Description of RSCW's algorithms

This page describes the algorithms used by my RSCW program. Furthermore, some advantages and disadvantages of these algorithms are mentioned.

The goal of this page is not just to inform the curious, but also to provide a starting point for discussions about better algorithms, improvement to the present algorithms, extension of the algorithms to non-machine-sent code, etc. Comments are most welcome!

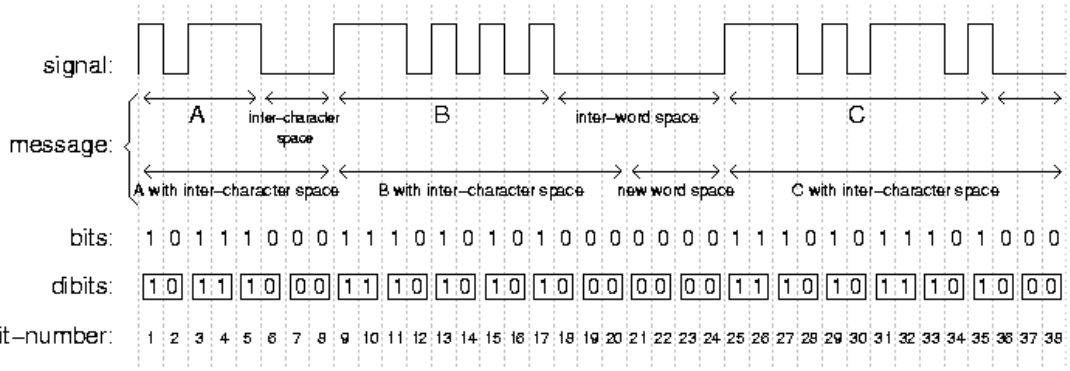
Some details and definitions about morse code

RSCW (at the moment) only aims to decode morse code that has perfect timing. Hand-sent morse may not have a very rigid timing, and still be easily decoded by the human ear and brain, but such code may be a lot harder to decode automatically. So practically, RSCW is mainly meant for machine-generated code, like the telemetry transmitted by satellites. (Actually, it is a trade-off: by making the algorithms expect perfect timing, we can dig weaker signals out of the noise than with an algorithm that also accepts sloppily sent code.)

Just to be sure, this is the timing specification for morse code:

- A dash is three times as long as a dot.
- The space between dots/dashes within one character is equally long as a dot.
- The space between two characters is three times as long as a dot.
- The space between two words is seven times as long as a dot. (Note: the RS12 satellite seems to use a factor of nine here.)

So the duration of a dot is the obvious unit of time.



From now on, I'll use the word **bit** for whatever is sent during one unit of time; a **1** indicates that the transmitter is on, a **0** indicates off. So the character 'R' (dot dash dot) would be sent as seven bits, namely 1011101 .

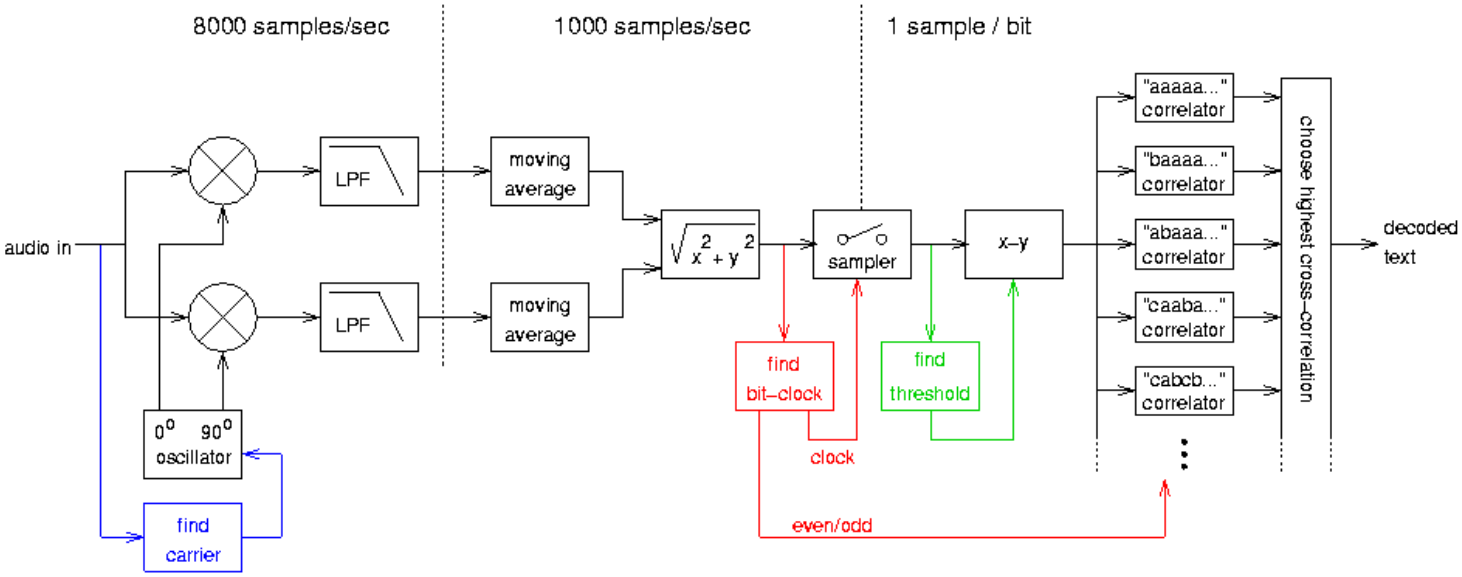
However, for the purpose of decoding morse by computer, this is not a very practical representation. If we receive stream of bits like 1011101 we cannot yet decide that this is actually an 'R': it could be that the next two bits would be 01, making the character an 'L'. This is solved easily by considering the inter-character space as part of the character itself. So we say that 'R' is 1011101000, while 'L' is 101110101000. Consistently, we can express the space between words as the bits 0000. (Technically, the code is prefix-free.)

Now take a look at how this code for an individual character is composed. For every dot, we have the sequence 10; for every dash, 1110; and at the end, we attach 00 (which, together with the 0 at the end of the last dot or dash, completes the group of three 0s that is the inter-character spacing). Consequently, every character is represented by an even number of bits. Furthermore, if we consider the bits in groups of two, called a **dibit**, we see that only the sequences 10, 11, and 00 occur; the sequence 01 never occurs! So if we number the bits starting from one, the odd-numbered bits are more likely to be 1 than the even-numbered bits. This phenomenon can be exploited to regenerate a clock signal, as we shall see below.

Note: either the 'bit' or the 'dibit' as defined above, would usually be called a symbol in communications theory. Since symbol in the context of morse code could also be mistaken as referring to the actual letters being transmitted, I avoid that word in this description.

The algorithm used in the RSCW program.

The below picture is a block diagram of what happens inside RSCW:



In black, the path of the signal itself is shown: on the left, it enters as an audio signal (sampled with 8 bits per sample, at 8000 samples per second), and on the right the algorithm outputs the decoded message. The blue, red and green parts deal with other issues, namely finding the carrier frequency, finding the bit clock and choosing a threshold.

Decoding of morse signals -- the signal path

This consists of the following steps:

1. Multiply the incoming signal with a locally generated carrier at the same frequency as the incoming signal. This mixes the incoming signal to 0 Hz (or nearly 0 Hz, if the local carrier is not exactly at the right frequency). In order not to loose information, two signal paths are created, one after mixing with the carrier itself, and one with the same carrier shifted in phase by 90 degrees. The paths are customarily called I (in-phase) and Q (quadrature).
2. Low-pass filter both I and Q signals. Note that, due to the mixing done in the previous stage, this corresponds to passing only signals which were originally near the carrier frequency: it acts as a band-pass filter.
   In RSCW, this filter is just a moving average over 48 samples, but it could be replaced by a steeper filter to attenuate signals away from the frequency of interest more.
   Furthermore, in RSCW this step also does a downsampling: 8000 samples per second enter the filter, but only 1000 samples per second leave it. This is acceptable since all high-frequency components have been removed, and it reduces the computational load of the rest of the algorithm.
3. Calculate a moving average over the duration of exactly 1 bit, again for both the I and Q signals.
   (Note: a moving average over say 10 samples is just first the average of samples 1,2,...,10, next the average of samples 2,3,...,11, next over 3,4,...,12, and so on.)
4. Since the phase of the incoming carrier is unknown, we next calculate a Pythagorean sum of the I and Q contributions. This makes the result independent of the phase difference of the received signal and our local carrier.
   The result of this step is the green line in the graphs produced by RSCW.
5. Once per bit, take a sample from the above Pythagorean sum. These samples should be taken at the right moment, namely such that the moving averages have just covered the duration of one entire bit (as opposed to part of one bit and part of the next bit).
   It is the job of the clock regeneration to find out the right sampling moments; see below.
6. Next, subtract from each sample a number which is the **threshold**. A suitable threshold is somewhere in between the typical signal level for a 0 (i.e., the noise received while the transmitter is off), and the level for a 1 (i.e., the result of both the transmitter's signal and the noise).
   The result of this is (ideally) a signal that is positive if a 1 was transmitted, and negative if a 0 was transmitted. However, due to noise this may not always be the case.
   In the graphs, this signal is shown in cyan (light blue).
7. Finally, calculate the cross-correlation between these samples and sequences of 1s and 0s (or rather, -1s) corresponding to every possible message. Then the most likely message, given what we have received, is the one that has the largest cross-correlation, so output that message as the result of the decoding process.
   See below for a more detailed discussion of how to do this practically.

Remarks:

- Assuming correct estimation of the carrier frequency, the bit clock and the threshold, the above procedure is the theoretical optimal one for decoding On-Off-Keying (OOK) signals in the presence of Additive White Gaussian Noise (AWGN).
- The I and Q branches actually constitute two matched filters (with a 90 degree phase difference) for one bit transmitted using OOK. In fact, there is no need to split each branch into several steps, it is possible to design a FIR (finite impulse response) DSP filter that does this in one step (using a convolution). But the multi-step implementation is more insightful.
- Up to and including the sampling switch, the above is very similar to the setup typically used for the reception of Coherent CW.
- The algorithm above assumes that between the individual bits, a random phase shift may occur. Indeed, the transmissions of the RS12 satellite do exhibit such random phase jumps between the dots and dashes.

If the transmitter were coherent (i.e., there would not be such phase jumps), that could be exploited to improve the decoder.
In fact, a signal like RS12's does not have phase jumps *within* a dash (which is composed of three bits). In principle, this could also be used to improve the decoder.

Finding the message with the highest cross-correlation

One possible algorithm for this would work as follows. (Note: this description (and the implementation in RSCW) assumes we already know which received bits are odd-numbered, so we can do the cross-correlation on dibits instead of individual bits.)

1. Start with one empty message.
2. Extend it with all possible dibits, namely 11, 10 and 00, yielding three messages, and for each of these calculate the crosscorrelation with the received signal for the first two bits. (Actually, the cross-correlation is not done using 1s and 0s, but using 1s and -1s for symmetry.)
3. Extend each of these (three) messages is again with a dibit in all possible ways. However, not all extensions are possible now. For example, 11 cannot be followed by 00, but only by 10. Which extensions are possible is governed both by the timing rules of morse code, and by the alphabet (e.g., 6 consecutive dashes are not the initial part of any morse character). For each resulting message, add to the previous cross-correlation the cross-correlation between the new dibit and the signal for the next two received bits.
4. Repeat step 3 until the end of the transmission.

Obviously, at every repetitition of step 2, the number of messages grows with a factor between 1 and 3 depending on how many extensions are possible. As a consequence, the number of messages to be considered would quickly become unpractically large. Furthermore, we would have to wait until the end of the transmission before we could decide which message has the highest cross-correlation.

However, it is not necessary to keep all messages until the end. To see this, consider the following example. Suppose after receiving 8 bits we have the following three messages stored (and others which are not relevant for this example):
10101000 (this is dot-dot-dot, i.e., an 'S') with cross-correlation 3.4
11101000 (this is dash-dot, i.e., an 'N') with cross-correlation 4.1
10111000 (this is dot-dash, i.e., an 'A') with cross-correlation 2.3
The above three messages each represent a morse character complete with its inter-character space. Therefore, each of these can be extended with the same dibits. So if we after receiving the rest of the message find an extension for the first of these messages which results in a high cross-correlation, than that same extension also fits after the second and the third message. And since the extension contributes equally to the cross-correlation of whichever message it is attached to, we can see that adding this particular extension will always give the highes total cross-correlation to the second of these three, since that one now already has the highest cross-correlation. In other words: the first and the third of these messages will never have the highest cross-correlation, so we can just forget them. (Readers who are familiar with the Viterbi decoder for convolutional codes will recognise this phenomenon.) The essential point here is that messages which end in an inter-character space are equal in the sense that they can all accept the same extensions, and therefore we only need to keep the best of them.

The above observation helps tremendously to reduce the number of messages considered, making the algorithm actually feasible. However, it has another nice consequence: in practice, it turns out that after a while all remaining messages have the same initial set of dibits. At that point, we can be sure those initial dibits will never change anymore: we have *decoded* them as the most likely bits, and can output them (or rather the character they represent). So we no longer need to wait until the end of the transmission before we can output at least part of the received message. (Note: theoretically, it is possible that this does not happen, i.e., that several messages with different initial dibits remain "alive" forever. The software needs to handle this case correctly.)

In fact, the idea of removing messages because we can be sure they won't win anymore, can be taken a step further: rather than only looking at messages that have just ended in an inter-character space, we can look at all messages, and remove them if for every possible extension of that message, another message is still there which can also accept that extension and already now has a higher cross-correlation. This again reduces the number of messages to be kept for consideration (thus reducing the computational effort), and also makes the algorithm decide (i.e., leave only messages with the same initial dibits) earlier.

A final note: if one knows even more about the message to be received (e.g., that it consists of sets of 3 letters followed by 2 digits) one could in principle use that knowledge to make the cross-correlator also reject messages that don't fit this pattern. In principle, this improves the decoder's ability to dig such signals out of the noise; however, it has not (yet) been implemented in RSCW.

Finding the carrier frequency

RSCW uses a rather simple algorithm for this: it performs an FFT (Fast Fourier Transform) on 1-second segments of the incoming signal, and chooses the frequency at which the power is largest (within a range of frequencies specified by the user). It does this twice a second. Linear interpolation is used to estimate the frequency between two successive FFT results.

This algorithm works well as long as there is only one significant narrow-band signal in the frequency range; otherwise, it may easily jump between the several peaks in the spectrum, rendering the decoder useless.

For the satellite signals, a "tracking" mode has been implemented: this tries to track a slowly drifting frequency (due to Doppler shift) by adapting the acceptable range of frequencies. This works fine once it has found the signal, but initially acquiring the signal is a problem unless the signal is already there when the program starts.

Clearly, this is an area where improvements can still be made.

Finding the bit clock

First, a bit of theory.
Consider a stream of 0s and 1s representing several characters of morse code, as discussed earlier on this page. Now start at an odd-numbered bit, and perform the following calculation: take this bit's value, subtract the values of the bit just before and the bit just after this bit, add the values of the bits two places before and two places after this one, and so on. In other words: add the values of bits that are an even number of positions away, and subtract those of the bits that are an odd number of positions away. Since the original bit was odd-numbered, all the added bits are also odd-numbered, while the subtracted bits are even-numbered. As noted before, odd-numbered bits are more likely to be 1 than the even-numbered bits; as a consequence, the above calculation will give a positive outcome. Conversely, if we had started the procedure with an even numbered bit, the outcome would have been negative. Thus, we can use this calculation to synchronize a stream of bits, in the sense of deciding whether a certain bit is odd or even.

Unfortunately, the above is just theory which we can't apply yet, since we don't have the bits yet. However, we can apply the same calculation to the stream of signal magnitudes provided by step 4: start with the magnitude at some point in time, add the magnitudes at points in time that are an even number of bit times earlier or later, and subtract the magnitudes at times that are an odd number of bit times earlier or later. If our initial time was right in the middle of an odd-numbered bit, the outcome would be positive, while if the initial time was right in the middle of an even-numbered bit, the outcome would be negative. And, as it turns out, if the initial time is somewhere in between, the outcome is also in between. Thus, doing this calculation at every point in time yields a signal that has a peak in the middle of the odd-numbered bits, and a minimum in the middle of the even-numbered bits.
This signal (based on a calculation about 10 bit times into past and future) is plotted in yellow by RSCW.

In practice, the signal derived as above is somewhat noisy. In order to estimate the time of the maxima and minima more accurately and reliably, RSCW fits a sine to the signal, and then uses the maxima and minima of that sine to choose the sampling instants for step 5.

This algorithm obviously fails if the difference between odd and even bits is not honored. Unfortunately, at least one amateur radio station transmitting machine-generated CW (the beacon DK0WCY on 10.144 MHz) seems to use an inter-character spacing of 4 instead of 3 bits, and can therefore not be decoded reliably with RSCW. Obviously, finding an algorithm without this limitation would be a useful improvement.

Establishing the threshold

In RSCW, the following algorithm is used. Given a set of samples (for 5 bits before and after the bit for which we're trying to set the threshold), choose the threshold such that the average distance between the threshold and samples above it, is equal to the average distance between the threshold and samples below it. If there are equally many samples above and below, then is just equivalent to simply taking the mean of the samples. But if the samples are unequally distributed (e.g., during reception of a character that contains a lot of dashes), this algorithm prevents a bias due to the unequal distribution. Actually, the threshold as defined above is not unique: there are typically several values which satisfy the criterion; in that case, the average of the highest and the lowest possible such value is taken. Furthermore, the 7 samples closest in time to the bit of interest are entered twice into the calculation, to make the resulting threshold behave more smoothly in the presence of large signal strength variation.

The resulting threshold is shown as a red line in the graphs.