Simple Conversational Amateur Messaging Protocol (SCAMP)

by Daniel Marks, KW4TI

Draft v0.3 2021-11-05

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 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 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 is 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 to 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.

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. These are reserved for future use.

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

```
11 1110 1101 0001 1001 1101 0001 1110 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. 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 one bit. To do this, a sequence of 20 or more one bit intervals are sent at the beginning of the transmission, followed by 3 or 4 zero bit intervals, then two one bit 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 1100 0011
11 1110 1101 0001 1001 1101 0001 1110
```

The initial 24 one bit intervals provides sufficient time for the receiver to begin to receive, and if the FSK mode is used, the receiver uses the received frequency to denote the one bit.

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)	! (exclamation 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 of the code page 437 of the original IBM character set, these may be sent using the a data Golay code, which is

MSB 1111 XXXX XXXX LSB

where XXXX XXXX is the 8-bit code page 437 representation of the symbol. 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.

```
const uint8_t ecc_6bit_codesymbols[60] = {'\0',   '\b',   '\r',    ' ',    '!',   0x22,   0x27,    '(',
                                           ')',    '*',    '+',    ',',    '-',    '.',    '/',    '0',
                                           '1',    '2',    '3',    '4',    '5',    '6',    '7',    '8',
                                           '9',    ':',    ';',    '=',    '?',    '@',    'A',    'B',
                                           'C',    'D',    'E',    'F',    'G',    'H',    'I',    'J',
                                           'K',    'L',    'M',    'N',    'O',    'P',    'Q',    'R',
                                           'S',    'T',    'U',    'V',    'W',    'X',    'Y',    'Z',
                                           0x5C,   '^',    '`',    '~' };
                                           /* Last 4 symbols can be interpreted as diacritical marks,
0x5C is diaeresis/umlaut */
                                           /* 0x27 can be interpreted as acute diacritical mark */


/* in memory buffer encoder / decoder callbacks */
typedef struct _eccfr_code_word_put_mem_buf_struct
{
  uint16_t *code_word_array;
  uint8_t cur_word;
  uint8_t max_words;
} eccfr_code_word_put_mem_buf_struct;

void eccfr_code_word_put_mem_buf(uint16_t code, void *st)
{
  eccfr_code_word_put_mem_buf_struct *s = (eccfr_code_word_put_mem_buf_struct *) st;
  if (s->cur_word < s->max_words)
     s->code_word_array[s->cur_word++] = code;
}

typedef struct _eccfr_code_word_get_mem_buf_struct
{
  uint16_t *code_word_array;
  uint8_t cur_word;
  uint8_t max_words;
} eccfr_code_word_get_mem_buf_struct;

uint16_t eccfr_code_word_get_mem_buf(void *st)
{
  eccfr_code_word_get_mem_buf_struct *s = (eccfr_code_word_get_mem_buf_struct *) st;
  if (s->cur_word < s->max_words)
     return s->code_word_array[s->cur_word++];
  return 0xFFFF;
}

/*  The input word has 24 bits.  The output word has 30 bits, with bits
    1, 5, 9, 13, 17, 21 preceded by its complement bit inserted into
    the word */

uint32_t eccfr_add_reversal_bits(uint32_t codeword)
{
  uint32_t outword = 0;
  uint8_t i;

  for (i=0;i<6;i++)
  {
      if (i>0)
          outword = (outword << 5);
      codeword = (codeword << 4);
      uint8_t temp = (codeword >> 24) & 0x0F;
      outword |= (temp | (((temp & 0x08) ^ 0x08) << 1));
  }
  return outword;
}

/* remove complement bits from inside 30 bit word to yield 24 bit Golay
   code word */

uint32_t eccfr_remove_reversal_bits(uint32_t outword)
{
  uint32_t codeword = 0;
  uint8_t i;

  for (i=0;i<6;i++)
  {
      if (i>0)
      {
          outword = (outword << 5);
          codeword = (codeword << 4);
      }
      uint8_t temp = (outword >> 25) & 0x0F;
      codeword |= temp;
  }
  return codeword;
}

/* decode 12 bit code words to 8 bit bytes */
uint8_t eccfr_code_words_to_bytes(eccfr_code_word_get ecwg, void *st, uint8_t *bytes, uint8_t
max_bytes)
{
    uint8_t cur_byte = 0;
    uint16_t last_code = 0xFFFF;
    while (cur_byte < max_bytes)
    {
        uint16_t code = ecwg(st);
        if (code == 0xFFFF) break;
        last_code = code;
        if ((code & 0xF00) == 0xF00)
        {
            bytes[cur_byte++] = code & 0xFF;
            continue;
        }
        if (code == last_code) continue;
        uint16_t code1 = (code & 0x3F);
        uint16_t code2 = ((code >> 6) & 0x3F);
        if ((code1 != 0) && (code1 < (sizeof(ecc_6bit_codesymbols)/sizeof(uint8_t))))
            bytes[cur_byte++] = ecc_6bit_codesymbols[code1];
        if ((code2 != 0) && (code2 < (sizeof(ecc_6bit_codesymbols)/sizeof(uint8_t))) && (cur_byte <
max_bytes))
            bytes[cur_byte++] = ecc_6bit_codesymbols[code2];
    }
    return cur_byte;
}

/* encode 8 bit bytes as 12 bit code words, always encoding as 8-bit raw */
void eccfr_raw_bytes_to_code_words(uint8_t *bytes, uint8_t num_bytes, eccfr_code_word_put ecwp, void
*st)
{
  uint8_t cur_byte = 0;
  while (cur_byte < num_bytes)
  {
     uint8_t b = bytes[cur_byte++];
     ecwp( ((uint16_t)(0xF00)) | b, st );
  }
}

/* convert character to 6-bit code if it exists */
/* should probably replace this with an inverse look up table later */
uint8_t eccfr_find_code_in_table(uint8_t c)
{
    uint8_t i;
    if ((c >= 'a') && (c <= 'z')) c -= 32;
       if (c =='\n') c = '\r';
       if (c == 127) c = '\b';
    for (i=1;i<(sizeof(ecc_6bit_codesymbols)/sizeof(uint8_t));i++)
        if (ecc_6bit_codesymbols[i] == c) return i;
    return 0xFF;
}

/* encode 8 bit bytes to 12 bit code words.
   code words that correspond to a 6-bit symbols are encoded as 6 bits.
   otherwise they are encoded as an 8-bit binary raw data word.
   If a byte that can be encoded as a 6 bit symbol precedes one that can
   not be encoded as a 6 bit symbol, and there is an extra symbol slot
   in the current word, fill it with a zero. */
void eccfr_bytes_to_code_words(uint8_t *bytes, uint8_t num_bytes, eccfr_code_word_put ecwp, void *st)
{
  uint8_t cur_byte = 0;
  uint16_t last_code_word = 0;
  uint16_t code_word;
  while (cur_byte < num_bytes)
  {
     uint8_t b = bytes[cur_byte++];
     uint8_t code1 = eccfr_find_code_in_table(b);
     if (code1 == 0xFF)
     {
         code_word = ((uint16_t)(0xF00)) | b;
     } else
     {
         code_word = (uint16_t)code1;
         if (cur_byte < num_bytes)
         {
            b = bytes[cur_byte];
            uint8_t code2 = eccfr_find_code_in_table(b);
            if (code2 != 0xFF)
            {
               code_word |= (((uint16_t)code2) << 6);
               cur_byte++;
            }
         }
         if (code_word == last_code_word)
             ecwp(0, st);
     }
     ecwp(code_word, st);
     last_code_word = code_word;
  }
}
```

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 following code below is an example of the Golay implementation:

```
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 */
}
```

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.

Implementation of the Finite Impulse Reponse (FIR) filters

The following describes a method of implementing the FIR filters that is suitable for small microcontrollers such as the ATMEGA328P.   16- and 32-bit multiplication are computationally intensive operations on an 8-bit microcontroller and are generally avoided. Because of this, these are avoided, and the operation of OOK and FSK modulations can be designed to mostly require additions, subtractions, comparisons, and bit shifts, all of which tend to be economical on small, energy efficient processors.

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. In particular, a quadrature demodulator can be implemented by using adds and subtracts from a counter oscillator, with the adds and subtracts for the in phase quadrature phase subtracts being delayed a quarter cycle from the in-phase add or subtract. A low-pass filter can be implemented using a moving average FIR filter, with the length of the filter designed to place the frequency nulls of the filter in an adjacent frequency channel to be rejected.

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, usually as the first operation in an interrupt service routine to apply the filter. A quadrature filter to be applied must have a period that is a multiple of *4T* so that the number of samples per period is *4N*, where *N* is an integer.   A counter tracks the current phase of the quadrature demodulator, counting from *0* to *4N-1* before resetting to *0* again.   There are accumulators for the in-phase and quadrature components of the filter.

If the counter is between *0* and *2N-1*, the current sample is subtracted from the in-phase component, and if the counter is between *2N* and *4N-1* the in-phase component is added. If the counter is between *N* and *3N-1* the current sample is subtracted from the quadrature-phase component, and if the counter is between *0* and *N-1* or *3N* and *4N-1*, it is added.

Furthermore, to implement low pass filtering, a moving average filter is applied to the samples. A circular buffer is maintained with the samples. Each frequency with period *4N* is averaged over *M* cycles, so that the total FIR filter length is *4NM*. The sample in the buffer that is lagged *4NM* samples behind the current sample is applied to the current in-phase and quadrature-phase components with the opposite sign as the current sample.

As an example, we consider a typical case of *f*=2000 Hz and T=0.5 ms. Typical frequencies to be filtered correspond to periods of 8 (N=2), 12 (N=3), 16 (N=4), 20 (N=5), and 24 (N=6) samples, or 250 Hz, 167 Hz, 125 Hz, 100 Hz, and 83 Hz respectively.

For OOK operation, the center frequency on which data is received could be chosen to be the N=3 channel, for example. The periods 8, 12, and 16 have a least common multiple of 48. The moving average filter corresponding to this 48 period interval corresponds to:

```
++++----++++----++++----++++----++++----++++----  (M=6, N=8, in-phase)
--++++----++++----++++----++++----++++----++++--  (M=6, N=8, quadrature-phase)
++++++------++++++------++++++------++++++------  (M=4, N=12, in-phase)
---++++++------++++++------++++++------++++++---  (M=4, N=12, quadrature-phase)
++++++++--------++++++++--------++++++++--------  (M=3, N=16, in-phase)
----++++++++--------++++++++--------++++++++----  (M=3, N=16, quadrature-phase)
```

These FIR filters are all orthogonal over the interval and therefore achieve the best discrimination between frequency channels for their length. The frequency nulls of each of these channels correspond to the two other channels. This is maintained for any interval that is a multiple of 48 periods. For the N=2, 3, 4, and 6 channels, 96 periods is similarly a common multiple.

For FSK operation, the two frequencies on which data is received could corresponding to the N=3 and N=5 channels. Because these are lengths 12 and 20, respectively, their least common multiple is 60. The FIR filters corresponding to this are:

```
++++++------++++++------++++++------++++++------++++++------  (M=5, N=12, in-phase)
---++++++------++++++------++++++------++++++------++++++---  (M=5, N=12, quadrature-phase)
++++++++++----------++++++++++----------++++++++++----------  (M=3, N=20, in-phase)
-----++++++++++----------++++++++++----------++++++++++-----  (M=3, N=20, quadrature-phase)
```

This FSK operation corresponds to a modulation similar to MSK, however, unlike MSK, it is insensitive to the phase of the carrier at the beginning of the interval. MSK modulation can be seen as alternating between modulating the carrier wave a half cycle or a full cycle over the 60 cycle interval. The orthogonality of the half and full cycle modulations requires that the phase of the half or full cycle be controlled at the beginning of the interval. The FSK modulation used here corresponds to one or two cycles modulated on the carrier over the 60 cycle interval. The orthogonality of one or two cycles over the interval does not depend on the initial phase of the carrier at the beginning of the interval. This is a much simpler to achieve with crude FSK modulation hardware where there could be phase discontinuities occuring when switching between the two FSK frequencies. The disadvantage is that only one quadrature signal of the two that could be used for sending separate data is being used at a time, halving the spectral efficiency and consequently reducing the signal-to-noise ratio.

The code example shows how to implement these FIR filters with the function dsp_new_sample() being called by an interrupt routine with a new sample to update the filters.

```
#define DSPINT_MAX_SAMPLEBUFFER 64
#define DSPINT_MAX_DEMODBUFFER 16
#define DSPINT_PWR_THR_DEF 32

#define DSPINT_AVG_CT_PWR2 9

#define DSPINT_OOK 0
#define DSPINT_OOK2 1
#define DSPINT_FSK 2
#define DSPINT_FSK2 3

#define DSPINT_SYNC_CODEWORD 0b111110110100011001110100011110ul
                              /* 0x3ED19D1E */
```

```c
#define DSPINT_BLANK_CODEWORD 0xAAAAAAAA

#define DSPINT_FRAME_FIFO_LENGTH 8

/* structure for FRAME FIFO */
typedef struct _dspint_frame_fifo
{
    uint8_t   head;
    uint8_t   tail;
    uint32_t  frames[DSPINT_FRAME_FIFO_LENGTH];
} dspint_frame_fifo;

/* this is stuff that is initialized when the modulation mode
   is changed and doesn't change otherwise */
typedef struct _dsp_state_fixed
{
  uint8_t   mod_type;
  uint8_t   buffer_size;
  uint8_t   fsk;

  uint8_t   dly_8;
  uint8_t   dly_12;
  uint8_t   dly_16;
  uint8_t   dly_20;
  uint8_t   dly_24;

  uint8_t   demod_samples_per_bit;
  uint8_t   demod_edge_window;
  uint16_t  power_thr_min;
} dsp_state_fixed;

/* this is the current state of the demodulator and is designed
   so that is might be reset quickly without disturbing the
   dsp_state_fixed state */
typedef struct _dsp_state
{
  uint8_t   slow_samp;
  uint8_t   slow_samp_num;
  uint16_t  total_num;

  uint8_t   sample_no;
  uint8_t   mag_new_sample;
  uint8_t   demod_sample_no;
  uint8_t   edge_ctr;
  uint8_t   next_edge_ctr;
  uint8_t   polarity;
  uint8_t   resync;
  uint8_t   demod_edge_window;

  uint8_t   bitflips_in_phase;
  uint8_t   bitflips_lag;
  uint8_t   bitflips_lead;
  uint8_t   bitflips_ctr;

  uint16_t  edge_thr;
  uint16_t  power_thr;

  uint16_t  ct_average;
  uint32_t  ct_sum;

  int8_t    current_bit_no;
  uint32_t  current_word;

  uint8_t   count_8;
  uint8_t   count_12;
  uint8_t   count_16;
  uint8_t   count_20;
  uint8_t   count_24;

  int16_t  state_i_8;
  int16_t  state_i_12;
  int16_t  state_i_16;
  int16_t  state_i_20;
  int16_t  state_i_24;
  int16_t  state_q_8;
  int16_t  state_q_12;
  int16_t  state_q_16;
  int16_t  state_q_20;
  int16_t  state_q_24;

  uint32_t state_m_8;
  uint32_t state_m_12;
  uint32_t state_m_16;
  uint32_t state_m_20;
  uint32_t state_m_24;

  uint16_t  mag_value_8;
  uint16_t  mag_value_12;
  uint16_t  mag_value_16;
  uint16_t  mag_value_20;
  uint16_t  mag_value_24;
  uint16_t  sample_buffer[DSPINT_MAX_SAMPLEBUFFER];
  int16_t   demod_buffer[DSPINT_MAX_DEMODBUFFER];

  uint16_t  bit_edge_val;
  uint16_t  max_bit_edge_val;
  int16_t   cur_bit;
} dsp_state;

extern dsp_state       ds;
extern dsp_state_fixed df;
extern volatile dspint_frame_fifo dsp_input_fifo;
extern volatile dspint_frame_fifo dsp_output_fifo;

void dsp_new_sample(uint16_t sample);
void dsp_initialize(uint8_t sample_buffer_size);
void dsp_reset_state(void);

void dsp_initialize_frame_fifo(volatile dspint_frame_fifo *dff);
uint8_t dsp_insert_into_frame_fifo(volatile dspint_frame_fifo *dff, uint32_t frame);
uint32_t dsp_remove_from_frame_fifo(volatile dspint_frame_fifo *dff);

dsp_state       ds;
dsp_state_fixed df;

volatile dspint_frame_fifo dsp_input_fifo;
volatile dspint_frame_fifo dsp_output_fifo;

/* initialize frame fifo */
void dsp_initialize_frame_fifo(volatile dspint_frame_fifo *dff)
{
    dff->head = dff->tail = 0;
}

/* insert a frame into the fifo.  this is intended to be interrupt safe */
uint8_t dsp_insert_into_frame_fifo(volatile dspint_frame_fifo *dff, uint32_t frame)
{
    uint8_t next = dff->head >= (DSPINT_FRAME_FIFO_LENGTH - 1) ? 0 : (dff->head+1);
    if (next == dff->tail) return 0;
    dff->frames[next] = frame;
    dff->head = next;
    return 1;
}

/* remove a frame from the fifo.  this is intended to be interrupt safe */
uint32_t dsp_remove_from_frame_fifo(volatile dspint_frame_fifo *dff)
{
    uint32_t frame;
    uint8_t next;
    if (dff->tail == dff->head) return 0xFFFF;
    next = dff->tail >= (DSPINT_FRAME_FIFO_LENGTH - 1) ? 0 : (dff->tail+1);
    frame = dff->frames[next];
    dff->tail = next;
    return frame;
}

/* calculate the hamming weight of a 16 bit integer
```

```
   GCC could use __builtin__popcount() */
uint8_t dsp_hamming_weight_16(uint16_t n)
{
  uint8_t s = 0, v;
  v = (n >> 8) & 0xFF;
  while (v)
  {
      v &= (v - 1);
      s++;
  }
  v = n & 0xFF;
  while (v)
  {
      v &= (v - 1);
      s++;
  }
  return s;
}

/* calculate the hamming weight a 30 bit number for
   to find hamming distance with sync word */
uint8_t dsp_hamming_weight_30(uint32_t n)
{
  uint8_t s = 0, v;
  v = (n >> 24) & 0x3F;
  while (v)
  {
      v &= (v - 1);
      s++;
  }
  v = (n >> 16) & 0xFF;
  while (v)
  {
      v &= (v - 1);
      s++;
  }
  v = (n >> 8) & 0xFF;
  while (v)
  {
      v &= (v - 1);
      s++;
  }
  v = n & 0xFF;
  while (v)
  {
      v &= (v - 1);
      s++;
  }
  return s;
}

void dsp_reset_codeword(void)
{
   ds.current_bit_no = 0;
   ds.current_word = DSPINT_BLANK_CODEWORD;
   ds.bitflips_in_phase = 0;
   ds.bitflips_lag = 0;
   ds.bitflips_lead = 0;
   ds.bitflips_ctr = 0;
}

/* reset state of buffer in case we get some kind
   of a dc offset mismatch */

void dsp_reset_state(void)
{
   memset(&ds,'\000',sizeof(dsp_state));
   dsp_reset_codeword();
   ds.demod_edge_window =  df.demod_edge_window;
   ds.power_thr = df.power_thr_min * 2;
   ds.edge_thr = ds.power_thr;
}

/* initialize the buffer including the signs to be subtracted
```

```
   from the end of the buffer */
void dsp_initialize(uint8_t mod_type)
{
    uint8_t i;
    df.mod_type = mod_type;
    switch (df.mod_type)
    {
        case DSPINT_OOK:  df.buffer_size = 32;
                          df.fsk = 0;
                          break;
        case DSPINT_OOK2: df.buffer_size = 64;
                          df.fsk = 0;
                          break;
        case DSPINT_FSK:  df.buffer_size = 60;
                          df.fsk = 1;
                          break;
        case DSPINT_FSK2: df.buffer_size = 24;
                          df.fsk = 1;
                          break;
    }
    df.dly_8 = (df.buffer_size / 8) * 8;
    df.dly_12 = (df.buffer_size / 12) * 12;
    df.dly_16 = (df.buffer_size / 16) * 16;
    df.dly_20 = (df.buffer_size / 20) * 20;
    df.dly_24 = (df.buffer_size / 24) * 24;
    df.demod_samples_per_bit = df.buffer_size / 4;
    df.power_thr_min = ((uint16_t)df.buffer_size) * DSPINT_PWR_THR_DEF * (df.fsk ? 2 : 1);
    df.demod_edge_window = (df.buffer_size + 6) / 12;
    dsp_reset_state();
}

/* this is an approximation to the sqrt(x^2+y^2) function that approximates the
   circular set as an octagon.  seems to work quite well */
#define SET_DSP_SQRT_APPROX(s,x,y) do { \
   uint16_t ux = (x < 0 ? -x : x); \
   uint16_t uy = (y < 0 ? -y : y); \
   s = ((ux + uy) >> 1)+(uy > ux ? uy : ux); \
} while(0)

/* called by the interrupt routine to update the spectral channels */
/* this is used to update the I & Q of each spectral channel and update
   the magnitude of the signal on each channel.  more channels are used (5)
   than needed (1 or 2) because this will help the operator align the signal into
   the correct channels */
void dsp_new_sample(uint16_t sample)
{
   uint8_t b, prep_sample;
   int16_t fir;

   if (ds.slow_samp_num > 1)
   {
       ds.total_num += sample;
       if ((++ds.slow_samp) >= ds.slow_samp_num)
       {
           sample = ds.total_num;
           ds.slow_samp = 0;
           ds.total_num = 0;
       } else
       {
           ds.mag_new_sample = 0;
           return;
       }
   }

   /* we do this so that the square roots are computed in the interrupt cycle
      before they are used to reduce the worst-case time for the interrupt
      execution */

   prep_sample = ds.count_8 & 0x03;
   ds.mag_new_sample = (prep_sample == 0);
   prep_sample = (prep_sample == 3);

   /* update 8 count I & Q */
   b = (ds.sample_no < df.dly_8) ? (ds.sample_no + df.buffer_size - df.dly_8) : (ds.sample_no - 
df.dly_8);
   fir = sample - ds.sample_buffer[b];
   if (ds.count_8 >= 4)
       ds.state_i_8 += fir;
   else
       ds.state_i_8 -= fir;
   if ((ds.count_8 >= 2) && (ds.count_8 < 6))
       ds.state_q_8 += fir;
   else
       ds.state_q_8 -= fir;
   ds.count_8 = (ds.count_8 >= 7) ? 0 : (ds.count_8 + 1);
   if (prep_sample)
       SET_DSP_SQRT_APPROX(ds.mag_value_8, ds.state_q_8, ds.state_i_8);

   /* update 12 count I & Q */
   b = (ds.sample_no < df.dly_12) ? (ds.sample_no + df.buffer_size - df.dly_12) : (ds.sample_no - 
df.dly_12);
   fir = sample - ds.sample_buffer[b];
   if (ds.count_12 >= 6)
       ds.state_i_12 += fir;
   else
       ds.state_i_12 -= fir;
   if ((ds.count_12 >= 3) && (ds.count_12 < 9))
       ds.state_q_12 += fir;
   else
       ds.state_q_12 -= fir;
   ds.count_12 = (ds.count_12 >= 11) ? 0 : (ds.count_12 + 1);
   if (prep_sample)
       SET_DSP_SQRT_APPROX(ds.mag_value_12, ds.state_q_12, ds.state_i_12);

   /* update 16 count I & Q */
   b = (ds.sample_no < df.dly_16) ? (ds.sample_no + df.buffer_size - df.dly_16) : (ds.sample_no - 
df.dly_16);
   fir = sample - ds.sample_buffer[b];
   if (ds.count_16 >= 8)
       ds.state_i_16 += fir;
   else
       ds.state_i_16 -= fir;
   if ((ds.count_16 >= 4) && (ds.count_16 < 12))
       ds.state_q_16 += fir;
   else
       ds.state_q_16 -= fir;
   ds.count_16 = (ds.count_16 >= 15) ? 0 : (ds.count_16 + 1);
   if (prep_sample)
        SET_DSP_SQRT_APPROX(ds.mag_value_16, ds.state_q_16, ds.state_i_16);

   /* update 20 count I & Q */
   b = (ds.sample_no < df.dly_20) ? (ds.sample_no + df.buffer_size - df.dly_20) : (ds.sample_no - 
df.dly_20);
   fir = sample - ds.sample_buffer[b];
   if (ds.count_20 >= 10)
       ds.state_i_20 += fir;
   else
       ds.state_i_20 -= fir;
   if ((ds.count_20 >= 5) && (ds.count_20 < 15))
       ds.state_q_20 += fir;
   else
       ds.state_q_20 -= fir;
   ds.count_20 = (ds.count_20 >= 19) ? 0 : (ds.count_20 + 1);
   if (prep_sample)
        SET_DSP_SQRT_APPROX(ds.mag_value_20, ds.state_q_20, ds.state_i_20);

   /* update 24 count I & Q */
   b = (ds.sample_no < df.dly_24) ? (ds.sample_no + df.buffer_size - df.dly_24) : (ds.sample_no - 
df.dly_24);
   fir = sample - ds.sample_buffer[b];
   if (ds.count_24 >= 12)
       ds.state_i_24 += fir;
   else
       ds.state_i_24 -= fir;
   if ((ds.count_24 >= 6) && (ds.count_24 < 18))
       ds.state_q_24 += fir;
   else
```

```
        ds.state_q_24 -= fir;
    ds.count_24 = (ds.count_24 >= 23) ? 0 : (ds.count_24 + 1);
    if (prep_sample)
         SET_DSP_SQRT_APPROX(ds.mag_value_24, ds.state_q_24, ds.state_i_24);

    /* store in circular buffer so that is can be subtracted from the end
       to make a moving average filter */
    ds.sample_buffer[ds.sample_no++] = sample;
    if (ds.sample_no >= df.buffer_size)
       ds.sample_no = 0;
}

/* this is called by the interrupt handle with a new sample from the DAC.
   first it updates the spectral channels.  then it figures out the magnitude of
   the signal given the current modulation type (OOK/FSK).  it tries to detect
   an edge to determine when the current bit has finished and when it should
   expect the next bit.  it resets the bit counter when the sync signal has
   been received, and when 30 bits of a frame have been received, stores the
   frame in the frame FIFO */
void dsp_interrupt_sample(uint16_t sample)
{
    int16_t demod_sample, temp;
    uint8_t received_bit;
    uint8_t hamming_weight;

    /* update the filter channel I & Q */
    dsp_new_sample(sample);

    /* only proceed if we have new magnitude samples */
    if (!ds.mag_new_sample)
        return;

    /* demodulate a sample either based on the amplitude in one frequency
       channel for OOK, or the difference in amplitude between two frequency
       channels for FSK */

    switch (df.mod_type)
    {
        case DSPINT_OOK:  demod_sample = ds.mag_value_8 - ds.power_thr;
                          ds.ct_sum += ds.mag_value_8;
                          break;
        case DSPINT_OOK2: demod_sample = ds.mag_value_16 - ds.power_thr;
                          ds.ct_sum += ds.mag_value_16;
                          break;
        case DSPINT_FSK:  demod_sample = ds.mag_value_20 - ds.mag_value_12;
                          ds.ct_sum += (ds.mag_value_20 + ds.mag_value_12);
                          break;
        case DSPINT_FSK2: demod_sample = ds.mag_value_12 - ds.mag_value_8;
                          ds.ct_sum += (ds.mag_value_12 + ds.mag_value_8);
                          break;
    }

    /* This is the automatic "gain" control (threshold level control).
       find the average of a certain number of samples (power of two for calculation
       speed) so that we can determine what to set the thresholds at for the bit on/off
       threshold for ook and the edge threshold for ook/fsk */
    if ((++ds.ct_average) >= (1 << DSPINT_AVG_CT_PWR2))
    {
       uint16_t temp;
       ds.ct_average = 0;
       temp = (ds.ct_sum >> DSPINT_AVG_CT_PWR2);
       /* don't allow threshold to get too low, or we'll be having bit edges constantly */
       ds.power_thr = temp > df.power_thr_min ? temp : df.power_thr_min;
       ds.edge_thr = ds.power_thr;
       printf("power %d edge %d min %d --------------------------------\
n",ds.power_thr,ds.edge_thr,df.power_thr_min);
       ds.ct_sum = 0;
    }

    /* calculate the difference between the modulated signal between now and one bit period ago to see
       if there is a bit edge */
    temp = ds.demod_buffer[ds.demod_sample_no];
    ds.bit_edge_val = demod_sample > temp ? (demod_sample - temp) : (temp - demod_sample);
    /* store the demodulated sample into a circular buffer to calculate the edge */
    ds.demod_buffer[ds.demod_sample_no] = demod_sample;
    if ((++ds.demod_sample_no) >= df.demod_samples_per_bit)
        ds.demod_sample_no = 0;

    if (ds.edge_ctr > 0)  /* count down the edge counter */
        --ds.edge_ctr;

    received_bit = 0;

    if (ds.edge_ctr < ds.demod_edge_window)  /* if it below the edge window, start looking for the bit
edge */
    {
        if (ds.bit_edge_val > ds.edge_thr)  /* the edge should come around now, does it exceed
threshold */
        {
            if (ds.bit_edge_val > ds.max_bit_edge_val)  /* if so, have we reached the peak of the edge
*/
            {
                ds.max_bit_edge_val = ds.bit_edge_val;  /* if so, reset the edge center counter */
                ds.next_edge_ctr = 1;
                ds.cur_bit = demod_sample;              /* save the bit occurring at the edge */
            } else
                ds.next_edge_ctr++;                     /* otherwise count that we have passed the edge
peak */
        } else
        {
            if (ds.max_bit_edge_val != 0)   /* if we have detected an edge */
            {
                ds.edge_ctr = df.demod_samples_per_bit > ds.next_edge_ctr ? df.demod_samples_per_bit -
ds.next_edge_ctr : 0;
                                /* reset edge ctr to look for next edge */
                ds.max_bit_edge_val = 0;    /* reset max_bit_edge_val for next edge peak */
                received_bit = 1;
                ds.demod_edge_window = df.demod_edge_window;
            } else /* we haven't detected an edge */
            {
                if (ds.edge_ctr == 0)
                {
                    ds.cur_bit = demod_sample;              /* save the bit */
                    ds.edge_ctr = df.demod_samples_per_bit;  /* reset and wait for the next bit edge to
come along */
                    received_bit = 1;                       /* an edge hasn't been detected but a bit
interval happened */
                }
            }
        }
    }
    if (!received_bit)   /* no bit available yet, wait */
        return;

    /* add the bit to the current word */
    ds.current_word = (ds.current_word << 1) | (ds.polarity ^ (ds.cur_bit > 0));
    ds.bitflips_ctr++;
    /* this is done on the bit before it is needed to reduce worst case latency */
    if (ds.bitflips_ctr == 4)
    {
        /* keep track of the number of complement bits in the word. should be 6 */
        uint8_t maskbits, lowerbyte = ((uint8_t)ds.current_word);
        maskbits = lowerbyte & 0x0C;
        ds.bitflips_in_phase += (maskbits == 0x08) || (maskbits == 0x04);
        maskbits = lowerbyte & 0x18;
        ds.bitflips_lag += (maskbits == 0x10) || (maskbits == 0x08);
        maskbits = lowerbyte & 0x06;
        ds.bitflips_lead += (maskbits == 0x04) || (maskbits == 0x02);
    } else if (ds.bitflips_ctr >= 5)
        ds.bitflips_ctr = 0;
    hamming_weight = dsp_hamming_weight_30(ds.current_word ^ DSPINT_SYNC_CODEWORD);
    printf("received: %08X %05d %02d %02d %02d\n", ds.current_word, ds.cur_bit, ds.current_bit_no,
hamming_weight, ds.polarity);
    if (hamming_weight < (ds.resync ? 4 : 6))  /* 30-bit resync word has occurred! */
    {
        printf("resync!\n");
        dsp_reset_codeword();
        ds.resync = 1;
        return;
    }
    /* if we 15 of the last 16 bits zeros with fsk, that means we have a reversed polarity start */
    hamming_weight = dsp_hamming_weight_16(ds.current_word);
    if ((hamming_weight < 2) && (df.fsk))
    {
        ds.demod_edge_window = df.demod_samples_per_bit;
        ds.polarity = !ds.polarity; /* reverse the polarity of FSK frequencies and 0/1 */
        dsp_reset_codeword();
        ds.resync = 0;
        return;
    }
    /* if we have 15 of the last 16 bits ones, that is a ook key down start, or a fsk start */
    if (hamming_weight >= 15)
    {
        ds.demod_edge_window = df.demod_samples_per_bit;
        dsp_reset_codeword();
        ds.resync = 0;
        return;
    }
    /* if we have synced, and we have 30 bits, we have a frame */
    ds.current_bit_no++;
    if ((ds.current_bit_no >= 30) && (ds.resync))  /* we have a complete frame */
    {
       if ((ds.bitflips_lead > ds.bitflips_in_phase) && (ds.bitflips_lead >= 6))
        /* we are at least one bit flip ahead, we probably registered a spurious bit flip */
       {
         /* back up and get one more bit.  clear bit_lead so we don't try a second time */
         ds.current_bit_no--;
         ds.bitflips_lead = 0;
       } else
       {
          if ((ds.bitflips_lag > ds.bitflips_in_phase) && (ds.bitflips_lag >= 4))
          /* we are at least one bit flip short, we probably fell a bit behind */
          {
            dsp_insert_into_frame_fifo(&dsp_output_fifo, ds.current_word >> 1);
            printf("inserted- word into fifo: %08X %02d %02d %02d ----------------------------\n",
                   ds.current_word >> 1, ds.bitflips_in_phase, ds.bitflips_lag, ds.bitflips_lead);
            /* start with the next word with one flip */
            ds.current_bit_no = 1;
            ds.bitflips_ctr = 1;
         } else
         {
             if (ds.bitflips_in_phase >= 4)
             {
               /* otherwise we just place in buffer and the code word is probably aligned */
               dsp_insert_into_frame_fifo(&dsp_output_fifo, ds.current_word);
               printf("inserted0 word into fifo: %08X %02d %02d %02d ----------------------------\n",
                   ds.current_word, ds.bitflips_in_phase, ds.bitflips_lag, ds.bitflips_lead);
             }
             ds.current_bit_no = 0;
             ds.bitflips_ctr = 0;
          }
          ds.bitflips_in_phase = 0;
          ds.bitflips_lag = 0;
          ds.bitflips_lead = 0;
       }
    }
}
```

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.