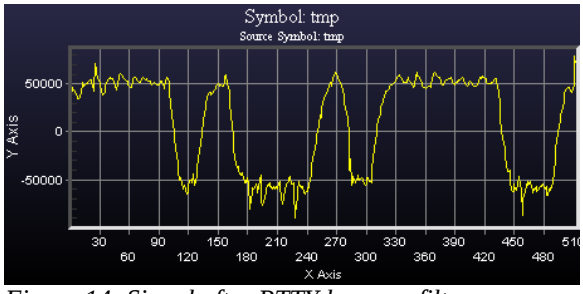

Figure 14: Signal after RTTY lowpass filter

For the 3db signal to noise signal in the above graphs, the simulated RTTY demodulator produced the entire message “HELLO WORLD.” For 0db, the simulated demodulator produced “HELO WORLD”, and for -3db the simulated demodulator produced “HELO TRGL.” As can be seen, as the noise becomes more prominent, the received message slowly degenerates to the point that it is uninterpretable.

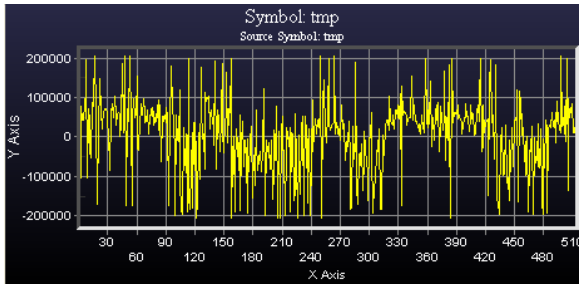

Figure 15: Signal after TDTL (no front end filter)

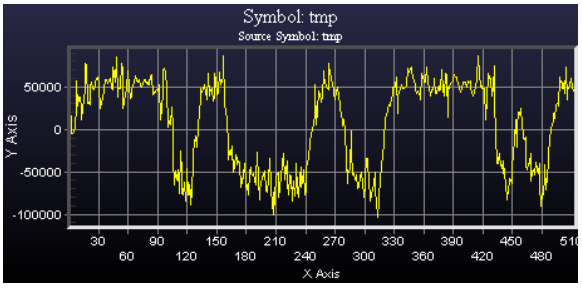

Figure 16: Signal after RTTY lowpass filter (no front end filter)

To make a comparison between the actual reference hardware platform and a standard MTTY computer decoder, MATLAB was used to generate a number of RTTY 'HELLO WORLD' messages with various awgn signal to noise ratios. The signal was sent to the targets

four times at each signal to noise level, and the best and worst cases were recorded. Table 1 shows the results for the reference hardware and Table 2 shows the results for MTTY.

Table 1: RTTY Error Rates for Reference Hardware		
MATLAB awgn Signal to Noise Ratio	Worst Case	Best Case
12dB	'ELLO WORLD'	'HELLO WORLD'
6dB	'ELLO WORLD'	'ELLO WORLD'
3dB	'EELLO WORLD'	'ELLO WORLD'
0dB	'ELLO ORLD'	'HELLO WORLD'
-3dB	'ELL QOLD'	'HELONWRD'
-6dB	' ORO'	'ZERO WRC'

Table 2: RTTY Error Rates for MTTY		
AWGN Signal to Noise Ratio	Worst Case	Best Case
12dB	'ELLO WORLDV'	'ELLO WORLD'
6dB	' WORLD'	'MELLO WORLD'
3dB	'JELLO WORLDX'	'ELLO WORLD'
0dB	'G WORLDV'	'HELLO WORLD'
-3dB	' WORLD'	'LLO WORLD'
-6dB	'BZOWOJPDM'	'AA WORLDRX'

As can be seen in the tables, the reference platform performs decently compared to MTTY, though it is unable to match performance down at very low noise levels. Of note though, the reference board was able to synchronize with the message and start recording much faster than MTTY. Unfortunately, this usually is not a problem, as RTTY sends synchronization messages during the time between when the transmitter is turned on and the user starts typing a message. However, as the reference platform was put together with very little effort, its performance is quite admirable.

As no error reduction mechanisms were put in place for PSK31, no attempt at comparing the noise performance against commercial software was made. It is almost certain that it would be terrible.

7. Where To Go From Here

There are two major pieces of work to be done to the project now. First of all, the noise performance must be increased dramatically, and secondly a better hardware platform should be developed.

a. Error rates

RTTY demodulation is quite suited to prediction, and it would be very straightforward to implement a mechanism by which the processor “guesses” about bits. If the processor has received only part of a character and then the signal drops out, it could use the values it has to look up what bits it is missing.

In combination with this, the timers could be modified to be more intelligent. Perhaps the program could interpret bits as not entirely marks or entire spaces, but fractional combinations of each (30% mark and 70% space, for instance). By doing this, the processor could take a probabilistic approach to decoding. If a bit is received as 50% mark and 50% space, then the bit prediction mechanism could be used to decide which is right (perhaps one of the characters is far more common that the other). It is not clear if this would improve error rates, but it would be worth trying.

The PSK31 decoder probably needs to be based on a signal detector other than the TDTL. While it works nicely in theory, it proved very difficult to get working in noisy situations. Even when integrated, the signal proved too finicky to process.

b. Hardware platform

Secondly, a better hardware platform needs to be defined. The current platform is power ignorant, and also suffers from clock problems associated with not having a crystal reference. A

board truly based on low power consumption needs to be designed and built with a decent clock reference. The Atmel megaAVR or XMEGA processors would be appropriate. Some of the higher end Texas Instruments MSP430 processors could be useful. Perhaps even a lower power PIC processor would work as well. All of this is speculation, but it seems reasonable that this demodulation scheme could be implemented on a processor with a 3Mhz clock at 1.8V, which, given a 3-4mA draw, would keep the demodulation power envelope below 10mW. Given batteries with 2000mAh ratings (AA size), a board drawing 3-4mA could run for 500 hours. Likely though, the audio and display circuitry would require a significantly larger amount of current and a higher voltage supply (5V) which would reduce battery life to a couple days. However, it is still worth exploring this realm, as the software in this project is most valuable for low power platforms where it is not possible to have full computers.

Appendix A: Code Design

This appendix attempts to describe how the organization of the code files.

/ – Root directory

schematic/

schematic.sch – Schematic of reference board

senior_design.lbr – Library of part symbols (incomplete)

simulations/ – Contains all the MATLAB code used for design and simulation of different components on the board

atan2_lookup.m – Performs arctangent calculation based on lookup tables for FSK TDTL simulations (algorithm defunct)

bandpass.m – Designs bandpass filters (used for front end filter). Prints F15 coefficients for IIR filter for given resonant frequencies and damping factors

dec2hexs.m – Converts decimal numbers to hexadecimal

gen_psk31_for_mplab_sim.m – Generate PSK31 signal, convert the samples to a 12 bit fractional type friendly with the MPLAB ADC simulator, and save these to a file

gen_psk.m – Generate a PSK31 signal for a given message and center frequency (on a -1.0 to 1.0 peak to peak sine wave)

gen_rtty.m – Generate a RTTY signal for a given message, mark frequency, and space frequency (on a -1.0 to 1.0 peak to peak output signal)

gen_rtty_for_mplab_sim.m – Generate RTTY signal, convert the samples to a 12 bit fractional type friendly with the MPLAB ADC simulator, and save these to a file

highpass.m – Designs highpass filters (was used for PSK31 noise reduction). Prints F15 coefficients for IIR filter

lowpass.m – Designs lowpass filters (used for RTTY noise reduction). Prints F15 coefficients for IIR filter

play_psk31.m – Play PSK31 message through computer speaker

play_rtty.m – Play RTTY message through computer speaker

tdtl.m – Full TDTL demodulating FSK signal

tdtl_psk.m – Full TDTL demodulating PSK signal

tdtl_rtty_fixed.m – TDTL using fixed point math and approximate arctangent demodulating FSK signal

arctanapprox.c – Code for arctangent approximation. Contains static allocations for lookup tables

arctanapprox.h – Header file for arctangent approximation

baudtimer.c – Code for interfacing with device timer for measuring signal times

baudtimer.h – Header file for interfacing with device timer

delay.s – Device specific assembly delay functions

DSP.mcp – MPLAB project file

dsp_data_monitor.dmci – MPLAB data monitor file

dsp_stimuli.sbs – MPLAB simulation stimulus

fractionaltypes.h – Header file containing macros and some test code for the F15 and F16 types

frontend.c – Code for front end filter. Contains static allocations for filter coefficients

frontend.h – Header for front end filter

HD44780.c – Code for interfacing with HD44780 compatible display in 4-bit mode. Contains device specific pin-interfacing code

HD44780.h – Header file for interfacing with HD44780 compatible display. Contains macros for character and command definitions

main.c – Main file. Contains run loop, sample interrupt code, and TDTL implementation

psk31.c – Code for PSK31 demodulator. Contains static allocations for tables translating between PSK31 characters and HD44780 characters

psk31.h – Header for PSK31 demodulator

README.txt – Describes how to use code

rtty.c – Code for RTTY demodulator. Contains static allocations for tables translating between RTTY characters and HD44780 characters

rtty.h – Header for RTTY demodulator

util.c – Code for some hand utility functions as well as the device specific ADC and sample-driving interrupt timers

util.h – Header including a few device specific constants and programming environment specific data types

Bibliography

Al-Araji, Saleh R., Hussain, Zahir M., Al-Qutayri, Mahmoud A. *Digital Phase Lock Loops Architectures and Applications*. Dordrecht, The Netherlands: Springer, 2006. Print.

Bartlett, Forest. A letter of recollections. Retrieved Dec. 5, 2010 from <http://www.rtty.com/history/w6owp.htm>

Henry, George W. Jr. ASCII, BAUDOT AND THE RADIO AMATEUR. Retrieved Dec. 5, 2010 from <http://www.digigrup.org/ccdd/rtty.htm>

Hill, Donald A. Getting Started on RTTY: Technical Discussion of Diddles. Retrieved Dec. 5, 2010 from <http://www.aa5au.com/gettingstarted/rtty_diddles_technical.htm>

Martinez, Peter. PSK31: A new radio-teletype mode with traditional philosophy. December 1998. RADCOM magazine. Retrieved Dec. 5, 2010 from <http://det.bi.ehu.es/~jtpjatae/pdf/p31g3plx.pdf>