Application Note 16

Implementing Fast Fourier Transforms on the ARM RISC Processor

Document Number: ARM DAI 0016C
Issued: January 1996

ARM POWERED ™

ARM
Advanced RISC Machines

Proprietary Notice

Change Log

Issue	Date	By	Change
A	June 94	PE/SG	Document creation
B	Dec 94	PB	Reformatted into FrameMaker
C	Jan 96	AP	Major edits

1 Introduction

The Fast Fourier Transform, known as the FFT, is an important algorithm used in digital signal processing. It converts a sampled signal, such as an audio or other waveform captured by an analog to digital converter, into a frequency spectrum. The Inverse Fourier Transform reverses the process.

This application note describes the implementation of the FFT using the ARM processor. How to call the algorithm from C or assembler is described in **2 Calling the algorithm** on page 2. The performance statistics are given in **3 Performance** on page 5. The mathematical theory concerning how the algorithm works is described in **A How the Algorithm Works** on page 11.

The implementation makes use of many ARM features to maximize performance including the following:

- The load and store multiple instructions greatly improve the handling of transfers between main memory and the general purpose registers.
- The combined arithmetic/logical and shift operations in a single instruction reduce the number of cycles needed for important computational parts of the program.
- Conditional execution of all instructions reduces the number of branches needed.
- The large number of general purpose registers is used to reduce the number of loads and stores to a minimum.

A number of optimizations are implemented, and conditional assembly is used to make them optional.

2 Calling the algorithm

2.1 Input and output

The algorithm takes four values on input:

- A pointer *x* to the input data consisting of *N* complex numbers *x[0], ..., x[N-1]*. Each complex number consists of a 32-bit integer containing the real part followed by a 32-bit integer containing the imaginary part and so is 8 bytes long.
- A pointer *y* to the output buffer for the transformed array to be stored. This buffer must be at least 8**N* bytes (the same size as the input *x* buffer).
- An integer *LogN* giving the base 2 logarithm of the number of points *N* in the fourier tranform.
- A flag *direction* which is either:

1	for a forward transform
0	for an inverse transform

On output, the algorithm returns an integer:

zero	if the transform was successful (in which case the *y* array contains the answer)
nonzero	if the transform was not successful (for example the trigonometry table was not large enough for the given size of *N*)

The forward FFT performs the calculation specified by the formula:

$$y[k] = \frac{1}{N}\sum_{n=0}^{N-1} x[n]\,(\cos(nk\omega) - i\sin(nk\omega))\,,$$

where $\omega = 2\pi/N$. The *1/N* scaling in front prevents any overflows within the algorithm. We will write $y = FFT(x)$ for the forward transform.

The Inverse transform performs the calculation:

$$y[k] = \frac{1}{N}\sum_{n=0}^{N-1} x[n]\,(\cos(nk\omega) + i\sin(nk\omega))\,,$$

which we will write as $y = IFT(x)$.

2.2 Calling the algorithm from C

To call the algorithm from C, include the header `fft.h` which declares the type *complex* and the function FFT as:

```
int FFT(complex *x, complex *y, int LogN, int direction);
```

Call the algorithm with *x, y, LogN, direction* set up as described above. Note the following points:

- The buffers *x* and *y* may coincide. In this case the FFT is said to be done *in place* rather than *out of place*.
- To prevent overflow within the algorithm, the real and imaginary values in the array x should be sign extended 16-bit quantities (between -32768 and +32767).
- When using the optimized version of the FFT, the smallest value of N allowed is 16 (LogN must be at least 4). N must always be a power of 2. The largest value of N allowed depends on the size of the trigonometry table included when the FFT code is assembled.

2.3 Calling the algorithm from assembler

To call the algorithm from assembler, import the symbol FFT and set up the registers as follows:

- set R0 to the address of the input buffer *x*
- set R1 to the address of the output buffer *y*
- set R2 to *LogN*
- set R3 to *direction* (0 or 1)
- R13 must point to a full descending stack

Call the algorithm via BL FFT. On exit R0 will contain the exit code (0 if successful), R1-R3 and R14 will have been corrupted and R4-R13 preserved.

See **2.2 Calling the algorithm from C** on page 2 for the restrictions on *LogN* and the contents of the *x* array.

2.4 Performing real FFTs

If the values placed in *x* are all real (zero imaginary part), the speed of the forward FFT can be almost doubled. The function `REALFFT` performs this and is declared in `fft.h` as:

```
int REALFFT(int *input, complex *output, int LogN);
```

The REALFFT function calculates the FFT of 2*N* real values using an *N* point complex FFT. The input buffer should be filled with the real parts of *x[0],...., x[2N-1]* (thus the size of the buffer is 8**N* bytes) and on output the output buffer will contain the first half of the FFT *y[0],..., y[N-1]*. The second half of the transform can be calculated by symmetry, for in the real case *y[2N-k]* is the complex conjugate of *y[k]*.

Note that the input and output buffers are the same size (8**N* bytes) and may coincide. The same restrictions on the size of *LogN* and the contents of *x* apply as for the FFT algorithm, with the addition that the trigonometry table must be at least size 2*N*.

2.5 Changing the size of the trigonometry table

The trigonometry table `fft_table.s` is generated by the C program `trig.c` provided. The syntax is:

```
trig [N [path]]
```

where *path* is the directory to place the file `fft_table.s`. For example:

```
trig 1024 /tmp/
```

will generate the table for 1024 point FFTs in `/tmp`. The FFT source code should now be recompiled so that it includes the new table. The labels TABLE_N and TABLE_LOGN at the start of the table tell the FFT code its size.

3 Performance

The FFT code contains two separate algorithms, one optimized for size and the other for speed. The `OPTIMISE` flag in the assembler source determines which is compiled. Use the following settings:

0	for the small, slower algorithm
1	for the large, highly optimized algorithm

3.1 The optimized algorithm

We start by listing the performance for the speed optimized algorithm. Timings are given for the ARM7 and ARM7M running at 40MHz. Note that the timings do *not* depend on the data passed to the FFT. In general, although the ARM multiply instruction takes a variable length of time, depending on the size of the numbers being multiplied, the fixed cos and sin values determine the number of cycles each multiply takes.

	40 MHz ARM7		**40 MHz ARM7M**	
Size of FFT N	Time for one FFT	FFTs per second	Time for one FFT	FFTs per second
16	0.028ms	36,000	0.026ms	38,000
32	0.080ms	12,500	0.069ms	14,500
64	0.20ms	4,700	0.17ms	5,700
128	0.52ms	1,900	0.41ms	2,400
256	1.24ms	800	1.00ms	1,040
512	2.9ms	340	2.2ms	450
1024	6.6ms	150	4.9ms	200
2048	15ms	67	11ms	89
4096	33ms	30	25ms	41
8192	73ms	14	53ms	19
16384	160ms	6.3	115ms	8.7

Table 1: Optimized timings

The timings are for *out of place* FFTs. The *in place* FFTs (where the input and output buffers are equal) will be slightly quicker for slow memory.

The code size for the optimized algorithm is 1592+*N*/2 bytes (including the lookup table) and the data size is 64 bytes (not including the input and output buffers).

3.2 The unoptimized algorithm

The unoptimized algorithm has a smaller code size of 548+*N*/2 bytes and data size of 64 bytes. **Table 2: Unoptimized timings** on page 6 lists the performance figures and shows that, on average, the unoptimized algorithm takes approximately 1.5 times as long as the optimized version for large FFTs and twice as long for smaller FFTs.

	40 MHz ARM7		**40 MHz ARM7M**	
Size of FFT N	Time for one FFT	FFTs per second	Time for one FFT	FFTs per second
16	0.071ms	14,000	0.057ms	17,500
32	0.17ms	5,900	0.13ms	7,600
64	0.39ms	2,500	0.29ms	3,400
128	0.90ms	1,100	0.65ms	1,500
256	2.0ms	500	1.4ms	700
512	4.5ms	220	3.1ms	320
1024	9.9ms	100	6.8ms	150
2048	22ms	46	15ms	68
4096	47ms	21	3ms	32
8192	100ms	10	67ms	15
16384	220ms	4.6	140ms	7.0

Table 2: Unoptimized timings

The timings are for *out of place* FFTs. **Figure 1: Timing data summary** on page 7 summarizes the timing data using logrithmic axes.

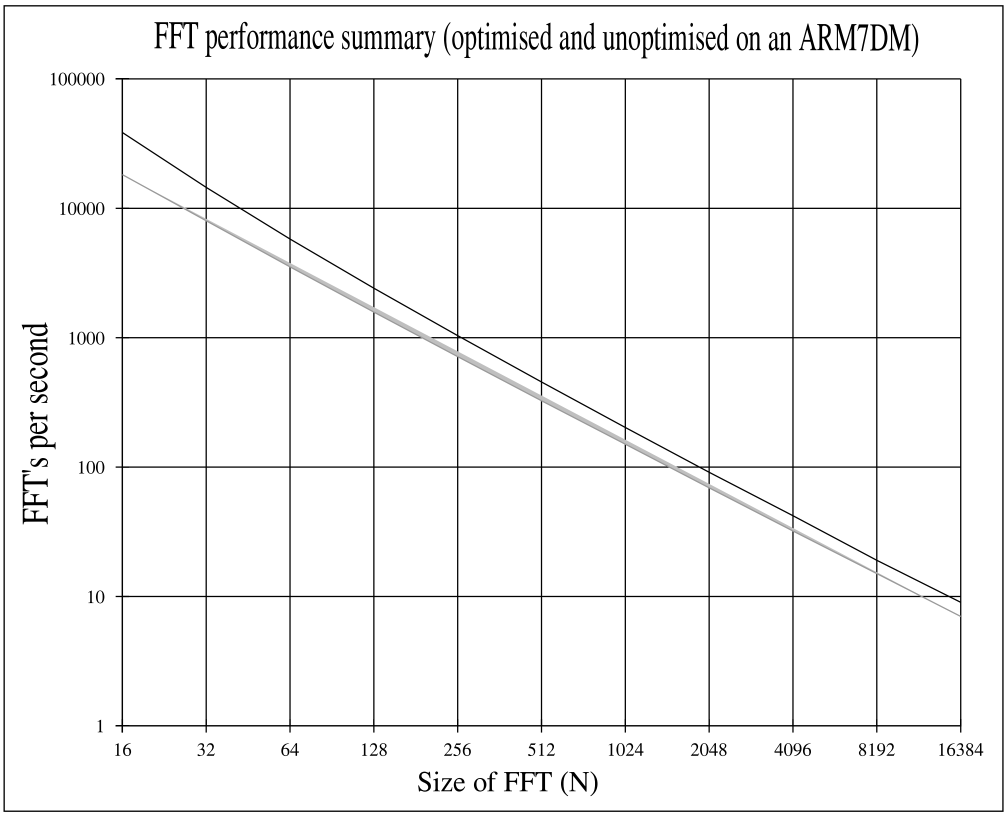

Figure 1: Timing data summary

4 Applications

4.1 Spectral analysis

Given a digitized signal, it is often useful to know the magnitude of each frequency within the signal—the spectral components. The FFT determines these components up to a frequency of half the sample rate. The fineness of the frequency resolution depends on the number of points in the FFT.

Suppose that *s[0], s[1], s[2], ... , s[N-1]* is a frame of *N* input samples, sampled at a frequency *f*. Set $x[k] = s[k] + 0i$ and calculate $y = FFT[x]$ (using the REALFFT function will be fastest). For *k=0, 1, 2, ... N/2-1,* $|y[k]|$ is the magnitude of the spectral component with frequency *fk/N*.

In order to pick out frequency bands of interest to an application, multirate techniques can be used to isolate and translate the frequency band before applying the FFT. See [1] for more details.

4.2 Digital filtering

Once a digital sample has been transformed to the frequency domain (as for spectral analysis) a filter may be applied to remove a range of frequency components. The inverse transform will reconstruct the signal. Due to the *1/N* scaling in the calculation of the FFT, the inverse transform will produce a signal *1/N* times the strength of the original. The inversion formula may be stated:

$$IFT[FFT[x]] = \frac{1}{N}x.$$

In practice, windowing techniques are required to prevent clicks at frame boundaries.

4.3 Calculation of correlations

Filtering using a traditional FIR filter requires the calculation of large correlations. FFTs may be used to speed up these calculations by exploiting the correlation theorem below. If *x* is the input data and *c* the vector of coefficients, the cyclic correlation of *x* and *c* is defined to be:

$$r[k] = \frac{1}{N}\sum_{n=0}^{N-1} x[k+n]\,c[n],$$

where x is assumed to be periodic with period *N*. If *X* and *C* are the Fourier transforms of *x* and *c* respectively, and $Z[k] = X[k]\overline{C[k]}$, the correlation theorem may be stated:

$$IFT[Z] = \frac{1}{N}r.$$

In practice this method is only faster for a large number of coefficients (>128 for example).

4.4 Calculation of convolutions

Another common calculation in the construction of filters is the convolution. The convolution of *x* and *c* is defined as:

$$v[k] = \frac{1}{N}\sum_{n=0}^{N-1} x[k-n]\,c[n] .$$

The Convolution theorem is similar to the correlation theorem:

$$IFT[Z] = \frac{1}{N}v ,$$

where this time $Z[k] = X[k]\,C[k]$.

4.5 Useful identities

If $X = FFT[x]$, and the input data *x* is real, then the following symmetry identities are useful in applications. In particular they show that the second half of the transform is determined by the first half.

$$Re[X[N-k]] = Re[X[k]] ,$$
$$Im[X[N-k]] = -Im[X[k]] .$$

4.6 References

[1] Digital Signal Processing, A Practical Approach, Emmanuel C. Ifeachor & Barrie W. Jervis, Addison-Wesley 1993 (ISBN 0-201-54413-X).

[2] The Fast Fourier Transform and its Applications, E. Oran Brigham, Prentice-Hall International Editions, 1988 (ISBN 0-13-307547-8).

A How the Algorithm Works

This appendix describes the internal workings of the ARM code routine and the mathematics behind it. None of this information is required to use the algorithm so this section may be skipped. Throughout, *x* denotes the vector to be transformed and $X = FFT[x]$.

A.1 The mathematical theory

Recalling the complex identity $\exp(i\theta) = \cos\theta + i\sin\theta$, the transform may be written:

$$X[k] = \frac{1}{N}\sum_{n=0}^{N-1} x[n]\exp(-ink\omega)$$

where $\omega = 2\pi/N$. Evaluating this sum directly will take N^2 multiplications. With rearrangements this may be reduced to $N \times \log N$.

First separate out the even and odd numbered elements:

$$X[k] = \frac{1}{N}\left(\sum_{n=0}^{N/2-1} x[2n]\exp(-2nik\omega) + \sum_{n=0}^{N/2-1} x[2n+1]\exp(-(2n+1)ik\omega)\right),$$

then take out a factor of $\exp(-ik\omega)$,

$$X[k] = \frac{1}{N}\left(\sum_{n=0}^{N/2-1} x[2n]\exp(-2nik\omega) + \exp(-ik\omega)\sum_{n=0}^{N/2-1} x[2n+1]\exp(-2nik\omega)\right).$$

Inspection shows that if we put Y=FFT(x[0], x[2], x[4], ... ,x[N-2]) and Z=FFT(x[1], x[3], x[5], ... , x[N-1]) then,

$$X[k] = \frac{1}{2}(Y[k] + \exp(-ik\omega)Z[k]) \qquad \text{if } 0 \le k < N/2,$$

$$X[k] = \frac{1}{2}(Y[k-N/2] - \exp(-ik\omega)Z[k-N/2]) \qquad \text{if } N/2 \le k < N.$$

The problem has been reduced to calculating two N/2 size FFTs and performing N complex multiplications. Note that the bottom bit of *k* determines whether *x[k]* is in the FFT calculation for *Y* or for *Z*. By repeating this process for *Y* and *Z* and recursing we derive the FFT algorithm of the next subsection. This process is known as *decimation in time*.

A.2 The FFT algorithm

Let x[0], x[1], ... x[N-1] be the input data and X[0], X[1], ... , X[N-1] the output buffer. The algorithm separates naturally into two stages:

STAGE 1: Bit reversal

Copy the data from the input buffer to the output buffer in bit-reversed order. If $k = (b_{N-1}\ldots b_2 b_1 b_0)_2$ is an integer between 0 and N-1, with binary representation as shown, for the bit reversal of k, we will write $\hat{k} = (b_0\ldots b_{N-3} b_{N-2} b_{N-1})_2$. So we perform the equivalent of the following loop:

```
int k,khat,bit;
for (k=0, khat=0; k<N; k++) {
      X[khat]=x[k];
      for (bit=N/2; (khat & bit)!=0; bit >>=1) khat ^= bit;
      khat ^= bit; /* finish incrementing khat */
      }
```

The bit loop inside increments `khat` as a bit reversed number. If the input buffers and output buffers are equal (X=x), more care needs to be taken to do the action in place, but it can be performed faster as fewer elements need to be moved. Putting the elements in bit reversed order has the effect of grouping together all the *Y*s and *Z*s of the previous subsection so no further rearrangement needs to be done.

STAGE 2: Iterate

Perform the equivalent of the following loop:

* denotes complex multiplication

```
# define pi 3.14159265358
for (n=2; n<=N; n <<= 1) {
      w=2*pi/n;
      for (m=0; m<N; m+=n) {
         for (k=0; k<n/2; k++) {
            y=X[m+k];
            z=X[m+k+n/2] * exp(-ikw);
            X[m+k]=(y+z)/2;
            X[m+k+n/2]=(y-z)/2;
         }
      }
}
```

After n=2 we have performed N/2 two-element FFTs, positioned at offsets m=0, 2, 4, ... N-2. After n=4 we have performed N/4 four-element FFTs positioned at offset m=0, 4, 8, ... N-4. After n=N we are left with the answer to the main FFT in the buffer *X*.

A.3 ARM implementation

The ARM implementation of the bit reversal stage is straightforward and contained between the labels FFT and FFTSTART. The second stage begins at the label FFTSTART. For the unoptimized algorithm, the calculation is as described in **A.2 The FFT algorithm** on page 11, except that the m and k loops are swapped over. This is done since $\exp(-ik\omega)$ does not depend on m. The code becomes:

```
# define pi 3.14159265358
for (n=2; n<=N; n <<= 1) {
      w=2*pi/n;
      for (k=0; k<n/2; k++) {
         c=cos(kw); s=sin(kw);
         for (m=0; m<N; m+=n) {
            y=X[m+k];
            z=X[m+k+n/2] * (c-is);
            X[m+k]=(y+z)/2;
            X[m+k+n/2]=(y-z)/2;
         }
      }
}
```

A sin/cos lookup table is used to calculate $\exp(-ik\omega) = \cos(k\omega) - i\sin(k\omega)$. Angles are stored in registers by expressing them as multiples of $2\pi/N$, which is the basic step size of the table. The lookup table only contains N/8+1 entries, describing the cos/sin values for angles between 0 and $\pi/4$ (inclusive). The values for other angles can be derived using the symmetry of the circle.

Register names can be identified with the pseudo-C code above in the following way:

GrpStage	n/2	number of times around the k loop to go
CalcsLoop	N/n	number of times around the m loop to go
Tab	kN/n	current angle kw in units of $2\pi/N$
StartElem	k	offset of the first element of the m loop
Yr, Yi		real and imaginary parts of Y
Zr, Zi		real and imaginary parts of Z
Yadr		address of next Y element = X + 8*(m+k)
Zadr		address of next Z element = X + 8*(m+k+n/2)
Cos, Sin		Cos(wk), Sin(wk) shifted left 14 bits

For the optimized algorithm, the first three stages k=0,1,2 are performed at the same time, in eight-element chunks. For these values of k, cos(kw) and sin(wk) take on simple values such as 0, 1 and sqr(2)/2. Multiplies by these constants can be done using shifts and adds. The MUL instruction is not needed for the first three stages.

The main loop of the optimized algorithm is unrolled to the extent that four m loops are performed within each k loop and k only varies between 0 and n/8. To illustrate exactly how this works, let us abbreviate the m loop to a function declared as m_loop(c,s) where c and s are the cos and sin values. Unoptmized, the k-loop is written:

```
for (k=0; k<n/2; k++) m_loop(cos(wk), sin(wk));
```

After optimization, the code becomes (the bracketed strings denote the label used in the ARM assembler for this loop):

```
m_loop(1,0) /* perform k=0 (EASYGROUP0 loop) */
m_loop(sqr(2)/2, sqr(2)/2) /* perform k=n/8 (EASYGROUP1)*/
m_loop(0,1) /* perform k=n/4 (EASYGROUP2) */
m_loop(-sqr(2)/2,sqr(2)/2)) /* perform k=3n/8 (EASYGROUP3)*/
for (knew=1; knew<n/8; knew++) {
      c=cos(knew*w); s=sin(knew*w);
      m_loop(c,s); /* perform k=knew (CALCLOOP1) */
      m_loop(s,c); /* perform k=n/4-knew (CALCLOOP2) */
      m_loop(-s,c); /* perform k=n/4+knew (CALCLOOP3) */
      m_loop(-c,s); /* perform k=n/2-knew (CALCLOOP4) */
}
```

There are two main advantages to writing the loop in this way.

- only one pair of cos/sin values needs to be looked up for all four inner m loops.
- since c and s are always positive, the mutiplies in the m_loop can be arranged so that the MUL instructions always multiply by a positive number, minimizing the number of cycles the MUL takes. Note that the above scheme only works if n>=8 and so cannot be used for the first two stages.

A.4 The inverse algorithm

To calculate the inverse FFT, the data is conjugated before and after passing it through the forward FFT algorithm. Conjugation is performed at the same time as bit reversal before the algorithm. An alternative approach, which would be slightly faster, is to duplicate the forward FFT code, but multiply by *c+is* in the inner loop instead of *c-is*. This approach has not been taken as the speed benefits are minimal and it would double the code size.

ENGLAND
Advanced RISC Machines Limited
Fulbourn Road
Cherry Hinton
Cambridge CB1 4JN
England
Telephone:+44 1223 400400
Facsimile:+44 1223 400410
Email:info@armltd.co.uk

GERMANY
Advanced RISC Machines Limited
Otto-Hahn Str. 13b
85521 Ottobrunn-Riemerling
Munich
Germany
Telephone:+49 (0) 89 608 75545
Facsimile:+49 (0) 89 608 75599
Email:info@armltd.co.uk

JAPAN
Advanced RISC Machines K.K.
KSP West Bldg, 3F 300D, 3-2-1 Sakado
Takatsu-ku, Kawasaki-shi
Kanagawa
213 Japan
Telephone:+81 44 850 1301
Facsimile:+81 44 850 1308
Email:info@armltd.co.uk

USA
ARM USA Incorporated
Suite 5
985 University Avenue
Los Gatos
CA 95030 USA
Telephone:+1 408 399 5199
Facsimile:+1 408 399 8854
Email:info@arm.com

World Wide Web Address: http://www.arm.com/