LB8, LB28, LB2Q & LB3Q

A new modulation and phase extraction technique for digital communication via electromagnetic waves.

Inventor and Author: Lawrence Byng
Original Publication Date: April 22nd 2025

Revised: May 27th 2025: Added LB8 and Phase Extraction sections
Revised: July 31st 2025: Added Doppler Shift and Alignment / Calibrate Sections
Revised: August 11th 2025: Added Pulse Train Disposition and Extrapolation Sections

Document Version: 1.20 - final

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.

Further, additional preliminary test results indicate the modes can also consistently decode data from noise with high accuracy at Eb/N0 values that far exceed the Shannon limit.

All of these preliminary test results will need to be independently verified / confirmed before any definitive conclusions can be drawn.

Table of Contents

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 consensus 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 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 consistency 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 frequency domain and 2 x 3 bit codes of 8PSK in the time domain. Each of these frequencies 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.

Scalability

Another key aspect of the design is scalability. The reference platform LB28 modes span from 20 characters per second on the higher baud rate with lower noise resilience end of the spectrum to a 6.4 seconds per character mode (0.15625 CPS) for lower baud rate with higher noise resilience at the other. This spectrum can be extended with additional carriers on the higher baud rate end and with more pulse repeats on the higher noise resilience end. Going in order from high baud rate to low baud rate, each successive mode is half as fast as the prior mode as it uses double the pulses per character but is twice as resilient in terms of signal to noise due to the doubling of the pulses per character. When adding additional carriers, blocks can be overlapped to increase carrier density thus resulting in more carriers per unit bandwidth of spectrum. This is described in greater detail in a later section.

It would also be possible to change modulation mid-stream i.e. in real time if the bit error rate changes significantly, as any changes to the block format would become immediately apparent. This could be utilized for example over a fully synchronous full duplex communication link or over a half duplex communication link as is often used in ham radio for sending critical information with an ARQ 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.

```
              |           |           |           |           |           |
Frequency 1:  |AAAA       |AAAA       |AAAA       |AAAA       |AAAA       |
              |           |           |           |           |           |
Frequency 2:  |       BBBB|       BBBB|       BBBB|       BBBB|       BBBB|
              |           |           |           |           |           |
```

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

```
|   |
|A  |
|   |
|  B|
|   |
```

LB2Q – ABB

```
|    |
|A   |
|    |
|  BB|
|    |
```

LB3Q - ABC

```
|    |
|A   |
|    |
| B  |
|    |
|   C|
|    |
```

LB2B – AAABBB

```
|       |
|AAA    |
|       |
|    BBB|
|       |
```

LB3B - AABBCC

```
|          |
|AA        |
|          |
|    BB    |
|          |
|        CC|
|          |
```

- Base 256 Character Set:-

LB2Q - ABBB

```
|    |
|A   |
|    |
| BBB|
|    |
```

LB2Q – AABB

```
|    |
|AA  |
|    |
|  BB|
|    |
```

LB3Q – ACBC

```
|    |
|A   |
|    |
|  B |
|    |
| C C|
|    |
```

LB3Q – ACAB

```
|    |
|A A |
|    |
|   B|
|    |
| C  |
|    |
```

LB3Q – ACBB

```
|    |
|A   |
|    |
|  BB|
|    |
| C  |
|    |
```

LB3Q – ABCC

```
|    |
|A   |
|    |
| B  |
|    |
|  CC|
|    |
```

LB3Q - ABCB

```
|    |
|A   |
|    |
| B B|
|    |
|  C |
|    |
```

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 mode 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 the following:-

```
""" 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 over the entire received chunk of data. The result is two lists of indices representing each of the pulses for each of the two frequency streams. Ideally, the lists of indices will be nice clean lists that match exactly the set of pulses that were modulated and transmitted. In reality, the lists will be anything but, especially if the signal has been distorted by noise; the lists will have indices missing at the start and end 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 an interpolation algorithm to process the received lists. The interpolation algorithm is described in the next section. python code excerpts from the test reference platform are included in appendix 1.

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

- The processed signal is then sent to a Costas Loop to convert to baseband and determine the phases along the full length of each of the two 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 utilize 3 FSK carriers + QPSK. 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 modulation technique is also extended to BPSK:-

- LB2B consists of 2 FSK carriers + BPSK. The block encoding for this is aaabbb for base 64 character set and aaaabbbb for base 256 character set.

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

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

Additionally, interpolated modes using standing waves with adjacent pulse phase extraction and doppler shift correction have additional code to align the pulse train and correct for doppler shift timing offsets and phase distortion from pulse compression / expansion.

Phase Extraction

This section describes the new technique for phase extraction using standing waves. Under certain conditions, standing waves can be made to form along the intra-block pulse train of the interpolated modes. Please note this technique is experimental and is currently being assessed for viability. Having said that, the technique has demonstrated sufficient consistency for initial publication pending further prototype development and testing.

Background

For interpolated modes without standing waves, each pulse along the pulse train for a given character block is modulated using the same phase; one phase on each pulse of the lower frequency and a different phase on each pulse of the higher frequency. This codes for a single character. Each block is sequential in time and provides a stream of characters forming the data transmission.

In a perfectly calibrated system, the sampling will be perfectly aligned with the signal as created and transmitted. This allows recreation of the original information without the need for correction of any sort. This is the simple case.

In reality there are many distortions that come into play which significantly impact the signal. Even distortions in the computer sampling, with respect to small timing delays, can have a significant impact on the phase of the received signal; phase rotation at the start of the signal due to mismatched timing and phase rotation of the signal itself from delays at points along the signal. These distortions modify the phase of the received signal and present some interesting challenges to achieve perfect decodes. Many techniques exist to correct for these distortions. The LB28 interpolated modes utilize a new technique for absolute phase recovery based on relative phase pulse values and pulse train standing waves.

Pulse Train Standing Waves

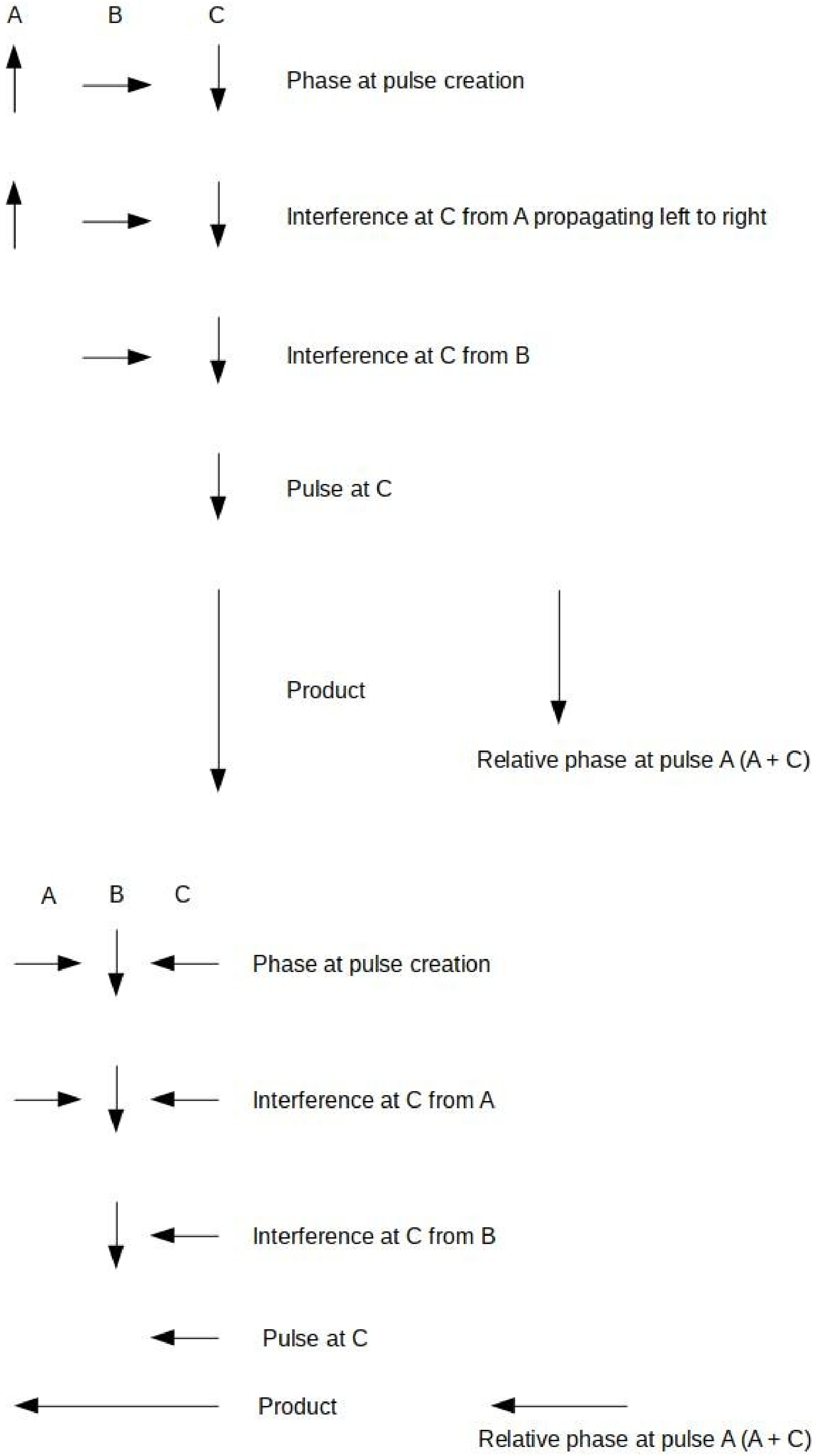
The central concept of this technique is the formation of standing waves along the pulse train. This is achieved by :

1. Sectioning each pulse train into repeating groups of 3 pulses A,B,C with each sequential pulse having a different phase according to its position in the grouping. One pulse train for the lower frequency and another pulse train for the higher frequency. Each pulse train encodes phase pulses according to the 3 pulse grouping i.e. ABC ABC ABC or BCA BCA BCA or CAB CAB CAB etc.

2. Adjusting the phase of each pulse along the pulse train according to the pulse's relative location in the pulse grouping. Phase pulses A, B and C each have a different phase relative to the required encode phase; to create the standing wave, each pulse of a three pulse grouping encodes the phase using a combination of three different phases relative to the original character phase ϕ and relative to the pulse train modulation frequency with wavelength λ . For this explanation, these phase values are as follows:- ϕ, ϕ + π/2 (phase at + λ/4), ϕ – π/2 (phase at - λ/4). The precise relative phase value for each pulse must adhere to certain principles which will be described in more detail later in this section, suffice to say, the numerical values chosen are in relation to modulation frequency and they determine both the location of the resulting standing wave phase pulse(s) as well as the number of standing wave phase pulses created. Referring back to the simple case of how phases combine, the above 3 phase combination would normally leave the wave phase unchanged however, because each phase is encoded on a different pulse in the pulse train, the resulting interaction from interference of wave pulses and phases is significantly more complex than the simple case.

The combined effect of these two steps is the formation of one or more standing waves of greater amplitude than the neighboring pulses at points along the pulse train. The standing waves repeat once at every 3 pulses and are relative to the 3 pulse groupings. A phasor representation is useful to explain further how the standing wave is utilized. The diagram below shows interference between the pulses and how the phase pulses propagate in time with a phase rotation and how they reinforce to increase amplitude and create the standing wave. Two phase vectors are shown; one at 0' and one at 90'. Similar product results are obtained with all other initial phase angles.

Diagram 1. Phasor



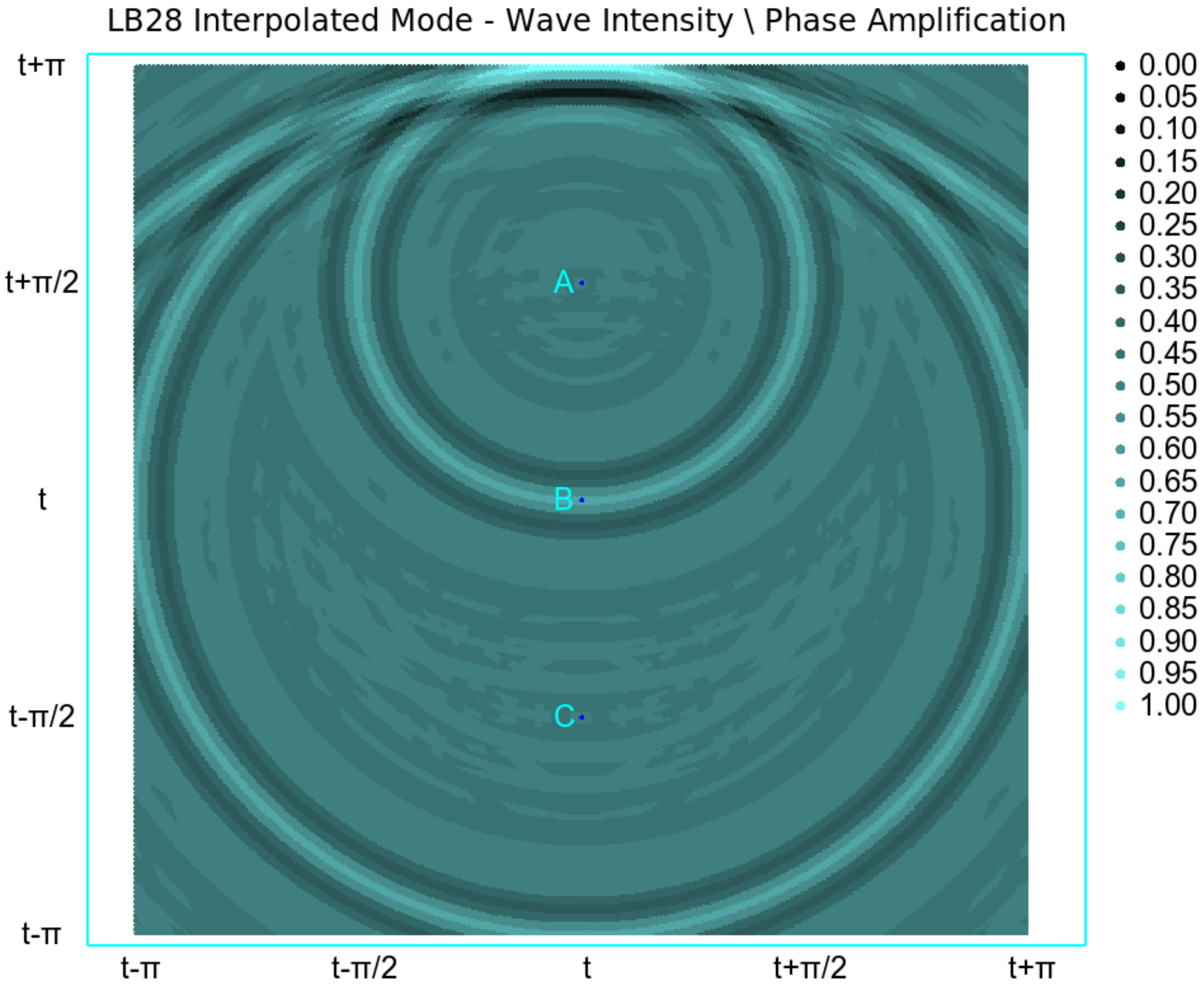
As can be seen in the above diagram, the relative phase obtained from the relative pulse addition of pulse A and pulse C gives an absolute phase value. The only inputs to deriving this phase value are the phase of pulse A and the phase of pulse C. The phasor diagram shows that the resulting phases are a result of both the magnitude and phase of adjacent pulses along the pulse train. Due to the standing wave effect from phase pulse interference, the inter-pulse phase difference between the standing wave pulse and the adjacent pulse at t+1 of the received wave, differs from the original transmitted wave’s zero phase by an amount equivalent to the absolute phase encoding of the original signal. The phasor diagram is a simplified version of the full range of interference effects but captures the essence of this mechanism.

Simulation

To further explain this effect, a computer simulation technique is utilized. This is achieved by plotting points on a 1000 x 1000 grid with an intensity value that represents phase. The precise phase at each point is derived from the combination of wave phases from source points A, B and C. These are equivalent to the phase pulses along the pulse train. These pulse origination points A, B and C emit pulses at different phases as described earlier. The phase values at each point on the simulation grid are proportional to the wavelength and the distance of the point from each of the source points A, B and C. Phase is equivalent to distance relative to wavelength.

The following simulation shows the three signal source points A, B and C for each of the phase pulses. The central point B is the in-phase ϕ character phase. The point A above this is ϕ + π/2 and point C is ϕ - π/2. The simulation shows the phase space as an X Y plane. The pulse train is aligned with the Y axis. Phase pulse offset is relative to wavelength.

Diagram 2. Standing Phase Wave



As can be seen, there is a significantly increased magnitude pulse above the location of the repeating pulse C along the pulse train. When the phase pulses combine, a wave of significantly greater intensity is formed at an offset of π. This pulse is the standing wave. This standing wave is essentially an amplified phase pulse. As the pulse spacing is many multiples of the wavelength, the standing wave that forms over every third pulse along the pulse train can be clearly differentiated from the other two pulses in the group as well as from the individual waves themselves. The presence of an amplified phase pulse alongside the normal phase pulses provides an absolute frame of reference for phase value extraction. The absolute phase value is obtained by a phase comparison between two points along the pulse train as explained in the earlier phasor diagram.

The significance of this phase derivation from relative pulse phase values, is that it is immune from what the individual waves are doing and consequently does not need to be synchronized, tracked or compared relative to anything else. This removes the need for phase tracking and simplifies the phase extraction process. In addition, the extra stage of amplification further complements the interpolated mode's signal to noise amplification process.

The precise mechanism for phase extraction can then be achieved in the downconversion process by taking the exponential of the standing wave at pulse t and adding it to the exponential of the adjacent pulse at pulse t+1 and applying a low pass filter to the resulting baseband phase. When the subsequent amplification process is performed by averaging every pulse along the pulse train to increase the signal intensity relative to the noise, the pulses retain these relative phase difference values.

As a further point of note, it is possible to create one or more standing waves relative to each pulse grouping. The precise nature of how the standing waves form is a direct result of the exact relative phase values used for the A, B and C phase pulses in the original signal modulation. The technique can also be extended to other groupings of pulses i.e. groups of 2 pulses or groups of 4 pulses. The precise technique used in the prototype application to generate the 3 phase signal is to interpose each of the individual components of the 3 phase signal by alternating at every sample.

Doppler Shift

Doppler shift correction via pulse train standing waves can be achieved with the following additional techniques:

1. Locate pulse peaks along the pulse train
2. Modify the signal to align pulses
3. Adjust extracted phase for pulse expansion / compression

Locate Pulse Peaks

The incoming signal is grouped into overlapping parts of around 18 pulses each with a 50% overlap. Each part is scanned to find the pulse offset resulting in maximum signal magnitude. The magnitude is derived from the sum of signal magnitudes over all 18 pulse peaks; a scan of all possible sample offsets from 0 to pulse length is done over each 18 pulse sample to determine at which offset the maximum occurs. If there is any measure of doppler shift, the resulting offsets for each part will have different values. If everything is perfectly aligned with no doppler shift, these values will be equal indicating perfect alignment. The values derived are within a margin of error that is dependent on sample rate; the higher the sample rate, the more offset errors arise however these remain small in relation to samples per wavelength.

Once the pulse peak center for each part has been identified, the resulting collection of parts is interpolated and smoothed using spline interpolation to interpolate the individual pulse offsets within each part.

The result is a series of points that represent the doppler shift at every pulse along the pulse train. This step is performed during the initial signal pre-processing prior to any fft filters and prior to pulse train detection and consequently does not need to conform to any fixed pulse train lengths.

Modify Signal

A new signal is then created by extracting each full pulse along the pulse train but allowing for the doppler shift to produce a correctly aligned pulse train. Any discontinuity in the underlying waves is irrelevant and inconsequential due to the nature of phase extraction using pulse train standing waves as described earlier.

It is important to note that the pulse peaks alter their displacement depending on the phase of the encoded signal. Aggregating these displacements into parts as described above is required for this process to function effectively. Additionally, each part should be divisible by 3. This is due to an effect that arises if the parts are not divisible by 3 that skews the phase disproportionately for each low / high signal component depending on the encoded phase.

Description of this effect follows, arrows are used to represent the phase shift of each pulse in a 3 pulse group sequence (‘<’ = left shift, ‘^’ = neutral, ‘>’ = right shift):

Low Signal: < ^ > < ^ >

High Signal: < ^ > < ^ >

Low / high signals follow sequentially and have equal length divisible by 3. Result is the signals have equal displacement (i.e. phase shift) as required for correct decodes.

Low Signal: < ^ > < ^ > <   (total displacement of <)

High Signal: ^ > < ^ > < ^   (total displacement of ^)

Although the low / high signals are sequential and of the same length, they are mismatched with unequal displacement / phase shift resulting in an incorrect decode.

Pulse Compression / Expansion

Due to the effects of doppler shift, pulses along the aligned pulse train will be compressed or expanded by different amounts. This amount is proportional to the amount of doppler shift corrected for in the previous alignment step.

To correct this, a phase adjustment value is derived by calculating the difference between the doppler shift that corresponds with the start and end of each character’s 3 bit encoding along the pulse train in terms of number of samples and then in terms of the number of full wavelengths. This is done by dividing the difference value by samples per wavelength. Samples per wavelength will be different for each of the two frequencies of an LB28 signal. The results of this calculation are then multiplied by a known factor and applied to the phase of each of the two signal components as derived during the phase extraction process. This step requires information that is not present during the pre processing stage and is therefore performed after pulse train calculation and during phase extraction.

These additional doppler shift correction techniques have been tested using a prototype and have successfully corrected for both constant velocity linear doppler shift as well as curved doppler shift resulting from acceleration and deceleration.

Diagram 3a. Doppler Effects - Acceleration and Deceleration

The following two diagrams show very typical doppler effects that are present on the computer test platform that uses CPU time slicing without hardware synchronization. The first is from the start of sampling and shows the initial acceleration then ramping up and deceleration of the receive sampling from offset -28 to offset 145. This positive gradient indicates signal expansion. At approximately mid way through the sample, the transmitted and received signals are in sync for a short period of time. The sample rates then diverge again with a short deceleration of the receive sample rate leading to a constant velocity negative gradient straight line indicating signal compression. The tail end of the chart is due to noise only where there is no signal present.

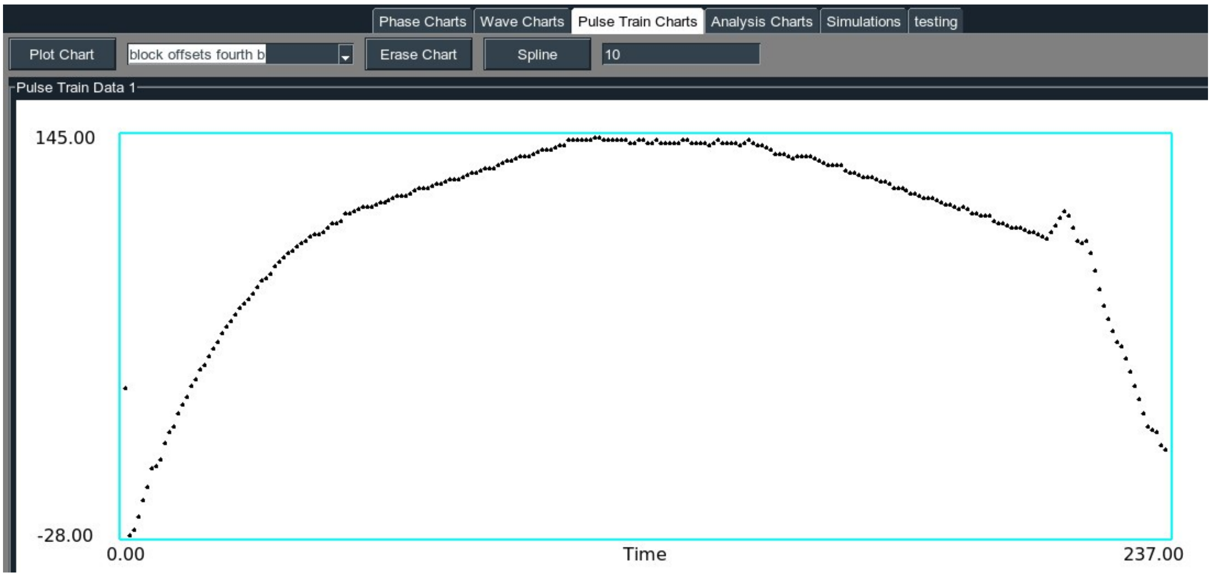
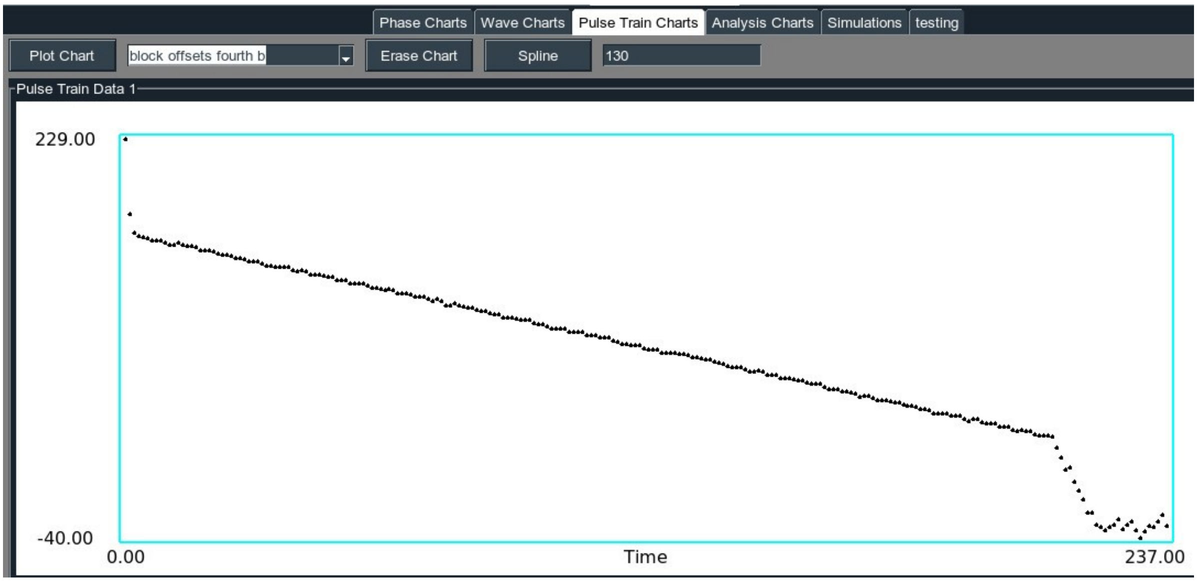


Diagram 3b. Doppler Effects – Constant Relative Velocity

The second diagram shows the doppler effects of subsequent samples. This shows a relatively linear constant relative velocity straight line indicating signal compression. The tail end of the chart has only noise and no signal present.



Alignment and Calibration

It is important to understand that any phase correction required for phase extraction using pulse train standing waves remains constant along the entire sequence of characters in the pulse train, as long as the disposition of the pulse train remains constant. Another way to put this is that all characters along the pulse train will retain their correct phase values relative to each other. The only unknown is the disposition of the pulse train itself i.e. where it starts relative to a full and complete pulse train, its length and where it ends. If the disposition is known, then a fixed phase rotation can be applied to extract all character phases. In the absence of a known pulse train disposition, it is necessary to use other techniques such as an auto rotation and alignment sequence to align the pulse train.

Fixed Alignment

Fixed alignment where the signal is assumed to have an absolute phase disposition has been proven to work but the technique is not resilient to high levels of AWGN; at relatively small amounts of AWGN, the accuracy of signal decodes decreases rapidly. This is most likely due to the changing disposition of the pulse train subject to both the start and length of the detected pulse train. Without any determination about the full disposition of the pulse train outside of what is actually observed at the received signal, fixed alignment becomes less practical in environments with moderate to high levels of white gaussian noise.

Auto Alignment

Auto Alignment is achieved by the use of a fixed length sequence of repeating characters at the start of the signal. These characters are used to match phase based on a minimum of 2 characters of equal phase. For example, an auto align sequence of 2_of_3 will attempt to locate the same phase in a minimum of two out of the first three characters. Each low / high signal part is aligned separately. Similarly, an auto align of 2_of_4 will attempt to locate a minimum of 2 characters with matching phase in the first four characters. Using this technique, the correct alignment of the pulse train can be determined.

Calibration

During the auto align process, the minimum phase division for 8psk i.e. $\pi / 4$ is divided into a number of potential alignments. Each alignment is checked separately to derive the maximum number of characters that match phase among all repeating auto alignment characters. This will typically produce a range of potential alignment values that can be used for subsequent decodes. The alignment value in the middle of this range is chosen as being the most likely to produce minimal errors in subsequent character decodes. Please note, the range typically wraps and will appear at both the start and end of the set of test alignments but not in the middle. This requires merging of the two ranges so that the correct mid-point can be obtained.

Pulse Train Disposition

Pulse train disposition is essentially the combined phase effects of all pulses in the detected pulse train for a given character’s 3 bit sequence. This depends on two things, the length of the pulse train and the offset at which the detected pulse train starts relative to where it should start.

These phase values can be calculated using formulae however, a more direct approach is to use simulation with a noise value of zero. The following tables of data shows a subset of rotations (lo, hi) derived via simulation for each offset of a given length pulse train starting at offset zero with lengths 20 thru 24.

Pulse Train Rotation Values (lo,hi) at Sequential Offsets Starting at 0	
20	[(4.705475688725958, 5.253871744772425), (0.3437505652066264, 5.0), (0.9214601836602547, 3.0), (2.764380550980766, 5.811946409141117), (4.4894911937916575, 5.4894911937916575), (0.38302047337649725, 3.4305863315368494), (0.764380550980766, 1.6662057805560844), (2.7447455968958288, 6.145740628585033), (4.293141652942296, 5.725110642810893), (0.22594084069700848, 3.6858407346410207), (0.3913164233669777, 1.8625553214054467), (3.9214601836602547, 0.12776607027232778)]
21	[(3.6858407346410207, 3.3127766070272333), (5.4894911937916575, 1.2735066988573607), (5.733406592801373, 5.753041546886308), (1.293141652942296, 5.607300918301275), (4.882190275490384, 3.5483960560464674), (0.5008301978861152, 1.5091261478765947), (0.7251106428108933, 6.027930904075416), (2.5483960560464674, 5.94109513774519), (4.391316423366977, 3.8036504591506386), (0.029591299847645303, 1.764380550980766), (0.23423679068748804, 6.224280444924777)]
22	[(3.6858407346410207, 3.3127766070272333), (5.4894911937916575, 1.2735066988573607), (5.733406592801373, 5.753041546886308), (1.293141652942296, 5.607300918301275), (4.882190275490384, 3.5483960560464674), (0.5008301978861152, 1.5091261478765947), (0.7251106428108933, 6.027930904075416), (2.5483960560464674, 5.94109513774519), (4.391316423366977, 3.8036504591506386), (0.029591299847645303, 1.764380550980766)]
23	[(3.6858407346410207, 1.3716814692820414), (5.4894911937916575, 5.635231822376691), (5.733406592801373, 5.548396056046467), (1.293141652942296, 3.6858407346410207), (4.882190275490384, 1.6073009183012754), (0.5008301978861152, 5.85121631731099), (0.7251106428108933, 5.921460183660255), (2.5483960560464674, 2.234236790687488), (4.391316423366977, 1.8625553214054467)]
24	[(2.410951377451914, 1.3716814692820414), (4.234236790687488, 5.635231822376691), (6.1850105367549055, 5.548396056046467), (1.7447455968958288, 3.6858407346410207), (3.58766596421634, 1.6073009183012754), (5.430586331536851, 5.85121631731099), (5.576326960121882, 5.921460183660255), (2.9214601836602547, 2.234236790687488)]

Extrapolation

An optional additional phase extraction technique for added noise resilience is with the use of extrapolation. This is a technique to derive the full pulse train disposition to further significantly improve noise resilience and maximize the effectiveness of decodes. The basic principle is this:-

- Required pulse train phase rotation is relative to pulse train start position and pulse train length i.e. its disposition
- Required pulse train phase rotation can be determined by the use of a fixed character sequence at the start of the transmission
- With values for actual rotation and known pulse train length it is possible, with the use of rotation tables, to extrapolate the entire pulse train to derive a complete set of extraction points thus maximizing the phase extraction relative to bit error rate.

A challenge for this process is that if any equal or near equal rotation values (lo,hi) occur in more than one place in the rotation table for a detected pulse train of given length, then extrapolation will need additional information to resolve this ambiguity. Without additional information, only a partial extrapolation of the pulse train using auto rotation and calibration can be achieved. To resolve this, a second set of lo hi rotation values can be derived using the next shortest pulse train length with substantially different unique values by re-processing the fixed character sequence. This provides an additional point of reference from which to determine the correct extrapolated pulse train; there will be one common extrapolation solution that exists for both sets of rotation values. A quick browse through the rotation tables shows that the vast majority of low hi rotation values for LB28 modes i.e. 2 carriers, appear to have unique fingerprints which makes this approach viable in many instances even with a single set of rotation values. With the second set of rotation values and two or more carriers, this technique provides a reliable and substantial improvement over non extrapolated decodes.

Just to round out the description of this technique:-

- the detected pulse train will be used to determine pulse train rotation
- the detected pulse train length will be used to determine which rotation table to search
- the pulse train rotation value or close match is then located in the rotation table. When a close match is found, this provides the remaining information of where the pulse train actually starts and ends for all codings along the entire pulse train and transmission.
- once the initial set of rotation characters has been processed and the pulse train disposition determined, the full pulse train is extrapolated and the pre-calculated precise fixed rotation value for the complete pulse train is then used for any subsequent decodes.

The initial sequence of characters at the start of the transmission is in essence an extrapolation sequence. This is similar in effect to the auto rotation sequence, but what sets this apart is that the rotation values derived do not need to be absolutely precise and do not need to be calibrated to any degree if at all. The values only need to be precise enough to determine which pulse train disposition is in effect.

Additionally, the extrapolation sequence can be considered separate from the remaining transmission and can make use of additional carriers. For example, an extrapolation sequence using an eight character sequence and 2 or 3 carriers could be used to determine pulse train disposition which then provides the basis for all subsequent transmission decodes of a single carrier transmission. This is possible as long as the transmission from extrapolation sequence to main transmission is continuous. Extending this idea a little further leads to the concept of an initialization sequence that contains not only the fixed character extrapolation sequence to determine pulse train disposition but additional information such as a transmission mode identifier used to identify the mode for the subsequent transmissions.

What makes this technique especially interesting, is that neither the rotation nor the pulse train need to be determined with any great accuracy. What matters is that the values retrieved are only sufficient to determine pulse train disposition so that extrapolation of the pulse train can be completed and the pre-calculated precise rotation values for the full pulse train can be applied to decode the signal to maximum effect. Details such as carrier separation and number of carriers used in the extrapolation sequence and how many rotation values are derived can positively impact the effectiveness of this technique.

Rotation Clusters

The following diagram shows a range of pulse train lengths (11 – 32) along with a count of the number of times each was detected in the sample data set. Using auto-rotation with an 8 character rotation and calibration sequence, the resulting rotations for low frequency and high frequency parts of the LB28 signal are plotted on a two dimensional plane; each test results in a single point with low rotation on the x axis and high rotation on the y axis from 0 to 2 * pi. The data set was generated over a wide variety of noise levels from AWGN 0 to AWGN 6. As can be seen, the resulting rotations for the decodes form a set of clusters or loci on the 2d plane. The precise set of loci changes considerably for each pulse train length as can be seen in the diagram. Pulse train length i.e. pulse train disposition therefore plays a significant role in the determination of the actual rotation for a correct decode.

Diagram 4a. Pulse Train Rotation Clusters – by Pulse Train Length

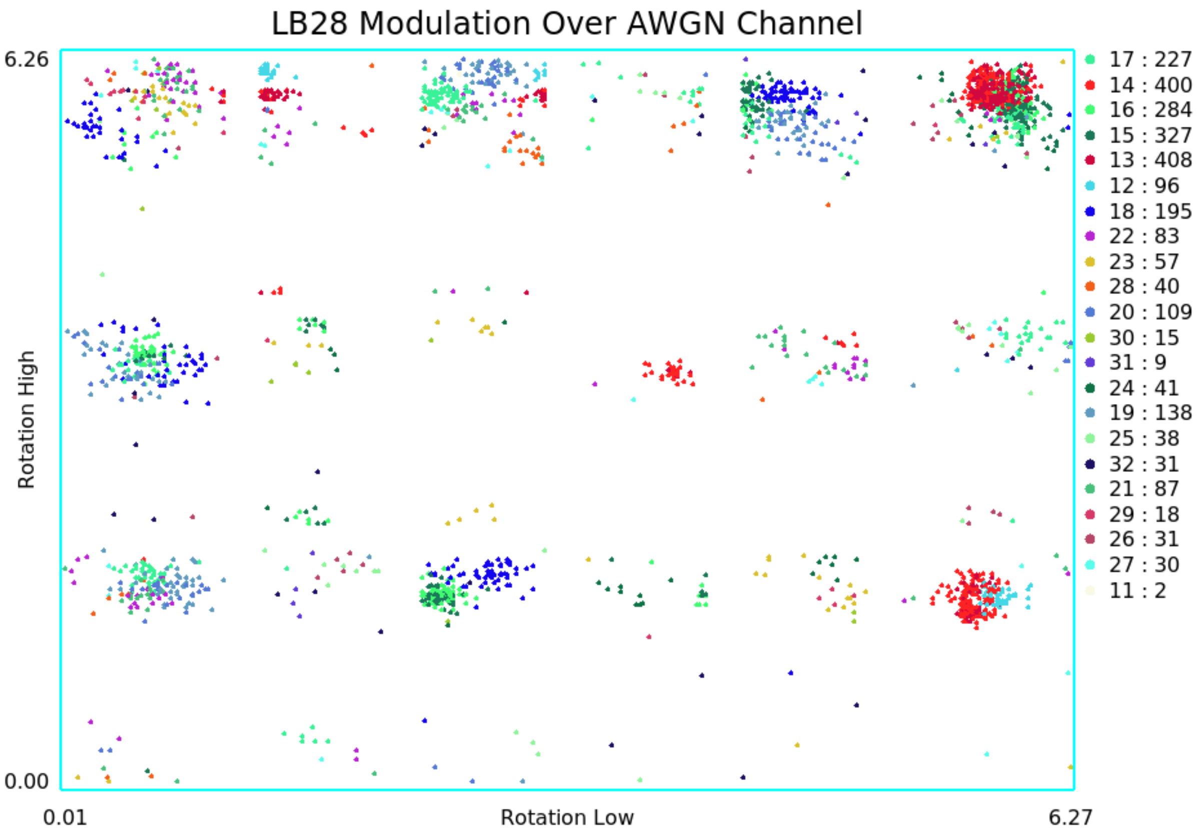
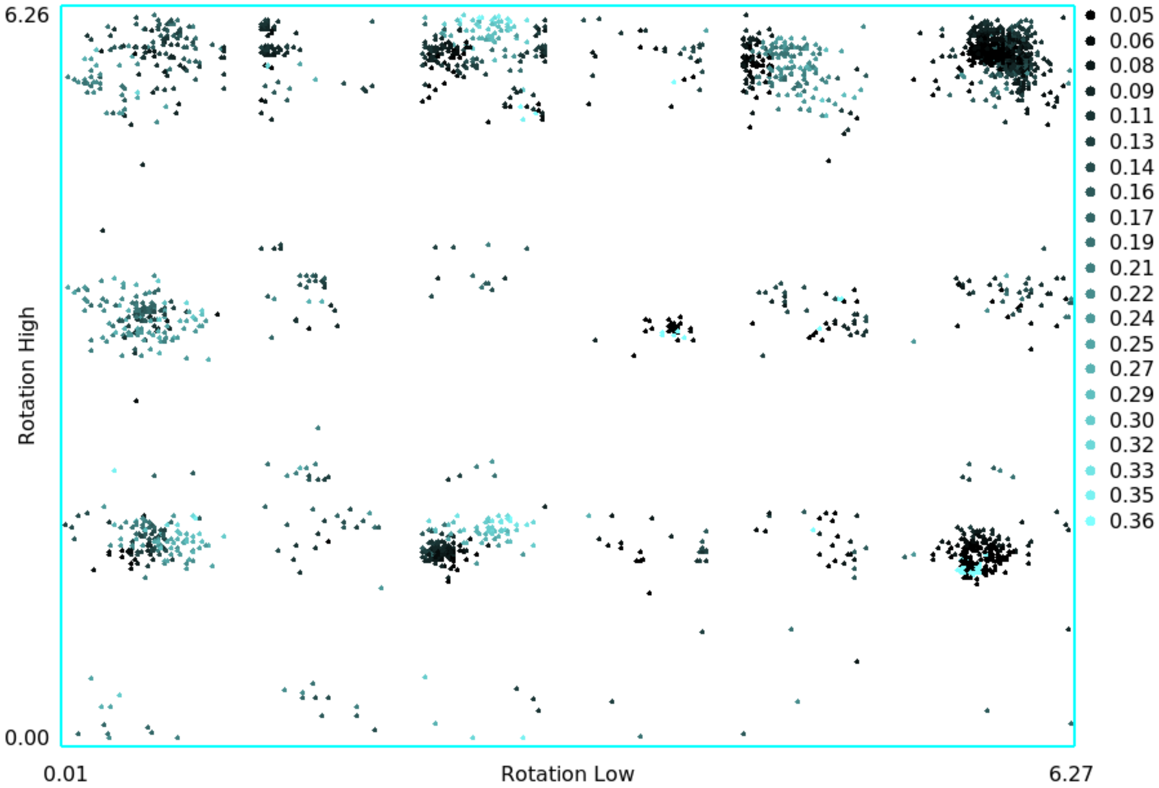


Diagram 4b. Pulse Train Rotation Clusters – by Bit Error Rate

The following diagram extends the diagram 4a above to show how accurately each point in the data set was decoded. The legend on the right represents bit error rate using a color intensity. As can be seen, each of the loci on the diagram represent highly accurate decodes, darker colors typically represent a bit error rate accuracy of 0.16 or greater.



Root Raised Cosine - Pulse Shapes

The following diagram shows the degree to which different pulse shapes as defined by their alpha and T values have an impact on both the power density of the signal as well as the bit error rate. The following samples were all taken using the same LB28 mode at a fixed 7.0 AWGN. The only difference is the precise nature of the actual random noise and the RRC pulse shape.

The small set of 14 pulse shapes has been fined tuned from a massive amount of data to select the pulse shapes that are the most efficient with respect to power density and bit error rate.

Diagram 5. RRC Pulse Shapes – by Shape

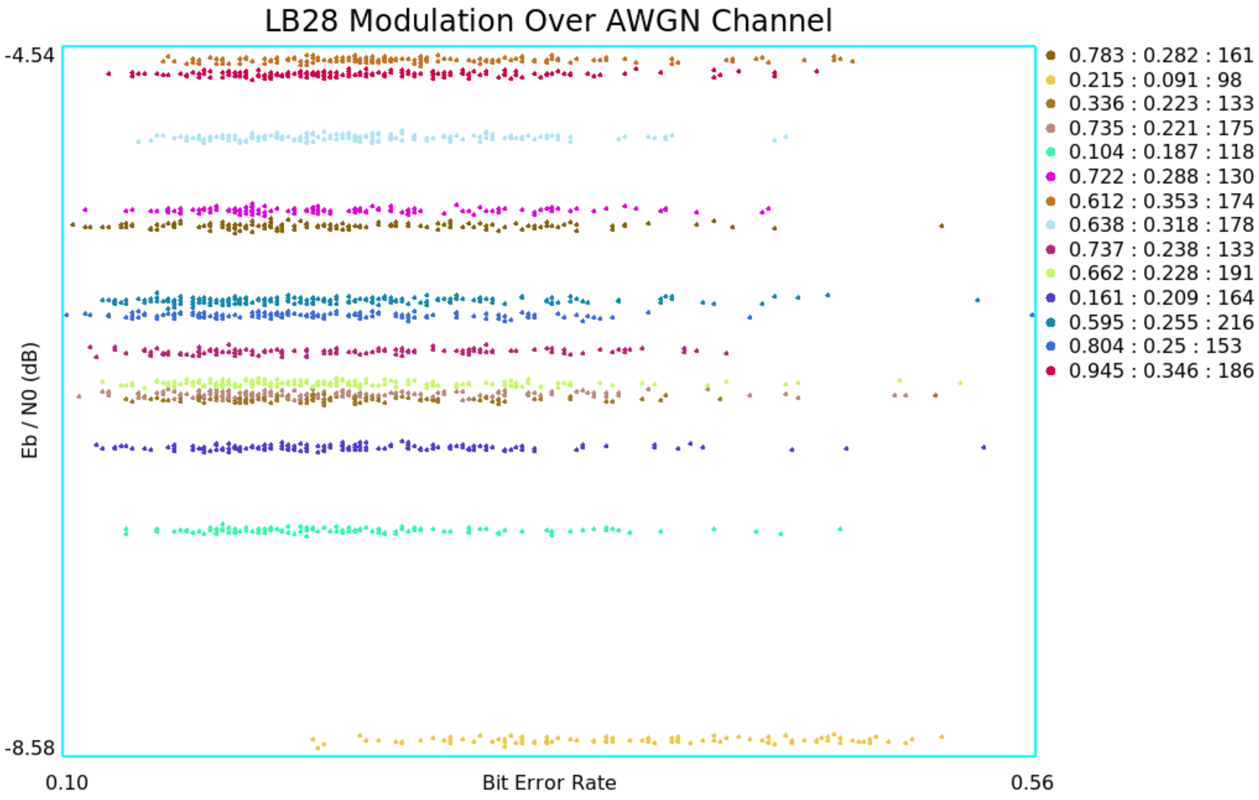
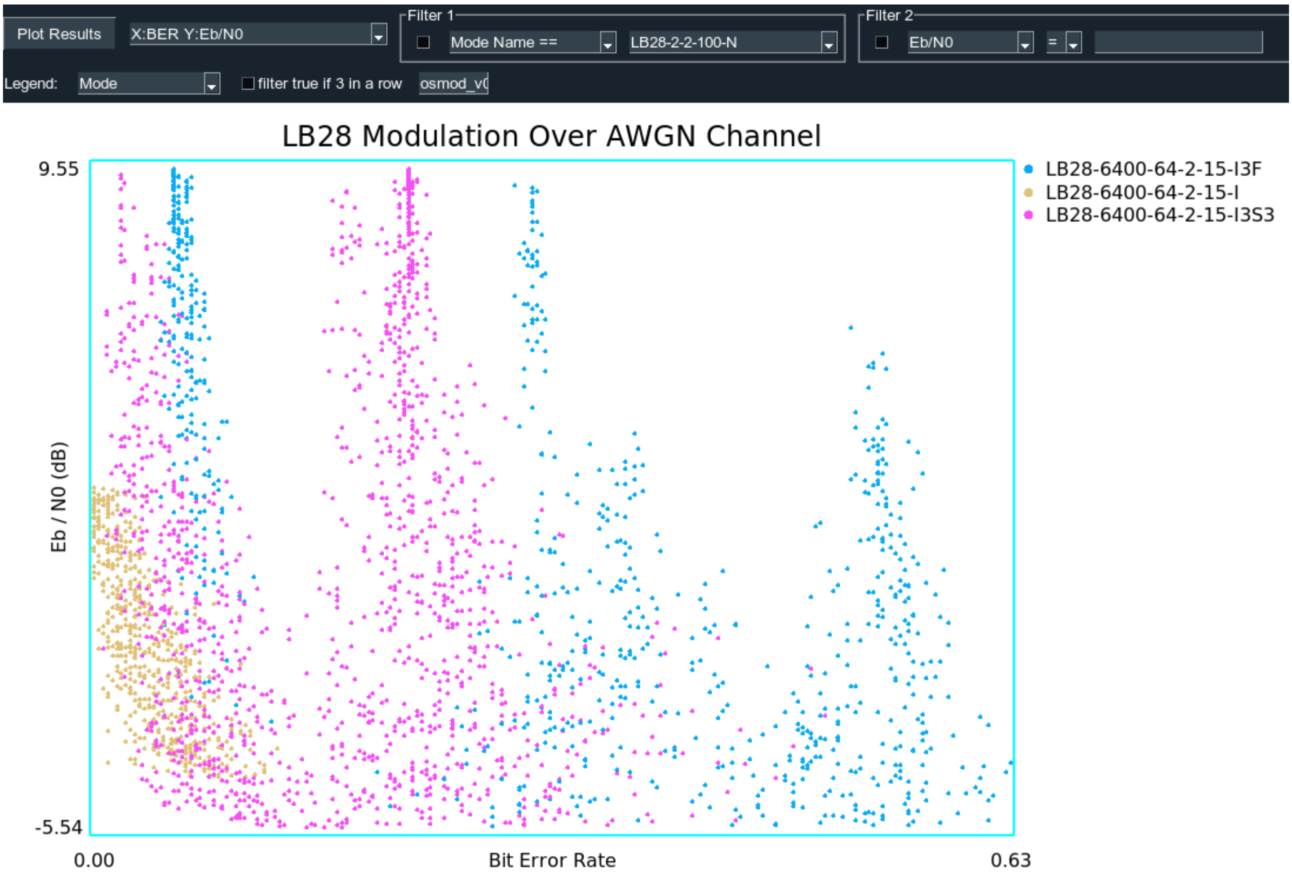


Diagram 6a. Phase Extraction Comparison – by Type

The diagram below shows comparison test data using three different phase extraction techniques; absolute phase recovery, auto rotation phase recovery, base mode using costas loop. The test is over an AWGN channel using LB28-6400-64-2-15-I, LB28-6400-64-2-15-I3F and LB28-6400-64-2-15-I3S3 modes. I denotes interpolated base mode, I3 at the end denotes this is an interpolated mode using intra-block 3 phase pulse delineation. F denotes a fixed phase rotation and S3 denotes 2_of_3 start sequence. The I3 modes both utilize relative phase obtained from two neighboring pulses in the baseband signal downconversion combined with doppler shift correction. Please note the diagram shows only AWGN and does not show any results relative to phase noise. Both I3 modes with doppler correction have demonstrated significantly improved doppler correction over the base mode. The modes are still experimental but appear to have potential for further development and refinement.



Plot Results X:BER Y:Eb/N0

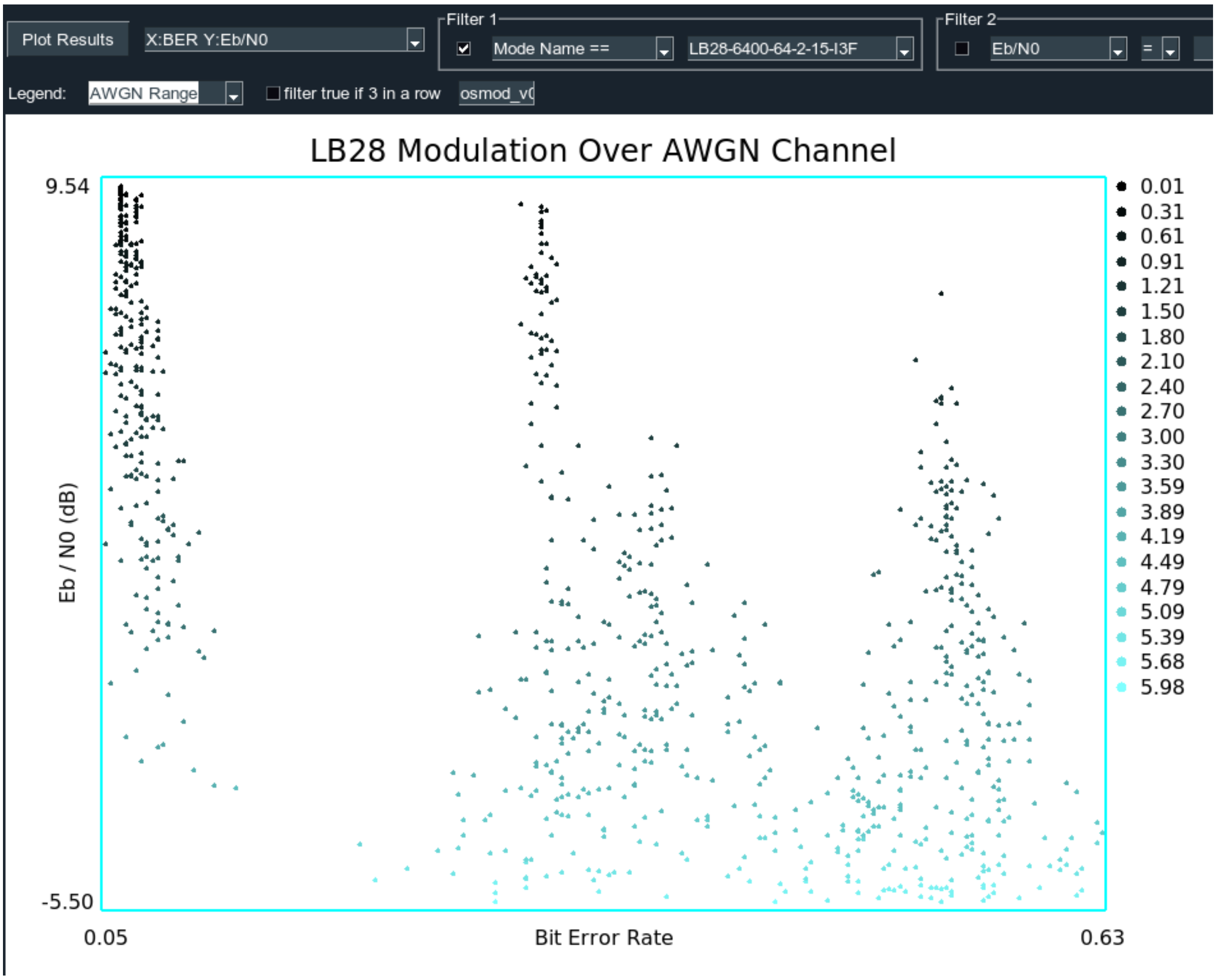
Filter 1 □ Mode Name == LB28-2-2-100-N

Filter 2 □ Eb/N0 =

Legend: Mode □ filter true if 3 in a row osmod_v(

Diagram 6b. Phase Extraction Comparison – by AWGN

The diagram below shows the I3F absolute phase recovery mode with different degrees of AWGN as denoted by the legend of intensity colors. As can be seen, the mode works most effectively at low AWGN (top left). The other interesting feature is there appear to be 3 distinct groupings to the data. This could be due to tolerance limitations and/or errors in the pulse detection and alignment process of the prototype application.



Plot Results X:BER Y:Eb/N0

Filter 1 ☑ Mode Name == LB28-6400-64-2-15-I3F

Filter 2 □ Eb/N0 =

Legend: AWGN Range □ filter true if 3 in a row osmod_v(

LB8

The ability to extract phase on an absolute basis from adjacent phase pulses, makes possible a new single carrier I3 mode with 8psk modulation, signal to noise amplification via interpolation and relative phase pulse phase extraction. This new mode is in theory highly resilient to:-

1. Misaligned timing between transmitting station and receiving station,
2. Phase distortion over the channel,
3. Gaussian white noise over the channel,
4. Frequency misalignment between transmit and receive hardware.

This new mode is LB8.

LB8 uses 2 blocks per character. Each 3 bit part of the 8psk sequence is sent as a serial stream i.e. 3bits + 3bits + 3bits + 3bits etc on a single carrier with 8psk. The character sequence is optionally enclosed in bookends; one or two repeats of a unique special character at the start of the sequence and one or two repeats of an additional special character at the end. The same technique can also be extended to 4psk/qpsk i.e. LB4 and 2psk/bpsk i.e. LB2.

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 statistical 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 iterations 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 proceed from low index to high index or high index to low index with wrap around. The end result is a clean contiguous set of indices. These represent the points along the signal wave.

- Additional interpolation steps are then performed 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 corresponding 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.

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.

The following additional optimizations are also possible but have not been included in the preliminary testing:-

- 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.

- Iterate through a list and fill out the partner list with the corresponding partner index if the index is not already present in the partner list. This needs to be done both ways i.e. on the first list and then on the second list.

- The list indices are originally derived from iterating over the entire chunk of data. In the preliminary tests, these chunks are approximately 33 characters in length. To further enhance signal amplification in the final stage where the phase value is mean averaged for the entire detected set of indices, a process could be incorporated to correlate indices over multiple chunks. The goal is to derive a full set of indices. At whatever point this occurs in the decode process, the full set of indices can then be used to pull out the data to the max for any previously received \ decoded data and any future data. For a given transmission, assuming that the transmit and receive stations are either stationary or moving at a constant velocity relative to each other, the signal will be essentially fixed to a clock and should not deviate on aggregate. Once the full set of indices is established, this effectively establishes a ‘clock lock’ and the full delineation of all blocks for this transmission, past present and future, is known i.e. where the block starts relative to the signal and also relative to a clock, where the block ends relative to its start point and where every piece of transmitted data is hiding inside the block. This data can then be extracted with maximum efficiency.

Carrier Optimization – Overlapping Blocks

Increasing the baud rate of the higher speed modes is achieved by adding additional carriers. Typically this is done by utilizing additional frequency blocks in the adjacent spectrum space.

To maximize carrier density, a technique of overlapping blocks can be implemented as follows:-

- Each parallel block is synchronized so that all parallel blocks are sending their respective modulated signals in-phase.
- By overlapping the blocks, it is possible to use bandwidth from the adjacent channel group space.

For example:-

Lets assume the following modulations:- LB28 in AB format and LB3Q in ACBB format. These blocks can be overlapped in the spectrum space as follows:-

LB28

No overlap 4 Carriers,

```
|    |
|A1  |
|    |
|  B1|
|    |
|A2  |
|    |
|  B2|
|    |
```

With overlap 3 Carriers

```
|    |
|A1  |
|    |
|A2B1|
|    |
|  B2|
|    |
```

LB3Q

No overlap 6 Carriers

```
|        |
|A1      |
|        |
|    B1B1|
|        |
|  C1    |
|        |
|A2      |
|        |
|    B2B2|
|        |
|  C2    |
|        |
```

With overlap 5 Carriers

```
|        |
|A1      |
|        |
|    B1B1|
|        |
|A2C1    |
|        |
|    B2B2|
|        |
|  C2    |
|        |
```

The above example shows that it is possible to achieve up to a 25% increase in carrier density by overlapping blocks in this manner. With more carrier groups this number tends towards a 50% increase for LB28 and other 2 carrier modes.

A key point to note is that the outer blocks in the frequency spectrum contain the information required to exactly delineate all in-phase block boundaries. For example, four groups of LB28 will look like the following with overlapping blocks:-

```
  |    |
  |A1  |
  |    |
  |A2B1|
  |    |
  |A3B2|
  |    |
  |A4B3|
  |    |
  |  B4|
  |    |
```

This is a reduction from 8 carriers to 5. The outer two blocks A1 and B4 show the block boundaries.

A further point of note, is that the higher baud rate modes are able to achieve tight carrier separation through orthogonality. When adjacent carrier groups with overlapping blocks are used, to maintain orthogonality and keep ISI to a minimum, the adjacent group(s) must be offset or interlaced by timing equivalent to between one and three additional RRC pulses. The net effect of this is that initially increasing carriers and baud rate is offset by a reduction in baud rate due to the increased time required for interlacing however, the total net gain in carrier density is significant. Once interlacing has been established, any further multiplication of carriers achieves a true 1 for 1 multiplication in baud rate.

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 2 chunks of modulated data. 67 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 interpolated 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)

    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
```

Appendix 2: Baseband Downconversion Using Relative Phase

The code snippet below shows the actual code used to derive the relative phases from two pulses (sig1 and sig2).

```
complex_exponential_lower  = np.exp(-1j * 2 * np.pi * frequency[0] * time)
complex_exponential_higher = np.exp(-1j * 2 * np.pi * frequency[1] * time)
complex_exponential = [complex_exponential_lower, complex_exponential_higher]

def deriveProduct(low_hi_index, sig1, sig2):
        baseband_sig = complex_exponential[low_hi_index] * (sig1 + sig2)
        return self.filter_low_pass(baseband_sig, frequency[low_hi_index] - 115)
```

Appendix 3: I3 Mode Pulse Averaging

```
     def deriveCombinationPulses(block, fine_tune_pulse_start_index):
        if index % 3 == 0 and max_a < max_useable:
          if has_pulse_a == True:
            pulse_a_real = pulse_a_real + block[offset + fine_tune_pulse_start_index + (index * 
pulse_length):offset + fine_tune_pulse_start_index + ((index+1) * pulse_length)].real
            pulse_a_imag = pulse_a_imag + block[offset + fine_tune_pulse_start_index + (index * 
pulse_length):offset + fine_tune_pulse_start_index + ((index+1) * pulse_length)].imag
          else:
            has_pulse_a = True
            pulse_a_real = block[offset + fine_tune_pulse_start_index + (index * 
pulse_length):offset + fine_tune_pulse_start_index + ((index+1) * pulse_length)].real
            pulse_a_imag = block[offset + fine_tune_pulse_start_index + (index * 
pulse_length):offset + fine_tune_pulse_start_index + ((index+1) * pulse_length)].imag

          max_a = max_a + 1
          add_pulse_to_all = True
        elif (index+2) % 3 == 0 and max_b < max_useable:
          if has_pulse_b == True:
            pulse_b_real = pulse_b_real + block[offset + fine_tune_pulse_start_index + (index * 
pulse_length):offset + fine_tune_pulse_start_index + ((index+1) * pulse_length)].real
            pulse_b_imag = pulse_b_imag + block[offset + fine_tune_pulse_start_index + (index * 
pulse_length):offset + fine_tune_pulse_start_index + ((index+1) * pulse_length)].imag
          else:
            has_pulse_b = True
            pulse_b_real = block[offset + fine_tune_pulse_start_index + (index * 
pulse_length):offset + fine_tune_pulse_start_index + ((index+1) * pulse_length)].real
            pulse_b_imag = block[offset + fine_tune_pulse_start_index + (index * 
pulse_length):offset + fine_tune_pulse_start_index + ((index+1) * pulse_length)].imag

          max_b = max_b + 1
          add_pulse_to_all = True
        elif (index+1) % 3 == 0 and max_c < max_useable:
          if has_pulse_c == True:
            pulse_c_real = pulse_c_real + block[offset + fine_tune_pulse_start_index + (index * 
pulse_length):offset + fine_tune_pulse_start_index + ((index+1) * pulse_length)].real
            pulse_c_imag = pulse_c_imag + block[offset + fine_tune_pulse_start_index + (index * 
pulse_length):offset + fine_tune_pulse_start_index + ((index+1) * pulse_length)].imag
          else:
            has_pulse_c = True
            pulse_c_real = block[offset + fine_tune_pulse_start_index + (index * 
pulse_length):offset + fine_tune_pulse_start_index + ((index+1) * pulse_length)].real
            pulse_c_imag = block[offset + fine_tune_pulse_start_index + (index * 
pulse_length):offset + fine_tune_pulse_start_index + ((index+1) * pulse_length)].imag

          max_c = max_c + 1
          add_pulse_to_all = True

        if add_pulse_to_all:
          if has_pulse_all == True:
            pulse_all_real = pulse_all_real + block[offset + fine_tune_pulse_start_index + (index * 
pulse_length):offset + fine_tune_pulse_start_index + ((index+1) * pulse_length)].real
            pulse_all_imag = pulse_all_imag + block[offset + fine_tune_pulse_start_index + (index * 
pulse_length):offset + fine_tune_pulse_start_index + ((index+1) * pulse_length)].imag
          else:
            has_pulse_all = True
            pulse_all_real = block[offset + fine_tune_pulse_start_index + (index * 
pulse_length):offset + fine_tune_pulse_start_index + ((index+1) * pulse_length)].real
            pulse_all_imag = block[offset + fine_tune_pulse_start_index + (index * 
pulse_length):offset + fine_tune_pulse_start_index + ((index+1) * pulse_length)].imag
```

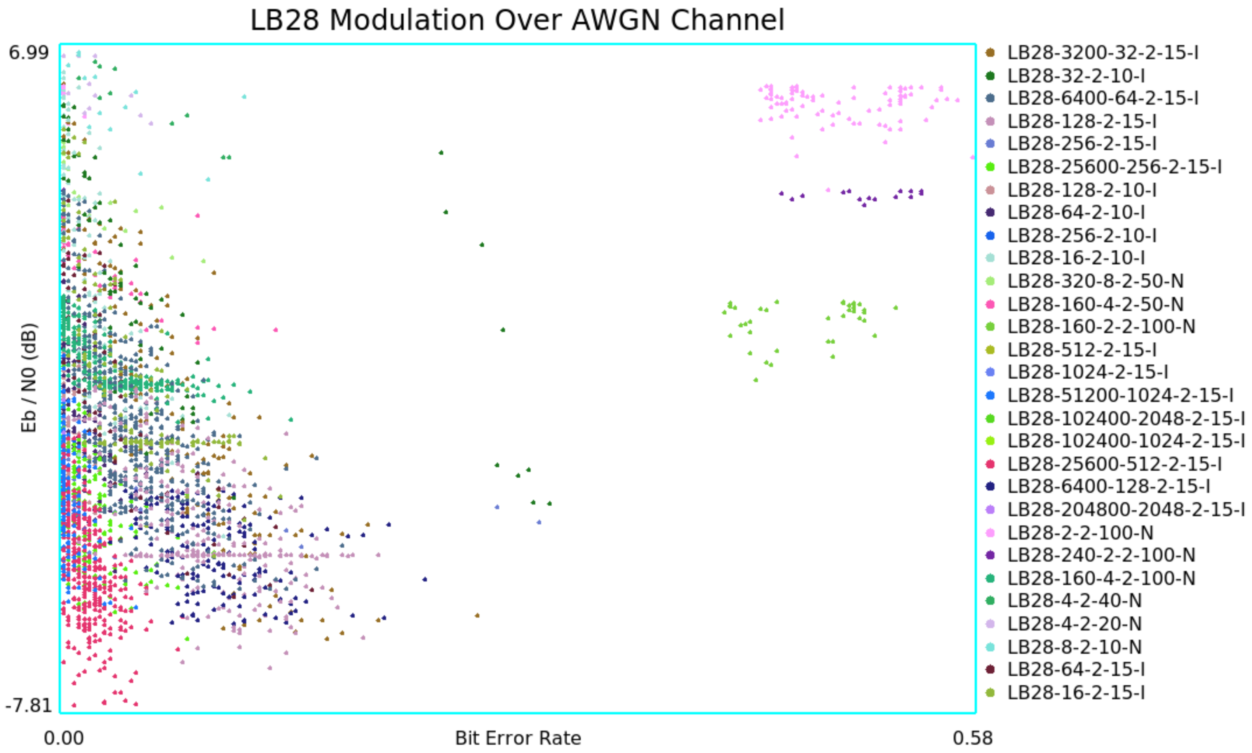
```
        for index in interpolated_lower:
          pulse_count = pulse_count + 1
          max_useable = max_useable_lower
          deriveCombinationPulses(audio_block1, fine_tune_pulse_start_index_lower)

        lower_pulse_a   = pulse_a_real + 1j   * pulse_a_imag
        lower_pulse_b   = pulse_b_real + 1j   * pulse_b_imag
        lower_pulse_c   = pulse_c_real + 1j   * pulse_c_imag
        product_pulse_lower3 = lower_pulse_b + lower_pulse_c


        for index in interpolated_higher:
          pulse_count = pulse_count + 1
          max_useable = max_useable_higher
          deriveCombinationPulses(audio_block2, fine_tune_pulse_start_index_higher)

        higher_pulse_a   = pulse_a_real + 1j   * pulse_a_imag
        higher_pulse_b   = pulse_b_real + 1j   * pulse_b_imag
        higher_pulse_c   = pulse_c_real + 1j   * pulse_c_imag
        product_pulse_higher3 = higher_pulse_b + higher_pulse_c

        """write the data back to the data stream """
        for i in range(0, self.osmod.pulses_per_block):
          if i in interpolated_lower:
            if i // max_useable_lower == 0:
              audio_block1[offset + fine_tune_pulse_start_index_lower+(i * pulse_length):offset +
fine_tune_pulse_start_index_lower + ((i+1) * pulse_length)] = product_pulse_lower3
            elif i // max_useable_lower == 1:
              audio_block1[offset + fine_tune_pulse_start_index_lower+(i * pulse_length):offset +
fine_tune_pulse_start_index_lower + ((i+1) * pulse_length)] = product_pulse_lower3
            elif i // max_useable_lower == 2:
              audio_block1[offset + fine_tune_pulse_start_index_lower+(i * pulse_length):offset +
fine_tune_pulse_start_index_lower + ((i+1) * pulse_length)] = product_pulse_lower3

          if i in interpolated_higher:
            if i // max_useable_higher == 0:
              audio_block2[offset + fine_tune_pulse_start_index_higher+(i * pulse_length):offset +
fine_tune_pulse_start_index_higher + ((i+1) * pulse_length)] = product_pulse_higher3
            elif i // max_useable_higher == 1:
              audio_block2[offset + fine_tune_pulse_start_index_higher+(i * pulse_length):offset +
fine_tune_pulse_start_index_higher + ((i+1) * pulse_length)] = product_pulse_higher3
            elif i // max_useable_higher == 2:
              audio_block2[offset + fine_tune_pulse_start_index_higher+(i * pulse_length):offset +
fine_tune_pulse_start_index_higher + ((i+1) * pulse_length)] = product_pulse_higher3
```

Appendix 4: Interposed Three Phase Signal Generator

```
for(int i = 0; i < num_phases; i++) {
        for(int k = 0; k < symbol_block_size; k++) {
          term5 = 2 * M_PI * time[k];
          term6 = term5 * frequency[0];
          term7 = term5 * frequency[1];
          /* frequency 0 */
          term8a = term6 + phases1[i];
          term8b = term6 + phases1[i] + (2*M_PI*offsets[0]);
          term8c = term6 + phases1[i] + (2*M_PI*offsets[1]);
          /* frequency 1 */
          term9a = term7 + phases2[i];
          term9b = term7 + phases2[i] + (2*M_PI*offsets[2]);
          term9c = term7 + phases2[i] + (2*M_PI*offsets[3]);
          pulses_wave1 = (double)symbol_wave_1_of_2[(int)(log2(pulses_per_block))-1][(int)(k /
pulse_length)];
          pulses_wave2 = (double)symbol_wave_2_of_2[(int)(log2(pulses_per_block))-1][(int)(k /
pulse_length)];

          intra_pulse_a = (3 - ((k+3) % 3)) / 3;
          intra_pulse_b = (3 - ((k+2) % 3)) / 3;
          intra_pulse_c = (3 - ((k+1) % 3)) / 3;

          if (intra_pulse_a == 1) {
            term10a = (amplitude * cos(term8a) + amplitude * sin(term8a)) * (pulses_wave1 *
filtRRC_coef_main[k % pulse_length]);
            term11a = (amplitude * cos(term9a) + amplitude * sin(term9a)) * (pulses_wave2 *
filtRRC_coef_main[k % pulse_length]);
            term10b = (amplitude * cos(term8b) + amplitude * sin(term8b)) * (pulses_wave1 *
filtRRC_coef_pre[k % pulse_length]);
            term11b = (amplitude * cos(term9b) + amplitude * sin(term9b)) * (pulses_wave2 *
filtRRC_coef_pre[k % pulse_length]);
            term10c = (amplitude * cos(term8c) + amplitude * sin(term8c)) * (pulses_wave1 *
filtRRC_coef_post[k % pulse_length]);
            term11c = (amplitude * cos(term9c) + amplitude * sin(term9c)) * (pulses_wave2 *
filtRRC_coef_post[k % pulse_length]);
             }
          else if(intra_pulse_b == 1 ){
            term10b = (amplitude * cos(term8b) + amplitude * sin(term8b)) * (pulses_wave1 *
filtRRC_coef_main[k % pulse_length]);
            term11b = (amplitude * cos(term9b) + amplitude * sin(term9b)) * (pulses_wave2 *
filtRRC_coef_main[k % pulse_length]);
            term10c = (amplitude * cos(term8c) + amplitude * sin(term8c)) * (pulses_wave1 *
filtRRC_coef_pre[k % pulse_length]);
            term11c = (amplitude * cos(term9c) + amplitude * sin(term9c)) * (pulses_wave2 *
filtRRC_coef_pre[k % pulse_length]);
            term10a = (amplitude * cos(term8a) + amplitude * sin(term8a)) * (pulses_wave1 *
filtRRC_coef_post[k % pulse_length]);
            term11a = (amplitude * cos(term9a) + amplitude * sin(term9a)) * (pulses_wave2 *
filtRRC_coef_post[k % pulse_length]);
          }
          else if(intra_pulse_c == 1){
            term10c = (amplitude * cos(term8c) + amplitude * sin(term8c)) * (pulses_wave1 *
filtRRC_coef_main[k % pulse_length]);
            term11c = (amplitude * cos(term9c) + amplitude * sin(term9c)) * (pulses_wave2 *
filtRRC_coef_main[k % pulse_length]);
            term10a = (amplitude * cos(term8a) + amplitude * sin(term8a)) * (pulses_wave1 *
filtRRC_coef_pre[k % pulse_length]);
            term11a = (amplitude * cos(term9a) + amplitude * sin(term9a)) * (pulses_wave2 *
filtRRC_coef_pre[k % pulse_length]);
            term10b = (amplitude * cos(term8b) + amplitude * sin(term8b)) * (pulses_wave1 *
filtRRC_coef_post[k % pulse_length]);
            term11b = (amplitude * cos(term9b) + amplitude * sin(term9b)) * (pulses_wave2 *
filtRRC_coef_post[k % pulse_length]);
          }

          modulated_wave_signal[(i * symbol_block_size) + k] = term10a + term11a + term10b +
term11b + term10c + term11c;
        }
    }
```

Appendix 5: LB28 Base Modes

Diagram 7. LB28 Base Modes Comparison – by Mode


LB28 Modulation Over AWGN Channel
Eb / N0 (dB)
Bit Error Rate
LB28-3200-32-2-15-I
LB28-32-2-10-I
LB28-6400-64-2-15-I
LB28-128-2-15-I
LB28-256-2-15-I
LB28-25600-256-2-15-I
LB28-128-2-10-I
LB28-64-2-10-I
LB28-256-2-10-I
LB28-16-2-10-I
LB28-320-8-2-50-N
LB28-160-4-2-50-N
LB28-160-2-2-100-N
LB28-512-2-15-I
LB28-1024-2-15-I
LB28-51200-1024-2-15-I
LB28-102400-2048-2-15-I
LB28-102400-1024-2-15-I
LB28-25600-512-2-15-I
LB28-6400-128-2-15-I
LB28-204800-2048-2-15-I
LB28-2-2-100-N
LB28-240-2-2-100-N
LB28-160-4-2-100-N
LB28-4-2-40-N
LB28-4-2-20-N
LB28-8-2-10-N
LB28-64-2-15-I
LB28-16-2-15-I