Simple Conversational Amateur Messaging Protocol (SCAMP)

by Daniel Marks, KW4TI

Draft v0.1 2021-10-12

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 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 blocks, in order of most significant bit to least significant bit (left to right as shown here). The format of each block 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 block 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.

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 block at the wrong point in the bit stream. Therefore a synchronization protocol is necessary that can synchronize the beginning of a 30-bit block. Because of the complement bits inserted into the 30-bit block, there will be no more than five consecutive mark or space (1 or 0) bits in the stream. This is used for synchronization. Synchronization is sent either at the beginning of the transmission, or can be reinitiated after sending a 30-bit block, with the exception of when text code words are sent multiple times without a delay in between for redundancy.

For OOK synchronization, a synchronization signal is detected by the receiver when 14 or 15 of the last 15 bit intervals decoded is a mark condition (one or key down). If this happens in the middle of receiving a codeword, the codeword being received is aborted and the synchronization phase is initiated. The receiver then waits for following sequence to occur: 3 space bits, 3 mark bits, 3 space bits, 3 mark bits, 3 space bits, 3 mark bits. The edges of the transitions between key on and off are used to synchronize the receiver’s clock with the sender. The sending of the first Golay code word occurs immediately after the last 3 mark bits.

For FSK synchronization, a synchronization signal is detected when either of the two FSK frequencies are detected by the receiver for 14 or 15 of the last 15 bit intervals decoded. The FSK frequency corresponding to the 14 or 15 bits is considered the mark condition. If this happens in the middle of receiving a codeword, the codeword being received is aborted and the synchronization phase is initiated. The receiver then waits for the following sequence to occur: 3 space bits, 3 mark bits, 3 space bits, 3 mark bits, 3 space bits, 3 mark bits. The receiver may identify the space frequency by looking at both frequencies above and below the mark frequency, or the direction of the mark to space frequency

shift may be manually specified. The edge of the transitions between frequencies are used to synchronize the receiver's clock with the sender. The sending of the first Golay code word occurs immediately after the last 3 mark bits.

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 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.

Golay Matrix

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(uint16_t n)
{
  uint8_t s = 0;
  while (n != 0)
  {
     s += (n & 0x1);
     n >>= 1;
  }
  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)
{
```

```
  uint16_t enc = codeword & 0xFFF;
  uint16_t parity = codeword >> 12;
  uint8_t i;
  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 undetected */
  syndrome = golay_mult(enc) ^ parity;
  if (golay_hamming_weight(syndrome) <= 3)
   return enc;

  /* check to see if the parity bits have no errors */
  parity_syndrome = golay_mult(parity) ^ enc;
  if (golay_hamming_weight(parity_syndrome) <= 3)
     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--;
      if (golay_hamming_weight(syndrome ^ golay_matrix[i]) <= 2)
        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];
      if (golay_hamming_weight(par_bit_synd) <= 2)
        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.