The JT4 Coding Process (Plain Text messages only)

This note describes my understanding of the JT4 coding process for Plain-Text messages. In the same way as for WSPR, I wanted a stand alone programme to generate the symbols for use in a PIC + DDS beacon generator for the mode, and just **had** to understand the coding process before being happy using it.

GENJT4.EXE and its associated source code ***GENJT4.BAS*** (16 bit PowerBasic) will generate a file *JT4SYMBS.INC* of compressed symbols for direct import into a PIC microcontroller assembly listing.

Source Coding :

For plain text messages input data can consist only of :

(1) 10 Numbers 0 - 9
(2) 26 Upper case letters A-Z
(3) 6 Punctuation symbols [space] + - . / ?

Giving a total of 42 possible characters.

A JT4 message consists of up to13 of these characters [char 1] through to [char 13].
So, for example, a callsign and locator can be accommodated eg. ‘ G4JNT IO90IV’
(nb. preceding space in this example) Shorter messages are padded with spaces.

The message is encoded by stepping through the data and generating a value from 0 to 41 for each character in the order they are listed above, ie ‘0’ = 0, ‘9’ = 9, ‘A’ = 10, ‘Z’ = 35, ‘?’ = 41.

Three long integers are now formed as follows :

```
N1 = [char1]                   N2 = [char6]
N1 = N1 * 42 + [char 2]        N2 = N2 * 42 + [char 7]
N1 = N1 * 42 + [char 3]        N2 = N2 * 42 + [char 8]
N1 = N1 * 42 + [char 4]        N2 = N2 * 42 + [char 9]
N1 = N1 * 42 + [char 5]        N2 = N2 * 42 + [char 10]

N3 = [char11]
N3 = N3 * 42 + [char 12]
N3 = N3 * 42 + [char 13]
```

N1 and N2 have a maximum value of $42^5 - 1 = 130691231$ which fits nicely into a 27 bit integer (maximum value 134217727). N3 is a 17 bit integer, having a maximum value of $42^3 = 74087$

So a total of 27 + 27 + 17 = 71 bits are needed; A single bit is added as a flag to indicate the message is plain text and gives a total of 72 bits of source data. The words are compressed and slightly rearranged into a contiguous 72 bit sequence

```
N1 = N1 * 2 + [bit 15] of N3   (or N3 \ 32768 MOD 2)   (28 bits total)
N2 = N2 * 2 + [bit 16] of N3   (or N3 \ 65536 MOD 2)   (28 bits total)
```

N3 = N3 MOD 32768 + 32768 (16 bits total)

Adding 32768 or setting bit 15 of N3 is the plain text flag

The contiguous 72 bit sequence is made up of N1 * $2^{(16 + 28)}$ + N2 * 2^{16} + N3
In GENJT4.BAS *the data is represented in a string format from this point on*.

31 zero bits are added at the end to make a 103 bit sequence for the next stage.

Convolutional Encoding

The data is now expanded to add FEC with a rate ½, constraint length 32, convolutional encoder.

The 103 bits (including trailing zeros) are read out MSB first:
(using the string representation, one-at-a-time from the left hand end)

The bits are clocked simultaneously into the right hand side, or least significant position, of two 32 bit shift registers [Reg 0] and [Reg 1]. Each shift register feeds an Exclusive-OR parity generator from feedback taps described respectively by the 32 bit values 0xF2D05351 and 0xE4613C47. Parity generation starts immediately the first bit appears in the registers (which must be initially cleared) and continues until the registers are flushed by the final 31st zero being clocked into them.

Each of the 103 bits shifted in generates a parity bit from each of the generators , a total of 206 bits in all. For each bit shifted in, the resulting two parity bits are taken in turn, in the order the two feedback tap positions values are given, to give a stream of 206 output bits.

The parity generation process is :

```
    Shift the next source bit into the LSB of both [Reg 0] and [Reg 1],
        moving the existing data in each one place left
    Take the contents of [Reg 0]
    AND with 0xF2D05351
    Calculate the single bit parity (XOR) of the resulting sum.
    Append to the output data stream
    Take the contents of [Reg 1]
    AND with 0xE4613C47
    Calculate the single bit parity (XOR) of the resulting sum.
    Append to the output data stream
```

The expansion from 72 source data bits to 206 has added sufficient redundancy in an optimised manner to give a code capable of very strong Forward Error Correction against random errors.

Interleaving

Errors over a radio link are rarely random , being more likely to occur in bursts against which this sort of convolutional coding is less effective,. So the final stage of encoding is to mix up, or interleave. the 206 data bits so as to move adjacent bits away from each other in time. The result is that close-together bits corrupted by burst interference are spread throughout the frame and therefore appear as random errors – which the FEC process can cope with.

The interleaving process is performed by taking the block of 206 starting bits labelled S[0] to S[205] and using a bit reversal of the address to reorder them, to give a pattern of destination bits referred to as D[0] to D[205].

```
Initialise a counter, P to zero
Take each 8-bit address from 0 to 255,   referred to here as  I
Bit-reverse I to give a value J.
    For example,  I = 1 gives J = 128,  I = 13  J = 176  etc.

If the resulting bit-reversed J yields a value less than 206 then :
    Set Destination bit  D[J] =  source bit  S[P]
    Increment  P
Stop when  P = 206
```

This completely shuffles and reorders the 206 bits on a one-to-one basis.

A single zero bit is now appended to the start of the start of the interleaved sequence. This serves as a reference tone if phase encoding is adopted (JT2 mode)

Merge With Sync Vector

The 207 bits of data are now merged with 207 bits of a pseudo random synchronisation word having good auto-correlation properties. Each source bit is combined with a sync bit taken in turn from the table below to give a four-state symbol value:

Symbol[n] = Sync[n] + 2 * Data[n]

```
207 bit Synchronisation  Vector
    0,0,0,0,1,1,0,0,0,1,1,0,1,1,0,0,1,0,1,0,0,0,0,0,0,0,1,1,0,0,0,0,0,0,0,0,0,0,0,0
    1,0,1,1,0,1,1,0,1,0,1,1,1,1,1,0,1,0,0,0,1,0,0,1,0,0,1,1,1,1,1,0,0,0,1,0,1,0,0,0
    1,1,1,1,0,1,1,0,0,1,0,0,0,1,1,0,1,0,1,0,1,0,1,0,1,1,1,1,1,0,1,0,1,0,1,1,0,1,0,1
    0,1,1,1,0,0,1,0,1,1,0,1,1,1,1,0,0,0,0,1,1,0,1,1,0,0,0,1,1,1,0,1,1,1,0,1,1,1,0,0
    1,0,0,0,1,1,0,1,1,0,0,1,0,0,0,1,1,1,1,1,1,0,0,1,1,0,0,0,0,1,1,0,0,0,1,0,1,1,0,1
    1,1,1,0,1,0,1
```

Resulting in 207 sequential symbols each with a value from 0 to 3

Modulation

Each symbol represents a frequency shift of 11025 / 2520, or 4.375Hz, per Symbol Value, multiplied by a constant K depending on the JT4a to JT4g mode and leading to a four-level Multi-FSK modulation. The transmitted symbol length is the reciprocal of the tone spacing, or approximately 0.229, so the complete message of 207 symbols takes around 47 seconds to send and occupies a bandwidth of approximately (3.K.+ 1)* 4.375 Hz (three spacing intervals plus the keying bandwidth).

```
Tone spacing
multipliers

  JT4a K = 1
  JT4b K = 2
  JT4c K = 4
  JT4d K = 9
  JT4e K = 18
  JT4f K = 36
  JT4g K = 72
```

Conventionally, and presumably for compatibility with other WSJT modes, the centre frequency of the transmission is defined as 11025Hz / 1024 * 118 = 1270.46Hz which lies at tone position 1.5 (midway between tones 1 and 2) So the audio frequency corresponding to Tone Zero is now :
11025/1024 * 118 – 1.5 * 4.375 * K

As a sanity-check, for JT4g, with K = 72 Tone zero lies at 797.959Hz (800 Hz is near enough!) and the tone spacing is 315Hz for a total signal bandwidth a few Hz over 945Hz

Packing for export and storage

For export, the 207 two-bit symbols are packed four to a byte, MSB first, into 52 locations with appropriate formatting for PIC assembly code.

```
;  JT4 Symbols  generated from GENJT4     G4JNT Jul 2009
    ;  Message data ' G4JNT IO90IV'

    de  0x20 0xDA 0x3E 0x50 0xCC 0xAA 0x2D 0x20
    de  0x00 0x82 0x65 0x34 0xC5 0xD4 0x4A 0xE1
    de  0x25 0xF4 0x06 0xC0 0x75 0x96 0x18 0x14
    de  0x6C 0xEE 0x55 0xC4 0xC7 0xBB 0x37 0x86
    de  0xF3 0xF4 0xA3 0x45 0x29 0xD9 0xD9 0xF2
    de  0x40 0xF1 0x63 0x03 0x5F 0xCB 0x48 0x16
    de  0x8C 0x71 0x54 0xCC
```

Andy Talbot G4JNT July 2009