The Q65 Coding Process used in WSJT-X.

Andy Talbot G4JNT October 2023

Introduction

The later versions of the WSJT-X suite introduced the Q65 Mode based on Q-ary Repeat Accumulation coding, originally proposed by Nico Palermo IV3NWV [1] , to generate the parity bits for Forward Error Correction. In the same way as done for the earlier modes, an in-depth study of the source code was made to gain an understanding of how the on-air symbols are generated from a user message [2] - [5]. Routines were then rewritten in an alternative programming language (PowerBasic) to check they all worked properly, giving the same result as the WSJT-X utility programme *Q65CODE.EXE*.

Only certain message types are considered in this study, a free text message payload of up to 13 characters and 'Telemetry Mode' for up to $17^3/_4$ hexadecimal characters. The latter has been introduced to give users direct access to the raw 71 data bits of the payload. Other message types are described in the WSJTX manual. The payload is made up of 77 bits with six of these reserved for message type. The two message types covered here are those most likely to be needed by users who wish to implement stand-alone transmitter sources such as beacons and for remote monitoring or telemetry systems.

Source Coding

Q65 uses the same source coding as FST4, FT8, FT4 and MSK144 to convert a user message into 71 bits of payload data. The compression technique for the free text mode is slightly easier to visualise than that for the slightly shorter message in the earlier modes like JT4 and JT9 but cannot be done using 32 bit, or even 64 bit integers in most high level languages. Nor can telemetry mode be generated using these standard numeric variable types. If a number has to be visualised at all, it needs to be a variable that can hold a 71 bit value or a bit array; special routines need to be written to work with this. In practice it is easier to work with a string or array of individual '1' or '0' bits and write routines to perform basic arithmetic on them a byte at a time.

Free text - Taking the 13 characters of the message from the left hand side, each character is converted to a value from 0 to 41 based on its position in the allowed alphabet.
" *0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ+-./?*". The first is [space] with a value of 0, '0' =1, '1'=2 …, 'A'=11 etc. up to '?' = 41. The 71 bit value is built up by progressively adding and multiplying by 42. The maximum possible value is therefore 42^{13}, which is less than 2^{71}. Taking this as a string of 71 bits, six more bits "*000000*" are appended to the RHS to label it a free text message. The resulting 77 bits go into the next part of the encoding process.

Telemetry Mode – A message expressed by up to 18 hexadecimal characters such as "5657A7EDEADBEEF123" or "0123456789CAFEEF01" converts directly to the 71 bits of the payload. Note that the left hand hex character can only take on a value of 0 to 7 to ensure the most significant bit is never used, so as to maintain just 71 bits. A hex string shorter than 18 characters is assumed to have leading zeroes to make up 18 hex digits. Six more bits "*101000*" are added to the RHS to make up the 77 bit source data.

A further "*0*" is appended to make the number of bits a multiple of 6 and the message is split up into 13 6 bit values, message symbols, each taking on a value of 0 – 63. The left-most block of 6 bits form the first message symbol, moving to the right 6 bits at a time to reach the final 13^{th} symbol.

Cyclic Redundancy Check with bit-flipping

A 12 bit CRC is added to allow the decoder software to reject all false decodes that can arise from chance events and incorrect application of error correction. Unlike all the other WSJT-X modes, Q65 has a peculiarity in defining the input to the CRC algorithm. For compatibility with Nico's original work on QRA codes, each input symbol or block of six bits is flipped end-to-end, or reversed, so bit '0' becomes bit '5', bit '4' becomes bit '1' etc. This is done to each one of the 13 input symbols / blocks of six bits. The result now has to be stored in a separate buffer purely for CRC computation.

The CRC is 12 bits long (two symbols) and is computed by shifting the 78 bit reordered sequence left one bit at a time. Each bit that pops out of the shift is then left-shifted through a 13 bit register R. If the bit appearing at the left hand side of R (actually its left most or 13^{th} bit) is a '1' the contents of R are XORed with the CRC polynomial '1100000001111'. If the 13^{th} bit is a '0' the value in R remains unchanged.

After all 78 input bits have been through the process, the 12 LSBs, or two symbols remaining in R give the CRC. The pair of six bit symbols are each bit-flipped to get the final value used in transmission.

The 12 reflipped bits in R forming the modified CRC are appended to the end of the original unflipped message to give a 90-bit sequence that goes into the parity generating process.

As an alternative to making a separate buffer for bit-flipped input data, each separate 6 bit symbol can be taken from the main store in the same order described, but each of the six bits is fed into the CRC process by RIGHT SHIFTING. This has the effect of doing the bit reversal as an integral part of the CRC generation process so does not require a separate buffer. Note that it is still necessary to bit-flip the two six bit symbols making up the CRC before transferring these to the end of the message symbol set.

Parity Bits

If described in mathematical terms, calculating the parity bits can appear horrendously complicated, involving multiplication and addition of values treated as polynomials in a Galois field GF(64) which, unless you have an intimate knowledge and understanding of how to do this, is utterly meaningless. However, ignoring the details of the GF maths and noting everything is done on six-bit symbols leads to a simple algorithmic approach. GF(64) addition turns out to be just a quite straightforward and simple XOR of the two six-bit values. No carry is involved as would be the case for 'normal' addition.

Multiplication is more involved, and is done 'using logarithms'. Two 64-entry precalculated tables have been generated. They can be found in the Fortran source code, in the file '*Q65_encoding_modules.F90*'. One table is called 'GF64Log' and consists of the GF(64) arithmetic version of the logarithm of each of the input values 1 – 63. Log of zero is impossible and this is treated separately, as is an input of 1 for either of the two input values since 1 multiplied by anything, even in GF maths, yields the same value. A table lookup for each of the two input values is made and the results added modulo (64). The resulting sum is then used as lookup into another table called 'GF64antilog' which does what its name implies. The result is the GF(64) product of the two input numbers since adding logs then taking the antilog is equivalent to multiplying the originals.

Parity Bits are generated by taking reordered or permuted input symbols, several times for each (the Q-ary name of the process). The 15 message symbols (13 data plus two parity) are permuted and repeated for an effective collection of 50 interim output symbols. Each of these is multiplied by a weighting value.

The permutation and weighting process is transparent, being performed in a two dimensional table called '*generator*' also to be found in the same .F90 source code file. The table matrix has 15 columns

corresponding to each of the 15 input symbols, referenced here by the 'i' variable, and 50 rows, referenced here by 'j'. Each input symbol is taken in turn and used as one coordinate for the table lookup. The other coordinate is a count from 1 to 50, labelled 'j' here. This is repeated for each input symbol.

A 50-long array of parity symbols pointed to by 'j' referred to as *'codeword'* is first initialised to zero. For each count of 'j' the two-dimensional *generator* table is accessed with the value (i , j). The value looked up is multiplied GF(64) by the associated message symbol msg(i) and added GF(64) to *codeword*(j). Thus each of the 50 values of the *codeword* are progressively built up via terms from the '*generator'* matrix that each include different weighted combinations of each of the source bits. It needs to be noted that many terms of the generator matrix are zero, so only certain combinations of input symbols influence a particular *codeword* symbol.

The 50 *codeword* symbols are appended to the end of the 15 message symbols to give, initially, 65 channel symbols. For convenience, a 65 long '*codeword*' array is used to hold both source and destination. The first 15 entries are set to be the input message, then the remaining 50 higher entries are pointed to by the 'j' counter with an offset of 15.

Done this way, the parity generation process can be seen in this **pseudo-code**. The GF(64) routines have been highlighted and renamed GF64+ and GF64* to indicate their similarity in use to normal mathematical operators. GF64+ could have simply been replaced by an XOR operation, but to keep faith with the original Fortran source code is shown this way.

```
codeword(1 .. 15)  =   message(1 .. 15)
For i = 1 to 15              'Count the input message symbols
   For j = 16 to 65          'jcount + 15 to allow one single input and output array
     codeword(j) = codeword(j) GF64+ (codeword(i) GF64* generator(i , j-15) )
   Next j
Next i
```

Puncturing

Of the resulting 65 symbols now in '*codeword*' the pair carrying the CRC at locations 14 and 15 are removed or punctured to give 63 information carrying symbols.

Synchronisation and Tones

The 63 information symbols are now merged with a 22 symbol synchronisation vector. An 85 symbol output array is initialised, and a value of 0 (corresponding to the lowest tone) inserted at locations 1, 9, 12, 13, 15, 22, 23, 26, 27, 33, 35, 38, 46, 50, 55, 60, 62, 66, 69, 74, 76, 85. This is the pseudo-random synchronisation vector.

The remaining empty locations 2 .. 8, 10, 11, 14, 16 .. 21, 24 … etc. are successively filled with the message symbols plus one to give a total of 65 different tone values. 0 for sync, 1 to 64 for information.

Depending on the speed and tone-spacing variant of Q65 in use, the tone number is multiplied by the tone-spacing and added to a baseline, or tone-zero frequency to give the final output tone for each symbol.

The utility Q65CODE.EXE included within the WSJT-X suite shows the output from each of these stages (except bit flipping in the CRC) and can be used to check progress when writing your own code. An example for the free-text message '*g4jnt testing*' is shown in the box below.

```
C:\WSJT\wsjtx\bin>q65code "g4jnt testing".

 Generated message plus CRC (90 bits)
6 bit :   13  63  22  63  36   8   6  57  56  24  38  26   0  47  38
binary: 001101111111010110111111100100001000000110111
        001111000011000100110011010000000101111100110

 Codeword with CRC symbols (65 symbols)
 13 63 22 63 36  8  6 57 56 24 38 26  0 47 38 47 55  8 44 22
 22 14 35 19 23  3 58 29 33 61 55 55 15 51 21 11  3 28 40 40
 60 34 59  4 30  8  4 34 46 40 51 33 33  6 15 17 28 46 30 43
 32 24 25 26 36

 Channel symbols (85 total)
  0 14 64 23 64 37  9  7  0 58 57  0  0 25  0 39 27  1 48 56
  9  0  0 45 23  0  0 23 15 36 20 24  0  4  0 59 30  0 34 62
 56 56 16 52 22  0 12  4 29  0 41 41 61 35  0 60  5 31  9  0
  5  0 35 47 41  0 52 34  0 34  7 16 18  0 29  0 47 31 44 33
 25 26 27 37  0
```

Output from *Q65CODE.EXE*

.

References

[1] **http://www.eme2016.org/wp-content/uploads/2016/08/EME-2016-IV3NWV-Presentation.pdf**

[2] http://g4jnt.com/WSPR_Coding_Process.pdf

[3] http://g4jnt.com/JT4_Coding_Process.pdf

[4] http://g4jnt.com/JT9_Coding_Process.pdf

[5] http://g4jnt.com/WSJT-X_LdpcModesCodingProcess.pdf

[6] WSJT-X Source Code can be found in a .TGZ archive which can be found by looking at the bottom of the WSJT-X Home page https://wsjt.sourceforge.io/wsjtx.html