Digital Signal Processing Libraries Using the ColdFire® eMAC and MAC
User’s Manual

by: Oleksandr Marchenko
       Antoly Khaynakov
       Andriy Tymkiv
       Igor Drozdinsky
       Dmitry Karpenko
       Artem Kovalev
       Michael Zinovjev
       Oleksandr Romaniuk
       Denis Ivanov
       Luis Reynoso
       Jaime Herrero

To provide the most up-to-date information, the revision of our documents on the World Wide Web will be the most current. Your printed copy may be an earlier revision. To verify that you have the latest information available, refer to:

http://www.freescale.com
The following revision history table summarizes changes contained in this document. For your convenience, the page number designators have been linked to the appropriate location.

**Revision History**

<table>
<thead>
<tr>
<th>Date</th>
<th>Revision Level</th>
<th>Description</th>
<th>Page Number(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>02/2006</td>
<td>1.2</td>
<td>Initial release</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Purpose**

This document provides a library of macros designed to ensure efficient programming of the ColdFire processor using MAC or eMAC modules.

**References**

The following documents were referenced to build this document:

1. *ColdFire Family Programmer's Reference*, Rev. 3
3. *MCF5282 ColdFire User’s Manual*, Rev. 2.3

**Definitions, Acronyms, and Abbreviations**

The following terms appear frequently in this manual:

- **DSP**  : Digital Signal Processor
- **FFT**  : Fast-Fourier Transform
- **FIR**  : Finite Impulse Filter
- **IIR**  : Infinite Impulse Filter
- **MAC**  : Multiply-and-Accumulate
- **eMAC** : Enhanced Multiply-and-Accumulate
# Table of Contents

## Chapter 1
### Overview

1.1 Introduction ............................................................................................................... 1-1
1.2 General Description of a Complex FFT Calculation .................................................. 1-1
1.3 General Filters Description ...................................................................................... 1-2
1.4 Library Structure .................................................................................................. 1-3
1.5 Getting Started .................................................................................................... 1-3
1.6 Definitions Used in Library .................................................................................... 1-4
   1.6.1 Data Types ....................................................................................................... 1-4
   1.6.2 Macros .............................................................................................................. 1-4

## Chapter 2
### Fast Fourier Transform (FFT)

2.1 Introduction ............................................................................................................ 2-1
2.2 FFT16 Functions ..................................................................................................... 2-1
   2.2.1 FFT16 ............................................................................................................. 2-1
      2.2.1.1 Call(s) ....................................................................................................... 2-1
      2.2.1.2 Parameters ............................................................................................... 2-1
      2.2.1.3 Returns .................................................................................................... 2-1
      2.2.1.4 Functional Description ............................................................................. 2-2
      2.2.1.5 Example of FFT16 Function .................................................................... 2-5
      2.2.1.6 Optimizations .......................................................................................... 2-5
      2.2.1.7 Observations ............................................................................................ 2-7
   2.2.2 INV_FFT16 ....................................................................................................... 2-7
      2.2.2.1 Call(s) ....................................................................................................... 2-7
      2.2.2.2 Parameters ............................................................................................... 2-7
      2.2.2.3 Returns .................................................................................................... 2-7
      2.2.2.4 Functional Description ............................................................................. 2-8
      2.2.2.5 Example of INV_FFT16 Function .............................................................. 2-8
      2.2.2.6 Observations ............................................................................................ 2-8
   2.3 FFT32 Functions .................................................................................................. 2-8
   2.3.1 FFT32 ............................................................................................................ 2-8
      2.3.1.1 Call(s) ....................................................................................................... 2-8
      2.3.1.2 Parameters ............................................................................................... 2-8
      2.3.1.3 Returns .................................................................................................... 2-9
      2.3.1.4 Functional Description ............................................................................. 2-9
      2.3.1.5 Example of FFT32 Function .................................................................... 2-12
Chapter 3
Finite Impulse Filter (FIR)

3.1 FIR16 Functions ................................................................. 3-1
   3.1.1 tFir16Struct Structure ........................................... 3-1
   3.1.2 FIR16Create ............................................................. 3-1
      3.1.2.1 Call(s) ......................................................... 3-1
      3.1.2.2 Parameters ................................................. 3-2
      3.1.2.3 Returns ..................................................... 3-2
   3.1.3 FIR16 .......................................................... 3-2
      3.1.3.1 Call(s) ......................................................... 3-2
      3.1.3.2 Parameters ................................................. 3-2
      3.1.3.3 Returns ..................................................... 3-2
   3.1.4 FIR16Destroy ....................................................... 3-2
      3.1.4.1 Call(s) ......................................................... 3-2
      3.1.4.2 Parameters ................................................. 3-3
      3.1.4.3 Returns ..................................................... 3-3
   3.1.5 Example of the FIR16 Function ............................... 3-3
   3.1.6 Optimizations ....................................................... 3-3
   3.1.7 Observations ....................................................... 3-5

3.2 FIR32 Functions .......................................................... 3-5
   3.2.1 tFir32Struct Structure ........................................... 3-5
   3.2.2 FIR32Create .......................................................... 3-5
      3.2.2.1 Call(s) ......................................................... 3-5
      3.2.2.2 Parameters ................................................. 3-6
      3.2.2.3 Returns ..................................................... 3-6
   3.2.3 FIR32 .......................................................... 3-6
      3.2.3.1 Call(s) ......................................................... 3-6
      3.2.3.2 Parameters ................................................. 3-6
      3.2.3.3 Returns ..................................................... 3-6
   3.2.4 FIR32Destroy ....................................................... 3-6
      3.2.4.1 Call(s) ......................................................... 3-6
      3.2.4.2 Parameters ................................................. 3-7
      3.2.4.3 Returns ..................................................... 3-7
   3.2.5 Example of FIR32 Function .................................. 3-7
## Chapter 4

### Infinite Impulse Filter (IIR)

#### 4.1 IIR16 Functions

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>tIir16Struct Structure</td>
<td>4-1</td>
</tr>
<tr>
<td>IIR16Create</td>
<td>4-1</td>
</tr>
<tr>
<td>4.1.2.1 Call(s)</td>
<td>4-1</td>
</tr>
<tr>
<td>4.1.2.2 Parameters</td>
<td>4-2</td>
</tr>
<tr>
<td>4.1.2.3 Returns</td>
<td>4-2</td>
</tr>
<tr>
<td>IIR16</td>
<td>4-2</td>
</tr>
<tr>
<td>4.1.3.1 Call(s)</td>
<td>4-2</td>
</tr>
<tr>
<td>4.1.3.2 Parameters</td>
<td>4-2</td>
</tr>
<tr>
<td>4.1.3.3 Returns</td>
<td>4-2</td>
</tr>
<tr>
<td>IIR16Destroy</td>
<td>4-2</td>
</tr>
<tr>
<td>4.1.4.1 Call(s)</td>
<td>4-3</td>
</tr>
<tr>
<td>4.1.4.2 Parameters</td>
<td>4-3</td>
</tr>
<tr>
<td>4.1.4.3 Returns</td>
<td>4-3</td>
</tr>
<tr>
<td>Example of the IIR16 Function</td>
<td>4-3</td>
</tr>
<tr>
<td>Optimizations</td>
<td>4-3</td>
</tr>
<tr>
<td>Observations</td>
<td>4-5</td>
</tr>
</tbody>
</table>

#### 4.2 IIR32 Functions

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>tFir32Struct Structure</td>
<td>4-5</td>
</tr>
<tr>
<td>IIR32Create</td>
<td>4-5</td>
</tr>
<tr>
<td>4.2.2.1 Call(s)</td>
<td>4-5</td>
</tr>
<tr>
<td>4.2.2.2 Parameters</td>
<td>4-6</td>
</tr>
<tr>
<td>4.2.2.3 Returns</td>
<td>4-6</td>
</tr>
<tr>
<td>IIR32</td>
<td>4-6</td>
</tr>
<tr>
<td>4.2.3.1 Call(s)</td>
<td>4-6</td>
</tr>
<tr>
<td>4.2.3.2 Parameters</td>
<td>4-6</td>
</tr>
<tr>
<td>4.2.3.3 Returns</td>
<td>4-6</td>
</tr>
<tr>
<td>IIR32Destroy</td>
<td>4-6</td>
</tr>
<tr>
<td>4.2.4.1 Call(s)</td>
<td>4-7</td>
</tr>
<tr>
<td>4.2.4.2 Parameters</td>
<td>4-7</td>
</tr>
<tr>
<td>Returns</td>
<td>4-7</td>
</tr>
<tr>
<td>Example of the IIR32 Function</td>
<td>4-7</td>
</tr>
<tr>
<td>Optimizations</td>
<td>4-7</td>
</tr>
<tr>
<td>Observations</td>
<td>4-9</td>
</tr>
</tbody>
</table>
Chapter 1
Overview

1.1 Introduction
This document describes a library of digital signal processing functions designed to work with the eMAC and MAC units in ColdFire processors. It includes three of the most common DSP functions:

1. Fast-fourier transform (FFT)
2. Finite impulse filter (FIR)
3. Infinite impulse filter (IIR)

The library contains functions used to execute these functions using the multiply-and-accumulate (MAC) module or the enhanced multiply-and-accumulate (eMAC) module available in ColdFire processors.

1.2 General Description of a Complex FFT Calculation

Figure 1-1 depicts the complex FFT butterfly.

Twiddle Factor:
\[ W_k = w_r + jw_i = \cos(2\pi k/N) + j\sin(2\pi k/N) \]

Equation of the butterfly:
\[ x_r = a_r + w_r * b_r - w_i * b_i \]
\[ x_i = a_i + w_i * b_r + w_r * b_i \]
\[ y_r = a_r - w_r * b_r + w_i * b_i = 2 * a_r - x_r \]
\[ y_i = a_i - w_i * b_r - w_r * b_i = 2 * a_i - x_i \]
Figure 1-2 depicts the stages of the complex FFT execution.

All variables (ai, bi, wi) on the diagram are complex values. Cross (X) means butterfly operation (see Figure 1-2).

For a 64-point FFT, six stages have to be completed as the following equation shows \((\log_2(64) = 6)\). At each input stage, the values for butterfly are a and b, and the output values are x and y. At the beginning of the next stage, the output values from the previous stage become the input values for the current stage. For example, at stage 2, x0 (after first stage) becomes a0, y0 becomes a1, x1 becomes b0, and so on.

### 1.3 General Filters Description

The infinite impulse response filter can be described by:

\[
Y[j] = \sum_{k=0}^{n-1} a[k]x[j - k] + \sum_{k=0}^{n-1} b[k]Y[j - k - 1] \quad 0 < j \leq n
\]  
**Eqn. 1-1**

where the output \(Y[j]\) is determined by past output values \(Y\) and input values \(x\). All input values and coefficients \(a\) and \(b\) are fractional. The number of coefficients must be less or equal the number of input samples.

The finite impulse response filter can be described by:

\[
Y[j] = \sum_{k=0}^{n-1} c[k]x[j - k] \quad 0 < j \leq n
\]  
**Eqn. 1-2**

where the output \(Y[j]\) is determined only by input values \(x\). All input values and coefficients \(c\) are fractional. The number of coefficients must be less or equal the number of input samples.

---

1. The FFT is processed in \(N\cdot\log_2(N)\) operations. The \(\log_2(N)\) determines the number of stages per each element (\(N\)).
1.4 **Library Structure**

The Library of Macros includes header files containing the function prototypes for each algorithm, as well as assembly source files containing the code for the corresponding module (MAC or eMAC). The IIR and FIR filters include C source files containing the required functions to create and destroy the structures needed by the algorithm. See **Figure 1-3**.

![Figure 1-3. DSP Library Structure](image)

1.5 **Getting Started**

To get started, include the appropriate header file to your current project’s source file along with the corresponding source files. Sample projects developed in Freescale’s CodeWarrior IDE tool are included for demonstration purposes. These examples are developed for the MCF5282 ColdFire processor and include examples for eMAC and MAC, as well as 32 and 16 bits. The files included have the structure shown in **Figure 1-4**.

![Figure 1-4. Structure of Included Files](image)

The structure of included files is explained in **Figure 1-4**:
- The `.mcp` file is the CodeWarrior project file.
- The Results folder contains some files with the results of the test.
- The Source folder contains all the source files needed for the project.
- The Other Files Folder is required by the CodeWarrior project.
1.6 Definitions Used in Library

In order to facilitate the use of these libraries, the features described in this subsection are used.

1.6.1 Data Types

Variables of types described in Table 1-1 are used as parameters of FIR16/FIR/IIR module functions.

### Table 1-1. Data Types Used in the Library

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frac16</td>
<td>Frac16 type is defined in the header files for each library and describes 16-bit fractional value.</td>
<td></td>
</tr>
</tbody>
</table>
| | | #define iNUM_OF_SAMPLE 7
| | | Frac16 aY[iNUM_OF_SAMPLE]; |
| Frac32 | Frac32 type is defined in the header files for each library and describes 32-bit fractional value. | 
| | | #define iNUM_OF_SAMPLE 7
| | | Frac32 aX[iNUM_OF_SAMPLE]; |

1.6.2 Macros

Macros are used to convert between floating point representation and Frac[16,32] constant, as well as for reverse converting. See Table 1-2.

### Table 1-2. Macros Types Used in the Library

<table>
<thead>
<tr>
<th>Macro</th>
<th>Definition</th>
<th>Description</th>
<th>Example of use</th>
</tr>
</thead>
</table>
| CFF16 | #define CFF16(X) (Frac16)(X*32768.0) | Macro CFF16 provides a conversion from floating-point representation to Frac16 constant. Parameter of this macros must be more than –1 and less than 1. In other cases, using this macros will lead to incorrect results of computations. | CFF16(0.01)
| | | This will result in 0.01*32768.0 = 327. |
| ICFF16 | #define ICFF16(X) (((double)(X))/32768.0) | Macro ICFF16 provides a conversion from Frac16 constant to floating-point representation constant. | ICFF16(327)
| | | This will result in 327 / 32768.0 = 0.00999. |
| CFF32 | #define CFF32(X) (Frac32)(X*2147483648.0) | Macro CFF32 is used to convert from floating point representation to Frac32 constant. Parameter of this macro must be more than –1 and less than 1. In other cases, using this macro will lead to incorrect results of computations. | CFF32(0.01)
| | | This will result in 0.01*2147483648.0 = 21474836. |
Table 1-2. Macros Types Used in the Library (continued)

<table>
<thead>
<tr>
<th>Macro</th>
<th>Definition</th>
<th>Description</th>
<th>Example of use</th>
</tr>
</thead>
<tbody>
<tr>
<td>ICFF32</td>
<td><code>#define ICFF32(X) (((double)(X))/2147483648.0)</code></td>
<td>Macro ICFF32 is used to convert from Frac32 constant to floating point representation constant.</td>
<td>ICFF32(21474836) This will result in $21474836 / 2147483648.0 = 0.00999$.</td>
</tr>
<tr>
<td>FRAC16</td>
<td><code>#define FRAC16(x) ((Frac16)((x) &lt; 1 ? ((x) &gt;= -1 ? CFF16(x) : MIN_16) : MAX_16))</code></td>
<td>Macro FRAC16 is used to convert from floating point representation to Frac16 constant, considering the case of the maximum and minimum values.</td>
<td>FRAC16(1.0) This will result in MAX_16 = 32767.</td>
</tr>
<tr>
<td>FRAC32</td>
<td><code>#define FRAC32(x) ((Frac32)((x) &lt; 1 ? ((x) &gt;= -1 ? (x) * MIN_32 : MIN_32) : MAX_32))</code></td>
<td>Macro FRAC32 is used to convert from floating point representation to Frac32 constant, considering the case of the maximum and minimum values.</td>
<td>FRAC32(1.0) This will result in MAX_32 = 2147483647.</td>
</tr>
</tbody>
</table>
Chapter 2
Fast Fourier Transform (FFT)

2.1 Introduction
This chapter discusses the following functions that compute the fast fourier transform (FFT) of an array of elements in their respective formats:

- FFT16
- INV_FFT16
- FFT32
- INV_FFT32

2.2 FFT16 Functions

2.2.1 FFT16
This function computes the fast fourier transform of an array of elements in Frac16 format.

2.2.1.1 Call(s)

```c
void fft16 (void * ReX, void * ImX)
```

2.2.1.2 Parameters

| void * ReX | In | Pointer to the 2048 bytes (1024 words) ReX[] buffer |
| void * ImX | In | Pointer to the 2048 bytes (1024 words) ImX[] buffer |

Upon entry, ReX[] contains the real input signal, while values in ImX[] are invalid. Indexes increment from 0 to 1023.

2.2.1.3 Returns
The real DFT takes a 1024 point time domain signal and creates two 513-point frequency domain signals. Upon return from the function, ReX contains the amplitudes of the cosine waves, and ImX contains the amplitudes of the sine waves. The indexes run from 0 to 512.
2.2.1.4 Functional Description

The real FFT is calculated by a complex FFT algorithm.

At the start of the calculation, the program stores odd-indexed values from the ReX to the ImX buffer. So, for the first nine stages of the FFT, the program only uses the first halves of each buffer (thus, the indexes increment not from 0 to 1023, but from 0 to 512). It is assumed, that the ReX[] buffer contains the real parts of the values, and the ImX[] buffer contains the imaginary parts of the values.

The values are then reordered using a bit-reversed addressing mode independently in each buffer.

Fractional values are represented in the following format:

\[
s.\ldots,\text{where:}\]
\[s\] is the sign bit, and
\[x\] is the data bit.

This format doesn’t allow us to represent decimal 1 (the max positive value is \(2^{15} - 1\)), but allows us to represent \(-1\) (0x8000). To minimize the error in multiplication when \(w_r = 1\) (W0), it is better to store in the table of twiddle factors \(-w_r\), with the following modification of the butterfly (the sign is changed for \(w_r\)):

\[
\begin{align*}
x_r &= a_r - w_r * b_r - w_i * b_i \\
x_i &= a_i + w_i * b_r - w_r * b_i \\
y_r &= 2 * a_r - x_r \\
y_i &= 2 * a_i - x_i
\end{align*}
\]

Therefore, the table of twiddle factors has the following structure:

\[-w_r 0 \quad w_i 0 \quad -w_r 1 \quad w_i 1 \quad -w_r 2 \quad w_i 2 \quad \ldots\]

For the first stage of calculation, all butterflies need only one complex twiddle factor, with real part \(w_r = -1\), and imaginary part \(w_i = 0\) (the twiddle factor having an index 0). Therefore, using \(-1\) instead of \(w_r\), and 0 instead of \(w_i\) in the butterfly simplifies the calculation:

\[
\begin{align*}
x_r &= a_r - (-1) * b_r - 0 * b_i = a_r + b_r \\
x_i &= a_i + 0 * b_r - (-1) * b_i = a_i + b_i \\
y_r &= 2 * a_r - x_r = 2 * a_r - a_r - b_r = a_r - b_r \\
y_i &= 2 * a_i - x_i = 2 * a_i - a_i - b_i = a_i - b_i
\end{align*}
\]

Thus, it is always possible to make some improvements in efficiency in the case of the calculation of butterflies for the first stage of the FFT separately from the other stages. Each iteration of this loop calculates one butterfly.

At the second stage of the FFT, the situation looks like that of the first stage, with some exceptions. To calculate butterflies with an even number (the number starts from 0) it is necessary to use a twiddle factor, with real part \(w_r = -1\), and imaginary part \(w_i = 0\) (the twiddle factor having an index of 0). To calculate butterflies with an odd number, it is also necessary to use a twiddle factor, with real part \(w_r = 0\), and imaginary part \(w_i = -1\) (the twiddle factor having an index of 256).
The butterflies with an even number (that is when the twiddle factor with index 0 is used) look like:

\[
\begin{align*}
    x_r &= a_r + b_r \\
    x_i &= a_i + b_i \\
    y_r &= a_r - b_r \\
    y_i &= a_i - b_i
\end{align*}
\]

The butterflies with an odd number (that is when a twiddle factor with index 256 is used) look like:

\[
\begin{align*}
    x_r &= a_r + b_i \\
    x_i &= a_i - b_r \\
    y_r &= a_r - b_i \\
    y_i &= a_i + b_r
\end{align*}
\]

For each iteration of the loop two butterflies (one with an even number and another with an odd number) can be calculated. From here on, stage 3 to 9 butterflies are calculated in a following fashion.

The third stage starts at label next_stage (this is the start point for stages 3–9). 64 sub DFTs should be calculated on this third stage with 4 butterflies per one sub DFT. The ColdFire programming model uses these registers:

- Register a0 points to the beginning of ReX[] buffer (first ar value)
- Register a1 points to the beginning of ImX[] buffer (first ai value)
- Register a2 points to the first br value
- Register a3 points to the first bi value
- Register a5 contains the number of butterflies per one sub DFT (for the third stage, it is 4) multiplied by 2 (2 bytes is the size of one value).

Label next_subDFT is the beginning of sub DFTs loops, which enumerates (calculates) all sub DFTs on the current stage (this loop is included into the loop, which enumerates stages).

Label next_bf is the beginning of the loop, which enumerates all butterflies of the current sub DFT. This loop is included into the previous one.

After completion of calculating of the butterfly, value 2 is added to the contents of registers a0, a1, a2, and a3. These registers now point to the input values for the next butterfly of the current sub DFT. There is no need to write any extra code to implement this addition. When the pointer becomes useless, addressing mode with post increment is used.
At the end of calculating sub DFT, the contents of a5 is added to the contents of a0, a1, a2, and a3. This indicates that these address registers now point to the input values for the first butterfly of the next sub DFT. Thus, calculation of the next sub DFT can be started.

Then, the contents of a5 will be multiplied by 2 (it is between instructions bcs.w next_subDFT and bcs.w next_stage). The number of butterflies per one sub DFT on the next stage is equal to the number of butterflies per one sub DFT on the current stage multiplied by 2. The number of sub DFTs on the next stage is equal to the number of sub DFT on the current stage divided by 2.

Next stage can be started (from label next_stage).

Butterflies on stages 3 to 9 are calculated using following equation:

\[
\begin{align*}
    x_r &= a_r + w_r \times b_r - w_i \times b_i \\
    x_i &= a_i + w_i \times b_r + w_r \times b_i \\
    y_r &= 2 \times a_r - x_r \\
    y_i &= 2 \times a_i - x_i
\end{align*}
\]

\[\text{Figure 2-1. Start Point for FET16 Stages 3 to 9}\]
A butterfly equation cannot be simplified, as it was done on stages 1 and 2, because of the values of twiddle factors used on stages 3 to 9.

Figure 2-2 shows the allocation of data inside the buffers. On each stage, register a0 points to the ar value, register a2 points to the br value, register a1 contains the address of the ai value, and register a3 points to the bi value. Register a4 points to the twiddle factor, which is used for the calculation of the current butterfly. Outputs of the butterfly are written back instead of the inputs (i.e. an in-place calculation). Xr is written back instead of ar, yr binstead of br, xi instead of ai, and finally, yi instead of bi.

The program then performs even/odd frequency domain decomposition.

Upon completion, the program calculates the last 10th stage for buffers of full length, which is equal to 1024 points (in stages 1 to 9, only the first halves of each buffer were used).

The values are scaled internally on this stage.

\[
\text{Figure 2-2. Memory Map (FFT16)}
\]

2.2.1.5 Example of FFT16 Function

```c
int16 ReX[1024];
int16 ImX[1024];
void main (void)
{
    ...
    fft16 (&ReX, &ImX);
    ...
}
```

2.2.1.6 Optimizations

Real and imaginary parts of samples are computed using only one accumulator in original code. eMAC has four accumulators. So, we can calculate real and imaginary parts simultaneously for better instruction pipelining. We don’t use all four accumulators because the code, which uses four accumulators, has too many surround codes.

Refer to Figure 2-3 and Figure 2-4.
Figure 2-3. Optimized for MAC Unit
Assembly Code (FFT16)

```assembly
and.l #0xffff0000, d2
move.l d2, ACC
msacl.w d0.u, d4.u, <<, (a3), d5
msacl.w d0.l, d5.u, <<, (a1), d7
move.l ACC, d3
asr.l d1, d3
move.w d3, (a0) +
add.l d2, d2
asr.l d1, d2
sub.l d3, d2
move.w d2, (a2) +
and.l #0xffff0000, d7
move.l d7, ACC
macl.w d0.l, d4.u, <<, (a0), d2
msacl.w d0.u, d5.u, <<, (a2), d4
move.l ACC, d3
asr.l d1, d3
move.w d3, (a1) +
add.l d7, d7
asr.l d1, d7
sub.l d3, d7
move.w d7, (a3) +
```

Figure 2-4. Optimized for eMAC Unit
Assembly Code (FFT16)

```assembly
move.l (a2), d4
move.w (a0), d2
move.w (a1), d7
msacl.w d0.u, d4.u, <<, (a3), d5, ACC0
msac.w d0.l, d5.u, <<, ACC0
mac.w d0.l, d4.u, <<, ACC1
msac.w d0.u, d5.u, <<, ACC1
movclr.l ACC0, d3
movclr.l ACC1, d1
add.l d2, d3
move.w d3, (a0) +
add.l d2, d2
sub.l d3, d2
move.w d2, (a2) +
add.l d7, d1
move.w d1, (a1) +
add.l d7, d7
sub.l d1, d7
move.w d7, (a3) +
```
2.2.1.7 Observations

Due to the nature of FFT processing and the range of fractional numbers, some intermediate or final results can exceed the range of fractional numbers and cause overflows. It is recommended to reduce the amplitude of the input signal properly before the FFT processing.

- The use of eMAC or MAC is specified in the fft.h header file by the following line:
  ```c
  #define __EMAC
  ```
- The FRAC16 macro is included in the fft.h header file in order to facilitate the conversion of floating point numbers to fractional numbers.

2.2.2 INV_FFT16

This function computes the inverse fast fourier transform of an array of elements in Frac16 format.

2.2.2.1 Call(s)

```c
void inv_fft16 (void * ReX, void * ImX)
```

2.2.2.2 Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void * ReX</td>
<td>In Pointer to the 2048 bytes (1024 words) ReX[] buffer</td>
</tr>
<tr>
<td>void * ImX</td>
<td>In Pointer to the 2048 bytes (1024 words) ImX[] buffer</td>
</tr>
</tbody>
</table>

Upon entry, ReX[] and ImX[] contain the real and imaginary parts accordingly of the frequency domain running from index 0 to 512. The remaining samples in ReX[] and ImX[] are ignored.

2.2.2.3 Returns

Upon return from the program, ReX[] contains the real time domain. The indexes run from 0 to 1023. In general, after execution of the following sequence:

1. Forward FFT
2. Inversed FFT

Output values from inversed FFT will be 1024 times \(2^{10}\) greater than the corresponding source input values for forward FFT without scaling. And, normally, these output values should be divided by 1024 after execution of the previous sequence of subroutines.

In the case of scaling, there is no need to normalize inversed FFT outputs. In most cases we should only remember that the values could be scaled.
2.2.2.4 Functional Description

The program first makes the frequency domain symmetrical, then adds the real and imaginary parts together. After that, it calculates forward real FFT. To do this, the program pushes into the stack addresses of the ReX[] and ImX[] buffers, which were passed into inv_fft() subroutine. Finally, it adds the real and imaginary parts together. ReX[] contains the real time domain.

2.2.2.5 Example of INV_FFT16 Function

```c
int16 ReX[1024];
int16 ImX[1024];
void main (void)
{
    ...
    fft16 (&ReX, &ImX);
    inv_fft16(&ReX, &ImX);
    ...
}
```

2.2.2.6 Observations

Due to the nature of FFT processing and the range of fractional numbers, some intermediate or final results can exceed the range of fractional numbers and thus cause overflows. It is recommended to reduce the amplitude of the input signal properly before the FFT processing.

The use of eMAC or MAC is specified in the fft.h header file by the following line:

```c
#define __EMAC
```

The FRAC16 macro is included in the fft.h header file in order to facilitate the conversion of floating point numbers to fractional numbers.

2.3 FFT32 Functions

2.3.1 FFT32

This function computes the fast fourier transform of an array of elements in Frac32 format.

2.3.1.1 Call(s)

```c
void fft32(void * ReX, void * ImX)
```

2.3.1.2 Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void * ReX</td>
<td>In Pointer to the 4096 bytes (1024 longs) ReX[ ] buffer</td>
</tr>
<tr>
<td>void * ImX</td>
<td>In Pointer to the 4096 bytes (1024 longs) ImX[ ] buffer</td>
</tr>
</tbody>
</table>
Upon entry, ReX[] contains the real input signal, while values in ImX[] are invalid. Indexes increment from 0 to 1023.

2.3.1.3 Returns

The real DFT takes a 1024 point time domain signal and creates two 513-point frequency domain signals. Upon return from the function, ReX contains the amplitudes of the cosine waves, and ImX contains the amplitudes of the sine waves. The indexes run from 0 to 512.

2.3.1.4 Functional Description

The real FFT is calculated by a complex FFT algorithm.

At the start of the calculation, the program stores odd-indexed values from the ReX to the ImX buffer. So, for the first nine stages of the FFT, the program uses only the first halves of each buffer (thus the indexes increment not from 0 to 1023, but from 0 to 512). It is assumed that the ReX[] buffer contains the real parts of the values, and the ImX[] buffer contains the imaginary parts of the values.

The values are then reordered using a bit-reversed addressing mode independently in each buffer.

Fractional values are represented in the following format:

\[ s.xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx, \text{ where} \]
\[ s \text{ is the sign bit, and} \]
\[ x \text{ is the data bit.} \]

This format doesn’t allow us to represent decimal 1 (the max positive value is \(2^{31} - 1\)), but allows us to represent –1 (0x80000000). Thus to minimize the error in multiplication when \(wr = 1\) (W0), it is better to store in the table of twiddle factors not \(wr\), but –\(wr\), with the following modification of the butterfly (the sign is changed for \(wr\)):

\[ \begin{align*}
x_r &= ar – wr * br – wi * bi \\
x_i &= ai + wi * br – wr * bi \\
y_r &= 2 * ar – xr \\
y_i &= 2 * ai – xi
\end{align*} \]

Therefore, the table of twiddle factors has the following structure:

\[-wr0\ wi0\ -wr1\ wi1\ -wr2\ wi2\ \ldots\]

For the first stage of calculation, all butterflies need only one complex twiddle factor, with real part \(wr = –1\) and imaginary part \(wi = 0\) (the twiddle factor having an index 0). Therefore, using –1 instead of \(wr\) and 0 instead of \(wi\) in the butterfly simplifies the calculation:

\[ \begin{align*}
x_r &= ar – (-1) * br – 0 * bi = ar + br \\
x_i &= ai + 0 * br – (-1) * bi = ai + bi \\
y_r &= 2 * ar – xr = 2 * ar – ar – br = ar – br \\
y_i &= 2 * ai – xi = 2 * ai – ai - bi = ai – bi
\end{align*} \]
Thus, it is always possible to make some improvements in efficiency in the case of the calculation of butterflies for the first stage of the FFT separately from the other stages. Each iteration of this loop calculates one butterfly.

At the second stage of the FFT, the situation looks like that of the first stage, with some exceptions. To calculate butterflies with an even number (the number starts from 0), it is necessary to use a twiddle factor, with real part $w_r = -1$ and imaginary part $w_i = 0$ (the twiddle factor having an index of 0). To calculate butterflies with an odd number, it is also necessary to use a twiddle factor, with real part $w_r = 0$ and imaginary part $w_i = -1$ (the twiddle factor having an index of 256).

The butterflies with an even number (that is when the twiddle factor with index 0 is used) look like:

\[
\begin{align*}
x_r &= a_r + b_r \\
x_i &= a_i + b_i \\
y_r &= a_r - b_r \\
y_i &= a_i - b_i
\end{align*}
\]

The butterflies with an odd number (that is when a twiddle factor with index 256 is used) look like:

\[
\begin{align*}
x_r &= a_r + b_i \\
x_i &= a_i - b_r \\
y_r &= a_r - b_i \\
y_i &= a_i + b_r
\end{align*}
\]

For each iteration of the loop, two butterflies (one with an even number and another with an odd number) can be calculated. From here on, stage 3 to 9 butterflies are calculated in a following fashion.

The third stage starts at label next_stage (this is the starting point for stages 3 to 9). 64 sub DFTs should be calculated on this third stage with 4 butterflies per one sub DFT. Register a0 points to the beginning of ReX[] buffer (first ar value), a1 points to the beginning of ImX[] buffer (first ai value), register a2 points to the first br value, and a3 points to the first bi value. The contents of registers a1 and a3 are calculated as the sum of the values of a0 and a2 respectively, and the value of register a5. a5 contains the number of butterflies per one sub DFT (for the third stage, it is 4) multiplied by 2 (2 bytes is the size of one value).

Label next_subDFT is the beginning of sub DFTs loops, which enumerates all sub DFTs on the current stage (this loop is included into the loop, which enumerates stages).

Label next_bf is the beginning of the loop, which enumerates all butterflies of the current sub DFT. This loop is included into the previous one.

After completion of calculating of the butterfly, value 2 is added to the contents of registers a0, a1, a2, and a3. These registers point to the input values for the next butterfly of the current sub DFT now. There is no need to write any extra code to implement this addition. When the pointer becomes useless, addressing mode with post increment is used.
At the end of calculation of each sub DFT, the contents of a5 is added to the contents of a0, a1, a2, and a3. This means that now these address registers point to the input values for the first butterfly of the next sub DFT. Thus, the calculating of next sub DFT can be started.

At the end of each stage, the contents of a5 will be multiplied by 2 (it is between instructions bcs.w next_subDFT and bcs.w next_stage). The number of butterflies per one sub DFT on the next stage is equal to the number of butterflies per one sub DFT on the current stage multiplied by 2. The number of sub DFTs on the next stage is equal to the number of sub DFT on the current stage divided by 2.

The ext stage can be started (from label next_stage).
Butterflies on stages 3 to 9 are calculated using the following equation:

\[
\begin{align*}
    x_r &= a_r + w_r \cdot b_r - w_i \cdot b_i \\
    x_i &= a_i + w_i \cdot b_r + w_r \cdot b_i \\
    y_r &= 2 \cdot a_r - x_r \\
    y_i &= 2 \cdot a_i - x_i.
\end{align*}
\]

An equation of butterfly can not be simplified, as was done on stages 1 and 2, because of the values of twiddle factors, which are used on stages 3 to 9.

Figure 2-6 shows the allocation of data inside the buffers. On each stage, register a0 points to the ar value, register a2 points to the br value, register a1 contains the address of the ai value, and register a3 points to the bi value. Register a4 points to the twiddle factor, which is used for the calculation of the current butterfly. Outputs of the butterfly are written back instead of the inputs (i.e. an in-place calculation). Xr is written back instead of ar, yr instead of br, xi instead of ai, and finally, yi instead of bi.

The program then performs even/odd frequency domain decomposition.

After that is complete, the program calculates the last 10th stage for buffers of full length, which is equal to 1024 points (in stages 1 to 9 only the first halves of each buffer were used).

The values are scaled internally on this stage.

![Figure 2-6. Memory Map (FFT32)](image)

### 2.3.1.5 Example of FFT32 Function

```c
int32  ReX[1024];
int32  ImX[1024];
void main (void)
{
    ...
    fft32(&ReX, &ImX);    
    ...
}
```
2.3.1.6 Optimizations

Real and imaginary parts of samples are computed using only one accumulator in original code. eMAC has four accumulators. So, we can calculate real and imaginary parts simultaneously for better instruction pipelining. Also, we can unroll next_bf loop (compute two butterflies per one iteration). But we don’t use all four accumulators because the code, which uses four accumulators, has too many surround codes.

Refer to Figure 2-7 and Figure 2-8.

```
next_bf:
  movem.l  (a4),d0-d1
  move.l   d2,ACC
  msaccl.l d0,d4,(a3),d5
  msaccl.l d1,d5,(a1),a6
  move.l   ACC,d3
  move.l   d3,(a0)+
  add.l    d2,d2
  sub.l    d3,d2
  move.l   d2,(a2)+
  move.l   a6,ACC
  maccl.l  d1,d4,(a0),d2
  msaccl.l d0,d5,(a2),d4
  move.l   ACC,d3
  move.l   d3,(a1)+
  adda.l   a6,a6
  suba.l   d3,a6
  move.l   a6,(a3)+
  adda.l   d6,a6
  addq.l   #4,d7
  cmp.l    a5,d7
  bcs.b    next_bf
```

**Figure 2-7. Optimized for MAC Unit**

Assembly Code (FFT32)
Figure 2-8. Optimized for eMAC Unit
Assembly Code (FFT32)
2.3.1.7 Observations

Due to the nature of FFT processing and the range of fractional numbers, some intermediate or final results can exceed the range of fractional numbers and cause overflows. It is recommended to reduce the amplitude of the input signal properly before the FFT processing.

The use of eMAC or MAC is specified in the fft.h header file by the following line:

```c
#define __EMAC
```

The FRAC32 macro is included in the fft.h header file in order to facilitate the conversion of floating point numbers to fractional numbers.

2.3.2 INV_FFT32

This function computes the inverse fast fourier transform of an array of elements in Frac32 format.

2.3.2.1 Call(s)

```c
void inv_fft32 (void * ReX, void *ImX)
```

2.3.2.2 Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void * ReX</td>
<td>Pointer to the 4096 bytes (1024 longs) ReX[] buffer</td>
</tr>
<tr>
<td>void * ImX</td>
<td>Pointer to the 4096 bytes (1024 longs) ImX[] buffer</td>
</tr>
</tbody>
</table>

Upon entry, ReX[] and ImX[] contain the real and imaginary parts accordingly of the frequency domain running from index 0 to 512. The remaining samples in ReX[] and ImX[] are ignored.

2.3.2.3 Returns

Upon return from the program, ReX[] contains the real time domain. The indexes run from 0 to 1023.

In general, after execution of the following sequence:

1. Forward FFT
2. Inversed FFT

Output values from inversed FFT will be 1024 times \(2^{10}\) greater than the corresponding source input values for forward FFT without scaling. Normally, these output values should be divided by 1024 after execution of the previous sequence of subroutines.

In the case of scaling, there is no need to normalize inversed FFT outputs. In most cases we should only remember that the values could be scaled.
2.3.2.4 Functional Description

The program first makes the frequency domain symmetrical, then adds the real and imaginary parts together. After this, it calculates the forward real FFT. Finally, it adds the real and imaginary parts together. ReX[] contains the real time domain.

2.3.2.5 Example of INV_FFT32 Function

```c
int32 ReX[1024];
int32 ImX[1024];
void main (void)
{
    ...
    fft32(&ReX, &ImX);
    inv_fft32(&ReX, &ImX);
    ...
}
```

2.3.2.6 Observations

Due to the nature of FFT processing and the range of fractional numbers, some intermediate or final results can exceed the range of fractional numbers and cause overflows. It is recommended to reduce the amplitude of the input signal properly before the FFT processing.

The use of eMAC or MAC is specified in the fft.h header file by the following line:

```c
#define __EMAC
```

The FRAC32 macro is included in the fft.h header file in order to facilitate the conversion of floating point numbers to fractional numbers.
Chapter 3
Finite Impulse Filter (FIR)

3.1   FIR16 Functions

These functions are used to filter an input signal of Frac16 numbers using the coefficients of a finite impulse filter.

3.1.1   tFir16Struct Structure

tFir16Struct is a structure containing data needed by the FIR16 function.

typedef struct       tFir16Struct{
    Frac16* pFirCoef;
    unsigned int         iFirCoefCount;
    Frac16* pFirHistory;
    unsigned int         iFirHistoryCount;
};

pFirCoef is a pointer to an array of FIR filter coefficients.

iFirCoefCount is a number of elements in filter coefficients’ array.

pFirHistory is a pointer to the history buffer of the FIR filter. The history buffer contains the last iFirCoefCount –1 input samples from the previous call of the FIR16 function. These last iFirCoefCount –1 input samples are used in computations of first iFirCoefCount – 1 results of the next call of the FIR16 function. The history buffer is not used in the first call of the FIR16 function.

iFirHistoryCount is a number of elements in the history buffer. It equals 0 during the first calling of the FIR16 function and iFirCoefCount – 1 during next calls.

3.1.2   FIR16Create

FIR16Create performs the initialization for the FIR16 filter function. FIR16Create allocates and initializes a data structure, which is used by FIR to preserve the filter’s state between calls. The data structure preserves a copy of the array of FIR filter coefficients. FIR16Create allocates a buffer to save the past history of N–1 data elements required by the FIR filter computation.

3.1.2.1   Call(s)

struct tFir16Struct* FIR16Create(Frac16* pCoef, unsigned int N);
3.1.2.2 Parameters

Table 3-1. FIR16Create Parameters

<table>
<thead>
<tr>
<th>pCoef</th>
<th>In</th>
<th>Pointer to a array of FIR filter coefficients {a0, a1, a2...}</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>In</td>
<td>Length of the array of FIR filter coefficients pointed to by pCoef; Must be less or equal length of the input and output arrays and more than one.</td>
</tr>
</tbody>
</table>

3.1.2.3 Returns

The FIR16Create function returns a pointer to the tFir16Struct data structure that it allocated if all allocations and initialization succeed. This pointer must then be passed to subsequent calls of the FIR16 function. If insufficient data memory is available, the FIR16Create function returns 0 (NULL).

3.1.3 FIR16

FIR16 computes a finite impulse response (FIR) filter for an array of fractional data values. Prior to any call to FIR16, the FIR filter must be initialized via a call to FIR16Create; the FIR filter uses coefficients passed to that FIR16Create call. FIR16 uses the private data structure established by FIR16Create to maintain the past history of data elements required by the FIR filter computation.

3.1.3.1 Call(s)

```c
void FIR16(struct tFir16Struct* pFIR, Frac16* pX, Frac16* pY, unsigned int n);
```

3.1.3.2 Parameters

Table 3-2. FIR16 Parameters

<table>
<thead>
<tr>
<th>pFIR</th>
<th>In</th>
<th>Pointer to a data structure containing private data for the FIR filter; this pointer is created by a call of FIR16Create</th>
</tr>
</thead>
<tbody>
<tr>
<td>pX</td>
<td>In</td>
<td>Pointer to the input array of n data elements</td>
</tr>
<tr>
<td>pY</td>
<td>Out</td>
<td>Pointer to the output array of n data elements</td>
</tr>
<tr>
<td>n</td>
<td>In</td>
<td>Length of the input and output arrays</td>
</tr>
</tbody>
</table>

3.1.3.3 Returns

The FIR filter computation generates output values, which are stored in the array, pointed to by pY.

3.1.4 FIR16Destroy

FIR16Destroy deallocates the data structure initially allocated by FIR16Create computation.

3.1.4.1 Call(s)

```c
void FIR16Destroy(struct tFir16Struct* pFIR);
```
3.1.4.2 Parameters

Table 3-3. FIR16Destroy Parameters

<table>
<thead>
<tr>
<th>pFIR</th>
<th>In</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Pointer to a data structure created by the FIR16Create function</td>
</tr>
</tbody>
</table>

3.1.4.3 Returns

Void.

3.1.5 Example of the FIR16 Function

```
{  
    Frac16 aCoef[iNUM_OF_COEF];
    Frac16 aX[iNUM_OF_SAMPLE];
    Frac16 aY[iNUM_OF_SAMPLE];
    struct tFir16Struct* pFIR;
    ...
    pFIR=FIR16Create(aCoef,iNUM_OF_COEF);
    ...
    FIR16(pFIR, aX, aY, iNUM_OF_SAMPLE);
    ...
    FIR16Destroy(pFIR);
}
```

3.1.6 Optimizations

Original code that uses MAC unit with one accumulator must fetch from memory coefficient for each input and output sample. Using the eMAC unit with four accumulators creates an opportunity to fetch coefficients only one time for four input or output values. Eight mac instructions (two instructions for each input sample) for one loop iteration can be performed instead of two.

Refer to Figure 3-1 and Figure 3-2.

```
.FORK3:
    cmp.l  d0, d2
    bcc .ENDFORk3
    move.l -(a1), d3
    mac.w  d3.u, d4.l, <<
    mac.w  d3.l, d4.u, <<, (a3)+, d4
    addq.l #2, d2
    bra .FORk3
.ENDORk3:
```

Figure 3-1. Optimized for MAC Unit
Assembly Code (FIR16)
```
move.w (a3)+, d4
move.w d2, d3
move.w -(a4), d2
swap d2
swap d3
mac.w d4.l, d2.u, <<, ACC0
mac.w d4.l, d2.l, <<, ACC1
mac.w d4.l, d3.l, <<, ACC3
subq #1, d5
beq EndIn1E

.ForIn1EBeg:
  move.l (a3)+, d4
  .ForIn1E:
    subq.l #2, d5
    blt EndIn1E
mac.w d4.u, d2.u, <<, ACC1
mac.w d4.u, d2.l, <<, ACC2
mac.w d4.u, d3.u, <<, ACC3
move.l d2, d3
move.l -(a4), d2
mac.w d4.u, d2.l, <<, ACC0
mac.w d4.l, d2.u, <<, ACC0
mac.w d4.l, d2.l, <<, ACC1
mac.w d4.l, d3.u, <<, ACC2
mac.w d4.l, d3.l, <<, (a3)+, d4, ACC3
bra .ForIn1E
.EndIn1E:
```

Figure 3-2. Optimized for eMAC Unit
Assembly Code (FIR16)
3.1.7 Observations

Due to the limited range of fractional numbers, FIR coefficients shouldn’t be numbers above 1 or below –1.

In some cases, the output can have an amplitude greater to the input (over-impulse). In such case, overflows can happen and it is recommended to reduce the amplitude of the input signal.

The use of eMAC or MAC is specified in the fir.h header file by the following line:

```
#define __EMAC
```

3.2 FIR32 Functions

These functions are used to filter an input signal of Frac32 numbers using the coefficients of a finite impulse filter.

3.2.1 tFir32Struct Structure

tFir32Struct is a structure containing data needed by the FIR32 function.

```
typedef struct tFir32Struct{
  Frac32* pFirCoef;
  unsigned int iFirCoefCount;
  Frac32* pFirHistory;
  unsigned int iFirHistoryCount;
};
```

pFirCoef is a pointer to an array of FIR filter coefficients.

iFirCoefCount is a number of elements in filter coefficients’ array.

pFirHistory is a pointer to the history buffer of the FIR filter. The history buffer contains the last iFirCoefCount - 1 input samples from the previous call of the FIR32 function. These last iFirCoefCount - 1 input samples are used in computations of the first iFirCoefCount –1 results of the next call of the FIR32 function. The history buffer is not used during the first call of the FIR32 function.

iFirHistoryCount is a number of elements in the history buffer. It equals 0 during the first calling of the FIR32 function and iFirCoefCount–1 during next calls.

3.2.2 FIR32Create

FIR32Create performs the initialization for the FIR32 filter function. FIR32Create allocates and initializes a data structure, which is used by FIR to preserve the filter’s state between calls. The data structure preserves a copy of the array of FIR filter coefficients. FIR32Create allocates a buffer to save the past history of N-1 data elements required by the FIR filter computation.

3.2.2.1 Call(s)

```
struct tFir32Struct* FIR32Create(Frac32* pCoef, unsigned int N);
```
3.2.2.2 Parameters

Table 3-4. FIR32Create Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pCoef</td>
<td>In</td>
<td>Pointer to a array of FIR filter coefficients {a0, a1, a2...}</td>
</tr>
<tr>
<td>N</td>
<td>In</td>
<td>Length of the array of FIR filter coefficients pointed to by pCoef; Must be less or equal length of the input and output arrays and more than one.</td>
</tr>
</tbody>
</table>

3.2.2.3 Returns

The FIR32Create function returns a pointer to tFir32Struct data structure that it allocated if all allocations and initializations succeed. This pointer must then be passed to subsequent calls of the FIR32 function. If insufficient data memory is available, the FIR32Create function returns 0 (NULL).

3.2.3 FIR32

FIR32 computes a finite impulse response (FIR) filter for an array of fractional data values. Prior to any call to FIR32, the FIR filter must be initialized via a call to FIR32Create; the FIR filter uses coefficients passed to that FIR32Create call. FIR32 uses the private data structure established by FIR32Create to maintain the past history of data elements required by the FIR filter computation.

3.2.3.1 Call(s)

void FIR32(struct tFir32Struct *pFIR, Frac32* pX, Frac32* pY, unsigned int n);

3.2.3.2 Parameters

Table 3-5. FIR32 Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pFIR</td>
<td>In</td>
<td>Pointer to a data structure containing private data for the FIR filter; this pointer is created by a call of FIR16Create</td>
</tr>
<tr>
<td>pX</td>
<td>In</td>
<td>Pointer to the input array of n data elements</td>
</tr>
<tr>
<td>pY</td>
<td>Out</td>
<td>Pointer to the output array of n data elements</td>
</tr>
<tr>
<td>n</td>
<td>In</td>
<td>Length of the input and output arrays</td>
</tr>
</tbody>
</table>

3.2.3.3 Returns

The FIR filter computation generates output values, which are stored in the array, pointed to by pY.

3.2.4 FIR32Destroy

FIR32Destroy deallocates the data structure initially allocated by FIR32Create.

3.2.4.1 Call(s)

void FIR32Destroy(struct tFir32Struct* pFIR);
3.2.4.2 Parameters

Table 3-6. FIR32Destroy Parameters

| pFIR  | In | Pointer to a data structure created by the FIR32Create function |

3.2.4.3 Returns

Void.

3.2.5 Example of FIR32 Function

```c
{
    Frac32 aCoef[iNUM_OF_COEF];
    Frac32 aX[iNUM_OF_SAMPLE];
    Frac32 aY[iNUM_OF_SAMPLE];
    struct tFir32Struct* pFIR;
    ...
    pFIR=FIR32Create(aCoef,iNUM_OF_COEF);
    ...
    FIR32(pFIR, aX, aY, iNUM_OF_SAMPLE);
    ...
    FIR32Destroy(pFIR);
}
```

3.2.6 Optimizations

The MAC unit has only one accumulator, so original code computes only one output sample per iteration of outer loop. Each outer loop has one inner loop in which only one mac instruction is performed, and one separate instruction to fetch operand from memory must be present. The eMAC unit has four accumulators, so optimized code computes four output samples per each iteration of outer loop. Each outer loop has one inner loop in which multiplications of input samples on corresponding coefficients are performed. Each inner loop has 16 mac instructions per iteration. Fetching operands from memory is executed at the same time with multiplication and there is no need to add separate instruction for this purpose.

**NOTE**

If the number of coefficients is less than 4, no optimization is performed, and execution time decreases only in cases where the number of coefficients is 4 or more.

MAC is a unit optimized for 16x16 multiplication, so original code uses only 16 upper bits of operands to perform the multiplication, although input and output operands are 32 bits long. eMAC is optimized for 32x32 multiplication, so in optimized code, all 32 bits of operands are used. Therefore, precision of computations increases.

Refer to Figure 3-3 and Figure 3-4.
### Figure 3-3. Optimized for MAC Unit
**Assembly Code (FIR32)**

```
.FORK3:
    move.l  -(a1),d3
    mac.w   d3.u,d4.u,<<,(a3)+,d4,ACC0
    addq.l  #1,d2
    cmp.l   d0, d2
    bcs     .FORk3
```

### Figure 3-4. Optimized for eMAC Unit
**Assembly Code (FIR32)**

```
.FORK4:
    cmp.l   d0, d2
    bhi     .ENDFORK4
    mac.l   a6,d5,<<,-(a1),d5,ACC3
    mac.l   a6,d4,<<,ACC2
    mac.l   a6,d3,<<,ACC1
    mac.l   a6,d6,<<,(a3)+,a6,ACC0
    addq.l  #4,d2
    bra     .FORk4
```

NXP Finite Impulse Filter (FIR)

Digital Signal Processing Libraries Using the ColdFire eMAC and Mac, Rev. 1.2

Freescale Semiconductor
3.2.7 Observations

Due to the limited range of fractional numbers, FIR coefficients shouldn’t be numbers above 1 or below –1.

In some cases, the output can have an amplitude greater to the input (over impulse). In such cases, overflow can happen and it is recommended to reduce the amplitude of the input signal.

The use of eMAC or MAC is specified in the fir.h header file by the following line:

```
#define __EMAC
```
Chapter 4  
Infinite Impulse Filter (IIR)

4.1 IIR16 Functions

These functions are used to filter an input signal of Frac16 numbers using the coefficients of a infinite impulse filter.

4.1.1 tIir16Struct Structure

tIir16Struct is a structure, containing data, needed by the IIR16 function.

```c
typedef struct tIir16Struct {
    Frac16* pIirCoef;
    unsigned int IIirCoefCount;
    Frac16* pIirHistory;
    unsigned int iIirHistoryCount;
} ;
```

pIirCoef is a pointer to an array of IIR filter coefficients.

iIirCoefCount is a number of elements in filter coefficients’ array.

pIirHistory is a pointer to the history buffer of the IIR filter. The history buffer contains the last iIirCoefCount / 2 + 1 input samples and iIirCoefCount / 2 computed samples from the previous call of the IIR16 function, which are used in computations of the first iIirCoefCount / 2 + 1 results of the next call of the IIR16 function. The history buffer is not used in the first call of the IIR16 function.

iIirHistoryCount is a number of elements in the history buffer. It equals 0 during the first calling of the IIR16 function and iIirCoefCount - 1 during next calls.

4.1.2 IIR16Create

IIR16Create performs the initialization for the IIR16 filter function. IIR16Create allocates and initializes a data structure, which is used by Iir to preserve the filter’s state between calls. The data structure preserves a copy of the array of IIR filter coefficients. IIR16Create allocates a buffer to save the past history of N-1 data elements required by the IIR filter computation.

4.1.2.1 Call(s)

```c
struct tIir16Struct* IIR16Create(Frac16* pCoef, unsigned int N);
```
4.1.2.2 Parameters

Table 4-1. IIR16Create Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pCoef</td>
<td>Pointer to a array of IIR filter coefficients {a0, a1, b1, a2, b2, ...}</td>
</tr>
<tr>
<td>N</td>
<td>Length of the array of IIR filter coefficients pointed to by pCoef; N=2n+1 (n=1,2,3..); Must be less or equal length of the input and output arrays and more than tree.</td>
</tr>
</tbody>
</table>

4.1.2.3 Returns

The IIR16Create function returns a pointer to tIir16Struct data structure that it allocated if all allocations and initializations succeed. This pointer must then be passed to subsequent calls of the IIR16 function. If insufficient data memory is available, the IIR16Create function returns 0 (NULL).

4.1.3 IIR16

IIR16 computes an infinite impulse response (IIR) filter for an array of fractional data values. Prior to any call to IIR16, the IIR filter must be initialized via a call to IIR16Create; the IIR filter uses coefficients passed to that IIR16Create call. IIR16 uses the private data structure established by IIR16Create to maintain the past history of data elements required by the IIR filter computation.

4.1.3.1 Call(s)

```c
void IIR16(struct iIir16Struct* pIIR, Frac16* pX, Frac16* pY, unsigned int n);
```

4.1.3.2 Parameters

Table 4-2. IIR16Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pIIR</td>
<td>Pointer to a data structure containing private data for the IIR filter; this pointer is created by a call of IIR16Create</td>
</tr>
<tr>
<td>pX</td>
<td>Pointer to the input array of n data elements</td>
</tr>
<tr>
<td>pY</td>
<td>Pointer to the output array of n data elements</td>
</tr>
<tr>
<td>n</td>
<td>Length of the input and output arrays</td>
</tr>
</tbody>
</table>

4.1.3.3 Returns

The IIR filter computation generates output values, which are stored in the array, pointed to by pY.

4.1.4 IIR16Destroy

IIR16Destroy deallocates the data structure initially allocated by IIR16Create computation.

4.1.4.1 Call(s)

```c
void IIR16Destroy(struct tIir16Struct* pIIR);
```
4.1.4.2 Parameters

Table 4-3. IIR16Destroy Parameters

<table>
<thead>
<tr>
<th>pIIR</th>
<th>In</th>
<th>Pointer to a data structure created by the IIR16Create function</th>
</tr>
</thead>
</table>

4.1.4.3 Returns

Void.

4.1.5 Example of the IIR16 Function

```c
{
    Frac16 aCoef[iNUM_OF_COEF];
    Frac16 aX[iNUM_OF_SAMPLE];
    Frac16 aY[iNUM_OF_SAMPLE];
    struct tIir16Struct* pIIR;
    ...
    pIIR=IIR16Create(aCoef,iNUM_OF_COEF);
    ...
    IIR16(pIIR, aX, aY, iNUM_OF_SAMPLE);
    ...
    IIR16Destroy(pIIR);
}
```

4.1.6 Optimizations

Original code that uses MAC unit with one accumulator must fetch from memory coefficient for each input and output sample. Using eMAC unit with four accumulators creates an opportunity to fetch coefficients only one time for four input or output values. So 16 mac instructions (two instructions for each input sample and two for each output) for one loop iteration can be performed instead of two.

```
.FORK3:
    cmp.l   d0,d2
    bcc    .ENDFORk3
    move.l   -(a1),d3
    mac.w   d3.1,d4.u,<<
    mac.w   -(a5),d3
    mac.w   d3.1,d4.1,<,<,(a3)+,d4
    addq.l   #1,d2
    bra    .FORk3
.ENDFORk3:
```

Figure 4-1. Optimized for MAC Unit
Assembly Code (IIR16)
move.l  (a3)+, d4
move.w  d2, d3
move.w  -(a4), d2
swap   d2
swap   d3
mac.w  d0, d1
mac.w  -(a5), d0
swap   d0
swap   d1
mac.w  d4.u, d2.u, <<, ACC0
mac.w  d4.u, d2.1, <<, ACC1
mac.w  d4.u, d2.u, <<, ACC2
mac.w  d4.u, d3.1, <<, ACC3
mac.w  d4.1, d2.u, <<, ACC0
mac.w  d4.1, d2.1, <<, ACC1
mac.w  d4.1, d2.u, <<, ACC2
mac.w  d4.1, d3.1, <<, ACC3
subq   #1, d5
beq    EndIn1E
ForIn1E:
  subq.l  #2, d5
  blt    EndIn1E
  mac.w  d4.u, d2.u, <<, ACC1
  mac.w  d4.u, d2.1, <<, ACC2
  mac.w  d4.u, d3.u, <<, ACC3
  mac.w  d4.1, d0.u, <<, ACC1
  mac.w  d4.1, d0.1, <<, ACC2
  mac.w  d4.1, d1.u, <<, ACC3
move.l  d2, d3
move.l  -(a4), d2
move.l  d0, d1
move.l  -(a5), d0
mac.w  d4.u, d2.1, <<, ACC0
mac.w  d4.1, d0.1, <<, ACC0
move.l  (a3)+, d4
mac.w  d4.u, d2.u, <<, ACC0
mac.w  d4.u, d2.1, <<, ACC1
mac.w  d4.u, d3.1, <<, ACC2
mac.w  d4.u, d3.1, <<, ACC3
mac.w  d4.1, d0.u, <<, ACC0
mac.w  d4.1, d0.1, <<, ACC1
mac.w  d4.1, d1.u, <<, ACC2
mac.w  d4.1, d1.1, <<, (a3)+, d4, ACC3
bra    .ForIn1E
.EndIn1E:

Figure 4-2. Optimized for eMAC Unit
Assembly Code (IIR16)
4.1.7 Observations

Due to the limited range of fractional numbers, IIR coefficients shouldn’t be numbers above 1 or below –1.

In some cases, the output can have an amplitude greater to the input (over-impulse). In such case, overflows can happen and is recommended to reduce the amplitude of the input signal.

The use of eMAC or MAC is specified in the iir.h header file by the following line:

```c
#define __EMAC
```

4.2 IIR32 Functions

These functions are used to filter an input signal of Frac32 numbers using the coefficients of an Infinite impulse filter.

4.2.1 tFir32Struct Structure

`tIir32Struct` is a structure, containing data, needed by the IIR32 function.

```c
typedef struct tIir32Struct {
    Frac32* pIirCoef;
    unsigned int  iIirCoefCount;
    Frac32* pIirHistory;
    unsigned int    iIirHistoryCount;
} tIir32Struct;
```

`pIirCoef` is a pointer to an array of IIR filter coefficients.

`iIirCoefCount` is a number of elements in filter coefficients’ array.

`pIirHistory` is a pointer to the history buffer of the IIR filter. The history buffer contains the last `iIirCoefCount/2 + 1` input samples and `iIirCoefCount/2` computed output samples from the previous call of the IIR32 function, which are used in computations of the first `iIirCoefCount/2 + 1` results of the next call of the IIR32 function. The history buffer is not used during the first call of the IIR32 function.

`iIirHistoryCount` is a number of elements in the history buffer. It equals 0 during the first calling of the FIR32 function and `iIirCoefCount -1` during next calls.

4.2.2 IIR32Create

IIR32Create performs the initialization for the IIR32 filter function. IIR32Create allocates and initializes a data structure, which is used by IIR to preserve the filter’s state between calls. The data structure preserves a copy of the array of IIR filter coefficients. IIR32Create allocates a buffer to save the past history of N-1 data elements required by the IIR filter computation.

4.2.2.1 Call(s)

```c
struct tIir32Struct* IIR32Create( Frac32* pCoef, unsigned int N);
```
4.2.2.2 Parameters

Table 4-4. IIR32Create Parameters

<table>
<thead>
<tr>
<th>pCoef</th>
<th>In</th>
<th>Pointer to a array of IIR filter coefficients; pIirCoef -&gt; {a0,a1,b1,a2,b2...}; ax are the coefficients for input samples, bx are the coefficients for output samples.</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>In</td>
<td>Length of the array of IIR filter coefficients pointed to by pCoef; N=2k+1 (k=1,2,3,...); k must be less or equal length of the input and output arrays (n)</td>
</tr>
</tbody>
</table>

4.2.2.3 Returns

The IIR32Create function returns a pointer to tIir32Struct data structure that it allocated if all allocations and initializations succeed. This pointer must then be passed to subsequent calls of the IIR32 function.

If insufficient data memory is available, the IIR32Create function returns 0 (NULL).

4.2.3 IIR32

IIR32 computes an infinite impulse response (IIR) filter for an array of fractional data values. Prior to any call to IIR32, the IIR filter must be initialized via a call to IIR32Create; the IIR filter uses coefficients passed to that IIR32Create call. IIR32 uses the private data structure established by IIR32Create to maintain the past history of data elements required by the IIR filter computation.

4.2.3.1 Call(s)

    void IIR32(struct tIir32Struct *pIIR, Frac32* pX, Frac32* pY, unsigned int n);

4.2.3.2 Parameters

Table 4-5. IIR32 Parameters

<table>
<thead>
<tr>
<th>pIFIR</th>
<th>In</th>
<th>Pointer to a data structure containing private data for the IIR filter; this pointer is created by a call of IIR32Create</th>
</tr>
</thead>
<tbody>
<tr>
<td>pX</td>
<td>In</td>
<td>Pointer to the input array of n data elements</td>
</tr>
<tr>
<td>pY</td>
<td>Out</td>
<td>Pointer to the output array of n data elements</td>
</tr>
<tr>
<td>n</td>
<td>In</td>
<td>Length of the input and output arrays</td>
</tr>
</tbody>
</table>

4.2.3.3 Returns

The IIR filter computation generates n output values, which are stored in the array, pointed to by pY.

4.2.4 IIR32Destroy

IIR32Destroy deallocates the data structure initially allocated by IIR32Create.

4.2.4.1 Call(s)

    void IIR32Destroy( struct tIir32Struct* pIIR);
4.2.4.2 Parameters

Table 4-6. IIR32Destroy Parameters

| pIIR | In   | Pointer to a data structure created by the IIR32Create function |

4.2.5 Returns

Void.

4.2.6 Example of the IIR32 Function

```c
{
    Frac32 aCoef[NUM_OF_COEF];
    Frac32 aX[NUM_OF_SAMPLE];
    Frac32 aY[NUM_OF_SAMPLE];
    struct tIir32Struct* pIIR;
    ...
    pIIR=IIR32Create(aCoef,NUM_OF_COEF);
    ...
    IIR32( pIIR, aX, aY, NUM_OF_SAMPLE);
    ...
    IIR32Destroy(pIIR);
}
```

4.2.7 Optimizations

MAC unit has only one accumulator, so original code computes only one output sample per iteration of outer loop. Each outer loop has one inner loop in which two mac instructions are performed and two separate instructions to fetch operands from memory must be present. eMAC unit has four accumulators so optimized code computes four output samples per each iteration of outer loop. Each outer loop has two inner loops. Multiplications of input samples on corresponding coefficients are performed in the first loop. Multiplications of output samples on corresponding coefficients are performed in the second loop. Each inner loop has 16 mac instructions per iteration. Fetching operands from memory is executed at the same time with multiplication, and there is no need to add separate instruction for this purpose.

**NOTE**

If the number of coefficients is less than 7, no optimization is performed, and execution time decreases only in cases where the number of coefficients is 7 or more.

MAC is a unit optimized for 16x16 multiplication, so original code uses only 16 upper bits of operands to perform the multiplication, although input and output operands are 32 bits long. eMAC is optimized for 32x32 multiplication, so in optimized code, all 32 bits of operands are used. Therefore precision of computations increases.

Optimized for MAC unit assembly code (see Figure 4-3 and Figure 4-4).
Infinite Impulse Filter (IIR)

Digital Signal Processing Libraries Using the ColdFire eMAC and Mac, Rev. 1.2

.FORK1:
    cmp.l  d1,d2
    bcc .ENDFORk1
    move.l  -(a1),d3
    mac.w  d3.u,d4.u,<<,(a3)+,d4,ACC0
    mac.w  -(a5),d3
    mac.w  d3.u,d4.u,<<,(a3)+,d4,ACC0
    addq.l  #1,d2
    bra .FORk1
.ENDFORk1:

Figure 4-3. Optimized for MAC Unit
Assembly Code (IIR32)

.FORK1:
    cmp.l  d1,d2
    bcc .ENDFORk1
    adda.l  #4,a3
    mac.l  a6,d5,<<,-(a1),d5,ACC3
    mac.l  a6,d4,<<,ACC2
    mac.l  a6,d3,<<,ACC1
    mac.l  a6,d6,<<,(a3)+,a6,ACC0
    adda.l  #4,a3
    mac.l  a6,d4,<<,-(a1),d4,ACC3
    mac.l  a6,d3,<<,ACC2
    mac.l  a6,d6,<<,ACC1
    mac.l  a6,d5,<<,(a3)+,a6,ACC0
    add1  #4,a3
    mac.l  a6,d3,<<,-(a1),d3,ACC3
    mac.l  a6,d6,<<,ACC2
    mac.l  a6,d5,<<,ACC1
    mac.l  a6,d4,<<,(a3)+,a6,ACC0
    adda.l  #4,a3
    mac.l  a6,d6,<<,-(a1),d6,ACC3
    mac.l  a6,d5,<<,ACC2
    mac.l  a6,d4,<<,ACC1
    mac.l  a6,d3,<<,(a3)+,a6,ACC0
    addq.l  #4,d3
    bra .FORk1
.ENDFORk1

Figure 4-4. Optimized for eMAC Unit
Assembly Code (IIR32)
4.2.8 Observations

Due to the limited range of fractional numbers, IIR coefficients shouldn’t be numbers above 1 or below –1. In some cases, the output can have an amplitude greater to the input (over impulse). In such case, overflows can happen, and it is recommended to reduce the amplitude of the input signal.

The use of eMAC or MAC is specified in the iir.h header file by the following line:

```c
#define __EMAC
```
Infinite Impulse Filter (IIR)
RoHS-compliant and/or Pb-free versions of Freescale products have the functionality and electrical characteristics of their non-RoHS-compliant and/or non-Pb-free counterparts. For further information, see http://www.freescale.com or contact your Freescale sales representative.

For information on Freescale’s Environmental Products program, go to http://www.freescale.com/epp.

Information in this document is provided solely to enable system and software implementers to use Freescale Semiconductor products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits or integrated circuits based on the information in this document.

Freescale Semiconductor reserves the right to make changes without further notice to any products herein. Freescale Semiconductor makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale Semiconductor assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. “Typical” parameters that may be provided in Freescale Semiconductor data sheets and/or specifications can and do vary in different applications and actual performance may vary over time. All operating parameters, including “Typicals”, must be validated for each customer application by customer's technical experts. Freescale Semiconductor does not convey any license under its patent rights nor the rights of others. Freescale Semiconductor products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Freescale Semiconductor product could create a situation where personal injury or death may occur. Should Buyer purchase or use Freescale Semiconductor products for any such unintended or unauthorized application, Buyer shall indemnify and hold Freescale Semiconductor and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Freescale Semiconductor was negligent regarding the design or manufacture of the part.

Freescale™ and the Freescale logo are trademarks of Freescale Semiconductor, Inc. All other product or service names are the property of their respective owners. © Freescale Semiconductor, Inc. 2006. All rights reserved.