EC581 DSP Projects: Lab Project #6

Dept. of Electrical and Computer Engineering
Rose-Hulman Institute of Technology

Radix 4 FFT and Real-time Spectrum Analyzer

Last Modified on 7/29/00 (KEH)

1. Study the radix-4 FFT algorithm explained in **Appendix A**. Then draw the (much simpler) 16-point radix-4 FFT butterfly pattern, similar to the 64-point radix-4 pattern on p. 3 of the handout. Be sure to indicate the ordering of the resulting transform points on the right.

2. Now assume the 16 input points consist of the following (real) values:

   {1000 1000 1000 0 0 0 0 0 0 0 0 0 0 0 0 0}

   Enter these numbers on the left side of your butterfly pattern, and then indicate the value of each remaining node in the butterfly pattern. Include as ***Attachment A.***

3. Now run a MATLAB FFT on this same 16-value input sequence. Show that the two results agree. Include MATLAB results as ***Attachment B.***

4. Next, run the radix-4 FFT C62x DSP board demo program that is listed in **Appendix B.** Note it is already set up to calculate the 16-point FFT of the sequence given above. Show that its results agree also. Include a copy of this output as ***Attachment C.***

5. Now, modify this program so that it now calculates a 64-point FFT. (You may want to initialize the twiddle factor array "w[ ]" using a software routine that calculates the sine and cosine values just once, at the beginning of the program. Or you may decide to use a MATLAB M file to help you generate the scaled short-integer values and enter it into a table, as was done in the 16-bit example.) Include your modified program as ***Attachment D.***

6. Test your 64-point FFT by comparing its results against a MATLAB FFT for at least 2 test cases. Include copies of the MATLAB FFT output and the DSP program output as ***Attachment E*** that prove the 64-point algorithm is working.

7. ***Continuously Updated Audio Spectrum Analyzer***

Integrate this 64-point, radix-4 FFT program with your interrupt-driven sampling program. Your program must sample 64 real input values at a sampling rate of 8 kHz. It should place these samples into even-indexed positions of a 128-element global input array (the odd-indexed positions correspond to the imaginary values of the input sequence, and these may be set to zero in the interrupt routine).

When the global index variable used to fill this input array reaches 127, the interrupt routine should stop filling the input array, though it should continue to sample and output the samples to the CODEC. The interrupt routine should wait until the main program sets the index variable back to zero, thereby signaling the interrupt routine that it is time to re-fill the input array. Once the main

program sees the global index variable reach 127, it can assume that the input array is full, and it should now call the radix-4 FFT routine to calculate the FFT of these points.

Finally, the squared magnitude of the first half of the transformed complex-valued points (the first 32) must be displayed, using *printf( ),* by the main DSP program. There is no need to display the remaining 32 points, since the first 32 points correspond to the usable frequency range between 0 Hz to fs/2 = 8000/2 = 4000 Hz. We really don't care about displaying the second half of the FFT, which corresponds to frequencies above the Nyquist sampling limit, fs/2 .

Once printing is complete, the main program can set the global index variable back to 0 to signal the interrupt routine that it is now time to re-fill the input array.

Demonstrate your working spectrum analyzer program and obtain the signature of the lab instructor on your adequately commented program listing. Include this signed listing as ***Attachment F***.

Appendix A. Radix 4 Decimation-in-Frequency FFT Algorithm Description, from "DSP Applications with TMS320", Texas Instruments, 1986 (SPRA012A)

From DSP Applications with TMS320, TI, 1986 (SPRA012A) (1)

RADIX-4 DECIMATION-IN-FREQUENCY (DIF) FFT

The implementation described thus far is that of a radix-2 FFT using Decimation In Time (DIT). The decimation-in-time FFT is calculated by breaking the input sequence x(n) into smaller and smaller sequences and computing their FFTs. In an alternate approach, the output sequence X(k), which represents the Fourier transform of x(n), can be broken down into smaller subsequences that are computed from x(n). This method is called Decimation In Frequency (DIF). Computationally, there is no real difference between the two approaches. DIF is introduced here for two reasons: (1) to give the reader a broader understanding of the different methods used for the computation of the FFT, and (2) to allow a comparison of this implementation with the FORTRAN programs provided in the book by Burrus and Parks.[4] The programs from that book were the basis for the development of the radix-4 FFT code on the TMS32020.

In a radix-4 FFT, each butterfly has four inputs and four outputs instead of two as in the case of radix-2 FFT. As shown in the following equations, this is advantageous because the twiddle factor W has special values when the exponent corresponds to multiples of $\pi/2$. The end result is that the computational load of the FFT is reduced, and the radix-4 FFT is computed faster than the radix-2 FFT.

To introduce the radix-4 DIF FFT, equation (3) is broken into four summations. These four summations correspond to the four components in radix-4. The choice of having N/4 consecutive samples of x(n) in each sum is dictated by the choice of Decimation In Frequency (DIF).

$$\begin{aligned} X(k) = & \sum_{n=0}^{N-1} x(n)\, W_N^{nk} = \sum_{n=0}^{(N/4)-1} x(n)\, W_N^{nk} \\ & + \sum_{n=N/4}^{(N/2)-1} x(n)\, W_N^{nk} + \sum_{n=N/2}^{(3N/4)-1} x(n)\, W_N^{nk} \\ & + \sum_{n=3N/4}^{N-1} x(n)\, W_N^{nk} = \sum_{n=0}^{(N/4)-1} x(n)\, W_N^{nk} \\ & + W_N^{Nk/4} \sum_{n=0}^{(N/4)-1} x(n+N/4)\, W_N^{nk} \\ & + W_N^{Nk/2} \sum_{n=0}^{(N/4)-1} x(n+N/2)\, W_N^{nk} \\ & + W_N^{3Nk/4} \sum_{n=0}^{(N/4)-1} x(n+3N/4)\, W_N^{nk} \end{aligned} \qquad (17)$$

$$k = 0,1,\ldots,N-1$$

From the definition of the twiddle factor, it can be shown that

$$W_N^{Nk/4} = (-j)^k,\ W_N^{Nk/2} = (-1)^k, \text{ and } W_N^{3Nk/4} = (j)^k$$

where j is the square root of -1. With this substitution, (17) can be rewritten as

$$\begin{aligned} X(k) = & \sum_{n=0}^{(N/4)-1} [x(n) + (-j)^k\, x(n+N/4) \\ & + (-1)^k\, x(n+N/2) + (j)^k\, x(n+3N/4)]\, W_N^{nk} \end{aligned} \qquad (18)$$

Equation (18) is not yet an FFT of length N/4, because the twiddle factor depends on N and not on N/4. To make it an N/4-point FFT, the sequence X(k) is broken into four sequences (decimation in frequency) for the cases where $k = 4r, 4r+1, 4r+2$, and $4r+3$.

Introducing this segmentation, and remembering that

$$W_N^{4nr} = W_{N/4}^{nr}$$

the following four equations (19) are derived from (18)

$$\begin{aligned} X(4r) = & \sum_{n=0}^{(N/4)-1} [x(n) + x(n+N/4) \\ & + x(n+N/2) + x(n+3N/4)]\, W_N^0\, W_{N/4}^{nr} \\ X(4r+1) = & \sum_{n=0}^{(N/4)-1} [x(n) - j\, x(n+N/4) \\ & - x(n+N/2) + j\, x(n+3N/4)]\, W_N^n\, W_{N/4}^{nr} \\ X(4r+2) = & \sum_{n=0}^{(N/4)-1} [x(n) - x(n+N/4) \\ & + x(n+N/2) - x(n+3N/4)]\, W_N^{2n}\, W_{N/4}^{nr} \\ X(4r+3) = & \sum_{n=0}^{(N/4)-1} [x(n) + j\, x(n+N/4) \\ & - x(n+N/2) - j\, x(n+3N/4)]\, W_N^{3n}\, W_{N/4}^{nr} \end{aligned} \qquad (19)$$

Each one of these equations is now an N/4-point FFT that can be computed by repeating the above procedure until N = 4. Note that the factors W_N^0, W_N^n, W_N^{2n}, and W_N^{3n} are considered part of the signal. In general, an N-point FFT (where N is a power of 4) can be reduced to the computation of four N/4-point FFTs by transforming the input signal x(n) into an intermediate signal y(n), as suggested by (19). Figure 17 shows the corresponding radix-4 DIF butterfly, which generates one term for each sum in (19).

For simplicity, the notation of Figure 18 is often used instead of that of Figure 17 for the butterfly of radix-4 DIF FFT.

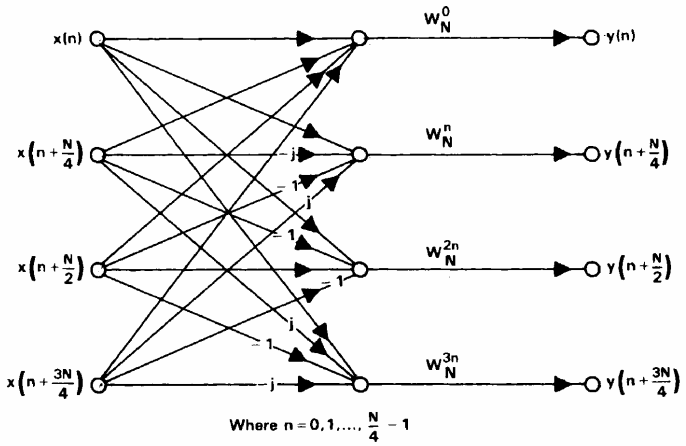


Figure 17. Radix-4 DIF Butterfly

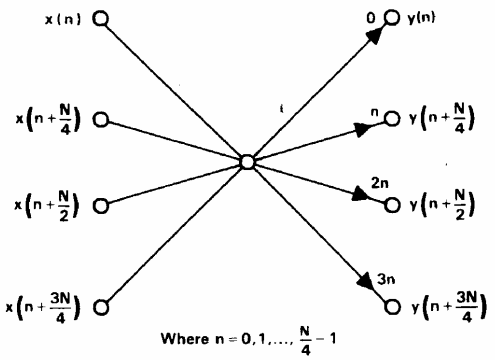


Figure 18. Alternate Form of the Radix-4 DIF Butterfly

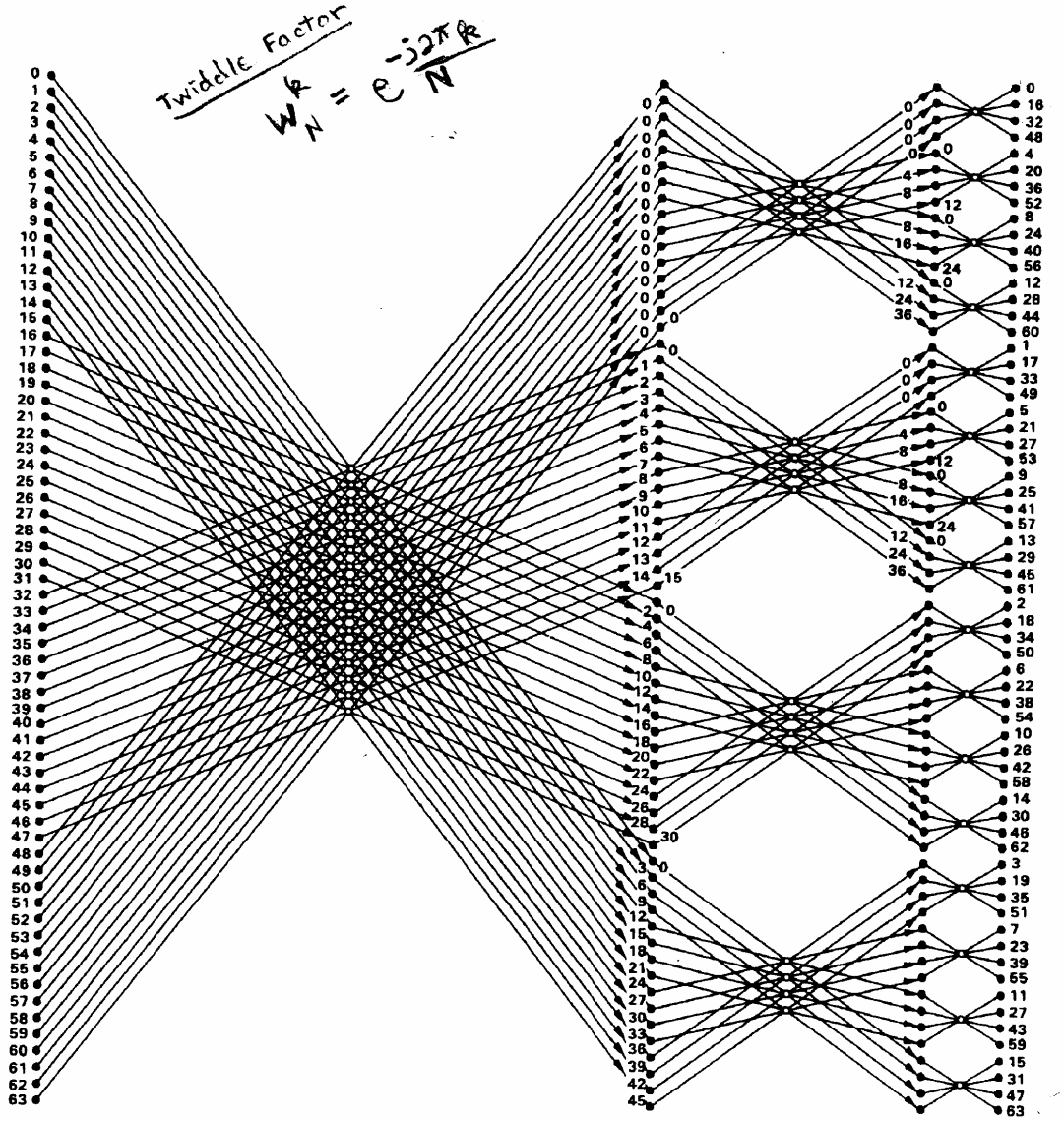
(3)

Figure 19 shows an example of a 64-point, radix-4 DIF FFT.

Note that the inputs are normally ordered while the outputs are presented in a digit-reversed order. The principle of digit reversal is the same as in radix-2 FFT, but now the digits are 0, 1, 2, and 3 (quaternary system) instead of 0 and 1 (binary system). The code for digit reversal is the same as that shown in Figure 11. For example, the datapoint occupying location 132 (quaternary number corresponding to decimal 30) exchanges positions with the datapoint at location 231 (corresponding to the decimal 45).

Another important point of the radix-4 algorithm regards scaling. Since each stage of the radix-4 algorithm corresponds to two stages of the radix-2 algorithm, equivalent results are obtained by dividing the output of each stage of the radix-4 algorithm by 4.

Appendix E contains the implementation of a 256-point, radix-4 DIF FFT on the TMS32020. This implementation follows the one described in FORTRAN code in the book by Burrus and Parks.[4]



Addendum to TI's Radix-4 FFT Article (KEH)

The "*Radix-4 Decimation-In-Frequency (DIF) FFT*" article (*Texas Instruments, 1986, SPRA012A*) begins by defining X(n) to be the DFT of an N-point input sequence, x(n), where N is restricted to be a power of 4, thus $N = 4^V$ for any integer V that is equal to or greater than 1.

This article starts with the basic definition of the discrete Fourier Transform (DFT):

$$X(k) = \sum_{n=0}^{N-1} x(n) \cdot W_N^{n \cdot k} \quad \text{where} \quad W_N = e^{-j \cdot \frac{2 \cdot \pi}{N}} \quad \text{(Twiddle Factor)}$$

From this starting point, this article shows in Eqns (19a-19d) that the original N-poir DFT calculation can be replaced by four independent N/4-point DFT calculations. E (19a) shows that every 4th point of the original N-point DFT, starting with X(0)

$$\{X(0), X(4), X(8),...X(N-4)\}$$

can be found by transforming the following N/4-point subsequence:

$$\left(x(n) + x\left(n + \frac{N}{4} \right) + x\left(n + \frac{N}{2} \right) + x\left(n + 3 \cdot \frac{N}{4} \right) \right) \cdot W_N^{0} \qquad \text{(20a)}$$

$$\text{for} \quad n = 0 .. \left(\frac{N}{4} - 1 \right)$$

In similar fashion, Eqn (19b) shows that every 4th point of the original N-point DFT starting with X(1)

$$\{X(1), X(5), X(9),...X(N-3)\}$$

can be found by transforming the following N/4-point subsequence:

$$\left(x(n) - j\,x\left(n + \frac{N}{4} \right) - x\left(n + \frac{N}{2} \right) + j\,x\left(n + 3 \cdot \frac{N}{4} \right) \right) \cdot W_N^{n} \qquad \text{(20b)}$$

$$\text{for} \quad n = 0 .. \left(\frac{N}{4} - 1 \right)$$

Likewise, Eqn (19c) shows that every 4th point of the original N-point DFT, starting with X(2)

$$\{X(2), X(6), X(10),...X(N-2)\}$$

can be found by transforming the following N/4-point subsequence:

$$\left(x(n) - x\left(n + \frac{N}{4} \right) + x\left(n + \frac{N}{2} \right) - x\left(n + 3 \cdot \frac{N}{4} \right) \right) \cdot W_N^{2n} \qquad (20c)$$

$$\text{for} \quad n = 0 .. \left(\frac{N}{4} - 1 \right)$$

Finally, Eqn (19d) shows that every 4th point of the original N-point DFT, starting with X(3)

$$\{X(3), X(7), X(11),...X(N-1)\}$$

can be found by transforming the following N/4-point subsequence:

$$\left(x(n) + j\, x\left(n + \frac{N}{4} \right) - x\left(n + \frac{N}{2} \right) - j\, x\left(n + 3 \cdot \frac{N}{4} \right) \right) \cdot W_N^{3n} \qquad (20d)$$

$$\text{for} \quad n = 0 .. \left(\frac{N}{4} - 1 \right)$$

The radix-4 DIF "Butterfly calculation" shown in Figs. 17 and 18, is the elementary calculation that must be performed N/4 times in order to transform the N original inpu points, x(n), into the four groups of points that must be individually, and separately, transformed, as indicated in Eqns (20a - 20d), to yield the (out-of-order) transformed points of the original N-point input series. The first of the N/4 Butterfly calculations corresponds to n = 0, and it produces the first entry in each of the four resulting N/4-p subseries indicated by Eqns (20a - 20d). The second Butterfly calculation correspond n = 1, and it produces the second entry in each of the four resulting N/4-point subserie The third Butterfly calculation corresponds to n = 2, and it produces the third entry in of the four subseries, etc. The final Butterfly calculation corresponds to n = N/4-1, an produces the final entry in each of the four subseries.

Fig. 19 shows an N=64-point DFT, broken down into a number of radix-4 Butterfly operations. Note that the original input points are listed (in order) on the extreme left, where the points are numbered 0 through 63. Then N/4 = 16 Butterfly operations are used to transform (or decompose) the original 64 input points into FOUR separate sets of N/4 = 16 input points, each of which may be individually transformed. The advantage of this decomposition is apparent when we recall that N^2 complex multiplications are required to evaluate an N-point DFT using its definition. Thus the original N = 64-point DFT would require $64^2 = 4096$ complex multiplications, while four N/4 = 16-point DFTs could be evaluated using only $4(16^2) = 1024$ complex multiplications (not counting the Butterfly calculations)! Note from Fig. 19 that the result of applying these 16 Butterfly operations to the original set of input values leaves us with a single 64-node vertical column near the middle of the page that containin four groups of values. If these four groups of points are run through four independent N/4 = 16-point DFT operations, this would yield the desired transformed values X(n) from the original N-point sequence, in the following order (progressing from the top to the bottom of the column), as predicted by Eqns (20):

0, 4, 8, 12, 16, 20, 24, 28, 32, 36, 40, 44, 48, 52, 56, 60
1, 5, 9, 13, 17, 21, 25, 29, 33, 37, 41, 45, 49, 53, 57, 61
2, 7, 10, 14, 18, 22, 26, 30, 34, 38, 42, 46, 50,54, 58, 62
3, 8, 11, 15, 19, 23, 27, 31, 35, 39, 43, 47, 51, 55, 59, 63

However, we can do much better than this! We can reduce the computational burden even further by not simply using the DFT definition to transform the resulting four 16-point subseries, but rather, by applying the radix-4 decomposition once again (in recursive, or hierarchical, fashion) to each of the individual 16-point DFTs.! This second level of decomposition requires 16/4 = 4 Butterfly operations for each of the four separate 16-point decompositions. (See Fig. 19.) This produces 4*4 = 16 groups of 16/4 = 4 points, each of which may be individually transformed. And once they are transformed, the ordering of the desired transform values X(n) from the original N-point sequence (as predicted by Eqns (20) is now found by taking every fourth value from the first row in the previous ordering sequence shown above, first starting at position 0, then starting at position 1, then starting at position 2, and finally starting at position 3. The remaining rows are treated in like fashion. This yields the new (and final ordering!) of:

0, 16, 32, 48, 4, 20, 36, 52, 8, 24, 40, 56, 12, 28, 44, 60
1, 17, 33, 49, 5, 21, 37, 53, 9, 25, 41, 57, 13, 29, 45, 61
2, 18, 34, 50, 7, 22, 38, 54, 10, 26, 42, 58, 14, 30, 46, 62
3, 19, 35, 51, 8, 23, 39, 55, 11, 27, 43, 59, 15, 31, 47, 63

Now, after two levels of radix-4 decomposition, we have reduced the original DFT calculation from a single 64-point DFT calculation down to evaluating sixteen 4-poi DFTs. *Note that up to this point, no DFTs have yet been evaluated!* So finally we are ready to evaluate these DFTs. How can this be accomplished as efficiently as possible? Eqns (19a - 19d) once again supplies the answer! Note that for a four poi transform (N = 4 and r takes on only one value, 0), so Eqns (19a - 19d) reduce to:

$$X(0) = x(0) + x(1) + x(2) + x(3)$$

$$X(1) = x(0) - j\cdot x(1) - x(2) + j\cdot x(3)$$

$$X(2) = x(0) - x(1) + x(2) - x(3)$$

$$X(3) = x(0) + j\cdot x(1) - x(2) - j\cdot x(3)$$

Note that the summation signs have vanished, and the multiplication by $W_{N/4}^{nr}$ has vanished! ***So the four-point Butterfly operation actually evalues the 4-point DFT, with the results being in the same order as the input points.*** This is great news!

Note the final stage in the DFT calculation depicted in Fig. 19 that consists of sixtee 4-point Butterfly operations. Note that the point ordering has not been disturbed any further.

The output point ordering turns out to have a simple "Base-4" digit-reversed relationship. For example, if we assign the topmost element to Position 0, and the next element to Position 1, etc., then we see from Fig. 19 that the output element at Position 9 is X(24). Since $N = 64 = 4^3$, each position number requires three Base-4 digits. Therefore, Position 9 would be written in 6-bit binary format as $(001001)_2$ and in "Base-4 format" (taking the bits in the binary number two at a time) as $(021)_4$. Digit-reversing this Base-4 number yields $(120)_4 = 1(4^2) + 2(4^1) + 0(4^0) = 24$, which agrees with the result we obtained. You should verify that this works for any other b position that you might choose as well.

Appendix B. Example C62x Radix-4 FFT Programs (Contained in three files: Radix4.c, Digitgen.c, and Digit.c) Written by Texas Instruments (See Application Report SPRA291, 1997)

File Radix4.c

```
#include <stdio.h>
#include <math.h>
 void R4DigitRevIndexTableGen(int, int *,unsigned short *,unsigned short *);
 void radix4(int n, short[], short[]);
 void digit_reverse(int *, unsigned short *, unsigned short *, int);
 short  w[32] = {
      32767,
          0,
      30273,
      12539,
      23170,
      23170,
      12539,
      30273,
          0,
      32767,
     -12539,
      30273,
     -23170,
      23170,
     -30273,
      12539,
     -32767,
          0,
     -30273,
     -12539,
     -23170,
     -23170,
     -12539,
     -30273,
          0,
     -32767,
      12539,
     -30273,
      23170,
     -23170,
      30273,
      -12539};
  short x[64];
 int i,count;
 unsigned short IndexI[64], IndexJ[64];
```

```
 void main(void)
 {
   for(i=0;i<32;i++) x[i]=0;
   x[0]=1000;
   x[2]=1000;
   x[4]=1000;
   R4DigitRevIndexTableGen(16, &count,IndexI,IndexJ);
   for (i=0;i<count;i++)
     printf("IndexI(%d) = %d IndexJ(%d) = %d \n",i,IndexI[i],i,IndexJ[i]);

   radix4(16,(short *)x,(short *)w);
   digit_reverse((int *)x,IndexI,IndexJ,count);
   for(i=0;i<16;i++)
   printf("X(%d) = %d + j%d\n",i,x[2*i],x[2*i+1]);
  }


/*************************************************************
File:  radix4.c FFT function   (Based on Burris, Parks p. 113)
          radix4(int n; short x[ ], short w[ ]);

      Efficiently calculates n-point DFT, defined as

            X[m] = SUM(o=0 to o=n-1){x(o)*Wn^(o*m)}

            (Where Wn = exp(j*2*Pi/n)

      n = number of points in FFT (must be a power of 4)
      x[ ] = complex input array of length 2n, where
            the kth element = x[2k] + j*x[2k+1].  Upon return
            this array holds the DFT result.
      w[ ] = complex twiddle factor array of length 2n, where
            Wn^k = = (exp(j*2*Pi/n))^k = exp(j*2*Pi*k/n)
            the kth complex element Wn^k = w[2k] + j*w[2k+1]

*************************************************************/

void radix4(int n, short x[ ], short w[ ])
  {
   int    n1,n2,ie,ia1,ia2,ia3,i0,i1,i2,i3,j,k;
   short          t,r1,r2,s1,s2,co1,co2,co3,si1,si2,si3;
   n2=n;
   ie=1;
    for (k=n;k>1;k=k>>2)
    {
     n1=n2;
```

```
    n2=n2>>2;
    ia1 = 0;
     for (j=0; j<n2; j++)
     {
      ia2=ia1+ia1;
      ia3=ia2+ia1;
      co1=w[ia1*2];
      si1=w[ia1*2+1];
      co2=w[ia2*2];
      si2=w[ia2*2+1];
      co3=w[ia3*2];
      si3=w[ia3*2+1];
      ia1=ia1+ie;
      for (i0=j;i0<n;i0=i0+n1)
        {
          i1=i0+n2;
          i2=i1+n2;
          i3=i2+n2;
          r1=x[2*i0]+x[2*i2];
          r2=x[2*i0]-x[2*i2];
          t=x[2*i1]+x[2*i3];
          x[2*i0]=r1+t;
          r1=r1-t;
          s1=x[2*i0+1]+x[2*i2+1];
          s2=x[2*i0+1]-x[2*i2+1];
          t= x[2*i1+1]+x[2*i3+1];
          x[2*i0+1]=s1+t;
          s1=s1-t;
          x[2*i2] = (r1*co2+s1*si2)>>15;
          x[2*i2+1] = (s1*co2-r1*si2)>>15;
          t = x[2*i1+1] - x[2*i3+1];
          r1=r2+t;
          r2=r2-t;
          t = x[2*i1]-x[2*i3];
          s1=s2-t;
          s2=s2+t;
          x[2*i1] = (r1*co1+s1*si1)>>15;
          x[2*i1+1]= (s1*co1-r1*si1)>>15;
          x[2*i3] = (r2*co3+s2*si3)>>15;
          x[2*i3+1] = (s2*co3-r2*si3)>>15;
        }
      }
      ie = ie << 2;
    }
}
```

File Digitgen.C

```
/*************************************************************************
 FILE
        digitgen.c - This is the C source code for a function used to generate
        index tables for a digit reversal function for a radix-4 FFT.

        Recall that the DIF Radix 4 method requires the transformed values to be bit
        reversed in a "mod-4" fashion.  Thus, in a 16-bit transform, element 1001
        must be swapped with 0110, and element 1010 would be swapped with
        1010 (and hence is not swapped at all).  Thus for an transform with
        n = 16 points, the following values would be generated by this function:

                Iindex[0] = %0001 = 1     Jindex[0] = %0100 = 4
                Iindex[1] = %0010 = 2     Jindex[1] = %1000 = 8
                Iindex[2] = %0011 = 3     Jindex[2] = %1100 = 12
                Iindex[3] = %0110 = 6     Jindex[3] = %1001 = 9
                Iindex[4] = %0111 = 7     Jindex[4] = %1101 = 13
                Iindex[5] = %1011 = 11    Jindex[5] = %1110 = 14

                count = 5   (=> 6 pairs of elements are to be swapped!)
 *************************************************************************/


void R4DigitRevIndexTableGen(int n, int *count, unsigned short *IIndex, unsigned short *JIndex)
{
        int j, n1, k, i;
        j = 1;
        n1 = n - 1;
        *count = 0;

        for(i=1; i<=n1; i++)
        {
                if(i < j)
                {
                        IIndex[*count] = (unsigned short)(i-1);
                        JIndex[*count] = (unsigned short)(j-1);
                        *count = *count + 1;
                }

                k = n >> 2;

                while(k*3 < j)
                {
                        j = j - k*3;
                        k = k >> 2;
```

```
                }
                j = j + k;
        }
}
```

File Digit.c

```
/*************************************************************************
 FILE
        digit.c - This is the C source code for a digit reversal function for
        a radix-4 FFT
 *************************************************************************/


void digit_reverse(int *yx, unsigned short *JIndex, unsigned short *IIndex, int count)
{   /* NOTE TRICK -- The value pointed at by yx (*yx) is
      declared to be of size int... thus both the short int real and imaginary part
      of each pair of transformed value are exchanged at the same time! */

        int i;
        unsigned short I, J;
        int YXI, YXJ;

        for (i = 0; i<count; i++)
        {
                I = IIndex[i];
                J = JIndex[i];
                YXI = yx[I];
                YXJ = yx[J];
                yx[J] = YXI;
                yx[I] = YXJ;
        }

}
```