LB28, LB2Q & LB3Q

A new digital radio modulation technique for HF QRP and QRPP.

Inventor and Author: Lawrence Byng

Original Publication Date: April 22nd 2025
Revised: April 23rd 2025
Document Version: 1.2

Preliminary test results over AWGN channel, indicate the new modes can consistently decode ***at*** the Shannon limit (-1.59 dB Eb/N0) with a 0.0 Bit Error Rate.

Inspiration

The inspiration for these modes originally came from using other ham radio digital modes and observing some of the limitations. This inspiration was originally in two main areas:-

1) A character encoding scheme to allow maximum flexibility and functionality while at the same time efficiently representing the data in a compact form and
2) A mode that does not require any special synchronization to time clocks or to concensus timing offsets to achieve optimal decodes.

Additionally, while researching digital modes and combining the different modulation techniques, I came across many instances where combining FSK with PSK was discouraged as being either ineffective, overly complex or impossible to achieve. In reality, combining FSK with PSK, when done using an effective design and technique, offers immense potential, not only for ham radio but for telecommunications in general.

Design Points

Points of consistency.

A key design consideration is how to modulate and encode the data so that each section is self contained. A well defined structure is paramount for building a process to go from known points of consistancy to less well defined points and even garbled data points. This key aspect of the design facilitates an accurate demodulation process.

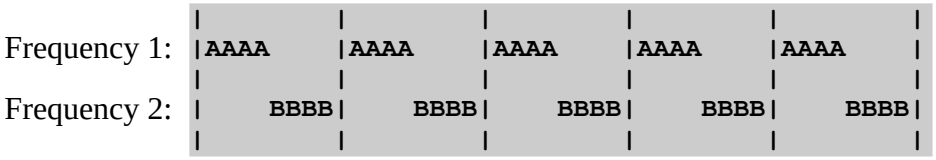
A modulation scheme of 2FSK + 8PSK was initially chosen. An 8PSK signal is modulated by 2 FSK carriers. The 2 FSK carriers function together to provide a clock signal anchor point. This anchor point is used in conjunction with a block format to represents 1 character i.e. 6 bits for a base 64 character encoding scheme.

The easiest way to explain how the modulation holds together is to explain it on the basis of a sequence of blocks. Each block represents 1 character. A block spans both the frequency domain and the time domain and has a specific format depending on which modulation scheme is being used. The simplest ‘AB’ block format has 2 FSK in the frequncy domain and 2 x 3 bit codes of 8PSK in the time domain. Each of these frequencies will contain an 8PSK signal for half of the block...first the low frequency 8PSK signal then switching to the higher of the two frequencies for an additional 8PSK signal. The frequencies can be very tightly packed...the 20 character per second mode uses a 100 Hz spacing and the weak signal interpolated modes have 10 Hz spacing between the frequencies.

The block gives complete referential integrity and provides a point of consistency from which to start a decode process.

Block Formats

The diagram below gives a visual representation of 5 Character blocks from left to right. The encoding scheme is LB28; 2 FSK and 8 PSK. ‘A’ represents a 3 bit pulse. Each pulse is repeated 4 times. ‘B’ represents a different 3 bit pulse and is repeated 4 times. Frequency on the vertical axis and Time on the horizontal axis. The complete set encodes for 5 x base 64 characters. This is the basic AB format.

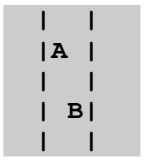


Time

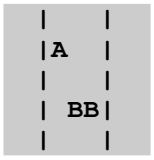
Detailed below are diagrammatic representations of some of the main block formats for base 64 character set and base 256 character set using both 2FSK and 3FSK combined with 8PSK, QPSK and BPSK with 1 pulse per 3 bit encode for 8psk, 1 pulse per 2 bit encode for QPSK or 1 pulse per 1 bit encode for BPSK.

- Base 64 Character set

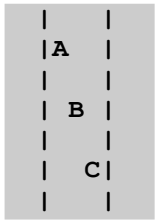
LB28 - AB



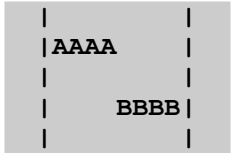
LB2Q – ABB



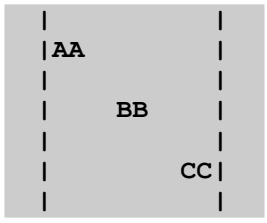
LB3Q - ABC



LB2B – AAAABBBB

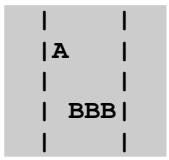


LB3B - AABBCC

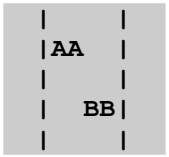


- Base 256 Character Set

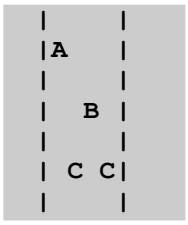
LB2Q - ABBB



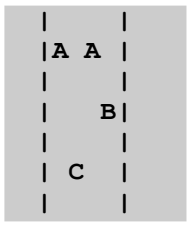
LB2Q – AABB



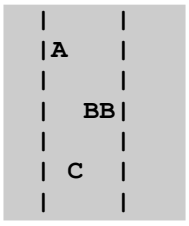
LB3Q – ACBC



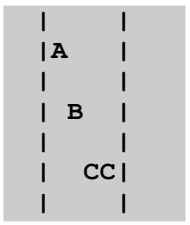
LB3Q – ACAB



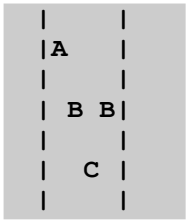
LB3Q – ACBB



LB3Q – ABCC



LB3Q - ABCB



There are additional block formats not mentioned here, however the above details should give an idea of how the different block formats are utilized for the various modulations.

Modulation

The modulation process proceeds in order as follows:-

- Two symbols waves, one for each frequency, are derived as follows:-

```
symbol_wave1 = amplitude * np.cos(2 * np.pi * time * frequency[0] + phase1) + amplitude * np.sin(2 *
np.pi * time * frequency[0] + phase1)

symbol_wave2 = amplitude * np.cos(2 * np.pi * time * frequency[1] + phase2) + amplitude * np.sin(2 *
np.pi * time * frequency[1] + phase2)
```

- Each symbol_wave is then further modulated with the root raised cosine (RRC) pulse shaping. For a signal with two pulses per symbol. The RRC shaping is as follows:-

```
shaped_symbol_wave = (  (symbol_wave[0] * filtRRC_fourth_wave[0]) + (symbol_wave[0] *
filtRRC_fourth_wave[1]) +
                        (symbol_wave[1] * filtRRC_fourth_wave[2]) +  (symbol_wave[1] *
filtRRC_fourth_wave[3])  )  / 4

     where  filtRRC_fourth_wave[n] is the root raised cosine shaper for the nth pulse of the block.
```

- Each of the above steps are performed once for each character. Each shaped symbol wave is joined onto the end of the previous to form a modulated stream of data. Each character comprises 2 x 8psk symbols to achieve a 6 bit character encoding of one of 64 characters.

Demodulation

Decode steps vary slightly between the non-interpolated modes and the interpolated modes, with the non-interpolated process being a subset of the interpolated mode process. Essentially, the non-interpolated modes skip the interpolation process.

The precise steps for the interpolated modes are as follows:-

- The received wave is sectioned into chunks of signal each approximately 32 characters long although this can vary.

- Locate the RRC peaks of the incoming signal and determine the location of the first full pulse in the received chunk. This process involves

```
For each sample i in the received chunk,

test_peak = signal[i * symbol_block_size : (I * symbol_block_size) + symbol_block_size]
test_max = np.max(test_peak)
test_min = np.min(test_peak)
max_indices = np.where((test_peak*(100/test_max)) > parameters[5])
min_indices = np.where((test_peak*(100/test_min)) > parameters[5])

Each of the minimum and maximum indices x are appended to a list of all indices:-
all_list.append(max_indices[0][x] % pulse_width)
all_list.append(min_indices[0][x] % pulse_width)

And the median value is then used to determine the most likely first peak location :-
pulse_start = (int(np.median(np.array(all_list)))  % pulse_width)  + pulse_length / 2
```

- A Fast Fourier Transform (FFT) bandpass filter is applied to each of the two signals. The width of this filter is absolutely critical for accurate decoding. For extreme at the limit decodes, this filter needs to be no more than 2 Hz wide.

- A similar process to the RRC peak location process described above is then used to locate all RRC pulse shaped pulses for each of the two frequencies. The result is two lists of indices representing each of the pulses for each of the streams. Ideally the list of indices would be a nice clean list that matches exactly the set of pulses that were modulated and transmitted. In reality the list will be anything but...especially if the signal has been distorted by noise; the lists will have indices missing at the ends making them shorter, indices will be missing from the middle and some indices will appear in both lists. An algorithm is then used to sort out and reconstruct the most probable two lists of indices. The more accurate and complete the resulting lists, then the more accurate can be the decoding that follows at a later stage.

- Apply interpolation algorithm to process received lists. Algorithm is described in the next section with python code excerpts in appendix 1.

- The original received signal is passed through a matching RRC filter and then filtered using a very sharp cutoff FFT bandpass filter. The filter width is absolutely critical and for decoding at the limit needs to be no more than 2Hz wide.

- The processed signal is then sent to a Costas Loop process to convert to baseband and determine the phases along the full length of each of the 2 frequency streams for the entire chunk of received data.

- A process of mean averaging is used to average all baseband pulses in a given stream that correspond with indices in the lists recovered in the prior steps. The averaging process effectively cancels out any remaining noise to a level of 1/N where N is the number of list indices in a given stream. For the LB28 mode which uses 2FSK + 8PSK and 256 pulses per block, this is equivalent to a noise reduction to a level of 1/128th if the list indices are recovered in full. At the same time, the baseband phase values remain unchanged. This equates to an amplification of the signal relative to the noise by a factor of 128. The net result is that the noise is reduced to minimal background and the signal is effectively amplified to allow for a successful decode.

- A phase value is extracted from the mean averaged data using the median index of the respective list.

- The result is two phase values, one from each frequency (2*8psk). This is then used to decode for the transmitted character of the base 64 character set.

The above process is the basis of all of the LB28, LB2Q and LB3Q Interpolated modes. The specifics vary slightly for example:-

- The LB2Q modes which use 2 FSK + QPSK, have 2 streams of 4 PSK pulses per block arranged in a abb configuration on the block. Note: all configurations aab, aba, bab, bba are tantamount to the same aab block configuration for 64 character modes. Also abbb/aabb can be used for 256 character modes.

- The LB3Q modes utilizes 3 FSK carriers. The block configurations include abc for 64 character modes and acbc/acab/acbb/abcc for 256 character modes. Other than that the process will be practically identical.

  The technique is also extended to BPSK:-

- LB2B consists of 2 FSK carriers + BPSK. The block encoding for this is aaaabbbb for base 64 character set.

- LB3B consists of 3 FSK carriers + BPSK. The block encoding for this is aabbcc for base 64 character set. A block envoding of aaccbbcc for base 256 character set

- The technique can in theory also be extended to other combinations such as FSK + QAM

Interpolation Algorithm

- The first step is to remove any list indices that appear in both lists. These are clearly erroneous and no determination can be made at this stage about which list the index should belong to.

- Next, interpolate the lists so that a sequence with gaps is filled in resulting in a contiguous set of indices. This involves several steps:-

    - First, find the median list index and remove any indices that deviate by more than num_pulses_per_block / 4...this represents half a stream for a 2 frequency decode.

    - If any indices were removed at the conclusion of this process, a new median index is found and the process repeats until there are no more statistical outliers in the list.

    - As this process proceeds, the median tends towards truth. The resulting median value is a more accurate median. This truth median is then used to purge any statstical outliers from the original list, a copy of which was saved at the start of this process. Take a final truth median measurement and compare with previous truth median. Repeat the above steps to a max of 5 interations or until the difference between the final truth median and the prior truth median is 0.

- Missing values are then interpolated using a walk algorithm to walk thru the list indices and determine by distance what the sequence is and ultimately determine how the indices wrap around i.e. do the indices proceeed from low index to high index or high index to low index with wrap around. The end result is a clean contiguouse set of indices. These represent the points along the signal wave.

- Additional interpolation steps are then perfomed to further maximize the completeness of the list of indices as follows:-

    - Firstly, if either of the two processed lists is a complete list of indices i.e. it has a full set of num_pulses_per_block / 2 and also if the other list is less than complete, then the short list is filled out by adding the corresponging partner index into the partner list.

    - Next, the same process is done if one list is longer than the other but the longer list is not a full set of indices.

    - An additional step could also be done to determine if the last index of one list is adjacent to the first index of the partner list. If so, then all indices can be computed. This final step was not included for the LB28 sample tests in Table 1, but may provide an additional optimization.

These steps are sufficient to provide a most probable set of indices for both signals. Each index directly relates to exactly where the data is hiding in the received signal block. This is paramount for the decode process that follows.

Parameter Block

Key parameters for each of the modulation modes appear in the parameter block section. There is one parameter block for each modulation type.

'LB28-0.15625-10I':-

```
'info'                  : '0.15625 characters per second, 0.9375 baud (bits per second)',
'symbol_block_size'     : 51200,
'symbol_wave_function'  : twohundredfiftysixths_symbol_wave_function,
'sample_rate'           : 8000,
'num_carriers'          : 2,
'carrier_separation'    : 10,
'detector_function'     : 'mode',
'baseband_conversion'   : 'costas_loop',
'phase_extraction'      : EXTRACT_INTERPOLATE,
'fft_filter'            : (-1, 1, -1, 1),
'fft_interpolate'       : (-1, 1, -1, 1),
'pulses_per_block'      : 256,
'parameters'            : (600, 0.70, 0.9, 10000, 2, 98) },
```

The last ‘parameters’ items are as follows respectively:-

- 600 Number for phase value constellation extraction to delineate +1, 0 and -1,
- 0.70 Alpha value for the RRC wave shape
- 0.9 T value for the RRC wave shape
- 10000 Baseband normalization value to normalize the resulting extraction phase
- 2 Extract phase num waves. Number of waves used for phase level determination
- 98 Percentage value of max peak to determine if a peak qualifies ‘at max’.

Reference Platform

A code reference platform has been included in Python to further illustrate each of the steps and the precise detail required. For optimal performance a JIT compiler could be used in conjunction with the costas loop method as this consumes the vast majority of the processing power for demodulation. Code excerpts from the reference platform are included in Appendix 1.

Table 1: Preliminary Test Results

Preliminary Test Results for LB28-0.625-10-I, LB28-0.3125-10-I and LB28-0.15625-10-I using length 32 and 33 block strings i.e. 65 characters total over 2 decodes with 6 bit characters.

Mode Name	**CPS/bits**	**Eb/N0 (dB)**	**BER**	**Equiv. SNR (dB)**	**Width**	**Total Bits**
AWGN Factor 7.85+						
LB28-0.625-10-I	0.625/6	-1.54	0.017412	-27.54	10 Hz	402
LB28-0.625-10-I	0.625/6	-1.53	0.019900	-27.54	10 Hz	402
LB28-0.625-10-I	0.625/6	-1.54	0.0	-27.54	10Hz	402
LB28-0.625-10-I	0.625/6	-1.52	0.007462	-27.53	10 Hz	402
LB28-0.625-10-I	0.625/6	-1.56	0.017412	-27.54	10 Hz	402
LB28-0.625-10-I	0.625/6	-1.60	0.004975	-27.55	10 Hz	402
LB28-0.625-10-I	0.625/6	-1.57	0.012437	-27.54	10 Hz	402
LB28-0.625-10-I	0.625/6	-1.58	0.019900	-27.54	10 Hz	402
LB28-0.625-10-I	0.625/6	-1.59	0.007462	-27.55	10 Hz	402
AWGN Factor 7.68+						
LB28-0.3125-10-I	0.3125/6	-1.70	0.009950	-30.57	10 Hz	402
LB28-0.3125-10-I	0.3125/6	-1.64	0.0	-30.56	10 Hz	402
LB28-0.3125-10-I	0.3125/6	-1.57	0.0	-30.55	10 Hz	402
LB28-0.3125-10-I	0.3125/6	-1.58	0.0	-30.55	10 Hz	402
LB28-0.3125-10-I	0.3125/6	-1.57	0.007462	-30.55	10 Hz	402
LB28-0.3125-10-I	0.3125/6	-1.59	0.0	-30.56	10 Hz	402
AWGN Factor 7.16+						
LB28-0.15625-10-I	0.15625/6	-1.55	0.0	-33.56	10 Hz	402
LB28-0.15625-10-I	0.15625/6	-1.59	0.0	-33.57	10 Hz	402
LB28-0.15625-10-I	0.15625/6	-1.59	0.0	-33.57	10 Hz	402
LB28-0.15625-10-I	0.15625/6	-1.60	0.007462	-33.57	10 Hz	402
LB28-0.15625-10-I	0.15625/6	-1.60	0.0	-33.57	10 Hz	402
LB28-0.15625-10-I	0.15625/6	-1.61	0.024875	-33.57	10 Hz	402
LB28-0.15625-10-I	0.15625/6	-1.62	0.007462	-33.57	10 Hz	402
LB28-0.15625-10-I	0.15625/6	-1.63	0.0	-33.57	10 Hz	402
LB28-0.15625-10-I	0.15625/6	-1.66	0.0	-33.58	10 Hz	402
LB28-0.15625-10-I	0.15625/6	-1.67	0.288557	-33.58	10 Hz	402
LB28-0.15625-10-I	0.15625/6	-1.72	0.002487	-33.59	10 Hz	402
LB28-0.15625-10-I	0.15625/6	-1.76	0.074626	-33.59	10 Hz	402

Appendix 1: Interpolation Algorithm

The following code excerpts are taken from the python reference platform used for the tests.

```
      """ Code snippet from higher level demodulation process """



      max_occurrences_lists = self.removeConflictingItemsTwoList([max_occurrences_lower,
max_occurrences_higher])
      max_occurrences_lower  = max_occurrences_lists[0]
      max_occurrences_higher = max_occurrences_lists[1]

      interpolated_lower  = self.interpolate_contiguous_items(max_occurrences_lower)
      interpolated_higher = self.interpolate_contiguous_items(max_occurrences_higher)

      half = int(self.osmod.pulses_per_block/2)

      """ if either of the interpolated lists is complete, fill out the other interpolated list if
incomplete """

      if len(interpolated_lower) == half and len(interpolated_higher) < half:
        for i in range(0,self.osmod.pulses_per_block):
          if i not in interpolated_lower and i not in interpolated_higher:
            interpolated_higher.append(i)
      elif len(interpolated_higher) == half and len(interpolated_lower) < half:
        for i in range(0,self.osmod.pulses_per_block):
          if i not in interpolated_lower and i not in interpolated_higher:
            interpolated_lower.append(i)

      """ if either of the inetrpolated lists is greater than the other list, fill out the other
interpolated list"""

      interpolated_lower  = self.sort_interpolated(interpolated_lower)
      interpolated_higher = self.sort_interpolated(interpolated_higher)

      if len(interpolated_lower) > len(interpolated_higher):
        for i in range(interpolated_lower[0],interpolated_lower[-1]):
          partner_offset = self.osmod.pulses_per_block / self.osmod.num_carriers
          partner_index = int((i + partner_offset) % self.osmod.pulses_per_block)
          if partner_index not in interpolated_higher:
            interpolated_higher.append(partner_index)
      elif len(interpolated_higher) > len(interpolated_lower):
        for i in range(interpolated_higher[0],interpolated_higher[-1]):
          partner_offset = self.osmod.pulses_per_block / self.osmod.num_carriers
          partner_index = int((i + partner_offset) % self.osmod.pulses_per_block)
          if partner_index not in interpolated_lower:
            interpolated_lower.append(partner_index)

      """ if the two halves are adjacent then can interpolate all!"""
      #if int((interpolated_higher[-1] + 1) % self.osmod.pulses_per_block) ==
interpolated_higher[0]:

    """ return the most probable and complete interpolated and sorted list set """
    return self.sort_interpolated(interpolated_lower), self.sort_interpolated(interpolated_higher)
```

```
  """ interpolate contiguous by distance and determine wrap around """
  def interpolate_contiguous_items(self, list_items):
      saved_list_items = list_items
      have_suspect_items = True

      while have_suspect_items:
        have_suspect_items = False
        median_input_list = int(np.median(np.array(list_items)))
        for i in list_items:
          if abs(median_input_list-i)>int((self.osmod.pulses_per_block/self.osmod.num_carriers)/2):
            list_items.remove(i)
            have_suspect_items = True
      median_input_list_truth = median_input_list

      """ now repeat process using final truth value as median """
      list_items = saved_list_items
      for i in list_items:
        if abs(median_input_list_truth-i)>int((self.osmod.pulses_per_block/self.osmod.num_carriers)/
2):
          list_items.remove(i)

      """ walk the list to determine extents  """
      half = int(self.osmod.pulses_per_block / 2)
      start_value = list_items[0]
      min_value = 0
      max_value = 0
      walk = 0
      for i, j in zip(list_items, list_items[1:]):
        if abs(i - j) < half:
          walk = walk - (i - j)
        elif abs(j - i) < half:
          walk = walk + (j - i)
        elif abs(j + self.osmod.pulses_per_block - i) < half:
          walk = walk + (j + self.osmod.pulses_per_block - i)
        elif abs(i + self.osmod.pulses_per_block - j) < half:
          walk = walk - (i + self.osmod.pulses_per_block - j)

        if walk > max_value:
          max_value = walk
        if walk < min_value:
          min_value = walk

      interpolated_list = []
      for x in range(start_value + min_value, start_value + max_value + 1):
        interpolated_list.append(x % self.osmod.pulses_per_block)

      return interpolated_list
```

```
 def removeConflictingItemsTwoList(self, max_occurrences_lists):
    max_occurrences_lower = max_occurrences_lists[0]
    max_occurrences_higher = max_occurrences_lists[1]

    in_both_lists = []
    for i in max_occurrences_lower:
      if i in max_occurrences_higher and i not in in_both_lists:
        in_both_lists.append(i)
    for i in max_occurrences_higher:
      if i in max_occurrences_lower and i not in in_both_lists:
        in_both_lists.append(i)
    self.debug.info_message("in_both_lists: " + str(in_both_lists))

    for i in in_both_lists:
      max_occurrences_lower.remove(i)
      max_occurrences_higher.remove(i)




def sort_interpolated(self, interpolated_lower):
    half = int(self.osmod.pulses_per_block/2)
    normalized_list = []
    restored_sorted_list = []
    for item in interpolated_lower:
      if item < half:
        normalized_list.append(item + self.osmod.pulses_per_block)
      else:
        normalized_list.append(item)

    normalized_list.sort()

    """ does the data wrap around? """
    if self.osmod.pulses_per_block - 1 in normalized_list:
      for item in normalized_list:
        if item < self.osmod.pulses_per_block:
          restored_sorted_list.append(item)
      for item in normalized_list:
        if item >= self.osmod.pulses_per_block:
          restored_sorted_list.append(item % self.osmod.pulses_per_block)
      return restored_sorted_list
    else:
      for item in normalized_list:
        if item >= self.osmod.pulses_per_block:
          restored_sorted_list.append(item % self.osmod.pulses_per_block)
      for item in normalized_list:
        if item < self.osmod.pulses_per_block:
          restored_sorted_list.append(item)
      return restored_sorted_list
```