OPTIMIZED FAST HARTLEY TRANSFORM FOR THE MC68000 WITH APPLICATIONS IN IMAGE PROCESSING

A THESIS
SUBMITTED TO THE FACULTY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE
DEGREE OF

MASTER OF SCIENCE

BY

A. ARLO REEVES

THAYER SCHOOL OF ENGINEERING
DARTMOUTH COLLEGE
HANOVER, NEW HAMPSHIRE

MARCH, 1990

Examining Committee:

_____
Professor Eric Hansen

_____
Professor Charles Daghlian

_____
Dean of Graduate Studies

_____
Professor Ian Baker

A. Arlo Reeves

THAYER SCHOOL OF ENGINEERING

DARTMOUTH COLLEGE

OPTIMIZED FAST HARTLEY TRANSFORM FOR THE MC68000 WITH APPLICATIONS IN IMAGE PROCESSING

A. ARLO REEVES

MASTER OF SCIENCE

ABSTRACT

Although Fourier transform techniques have seen wide use in image processing since the development of the Fast Fourier Transform algorithm in 1965, the process remains computation intensive and is normally used only by those with large computers or specialized hardware. This work describes the development of code which takes advantage of recent advancements in transform algorithms and microprocessor technology to make frequency domain image manipulation available and convenient for the personal computer user. Specifically, a one dimensional, decimation-in-time Fast Hartley Transform algorithm and a host of support routines have been coded and optimized in MC68000 assembly language using integer arithmetic. This code was then incorporated into *Image*, a full-featured image processing program for the Macintosh II developed at the National Institutes of Health. The resulting program can compute and display the power spectrum of a 256x256 pixel image in 10 seconds and allows frequency domain editing and inverse transformation. Dyadic frequency domain operations are also supported, providing convolution and correlation capability. All of these features are made particularly accessible through a consistent implementation of the Macintosh user interface. To illustrate *Image*'s utility in frequency domain image processing applications, several examples are presented.

ACKNOWLEDGEMENTS

I would like to thank Charles Daghlian for supporting me in this endeavor from the start. Thanks, too, to Adam Erickson who undertook a hardware implementation of the fast Hartley transform as I developed my software; our discussions often took us right down to the bits and this was very helpful. Mark Valence was always there to patiently answer my questions regarding the complex workings of the Macintosh Toolbox. Eric Hansen provided many informative discussions and the creation of Thayer's first course in image processing. Without the help of these people, completing this project would not have been possible.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF ILLUSTRATIONS

For
Hanneli

0. INTRODUCTION

Since its development in 1807 by Baron Jean-Baptiste-Joseph Fourier, the Fourier transform has seen countless applications ranging from Fourier's original studies in heat conduction to modern molecular modeling [1]. While it provided a powerful analytical tool, however, numerical calculation of the Fourier transform remained too computation intensive for most applications until 1965. In that year, James Cooley and John Tukey introduced an efficient algorithm for the calculation of Fourier transforms known as the Fast Fourier Transform or FFT[1] [2]. Because it offers a vast savings in execution time over direct Fourier transform calculation[2], the FFT has revolutionized many facets of scientific analysis and earned a place among the greatest numerical methods developed in the 20th century [4].

As a prism decomposes white light into a spectrum of its constituent colors, the Fourier transform decomposes a time- or space-varying signal into a spectrum of its constituent frequencies. The ear continuously performs a Fourier transform by converting complex time-varying sound waves into a series of volumes at distinct pitches [1]. The spectrum produced by the Fourier transform process is a complete and complementary representation of the original signal; the utility of this representation lies both in the alternative perspective it provides and in the fact that some operations are more easily performed on the spectrum than on the original signal.

While the FFT expedites Fourier transform calculations, it still remains a computation intensive task. Large FFT calculations, such as the two dimensional FFTs often used in image processing applications, require so much computation that they are most often run on powerful mainframes or specialized hardware. For example, engineers at Dartmouth's Thayer School of Engineering found that instead of waiting for a two dimensional FFT calculation to complete on their Sun 3/160, it was quicker to upload the image to a powerful Convex computer, perform the

[1]Since that time it has been found that the foundations for the FFT were laid in 1942 by Danielson & Lanczos [3], who in turn based their work on that of Runge [4]. Since they were the first to implement this technique on a digital computer, however, Cooley and Tukey are still considered the fathers of the FFT.

[2]The FFT requires $O(N\log_2 N)$ operations, while direct computation requires $O(N^2)$ operations.

FFT calculation there, and download the result[3]. Those who don't have access to such resources must show some patience: Recent research using a 68000 based VICOM Digital Image Processor required scientists to wait 30 minutes for a 512x512 point FFT calculation to complete [5]. Because most people lack either the computational resources required for rapid two dimensional FFT calculations or the patience necessary when such resources are absent, such calculations remain a tool of the privileged few.

Microprocessors have developed to the point, however, where many personal computers boast more computational power than yesteryear's mainframes. FFT algorithms have also evolved to provide the same functionality for a significantly reduced number of operations. This work describes how one such algorithm, the fast Hartley transform (FHT), was carefully coded and optimized to run on the 68000 family of microprocessors, resulting in code that can compute the power spectrum of a 512x512 point FFT in less than 30 *seconds* on a Macintosh IIci. To make this fast FFT capability available to every Macintosh II owner, the code was incorporated into a popular public domain image processing program for the Macintosh II. Careful attention was paid to the user interface extensions made, resulting in a program that provides FFT capabilities of both speed and accessibility unprecedented on a personal computer.

This paper begins by discussing the early decisions that shaped the development of this project. The Fourier and Hartley transforms are then described in more detail, and the fast Hartley transform algorithm is introduced. Next, the details of implementation that speed the FHT on the 68000 are related as are the details of the utility routines, each followed in turn with execution timing results. The next section elaborates on the incorporation of an FFT capability into the *Image* program and the feature set these extensions provide. Finally, sample images are enhanced to illustrate the power of Fourier transform techniques and the way in which *Image* brings this capability to the user. The paper concludes with suggestions for future development.

---

[3]Eric Hansen, private communication.

1. THE FIRST DECISIONS

Several decisions made early on in the course of this project fundamentally shaped its development and outcome. The first was to implement the transform code produced in *Image*, an existing image processing program for the Macintosh II. Because it promised to be most efficient, integer computation was chosen over floating point computation. Finally, an appropriate transform algorithm had to be chosen. Because of their importance, all three of these decisions are described in detail here.

1.1 Choosing an Existing Platform: The Image Program

Image is a full featured image processing program for the Macintosh II produced by Wayne Rasband at the National Institutes of Health. In his words, *Image* can [6]

> acquire, display, edit, enhance, analyze, print, and animate images. It reads and writes TIFF, PICT, and MacPaint files, providing compatibility with many other Macintosh applications, including programs for scanning, processing, editing, publishing, and analyzing images. It supports many standard image processing functions, including histogram equalization, contrast enhancement, density profiling, smoothing, sharpening, edge detection, and noise reduction. Spatial convolutions, with user defined kernels up to 63x63, are also supported.

This is only the beginning of a long list of *Image*'s capabilities. What makes *Image* even more attractive, however, is that it is distributed with its source code (in THINK Pascal) in the public domain. Anyone can download the latest version of *Image*, its documentation and source code from a number of publicly accessible computers and bulletin board services for free[4].

Because of *Image*'s singular performance/price ratio in a market otherwise dominated by pricey packages, it has gained a wide following. *Image* is also popular because users can modify the program to fit their individual needs. Several of these modifications have made their way back to the National Institutes of Health for permanent incorporation into the *Image* program. The FFT extensions to *Image* described here are one such example.

Despite all its virtues, *Image* did not originally provide the ability to take the Fourier transform of an image. Nor, because of the FFT's reputation of being everything but fast on personal computers, was this modification forthcoming. By already providing so much

[4]The *Image* documentation [6] includes a list of these sources.

functionality, however, *Image* provided an ideal platform on which to build FFT code. Any extension to *Image* automatically has the benefit of its support; because *Image* was already a 'complete' program, all effort could be focused on the FFT extensions with the assurance that their utility would be enhanced by their environment.

The success of this endeavor was by no means certain at the outset. Consequently, a further benefit of using *Image* as a development platform was not anticipated: its users. Already familiar with *Image*'s environment, *Image* users have taken quickly to the FFT extensions and are eager to use the FFT source code soon to be released into the public domain.

1.2 Choosing a Number System

Among personal computers, the Macintosh II is unique in that every unit comes equipped with a floating point coprocessor[5]. The MC68881 and MC68882 floating point units (FPUs) have specialized hardware that speed some floating point operations by over two orders of magnitude[6]. Since the fast Fourier transform is commonly known to involve many floating point computations, one is tempted to exploit the Macintosh II's FPU to accelerate FFT calculations. It may therefore come as a surprise to some that the quickest FFT calculations actually eliminate use of the FPU.

While the FPU can compute logarithms, transcendentals and other complex functions much more quickly than the software routines otherwise used by the host processor, none of these complex operations are needed to compute the FFT[7]. The basic routines, such as multiplication and addition, which make up the bulk of the transform calculation, actually take longer to compute on the FPU than on the host processor because the FPU operates on 80 bit floating point numbers while the host processor operates on 16 and 32 bit integers. Numbers moved to/from the FPU also undergo automatic conversion to/from its 80 bit format, making CPU to

---

[5]On IBM PCs and PC-compatibles, a floating point unit (FPU) is typically offered as an option. The Macintosh II comes with a 68020/68881 CPU/FPU processor set, while the Macintosh IIx, IIcx, IIci and IIfx come with 68030/68882 processor sets.

[6]Transcendental operations are accelerated most, since they are otherwise computed in software.

[7]Actually, the transcendental functions sine and cosine are needed to compute the Fourier transform, but usually their values are stored in a lookup table to speed execution.

FPU data transfers even slower that transfers between the CPU and main memory. Finally, a single precision real number occupies twice as much memory as a 16 bit integer.

To speed the execution of the transform calculation and minimize its memory demands, we therefore chose to perform all calculations using integer arithmetic. Because integers have a very limited dynamic range in comparison to real numbers, special care had to be taken to preserve as much information during the transform computation as possible. The details of these techniques are described under 'Making the FHT Accurate', below.

1.3 Choosing An Algorithm

In the last twenty five years, an abundance of new algorithms for efficient calculation of the Fourier transform have been developed and it continues to be an area of active research. Which algorithm is best? The answer to this question depends on the application at hand.

Choosing the Macintosh platform placed several constraints on the code to be developed. First, while a Macintosh II can hold 32 megabytes of RAM on the motherboard, most Macintosh II's have somewhat less memory and *Image* was designed for a minimum configuration of 2 megabytes of RAM. Since RAM promised to be a critical constraint, it was important that the calculation require a minimum of RAM. The fact that the Cooley-Tukey FFT algorithm is computed in place (the transform output is computed and stored in the same memory used for data input) made it and other algorithms with this characteristic particularly appealing. The Winograd Fourier transform [7], although very efficient in the number of multiplications it requires, is not computed in place. This, and the fact that fixed point implementations of the Winograd transform have inferior error characteristics [8], eliminated it from consideration here.

Although the simplest and most common way to compute the two dimensional Fourier transform of a matrix is to apply a one dimensional Fourier transform to each of the rows and columns of the matrix, this technique is not optimal. Vector radix transform algorithms exploit the multi-dimensionality of a given data set to reduce the number of operations required to compute its Fourier transform. For example, in two dimensions a vector radix-4 Fourier transform algorithm requires less than half of the multiplications required by the row-column technique [9,10]. Since

vector radix algorithms also use in place calculation, their efficiency singled them out for implementation and many days were spent studying and coding this technique.

Unfortunately, a small number of multiplications alone does not a fast transform make [11]. High level language implementations of the vector radix algorithms were quite complex in comparison to their one dimensional counterparts; while the number of multiplications required for transform computation were reduced, the operations required for address computation increased resulting in an overall performance only moderately better than for the row-column technique. While the address computations could probably have been optimized for better performance, another characteristic of the vector radix technique caused it to fall out of favor.

The vector radix technique's efficiency arises from the fact that it operates on the transform data in large blocks, reducing both the number of loop iterations and the total operation count involved in the computation. Unfortunately, not all of the variables in a such block can fit in the 68000's register set, forcing them to be stored in main memory instead. Since going 'off chip' to fetch a number from memory takes about 5 times longer than fetching a number directly from a register, performance of the vector radix technique was compromised by time consuming memory movement operations.

In seeking a quantitative metric by which to evaluate the performance of various Fourier transform algorithms, the scientific community has focused on the number of arithmetic operations required by a particular algorithm. Multiplication and division were invariably implemented in microcode on older processors (even the 68000 is built this way) making these iterative instructions the slowest in the instruction set. For this reason, the performance of early fast Fourier transform implementations was often truly 'multiplication bound' and an algorithm's performance correlated well with its arithmetic operation count.

As the density of semiconductor devices increases, however, operations like multiplication are no longer constrained to exist in microcode. Many newer processors have dedicated parallel multiplication units, reducing the time required for this instruction considerably. Even in the 68000 family of microprocessors, the execution time for a 16 bit multiplication has been reduced from 70 cycles on the 68000 to 28 on the 68030. At this speed, multiplication no longer stands out as the slow poke of the instruction set, but executes in a time comparable to many other

instructions. Arithmetic operation count, although still widely used as a basis for algorithm comparison, is therefore no longer as useful a metric in the evaluation of algorithm performance. For example, a 68000 implementation of Despain's algorithm [12], which completely eliminates all multiplications from the transform computation, was found to be limited by address calculations [11]. At this point it became apparent that the speed of the code developed would likely depend as much on the details of its implementation as on the algorithm used.

While this subdued the search for the ultimate Fourier transform algorithm, it did not eliminate the need to find an appropriate algorithm for incorporation into *Image*. Thus far it had become clear that simple, compact algorithms were best suited for implementation on the 68000, yet even this subset of algorithms is quite large.

The Final Choice: The Fast Hartley Transform

Images are inherently real valued, yet most general Fourier transform algorithms accept complex valued input and return complex valued output. The generality of these algorithms is also their weakness, for in the process of transforming real data they perform twice as many operations (arithmetic, address and transfer) as is necessary. Since the Fourier transform is commonly used in the analysis of real signals, special versions of almost every transform algorithm have been developed to deal more efficiently with real data [13,14]. Unfortunately, when it comes to inverse transformation, *another* special version of the algorithm is required to efficiently transform the complex output back into the real sequence [15].

The Hartley transform [16] distinguishes itself from its close cousin, the Fourier transform, by being real valued; it produces real output from real input. Even so, it provides the same phase and amplitude information about the data as the Fourier transform. The Hartley transform may also be computed using a 'fast' algorithm which requires $O(N\log_2 N)$ operations. Finally, the fast Hartley transform (FHT) is twice as fast as a complex valued FFT, requiring virtually the same number of operations as the real valued FFT algorithms [17,18].

In addition to its speed, the FHT provided two advantages over the FFT counterparts in the current application. First, since the Hartley transform is real valued, it could be efficiently stored in the limited memory available. While the symmetry of the complex output of real valued FFT

routines can be exploited to produce comparably dense storage, this requires some extra computation. Second, the Hartley transform is so symmetric that the forward and inverse transforms differ only by a multiplicative constant. This obviated the need to write two separate routines for transformation and inverse transformation.

While further research on available algorithms might have produced even better candidates for implementation, the Hartley transform's compactness, simplicity and symmetry made it a sound starting point.

2. MATHEMATICAL FOUNDATIONS

Having narrowed the scope of interest down to the Hartley transform, more mathematical details can now be given. To provide a standard against which to compare the Hartley transform, we introduce it in parallel with the Fourier transform.

2.1 The Fourier and Hartley Transforms

A physical process can be described either in the *time domain* by some function V(t) or in the *frequency domain* by some (generally complex) function F(f). Both descriptions contain the same information and can therefore be thought of as different representations of the same function [14]. To produce one representation of the function from the other, one uses the *Fourier transform* and the *inverse Fourier transform*[8]:

$$F(f) = \int_{-\infty}^{\infty} V(t)\, e^{-2\pi i f t}\, dt$$

$$V(t) = \int_{-\infty}^{\infty} F(f)\, e^{2\pi i f t}\, df \qquad (1)$$

The Hartley transform and its inverse are very similar to their Fourier counterparts [15]:

$$H(f) = \int_{-\infty}^{\infty} V(t)\, \mathrm{cas}(2\pi f t)\, dt$$

$$V(t) = \int_{-\infty}^{\infty} H(f)\, \mathrm{cas}(2\pi f t)\, df \qquad (2)$$

where the cas function is defined

[8]The Fourier transform of V(t) exists if V(t) it is bounded and absolutely integrable [4].

$$\mathrm{cas}(t) \equiv \cos(t) + \sin(t). \tag{3}$$

Comparing the Fourier and Hartley transform pairs, two distinctions are immediately evident. First, the Hartley transform of a real valued function is itself real valued. Second, there is absolutely no difference between the forward and inverse Hartley transform. These two characteristics, combined with the fact that the Fourier transform of a real function can be easily derived from its Hartley transform, make the Hartley transform an efficient and convenient vehicle for Fourier transform calculation, as will be shown below.

The relationship between the Fourier and Hartley transforms hinges upon symmetry conditions. Splitting the Hartley transform $H(f)$ into its even and odd components, $E(f)$ and $O(f)$, we obtain [15]

$$E(f) = \frac{H(f) + H(-f)}{2} = \int_{-\infty}^{\infty} V(t)\cos(2\pi f t)\,dt$$

$$O(f) = \frac{H(f) - H(-f)}{2} = \int_{-\infty}^{\infty} V(t)\sin(2\pi f t)\,dt \tag{4}$$

From these relations, we see that the Fourier transform can be obtained from the Hartley transform by forming the difference $E(f) - iO(f)$:

$$\begin{aligned} E(f) - iO(f) &= \int_{-\infty}^{\infty} V(t)(\cos 2\pi f t - i \sin 2\pi f t)\,dt \\ &= \int_{-\infty}^{\infty} V(t)\, e^{-i2\pi f t}\,dt \\ &= F(f) \end{aligned} \tag{5}$$

Conversely, the Hartley transform can be obtained from the Fourier transform by computing

$$H(f) = F_{\mathrm{real}}(f) - F_{\mathrm{imaginary}}(f). \tag{6}$$

The real and imaginary parts of a signal's Fourier transform are often of less interest than their derivates: The signal's amplitude, phase and power spectrum. Using the definitions of

these quantities and the equations (4) and (5), these values can be computed directly from both the Fourier and Hartley transforms as follows:

Function	Fourier Calculation	Hartley Calculation
Power Spectrum	$[F_r(f)]^2 + [F_i(f)]^2$	$\frac{[H(f)]^2 + [H(-f)]^2}{2}$
Amplitude	$\sqrt{[F_r(f)]^2 + [F_i(f)]^2}$	$\sqrt{[H(f)]^2 + [H(-f)]^2}$
Phase	$\arctan\left[\frac{F_i(f)}{F_r(f)}\right]$	$\arctan\left[\frac{H(-f)}{H(f)}\right] + \frac{\pi}{8}$

Table 1: Power Spectrum, Magnitude and Phase as calculated from the Fourier and Hartley transforms [15].

As Table 1 clearly shows, the calculations involved in recovering the power spectrum, amplitude and phase of a signal from its Fourier or Hartley transform are very similar and demand the same number of operations.

The similarity between the two transforms extends to the theorems commonly used in Fourier analysis. For virtually all Fourier transform theorems, there is a corresponding Hartley transform theorem. Both transforms are linear and therefore have Addition theorems (see Table 2). The Similarity theorem, which states that contraction of a signal's space domain representation corresponds to a dilation of its it frequency domain representation, also exists for the Hartley transform. This trend holds for the important Convolution and Correlation theorems as well, enabling these operations to be directly computed from the Hartley transform.

Theorem	V(t)	F(f)	H(f)
Addition	$V_1(t) + V_2(t)$	$F_1(f) + F_2(f)$	$H_1(f) + H_2(f)$
Similarity	V(t/T)	|T|F(Tf)	|T|H(Tf)
Shift	V(t - T)	$e^{-2\pi iTf/N}F(f)$	$\sin(2\pi Tf)H(-f) + \cos(2\pi Tf)H(f)$
Reversal	V(-t)	F(-f)	H(-f)
Convolution	$V_1(t) * V_2(t)$	$F_1(f)F_2(f)$	$\frac{1}{2}[H_1(f)H_2(f) - H_1(-f)H_2(-f) + H_1(f)H_2(-f) + H_1(-f)H_2(f)]$
Correlation	$V_1(t) \bigstar V_2(t)$	$F_1(f)[F_2(f)]^*$	$\frac{1}{2}[H_1(f)H_2(f) + H_1(-f)H_2(-f) + H_1(f)H_2(-f) - H_1(-f)H_2(f)]$

★ Denotes Correlation

Table 2: Theorems for the Fourier and Hartley transforms [15].

The strong similarity between the Hartley and Fourier transforms enables them to be interchanged in most situations. As we will see, making this exchange is worthwhile when computing the Fourier transform of real sequences.

2.2 The Discrete Fourier & Hartley Transform

While people tend to think in terms of continuous variables, it is usually necessary to use discrete variables when making measurements and performing computations. Computing the Fourier transform is no exception, so a discrete form of the Fourier transform and its inverse is needed:

$$F(k) = \sum_{n=0}^{N-1} V(n)\, e^{-2\pi ikn/N}$$

$$V(n) = \frac{1}{N} \sum_{n=0}^{N-1} F(k)\, e^{2\pi ikn/N} \qquad (7)$$

Here, our continuous function of time, V(t), has been sampled at N equispaced points producing a discrete sequence of N numbers, V(n). The Fourier transform, F(k), has likewise become a discrete sequence consisting of N complex numbers[9]. The discrete Fourier transform (DFT) is very

[9]In accordance with popular convention, we denote these indices as ranging from 0 to N-1, with the result that positive frequencies correspond to k in the range [1..N/2-1], while negative frequencies correspond to the 'second half of the array' or k in [N/2+1..N-1]. Note also that k = 0 is the 'DC' or zero frequency component of the

similar to is continuous counterpart, except for the factor of 1/N in the inverse transform. This factor is as often associated with the the forward transform as its inverse; while its location is not critical[10], it is necessary to restore the proper scale to a sequence that has been transformed and then inverse transformed [15].

Using the same discrete sequence, V(n), the discrete Hartley transform (DHT) and its inverse become:

$$H(k) = \sum_{n=0}^{N-1} V(n)\,\mathrm{cas}(2\pi kn/N)$$
$$V(n) = \frac{1}{N}\sum_{n=0}^{N-1} H(k)\,\mathrm{cas}(2\pi kn/N) \tag{8}$$

Except for a factor of 1/N, the DHT and its inverse are again identical. The relation between the DFT and DHT is also analogous to the continuous case; the DFT may be obtained from the DHT using

$$F(k) = E(k) - iO(k) \tag{9}$$

where

$$E(k) = \frac{H(k) + H(N-k)}{2}$$
$$O(k) = \frac{H(k) - H(N-k)}{2} \tag{10}$$

Conversely, the DHT can be computed from the DFT using

$$H(k) = F_r(k) - F_i(k) \tag{11}$$

as before. The direct analogy between the discrete and continuous cases also extends to the computations of amplitude, phase and power spectra, the only difference being that H(-*f*) is replaced with H(N-k)[11]. Likewise, the discrete theorems are very similar, except in two

---

spectrum, while k = N/2 is at once the maximum positive and negative frequency in the spectrum (these are equal due to the spectrum's periodicity).

[10]The transform pair can even be made symmetrical by multiplying both sums by $1/\sqrt{N}$.

[11]Which arises from letting k range from 0 to N-1, and not from -N/2 to N/2.

important cases which underscore the difference between the continuous and discrete transforms: The Convolution and Correlation theorems.

Discrete sampling of a signal can be modeled as multiplying the signal by a sequence of delta functions spaced some sampling interval, □T, apart. The transform of this delta function sequence is itself a delta function sequence, spaced 1/□T apart in the frequency domain. Since multiplication in the time domain is equivalent convolution in the frequency domain, the sampling of the signal results in the periodic reproduction of its transform with a 1/□T spacing in the frequency domain[12]. Likewise, the sampling or discretization of the transform results in the signal being periodically extended in the space domain. Consequently the signal, its DFT and its DHT are all periodic sequences with period N, as equations (7) and (8) will verify.

The discrete convolution and correlation theorems therefore describe cyclic convolution and cyclic correlation: In contrast to the continuous case, the shift and multiply operation characterizing convolution and correlation produces overlap between the sampled signal and its periodic extension. Consequently, the cyclic convolution and cyclic correlation are also periodic functions[13].

Theorem	V(n)	F(k)	H(k)
Addition	$V_1(n) + V_2(n)$	$F_1(k) + F_2(k)$	$H_1(k) + H_2(k)$
Shift	V(n - T)	$e^{-2\pi iTk/N}F(k)$	$\cos(2\pi Tk/N)H(k) - \sin(2\pi Tk/N)H(N-k)$
Reversal	V(-n)	F(N-k)	H(N-k)
Convolution	$V_1(n) \circledast V_2(n)$	$NF_1(k)F_2(k)$	$\frac{1}{2}N[H_1(k)H_2(k) - H_1(N-k)H_2(N-k) + H_1(k)H_2(N-k) + H_1(N-k)H_2(k)]$
Correlation	$V_1(n) \P V_2(n)$	$NF_1(k)[F_2(k)]^*$	$\frac{1}{2}N[H_1(k)H_2(k) + H_1(-k)H_2(N-k) + H_1(k)H_2(N-k) - H_1(N-k)H_2(k)]$

⊛ Denotes cyclic convolution.
¶ Denotes cyclic correlation.

Table 3: Theorems for the DFT and DHT[14] [15].

[12]To develop a graphical understanding of this, read E. O. Brigham's 'Graphical Development of the Discrete Fourier Transform' in his fine book, *The Fast Fourier Transform* [4].
[13]If one wishes to compute a normal, non-cyclic convolution or correlation, this overlap effect can be avoided by padding the data with zeros, as is described in Appendix A
[14]For the discrete Similarity theorem, see Bracewell [15].

Sampling also limits the accuracy with which a DFT models the continuous Fourier transform sought. Because of the transform's duplication every 1/□T cycles, it is necessary that the transform go to zero for frequencies exceeding 1/2□T (the Nyquist Frequency). Frequencies higher than this are *aliased* into the interval ±1/2□T. Furthermore, since sequences always have finite length, their transforms are convolved with with the transform of their windowing function, decreasing the spectral resolution in an effect known as *leakage* [4]. Once aware of these limitations of the DFT and DHT, however, one can minimize their effects.

The Two Dimensional DFT and Two Dimensional DHT

Until now we have considered only one dimensional signals and sequences in the time domain. Images are two dimensional *space domain* objects. The only difference between the two domains is that the temporal causality of the time domain is absent in the space domain; every pixel of an image can be accessed simultaneously, whereas a time series is inherently sequential.

While white light's rainbow colored spectrum is an easily visualized manifestation of Fourier transformation in one dimension, how does one visualize a two dimensional Fourier transform? As a diffraction pattern; if an object is considered to be a two dimensional aperture function, A(y,z), its far-field Fraunhofer diffraction pattern is the Fourier transform of A(y,z) [19]. If this description does little for your intuition, then using the FFT extensions to *Image* will; there is no better way to develop this intuition than by direct experience.

The two dimensional DFT and its inverse are expressed:

$$F(k_1,k_2) = \sum_{n_2=0}^{N_2-1} \sum_{n_1=0}^{N_1-1} V(n_1,n_2) \exp\left[-2\pi i\left(\frac{k_1 n_1}{N_1} + \frac{k_2 n_2}{N_2}\right)\right]$$

$$V(n_1,n_2) = \frac{1}{N_1 N_2} \sum_{n_2=0}^{N_2-1} \sum_{n_1=0}^{N_1-1} F(k_1,k_2) \exp\left[2\pi i\left(\frac{k_1 n_1}{N_1} + \frac{k_2 n_2}{N_2}\right)\right] \qquad (12)$$

Because of the exp function's separability ($e^A e^B = e^{(A+B)}$), the two dimensional (2D) DFT can be computed as the DFT of one dimensional DFTs:

$$F(k_1,k_2) = \sum_{n_2=0}^{N_2-1}\left[\sum_{n_1=0}^{N_1-1} V(n_1,n_2)\exp\left[-2\pi i\left(\frac{k_1 n_1}{N_1}\right)\right]\right]\exp\left[-2\pi i\left(\frac{k_2 n_2}{N_2}\right)\right]$$

$$V(n_1,n_2) = \frac{1}{N_2}\sum_{n_2=0}^{N_2-1}\left[\frac{1}{N_1}\sum_{n_1=0}^{N_1-1} F(k_1,k_2)\exp\left[2\pi i\left(\frac{k_1 n_1}{N_1}\right)\right]\right]\exp\left[2\pi i\left(\frac{k_2 n_2}{N_2}\right)\right] \tag{13}$$

This means that the 2D DFT of an image, for example, can be computed by first transforming each of the rows of the image and then transforming each of the columns of the image or vice versa.

The 2D DHT is expressed

$$F(k_1,k_2) = \sum_{n_2=0}^{N_2-1}\sum_{n_1=0}^{N_1-1} V(n_1,n_2)\,\mathrm{cas}\left[2\pi\left(\frac{k_1 n_1}{N_1}+\frac{k_2 n_2}{N_2}\right)\right]$$

$$V(n_1,n_2) = \frac{1}{N_1 N_2}\sum_{n_2=0}^{N_2-1}\sum_{n_1=0}^{N_1-1} F(k_1,k_2)\,\mathrm{cas}\left[2\pi\left(\frac{k_1 n_1}{N_1}+\frac{k_2 n_2}{N_2}\right)\right] \tag{14}$$

As in the one dimensional case, the forward and inverse transforms are identical but for a multiplicative factor. Unfortunately, however, the cas function is not separable and therefore the 2D DHT cannot be computed by simply applying a 1D DHT to the rows and columns of the image matrix. If a row-column DHT is computed, however, the two dimensional DHT can be recovered from the result, as is described under 'Row Column HT to Two Dimensional HT Conversion' below.

2.3 The Fast Fourier Transform

Because computing the DFT of an N point sequence requires N summations each involving N operations, the total computation requires $O(N^2)$ operations. Writing out the entire computation by hand will show, however, that many of these operations are redundant and can be eliminated. Using Danielson and Lanczos' [3] observation that an N point DFT can be expressed as the summation of two N/2 point DFTs, these redundancies can be eliminated as we now show. Adopting the conventional definition

$$W \equiv e^{-2\pi i/N}, \tag{15}$$

where W is commonly referred to as a *twiddle factor*[15], the DFT can be divided in two as follows:

$$\begin{aligned} F(k) &= \sum_{n=0}^{N-1} V(n)W^{nk} \\ &= \sum_{n=0}^{N/2-1} V(2n)W^{(2n)k} + \sum_{n=0}^{N/2-1} V(2n+1)W^{(2n+1)k} \\ &= \sum_{n=0}^{N/2-1} V(2n)W^{(2n)k} + W^k \sum_{n=0}^{N/2-1} V(2n+1)W^{(2n)k} \\ &= F_{even}(k) + W^k F_{odd}(k) \end{aligned} \tag{16}$$

$F_{even}(k)$ is the N/2 point DFT of the even elements of V(n), while $F_{odd}(k)$ is the N/2 point DFT of the odd elements of V(n). The second N/2 points of the transform are likewise computed using

$$\begin{aligned} F(k + N/2) &= F_{even}(k + N/2) + W^{k+N/2}F_{odd}(k + N/2) \\ &= F_{even}(k) - W^k F_{odd}(k) \end{aligned} \tag{17}$$

since F_{even} and F_{odd} both have period N/2 and $W^{N/2} = -1$.

While using this division reduces the total computation involved by almost a factor of 2, there is no reason to stop here. The beauty of the Danielson Lanczos Lemma is that it can be recursively applied to sub-sequences of length N/4, N/8, etc. [14]. When this technique is used on sequences an integer power of two in length, the division process can proceed $\log_2 N$ times, producing $\log_2 N$ *stages* each of which require O(N) operations. The resulting total operation count of $O(N\log_2 N)$ provides a vast improvement over $O(N^2)$ for large N, making this the *Fast Fourier Transform* or FFT.

The fundamental computational unit of the FFT is a two point transform, called a *butterfly* because of its appearance:

---

[15]The W^k are the N complex roots of unity and therefore lie on the unit circle in the complex plane.

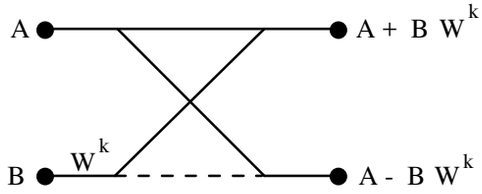


Figure 1: FFT Butterfly. The dashed line indicates negation.

Two input data points, A & B, produce two output points after one complex twiddle factor multiplication and two complex additions or a total of 4 real multiplications and 5 real additions. Since only two elements are accessed per butterfly, the same storage can be used for input and output, and the computation is performed *in place*. In the first stage of the FFT computation, N/2 such butterflies are applied to the input sequence producing N/2 two point transforms. The second and subsequent stages combine butterflies into groups of 2, 4, 8, etc. to perform longer transforms. This is depicted here in a butterfly flow diagram for a 16 point FFT:

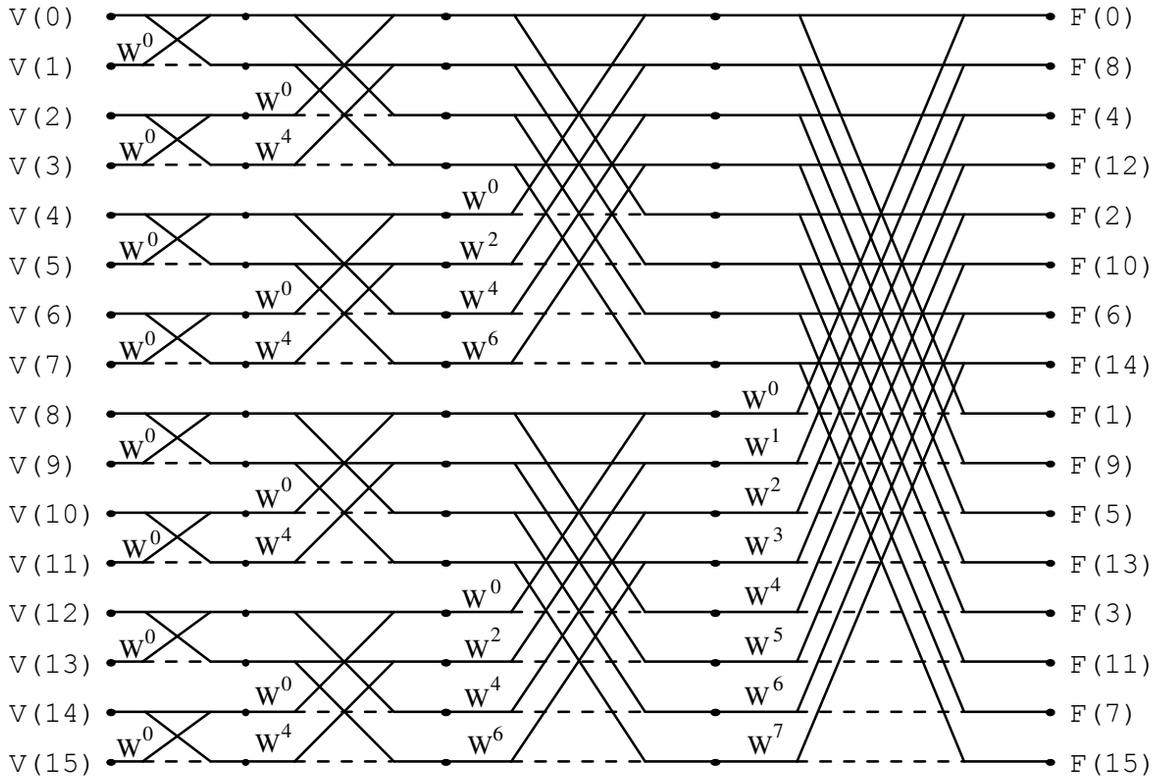


Figure 2: Butterfly flow diagram for a 16 point FFT.

The high degree of regularity in the butterfly diagram makes its implementation in code very compact. An FFT routine consists of three small loops, the outermost loop determines the stage

and repeats $\log_2 N$ times, while the inner two loops control the number of butterflies in a group and the number of groups[16], together performing N/2 two point butterflies.

By repeatedly decimating the input into even and odd sub-sequences, the radix 2 FFT returns a permutation of the desired transform. To restore sequential ordering, the elements whose indices bit-wise mirror one another are swapped (e.g. the element at index 0011 (3) is swapped with the element at index 1100 (12))[17]. This *bit reversal* operation, described under 'Details of the Utility Routines' below, may be performed before or after the transform.

The basic, radix-2 FFT algorithm is very symmetrical, but it accepts general complex input when all that we need here is the ability to transform real sequences. The Fourier transform of a real sequence has conjugate symmetry (the real part of the transform is even while the imaginary part is odd) which can be exploited to reduce the number of computations in an FFT by one half [14].

2.4 The Fast Hartley Transform

Like the DFT, the DHT summation can be split in two to reduce total operation count. Expressing equation (9) as

$$F(k) = \frac{H(k) + iH(N-k)}{1+i} \tag{18}$$

and substituting this expression into equation (16), we obtain

$$H(k) + iH(N-k) = [H_e(k) + iH_e(\frac{N}{2}-k)] + W^k[H_o(k) + iH_o(\frac{N}{2}-k)] \tag{19}$$

where $H_e(k)$ is the N/2 point DHT of the even indexed elements of H(k) and $H_o(k)$ is the N/2 point DHT of the odd indexed elements of H(k)[18]. Equating real and imaginary parts gives us the Hartley analog of equations (16) and (17) [21]:

---

[16]This terminology of *stages*, *group size* and *number of groups* is carried through to the source code level.

[17]For higher radix algorithms, the indices are digit reversed in the radix used; for radix 4 permutations, the array elements whose base-four indices mirror one another are swapped. The general radix permutation is therefore described as digit-reversal [20].

[18]Not to be confused with E(k) and O(k), the even and odd parts of H(k).

$$H(k) = H_e(k) + [H_o(k)\cos(2\pi k/N) + H_o((N/2) - k)\sin(2\pi k/N)]$$
$$H(k + N/2) = H_e(k) - [H_o(k)\cos(2\pi k/N) + H_o((N/2) - k)\sin(2\pi k/N)] \quad (20)$$

While k ranges from 0 to N-1, the indices for the even and odd sub-transforms are evaluated modulo N/2 [18]. Like the FFT, this decomposition can be recursively applied until length two transforms are obtained. Because the even-odd decimation is identical to that of the FFT, the bit reversal permutation is also the same.

Retrograde Indexing

In place computation is desired, yet equation (20) shows that both the kth element and the (N/2-k)th element of an N/2 point DHT must be accessed to compute one output point. Consequently, four elements must be processed at once to avoid overwriting an element that will be needed later. This is a manifestation of the FHT's *retrograde indexing* because the index, N/2-k, of the element multiplied by the sine term decreases with increasing k while the indices for the other elements increase with k. This is most easily visualized in the structured butterfly flow diagram for the FHT [22]:

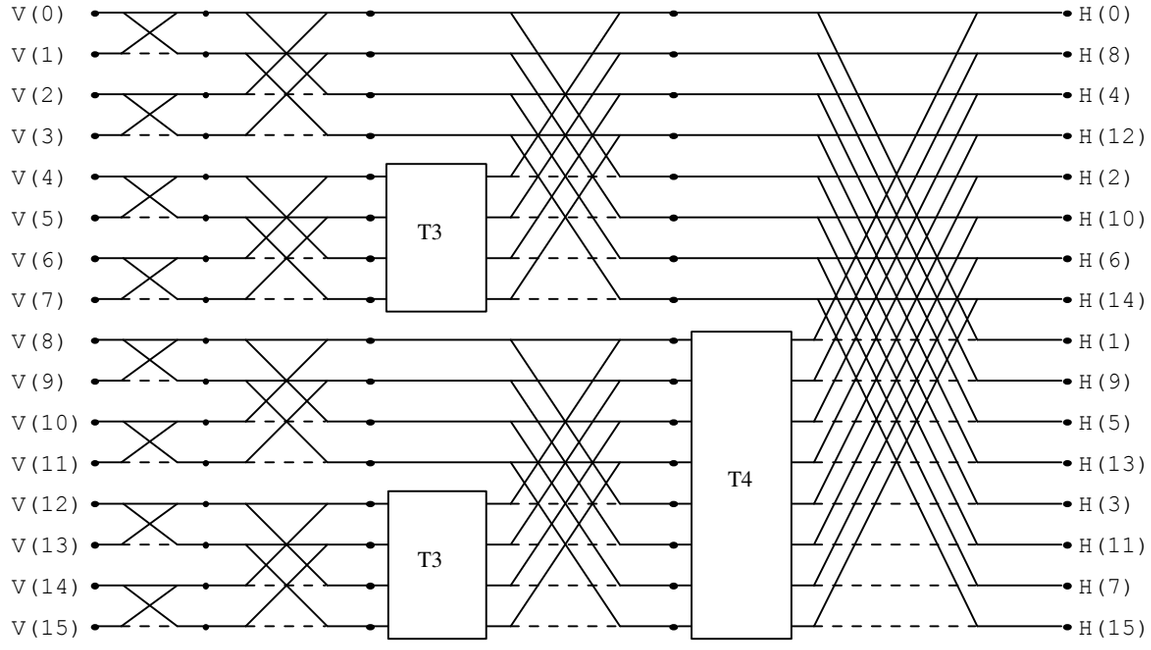


Figure 3: Structured Butterfly Flow Diagram for a 16 point FHT.

In Figure 3, butterflies operate in the same way as they do for the FFT. The absence of twiddle factors in the first stage is no mistake; the sine and cosine terms are either 1, 0, or -1 in these stages and are never both simultaneously non-zero.

The retrograde indexing becomes apparent in the third and subsequent stages, where the twiddle factor multiplication takes place in the boxes labeled T3 and T4. The data paths and twiddle factor multiplications inside these boxes are shown here:

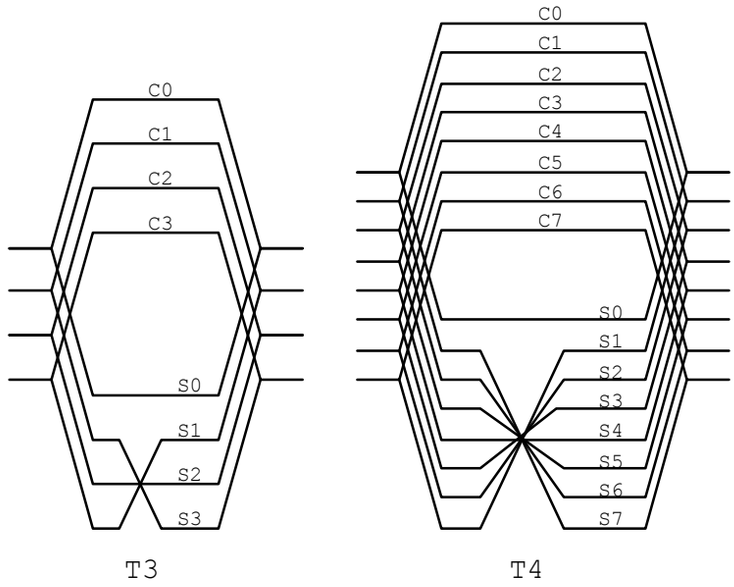
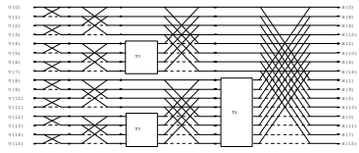


Figure 4: FHT Retrograde Indexed Twiddle Factor Multiplication.

In Figure 4, units T3 and T4 show how the twiddle factor multiplication takes place. The Ck and Sk denote the terms cos(2□k/N) and sin(2□k/N) respectively. The retrograde indexing of the sine terms can easily be seen to produce the requirement of two input points for each output point. To attain in place computation, two butterflies must therefore performed at once. By exploiting the symmetries



(21)

the total number of multiplications for the 4 point dual-butterfly is reduced to a total of four real multiplications and six real additions [18]. Since two complex FFT butterflies require 8 real multiplications and 10 real additions, the operation count is reduced by almost one half.

Operation count of the FHT can be reduced yet further. Since the first two stages involve no multiplications, they can be computed separately. To minimize the number of memory accesses in the first two stages, they can be collapsed into one stage made up of N/4 radix 4 butterflies. The first butterfly of each group starting with the third stage likewise involves no multiplications and can be treated separately.

Because of the similarity between the four twiddle factor multiplications necessary to compute a dual-butterfly and a general complex multiplication, a trick to reduce the operation count in the latter computation can also be exploited. The product

$$(a + ib)(c + id) = (ac - bd) + i(ad + bc) = Re + iIm \tag{22}$$

which requires 4 multiplies and 3 adds can be computed with 3 multiplies and 5 adds by using a temporary variable T[19]:

$$\begin{aligned} T &= (a - b)d \\ Re &= T + a(c - d) \\ Im &= T + b(c + d) \end{aligned} \tag{23}$$

Further reductions in operation count can be achieved by exploiting the symmetries of the twiddle factors[20], but since they occur only under specific conditions, continuously checking for these conditions may outweigh the benefits of their 'optimization.'

[19]Because an ADD instruction runs between 7 and 18 times faster than a MUL instruction on the 680x0 processors, this is a worthwhile optimization.

[20]For example, when the sine and cosine terms are equal, a multiplication can be avoided.

3. DETAILS OF THE FHT ALGORITHM

The current implementation of the FHT algorithm evolved over more than six months. The earliest versions had the same basic structure as the final ones, but many changes were made that significantly enhanced both the speed and accuracy of the final implementation.

3.1 Making the FHT Fast

In making the FHT implementation execute as fast as possible, several basic tenets of 68000 assembly language programming were observed. In addition, there were aspects of the Macintosh system software that had to be considered to speed the code on this system. Finally, common sense was used in evaluating implementation tradeoffs. These are all discussed below.

Use The Registers To Minimize Memory Access

The designers of the 68000 were very generous is giving the programmer sixteen 32 bit registers to work with. Eight of these are general purpose data registers, while the other 8 are address registers. Since register access is about 5 times faster than memory access, the registers are to be exploited to their fullest, while memory is be accessed only when absolutely necessary. This includes minimizing the use of the memory-based stack and local variables, which are the second most convenient place for temporary data storage. To squeeze the most functionality out of the 8 data registers, they can be treated as sixteen 16 bit registers; the SWAP instruction swaps the high and low words of a given data register, allowing one to pack two numbers in each register. While they do not support all data operations, address registers can also be used for temporary data storage as well.

To further minimize memory accesses, data should always be fetched from memory in the largest possible denomination. For example, two word accesses require a minimum of 32 cycles on the 68000, while one long word access requires only 20 cycles. Similarly, it is more efficient to fetch a word (and process the bytes in the registers) than to fetch 2 bytes[21]. This simple optimization has been exploited whenever possible.

[21]Only the 68020 and 68030 allow byte access at an odd address.

Make the Code Small

The processor spends a non-negligible part of its time simply fetching the instructions to be executed from memory. All else being equal, a 10 byte implementation of function X will be faster than a 12 byte implementation. This may sound simple enough, but there are at least two other important considerations to be made here.

First, the 68020 and 68030 have 256-byte instruction caches. If an entire routine, or at least its innermost loop, can be made small enough to fit in this cache, it will execute much faster since the processor will not need to go off chip to fetch instructions. The 68040's two 4K caches (data and instructions) would greatly enhance the performance of this FHT routine since all 938 bytes of the code as well as most of the twiddle factor lookup table (described below) could fit into these caches.

The are, however, limits to the 'small is beautiful' coding philosophy. Like any other author, a programmer must also take into consideration his readership. If the code one is writing is to be burned into a ROM and never seen again, every trick available for making that code small and fast should be employed. Code like this that has undergone severe 'human compilation' can be very difficult to read, however. Because it is to be distributed in the public domain, an effort has been made to maintain the code's readability without making too many sacrifices in compactness.

Avoid Slow Instructions

In every microprocessor's users manual, a section is devoted to instruction execution timing. Perusal of the 68000's execution timing section quickly shows that while most instructions require less than 20 cycles to execute, the Multiply and Divide instructions take about 70 and 150 cycles, respectively[22]. These execution times are long enough that the use of these instructions inside loops should be avoided or at least minimized. For example, multiplication or division by powers of two should always be done using left and right arithmetic shift operations. If a number is to be doubled, however, adding it to itself is quicker than shifting it left by one bit.

[22]These have been reduced to 28 and 56 cycles, respectively, on the 68030.

Avoid the Trap Dispatcher

The Macintosh provides a rich set of over 1100 system routines for performing everything from drawing to file I/O. Most of these routines are stored in ROM, yet because their specific ROM locations may differ from one Macintosh model to the next, applications do not access ROM routines directly (by address) but instead through the *Trap Dispatcher* [23]. Before we describe the Trap Dispatcher, however, it is necessary to know a little more about how the 68000 works.

The shortest 68000 instruction is one 16 bit word, the most significant 4 bits of which is called the *opcode*. The opcode, represented in hexadecimal from 0 to F, separates the 68000 instruction set into functional groups. Only 14 of the opcodes are used on the 68000; there are no instructions beginning with the opcodes A or F. Since the development of the 68000, Motorola has made use of the F opcodes to describe instructions for their floating point coprocessors, the 68881 and 68882. Instructions beginning with A, however, will still generate a 'Line 1010' exception (or A-Trap) when encountered by the 68000.

Apple has taken advantage of this behavior by making all of their ROM calls be single word instructions beginning with A. When such an instruction is encountered, 68000 state is saved and execution begins at the address stored in A-Trap exception vector location $28. By storing the address of the Trap Dispatcher in this location at startup, all subsequent A-Traps are processed by this routine. The Trap Dispatcher inspects the rest of the word that generated the A-Trap, looks at the 'Trap Dispatch Table' to determine where the routine corresponding to that word is stored on the current machine, and finally passes execution to it.

The advantage of this technique is that the ROM configuration can change from machine to machine, yet the Trap Dispatcher will always find the correct routine. The disadvantage is that the Trap Dispatcher is slow. First, the Line 1010 exception requires 34 cycles to complete, then the Trap Dispatcher routine, involving at least 14 more instructions, must execute. While this penalty is tolerable for individual ROM calls, it becomes quite a burden when included in a tight

loop. For this reason, ROM calls were never made inside loops and were eliminated from virtually all routines in the assembly language routine library[23].

Use Lookup Tables

Whenever successive calls to a routine result in the duplication of calculations, gains in execution time may be made by storing their outcome in a lookup table. Then instead of repeating the calculation on every call, the lookup table can be accessed instead. As long as accessing the lookup table is faster than performing the calculation itself, precious time can be saved. In the case of the FHT, bit reversal, twiddle factor and address calculations are all duplicated each time the routine is called, yet for a given sequence length N, these calculations always produce the same results. Should these calculations be replaced with lookup tables? In two cases, the answer is yes.

One negative aspect of lookup table usage is that the code loses its independence; it becomes the programmer's responsibility to bring the code and the table together in cooperative union. Another drawback is the implicit limitation on sequence length imposed by a lookup table. Finally, lookup tables use up valuable memory. In applications where memory is limited, the speed advantages a lookup table might bring may be outweighed by its size. While the latter disadvantage wanes with RAM prices, the former two remain important; self contained code is always nicer to work with. Fortunately, in the Resource Manager (described in more detail below), the Macintosh provides a mechanism which facilitates the storage and dynamic allocation of memory for lookup tables, greatly facilitating their use.

The decision to put the twiddle factors in a lookup table was not hard to make: Each twiddle factor table entry requires a sine or cosine call and transcendental functions are notoriously slow. Furthermore, the size of the table needed is not prohibitive. For a sequence of length N, N/4 sine terms and N/4 cosine terms are needed, giving a table N/2 terms in length. While the similarity of the sine and cosine terms could be exploited to reduce this the table size to N/4, this was not done here. The maximum N required by the *Image* program is 2048, giving a twiddle factor

[23]An alternate solution is to use the routine *GetTrapAddress* once to determine the physical address of the ROM routine of interest. Jumping directly to this address then eliminates the delay incurred in repeatedly using the Trap Dispatch mechanism.

lookup table 4K in size. This is small when compared to the images being operated on, often 300K in size.

A lookup table for bit reversal was also used to speed up that operation as is described in more depth under 'Evolution of the Utility Routines' below. For the bit reversal operation, a separate lookup table is required for each sequence length in [64, 128, 256, 512, 1024, 2048], the largest requiring 3972 bytes of storage space. Again, this is small compared with the size of the images in memory, and since the memory for the lookup table needed is dynamically allocated, it can be recovered if necessary after the bit reversal computation is complete.

As discussed above, the retrograde indexing characteristic of the FHT leads to a sequence of addresses (array indexes) that is not as regular as in the fast Fourier transform. Because of the added complexity of the retrograde index calculation, the addresses were originally put in a lookup table. The rationalization used was that the memory fetch and pointer increment required for the lookup table could always be executed more quickly than the retrograde index calculation itself[24]. The drawback was that the address lookup table required 20K of storage space for a maximum sequence length of 1024[25]. Increasing the sequence length to 2048 would have required doubling the lookup table size to 40K. This rather severe restriction on sequence length was overcome by studying the address sequence generated by the FHT and developing an efficient means for its calculation[26].

Eliminating the address lookup table removed one restriction on sequence length in this FHT routine. The bit reversal and twiddle factor lookup tables, however, still impose restrictions on sequence length. Even so, these lookup tables could be made large enough to accommodate arbitrarily long sequences. Therefore, it is not the lookup tables that set a practical limit on sequence length, but the way in which data is addressed. Study of the FHT source code in Appendix F shows that offsets into the data[27] are stored as 16 bit words. This limits the

---

[24]A memory fetch and register increment (ADDA) take about 25 cycles, while the quickest instructions require about 4 cycles. If the address can be calculated using 4 fast instructions, then no lookup table is needed.

[25]The table included 5 words (10 bytes) for each dual-radix butterfly; 4 words for the addresses of the dual butterfly elements and 1 for the address of the twiddle factor. For the 8 stages requiring the table for N = 1024 (the first two stages do not), there are 256 dual-radix butterflies. Total table size was therefore 8 x 256 x 10 = 20,480 bytes.

[26]The addresses, names Ad1-Ad4 and CSAd, are easily generated with program DFHT3.p in Appendix E

[27]Documented as Ad1 - Ad4 in Appendix E and Appendix F

maximum array index to 2^{15} - 1, or 32K. Data are stored as 16 bit words, however, so the maximum index is actually limited to 16K. Since few people ever need to take transforms longer than 16K, this restriction seems justified.

3.2 Making the FHT Accurate

By using integer arithmetic, the speed of computation is enhanced at the expense of its accuracy and ease of coding. Many techniques for dealing with the limitations of integer arithmetic in digital signal processing applications have been developed over the years, several of which have been exploited here to minimize error.

Scaling

In each of the FHT butterflies, three input terms are summed to generate two output terms[28]:

$$\begin{aligned} H(k) &= H_e(k) + [H_o(k)\cos(2\pi k/N) + H_o((N/2) - k)\sin(2\pi k/N)] \\ H(k + N/2) &= H_e(k) - [H_o(k)\cos(2\pi k/N) + H_o((N/2) - k)\sin(2\pi k/N)] \end{aligned} \qquad (20)$$

If the input terms are all normalized to 1 and $2\pi k/N$ is set to $\pi/4$ to maximize the contribution of the trigonometric terms, the largest possible result is $1 + \sqrt{2}$. The maximum growth per stage is therefore 2 bits, although in practice the maximum average growth per stage is one bit [24].

This overflow characteristic effectively reduces the dynamic range of the integers used by 2 bits. Since images are stored as matrices of 8 bit pixels in the *Image* program, only 6 bits of dynamic range would be available for an in-place transform calculation. This is far too little dynamic range in which to perform a complex, iterative integer calculation, so the transform calculation is instead performed on a copy of the image which has been vertically scaled to fill a 14 bit dynamic range in a 16 bit deep image buffer. This not only provides the needed dynamic range for an accurate transform calculation, but also insures that an image can be transformed and inverse transformed several times before the inevitable degradations due to integer computations become apparent in its 8 bit representation.

These degradations stem in part from the persistent, stage by stage growth of the data and the limited dynamic range of integers. Several techniques have been developed to deal with this

[28]In the first two stages, the transcendental terms are either 0 or 1 and overflow is limited to one bit.

stage-wise data growth [24,25]. One simple solution, called *unconditional scaling*, right-shifts all the data in an input sequence of length N by $\log_2 N$ bits before the transform is computed. This straightforward technique ensures that no overflow will occur during the $\log_2 N$ stage FHT computation and eliminates the need for the routine itself to monitor data growth, yet has obvious drawbacks. First, if the input data are all less than $\log_2 N$ bits in magnitude, the unconditional shift will zero out the entire sequence. This situation could be avoided by finding the extrema of the data and right shifting only the minimum number of bits necessary to prevent overflow. Still, by assuming the worst case overflow will always occur, unconditional shifting 'throws out' information before the transform computation even starts.

In practice, overflow does not occur on every stage and will often average less than 1/2 bit per stage. The *block floating point* scaling technique exploits this behavior by monitoring the data growth stage by stage. If, after a given stage, the data has overflowed (exceeded its maximum allowable range), then the whole data sequence is right-shifted by the number of bits of overflow and a scaling counter is incremented to reflect the total accumulated scaling. Because it is necessary to monitor numbers exceeding the 'maximum allowable range' without actually experiencing arithmetic errors, this range is 2 bits smaller than the maximum range of the integers used. Since we use 16 bit integers, our dynamic range is reduced to 14 bits, or numbers in the range -8192 to 8191. In this way, a data sequence that generates 2 bits of overflow before the first scaling operation takes place will not produce any fatal arithmetic errors.

By monitoring data growth during the transform computation, the block floating point technique minimizes the amount of information 'thrown out' and was therefore adopted for use in our FHT routine. In implementing this technique, however, it was considered undesirable to re-scan the data after every stage which generated overflow to restore the proper scaling[29]. The additional memory accesses involved in this process were avoided by using three local variables and a trick.

The trick makes use of the fact that we are interested only in the bit-magnitude of the data generated in each stage and not in the decimal value; only if a number exceeds 14 bits do we

[29]Adam Erickson, private communication.

need to perform data scaling. The bit-magnitude of the sequence is monitored most quickly by bitwise OR-ing the absolute value of each element. If any of the values generated in a given stage fall outside the range ±8191 but within the range ±16383 (are 15 bits in magnitude), one bit of overflow occurred; if any values attain 16 bits in magnitude, 2 bits of overflow occurred. Once one bit of overflow has occurred on a given stage, further one bit overflows needn't be monitored. Likewise, when the maximum overflow of two bits has occurred on a given stage, no overflow checking is necessary for the remainder of the butterflies in that stage. In this way, the amount of time spent checking for overflows is minimized.

The number of bits of overflow encountered in a given stage's computation is stored in a local variable, NewShift (Appendix F). At the end of that stage, the value of NewShift is added to the variable ShiftTotal, recording the total number of bits shifted during the transform. NewShift is then copied into another local variable, OldShift, before being zeroed for the coming stage. During the next stage each incoming data element is right shifted by OldShift bits. This method avoids the unnecessary memory accesses that an explicit scaling subroutine would involve in every stage but the last. Should the last stage produce overflow, an explicit scaling loop is still necessary.

When the transform computation is complete, the variable ShiftTotal is returned to the caller. If the programmer wants to restore the proper scaling to the data, the sequence can be multiplied by $2^{\text{ShiftTotal}}$ (shifted left by ShiftTotal bits), but this will often cause the data to exceed 16 bits in magnitude. The most compact form of storing the transformed sequence is to maintain its exponent, ShiftTotal, in a separate variable.

Part the Hartley transform's beauty is that the forward and inverse transforms differ only by a factor of 1/N. Our routine therefore makes no distinction between the forward and inverse transforms and leaves it up to those programmers interested in maintaining proper scaling to subtract $\log_2 N$ from the exponent after an 'inverse' transform has been calculated.

Rounding

The accuracy of the FHT routine can be further enhanced by using rounding techniques to minimize the inevitable error introduced when data are shifted out during scaling. There are

many alternatives to simple truncation, which ignores the bits being shifted out. Five of these are listed here [26]:

1. *Up/Down Rounding*. The mid way points are are shifted up/down to the next allowable value.
2. *Magnitude Up/Down Rounding*. The mid-way points are moved away from/toward zero .
3. *Value-Alternate Rounding*. Adjacent mid-way points are alternately moved up and down.
4. *Random Rounding*. The direction of rounding is pseudo-random.
5. *Stage-Alternate Rounding*. The direction of rounding alternates on each stage of the FFT.

Like scaling, however, the accuracy gained by rounding is paid for in slower execution times. Obviously, some techniques, like random rounding, may be more difficult to implement efficiently than others. We chose to implement up/down rounding because it could be performed by inspecting the last bit shifted out. This is conveniently stored in the carry bit of the 68000's condition code register. If the carry bit is 1 after a right shift operation, then the m bit number shifted out was larger than $2^m/2$ and a round up operation is performed by adding 1 to the result (this is easily shown to work for negative numbers as well). If the carry bit is zero, the result is rounded down by being left as is.

This very simple form of rounding only takes two instructions after the shift operation:

```
            ASR.W  D7, D4              ; arithmetic shift right
            BCC.S  RoundDn1            ; branch on carry clear to RoundDn1
            ADDQ.W #1, D4              ; if carry set then round up
RoundDn1    …                          ; continue here
```

When shifting by more than one bit, this is a cheap and worthwhile form of up/down rounding. When there is only one bit to shift, however, this implementation reduces to a simple round up operation and is no better than truncation's round down. Since two bits of overflow occur only rarely in a given stage[30], no rounding was used during the overflow shift operations. Instead, this rounding scheme was employed when the 32 bit result of the twiddle factor multiplications was scaled back down to 16 bits for storage and when the row transforms were normalized to a uniform scale. The latter case is described in more detail below under 'Row Shifting', while the former is discussed immediately below.

[30]So rarely, in fact, that at one point a version of the routine was developed to accommodate only 1 bit of overflow. This increased the dynamic range of the input and output to 15 bits and worked admirably in practice. Unfortunately, certain sequences (like 16383, 0, 0, 0, 16383, 0, 0, 0), could make this routine fail by producing two bits of overflow on the third stage.

Double Precision Accumulation

The twiddle factors, cos(2□k/N) and sin(2□k/N), are approximated by 16 bit integers and stored in a lookup table (see Appendix D, Resource Formats). To make the twiddle factors fill the dynamic range of the integers used to represent them, they are multiplied by 2^{15}. Therefore, twiddle factor multiplication in the FHT routine implicitly scales the data up by 2^{15}. Since the 68000's multiplication instruction takes two 16 bit words as arguments and returns a 32 bit long word product, the 32 bit twiddle factor product must be scaled back down to 16 bits for in place storage. This is done by using the arithmetic right shift instruction and the up/down rounding scheme just discussed.

This is not done immediately after the multiplication, however, but is postponed until just before the data are stored. By performing the additions and subtractions that follow the twiddle factor multiplication in double precision, the accuracy of the overall calculation is improved.

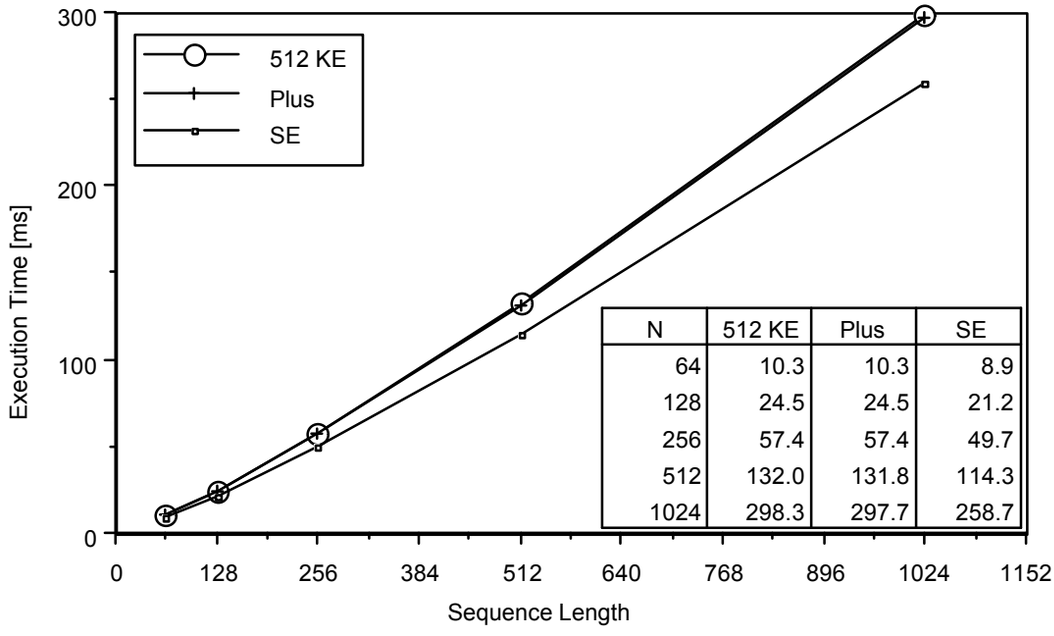
3.3 The Fully Evolved FHT Algorithm

The final 68000 assembly language implementation of the FHT algorithm exploits all the points discussed above to maximize its speed and minimize its error. For the inquisitive reader, a listing of an FHT routine in Pascal is included in Appendix E, while the assembly listing is in Appendix F.

Timing Results

Because the FHT routine only uses instructions in the 68000 instruction set[31], it can run on all Macintosh computers. Figure 5 shows the execution times in milliseconds as a function of sequence length for three 68000 based Macintosh computers: the Macintosh 512KE, Plus and SE.

[31]Some of the utility routines use the FPU, restricting their use to computers with the 68881 or 68882 FPU.

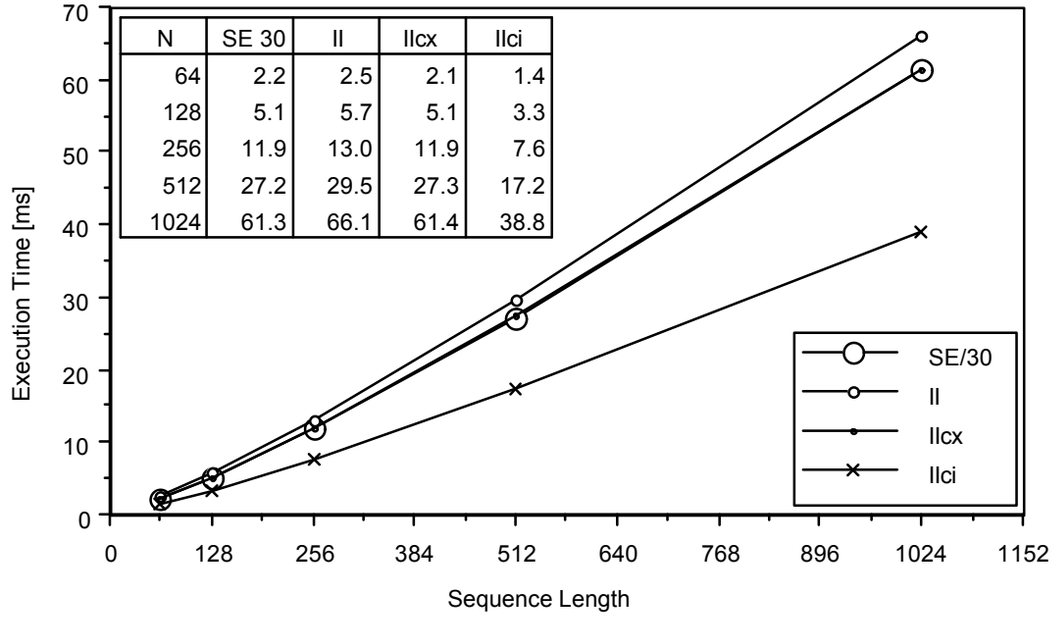


N	512 KE	Plus	SE
64	10.3	10.3	8.9
128	24.5	24.5	21.2
256	57.4	57.4	49.7
512	132.0	131.8	114.3
1024	298.3	297.7	258.7

Figure 5: FHT Execution times for 68000-based Macintosh computers.

The Macintosh Plus and 512 KE are seen to have very similar execution times, while the SE is about 13% faster, requiring only 259 milliseconds for a 1024 point FHT.

Execution times for the Macintosh SE30, II, IIcx and IIci computers are shown in Figure 6.



N	SE 30	II	IIcx	IIci
64	2.2	2.5	2.1	1.4
128	5.1	5.7	5.1	3.3
256	11.9	13.0	11.9	7.6
512	27.2	29.5	27.3	17.2
1024	61.3	66.1	61.4	38.8

Figure 6: FHT Execution times for 68020- and 68030-based Macintosh computers.

Of all of these machines, only the Macintosh II is 68020 based; even though it has the same 16 MHz clock frequency as the Macintosh SE30 and IIcx computers, the latter two both use the 68030 processor and run about 7% faster. The Macintosh IIci's 25 MHz 68030 completes a

1024 point FHT in only 39 milliseconds, making it 37% faster yet, or almost 8 times faster than the original Macintosh.

While these execution times are interesting, they alone do not determine the execution speed of the final two dimensional FFT. Before such a computation can be undertaken, there are many utility routines that need to be implemented, any one of which could slow down the whole process considerably. The implementation of these routines is discussed in detail below.

4. DETAILS OF THE UTILITY ROUTINES

Bit Reversal

A natural consequence of the stage-wise 'decimation' occurring in every in-place FFT or FHT is the bit-reversed permutation of the transformed sequence [14,20]. 'Un-shuffling' the sequence is therefore a task that must be performed either before or after every transform. While this task doesn't require much computation in comparison to the transform itself, it can approach 10% of the total execution time required for a particular transform [20] and therefore bears inspection and optimization.

First note that bit reversal is simply an in-place data swapping operation. Consider the elements of an array of length N, where N is an integer power of two. The bit-reversal operation scans this array and swaps the element at index I with the element at index I', where I' is the mirror image of I when both numbers are represented base 2. Since some numbers are symmetrical under the binary mirror operation of bit reversal, the array elements with these indices don't need to be swapped at all. As illustrated in Table 4, there are 4 such elements in an array of length 16. In sequences of 32 and 64 elements in length, there are 8 such elements. In general, there are

$$2^{\left\lceil \frac{\log_2 N}{2} \right\rceil}$$

bit-symmetric indices for an array whose length, N, is an integer power of 2. Although the proportion of symmetric indices decreases with increasing N, avoiding unnecessary swaps saves precious time.

Index Base 10	Index Base 2	Bit-Reversed Index Base 2	Bit-Reversed Index Base 10
0	0000	0000	*0
1	0001	1000	8
2	0010	0100	4
3	0011	1100	12
4	0100	0010	2
5	0101	1010	10
6	0110	0110	*6
7	0111	1110	14
8	1000	0001	1
9	1001	1001	*9
10	1010	0101	5
11	1011	1101	13
12	1100	0011	3
13	1101	1011	11
14	1110	0111	7
15	1111	1111	*15

* Numbers symmetric under bit-reversal operation.

Table 4: Bit Reversal Made Clear.

Typically, the bit reversed index calculation is done by a straightforward sequence of bit shifting operations. Since the bit-reversed indices are unique for a given N, repeating this bit reversal calculation for every sequence of length N is wasteful. As mentioned earlier, the solution is to use a lookup table to store the bit-reversed indices. This allows the bit-reversal operation to proceed without any bit-shifting operations or conditional tests to find bit-symmetric indices. In the final implementation, the lookup tables for N in [64, 128, 256, 512, 1024, 2048] were stored in a resource[32] as arrays of integers. The table for N=1024, at only 2K in size, is small when compared to the 300K required by many images, but is the same size as a linear 16 bit sequence of length 1024. Either way, the memory for the table is allocated dynamically and can be released when no longer needed. Using an assembly language implementation of this lookup table technique proved to be over two orders of magnitude faster than than its direct-computation counterpart coded in THINK Pascal (see Appendix B for complete timing results).

To further speed the bit-reversal operation, the routine BitRevRows was assembly coded to perform the bit reversal on each row of the image matrix. By bit reversing all the rows together, only two calls to this routine are required during the transformation of an NxN image matrix, instead of 2N calls.

[32]The details of the resource formats used can be found in Appendix D, while more general information on Macintosh resources is to be found in the section on the Macintosh Toolbox below.

Row Shifting

As described above, the FHT routine returns the number of bits, m, of overflow that occurred during the transform calculation. In order to obtain a result that is properly scaled, the transform sequence must be multiplied by 2^m. When transforming an NxN image matrix row by row, the amount of overflow will vary from row to row. In order to insure that all rows have the same scaling before the computation proceeds, it is necessary to perform a row-wise scaling adjustment.

During the sequence of row transformations, the scaling, m_i, of each row is recorded in an array. The maximum row scaling, M_r, of all rows is also recorded in a separate variable. When the row transformations are complete, each row i is then right shifted by M_r-m_i bits to give it the same scaling as the row which incurred M_r bits of overflow. This is performed by the routine ShiftRowsR, which, like BitRevRows, performs the shift operation for each row of the image matrix and therefore need only be called twice during the two dimensional transform routine. The total scaling undergone by the image matrix during the 2D transform is then $M_r + M_c$, where M_c is the maximum overflow incurred during the column transforms. This scaling, or exponent, is returned to the calling program.

The R at the end of ShiftRowsR stands for Rounding. In this routine, the same Up/Down rounding technique described above in 'Making the FHT Accurate' was used. Since rows are often shifted by more than one bit, this was an appropriate use of this rounding technique. While no quantitative study was conducted here to determine the effect this rounding scheme had on signal to noise ratio, the qualitative improvement was immediately apparent.

Matrix Transposition

As described above, the two dimensional Hartley transform can be recovered from a matrix which has been row- and column-transformed using a one dimensional Hartley transform routine. Since the image is stored as a sequence of rows in memory, transforming each row is straightforward. Elements of a particular column, however, are interleaved in memory at intervals equal to the row length, complicating the addressing required to access them. Therefore, instead of creating special code to access column elements directly, the whole matrix is transposed in place. The

column transforms are then performed using the same Hartley transform routine as was used for the rows and, once complete, the matrix is transposed back to its original orientation.

Block Swapping

The indexing convention used places the spectral components corresponding to low spatial frequencies in the corners of the matrix as shown in Figure 7a. For example, the zero frequency, or 'DC', component of the matrix is in the upper left corner of the power spectrum matrix.

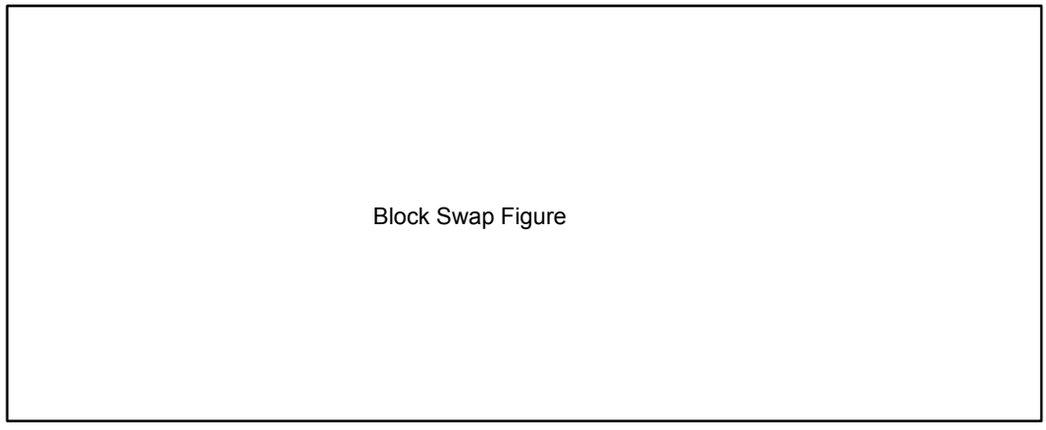


Figure 7a & b: Block Swap Example.

Power spectra are normally displayed with the low spatial frequencies at the center of the matrix as in Figure 7b, corresponding to the form of natural diffraction patterns. To create this preferred viewing arrangement, quadrants 1 and 3 of the power spectrum matrix are swapped, as are quadrants 2 and 4. This 'block-swapping' operation is carried out every time the power spectrum is displayed. Like matrix transposition, block-swapping's simple memory movement does not take much time.

Row-Column HT to Two Dimensional HT Conversion

In contrast to the two dimensional Fourier transform's separable exponential kernel, $\exp[\pm 2\pi i(n_1k_1/N_1 + n_2k_2/N_2)]$, the two dimensional Hartley transform's kernel, $\mathrm{cas}[2\pi(n_1k_1/N_1 + n_2k_2/N_2]$, is not separable. Therefore, applying the one dimensional Hartley transform to each row and column of the image matrix does not give us a two dimensional Hartley transform, but the two dimensional Hartley transform can be derived from the function that results from this row-column operation,

$$T(k_1,k_2) = \sum_{n_2=0}^{N_2-1} \sum_{n_1=0}^{N_1-1} V(n_1,n_2)\, cas(2\pi k_1 n_1/N_1) cas(2\pi k_2 n_2/N_2), \tag{24}$$

by a simple computation [27]. Using the trigonometric identity

$$2cas(\alpha+\beta) = cas(\alpha)cas(\beta) + cas(\alpha)cas(-\beta) + cas(-\alpha)cas(\beta) - cas(-\alpha)cas(-\beta), \tag{25}$$

it is easily seen that the Hartley transform $H(k_1,k_2)$ can be computed from T(u,v) using

$$\begin{aligned} 2H(k_1, k_2) &= T(k_1, k_2) + T(N_1-k_1, k_2) + T(k_1, N_2-k_2) - T(N_1-k_1, N_2-k_2) \\ &= A + B + C - D. \end{aligned} \tag{26}$$

This computation is carried out most efficiently if one first computes the value

$$E = \frac{1}{2}[(A + D) - (B + C)]. \tag{27}$$

Then T(u,v) can be converted into $H(k_1,k_2)$ at the four matrix locations A-D using the in-place calculation

$$\begin{aligned} A &= A - E \\ B &= B + E \\ C &= C + E \\ D &= D - E. \end{aligned} \tag{28}$$

For an N_1xN_2 matrix, this calculation need be only carried out $N_1N_2/4$ times, each calculation involving 7 additions and one shift operation. While the extra post processing required to create a two dimensional Hartley transform from the row-column transform increases the operation count by 8% for an 8x8 matrix, this proportion diminishes with increasing array size [27].

Computing the Power Spectrum

While the magnitude and phase of the Fourier transform can be recovered from the Hartley transform of an image, they comprise two separate images that can be difficult to interpret. Although it doesn't contain phase information, the power spectrum, or squared modulus of the Fourier transform, is more commonly displayed since it corresponds directly to optical diffraction patterns. In one dimension, the power spectrum, P(*f*), is computed from the Hartley transform as follows

$$
\begin{aligned}
P(f) &= F_r^2 + F_i^2 \\
&= E(f)^2 + O(f)^2 \\
&= \left[\frac{H(f) + H(-f)}{2}\right]^2 + \left[\frac{H(f) - H(-f)}{2}\right]^2 \\
&= \frac{[H(f)]^2 + [H(-f)]^2}{2}
\end{aligned}
\tag{29}
$$

and is therefore an even function of f.

When computing the discrete power spectrum of a sequence of length N, the 'negative frequencies' occur in the second half of transform array and H(-k) becomes H(N-k), giving

$$
\begin{aligned}
P(k) &= [H(k)]^2 && ; k = 0 \\
&= \frac{[H(k)]^2 + [H(N-k)]^2}{2} && ; k = [1..\frac{N}{2}],
\end{aligned}
\tag{30}
$$

where the special case when k is 0 can be eliminated if one computes N-k modulo N. Using this modulo index arithmetic, the two dimensional power spectrum $P(k_1,k_2)$ of the N_1xN_2 discrete Hartley transform $H(k_1,k_2)$ is likewise computed using

$$
\begin{aligned}
P(k_1,k_2) &= \frac{[H(k_1,k_2)]^2 + [H((N_1-k_1) \bmod N_1, (N_2-k_2) \bmod N_2)]^2}{2} \\
& k_1 = [0..\frac{N_1}{2}],\ k_2 = [0..\frac{N_2}{2}].
\end{aligned}
\tag{31}
$$

While the consequence of the modulo index arithmetic is easily understood in the one dimensional case, in two dimensions it leads to a non-intuitive traversal of the $H(k_1,k_2)$ matrix. Figure 8 shows one possible $H(k_1,k_2)$ traversal for an 8x8 power spectrum calculation. The numbers in the matrix correspond to the computation sequence taken: two elements marked with the same number correspond to an $H(k_1,k_2)$, $H((N_1-k_1) \bmod N_1, (N_2-k_2) \bmod N_2)$ pair, while bold numbered elements are squared in place.

0	30	31	32	**33**	32	31	30
8	1	2	3	4	5	6	7
16	9	10	11	12	13	14	15
24	17	18	19	20	21	22	23
29	25	26	27	**28**	27	26	25
24	23	22	21	20	19	18	17
16	15	14	13	12	11	10	9
8	7	6	5	4	3	2	1

Figure 8: $H(k_1,k_2)$ array traversal for power spectrum calculation.

Immediately one sees that the first row and the first column of the two dimensional power spectrum are computed exactly as in the one dimensional case. The remainder of the calculation operates on pairs of numbers diametrically opposing one another about the matrix's central element.

Computing the power spectrum of $H(k_1,k_2)$ in the present application was complicated by issues of memory utilization and data scaling. While the power spectrum could be computed in place, that would destroy the original Hartley transform, making further operations such as inverse transformation impossible. The size of the 'FFT Buffer' in which $H(k_1,k_2)$ was stored is also so large (512K for a 512x512 point transform) that temporary allocation of a comparably sized scratch buffer was an unattractive solution. It was therefore resolved to create a routine that computed the power spectrum and put the scaled result into the 8 bit deep off-screen graphics port (required by every window) without disturbing the original transform. This minimizes the routine's memory demands, but there is still a problem of data scaling. During the power spectrum calculation, two 14 bit numbers are squared and summed to give a 29 bit result. This number must then scaled down to 8 bits for storage using a mapping based on the power spectrum's dynamic range. But the power spectrum's dynamic range cannot be determined until the power spectrum calculation is complete! Consequently the power spectrum is actually calculated *twice*: The first $H(k_1,k_2)$ traversal finds the extrema of the power spectrum and the second $H(k_1,k_2)$ traversal uses the mapping based on these extrema to store the re-computed result.

Typically, power spectra are logarithmically scaled to bring out detail in low amplitudes. There are times, however, when a linear scaling is desirable or when an intermediate scaling between the linear and logarithmic extremes is most appropriate. The power spectrum routine was therefore designed to support both linear and log scaling as well as Nth root scaling for N in [2..9]. In offering this variety of scaling, it was necessary to use the Macintosh II's floating point unit, an 68881 or 68882. While the FPU is typically slower than the host processor (a 68020 or 68030) for simple arithmetic operations, its logarithm and exponentiation functions were convenient and, even at about 500 cycles apiece, speedier and more accurate than the alternatives using integer arithmetic which were considered.

Computing Convolution & Deconvolution

Like the Fourier transform, the Hartley transform has a convolution theorem which can be exploited to speed convolution calculations. In the Fourier domain, the convolution theorem is expressed by the transform pair

$$V_1(t) * V_2(t) \quad \Leftrightarrow \quad F_1(f)F_2(f),$$

where V_1 and V_2 are space domain functions with Fourier transforms F_1 and F_2. Using the Hartley transforms H_1 and H_2 of V_1 and V_2, the Hartley transform convolution pair is

$$V_1(t) * V_2(t) \quad \Leftrightarrow \quad \frac{1}{2}[H_1(f)H_2(f) - H_1(-f)H_2(-f) + H_1(f)H_2(-f) + H_1(-f)H_2(f)],$$

which can be verified by substituting equations (4) into the expression for Fourier convolution. By factoring out the even and odd components of H_2, this transform pair can be more succinctly expressed [15]

$$V_1(t) * V_2(t) \quad \Leftrightarrow \quad H_1(f)H_{2e}(f) + H_1(-f)H_{2o}(f).$$

While the Hartley convolution appears to be more complicated than its Fourier counterpart, note that the complex multiplication in Fourier convolution involves 4 multiplications and 3 additions — the same number of operations required by Hartley convolution. Furthermore, if either or both of the functions V_1 and V_2 is even or odd, then the Hartley convolution product reduces to a form similar to its Fourier counterpart: a point by point multiplication of the transforms [15].

Similarly, the deconvolution of V_1 and V_2 is computed by performing a complex division of F_1 and F_2. Denoting deconvolution as ⁂, the deconvolution Fourier transform pair becomes:

$$V_1(t) \text{ ⁂ } V_2(t) \quad \Leftrightarrow \quad \frac{F_1(f)}{F_2(f)} = \frac{F_1(f)F_2^*(f)}{||F_2(f)||^2}.$$

Hartley deconvolution involves a similar computation and gives the following transform pair:

$$V_1(t) \text{ ⁂ } V_2(t) \quad \Leftrightarrow \quad \frac{H_1(f)H_{2e}(f) - H_1(-f)H_{2o}(f)}{[H_2(f)]^2 + [H_2(-f)]^2}.$$

In contrast to the monadic operations discussed thus far, dyadic operations like convolution and deconvolution require that the relative scales of the transforms H_1 and H_2 be preserved. Recall that the scale or exponent of a two dimensional FHT is the sum of the maximum overflow incurred during both the row and column transforms. Unfortunately, keeping track of the scaling is not as simple as just adding exponents when multiplying and subtracting exponents when dividing.

Computing Hartley transform products and quotients also involves addition and subtraction operations that can influence the overall scale of the result. Furthermore, some portions of the Hartley product may encounter a net growth in their exponent during the calculation, while others may not. This situation is similar to the FHT calculation itself, where the maximum overflow must be monitored during the computation and the result scaled to reflect this exponent. Since time was limited[33] and implementing such block floating point calculations in assembly language is non-trivial, more simple implementation alternatives were sought.

The exponent of a two dimensional transform can be as large as $2\log_2 N$, or 22 bits for a 2048x2048 image (the largest allowed by the *Image* program). Since the dynamic range of the Hartley transform is 14 bits, the total dynamic range can exceed the 32 bit size of the 68000's registers. Because the scaled data could not be easily manipulated in an integer format, the Macintosh II's floating point unit was used.

---

[33]The code for the dyadic frequency domain operations was originally written to satisfy the requirements of a course in image processing and has been included here for completeness. Because a quick implementation was sought, not as much emphasis was placed on minimizing execution time.

Using floating point operations frees the programmer from the complications of the block floating point technique and results in code that is both easy to read and easy to write. The price paid for this luxury, however, is in execution speed, as is easily seen in the section on timing results below.

Computing Correlation

Another frequently used dyadic frequency domain operation is correlation. Like convolution, correlation may be more efficiently computed by exploiting the correlation theorem. The Fourier transform pair for correlation is

$$V_1(t) \bigstar V_2(t) \quad \Leftrightarrow \quad F_1(f)F_2^*(f),$$

where ★ denotes correlation. The Hartley transform equivalent is expressed

$$V_1(t) \bigstar V_2(t) \quad \Leftrightarrow \quad H_1(f)H_{2e}(f) - H_1(-f)H_{2o}(f).$$

Frequently the autocorrelation of an image is sought. It follows directly from the above relations that the Fourier and Hartley transform pairs for autocorrelation are

$$V_1(t) \bigstar V_1(t) \quad \Leftrightarrow \quad \|F_1(f)\|^2 \qquad \text{and}$$
$$V_1(t) \bigstar V_1(t) \quad \Leftrightarrow \quad \frac{1}{2}\left[H_1(f)^2 + H_1(-f)^2\right],$$

respectively. While the autocorrelation's symmetry reduces its operation count to half of that involved in a general cross correlation, this was not exploited here. Instead, the computational similarity of convolution, deconvolution and correlation was used to incorporate all three operations into one routine[34]. Since all three operations involved sequential accesses of the Hartley transform at both positive and negative frequencies, the $H(k_1,k_2)$ array traversal was identical to that for the power spectrum calculation. In order to conserve memory, it was again necessary to perform the calculation twice, once to monitor output extrema and develop mappings, again to store the mapped output. In contrast to the power spectrum calculation,

[34]Named Hcdc2BufsF for Hartley Convolution, Deconvolution & Corrleation of 2 Buffers using the FPU.

however, both traversals involved heavy use of the FPU in this routine, slowing it down significantly.

4.1 The Fully Evolved Two Dimensional FHT Algorithm

Using the FHT and utility routines in concert, the power spectrum of an image is computed as follows[35]:

0. Copy the memory from the image into the FHTBuffer, expanding the image's 8 bit dynamic range to fill the buffer's 16 bit dynamic range (routine DblMem).
1. Bit-reverse the rows of the FHTBuffer (BitRevRows).
2. Compute the FHT of each row of the FHTBuffer (FHT).
3. Normalize the scaling of all rows (ShiftRowsR).
4. Transpose the FHTBuffer (Transpose).
5. Bit-reverse the rows of the FHTBuffer (BitRevRows).
6. Compute the FHT of each row of the FHTBuffer (FHT).
7. Normalize the scaling of all rows (ShiftRowsR).
8. Transpose the FHTBuffer to its original orientation (Transpose).
9. Compute the two dimensional FHT from the row-column FHT (ToRCFHT).
10. Compute the power spectrum, storing the scaled output in the 8 bit deep image buffer of the newly created frequency domain window (PSFHT2D).
11. Swap image quadrants to reflect natural orientation of a diffraction pattern (SwapBBlock).

Because the actual FHT accounts for only a portion total power spectrum computation, it was important that the utility routines be optimized as well. The combined efficiency of the bit reversal, matrix transposition, power spectrum and FHT routines brought the total execution time down to a tolerable number of seconds. The source code incorporating these routines to perform the algorithm above was written in Pascal and can be studied in Appendix E. Complete timing results may be found in Appendix B, while the execution times of most interest are presented here.

Timing Results

The complete power spectrum computation was timed on two different machines, a Macintosh II with 8 Megabytes of RAM and an Macintosh IIci with 5 Megabytes of RAM. In both cases, *Image* was run under the Finder[36] and a logarithmically scaled power spectrum was

---

[35]The inverse transform differs only in that the power spectrum and block swapping operations are replaced with a routine (IntToByteF) that maps the 16 bit FHTBuffer into the 8 bit output image matrix.

[36]Multifinder was observed to increase the overhead from 72% to 85% for a 64x64 point power spectrum calculation on the IIci, so the Finder was used in all benchmarks.

calculated with the 'Mean Zero' and 'Clip Output to [1..254]' options disabled (these options are described in Appendix A). The total execution times for both machines are shown here:

2D FFT Size	Total Execution Time	
	II	IIci
64	0.8	0.9
128	2.6	2.0
256	10.5	6.5
512	44.9	28.0

Table 5: Total execution times for 2D power spectrum calculation.

The Macintosh IIci outperforms the Macintosh II by the ratio of their clock speeds (25/16 MHz), completing a 256x256 point transform and power spectrum calculation in only 6.5 seconds. This is quick enough to satisfy most users. For those that are less patient, the recently released Macintosh IIfx, which boasts a 40 MHz 68030, reduces this time to 4.5 seconds, while a 512x512 point transform on this machine takes only 18 seconds[37]. If one has lots of RAM and is very patient, the Macintosh II can compute a 1024x1024 point transform in 3 minutes and 16 seconds.

It is interesting to see how the total execution time is distributed among the different tasks involved in a power spectrum calculation. This is illustrated for the Macintosh II and IIci in Figures 9 and 10 respectively:

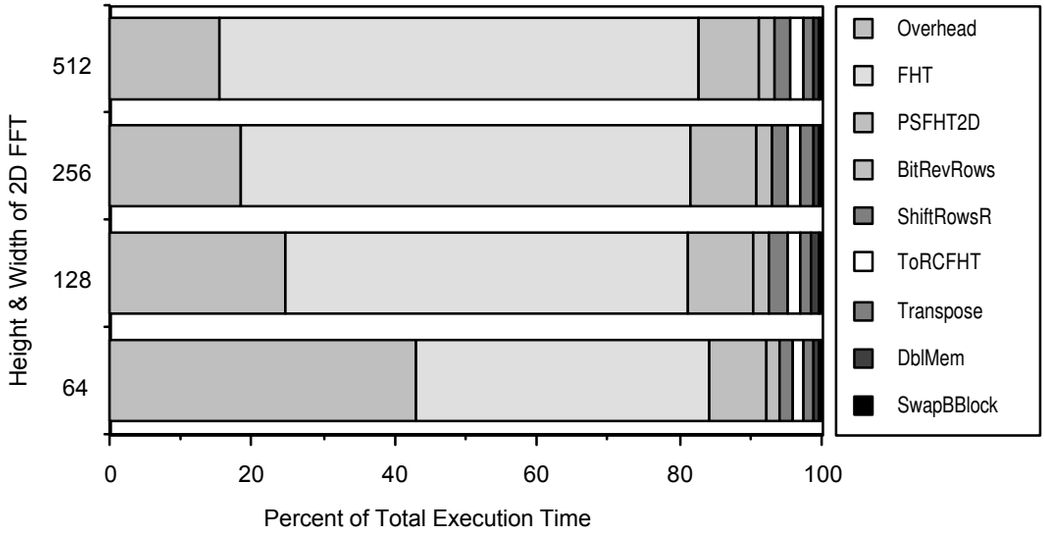


Figure 9: Percent of Total Power Spectrum Execution Time vs. Image Size for the Macintosh II computer.

[37]Wayne Rasband, private communication.

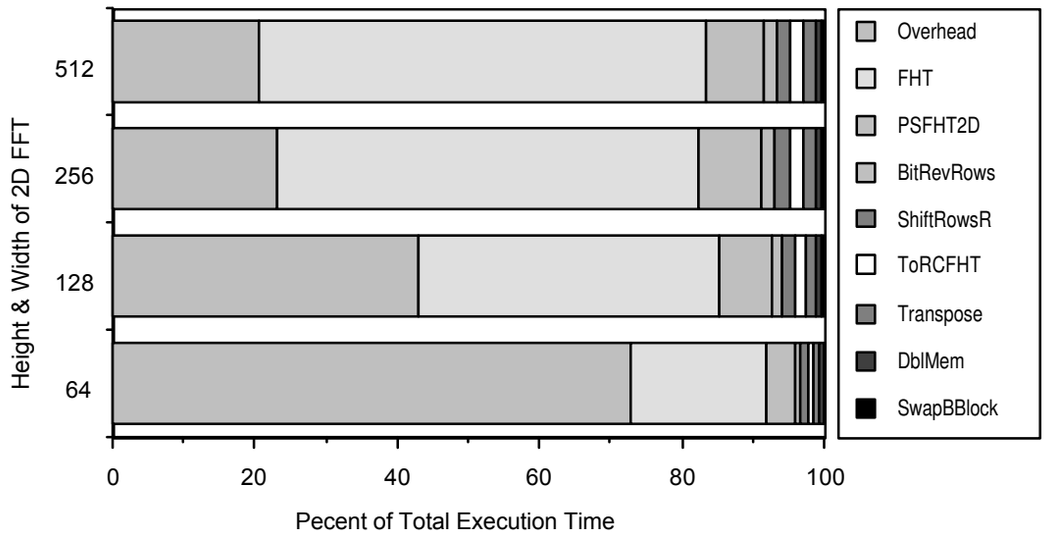


Figure 10: Percent of Total Power Spectrum Execution Time vs. Image Size for the Macintosh IIci computer.

In these figures, the relative amount time required by the FHT and each of the utility routines used in the power spectrum computation is plotted versus the dimension of the image being transformed. 'Overhead' is the difference between the the measured execution time and the execution time predicted from a simple summation of the execution times of the individual routines. Overhead, which includes memory allocation, opening a new window, and address calculations, dominates the total execution time on the IIci for 64x64 images. This is due in part to the memory allocation process.

The memory of a Macintosh application is divided into two regions, the Heap and the Stack [28]. All globally and locally declared variables are stored on the stack, while all dynamic memory allocation occurs in the Heap. In the Heap, memory is divided into disjoint blocks where each block has several attributes. A block's relocatability is its most important attribute; most blocks on the heap are relocatable, so that when an application requests a block of memory larger than the largest existing contiguous block, the Macintosh 'Memory Manager' can move relocatable blocks until it frees a contiguous block of the size requested. Non-relocatable blocks of memory can interfere with this process by fragmenting the heap into sections. For this reason, when an application requests a nonreolcatable block of memory, the Memory Manager does everything it can (by relocating other blocks) to place the non-relocatable block at the bottom of the Heap.

Macintosh windows and the FHTBuffer data structures are stored in non-relocatable blocks, so each time memory for these objects is requested, the Memory Manager does its best to place them where they will cause no fragmentation. In the process, so much memory movement may take place that a delay, for example in the creation of a new window, becomes apparent. This contributes to the computation overhead.

The total overhead decreases with increasing transform size but at 15% and 20% for the Mac II and IIci respectively, is still significant for 512x512 transforms. The FHT's proportion of the total execution time increases with transform size, accounting for 67% and 63% of the time spent computing a 512x512 power spectrum on the Macintosh II and IIci, respectively. Third most demanding is the power spectrum calculation itself, requiring about 10% of the total execution time for both machines. The remaining six operations, which include bit reversal, row shifting and transposition, together account for less than 10% of the total execution time. Because they contribute so little to the total computational load, these utility routines deserve as much credit for minimizing the total execution time as the FHT itself.

As described above, the dyadic frequency domain operations were implemented using the FPU, with the result that their performance is sub-optimal. The execution times of these routines on a Macintosh IIci computer are given in Table 6.

Image Size	Add/Sub	Convolve	Deconvolve	Correlate
32	0.04	0.09	0.12	0.09
64	0.14	0.37	0.48	0.37
128	0.56	1.48	1.93	1.48
256	2.24	5.89	7.69	5.92
512	8.96	23.57	30.77	23.68

Table 6: Execution times (in seconds) for the dyadic frequency domain operations vs. image size on the Macintosh IIci computer.

For a 512x512 image, the Hartley transform division operation corresponding to deconvolution takes 31 seconds on the Mac IIci which is longer than the time required to compute an FHT of similar size. Furthermore, this time does not include the overhead involved in allocating memory or opening windows. This clearly illustrates why use of the FPU should be minimized.

5. INCORPORATING THE FHT INTO *IMAGE*

Image consists of about 20,000 lines of uncommented THINK Pascal[38] source code, but it is well structured and fairly easy to read[39]. The code is divided into 14 units, most of which specialize in one functional area, such as the units for file I/O, graphics and analysis. The FFT extensions made to *Image* also reside in their own unit, and the modifications to the rest of the code were kept to a minimum.

In making the FFT extensions to *Image*, much thought and consideration were given to the details of the user interface and which FFT operations to support. During the development process, several questions had to be answered: How should power spectra be displayed? What tools does one need to operate on power spectra? How should these tools be made available? The way the answers to these questions evolved are discussed here, while detailed documentation of the FFT extensions made to *Image* may be found Appendix A.

5.1 User Interface Considerations

Introducing Bimodality into *Image*

Inside Macintosh, the five volume programmer's reference for the Macintosh, begins with a chapter on 'The Macintosh User Interface Guidelines.' To maintain the regularity among Macintosh applications that makes them so easy to use, developers are encouraged to follow these guidelines. For example, it is the developer's responsibility to support the 'select an object, then operate on it' metaphor common throughout Macintosh software.

Macintosh developers are also warned to avoid introducing *modes* into their code, where a mode is 'a part of an application that the user has to formally enter and leave, and that restricts the operations that can be performed while it's in effect' [28][40]. Many kinds of modality are

[38]THINK Pascal is a product of the SYMANTEC® Corporation.

[39]Because the source code for *Image* is so large, it has not been included here. It can be obtained with the documentation and sample image files via anonymous FTP from alw.nih.gov. Dartmouth community members can easily download from this site using the application *Fetch*.

[40]Anyone familiar with the UNIX editor *vi* knows how many have struggled with that program's Edit and Append modes. Even Microsoft Word 4.0 has four different modes (Galley View, Page View, Outline and Print Preview), but they merge well to enhance the program's functionality with a minimum of confusion.

unavoidable, however, such as the frequency and space domain modes introduced with *Image*'s FFT extensions. To make this modality as transparent as possible to the user, several measures were taken.

New Windows

First, it was decided that the power spectrum of an image would be displayed in a new window and not in the window of the original image. By keeping the space and frequency domains in separate windows, the user can toggle between domains by simply clicking in windows. If a frequency domain window is front-most, then the program operates in 'frequency domain mode'; clicking in a space domain window brings it to the front and puts the program in 'space domain mode.'

Although an image and its power spectrum are usually readily distinguished, the distinction between domains is reinforced by using different window types for each domain. The space domain windows are predefined by the Macintosh and have title bars containing horizontal stripes when the window is active. The window definition function, which is the code that draws the window, was modified for frequency domain windows to have a diagonal cross-hatching in the title bars is shown in Figure 11[41] . This visual aid reminds the user which domain he is currently in.

Figure 11: Title bars for Space and Frequency Domain Windows.

New Tool Behavior

Since the operations required in the space and frequency domains differ, the initial impulse was to provide different tools for each domain. This was actually implemented, requiring that the *Image*'s tool palette change depending on the type of the front-most window: When a frequency domain window was front-most, the tool palette provided frequency domain tools and

[41]The pattern used to paint the title bar may be found at offset $03B8 in the WDEF 0 resource.

when the space domain window was front-most, the tool palette provided space domain tools. This context sensitive tool palette is found in commercial drawing applications like SuperPaint™ that support two drawing layers requiring different tools. Because *Image*'s bimodality is very similar, using a dynamic tool palette seemed appropriate here as well.

There are, however, three reasons why this dynamic tool palette was abandoned. First, far fewer tools were needed in the frequency domain than in the space domain, making the frequency domain tool palette appear very sparse. Second, the functionality provided by the frequency domain tools differed little from the space domain's selection tools. Finally, redrawing the tool palette every time one switched domains was found unsightly. For these reasons, the tool palette's appearance was left unchanged and the functionality of the selection tools was made to depend on the domain in which they were used. Consequently, it also became necessary to alert the user of her error whenever he tried to use a space domain tool whose functionality was not implemented in the frequency domain[42].

When displayed in the format of a natural diffraction pattern, power spectra are even functions and therefore redundant. If one selects a part of the power spectrum at location (x, y), the part of the power spectrum diametrically opposite the origin at (-x, -y) is therefore also selected. The tool palette's selection tools were made to reflect this symmetry when a frequency domain window is front-most by highlighting both the region selected and its reflection through the origin. When a selection and its reflection overlap, their union is highlighted. Because this selection tool behavior is very different from the space domain, it reinforces the distinction between the two domains.

A New Menu

Since all of the FFT extensions made to *Image* are related, they are included in a single menu titled 'FFT.' Depending on the validity of a given operation, its corresponding menu item is either enabled or disabled (grayed). This prevents the user from trying to perform operations that don't make sense in the current context, like performing an FFT of an oval selection. It also simplifies the code by obviating the need to check for such spurious requests explicitly.

---

[42]This may not be as ideal as disabling the tools which have no use in the frequency domain.

5.2 Operations Supported

While the FFT extensions to *Image* at least had to support Fourier transformation and inverse transformation, there was a good deal of flexibility in the choice and implementation of additional operations. The final feature set summarized here, while more detail may be found in Appendix A.

Transformation and Inverse Transformation

When using *Image*'s rectangular selection tool in the space domain, holding down the shift key constrains the selection to be square[43]. Since the FHT algorithm used works only for square selections whose height and width are an integer power of 2, the selection can be constrained to these dimensions by holding down the command key[44]. The power spectrum of such a selection can then be computed by selecting 'FFT' from the FFT menu. This produces a frequency domain window in which the power spectrum of the selection is displayed with the desired scaling (logarithmic, Nth root or linear). Since the power spectrum is more often of interest that either the amplitude or the phase, the FFT extensions to image do not provide amplitude or phase display or manipulation. Saving the FHT to disk is also not supported, although the power spectra can be saved and re-opened as space domain images.

When a frequency domain window is front-most, its inverse FFT may be computed by selecting 'Inverse FFT' from the FFT menu. This produces a new space domain window of the same dimensions as the frequency domain window in which the inverse FFT is displayed.

Masking

Masking is a space domain operation used to eliminate cross-shaped power spectral signatures generated by the discontinuities in an image's periodic extension. By multiplying the image with a window function that goes to zero at its edges, masking eliminates the discontinuities in its periodic extension and therefore the 'cross' in its power spectrum as well. It

---

[43]This behavior is also common across Macintosh graphics applications and has its roots in Bill Atkinson's original MacPaint program.

[44]An option to compute the 2D FFT of a square region of any size by first expanding it (using bilinear interpolation) to the next integer power of 2 dimension was not yet implemented at the time of writing.

also alters the spectrum's leakage function, making tradeoffs between the narrowness of its central peak and the magnitude of its side lobes [14]. The transition width of the window function is user-definable, while its cross section shape may be Gaussian, cosine, linear or parabolic. The latter three correspond to the Hanning, Parzen and Welch window functions, respectively.

Passing & Filtering

Filtering and Passing are frequency domain operations that allow one to suppress or eliminate certain spatial frequencies in an image's power spectrum before inverse transformation. In analogy to Masking, Filtering and Passing multiply the power spectrum by a window function. In contrast to Masking, however, this window function can be very complex: Its shape is governed by the current frequency domain selection, while its transition width and extrema are user-definable. Filtering is used to suppress or eliminate the spatial frequencies in the current selection, while Passing does the opposite.

Threshold Zeroing

Threshold Zeroing is a frequency domain operation that can be used to remove spatial frequencies whose power spectral density falls within a certain threshold interval. During thresholding, *Image* highlights the pixels with values in the threshold interval and allows easy manipulation of the threshold interval's size and range. In the frequency domain, the threshold interval can be adjusted to select a range of power spectral densities to be removed with the Threshold Zero command. An example using Threshold Zero to eliminate periodic noise from an image is given under 'Applications,' below.

Dyadic Frequency Domain Operations

Once one or more frequency domain windows are displayed, dyadic frequency domain operations can be performed. These include addition, subtraction, multiplication, division and conjugate multiplication. In conjunction with the inverse transformation, these operations provide the ability to convolve, deconvolve and correlate two images. While they provide a logical extension

to *Image*'s FFT extensions, the dyadic operations are compromised by their lack of speed and, in the case of deconvolution, their accuracy.

Getting Around Undo — Automatically

The Mask, Pass, Filter and Threshold Zero operations can be performed either directly on an image or power spectrum or they can be performed 'automatically' on the copy of the image or power spectrum generated during the transform or inverse transform computation. Except for Masking, these operations are not undo-able (because that would require storing a duplicate of the large FHTBuffer), so they permanently alter the transform to which they are applied. The operation normally provided by 'Undo' is therefore provided for in 'automatic' calculation. Since each FFT and inverse FFT operates a copy of the original data, the desired operation can be automatically applied to this copy during the transform process, leaving the original undisturbed.

5.3 Using the Macintosh Toolbox

Resources as Lookup Tables

On the Macintosh, every kind of file (programs included) is divided into two *forks*, the *data fork* and the *resource fork*. The data fork is in a format completely determined by the application that created the file and is accessed with primitive file I/O routines. The resource fork, however, has a well defined structure and may be accessed with the Resource Manager routines. Programs frequently have no data fork whatsoever, so everything, from menus to text strings to the code itself, is stored in resources. By separating the components of a program this way, any individual component (except the code) can be changed without rebuilding the program. All that is necessary to translate many programs into another (human) language, for example, is to translate all the resources containing text strings.

The Resource Manager makes accessing resources very easy for the programmer and provides a pseudo-virtual memory behavior. With a simple 'GetResource' call, a program can gain access to a resource. The Resource Manager will allocate memory for that resource and

load it from disk if it has not already done so during a previous GetResource call. Resources can also be made purgeable, so that the memory they occupy can be recovered when needed.

The Resource Manager provides a very convenient means of storing the twiddle factor and lookup tables required for the FHT calculation. As resources, these tables can be changed independently of the code and benefit from the Resource Manager's dynamic allocation techniques. Typically, the memory required by these tables is so small that they are loaded once during the first FFT calculation and never purged.

6. APPLICATIONS

To illustrate the power and breadth of Fourier domain image processing techniques and how *Image* can be used to exploit them, five sample applications are described here. These are divided into three sections including examples of image restoration, image enhancement and pattern recognition.

6.1 Image Restoration

Noise Removal Using Spatial Filtering

If periodic noise corrupts an image, it can often be removed more easily in the frequency domain than in the space domain. Figure 12a shows a portrait so corrupted by periodic noise that the details are difficult to make out. In Figure 12b, the power spectrum of this portrait reveals that all of the power in the noise is conveniently focused in six localized high spatial frequencies (actually three and their reflections through the origin). One could remove these and all higher frequencies by performing a low pass operation on the power spectrum, but since the noise spikes are localized and of large amplitude, the Threshold Zero operation can be used to remove them more selectively. This is done by enabling thresholding and adjusting the threshold interval until only the noise spikes are highlighted. Using Threshold Zero then removes these spatial frequencies and the power spectrum appears as in Figure 12c. Inverse transformation finally gives the restored image in Figure 12d, which has been contrast enhanced to fill the dynamic range of the 8 bit pixels. While the noise dominates the signal in Figure 12a, its removal in Figure 12d reveals a very clean portrait.

By having a spectrum made up of delta functions at high frequencies, the noise in this example was unusually well behaved. In practice noise spectra will not be as localized and may interfere more directly with the spatial frequencies of interest. Even so, the spatial filtering technique is widely used in image restoration [19].

Figure 12: Spatial Filtering Example.(a) Image corrupted by noise. (b) Power spectrum of the Fourier transform of a, showing noise spikes. (c) Power spectrum after removal of noise spikes with Threshold Zeroing. (d) Inverse Fourier transform of c, showing restored image.

Motion Blur Removal Using Deconvolution

Figure 13a shows an image exhibiting a horizontal blurring similar to that obtained when photographing a quickly moving object with a slow shutter speed. As the planetary probe Voyager ventured to the outer planets, the waning sunlight demanded longer exposure times and such motion blur affected every image recorded. Fortunately, if the nature of the motion is known and the object being imaged is at a constant distance from the camera (a good approximation for planetary imaging), motion blurring can be easily corrected.

Motion blur can be modeled as the convolution of the image with a line whose length and orientation reflect the camera's shutter speed and the object's direction of motion, respectively (Figure 13a was generated by convolving the original image with a horizontal line 13 pixels long). Since convolving in the space domain corresponds to multiplying in the frequency domain, the Fourier transform of the undegraded image is multiplied by the Fourier transform of the line to produce the power spectrum of Figure 13b. The spacing and orientation of the zeroes of Figure 13b belie the length and orientation of the blurring line, allowing one to generate its Fourier transform independently. Dividing the Fourier transform corresponding to Figure 13b by the Fourier transform of the blurring line gives the power spectrum of Figure 13c. Although some of the original power spectrum is lost, inverse transformation shows this deconvolution operation successfully restores the original image in Figure 13d.

Note that the vertically oriented zeros of the blurring line's power spectrum would cause the quotient transform of Figure 13c to 'blow up' along these lines. Normally, the inverse filter is apodized to avoid this effect. With the limited dynamic range of our integer calculations, these singular quotients occur more frequently than when real numbers are used and consequently more of the quotient transform is lost (the white area in Figure 13c). While this limits the use of deconvolution in *Image*, it is still of practical use as this example shows.

Figure 13: Deconvolution Example. (a) Motion blurred image. (b) Power spectrum of Fourier transform of a. (c) Power spectrum of quotient of Fourier transform corresponding to b and Fourier transform of blurring line. (d) Inverse transform of c, showing successful restoration through deconvolution.

6.2 Image Enhancement

T4 Bacteriophage Tail Structure

The T4 bacteriophage shown in Figure 14a 'represents the extreme in structural complexity among bacterial viruses' and has been widely studied by biologists [29]. Its tail structure, a detail of which is shown Figure 14b, has been modeled as a tube of helically interwoven proteins. The structure apparent in most TEM images, however, makes the T4 tail appear like a stack of disks viewed on edge. Consequently, the power spectrum of Figure 14b, shown in Figure 14c, is dominated by the frequency of the disk spacing. Since both the image and its power spectrum are spatially calibrated, *Image* allows one to easily establish that the disk spacing is 3.9 nm by simply moving the mouse cursor over the power spectral peaks (more detail on this may be found in Appendix A).

Also discernible in the power spectrum of Figure 14c are spectral features of frequency similar to the disk spacing, but oriented at an angle. These have been enhanced in Figure 14d by multiplying all other spatial frequencies by a factor of 0.25 using *Image*'s Pass operation (Appendix A). Inverse transformation yields Figure 14e, which more clearly reveals the helical structure of the T4 tail.

While the emphasis of specific spatial frequencies has been used to advantage here, it should be emphasized that this technique can be exploited to provide 'enhancements' that are unrealistic. Microscopists generally emphasize or filter out complete annuli in the frequency domain, avoiding the preferential treatment otherwise given to spatial frequencies of a particular orientation. This technique may also be easily applied using *Image*'s FFT extensions.

Figure 14: T4 Bacteriophage Tail Structure Enhancement. (a) Image of T4 bacteriophage virus. (b) Detail of the 'tail' of a T4 bacteriophage. (c) Power spectrum of Fourier transform of b, showing spectral peaks corresponding to horizontal spacing dominant in b. (d) Same as c, but selected spectral peaks are emphasized by multiplying all other spatial frequencies by 0.25. (e) Inverse transform of d, showing enhanced helical structure of T4 bacteriophage tail.

Myofibril Structure

Skeletal muscle is made up of long, thin cells each containing many *myofibrils*. Myofibrils in turn contain long *myocin* and *actin* molecules forming a regular bundles of *thick* and *thin filaments*, respectively. When a skeletal muscle cell is viewed in cross section, as in the TEM image of Figure 15a, the regular arrangement of the thick filaments becomes apparent only after some scrutiny. The regularly spaced spectral features of Figure 15a's power spectrum, shown in Figure 15b, reveal that there is in fact a great deal of regular structure in the original image.

Using the Threshold Zero and Filter operations, the dominant peaks of the power spectrum in Figure 15b are isolated, producing the power spectrum in Figure 15c. The inner ring of six spectral peaks corresponds to the regular thick filament spacing, while the outer ring corresponds to the higher frequency thin filament spacing. The inverse transform of this modified power spectrum is shown in Figure 15d, revealing the basic periodic structure of Figure 15a, including that of the thin filaments which could not be seen in the original image.

Figure 15: Myofibril Structure. (a) TEM image of myofibril cross section. (b) Power spectrum of a, showing spectral features indicating a high degree of regular structure. (c) Modified power spectrum in which the spectral peaks of b have been isolated by using the Threshold Zero and Filter operations. (d) Inverse transform of c, revealing the basic periodic structure of the myofibril cross section. Inset shows the hexagonal arrangement of thick and thin filaments.

6.3 Pattern Recognition

Using Cross Correlation for Matched Filtering

A simple form of pattern recognition can be employed by using *Image*'s capability to cross correlate two images. In Figure 16a and 16b, a sequence of letters and the power spectrum of their Fourier transform are shown, respectively. Figures 16c and 16d likewise show the letter 'J' and the power spectrum of its Fourier transform. Calling the Fourier transforms corresponding to Figures 16b and 16d **A** and **B** respectively, the power spectrum of their conjugate product, $\mathbf{AB}^*$, is shown in Figure 16f. Figure 16e is the inverse transform of 16f and therefore the cross correlation of Figures 16a and 16c.

In the space domain, the cross correlation process can be viewed as sliding the image of J over the other letters and noting at each offset how well the two images match. Since they match best only at the offset where the two J images exactly overlap, the cross correlation function is maximized at this point. This *matched filtering* technique can be used in pattern recognition applications but is very sensitive variations in image orientation and scale. Furthermore, had 'I' been used instead of 'J', the cross correlation function would have had several maxima of equal amplitude where the 'I' matched the vertical components of other letters (e.g. 'H').

Figure 16: Correlation Example. (a) Image of letters. (b) Power spectrum of the Fourier transform of a. (c) Image of 'J'. (d) Power spectrum of the Fourier transform of c. (f) Power spectrum of the conjugate product of the Fourier transforms corresponding to b and d. (e) Inverse transform of f, the cross correlation of a and c, showing a maximum where the two 'J's coincide.

7. CONCLUSION

The combination of an appropriate algorithm, integer arithmetic and careful coding in 68000 assembly language has reduced the execution time required for FFT calculations to the point that they can be conveniently used in image processing applications on a mere Macintosh II. The incorporation of this capability into *Image* has made these capabilities available to everyone in the public domain. Professionals and students alike can profit from using *Image*'s new capabilities, whether it be for real image processing applications or simply to develop a better understanding of the frequency domain and its uses.

7.1 Future Development

As with any large project, there is much more work that can be done.

Efficient alternatives to the simple up/down rounding scheme were not explored and deserve further study. The stage alternate rounding technique is particularly promising [26] and should be quantitatively compared to other rounding schemes.

Further research into algorithms could also produce better candidates for 68000 implementation. Split-radix techniques are very attractive since they provide both compact size and minimum operation counts. As processors evolve, the finite register set limitation also becomes less stringent. For example, the 68040 processor has a 4K on chip data cache providing access speed approaching that of registers. Such an environment could be used to exploit the advantages of the larger and more complex algorithms like vector radix techniques. Many promising hybrid techniques have been also developed and deserve attention. These include a split-radix FHT, a split-vector radix FFT [30,31] and a vector-radix FHT [32].

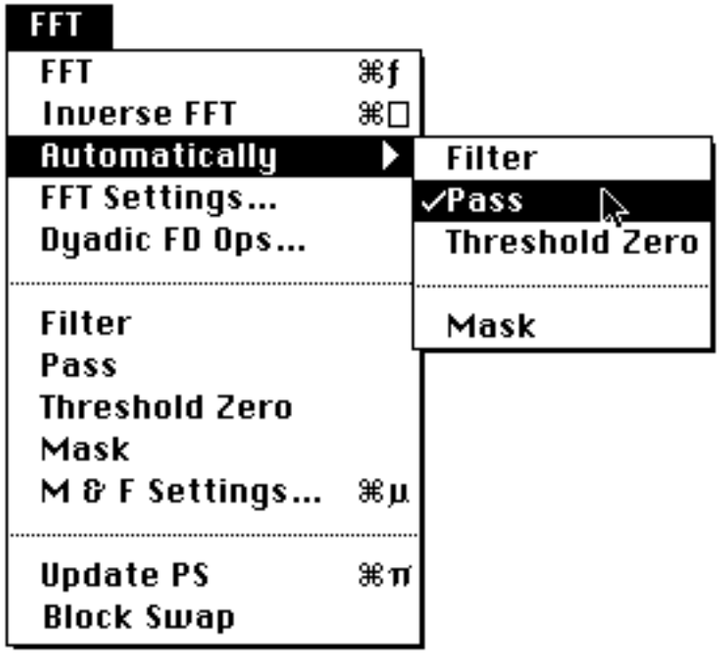
Image should also support saving the FHTBuffer to disk as well as its power spectrum. The ability to view and alter the amplitude and phase of images should also be supported. Finally, the dyadic frequency domain operations deserve optimization, since their speed could doubled without too much difficulty.

APPENDIX A
FFT EXTENSIONS TO *IMAGE* 1.25

Described herein are the extensions to *Image* 1.25 allowing frequency domain (power spectrum) display and editing of gray scale images. A description of each menu item and dialog box is followed below by a Quick Reference. If you have experience with earlier versions of this program or are simply ambitious, you may wish to skip ahead to the Quick Reference section which summarizes important key-combinations necessary to make full use of *Image*'s new capability.

• • •

The addition of FFT capability to *Image* has added one menu to the program:



FFT

The FFT menu item is enabled if the current selection is a valid FFT-able selection. Currently only square regions which are an integer power of 2 in size are FFT-able (the FFT Settings dialog box has a 'continuous selection size' setting, but that is not yet supported). Selections of

this size are made most easily by holding down the command key while making a selection with the rectangle selection tool.

When the FFT is computed, the power spectrum is displayed in a new window (titled 'FFT #' where # increases serially from 1) with a slashed title bar. This title bar indicates that the window is a frequency domain window and that different editing rules apply. No cutting or pasting is allowed in a frequency domain window, but its contents may be copied and pasted into a space domain window. Frequency domain editing is limited to the operations Filter, Pass and Threshold Zero (described below), and selections may only be made with the rectangle, oval and rounded rectangle tools. Furthermore, these tools behave differently than in the space domain as is described below under 'Making Selections In the Frequency Domain'.

During the FFT computation, the portion of the image being transformed is displayed in the newly created frequency domain window. If automatic masking (described below) is enabled, a masked version of the original image will appear in the frequency domain window while the computation is taking place.

Once the FFT computation is complete and the power spectrum is displayed in the new frequency domain window, the results window shows the mouse location in a new format. If the mouse is over an active frequency domain window, its location is displayed in polar coordinates. The angle is expressed in degrees (°), while the radius is expressed in either pixels per cycle (p/c) or, if a spatial scale calibration has been made using Set Scale…, in [units] per cycle (e.g. mm/c).

The contents of frequency domain windows can be saved as normal space domain images; the ability to save inverse-transformable power spectra is not yet supported.

The key-equivalent for FFT is Command-Option-F.

An Important Aside: Making Selections in the Frequency Domain

Selections in the frequency domain can be made with the rectangle, oval and rounded rectangle selection tools (the polygon and freehand selection tools will eventually be supported as well). These selection tools behave differently in the frequency domain because the frequency domain is radially symmetric and therefore redundant. When a selection is made with one of

these tools, the effective selection is the union of the original selection with its reflection through the origin. The 'marching ants' marquee portrays this graphically as you make the selection.

It is often desirable to center the selection on the origin and this can be done by pressing the command key while making the selection. As in the space domain, pressing the shift key during selection constrains the selection to have equal width and height. Finally, if you want to make complex selections, the active selection, or 'region of interest', can be added to and subtracted from using the same technique as in the space domain: To subtract a region from the currently active region, hold down the option key while making a new selection; when you're done it will be subtracted from the current region. Do the same while holding down the control key if you want to add to the current selection. It might seem complicated at first using e.g. option-command-shift to subtract a centered square region from the current region of interest, but after a while you'll get the hang of it. All the key combinations are summarized in the Quick Reference below, so you might want to print that page out and keep it around while you're learning.

Inverse FFT

The inverse FFT menu item is enabled when a frequency domain window is front-most. When the inverse FFT is computed, the inverse transform is displayed in a new space domain window with title 'I FFT #' where # increases serially from 1. During the inverse FFT computation, a copy of the power spectrum of the original transform is displayed in this window. If automatic Filtering, Passing or Threshold Zeroing (described below) are enabled, the copy of the power spectrum displayed will reflect these operations.

The key-equivalent for Inverse FFT is Command-Option-Shift F.

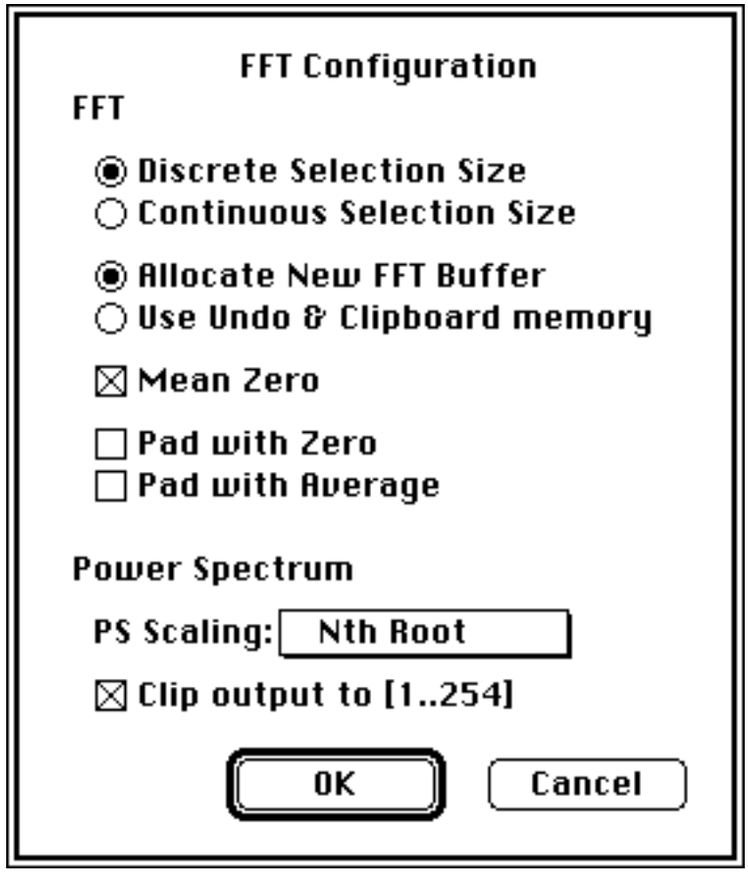
Automatically

The 'Automatically' sub-menu allows one to turn automatic Filtering, Passing, Threshold Zeroing and Masking on and off. The first three operations are used in Inverse FFTs, while Masking is used in forward FFTs. All four of these operations can either be done in place using the menu items in the main FFT menu or they can be done automatically to the copy generated

during the transform process. By doing these operations automatically, the original copy of the image or power spectrum remains unchanged, eliminating the need to restore or re-compute it.

FFT Settings…

Choosing FFT Settings… brings up the following dialog box:



Discrete & Continuous Selection Size

The first radio button pair allows one to choose between discrete and continuous selection sizes. Currently, only discrete selection sizes are supported, but a new version due out 'any time now' will allow square selections of any size to be transformed.

Allocate New FFT Buffer / Use Undo & Clipboard memory

The second radio button pair is only to be used by those with severe memory constraints (those with only 2 Megabytes of memory, for example). Normally, *Image* will try to allocate a

new power spectrum buffer for every new frequency domain window it creates, but if you choose 'Use Undo & Clipboard memory', the power spectrum will be computed in the memory normally reserved for the Clipboard and Undo buffer. This allows those with only 2M to compute larger transforms, but the power spectra generated in this way cannot be Inverse transformed, filtered or updated (re-computed).

Mean Zero

Mean Zero before FFT, when enabled, will cause the mean of the image being transformed to be computed and subtracted from the copy of the image being transformed. This effectively removes the DC component (the central pixel) of the transform and, because this component can otherwise dominate the power spectrum, improve the overall scaling.

Pad with Zero / Pad with Average

When computing the convolution or correlation of two images by performing dyadic frequency domain operations (described below), one may choose to pad the data to be transformed. When this option is enabled, the transform window will be twice the size (four times the area) of the original transform selection; during the transform computation, the selection will appear in the upper left quadrant of the new frequency domain window. If zero padding is enabled, the other three quadrants will be filled with zeros, while if average padding is enabled, they will be filled with the average value of the selection. Zero padding is necessary if the convolution of two images is to be subsequently sought; without zero padding, the cyclic, or circular, convolution will result. Likewise, padding is necessary for the correlation and autocorrelation operations, however, in this case more detail is brought out if the data is padded with its average value.

Power Spectrum Scaling

Power spectra are normally log-scaled, but this scaling sometimes brings out more spectral detail at low spatial frequencies than is desired. For this reason, Nth Root scaling, for N in [2..9] and Linear scaling are also supported. After the scaling has been changed using this pop-up menu, the power spectrum displayed in the front-most frequency domain window can be updated to reflect this scaling with the Update PS menu item.

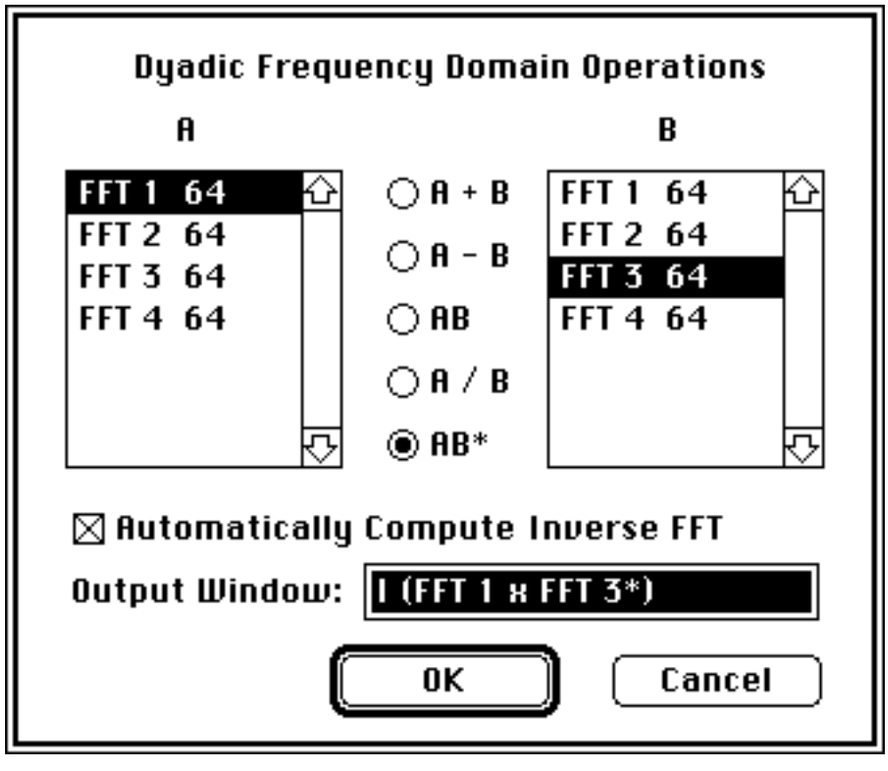
Clip Output to [1..254]

When the power spectrum is computed, it normally is scaled to fill the output range [0..255]. Thresholding, however, allows the user to threshold only pixels in the range [1..254]. Therefore, if the user wants to use Threshold Zero (described below), he will be able to threshold any portion of the power spectrum when 'Clip Output to [1..254]' was enabled prior to the last power spectrum update.

All FFT Settings can be recorded permanently by choosing the 'Record Preferences' menu item in the file menu.

Dyadic FD Ops…

New in this version of image is the support of Dyadic Frequency Domain Operations. With them, *Image* is now capable computing convolutions, deconvolutions, correlations and autocorrelations.



When a frequency domain window is open, this menu item is enabled and choosing it will bring up the dialog box above. The titles of the currently open frequency domain windows appear in two lists, followed by their width (=height) in pixels (Dyadic operations are allowed

only on images of equal dimensions). Between these two lists, five radio buttons allow one to select the frequency domain operation desired. In addition to the four common operations, conjugate multiplication is also supported for correlation computations.

The result of the dyadic frequency domain operation appears in a new window, the title of which may be entered into the Output Window field. A default title generated from the selections is provided until one types or clicks in this field.

Automatically Compute Inverse FFT

Often one is more interested in the inverse transform of a dyadic frequency domain operation than in the transform itself. When this is the case, the inverse transform can be automatically computed from the result of the frequency domain operation by enabling this item. The new window created by the operation is then a space domain window which, during the inverse transform computation, displays the power spectrum of the dyadic operation's result. Not only does this proceed automatically, but it spares the memory otherwise used by the intermediate frequency domain window and its buffer.

Addition and Subtraction

The addition and subtraction operations in the frequency domain are equivalent to those in the space domain. They therefore do not hold much interest except to verify the linearity of the transform process. Still, they are included here for completeness.

Multiplication (Convolution)

Multiplication in the frequency domain translates to convolution in the space domain; two images can be convolved by transforming each of them, multiplying their transforms and inverse transforming the result. While the same computation can be done directly in the space domain using *Image*'s convolution feature, the space domain kernels must be typed in as text files and are limited in size to 63x63. Using the frequency domain operation removes the size restriction and allows one to draw the convolution kernels using *Image*'s drawing tools. While the set of practical applications that demand this new flexibility may be small, the process is a great educational tool.

If data padding (see FFT Settings… above) is disabled, the inverse transform of the product of two transforms will produce the *cyclic* convolution of the two original images. This arises from the sampling-induced periodicity of the image plane. Since the Fourier transform of a sampled image is equivalent to the transform of an infinite periodic replication of the image, the shifting inherent in the convolution operation results in the image overlapping its periodic extension. This creates a circular convolution and the wrap around affect can be easily seen by experimenting with this operation.

In order to produce normal, not circular, convolution, one can avoid the overlap artifact by padding the image to be convolved with zeros. Unlike the autocorrelation function, however, convolution is sensitive to the placement of the extra zeros. By placing them to the lower right (see description under FFT Settings…), no artificial translation is induced.

Division (Deconvolution)

Division in the frequency domain corresponds to deconvolution in the space domain and serves as the inverse of the multiplication (convolution) operation. For example, if one first multiplies the transforms FFT1 and FFT2 together, leaving the result in a new frequency domain window, FFT1 x FFT2, and then divides this spectrum by either FFT1 or FFT2, the complementary FFT results. Unfortunately, however, the nature of the division operation creates output with a wide dynamic range (not to mention some singularities) that cannot be fully accommodated by the 16 bit integer arithmetic used throughout these routines. Consequently, often large portions of the quotient transform are lost and the resulting inverse transform suffers accordingly. While this limits the practical use of this deconvolution operation, it is nonetheless impressive to see how well the original object can be recovered from its image when one knows the point spread function with which it was convolved.

Conjugate Multiplication (Correlation)

Conjugate multiplication in the frequency domain translates to correlation in the space domain. The wide use of correlation and autocorrelation operations in imaging science makes this perhaps the most important dyadic frequency domain operation supported by *Image*.

The best way to compute the autocorrelation of an image is to select 'Pad with Average' in FFT Settings before Fourier transforming. This accomplishes two things. First, data padding avoids the characteristic wrap around affect to which correlation, like convolution, is subject. Second, by padding the data with its average value instead of with zeros, more detail is brought out in the autocorrelation function. By padding with zeros, the extended data looks like a modulated rectangle function. In the autocorrelation operation, the signature of the rect function dominates over that of the modulation (your data). By padding with the average value of the data, this rect function autocorrelation signature is removed and the autocorrelation features of the data stand out.

Finally, the result of the autocorrelation operation produces an image that is inherently 'block swapped' or in its 'wrap around' state. If you prefer the more common 'un-swapped' representation of the data, it can be rearranged using the Block Swap menu item described below.

Like the FFT Settings, the preferred Dyadic Operation settings can be recorded permanently using 'Record Preferences.'

Filter & Pass

Once a region is selected in the frequency domain, these menu items become enabled and the selection can be either Filtered or Passed. 'Filter' removes or suppresses the spatial frequencies in the current selection while 'Pass' performs the complementary operation. The details of these operations are described under 'Mask & Filter Settings…' below.

Threshold Zero

Another means of frequency domain editing is provided by the Threshold Zero operation. When thresholding in the frequency domain, this menu item is enabled and allows one to select, by using the LUT tool to manipulate the threshold 'density slice', which part of the power spectrum to filter out. Choosing Threshold Zero will zero out all parts of the power spectrum within the thresholding interval (colored red by default). A typical application of this would be to remove all components of the power spectrum except the most dominant peaks.

In order to make the entire power spectrum fall within the threshold-able interval [1..254], the ‘Clip Output to [1..254]’ option in the FFT Settings dialog box should be enabled.

Like the Filter and Pass operations, Threshold Zeroing can be performed automatically on inverse transformation to avoid permanent (non-undoable) alteration of the original power spectrum.

Mask

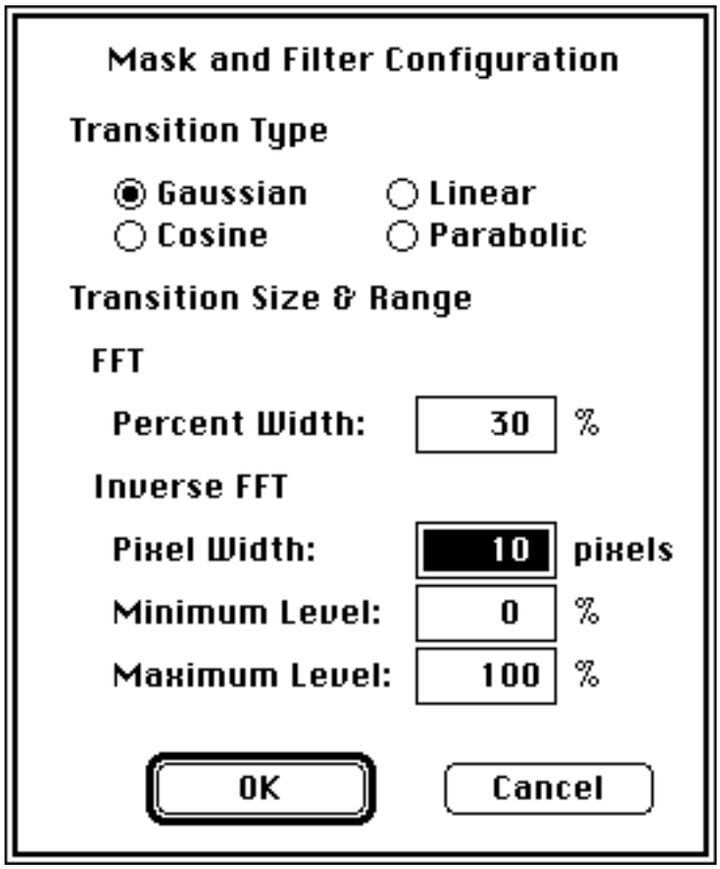
Masking is the space domain analog of Passing: when Masked, an image selection is given a smooth transition to zero at its edges. In contrast to Passing, however, masking is less flexible because it is only enabled for rectangular selections in the space domain and regions inside the mask transition region are passed completely while those outside are zeroed completely.

Why use masking? The discrete two dimensional Fourier transform of an image is effectively the transform of the periodic extension of the image. Tiling the image in the plane can give rise to sharp edges where the right (top) edge of the image meets the left (bottom) edge of its periodic extension. This in turn increases the image’s power spectral density in the horizontal and vertical directions; in extreme cases the power spectrum of an unmasked image can be dominated by this cross shaped signature. By masking, however, one guarantees that an image’s periodic extension has no sharp edge discontinuities, effectively removing the cross-shaped artifact from the power spectrum. Be aware, however: masking (multiplying) in the space domain is equivalent to convolving in the frequency domain; by masking an image, you are convolving its transform with the transform of the mask, which can also have undesirable effects. In general, masking is used to remove the cross from a power spectrum; if the power spectrum is to be modified and inverse transformed, masking is not used.

Like Filtering, Passing and Threshold Zeroing, Masking can be done automatically during the forward FFT, eliminating the need modify the original image. Unlike the other operations, however, masking *is* undoable.

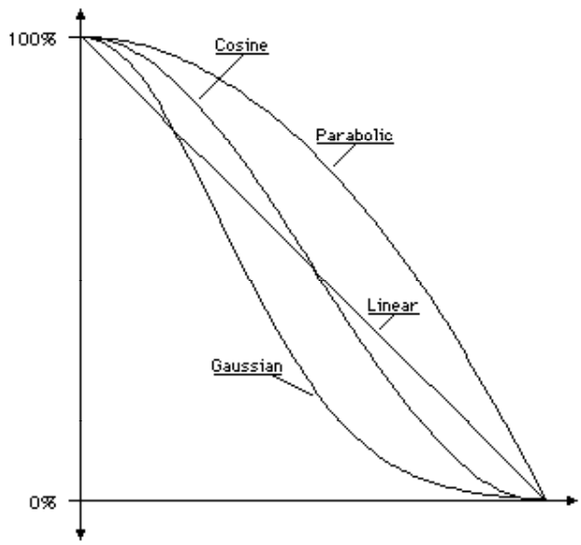
Mask & Filter Settings…

Choosing ‘M & F Settings…’ brings up the following dialog box:



Transition Type

Both the space domain operation Mask as well as the frequency domain operations Filter and Pass support the four different transition types shown below:



These are the transitions that Masks or Filters take from full scale to zero. Both the space and frequency domain images are altered by such Masks and Filters using a multiply operation; each pixel in the transition region is reduced in proportion to the transition function at that point.

Percent Width

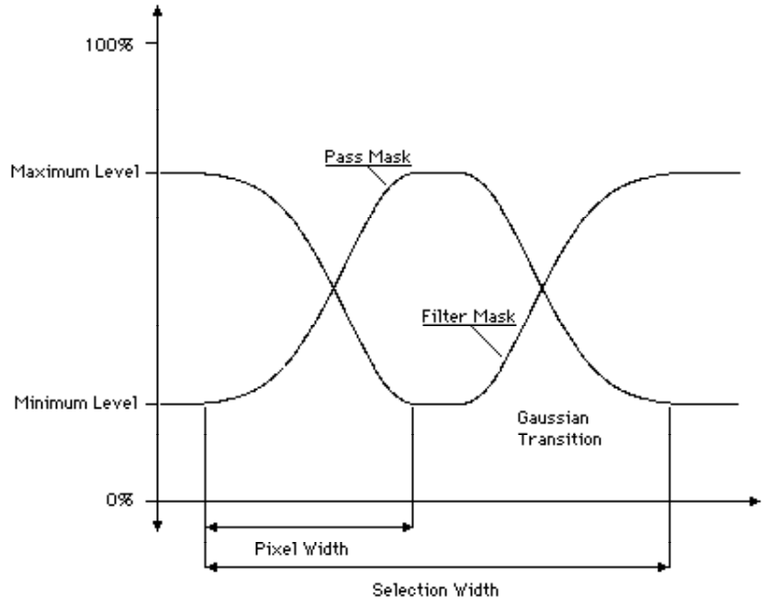
For Masking, the width of the transition region is specified as a proportion of the minimum dimension of the selection rectangle, denoted Percent Width. A Percent Width of 100% will set the width of the transition region to half of the minimum dimension of the selection rectangle and the entire region inside the rectangle will be altered by the mask. Any percent width in [0..100%] can be entered in the M & F Settings dialog box.

Pixel Width, Minimum & Maximum Level

Since the Filter and Pass operations can be applied to much more complicated regions in the frequency domain, the transition width is specified in pixels instead. Furthermore, since it may be desirable to suppress, instead of completely eliminate, particular spatial frequencies, you can specify a maximum and minimum level of the transition function.

For example, if the user wants to suppress all spatial frequencies outside the selection by 50% while completely passing all spatial frequencies within the selection, he can set the transition pixel width to 0, the minimum level to 50%, the maximum level to 100% and use the Pass operation.

A graphical representation of a Pass and Filter mask cross section is shown below:



The key equivalent for Mask and Filter Settings is Command-Option-M.

All Mask & Filter Settings can be recorded permanently by choosing the 'Record Preferences' menu item in the file menu.

Update PS

At any time a frequency domain window is front-most, its power spectrum can be re-computed using Update PS. If, however, the power spectrum was computed using the 'Use Undo & Clipboard memory' option enabled, the power spectrum cannot be re-computed because the memory used in the original power spectrum computation has likely been corrupted by Undo and Clipboard operations.

Typically, Update PS is used to re-compute the power spectrum after the default scaling or the 'Clip Output to [1..254]' variables have changed.

The key equivalent for Update Power Spectrum is Command-Option-P.

Block Swap

Block swapping is enabled for space domain windows whose width and height are equal and an integer power of two. This happens to include all windows produced by inverse Fourier transformation. Why Block Swap? What *is* Block Swap?

Block Swapping (I'm sure it has a more official name) is performed automatically every time a power spectrum is computed. If you think of an image's origin as being at its geometric center, block swapping simply swaps quadrant 1 of the image with quadrant 3; likewise for quadrants 2 and 4. In the normal, un-swapped, state, the power spectrum's central peak is distributed among the four corners of the image matrix. While this is the format used in all computations, it doesn't correspond to nature's FFT analog, the diffraction pattern, so most spectra are block swapped before display. This menu item allows block swapping for space domain windows only, however, because it is useful in two other contexts.

First, convolution kernels are typically arranged in a block swapped or 'wrap around' state. This puts the maximum of the kernel at matrix index 0,0 (upper left corner) so that convolving with the kernel will not result in a translation. Second, the cross and auto-correlation operations result in inherently un-swapped space domain images, yet like power spectra, correlation functions are often displayed in block swapped format. The Block Swap menu item lets you choose the format you desire.

FFT QUICK REFERENCE

Space Domain

- With the rectangle selection tool, hold down the command key to make a square selection an integer power of 2 in size.
- To make the largest FFT selection allowable in the current window, hold down the command key while double clicking the rectangle selection tool.

- Masking vignettes a rectangular image selection so that it has a smooth transition to zero at its edges.
- Automatic Masking, when enabled, will cause the copy of the image selection used in computing the FFT to be masked, leaving the original selection unchanged.

Frequency Domain

- Only the rectangle, oval and rounded rectangle selection tools allow selections to be made (Some other tools work too, like the cross-section plot tool).
- Holding down the command key will center these selections on the origin.
- Holding down the shift key will constrain these selections to have equal height and width (whether or not they are centered at the origin).
- Holding down the option key before making a new selection will cause it to be subtracted from the current selection
- Holding down the control key before making a new selection will cause it to be added to the current selection.

- Filtering removes or suppresses the spatial frequencies in the current selection.
- Passing removes of suppresses the spatial frequencies outside the current selection.
- Threshold Zeroing zeroes out all spatial frequencies within the current threshold density slice.
- Automatic Filtering, Passing and Threshold Zeroing performs these operations on the copy of the power spectrum used in computing the Inverse FFT without altering the original power spectrum.

- Update Power Spectrum re-computes the power spectrum to reflect the current scaling default (chosen in FFT Settings).

Key Equivalents

- FFT Command-Option-F
- Inverse FFT Command-Shift-Option-F
- M & F Settings… Command-Option-M
- Update PS Command-Option-P

APPENDIX B
EXECUTION TIMES FOR THE FHT ROUTINE LIBRARY

N	512 KE	Plus	SE	SE 30	II	IIcx	IIci
64	10.3	10.3	8.9	2.2	2.5	2.1	1.4
128	24.5	24.5	21.2	5.1	5.7	5.1	3.3
256	57.4	57.4	49.7	11.9	13.0	11.9	7.6
512	132.0	131.8	114.3	27.2	29.5	27.3	17.2
1024	298.3	297.7	258.7	61.3	66.1	61.4	38.8

Table B1: FHT execution times in milliseconds vs. sequence length and Macintosh computer type.

	Height & Width of Image				
Procedure	32	64	128	256	512
ShiftRowsR	1.7	7.8	30.6	120.0	480.0
BitRevRows	1.7	6.7	28.3	122.2	487.2
MeanZero	37.2	45.0	73.9	188.9	651.1
ClipMinMax	2.2	8.3	35.0	140.6	564.4
SwapBBlock	0.6	1.7	7.8	31.1	123.9
Transpose	1.7	5.6	21.1	85.0	339.4
ToRCFHT	3.3	12.2	48.3	189.4	753.3
DblMem	1.7	6.1	25.6	103.9	415.6
IntToByteF	11.1	45.6	182.2	729.4	2917.8
PSFHT2D Linear	13.3	53.3	213.3	853.3	3406.7
PSFHT2D 5th Root	16.7	63.3	253.3	1013.3	4043.3
PSFHT2D Log	16.7	63.3	243.3	963.3	3850.0
AddSub2BufsF	60.0	241.7	965.0	3865.0	15455.0
Convolve	161.1	644.4	2577.8	10300.0	41211.1
Deconvolve	211.1	844.4	3377.8	13527.8	54116.7
Correlate	166.7	650.0	2605.6	10422.2	41688.9

Table B2: Utility routine execution times in milliseconds vs. image size for the Macintosh II.

Procedure	Height & Width of Image 32	64	128	256	512
ShiftRowsR	1.1	3.9	16.7	66.1	262.2
BitRevRows	0.6	3.9	13.9	63.9	268.9
MeanZero	22.2	26.1	41.1	100.6	338.3
ClipMinMax	1.1	5.0	20.6	83.3	333.3
SwapBBlock	0.6	1.7	6.1	23.9	95.6
Transpose	1.1	3.9	14.4	57.8	232.8
ToRCFHT	2.2	8.3	31.1	123.3	490.0
DblMem	1.1	3.9	15.0	59.4	238.3
IntToByteF	7.8	31.1	125.0	498.3	1995.0
PSFHT2D Linear	10.0	33.3	130.0	530.0	2113.3
PSFHT2D 5th Root	10.0	40.0	153.3	603.3	2413.3
PSFHT2D Log	10.0	36.7	146.7	580.0	2320.0
AddSub2BufsF	35.0	140.0	560.0	2240.0	8956.7
Convolve	88.9	372.2	1477.8	5894.4	23572.2
Deconvolve	116.7	483.3	1927.8	7694.4	30772.2
Correlate	88.9	372.2	1477.8	5922.2	23683.3

Table B3: Utility routine execution times in milliseconds vs. image size for the Macintosh IIci.

APPENDIX C
USE WITH OTHER LANGUAGES AND OTHER 68000 SYSTEMS

The FHT code and many of the utility routines are written using generic 68000 instructions and can be ported to any other 68000 based machine. For example, it would not take much effort to bring these routines up on a 680x0 based Sun workstation.

Incompatibilities may exist for those utility routines that make use of the Macintosh II's floating point unit. These routines (including PSFHT1D, PSFHT2D, IntToByteF, AddSub2BufsF & Hcdc2BufsF) are clearly marked in the source listing in Appendix F. Based on their description under 'Details of the Utility Routines' and the inline documentation, the enterprising individual could easily write high level routines to perform the same functions.

Otherwise, all of the code is written for a generic 68000 and should be very portable.

Pascal Calling Conventions

Because *Image* is written in Pascal, the FHT and utility routines expect to receive their parameters on the stack in the format typical of Pascal on the Macintosh. Upon routine entry, the top of the stack contains return address. The first operations typically performed by the routine are a) a LINK to create a stack frame and allocate space on the stack for local varables, and b) to save the contents of the 68000 registers changed by the routine by pushing them on the stack. This leaves the stack in the state shown in Figure C1.

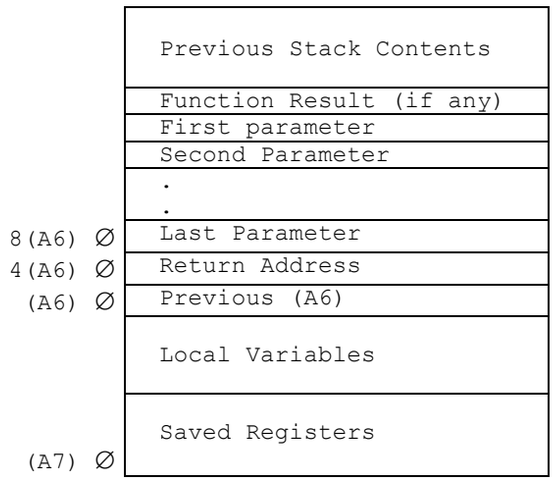


Figure C1: Stack Frame.

Any of the routines listed in Appendix F can be used to develop an understanding of Pascal calling conventions, as the routine entry and exit sequence is uniform throughout the code and reflects the conventions outlined in Inside Macintosh, Volume 1, Chapter 4 [28]. Furthermore, if one understands the calling conventions of one's preferred language, it may be possible to use the routine library from that language with little or no change to the assembly source code.

APPENDIX D
RESOURCE FORMATS

BREV

The custom Macintosh resource type 'BREV' was generated for use with the FHT routines to speed the bit reversal operation as described under 'Details of the Utility Routines.' It is a lookup table consisting of a sequence of 16 bit words. The first word, N, is the number of swaps necessary to bit reverse the sequence. Following this are N pairs of words which are the addresses of data pairs to be swapped. Since all operations are on two byte words, the addresses are all even. This format is shown here:

N
Swap 1 Address 1
Swap 1 Address 2
Swap 2 Address 1
Swap 2 Address 2
. . .
Swap N Address 1
Swap N Address 2

Figure D1: BREV resource format.

Since a unique BREV lookup table is required for each integer power of 2, instances of this resource for transform lengths in [64, 128, 256, 512, 1024, 2048] are provided. The resource ID of each instance is its length *plus* 124. The BREV resource ID for N=64 is therefore 188. If longer sequences are to be bit reversed, larger BREV tables must be generated.

TWID

The custom resource type TWID was generated to hold the twiddle factors necessary in the computation of the FHT. It consists of a sequence of 32 bit longwords in the following format:

High Word	Low Word
2^{15}*cos(2□*0/4096)	2^{15}*sin(2□*0/4096)
2^{15}*cos(2□*1/4096)	2^{15}*sin(2□*1/4096)
2^{15}*cos(2□*2/4096)	2^{15}*sin(2□*2/4096)
2^{15}*cos(2□*3/4096)	2^{15}*sin(2□*3/4096)
.	.
.	.
.	.
2^{15}*cos(2□*1023/4096)	2^{15}*sin(2□*1023/4096)

Figure D2: TWID resource format.

This lookup table is 4K in size and can accommodate transforms up to 4K in length. If longer transforms are to be calculated, then a larger TWID resource must be generated using this format. In addition, the variable CSAdIncr (for cosine/sine address increment) in source code for FHT.a (Appendix F) must also be increased accordingly[45].

[45]This can be done without reassembling the code by simply modifying the word at hex offset $126 in the compiled code. For the current maximum transform length of 4K, this number is $0800 (2K). For a TWID resource twice the size (accommodating transform lengths up to 8K), this number should be doubled to $1000 (4K).

APPENDIX E
THINK PASCAL SOURCE CODE

The complete Pascal source code for the FFT extensions to *Image* is not included here because it makes little sense out of the context of the rest of the *Image* source code. Since the *Image* program is so large, the source code cannot be reproduced in its entirety here. When the FFT extensions are permanently incorporated into the *Image* program, their source code will be available via anonymous FTP from alw.nih.gov. Dartmouth community members can most easily download files from this site using the *Fetch* application. Until that time, the source code for *Image* and its FFT extentions may be obtained from Charles.Daghlian@mac.dartmouth.edu.

The rest of this Appendix includes two routines. The first implements the two dimensional FHT algorithm, while the second is a real valued FHT routine that can be used to test the accuracy of the integer FHT routine in Appendix F.

RCFHT

RCFHT computes a two dimensional FHT using the 11 step algorithm outlined under 'The Fully Evolved Two Dimensional FHT Algorithm'. It appears here as implemented in *Image* and makes use of several of the utility routines described under 'Details of the Utility Routines.'

```
function RCFHT(x: ptr;
               rowWords: integer;
               var scale: integer): boolean;

  const
    CheckInterval = 64;

  type
    IntArr = array [0..0] of integer;
    IntArrPtr = ^IntArr;

  var
    BitRevH, TwidH: handle;
    shiftArr: IntArrPtr;
    maxShift, r, ShiftMag, rowBytes: integer;
    bufSize, offset: longint;

  procedure CheckAndQuit;

    begin
      if CommandPeriod then
        begin
          HUnlock(Handle(BitRevH));
          HUnlock(Handle(TwidH));
          Dispose(ptr(shiftArr));
          exit(RCFHT);
        end;
    end;

  begin
    RCFHT := false;
    shiftArr := IntArrPtr(NewPtr(BSL(rowWords, 1)));
    bufSize := longint(rowWords) * rowWords;
```

```
    BitRevH := GetResource('BREV', rowWords + 124);
    TwidH := GetResource('TWID', 129);
    if (TwidH = nil) or (BitRevH = nil) then
      begin
        PutMessage('NIL resource handles in RCFHT9', '', '');
        exit(RCFHT);
      end;
    Hlock(Handle(BitRevH));
    BitRevRows(x, rowWords, BitRevH^);
    rowBytes := rowWords + rowWords;
    maxShift := 0;
    offset := 0;
    Hlock(Handle(TwidH));
    for r := 0 to rowWords - 1 do
      begin
        ShiftArr^[r] := FHT(ptr(ord4(x) + offset), rowWords, TwidH^);
        offset := offset + rowBytes;
        if ShiftArr^[r] > maxShift then
          maxShift := ShiftArr^[r];
        if (r mod CheckInterval = 0) then
          CheckAndQuit;
      end;
    scale := maxShift;
    ShiftRowsR(x, rowWords, maxShift, ptr(shiftArr));
    transpose(x, rowWords);
    BitRevRows(x, rowWords, BitRevH^);
    HUnlock(Handle(BitRevH));
    maxShift := 0;
    offset := 0;
    for r := 0 to rowWords - 1 do
      begin
        ShiftArr^[r] := FHT(ptr(ord4(x) + offset), rowWords, TwidH^);
        offset := offset + rowBytes;
        if ShiftArr^[r] > maxShift then
          maxShift := ShiftArr^[r];
        if (r mod CheckInterval = 0) then
          CheckAndQuit;
      end;
    scale := scale + maxShift;
    HUnlock(Handle(TwidH));
    ShiftRowsR(x, rowWords, maxShift, ptr(shiftArr));
    transpose(x, rowWords);
    ToRCFHT(x, rowWords);
    Dispose(ptr(shiftArr));
    RCFHT := true;
  end; { RCFHT }
```

DFHT

DFHT is a high level implementation of the FHT, included here for those who want more than just assembly source. This implementation is not particularly beautiful, but its structure (and variable names) is similar to that used in the assembly language implementation. Namely, the first two stages are joined in one stage of radix 4 butterflies and the first butterfly of each subsequent group is computed without multiplications. Further optimizations are not implemented here.

```
const
  maxStore = 4096;
  maxStoreLess1 = 4095;

type
  rArr = array[0..maxStoreLess1] of real;
  rArrPtr = ^rArr;

procedure dfht(var x: rArr;
               inverse: boolean);
{ an optimized real FHT }

  var
    i, stage, gpNum, gpIndex, gpSize, numGps, Nlog2: integer;
    bfNum, numBfs: integer;
    Ad0, Ad1, Ad2, Ad3, Ad4, CSAd: integer;
```

```
  C, S: rArrPtr;
  rt1, rt2, rt3, rt4: real;
  theta, dTheta: real;

begin
  if maxN < 4 then
    begin
      writeln('Sequence Length must be >= 4 in Fast FHT');
      ExitToShell;
    end;

  C := rArrPtr(NewPtr(SizeOf(rArr)));
  S := rArrPtr(NewPtr(SizeOf(rArr)));         { Should Check for NIL pointer here }

  theta := 0;
  dTheta := 2 * pi / maxN;
  for i := 0 to maxN div 4 - 1 do
    begin
      C^[i] := cos(theta);
      S^[i] := sin(theta);
      theta := theta + dTheta;
    end;

  Nlog2 := log2(maxN);
  BitRevRArr(x, Nlog2); { bitReverse the input array }

  gpSize := 2; { first & second stages - do radix 4 butterflies once thru }
  numGps := maxN div 4;
  for gpNum := 0 to numGps - 1 do
    begin
      Ad1 := gpNum * 4;
      Ad2 := Ad1 + 1;
      Ad3 := Ad1 + gpSize;
      Ad4 := Ad2 + gpSize;
      rt1 := x[Ad1] + x[Ad2]; { a + b }
      rt2 := x[Ad1] - x[Ad2]; { a - b }
      rt3 := x[Ad3] + x[Ad4]; { c + d }
      rt4 := x[Ad3] - x[Ad4]; { c - d }
      x[Ad1] := rt1 + rt3;    { a + b + (c + d ) }
      x[Ad2] := rt2 + rt4;    { a - b + (c - d ) }
      x[Ad3] := rt1 - rt3;    { a + b - (c + d ) }
      x[Ad4] := rt2 - rt4;    { a - b - (c - d ) }
    end;

  if Nlog2 > 2 then
    begin { third + stages computed here }

      gpSize := 4;
      numBfs := 2;
      numGps := numGps div 2;
      for stage := 2 to Nlog2 - 1 do
        begin
          for gpNum := 0 to numGps - 1 do
            begin
              Ad0 := gpNum * gpSize * 2;
              Ad1 := Ad0; { 1st butterfly is different from others - no mults
                            needed }
              Ad2 := Ad1 + gpSize;
              Ad3 := Ad1 + gpSize div 2;
              Ad4 := Ad3 + gpSize;
              rt1 := x[Ad1];
              x[Ad1] := x[Ad1] + x[Ad2];
              x[Ad2] := rt1 - x[Ad2];
              rt1 := x[Ad3];
              x[Ad3] := x[Ad3] + x[Ad4];
              x[Ad4] := rt1 - x[Ad4];
              for bfNum := 1 to numBfs - 1 do
                begin { subsequent BF's dealt with together }
                  Ad1 := bfNum + Ad0;
                  Ad2 := Ad1 + gpSize;
                  Ad3 := gpSize - bfNum + Ad0;
                  Ad4 := Ad3 + gpSize;

                  CSAd := bfNum * numGps;
                  rt1 := x[Ad2] * C^[CSAd] + x[Ad4] * S^[CSAd];
                  rt2 := x[Ad4] * C^[CSAd] - x[Ad2] * S^[CSAd];

                  x[Ad2] := x[Ad1] - rt1;
                  x[Ad1] := x[Ad1] + rt1;
                  x[Ad4] := x[Ad3] + rt2;
                  x[Ad3] := x[Ad3] - rt2;

                end; { for bfNum := 0 to ... }
            end; { for gpNum := 0 to ... }
          gpSize := gpSize * 2;
          numBfs := numBfs * 2;
          numGps := numGps div 2;
        end;
    end; { if }
```

```
    if inverse then
      for i := 0 to maxNless1 do
        x[i] := x[i] / maxN;

    dispose(ptr(C));
    dispose(ptr(S));
  end;
```

APPENDIX F
MC68000 ASSEMBLY LANGUAGE SOURCE CODE

Included here are the Macintosh Programmer's Workshop assembly source files for the FHT routine, FHT.a, and the utility routines, FHTUtils.a. No macros or special assembler directives were used in writing to code to make it as portable as possible to other assemblers and 68000 systems.

FHT.a

```
                PRINT      OFF
                INCLUDE    'Traps.a'
                PRINT      ON

                TITLE      'FHT.a'
                STRING     ASIS

; This file contains a 68000 assembly language implementation of the
; Fast Hartley Transform algorithm which is covered under United States
; Patent Number 4,646,256.
;
; This code may therefore be freely used and distributed only under the
; following conditions:
;
;                1)  This header is included in every copy of the code; and
;
;                2)  The code is used for noncommercial research purposes only.
;
; Firms using this code for commercial purposes will be infringing a United
; States patent and should contact the
;
;                Office of Technology Licensing
;                Stanford University
;                857 Serra Street, 2nd Floor
;                Stanford, CA   94305-6225
;                (415) 723 0651
;
; Questions about the implementation of this routine can be directed to me at:
;
; Until 4/15/90:  Arlo Reeves
;                 Thayer School of Engineering
;                 Dartmouth College
;                 Hanover, NH   03755
;                 (603) 643 9076
;                 BITNET: arlo@mac.dartmouth.edu
;
; Permanently:    Box 345
;                 Mendocino, CA   95460
;                 (707) 937 5686
;

FHT             PROC       EXPORT

; function FHT(baseAddr:     ptr;
;              rowWords:     integer;
;              TwidTbl:      ptr): ShiftTotal; (integer)
;
; This routine performs a simple decimation in time Radix 2 Fast Hartley Transform.
; baseAddr is a pointer to an array of words which is rowWords long.  These
; integers must fall in the range [-8192..8191] because 2 bits of overflow can
; occur before any block floating point scaling is performed; the output is also
; limited to a 14 bit dynamic range.
; TwidTbl is a pointer to a twiddle factor lookup table with the following
; format:         Longword Offset   1st word              2nd word
;                 k:                (2^15)*cos(2□k/4096)  (2^15)*sin(2□k/4096)
; where k ranges from 0 to 1023.  The twiddle table is therefore 4K in size, so
; this routine can transform sequences and integer power of 2 in length from
; 4 to 4096.
; To process sequences up to 16K in length, a larger twiddle table must be created
; and the CSAdIncr (Cos/Sin address increment) variable must be increased
; from its current value of 2048 (doubled to 4096 for 8K sequence lengths).
; The routine, however, has not been tested for sequences longer than 4K.
; Bitreversal of the data must be done prior to calling this routine, e.g. with
; BitRevQ or BitRevEZ in one dimension, or with BitRevRows in two dimensions.
```

```
; The function returns the total number of bits shifted during the calculation.
; The correct scaling can be obtained by multiplying the transform by 2^ShiftTotal
; (though this may result in numbers bigger than 16 bits and does not add to the
; result's information content).
; Because of the Hartley Transform's symmetry, this routine can be used for
; both forward and inverse transformations.
; No checking is done for incorrect input; this is relegated to the calling routine.
; This routine will run on any 680x0 processor.
;
; The register map for this routine is as follows:
;                D0   csAdOffset            A0   x^
;                D1   numGps | gpNum        A1   cs^
;                D2   gpSize | gpIndex      A2   scratch
;                D3   scratch               A3   Ad1 | Ad2
;                D4   scratch               A4   Ad3 | Ad4
;                D5   scratch               A5   N   | stage
;                D6   scratch               A6   FramePtr
;                D7   scratch               A7   StackPtr
;
; The other values used are
                                                ; A6 Offsets
ShiftTot        EQU     18                      ; total bits shifted
baseAddr        EQU     14                      ; Ptr to data  AND
N               EQU     12                      ; length of xform
TwidTbl         EQU     8                       ; ptr to table of twiddle factors
OldShift        EQU     -2                      ; numBits to shift on this load this stage
NewShift        EQU     -4                      ; numBits to shift on this load next stage
Ad2             EQU     -6                      ;
Ad1             EQU     -8                      ;
Ad1Ad2          EQU     -8                      ; starting stage address of 1st & 2nd data element
Ad4             EQU     -10                     ;
Ad3             EQU     -12                     ;
Ad3Ad4          EQU     -12                     ; starting stage address of 3rd & 4th data element
gpAdIncR        EQU     -14                     ;
gpAdIncF        EQU     -16                     ;
gpAdIncFR       EQU     -16                     ; group Address Increment Forward & Retrograde indexing
CSAdInc         EQU     -18                     ; COS/SIN Address Increment

UsedRegs        REG     A2-A5/D2-D7

start           LINK    A6, #-18                ; Lotsa Locals
                MOVEM.L UsedRegs, -(SP)         ; Save registers
                MOVE.L  baseAddr(A6), A4        ; A4: baseAddr
                MOVE.W  N(A6), D0               ; move N into lo(D0)
                MOVE.W  D0, D1                  ;
                SWAP    D0                      ;
                MOVE.W  D1, D0                  ;
                MOVEA.L D0, A5                  ; A5: N | N
                MOVE.W  #0, ShiftTot(A6)        ; ShiftTotal = 0 initially

; First 2 butterflys done in one loop
Stage12         MOVE.W  A5, D0                  ; D0: N
                ASR.W   #2, D0                  ; D0: N div 4
                SUBQ.W  #1, D0                  ; D0: N div 4 - 1
                MOVEQ.L #0, D7                  ; D7: NewShift | maxVal

S12Loop         MOVE.W  D0, D1                  ; calculate address offsets of data
                ASL.W   #3, D1                  ; #3 -> 2wice N for even addr.
                MOVE.L  D1, D2                  ;
                ADDQ.W  #4, D2                  ; D1: ?? | Ad1,2
                SWAP    D1                      ;
                MOVE.W  D2, D1                  ; D1: Ad1,2 | Ad3,4
                MOVE.L  0(A4,D1.W), D5          ; D5: c | d
                MOVE.L  D5,D6                   ;
                SWAP    D6                      ; D6: d | c
                SWAP    D1                      ;
                MOVE.L  0(A4,D1.W), D3          ; D3: a | b
                MOVE.L  D3,D4                   ;
                SWAP    D4                      ; D4: b | a

                                                ;   D4       D3       D2
                MOVE.W  D3, D2                  ;    b   a    a   b    a   b
                ADD.W   D4, D3                  ;    b   a    a   a+b  a   b
                SUB.W   D2, D4                  ;    b   a-b  a   a+b  a   b

                                                ;   D6       D5       D2
                MOVE.W  D5, D2                  ;    d   c    c   d    c   d
                ADD.W   D6, D5                  ;    d   c    c   c+d  c   d
                SUB.W   D2, D6                  ;    d   c-d  c   c+d  c   d

                MOVE.W  D3, D2                  ;    c        a+b
                ADD.W   D5, D2                  ;    c        a+b+c+d
                SWAP    D2                      ;    a+b+c+d  c
                MOVE.W  D4, D2                  ;    a+b+c+d  a-b
                ADD.W   D6, D2                  ;    a+b+c+d  a-b+c-d
                MOVE.L  D2, 0(A4,D1.W)          ; copy x[Ad1,2] back to memory
                SWAP    D1

                MOVE.W  #0, D7                  ; maxVal := 0
```

```
                BTST.L     #17, D7             ; already 2 bits OF detected?
                BNE.S      GoOn1               ; No -> check for OF

                TST.W      D2                  ;
                BGE.S      pos1                ;
                NEG.W      D2                  ;
pos1            OR.W       D2, D7              ;
                SWAP       D2                  ;
                TST.W      D2                  ;
                BGE.S      pos2                ;
                NEG.W      D2                  ;
pos2            OR.W       D2, D7              ; set up maxVal

GoOn1           MOVE.W     D3, D2              ;   a+b+c+d  a+b
                SUB.W      D5, D2              ;   a+b+c+d  a+b-c-d
                SWAP       D2                  ;   a+b-c-d  a+b+c+d
                MOVE.W     D4, D2              ;   a+b-c-d  a-b
                SUB.W      D6, D2              ;   a+b-c-d  a-b-c+d
                MOVE.L     D2, 0(A4,D1.W)      ; copy x[Ad3,4] back to memory

                BTST.L     #17, D7             ; Already 2 bits of OF?
                BNE.S      GoOn3               ; No -> check for OF

                TST.W      D2                  ;
                BGE.S      pos3                ;
                NEG.W      D2                  ;
pos3            OR.W       D2, D7              ;
                SWAP       D2                  ;
                TST.W      D2                  ;
                BGE.S      pos4                ;
                NEG.W      D2                  ;
pos4            OR.W       D2, D7              ; set up maxVal
                CMP.W      #$1FFF, D7          ; now compare with mag limit
                BLE.S      GoOn2               ; less than -> no OF
                BSET.L     #16, D7             ; otherwise, shift next stage by 1 bit when loading
GoOn2           CMP.W      #$3FFF, D7          ;
                BLE.S      GoOn3               ;
                BSET.L     #17, D7             ; 2 bits of OF -> set Bit 17

GoOn3           DBRA.W     D0, S12Loop         ; end loop of first two stages

                SWAP       D7                  ;
                BTST.L     #1, D7              ; 2 bits of OF?
                BEQ.S      LEOneBitOF          ;
                MOVE.W     #2, D7              ;
                BRA.S      Cont1               ;
LEOneBitOF      BTST.L     #0, D7              ;
                BEQ.S      NoOF                ;
                MOVE.W     #1, D7              ;
                BRA.S      Cont1               ;
NoOF            MOVE.W     #0, D7              ;
Cont1           ADD.W      D7, ShiftTot(A6)    ;
                MOVE.W     D7, OldShift(A6)    ; for next stage

                MOVE.W     A5, D7              ; D7 : N
                CMPI.W     #8,D7               ; If transform length only 4 then end
                BLT        exit                ;

                MOVE.L     TwidTbl(A6), A1     ; A1: cs^
                MOVE.L     A4, A0              ; A0: x^

; set up the loop variables stage, gpNum and bfNum as well as gpSize numGps and numBfs

                MOVE.L     A5, D0              ;
                MOVE.W     #15, D1             ;
logLoop         BTST.L     D1, D0              ;
                DBNE.W     D1, logLoop         ;
                MOVE.W     D1, D0              ;
                MOVE.W     D0, D3              ; copy
                SUBQ.W     #3, D0              ;
                MOVE.L     D0, A5              ; A5: N | stage ;; stage := Nlog2 - 3
                MOVE.L     A5, D1              ;
                SWAP       D1
                ASR.W      #3, D1              ; numGps := maxN div 8;
                SWAP       D1                  ; D1: numGps | gpNum
                MOVE.L     #$00020000, D2      ; D2: gpSize | gpIndex
                MOVE.L     #$00000008, Ad1Ad2(A6)       ; Set up Ad1 and Ad2
                MOVE.L     #$0004000C, Ad3Ad4(A6)       ; Set up Ad3 and Ad4
                MOVE.L     #$000C0010,gpAdIncFR(A6); Set up Group Address Increments forward & retro
                MOVE.W     #2048, CSAdInc(A6)           ; Set up Cos/Sin Address Increment
                ; this address increment tailored to a TWID resource
                ; allowing 4K transform length - a TWID resource for 8K length needs this CSAdInc to be 4K, etc.

Stage           MOVE.W     #0, NewShift(A6)    ; no OF seen yet

                MOVE.L     D1, D3              ;
                SWAP       D3                  ;
                SUBQ.W     #1, D3              ;
                MOVE.W     D3, D1              ; gpNum := numGps - 1 downto 0
```

```
                MOVE.L     Ad1Ad2(A6), A3      ; A3: Ad1 | Ad2
                MOVE.L     Ad3Ad4(A6), A4      ; A4: Ad3 | Ad4

; fetch addresses Ad1 - Ad4 & calculate 1st butterfly

Group           MOVEQ.L    #0, D7              ;
                MOVE.W     OldShift(A6), D7    ; D7: maxVal | OldShift
                MOVE.L     A3, D3              ; D3: Ad1 | Ad2
                MOVE.W     0(A0, D3.W), D4     ;
                ASR.W      D7, D4              ; shift if OF dictates
                SWAP       D4                  ;
                SWAP       D3                  ;
                MOVE.W     0(A0, D3.W), D4     ; D4: x[Ad2] | x[Ad1]
                ASR.W      D7, D4              ; shift if OF dictates
                MOVE.L     D4, D5              ;
                SWAP       D5                  ;
                MOVE.L     D5, D6              ;
                ADD.W      D4, D5              ;
                SUB.W      D6, D4              ;
                MOVE.W     D5, 0(A0, D3.W)     ; x[Ad1] := x1 + x2
                SWAP       D3                  ;
                MOVE.W     D4, 0(A0, D3.W)     ; x[Ad2] := x1 - x2

                TST.W      NewShift(A6)        ; Over Flow already?
                BNE.S      GoOn4               ; yes -> no need to check

                SWAP       D7                  ; now accumulate maxVal
                TST.W      D4                  ;
                BGE.S      pos5                ;
                NEG.W      D4                  ;
pos5            OR.W       D4, D7              ;
                TST.W      D5                  ;
                BGE.S      pos6                ;
                NEG.W      D5                  ;
pos6            OR.W       D5, D7              ;
                SWAP       D7                  ;

GoOn4           MOVE.L     A4, D3              ;
                MOVE.W     0(A0, D3.W), D4     ;
                ASR.W      D7, D4              ; shift if OF dictates
                SWAP       D4                  ;
                SWAP       D3                  ;
                MOVE.W     0(A0, D3.W), D4     ; D4: x[Ad4] | x[Ad3]
                ASR.W      D7, D4              ; shift if OF dictates
                MOVE.L     D4, D5              ;
                SWAP       D5                  ;
                MOVE.L     D5, D6              ;
                ADD.W      D4, D5              ;
                SUB.W      D6, D4              ;
                MOVE.W     D5, 0(A0, D3.W)     ; x[Ad3] := x3 + x4
                SWAP       D3                  ;
                MOVE.W     D4, 0(A0, D3.W)     ; x[Ad4] := x3 - x4

                TST.W      NewShift(A6)        ; Over Flow already?
                BNE.S      GoOn5               ; yes -> no need to check

                SWAP       D7                  ; now accumulate maxVal
                TST.W      D4                  ;
                BGE.S      pos7                ;
                NEG.W      D4                  ;
pos7            OR.W       D4, D7              ;
                TST.W      D5                  ;
                BGE.S      pos8                ;
                NEG.W      D5                  ;
pos8            OR.W       D5, D7              ;
                CMP.W      #$1FFF, D7          ; maxVal > #$1FFF => OverFlow
                BLT.S      GoOn5               ; no OF -> go on
                MOVE       #1, NewShift(A6)    ;
                SWAP       D7                  ; D7: maxVal | oldShift

GoOn5           MOVE.L     D2, D3              ;
                SWAP       D3                  ;
                SUBQ.W     #2, D3              ;
                MOVE.W     D3, D2              ; gpIndex := gpSize - 2 downto 0

                MOVE.L     A3, D3              ; update Ad1 & Ad2
                ADDQ.W     #2, D3              ; Ad2 := Ad2 + 2
                SWAP       D3                  ;
                ADDQ.W     #2, D3              ; Ad1 := Ad1 + 2
                SWAP       D3                  ;
                MOVEA.L    D3, A3              ;

                MOVE.L     A4, D3              ; update Ad3 & Ad4
                MOVE.L     D2, D4              ;
                SWAP       D4                  ; D4: gpIndex| gpSize
                ASL.W      #1,D4               ;
                SWAP       D3                  ;
                ADD.W      D4, D3              ;
```

```
                SUBQ.W     #2, D3             ;
                SWAP       D3                 ; Ad4 := Ad4 + gpSize - 2
                ADD.W      D4, D3             ;
                SUBQ.W     #2, D3             ; Ad3 := Ad3 + gpSize - 2
                MOVEA.L    D3, A4             ;
                MOVE.W     CSAdInc(A6), D0    ; D0: CSAdOffset

; fetch addresses Ad1 - Ad4 & put CS[CSAd] into D7
bfLoop          MOVE.W     OldShift(A6), D7   ;
                MOVE.L     A3, D3             ; D3: Ad1 | Ad2
                MOVE.W     0(A0,D3.W), D5     ; D5: * | x[Ad2]
                ASR.W      D7, D5             ; shift if OF dictates
                SWAP       D3                 ;
                MOVE.W     0(A0,D3.W), D6     ; D6: * | x[Ad1]
                ASR.W      D7, D6             ; shift if OF dictates
                SWAP       D6                 ; D6: x[Ad1] | *

                MOVE.L     A4, D3             ; D3: Ad3 | Ad4
                MOVE.W     0(A0,D3.W), D4     ; D4: * | x[Ad4]
                ASR.W      D7, D4             ; shift if OF dictates
                SWAP       D3                 ;
                MOVE.W     0(A0,D3.W), D6     ; D6: x[Ad1] | x[Ad3]
                ASR.W      D7, D6             ; shift if OF dictates
                MOVEA.L    D6, A2             ; save

                MOVE.L     0(A1,D0.W), D7     ; D7: cos | sin
                MOVE.L     D7, D6             ; performing long subtraction avoids need
                CLR.W      D6                 ; to check for case of sin/cos = $8000 (N>=2048)
                SWAP       D6                 ; D6: 0 | cos
                MOVEQ.L    #0, D3             ;
                MOVE.W     D7, D3             ; D3: 0 | sin
                SUB.L      D3, D6             ; D6: * | cos - sin (difference always < $8000)
                MULS.W     D5, D6             ; D6: X2(c - s)

                MOVE.W     D4, D3             ; D3: X4
                ADD.W      D5, D3             ; D3: X4 + X2
                SUB.W      D5, D4             ; D4: X4 - X2

                CMPI.W     #$8000, D7         ; is sin factor = 2^15?
                BEQ.S      noMult1            ; long sequences (look at beg/end of TWID table)
                MULS.W     D7, D3             ; D3: s(X4 + X2)
                BRA.S      GoOn6              ;

noMult1         SWAP       D3                 ;
                CLR.W      D3                 ; Equivalent to multiplying by 2^16
                ASR.L      #1, D3             ; D3: s(X4 + X2)

GoOn6           SWAP       D7                 ;
                CMPI.W     #$8000, D7         ;
                BEQ.S      noMult2            ;
                MULS.W     D7, D4             ; D4: c(X4 - X2)
                BRA.S      GoOn7              ;

noMult2         SWAP       D4                 ;
                CLR.W      D4                 ; Equivalent to multiplying by 2^16
                ASR.L      #1, D4             ; D4: c(X4 - X2)

GoOn7           ADD.L      D6, D3             ; D3: cX2 + sX4
                ADD.L      D6, D4             ; D4: cX4 - sX2
                MOVE.W     #15, D7            ;
                ASR.L      D7, D3             ; Because mult'd by cos*2^15
                BCC.S      RoundDn1           ;
                ADDQ.W     #1, D3             ;
RoundDn1        ASR.L      D7, D4             ; Because mult'd by cos*2^15
                BCC.S      RoundDn2           ;
                ADDQ.W     #1, D4             ;

RoundDn2        MOVE.L     A2, D7             ; D7:  x[Ad1] | x[Ad3]
                MOVE.W     D7, D5             ;
                SUB.W      D4, D5             ; D5: *  | x3
                ADD.W      D4, D7             ; D7: *  | x4
                MOVE.L     D7, D6             ;
                SWAP       D6                 ;
                MOVE.W     D6, D4             ;
                SUB.W      D3, D4             ; D4: *  | x2
                ADD.W      D3, D6             ; D6: *  | x1

                MOVE.L     A3, D3
                MOVE.W     D4, 0(A0,D3.W)     ; store x2
                ADDQ.W     #2, D3             ; update Ad2
                SWAP       D3
                MOVE.W     D6, 0(A0,D3.W)     ; store x1
                ADDQ.W     #2, D3             ; update Ad1
                SWAP       D3                 ;
                MOVEA.L    D3, A3             ; store updated Ad1 | Ad2

                MOVE.L     A4, D3
                MOVE.W     D7, 0(A0,D3.W)     ; store x4
                SUBQ.W     #2, D3             ; update Ad4
```

```
                SWAP        D3                  ;
                MOVE.W      D5, 0(A0,D3.W)      ; store x3
                SUBQ.W      #2, D3              ; update Ad3
                SWAP        D3                  ;
                MOVEA.L     D3, A4              ; store updated Ad3 | Ad4

                TST.W       NewShift(A6)        ;
                BNE.S       GoOn8               ;

                MOVE.W      #0, D3              ; D4: maxVal
                TST.W       D4                  ;
                BGE.S       pos9                ;
                NEG.W       D4                  ;
pos9            OR.W        D4, D3              ;
                TST.W       D5                  ;
                BGE.S       pos10               ;
                NEG.W       D5                  ;
pos10           OR.W        D5, D3              ;
                TST.W       D6                  ;
                BGE.S       pos11               ;
                NEG.W       D6                  ;
pos11           OR.W        D6, D3              ;
                TST.W       D7                  ;
                BGE.S       pos12               ;
                NEG.W       D7                  ;
pos12           OR.W        D7, D3              ;
                CMP.W       #$1FFF, D3          ;
                BLT.S       GoOn8               ;
                MOVE.W      #1, NewShift(A6)    ; for next stage

GoOn8           ADD.W       CSAdInc(A6), D0     ; Update Cos/Sin Address

                DBRA        D2, bfLoop          ; gpIndex := gpIndex - 1, Loop if >= 0

                MOVE.L      gpAdIncFR(A6), D4   ; D4: gpAdIncF | gpAdIncR
                MOVE.L      A4, D3              ; D3: Ad3 | Ad4
                ADD.W       D4, D3              ; gpAddress base := gpAddress base + adIncrement
                SWAP        D3                  ;
                ADD.W       D4, D3              ;
                SWAP        D3                  ;
                MOVEA.L     D3, A4              ; update Ad3 & Ad4

                SWAP        D4
                MOVE.L      A3, D3              ;
                ADD.W       D4, D3              ; gpAddress base := gpAddress base + adIncrement
                SWAP        D3                  ;
                ADD.W       D4, D3              ;
                SWAP        D3                  ;
                MOVEA.L     D3, A3              ; update Ad1 & Ad2

                DBRA        D1, Group

                MOVE.L      D1, D3              ;
                SWAP        D3                  ;
                ASR.W       #1, D3              ; numGps := numGps div 2
                SWAP        D3                  ;
                MOVE.L      D3, D1              ;

                MOVE.L      D2, D3              ;
                SWAP        D3                  ;
                ASL.W       #1, D3              ; gpSize := gpSize * 2
                SWAP        D3                  ;
                MOVE.L      D3, D2              ;

                ASL.W       Ad1(A6)             ; update group Address bases
                ASL.W       Ad2(A6)             ;
                ASL.W       Ad3(A6)             ;
                ASL.W       Ad4(A6)             ;

                ASL.W       gpAdIncF(A6)        ; update group address increments
                ASL.W       gpAdIncR(A6)        ;

                ASR.W       CSAdInc(A6)         ; update Cos/Sin Address Increment

                MOVE.L      A5, D3              ;
                MOVE.L      D3, D4              ;
                SUBQ.W      #1, D3              ;
                MOVEA.L     D3, A5              ; stage := stage - 1

                MOVE.W      NewShift(A6), D7    ;
                ADD.W       D7, ShiftTot(A6)    ; Update ShiftTot
                MOVE.W      NewShift(A6), OldShift(A6)   ; Update OldShift

                DBRA.W      D4, Stage           ;

                TST.W       OldShift(A6)        ; was there to be a shift @ end?
                BEQ.S       exit                ; no -> exit
                MOVE.L      A5, D0              ;
                SWAP        D0                  ;
```

```
                SUBQ.W      #1, D0              ; D0: counter
                MOVE.W      OldShift(A6), D1    ; D1: NumBits to shift

                CMPI.W      #1, D1              ; if only one bit to shift, can do fast!
                BEQ.S       OneBitLoop          ;

TwoBitLoop      MOVE.W      (A0), D2            ; Two bits to shift -> slower
                ASR.W       D1, D2              ;
                MOVE.W      D2, (A0)+           ;
                DBRA.W      D0, TwoBitLoop      ;
                BRA.S       exit                ; then exit

OneBitLoop      ASR.W       (A0)+               ; only way to use this instr directly on mem
                DBRA.W      D0, OneBitLoop      ;

exit            MOVEM.L     (SP)+,UsedRegs      ; restore reg's
                UNLK        A6                  ; restore old A6
                MOVEA.L     (SP)+, A0           ; pop return address
                ADD.L       #10, SP             ; discard parameters
                JMP         (A0)                ; return to caller

                DC.B        'FHT     '          ; MacsBug Name

                ENDPROC


                END
```

FHTUtils.a

```
                PRINT      OFF
                INCLUDE    'Traps.a'
                PRINT      ON

                TITLE      'FHTutils.a'
                STRING     ASIS                ; to make DC.B work w/ normal 'strings'
                MC68881                        ; Needed for routines using FPU

; Included in this file are several routines needed in the computation of 1 and 2
; dimensional FFT's, such as bit-reversal and power spectrum calculation.  They
; may be freely duplicated and used (at your own risk).
;
; Questions about the implementation of these routines can be directed to me at:
;
; Until 4/15/90:  Arlo Reeves
;                 Thayer School of Engineering
;                 Dartmouth College
;                 Hanover, NH   03755
;                 (603) 643 9076
;                 BITNET: arlo@mac.dartmouth.edu
;
; Permanently:    Box 345
;                 Mendocino, CA   95460
;                 (707) 937 5686
;

BitRevEZ        PROC       EXPORT

; function BitRevEZ(x: ptr;
;                   length: integer):ErrCode;
;
; function BitRevEasy bitreverses the integer array of 'length' elements pointed
; to by x.  It does a getresource call to get a bitreversal lookup table of type
; 'BREV' of the appropriate size.  If BitRevEZ finds the resource, ErrCode is returned
; as 0, otherwise, it is returned as 1 (Nil resource handle).
; NOTE: Length must be a power of 2.

ErrCode         EQU        14                  ; ErrCode result A6 offset
x               EQU        10                  ;
length          EQU        8                   ;
NilHErr         EQU        1                   ;

UsedRegs        REG        A2/A4/D2-D4         ;

                LINK       A6, #0              ; No Locals
                MOVEM.L    UsedRegs, -(SP)     ; Save registers
                MOVE.W     #0, ErrCode(A6)     ; Default ErrCode is 0
                MOVE.L     x(A6), A1           ; A1: x
                MOVE.W     length(A6), D0      ; D0: length
                MOVE.W     D0, D1              ; Duplicate D0
                ADD.W      #124, D1            ; Add 124 to D1 to get Rsrc ID

                SUBQ.L     #4,SP               ; Reserve space for Rsrc handle
                MOVE.L     #'BREV', -(SP)      ; Pass RSRC Type and
                MOVE.W     D1, -(SP)           ; RSRC ID
                _GetResource                   ; Get the resource
                MOVE.L     (SP)+, D0           ; Put Resource Handle in D0 to set CCR
                BNE.S      continue            ; just in case handle is nil
                MOVE.W     NilHErr, ErrCode(A6)    ; set ErrCode = 1 => NilRHandle
                BRA.S      exit                ; and exit

continue        MOVE.L     D0, A2              ; use the handle
                MOVEA.L    A2, A4              ; Copy the handle into A4
                MOVE.L     (A4), A2            ; Dereference the Rsrc Handle in A2
                MOVE.W     (A2)+, D1           ; Move numSwaps into D1
                SUBQ.W     #1, D1              ; numSwaps -1 needed for DBRA
loop            MOVE.L     (A2)+, D2           ; Displacement of first swap add
                MOVE.W     D2,D0               ; move loword disp into D0
                SWAP       D2                  ; hiword disp in D2
                MOVE.W     0(A1,D0.W), D3      ; 1st integer in D3
                MOVE.W     0(A1,D2.W), D4      ; 2nd integer in D4
                MOVE.W     D3, 0(A1,D2.W)      ; Store back in reversed order
                MOVE.W     D4, 0(A1,D0.W)      ; "
                DBRA       D1, loop            ; until numSwaps (left) = 0

                MOVE.L     A4, -(SP)           ; push resource handle
                _ReleaseResource               ; release Resource

exit            MOVEM.L    (SP)+,UsedRegs      ; restore reg's
                UNLK       A6                  ; restore old A6
                MOVEA.L    (SP)+, A0           ; pop return address
                ADDQ.L     #6, SP              ; discard parameters
                JMP        (A0)                ; return to caller

                DC.B       'BITREVEZ'          ; MacsBug Name
```

```
                ENDPROC                         ; BitRevEZ


BitRevQ         PROC        EXPORT

; procedure BitRevQ(x: ptr;
;                   length: integer;
;                   BitRevTbl: ptr);
;
; function BitRevQuick bitreverses the integer array of 'length' elements pointed
; to by x.  It relies on the caller to provide a pointer to the appropriate
; resource of type 'BREV'.  If this pointer is nil, BitRevQ returns w/o having
; done anything to the input data.  By eliminating the getresource call, BitRevQ
; is quicker than BitRevEZ.
; NOTE: Length must be a power of 2.

data            EQU         14                  ; A6 offsets
length          EQU         12                  ;
BitRevTbl       EQU         8                   ;

UsedRegs        REG         A2/A4/D2-D4         ;

                LINK        A6, #0              ; No Locals
                MOVEM.L     UsedRegs, -(SP)     ; Save registers
                MOVE.L      BitRevTbl(A6), D0   ; to set condition codes
                BEQ.S       exit                ; if nil then quit
                MOVEA.L     D0, A2              ; A2: BitRevTbl
                MOVE.L      data(A6), A1        ; A1: data
                MOVE.W      length(A6), D0      ; D0: length

continue        MOVE.W      (A2)+, D1           ; Move numSwaps into D1
                SUBQ.W      #1, D1              ; numSwaps - 1 needed for DBRA
loop            MOVE.L      (A2)+, D2           ; Displacement of first swap add
                MOVE.W      D2,D0               ; move loword disp into D0
                SWAP        D2                  ; hiword disp in D2
                MOVE.W      0(A1,D0.W), D3      ; 1st integer in D3
                MOVE.W      0(A1,D2.W), D4      ; 2nd integer in D4
                MOVE.W      D3, 0(A1,D2.W)      ; Store back in reversed order
                MOVE.W      D4, 0(A1,D0.W)      ; "
                DBRA        D1, loop            ; until numSwaps (left) = 0

exit            MOVEM.L     (SP)+,UsedRegs      ; restore reg's
                UNLK        A6                  ; restore old A6
                MOVEA.L     (SP)+, A0           ; pop return address
                ADDA.W      #10, SP             ; discard parameters
                JMP         (A0)                ; return to caller

                DC.B        'BITREVQ '          ; MacsBug Name

                ENDPROC                         ; BitRevQ


BitRevRows      PROC        EXPORT

; procedure BitRevRows(iMatPtr: ptr;
;                      rowWords: integer;
;                      BitRevTbl: ptr);
;
; BitRevRows is similar to BitRevQ, but it BitReverses EACH row of the
; rowWords x rowWords matrix pointed to by iMatPtr.  Like BitRevQ, it also relies on the
; calling routine to supply a pointer to the lookup table (resource BREV),
; instead of loading it itself (as in BitRev).
; A NIL BitRevTbl will cause this routine to return w/o having done anything.

iMatPtr         EQU         14                  ; A6 Offsets
rowWords        EQU         12                  ;
BitRevTbl       EQU         8                   ;

UsedRegs        REG         A2/D2-D6            ;

                LINK        A6, #0              ; No Locals
                MOVEM.L     UsedRegs, -(SP)     ; Save registers
                MOVE.L      BitRevTbl(A6), D0   ;
                BEQ.S       exit                ; if BitRevTbl is NIL then exit
                MOVEA.L     D0, A1              ; A1: BitRevTbl
                MOVEA.L     iMatPtr(A6), A0     ; A0: iMatPtr
                MOVE.W      rowWords(A6), D0    ; D0: rowWords

                MOVE.W      (A1)+, D1           ; D1: numSwaps
                MOVEA.L     A1, A2              ; A2: swap start
                SUBQ.W      #1, D1              ; D1: numSwaps cntr.
                MOVE.W      D1, D2              ; D2: numSwaps cntr store
                MOVE.W      D0, D3              ;
                EXT.L       D3                  ;
                ADD.L       D3, D3              ; D3: rowOffset
                SUBQ.W      #1, D0              ; D0: rowWords cntr

rowLoop         MOVE.L      (A1)+, D4           ; D4: swap indices
```

```
                MOVE.W     (A0, D4.W), D5      ;
                SWAP       D4                  ;
                MOVE.W     (A0, D4.W), D6      ;
                MOVE.W     D5, (A0, D4.W)      ;
                SWAP       D4                  ;
                MOVE.W     D6, (A0, D4.W)      ;
                DBRA.W     D1, rowLoop         ; for each Swap

                MOVE.W     D2, D1              ; restore numSwaps cntr
                ADDA.L     D3, A0              ; increment addr for next row
                MOVEA.L    A2, A1              ; restore Tbl Addr
                DBRA.W     D0, rowLoop         ; for each row

exit            MOVEM.L    (SP)+,UsedRegs      ; restore reg's
                UNLK       A6                  ; restore old A6
                MOVEA.L    (SP)+, A0           ; pop return address
                ADDA.W     #10, SP             ; discard parameters
                JMP        (A0)                ; return to caller

                DC.B       'BTRVROWS'          ; MacsBug Name

                ENDPROC                        ; BitRevRows


ShiftRowsR      PROC       EXPORT

; procedure ShiftRowsR(iMatPtr: ptr;
;                      rowWords: integer;
;                      maxShift: integer;
;                      RowShfArr: ptr);
;
; Each row of the rowWords x rowWords integer matrix pointed to by iMatPtr is right
; shifted by a number of bits specified by the difference maxShift - RowShfArr[r]
; where RowShfArr (= array[0..rowWords -1] of integer) contains the number of
; bits each row has shifted (during transformation).
; During the right shift operation ShiftRowsR rounds the numbers instead of
; throwing out the bits shifted out.  It does this in a very simple way by
; checking the carry bit (the value of the last bit shifted out).  If it is
; set, then the number is rounded up; if it is not, the number is left as is.
; Because each word must be manipulated in this routine, there is no fast loop
; for the common 1 bit shift case.
; maxShift is the maximum number of bits by which any row was shifted during the
; transform (in FHT).  RowShfArr contains the number of bits by which each row
; was shifted.  To make all rows shifted by maxShift, each row is shifted here
; by maxShift - RowShfArr[rowIndex] bits.

iMatPtr         EQU        16                  ; Ptr to data
rowWords        EQU        14                  ; length of row
maxShift        EQU        12                  ; maximum of array
RowShfArr       EQU        8                   ; Array w/number of bits to shift right

UsedRegs        REG        D2-D6/A2

start           LINK       A6, #0              ; No Locals
                MOVEM.L    UsedRegs, -(SP)     ; Save registers
                MOVE.L     iMatPtr(A6), A0     ; A0: iMatPtr
                MOVEA.L    A0, A2              ; A2: iMatPtr (row modified)
                MOVE.L     RowShfArr(A6), A1   ; A1: RowShfArr
                MOVE.W     rowWords(A6), D0    ; D0: rowWords

                MOVE.W     D0, D4              ;
                EXT.L      D4                  ;
                ADD.L      D4, D4              ; D4: rowOffset
                SUBQ.W     #1, D0              ; D0: rowWords cntr store
                MOVE.W     D0, D1              ; D1: row cntr
                MOVE.W     D1, D2              ; D2: col cntr
                MOVE.W     maxShift(A6), D3    ; D3: maxShift

rowLoop         MOVEA.L    A2, A0              ; A0: points to current row
                MOVE.W     (A1)+, D5           ; D5: numBits already shifted
                SUB.W      D3, D5              ;
                NEG.W      D5                  ; D5: maxShift - BitShfArr[r]
                BLE.S      ShfDone             ; if numBits <= 0 then No Shift necc.

GenShift        MOVE.W     (A0), D6            ; D4: data
                ASR.W      D5, D6              ; shift it
                BCC.S      NoRound             ;
                ADDQ.W     #1, D6              ;
NoRound         MOVE.W     D6, (A0)+           ; put it back
                DBRA.W     D2, GenShift        ; for each col

ShfDone         ADDA.L     D4, A2              ;
                MOVE.W     D0, D2              ; restore col cntr
                DBRA.W     D1, rowLoop         ; for each row

exit            MOVEM.L    (SP)+,UsedRegs      ; restore reg's
                UNLK       A6                  ; restore old A6
                MOVEA.L    (SP)+, A0           ; pop return address
                ADDA.W     #12, SP             ; discard parameters
```

```
                JMP         (A0)                ; return to caller

                DC.B        'SFTROWSR'          ; MacsBug Name

                ENDPROC                         ; ShiftRowsR

MeanZero        PROC        EXPORT

; procedure MeanZero(srcPtr: ptr;
;                    bufSize: longint);
;
; Procedure MeanZero traverses the bufSize integer array pointed to by srcPtr
; and calculates the average value of the matrix elements.  It the traverses the
; array again, subtracting the average from each element.  This removes the DC
; component of the 2D transform of this matrix.
;
srcPtr          EQU         12                  ; A6 offsets
bufSize         EQU         8                   ;

UsedRegs        REG         D2-D6

                LINK        A6, #0              ; no locals
                MOVEM.L     UsedRegs, -(SP)     ;
                MOVE.L      srcPtr(A6), A0      ; A0: srcPtr
                MOVE.L      bufSize(A6), D0     ;
                MOVE.L      D0, D1              ; D1: counter
                MOVEQ.L     #0, D2              ; D2: lo longword of accumulator
                MOVEQ.L     #0, D3              ; D3: hi longword of accumulator
                MOVEQ.L     #0, D4              ; D4: zero needed for addx
                MOVEQ.L     #0, D5              ; D5: Hi word must be zero

sumLoop         MOVE.W      (A0)+, D5           ;
                ADD.L       D5, D2              ;
                ADDX.L      D4, D3              ; if carry bit, carry on
                SUBQ.L      #1, D1              ;
                BNE.S       sumLoop             ;

; have accumulated 64 bit sum in D3:D2; now simulate divide by subtracting DO repeatedly

                MOVEQ.L     #0, D1              ; D1: zero needed for subx
divLoop         SUB.L       D0, D2              ;
                SUBX.L      D1, D3              ;
                BLT.S       divDone             ;
                ADDQ.L      #1, D4              ; D4: quotient
                BRA.S       divLoop             ;

divDone         MOVE.L      srcPtr(A6), A0      ;
subtLoop        SUB.W       D4, (A0)+           ;
                SUBQ.L      #1, D0              ;
                BNE.S       subtLoop            ;

exit            MOVEM.L     (SP)+,UsedRegs      ; restore reg's
                UNLK        A6                  ; restore old A6
                MOVEA.L     (SP)+, A0           ; pop return address
                ADDQ.L      #8, SP              ; discard parameters
                JMP         (A0)                ; return to caller

                DC.B        'MEANZERO'          ; MacsBug Name

                ENDPROC                         ; MeanZero


FHTtoFFT1D      PROC        EXPORT

; procedure HtoF1D(H: iArrPtr;
;                  F: icArrPtr;
;                  numPts: integer);
;
; procedure FHTtoFFT1D takes a numPts long Hartley transform pointed to by x and
; converts it into a complex Fourier transform pointed to F.  F is of type cArrPtr:
;  icPoint = record
;    re, im: integer;
;   end;
;  icArr = array[0..maxStoreLess1] of icPoint;
;  icArrPtr = ^icArr;
; Both pointers H & F must be valid (not nil) and numPts must be a power of 2.
; The memory pointed to by F needn't be initialized.

H               EQU         14                  ; A6 offsets
F               EQU         10                  ;
numPts          EQU         8                   ;

UsedRegs        REG         D2-D6

                LINK        A6, #0              ; no locals
                MOVEM.L     UsedRegs, -(SP)     ;
                MOVE.W      numPts(A6), D1      ; D1: numPts
                MOVEA.L     H(A6), A0           ; A0: H
                MOVEA.L     F(A6), A1           ; A1: F
```

```
; Do zero'th element of array first - special case

                MOVE.W     (A0), D0            ; D0: ??? | H(0)
                SWAP       D0                  ;
                CLR.W      D0                  ; D0: H(0) | 0 = F(0)
                MOVE.L     D0, (A1)            ; Store result

; Do N div 2 + 1'th element next - also a special case

                MOVE.W     0(A0, D1.W), D0     ; N div 2 + 1 th element - middle of spectrum
                SWAP       D0                  ;
                CLR.W      D0                  ; D0: H(N div 2 + 1) | 0 = F(N div 2 + 1)
                MOVE.W     D1, D2              ;
                ASL.W      #1, D2              ;
                MOVE.L     D0, 0(A1, D2.W)     ; Store result

; Now do rest of the points in a loop

                MOVE.W     D1, D0              ; D0: N
                ASR.W      #1, D1              ;
                SUB.W      #2, D1              ; D1: counter
                MOVE.W     #2, D2              ; D2: f      (integer index)
                MOVE.W     D0, D3              ;
                ASL.W      #1, D3              ;
                SUB.W      #2, D3              ; D3: N - f (integer index)

loop            MOVE.W     0(A0, D2.W), D4     ; D4: H(f)
                MOVE.W     0(A0, D3.W), D5     ; D5: H(N - f)
                MOVE.W     D5, D6              ; D6: H(N - f)
                ADD.W      D4, D5              ; D5:   H(f) + H(N - f)  = re
                SUB.W      D4, D6              ; D6: -(H(f) - H(N - f)) = im
                ASR.W      #1, D5              ;
                ASR.W      #1, D6              ;
                SWAP       D5                  ;
                MOVE.W     D6, D5              ; D5: re | im

                MOVE.W     D2, D6              ;
                ASL.W      #1, D6              ;
                MOVE.L     D5, 0(A1, D6.W)     ; store F(f)
                MOVE.W     D3, D6              ;
                ASL.W      #1, D6              ;
                NEG.W      D5                  ; CHS of im part
                MOVE.L     D5, 0(A1, D6.W)     ; store F(N - f)

                ADDQ.W     #2, D2              ;
                SUBQ.W     #2, D3              ;

                DBRA       D1, loop            ;

exit            MOVEM.L    (SP)+,UsedRegs      ; restore reg's
                UNLK       A6                  ; restore old A6
                MOVEA.L    (SP)+, A0           ; pop return address
                ADDA.W     #10, SP             ; discard parameters
                JMP        (A0)                ; return to caller

                DC.B       'HtoF1D  '          ; MacsBug Name

                ENDPROC                        ; HtoF1D


FFTtoFHT1D      PROC       EXPORT

; procedure FtoH1D(F: icArrPtr;
;                  H:        iArrPtr;
;                  numPts: integer);
;
; procedure FtoH1D converts an integer Fourier transform of length numPts into
; a Hartley transform.  icP points to the complex integer array holding the Fourier
; transform and ixP points to the integer array Hartley transform result.
; For a definition of icArrPtr type, see HtoF1D.
; Again, no checking is done for nil pointers and numPts must be a power of 2.

F               EQU        14                  ; A6 offsets
H               EQU        10                  ;
numPts          EQU        8                   ;

UsedRegs        REG        D2-D3

                LINK       A6, #0              ; no locals
                MOVEM.L    UsedRegs, -(SP)     ;
                MOVE.W     numPts(A6), D1      ; D1: numPts
                MOVEA.L    H(A6), A0           ; A0: H
                MOVEA.L    F(A6), A1           ; A1: F

                SUBQ.W     #1, D1              ; D1: counter
loop            MOVE.L     (A1)+, D2           ; D2: re | im
                MOVE.L     D2, D3              ;
                SWAP       D3                  ; D3: im | re
```

```
                SUB.W       D2, D3              ; D3: im | re - im
                MOVE.W      D3, (A0)+           ;
                DBRA        D1, loop            ;

exit            MOVEM.L     (SP)+,UsedRegs      ; restore reg's
                UNLK        A6                  ; restore old A6
                MOVEA.L     (SP)+, A0           ; pop return address
                ADDA.W      #10, SP             ; discard parameters
                JMP         (A0)                ; return to caller

                DC.B        'FtoH1D  '          ; MacsBug Name

                ENDPROC                         ; FtoH1D

psFHT1D         PROC        EXPORT

; procedure psFHT1D(h: iArrPtr;
;                   ps: iArrPtr;
;                   numPts: integer;
;                   scaleType: integer);
;
; procedure psFHT1D computes the power spectrum from the numPts long Hartley transform
; pointed to by h and puts the result in the integer array ps.  The multiplication
; inherent in the ps calculations gives 32 bit results from 16 bit words, so these values
; are mapped into a their 16 bit destinations.  The mapping chosen is based on the
; scaleType argument as follows:
; scaleType       scaling implemented:
;                 <=0         linear
;                 1-9         nth root            (1st root = linear)
;                 >=10        log
; In order to first determine the scale factor and offset of the linear mapping,
; h is first scanned to determine the range of output by actually calculating
; the power spectrum.  During the second scan, the computation again takes
; place and the scaled data is stored in the dest.  This redundancy in computation
; could only be eliminated by an increase in storage requirements.
;
; WARNING: the use of this routine requires the existance of an FPU!

h               EQU         16                  ; A6 offsets
ps              EQU         12                  ;
numPts          EQU         10                  ;
scaleType       EQU         8                   ;

UsedRegs        REG         D2-D7

                LINK        A6, #0              ;
                MOVEM.L     UsedRegs, -(SP)     ;
                FMOVE.X     FP2, -(SP)          ; save FP2
                MOVE.W      numPts(A6), D0      ; D0: numPts
                MOVEA.L     h(A6), A0           ; A0: h
                MOVEA.L     ps(A6), A1          ; A1: ps

; first find the maximum and minimum of the range

                MOVE.L      #$7FFFFFFF, D6      ; D6: min
                MOVE.L      #$80000000, D7      ; D7: max

; first special case: H(0)
                MOVE.W      (A0), D1            ; D1: H(0)
                MULS.W      D1, D1              ;
                ADD.L       D1, D1              ;
                CMP.L       D6, D1              ;
                BGE.S       notMin1             ;
                MOVE.L      D1, D6              ;
notMin1         CMP.L       D7, D1              ;
                BLE.S       notMax1             ;
                MOVE.L      D1, D7              ;

; second special case: H(N div 2 + 1)
notMax1         MOVE.W      0(A0, D0.W), D1     ; D1: H(N Div 2 + 1)
                MULS.W      D1, D1              ;
                ADD.L       D1, D1              ;
                CMP.L       D6, D1              ;
                BGE.S       notMin2             ;
                MOVE.L      D1, D6              ;
notMin2         CMP.L       D7, D1              ;
                BLE.S       notMax2             ;
                MOVE.L      D1, D7              ;

; now for the rest of the spectrum
notMax2         MOVE.W      D0, D1              ; D0: N
                ASR.W       #1, D1              ;
                SUB.W       #2, D1              ; D1: counter
                MOVE.W      #2, D2              ; D2: f      (integer index)
                MOVE.W      D0, D3              ;
                ASL.W       #1, D3              ;
                SUB.W       #2, D3              ; D3: N - f (integer index)

loop1           MOVE.W      0(A0, D2.W), D4     ;
```

```
                MOVE.W     0(A0, D3.W), D5     ;
                MULS.W     D4, D4              ;
                MULS.W     D5, D5
                ADD.W      D5, D4              ;
                CMP.L      D6, D4              ;
                BGE.S      notMin3             ;
                MOVE.L     D4, D6              ;
notMin3         CMP.L      D7, D4              ;
                BLE.S      notMax3             ;
                MOVE.L     D4, D7              ;

notMax3         ADDQ.W     #2, D2              ;
                SUBQ.W     #2, D3              ;
                DBRA       D1, loop1           ;

; Now the values of max and min are recorded in D7 and D6 resp.
;
; Find which scaling to use : D6:               min
;                             FP1:              mulFact
;                             FP2:              1/n (for nth root calculation)
; for any x the mapping is: scaleType(x - xmin) * mulFact
; where scaleType produces the scaling desired and mulFact maps that range
; onto 0..maxint.

                SUB.L      D6, D7              ; D7: max - min :: range
                BNE.S      findScale           ;
                MOVEQ.L    #1, D7              ; don't want a range of 0!

findScale       MOVE.W     scaleType(A6), D4   ;
                BLE.S      linMap1             ; scaleType is linear
                CMP.W      #10, D4             ;
                BGE.S      logMap1             ; scaleType is log

rootMap1        FMOVE.L    #1, FP2             ; scaleType is nth root
                FDIV.W     D4, FP2             ; FP1: 1/n
                FLOGN.L    D7, FP0             ; FP0: ln(range)
                FMUL.X     FP2, FP0            ; FP0: 1/n*ln(range)
                FETOX.X    FP0, FP0            ; FP0: (range)^(1/n)
                MOVE.W     #2, D5              ; D5 : scaleType flag
                BRA.S      GoOn1               ;

linMap1         FMOVE.L    D7, FP0             ;
                MOVE.W     #1, D5              ; D5: scaleType flag
                BRA.S      GoOn1               ;

logMap1         SUBQ.L     #1, D6              ; min := min - 1 ;; to prevent log(0)
                ADDQ.L     #1, D7              ; increases range by 1
                FLOG2.L    D7, FP0             ; FP0: range !!! 6888x
                MOVE.W     #3, D5              ; D5: scaleType flag

; Calculate the mulFact and put it in FP1

GoOn1           FMOVE.L    #$7FFF, FP1         ;
                FDIV.X     FP0, FP1            ; FP1: mulFact
                MOVE.W     D5, D7              ; D7: scaleType flag
;
; D7 contains a scaleType flag: 1 -> linear; 2 -> nth root; 3 -> log
;
; now calculate actual ps values
; special case 1: H(0)

                MOVE.W     (A0), D2            ;
                MULS.W     D2, D2              ;
                ADD.L      D2, D2              ;
                SUB.L      D6, D2              ;
                MOVE.W     D7, D5              ;
                SUBQ.W     #1, D5              ;
                BEQ.S      linMap2             ;
                SUBQ.W     #1, D5              ;
                BEQ.S      rootMap2            ;
logMap2         FLOG2.L    D2, FP0             ;
                FMUL.X     FP1, FP0            ;
                BRA.S      GoOn2               ;

linMap2         FMOVE.L    D2, FP0             ;
                FMUL.X     FP1, FP0            ;
                BRA.S      GoOn2               ;

rootMap2        FLOGN.L    D2, FP0             ;
                FMUL.X     FP2, FP0            ; FP0: 1/n * ln(x)
                FETOX.X    FP0, FP0            ;
                FMUL.X     FP1, FP0            ;

GoOn2           FMOVE.W    FP0, D2             ;
                MOVE.W     D2, (A1)            ;

; special case 2: H(N div 2 + 1)

                MOVE.W     0(A0, D0.W), D2     ;
```

```
                MULS.W     D2, D2              ;
                ADD.L      D2, D2              ;
                SUB.L      D6, D2              ;
                MOVE.W     D7, D5              ;
                SUBQ.W     #1, D5              ;
                BEQ.S      linMap3             ;
                SUBQ.W     #1, D5              ;
                BEQ.S      rootMap3            ;
logMap3         FLOG2.L    D2, FP0             ;
                FMUL.X     FP1, FP0            ;
                BRA.S      GoOn3               ;

linMap3         FMOVE.L    D2, FP0             ;
                FMUL.X     FP1, FP0            ;
                BRA.S      GoOn3               ;

rootMap3        FLOGN.L    D2, FP0             ;
                FMUL.X     FP2, FP0            ; FP0: 1/n * ln(x)
                FETOX.X    FP0, FP0            ;
                FMUL.X     FP1, FP0            ;

GoOn3           FMOVE.W    FP0, D2             ;
                MOVE.W     D2, 0(A1, D0.W)     ;

; now do it for the rest
                MOVE.W     D0, D1              ; D0: N
                ASR.W      #1, D1              ;
                SUB.W      #2, D1              ; D1: counter
                MOVE.W     #2, D2              ; D2: f      (integer index)
                MOVE.W     D0, D3              ;
                ASL.W      #1, D3              ;
                SUB.W      #2, D3              ; D3: N - f (integer index)

loop2           MOVE.W     0(A0, D2.W), D4     ;
                MOVE.W     0(A0, D3.W), D5     ;
                MULS.W     D4, D4              ;
                MULS.W     D5, D5              ;
                ADD.L      D4, D5              ;
                SUB.L      D6, D5              ;
                MOVE.W     D7, D4              ;
                SUBQ.W     #1, D4              ;
                BEQ.S      linMap4             ;
                SUBQ.W     #1, D4              ;
                BEQ.S      rootMap4            ;
logMap4         FLOG2.L    D5, FP0             ;
                FMUL.X     FP1, FP0            ;
                BRA.S      GoOn4               ;

linMap4         FMOVE.L    D5, FP0             ;
                FMUL.X     FP1, FP0            ;
                BRA.S      GoOn4               ;

rootMap4        FLOGN.L    D5, FP0             ;
                FMUL.X     FP2, FP0            ; FP0: 1/n * ln(x)
                FETOX.X    FP0, FP0            ;
                FMUL.X     FP1, FP0            ;

GoOn4           FMOVE.W    FP0, D5             ;
                MOVE.W     D5, 0(A1, D2.W)     ;
                MOVE.W     D5, 0(A1, D3.W)     ;

                ADDQ.W     #2, D2              ;
                SUBQ.W     #2, D3              ;
                DBRA       D1, loop2           ;

                FMOVE.X    (SP)+, FP2          ; restore FP2
exit            MOVEM.L    (SP)+,UsedRegs      ; restore reg's
                UNLK       A6                  ; restore old A6
                MOVEA.L    (SP)+, A0           ; pop return address
                ADDA.W     #12, SP             ; discard parameters
                JMP        (A0)                ; return to caller

                DC.B       'psFHT1D '          ; MacsBug Name

                ENDPROC                        ; psFHT1D


ByteAvgRect     PROC       EXPORT

; function ByteAvgRect(baseAddr: ptr;
;                      pixelsPerLine: integer;
;                      roiWidth: integer;
;                      roiHeight: integer): integer;
;
; function ByteAvgRect returns the average byte of a rectangular part of a PixMap.
; baseAddr is the address of the first byte in the rectangle, pixelsPerLine is the
; width of the PixMap (right out of the Image's PicInfo record), roiWidth is the
; width of the rectangle (osRoiRect's width) and roiHeight is its height (osRoiRect's
; height).
```

```
; NOTE: lots of odd addresses generated here.

result          EQU       18                ; A6 Offsets
baseAddr        EQU       14                ;
pixelsPL        EQU       12                ;
roiWidth        EQU       10                ;
roiHeight       EQU       8                 ;

UsedRegs        REG       D2-D5/A2

start           LINK      A6, #0            ;
                MOVEM.L   UsedRegs, -(SP)   ;
                MOVEA.L   baseAddr(A6), A0  ; A0: baseAddr
                MOVEA.L   A0, A1            ; A1: lineAddr stored
                MOVE.W    roiHeight(A6), D0 ; D0: numLines (row)
                MOVE.W    roiWidth(A6), D1  ; D0: rowBytes (col)
                MOVE.W    D1, D5            ;
                MULS.W    D0, D5            ; D5: numBytes
                SUBQ.W    #1, D0            ; D0: row counter
                SUBQ.W    #1, D1            ; D1: col counter
                MOVE.W    D1, D2            ; D2: column count (for restore)
                MOVE.W    pixelsPL(A6), A2  ; A2: pixelsPerLine (bytes offset)
                MOVEQ.L   #0, D3            ; temp byte store
                MOVEQ.L   #0, D4            ; sum of bytes

loop            MOVE.B    (A0)+, D3         ;
                ADD.L     D3, D4            ;
                DBRA.W    D1, loop          ;

                MOVE.W    D2, D1            ; restore column counter
                ADDA.L    A2, A1            ;
                MOVEA.L   A1, A0            ; update address
                DBRA.W    D0, loop          ;

                CLR.W     D3                ; D3: quotient
divLoop         SUB.L     D5, D4            ;
                BLT.S     done              ;
                ADDQ.W    #1, D3            ;
                BRA.S     divLoop           ;

done            MOVE.W    D3, result(A6)    ;

exit            MOVEM.L   (SP)+, UsedRegs   ;
                UNLK      A6                ;
                MOVEA.L   (SP)+, A0         ;
                ADDA.L    #10, SP           ;
                JMP       (A0)              ;

                DC.B      'BTAVGRCT'        ; Macsbug name

                ENDPROC


ByteRange       PROC      EXPORT

; function ByteRange(baseAddr: ptr;
;                    bufSize: longint):longint;
;
; function ByteRange scans the array of bytes starting at baseAddr that is bufSize
; bytes in length, recording max and min.  The max is returned in the high word,
; the min in the low word of the result.
;
;
result          EQU       16                ;
baseAddr        EQU       12                ;
bufSize         EQU       8                 ;

UsedRegs        REG       D2-D3/D6-D7       ;

start           LINK      A6, #0            ;
                MOVEM.L   UsedRegs, -(SP)   ;
                MOVE.L    baseAddr(A6), A0  ;
                MOVE.L    bufSize(A6), D0   ;
                ASR.L     #1, D0            ; fetching in bytes
                MOVE.W    #$00FF, D3        ; D3: lo byte mask
                MOVE.W    D3, D6            ; D6: min
                MOVE.W    #0, D7            ; D7: max

loop            MOVE.W    (A0)+, D1         ;
                MOVE.W    D1, D2            ;
                LSR.W     #8, D1            ; D1: hibyte
                AND.W     D3, D2            ; D2: lobyte

                CMP.W     D6, D2            ; check hibyte
                BGE.S     notMin1           ;
                MOVE.W    D2, D6            ;
notMin1         CMP.W     D7, D2            ;
                BLE.S     notMax1           ;
                MOVE.W    D2, D7            ;
```

```
notMax1         CMP.W     D6, D1            ; check lobyte
                BGE.S     notMin2           ;
                MOVE.W    D1, D6            ;
notMin2         CMP.W     D7, D1            ;
                BLE.S     notMax2           ;
                MOVE.W    D1, D7            ;

notMax2         SUBQ.L    #1, D0            ;
                BGT.S     loop              ;

                SWAP      D7                ;
                MOVE.W    D6, D7            ;
                MOVE.L    D7, result(A6)    ;

exit            MOVEM.L   (SP)+, UsedRegs   ;
                UNLK      A6                ;
                MOVE.L    (SP)+, A0         ;
                ADDQ.L    #8, SP            ;
                JMP       (A0)              ;

                DC.B      'BYTERNGE'        ; MacsBug Name

                ENDPROC


IntRange        PROC      EXPORT

; function IntRange(baseAddr: ptr;
;                   bufSize: longint): result;
;
; function IntRange scans an array of integers bufSize long starting at baseAddr
; and returns the max in the high word, the min in the low word of the result
;
result          EQU       16                ;
baseAddr        EQU       12                ;
bufSize         EQU       8                 ;

UsedRegs        REG       D2/D6-D7          ;

start           LINK      A6, #0            ;
                MOVEM.L   UsedRegs, -(SP)   ;
                MOVE.L    baseAddr(A6), A0  ;
                MOVE.L    bufSize(A6), D0   ;
                ASR.L     #1, D0            ;
                MOVE.W    #$7FFF, D6        ; D6: min
                MOVE.W    #$8000, D7        ; D7: max
loop            MOVE.L    (A0)+, D1         ;
                MOVE.L    D1, D2            ;
                SWAP      D2                ;

                CMP.W     D6, D2            ; check 1st #
                BGE.S     notMin1           ;
                MOVE.W    D2, D6            ;
notMin1         CMP.W     D7, D2            ;
                BLE.S     notMax1           ;
                MOVE.W    D2, D7            ;

notMax1         CMP.W     D6, D1            ; check 2nd #
                BGE.S     notMin2           ;
                MOVE.W    D1, D6            ;
notMin2         CMP.W     D7, D1            ;
                BLE.S     notMax2           ;
                MOVE.W    D1, D7            ;

notMax2         SUBQ.L    #1, D0            ;
                BGT.S     loop              ;

                SWAP      D7                ; D7: max | ?
                MOVE.W    D6, D7            ; D7: max | min
                MOVE.L    D7, result(A6)    ;

exit            MOVEM.L   (SP)+, UsedRegs   ;
                UNLK      A6                ;
                MOVE.L    (SP)+, A0         ;
                ADDQ.L    #8, SP            ;
                JMP       (A0)              ;

                DC.B      'INTRANGE'        ; MacsBug Name

                ENDPROC


AddSub2BufsF    PROC      EXPORT

; procedure AddSub2BufsF(srcPtr1:      ptr;
;                        srcPtr2:      ptr;
;                        destPtr:      ptr;
;                        srcScale1:    integer;
```

```
;                       rcScale2:      integer;
;                   var destScale:     integer;  (ptr)
;                       srcSize:       longint;
;                       operation:     integer);
;
; procedure AddSub2BufsF adds/subtracts the contents of the FHTBuf pointed
; to by srcPtr2 to/from the FHTBuf pointed to by srcPtr1 and puts the result
; in the FHTBuf pointed to by destPtr (all Bufs are srcSize words in length).
; If operation is >= 0 then the two arrays are added;
; If operation is <  0 then the two arrays are subtracted.
; srcScale is the power of 2 by which FHTBuf is multiplied to give a
; 'correctly' scaled FHTBuf.  Since the scalings for the two buffers may
; differ, one must be scaled according to the difference of srcScale1
; and srcScale2 before adding/subtracting.
; NOTE: FPU used here.

srcPtr1          EQU        30                ; A6 Offsets
srcPtr2          EQU        26                ;
destPtr          EQU        22                ;
srcScale1        EQU        20                ;
srcScale2        EQU        18                ;
destScale        EQU        14                ;
srcSize          EQU        10                ;
operation        EQU        8                 ;

UsedRegs         REG        A2-A3             ;
UsedFRegs        FREG       FP2-FP7           ;

start            LINK       A6, #0            ;
                 MOVEM.L    UsedRegs, -(SP)   ;
                 FMOVEM.X   UsedFRegs, -(SP)  ;

                 FMOVE.W    srcScale1(A6), FP6  ;
                 FTWOTOX.X  FP6                 ; FP6: scale1
                 FMOVE.W    srcScale2(A6), FP7  ;
                 FTWOTOX.X  FP7                 ; FP7: scale2

                 MOVEA.L    destScale(A6), A0   ; @destScale
                 MOVEA.L    srcPtr1(A6), A1     ;
                 MOVEA.L    srcPtr2(A6), A2     ;
                 MOVE.L     srcSize(A6), D0     ;
                 FMOVE.L #$7FFFFFFF, FP4        ; FP4: min
                 FMOVE.L    #$80000000, FP5     ; FP5: max

                 TST.W      operation(A6)       ;
                 BLT.S      SubLoop1            ;

AddLoop1         FMOVE.W    (A1)+, FP1          ;
                 FMOVE.W    (A2)+, FP2          ;
                 FMUL.X     FP6, FP1            ;
                 FMUL.X     FP7, FP2            ;
                 FADD.X     FP2, FP1            ;
                 FCMP.X     FP4, FP1            ;
                 FBGE.W     notMin1             ;
                 FMOVE.X    FP1, FP4            ;
notMin1          FCMP.X     FP5, FP1            ;
                 FBLE.W     notMax1             ;
                 FMOVE.X    FP1, FP5            ;
notMax1          SUBQ.L     #1, D0              ;
                 BGT.S      AddLoop1            ;
                 BRA.S      GoOn1               ;

SubLoop1         FMOVE.W    (A1)+, FP1          ;
                 FMOVE.W    (A2)+, FP2          ;
                 FMUL.X     FP6, FP1            ;
                 FMUL.X     FP7, FP2            ;
                 FSUB.X     FP2, FP1            ;
                 FCMP.X     FP4, FP1            ;
                 FBGE.W     notMin2             ;
                 FMOVE.X    FP1, FP4            ;
notMin2          FCMP.X     FP5, FP1            ;
                 FBLE.W     notMax2             ;
                 FMOVE.X    FP1, FP5            ;
notMax2          SUBQ.L     #1, D0              ;
                 BGT.B      SubLoop1            ;

GoOn1            FABS.X     FP4                 ; FP4: abs(min)
                 FABS.X     FP5                 ; FP5: abs(max)
                 FCMP.X     FP4, FP5            ;
                 FBGT.W     Bigger1             ;
                 FMOVE.X    FP4, FP5            ;
Bigger1          FMOVE.W    #0, FP4             ; FTST wouldn't work in MPW 2.02's asm!
                 FCMP.X     FP4, FP5            ;
                 FBNE.W     notZero1            ;
                 FMOVE.W    #1, FP5             ;
notZero1         CLR.W      D0                  ; D0: destScale
                 FLOG2.X    FP5                 ;
                 FMOVE.W    #13, FP4            ;
                 FCMP.X     FP4, FP5            ;
```

```
                FBLT.W      GoOn2               ;
                FSUB.X      FP4, FP5            ;
                FADD.W      #1, FP5             ;
                FINTRZ.X    FP5                 ; Round Up to next integer
                FMOVE.W     FP5, D0             ; D0: destScale
GoOn2           MOVE.W      D0, (A0)            ; destScale stored
                FMOVE.W     D0, FP4             ; FP4: destScale
                FTWOTOX.X   FP4                 ; FP4: 2^destScale (divFactor)
                FMOVE.W     #1, FP5             ;
                FDIV.X      FP4, FP5            ; FP5: 1/(2^destScale (mulFactor)

                MOVEA.L     srcPtr1(A6), A1     ;
                MOVEA.L     srcPtr2(A6), A2     ;
                MOVEA.L     destPtr(a6), A3     ;
                MOVE.L      srcSize(A6), D0     ;
                TST.W       operation(A6)       ;
                BLT.S       SubLoop2            ;

AddLoop2        FMOVE.W     (A1)+, FP1          ;
                FMOVE.W     (A2)+, FP2          ;
                FMUL.X      FP6, FP1            ;
                FMUL.X      FP7, FP2            ;
                FADD.X      FP2, FP1            ;
                FMUL.X      FP5, FP1            ;
                FMOVE.W     FP1, (A3)+          ;
                SUBQ.L      #1, D0              ;
                BGT.B       AddLoop2            ;
                BRA.S       exit                ;

SubLoop2        FMOVE.W     (A1)+, FP1          ;
                FMOVE.W     (A2)+, FP2          ;
                FMUL.X      FP6, FP1            ;
                FMUL.X      FP7, FP2            ;
                FSUB.X      FP2, FP1            ;
                FMUL.X      FP5, FP1            ;
                FMOVE.W     FP1, (A3)+          ;
                SUBQ.L      #1, D0              ;
                BGT.B       SubLoop2            ;

exit            FMOVEM.X    (SP)+, UsedFRegs    ;
                MOVEM.L     (SP)+, UsedRegs     ;
                UNLK        A6                  ;
                MOVE.L      (SP)+, A0           ;
                ADDA.W      #26, SP             ;
                JMP         (A0)                ;

                DC.B        'ADSB2BFF'          ; MacsBug Name

                ENDPROC


Hcdc2BufsF      PROC        EXPORT

; procedure Hcdc2BufsF(srcPtr1:    ptr;
;                      srcPtr2:    ptr;
;                      destPtr:    ptr;
;                      srcScale1:  integer;
;                      srcScale2:  integer;
;                  var destScale:  integer;     (ptr)
;                      rowWords:   integer;
;                      operation:  integer);
;
; procedure Hcdc2BufsF (for Hartley Convolve or Deconvolve or Correlate two Buffers w/ FPU)
; is a feature packed routine that performs the Hartley analog of complex multiplication
; or division or conjugate multiplication on 2 rowWords x rowWords integer Hartley
; transform buffers pointed to by srcPtr1 and srcPtr2, placing the result in
; the Hartley transform buffer pointed to by destPtr.  All three of these computations
; are so similar (and the addressing so complex) that they have been lumped togehter
; in one routine.
; The operations on H1 and H2 (src1 and src2) can be expressed using Bracewell's notation
; (p 43, 'The Hartley Transform') as
; (Convolution):              H1(+f)*H2e + H1(-f)*H2o
; (Deconvolution):           (H1(+f)*H2e - H1(-f)*H2o)/(H2(+f)^2 + H2(-f)^2)
; (Correlation):             H1(+f)*H2e - H1(-f)*H2o
; The operation desired is specified by the operation paramter as follows:
;                 operation  function performed
;                 < 0        Hartley Multiplication   (for Convolution)
;                 = 0        Hartley Division         (for Deconvolution)
;                 > 0        Hartley Conjugate Multiplication (for Correlation)
; As in PSFHT2D and other routines, this calculation must be performed twice.  The first
; time through, only the maximum and minimum values of the output are recorded.  From the
; extrema, a linear mapping from the 32 bit output into the 16 bit output buffer is
; devised.  On the second time through, this mapping is used to store the computed result
; into the output buffer.
; The order of arguments is not important for multiplication (convolution) as it commutes,
; however, for division and conjugate multiplication the operations are
; src1/src2 and src1(src2)* respectivley, where * indicates conjugation.
; NOTE: this accomplishes with one loop what PSFHT2D does with two loops and two special
;                 cases; the addressing calculation involved here does not take long in comparison
```

```
;                with PSFHT2D, so that routine should be modified to use this more compact format.
; NOTE: rowWords MUST be a power of 2; this routine is designed to work exclusively on
;                     square, two-dimensional integer matrices.
; NOTE: Autocorrelation needs only half the operations that correlation needs; this is
;                not taken advantage of here; THIS ROUTINE IS NOT OPTIMIZED AT ALL.
; NOTE: FPU used here.

srcPtr1          EQU        28                ; A6 Offsets
srcPtr2          EQU        24                ;
destPtr          EQU        20                ;
srcScale1        EQU        18                ;
srcScale2        EQU        16                ;
destScale        EQU        12                ;
rowWords         EQU        10                ;
operation        EQU        8                 ;

UsedRegs         REG        D2-D7/A2-A4
UsedFRegs        FREG       FP2-FP7

start            LINK       A6, #0            ;
                 MOVEM.L    UsedRegs, -(SP)   ;
                 FMOVEM.X   UsedFRegs, -(SP)  ;

                 FMOVE.W    srcScale1(A6), FP6  ;
                 FTWOTOX.X  FP6                 ; FP6: scale1
                 FMOVE.W    srcScale2(A6), FP7  ;
                 FTWOTOX.X  FP7                 ; FP7: scale2

                 MOVE.W     operation(A6), D0   ; D0: * | operation
                 SWAP       D0                  ;
                 MOVE.W     rowWords(A6), D0    ; D0: operation | rowWords
                 MOVEA.L    srcPtr1(A6), A1     ; A1: srcPtr1
                 MOVEA.L    srcPtr2(A6), A2     ; A2: srcPtr2

                 MOVE.W     #15, D2             ;
logLoop          BTST.L     D2, D0              ;
                 DBNE.W     D2, logLoop         ; D2: log2(rowWords)
                 MOVE.W     D2, D1              ;
                 SWAP       D1                  ;
                 MOVE.W     D0, D1              ;
                 SUBQ.W     #1, D1              ; D1: log2(rowWords) | mod mask & column count
                 MOVE.W     D1, D2              ; D2: column counter
                 MOVE.W     D1, D3              ; D3: row counter
                 FMOVE.L    #$7FFFFFFF, FP4     ; FP4: min
                 FMOVE.L    #$80000000, FP5     ; FP5: max

RCLoop1          MOVE.W     D3, D4              ;
                 EXT.L      D4                  ;
                 SWAP       D1                  ;
                 ASL.L      D1, D4              ; D4: N * row
                 SWAP       D1                  ;
                 MOVE.W     D2, D5              ;
                 EXT.L      D5                  ;
                 ADD.L      D5, D4              ; D4: N * row + col
                 ASL.L      #1, D4              ; D4: even (word) address of H(r, c)
                 MOVEA.L    D4, A3              ; A3: address of H(row, col)

                 MOVE.L     D3, D4              ;
                 MOVE.W     D0, D5              ; D5: N
                 SUB.W      D4, D5              ; D5: N - row
                 AND.W      D1, D5              ; D4: (N - row) mod N = modRow
                 EXT.L      D5                  ;
                 SWAP       D1                  ;
                 ASL.L      D1, D5              ; D4: N * modRow
                 SWAP       D1                  ;
                 MOVE.W     D0, D6              ; D6: N
                 SUB.W      D2, D6              ; D6: N - col
                 AND.W      D1, D6              ; D6: (N - col) mod N = modCol
                 EXT.L      D6                  ;
                 ADD.L      D6, D5              ; D4: N * modRow + modCol
                 ASL.L      #1, D5              ; D4: even (word) address of H(modRow, modCol)
                 MOVEA.L    D5, A4              ; A4: address of H(modRow, modCol)

; address calculation done

                 FMOVE.W    0(A2, A3.L), FP0    ; FP0: H2(r,c)
                 FMUL.X     FP7, FP0            ;
                 FMOVE.X    FP0, FP1            ;
                 FMOVE.W    0(A2, A4.L), FP2    ; FP2: H2(modRow, modCol)
                 FMUL.X     FP7, FP2            ;
                 FADD.X     FP2, FP0            ;
                 FSUB.X     FP2, FP1            ;
                 FDIV.W     #2, FP0             ; FP0: H2even
                 FDIV.W     #2, FP1             ; FP1: H2odd

                 MOVE.L     D0, D7              ; because a swap will change CCR's Z & N bits
                 SWAP       D7                  ;
                 TST.W      D7                  ;
                 BGT.W      Corr1               ;
```

```
                BLT.B     Conv1              ;

Deconv1         FMOVE.X   FP0, FP2           ;
                FMUL.X    FP2, FP2           ; FP2: sqr(H2e)
                FMOVE.X   FP1, FP3           ;
                FMUL.X    FP3, FP3           ; FP3: sqr(H2o)
                FADD.X    FP3, FP2           ; FP2: sqr(H2e) + sqr(H2o) [= (H2^2 + H2(-)^2)/2]
                FMUL.W    #2, FP2            ; FP2: = sqr(H2) + sqr(H2(-)) = denominator
                FBNE.W    notZero1           ;
                FMOVE.W   #1, FP2            ; to prevent divide by zero
notZero1        FMOVE.W   0(A1, A3.L), FP3   ;
                FMUL.X    FP6, FP3           ; FP3: H1(r,c)
                FMUL.X    FP3, FP0           ; FP0: H1(r,c)* H2even
                FMOVE.W   0(A1, A4.L), FP3   ;
                FMUL.X    FP6, FP3           ; FP3: H1(modRow, modCol)
                FMUL.X    FP3, FP1           ; FP1: H1(modRow, modCol)* H2odd
                FSUB.X    FP1, FP0           ; FP0: deconv numerator
                FDIV.X    FP2, FP0           ; FP0: ratio
                FCMP.X    FP4, FP0           ;
                FBGE.W    notMin1            ;
                FMOVE.X   FP0, FP4           ;
notMin1         FCMP.X    FP5, FP0           ;
                FBLE.W    notMax1            ;
                FMOVE.X   FP0, FP5           ;
notMax1         BRA.S     GoOn1              ;

Conv1           FMOVE.W   0(A1, A3.L), FP2   ; FP2: H1(r,c)
                FMOVE.W   0(A1, A4.L), FP3   ; FP3: H1(modRow, modCol)
                FMUL.X    FP6, FP2           ;
                FMUL.X    FP6, FP3           ;
                FMUL.X    FP2, FP0           ; FP0: H1(r,c)*H2even
                FMUL.X    FP3, FP1           ; FP1: H1(modRow, modCol)*H2odd
                FADD.X    FP1, FP0           ;
                FCMP.X    FP4, FP0           ;
                FBGE.W    notMin2            ;
                FMOVE.X   FP0, FP4           ;
notMin2         FCMP.X    FP5, FP0           ;
                FBLE.W    notMax2            ;
                FMOVE.X   FP0, FP5           ;
notMax2         BRA.S     GoOn1              ;

Corr1           FMOVE.W   0(A1, A3.L), FP2   ; FP2: H1(r,c)
                FMOVE.W   0(A1, A4.L), FP3   ; FP3: H1(modRow, modCol)
                FMUL.X    FP6, FP2           ;
                FMUL.X    FP6, FP3           ;
                FMUL.X    FP2, FP0           ; FP0: H1(r,c)*H2even
                FMUL.X    FP3, FP1           ; FP1: H1(modRow, modCol)*H2odd
                FSUB.X    FP1, FP0           ;
                FCMP.X    FP4, FP0           ;
                FBGE.W    notMin3            ;
                FMOVE.X   FP0, FP4           ;
notMin3         FCMP.X    FP5, FP0           ;
                FBLE.W    GoOn1              ;
                FMOVE.X   FP0, FP5           ;

; calculation complete

GoOn1           DBRA.W    D2, RCLoop1        ;

                MOVE.W    D1, D2             ; Restore column counter
                DBRA.W    D3, RCLoop1        ;

; First traversal done - max and min found

                MOVEA.L   destScale(A6), A0  ; A0: @destScale
                FABS.X    FP4                ; FP4: abs(min)
                FABS.X    FP5                ; FP5: abs(max)
                FCMP.X    FP4, FP5           ;
                FBGT.W    Bigger2            ;
                FMOVE.X   FP4, FP5           ;
Bigger2         FMOVE.W   #0, FP4            ; FTST wouldn't work in MPW 2.02's asm!
                FCMP.X    FP4, FP5           ;
                FBNE.W    notZero2           ;
                FMOVE.W   #1, FP5            ;
notZero2        CLR.W     D5                 ; D5: destScale
                FLOG2.X   FP5                ;
                FMOVE.W   #13, FP4           ;
                FCMP.X    FP4, FP5           ;
                FBLT.W    GoOn2              ;
                FSUB.X    FP4, FP5           ;
                FADD.W    #1, FP5            ;
                FINTRZ.X  FP5                ; Round Up to next integer
                FMOVE.W   FP5, D5            ; D5: destScale
GoOn2           MOVE.W    D5, (A0)           ; destScale stored
                FMOVE.W   D5, FP4            ; FP4: destScale
                FTWOTOX.X FP4                ; FP4: 2^destScale (divFactor)
                FMOVE.W   #1, FP5            ;
                FDIV.X    FP4, FP5           ; FP5: 1/(2^destScale (mulFactor)
```

```
; mapping set up - restore row and column counters, etc. for next traversal

                MOVEA.L    srcPtr1(A6), A1     ;
                MOVEA.L    srcPtr2(A6), A2     ;
                MOVEA.L    destPtr(A6), A0     ;
                MOVE.W     D1, D2              ; D2: col counter
                MOVE.W     D1, D3              ; D3: row counter

RCLoop2         MOVE.W     D3, D4              ;
                EXT.L      D4                  ;
                SWAP       D1                  ;
                ASL.L      D1, D4              ; D4: N * row
                SWAP       D1                  ;
                MOVE.W     D2, D5              ;
                EXT.L      D5                  ;
                ADD.L      D5, D4              ; D4: N * row + col
                ASL.L      #1, D4              ; D4: even (word) address of H(r, c)
                MOVEA.L    D4, A3              ; A3: address of H(row, col)

                MOVE.L     D3, D4              ;
                MOVE.W     D0, D5              ; D5: N
                SUB.W      D4, D5              ; D5: N - row
                AND.W      D1, D5              ; D4: (N - row) mod N = modRow
                EXT.L      D5                  ;
                SWAP       D1                  ;
                ASL.L      D1, D5              ; D4: N * modRow
                SWAP       D1                  ;
                MOVE.W     D0, D6              ; D6: N
                SUB.W      D2, D6              ; D6: N - col
                AND.W      D1, D6              ; D6: (N - col) mod N = modCol
                EXT.L      D6                  ;
                ADD.L      D6, D5              ; D4: N * modRow + modCol
                ASL.L      #1, D5              ; D4: even (word) address of H(modRow, modCol)
                MOVEA.L    D5, A4              ; A4: address of H(modRow, modCol)

; address calculation done

                FMOVE.W    0(A2, A3.L), FP0    ; FP0: H2(r,c)
                FMUL.X     FP7, FP0            ;
                FMOVE.X    FP0, FP1            ;
                FMOVE.W    0(A2, A4.L), FP2    ; FP2: H2(modRow, modCol)
                FMUL.X     FP7, FP2            ;
                FADD.X     FP2, FP0            ;
                FSUB.X     FP2, FP1            ;
                FDIV.W     #2, FP0             ; FP0: H2even
                FDIV.W     #2, FP1             ; FP1: H2odd

                MOVE.L     D0, D7              ; because a swap will change CCR's Z & N bits
                SWAP       D7                  ;
                TST.W      D7                  ;
                BGT.S      Corr2               ;
                BLT.S      Conv2               ;

Deconv2         FMOVE.X    FP0, FP2            ;
                FMUL.X     FP2, FP2            ; FP2: sqr(H2e)
                FMOVE.X    FP1, FP3            ;
                FMUL.X     FP3, FP3            ; FP3: sqr(H2o)
                FADD.X     FP3, FP2            ; FP2: sqr(H2e) + sqr(H2o) [= (H2^2 + H2(-)^2)/2]
                FMUL.W     #2, FP2             ; FP2: = sqr(H2) + sqr(H2(-)) = denominator
                FBNE.W     notZero3            ;
                FMOVE.W    #1, FP2             ; to prevent divide by zero
notZero3        FMOVE.W    0(A1, A3.L), FP3    ;
                FMUL.X     FP6, FP3            ; FP3: H1(r,c)
                FMUL.X     FP3, FP0            ; FP0: H1(r,c)* H2even
                FMOVE.W    0(A1, A4.L), FP3    ;
                FMUL.X     FP6, FP3            ; FP3: H1(modRow, modCol)
                FMUL.X     FP3, FP1            ; FP1: H1(modRow, modCol)* H2odd
                FSUB.X     FP1, FP0            ; FP0: deconv numerator
                FDIV.X     FP2, FP0            ; FP0: ratio
                BRA.S      GoOn3               ;

Conv2           FMOVE.W    0(A1, A3.L), FP2    ; FP2: H1(r,c)
                FMOVE.W    0(A1, A4.L), FP3    ; FP3: H1(modRow, modCol)
                FMUL.X     FP6, FP2            ;
                FMUL.X     FP6, FP3            ;
                FMUL.X     FP2, FP0            ; FP0: H1(r,c)*H2even
                FMUL.X     FP3, FP1            ; FP1: H1(modRow, modCol)*H2odd
                FADD.X     FP1, FP0            ;
                BRA.S      GoOn3               ;

Corr2           FMOVE.W    0(A1, A3.L), FP2    ; FP2: H1(r,c)
                FMOVE.W    0(A1, A4.L), FP3    ; FP3: H1(modRow, modCol)
                FMUL.X     FP6, FP2            ;
                FMUL.X     FP6, FP3            ;
                FMUL.X     FP2, FP0            ; FP0: H1(r,c)*H2even
                FMUL.X     FP3, FP1            ; FP1: H1(modRow, modCol)*H2odd
                FSUB.X     FP1, FP0            ;

; calculation complete
```

```
GoOn3           FMUL.X      FP5, FP0            ; FP0: result scaled down
                FMOVE.W     FP0, 0(A0, A3.L)    ; result stored
                DBRA.W      D2, RCLoop2         ;

                MOVE.W      D1, D2              ; Restore column counter
                DBRA.W      D3, RCLoop2         ;

exit            FMOVEM.X    (SP)+, UsedFRegs    ;
                MOVEM.L     (SP)+, UsedRegs     ;
                UNLK        A6                  ;
                MOVE.L      (SP)+, A0           ;
                ADDA.L      #24, SP             ;
                JMP         (A0)                ;

                DC.B        'HCDC2BFF'          ; MacsBug Name

                ENDPROC


ClipMinMax      PROC        EXPORT

; procedure ClipMinMax(baseAddr: ptr            ;
;                      bufSize: longint);
;
; procedure ClipMinMax is used on a PixMap to convert all of its 0 pixels to 1 and
; all of its 255 pixels to 254, thereby avoiding the sacred foreColor and bgColor
; extremeties.  baseAddr points to an array of bytes bufSize long.

baseAddr        EQU         12                  ;
bufSize         EQU         8                   ;

UsedRegs        REG         D2-D3               ;

start           LINK        A6, #0              ;
                MOVEM.L     UsedRegs, -(SP)     ;
                MOVE.L      baseAddr(A6), A0    ; A0: baseAddr
                MOVE.L      bufSize(A6), D0     ; D0: bufSize
                ASR.L       #1, D0              ; D0: bufSize words
                MOVE.W      #$00FF, D1          ; D1: low Byte mask

loop            MOVE.W      (A0), D2            ;
                MOVE.W      D2, D3              ;
                LSR.W       #8, D3              ; D3: 1st byte
                AND.W       D1, D2              ; D2: 2nd byte

                BNE.S       notZero1            ; check 2nd byte
                MOVE.W      #1, D2              ;
notZero1        CMPI.W      #254, D2            ;
                BLE.S       notBig1             ;
                MOVE.W      #254, D2            ;

notBig1         TST.W       D3                  ; check 1st byte
                BNE.S       notZero2            ;
                MOVE.W      #1, D3              ;
notZero2        CMPI.W      #254, D3            ;
                BLE.S       notBig2             ;
                MOVE.W      #254, D3            ;

notBig2         LSL.W       #8, D3              ; store bytes back
                OR.W        D2, D3              ;
                MOVE.W      D3, (A0)+           ;
                SUBQ.L      #1, D0              ;
                BGT.S       loop                ;

exit            MOVEM.L     (SP)+, UsedRegs     ;
                UNLK        A6                  ;
                MOVE.L      (SP)+, A0           ;
                ADDQ.L      #8, SP              ;
                JMP         (A0)                ;

                DC.B        'CPMINMAX'          ; MacsBug Name

                ENDPROC


DblMem          PROC        EXPORT

; procedure DblMem(srcPtr:   ptr;
;                  destPtr:  ptr;
;                  srcSize:  longint);
;
; This procedure takes an input block of memory, a sequence of bytes (e.g. from
; an 8 bit deep pixel map), and copies it into a block of memory twice the size -
; a 16 bit deep pixel map. The byte with value 1 is mapped into 32, while the byte
; with value 255 is mapped into 8160; the values are simply left shifted by 5 bits.
; srcSize is measured in bytes.

srcPtr          EQU         16                  ; A6 offsets
```

```
destPtr         EQU     12              ;
srcSize         EQU     8               ;

UsedRegs        REG     D2-D4

                LINK    A6, #0          ; no locals (disco surfer!)
                MOVEM.L UsedRegs, -(SP) ;
                MOVEA.L srcPtr(A6), A0  ; A0: srcPtr
                MOVEA.L destPtr(A6), A1 ; A1: destPtr
                MOVE.L  srcSize(A6), D0 ; D0: srcSize
                ASR.L   #1, D0          ; D0: srcSize in words
                MOVE.W  #5, D1          ; D1: num bits to shift
                MOVE.W  #$00FF, D2      ; D2: low byte mask

loop            MOVE.W  (A0)+, D3       ;
                MOVE.W  D3, D4          ;
                LSR.W   #8, D3          ; D3: 1st byte
                AND.W   D2, D4          ; D4: 2nd byte
                ASL.W   D1, D3          ;
                ASL.W   D1, D4          ;
                SWAP    D3              ;
                MOVE.W  D4, D3          ;
                MOVE.L  D3, (A1)+       ;
                SUBQ.L  #1, D0          ;
                BGT.S   loop            ;

exit            MOVEM.L (SP)+, UsedRegs ; restore registers
                UNLK    A6              ;
                MOVE.L  (SP)+, A0       ;
                ADDA.W  #12, SP         ; pop parameters
                JMP     (A0)            ;

                DC.B    'DBLMEM  '      ; MacsBug Name

                ENDPROC


IntToByteF      PROC    EXPORT

; procedure IntToByteF(srcPtr: ptr;
;                      destPtr: ptr;
;                      rowWords: integer);
;
; procedure IntToByte scans the rowWords^2 matrix of integers pointed
; to by srcPtr recording the maximum and minimum values.
; These are then used to define a linear mapping onto the byte range 0-255.
; A second pass over the integer array is then made to map the integers
; into the rowWords^2 byte matrix pointed to by destPtr.
; IntToByteT differs from IntToByte in that it uses the FPU for the linear
; mapping and is therefore slower, but simpler to read, smaller and more accurate.

srcPtr          EQU     14              ; A6 offsets
destPtr         EQU     10              ;
rowWords        EQU     8               ;

UsedRegs        REG     D2/D6-D7        ;

start           LINK    A6, #0          ;
                MOVEM.L UsedRegs, -(SP) ;
                MOVE.L  srcPtr(A6), A0  ; A0: srcPtr
                MOVE.W  rowWords(A6), D0 ; D0: rowWords

                MOVE.W  #$7FFF, D6      ; D6: minVal
                MOVEQ.L #0, D7          ;
                BSET.L  #15, D7         ; D7: maxVal

                MULS.W  D0, D0          ; D0: srcSize
                MOVE.L  D0, D1          ; D1: count

scanLoop        MOVE.W  (A0)+, D2       ;
                CMP.W   D6, D2          ;
                BGT.S   notMin          ;
                MOVE.W  D2, D6          ;
notMin          CMP.W   D7, D2          ;
                BLE.S   notMax          ;
                MOVE.W  D2, D7          ;
notMax          SUBQ.L  #1, D1          ;
                BGT.S   scanLoop        ; max & min found
                SUB.W   D6, D7          ; D7: range
                BNE.S   notZero         ;
                MOVE.W  #1, D7          ;

notZero         FMOVE.W D7, FP0         ; FPO: range
                FMOVE.W #255, FP1       ;
                FDIV.X  FP0, FP1        ; FP1: mul Factor (scale)

                MOVE.L  srcPtr(A6), A0  ; A0: srcPtr
                MOVE.L  destPtr(A6), A1 ; A1: destPtr
```

```
                ASR.L       #1, D0              ; D0: count
storeLoop       MOVE.L      (A0)+, D1           ; D1: 2nd word (2 words at at time)
                MOVE.L      D1, D2              ;
                SWAP        D2                  ; D2: 1st word
                SUB.W       D6, D1              ;
                SUB.W       D6, D2              ;
                FMOVE.W     D2, FP0             ;
                FMUL.X      FP1, FP0            ;
                FMOVE.W     FP0, D2             ;

                FMOVE.W     D1, FP0             ;
                FMUL.X      FP1, FP0            ;
                FMOVE.W     FP0, D1             ;

                LSL.W       #8, D2              ;
                OR.W        D2, D1              ;
                MOVE.W      D1, (A1)+           ;

                SUBQ.L      #1, D0              ;
                BGT.S       storeLoop           ;

exit            MOVEM.L     (SP)+, UsedRegs     ;
                UNLK        A6                  ;
                MOVE.L      (SP)+, A0           ;
                ADDA.W      #10, SP             ;
                JMP         (A0)                ;

                DC.B        'INTOBYTF'          ; MacsBug Name

                ENDPROC


SwapBBlock      PROC        EXPORT

; procedure SwapBBlock(matPtr: ptr;
;                      rowBytes: integer);

; procedure SwapBBlock rearranges the square matrix of BYTES pointed to by matPtr.
; rowBytes must be a power of 2 >= 4. The moves are word sized.
; It swaps the matrix's 2nd and 4th quadrants and its 1st and 3rd quadrants, so
; that the transform is centered in the matrix.

matPtr          EQU         10                  ; A6 offsets
rowBytes        EQU         8                   ;

UsedRegs        REG         D2-D7/A2-A3         ;

start           LINK        A6, #0              ;
                MOVEM.L     UsedRegs, -(SP)     ;
                MOVE.L      matPtr(A6), A0      ;
                MOVE.W      rowBytes(A6), D0    ;

                EXT.L       D0                  ; D0: rowBytes
                MOVE.L      D0, D1              ;
                ASR.L       #1, D1              ; D1: rowBytes div 2
                MOVE.W      #15, D2             ;
logLoop         BTST.L      D2, D0              ;
                DBNE.W      D2, logLoop         ; D2: ?? | Log2(rowBytes)
                SUBQ.W      #1, D2              ;
                MOVE.L      D0, D3              ;
                ASL.L       D2, D3              ; D3: rowBytes^2 div 2

                MOVEA.L     A0, A1              ; A0: Quad 2 ptr
                MOVEA.L     A0, A2              ;
                ADDA.L      D1, A1              ; A1: Quad 1 ptr
                ADDA.L      D3, A2              ; A2: Quad 3 ptr
                MOVEA.L     A2, A3              ;
                ADDA.L      D1, A3              ; A3: Quad 4 ptr

                MOVE.W      D1, D3              ; this must be pos, word sized.
                ASR.W       #1, D3              ; D2: rowWords
                SUBQ.W      #1, D3              ; D2: rowWords - 1
                MOVE.W      D3, D2              ;
                SWAP        D2                  ;
                MOVE.W      D3, D2              ; D2: col | col
                MOVE.W      D1, D3              ;
                SUBQ.W      #1, D3              ; D3: row

loop            MOVE.W      (A0), D4            ;
                MOVE.W      (A1), D5            ;
                MOVE.W      (A2), D6            ;
                MOVE.W      (A3), D7            ;
                MOVE.W      D4, (A3)+           ;
                MOVE.W      D5, (A2)+           ;
                MOVE.W      D6, (A1)+           ;
                MOVE.W      D7, (A0)+           ;

                DBRA.W      D2, loop            ;
```

```
                MOVE.L      D2, D4              ;
                SWAP        D4                  ;
                MOVE.W      D4, D2              ; restore col counter

                ADDA.L      D1, A0              ;
                ADDA.L      D1, A1              ;
                ADDA.L      D1, A2              ;
                ADDA.L      D1, A3              ; update the address ptrs

                DBRA.W      D3, loop            ;

                MOVEM.L     (SP)+, UsedRegs     ;
                UNLK        A6                  ;
                MOVE.L      (SP)+, A0           ;
                ADDQ.L      #6, SP              ;
                JMP         (A0)                ;

                DC.B        'SWPBBLOK'          ; MacsBug Name

                ENDPROC


Transpose       PROC        EXPORT

; procedure transpose(matPtr: ptr;
;                     rowWords: integer);

; procedure transpose transposes a rowWords x rowWords matrix of words.
; WARNING: the integer block pointed to by matPtr must have rowWords^2
; words allocated to it. (it must be a square matrix).  No checking for
; this is done.

matPtr          EQU         10                  ; A6 offsets
rowWords        EQU         8                   ;

UsedRegs        REG         D2-D7               ;

                LINK        A6, #0              ; no locals (my wave!)
                MOVEM.L     UsedRegs, -(SP)     ;
                MOVE.L      matPtr(A6), A0      ; Base Address
                MOVE.W      rowWords(A6), D0    ; counter1
                EXT.L       D0                  ; make it long
                MOVE.L      D0, D7              ;
                ASL.L       #1, D0              ; integers = 2 bytes long -> all off's even
                MOVE.L      D0, D1              ;

                SUBQ.L      #2, D1              ; D1: rowWords - 1 == col
                MOVE.L      D1, D2              ;
                SUBQ.L      #2, D2              ; D2: rowWords - 2 == row
                MOVE.L      D1, D3              ;
                MOVE.L      D2, D4              ;
                MULU.W      D7, D3              ; D3: col * N
                MULU.W      D7, D4              ; D4: row * N

rowLoop         MOVE.L      D1, D5              ;
                ADD.L       D4, D5              ; D5: Add1 = row*N + col
                MOVE.L      D2, D6              ;
                ADD.L       D3, D6              ; D6: Add2 = col*N + row

colLoop         MOVE.W      0(A0,D6.L), D7      ;
                MOVE.W      0(A0,D5.L), 0(A0,D6.L)  ;
                MOVE.W      D7, 0(A0,D5.L)      ; swap Add1 & Add2

                SUB.L       D0, D6              ;
                SUBQ.L      #2, D5              ;
                CMP.L       D5, D6              ;
                BNE.S       colLoop             ;

                SUBQ.L      #2, D2              ;
                SUB.L       D0, D4              ;
                BGE.S       rowLoop             ;

exit            MOVEM.L     (SP)+, UsedRegs     ; restore registers
                UNLK        A6                  ;
                MOVE.L      (SP)+, A0           ;
                ADDQ.L      #6, SP              ; pop parameters
                JMP         (A0)                ;

                DC.B        'TRANSPOS'          ; MacsBug Name

                ENDPROC


ToRCFHT         PROC        EXPORT

; procedure toRCFHT(matPtr: Ptr;
;                   rowWords: integer);
;
; procedure ToRCFHT takes a matrix of words which has been 1D FHT'd row
```

```
; by row and col by col and produces a real 2D FHT from the result
; From the pascal unit FFTnFHT.p, the algorithm is:

;  for row := 0 to maxN div 2 do                { Now calculate actual Hartley transform }
;   for col := 0 to maxN div 2 do begin
;     mRow := (maxN - row) mod maxN;
;     mCol := (maxN - col) mod maxN;
;     A := x[row, col];      { see Bracewell, 'Fast 2D Hartley Transf.' IEEE Procs. 9/86 }
;     B := x[mRow, col];
;     C := x[row, mCol];
;     D := x[mRow, mCol];
;     E := ((A + D) - (B + C)) / 2;
;     x[row, col] := A - E;
;     x[mRow, col] := B + E;
;     x[row, mCol] := C + E;
;     x[mRow, mCol] := D - E;
;    end;

; WARNING: the integer block pointed to by matPtr must have rowWords^2
; words allocated to it. (it must be a square matrix).  No checking for
; this is done.  Furthermore, rowWords must be a power of 2; no checking is
; done for this either.

matPtr          EQU     10                  ; A6 Offsets
rowWords        EQU     8                   ;

UsedRegs        REG     D2-D7/A2-A4

                LINK    A6, #0              ; no locals (shoulder-hopper!)
                MOVEM.L UsedRegs, -(SP)     ;
                MOVE.L  matPtr(A6), A0      ; Base Address
                MOVE.W  rowWords(A6), D0    ; counter

                MOVE.W  D0, D1              ;
                SUBQ.W  #1, D0              ;
                SWAP    D0                  ;
                MOVE.W  D1, D0              ; D0: mask | maxN
                MOVE.W  #15, D2             ;
logLoop         BTST.L  D2, D1              ;
                DBNE.W  D2, logLoop         ;
                MOVE.W  D2, D1              ;
                SWAP    D1                  ;
                MOVE.W  D0, D1              ;
                LSR.W   #1, D1              ; D1: log2(maxN) | row □
                MOVE.W  D1, D2              ;
                SWAP    D2                  ;
                MOVE.W  D1, D2              ; D2: col | col

rowLoop         MOVEQ.L #0, D3              ;
                MOVEQ.L #0, D4              ;
                MOVEQ.L #0, D5              ;
                MOVEQ.L #0, D6              ; clear the registers…
                MOVEQ.L #0, D7              ; for mixed w/l operations
                MOVE.W  D0, D3              ;
                MOVE.W  D0, D4              ;
                SUB.W   D1, D3              ; D3: MaxN - row
                SUB.W   D2, D4              ; D4: MaxN - col
                SWAP    D0                  ;
                AND.W   D0, D3              ; D3: (MaxN - row) mod maxN = mRow
                AND.W   D0, D4              ; D4: (MaxN - col) mod maxN = mCol
                SWAP    D0                  ;
                MOVE.W  D1, D5
                SWAP    D1                  ;
                ASL.L   D1, D3              ; D3: maxN * mRow
                ASL.L   D1, D5              ; D5: maxN * row
                SWAP    D1                  ;

                MOVE.W  D2, D6              ;
                ADD.L   D5, D6              ; maxN * row + col
                ASL.L   #1, D6              ; integers -> double addr
                MOVEA.L D6, A1              ; A1: (A)

                MOVE.W  D2, D7              ;
                ADD.L   D3, D7              ; maxN * mRow + col
                ASL.L   #1, D7              ; integers take 2 bytes
                MOVEA.L D7, A2              ; A2: (B)

                MOVEQ.L #0, D6              ; make space
                MOVE.W  D4, D6              ;
                ADD.L   D5, D6              ; maxN * row + mCol
                ASL.L   #1, D6              ; integers take 2 bytes
                MOVEA.L D6, A3              ; A3: (C)

                MOVEQ.L #0, D7              ; make space
                MOVE.W  D4, D7              ;
                ADD.L   D3, D7              ; maxN * mRow + mCol
                ASL.L   #1, D7              ; even addresses for int's
                MOVEA.L D7, A4              ; A4: (D)
```

```
                MOVE.W      0(A0, A1.L), D3     ; D3: A
                MOVE.W      0(A0, A2.L), D4     ; D4: B
                MOVE.W      0(A0, A3.L), D5     ; D5: C
                MOVE.W      0(A0, A4.L), D6     ; D6: D

                MOVE.W      D6, D7              ;
                ADD.W       D3, D7              ; D7: A + D
                SUB.W       D4, D7              ; D7: A+D-B
                SUB.W       D5, D7              ; D7: (A+D)-(B+C)
                ASR.W       #1, D7              ; D7: E (no Rounding!!)
                BEQ.S       continue            ; E=0 -> no changes

                SUB.W       D7, D3              ; D3: A - E
                ADD.W       D7, D4              ; D4: B + E
                ADD.W       D7, D5              ; D5: C + E
                SUB.W       D7, D6              ; D6: D - E

                MOVE.W      D3, 0(A0, A1.L)     ;
                MOVE.W      D4, 0(A0, A2.L)     ;
                MOVE.W      D5, 0(A0, A3.L)     ;
                MOVE.W      D6, 0(A0, A4.L)     ; all values stored

continue        DBRA        D2, rowLoop         ;

                MOVE.L      D2, D3              ;
                SWAP        D3                  ;
                MOVE.W      D3, D2              ; restore col counter

                DBRA        D1, rowLoop         ;

exit            MOVEM.L     (SP)+, UsedRegs     ; restore registers
                UNLK        A6                  ;
                MOVE.L      (SP)+, A0           ;
                ADDQ.L      #6, SP              ; pop parameters
                JMP         (A0)                ;

                DC.B        'TORCFHT '          ; MacsBug Name

                ENDPROC


psFHT2D         PROC        EXPORT

; procedure psFHT2D(srcPtr: ptr;
;                   destPtr: ptr;
;                   rowWords: integer;
;                   scaleType: integer);
;
; function psFHT2D computes the power spectrum of a 2D FHT.
; srcPtr points to a matrix of integers containing the transform.
; destPtr points to a matrix of bytes in which the scaled power spectrum of
; the transform is written. the matrix pointed to by srcPtr is
; rowWords x rowWords in size.  rowWords must be a power of 2.
;
; psFHT2D first scans the src matrix to determine the largest and
; smallest elements of the power spectrum.  A scaling function
; based on these extremes is developed according to 'scaleType'
; as follows:
; scaleType       scaling implemented:
;                 <=0         linear
;                 1-9         nth root            (1st root = linear)
;                 >=10        log
; Next, the src matrix is scanned again, this time the power
; spectrum at each point is again computed and scaled to a byte size
; before being stored in the dest matrix.
; The necessity of scanning the matrix twice was originally avoided by
; approximating the power spectrum element (a^2 + b^2)/2 by (|a| + |b|)/2,
; but while this avoided many multiplications, it occasionally produced
; spurious results.
; Here, some addressing calculations have been avoided by splitting up the
; matrix traversal into four separate sections: two loops and two elements.
; A more compact form of similar addressing may be found above in Hcdc2Bufs.

; register map
; D0              rowWords    A0                  srcPtr
; D1              mask        A1                  destPtr
; D2              loop counter                    A2   longOffset
; D3              scratch     A3                  shortOffset
; D4              scratch
; D5              scratch
;
; FP0             scratch
; FP1             mulFactor
; FP2             1/n (for nth root mapping)

srcPtr          EQU         16                  ; A6 offsets
destPtr         EQU         12                  ;
rowWords        EQU         10                  ;
scaleType       EQU         8
```

```
UsedRegs        REG         D2-D7/A2-A3         ;

start           LINK        A6, #0              ;
                MOVEM.L     UsedRegs, -(SP)     ;
                FMOVE.X     FP2, -(SP)          ;
                MOVE.L      srcPtr(A6), A0      ; A0: srcPtr
                MOVE.W      rowWords(A6), D0    ; D0: rowWords
                EXT.L       D0                  ;

                MOVE.W      D0, D1              ;
                SUBQ.W      #1, D1              ;
                EXT.L       D1                  ; D1: mod rowWords mask

                MOVE.W      D0, D2              ;
                MULS.W      D2, D2              ;
                MOVE.L      D2, -(SP)           ; for later
                MOVEA.L     D2, A2              ;
                ADDA.L      D2, A2              ;
                SUBA.L      #2, A2              ; A2: longOffset

                MOVE.W      D0, D3              ;
                EXT.L       D3                  ;
                MOVEA.L     D3, A3              ;
                ADDA.L      D3, A3              ;
                ADDA.L      #2, A3              ; A3: shortOffset

                MOVE        D0, D3              ;
                EXT.L       D3                  ;
                ASR.L       #1, D3              ; D3: rowWords div 2
                ASR.L       #1, D2              ; D2: rowWords^2 div 2
                SUB.L       D3, D2              ; D2: count
                MOVE.L      #$7FFFFFFF, D6      ; D6: minVal
                MOVEQ.L     #0, D7              ;
                BSET.L      #31, D7             ; D7: maxVal

loop1           MOVE.L      A3, D3              ;
                MOVE.L      D3, D4              ;
                ASR.L       #1, D4              ;
                AND.L       D1, D4              ; mod rowWords
                BNE.S       noMod1              ;
                MOVE.W      D0, D4              ;
                EXT.L       D4                  ;
                ADD.L       D4, D4              ;
                SUB.L       D4, D3              ;
noMod1          MOVE.W      0(A0, D3.L), D4     ;
                MOVE.W      0(A0, A2.L), D5     ;
                MULS.W      D4, D4              ; D4: a^2
                MULS.W      D5, D5              ; D5: b^2
                ADD.L       D5, D4              ; D4: a^2 + b^2
                CMP.L       D6, D4              ;
                BGE.S       notMin1             ;
                MOVE.L      D4, D6              ;
notMin1         CMP.L       D7, D4              ;
                BLE.S       notMax1             ;
                MOVE.L      D4, D7              ;

notMax1         SUBQ.L      #2, A2              ;
                ADDQ.L      #2, A3              ;
                SUBQ.L      #1, D2              ;
                BGT.S       loop1               ;

                MOVE.L      (SP), D3            ; check middle element, 1st column
                MOVE.W      0(A0, D3.L), D4     ;
                MULS.W      D4, D4              ;
                ADD.L       D4, D4              ;
                CMP.L       D6, D4              ;
                BGE.S       notMin2             ;
                MOVE.L      D4, D6              ;
notMin2         CMP.L       D7, D4              ;
                BLE.S       notMax2             ;
                MOVE.L      D4, D7              ;

notMax2         MOVE.W      (A0), D4            ; check 0th matrix element
                MULS.W      D4, D4              ;
                ADD.L       D4, D4              ;
                CMP.L       D6, D4              ;
                BGE.S       notMin3             ;
                MOVE.L      D4, D6              ;
notMin3         CMP.L       D7, D4              ;
                BLE.S       notMax3             ;
                MOVE.L      D4, D7              ;
                ; now check top Row
notMax3         MOVE.L      #2, A3              ; A3: shortOffset
                MOVE.W      D0, D3              ;
                EXT.L       D3                  ;
                ADD.L       D3, D3              ;
                SUBQ.L      #2, D3              ;
                MOVE.L      D3, A2              ; A2: longOffset
```

```
                MOVE.L      D0, D2              ;
                ASR.L       #1, D2              ; D2: counter

loop2           MOVE.W      0(A0, A2.L), D4     ;
                MOVE.W      0(A0, A3.L), D5     ;
                MULS.W      D4, D4              ;
                MULS.W      D5, D5              ;
                ADD.L       D5, D4              ; D4: a^2 + b^2
                CMP.L       D6, D4              ;
                BGE.S       notMin4             ;
                MOVE.L      D4, D6              ;
notMin4         CMP.L       D7, D4              ;
                BLE.S       notMax4             ;
                MOVE.L      D4, D7              ;

notMax4         SUBA.L      #2, A2              ;
                ADDA.W      #2, A3              ;
                SUBQ.L      #1, D2              ;
                BGT.S       loop2               ; done checking at last!
lg
                SUB.L       D6, D7              ; D7: range
                BNE.S       findScale           ;
                MOVEQ.L     #1, D7              ;

findScale       MOVE.W      scaleType(A6), D4   ;
                BLE.S       linMap1             ; scaleType <= 0 -> linear Mapping
                CMP.W       #10, D4             ;
                BGE.S       logMap1             ; scaleType >= 10 -> log Mapping

rootMap1        FMOVE.L     #1, FP2             ; 0 < scaleType < 10 -> nth root mapping
                FDIV.W      D4, FP2             ; FP2: 1/n
                FLOGN.L     D7, FP0             ; FP0: ln(range)
                FMUL.X      FP2, FP0            ; FP0: 1/n * ln(range)
                FETOX.X     FP0, FP0            ; FP0: e^(1/n * ln(range)) = range^(1/n)
                MOVE.W      #2, D5              ; D5: ScaleType = 2 -> nth root
                BRA.S       GoOn1               ;

linMap1         FMOVE.L     D7, FP0             ;
                MOVE.W      #1, D5              ; D5: ScaleType = 1 -> linear
                BRA.S       GoOn1               ;

logMap1         SUBQ.L      #1, D6              ; min := min - 1 ; to prevent log(0)
                ADDQ.L      #1, D7              ; which increases range by 1
                FLOG2.L     D7, FP0             ;
                MOVE.W      #3, D5              ; D5: ScaleType = 3 -> log

GoOn1           FMOVE.L     #255, FP1           ;
                FDIV.X      FP0, FP1            ; FP1: mul factor
                MOVE.L      D5, D7              ; D7: Copy of ScaleType
                ; D6: minVal

doPS            MOVE.L      destPtr(A6), A1     ; compute body of PS
                MOVE.L      (SP), D2            ;
                MOVEA.L     D2, A2              ;
                ADDA.L      D2, A2              ;
                SUBA.L      #2, A2              ; A2: longOffset

                MOVE.W      D0, D3              ;
                EXT.L       D3                  ;
                MOVEA.L     D3, A3              ;
                ADDA.L      D3, A3              ;
                ADDA.W      #2, A3              ; A3: shortOffset

                MOVE        D0, D3              ;
                EXT.L       D3                  ;
                ASR.L       #1, D3              ; D3: rowWords div 2
                ASR.L       #1, D2              ; D2: rowWords^2 div 2
                SUB.L       D3, D2              ; D2: count

loop3           MOVE.L      A3, D3              ;
                MOVE.L      D3, D4              ;
                ASR.L       #1, D4              ;
                AND.L       D1, D4              ; mod rowWords
                BNE.S       noMod2              ;
                MOVE.W      D0, D4              ;
                EXT.L       D4                  ;
                ADD.L       D4, D4              ;
                SUB.L       D4, D3              ;
noMod2          MOVE.W      0(A0, D3.L), D4     ;
                MOVE.W      0(A0, A2.L), D5     ;
                MULS.W      D4, D4              ;
                MULS.W      D5, D5              ;
                ADD.L       D5, D4              ; D4: x^2 + y^2
                SUB.L       D6, D4              ; D4: " - minVal
                MOVE.W      D7, D5              ; D5: scaleType
                SUBQ.W      #1, D5              ;
                BEQ.S       linMap2             ;
                SUBQ.W      #1, D5              ;
                BEQ.S       rootMap2            ;
```

```
logMap2         FLOG2.L    D4, FP0            ; take log
                FMUL.X     FP1, FP0           ; scale up to [0..255]
                BRA.S      GoOn2              ;

linMap2         FMOVE.L    D4, FP0            ;
                FMUL.X     FP1, FP0           ;
                BRA.S      GoOn2              ;

rootMap2        FLOGN.L    D4, FP0            ;
                FMUL.X     FP2, FP0           ;
                FETOX.X    FP0, FP0           ;
                FMUL.X     FP1, FP0           ;

GoOn2           FMOVE.L    FP0, D4            ; D4: scaled out: should be in [0,255]
                MOVE.L     D3, D5             ;
                ASR.L      #1, D5             ;
                MOVE.B     D4, 0(A1, D5.L)    ; !!!doesn't work on 68000
                MOVE.L     A2, D5             ;
                ASR.L      #1, D5             ;
                MOVE.B     D4, 0(A1, D5.L)    ; stored result bytes

                SUBQ.L     #2, A2             ;
                ADDQ.L     #2, A3             ;
                SUBQ.L     #1, D2             ;
                BGT.S      loop3              ;

                MOVE.L     (SP)+, D3          ; compute middle element, 1st column
                MOVE.W     0(A0, D3.L), D4    ;
                MULS.W     D4, D4             ; D4: (x^2 + y^2) div 2
                ADD.L      D4, D4             ;
                SUB.L      D6, D4             ; D4: " - minVal
                MOVE.W     D7, D5             ; D6: scaleType
                SUBQ.W     #1, D5             ;
                BEQ.S      linMap3            ;
                SUBQ.W     #1, D5             ;
                BEQ.S      rootMap3           ;

logMap3         FLOG2.L    D4, FP0            ; take log
                FMUL.X     FP1, FP0           ; scale up to [0..255]
                BRA.S      GoOn3              ;

linMap3         FMOVE.L    D4, FP0            ;
                FMUL.X     FP1, FP0           ;
                BRA.S      GoOn3              ;

rootMap3        FLOGN.L    D4, FP0            ;
                FMUL.X     FP2, FP0           ;
                FETOX.X    FP0, FP0           ;
                FMUL.X     FP1, FP0           ;

GoOn3           FMOVE.L    FP0, D4            ; D4: scaled out: should be in [0,255]
                MOVE.L     D3, D5             ;
                ASR.L      #1, D5             ;
                MOVE.B     D4, 0(A1, D5.L)    ; !!!doesn't work on 68000

                MOVE.W     (A0), D4           ; compute 0th matrix element
                MULS.W     D4, D4             ; D4: (x^2 + y^2) div 2
                ADD.L      D4, D4
                SUB.L      D6, D4             ; D4: " - minVal
                MOVE.W     D7, D5             ; D6: scaleType
                SUBQ.W     #1, D5             ;
                BEQ.S      linMap4            ;
                SUBQ.W     #1, D5             ;
                BEQ.S      rootMap4           ;

logMap4         FLOG2.L    D4, FP0            ; take log
                FMUL.X     FP1, FP0           ; scale up to [0..255]
                BRA.S      GoOn4              ;

linMap4         FMOVE.L    D4, FP0            ;
                FMUL.X     FP1, FP0           ;
                BRA.S      GoOn4              ;

rootMap4        FLOGN.L    D4, FP0            ;
                FMUL.X     FP2, FP0           ;
                FETOX.X    FP0, FP0           ;
                FMUL.X     FP1, FP0           ;

GoOn4           FMOVE.L    FP0, D4            ; D4: scaled out: should be in [0,255]
                MOVE.B     D4, (A1)           ; !!!doesn't work on 68000

                ; now compute top Row PS
                MOVE.L     #2, A3             ; A3: shortOffset
                MOVE.W     D0, D3             ;
                EXT.L      D3                 ;
                ADD.L      D3, D3             ;
                SUBQ.L     #2, D3             ;
                MOVE.L     D3, A2             ; A2: longOffset
```

```
                MOVE.L      D0, D2              ;
                SUBQ.L      #1, D2              ; D2: counter

loop4           MOVE.W      0(A0, A2.L), D4     ;
                MOVE.W      0(A0, A3.L), D5     ;
                MULS.W      D4, D4              ;
                MULS.W      D5, D5              ;
                ADD.L       D5, D4              ;
                SUB.L       D6, D4              ; D4: " - minVal
                MOVE.W      D7, D5              ; D6: scaleType
                SUBQ.W      #1, D5              ;
                BEQ.S       linMap5             ;
                SUBQ.W      #1, D5              ;
                BEQ.S       rootMap5            ;

logMap5         FLOG2.L     D4, FP0             ; take log
                FMUL.X      FP1, FP0            ; scale up to [0..255]
                BRA.S       GoOn5               ;

linMap5         FMOVE.L     D4, FP0             ;
                FMUL.X      FP1, FP0            ;
                BRA.S       GoOn5               ;

rootMap5        FLOGN.L     D4, FP0             ;
                FMUL.X      FP2, FP0            ;
                FETOX.X     FP0, FP0            ;
                FMUL.X      FP1, FP0            ;

GoOn5           FMOVE.L     FP0, D4             ; D4: scaled out: should be in [0,255]
                MOVE.L      D3, D5              ;
                ASR.L       #1, D5              ;
                MOVE.B      D4, 0(A1, D5.L)     ; !!!doesn't work on 68000
                MOVE.L      A2, D5              ;
                ASR.L       #1, D5              ;
                MOVE.B      D4, 0(A1, D5.L)     ; stored result bytes

                SUBA.L      #2, A2              ;
                ADDA.W      #2, A3              ;
                SUBQ.L      #1, D2              ;
                BGT.S       loop4               ; done at last!

exit            FMOVE.X     (SP)+, FP2          ; restore FP2
                MOVEM.L     (SP)+, UsedRegs     ;
                UNLK        A6                  ;
                MOVE.L      (SP)+, A0           ;
                ADD.L       #12, SP             ;
                JMP         (A0)                ;

                DC.B        'PSFHT2D '          ; MacsBug Name

                ENDPROC

                END
```

BIBLIOGRAPHY

[1] Bracewell, R. N., *The Fourier Transform,* Scientific American, June 1989.

[2] Cooley, J. W., and Tukey, J. W., *An Algorithm for the machine calculation of complex Fourier series*, Mathematics of Computation, 1965.

[3] Danielson, G.C., and C. Lanczos, *Some Improvements in Practical Fourier Analysis and Their Application to X-Ray Scattering from Liquids*, J. Franklin Institute, Vol. 233, 1942.

[4] Brigham, E. O., *The Fast Fourier Transform*, Prentice-Hall, 1974.

[5] Artal, P., Avalos-Borja, M., Soria, A., Poppa, H., & Heinemann, K., *Image Processing Enhancement of High-Resolution TEM Micrographs of Nanometer-Size Metal Particles*, Ultramicroscopy, July/August 1989.

[6] Rasband, W., *Image 1.25* (Documentation), National Institues of Health, February 1990.

[7] Silverman, H. F., *An Introduction to Programming the Winograd Fourier Transform Algorithm (WFTA)*, IEEE Trans. ASSP, April 1977.

[8] Patterson, R. W. & McClellan, J. H. *Fixed Point Error Analysis of Winograd Fourier Transform Algorithms*, IEEE Trans. ASSP, October 1978.

[9] Harris, D. B. McClellan, J. H., Chan, D. H. K. & Schuessler, H. W., *Vector Radix Fast Fourier Transform*, IEEE ICASSP, 1977.

[10] Dudgeon, D. E., & Mersereau, R. M., *Multidimensional Digital Signal Processing*, Prentice-Hall, 1984.

[11] Gibson, R. M., & McCabe, D. P., *Fourier Transform Algorithm Implementations on a General Purpose Microprocessor*, IEEE ICASSP, Vol. 2., 1981.

[12] Despain, A. M., *Very Fast Fourier Transform Algorithms Hardware for Implementation*, IEEE Trans. Computer, C-28, May 1979.

[13] Sorensen, H. V., Jones, D. L., Heideman, M. T., & Burrus, C. S., *Real Valued Fast Fourier Transform Algorithms*, IEEE Trans. ASSP, Vol. 35, No. 6, June 1987.

[14] Press, W. H., Flannery, B. P., Teukolsky, S. A., & Vetterling, W. T., *Numerical Recipes*, Cambridge University Press, 1986.

[15] Bracewell, R. N., *The Hartley Transform*, Oxford University Press, 1986.

[16] Hartley, R. V. L., *A more symmetrical Fourier analysis applied to transmission problems,* Proc. IRE, March 1942.

[17] Bracewell, R. N., *The Fast Hartley Transform*, Proc. IEEE. Vol. 72, No. 8, August 1984.

[18] Sorensen, H. K., Jones, D. L, Burrus, C. S. & Heideman, M. T., *On Computing the Discrete Hartley Transform*, IEEE Trans. ASSP, Vol. 33, No. 4, October 1985.

[19] Hecht, E. & Zajac, A., *Optics*, Addison-Wesley, 1979.

[20] Evans, David M. W., *An Improved Digit Reversal Permutation Algorithm for the Fast Fourier and Hartley Transforms*, IEEE Trans. ASSP, Vol. 35, No. 8, August 1987.

[21] Le-Ngoc, T., & Vo, M.T., *Implementation and Performance of the Fast Hartley Transform*, IEEE Micro, October 1989.

[22] Kwong, C. P., & Shiu, K. P., *Structured Fast Hartley Transform Algorithms*, IEEE Trans. ASSP, Vol. 34, No. 4, August 1986.

[23] Knaster, Scott, *How to Write Macintosh Software, The Debugging Reference for Macintosh*, Second Ed., Hayden Book Co., 1988.

[24] _____, Analog Devices ADSP 2100 Applications Handbook and Users Manual. Analog Devices Signal Processing Division, 1988.

[25] Oppenheim, A. V., & Schafer, R. W., *Digital Signal Processing*, Prentice-Hall, 1975.

[26] Kabal, P. & Sayar, B., *Performance of Fixed-Point FFT's: Rounding and Scaling Considerations*, IEEE ICASSP, 1986.

[27] Bracewell, R. N., Buneman, O., Hao, H. & Villasenor, J., *Fast Two-Dimensional Hartley Transform*, Proc. IEEE. Vol. 74, No. 9, September 1986.

[28] _____, Inside Macintosh, Vol 1 - 5, Addison Wesley, 1985.

[29] Eiserling, F. A., *Structure of T4 virion*, from Bacteriophage T4, Edited by C. Matthew, E. Kutter, G. Mosig & P. Berget, American Society for Microbiology, Washington, DC, 1983.

[30] Pei, S. C., & Wu, J. L., *Split-Radix Fast Hartley Transform*, Electronics Letters, January 1986.

[31] Pei, S. C., & Wu, J. L., *Split Vector Radix 2-D Fast Fourier Transform*, IEEE Trans. Circuits and Systems, August 1987.

[32] Kumaresan, R. & Gupta, P. K., *Vector Radix Algorithm for a 2-D Discrete Hartley Transform*, Proc. IEEE, Vol. 74, No. 5, May 1986.

Additional Reading

Chamberlin, H., *Musical Applications of Microprocessors*, Hayden Book Co., 1985.

Lord, R. H., *Fast Fourier for the 6800*, BYTE, February 1989.

O'Neill, M. A., *Faster than Fast Fourier*, BYTE, April 1988.

Paik, C. H. & Martin D. Fox, *Fast Hartley Transforms for Image Processing*, IEEE Trans. Medical Imaging, Vol. 7, No. 2, June 1988.

Rivard, G. E., Direct Fast Fourier Transform of Bivariate Functions, IEEE Trans. ASSP, Vol. 25, No. 3, June 1977.

Said, S. M. & K. R. Dimond, *Improved Implementation of FFT Algorithm on a High-Performance Processor*, Electronics Letters, Vol. 20, No. 8, 12 April 1984.

Stigall, P. D., Ziemer, R. E. & Hudec, L., *A Performance Study of 16-bit Microcomputer-implemented FFT Algorithms*, IEEE Micro, November 1982.

Zhi-Jian, M. & Duhamel, P., *In-Place Butterfly-Style FFT of 2-D Real Sequences*, IEEE Trans. ASSP Vol. 36, No. 10, October 1988.