A DDS Based Beacon Source for FST4 and FST4W Modes

Andy Talbot G4JNT March 2021

Overview

When the WSPR weak signal beacon mode first arrived in 2008 I put together a beacon generator that used the symbols from a pre-generated message to drive an AD9852 DDS source to give a WSPR beacon transmission that did not need a host PC and upconversion from audio in an SSB transmitter. The latest version of that beacon source [1] uses a pseudo random sequence to give a range of transmission duty cycles, from 100% down to 20% in randomised timeslots and uses frequency hopping, where each new transmission cycle is in a different randomised part of the 200Hz wide WSPR band.

Subsequently, several other designs for WSPR beacons have appeared on the market doing a similar task and openly published code has allowed many home-brew designs to proliferate. Generating a WSPR waveform is straightforward; an RF carrier is stepped between four spot frequencies spaced 1.46Hz apart at a rate of 1.46 symbols per second, for a transmission period of a little under two minutes. For a period when a 15 minute version of WSPR was in use, the beacon source was modified to transmit this mode. A version of the source even randomly changed between the two modes. WSPR-15 was subsequently dropped and no longer appears in the WSJT software.

With WSJT-X version 2.3 a new mode specifically aimed at low data rate LF was introduced; called FST4 there is also version for beacon type operation similar to WSPR termed FST4W. Both FST4 and FST4W use the same modulation, 160 symbols of 4 tone MFSK, although having different message formats. The major change with FST4 from that of WSPR and other earlier MFSK modes is that the tone-to-tone transition is specially shaped to reduce the bandwidth of the transmission at the symbol intervals so it glides from one tone to the next. This is done to considerably reduce the bandwidth of the waveform at the tone transitions A beacon source that merely switches between four preset tones is no longer suitable for generating this mod. Seven different speeds / bandwidth options were offered, with transmission cycle times ranging from 15 seconds to 30 minutes for FST4; FST4W offering the slowest four of these from ranging from 2 to 30 minutes per cycle.

Gaussian Pulse Shaping

The transition from one tone to the next is shaped to follow a Gaussian curve, and starts before the current symbol has completed, finishing after the next one has started. Gaussian FSK (GFSK) was introduced initially for the FT8 and FT4 data modes targeted at the crowded HF bands where the need to reduce bandwidth was essential. There is an excellent write up on FT4/8 in QEX [2] where the Gaussian pulse shaping is described in considerable detail.

There are several ways to generate a smooth transition from one tone to the next but it was decided to implement what is more-or-less described in the QEX article, a summation of weighted contributions from adjacent symbols. Consider the shape of the Gaussian two-sided curve which is defined by *equation (3)* and shown in *Figure 1* of the QEX article. Our wanted train of symbols effectively slides across this curve which determines at any time how much, or the weighting, of each adjacent symbol is added into the final waveform. At any one instant the sum of the weightings should equal unity. When the current symbol is central to the curve, only that one contributes to the waveform, but at the half-way point either side there is an equal contribution from the current, and either the last or the next symbol depending on the point in the transmission cycle.

The first decision to be made is what subdivision, or sampling rate of the Gaussian waveform is to be used. As a simple PIC processor was to be used for the controller, making this a binary number would be advantageous. Experience suggested that 64 samples at 8-bit resolution over the whole symbol would suffice. Even for the fastest FST mode with a symbol rate of 16.67Hz, the resultant sampling would only be at around 1kHz which would present no timing difficulty to the processor. Of these 64 samples, due to 8-bit rounding of coefficients only some 25 of them actually sit on the slope of the curve, the rest have either zero amplitude or are at maximum.

An Excel spreadsheet implementing the equations in the QEX article was used to generate the values to be used in a lookup table in the PIC. Due to the vagaries of binary arithmetic and to keep things simple, the value for the amplitude could not be allowed to go to the maximum allowed for a byte, so the curve was designed to go from 0 to 250 maximum. The symmetrical nature of the curve means that only half the values need to be stored; the other half are generated by reflection of the address. The half-curve of Figure 1 shows the actual contents of the lookup table. To keep the summation of contributions symmetrical, a 64 times sampling waveform actually needs 65 points, numbered 0 – 64 in the lookup table.

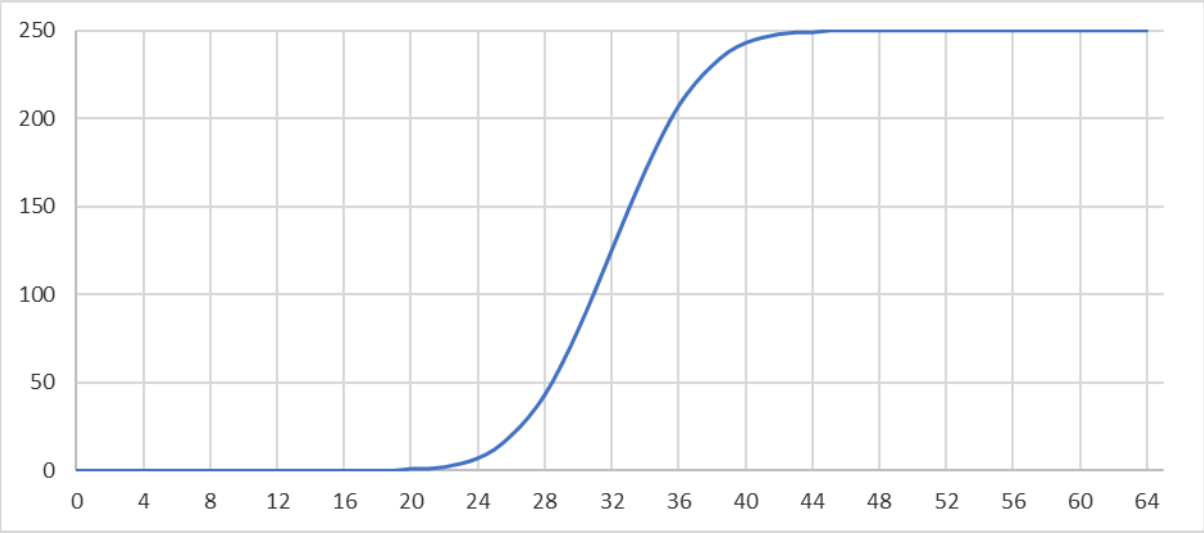

Figure 1 The Gaussian pulse shaping – axis values correspond to those in the processor lookup table.

PIC Coding Details

For the first test, the WSPR beacon code from [1] was used, with the CWident, frequency hopping and duty cycle settings all disabled to allow continuous waveform transmission. The interrupt routine that generated the symbol timing, and now the sampling, was completely rewritten. An interrupt is generated using TIMER1 as a clock divider to give the sampling rate, 64 times the symbol rate, enabled whenever the waveform is to be generated. A sample counter (*SampCount*) runs repeatedly from zero to 63, and each time it rolls over a symbol counter (*SymbCount*) runs from 0 to 159. When both counts are complete, the timer interrupt is stopped.

Initially it was thought that only two symbols need be used in the waveform, the last one (*LastSymb*) and the current one (*ThisSymb*), going from 100% of *LastSymb*, through 50% of each, to 100% of *ThisSymb* as defined by the curve. Ignoring any contribution from *NextSymb* would result in the waveform being transmitted with an overall delay of half a symbol. First thoughts were that this would not matter as the

corresponding delay is just 32ms at the fastest rate for FTS4-15 and only a second or so for the five minute long sequence. So for each sample, the instantaneous frequency data sent to the DDS is made up from

ThisSymb * Table(SampCount) + LastSymb * Table(64 – SampCount)

A version of code using this algorithm was tested and the resulting waveform decoded successfully both locally and, using FST4W, when tested on air via WSPRNET [3] for all the mode from FST4-15 to FST4-900. However the slowest one, FST4-1800 with s 30 minute cycle time never gave any decodes. The clue was in the reported DT values from WSJT-X when decoding the -300 and -900 modes, of 1 second and around 2 - 3 seconds respectively. Clearly the half symbol delay, 5.5 seconds for FST4W-1800 mode, was throwing the decoder. The waveform would have to be generated properly, using *LastSymb*, *ThisSymb* and now *NextSymb*. The equation for each sample was changed to :

LastSymb * Table(32 – SampCount) + ThisSymb * Table(SampCount + 32 or 96 – SampCount) +
NextSymb * Table(SampCount – 32)

In each case, if the table address falls outside the range 0 – 64 it is frozen at 0 or 64 respectively, giving zero or full weighting in each case. The address for *ThisSymb* has to range from 32 at the start for a 50% contribution, through 100% at count 32 for the middle of the pulse and back to 32 for the final sample. The addresses for *LastSymb* and *NextSymb* range over 0 – 32 giving a zero to 50% contribution from each.

Just before the start of each new symbol, the three values are shifted so *ThisSymb* becomes *LastSymb* and *NextSymb* become *ThisSymb*. *SymbCount* is now used to lookup *NextSymb* each time. Clearly, just before the start of each sequence *NextSymb* has to be preloaded so it gets the first value to be sent and then transfers it to *ThisSymb* at the start. *LastSymb* has a value of zero at the start, by definition.

Using this algorithm, all mode and speed combinations decoded successfully, with a very satisfying report of DT=0 in all cases.

Bells and Whistles

Now that the basic code generation was working and properly generating all seven speed variants, it was time to work out a way to make the modes user-selectable. The original WSPR source had no need for user selection, the message and duty cycle were decided at the start and blown into the PIC's EE memory along with frequency codes specific to the RF output frequency and DDS clock. But for FST4 / FST4W it would be advantageous to include some user input via a front panel control, to at least adjust the speed.

The WSPR source hardware that was being reused for this beacon source had only two spare I/O pins on the PIC device, so a binary thumbwheel switch or anything similar wasn't a possibility. One of the two spare pins was PORT A4 which is a Schmitt trigger input. By connecting a capacitor from this pin to ground, and a variable resistor from it to the supply rail, the capacitor is charged at a rate depending on the variable resistor setting. So a pseudo-analogue input can be achieved by measuring the time it takes for the capacitor to charge up to the Schmitt threshold. It works like this:

Pin A4 is set as an output and cleared for a few milliseconds to fully discharge the capacitor. It is then set as an input with a resulting high impedance and the capacitor starts to charge up via the variable resistor. Simultaneously a fast counter is started. As soon as the voltage on the capacitor reaches the Schmitt threshold the counter is stopped. By suitable choice of count speed versus the range of the CR time-constant, numbers can be kept realistic and the resulting count is a measure of the variable resistor's setting from min to max resistance. A lookup table or set of comparison thresholds can be used to turn this value into the required range of values wanted. With typical tolerances of components, especially the

capacitor, some adjust-on-test coding needs to be made to give a workable range. Here a counter reading that varied nominally over the range 20 to 190 was turned into a setting of 1 to 8. This number corresponds to the seven speed values for FST4 with the final setting giving an extra-wide tone spacing and medium symbol rate for display and test purposes; something that showed exactly the waveform being generated when viewed on a waterfall display and 'sounds good'. A waterfall plot of the signal in wide spaced demo mode can be seen in Figure 2.

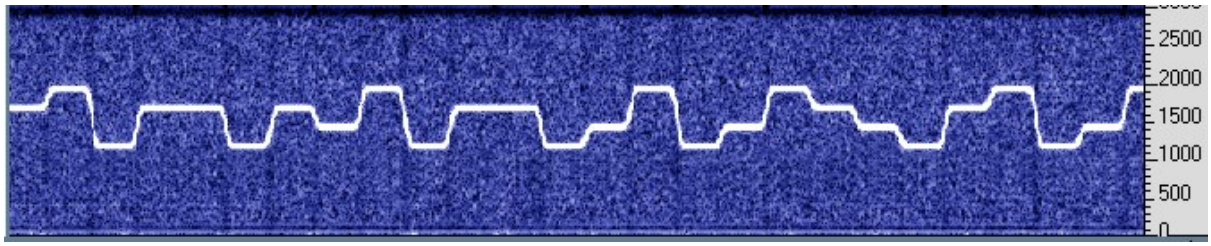

Figure 2 Spectrogram of the Output Waveform in the Wide Spaced Demo Mode

The CR network is read when the processor starts. To change the speed setting the variable resistor has to be placed in one of eight positions then the processor's reset button pressed. Feedback of the position selected is given by flashing the status LED with the character '1' to '8' in Morse, and also by playing this over the RF link so the character can be heard in a local receiver tuned to the RF output.

Since a means of user-adjustable input was now available giving eight settings, it was also used to allow adjustment of the Tx duty-cycle and to turn off or on the frequency hopping. To enable these to be adjusted the test switch, that in normal operation is used to generate a plain carrier, is turned on and the processor reset. When activated at turn on or processor reset, an input setting of '1' – '5' sets the Tx duty cycle to 100/50/33/25/20% respectively. '6' is reserved for future use, '7' turns on frequency hopping, and '8' turns it off. Settings are reported by the LED flashing and RF playing CW in the format 'D1' to 'D5' for duty cycle, followed by 'H' or 'X' for hopping on or off. All CW messages are sent twice. The Test Switch has to then be set off before the beacon source will go into normal operation so the variable resistor now has to be set to the desired speed setting. Both duty cycle and hopping status are stored in non-volatile memory so they don't have to be set each time the source is turned on. Just remember not to turn it on with the test switch activated.

This somewhat convoluted setting routine is in place only because existing hardware was adapted to use. Starting a new design from scratch, a processor with more I/O pins would allow a more conventional user interface.

A CW Ident is played after the FST4 message completes. The CW waveform uses raised cosine amplitude ramping to reduce key-clicks and control bandwidth.

Generating Message Data and Storing Parameters

The 160 two-bit symbols are stored within the PIC code, in its EE area, in compressed form with four symbols per byte, with the most significant bits transmitted first. To generate the symbols the utility *FST4SIM.EXE* which is part of the WSJT-X suite needs to be invoked. It be found in the .BIN directory under the WSJT-X install directory. In normal use this utility directs output to the text , or command screen. A number of command tags have to be included to get the utility to run, most of which are irrelevant here as only the symbol output is needed. A typical minimum command would be :

C:\Coding>fst4sim "G4JNT IO90 10" 15 1000 0 0 0 0 0 F

The meaning of all the command tags can be seen by just typing *FST4SIM* on its own to get a help message. A wrapper utility encapsulates and simplifies the call to *FST4SIM*, redirecting its output to the software that then assembles the symbols into bytes which are then put into a form suitable for copying into a PIC assembler file. As well as being saved to a .INC file, the four-symbol bytes are also copied to the clipboard ready for direct pasting into the PIC assembly file. The utility is *GENFST4* and a typical example of its usage can be seen in the screen dump of Figure 3.

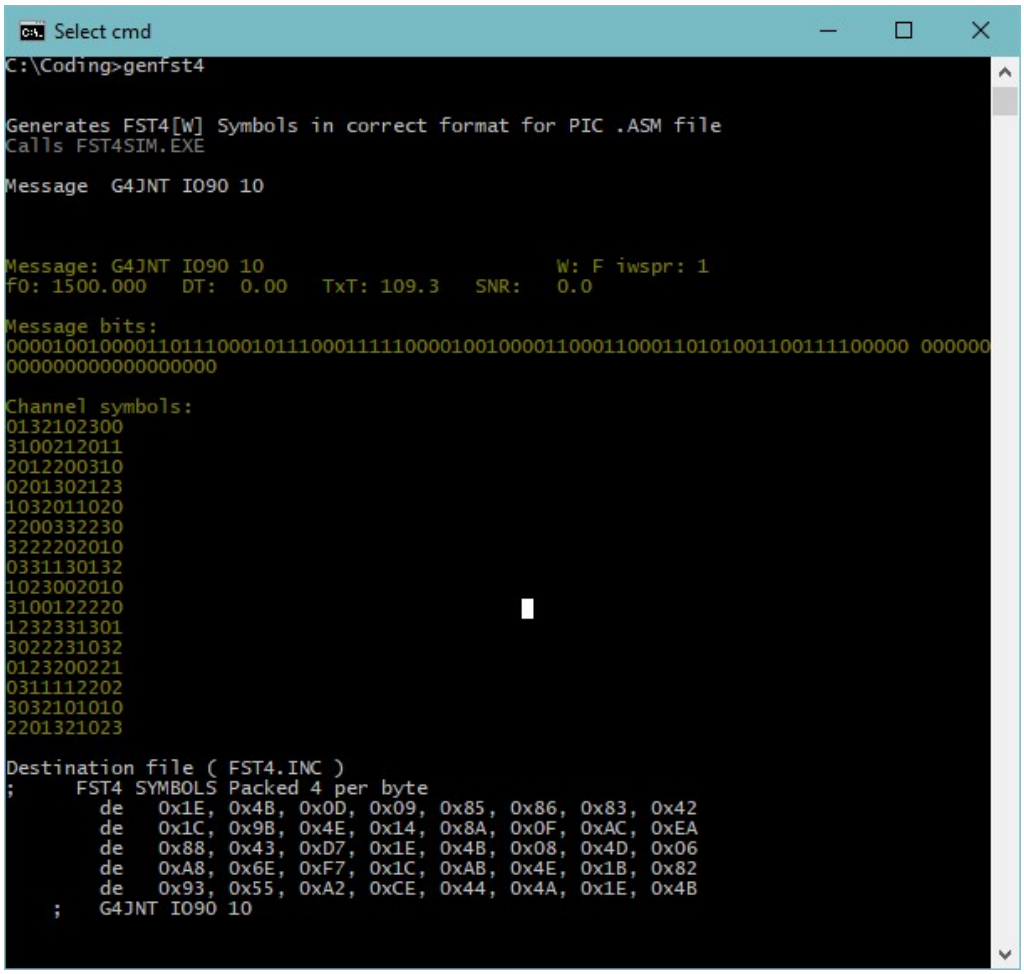

Figure 3 Screen dump from the utility *GENFST4*. The text in yellow is the output generated by *FST4SIM*.

GENFST4 needs to be in the same folder as *FST4SIM*. If copying the latter from the standard WSJTX folder to another folder, a number of .DLLs from the wsjt\bin will need to be copied as well. If they are not present, running *FST4SIM* will generate an error message and list which ones are required. The duty cycle and hopping details are stored in EE memory and can be set in the .INC file, so avoiding the need for the convoluted setup procedure for a fixed beacon setup.

Other data stored in the PIC's EE area contain the values for RF carrier frequency, the delta step (1/250 of the tone spacing) for each of the speeds, and the Frequency Hopping grid. This latter value will typically be around 1.5Hz allowing the up to 128 multiples to cover most of the 200Hz wide WSPR band. All these parameters can conveniently be placed in a separate .INC file, an example is shown below. It is easier to enter the frequency constants as numbers and let the assembler turn them into EE contents (the 'de' expressions), done here just for the *FREQCONST* values.

```
;Specific values for each mode  stored as constants here for ease of placing in EE format.  DDS code for sampling delta freq.
        FREQCONST15             =       0x01CA22 ;      16.6666Hz / 250  at  160 MHz Fs
        FREQCONST30             =       0x0C459  ;      7.143Hz     Let the assembler put into EE format
        FREQCONST60             =       0x54E9   ;      3.088Hz
        FREQCONST120            =       0x283E   ;      1.464Hz
        FREQCONST300            =       0x0F57   ;      0.558Hz
        FREQCONST900            =       0x04F5   ;      0.1803Hz
        FREQCONST1800           =       0x0275   ;      0.08929Hz
        FREQCONSTTEST           =       0x1AD7F3 ;      250Hz tone spacing  (delta = 1Hz) for testing / demo

org  0x2100

ControlPLL  de  d'8'    ;PLL Multiplier.   In this case 8 * 20MHz for 160MHz clock
DutyCycle   de  0x81    ;MSB is Hopping Flag.  Duty cycle in LSBs, 1 =100%, 2 = 50%, 3 = 33%, 4 = 25%, 5 = 20%

FreqConsts              ;Stored as 4 byte values for convenience, derived from values above.
        de  (FREQCONST15 >> d'24') & 0xFF, (FREQCONST15 >> d'16') & 0xFF, (FREQCONST15 >> d'8') & 0xFF, FREQCONST15 & 0xFF
        de  (FREQCONST30 >> d'24') & 0xFF, (FREQCONST30 >> d'16') & 0xFF, (FREQCONST30 >> d'8') & 0xFF, FREQCONST30 & 0xFF
        de  (FREQCONST60 >> d'24') & 0xFF, (FREQCONST60 >> d'16') & 0xFF, (FREQCONST60 >> d'8') & 0xFF, FREQCONST60 & 0xFF
        de  (FREQCONST120 >> d'24') & 0xFF, (FREQCONST120 >> d'16') & 0xFF, (FREQCONST120 >> d'8') & 0xFF, FREQCONST120 & 0xFF
        de  (FREQCONST300 >> d'24') & 0xFF, (FREQCONST300 >> d'16') & 0xFF, (FREQCONST300 >> d'8') & 0xFF, FREQCONST300 & 0xFF
        de  (FREQCONST900 >> d'24') & 0xFF, (FREQCONST900 >> d'16') & 0xFF, (FREQCONST900 >> d'8') & 0xFF, FREQCONST900 & 0xFF
        de  (FREQCONST1800 >> d'24') & 0xFF, (FREQCONST1800 >> d'16') & 0xFF, (FREQCONST1800 >> d'8') & 0xFF, FREQCONST1800 & 0xFF
        de  (FREQCONSTTEST >> d'24') & 0xFF, (FREQCONSTTEST >> d'16') & 0xFF, (FREQCONSTTEST >> d'8') & 0xFF, FREQCONSTTEST & 0xFF


FreqData    de          0x00,0x38,0x47,0x6F,0x2A,0x5B           ;137.4kHz  at 160MHz DDS clock
RandJump  de            0x00,0x28,0x43,0xEC                     ;1.5Hz * (3 - 131) spreading covers much of 200Hz of band


FST4MsgData
 ;      FST4W SYMBOLS Packed 4 per byte
  de  0x1E, 0x4B, 0x0D, 0x09, 0x85, 0x86, 0x83, 0x47
  de  0x1C, 0x8B, 0x4E, 0x1E, 0xAC, 0x23, 0x9E, 0xAC
  de  0xA2, 0x72, 0x87, 0x1E, 0x4B, 0x1D, 0x03, 0xD2
  de  0x9A, 0x3B, 0xA9, 0x21, 0xCB, 0x4E, 0x1B, 0xFB
  de  0x39, 0xCE, 0x77, 0x37, 0x15, 0xB6, 0x1E, 0x4B
 ; G4JNT IO90 20

CWMsg   de  "G4JNT",0
```

Figure 4 Typical Contents of the FST4BCN.INC include file. This contains all frequency setting data as well as the FST4 symbols and optional CW identifier sent after every transmission.

Conclusions

This design has demonstrated how a simple beacon source can generate a correctly formatted and timed FST4 message using a basic level PIC processor and 48 bit DDS. The Gaussian interpulse waveform is sampled at 64 times the symbol rate and generated with a lookup table.

Direct replication of this design is probably not a good idea, but for anyone wishing to adapt it for other processors and frequency generators the PIC source code *FST4Bcn_9852.asm*, an example .INC file *FST4Bcn.inc*, the *GENFST4* utility and the *Excel* spreadsheet for generating the Gaussian lookup table can be found at [4]. The PIC source code is fully commented with variables given appropriate names, and in conjunction with the text above should enable a full understanding of the process.

References

[1] Description and PIC assembler code for a WSPR beacon using a DDS source
http://g4jnt.com/WSPRBCNS.ZIP

[2] The FT4 and FT8 Communication Protocols. QEX July / August 2020
https://physics.princeton.edu/pulsar/k1jt/FT4_FT8_QEX.pdf

[3] WSPRNET, the global WSPR (and FST4W) reporting network. http://wsprnet.org/drupal/wsprnet/spots

[4] PIC code and *GENFST4* utility. http://g4jnt.com/FST4Beacon.zip