LF / Audio Beacon Source for WSJT Modes

Andy Talbot G4JNT October 2011

Overview

The unit described here is a development of the JT4 Generator and 4-Frequency DDS modules published previously. That could only generate modulation types that used four tones and was only suitable for JT4 or WSPR. JT65 requires that the DDS supplies 65 equally spaced tone frequencies, so an alternative solution for setting the PIC used for the DDS from the one generating the code was needed. A serial interface meets the requirements.

The DDS interrupt, typically at 100kHz or higher, must run continuously. generating the output waveform. The serial data transfer has to be robust enough to withstand timing jitter and not interfere with the interrupt process in any way. A synchronous three-wire interface was chosen, with Clock, Data and Strobe signals. It would have been preferable to keep to a two wire interface so hardware could remain almost unchanged from the parallel-programmed four tone version, but although a protocol based on clock counting was tried, it was not a robust as needed. A modified I2C type protocol was also rejected due to complexity of implementation of the slave end in PIC software. Details of the three wire signalling are given in the section describing the frequency source.

As the frequency setting data is now sent over the serial interface, there is now little point in having the user setting switches (originally for selecting JT4A/c/d/g or whatever) on the DDS. These have now been moved to the Code generator PIC with the advantage that the frequency source is now a completely dumb DDS. All frequency information is now stored in the same device as the code details, message etc. This simplifies reprogramming the beacon source when changing modes.

JT4 in all its variants (A-G) consists of a four tone Multi Frequency Shift Keyed (4-MFSK) waveform, with the spacing between the tones chosen depending on the frequency band and expected spreading. [1] The MFSK message consists of 207 symbols (one of four sequential tones) transmitted at a rate of 4.375Hz, the whole message therefore taking about 48 seconds to send. A rigid timing structure is in use, and the start of the transmission must coincide with the UTC minute interval. For beacon usage, the even minute has been universally chosen as the reference start time for beacons using this mode. However, the decoding software does have a monitor function whereby transmissions in both even and odd minute slots are decoded.

WSPR is very similar to JT4, although is has a symbol rate of 1.46Hz and transmits 162 symbols over 110 seconds on a 2 minute repeat cycle.

For the decoder to work correctly, the start point must be accurately defined, being no more than a few seconds late, and no more than one second early (the protocol was originally designed for EME with its 2 seconds delay). The entire message for JT4 and JT65 modes contains exactly 13 characters taken from an alphabet of letters, numbers and a few punctuation symbols. WSPR contains somewhat more information coded in a different way More details of WSJT coding can be found at [2]

JT65 also works with a precisely timed 48 second sequence, but now transmits one-of 65 tones using 126 symbols at a rate of 2.69Hz. 64 tones are dedicated to message data and

the other one, the lowest, is a synchronisation tone sent at pseudo random intervals. More details on JT65 coding can be found at [1]

Code Generator Module

The unit described will generate correctly timed and formatted JT4, JT65 or WSPR words on the serial interface, sending the frequency data to the DDS for every symbol transmitted.

A 16F627 or 16F628 PIC monitors the GPS serial data line and decodes the real time information from the GPS. Every even minute, or every minute depending on requirements, the 00 seconds marker is identified and the pre-stored symbols are sequentially output on the serial output lines.

Depending on the format transmitted, the PIC has to be programmed with either ***JT4GEN_SER.HEX***, ***WSPRGEN_SERIAL.HEX*** or ***JT65GEN_SER.HEX*** as appropriate. The PIC does generate a CW message at the end of the sequence, but at the moment this is unused. The external keying option has been removed.

Connecting the GPS module.

The description that follows, as well as the PIC firmware supplied, assumes that serial data in one of two formats is available. The proprietary binary format given by the Motorola Oncore or M12 type GPS module at 9600 baud or standard NMEA text messages at 4800 baud carrying the *$GPRMC* string. The polarity of the data can be selected at the time the PIC firmware is compiled. Either native 5V logic or RS232 polarity can be catered-for

Figure 1 shows the circuit diagram of the generator module. Two input lines carry the serial data and 1 pulse-per-second synchronising signal. There is also a third GPS interface connection shown, an output from the PIC to the GPS receiver. At the moment this is not used and does not have to be connected. It has been included for any future version that could include GPS receiver initialisation. A red-green LED shows the operating status and indicates whether the GPS is synchronised. When valid data appears from the GPS module long green flashes are shown. When the GPS receiver is not synchronised, these change to short flashes. When WSJT data is being sent, the LED flashes red at half-symbol rate, and then shows the CW data.

> *Note that at the time of writing, this actual PIC code has not been tested using the Motorola Oncore family of GPS receiver modules. However, Oncore modules have been used in other DDS based WSJT sources.*

PIC Coding Details

All information relating to the message, frequencies and setup need to be programmed into the PIC at the start. All values need to be calculated beforehand and included within the source file which is compiled to give the .HEX file for download to the PIC device. These include frequency codes for the DDS, WSJT message data and the CW message (when this

is implemented). Compile-time flags are used to define the data polarity and format from the GPS.

The PIC firmware is contained in the source files *JT4GEN_SER.ASM, JT65GEN_SER.ASM* and *WSPRGEN_SERIAL.ASM* .The symbol information resides in auxiliary include files *JT4SYMBS.INC* , *JT65SYMB.INC* or *WSPTONES.INC* which can be generated automatically by appropriate utilities described later. Alternatively, the symbols can be derived from the WSJT software, following Joe's instructions supplied with the software suite, formatted and entered manually into the include file.

For JT4 and WSPR Each of the 207 /162 symbols is formed from two bits giving a value from 0 – 3 which are packed four to a byte, most significant first to give 51 / 41 bytes in total. (As listed, they are read out in order left to right, top to bottom) For JT65 the 63 symbols are stored directly as values from 0 to 63

Customise the Source File

Change the compile-time flags and CW message data to suit your requirements, and generate a new symbols .INC include file. Save the new assembler file and use a utility such as MPASM (available from the Microchip website or included within the MPLAB suite) to generate a new .HEX file for programming into the PIC device

The code supplied is designed for 16F627A and 16F628A type devices and either are suitable. If using the 16F628 it is not essential to change the type specified in the assembly code. Some PIC - programmers may object, but they can usually be overridden.

Compile-time flags.

These appear at the start of the assembler listing as shown in the table below

NMEAPol defines if the polarity of the data coming from the GPS receiver is 0/5V logic level as supplied directly by most GPS modules, or RS232 polarity for direct connection to a PC. Some early Garmin modules supply this latter polarity, as do some GPS receiver systems. . Use **0** for 5V Logic level / polarity, **1** for RS232.

Please note that if true positive/negative RS232 voltage levels are encountered, an additional resistor of around 4k7 needs to be inserted in the Data In line to prevent excessive current into the PIC interface pin

GPSDataType should be set to **0** for Motorola binary format data at 9600 baud; Use **1** for NMEA ASCII format at 4800 baud

BOTHMINUTES (not applicable to WSPR) defines if the JT4 or JT65 is sent every minute, or every two minutes on the even minute boundary. It should be set to **0** for conventional even minute transmissions, and set to **1** for near 100% duty cycle transmission of the message every minute. There is no option for transmitting only on the odd minutes.

IGNOREPPS allows timing information to be derived from the GPS data stream alone, without any need for the 1-PPS signal. This simplifies the connection for some GPS receiver modules, but does mean the transmission timing could have up to one second

uncertainty. Set to **0** for normal high accuracy timing using the 1-PPS signal, Use **1** for GPS serial data based timing only.

Compiler Constants

CWSPEED is a compiler constant and defines the dot length of the CW, in milliseconds. Use **d'100'** for 12WPM, **d'75'** for 16WPM etc.

BAUD9600 and BAUD4800 should not be changed.

```
GPSDataType        =       1       ;1 = NMEA  0 = Motorola Binary
NMEAPol            =       1       ;1 = RS232 Levels, 0 = TTL
BOTHMINUTES        =       1       ;1 = every minute, 0 = Alternate (even) minutes

CWSPEED    =  d'50'        ;CW Dot length, ms
BAUD9600   =       d'40'   ;6.N + 18 = Fc/Baud  or  N ~ (Fxtal/Baud)/24 - 3
BAUD4800   =       d'83'
```

..... Data Type Specific coding .....:

JT4SYMBS.INC

This include file is generated automatically in exactly the form shown as a result of running the utility ***GENJT4.EXE***. It should not be necessary to alter the file in any way. As the file is regenerated and overwritten each time ***GENJT4*** is run, it is advisable to save a copy under a different name – eg*, GB3SCS_JT4SYMBS.INC*.

The WSJT software does offer the ability to generate the symbol data in a listed form, and users may want to use this route instead – for example to include a 'QSO-type' message into the beacon data instead of 13 characters of plain text. In this case, the individual symbol data in the form of 207 numbers with values 0 – 3 will have to be assembled manually into the EE data bytes, four-at-a-time starting with the most significant pair of bits. For example, if the first eight symbols generated are 3,1,2,0,2,1,3,0, the resulting first two bytes will be b'11011000' and b'10011100' or in hex 0xD8, 0xC0. Both these formats, binary or hex, (or even decimal as d'nn') are acceptable to the compiler. Read the WSJT documentation for further details of how to generate the symbol list.

```
;  JT4 Symbols  generated from GENJT4     G4JNT Jul 2009
;  Message data 'GB3SCS IO80UU'

    de  0x00, 0xD8, 0x14, 0xDA, 0xC4, 0x02, 0x8D, 0x28
    de  0xAA, 0x0A, 0xC7, 0x9C, 0xEF, 0xD6, 0x68, 0xC3
    de  0xA5, 0x74, 0x2C, 0x6A, 0x75, 0x1E, 0xB8, 0x34
    de  0xC4, 0xC6, 0xF5, 0xC4, 0x67, 0x33, 0x9D, 0xA4
    de  0x59, 0x76, 0xA9, 0x65, 0x83, 0x53, 0x73, 0x50
    de  0xC0, 0x51, 0xE9, 0x2B, 0x57, 0x63, 0xE2, 0x34
    de  0x26, 0x73, 0xD6, 0x6C
```

JT65SYMB.INC

This include file is generated automatically in exactly the form shown as a result of running the utility ***GENJT65.EXE***.

GENJT65 is a “wrapper” programme that calls up another piece of code written by Joe Taylor for generating the JT65 symbols, ***JT65CODE.EXE***. This is included in the archive. It is possible to generate the symbols from the WSJT software; see its documentation for details.

WSPTONES.INC

This include file is generated automatically in exactly the form shown as a result of running the utility ***GENWSPR.EXE***.

DDS Frequency Source

The frequency source implements a simple DDS designed around a 16F627A or 16F628A PIC programmed with the code named ***SER_FreqSource.HEX*** Source code can be seen in ***SER_FreqSource.asm*** .

The desired frequency is sent via the three input lines, Clock, Data and Strobe which transfers the 24 bit word supplied from the Code Generator.

The DDS runs at a clock rate equal to the device oscillator / 192, or the internal clock divided by 48. The maximum specified frequency for this device is 20MHz, so with a crystal of this value, the DDS clock runs at 104.167kHz. Higher crystal frequencies will usually work, and 24MHz is more than likely OK, giving a convenient DDS clock of 125kHz. Another useful low cost off-the-shelf crystal frequency is 22.1194MHz giving a 115.2kHz clock

The DDS word is based on a 24 bit register, so the resolution is around 6mHz at this clock. An 8 bit D/A converter is implemented by a discrete 2-2R ladder. The 8 bit D/A, using the 6dB / bit rule of thumb, suggests spurii will be around -48dBc. A plot of the spectrum generating an output at 25kHz can be seen in Figure 5. The worst case spurious in this case is at -42dBc.

Figure 2 shows the circuit diagram with a PCB layout in Figure 3 and a mirror imaged copper track pattern at 1:1 scale in Figure 4 The typical output spectrum can be seen in Figure 5

Frequency information is kept in EEPROM in the code Generator , and is stored in different ways for JT4 / WSPR and JT65 versions.

JT4 and WSPR Frequency Calculation and Programming

For JT4, up to four sets of four tones are stored as four bytes each. Only the first three are used for defining the frequency, the fourth is ignored The set of four in use at any time is selected via the user switches on the PIC lines A3 and A4. This way, moving between JT4A, JT4C, JT4E, JT4G for example can be made via switch selection without having to reprogramme the chip. For WSPR only one set of four frequencies is stored.

The example below is for JT4. For WSPR frequencies follow the same procedure but store only the first block of four frequencies.

1) Determine the DDS clock, $F_{clock} = F_{osc} / 192$

2) For each of the four desired frequencies, F_N calculate the value N from $N = F_N / F_{clock} * 2^{24}$.

2) Convert to Hex notation and store in the PIC assembly file in the format shown below

4) Repeat for each of the other three frequencies.

5) Repeat all again for the other three switch-selected blocks (JT4 only).

6) Save the data in the appropriate source file.

Example : Tone frequency 24997.8Hz with a 20MHz crystal

Fclock = 20MHz / 192 = 104166.7Hz

N = 24997.8 / 104166.7 * 2^{24} = 402617

Convert to Hex = 0x3D6F42

Store as three bytes (with a dummy 4th one)

Assemble the code, using, for example the *MPASM* assembler

Programme the chip

```
   Org   0x2100
FreqData
         ;Direct Freq Gen   JT4A   Ref 0.125MHz
    de  0x33, 0x2F, 0xC2, 0x65    ;  Tone 0  00.0249934MHz
    de  0x33, 0x32, 0x0D, 0x99    ;  Tone 1  00.0249978MHz
    de  0x33, 0x34, 0x58, 0xCD    ;  Tone 2  00.0250022MHz
    de  0x33, 0x36, 0xA4, 0x00    ;  Tone 3  00.0250066MHz

         ;Direct Freq Gen   JT4C   Ref 0.125MHz
    de  0x33, 0x25, 0x6F, 0xFC    ;  Tone 0  00.0249737MHz
    de  0x33, 0x2E, 0x9C, 0xCB    ;  Tone 1  00.0249912MHz
    de  0x33, 0x37, 0xC9, 0x9A    ;  Tone 2  00.0250087MHz
    de  0x33, 0x40, 0xF6, 0x6A    ;  Tone 3  00.0250262MHz

         ;Direct Freq Gen   JT4E   Ref 0.125MHz
    de  0x32, 0xF5, 0x44, 0xBB    ;  Tone 0  00.0248819MHz
    de  0x33, 0x1E, 0x8E, 0x60    ;  Tone 1  00.0249606MHz
    de  0x33, 0x47, 0xD8, 0x05    ;  Tone 2  00.0250394MHz
    de  0x33, 0x71, 0x21, 0xAB    ;  Tone 3  00.0251181MHz

        ;Direct Freq Gen   JT4G   Ref 0.125MHz
    de  0x32, 0x3B, 0x79, 0x52    ;  Tone 0  00.0245275MHz
    de  0x32, 0xE0, 0x9F, 0xE8    ;  Tone 1  00.0248425MHz
    de  0x33, 0x85, 0xC6, 0x7D    ;  Tone 2  00.0251575MHz
    de  0x34, 0x2A, 0xED, 0x13    ;  Tone 3  00.0254725MHz
```

(the 'de' before each line defines the four numbers as EE data. Anything after a semi-colon is a comment for reference purposes only. The 'FreqData' label must be on the extreme left hand side, immediately preceding the data itself. The 'org 0x2100' must be immediately before

JT65 Frequency Calculation and Programming

For JT65, Frequency data is stored in a different way

1) First decide on a reference frequency for the JT65 sync tone – the lowest of the set. For normal receiver and decoder operation this is tuned so it appears as a 1270Hz audio tone.

2) Calculate the four byte code to generate this frequency in the same way as is done for each JT4 tone

3) Save this in the ***JT65GEN_SER.ASM*** assembly file in exactly the format shown below. This is the *SyncReference*

4) Calculate the lowest tone spacing, that for JT65A and multiply by 256. This is given exactly by 11025/16 and is 689.0625Hz DO NOT TRUNCATE

5) Calculate the hex code for generating this frequency using the DDS clock rate determined earlier and as used for the reference. Save this as a four byte value as shown. This is the *ToneInterval*

For each transmitted symbol the *ToneInterval* is multiplied by its symbol code, 0- 63, and then by 1, 2, or 4 (determined by reading the user switches), divided by 256 and the result is then added to the *SyncReference* to give the tone value sent to the DDS.

The user switches are read as follows

00 = JT65A

01 or 10 = JT65B

11 = JT65C

```
; org   0x2100

FreqData      ;based on 125kHz sampling rate
       de     0x32,0xBA,0xDA,0xBA ; JT Sync tone  1270.46Hz + 23.5kHz
       de     0x01,0x69,0x44,0x67 ; JT65A Tone spacing * 256


JT65MsgData
 include "jt65symb.inc"     ;JT65 tones
```

Example, JT65 frequencies transmitted for SSB Rx carrier tuning at 23.5kHz.

125kHz DDS clock (24MHz crystal)

Reference Frequency for Sync tone = 23500 + 1270.46 = 24770.46Hz

N = 24770 / 125000 * 2^{32} = 851106525

Convert to Hex = 0x32BADADD (not quite the same as the table, the difference is approx 1mHz)

Store as four bytes 0x32, 0xBA, 0xDA, 0xDD under the label *FreqData*

Take the tone spacing * 256 and calculate the 32 bit equivalent

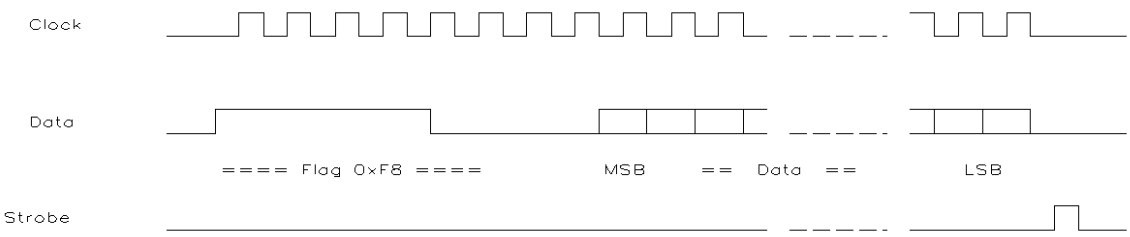
N = (11025 / 16) / 125000 * 2^{32} = 0x1694467

Store as four bytes 0x01, 0x69, 0x44, 0x67 directly under the Reference Frequency data

Utilities for helping in the calculation and programming are included in the archive

Serial Interface Protocol

32 bits are sent MSB first on the Data line. The first 8 are a fixed header flag, consisting of 0xF8 in hex, used for checking validity of the received data. This is followed by the 24 bits of frequency data. All are sent MSB first. Data changes are roughly coincident with the falling edge of the Clock signal. The clock exits in the low state. After 32 clock pulses have been sent, when all data has been transferred, a positive going strobe signals that data in the input register is to be loaded into the DDS frequency register. The strobe is only recognised if the clock signal is low during its rising edge. The serial clock / data duration is approximately 2ms per bit, so takes about 64ms to transfer the entire frequency word. This ensures it straddles many interrupt periods and the resulting timing jitter will not compromise data transfer.



References

[1] http://physics.princeton.edu/pulsar/K1JT/

[2] http://www.g4jnt.com/JTModesBcns.htm

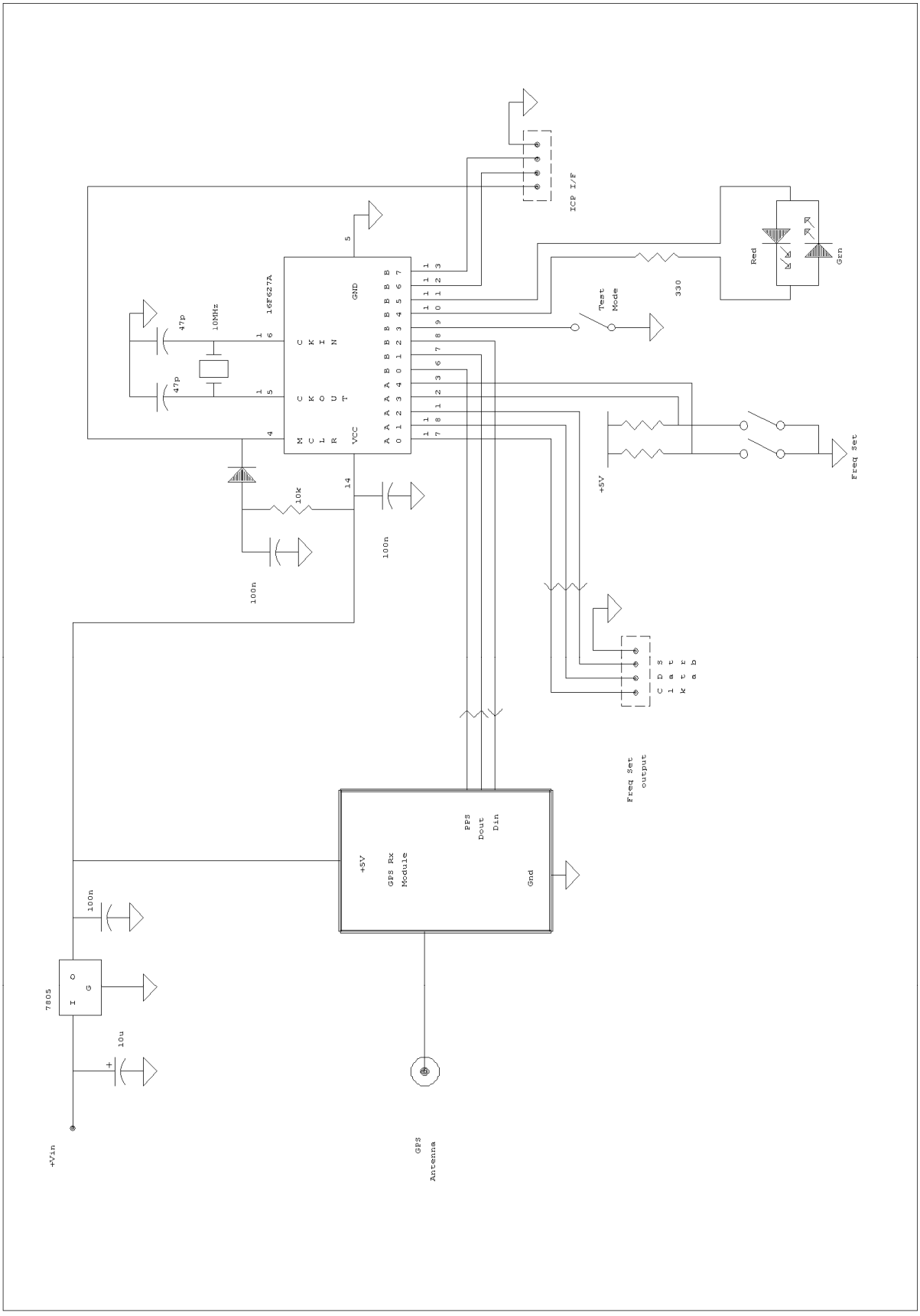


Figure 1 Code Generator module, provides frequency data on a three wire synchronous serial interface

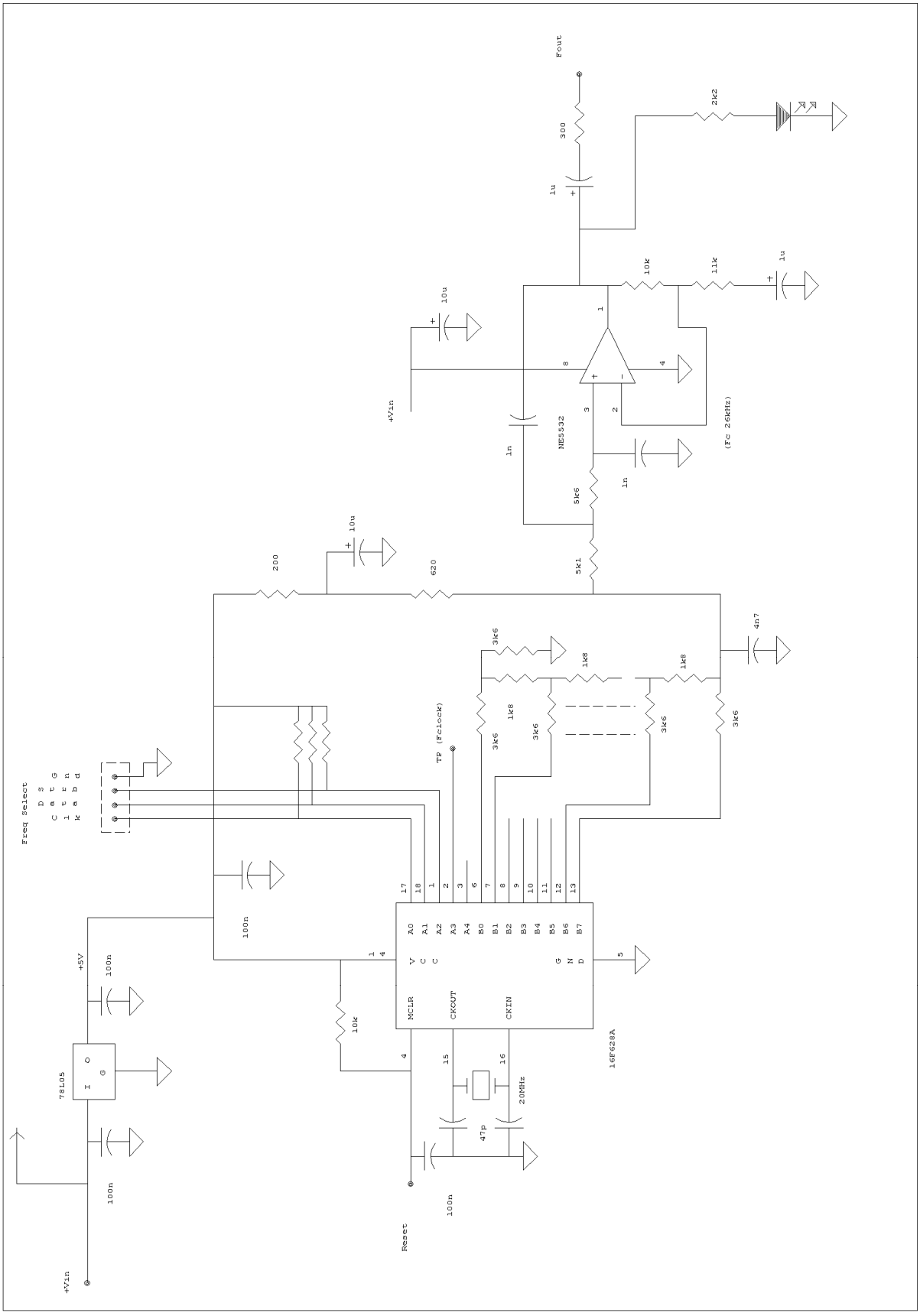
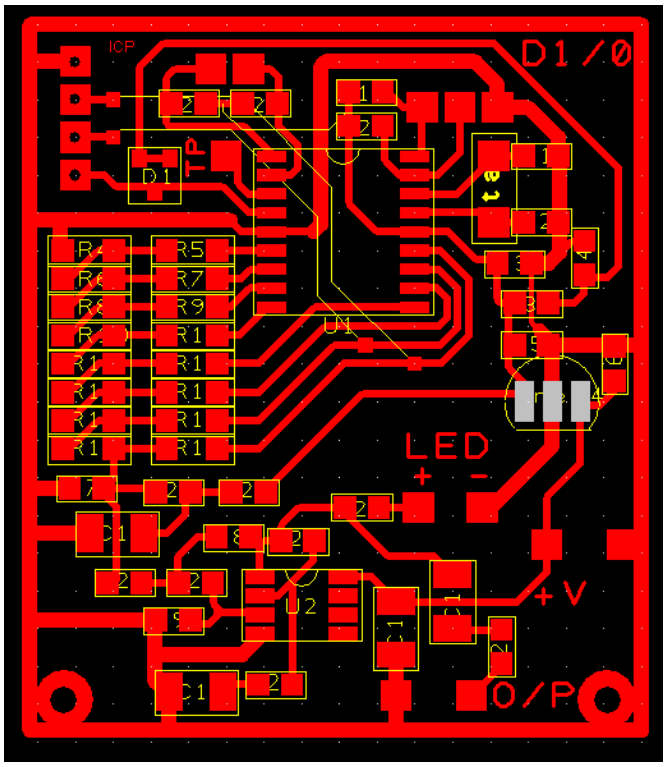


Figure 2 PIC Based LF DDS. Programmed via three wire synchronous serial interface



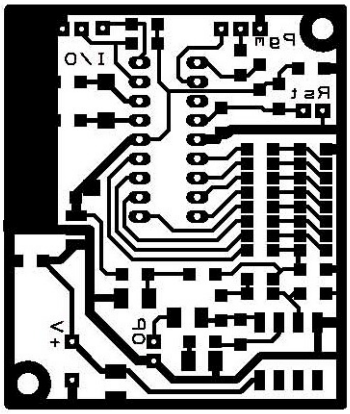


Figure 4 Mirrored 1:1 PCB

Figure 3 PCB Layout for the DDS Frequency Source

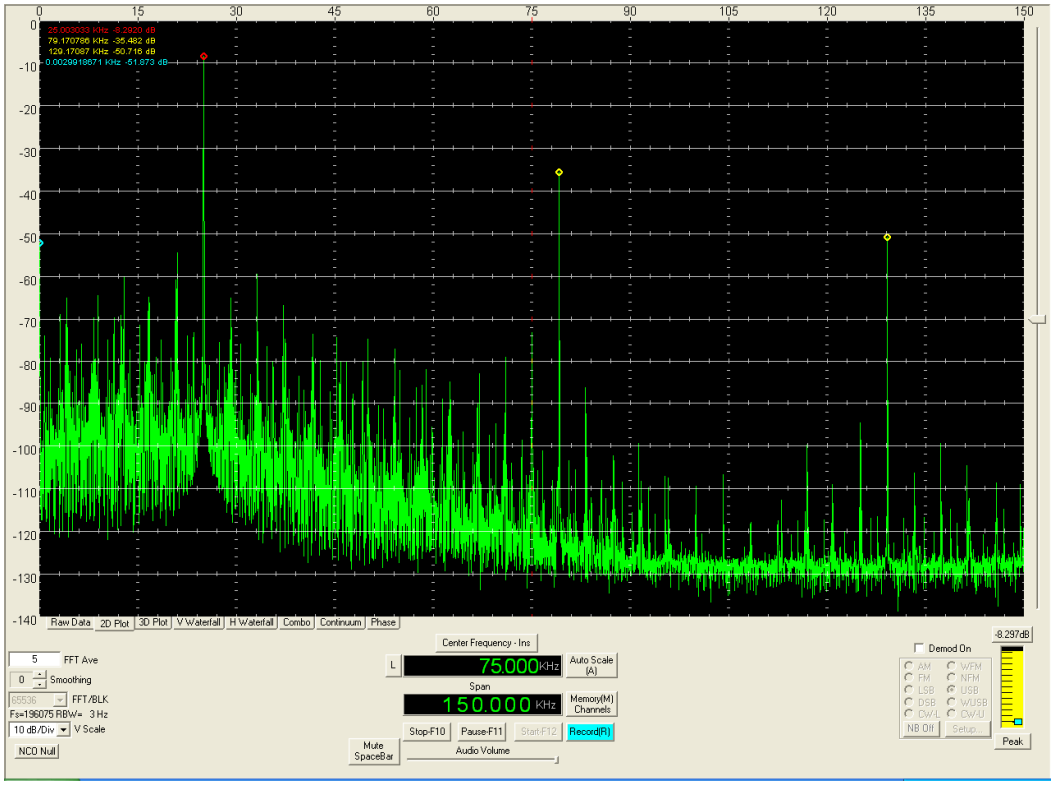


Figure 5 Output Spectrum at 25kHz

Annex 1

Free Running Versions

The PIC Code ***JT4GEN_freerun_serial*** , ***wsprgen_freerun_Serial*** and ***jt65gen_freerun_serial*** (included in the archive) do not need any external timing information and use the PIC's own oscillator to generate the appropriate intervals. All connections to the PIC are the same as for the standard version, except that ports B0-B2 are not used. Provisions has been built into the software for use – with care – of crystals of other frequencies. The timing for both the JT4 sequence and the real time count of seconds has to be derived by dividing down from a regular interrupt, which itself is generated by dividing down from the PIC internal clock. Only certain frequencies will meet the requirements for this,; the seconds count has to be exact, and the JT4 timing has to be close enough that at the end of the 48 second sequence, the timing has drifted by no more than a fraction of a symbol.

A spreadsheet ***WSJT_PIC_Timings.XLS*** has been included to assist in calculating the values.

The source code is customised in the same way as for JT4GEN , except that now only the ***BOTHMINUTES*** user flag is available. If crystal frequencies other than 10MHz are to be used, the divider values have to be changed at the start of the listing. See table A1 for calculation of these.

To synchronise timing, the module must be powered up or reset as close to the 00 seconds point as possible. Timing has been arranged so that it launches into code transmission immediately after power up / reset. When using the alternative minutes option, odd or even slots are possible depending on the reset point.

The LED flashes red at half symbol rate while sending, and for the CW message; it remains green during the idle period.

Timing accuracy depends on the PIC oscillator remaining on frequency. Assuming a reasonable 10 parts-per-million accuracy, this equates to a drift of one second per 100000s or less than 1s per day. If the frequency drifts low and the clock slows, the resulting permissible timing for successful decoding using the WSJT software can be up to 6 seconds late. It is not so forgiving of early timing, and only 1 second fast is permissible. So if you are unsure of timing drift, or long transmission periods are anticipated, it is safer to start / reset a few seconds after the minute marker.

For long term usage, a TCXO with 1ppm stability should give acceptable timing for several weeks or even months of operation

TMR0 Programmed value

Table A1 Selecting divider values for arbitrary oscillator frequencies

Choose a crystal frequency and enter this into the box under 'PIC Oscillator' in the ***WSJT_PIC_Timings*** spreadsheet. Do not change the values in 'Prescalar, and 'Interrupt Overhead' (4 and 12 respectively). These cannot be altered in this PIC code.

Check the timings are OK against the mode of choice and "1s timing". If an unsuitable crystal frequency is chosen, "**X no**" will appear in red against the associated entry. If this happens, choose another crystal frequency, or with care, adjust the "Interrupt Division" value. Do not use any value less than 100 in here; values from approximately 100 to 240 are acceptable. NOTE *a value of 256 is a special case and makes the PIC code run a lot happier ! In this case it is not necessary to reprogramme TMR0 every interrupt, and the appropriate lines of code are* remmed *out. The examples included, with a 19.6608MHz crystal frequency, use this option.*

When a satisfactory set of values are found, note the values generated for "TMR0 Programmed value", and the "Int Div N" for JT4 and 1s timing. Also the value for MSDELAY.

Copy these values into the assemble file listing and assemble the code.

How the timing is generated:

The oscillator frequency is divided by 16 (fixed) and clocks the 8 bit TMR0 counter. When this overflows an interrupt is generated. By pre-loading with a constant, ***INTDIVIDER*** each time it overflows, the resulting interrupt rate can be modified. The Interrupt frequency is therefore :Fxtal / 16 / (256 – N') where N' is the value preloaded each time it overflows. There is a bit of a snag as some clock cycles are taken up servicing the interrupt before N' is loaded, so the value has to be modified. The end result is to generate an interrupt at a precise rate – shown as the values against "1s timing" ***See the note above for the case where the divider is 256***

Two separate counters now count each interrupt. The seconds counter is checked against a value ***INTSPERSECOND*** that is numerically equal to the interrupt rate in Hz, or the "1s Timing" value. When this value is reached it is reset, and the seconds / minutes updated. All are started off at zero when the PIC is switched on / reset.

The Symbol timing is checked in the same way against a value ***JTDIVIDER*** that results in a rate acceptably close to 4.375Hz to give insignificant symbol overrun at the end of the