## Summary

The following explains the Fast Fourier Transforms of real waveforms on the QED Board.

## Description

The QED Board contains firmware programs that find the Fast Fourier Transforms (FFT) or Inverse Fourier Transforms (IFFT) of either complex or real, floating point or integer, arrays of one dimension. Documentation supplied with the QED Board (the QED Sortware Manual) shows how to find FFTs of complex floating point data. This application note provides further information and shows how to perform the FFT for real integer data. This is generally the most useful FFT because data that is acquired by the QED Board's 8- or 12-bit A/Ds is real integer data and the algorithms for transforming real integer data are the fastest of the FFT programs supplied.

## Performing Integer FFTs

The Integer FFT program, called Real.Integer.FFT, and the corresponding program for the inverse FFT, Real.Integer.IFFT, operate on data arrays that must be stored either on the same page as the FFT routine or in the common ram. These programs refer to their input data using starting addresses rather than by treating the data like formal arrays or matrices. The input data array must be organized along a single index, it must contain at least four elements, and it should have a number of elements equal to a power of two. There is an unchecked error if the number of elements is less than four or not a power of two. If you need to transform a two-element array use the complex FFT routine with the imaginary part of the waveform packed with zeros, or don't use the FFT routines at all and just compute the DC (or zero frequency) component as the average of the two points, and the aliased single frequency transform as the difference between the two points.

Another data array, a read-only look-up table containing the first quadrant of the cosine function, is also needed and must be either on the same page as the FFT routine or in common RAM, so that it can be addressed without a page change. An initialization routine for this table is provided, see the file "Using Integer FFTs".

The integer version of the FFT routine uses 16-bit signed integers; however, to avoid internal overflow the absolute values of the inputted data must all be less than $2^{14} = 16384$. This is not a significant limitation because the results of a 12-bit A/D conversion are always less than $2^{14}$; they can be used directly as 16-bit integers if unsigned, and if they are signed they can be sign extended to 16-bit integers. Eight-bit integers can be used too but they should also first be sign extended to 16-bits.

Interpreting the Time Series Waveform and Its Fourier Spectrum

The input data consists of N sequential, equally spaced samples of an input *signal*, for example a voltage waveform, sampled every $\bullet t$ seconds, for a total duration of T = N*$\bullet t$. We'll call these samples $S_k$ for $k = 0$ to N-1. These samples are stored sequentially in memory as 16-bit integers and the input to the FFT routine is just the starting address of this list of samples. The set of sampling times for this discretely sampled signal consisting of N values of $S_k$ are:

$$t_k = \{ 0, \bullet t, 2\bullet t, 3\bullet t, ..., (N-1)\bullet t \} \text{ seconds; or } t_k = k\bullet t \text{ for } k = 0 ... N-1 \qquad \text{Eqn. 1}$$

For example, we might take 1024 samples with the first sample taken at time zero and each sample thereafter taken at 50 µsec intervals. This corresponds to a sampling rate of 1/(50 µsec), or 20 kilosamples per second, or 20 kHz. We will assume that before sampling, in order to prevent aliasing, the waveform had been filtered to exclude frequencies of greater than one half of the sampling rate, that is, greater than 10 kHz in this case. Our total sample duration is 1024 x 50 µsec. = 51.2 msec. Our sampling times for the 1024 samples are,

$$t_k = k\bullet t = \{ 0, 50 \text{ µsec.}, 100 \text{ µsec.}, 150 \text{ µsec.}, ..., 51.2 \text{ msec} \} \qquad \text{Eqn. 2}$$

Taking the FFT of this discrete waveform will result in a discrete *spectrum* of magnitudes, $F_n$, for $(N/2)+1$ frequency components with a lowest nonzero frequency of $1/(N\bullet t)$ (or $1/T$) Hz, a greatest frequency of $(N/2)/(N\bullet t)$ Hz (or $1/(2\bullet t)$ Hz) and individual frequency components of,

$$f_m = m/(N\bullet t) = \{ 0, 1/(N\bullet t), 2/(N\bullet t), 3/(N\bullet t), ..., (N/2)/(N\bullet t) \} \text{ Hertz;} \qquad \text{Eqn. 3}$$

for the $(N/2)+1$ values of $m$, $m = \{0 ... N/2\}$

The lowest nonzero frequency, and the interval between successive frequencies, is called the *resolution* of the spectrum; it is $1/T$ Hz (or $1/(N\bullet t)$ Hz). The lowest nonzero frequency is also called the *fundamental*, *sampling fundamental*, or *first harmonic*; and the other nonzero frequencies are often called m'th harmonics. In our example, the fundamental is given by $1/(N\bullet t)$ = 19.53 Hz, the second harmonic is twice this frequency, the third harmonic is three times the fundamental, and all the frequencies are,

$$f_m = \{ 0, 19.53 \text{ Hz}, 39.06 \text{ Hz}, 58.59 \text{ Hz}, ..., 10.0 \text{ kHz} \} \qquad \text{Eqn. 4}$$

The frequencies of the spectrum are found with approximately 20 Hz resolution from zero frequency through the highest harmonic of 10 kHz. This highest harmonic, numerically equal to one half of the sampling rate, is called the *sampling cut-off frequency* or the *Nyquist critical (or cut-off) frequency*.

$$f_c = f_{N/2} = 1/(2\bullet t) \text{ Hz} \qquad \text{Eqn. 5}$$

At each frequency but the lowest (the zero frequency) and greatest (the $1/(2\bullet t)$ frequency) the spectrum comprises two numbers, one for the signed magnitude of the cosine phase of the signal at that frequency and one for the signed magnitude of the sine phase of the signal at that frequency.

The FFT is done in place so that the resulting array of magnitudes, $F_n$, replaces the original array, $S_k$. The result is formatted as pairs of cosine and sine components, except for the first pair. The first pair contains the signed magnitude of the zero frequency (DC) component, and the signed magnitude of the (aliased) highest frequency component (as a real, or cosine, frequency). The second pair contains the signed magnitudes of the cosine and sine of the first frequency (the fundamental or harmonic #1), the next pair those of the second harmonic and so on. For each pair but the first, the first element of the pair contains the signed magnitude of the cosine component, and the second element contains the signed magnitude of the sine component.

Note that in accordance with most engineering conventions the waveform is treated as though it is composed of only positive frequency components. The zero frequency component of the spectrum is the average DC magnitude of the input waveform. For the greatest frequency component (the Nyquist critical frequency) only the cosine phase is found and its coefficient in the spectrum is equal to the magnitude of the cosine wave of the greatest frequency.

If we think of the input waveform as being the sum of sine and cosine waves then the output spectrum is a list of the magnitudes of each of those sine or cosine components. For example, consider an input waveform, $S_k$, given by N samples indexed as $k$ = {0, 1, ... N-1} at times {0, $\bullet t$, $2\bullet t$, $3\bullet t$, ... $(N-1)\bullet t$ } where the waveform is the sum of sines and cosines in time as,

$$\begin{aligned} S_k = A_0 &+ A_1 \cos(2\bullet k/N) + A_2 \cos(2*2\bullet k/N) + A_3 \cos(3*2\bullet k/N) + \ldots \\ &\quad \ldots + A_{N/2-1} \cos((N/2-1)*2\bullet k/N) + A_{N/2} \cos((N/2)*2\bullet k/N) \\ &+ B_1 \sin(2\bullet k/N) + B_2 \sin(2*2\bullet k/N) + B_3 \sin(3*2\bullet k/N) + \ldots \\ &\quad \ldots + B_{N/2-1} \sin((N/2-1)*2\bullet k/N) + B_{N/2} \sin((N/2)*2\bullet k/N) \end{aligned} \qquad \text{Eqn. 6}$$

or, more simply,

$$S_k = A_0 + \bullet [\, A_m \cos(2\bullet m k/N) + B_m \sin(2\bullet m k/N) \,] \qquad \text{Eqn. 7}$$

where the sum is taken for $m$=0 to N/2.

Note that in the above $2\bullet k\bullet t/T$ has been simplified as $2\bullet k\bullet t/T = 2\bullet k\bullet t/N\bullet t = 2\bullet k/N$. Executing Real.Integer.FFT generates a spectrum that consists of the coefficients in the following order:

$$F_n = \{A_0, A_{N/2}, A_1, B_1, A_2, B_2, A_3, B_3, \ldots A_{N/2-1}, B_{N/2-1}\} \qquad \text{Eqn. 8}$$

corresponding to array indices,

$$n = \{ 0, 1, 2, 3, 4, 5, 6, 7, \ldots N-2, N-1 \} \qquad \text{Eqn. 9}$$

and frequencies $f_m = m/(N\bullet t)$, where the harmonic number, $m$, ranges as,

$$m = \{ 0, N/2, 1, 1, 2, 2, 3, 3, \ldots (N/2)-1, (N/2)-1 \} \qquad \text{Eqn. 10}$$

Note that this spectrum does not contain both phases of the Nyquist critical frequency component; $B_{N/2}$ can not be determined because it is not actually represented in the discretely sampled waveform of only N points. That is, in the time domain equation above, $B_{N/2} \sin((N/2)*2\bullet k\bullet t/N\bullet t)$ always equals zero because $\sin((N/2)*2\bullet k\bullet t/N\bullet t) = \sin(n*\bullet) = 0$. Sampling a waveform every $\bullet t$ seconds starting at time zero can not capture any of the sine-phase energy at frequency $1/(2\bullet t)$. This is why we say that this highest frequency component is aliased to zero. The magnitude of the cosine-phase component is also aliased in the sense that it is sampled only at its peak magnitude excursions. Any frequencies of the waveform greater than $1/(2\bullet t)$, both cosine- and sine-phases, are aliased during sampling by being folded into (that is, increasing the magnitude of) lower frequency components. This is why the input waveform should be filtered to exclude frequencies greater than $1/(2\bullet t)$ Hz before it is discretely sampled. (See the discussion below on Aliasing.)

Converting the Magnitude Spectrum to a Power Density Spectrum

A common use for the FFT is to determine the amount of energy, $P_m$, in a waveform accounted for by a particular frequency, $f_m$, or band of frequencies. The waveform's energy as a function of frequency is called its *Power Density Spectrum*, or *Periodogram*.

In the *time* domain the power density of the waveform is estimated as the average of the squared values of the sampled waveform as,

$$P = ( S_0^2 + S_1^2 + ... S_{N-1}^2 ) / N \qquad \text{Eqn. 11}$$

Note that this is an estimate because we are using only a finite number of samples to calculate the power of a continuous function. If we were to find the power, *P*, of the next sequence of N samples we would obtain a different result. As the number of samples increases our estimate of the average power density, *P*, improves; its standard deviation decreases as $(1/N)^{1/2}$.

Note also that *P* is not actually the power but is only proportional to it. The actual power depends not only on the average squared *magnitude* but also on the *impedance* (or resistance) of the medium carrying the signal. For example, if our sampled waveform is a voltage and we actually need to know the electrical power then to compute it we also need to know either the circuit impedance or the current. However, it is conventional, if not technically correct, to call the averaged squared voltage the power since they are proportional. The rms average voltage value of the waveform is just the square root of this value.

$$V_{rms} = P^{1/2} = [( S_0^2 + S_1^2 + ... S_{N-1}^2 ) / N ]^{1/2} \qquad \text{Eqn. 12}$$

This same power is computed in the *frequency* domain as the sum of the squares of the Fourier coefficients (this equivalence is known as Parseval's theorem) as,

$$P = A_0^2 + A_{N/2}^2 + (A_1^2 + B_1^2 + A_2^2 + B_2^2 + ... + A_{N/2-1}^2 + B_{N/2-1}^2 ) / 2 \qquad \text{Eqn. 13}$$

Because this is the same power as that computed by summing the squares of the sampled voltage, the rms voltage can also be computed indirectly from the spectral coefficients as,

$$V_{rms} = P^{1/2} = [A_0^2 + A_{N/2}^2 + (A_1^2 + B_1^2 + A_2^2 + B_2^2 + ... + A_{N/2-1}^2 + B_{N/2-1}^2) / 2 ]^{1/2} \qquad \text{Eqn. 14}$$

Note that we take one half of the squares of each frequency component except the zero frequency and the maximum (or N/2) frequency, which don't have the factor of 1/2.
The *periodogram* is the set of N/2+1 coefficients for the power at each harmonic; it is given by,

$$P_0 = A_0^2 \quad \text{for } m = 0 \qquad \text{Eqn. 15}$$

$$P_m = ( A_m^2 + B_m^2 ) / 2 \quad \text{for } 0 < m < N/2$$

$$P_{N/2} = A_{N/2}^2 \quad \text{for } m = N/2$$

and this can be expressed as a fraction of the total power as $P_m/P$. The rms magnitude value of the *m*'th frequency component is just the square root of $P_m$ or,

$$V_{rms,m} = P_m^{1/2} \qquad \text{Eqn. 16}$$

This form, as rms magnitude, is the most generally useful form for the spectrum. To convert our FFT spectrum of N points, $F_n$, to an N/2+1 point rms magnitude spectrum, $V_{rms,m}$, of this form, we convert,

$$F_n = \{ A_0, A_{N/2}, A_1, B_1, A_2, B_2, A_3, B_3, \ldots A_{N/2-1}, B_{N/2-1} \} \quad \text{Eqn. 17}$$

into,

$$V_{rms,m} = \{ |A_0|, [(A_1^2+B_1^2)/2]^{1/2}, [(A_2^2+B_2^2)/2]^{1/2}, \ldots \quad \text{Eqn. 18}$$
$$\ldots, [(A_{N/2-1}^2+B_{N/2-1}^2)/2]^{1/2}, |A_{N/2}| \}$$

Notice that in $V_{rms,m}$ we have reordered the terms to place the highest frequency term last. A routine named >MAGNITUDE is provided to do this conversion for you, please see the files "v3.0 Fast Integer FFT" and "Using Integer FFTs". If the elements of $V_{rms,m}$ are squared we have the power density spectrum (or periodogram). Then, the elements corresponding to different frequency regions can simply be summed to find the spectral power in those regions.

For Experts: Details of the Algorithm

The most straightforward implementation of a Discrete Fourier Transform (DFT) generally takes of the order of $N^2$ computations. But if the number of samples is an integer power of two, then an algorithm called the Fast Fourier Transform (FFT) can be used which takes only of the order of N*log(N) computations. The algorithm we use is a decimation-in-time FFT as described in standard references. (See for example [1] and [2].) Note that these two references use two slightly different definitions of the Fourier Transform. *Digital...* [1] uses

$$X_f = \bullet x_t \exp[-2 \bullet i f t] \quad \text{Eqn. 19}$$

for the FFT where the sum is taken over the discrete times, *t*, and

$$x_t = (1/N) \bullet X_f \exp[+2 \bullet i f t] \quad \text{Eqn. 20}$$

for the inverse FFT where the sum is taken over the discrete frequencies, *f*, whereas *Numerical Recipes...* [2] uses

$$X_f = \bullet x_t \exp[+2 \bullet i f t] \quad \text{Eqn. 21}$$

for the FFT where the sum is taken over the times, *t*, and uses

$$x_t = (1/N) \bullet X_f \exp[-2 \bullet i f t] \quad \text{Eqn. 22}$$

for the inverse FFT where the sum is taken over the frequencies, *f*. These definitions differ in the sign of the argument of the complex exponential.

We follow the convention used by *Numerical Recipes...* [ref 2]. Among theorists it seems to be the lesser used convention, but among engineers it is more often used because when evaluating real functions the magnitudes of the real part of the transform can be directly taken to represent the cosine components of the waveform, and the magnitudes of the imaginary part of the transform represent the sine components. However, we depart from *Numerical Recipes...* convention in that we normalize (divide by N) when doing the FFT instead of when doing the IFFT. This makes better engineering sense in that it returns magnitudes that are already scaled to be independent of the length of the sampled waveform; and therefore produces power *densities* directly. It also makes it easier to keep the data array scaled within the magnitude limits of 16-bit integers.

These routines use the algorithm described in the *Numerical Recipes...* book for doing a Cooley-Tukey (or Danielson-Lanczos) decimation-in-time FFT. Our (complex) integer FFT routine implements the following transformations for the complex FFT and the complex inverse FFT (or IFFT):

FFT: $F_n = (1/N) \bullet S_k \exp[+2 \bullet i k n / N]$ where the sum is taken over $k = 0$ to $N-1$ Eqn. 23

IFFT: $S_k = \bullet F_n \exp[-2 \bullet i k n / N]$ where the sum is taken over $n = 0$ to $N-1$ Eqn. 24

where,

$S_k$ is a sequence of N complex-valued samples, for index $k = \{ 0, 1, 2, \ldots , N-1\}$, spaced equally in time (where $\bullet t$ is the time interval between successive samples), and

$F_n$ are complex-valued magnitudes, for index $n = \{ 0, 1, 2, \ldots , N-1\}$, of the frequency components in which the real part of the complex magnitude corresponds to the cosine-phase of the frequency component and the imaginary component of the magnitude corresponds to the sine-phase of the frequency component, the FFT being done in place so that $S_k$ is replaced with $F_n$.

For this complex FFT algorithm complex-valued magnitudes are found for both positive and negative frequency components. The frequencies, $f_m = m/(N \bullet t)$, that correspond to the $F_n$ are ordered as,

$n = \{ 0, 1, 2, \ldots N/2-1, N/2, N/2+1, \ldots N-2, N-1 \}$ Eqn. 25

$m = \{ 0, 1, 2, \ldots N/2-1, +/-N/2, -(N/2-1), \ldots -2, -1 \}$ Eqn. 26

The algorithm for transforming a real-valued waveform rather than a complex-valued waveform also uses Eqns. 23-24. For greater efficiency, this algorithm packs even and odd samples of the real-valued waveform into real and imaginary portions of a complex waveform of half the length, performs a complex FFT, and then sorts out the results. Consult the *Numerical Recipes...* book [2] for details of the algorithm. I have used their notation in the source code wherever possible. The routines that implement the real-valued FFT and IFFT use Eqns. 23-24, but in this case the $S_k$ and $F_n$ have the following meaning:

$S_k$ are N sequential real-valued samples, for index $k = \{ 0, 1, 2, \ldots , N-1 \}$, spaced equally in time (where $\bullet t$ is the time interval between successive samples);

$F_n$ are ordered as pairs of real-valued magnitudes, for index $n = \{ 0, 1, 2, \ldots , N-1 \}$, of the frequency components in which for all but the first pair the first element of each pair corresponds to the cosine phase of the frequency component and the second element corresponds to the sine phase of the frequency component, the FFT being done in place so that $S_k$ is replaced with $F_n$ ; and,

$f_m$ are the frequencies corresponding to the array of magnitudes $F_n$, where the frequencies are given by $f_m = m/(N \bullet t)$, and are ordered at the output of the routine as follows:

for $n = \{ 0, 1, 2, 3, 4, 5, 6, 7, \ldots N-2, N-1 \}$ Eqn. 27

$m = \{ 0, N/2, 1, 1, 2, 2, 3, 3, \ldots N/2-2, N/2-1 \}$ Eqn. 28

Avoiding Aliasing

In our discussion so far we have been assuming that the input waveform does not contain any signal energy at frequencies greater than the Nyquist critical frequency, $f_C = 1/(2 \bullet t)$, determined by the sampling interval, $\bullet t$. But what if the input waveform is not bandlimited to this range? In that case, through the act of discrete sampling, all the power that lies at greater frequencies is spuriously moved to lower frequencies. This is called *aliasing*. Energy at a frequency a little greater than the Nyquist cut-off, $f > f_C$, is *aliased* (falsely translated) or folded back into a new frequency, $f'$, less than the Nyquist cut-off, where $f' = 2fc - f$. This occurs because sine or cosine waves of any frequency components that differ by a multiple of $1/\bullet t$ give the same samples at intervals of $\bullet t$. Once aliasing has occurred there is no way to correct the resulting Fourier Transform.

To prevent aliasing, the input waveform must be bandlimited to frequencies less than the Nyquist cut-off *before* sampling, or the sampling rate must be chosen great enough so that the Nyquist frequency is greater than the highest frequency present in the waveform to be sampled.

For some signals aliasing may be detected by examining the resulting Fourier spectrum. If the magnitudes of the frequency components smoothly decrease as the Nyquest cut-off is approached, diminishing toward zero at the cut-off, then it is likely that the input signal was properly bandlimited. If however the magnitudes approach a finite value it can be assumed that components outside of the transform range have been folded back into the range.

Windowing and the Resolution of Power Density Estimation

An FFT provides a way of decomposing any waveform, whatever its spectral content in terms of number of frequencies and their magnitudes, into a fixed number of discrete frequencies that when combined, perfectly reproduce the sampled waveform *at the sample points*. So far we have assumed that our input waveform consists only of discrete frequency components that are all integer multiples of the inverse of the sample period, that is $f_m = m/(N \bullet t)$. In this case the magnitudes of those frequency components are sorted into discrete bins as shown in Eqns. 6-10. For this case too, the decomposition into frequency components is perfect in the sense that if the components are recombined the original waveform is reproduced at all points, not just the sampled points. Eqn. 7 can be used to reproduce the waveform at the discrete sampling times and, through interpolation, at any other times.

But what happens in the more general case of an input waveform that contains frequencies that are not integer multiples of the (often arbitrarily chosen) fundamental frequency? There are two implications of this, one pertaining to waveform reproduction and one to the estimation of power spectral density.

First, as far as faithful reproduction of the signal goes, the FFT is still perfectly suited for applications that require compression or reproduction of the sampled waveform because it finds a set of frequencies and their magnitudes that do faithfully reproduce the input waveform *at the sample points*. Eqn. 7 can be used to recompute these sample values from the spectral magnitudes exactly. However Eqn. 7 can not be used to interpolate for other times. Nevertheless, if the input waveform had been bandlimited to exclude all signal at frequencies greater than the Nyquist cut-off, $f_c$, then the input waveform is still completely determined from its samples and the following is an interpolation formula that does work:

$$S(t) = \bullet S_k \bullet t \quad \sin[2 \bullet f_c (t - k \bullet t)] \; / \; (\bullet (t - k \bullet t)) \qquad \text{Eqn. 29}$$

in which the $S_k$ are assumed to be periodic with period N and the sum is taken over all $k$, $-\bullet < k < +\bullet$. For good discussions of this interpolation consult the references [1-2].

The second implication concerns the estimation of power spectral density. For applications in which we need to estimate the amount of energy at various frequencies in a waveform, the process of discrete sampling and Fourier Transform does introduce errors. This process necessarily shifts the energy at the waveform's infinite number of continuous frequencies into a limited set of harmonics of the sampling fundamental.

For example, consider a signal that contains only a single pure sine wave at a frequency of 3.5 times the sampling fundamental, or $f = 3.5/(N \cdot t)$. The spectrum of this signal can not show the single frequency component because it can only represent frequencies that are integer multiples of the fundamental. Consequently, its energy is primarily shifted into spectral bins at the third and fourth harmonics (on each side of the "true" frequency), with some additional *leakage* (that is the technical term) into still lower and higher bins. The actual distribution of energy (or power) over the spectrum is given by a leakage function, *L(s)*, as a function of the frequency offset,

Hanning (cosine), Welch (parabolic), or Parzen (triangle) windows work about the same and are an improvement over boxcar windows. One window we often use at Mosaic is a computationally efficeint close approximation of the Hanning window, formed from piecewise quadratic sections. Of these windows the Hanning and Mosaic give the smallest leakage widths at the expense of the greatest FWHM. These various windows are defined as follows for $0 <= k < N$:

| | | | |
|---|---|---|---|
| Boxcar: | $w_k = 1$ | | Eqn. 32 |
| Parzen: | $w_k = 1 - \lvert (k - (N-1)/2)/((N+1)/2) \rvert$ | | Eqn. 33 |
| Welch: | $w_k = 1 - [ (k - (N-1)/2)/((N+1)/2) ]^2$ | | Eqn. 34 |
| Hanning: | $w_k = [ 1 - \cos\{2 \cdot k/(N-1)\} ] / 2$ | | Eqn. 35 |
| Mosaic: | $w_k = 2 ( k / (N/2) )^2$ | for $k <= N/4$ | Eqn. 36 |
| | $w_k = 1 - 2 ( (N/2 - k) / (N/2) )^2$ | for $N/4 <= k <= 3N/4$ | |
| | $w_k = 2 ( (N-k) / (N/2) )^2$ | for $k >= 3N/4$ | |

Whichever window is used, the input data is multiplied by the window point-by-point so that the FFT is taken of the product $w_k S_k$. In this case the power spectral density should be renormalized so that Eqn. 15 becomes,

$$P_0 = A_0^2 / W^2 \quad \text{for} \quad m = 0, \qquad \text{Eqn. 37}$$

$$P_m = ( A_m^2 + B_m^2 ) / 2W^2 \quad \text{for} \quad 0 < m < N/2, \quad \text{and,}$$

$$P_{N/2} = A_{N/2}^2 / W^2 \quad \text{for} \quad m = N/2$$

in which $W^2$ is defined as the per-point variance of the window function as,

$$W^2 = (1/N) \cdot w_k^2 \quad \text{for} \quad k = 0 \text{ to } N-1 \qquad \text{Eqn. 38}$$

If the window weights are normalized so that they have an rms magnitude of unity then $W^2$ is equal to one and we can still use Eqn. 15 directly.

The leakage function of Eqn. 30 holds true for a boxcar window function. If a window function other than the boxcar function is used the more general leakage function is given by the following,

$$L(s) = \lvert \cdot (w_k \exp[2 \cdot i s k / N]) \rvert^2 / (W^2 N^2) \quad \text{summed for } k = 0 \text{ to } N-1, \text{ or} \qquad \text{Eqn. 39}$$

$$L(s) \cdot \lvert \cdot w(k - N/2) \cos[2 \cdot s k / N] \, dk \rvert^2 / (W^2 N^2) \quad \text{integrated from } k = -N/2 \text{ to } N/2$$

in which the continuous function, $w(k-N/2)$, is meant to be any smooth continuous function that passes through the points $w_k$. The approximation is useful for practical estimates of leakage into nearby bins. It is valid for any window that is right/left symmetric (i.e., symmetric about $n = N/2$), and for $s << N$.

Spectral Averaging for Better Power Density Estimation

We might ask ourselves how accurate are each of the N/2+1 coefficients of the power density spectrum of Eqn. 15 or Eqn. 37 ? Or, equivalently, to what extent are these coefficients a true estimate of the power spectrum of the input waveform? Obviously a finite number of discrete samples will only approximate the power density spectrum of a continuous infinite waveform. If the waveform is stationary (its power spectrum does not vary with time), so that it can be characterized by a single power density spectrum, then the power density spectrum we measure depends on the finite number of samples we take. We might ask, what is the relationahip between the accuracy of our estimation and the number of samples taken? The answer turns out to be that each of the coefficients has a variance equal to the square of its expectation value; that is, the standard deviation is always 100 percent of the value, independent of N! If more points are sampled in the waveform (either by sampling at smaller intervals or for a greater total time) then more frequencies are added to the spectrum but the standard deviation of each frequency's power density coefficient remains equal to its value.

Even so, there are ways to get more accurate estimation, and they all involve averaging:

1. Smoothing the Spectrum: Sample over a longer time interval and compute a periodogram with finer frequency resultion than is actually needed. Then, instead of using all of the $P_m$, the periodogram can be divided into sections and the $P_m$ within each section summed together to get smoother estimates at the mid frequencies of each section. If $K$ frequency bins are summed then the variance of the summed estimate is smaller than the estimate itself by a factor of $1/K$; the standard deviation of that sum is smaller by a factor of $(1/K)^{1/2}$. Summing is used rather than averaging so that the sum of the elements of the resulting periodogram of greater granularity still equals the mean square value of the original time-domain waveform.

2. A second way to get a good estimate of the power spectral density is to segment the original data into $K$ segments, take the FFT of each segment, and average the resulting periodograms at each frequency. The averaging reduces the variance of the periodogram coefficients by a factor of $K$, and the standard deviation by a factor of $(1/K)^{1/2}$. As discussed above, leakage of energy from one frequency bin into surrounding bins depends on the shape window used to frame the $K$ data segments.

If implementing the second technique there are two ways the data can be segmented and windowed. The first is used if the goal is to obtain the smallest variance given a fixed amount of computation. In this case it is best to segment the data without any overlapping. For example the first $2M$ points are used for the first segment, the next $2M$ points for the second segment, up to $K$ segments for a total number of points of $N = 2KM$. Each $2M$-point segment is windowed, its FFT is taken, and an $M+1$ frequency periodogram computed. These $K$ periodograms are then averaged to produce a single $M+1$ frequency periodogram with a standard deviation on each power spectral coefficient of $(1/K)^{1/2}$ times its value. This is generally best if the data are being collected in real time, with the data-reduction being computer-limited.

A second goal might be to make the best use (in terms of reducing variance on the periodogram) of a limited amount of data. In that case it turns out to be best to overlap the data segments by one half of their length. Ther first and second sets of $M$ points are the first $2M$-point segment, the second and third sets of $M$ points are the second segment, up to the $K$'th and ($K$+1)'th set of $M$ points for the $K$'th segment. The total number of points is $N = (K+1)M$, a little over half as many as would be used if we did not overlap the segments. As with the first method, each $2M$-point segment is windowed, its FFT is taken, and an $M$+1 frequency periodogram computed. These $K$ periodograms are then averaged to produce a single $M$+1 frequency periodogram. In this case however the reduction in variance is not a factor of $K$ because the segments are not statistically independent. However, the variance is reduced nearly as much, by a factor of about $9K/11$. This is much better than the factor of about $K/2$ which would result if the first method (nonoverlapping data segments) were used on the same number of data points. The interested reader is referred to reference [2] from which this discussion was abstracted.

Useful References

[1] Digital Signal Processing, A.V. Oppenheim and R.W. Schafer, Prentice-Hall, Inc., 1975, pg 291-302

[2] Numerical Recipes in C, The Art of Scientific Computing, William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling, Cambridge University Press, 1988, Chapter 12

**MI-AN-055**
**Appendix A: Complex and Real Integer FFT Code**

```
\ ***************************************************************************
\ ***************                                         ***************
\ ***************            v3.0 Integer FFT Code        ***************
\ ***************                                         ***************
\ ***************************************************************************


\ Last Revision: 09/02/95  02:19:12 PM  Paul K. Clifford
\ ( I made many small changes to speed this up a bit.)
\ Last Revision: 03/18/96  02:14:26 PM  PKC
\   I tried storing bit-reversed address indices in a table rather than
\   computing them in real time but the very small improvement in speed
\   (appx 0.8%) didn't justify the table storage space on the common
\   page.
\
\ This file is provided by Mosaic Industries to those
\ wishing to perform faster FFTs on the QED Board.  Please see the
\ accompanying documentation file, "Using Integer FFTs", and read the
\ extensive description and comments that appear throughout this file.
\ If after that you have any questions about the use of these routines
\ feel free to call Mosaic Industries, Inc. at 510-790-1255.

\ The programs in this file will perform integer FFTs of real waveforms
\ on a 16 MHz QED Board in the following times:
\
\       Number of Points        Time
\              4                 0.45 millisecond
\              8                 1.2  millisecond
\             16                 3.4  millisecond
\             32                 9.25 millisecond
\             64                24.   millisecond
\            128                60.   millisecond
\            256              0.142    second
\            512              0.330    second
\           1024              0.754    second
\           2048              1.693    second
\
\ or, Time(msec) = 0.092 N (Log(N) - 2) + 0.45 msec.  where Log is done in
\ base 2.  These times include the timeslicer switch time taken by the
\ Benchmark: word.

ANEW <FOURIER>

\ This file contains code for an updated integer FFT for QED 3.x software.
\ The user available words are:
\   Initialize.Real.FFT     - Initializes a lookup table to enable doing real
\                             FFTs or IFFTs.  Only needs to be called initially
\                             and whenever the size of the FFT is changed.  For
\                             fixed sized FFTs Initialize.Real.FFT only needs to
\                             be called once at compile time to initialize a
\                             table in ROM (on the same page as this code).  If
\                             this is done, the variables, Table.Address and
\                             Number.of.Complex.Points, should be left as
\                             variables but initialized and compiled into
\                             ROM instead of variable space, or converted to
\                             constants.
```

```
\   Initialize.Complex.FFT  - Just like Initialize.Real.FFT but for doing FFTs
\                             or IFFTs of complex waveforms.
\
\   Complex.Integer.FFT     - Performs an in-place FFT of a complex integer
\                             waveform.  The waveform must comprise signed
\                             16-bit integers of absolute magnitude less than
\                             2^14.
\
\   Complex.Integer.IFFT    - Does the inverse FFT.  If the spectrum resulted from
\                             a waveform of the right magnitude ( <2^14 ) then
\                             there will be no overflow when the IFFT is taken.
\                             If there is any doubt that the waveform might overflow
\                             then scale the spectrum towards smaller values before
\                             calling Complex.Integer.IFFT and/or check the result
\                             by taking its FFT.
\
\   Real.Integer.FFT        - Performs an in-place FFT of a real integer
\                             waveform.  The waveform must comprise signed
\                             16-bit integers of absolute magnitude less than
\                             2^14.
\
\   Real.Integer.IFFT       - Does the inverse FFT.  Input must be scaled so that
\                             all values are less than |2^14| and so that the peak
\                             values of the resulting waveform are also less
\                             than |2^14|.
\
\   >Magnitude              - Converts the integer spectrum created by the
\                             routine Real.Integer.FFT into an rms magnitude
\                             spectrum stored as a floating point matrix.  This
\                             is the most useful form of the spectrum for graphing
\                             and for computing the power in different
\                             frequency regions.


\ This program operates on a data array in the common ram, and refers to it
\ by its starting address rather than by treating it like an array or matrix.
\ Also, another data array, a read-only look-up table containing the first
\ quadrant of the cosine function must be on the same page as this code, or
\ in common ram, so that we can address it without a page number.

\ The discrete Fourier transform generally takes of the order of
\ N^2 computations.  But if the number of samples is an integer
\ power of two, then a Fast Fourier Transform can be used which
\ takes only of the order of N*log(N) computations.  This program
\ is a decimation-in-time FFT as described in standard references.
\ (See for example Digital Signal Processing, A.V.Oppenheim and
\ R.W.Schafer, Prentice-Hall, Inc., 1975, pg 291-302, or, "Numerical
\ Recipes in C, The Art of Scientific Computing", William H. Press,
\ Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling,
\ Cambridge University Press, 1988, Chapter 12.)

\ These routines use the algorithm described in the "Numerical Recipes..."
\ book for doing a Cooley-Tukey (or Danielson-Lanczos) decimation-in-time FFT.
\ The algorithm for doing the FFT of a single real function packs even and odd
\ samples into real and imaginary portions of a complex waveform of half the
\ length, does a complex FFT, and then sorts out the results.  Consult the
\ "Numerical Recipes..." book for details of the algorithm.  I have preserved
\ their notation in the source code wherever possible.
```

```
\ Note that these two references use two different definitions of the Fourier
\ Transform. "Digital..." uses X(f) =   x(t)exp[-2PIift/N] and "Numerical..."
\ uses X(f) =   x(t)exp[+2PIift/N].  We follow the convention used by
\ "Numerical...", even though it seems to be the lesser used convention,
\ because when evaluating real functions the magnitudes of the complex
\ part of the transform can be directly taken to represent the sine
\ components of the waveform.  However, we depart from "Numerical"'s convention
\ in that we normalize (divide by N) when doing the FFT instead of when
\ doing the IFFT.



\ The following program, Complex.Integer.FFT, defines the Discrete Fourier
\ Transform (DFT) for complex numbers as,
\
\   F(n) = (1/N) *   S(k)exp[+2*PI*ikn/N]
\
\ where
\
\   the sum is taken over k=0 to N-1,
\
\   S(k) for k = { 0, 1, 2, ... , N-1 } is a sequence of N complex input
\       samples spaced equally in time ( where t is the time interval between
\       successive samples),
\
\   F(n) for n = { 0, 1, 2, ... , N-1 } are complex magnitudes of the frequency
\       components, for frequencies of 0, +/- 1/Nt, +/- 2/Nt, ... to +/- 1/2t,
\
\   f(n), the frequencies corresponding to the array of magnitudes F(n),
\       are ordered as
\           f(n) = { 0, 1, 2, ... N/2-1, +/-N/2, -(N/2-1), ...  -2,  -1 }*(1/Nt)
\         for n  = { 0, 1, 2, ... N/2-1,    N/2,   N/2+1 , ... N-2, N-1 }
\
\   and the DFT is done in place so that S(k) is replaced with F(n).
\
\ The computation is done in-place and the format of the data and resulting
\ spectrum is the standard described in "Numerical Recipes...", and in the
\ QED Software Manual, Chapter 7.

\ The inverse DFT follows the same conventions as the forward DFT and is defined
\ as,
\
\   S(k) =   F(n)exp[-2PIink/N]
\
\ where
\   the sum is taken over n=0 to N-1.
\
\ Note that both the input samples, S(k), and the magnitudes of the frequencies,
\ F(n), are periodic with a period of N.  Therefore F(n) = F(n+N), S(k) = S(k+N),
\ and f(n) = f(n+N) = n, so that in the exponential's arguments of above
\ definitions the 2PIink/N is equivalent to 2PIif(n)k/N.
```

```
\ The sign of the argument of the exp[] that I chose is consistent in the sign
\ of the argument used in the Numerical Recipes book. The QED 2.x software used
\ a definition with the opposite signs for the FFT and the IFFT to be
\ consistent with the "Digital Signal Processing" and other books.
\ Unlike the algorithms described in either of these books, and unlike the
\ QED v2.0 kernel routines FFT and IFFT, we normalize by dividing by N when doing
\ the FFT rather than the IFFT because that gives us better use of the limited
\ magnitude range of the 16-bit integers we use, and because it makes better
\ engineering sense (the magnitudes of the sine and cosine components that are
\ computed correspond directly to the magnitude of the input waveform; see the
\ example in the corresponding documentation file).
\
\ When doing FFTs, the peak excursions of the input waveform must be less than
\ +/- 2^14 to prevent internal overflows in the routines.


DECIMAL  \ Must be DECIMAL !

7 WIDTH !

CODE (M*) ( n1\n1 -- d )
    \ a somewhat faster version of M* than the kernel's version
    00 IND,Y LDD PSHA   \ save first operand's sign on rtn stack
    MI IF,              \ two's complement if needed to get abs value of operand
        COMA COMB 01 IMM ADDD 00 IND,Y STD
    ENDIF,
    PULA 02 IND,Y EORA PSHA \ find and save the sign of the result
    02 IND,Y LDD
    MI IF,              \ two's complement if needed to get abs value of operand
        COMA COMB 01 IMM ADDD 02 IND,Y STD
    ENDIF,
    \ do the four part multiply with partial sums:
    01 IND,Y LDD   PSHB
    MUL  XGDX           00 IND,Y LDAA  PSHA  03 IND,Y LDAB
    MUL  ABX      PSHA  01 IND,Y LDAB        03 IND,Y LDAA
    MUL  03 IND,Y STAB        TAB ABX        01 IND,Y STX  PULB PULX XGDX
    MUL  XGDX ABX       01 IND,Y LDAB  ABX   00 IND,Y STX
    PULA TSTA           \ get the sign of the result
    MI IF,              \ if negative, two's complement the result
        02 IND,Y LDD COMA COMB 01 IMM ADDD \ complement the lower word
        02 IND,Y STD    \ and store it
        00 IND,Y LDD
        CS IF,          \ complement higher word, adding one if a one was carried
            COMA COMB 01 IMM ADDD \ out of the lower word.
        ELSE,
            COMA COMB
        ENDIF,
        00 IND,Y STD    \ store higher word of result
    ENDIF,
    RTS
    END.CODE
```

```
CODE D-.1.DSCALE.NIP
    \ Does a D- then divides the result by 2^15 leaving a signed 16-bit integer
    \ Does not check for overflow.
    06 IND,Y LDD  02 IND,Y SUBD  06 IND,Y STD
    04 IND,Y LDD  CS IF,  01 IMM SUBD  ENDIF,
    00 IND,Y SUBD  04 IND,Y STD
    06 IND,Y LDAA ROLA  04 IND,Y LDD  ROLB ROLA
    06 IND,Y STD
    06 IMM LDAB ABY RTS
    END.CODE

CODE D+.1.DSCALE.NIP
    \ Does a D+ then divides the result by 2^15 leaving a signed 16-bit integer
    \ Does not check for overflow.
    02 IND,Y LDD  06 IND,Y ADDD  06 IND,Y STD
    04 IND,Y LDD  CS IF, 01 IMM ADDD ENDIF,
    00 IND,Y ADDD  04 IND,Y STD
    06 IND,Y LDAA ROLA  04 IND,Y LDD  ROLB ROLA
    06 IND,Y STD
    06 IMM LDAB  ABY  RTS
    END.CODE

\ : BIT-REVERSE ( N \ Data.addr -- )
\   \ This is the high level definition of Bit-Reverse for documentation
\   \ purposes only.
\   \ Data.addr is the base address of the complex integer array in common
\   \ memory (or on the same page as these routines).  N is the number of
\   \ complex points in the spectrum.
\   LOCALS{ &data.vector &N | &N-1 &N/2 &A &B &C }
\   0 TO &B     &N 2/ TO &N/2
\   0 TO &C     &N 1- TO &N-1
\   BEGIN   \ Written as a BEGIN loop instead of a DO or FOR loop for ease in
\           \ converting to assembly code.
\       &B I >  \ Swap the numbers whenever the bit-reversed counter exceeds the
\               \ straight counter.
\        IF &data.vector &B 4* +
\           &data.vector  I 4* + 2@SWAP2!
\           \ 2@SWAP2! takes two addresses and swaps the double number contents
\           \ of those addresses.
\       ENDIF
\       \ The following BEGIN/WHILE/REPEAT loop executes an algorithm that
\       \ efficiently counts in bit-reversal order.  The variable &B is the
\       \ bit-reversed counter.  For example, for N=8, &B counts as 0,4,2,6,
\       \ 1,5,3,7.
\       &N/2 TO &A
\       BEGIN
\           &A &B > NOT
\       WHILE
\           &B &A - TO &B   &A 2/ TO &A
\       REPEAT
\       &B &A + TO &B       &C 1+ TO &C
\       &C &N-1 =
\   UNTIL
\   ;
```

```
CODE BIT-REVERSE ( N \ Data.addr -- )
    \ This is the assembly language version of the above high-level program.
    \ It's almost a one-for-one translation of that program.  A stack frame on
    \ the data stack holds the following variables, referenced to the Y register
    \ as:
    \
    \   00 Temporary storage for SWAPs
    \   02 Swap address#1
    \   04 Swap address#2
    \   06 C          - loop counter that steps through data
    \   08 B          - bit-reversed counter
    \   10 A          - used in bit-reversal algorithm
    \   12 N/2        - used in bit-reversal algorithm
    \   14 Data.addr - base of data array
    \   16 N-1        - N is the number of complex elements in the array
    \
    PSHY PULA PULB 14 IMM SUBD XGDY \ Create stack frame for 7 more stack items
    PSHY PULX                       \ We'll index with X because it's faster
    16 IND,X LDD ASRA RORB 12 IND,X STD      \ N/2
    16 IND,X LDD 0001 IMM SUBD  16 IND,X STD \ N-1
    0000 IMM LDD  06 IND,X STD  08 IND,X STD \ C=0, B=0

    BEGIN,  \ Loop through the addresses of the data.array
        08 IND,X LDD 06 IND,X SUBD          \ B-C
        HI                                  \ U>
        IF, \ If bit-reversed counter is greater than array index then swap

            \ Addr1 =  4*B + Data.addr :
                08 IND,X LDD   ASLD ASLD   14 IND,X ADDD   02 IND,X STD

            \ Addr2 =  4*C + Data.addr :
                06 IND,X LDD   ASLD ASLD   14 IND,X ADDD   04 IND,X STD

            \ Swap the double numbers pointed to by Addr1 and Addr2:
                02 IND,Y LDX  00 IND,X LDD  00 IND,Y STD
                04 IND,Y LDX  00 IND,X LDD  02 IND,Y LDX  00 IND,X STD
                00 IND,Y LDD  04 IND,Y LDX  00 IND,X STD
                02 IND,Y LDX  02 IND,X LDD  00 IND,Y STD
                04 IND,Y LDX  02 IND,X LDD  02 IND,Y LDX  02 IND,X STD
                00 IND,Y LDD  04 IND,Y LDX  02 IND,X STD
                PSHY PULX   \ restore stack pointer into X
        ENDIF,
        12 IND,X LDD 10 IND,X STD               \ A=N/2

        BEGIN,
            08 IND,X LDD 10 IND,X SUBD          \ B-A
            HS                                  \ While B >= A
        WHILE,
            08 IND,X STD                        \ B=B-A
            10 IND,X LDD ASRA RORB 10 IND,X STD \ A=A/2
        REPEAT,

        08 IND,X LDD 10 IND,X ADDD 08 IND,X STD \ B=B+A
        06 IND,X LDD 0001 IMM ADDD 06 IND,X STD \ C=C+1
        16 IND,X SUBD EQ                        \ C=N-1 ?
    UNTIL,              \ Finished when we've stepped through the array

    18 IMM LDAB ABY     \ Drop the stack frame of 9 stack cells
    RTS                 \ Return with the array in bit-reversed order.
END.CODE
```

```
\ The following butterfly code, used as,
\           &A1 &A2 &R* &I* &FLAG BUTTERFLY
\ is equivalent to:
\ &FLAG
\ IF
\   &A1    (@) &R* +    &A2    (!)
\   &A1 2+ (@) &I* -    &A2 2+ (!)
\   &A1 2+ (@) &I* +    &A1 2+ (!)
\   &A1    (@) &R* -    &A1    (!)
\ ELSE
\   &A1    (@) &R* + 2/ &A2    (!)
\   &A1 2+ (@) &I* - 2/ &A2 2+ (!)
\   &A1 2+ (@) &I* + 2/ &A1 2+ (!)
\   &A1    (@) &R* - 2/ &A1    (!)
\ ENDIF

CODE BUTTERFLY ( &A1\&A2\&R*\&I*\&Flag -- &A1\&A2\&R*\&I*\&Flag )
    \ Leaves everything on the stack on exit.
    08 IND,Y LDX                    \   &A1 to X
    00 IND,Y LDAA                   \   &FLAG IF
    NE IF,
        00 IND,X LDD  04 IND,Y ADDD \   &A1 (@) &R* +
        06 IND,Y LDX  00 IND,X STD  \   &A2 (!)
        08 IND,Y LDX  02 IND,X LDD  \   &A1 2+ (@)
        02 IND,Y SUBD               \   &I* -
        06 IND,Y LDX  02 IND,X STD  \   &A2 2+ (!)
        08 IND,Y LDX  02 IND,X LDD  \   &A1 2+ (@)
        02 IND,Y ADDD               \   &I* +
        02 IND,X STD  00 IND,X LDD  \   &A1 2+ (!) &A1 (@)
        04 IND,Y SUBD 00 IND,X STD  \   &R* -  &A1 (!)
    ELSE,                           \   ELSE
        00 IND,X LDD                \   &A1 (@)
        04 IND,Y ADDD ASRA RORB     \   &R* + 2/
        06 IND,Y LDX  00 IND,X STD  \   &A2 (!)
        08 IND,Y LDX  02 IND,X LDD  \   &A1 2+ (@)
        02 IND,Y SUBD ASRA RORB     \   &I* - 2/
        06 IND,Y LDX  02 IND,X STD  \   &A2 2+ (!)
        08 IND,Y LDX  02 IND,X LDD  \   &A1 2+ (@)
        02 IND,Y ADDD ASRA RORB     \   &I* + 2/
        02 IND,X STD  00 IND,X LDD  \   &A1 2+ (!)  &A1 (@)
        04 IND,Y SUBD ASRA RORB     \   &R* - 2/
        00 IND,X STD                \   &A1 (!)
    ENDIF,                          \   ENDIF
    RTS
END.CODE
```

```
CODE FIRST.POINT ( stack.frame... -- stack.frame... )
    \ Butterfly computations that use SINEs and COSINEs of 0 or 90 degrees, and
    \ for which the multiplies by 0.0 and 1.0 can be replaced with additions,
    \ are done separately for speed.
    22 IND,X LDD 08 IND,X STD                    \ ADDR.COUNTER @ A1 !
    BEGIN,                        \ step through addresses in increments of 2^(I+3) @
        08 IND,X LDD 16 IND,X ADDD 06 IND,X STD \ A1 @ OUTER.COUNTER @ + A2 !
        XGDX 00 IND,X LDD                        \ A2 @ (@)
        COMA COMB 01 IMM ADDD                    \ NEGATE
        04 IND,Y STD  02 IND,X LDD 02 IND,Y STD \ R* !  A2 @ (2+@)  I* !
        CALL BUTTERFLY                           \ Do the butterfly computation
        PSHY PULX                                \ Put data stack pointer into X
        08 IND,X LDD 18 IND,X ADDD 08 IND,X STD \ 2^(I+3) @ A1 +!
        32 IND,X SUBD                            \ 4(N-1)+Data.Addr @ -
        HS \ U>=
    UNTIL,
    RTS
END.CODE
CODE MIDDLE.POINT ( stack.frame... -- stack.frame... )
    22 IND,X LDD 08 IND,X STD                \ ADDR.COUNTER @ A1 !
    BEGIN,      \ Step through addresses in increments of 2^(I+3) @
        08 IND,X LDD 16 IND,X ADDD 06 IND,X STD \ A1 @ OUTER.COUNTER @ + A2 !
        00 IND,X LDAA NE                         \ Flag @ 0 <>
        IF,
            \ The IF clause is equivalent to the high level line:
            \ A1 @ A2 @ A2 @ (2+@) NEGATE A2 @ (@) NEGATE FLAG @ BUTTERFLY
            06 IND,X LDD XGDX 02 IND,X LDD  \ A2 @ (2+@)
            COMA COMB 01 IMM ADDD           \ NEGATE
            04 IND,Y STD                    \ R* !
            00 IND,X LDD                    \ A2 @ (@)
            COMA COMB 01 IMM ADDD           \ NEGATE
            02 IND,Y STD                    \ I* !
            CALL BUTTERFLY                  \ Do the butterfly computation
        ELSE,
            \ The ELSE clause is equivalent to the high level line:
            \ A1 @ A2 @ A2 @ (2+@) A2 @ (@) FLAG @ BUTTERFLY
            06 IND,X LDD XGDX 02 IND,X LDD  \ A2 @ (2+@)
            04 IND,Y STD                    \ R* !
            00 IND,X LDD                    \ A2 @ (@)
            02 IND,Y STD                    \ I* !
            CALL BUTTERFLY                  \ Do the butterfly computation
        ENDIF,
        PSHY PULX
        08 IND,X LDD 18 IND,X ADDD 08 IND,X STD \ 2^(I+3) @ A1 +!
        32 IND,X SUBD                           \ 4(N-1)+Data.Addr @ -
        HS                                      \ U>=
    UNTIL,
    RTS
END.CODE

CODE TABLE.LOOKUP ( stack.frame... -- stack.frame... )
    \ The table holds the first quadrant of a cosine waveform.  COS is taken by
    \ addressing it from the beginning and SINE is taken by addressing it in
    \ reverse order, from the end.  The second quadrant of the COS is also taken
    \ by addressing it from the end, and NEGATEing the result.  Stores the COS
    \ in UR and the SINE in UI.
    38 IND,X LDD                    \ Number.of.Complex.Points @
```

```
26 IND,X SUBD                    \ Table.Pointer @ -
    HI                           \ U>
    IF,                          \ Fetch cosine (first quadrant)
        40 IND,X LDD 26 IND,X ADDD  \ Table.address @ Table.Pointer @ +
        XGDX 00 IND,X LDD        \ (@)
        PSHY PULX 12 IND,X STD   \ UR !
        26 IND,X LDD 28 IND,X STD   \ Table.Pointer @ Reflected.Table.Pointer !
    ELSE,                        \ Fetch second quadrant of cosine
        38 IND,X LDD ASLD        \ Number.of.Complex.Points @ 2*
        26 IND,X SUBD            \ Table.Pointer @ -
        28 IND,X STD             \ Reflected.Table.Pointer !
                                 \ Reflected.Table.Pointer @
        40 IND,X ADDD            \ Table.address @ +
        XGDX 00 IND,X LDD        \ (@)
        COMA COMB 01 IMM ADDD    \ NEGATE
        PSHY PULX 12 IND,X STD   \ UR !
    ENDIF,
    \ Fetch sine; for IFFT use complex conjugate.
    \ Address the table backwards to get the sine value.
    \ Table contains 1/4 wave.
    24 IND,X LDD 28 IND,X SUBD   \ Table.Top.addr @ Reflected.Table.Pointer @ -
    XGDX 00 IND,X LDD PSHY PULX  \ (@)
    00 IND,X TST                 \ FLAG @
    NE IF,                       \ 0<> IF
        COMA COMB 01 IMM ADDD    \ NEGATE
    ENDIF,
    10 IND,X STD                 \ UI !
    RTS
END.CODE

CODE INNER.LOOP ( 21 stack items ... -- 21 stack items ... )
        \ A1 @ OUTER.COUNTER @ + A2 !
        08 IND,X LDD 16 IND,X ADDD 06 IND,X STD
        02 IMM ADDD              \ A2 @ 2+
        XGDX 00 IND,X LDD        \ (@)
        DEY DEY 00 IND,Y STD     \ push
        12 IND,Y LDD             \ UI @
        DEY DEY 00 IND,Y STD     \ push
        CALL (M*)                \ multiply
        10 IND,Y LDX             \ A2 @
        00 IND,X LDD             \ (@)
        DEY DEY 00 IND,Y STD     \ push
        18 IND,Y LDD             \ UR @
        DEY DEY 00 IND,Y STD     \ push
        CALL (M*) CALL D-.1.DSCALE.NIP \ multiply and scale
        00 IND,Y LDD 06 IND,Y STD   \ R* !
        08 IND,Y LDD 02 IMM ADDD    \ A2 @ 2+
        XGDX 00 IND,X LDD        \ (@)
        00 IND,Y STD             \ push
        14 IND,Y LDD             \ UR @
        DEY DEY 00 IND,Y STD     \ push
        CALL (M*)                \ multiply
        10 IND,Y LDX             \ A2 @
        00 IND,X LDD             \ (@)
        DEY DEY 00 IND,Y STD     \ push
        16 IND,Y LDD             \ UI @
        DEY DEY 00 IND,Y STD     \ push
```

```
CALL (M*) CALL D+.1.DSCALE.NIP \ multiply and scale
        00 IND,Y LDD 04 IND,Y STD   \ I* !
        02 IMM LDAB ABY             \ pop
        CALL BUTTERFLY              \ do the butterfly computation
        PSHY PULX
    RTS
END.CODE

\ The word FFT-Kernel keeps all its local variables in a stack frame on the
\ parameter stack.  The variables are referenced by offsets to the Y stack
\ pointer as:
\   00 Flag                    02 I*                        04 R*
\   06 A2                      08 A1                        10 UI
\   12 UR                      14 OUTER.COUNTER.LIMIT       16 OUTER.COUNTER
\   18 2^(I+3)                 20 ADDR.COUNTER.LIMIT        22 ADDR.COUNTER
\   24 Table.Top.addr          26 Table.Pointer           28 Reflected.Table.Pointer
\   30 Table.Pointer.Increment 32 4(N-1)+Data.Addr          34 Flag
\   36 Data.Addr               38 Number.of.Complex.Points 40 Table.Address

CODE FFT-KERNEL ( Table.address\#Complex.Points\Data.vector.addr\IFFT.flag -- )
    PSHY PULA PULB 34 IMM SUBD XGDY \ Create stack frame for 17 more stack items
    PSHY PULX                   \ We'll index with X because it's faster
    38 IND,X LDD                \ Number.of.Complex.Points @
    0001 IMM SUBD  ASLD ASLD    \ 1- 4*
    36 IND,X ADDD  32 IND,X STD \ Data.Addr @ + 4(N-1)+Data.Addr !
    38 IND,X LDD                \ Number.of.Complex.Points @
    40 IND,X ADDD  24 IND,X STD \ Table.address @ +  Table.Top.addr !
    38 IND,X LDD   ASLD         \ Number.of.Complex.Points @  2*
    14 IND,X STD   30 IND,X STD \ DUP OUTER.COUNTER.LIMIT !
                                \ Table.Pointer.Increment !
    04 IMM LDD     16 IND,X STD \ 4 OUTER.COUNTER !
    08 IMM LDD     18 IND,X STD \ 2^(I+3) !
    34 IND,X LDAA   00 IND,X STAA \ transfer Flag to tos
    BEGIN,  \ loops LOG(N) times by doubling loop increment each iteration
        26 IND,X CLR    27 IND,X CLR        \ 0 Table.Pointer !
        36 IND,X LDD    22 IND,X STD        \ Data.Addr @ ADDR.COUNTER !
        16 IND,X ADDD   20 IND,X STD        \ OUTER.COUNTER @ + ADDR.COUNTER.LIMIT !
        BEGIN,                              \ loop on data array addresses
            36 IND,X LDD 22 IND,X SUBD EQ   \ Data.Addr @ ADDR.COUNTER @  =
            IF,                             \ for the first point in the array
                CALL FIRST.POINT            \ optimized for COS(0)=1.0 and
                                            \ SIN(0)=0.0
            ELSE,
                16 IND,X LDD ASRA RORB      \ OUTER.COUNTER @ 2/
                36 IND,X ADDD               \ Data.Addr @  +
                22 IND,X SUBD EQ            \ ADDR.COUNTER @ =
                IF,                         \ optimized for COS(90)=0.0
                                            \ and SIN(90)=1.0
                    CALL MIDDLE.POINT
```

```
ELSE,
                    \ For values other than COS and SIN of zero or ninety degrees
                    \ we'll get the cosine and sine factors from the table.
                    CALL TABLE.LOOKUP
                    22 IND,X LDD 08 IND,X STD    \ ADDR.COUNTER @ A1 !
                    BEGIN,                       \ Do the butterfly computation:
                        CALL INNER.LOOP
                        08 IND,X LDD 18 IND,X ADDD 08 IND,X STD \ 2^(I+3) @ A1 +!
                        32 IND,X SUBD            \ 4(N-1)+Data.Addr @ -
                        HS                       \ U>=
                    UNTIL,
                ENDIF,
            ENDIF,
            26 IND,X LDD  30 IND,X ADDD \ Table.Pointer @
                                        \ Table.Pointer.Increment @ +
            26 IND,X STD                \ Table.Pointer !
            22 IND,X LDD  04 IMM ADDD   22 IND,X STD \ 4 ADDR.COUNTER +!
            20 IND,X SUBD               \ ADDR.COUNTER @ ADDR.COUNTER.LIMIT @ -
            HS                          \ U>=
        UNTIL,
        30 IND,X LDD  ASRA RORB         \ Table.Pointer.Increment @ 2/
        30 IND,X STD                    \ Table.Pointer.Increment !
        16 IND,X LDD  ASLD              \ OUTER.COUNTER @ 2* \ double loop increment
        16 IND,X STD  ASLD              \ DUP OUTER.COUNTER ! 2*
        18 IND,X STD                    \ 2^(I+3) !
        16 IND,X LDD  14 IND,X SUBD     \ OUTER.COUNTER @ OUTER.COUNTER.LIMIT @ -
        HI                              \ U>
    UNTIL,
    42 IMM LDAB ABY                     \ drop 21 stack cells to return
    RTS
END.CODE

: (FFT) ( Data.addr \ Table.addr \ N \ flag -- )
    \ flag is true for IFFT and false for FFT
    LOCALS{ &IFFT.flag &N &Table.addr &Data.vector.addr }
    &N &Data.vector.addr BIT-REVERSE
    &Table.addr &N &Data.vector.addr &IFFT.flag FFT-KERNEL
    ;

: Complex.Integer.FFT   ( Data.addr \ Table.addr \ N -- )
    \ N is the number of samples in the Data array
    \ and Data.addr is its starting address in common memory
    \ and Table.addr is the address of the lookup table.
    FALSE (FFT) ;

: Complex.Integer.IFFT  ( Data.addr \ Table.addr \ N -- )
    \ N is the number of samples in the Data array
    \ and Data.addr is its starting address in common memory
    \ and Table.addr is the address of the lookup table.
    TRUE (FFT) ;
```

```
\ *********    Real.Integer.FFT and Real.Integer.IFFT ***************
\
\ Finds the fourier transform of a real integer array of 1 dimension.
\ The array must be organized along a single index and contain at least
\ 4 elements, and have a number of elements equal to a power of 2.
\
\ THERE IS AN UNCHECKED ERROR IF THE NUMBER OF ELEMENTS IS LESS THAN 4
\ OR NOT A POWER OF TWO!
\
\ If you need to transform a smaller array use the complex FFT routine
\ with the imaginary part of the waveform packed with zeros. If the
\ number of elements in the array is not a power of two there is an
\ unchecked error.
\
\ We'll use the "Numerical Recipes" convention for the forward FFT,
\ except that when we do the FFT we'll normalize by dividing by N.
\ ("Numerical Recipies" divides by N for the IFFT not the FFT.)
\ For the FFT we implement the following:
\
\   F(n) = (1/N) *   S(k)exp[+2PIikn/N]
\
\ where
\   the sum is taken over k=0 to N-1,
\   S(k) for k = { 0, 1, 2, ... , N-1 } is a sequence of N real input
\       samples spaced equally in time ( where t is the time interval between
\       successive samples),
\   F(n) for n = { 0, 1, 2, ... , N-1 } are real magnitudes (COSINEs)
\       of the frequency components,
\   f(n), the frequencies corresponding to the array of magnitudes F(n),
\       are ordered as
\           f(n) = { 0, N/2, 1, i, 2, 2i, ..., (N/2-1), (N/2-1)i }*(1/t)
\         for n  = { 0, 1,   2, 3, 4,  5, ...,     N-2,      N-1 }
\       where real frequencies represent COSINE components and imaginary
\       frequencies represent SINE components,
\   and the DFT is done in place so that S(k) is replaced with F(n),

\ The result is in spectral format in pairs of cosine and sine components,
\ except for the first pair. The first pair contains the magnitudes of the
\ zero frequency (DC) component, and the (aliased) highest frequency
\ component (as a real, or cosine, frequency).  The second pair contains
\ the first frequency (the fundamental or harmonic #1), the next pair the
\ second frequency (harmonic #2) and so on.  For all pairs but the first
\ the first elements of the pair contain the cosine components and the
\ second elements contain the sine components.
\
\ If flag is true we do a real FFT, if false we do a real Inverse FFT.
\ If we are doing an inverse FFT the output is presented as a single
\ column vector.
\

\ The following is a high level version of Real.Integer.FFT.Kernel provided so
\ that it may be maintained more easily.
```

```
\ : Real.Integer.FFT.Kernel
\   ( Data.addr \ Table.address \ Number.of.Complex.Points \ Flag -- )
\   LOCALS{ &FFT?   &N      &Table.addr &Data.addr |
\           &I1     &I2     &I3         &I4
\           &WR     &WI     &WR.addr    &WI.addr
\           &H1R    &H1I    &H2R        &H2I        }
\   &FFT?
\   IF      \ if we're doing forward FFT
\      &Data.addr Complex.Integer.FFT      \ Do the FFT
\   ENDIF
\
\   &Table.addr      TO &WR.addr    \ We'll step these addresses
\   &Table.addr &N + TO &WI.addr    \ through the look-up table.
\
\   \ Initialize the addresses into the array:
\      &Data.addr            4+ DUP TO &I1 2+ TO &I2
\      &Data.addr &N 2* 1- 2* + DUP TO &I4 2- TO &I3
\
\   \ Step through the data array:
\   &N 2 >       \ For the case &N=2, or 4 points, we don't wany to execute
\                \ the FOR loop 65536 times !  We want to execute it zero
\                \ times.
\   IF
\   &N 2/ 2-     \ N/2-1 iterations
\   FOR          \ Case for elements 0 and 1 is done separately below.
\   \ The xaddresses step through the matrix as though it is a 2N-element
\   \ linear array with element addresses of { 0, 1, 2, ... 2N-2, and 2N-1 }.
\   \ I1 steps through array indices = {    2,    4,    6, ...   N-4, N-2 }
\   \ I2 steps through array indices = {    3,    5,    7, ...   N-3, N-1 }
\   \ I3 steps through array indices = { 2N-2, 2N-4, 2N-6, ...   N+4, N+2 }
\   \ I4 steps through array indices = { 2N-1, 2N-3, 2N-5, ...   N+5, N+3 }
\   \ The I1-I4 notation preserves the variable names from the listing
\   \ in the Numerical Recipes book.
\      \ Increment the look-up table addresses:
\      &WR.addr 2+ DUP TO &WR.addr  &WI.addr 2- DUP TO &WI.addr
\      \ and fetch the cosine (real) and sine (imaginary) factors:
\      (@) &FFT? NOT.IF.NEGATE.ENDIF   TO &WI
\      (@)                             TO &WR
\
\      \ First the two transforms are separated from the data:
\      &I1 (@) &I3 (@) 2DUP +.2/ TO &H1R -.2/
\          &FFT?     IF.NEGATE.ENDIF TO &H2I
\      &I2 (@) &I4 (@) 2DUP -.2/ TO &H1I +.2/
\          &FFT? NOT.IF.NEGATE.ENDIF TO &H2R
```

```
\
\       \ Then they are recombined to form the true transform of the
\       \ original real data:
\       &H1R &WR &H2R M* &WI &H2I M* D- 1.DSCALE.NIP
\                  2DUP + &I1 (!) - &I3 (!)
\            &WR &H2I M* &WI &H2R M* D+ 1.DSCALE.NIP
\               &H1I 2DUP + &I2 (!) - &I4 (!)
\
\       \ And finally the addresses are each incremented by two indices:
\       &I1 4+ TO &I1
\       &I2 4+ TO &I2   \ every other integer number
\       &I3 4- TO &I3
\       &I4 4- TO &I4
\   NEXT
\   ENDIF
\
\   \ Do the case for samples 0 and 1:
\   &Data.addr DUP TO &I1 (@) &Data.addr 2+ DUP TO &I2 (@)
\   &FFT?
\   IF      2DUP -.2/ &I2 (!) +.2/  &I1 (!) \ if we're doing FFT
\   ELSE    2DUP -    &I2 (!) +     &I1 (!) \ if we're doing IFFT
\           &Data.addr Complex.Integer.IFFT
\   ENDIF
\   ;


CODE Real.Inner.Loop ( ...stack.frame... -- ...same.stack.frame... )
    \ Case for elements 0 and 1 is done separately outside of the loop.
    14 IND,X LDD 02 IMM ADDD 14 IND,X STD   \ WR.addr @ 2+ WR.addr !
    12 IND,X LDD 02 IMM SUBD 12 IND,X STD   \ WI.addr @ 2- WI.addr !

    \ Fetch the cosine (real) and sine (imaginary) factors from the table:
    XGDX 00 IND,X LDD                       \ WI.addr @ (@)
    30 IND,Y LDX                            \ FLAG @
    EQ IF, COMA COMB  0001 IMM ADDD ENDIF,  \ NOT.IF.NEGATE.ENDIF
    16 IND,Y STD                            \ WI !
    14 IND,Y LDX    00 IND,X LDD            \ WR.addr @ (@)
    18 IND,Y STD                            \ WR !

    \ First the two transforms are separated from the data:
    26 IND,Y LDX 00 IND,X LDD 02 IND,Y STD  \ I1 @ (@)  TEMP1 !
    22 IND,Y LDX 00 IND,X LDD 00 IND,Y STD  \ I3 @ (@)  TEMP2 !
    02 IND,Y ADDD ASRA RORB 10 IND,Y STD    \ TEMP1 @ TEMP2 @ +.2/ H1R !
    02 IND,Y LDD 00 IND,Y SUBD ASRA RORB    \ TEMP1 @ TEMP2 @ -.2/
    30 IND,Y LDX                            \ FLAG @
    NE IF, COMA COMB  0001 IMM ADDD ENDIF,  \ IF.NEGATE.ENDIF
    04 IND,Y STD                            \ H2I !

    20 IND,Y LDX 00 IND,X LDD 00 IND,Y STD  \ I4 @ (@)  TEMP2 !
    24 IND,Y LDX 00 IND,X LDD 02 IND,Y STD  \ I2 @ (@)  TEMP1 !
    00 IND,Y SUBD  ASRA RORB  08 IND,Y STD  \ TEMP1 @ TEMP2 @ -.2/ H1I !
    02 IND,Y LDD 00 IND,Y ADDD  ASRA RORB   \ TEMP1 @ TEMP2 @ +.2/
    30 IND,Y LDX                            \ FLAG @
    EQ IF, COMA COMB  0001 IMM ADDD ENDIF,  \ NOT.IF.NEGATE.ENDIF
    06 IND,Y STD
```

```
\ Then they are recombined to form the true transform of the
    \ original real data:
    DEY DEY DEY DEY
    22 IND,Y LDD 00 IND,Y STD                \ WR @
    10 IND,Y LDD 02 IND,Y STD  CALL (M*)     \ H2R @ M*

    DEY DEY DEY DEY
    24 IND,Y LDD 00 IND,Y STD        \ WI @
    12 IND,Y LDD 02 IND,Y STD        \ H2I @
    CALL (M*) CALL D-.1.DSCALE.NIP   \ M* D- 1.DSCALE.NIP
    00 IND,Y LDD 04 IND,Y STD        \ TEMP1 !
    02 IMM LDAB ABY

    10 IND,Y LDD 02 IND,Y ADDD       \ H1R @ TEMP1 @ +
    26 IND,Y LDX 00 IND,X STD        \ I1 @ (!)
    10 IND,Y LDD 02 IND,Y SUBD       \ H1R @ TEMP1 @ -
    22 IND,Y LDX 00 IND,X STD        \ I3 @ (!)

    DEY DEY DEY DEY
    22 IND,Y LDD 00 IND,Y STD        \ WR @
    08 IND,Y LDD 02 IND,Y STD  CALL (M*) \ H2I @ M*

    DEY DEY DEY DEY
    24 IND,Y LDD 00 IND,Y STD        \ WI @
    14 IND,Y LDD 02 IND,Y STD        \ H2R @
    CALL (M*) CALL D+.1.DSCALE.NIP   \ M* D+ 1.DSCALE.NIP
    00 IND,Y LDD 04 IND,Y STD        \ TEMP1 !
    02 IMM LDAB ABY

    02 IND,Y LDD 08 IND,Y ADDD       \ TEMP1 @ H1I @ +
    24 IND,Y LDX 00 IND,X STD        \ I2 @ (!)
    02 IND,Y LDD 08 IND,Y SUBD       \ TEMP1 @ H1I @ -
    20 IND,Y LDX 00 IND,X STD        \ I4 @ (!)

    \ And finally the addresses are each incremented by two indices:
    PSHY PULX
    26 IND,X LDD 04 IMM ADDD 26 IND,X STD   \  4 I1 +!
    24 IND,X LDD 04 IMM ADDD 24 IND,X STD   \  4 I2 +!
    22 IND,X LDD 04 IMM SUBD 22 IND,X STD   \ -4 I3 +!
    20 IND,X LDD 04 IMM SUBD 20 IND,X STD   \ -4 I4 +!
    RTS
END.CODE
CODE Real.Integer.FFT.Kernel
    ( Data.addr \ Table.Address \ Number.of.Complex.Points \ Flag -- )
    \ We set up a stack frame with the following variables indexed by the
    \ Y register:
    \
    \   00 Temp2        02 Temp1            04 H2i              06 H2r
    \   08 H1i          10 H1r              12 Wi.addr          14 Wr.addr
    \   16 Wi           18 Wr               20 I4               22 I3
    \   24 I2           26 I1               28 LOOP.COUNTER     30 Flag
    \   32 N            34 TABLE.ADDRESS    36 Data.addr
    \
```

```
PSHY PULA PULB 30 IMM SUBD XGDY \ Create stack frame for 15 more stack items
                                \ bottom four items are already there
    30 IND,Y LDAA               \ examine flag
    NE IF,                      \ if we're doing forward FFT
        DEY DEY  38 IND,Y LDD   00 IND,Y STD    \ push DATA.ADDR @
        DEY DEY  38 IND,Y LDD   00 IND,Y STD    \ push TABLE.ADDRESS @
        DEY DEY  38 IND,Y LDD   00 IND,Y STD    \ push N @
        CALL Complex.Integer.FFT    \ Do the FFT
    ENDIF,
    PSHY PULX                   \ We'll index with X because it's faster

    34 IND,X LDD    14 IND,X STD    \ Table.addr @ WR.addr !
    32 IND,X ADDD   12 IND,X STD    \ Table.addr @ N @  + WI.addr !

    \ Initialize the address pointers into the array:
    36 IND,X LDD    04 IMM ADDD     \ Data.addr @ 4+
    26 IND,X STD                    \ DUP I1 !
    02 IMM ADDD     24 IND,X STD    \ 2+ I2 !
    32 IND,X LDD    ASLD            \ N @ 2*
    0001 IMM SUBD   ASLD            \ 1- 2*
    36 IND,X ADDD   20 IND,X STD    \ Data.addr @  + DUP I4 !
    0002 IMM SUBD   22 IND,X STD    \ 2- I3 !

    \ Step through the data array:
    32 IND,X LDD 0002 IMM SUBD HI   \ N @ 2 U>
    IF, \ For the case N=2, or 4 points, we don't wany to execute
        \ the FOR loop 65536 times !  We want to execute it zero times.
        \ Set up the loop counter:
        32 IND,X LDD ASRA RORB          \ N @ 2/
        0001 IMM SUBD   28 IND,X STD    \ 1- LOOP.COUNTER !

        BEGIN,                          \ N/2-1 iterations
            CALL Real.Inner.Loop
            28 IND,X LDD 01 IMM SUBD 28 IND,X STD \ Decrement the loop counter
        EQ UNTIL,                       \ finish at zero
    ENDIF,

    \ Do the case for samples 0 and 1:
    36 IND,X LDD    26 IND,X STD        \ Data.addr @    I1 !
    XGDX  00 IND,X LDX 02 IND,Y STX     \ Data.addr @    (@) TEMP1 !
    36 IND,Y LDD                        \ Data.addr @
    02 IMM ADDD     24 IND,Y STD        \ 2+ I2 !
    XGDX  00 IND,X LDX 00 IND,Y STX     \ Data.addr @ 2+ (@) TEMP2 !
    30 IND,Y LDAA                       \ examine flag
    NE IF,                              \ if we're doing forward FFT
        02 IND,Y LDD    00 IND,Y SUBD   \ TEMP1 @  TEMP2 @ -
        ASRA RORB       24 IND,Y LDX  00 IND,X STD  \ 2/ I2 @ (!)
        02 IND,Y LDD    00 IND,Y ADDD               \ TEMP1 @  TEMP2 @ +
        ASRA RORB       26 IND,Y LDX  00 IND,X STD  \ 2/ I1 @ (!)
    ELSE, \ if we're doing IFFT
        02 IND,Y LDD    00 IND,Y SUBD       \ TEMP1 @  TEMP2 @ -
        24 IND,Y LDX    00 IND,X STD        \ I2 @ (!)
        02 IND,Y LDD    00 IND,Y ADDD       \ TEMP1 @  TEMP2 @ +
        26 IND,Y LDX    00 IND,X STD        \ I1 @ (!)

        DEY DEY  38 IND,Y LDD   00 IND,Y STD    \ push DATA.ADDR @
        DEY DEY  38 IND,Y LDD   00 IND,Y STD    \ push TABLE.ADDRESS @
        DEY DEY  38 IND,Y LDD   00 IND,Y STD    \ push N @
        CALL Complex.Integer.IFFT
```

```
ENDIF,
    \ Drop the stack frame
    38 IMM LDAB ABY
    RTS
END.CODE

: Real.Integer.FFT      ( Data.addr \ Table.addr \ N -- )
    2/ TRUE Real.Integer.FFT.Kernel  ;
: Real.Integer.IFFT     ( Data.addr \ Table.addr \ N -- )
    2/ FALSE Real.Integer.FFT.Kernel ;
: Initialize.FFT.Table ( Table.addr \ n -- )
    \ M is the number of points in a 1/4 COSINE wave
    \ M=N/2 for doing complex FFTs and M=N/4 for doing real FFTs.
    2/ LOCALS{ &M &Table.addr }
    &M 0
    DO
        PI &M 2* FLOT F/ I FLOT F* FCOS \ Get COS value.
        32768. F* 32767. FMIN FIXX      \ Scale and convert it to an integer.
        I 2* &Table.addr + (!)          \ Store it in the table.
    LOOP
    ;
: Initialize.Complex.FFT ( Table.addr \ N  -- )
    \ Initializes the FFT table so that integer complex FFTs can be done
    \ N is the number of points in the input waveform
    \ For complex waveforms each point contains two 16-bit integers stored
    \ in memory as real followed by imaginary.
    \ Table.addr is the 16-bit starting address of a variable table in
    \ common RAM. The table must have room for N/2 integers.
    Initialize.FFT.Table ;

: Initialize.Real.FFT ( Table.addr \ N -- )
    \ Initializes the FFT table so that integer real FFTs can be done
    \ N is the number of points in the input waveform
    \ For real waveforms each point consists of a single 16-bit integer.
    \ Points are stored in memory sequentially.
    \ Table.addr is the 16-bit starting address of a variable table in
    \ common RAM. The table must have room for N/4 integers.
    2/ Initialize.FFT.Table ;

: >Magnitude ( data.addr \ N \ matrix.xpfa -- )
    \ Converts the output of the integer FFT routine into a magnitude spectrum
    \ in which the first element of the matrix is the magnitude of the zero
    \ frequency component, the next element is the rms magnitude of the first
    \ frequency component, the next element is the rms magnitude of the second,
    \ and so on, until the last element which is the
    \ magnitude of highest frequency component.  Note that this order is different
    \ from the order of the input data vector.  The input vector is in the order:
    \ a(0), a(N/2), a(1), b(1), a(2), b(2), a(3), b(3), ..., a(N/2-1), b(N/2-1)
    \ in which the a()'s are the magnitudes of the cosine components and
    \ the b()'s are the magnitudes of the sine components.  The output is ordered:
    \ M(0), M(1), M(2), M(3), ... M(N/2-1), M(N/2)
    \ in which the M() are the rms magnitudes of the frequency components.
    \ The output is a matrix of floating point numbers.
    \ data.addr is the 16-bit address of an integer FFT spectrum
    \ N is the size of the data array
    \ matrix.xpfa is the matrix to store the power spectrum in
```

```
LOCALS{ X&Matrix &N &Data.addr }
    \ Dimension the matrix as a row vector to hold N/2 + 1 elements:
       &N 2/ 1+ 1 X&Matrix DIMMED
    \ Convert the DC component and store it:
       &Data.addr (@) ABS FLOT 0 0 X&Matrix M[]!
    \ Convert the highest frequency component and store it:
       &Data.addr 2+ (@) ABS FLOT &N 2/ 0 X&Matrix M[]!
    \ Convert all the other values:
       &N 2/ 1
       DO
           &Data.addr I 4* +    (@) FLOT FDUP F*
           &Data.addr I 4* + 2+ (@) FLOT FDUP F*
           F+ F2/ FSQRT  I 0 X&Matrix M[]!
       LOOP
    ;

AXE D+.1.DSCALE.NIP     AXE BIT-REVERSE               AXE BUTTERFLY
AXE FIRST.POINT         AXE MIDDLE.POINT              AXE TABLE.LOOKUP
AXE INNER.LOOP          AXE FFT-KERNEL                AXE (FFT)
AXE Real.Inner.Loop     AXE Real.Integer.FFT.Kernel   AXE Initialize.FFT.Table
AXE D-.1.DSCALE.NIP     AXE (M*)
```

**MI-AN-055**
**Appendix B: Examples of Using Real Integer FFTs**

```
\ ************************************************************************
\ ***************                                          ***************
\ ***************          v3.0 Using Integer FFTs         ***************
\ ***************                                          ***************
\ ************************************************************************
\ Last Revision: 03/15/95  10:38:46 AM  Paul K. Clifford
\ This documentation file is provided by Mosaic Industries as a description of
\ the fast integer FFT program in the file "v3.0 Fast Integer FFT".
\ If after reading these comments you have any remaining questions about the
\ use of these routines please feel free to call Mosaic Industries, Inc.
\ at 510-790-1255.

ANEW FFT.TESTER
DECIMAL

\ To use integer FFTs or inverse FFTs first compile the file titled
\ "v3.0 Fast Integer FFT".  There are seven user words which I will first
\ briefly describe and then for which I will provide an example of use.  These
\ user words are:

\   Initialize.Real.FFT     - Initializes a lookup table to enable doing real
\                             FFTs or IFFTs.  Only needs to be called initially
\                             and whenever the size of the FFT is changed.  For
\                             fixed sized FFTs, Initialize.Real.FFT only needs
\                             to be called once at compile time to initialize a
\                             table in ROM (on the same page as this code).  If
\                             this is done, the variables Table.Address and
\                             Number.of.Complex.Points should be either changed
\                             to constants or left as variables but compiled into
\                             ROM instead of variable space.
\
\   Initialize.Complex.FFT  - Just like Initialize.Real.FFT but for doing FFTs
\                             or IFFTs of complex waveforms.
\
\   Complex.Integer.FFT     - Performs an in-place FFT of a complex integer
\                             waveform.  The waveform must comprise pairs of
\                             signed 16-bit integers of absolute magnitude less
\                             than 2^14. ( Less than or equal to 2^14 - 1 )
\
\   Complex.Integer.IFFT    - Does the inverse FFT.
\
\   Real.Integer.FFT        - Performs an in-place FFT of a real integer
\                             waveform.  The waveform must comprise signed
\                             16-bit integers of absolute magnitude less than
\                             2^14.
\
\   Real.Integer.IFFT       - Does the inverse FFT.
\
\   >Magnitude              - Converts the integer spectrum created by the
\                             routine Real.Integer.FFT into an rms magnitude
\                             spectrum stored as a floating point matrix.  This
\                             is the most useful form of the spectrum for graphing
\                             and for computing the power in different
\                             frequency regions.
```

```
\ In the following example I will assume that we would like to perform an FFT
\ of a real waveform of 64 points.  Lets define a constant of that size:

    64 CONSTANT N   \ Size of real FFTs

\ Before we can do an FFT we must first initialize a look-up table in
\ common ram (or in ROM).  If N is the number of real points (samples) in the
\ waveform then the look-up table must have M = N/4 integer entries.  If we
\ were doing an FFT of N complex points then our table must have M = N/2
\ integer entries.  In either case the look-up table contains 16-bit signed
\ integer values of the cosine function scaled so that 1.0 is represented as
\ 2^15, and -1.0 is represented as -2^15.  (Don't worry that 2^15 can not
\ actually be represented as a signed 16-bit value (the maximum positive number
\ is actually 2^15 - 1) because the first entry of the table is not actually used.
\ The table contains the values COS(PI*I/4M) for I=0,1,2, ... M-1.  We don't
\ actually need to place the values in there, Initialize.Real.FFT or
\ Initialize.Complex.FFT will do that for us, but we do need to decide where
\ the table will go, and to reserve space for it.

\ We will also need to reserve space in the common memory for the data array
\ that will hold the input waveform and its Fourier spectrum.  We can reserve
\ space for the data and the look-up table in common RAM (assuming we have
\ unused space starting at hex 8E00) using the following code:

\ Save the old variable pointer on the stack and set the variable pointer to
\ common RAM if it was not there already.
VP X@ XDUP NIP 0= NOT IFTRUE HEX 8E00 00 VP X! ENDIFTRUE

VARIABLE (DATA)  N          \ N is the number of integer elements
                 2*         \ There are 2 bytes for each element
                 2- VALLOT  \ VARIABLE already allotted two bytes so we use
                            \ VALLOT to allot the remaining bytes

VARIABLE (TABLE) N 4/       \ N/4 is the number of elements for a look-up table
                            \ for real FFTs; if we were doing complex FFTs we
                            \ would use N/2.
                 2*         \ There are 2 bytes for each element
                 2- VALLOT  \ VARIABLE already allotted two bytes

\ Restore the old VP from the stack:
XDUP NIP 0= NOT IFTRUE VP X! OTHERWISE XDROP ENDIFTRUE

DECIMAL

\ Now (DATA) and (TABLE) leave their xaddresses on the stack when invoked.  But
\ we don't really need the page information, so we can define some constants
\ that just leave the 16-bit addresses when invoked as,

    (DATA)  DROP CONSTANT DATA
    (TABLE) DROP CONSTANT TABLE

\ Because the look-up table is in volatile RAM it must be initialized every
\ time the QED board is turned on.  Initialize.Real.FFT will fill it with the
\ proper cosine values for us if we execute:
```

```
TABLE N Initialize.Real.FFT

\ If we are doing FFTs of a complex array of N points then we would initialize
\ the look-up table using,

\       TABLE N Initialize.Complex.FFT

\ If we are always going to be doing FFTs of a certain length, instead of
\ storing the table in common RAM where we need to reinitialize it on
\ each power-up, we can compile it into ROM on the same page as the FFT code,
\ where it's more secure.  Note that it must be compiled onto the same page as
\ the rest of the FFT code !  We can do this similarly to above as,

\ Reserve space for the table in the next available code space:
\
\ HEX VP X@                  \ Save the old variable pointer
\ DP X@ VP X!                \ and set the variable pointer to the next available
\                            \ location in ROM
\ VARIABLE (TABLE) N 4/      \ N/4 is the number of elements for a look-up table
\                            \ for real FFTs, if we were doing complex FFTs we
\                            \ would use N/2.
\                2*          \ There are 2 bytes for each element
\                2- VALLOT   \ VARIABLE already allotted two bytes
\ VP X@ DP X!                \ Update DP
\ VP X!                      \ and restore the old VP
\ DECIMAL
\
\ We would still reserve space for the data array in the common RAM just like
\ before:
\
\ Save the old variable pointer on the stack and set the variable pointer to
\ common RAM if it was not there already.
\ VP X@ XDUP NIP 0= NOT IFTRUE HEX 8E00 00 VP X! ENDIFTRUE
\ VARIABLE (DATA)  N         \ N is the number of integer elements
\                2*          \ There are 2 bytes for each element
\                2- VALLOT   \ VARIABLE already allotted two bytes
\
\ Restore the old VP from the stack:
\ XDUP NIP 0= NOT IFTRUE VP X! OTHERWISE XDROP ENDIFTRUE
\ DECIMAL

\ And just like before we would define constants to return the 16-bit addresses:
\   (DATA)  DROP CONSTANT DATA
\   (TABLE) DROP CONSTANT TABLE
\ Now we initialize the table only once, DURING COMPILATION, by executing,
\       TABLE N Initialize.Real.FFT
\ The table now resides with the code, and once burned into ROM never needs
\ to be reinitialized.

\ Now, to perform an FFT on a real waveform comprising 16-bit integers stored
\ at the address returned by DATA we simply execute:

\   DATA TABLE N Real.Integer.FFT

\ Or, to perform an FFT on a complex waveform comprising pairs of 16-bit integers
\ (real,imaginary,real,imaginary,...) stored at the address returned by DATA
\ we would execute:
```

```
\   DATA TABLE N Complex.Integer.FFT

\ To test the FFTs, we will write a word that fills DATA with the sum of some
\ sine and cosine waves as,

: FILL.IT ( Data.addr \ N -- )
    \ Data.addr is the starting address of the array
    \ N is the number of samples
    LOCALS{ &N &Data.addr | F&2PI/N F&N/2 }
    &N 2/ FLOT TO F&N/2
    PI F2* &N FLOT F/ TO F&2PI/N
    &N  0
    DO
        \ Add a bunch of COS and SIN waves all of unity magnitude:
       -0.5                                      \ constant, zero freq, term
        1.0        I FLOT F* F&2PI/N F* FCOS  1.0 f* F+ \ fundamental frequency, COS
        2.0        I FLOT F* F&2PI/N F* FSIN -0.5 f* F+ \ twice freq, SIN
        3.0        I FLOT F* F&2PI/N F* FCOS  1.0 f* F+ \ 3rd freq, COS
        3.0        I FLOT F* F&2PI/N F* FSIN  1.0 F* F+ \ 3rd freq, SIN
        4.0        I FLOT F* F&2PI/N F* FSIN  1.5 f* F+ \ 4th freq, SIN
        F&N/2      I FLOT F* F&2PI/N F* FCOS  1.5 f* F+ \ greatest frequency, COS
        F&N/2 1. F- I FLOT F* F&2PI/N F* FSIN 0.5 f* F+ \ penultimate freq, SIN
        \ We'll scale the waveform so that we know it can't exceed 2^14:
        2000. F* 16383. FMIN -16383. FMAX
        \ convert it to integers and store in DATA
        FIXX &Data.addr I 2* + (!)  \ 2 bytes per element
    LOOP
    ;

\ And we'll also write a word that will print the integer array as a two column
\ matrix of floating point numbers (1st column for COS components and 2nd column
\ for SIN components of the transform):

: PRINT.DATA ( addr \ N -- )
    \ addr is starting address of integer array to print and N is the number
    \ of elements
    LOCALS{ &N &Data.addr }

    RIGHT.PLACES @ 2 RIGHT.PLACES ! \ make a nice printing format
    &N  0
    DO
        I 2 MOD 0= IF CR ENDIF      \ print as two columns
        &Data.addr I 2* + (@) FLOT  \ get the element and float it
        2000. F/                    \ unscale using same factor as FILL.IT used
        FIXED.                      \ print it out
    LOOP
    RIGHT.PLACES !              \ restore prior printing format
    ;
```

```
\ For an input waveform given by N samples in the time domain by,
\
\ S(n) = a(0)   + a(1) COS(1*2*PI*n/N)          + b(1) SIN(1*2*PI*n/N)
\               + a(2) COS(2*2*PI*n/N)          + b(2) SIN(2*2*PI*n/N)
\               + a(3) COS(3*2*PI*n/N)          + b(3) SIN(3*2*PI*n/N)
\               + ...
\               + a(N/2-1) COS((N/2-1)*2*PI*n/N) + b(N/2-1) SIN((N/2-1)*2*PI*n/N)
\               + a(N/2) COS((N/2)*2*PI*n/N)    + b(N/2) SIN((N/2)*2*PI*n/N)
\   (for n = 0, 1, ... N-1)
\ our print word, PRINT.DATA, should print f(n) in the format,
\   S(0)    S(1)
\   S(2)    S(3)
\   ...     ...
\   S(N-2)  S(N-1)
\ and after we do the FFT the word PRINT.DATA should print the spectrum as,
\
\   a(0)        a(N/2)
\   a(1)        b(1)
\   a(2)        b(2)
\   a(3)        b(3)
\   a(N/2-1)    b(N/2-1)
\
\ Note that b(N/2), as the first aliased component, is not part of the spectrum.
\ Only the cosine phase of the hightest frequency component, a(N/2), is actually
\ present in the sampled waveform and shows up in the spectrum, the sine phase
\ is aliased to zero.  The cosine phase is also aliased in magnitude because it
\ is sampled only at its peak excursions.
\ For the waveform produced by the FILL.IT word above the spectrum should
\ look like:
\
\    -0.5     1.5
\     1.0     0.0
\     0.0    -0.5
\     1.0     1.0
\     0.0     1.5
\     0.0     0.0
\     0.0     0.0
\     ...     ...
\     ...     ...
\     0.0     0.0
\     0.0     1.5
\ Now try the following commands from the terminal to see how they behave:
\   Data  N Fill.It                 \ Fill the data with the waveform
\   Data  N Print.Data              \ examine the input waveform
\   TABLE N Initialize.Real.FFT     \ initialize the FFT look-up table if we
\                                   \ haven't already
\   Start.Timeslicer                 \ prepare to measure execution times
\   Data Table N Benchmark: Real.Integer.FFT \ do the FFT
\   Data  N Print.Data              \ print the spectrum of the input waveform
\                                   \ we can recover the original waveform
\                                   \ by doing an Inverse FFT
\   Data Table N Benchmark: Real.Integer.IFFT \ do the IFFT
\   Data  N Print.Data              \ and we can examine the regenerated waveform

\   MATRIX: SPECTRUM                \ Define a matrix to hold the rms magnitude
\                                   \ spectrum
\   Data  N ' SPECTRUM >Magnitude   \ Convert the spectrum to rms magnitudes
\                                   \ stored a floating point numbers in a
\                                   \ matrix.
```

**MI-AN-055**
**Appendix C: Floating Point Complex and Real FFT Code**

```
\ ********************************************************************
\ ***************                                      ***************
\ ***************          v3.0 Floating Point FFT Code ***************
\ ***************                                      ***************
\ ********************************************************************



\ Last Revision: 02/17/95  03:14:29 PM  PKC
\ Last Revision: 02/19/95  11:01:01 PM  PKC

\   The only way I can see to make these routines much faster would be to
\ require that the data matrices be placed in common ram, so that all fetches
\ and stores, currently F@ and F!, can be replaced with pageless fetches and
\ stores (F@) and (F!).  I wouldn't need to keep the page information at all
\ anymore, and both the FFT.Kernel word and the Bit.Reverse word could be
\ rewritten to be much faster.  This is a major rewrite however, and would
\ produce a much less generally useful program, but it might be necessary for
\ a highly specialized application.
\
\ Some speed gain could be had by requiring the FFT.Table to be on the same
\ page as this code.  In that case we can do away with its page info and just
\ use (F@) to fetch from it.  We can also use a full look-up table instead of
\ using trigonometric recurrances.

ANEW <FOURIER>

\ Updated FFT for QED 3.x software.

\ Please see the comments in the program that contain the "***" characters.
\ This program for QED 3.x is different than the QED 2.x software in that it
\ follows a different definition of the Fourier transform. It defines the DFT
\ as
\   X(f) = (1/N) *   x(t)exp[+2ðift/N]
\ to be consistent with the Numerical Recipes book. The QED 2.x software used
\ the following definition (note the change in sign of the argument of the
\ exponential):
\   X(f) =   x(t)exp[-2ðift/N]
\ which is consistent with the "Digital Signal Processing" and other books.

DECIMAL  \ Must be DECIMAL !

5 WIDTH !
CODE IF.FNEGATE.ENDIF
    00 IND,Y LDD
    NE IF, 02 IND,Y NEG ENDIF,
    02 IMM LDAB
    ABY
    RTS
END.CODE
```

```
CODE NOT.IF.FNEGATE.ENDIF
    00 IND,Y LDD
    EQ IF, 02 IND,Y NEG ENDIF,
    02 IMM LDAB
    ABY
    RTS
END.CODE

FIND 8XN- NOT
IFTRUE
    HEX
    CODE 8XN-               \ The kernel has an 8XN+ but no 8XN-
        02  IND,Y   LDD     \ so we define it here.
        08  IMM     SUBD
        CS IF,
            01  IND,Y   DEC
            7F  IMM     ANDA
        THEN,
        02  IND,Y   STD
        RTS
    END.CODE
    DECIMAL
OTHERWISE
    XDROP
ENDIFTRUE

8192 LOG2 1+ 2* 1 DIM.CONSTANT.MATRIX: FFT.TABLE

: FFT.INIT ( -- )
    \ FFT.TABLE must be already dimensioned as a constant matrix as
    \   8192 LOG2 1+ 2* 1 DIM.CONSTANT.MATRIX: FFT.TABLE
    \ for doing transforms of up to 8192 in size
    \ This initialization routine fills FFT.TABLE with cosines and sines as,
    \ ***  QED 2.x version: exp[-i(PI/2^I)]= cos(PI/2^I) - i sin(PI/2^I)
    \ New QED 3.x version:  exp[+i(PI/2^I)]= cos(PI/2^I) + i sin(PI/2^I)
    \
    \ #ROWS = 2* [ log2(M) + 1 ] , for up to M-point xforms
    \ 11 real/imaginary number pairs, real in low memory, for 1024 point FFT
    \ For the trigonometric recurrance we need cosine/sine pairs.  However,
    \ instead of using the cosine directly, it is more accurate to use a recurrance
    \ formula that makes use of the trigonometric relationship:
    \   COS(x) = 1 - 2[SIN(x/2)]^2
    \ So instead of storing the COS(x) we will store -2[SIN(x/2)]^2
    ' FFT.TABLE ?DIM.MATRIX DROP 2/ 0
    DO
        PI I NEGATE FSCALE FDUP          \ argument, PI/2^I
        F2/ FSIN FDUP F* F2* FNEGATE     \
        I 2*    0 FFT.TABLE F!           \ store real part
        FSIN                             \ imag = +sin(PI/2^I)
        I 2* 1+ 0 FFT.TABLE F!           \ store imaginary part
    LOOP
    ;
```

```
\ FFT.INIT emplaces the table of trigonometric values in memory and then can
\ be forgotten:

FFT.INIT    FORGET FFT.INIT


CODE (BIT.REVERSE) ( n2 \ n1  --  n2 \ n3 )
    \ n2 is the number of bits to reverse, must be 0 < n2 <= 16 or there
    \ is an unchecked error.
    \ n1 is the 16-bit integer to reverse, only the lowest n2 bits are used
    \ n3 is the bit-reversed version of n1
    DEY DEY             \ Duplicate the top item on stack (n1)
    02 IND,Y LDD        \
    00 IND,Y STD        \
    0000 IMM LDD        \ Zero 2nd item on stack, will hold result (n3)
    02 IND,Y STD
    05 IND,Y LDAB       \ get number of bits to reverse (low byte of n2)
    BEGIN,
        00 IND,Y ROR    \ rotate high byte of n1, shifting in/out the carry
        01 IND,Y ROR    \ rotate low  byte of n1, shifting in/out the carry
        03 IND,Y ROL    \ rotate low  byte of n2, shifting in/out the carry
        02 IND,Y ROL    \ rotate high byte of n1, shifting in/out the carry
        DECB            \ decrement number of bits
    EQ UNTIL,
    02 IMM LDAB         \ Pop top item from stack
    ABY
    RTS                 \ return
    END.CODE

: BIT-REVERSE           ( N\Data.vector.xpfa -- )
    [0]
    LOCALS{ x&data.vector &N | &Index X&C2 X&C1 X&R2 X&R1 }
    &N LOG2     \ leave the #bits to bit.reverse on stack
    &N 1- 0
    DO
            I (BIT.REVERSE) DUP TO &Index I
            >
            IF
                x&data.vector      I 8* XN+ XDUP TO X&R2 4XN+ TO X&C2
                x&data.vector &Index 8* XN+ XDUP TO X&R1 4XN+ TO X&C1
                \ @CXSWAP! :
                X&C2 F@ X&C1 F@ X&C2 F! X&C1 F!
                X&R2 F@ X&R1 F@ X&R2 F! X&R1 F!
            ENDIF
    LOOP
    DROP        \ drop the #bits to bit.reverse from stack
    ;
```

```
: FFT-KERNEL ( IFFT.flag\N\FFT.Table.xpfa\Data.vector.xpfa -- )
    [0] XSWAP [0]
    LOCALS{ x&Table &Page &DVADR &N &FLAG
          | &8(N-1)+DVADR F&R* F&I* F&WR F&WI X&A1 X&A2 F&UR F&UI }
    &DVADR &N 1- 8* + TO &8(N-1)+DVADR
    &N LOG2 0
    DO
        ZERO TO F&UI ONE TO F&UR                \ U=1.0 real for starters
        x&Table I 8* XN+ XDUP F@ TO F&WR        \ W from FFT.TABLE
        \ For IFFT use complex conjugate
        4XN+ F@ &FLAG IF.FNEGATE.ENDIF TO F&WI
        1 I 4+ SCALE                            \ leave on stack 2^(I+4)
        DUP 2/ &DVADR +   &DVADR                \ loop limits
        DO
            I &DVADR =
            IF
                >ASSM DEY DEY 02 IND,Y LDD      \ Code for
                ASRA RORB  00 IND,Y STD >FORTH  \ DUP 2/
                &8(N-1)+DVADR  I
                DO  \ Do the butterfly computation:
                    \ I 2DUP +
                    >ASSM TSX 00 IND,X LDD DEY DEY 00 IND,Y STD \ Code for
                    02 IND,Y ADDD DEY DEY 00 IND,Y STD >FORTH   \ I 2DUP +
                    &Page TO X&A2 &Page TO X&A1
                    X&A2 4XN+  F@    TO F&I*
                    x&A2 F@ FNEGATE  TO F&R*
                    X&A1       F@ FDUP F&R* F+ X&A2      F!
                    X&A1 4XN+ F@ FDUP F&I* F- X&A2 4XN+ F!
                                 F&I* F+ X&A1 4XN+ F!
                                 F&R* F- X&A1      F!
                OVER +LOOP
                DROP
                ELSE
                I OVER 4/ &DVADR + =
                IF
                    1.0 &FLAG IF.FNEGATE.ENDIF TO F&UI 0.0 TO F&UR

                    >ASSM DEY DEY 02 IND,Y LDD      \ Code for
                    ASRA RORB  00 IND,Y STD >FORTH  \ DUP 2/
                    &8(N-1)+DVADR  I
                    DO  \ Do the butterfly computation:
                        \ I 2DUP +
                        >ASSM TSX 00 IND,X LDD DEY DEY 00 IND,Y STD \ Code for
                        02 IND,Y ADDD DEY DEY 00 IND,Y STD >FORTH   \ I 2DUP +
                        &Page TO X&A2 &Page TO X&A1

                        x&A2      F@ &FLAG IF.FNEGATE.ENDIF TO F&I*
                        X&A2 4XN+ F@ &FLAG IF.FNEGATE.ENDIF TO F&R*

                        X&A1      F@ FDUP F&R* F+ X&A2      F!
                        X&A1 4XN+ F@ FDUP F&I* F- X&A2 4XN+ F!
                                  F&I* F+ X&A1 4XN+ F!
                                  F&R* F- X&A1      F!
                    OVER +LOOP
                    DROP
```

```
ELSE
                    >ASSM DEY DEY 02 IND,Y LDD      \ Code for
                    ASRA RORB  00 IND,Y STD >FORTH  \ DUP 2/
                    &8(N-1)+DVADR  I
                    DO  \ Do the butterfly computation:
                        \ I 2DUP +
                        >ASSM TSX 00 IND,X LDD DEY DEY 00 IND,Y STD \ Code for
                        02 IND,Y ADDD DEY DEY 00 IND,Y STD >FORTH   \ I 2DUP +
                        &Page TO X&A2 &Page TO X&A1
                        X&A2 4XN+  F@  FDUP
                        F&UI  F*  x&A2 F@ F&UR F* F-  TO F&R*
                        F&UR  F*  X&A2 F@ F&UI F* F+  TO F&I*
                        X&A1      F@ FDUP F&R* F+ X&A2      F!
                        X&A1 4XN+ F@ FDUP F&I* F- X&A2 4XN+ F!
                                  F&I* F+ X&A1 4XN+ F!
                                  F&R* F- X&A1      F!
                    OVER +LOOP
                    DROP
                ENDIF
            ENDIF
            \ Now the trigonometric recurrance:
                F&UR FDUP F&WR F* F&UI F&WI F* F- F+
                F&UI FDUP F&WR F* F&UR F&WI F* F+ F+
                TO F&UI TO F&UR
        8 +LOOP
        DROP \ drop the 2^(I+4)
    LOOP
    ;

\ : IFFT.SCALE ( n\...\r -- n\...\r )
\   \ Tricky word: This is called by V.TRANSFORM which leaves many things
\   \ on the stack.  We have to reach below them to get our scale factor.
\   16 PICK FSCALE
\   ;

\ The above high level code is replaced with the following assm code:

CODE IFFT.SCALE  ( n\...\r -- n\...\r )
    DEY DEY
    34 IND,Y LDD
    00 IND,Y STD
    CFA.FOR FSCALE HERE NIP =       \ Is FSCALE on the same page as this
    OVER 0< OR                      \ routine or is it in the common memory?
    IFTRUE     EXT JMP              \ If so just jump to it.
    OTHERWISE  DROP CALL FSCALE RTS \ Else compile a call with a page
    ENDIFTRUE                       \ if necessary.
END.CODE
```

```
: NORMALIZE ( N\Data.vector.xpfa -- | divide data.vector by N if an IFFT was done)
    [0]
    LOCALS{ X&Data.addr &N }
    &N LOG2 NEGATE          \ scale factor ( number of bits to shift )
                            \ we'll just leave it on the stack where IFFT.SCALE
                            \ can find it
    X&Data.addr 1           \ source vector starting xaddress and element separation
    X&Data.addr 1           \ dest vector starting address and sep
    &N  2* U>D              \ number of fp elements to scale as a double number
    CFA.FOR IFFT.SCALE
    V.TRANSFORM             \ scale the entire vector
    DROP                    \ drop the scale factor
    ;

: (FFT)     ( FFT.Table.xpfa\Data.xpfa\[-1] or [0] -- | -1 for IFFT and 0 for FFT )
    LOCALS{ &IFFT.flag x&Data.vector.pfa x&Table | &N }
    x&Data.vector.pfa ?DIM.MATRIX * 2/ TO &N
    &N                      x&Data.vector.pfa BIT-REVERSE
    &IFFT.flag &N x&Table x&Data.vector.pfa FFT-KERNEL
    &IFFT.flag NOT
    IF
        &N x&Data.vector.pfa NORMALIZE
    THEN
    ;

: FFT   ( Data.xpfa -- )
    ' FFT.TABLE XSWAP FALSE (FFT) ;

: IFFT  ( Data.xpfa -- )
    ' FFT.TABLE XSWAP TRUE (FFT) ;
```

```
\ The result is in spectral format, as two rows and many columns.
    \ Each column but the first contains the sine and cosine components of
    \ each frequency.  The first column contains the magnitudes of the
    \ zero frequency (DC) component, and the (aliased) highest frequency
    \ component (as a real, or cosine, frequency).  The second column contains
    \ the first frequency (the fundamental or harmonic #1), the next column the
    \ second frequency (harmonic #2) and so on.  For all columns but the first
    \ the first row contains the real (cosine) components and the second row
    \ the imaginary (sine) components.
    \
    \ If flag is true we do a real FFT, if false we do a real Inverse FFT.
    \ If we are doing an inverse FFT the output is presented as a single
    \ column vector.
    \
    3 NEEDED
    LOCALS{ &FFT? X&Matrix
          | &N  X&I1  X&I2  X&I3  X&I4
               F&WR  F&WI  F&WPR F&WPI
               F&H1R F&H1I F&H2R F&H2I }
    X&Matrix ?DIM.MATRIX * 2/ TO &N
    &FFT?
    IF                                  \ if we're doing forward FFT
        2 &N X&Matrix REDIMMED          \ Redimension the input waveform
        X&Matrix FFT                    \ Do the FFT
    ENDIF

    \ Fetch the COS term from the FFT.TABLE.  The COS term is actually stored
    \ as a SIN^2 term for better accuracy in the recurrance formula.
    \   COS(x) = 1 - 2[SIN(x/2)]^2
    \ So instead of fetching COS(x) directly we fetch -2[SIN(x/2)]^2
    &N Log2 2* 0 FFT.TABLE XDUP  F@ FDUP TO F&WPR  1.0 F+ TO F&WR
    \ And fetch the SIN term:
        4XN+ F@ &FFT? NOT.IF.FNEGATE.ENDIF FDUP TO F&WPI TO F&WI

    \ *** In the above line we negated the SIN term which is equivalent
    \ to negating the angle used in exp[ +/- 2ðift/N].
    \ If our FFT routine uses the convention found in the book "Digital Sig
    \ Processing", that is X(f) =   x(t)exp[-2ðift/N] then we need the FNEGATE
    \ when doing an FFT, not for the IFFT.  If on the other hand we use the
    \ convention X(f) =   x(t)exp[+2ðift/N] as does the "Numerical Recipes"
    \ book, then we need the FNEGATE for the IFFT instead.  We'll use the
    \ "Numerical Recipes" convention.

    \ Initialize the addresses into the array:
            X&Matrix [0] 8XN+ XDUP TO X&I1 4XN+ TO X&I2
    1 &N 1- X&Matrix M[]      XDUP TO X&I4 4XN- TO X&I3


    \ Step through the array:
    &N 2 >          \ For the case &N=2, or 4 points, we don't wany to execute
                    \ the FOR loop 65536 times !  We want to execute it zero
                    \ times.
```

```
IF
    &N 2/ 2-        \ N/2-1 iterations
    FOR             \ Case for elements 0 and 1 is done separately below.
    \ The xaddresses step through the matrix as though it is a 2N-element
    \ linear array with element addresses of { 0, 1, 2, ... 2N-2, and 2N-1 }.
    \ I1 steps through array indices = {    2,    4,    6, ...   N-4, N-2 }
    \ I2 steps through array indices = {    3,    5,    7, ...   N-3, N-1 }
    \ I3 steps through array indices = { 2N-2, 2N-4, 2N-6, ...   N+4, N+2 }
    \ I4 steps through array indices = { 2N-1, 2N-3, 2N-5, ...   N+5, N+3 }
    \ The I1-I4 notation preserves the variable names from the listing
    \ in the Numerical Recipes book.

        \ First the two transforms are separated from the data:
        X&I1 F@ X&I3 F@ F2DUP F+ F2/ TO F&H1R F- F2/ &FFT?     IF.FNEGATE.ENDIF
            TO F&H2I
        X&I2 F@ X&I4 F@ F2DUP F- F2/ TO F&H1I F+ F2/ &FFT? NOT.IF.FNEGATE.ENDIF
            TO F&H2R

        \ Then they are recombined to form the true transform of the
        \ original real data:
        F&H1R F&WR  F&H2R F*   F&WI  F&H2I F* F-    F2DUP  F+ X&I1 F!  F- X&I3 F!
        F&WR  F&H2I F*    F&WI F&H2R F*    F+ F&H1I F2DUP  F+ X&I2 F!  F- X&I4 F!

        \ Then the recurrence is implemented:
        F&WR F&WPR F* F&WI F&WPI F* F- F&WR F+
        F&WI F&WPR F* F&WR F&WPI F* F+ F&WI F+
        \ Must occur in this order, prior value of F&WR is used in above line:
            TO F&WI TO F&WR

        \ And finally the addresses are each incremented by two indices:
        X&I1 8XN+ TO X&I1
        X&I2 8XN+ TO X&I2   \ every other floating point number
        X&I3 8XN- TO X&I3
        X&I4 8XN- TO X&I4
    NEXT
    ENDIF

    &FFT?   \ if we're doing FFT
    IF      \ Do the case for samples 0 and 1:
        X&Matrix [0] XDUP TO X&I1 F@ 1 0 X&Matrix M[] XDUP TO X&I2 F@
            F2DUP F- F2/ X&I2 F! F+ F2/ X&I1 F!
    ELSE            \ if we're doing IFFT
        X&Matrix [0] XDUP TO X&I1 F@ 1 0 X&Matrix M[] XDUP TO X&I2 F@
            F2DUP F- X&I2 F! F+ X&I1 F!
        X&Matrix IFFT
        &N 2* 1 X&Matrix REDIMMED           \ Redimension the output waveform
    ENDIF
    ;

: Real.FFT      ( matrix.xpfa -- )
    TRUE Real.FFT.Kernel
    ;
: Real.IFFT     ( matrix.xpfa -- )
    FALSE Real.FFT.Kernel
    ;

\ AXE FFT.TABLE         \ This table may be useful if anyone needs
                        \ some sines and cosines
```

```
AXE BIT-REVERSE          AXE FFT-KERNEL          AXE IFFT.SCALE
AXE NORMALIZE            AXE (FFT)
AXE Real.FFT.Kernel


\ A NOTE ON THE FREQUENCY COMPONENTS RETURNED BY THE ROUTINE REAL.FFT:

\ Only the coefficients of positive frequencies are saved, but by symmetry
\ (for real-valued waveforms) the coefficients of the negative frequencies are
\ the same as for the positive frequencies.
\ The DC component of the spectrum equals the number of points, N, times the DC
\ magnitude of the input waveform.  This is also true of the highest frequency
\ component, the aliased component.  For this component only the cosine phase
\ is found and its coefficient in the spectrum is equal to N times the magnitude
\ of the cosine wave at the highest frequency.

\ All of the other frequency coefficients of the spectrum are equal to one half
\ of the number of points, N/2, times the magnitudes of the individual sine or
\ cosine contituents of the input waveform.

\ The output spectrum is represented by two rows.  Aside from the two values in
\ the first column, the first row represents the cosine components for frequencies
\ of 1, 2, ... N/2-1 and the second row represents the sine components for these
\ same frequencies.  The first value of the first column represents the DC or
\ zero frequency component, and the second value represents the aliased highest
\ frequency component.

\ An input waveform given by N samples in the time domain by,
\
\ f(n) = a(0) + a(1) COS(1*2PIn/N) + a(2) COS(2*2PIn/N) + a(3) COS(3*2PIn/N) +
\
\               ... a(N/2-1) COS((N/2-1)*2PIn/N) + a(N/2) COS((N/2)*2PIn/N)
\
\            + b(1) SIN(1*2PIn/N) + b(2) SIN(2*2PIn/N) + b(3) SIN(3*2PIn/N) +
\
\               ... b(N/2-1) SIN((N/2-1)*2PIn/N) + b(N/2) SIN((N/2)*2PIn/N)
\
\       (for n = 0, 1, ... N-1)
\
\ generates a Real.FFT given by a two row matrix in the following form:
\
\       a(0)    a(1)    a(2)    a(3)    ... a(N/2-1)
\
\       a(N/2)  b(1)    b(2)    b(3)    ... b(N/2-1)
\
\
\ Note that this spectrum does not contain both phases of the highest frequency
\ component; b(N/2) can not be determined because it is not actually
\ represented in the discretely sampled waveform of only N points.  That is,
\ in the time domain equation above, b(N/2) SIN((N/2)*2PIn/N) always equals
\ zero because SIN((N/2)*2PIn/N) = SIN(n*PI) = 0.  Sampling a waveform with
\ N samples starting at time zero can not capture any of the SIN-phase energy at
\ frequency N/2.  This is why we say that this highest frequency component is
\ aliased.  Any frequencies of the waveform greater than N/2, both COS and SIN
\ phases, are aliased during sampling by being folded into lower frequency
\ components.
```

```
ANEW FFT.TESTER

Hex 0000 07 8000 07 is.heap

DECIMAL

MATRIX: DATA1  128 1 ' DATA1 DIMMED

: FILL.IT ( MATRIX.XPFA -- )
    LOCALS{ X&MATRIX | &N X&ADDR }
    X&MATRIX ?DIM.MATRIX * TO &N
    X&MATRIX [0] TO X&ADDR
    &N  0
    DO
    \       PI    F2* I FLOT F* &N FLOT F/ FCOS
    \   2.0 PI F* F2* I FLOT F* &N FLOT F/ FSIN F+
    \   3.0 PI F* F2* I FLOT F* &N FLOT F/ FCOS F+
    \   4.0 PI F* F2* I FLOT F* &N FLOT F/ FSIN F+
    \   5.0 PI F* F2* I FLOT F* &N FLOT F/ FCOS F+
    \   8.0 PI F* F2* I FLOT F* &N FLOT F/ FCOS F+
    \   1.0 F+
        FRANDOM
    \ F+
        X&ADDR F!
        X&ADDR 4XN+ TO X&ADDR
    LOOP
    ;

' DATA1 FILL.IT

: test 10 0 do ' DATA1 FFT  ' DATA1 IFFT LOOP ;

start.timeslicer
\ ' DATA1 BENCHMARK: Real.FFT
\ BENCHMARK: test
' DATA1 BENCHMARK: FFT


\ ****************************** End of Code ********************************
```



Mosaic Industries

5437 Central Ave Suite 1, Newark, CA 94560 Telephone: (510) 790-8222 Fax: (510) 790-0925