WOLF Coding Process

Radix40 Coding

Start with a 15 character message containing only upper case letters, numbers 0-9 or the characters "." "/" or [space]

Take the first three characters of the message from left to right, numbering them 1 to 3. Code each character into a number (n) from 0 to 39 as :

Letters A to Z = 1 to 26 (subtract 64 from the ASCII value of upper case)
Numbers 0 to 9 = 27 to 36 (subtract 21 from ASCII value)

[space] = 0
. [dot] = 37
/ [slash] = 38
All other characters = 39

Take the three numbers n1, n2, n3 and form a 16 bit value :
V0 = n1 * 1600 + n2 * 40 + n3

Repeat for each group of three, working from the left of the input message to give five 16 bit numbers V0, V1, V2, V3, V4 stacked in this order to form an 80 bit sequence V

Convolutional Coding

Preload a 16 bit shift register (SR) with the final value, V4 (Tail biting)

Shift SR left by 1 bit
Shift the 80 bits of V left, moving the MS bit shifted out into the LSB of the shift register [Note 1]

In the order given, **AND** the contents of the shift register successively with the six convolution codes : 042631, 047245, 073363, 056507, 077267, 064537 [expressed in octal]

Calculate the single bit parity (B) for each result of the six **AND** operations,
For each shift of the 80 source bits, 6 parity bits are therefore generated in sequence, giving 480 bits out. These are placed into an output buffer at positions defined by the interleaver.

Interleaving

The locations of each convolver output bit are generated by making use of the three counters used in the convolution process:

The count (0 to 4) of the five 16-bit V values containing the radix40 source data;
The bit-count for each of those (0 – 15) *working from left to right*
The convolution count (0 to 5) in the order given above.

The counters are referred to, here, respectively as i_{VAL} , j_{BIT} and k_{CONV}. (the letters i, j, k are derived from the names of their variables used in the original Wolf source code; the subscripts have been added for clarity.

The process of shifting the 80 bits, in practice involves an outer loop counting the five values $\mathbf{i_{VAL}}$ a middle loop counting the 16 bits $\mathbf{j_{BIT}}$ and the innermost loop counting the convolution calculation $\mathbf{k_{CONV}}$

At each step of the above sequence, as the 480 output bits are generated, a pointer to an output array (outdata) is formed from :

Pointer = $\mathbf{k_{CONV}}$ * 80 + ($\mathbf{j_{BIT}}$ **&** 7) * 10 + ($\mathbf{j_{BIT}}$ >> 8) + $\mathbf{i_{VAL}}$ * 2 or
Pointer = $\mathbf{k_{CONV}}$ * 80 + ($\mathbf{j_{BIT}}$ MOD 8) * 10 + ($\mathbf{j_{BIT}}$ \ 8) + $\mathbf{i_{VAL}}$ * 2

Each of the sequentially generated 480 parity bits, B_N, are placed in the array pointed to by the Pointer calculation:

Outdata(pointer) = B_N

Note that the interleaving process only mixes up bits over a single convolution code. So each output bit of the six innermost convolution calculations will be "spread out" over a contiguous space of 80 bits. The resulting six blocks of 80 bits sit end-to-end, meaning the entire contents of the first convolution over the whole message are sent first, followed by the whole second convolution and so on. This is the means that allows strong signals to be decoded before a complete transmission sequence has been received.

The piecewise interleaving will also simplify the encoding process in small microcontrollers with limited working memory.

The result is an array of 480 bits, consisting of six groups of 80 'mixed up' parity results from each calculation in turn

Merge with Sync Vector

Bits are taken alternately, starting with B0 of the *Outdata* array, followed by the first bit of the sync vector (shown in the listing below), then B1 of *Outdata*, then the next bit of the sync vector, B2, sync, B3, sync….. and so on.

The result is 960 bits (symbols) that form the final PSK modulation.

Notes

Note 1 The original C Source code for Wolf, and my PowerBasic version of that both use a SHIFT RIGHT command on the Radix40 coded values which confuses understanding of the encoding process :

In C code this is : in[i] >> (15-j)
In Basic SHIFT RIGHT radix40(i), (15-j)

This is, in reality, equivalent to a shift left, extracting the MSB of the 16 bit value at each iteration. The Pascal source code mentions this but the whole process there is complicated by having to use signed variables:

item:=item+item; isr:=(isr+isr) and $7fff; {shift our 15-bit shiftreg left 1 pl}

Sync Pattern

0,0,1,1,1,1,1,0,1,0,1,0,1,1,0,0,0,0,1,1,0,1,0,1,1,0,0,1,0,1,0,0,1,0,0,0,1,0,0,1,
1,0,1,0,0,0,0,0,1,1,1,0,1,0,1,0,1,1,1,0,0,0,1,0,1,1,0,0,0,1,0,1,1,1,1,1,0,1,0,1,1,
1,1,0,0,1,0,0,0,1,0,0,1,1,1,0,1,0,1,0,0,0,1,0,1,1,0,0,0,1,0,1,0,1,1,0,1,1,0,1,0,
1,0,1,1,0,0,0,0,0,1,0,1,1,0,1,1,0,1,0,0,0,1,0,1,1,1,0,0,0,1,1,1,1,1,0,1,1,1,0,0,1,
1,0,1,1,0,1,0,0,1,1,1,0,1,0,0,1,1,0,1,0,1,0,0,1,0,0,0,1,1,1,0,1,0,0,1,0,0,0,1,1,
0,0,1,0,1,0,1,1,1,1,0,1,0,1,0,0,0,0,0,0,0,0,0,0,1,0,1,1,1,1,1,1,1,0,1,1,0,0,1,1,1,
1,1,0,1,0,0,1,0,1,0,1,0,1,1,0,1,0,0,1,0,1,0,0,0,1,0,1,0,1,1,0,0,0,0,0,0,0,0,0,0,
1,1,1,1,1,1,0,1,0,1,1,0,0,1,0,0,1,1,0,1,0,1,1,0,1,1,1,0,0,1,0,1,0,1,0,1,1,1,1,0,
1,1,0,0,1,0,0,0,0,0,0,1,1,1,1,1,1,0,0,1,1,0,0,0,1,1,1,1,1,1,0,1,0,0,0,0,0,0,1,0,0,1,
0,1,0,0,0,1,1,0,0,1,0,1,0,1,0,0,1,1,0,1,1,1,0,0,1,1,1,0,0,1,1,1,1,1,1,0,1,1,0,1,0,
1,0,0,1,1,1,0,0,0,0,0,1,0,0,1,0,1,0,0,0,0,0,1,1,0,0,0,0,0,1,1,0,1,0,1,1,1,0,1,0,1,1,
1,0,0,0,1,1,1,1,1,1,0,0,0,1,1,0,1,0,0,1,1,1,1,1,1,1,0,1,0,0,1,0,0,1,0,1,0,1,0,0,0,1,1,1

Or shown grouped into bytes :

b'00111110', b'10101100', b'00110101', b'10010100', b'10001001
b'10100000', b'11101010', b'11100010', b'11000101', b'11101011'
b'11001000', b'10011101', b'01000101', b'10001010', b'11011010'
b'10110000', b'01011011', b'01000101', b'11000111', b'10111001'
b'10110100', b'11101001', b'10101001', b'00011101', b'00100011'
b'00101011', b'11010100', b'00000000', b'10111111', b'01100111'
b'11010010', b'10101101', b'00101000', b'10101100', b'00000000'
b'11111101', b'01100100', b'11010110', b'11100101', b'01011110'
b'11001000', b'00011111', b'00110001', b'11110100', b'00001001'
b'01000110', b'01010100', b'11011100', b'11100111', b'11011010'
b'10011100', b'00010010', b'10000110', b'00011010', b'11101011'
b'10001111', b'10001101', b'00111111', b'01001001', b'01000111'

Andy Talbot G4JNT March 2016