transforming a series of samples taken in the time domain into a frequency domain representation is useful in a wide variety of signal processing applications. This includes direct analysis of the frequency spectrum as well as operations like convolutions, which are used in digital filtering. The usual tool for performing this transformation is the Fourier transform, or to be more specific, the fast Fourier transform (FFT). But for most applications, there is an even faster method called the Hartley transform. This article reviews some published data on the fast Hartley transform (FHT) and then develops an optimized implementation of this algorithm.

A typical FFT application starts with a series of samples taken in the time domain. For example, suppose you want to determine the vibration characteristics of an automobile engine. You could attach an accelerometer to the engine block and sample the output of the accelerometer at regular intervals. Suppose you take 4,096 samples over a one-second period. These 4,096 samples would then be transformed, using an FFT, into an array of 4,096 elements where each element specifies the vibrational energy at a particular frequency. The frequencies represented in the FFT array run from 0Hz to 4,095Hz in steps of 1Hz. So if you needed to determine the total energy in the frequency band from 200Hz to 300Hz, you could add up the amplitudes of FFT entries 200 through 300. Although this example is described using the FFT, the FHT works just as well.

When I first saw an article about the Hartley transform, I must admit that I didn’t give it serious consideration. The formulae looked similar to the ones for the Fourier transform, except that the imaginary, $i$, was missing. I thought there must be some catch. But then, years later, I ran across
that article again and decided to delve into it to see what the story was. I was so impressed with what I learned that I immediately wrote a little freeware utility for general aviation enthusiasts—a PC program to measure the RPM of an airplane engine based on the sound of that engine. The program (available at www.wnet.net/~rscott/rpmsound.html) uses the Hartley transform to find the peaks in the frequency spectrum. The pattern and location of the frequency peaks determine the RPM.

**Hartley transform defined**

The Hartley transform was first proposed in 1942 by Ralph Hartley. This is the same fellow, by the way, who invented the well-known Hartley oscillator circuit. Just as in the Fourier transform, the Hartley transform starts with a sequence of samples in the time domain. Let \( X(t) \) for \( t = 0 \ldots N-1 \) be such a sequence. The Hartley transform of this sequence is another sequence, \( H(f) \) for \( f = 0 \ldots N-1 \) given by:

\[
H(f) = \frac{1}{N} \sum_{n=0}^{N-1} X(t) \left[ \cos \left( \frac{2\pi n f}{N} \right) + \sin \left( \frac{2\pi n f}{N} \right) \right]
\]

(1)

If, for some reason, you want to undo this transform, the formula is almost identical, except for the absence of the \( 1/N \) factor:

\[
X(t) = \sum_{f=0}^{N-1} H(f) \left[ \cos \left( \frac{2\pi n f}{N} \right) + \sin \left( \frac{2\pi n f}{N} \right) \right]
\]

The first thing that made me suspicious about these formulae was that the sum of the cosine and the sine of an angle is just another phase-shifted sine function (shifted by 45 degrees, to be exact). What could be so magical about correlating against a sine function that was shifted by an arbitrary amount? Well, the answer is “nothing.” In fact, it is possible to define a whole class of reversible transforms based on other phase-shifted sine functions. But the specific sine function used in the Hartley transform exhibits a certain symmetry between the transform and its inverse. And when it comes to implementing the transform, we will see that we end up taking pure sines and cosines anyway and not their sums.

**Comparisons with the FFT**

In a 1988 article in *Byte*, Mark O’Neill gave formulae similar to the ones shown above; but note the typo in his Equation 5, which I have corrected. In addition to introducing the Hartley transform, O’Neill went on to develop the fast Hartley transform, based on a 1984 article by Ronald Bracewell in *Proceedings of the IEEE.*

---

**FIGURE 1 A single FHT butterfly**

The results of recursion Equations 3 and 4 are combined in this data-flow diagram. Blue represents terms with a cosine factor. Red represents terms with a sine factor. Black represents terms with no trig factors.

\[
\begin{align*}
H_0(f) + H(f) \\
H_1(f) + H(f+N/2) \\
H_1(N-f)
\end{align*}
\]

**FIGURE 2 FHT data flow**

A 16-point sequence, \( X \), generates its Hartley transform, \( H \). The lines represent instances of the single butterfly calculation from Figure 1. Lines have been omitted for null terms.
While the Fourier transform works in the complete generality of complex-valued sequences, the Hartley transform only works with real-valued sequences. This should not be a problem for most real-world applications, which are limited to real-valued time domain samples anyway. To show that nothing is lost in using the Hartley transform, O’Neill gave the following formulae for the real and imaginary parts of the Fourier transform in terms of the Hartley transform:

\[ F_{\text{real}}(f) = H(f) + H(N - f) \]
\[ F_{\text{imag}}(f) = H(f) - H(N - f) \]  

It turns out that using the Hartley transform to compute the Fourier transform is faster than computing the Fourier transform directly. This is an important use of the Hartley transform because the Fourier transform has immediate natural interpretation while, as far as I know, the Hartley transform does not. Each complex value in the Fourier transform represents the amplitude and phase of a particular frequency component in the frequency spectrum of a time domain sequence. Often only the amplitude is interesting, so applications will then form the array of magnitudes of the complex-valued Fourier series. The implementation of the Hartley transform described here provides for a direct calculation of this array of amplitudes.

Note that Equation 2 leads to out-of-range indices when \( f = 0 \), since there is no \( H(N) \). The interpretation of the indices must, therefore, be extended modulo-\( N \). Therefore, \( H(N) \) takes the value \( H(0) \). And, in fact, this modulo-\( N \) interpretation of the indices is supported by the formulae that define the Hartley transform, which give the same values if you add \( N \) to the indices.

If you have ever used the Fourier transform to process real-valued data, you may have been aware of a sense of wasted effort. To begin with, you prob-

---

**Listing 1** Working implementation of FHT

```c
// Fast Hartley Transform - optimized for speed
#include <math.h>
typedef float DataType;

void InitTrigTable(DataType *TrigPtr, int nBits)
// Pre-compute sine function for first quadrant only
{
    unsigned int dataNdiv4,i;
    DataType RadianIncrement;

dataNdiv4 = 1 << (nBits-2);
RadianIncrement = 3.14159265358979238 / 2 / dataNdiv4;
for(i=0; i<dataNdiv4; i++)
    TrigPtr[i] = sin( RadianIncrement * i );
}

DataType *FastHartleyTransform( DataType *DataArray,
    DataType *WorkArray,
    DataType *TrigPtr,
    int nBits )
// DataArray[0...N] are to be transformed. Results will be in either
// DataArray[] or WorkArray[]. This function returns a pointer to
// whichever array it is. TrigPtr is a pointer to a pre-computed
// table of sines for angles in the first quadrant. If there are
// dataN elements in the first two arrays, then there are dataN/4
// elements in TrigPtr[, and nBits is the log base 2 of dataN.
// dataN must be a power of 2 and be at least 16.
{
    DataType *FromArray,*ToArray,*SwapTemp;
    DataType *From,*To,*FromRetrograde;
    DataType *To_ArrayEnd,*To_QuadrantEnd;
    DataType *SinPtr,*CosPtr;
    unsigned int ButterflyBlocks,ButterflyWidth;
    unsigned int dataN,baseN,TrigArraySize;
    unsigned int RegularBase,ReversedBase;

dataN = 1 << nBits;
TrigArraySize = dataN/4;
From = DataArray;
To = WorkArray;

    // Reorder the From[] array and place it in the To[] array...
    // according to bit-reversed indices.

baseN = dataN/16;
for(RegularBase=0; RegularBase<baseN; RegularBase++)
{
    unsigned int RemainingRegularBase;
    Listing 1 continued on p. 132
```

---

130 SEPTEMBER 2000 Embedded Systems Programming
ably stuffed your real-valued sequence into the real parts of an array of complex numbers and zeroed out the imaginary parts. Having all those zeroes using up space but providing no information always bothered me. Then, when the Fourier transform was done, the resulting array of complex numbers had a lot of redundancy. In particular, you never had to look past the midpoint of the array (the so-called Nyquist frequency) because the frequency domain data in the second half of the array was a kind of mirror image of the first half. The complex-valued $F(N-f)$ is just the complex conjugate of $F(f)$. (This is only true for the Fourier transform of real-valued sequences. In general, the Fourier transform has no such redundancy.) When I saw how the power of the Fourier transform was being “wasted” on real-valued sequences, I finally understood how something like the Hartley transform could be more efficient.

Butterflies

Now let’s get down to business and see how the Hartley transform can be implemented efficiently as the FHT. The development is similar to that of the FFT, which is not presented here but can be found in any textbook on digital signal processing.

The heart of the algorithm is a computation whose data-flow diagram looks like a butterfly. But unlike the symmetric butterfly of the FFT, the FHT butterfly, as shown in Figure 1, is a bit ugly. It has a vestigial third wing.

From now on we will be assuming that $N$ is a power of two. Since $N$ is even, the time sequence, $X(t)$, which appears in the definition of the Hartley transform (Equation 1), can be divided into two intertwined sequences, $X_0$ and $X_1$, given by:

$$X_0(t) = X(2t)$$
$$X_1(t) = X(2t+1)$$

That is, sequence $X_0$ consists of the even-indexed values from sequence $X$ and sequence $X_1$ consists of the odd-indexed values.
indexed values. As before, we will consider all indices to be interpreted modulo-$N$. But since $X_0$ and $X_1$ are defined in terms of $2^t$, they actually repeat modulo $N/2$. So consider $X_0$ and $X_1$ as sequences of length $N/2$ and let $H_0$ and $H_1$ be their Hartley transforms. Let $M$ stand for $N/2$. Then, by the definition of the Hartley transform in Equation 1 we have:

$$H(f) = \frac{1}{M} \sum_{n=0}^{N-1} X_n(f) \left[ \cos \left( \frac{2\pi f}{M} \right) + \sin \left( \frac{2\pi f}{M} \right) \right]$$

$$H_0(f) = \frac{1}{M} \sum_{n=0}^{N/2-1} X_n(f) \left[ \cos \left( \frac{2\pi f}{M} \right) + \sin \left( \frac{2\pi f}{M} \right) \right]$$

I leave it as an exercise for the reader to verify that:

$$H(f) = \frac{1}{2} \left\{ H_0(f) + H_1(f) \cos \left( \frac{2\pi f}{N} \right) \right\} + \frac{1}{2} \left\{ H_0(f) - H_1(f) \sin \left( \frac{2\pi f}{N} \right) \right\}$$

(If you are reading the Byte article, note that this equation is incorrectly shown with $N_s$ instead of $N$.) It took me about half an hour to verify this equation with paper and pencil, using only the trig identities:

$$\sin(A)\cos(B) = \frac{\sin(A + B) + \sin(A - B)}{2}$$
$$\sin(A)\sin(B) = \frac{\cos(A - B) - \cos(A + B)}{2}$$
$$\cos(A)\cos(B) = \frac{\cos(A - B) + \cos(A + B)}{2}$$

If you want to develop an appreciation of the basis of the Hartley transform, I recommend performing the exercise.

Keeping in mind that the Hartley transforms on the right side of Equation 3 repeat modulo $N/2$, we can recognize the following symmetry:

$$H\left( f + \frac{N}{2} \right) = \frac{1}{2} \left\{ H_0(f) - H_1(f) \cos \left( \frac{2\pi f}{N} \right) \right\} - \frac{1}{2} \left\{ H_0(f) + H_1(f) \sin \left( \frac{2\pi f}{N} \right) \right\}$$

Equations 3 and 4 taken together
define the complete FHT butterfly computation, as diagrammed in Figure 1. These two equations express the Hartley transform of length \( N \) in terms of two smaller (half-length) Hartley transforms. This is the beginning of a recursive process. At each step in the recursion we recognize that all of the Hartley transforms from the previous step can be further broken down into Hartley transforms of half the length. The process continues until we have only Hartley transforms of length 1, which can be trivially computed from Equation 1.

At this point I will abandon mathematical rigor and proceed by example alone. Suppose \( N = 16 \). Figure 2 shows a data-flow diagram that illustrates the recursive decomposition of a Hartley transform of length 16. Although the data-flow diagram is computed from left to right, we shall analyze it from right to left. On the far right, the numbers 0 to 15 represent the values of \( H(0) \) through \( H(15) \), which is the Hartley transform of the original sequence \( X(0) \) through \( X(15) \).

If you let \( f \) run from 0 to 7 in Equations 3 and 4, you can see how the 16 values of \( H \) can be computed from the two smaller Hartley transforms, \( H_0 \) and \( H_1 \), each of length 8. The right-most column of data-flow lines in Figure 2 illustrates this process as the overlay of eight instances of the butterfly diagram from Figure 1. The lines have been omitted where the corresponding trig factor is zero. In the column of numbers to the left of the final butterfly diagrams, the two sets of indices, 0 through 7, represent the value of \( H_0 \) and \( H_1 \) with \( H_0 \) on top. For example, the first three instances of Equations 3 and 4 in this column of butterflies is:

\[
\begin{align*}
H(0) &= \frac{1}{2} [H_0(0) + H_1(0)\cos(0) + H_1(0)\sin(0)] \\
H(8) &= \frac{1}{2} [H_0(0) - H_1(0)\cos(0) - H_1(0)\sin(0)] \\
H(1) &= \frac{1}{2} [H_0(1) + H_1(1)\cos(\pi/8) + H_1(1)\sin(\pi/8)] \\
H(9) &= \frac{1}{2} [H_0(1) - H_1(1)\cos(\pi/8) - H_1(1)\sin(\pi/8)] \\
H(2) &= \frac{1}{2} [H_0(2) + H_1(2)\cos(\pi/4) + H_1(2)\sin(\pi/4)] \\
H(10) &= \frac{1}{2} [H_0(2) - H_1(2)\cos(\pi/4) - H_1(2)\sin(\pi/4)]
\end{align*}
\]

Note that the indices of \( H_0 \) and \( H_1 \) have been reduced modulo 8, since they are Hartley transforms of length 8.

The decomposition then continues with the next column to the left, showing how \( H_0 \) and \( H_1 \) are each repre-
sented in terms of Hartley transforms of length 4. $H_0$ is decomposed into $H_{00}$ and $H_{01}$, while $H_1$ is decomposed into $H_{10}$ and $H_{11}$. The numbers 0 through 3 are indices into these smaller Hartley transforms.

The recursion process continues all the way back to the first column, where we would have 16 Hartley transforms of length 1. Equation 1 shows that a Hartley transform of a time sequence of length 1 is just the original time sequence itself. So the first column represents values from the original time sequence $X(t)$. The numbers in that column are the indices into the original time sequence, $X(t)$. But how did they get into that strange order?

To answer this question we must determine exactly which time sequence is transformed into each of the Hartley transforms in Figure 2. Starting again at the right, we know that the full Hartley transform is derived from the original time sequence $X(0)$ through $X(15)$ in normal order. Going back one step in the recursion and recalling how the smaller Hartley transforms in Equations 3 and 4 were defined, we see that $H_1$ is the Hartley transform of the sequence $X(0), X(2), \ldots, X(14)$ and $H_0$ is the Hartley transform of the sequence $X(1), X(3), \ldots, X(15)$. Purely as a computational convenience, we place the time sequence for $H_1$ after the time sequence for $H_0$. We have effectively sorted the sequence, $X$, according to the least-significant bit of its indices.

In going from Hartley transforms of length 8 to Hartley transforms of length 4, we further refine the sort on $X$ based on the next-least-significant bit of its indices.

In Listing 1, we present a working implementation of the Fast Hartley Transform (FHT). The code is designed to efficiently compute the Hartley transform of a given time sequence.

```c
while(1)
{
    From++; To++; FromRetrograde--;
    if (To == To_QuadrantEnd) break;
    CSTerm = *SinPtr * FromRetrograde[0] -
             *CosPtr * FromRetrograde[ButterflyWidth];
    To[0] = From[0] + CSTerm;
    To[ButterflyWidth] = From[0] - CSTerm;
    SinPtr -= ButterflyBlocks;
    CosPtr += ButterflyBlocks;
}
```

```c
void PowerSpectrum(DataType *power, DataType *dht, int nBits)
// converts Hartley transform dht[0...N-1] into power spectrum
// power[0...N/2]. The power spectrum array is one entry longer than
// half as long as the DHT because it is symmetrical after the
// mid-frequency anyway. It is OK to call with power=dht and use
// only the first N/2+1 elements of dht on return.
{
    unsigned int N,i,NminusI;
    N = 1 << nBits;
    for(i=0; i <= N/2; i++)
    {
        NminusI = (i==0)? 0 : (N-i);
        power[i] =
            sqrt( (dht[i] * dht[i] + dht[NminusI] * dht[NminusI]) / 2 );
    }
}
```

```c
void ScaleHartleyTransform(DataType *data, int nBits)
{
    unsigned int i,N;
    DataType Nrecip;
    N = 1<<nBits;
    Nrecip = 1 / (DataType)N;
    for(i=0; i<N; i++)
        data[i] *= Nrecip;
}
```

At each stage in this progressive sort, the results of the previous sort are preserved. Therefore when we
get back to the beginning of the data-flow diagram, we have the original sequence, \( X \), listed in an order where the significance of the bits in the indices has been reversed. This is called bit-reversed order, and it is just as applicable to the FFT as it is to the FHT. Thus, to perform the FHT, we must first reorder the sequence, \( X \), into bit-reversed order. Then we perform the butterfly computations as they are diagramed in Figure 2.

### Implementation

If you are familiar with the structure of the FFT, you will recognize similarities in the FHT. One important difference is that you cannot do the FHT “in place.” Computing the transform “in place” means that your initial array of time domain values is used to hold all the intermediate calculations and the resulting transform ends up in the same array. This is possible with the FFT because the butterfly operations all have two inputs and two outputs which are in the same array location as the inputs. You can load your time domain sequence into an array and then perform butterfly calculations two elements at a time. The reason this is not possible with the FHT is that the butterfly operation takes three inputs to produce two outputs. If the outputs are immediately stored in the original array, then an input to some other butterfly operation will get overwritten before it is used.

When we calculate the FHT, we must have two arrays. Each round of calculations uses one array for input (the FromArray in the code) and the other array for output (the ToArray in the code). Which array the final transform ends up in depends on whether you had to do an even or an odd number of butterfly calculation rounds. In any case, this inability to do a transform in place contradicts the opening claim of O’Neill’s article that only half the memory is needed for the FHT. It is true that the FHT uses only real-valued arrays instead of the complex-valued arrays needed by the FFT, and so the arrays use half as much memory. But you need two of those arrays with the FHT and only one array with the FFT, therefore the memory usage is a wash.

In general, each round of calculations requires \( N/2 \) evaluations of
Equations 3 and 4. Each such evaluation involves a sine, a cosine, two multiplications, and two additions. To make things quick, the trig values can be in pre-computed lookup tables. Additionally, we recognize that in each butterfly calculation, the terms in Equation 3 are the same as the terms in Equation 4, except for a difference in signs. Referring to the example of \( N=16 \), this allows us to calculate terms \( H(0) \) through \( H(7) \) and get \( H(8) \) through \( H(15) \) almost for free. This also means that we only need values of sine and cosine for angles from zero to just under 180 degrees. In fact, I choose to implement the smallest possible trig lookup table, namely, the values of \( \sin(x) \) in the first quadrant (zero to 90 degrees). The values of sine in the second quadrant as well as the values of cosine in both quadrants will come from this table using the trig identities:

\[
\begin{align*}
\sin(x + \pi/2) &= \sin(x) \\
\cos(x) &= \sin(\pi/2 - x) \\
\cos(x + \pi/2) &= -\cos(x)
\end{align*}
\]

Thus to perform a Hartley transform of length \( N \), we only need to precompute the \( N/4 \) values of \( \sin(x) \) in the first quadrant. In order to simplify accessing this table, I have separate code for the first and second quadrants. The third and fourth quadrants are done as a byproduct using the symmetry noted in Equation 4.

Let’s jump right to the heart of the implementation. The inner-most loop is essentially:

```c
while(in first quadrant)
{
    From++; To++; FromRetrograde--; 
    SinPtr += ButterflyBlocks; 
    CosPtr -= ButterflyBlocks; 
    CSTerm = *CosPtr * 
        From[ButterflyWidth] + *SinPtr * FromRetrograde[0]; 
    To[0] = From[0] + CSTerm; 
    To[ButterflyWidth] = From[0] - CSTerm; 
}
```

This has been extracted and simplified from Listing 1. The actual implementation of Equation 3 is in the calculation for \( To[0] \). The calculation for \( To[ButterflyWidth] \) shows the use of the symmetry of Equation 4. As you can see, \( CSTerm \) only needs to be calculated once for each pair of Hartley transform values. Both \( SinPtr \) and \( CosPtr \) point into the pre-computed sine lookup table, but they run in opposite directions to give the sine and cosine. The values for \( ButterflyWidth \) and \( ButterflyBlocks \) change for each round. \( ButterflyWidth \) is the index offset between the two outputs of the butterfly. It starts at four for round three and doubles in each round. \( ButterflyBlocks \) is the number of Hartley transforms on the output side of each round. It is cut in half after each round, ending up at one for the last round.

I should point out here that the code actually computes double the Hartley transform at each row because we ignore the factor of 1/2 in Equations 3 and 4. This makes our final result too large by a factor of \( N \). In many applications this error is irrelevant since only relative values are used. But if correct scaling is required, a separate scaling function, \( \text{ScaleHartleyTransform()} \), is provided in the code. Without calling this function you actually get the reverse Hartley transform.

You will notice that two of the three terms in Equation 3 involve indices that increment as the target index increments. But the third term in that equation uses a backwards-running index. This has been called the “retrograde index” in the literature. It is what I call the ugly third wing of the
FHT butterfly. It is represented by FromRetrograde in the code.

Bit reversal and special cases
The transform code starts by permuting the input array according to the bit reversal pattern described earlier. This could take a lot of time if done in a brute-force fashion, peeling off one bit at a time. I have chosen to do it four bits at a time by using a pre-computed table of 16 bit-reversed nibbles. I generate all N bit-reversed pairs and, for each pair I interchange and transfer the input array values to the other array if the pair of indices is increasing or equal. If I did not check the relative magnitude of the paired indices, we would be transferring most values twice, which would not be bad but would waste some time. Note that the bit reversal could have been done in place, but the other work array was available so I used that instead. Because of the assumptions made in this optimized bit reversal, the implementation is limited to transforms of sequences whose length is at least 16.

The first two rounds of the FHT are so simple that I do them all at once as a special case. They involve only degenerate trig values (0 or 1). Also, the first computation of every set of butterflies is special because it has no natural interpretation as a frequency spectrum. The most natural use for the Hartley transform is as a means to develop the Fourier transform, as shown by Equation 2. The magnitude of the first half of the Fourier transform is calculated by the PowerSpectrum() function.

Performance
Classically, FFT performance has been evaluated by counting the number of multiplications, additions, and subtractions that are involved. In these terms, the FHT does very well. If we disregard the simplicity of the first two rounds, we have each round requiring N multiplications and 2N additions or subtractions. The number of rounds is \( \log_2 N \). By comparison, the FFT requires 2N multiplications and 7N/2 additions or subtractions in each round. In both cases these numbers assume that no redundant operations are performed.

But this may not tell the whole story. On modern processors with floating point capability, the time spent in computing indices and other "overhead" operations may actually take more time than all the multiplications, especially in some of the demonstration programs that are given as examples of a Hartley implementation (as in O’Neill’s).

The implementation in Listing 1 is meant as a real working implementation, not just a demonstration. I admit that the cost-savings of performing the first two rounds as a special case does become a smaller percentage of the whole as the number of points in the sequence increases. But the savings gained by minimizing the number of function calls and parameters passed in the inner-most loop is significant for transforms of any length.

If the Hartley transform is used only as a means to develop the Fourier transform, the cost of converting to the Fourier transform using Equation 2 should be taken into account in performance comparisons. This will add 2N additions to the total number of operations if the FHT is used to generate the FFT. However, if the goal is to develop the power spectrum, which is the magnitude of the complex Fourier transform values, then there is no additional cost to the FHT. The operations in the PowerSpectrum() function amount to no more than what would be required to perform the magnitude calculations on the Fourier transform values.

By the use of a typedef, the code in Listing 1 can be made to operate on either doubles or floats. Table 1 contains some benchmarks run on a 75MHz Pentium.

Doing the job quickly
Whenever real-valued sample sequences need to be transformed into the frequency domain, the fast Hartley transform does the job in about half the time of the fast Fourier transform. The FHT algorithm is simple enough to code that there is little reason to resort to third-party libraries. With direct control over the implementation you can achieve optimizations that surpass the typical demonstration implementation.

Robert Scott is the president of Real-Time Specialties, which provides consulting services particular to embedded systems in the automotive industry. He can be reached at rscott@wwnet.net.

The code in this article is available at www.embedded.com/code/htm.

References