The Coding Process for the FST4, FTS4W, FT8, FT4 and MSK144 Modes used in WSJT-X.

Andy Talbot G4JNT December 2021

Introduction

The later versions of the WSJT-X weak signal data modes suite introduced several new modes based on a new method of Forward Error Correction; Low Density Parity Checking or LDPC codes. In the same way as was done for the earlier modes [1] - [3], I studied the source code in depth to gain an understanding of how the on-air symbols are generated from a user message. In each case, routines were written in an alternative programming language (PowerBasic) to check I had implemented all the routines properly, by its giving the same results as the various WSJT-X utility programmes.

As for the earlier modes, only certain message types are considered for the communication modes FST4, FT8, FT4 and MSK144. A similar free text message payload of up to 13 characters is similar to that in the earlier modes, and there is also a new 'Telemetry Mode'. The latter has been introduced to give users direct access to the raw 71 data bits of the payload. There are actually 77 bits in the payload, but six of these are reserved for message type. These two message types are those most likely to be needed by users who wish to implement stand-alone transmitter sources like beacons and remote monitoring or telemetry systems.

Similarities and Differences

The modes covered here all have basic similarities in their coding structure, as well as some subtle differences, so all of them will be covered in this same document with the differences highlighted as needed. While FST4W has a lot in common with FST4 resulting in a nearly identical final symbol set, like WSPR it carries a shorter and very different type of message payload.

Source Coding

FST4, FT8, FT4 and MSK144 all use an identical means 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 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 have to be written to work with this. In practice it is easier to work with as a string 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 – In the WSJT-X software 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 not 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 of the 71 bits to make up the 77 bit source data**.**

FST4W Compression – This is a special encoding just used for iWSPR type messages, and contains callsign, four-character locator, and power level in dBm. Basic callsign encoding is performed in a similar but not identical manner to that for WSPR, [1] building up a 28 bit value based on the allowed letters or numbers in any position of a standard callsign.

A callsign has a maximum of six characters consisting only of A-Z, 0-9 and [space] The *third* character is forced to be *always* a number. To cope with callsigns that start with a letter followed by a number, a space is appended to the front if necessary. So, for example, 'G4JNT' will become '[space]G4JNT' whereas GD4JNT stays as-is. Short callsigns are then further padded out to six characters by appending spaces to the end

The 37 allowed characters are allocated values from 0 to 36 such that '0' – '9' give 0 – 9, 'A' to 'Z' give 10 to 35 and [space] is given the value 36. Further coding rules on callsigns mean that the final three characters (of the now padded out callsign) can only be letters or [space] so will only take the values 10 – 36. With the characters, designated [Ch *X*], taking on values from 0 to 36 as defined, the callsign is now compressed into a single integer **N** by successively building up.

N_1 = [Ch 1]	The first character can take on any of the 37 values including [sp],
$N_2 = N_1 * 36 + [Ch\ 2]$	but the second character cannot then be a space so can have 36 values
$N_3 = N_2 * 10 + [Ch\ 3]$	The third character must always be a number, so only 10 values are possible.
$N_4 = 27 * N_3 + [Ch\ 4] - 10]$	
$N_5 = 27 * N_4 + [Ch\ 5] - 10]$	Characters at the end cannot be numbers,
$N_6 = 27 * N_5 + [Ch\ 6] - 10]$	so only 27 values are possible.

Giving an absolute maximum value for **N** of 37 * 36 * 10 * 27 * 27 * 27 = 262177560

Which is just comfortably less than 2^{28} = 268435456 and means the callsign can be represented by 28 bits with a range of codes left over for the future, eg. For allocation to special cases and flags.

In a change from WSPR encoding, 2063592 + 2^{22} is now added to the result as a flag to show it is a standard callsign, but the result is still a 28 bit number. (Adding this offset takes the maximum possible value up to the limit for 28 bits). Doing this means there is now provision for special types of callsigns, and extensions but these are not discussed here. Visit the WSJT-X source code for full details of the enhanced callsign encoding.

The locator, characters CH1 to CH4, is compressed to 15 bits but in a way that is simpler than used for WSPR, just encoding the first pair of letters A – R directly as numbers from 0 to 17, then progressively building up from the left:

N = CH1 * 18 * 10 * 10 + CH2 * 10 * 10 + CH3 * 10 + CH4

The power level is encoded as a five bit value 0 – 63, giving units from -30dBm to +30dBm

The resulting 48 bit number, (28 bits of callsign, 15 of locator and 5 of power) has the two bits "00" appended to make up a 50 bit source for the FST4W standard '*iWSPR*' mode.

Scrambling

For the FST4 and FT4 Modes only, the 77 source bits are scrambled by bitwise exclusive ORing with the scrambling vector :
"*0100101001011110100010011011010010110000100010100111100101010101101111100010 1*".

FT8, MSK144 and FST4W are not scrambled and the raw 77 bits pass on to the next stage of processing.

Cyclic Redundancy Check

Each of the modes has a CRC added to allow the decoder software to reject all false decodes that can arise from noise and incorrect application of error correction. The CRC is of a different length and has a different generating polynomial for each of the three main mode types. CRC generation is subtly different between that for FST4 / FST4W , FT4 / FT8 and MSK144

FST4 / FST4W CRC – this is 24 bits long and is generated as follows. 24 '0's are appended to the end of the 77 bit (for FST4) or 50 bit (for FST4W) source data making a bit pattern 101 or 74 bits long. The bits of the source data are then left shifted, one bit at a time, through a 24 bit shift register that is initialised to all zeroes at the start. At each stage of the shift, the 'lost' bit that has just popped out from the left hand side of the shift register is tested. If it is '0' the contents of the shift register remain unaltered. If it is a '1' the 24 bit shift register contents are XORed with the generator polynomial '*100000000000000011001011011*' or 0x100065B. (Note that the left hand, most significant bit of the polynomial is always 1 in a CRC generator and is ignored here – only the 24 right-most bits take part in the XOR process). After the source bits have all been shifted in, including the 24 '0's that were appended at the end, the 24 bits remaining in the shift register form the CRC.

This CRC is then appended to the end of the *original* source bits to form a 101 or 74 bit data-set for the next stage.

FT8 and FT4 CRC – is 14 bit with the generator polynomial 0x6757 or '*110011101010111*' (again, the left-most bit is always '1' and not used). 14 '0's are appended to the 77 bit source data, in the same was as described above, but there is another quirk in this (and for MSK144) CRC generation. Further '0's are now added so the resulting source plus extra '0's is padded out to have a length that is a multiple of 8 bits. For the 77+14 bits here, that requires '00000' to be appended making a 96 bit pattern. The 96 bits are left-shifted into the 14 bit shift register, and if the bit just popped out at each shift is a '1', the contents are XORed with the polynomial.

The 14 bits left in the shift register after the final shift-left form the CRC which is appended to the original source data to give a data set 91 bits in length

MSK144 CRC – is 13 bit and has to have six more '0's appended on top of the 13, to make a multiple of 8 bits, again resulting in 96 in total. The generator polynomial is 0x15D7 or '*1010111010111*' (the leading '1' has been left out this time, but this is the value as quoted in the source code).

The 13 bits left in the shift register after the final shift-left form the CRC which is appended to the original source data to give a data set of 90 bits in length.

Parity Bits

All five modes are now expanded by adding parity bits formed by multiplying (ANDing) the entire source + CRC with successive rows of a generator matrix, then taking the single bit parity of the result after each operation. The width of the generator matrix is equal to the length of the input data (77 or 50),

and the depth (number of rows) is equal to the number of parity bits to be generated. These parity matrices can be found in the WSJT-X source documentation, with file names like *ldpc_240_74_generator.f90*.

This one is for FST4W and generates 240 parity bits (240 rows of data) from an input 74 bits wide. The first five rows are :

"de8b3201e3c59f55a14"
"2e06d352ebc5b74c4fc"
"2e16d6cf5a725c3244c"
"84f5587edca6d777de4"
"e152b1e2b5965093ecc"

The hex code for these matrices as listed are left justified. Note how the rightmost hex digit only ever has its two MSBs in use, yielding values of only 0, 4, 8 or 0x0C , since the width is just the 74 bits of the source data + CRC.

The other parity generator matrices are:

FST4 - "*ldpc_240_101_generator.f90*"
FT8/FT4 - "*ldpc_174_91_c_generator.f90*"
MSK144 - "*ldpc_128_90_generator.f90*"

The parity bits for all four modes are generated as follows:

AND the entire source + CRC with the first row of the generator matrix, starting with the left-most bit of the source and the left-most bit of the generator and so on. Form the single parity bit of the result of this multiple AND operation and append this to the source. Repeat for each row of the generator matrix, forming as many parity pits as there are rows in the matrix. This process is equivalent to adding, modulo two, the source+CRC to each line of the parity matrix.

The resulting source data + CRC + parity bits are the FEC encoded final data that will go onto the next stage of symbol generation and modulation. For each of the four modes this data has a structure to it, albeit each of different length, in the form Source data + CRC + parity. So it can be seen that the raw 77 or 50 source bits appear at the start of the final transmission unaltered, apart from the case with FST4 and FT4 where they have been scrambled by the fixed scrambling vector. The overall bit positions are:

Mode	Source		CRC		Parity		
FST4	77	+	24	+	139	=	240 bits
FST4W	50	+	24	+	166	=	240
FT8/FT4	77	+	14	+	83	=	174
MSK144	77	+	13	+	38	=	128

Note the considerably fewer number of parity bits in MSK144 as well as its shorter CRC. This reduction in FEC strength is offset by the fact that an MSK144 message is repeated many times in a frame so a significant part of the error correction is done by overlaying information from successive messages.

Synchronisation and Conversion to Symbols

So far, apart from different lengths of the bit fields and the message data in FST4W, the encoding of each mode has all followed essentially the same process. It is the modulation and addition of synchronisation that makes the three modes very different.

FST4/FST4W – The 240 bits are taken in pairs and coded via a Gray code mapping ‘00’ > 0, ‘01’ > 1, ‘10’ > 3, ‘11’ > 2 to give 120 four-level symbols, 0 – 3.

Two synchronisation Costas arrays of eight symbols each are taken, corresponding to the symbol sequences, *sync1* = "01321023" and *sync2* = "23103201"

The 120 symbols are split up into four blocks, *Block1 – Block4*, of 30 symbols each and merged with the Costas arrays in the following manner to give a 160 symbol sequence. The two distinct Costas arrays are repeated and interspersed throughout the transmitted symbol set:

Sync 1 + Block1 + Sync2 + Block2 + Sync1 + Block3 + Sync2 + Block4 + Sync1

Modulation is by 4-level MFSK with equally spaced frequencies, each corresponding to one of the four symbols. A range of symbol rate and tone spacing options are provided, the submode qualifier is the cycle time of the transmission in seconds. The Tx period is a few seconds less than this

Mode/Duration	Baud Interval, s	Tone Spacing Hz
FST4- 15	0.06	16.67
FST4- 30	0.14	7.14
FST4- 60	0.32	3.09
FST4(W)- 120	0.68	1.46
FST4(W)- 300	1.79	0.558
FST4(W)- 900	5.56	0.18
FST4(W)- 1800	11.2	0.089

FT8 - The 174 bits are taken three at a time and Gray code mapped as:

‘000’ > 0 ‘001’ > 1 ‘011’ > 2 ‘010’ > 3 ‘110’ > 4 ‘100’ > 5 ‘101’ > 6 ‘111’ > 7 To give 58 symbols. A Costas array comprising of seven symbols "3140652" forms a sync vector. The 58 message symbols are spilt into two blocks of 29 and are merged with the sync arrays to form:
Sync + Block1 + Sync + Block2 + Sync to give a 79 symbol pattern.

Modulation is 8-level FSK with a tone spacing of 6.25Hz and a symbol interval of 0.16s

FT4 – adopts the same encoding (albeit with the scrambling of the message data) as for FT8 up to the addition of the parity bits. The 174 bits are taken in pairs and Gray coded mapped as :

‘00’ > 0, ‘01’ > 1, ‘10’ > 3, ‘11’ > 2 to give 87 symbols. This is split into three 29 symbol blocks,

Four different 4x4 Costas arrays are used as sync vectors: Sync1 = ‘0132’ Sync2 =’1023’ Sync3 = ‘2310’ Sync4 = ‘3201’.

The sync vectors are interspersed with the three 29 bit blocks to give 103 symbols

Sync1 + Block1 + Sync2 + Block2 + Sync3 + Block3 + Sync4.

A special amplitude ramped null symbol is inserted at the beginning and end of the transmitted sequence, but this does not affect the coding process

Modulation is 4-level FSK with a tone spacing of 20.833Hz and a symbol interval of 48ms

MSK144 - Is a binary modulation so there is no assembling into multi-level symbols. A sync pattern of "*01110010*" is chosen. The synchronisation is added by splitting the 128 bit message data into two unequal blocks, the one containing the first 48 bits of data and the second block the remaining 80 bits. The over-air stream is then assembled from Sync + Block1 + Sync + Block2, making 144 bits in all

MSK144 uses Minimum Shift Keying, or MSK. There are two way of viewing an MSK waveform, either as offset QPSK with alternate symbols modulating the I or Q data, or as FSK with a frequency shift exactly half the symbol rate. The WSJT software actually includes two methods of generating the resulting waveform. Within the software the 144 bit pattern is used directly, with alternate bits used to form the I and Q channels of the sampled audio waveform to be sent to an SSB transmitter. For direct FSK generation at half the symbol rate this 144 bit pattern needs to be further manipulated to give the same result as the O-QPSK method. This involves differential coding and inverting alternate symbols. For each input bit labelled B1 to B144, the symbols S1 to S144 are generated from:

S1 = B1 XOR B2 S2 = B2 XNOR B3 S3 = B3 XOR B4 ……. S144 = B144 XNOR B1 (note the wrap-around on the final symbol). XNOR is simply XOR with the result inverted. Using the resulting '0' or '1' of the symbol set S1 to S144 as drive to an FSK source generating two frequencies, separated by exactly half the symbol rate to give an MSK waveform.

The WSJT software includes a routine to do this conversion which is used for the *MSK144CODE* utility that generates the binary FSK frequency shift bit pattern.

MSK144 uses a symbol rate of 2000 symbols / second, so the tone shift is 1000Hz which is usually applied to a DDS-based source by setting values giving +/-500Hz of the nominal centre frequency. The burst of 144 symbols takes 72ms to send, with the burst repeated continuously for the duration of the transmission cycle time, usually 5 to 30 seconds.

Gaussian Frequency Shift Keying

While not relevant to the coding process, this is important in generating the FSK symbols when a standalone RF source is being used to generate the transmitted RF. To avoid excessive keying sidebands, the transition from one FSK tone to the next is not an abrupt switch in the way JT4, JT9 and WSPR modes are formed. Instead the change from one symbol to the next follows a Gaussian shaped transition. This means the beginning and end of each symbol contain elements of the previous, current and next pulse. Full details of the Gaussian shaping can be found at [5] and [6]

Summary of Coding Differences

Mode	Source Bits	Scramble	CRC	CRC Padding	Parity Bits	Parity Matrix	Bits / Symbol	Symbols / Sync	Modulation	Rate/Tone Baud ms, Hz
FST4	77	Yes	24	0	166	"…240_101…"	2	2x 8x8 Costas, 5 equi-placed	4-GFSK	Variable
FST4W	50	No	24	0	139	"…240_74…"	2	2x 8x8 Costas, 5 equi-placed	4-GFSK	Variable
FT8	77	No	14	5	83	"…174_91…"	3	1x 7x7 Costas, 3 equi-placed	8-GFSK	160/6.25
FT4	77	Yes	14	5	83	"…174_91…"	2	4x 4x4 Costas, 4 equi-placed	4-GFSK	48/20.83
MSK144	77	No	13	6	38	"…128_90…"	1-Diff +/-Pol.	8bits, x2, at start & 48 bits in	MSK	0.5/1000

References

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

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

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

[4] WSJT-X Source Code can be found in a .TGZ archive file which can be found by looking at the bottom of the WSJT-X Home page https://physics.princeton.edu/pulsar/k1jt/wsjtx.html

[5] The FT4 and FT8 Communication Protocols. QEX July / August 2020 https://physics.princeton.edu/pulsar/k1jt/FT4_FT8_QEX.pdf Contains details of the Gaussian shaping

[6] http://g4jnt.com/FST4_Beacon_Source.pdf