# Simple Conversational Amateur Messaging Protocol (SCAMP)

by Daniel Marks, KW4TI

Draft v0.9 2023-9-12

## Abstract

Modern digital modes for amateur radio tend to require complicated transceiver architectures. For example, most need upper sideband modulation with a very stable local oscillator. Furthermore, methods such as multiple FSK can be susceptible to effects such as intermodulation from strong adjacent stations. Earlier modulations such as CW, which uses on-off keying (OOK), and RTTY45, which uses 2 frequency-shift-keying (FSK) were usable on transceivers with much less demanding requirements. Recently, there has been a trend towards sending data with a very low symbol rate (e.g. FT8 and QRSS) in order to achieve communication with very low transmitter power and/or with poor receiving conditions. This is a proposal for a simple digital mode, Simple Conversational Amateur Messaging Protocol (SCAMP), that can be implemented with OOK or 2FSK for conversational or amateur radio contest QSOs, can be implemented with relatively crude transceivers, and on simple 8-bit microprocessors such as the ATMEGA328P used for the Arduino Uno or Nano. It has both conversational and data transfer modes, but is tailored more to low bit rate text connections.

## Introduction

There is now an abundance of amateur radio data modes. Early digital modes used FSK, for example, RTTY45, AMTOR, and its successors in packet radio. As personal computers became widely available with increasing processing power, as well as receivers with stable local oscillators, phase sensitive modes such as PSK31 and methods with small frequency shifts such as multiple frequency-shift keying (MFSK) (Olivia/Contestia) became possible, adding increased resistance to noise. Protocols specializing in very short messages, synchronized with a global time standard such as JT65/FT8/FT4 were designed for making QSOs with low power and in poor conditions. While there have been remarkable improvements in amateur radio data modes, with FT8 especially becoming quite popular, these require transceivers with very stable local oscillators that are resistant to intermodulation and other distortions. Furthermore, the computation required to decode some of these protocols requires a computer with powerful digital signal processing capability. There are instances when simple transmitters and receivers are desirable, that for example may only be capable of OOK (such as is used for CW transmission) or FSK (by selecting a PLL frequency or modulating a crystal oscillator). It would be further desirable if the processing ability of a microcontroller (for example, ATMEGA328P) was sufficient to encode and decode the data mode modulation. QRP transceivers, for example, often use simplified hardware, based on a PLL synthesizer or crystal oscillator with a direct conversion

receiver. Controlled by a microprocessor, such QRP radios can be solar powered, are highly portable, and can serve as message relays, for telemetry, or for emergency use.

A de-facto specification for an efficient, reliable, portable HF communication protocol called Automatic Link Establishment exists, as specified in MIL-STD-188/141A. This protocol uses MFSK, forward error correction (FEC), frame interleaving, and automatic repeat request (ARQ), among other methods. This protocol has been very successful for its intended use, however, it requires the kind of complex transceivers and modems that this method wishes to avoid. However, there are aspects of such methods that can be adapted. In particular, few amateur radio modes combine simple modulation and forward error correction. Simple forms of forward error correction can be implemented on 8-bit processors, and other techniques like interleaving may also be used. Most forms of FEC require significant processing power to decode and use either dedicated hardware or high speed processors to implement methods like the Viterbi or Berlekamp-Massey algorithm. The extended (24,12,8) Golay code has been used in ALE and is a happy medium which achieves a ½ code rate and is able to correct 3 of 24 bits. Its primary disadvantage is that its short code length requires that the data is interleaved to be resistant to long error bursts. This is problematic for conversational modes for which long latency is an issue, especially for contesting. The Walsh/Hadamard codes used in Olivia/Contestia have this problem in particular because of their low code rate. The venerable extended (24,12,8) Golay code is a compromise solution that can be decoded by an 8-bit microcontroller, has reasonable latency, good code rate, and can correct up to 12.5% bit errors.

# Modulation layer

The modulation layer is of one of two types:

**On Off Keying** (OOK)– The carrier alternates between full power transmission and no transmission as a non-return to zero (NRZ) line code. The mark condition (or one bit) is transmitting and the space condition (or zero bit) is no transmission. Care should be taken that the transmitter does not significantly chirp the carrier when initiating a transmission. Envelope modulation may be built into the keying circuitry to prevent keyclick. The interval of each bit (transmitting or not transmitting) is identical and is given by the reciprocal of the baud rate.

**2 Frequency Shift Keying** (FSK) – When transmitting, the carrier alternates between two frequencies as a return-to-zero (RZ) line code. The mark condition corresponds to transmitting at one frequency and the space condition is transmitting on the other frequency. The mark frequency may be higher or lower than the space frequency. The separation between the two is determined by the baud rate. Ideally the separation is a multiple of ½ the baud rate, with a multiple of one corresponding to minimum shift keying (MSK). However, it is not expected that the transmitter can achieve a perfect separation frequency, nor can the receiver perfectly coherently decode a MSK signal. Therefore a separation equaling a multiple of the baud rate is used, so that for 100 mark or space intervals per second, these would be separated by 100 Hz. This nominally retains the orthogonality of the mark and space conditions but increases the tolerance to error in the separation frequency or local oscillator phase.

# Digital Encoding

Messages are sent as 30 bit data frames, in order of most significant bit to least significant bit (left to right as shown here). The format of each frame is

```
C XXXX C XXXX C XXXX C XXXX C XXXX C XXXX
MSB                                     LSB
```

The 24 "X" are 24 data bits to be sent in the frame which are a Golay code word. Each "C" is the complement of the bit immediately following it. This ensures there are no more than five consecutive mark or spaces (ones or zeros) in a valid codeword or consecutive codewords. The transition between the mark and space condition is used to aid in clock recovery and to allow an initial or resynchronization phase to be recognized.

# Golay code word

Each Golay code word consists of two halves.

```
PPPP PPPP PPPP XXXX XXXX XXXX
MSB                       LSB
```

The Golay code word is a 24 bit code word that contains 12 bits of data payload to be sent, represented by "X" and 12 bits of parity, represented by "P". The parity bits are calculated from the data bits using the (24,12,8) extended Golay encoding algorithm as given in the Appendix.

# Golay code word types

The Golay code word types are the 12-bit payload to be sent in the Golay code word. There are two Golay code word types.

**Data code word type**

```
1111 XXXX XXXX
MSB       LSB
```

This encodes 8-bit raw binary data XXXX XXXX in order of MSB to LSB. This message is intended to be used for exchanging data for file transfer protocols and other uses left up to the users. It is not an efficient encoding of this data but is included so that the connection may be used for purposes other than text exchange.

**Text code word type**

YYYYYY XXXXXX
MSB LSB

These encode two 6 bit symbols XXXXXX and YYYYYY. The symbol XXXXXX precedes that of YYYYYY in the data stream, that is, when considered as part of a message, the symbol corresponding to XXXXXX precedes YYYYYY. The symbols encode characters as specified in a table in the Appendix. The 6-bit code corresponding to 000000 indicates “No symbol” so that no character should be decoded in the message for this 6-bit code. If only one code is to be sent in the code word, then XXXXXX should be the code, and YYYYYY should be 000000. A code word with both XXXXXX and YYYYYY being 000000 is valid and should be considered as two no symbols. This symbol may be sent continuously during a connection one wishes to send a real-time typing message, or alternatively one can initiate a resynchronization.

For additional redundancy, the same text code word may be sent multiple times in a row. If the receiver decodes the same code multiple times before receiving a different code, it should discard the redundant decodes of the code word. If the same code word needs to be sent multiple times and not have its redundant copies discarded, at least one no symbol code (000000 000000) should be sent between the code word and its next copy so that the receiver decodes a different code. Redundant copies should be sent in immediate succession, that is, there should be no delay between sending the redundant copies of the code word. This enables the copies of the code word to be coherently summed by the receiver. Redundantly sent data code words (as opposed to text code words) should not be discarded.

**Reserved code word type**

XXXXYY 1111YY
MSB LSB

where XXXX is not 1111. There are 240 possible reserved codes. These codes are listed in the Appendix.

# Frame Synchronization

One of the aspects of a protocol that is most susceptible to corruption is desynchronization of the bit stream, that is, incorrectly starting a 30-bit frame at the wrong point in the bit stream. Therefore a synchronization protocol is necessary that can synchronize the beginning of a 30-bit frame. A fixed 30-bit synchronization frame is used that the receiver can recognize to indicate when a full frame has been received, with the next 30-bit data frame starting immediately afterwards. To not be mistaken as a data frame, the synchronization codeword must not be likely to be received as a valid data frame, even with added noise corruption. As the 30-bit data frame has complement bits inserted into it, the synchronization code word should not have complement bits at the same positions for as few shifts of the code word as possible so that the synchronization and data frames are unlikely to be confused with added noise. Furthermore, the synchronization frame should have low autocorrelation sidelobes so that it is unlikely that a shift of the frame in the presence of noise triggers a synchronization.

The synchronization frame is given by the 30 bit number

```
11111 01101 00011 00111 01000 11110 or 0x3ED19D1E
```

which is sent MSB to LSB, or in order of left to right as written here. This codeword has a maximum of two complemented bits for any shift and an autocorrelation, given a bipolar encoding such that 1 is +1 and 0 is -1:

```
30 3 -2 -3 0 -1 -2 3 0 -3 -2 -1 4 1 2 -1 -4 1 2 3 -4 1 -2 -1 4 3 2 1 0 -1
```

so that the maximum off-peak autocorrelation without noise has a magnitude of 4 for a noise rejection capability of $20 \log_{10} (30/4) = 17.5$ dB. The receiver should continuously cross-correlate the incoming bit stream with this code word and reset the clock edge timer so that the first bit of the subsequent data code word is sampled at one bit interval after the peak of the cross-correlation occurs, or the synchronization frame is approximately matched by the last 30 bits. The codeword may be resent periodically to aid in resynchronization.

When first keying up a transmission, a receiver must be given time to start receiving. Furthermore, if the FSK mode is used, the receiver must determine which of the two FSK frequencies corresponds to the mark condition. To do this, a sequence of 20 or more mark intervals are sent at the beginning of the transmission, followed by 3 or 4 space intervals, then two mark intervals, and finally the synchronization code word. This pattern may be restarted at any time during the transmission to resynchronize the receiver with the transmitter. So in full, at the beginning of the transmission the following is sent:

```
11 1111 1111 1111 1111 1111 1101 0101
11 1110 1101 0001 1001 1101 0001 1110
```

The initial 24 one bit intervals provides sufficient time for the receiver to begin to receive. If the FSK mode is used, the receiver uses the received frequency with at least 15 consecutive bit lengths to denote the one bit. The 010101 provides six bit edges to synchronize with to make 30 bits for the first frame. The second frame is the synchronization frame to help ensure that the subsequent frames of 30 bits are correctly separated.

When a data frame is received without error, six pairs of complement bits should be received corresponding to the bit pairs 1 and 2, 6 and 7, 11 and 12, 16 and 17, 21 and 22, and 26 and 27. If six pairs of complement bits are not received, there are several possibilities as to what caused the error. Firstly, one of the bits may have been inverted, so for example only five rather than six pairs of complement bits are observed. However, another possibility is that an extra bit transition was observed, or a bit transition was missed. With the erroneous bit transition, it is unlikely that the six complemented bits will be observed in their proper positions. If an extra bit transition occurs in error, then the thirty bits of a frame occurs one bit sooner than it should for the uncorrupted frame. In this case, if the number of complement bits are counted for the frame with the frame being advanced one bit, ideally five complement bits should be counted, as this indicates that waiting for another bit transition would add the sixth complement bit pair and therefore complete the frame, though with bit errors (which might be corrected using FEC). If a bit transition is missed, then by examining the frame delayed by one bit, one should ideally count six complement bits, and this indicates that one counted an extra bit during the frame, and that the frame is the thirty bits that are delayed by one bit from the

current bit. Furthermore, the most recently received bit is the first bit of the next frame. Importantly, by examining the number of complement bits in frames delayed or advanced by one bit, one can detect frame synchronization errors and possibly correct them by advancing or delaying the bit stream by one bit to compensate for the error.

# Appendices

## Six-bit symbol encoding

The following is a table of the six bit code. One or two of these codes form a 12-bit Golay code word which can send one or two character symbols. The six bit code is as follows:

| | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
|---|---|---|---|---|---|---|---|---|
| 000xxx | No symbol | Backspace | End of Line | (space) | ! (exclam-ation mark) | “ (double quote) | ‘ (single quote) | ( left parenthesis |
| 001xxx | ) right parenthesis | * (asterisk) | + (plus) | , (comma) | - (minus) | . (period) | / (slash) | 0 |
| 010xxx | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 011xxx | 9 | : (colon) | ; (semi colon) | = (equal) | ? (question mark) | @ (at) | A | B |
| 100xxx | C | D | E | F | G | H | I | J |
| 101xxx | K | L | M | N | O | P | Q | R |
| 110xxx | S | T | U | V | W | X | Y | Z |
| 111xxx | \ (backslash) | ^ (carat) | ` (grave) | ~ (tilde) | INVALID | INVALID | INVALID | INVALID |

The codes 111100, 111101, 111110, and 111111 are invalid and are never to be used.

For languages that have diacritics, the single quote, carat, grave, tilde, and backslash may be interpreted as diacritics, with the diacritic applying to the previously sent character. The backslash may be interpreted as an umlaut. It is highly recommended to send the character and the diacritic in the same Golay codeword so that these are decoded in the same word. A “No symbol” (000000) can be placed in the previous codeword to ensure that the next symbol is included with its diacritic.

For other characters representable by 8-bit bytes, for example UTF-8 encoding could be used. However, as UTF-8 has variable length symbols, missed bytes could result in sustained decoding failures, which is a serious drawback for amateur communications unless there is a redundancy check and an automatic retransmit request mechanism.

MSB 1111 XXXX XXXX LSB

where XXXX XXXX is the byte to be sent.  For example, to send lower case a, the 12-bit word 111101100001 or 0xF61 would be sent.  However, no differentiation is made between these 8-bit symbols sent as text, and those sent as data, for example, as part of a file transfer protocol.

## Reserved Word Code Types

The following reserved words are defined. Note that the first four binary digits of the reserved word must not be 1111.

`000000 111100`: End of transmission. This instructs the receiver to return to listening for the synchronization protocol so that further data/text frames are not decoded until another synchronization protocol is completed (see next section). This may be sent multiple times at the end of transmission as it is idempotent.

`XXXXXX 111111`: 28 codes that are reserved for switching to a character set. A reserved word to switch to a character set may be sent multiple times to ensure the correct character set is selected; these codes are idempotent. For standard six bit encodings, there are only 59 available characters to define in each set (000001-111011) with 0 and 60-63 reserved. Because of this, there many be some compromises to a language's orthography. For twelve bit encodings which encode only one character in each frame (for example Kana or Devanagari), there are 3481 possible codes which are likely to correspond to Unicode characters. Here are tentative codes designated for some common character sets:

`000000 111111`: Latin (6 bit)
`000001 111111`: Augmented Latin (12 bit)
`000010 111111`: Cyrillic (6 bit)
`000011 111111`: Greek (6 bit)
`000100 111111`: Arabic (6 bit)
`000101 111111`: Hebrew (6 bit)
`000110 111111`: Simplified Chinese (12 bit)
`000111 111111`: Traditional Chinese (12 bit)
`001000 111111`: Japanese (Wabun Japanese text in Morse code character set) (6 bit)
`001001 111111`: Japanese (Hiragana, Katakana, and Kanji) (12 bit)
`001010 111111`: Hangul (6 bit)
`001011 111111`: Devanagari (12 bit)

It is quite likely that embedded applications will only support subsets of these characters, for example, Hiragana/Katakana only for Japanese.

# Golay Matrix and Encoding/Decoding

The Golay matrix is for the extended (24,12,8) Golay code. The code is implemented using the perfect (23,11,7) Golay code with an extra parity bit included. This can be implemented by multiplying the 12-bit code word with the Golay matrix to obtain the 12-bit Golay parity check word. The Golay matrix is its own inverse, so that applying the matrix to a 12-bit word, and then again to the result, yields the original word. The implementation of the Golay coding is in the code Appendix.

The encoded word is simply the 12-bit word to be send with the 12-bit parity code prepended to it. The Golay code can correct up to 3 bit errors, and so to decode every possible error up to 3 bits, this code performs the following:

1. Check to see if all three error bits are in the sent parity bits by calculating the parity code of the sent 12-bit word and seeing if there are three or fewer difference bits between the sent parity and the calculated parity. If so, the sent 12-bit word is correct.

2. Check to see if all three errors are in the sent 12-bit word. Calculate the 12-bit word that would be obtained with the given parity code and see if there are three or fewer difference bits between the sent word and the calculated word. If so, the calculated word is correct.

3. Try flipping every bit in the sent 12-bit word and see if there are 2 or fewer errors between the calculated parity and the sent parity. If so, we know which bit of the 12-bit word is wrong and it is corrected.

4. Try flipping every bit in the sent parity code and see if there are 2 or fewer errors between the calculated 12-bit word and the sent word. If so, we know which bits are wrong in the sent word and it is corrected.

5. Otherwise, the error is uncorrectable or undetectable.

# Golay Implementation Example

```
const uint16_t golay_matrix[12] =
{
    0b110111000101,
    0b101110001011,
    0b011100010111,
    0b111000101101,
    0b110001011011,
    0b100010110111,
    0b000101101111,
    0b001011011101,
    0b010110111001,
    0b101101110001,
    0b011011100011,
    0b111111111110
};

uint16_t golay_mult(uint16_t wd_enc)
{
   uint16_t enc = 0;
   uint8_t i;
   for (i=12;i>0;)
   {
      i--;
      if (wd_enc & 1) enc ^= golay_matrix[i];
      wd_enc >>= 1;
   }
   return enc;
}

uint8_t golay_hamming_weight_16(uint16_t n)
{
  uint8_t s = 0, v;
  v = (n >> 8) & 0xFF;
  while (v)
  {
      v = v & (v - 1);
      s++;
  }
  v = n & 0xFF;
  while (v)
  {
      v = v & (v - 1);
      s++;
  }
  return s;
}

uint32_t golay_encode(uint16_t wd_enc)
{
  uint16_t enc = golay_mult(wd_enc);
  return (((uint32_t)enc) << 12) | wd_enc;
}

uint16_t golay_decode(uint32_t codeword, uint8_t *biterrs)
{
  uint16_t enc = codeword & 0xFFF;
  uint16_t parity = codeword >> 12;
  uint8_t i, biterr;
  uint16_t syndrome, parity_syndrome;

  /* if there are three or fewer errors in the parity bits, then
     we hope that there are no errors in the data bits, otherwise
     the error is uncorrected */
  syndrome = golay_mult(enc) ^ parity;
  biterr = golay_hamming_weight_16(syndrome);
  if (biterr <= 3)
  {
     *biterrs = biterr;
     return enc;
  }
```

```
  /* check to see if the parity bits have no errors */
  parity_syndrome = golay_mult(parity) ^ enc;
  biterr = golay_hamming_weight_16(parity_syndrome);
  if (biterr <= 3)
  {
     *biterrs = biterr;
     return enc ^ parity_syndrome;
  }

  /* we flip each bit of the data to see if we have two or fewer errors */
  for (i=12;i>0;)
  {
      i--;
      biterr = golay_hamming_weight_16(syndrome ^ golay_matrix[i]);
      if (biterr <= 2)
      {
          *biterrs = biterr+1;
          return enc ^ (((uint16_t)0x800) >> i);
      }
  }

  /* we flip each bit of the parity to see if we have two or fewer errors */
  for (i=12;i>0;)
  {
      i--;
      uint16_t par_bit_synd = parity_syndrome ^ golay_matrix[i];
      biterr = golay_hamming_weight_16(par_bit_synd);
      if (biterr <= 2)
      {
          *biterrs = biterr+1;
          return enc ^ par_bit_synd;
      }
  }
  return 0xFFFF;   /* uncorrectable error */
}
```

# Implementation of the Finite Impulse Reponse (FIR) filters

The following describes a method of implementing the FIR filters that can be performed with only two multiplications per sample per filter. The FIR filter prevents cumulative error from precision loss. Therefore low precision multiplication operations (even 8 by 8 bit or 16 by 16 bit multiplication operations) can be used.

The following approach is used. An analog anti-aliasing filter can be applied before the signal from the mixer is samples by an analog-to-digital converter (ADC). This removes harmonics that do not need to be filtered in software, making the application of a quadrature demodulator and low-pass filter relatively easy.

The filters are driven by a master sample clock which is typically the sample rate of the ADC. This frequency is denoted here by *f* and the corresponding sampling period *T=1/f*. Each period *T*, the ADC is sampled as a sample `x[n]`, usually as the first operation in an interrupt service routine to apply the filter. In-phase and quadrature filters must be applied to the data stream. Because of this, the filter length *N* must be divisible by four. The recursive FIR filter formula for a filter of length *m* is

$y_i[n] = y_i[n-1] + I_m[n \bmod m](x[n] - x[n-N])$

$y_q[n] = y_q[n-1] + Q_m[n \bmod m](x[n] - x[n-N])$

A circular buffer of samples of length *N* may be used to hold previous samples. Because the same value is subtracted at the beginning and end of the filter, there is no cumulative error. An approximation to the magnitude of the signal may be calculated as

$m_r[n] = \max(\text{abs}(y_i[n]),\text{abs}(y_q[n])) + \text{floor}[(\text{abs}(y_i[n])+\text{abs}(y_q[n])) / 2]$

The divide by two operation is an unsigned right shift by one bit operation.

The 16-bit representations of the filters used in SCAMP are as follows:

$I_8[n]$={ 16384,-11585,0,11585,-16384,11585,0,-11585 } (750 Hz)

$Q_8[n]$={ 0,11585,-16384,11585,0,-11585,16384,-11585 }

$I_{12}[n]$={ 16384,-8192,-8192,16384,-8192,-8192,16384,-8192,-8192,16384,-8192,-8192 } (667 Hz)

$Q_{12}[n]$={ 0,14189,-14189,0,14189,-14189,0,14189,-14189,0,14189,-14189 }

$I_{16}[n]$={ 16384,-6270,-11585,15137,0,-15137,11585,6270,-16384,6270,11585,-15137,0,15137,-11585,-6270 } (625 Hz)

$Q_{16}[n]$={ 0,15137,-11585,-6270,16384,-6270,-11585,15137,0,-15137,11585,6270,-16384,6270,11585,-15137 }

$I_{20}[n]$={ 16384,-5063,-13255,13255,5063,-16384,5063,13255,-13255,-5063,16384,-5063,-13255,13255,5063,-16384,5063,13255,-13255,-5063 } (600 Hz)

$Q_{20}[n]$={ 0,15582,-9630,-9630,15582,0,-15582,9630,9630,-15582,0,15582,-9630,-9630,15582,0,-15582,9630,9630,-15582 }

$I_{24}[n]$={ 16384,-4240,-14189,11585,8192,-15826,0,15826,-8192,-11585,14189,4240,-16384,4240,14189,-11585,-8192,15826,0,-15826,8192,11585,-14189,-4240 } (583 Hz)

$Q_{24}[n]$={ 0,15826,-8192,-11585,14189,4240,-16384,4240,14189,-11585,-8192,15826,0,-15826,8192,11585,-14189,-4240,16384,-4240,-14189,11585,8192,-15826 }

Two of these filters are mutually orthogonal if a filtering period is taken that is the least common multiple of the length of the filter. For example, the filters of length 12 and 20 have a least common multiple of 60, so *N* must be a multiple of 60 for the filters to be orthogonal. Similarly, the filters 8 and 24 have a least common multiple of 24, so *N* must be a multiple of 24 for the filters to be orthogonal. The symbol period is then a multiple of *N* samples for the orthogonality to be preserved.

The standard sample rate for SCAMP is 2000 Hz, so that the filter 8 length frequency corresponds to 750 Hz, the filter 12 length frequency corresponds to 667 Hz, the filter 16 length frequency corresponds to 625 Hz, the filter 20 length frequency corresponds to 600 Hz, and the filter 24 length frequency corresponds to 583 Hz.

# Standard SCAMP Modes and Variable Frequency Oscillator (VFO) frequency selection

SCAMP is intended to be used with both double sideband (DSB) transceivers and single sideband (SSB) transceivers in the upper sideband. To accommodate this, the following conventions are used.

For direct conversion double sideband transceivers (such as the RFBitBanger), there is an ambiguity because a SCAMP signal may be decoded either on the upper or lower sideband. For SCAMP OOK modes, there is a single frequency $f_{mark}$, while for FSK modes there are two frequencies $f_{mark}$ and $f_{space}$. If the VFO is set to a frequency $f_{vfo}$, for OOK modes the transmitted SCAMP signal mark frequency is $f_{vfo}$ while for FSK modes, the mark transmission is at the frequency $f_{vfo}+(f_{mark}-f_{space})/2$ and the space transmission is at the frequency $f_{vfo}+(f_{space}-f_{mark})/2$ (note the difference in order) so that the mark and space frequencies are symmetrically separated about $f_{vfo}$. When receiving, the local oscillator (LO) is set to $f_{vfo}-f_{mark}$ for OOK modes, and $f_{vfo}-(f_{space}+f_{mark})/2$ for FSK modes. Therefore the beat frequency for the mark tone is at $f_{mark}$ and the beat frequency for the space tone (for FSK modes) is $f_{space}$. This will place the mark and space beat frequencies aligned with the digital filters so that the audio sampling can demodulate them. Note that if the VFO is tuned above a received SCAMP transmission center frequency by $f_{mark}$ (OOK) or $(f_{space}+f_{mark})/2$ (FSK) (rather than below as would normally occur), reception will occur successfully on the lower sideband rather than the upper sideband, however, response transmissions will be above the frequency at which the other station expects the response by a frequency of $2f_{mark}$ (OOK) or $(f_{space}+f_{mark})$ (FSK). Therefore an operator of the DSB radio must ensure that the upper sideband is selected by tuning the VFO such that the tone frequency of the receiving station decreases as the VFO frequency is increased.

For single sideband transceivers operated in the upper sideband (USB), the choice of VFO frequency depends on the type of modem used. A software modem implemented on a personal computer or mobile phone with sufficient signal processing power can dynamically change filter frequencies and/or transmitted signal frequencies easily and so can accommodate processing the SCAMP signals at any frequency within the USB passband (usually 1.5 -3.0 kHz wide), and furthermore, frequently such software modems may decode several received signals simultaneously within the USB passband. For modems without much processing power (for example ATMEGA328P-based hardware), however, the audio signal must be aligned with digital filter frequencies programmed into the SCAMP modem. The VFO of the SSB transceiver must then be set to place the received SCAMP frequencies to be aligned with the digital filter frequencies within the USB passband. This would typically be accomplished by tuning the VFO of the transceiver with a signal strength indicator on the modem determining when the received signal is aligned with the digital filter frequencies.

The following are the standard modes for SCAMP. The sample clock signal is 2 kHz, so one sample corresponds to 0.5 ms. For FSK modes, the MARK and SPACE frequencies are interchangeable. Note that the tuning sensitivity is approximately ¼ to 1/3 the symbol rate, so for example, the SCAMP FSK SLOW mode with a symbol rate of 16.7 bits/second needs to be tuned within 2-4 Hz to be properly decoded. The SCAMP FSK VSLW mode is even more stringent, with tuning within 1-2 Hz for the most reliable decoding. This is a reason why a method of easily and accurately aligning transmitting and receiver LO is needed.

SCAMP OOK:
MARK IF = 625 Hz (length 16 filter)
Symbol length *N*=64 samples (32 ms, 31.25 bits per second, 25.0 WPM)

SCAMP OOK SLOW:
MARK IF = 625 Hz (length 16 filter)
Symbol length *N*=144 samples (72 ms, 13.89 bits per second, 11.1 WPM))

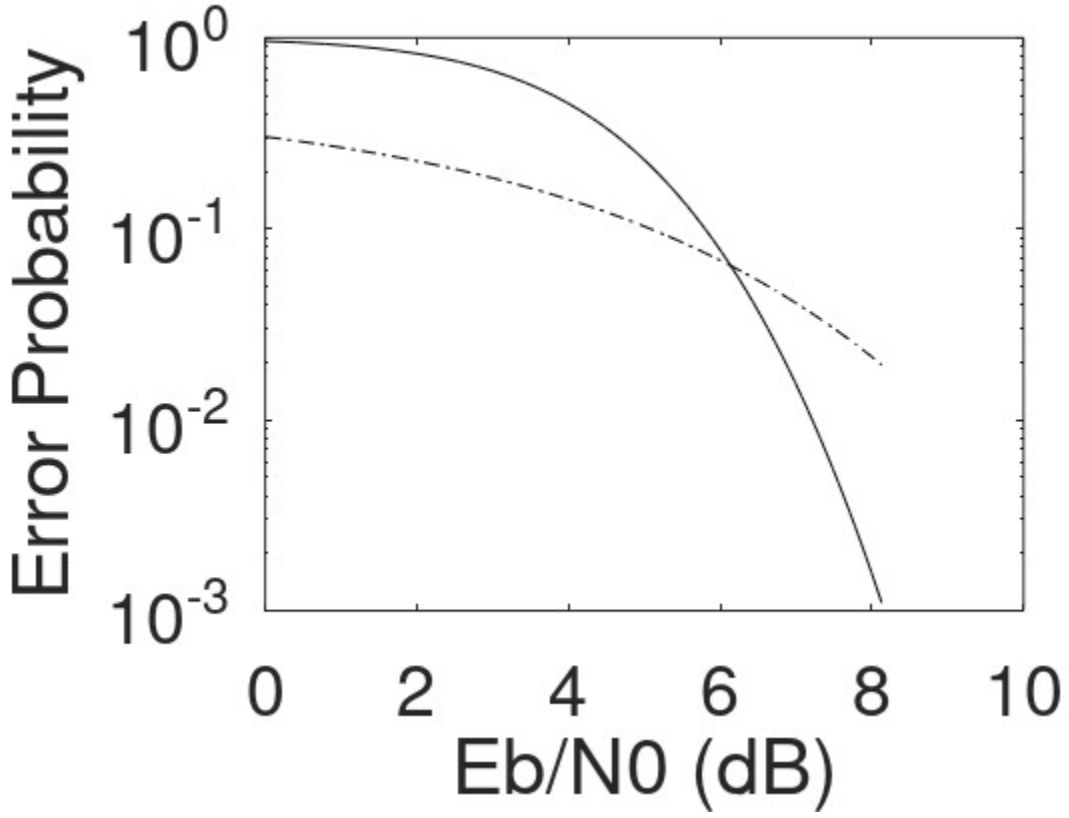
SCAMP FSK:
SPACE IF=600 Hz (length 20 filter), MARK IF=667 Hz (length 12 filter).
Symbol length *N*=60 samples (30 ms, 33.3 bits per second, 26.6 WPM).

SCAMP FSK FAST:
SPACE IF=583 Hz (length 24 filter), MARK IF=750 (length 8 filter).
Symbol length *N*=24 samples (12 ms, 83.3 bits per second, 66.7 WPM).

SCAMP FSK SLOW:
SPACE IF=625 Hz (length 16 filter), MARK IF=667 Hz (length 12 filter).
Symbol length *N*=144 samples (72 ms, 13.89 bits per second, 11.1 WPM).

SCAMP FSK VERY SLOW:
SPACE IF=312.5 Hz (length 16 filter), MARK IF=333.3 Hz (length 12 filter,
1000 Hz sample rate achieved by summing together every two samples at 2000 Hz).
Symbol length *N*=144 samples (144 ms, 6.94 bits per second, 5.56 WPM).

## Error probability and frame probabilty vs. Eb/N0




Dotted curve, bit error rate (BFSK).
Solid curve, frame error rate (probability of error in 30 bit frame).

Conversion between 2500 Hz SNR_per_bit, SNR@2500 Hz and Eb/N0:

Eb/N0 = SNR_per_bit / k (k=1 for BFSK, k=2 for 4FSK, k=3 for 8FSK, etc.)

SNR@2500Hz = SNR_per_bit (bits per second / 2500 Hz)

For $10^{-3}$ frame error, the Eb/N0 required is 8.1 dB, and the SNR_per_bit is 8.1 dB.

For SCAMP FSK FAST, the SNR@2500Hz required is 8.1 dB – 10 log(83.3/2500) = -6.67 dB
For SCAMP FSK, the SNR@2500Hz required is 8.1 dB – 10 log(26.6/2500) = -11.6 dB
For SCAMP FSK SLOW, the SNR@2500Hz required is 8.1 dB – 10 log(13.9/2500) = -14.5 dB
For SCAMP FSK VERY SLOW, the SNR@2500Hz required is 8.1 dB – 10 log(6.9/2500) = -17.5 dB

# A crude implementation of the FIR filters

If one does not have a hardware integer multiplication unit available, the processor may be too slow even to apply two multiplications per sample per filter. In this case, an approximation that uses only adds and subtracts may be used. These filters admit odd harmonics of the fundamental frequency and so analog filtering is required to reduce these to prevent aliasing. Note that the length 8 and length 24 filters are not orthogonal, and so this method can not be used for the SCAMP FSK FAST mode. Here are the filters in FIR form:

$I_8[n]=\{ 1,1,1,1,-1,-1,-1,-1 \}$ (250 Hz)

$Q_8[n]=\{ 1,1,-1,-1,-1,-1,1,1 \}$

$I_{12}[n]=\{ 1,1,1,1,1,1,-1,-1,-1,-1,-1,-1 \}$ (167 Hz)

$Q_{12}[n]=\{ 1,1,1,-1,-1,-1,-1,-1,-1,1,1,1 \}$

$I_{16}[n]=\{ 1,1,1,1,1,1,1,1,-1,-1,-1,-1,-1,-1,-1,-1, \}$ (125 Hz)

$Q_{16}[n]=\{ 1,1,1,1,-1,-1,-1,-1,-1,-1,-1,-1,1,1,1,1 \}$

$I_{20}[n]=\{ 1,1,1,1,1,1,1,1,1,1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1 \}$ (100 Hz)

$Q_{20}[n]=\{ 1,1,1,1,1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,1,1,1,1,1,1 \}$

$I_{24}[n]=\{ 1,1,1,1,1,1,1,1,1,1,1,1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1 \}$ (83.3 Hz)

$Q_{24}[n]=\{ 1,1,1,1,1,1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,1,1,1,1,1,1 \}$

These have the simple form that is more convenient for implementation:

$I_m[n] = 1 \text{ for } 0 \le n < m/2$

$I_m[n] = -1 \text{ for } m/2 \le n < m$

$Q_m[n] = 1 \text{ for } 0 \le n < m/4 \text{ or } 3m/4 \le n < m$

$Q_m[n] = -1 \text{ for } m/4 \le n < 3m/4$

Note that the offsets of the filters are 500 Hz lower than the offsets in the integer multiplication versions of the FIR filters. To match up the send and receive frequencies between a transmitter using the higher offsets, a receive incremental tuning (RIT) method could be used to offset the receive frequency from the VFO frequency by 500 Hz.

# Other candidate synchronization code words

During the search for code words, these other candidate 30-bit code words were discovered. The chosen code word was decided by having the maximum number of preceding ones so it would be part of a constant carrier signal. These code words all have a maximum of two complement bits for any shift of the word and a correlation magnitude of 4. These were found by exhaustive computer search.

```
syncword: 1,0,0,0,0,1,1,1,0,1,0,0,0,1,1,0,0,1,1,1,0,1,0,0,1,0,0,0,0,0
autocor: 30 3 -2 -3 0 -1 -2 3 0 -3 -2 -1 4 1 2 -1 -4 1 2 3 -4 1 -2 -1 4 3 2 1 0 -1

syncword: 1,0,0,0,0,1,1,1,1,0,1,1,1,1,1,0,0,1,0,1,0,0,1,1,1,0,1,1,0,0
autocor: 30 3 -2 -1 -4 3 -2 1 -4 -3 -4 -1 4 3 4 -3 2 3 -2 3 -2 -3 -2 -3 0 -1 2 3 0 -1

syncword: 1,0,0,0,0,1,1,1,1,1,0,0,1,1,1,0,1,1,1,0,1,1,1,0,1,0,0,1,0,0
autocor: 30 3 -2 -1 2 -3 0 1 4 3 -2 -3 -2 1 0 1 -2 -3 0 -3 -4 -1 2 -3 2 1 0 3 0 -1

syncword: 1,0,0,0,0,1,1,1,1,1,0,1,1,1,1,0,0,1,1,0,1,1,0,1,1,0,0,0,1,0
autocor: 30 3 -4 3 -2 -3 4 3 -4 -1 -2 1 4 -1 0 1 -2 -1 -2 -3 -4 -1 -2 3 2 1 0 -1 2 -1

syncword: 1,0,1,1,1,0,0,1,0,0,1,0,0,1,1,0,0,0,0,1,0,0,0,0,0,1,1,1,1,0
autocor: 30 3 -4 3 -2 -3 4 3 -4 -1 -2 1 4 -1 0 1 -2 -1 -2 -3 -4 -1 -2 3 2 1 0 -1 2 -1

syncword: 1,0,1,1,1,0,1,0,0,1,0,0,0,0,1,0,0,0,1,1,0,0,0,0,0,1,1,1,0,0
autocor: 30 3 -2 -3 0 3 2 3 2 1 -2 -1 4 -1 -2 -1 4 -1 -2 -1 -4 1 0 3 0 -1 -2 1 0 -1

syncword: 1,1,0,0,0,1,1,1,1,0,1,0,0,0,1,0,0,0,1,1,0,0,1,0,0,1,0,0,0,0
autocor: 30 3 -4 -1 2 -1 0 1 4 1 -4 3 4 3 2 -1 -4 3 2 -1 -2 -3 -2 -3 2 3 0 -1 -2 -1

syncword: 1,1,0,0,0,1,1,1,1,1,0,0,1,1,1,0,1,1,1,1,0,1,1,0,1,0,0,0,1,0
autocor: 30 3 -2 -3 0 3 2 3 2 1 -2 -1 4 -1 -2 -1 4 -1 -2 -1 -4 1 0 3 0 -1 -2 1 0 -1

syncword: 1,1,0,0,1,0,0,0,1,1,0,1,0,1,1,0,0,0,0,0,1,0,0,0,0,1,1,1,1,0
autocor: 30 3 -2 -1 -4 3 -2 1 -4 -3 -4 -1 4 3 4 -3 2 3 -2 3 -2 -3 -2 -3 0 -1 2 3 0 -1

syncword: 1,1,0,1,1,0,1,0,0,0,1,0,0,0,1,0,0,0,1,1,0,0,0,0,0,1,1,1,1,0
autocor: 30 3 -2 -1 2 -3 0 1 4 3 -2 -3 -2 1 0 1 -2 -3 0 -3 -4 -1 2 -3 2 1 0 3 0 -1

syncword: 1,1,0,1,1,0,1,1,1,1,0,0,1,1,1,0,0,0,1,0,1,1,1,1,0,1,0,0,0,0
autocor: 30 3 0 -1 -4 3 -2 -3 0 3 -2 1 2 3 4 -3 -2 3 0 1 -4 -1 2 -3 0 -1 -2 -1 -2 -1

syncword: 1,1,1,0,0,1,1,0,1,0,1,0,1,1,0,1,1,1,1,1,0,1,1,1,1,0,0,0,0,0
autocor: 30 3 4 1 0 3 2 1 -2 -3 -4 3 -4 -1 -2 -1 4 3 2 -1 -2 1 4 1 0 -1 -2 -3 -2 -1

syncword: 1,1,1,1,0,1,0,0,0,0,1,0,1,1,1,0,0,0,1,1,0,0,0,0,1,0,0,1,0,0
autocor: 30 3 0 -1 -4 3 -2 -3 0 3 -2 1 2 3 4 -3 -2 3 0 1 -4 -1 2 -3 0 -1 -2 -1 -2 -1

syncword: 1,1,1,1,0,1,1,0,1,1,0,0,1,1,1,0,1,1,1,0,1,0,0,0,0,1,1,1,0,0
autocor: 30 3 -4 -1 2 -1 0 1 4 1 -4 3 4 3 2 -1 -4 3 2 -1 -2 -3 -2 -3 2 3 0 -1 -2 -1

syncword: 1,1,1,1,1,0,0,0,0,1,0,0,0,0,0,1,0,0,1,0,1,0,1,0,0,1,1,0,0,0
autocor: 30 3 4 1 0 3 2 1 -2 -3 -4 3 -4 -1 -2 -1 4 3 2 -1 -2 1 4 1 0 -1 -2 -3 -2 -1

syncword: 1,1,1,1,1,0,1,1,0,1,0,0,0,1,1,0,0,1,1,1,0,1,0,0,0,1,1,1,1,0
autocor: 30 3 -2 -3 0 -1 -2 3 0 -3 -2 -1 4 1 2 -1 -4 1 2 3 -4 1 -2 -1 4 3 2 1 0 -1
```

# Software License for Code

The code examples in this document are licensed under the zlib license, the text of which is included below.

```
/*
 * Copyright (c) 2021 Daniel Marks

This software is provided 'as-is', without any express or implied
warranty. In no event will the authors be held liable for any damages
arising from the use of this software.

Permission is granted to anyone to use this software for any purpose,
including commercial applications, and to alter it and redistribute it
freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not
   claim that you wrote the original software. If you use this software
   in a product, an acknowledgment in the product documentation would be
   appreciated but is not required.
2. Altered source versions must be plainly marked as such, and must not be
   misrepresented as being the original software.
3. This notice may not be removed or altered from any source distribution.
 */
```