Embedded SDK
(Software Development Kit)
Digital Signal Processing (DSP) Function Library

SDK107/D
Rev. 3, 10/08/2002
# Contents

## About This Document

- Audience ................................................................. xvii
- Organization ............................................................. xvii
- Suggested Reading ..................................................... xviii
- Conventions ............................................................. xviii
- Definitions, Acronyms, and Abbreviations ....................... xix
- References ................................................................... xx

## Chapter 1
### Introduction

- 1.1 Digital Signal Processing Algorithms .......................... 1-1
- 1.2 Advantages ............................................................. 1-2
- 1.3 Quick Start ............................................................. 1-2
- 1.3.1 Creating a New Project .......................................... 1-3
- 1.3.2 Initializing the DSP Function Library ......................... 1-6
- 1.3.3 Using the DSP Function Library Routines .................... 1-7

## Chapter 2
### Target Architectures

- 2.1 DSP56800 Family ..................................................... 2-1
- 2.1.1 Fractional Data Representation ............................... 2-1
- 2.1.2 Fractional Arithmetic ............................................ 2-3
- 2.1.3 Accumulators ....................................................... 2-3
- 2.1.4 Data Limiting ....................................................... 2-4
- 2.1.5 Saturation ............................................................ 2-4
- 2.1.6 Limit Bit .............................................................. 2-5
- 2.1.7 Rounding ............................................................ 2-5
- 2.1.8 Internal vs External Memory ................................. 2-5
- 2.1.9 DSP Function Library Implementation ...................... 2-6

## Chapter 3
### CodeWarrior C

- 3.1 Frac16 vs __fixed__ ................................................... 3-1
- 3.2 Basic Types in port.h ................................................. 3-2
- 3.3 Constants ............................................................... 3-4
- 3.4 Data Storage ........................................................... 3-4
- 3.5 Intrinsics ................................................................. 3-5
- 3.6 Calling Conventions .................................................. 3-5
- 3.6.1 DSP56800 Calling Conventions ............................... 3-5
# Chapter 4
Software Development Kit

4.1 SDK Directory Structure ............................................. 4-1  
4.2 appconfig.h .......................................................... 4-3  
4.3 CodeWarrior Project .................................................. 4-4

# Chapter 5
Include and Library Files

5.1 Include Files .......................................................... 5-1  
5.2 dspfunc.h .............................................................. 5-2  
5.3 dspfunc.lib ........................................................... 5-3  
5.4 Rebuilding dspfunc.lib .............................................. 5-3  
5.5 Sample Code ........................................................... 5-3

# Chapter 6
Basic Fractional Math Library

6.1 Introduction ........................................................... 6-1  
6.2 16-bit Fractional Math Interface - mfr16.h ......................... 6-1  
6.3 32-bit Fractional Math Interface - mfr32.h ......................... 6-2  
6.4 Basic Fractional Math Function Specifications ...................... 6-4  
6.4.1 abs ................................................................. 6-5  
6.4.2 add ................................................................. 6-6  
6.4.3 div ................................................................. 6-7  
6.4.4 deposit ............................................................ 6-8  
6.4.5 extract ............................................................. 6-9  
6.4.6 mac ............................................................... 6-10  
6.4.7 msu ............................................................... 6-11  
6.4.8 mult ............................................................... 6-12  
6.4.9 negate ............................................................. 6-13  
6.4.10 norm ............................................................. 6-14  
6.4.11 rand .............................................................. 6-15  
6.4.12 randseed ......................................................... 6-16  
6.4.13 round .............................................................. 6-17  
6.4.14 shl ............................................................... 6-18  
6.4.15 shr ............................................................... 6-19  
6.4.16 sqrt .............................................................. 6-20  
6.4.17 sub ............................................................... 6-21

# Chapter 7
Trigonometric Math Library

7.1 Introduction ........................................................... 7-1  
7.2 16-bit Fractional Interface .......................................... 7-1  
7.3 Fractional Trigonometric Function Specs ............................ 7-6  
7.3.1 AcosOverPI .......................................................... 7-7  
7.3.2 AsinOverPI .......................................................... 7-8
MOTOROLA

For More Information On This Product,
Go to: www.freescale.com

Chapter 8
Array Library

8.1 Introduction ......................................................... 8-1
8.2 16-bit Fractional Interface ..................................... 8-1
8.3 32-bit Fractional Interface ..................................... 8-2
8.4 Array Function Specifications ............................... 8-4
   8.4.1 abs .......................................................... 8-5
   8.4.2 add .......................................................... 8-6
   8.4.3 div .......................................................... 8-7
   8.4.4 equal ...................................................... 8-9
   8.4.5 mac ......................................................... 8-10
   8.4.6 max ........................................................ 8-12
   8.4.7 min ........................................................ 8-13
   8.4.8 msu ......................................................... 8-14
   8.4.9 mult ......................................................... 8-16
   8.4.10 negate ................................................... 8-17
   8.4.11 rand ..................................................... 8-18
   8.4.12 round .................................................... 8-19
   8.4.13 sqrt ..................................................... 8-20
   8.4.14 sub ....................................................... 8-21

Chapter 9
Vector Library

9.1 Introduction ........................................................ 9-1
9.2 16-bit Fractional Interface ..................................... 9-1
9.3 Vector Function Specifications .............................. 9-3
   9.3.1 add ........................................................ 9-4
   9.3.2 dotProd .................................................... 9-5
   9.3.3 equal ...................................................... 9-6
   9.3.4 length ..................................................... 9-7
   9.3.5 mult ....................................................... 9-8
   9.3.6 scale ....................................................... 9-9
   9.3.7 sub ........................................................ 9-10

Table of Contents
Chapter 10
Matrix Library

10.1 Introduction ................................................................. 10-1
10.2 16-bit Fractional Interface ........................................... 10-1
10.3 Matrix Function Specifications ....................................... 10-3
    10.3.1 add ................................................................. 10-4
    10.3.2 det (determinant) ............................................... 10-5
    10.3.3 equal ............................................................ 10-6
    10.3.4 inv (inverse) ................................................... 10-7
    10.3.5 mult (multiply) ................................................ 10-8
    10.3.6 sub (subtract) ................................................. 10-10
    10.3.7 trans (transpose) .............................................. 10-12

Chapter 11
Signal Processing Library

11.1 Introduction ................................................................. 11-1
11.2 16-bit Fractional Interface ........................................... 11-1
11.3 Signal Processing Function Specifications ......................... 11-6
    11.3.1 autoCorr - Auto-Correlation .................................. 11-7
    11.3.2 cbittrev - Complex Bit Reverse ................................ 11-9
    11.3.3 cfft - Complex Fast Fourier Transform ...................... 11-10
    11.3.4 cfftCreate - Create Complex FFT ............................ 11-13
    11.3.5 cfftDestroy - Destroy Complex FFT ......................... 11-15
    11.3.6 cfftInit - Initialize Complex FFT ............................ 11-16
    11.3.7 ciff - Complex Inverse Fast Fourier Transform ............ 11-19
    11.3.8 ciffCreate - Create Complex Inverse FFT ................... 11-22
    11.3.9 ciffDestroy - Destroy Complex Inverse FFT ................ 11-24
    11.3.10 ciffInit - Initialize Complex Inverse FFT ................. 11-25
    11.3.11 rfft - Real Fast Fourier Transform ......................... 11-28
    11.3.12 rfftCreate - Create Real FFT ................................ 11-31
    11.3.13 rfftDestroy - Destroy Real FFT ............................ 11-33
    11.3.14 rfftInit - Initialize Real FFT .............................. 11-34
    11.3.15 rifft - Real Inverse Fast Fourier Transform ............... 11-36
    11.3.16 rifftCreate - Create Real Inverse FFT .................... 11-39
    11.3.17 rifftDestroy - Destroy Real Inverse FFT ................... 11-40
    11.3.18 corr - Correlation ........................................... 11-41
    11.3.19 fir - Finite Impulse Response Filter ....................... 11-43
    11.3.20 firs - FIR Filter for a Single Input ........................ 11-46
    11.3.21 firCreate - Create Finite Impulse Response Filter ........ 11-48
    11.3.22 firDestroy - Destroy Finite Impulse Response Filter .... 11-50
    11.3.23 firHistory - Reinitialize FIR History Buffer .............. 11-51
    11.3.24 firInit - Initialize Finite Impulse Response Filter ...... 11-53
    11.3.25 firdec - Decimating FIR Filter ............................ 11-55
    11.3.26 firdecCreate - Create Decimating FIR Filter .............. 11-57
    11.3.27 firdecDestroy - Destroy Decimating FIR Filter .......... 11-58
    11.3.28 firdecInit - Initialize Decimating FIR Filter ............. 11-59
11.3.29 firint - Interpolating FIR Filter ........................................ 11-61
11.3.30 firintCreate - Create Interpolating FIR Filter ....................... 11-62
11.3.31 firintDestroy - Destroy Interpolating FIR Filter ..................... 11-63
11.3.32 firintInit - Initialize Interpolating FIR Filter ....................... 11-64
11.3.33 iir - Infinite Impulse Response Filter .................................. 11-66
11.3.34 iirCreate - Create IIR Filter ........................................... 11-69
11.3.35 iirDestroy - Destroy IIR Filter .......................................... 11-71
11.3.36 iirInit - Initialize IIR Filter ........................................... 11-72

Chapter 12
Performance

12.1 DSP56824 Performance ................................................. 12-1
12.2 Trigonometric Math Function Performance .............................. 12-1
12.3 Signal Processing Function Performance .............................. 12-1
12.3.1 autoCorr .......................................................... 12-2
12.3.2 cbitrev ........................................................... 12-2
12.3.3 cfft ............................................................... 12-2
12.3.4 cifft .............................................................. 12-3
12.3.5 corr ............................................................... 12-4
12.3.6 fir ................................................................. 12-4
12.3.7 iir ................................................................. 12-4
12.3.8 rfft ............................................................... 12-5
12.3.9 rffft ............................................................. 12-6

Chapter 13
License

13.1 Limited Use License Agreement ........................................ 13-1
# List of Tables

<table>
<thead>
<tr>
<th>Table 2-1</th>
<th>Interpreting 16-bit Data Values</th>
<th>2-2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 2-2</td>
<td>Interpreting 32-bit Data Values</td>
<td>2-3</td>
</tr>
<tr>
<td>Table 2-3</td>
<td>Maximum/Minimum Saturated Values</td>
<td>2-5</td>
</tr>
<tr>
<td>Table 2-4</td>
<td>Default Saturation and Rounding Modes</td>
<td>2-7</td>
</tr>
<tr>
<td>Table 3-1</td>
<td>CodeWarrior Fractional Typedefs</td>
<td>3-1</td>
</tr>
<tr>
<td>Table 3-2</td>
<td>Data Storage for Long and Unsigned Long</td>
<td>3-4</td>
</tr>
<tr>
<td>Table 6-1</td>
<td>abs arguments</td>
<td>6-5</td>
</tr>
<tr>
<td>Table 6-2</td>
<td>add arguments</td>
<td>6-6</td>
</tr>
<tr>
<td>Table 6-3</td>
<td>div arguments</td>
<td>6-7</td>
</tr>
<tr>
<td>Table 6-4</td>
<td>deposit arguments</td>
<td>6-8</td>
</tr>
<tr>
<td>Table 6-5</td>
<td>extract arguments</td>
<td>6-9</td>
</tr>
<tr>
<td>Table 6-6</td>
<td>mac arguments</td>
<td>6-10</td>
</tr>
<tr>
<td>Table 6-7</td>
<td>msu arguments</td>
<td>6-11</td>
</tr>
<tr>
<td>Table 6-8</td>
<td>mult arguments</td>
<td>6-12</td>
</tr>
<tr>
<td>Table 6-9</td>
<td>negate arguments</td>
<td>6-13</td>
</tr>
<tr>
<td>Table 6-10</td>
<td>norm arguments</td>
<td>6-14</td>
</tr>
<tr>
<td>Table 6-11</td>
<td>rand arguments</td>
<td>6-15</td>
</tr>
<tr>
<td>Table 6-12</td>
<td>randseed arguments</td>
<td>6-16</td>
</tr>
<tr>
<td>Table 6-13</td>
<td>round arguments</td>
<td>6-17</td>
</tr>
<tr>
<td>Table 6-14</td>
<td>shl arguments</td>
<td>6-18</td>
</tr>
<tr>
<td>Table 6-15</td>
<td>shr arguments</td>
<td>6-19</td>
</tr>
<tr>
<td>Table 6-16</td>
<td>sqrt arguments</td>
<td>6-20</td>
</tr>
<tr>
<td>Table 6-17</td>
<td>sub arguments</td>
<td>6-21</td>
</tr>
<tr>
<td>Table 7-1</td>
<td>AcosOverPI arguments</td>
<td>7-7</td>
</tr>
<tr>
<td>Table 7-2</td>
<td>AsinOverPI arguments</td>
<td>7-8</td>
</tr>
<tr>
<td>Table 7-3</td>
<td>AtanOverPI arguments</td>
<td>7-9</td>
</tr>
<tr>
<td>Table 7-4</td>
<td>Atan2OverPI arguments</td>
<td>7-10</td>
</tr>
<tr>
<td>Table 7-5</td>
<td>CosPix arguments</td>
<td>7-11</td>
</tr>
<tr>
<td>Table 7-6</td>
<td>SinPix arguments</td>
<td>7-12</td>
</tr>
<tr>
<td>Table 7-7</td>
<td>SineWaveGenIDTL arguments</td>
<td>7-13</td>
</tr>
<tr>
<td>Table 7-8</td>
<td>SineWaveGenRDTL arguments</td>
<td>7-16</td>
</tr>
<tr>
<td>Table 7-9</td>
<td>SineWaveGenRDITL arguments</td>
<td>7-19</td>
</tr>
<tr>
<td>Table 7-10</td>
<td>SineWaveGenRDITLQ arguments</td>
<td>7-22</td>
</tr>
<tr>
<td>Table 7-11</td>
<td>SineWaveGenPAM arguments</td>
<td>7-25</td>
</tr>
<tr>
<td>Table 7-12</td>
<td>SineWaveGenDOM arguments</td>
<td>Table 7-13</td>
</tr>
<tr>
<td>-----------</td>
<td>--------------------------</td>
<td>------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Table 8-1</td>
<td>abs (array) arguments</td>
<td>Table 8-2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Table 8-5</td>
<td>mac (array) arguments</td>
<td>Table 8-6</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Table 8-9</td>
<td>mult (array) arguments</td>
<td>Table 8-10</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Table 8-13</td>
<td>sqrt (array) arguments</td>
<td>Table 8-14</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Table 9-3</td>
<td>equal (vector) arguments</td>
<td>Table 9-4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Table 9-7</td>
<td>sub (vector) arguments</td>
<td>Table 10-1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Table 10-4</td>
<td>inv (matrix) arguments</td>
<td>Table 10-5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Table 11-1</td>
<td>autoCorr arguments</td>
<td>Table 11-2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Table 11-5</td>
<td>cfft options</td>
<td>Table 11-6</td>
</tr>
<tr>
<td>Table 11-8</td>
<td>cfftDestroy arguments</td>
<td>11-15</td>
</tr>
<tr>
<td>Table 11-9</td>
<td>cfftInit arguments</td>
<td>11-16</td>
</tr>
<tr>
<td>Table 11-10</td>
<td>cfftCreate options</td>
<td>11-16</td>
</tr>
<tr>
<td>Table 11-11</td>
<td>cfft arguments</td>
<td>11-19</td>
</tr>
<tr>
<td>Table 11-12</td>
<td>cfft options</td>
<td>11-19</td>
</tr>
<tr>
<td>Table 11-13</td>
<td>cfftCreate arguments</td>
<td>11-22</td>
</tr>
<tr>
<td>Table 11-14</td>
<td>cfftCreate options</td>
<td>11-22</td>
</tr>
<tr>
<td>Table 11-15</td>
<td>cfftDestroy arguments</td>
<td>11-24</td>
</tr>
<tr>
<td>Table 11-16</td>
<td>cfftInit arguments</td>
<td>11-25</td>
</tr>
<tr>
<td>Table 11-17</td>
<td>cfftInit options</td>
<td>11-25</td>
</tr>
<tr>
<td>Table 11-18</td>
<td>rfft arguments</td>
<td>11-28</td>
</tr>
<tr>
<td>Table 11-19</td>
<td>rfft options</td>
<td>11-28</td>
</tr>
<tr>
<td>Table 11-20</td>
<td>rfftCreate</td>
<td>11-31</td>
</tr>
<tr>
<td>Table 11-21</td>
<td>rfftCreate options</td>
<td>11-31</td>
</tr>
<tr>
<td>Table 11-22</td>
<td>rfftDestroy arguments</td>
<td>11-33</td>
</tr>
<tr>
<td>Table 11-23</td>
<td>crftInit arguments</td>
<td>11-34</td>
</tr>
<tr>
<td>Table 11-24</td>
<td>rfftInit options</td>
<td>11-34</td>
</tr>
<tr>
<td>Table 11-25</td>
<td>rfft arguments</td>
<td>11-36</td>
</tr>
<tr>
<td>Table 11-26</td>
<td>rfft options</td>
<td>11-36</td>
</tr>
<tr>
<td>Table 11-27</td>
<td>crfftCreate arguments</td>
<td>11-39</td>
</tr>
<tr>
<td>Table 11-28</td>
<td>crfftCreate options</td>
<td>11-39</td>
</tr>
<tr>
<td>Table 11-29</td>
<td>crfftDestroy arguments</td>
<td>11-40</td>
</tr>
<tr>
<td>Table 11-30</td>
<td>corr arguments</td>
<td>11-41</td>
</tr>
<tr>
<td>Table 11-31</td>
<td>corr options</td>
<td>11-41</td>
</tr>
<tr>
<td>Table 11-32</td>
<td>fir arguments</td>
<td>11-43</td>
</tr>
<tr>
<td>Table 11-33</td>
<td>firs arguments</td>
<td>11-46</td>
</tr>
<tr>
<td>Table 11-34</td>
<td>firCreate arguments</td>
<td>11-48</td>
</tr>
<tr>
<td>Table 11-35</td>
<td>firDestroy arguments</td>
<td>11-50</td>
</tr>
<tr>
<td>Table 11-36</td>
<td>firHistory arguments</td>
<td>11-51</td>
</tr>
<tr>
<td>Table 11-37</td>
<td>firInit arguments</td>
<td>11-53</td>
</tr>
<tr>
<td>Table 11-38</td>
<td>firdec arguments</td>
<td>11-55</td>
</tr>
<tr>
<td>Table 11-39</td>
<td>firdecCreate arguments</td>
<td>11-57</td>
</tr>
<tr>
<td>Table 11-40</td>
<td>firdecDestroy arguments</td>
<td>11-58</td>
</tr>
<tr>
<td>Table 11-41</td>
<td>firdecInit arguments</td>
<td>11-59</td>
</tr>
<tr>
<td>Table 11-42</td>
<td>firint arguments</td>
<td>11-61</td>
</tr>
<tr>
<td>Table 11-43</td>
<td>firintCreate arguments</td>
<td>11-62</td>
</tr>
<tr>
<td>Table 11-44</td>
<td>firintDestroy</td>
<td>11-63</td>
</tr>
<tr>
<td>Table 11-45</td>
<td>firintInit arguments</td>
<td>11-64</td>
</tr>
<tr>
<td>Table 11-46</td>
<td>iir arguments</td>
<td>11-66</td>
</tr>
<tr>
<td>Table</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>------------</td>
<td>------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>11-47</td>
<td>iirCreate arguments</td>
<td>11-69</td>
</tr>
<tr>
<td>11-48</td>
<td>iirDestroy arguments</td>
<td>11-71</td>
</tr>
<tr>
<td>11-49</td>
<td>iirCreate arguments</td>
<td>11-72</td>
</tr>
<tr>
<td>12-1</td>
<td>Trigonometric Math Function Performance</td>
<td>12-1</td>
</tr>
<tr>
<td>12-2</td>
<td>autoCorr Performance</td>
<td>12-2</td>
</tr>
<tr>
<td>12-3</td>
<td>cbitrev Performance</td>
<td>12-2</td>
</tr>
<tr>
<td>12-4</td>
<td>cfft Performance</td>
<td>12-3</td>
</tr>
<tr>
<td>12-5</td>
<td>cifft Performance</td>
<td>12-3</td>
</tr>
<tr>
<td>12-6</td>
<td>corr Performance</td>
<td>12-4</td>
</tr>
<tr>
<td>12-7</td>
<td>iir Performance</td>
<td>12-5</td>
</tr>
<tr>
<td>12-8</td>
<td>rfft Performance</td>
<td>12-5</td>
</tr>
<tr>
<td>12-9</td>
<td>rifft Performance</td>
<td>12-6</td>
</tr>
</tbody>
</table>
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1-1</td>
<td>Select File/New</td>
<td>1-3</td>
</tr>
<tr>
<td>Figure 1-2</td>
<td>Select Stationery</td>
<td>1-4</td>
</tr>
<tr>
<td>Figure 1-3</td>
<td>Select Target Platform</td>
<td>1-4</td>
</tr>
<tr>
<td>Figure 1-4</td>
<td>Select Build Type</td>
<td>1-5</td>
</tr>
<tr>
<td>Figure 1-5</td>
<td>A New Project</td>
<td>1-5</td>
</tr>
<tr>
<td>Figure 1-6</td>
<td>Opening the appconfig.h file</td>
<td>1-6</td>
</tr>
<tr>
<td>Figure 1-7</td>
<td>Initializing the DSP Function Library</td>
<td>1-7</td>
</tr>
<tr>
<td>Figure 1-8</td>
<td>Create a New File</td>
<td>1-8</td>
</tr>
<tr>
<td>Figure 1-9</td>
<td>Enter Application Code</td>
<td>1-9</td>
</tr>
<tr>
<td>Figure 1-10</td>
<td>Save the Text File</td>
<td>1-10</td>
</tr>
<tr>
<td>Figure 1-11</td>
<td>Add File</td>
<td>1-11</td>
</tr>
<tr>
<td>Figure 1-12</td>
<td>Name File</td>
<td>1-11</td>
</tr>
<tr>
<td>Figure 1-13</td>
<td>Execute the Program</td>
<td>1-12</td>
</tr>
<tr>
<td>Figure 1-14</td>
<td>Debug the Code</td>
<td>1-13</td>
</tr>
<tr>
<td>Figure 4-1</td>
<td>SDK Directory Structure</td>
<td>4-2</td>
</tr>
<tr>
<td>Figure 4-2</td>
<td>Contents of an Example CodeWarrior Project</td>
<td>4-4</td>
</tr>
<tr>
<td>Figure 11-1</td>
<td>Biquad Structure with a 1Scaled</td>
<td>11-66</td>
</tr>
</tbody>
</table>
## List of Examples

<table>
<thead>
<tr>
<th>Code Example</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3-1</td>
<td>port.h</td>
<td>3-2</td>
</tr>
<tr>
<td>5-1</td>
<td>dspfunc.h</td>
<td>5-2</td>
</tr>
<tr>
<td>6-1</td>
<td>mfr16.h</td>
<td>6-1</td>
</tr>
<tr>
<td>6-2</td>
<td>mfr32.h</td>
<td>6-3</td>
</tr>
<tr>
<td>6-3</td>
<td>abs</td>
<td>6-5</td>
</tr>
<tr>
<td>6-4</td>
<td>add</td>
<td>6-6</td>
</tr>
<tr>
<td>6-5</td>
<td>div</td>
<td>6-7</td>
</tr>
<tr>
<td>6-6</td>
<td>deposit</td>
<td>6-8</td>
</tr>
<tr>
<td>6-7</td>
<td>extract</td>
<td>6-9</td>
</tr>
<tr>
<td>6-8</td>
<td>mac</td>
<td>6-10</td>
</tr>
<tr>
<td>6-9</td>
<td>msu</td>
<td>6-11</td>
</tr>
<tr>
<td>6-10</td>
<td>mult</td>
<td>6-12</td>
</tr>
<tr>
<td>6-11</td>
<td>negate</td>
<td>6-13</td>
</tr>
<tr>
<td>6-12</td>
<td>norm</td>
<td>6-14</td>
</tr>
<tr>
<td>6-13</td>
<td>rand</td>
<td>6-15</td>
</tr>
<tr>
<td>6-14</td>
<td>randseed</td>
<td>6-16</td>
</tr>
<tr>
<td>6-15</td>
<td>round</td>
<td>6-17</td>
</tr>
<tr>
<td>6-16</td>
<td>shl</td>
<td>6-18</td>
</tr>
<tr>
<td>6-17</td>
<td>shr</td>
<td>6-19</td>
</tr>
<tr>
<td>6-18</td>
<td>sqrt</td>
<td>6-20</td>
</tr>
<tr>
<td>6-19</td>
<td>sub</td>
<td>6-21</td>
</tr>
<tr>
<td>7-1</td>
<td>C Header File tfr16.h</td>
<td>7-1</td>
</tr>
<tr>
<td>7-2</td>
<td>AcosOverPI</td>
<td>7-7</td>
</tr>
<tr>
<td>7-3</td>
<td>AsinOverPI</td>
<td>7-8</td>
</tr>
<tr>
<td>7-4</td>
<td>AtanOverPI</td>
<td>7-9</td>
</tr>
<tr>
<td>7-5</td>
<td>Atan2OverPI</td>
<td>7-10</td>
</tr>
<tr>
<td>7-6</td>
<td>CosPIx</td>
<td>7-11</td>
</tr>
<tr>
<td>7-7</td>
<td>SinPIx</td>
<td>7-12</td>
</tr>
<tr>
<td>7-8</td>
<td>SineWaveGenIDTL</td>
<td>7-15</td>
</tr>
<tr>
<td>7-9</td>
<td>SineWaveGenRDTL</td>
<td>7-18</td>
</tr>
<tr>
<td>7-10</td>
<td>SineWaveGenRDITL</td>
<td>7-21</td>
</tr>
<tr>
<td>7-11</td>
<td>SineWaveGenRDITLQ</td>
<td>7-24</td>
</tr>
<tr>
<td>7-12</td>
<td>SineWaveGenPAM</td>
<td>7-27</td>
</tr>
<tr>
<td>7-13</td>
<td>SineWaveGenDOM</td>
<td>7-30</td>
</tr>
<tr>
<td>Code Example</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>--------------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>11-4 cfftInit</td>
<td>11-18</td>
<td></td>
</tr>
<tr>
<td>11-3 cfft</td>
<td>11-12</td>
<td></td>
</tr>
<tr>
<td>11-2 autoCorr</td>
<td>11-11</td>
<td></td>
</tr>
<tr>
<td>11-1 C Header File dfr16.h</td>
<td>11-1</td>
<td></td>
</tr>
<tr>
<td>10-8 trans.</td>
<td>10-12</td>
<td></td>
</tr>
<tr>
<td>10-7 sub</td>
<td>10-11</td>
<td></td>
</tr>
<tr>
<td>10-6 mult</td>
<td>10-9</td>
<td></td>
</tr>
<tr>
<td>10-5 inv</td>
<td>10-7</td>
<td></td>
</tr>
<tr>
<td>10-4 equal</td>
<td>10-6</td>
<td></td>
</tr>
<tr>
<td>10-3 det</td>
<td>10-5</td>
<td></td>
</tr>
<tr>
<td>10-2 add</td>
<td>10-4</td>
<td></td>
</tr>
<tr>
<td>10-1 C Header File xfr16.h</td>
<td>10-2</td>
<td></td>
</tr>
<tr>
<td>9-8 sub</td>
<td>9-10</td>
<td></td>
</tr>
<tr>
<td>9-7 scale</td>
<td>9-9</td>
<td></td>
</tr>
<tr>
<td>9-6 mult</td>
<td>9-8</td>
<td></td>
</tr>
<tr>
<td>9-5 length</td>
<td>9-7</td>
<td></td>
</tr>
<tr>
<td>9-4 add</td>
<td>9-4</td>
<td></td>
</tr>
<tr>
<td>9-3 dotProd</td>
<td>9-5</td>
<td></td>
</tr>
<tr>
<td>9-2 add</td>
<td>9-2</td>
<td></td>
</tr>
<tr>
<td>9-1 C Header File vfr16.h</td>
<td>9-2</td>
<td></td>
</tr>
<tr>
<td>8-15 sqrt</td>
<td>8-20</td>
<td></td>
</tr>
<tr>
<td>8-14 round</td>
<td>8-19</td>
<td></td>
</tr>
<tr>
<td>8-13 min</td>
<td>8-13</td>
<td></td>
</tr>
<tr>
<td>8-12 max</td>
<td>8-12</td>
<td></td>
</tr>
<tr>
<td>8-11 negate</td>
<td>8-17</td>
<td></td>
</tr>
<tr>
<td>8-10 msu</td>
<td>8-15</td>
<td></td>
</tr>
<tr>
<td>8-9 equal</td>
<td>8-9</td>
<td></td>
</tr>
<tr>
<td>8-8 sub</td>
<td>8-21</td>
<td></td>
</tr>
<tr>
<td>8-7 mac</td>
<td>8-11</td>
<td></td>
</tr>
<tr>
<td>8-6 mult</td>
<td>8-16</td>
<td></td>
</tr>
<tr>
<td>8-5 div</td>
<td>8-8</td>
<td></td>
</tr>
<tr>
<td>8-4 add</td>
<td>8-6</td>
<td></td>
</tr>
<tr>
<td>8-3 abs</td>
<td>8-5</td>
<td></td>
</tr>
<tr>
<td>8-2 C Header File afr16.h</td>
<td>8-1</td>
<td></td>
</tr>
<tr>
<td>8-1 C Header File afr32.h</td>
<td>8-2</td>
<td></td>
</tr>
<tr>
<td>7-16 CosPIxLUT</td>
<td>7-38</td>
<td></td>
</tr>
<tr>
<td>7-15 SinPIxLUT</td>
<td>7-35</td>
<td></td>
</tr>
<tr>
<td>7-14 WaveGenRDITLQ</td>
<td>7-33</td>
<td></td>
</tr>
<tr>
<td>7-13 WaveGenRDITLQ</td>
<td>7-33</td>
<td></td>
</tr>
<tr>
<td>7-12 WaveGenRDITLQ</td>
<td>7-33</td>
<td></td>
</tr>
<tr>
<td>7-11 WaveGenRDITLQ</td>
<td>7-33</td>
<td></td>
</tr>
<tr>
<td>7-10 WaveGenRDITLQ</td>
<td>7-33</td>
<td></td>
</tr>
<tr>
<td>7-9 WaveGenRDITLQ</td>
<td>7-33</td>
<td></td>
</tr>
<tr>
<td>7-8 WaveGenRDITLQ</td>
<td>7-33</td>
<td></td>
</tr>
<tr>
<td>7-7 WaveGenRDITLQ</td>
<td>7-33</td>
<td></td>
</tr>
<tr>
<td>7-6 WaveGenRDITLQ</td>
<td>7-33</td>
<td></td>
</tr>
<tr>
<td>7-5 WaveGenRDITLQ</td>
<td>7-33</td>
<td></td>
</tr>
<tr>
<td>7-4 WaveGenRDITLQ</td>
<td>7-33</td>
<td></td>
</tr>
<tr>
<td>7-3 WaveGenRDITLQ</td>
<td>7-33</td>
<td></td>
</tr>
<tr>
<td>7-2 WaveGenRDITLQ</td>
<td>7-33</td>
<td></td>
</tr>
<tr>
<td>7-1 WaveGenRDITLQ</td>
<td>7-33</td>
<td></td>
</tr>
<tr>
<td>Code Example</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>---------------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>11-5 cifft</td>
<td>11-21</td>
<td></td>
</tr>
<tr>
<td>11-6 cifftInit</td>
<td>11-26</td>
<td></td>
</tr>
<tr>
<td>11-7 rfft</td>
<td>11-29</td>
<td></td>
</tr>
<tr>
<td>11-8 rifft</td>
<td>11-38</td>
<td></td>
</tr>
<tr>
<td>11-9 corr</td>
<td>11-42</td>
<td></td>
</tr>
<tr>
<td>11-10 fir</td>
<td>11-44</td>
<td></td>
</tr>
<tr>
<td>11-11 firs</td>
<td>11-46</td>
<td></td>
</tr>
<tr>
<td>11-12 firCreate</td>
<td>11-49</td>
<td></td>
</tr>
<tr>
<td>11-13 firDestroy</td>
<td>11-50</td>
<td></td>
</tr>
<tr>
<td>11-14 firHistory</td>
<td>11-52</td>
<td></td>
</tr>
<tr>
<td>11-15 firInit</td>
<td>11-54</td>
<td></td>
</tr>
<tr>
<td>11-16 iir</td>
<td>11-68</td>
<td></td>
</tr>
<tr>
<td>11-17 iirInit</td>
<td>11-72</td>
<td></td>
</tr>
</tbody>
</table>
About This Document

This manual describes a function library of digital signal processing algorithms using fractional math.

These digital signal processing algorithms cover the following categories:

- Basic fractional math primitives
- Trigonometric functions
- Sine wave generation
- Single-dimensioned array functions
- Vector processing functions
- Matrix functions
- Fast Fourier Transforms (FFT)
- Finite Impulse Response (FIR) filters
- Infinite Impulse Response (IIR) filters
- Correlation

Audience

This document targets software developers performing digital signal processing functions, especially those using the DSP56800 family of processors.

Organization

This manual is arranged in the following sections:

- **Chapter 1, Introduction**—provides a brief overview of this document
- **Chapter 2, Target Architectures**—Describes details of the Motorola DSP architectures relating to fractional data types and fractional algorithms
- **Chapter 3, CodeWarrior C**—discusses the use of Metrowerks’ CodeWarrior for Motorola Embedded Digital Signal Processors (DSPs)
- **Chapter 4, Software Development Kit**—describes the basic directory structure, configuration file, and make facility for the SDK
- **Chapter 5, Include and Library Files**—describes the files comprising the DSP Function Library
- **Chapter 6, Basic Fractional Math Library**—describes the basic math primitives for 16-bit and 32-bit fractional systems
- **Chapter 7, Trigonometric Math Library**—describes the trigonometric functions for 16-bit fractional values
- **Chapter 8, Array Library**—describes functions for single-dimensioned arrays of 16-bit and 32-bit fractional values
- **Chapter 9, Vector Library**—describes functions for vectors of 16-bit fractional values
- **Chapter 10, Matrix Library**—describes functions for matrices of 16-bit and 32-bit fractional values
Chapter 11, Signal Processing Library—describes signal processing functions for 16-bit fractional values

Chapter 12, Performance—details the performance of each of the DSP Function Library functions on the DSP56825

Chapter 13, License—provides the license required to use this product

Suggested Reading

We recommend that you have a copy of the following references:

- DSP56800 Family Manual, DSP56800FM/AD
- DSP56824 User’s Manual, DSP56824UM/AD
- Inside CodeWarrior: Core Tools, Metrowerks Corp.

Conventions

This document uses the following notational conventions:

<table>
<thead>
<tr>
<th>Typeface, Symbol or Term</th>
<th>Meaning</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Courier Monospaced Type</td>
<td>Commands, command parameters, code examples, expressions, datatypes, and directives</td>
<td>&quot;*Foundational include files...&quot; &quot;a data structure of type vad_tConfigure...&quot;</td>
</tr>
<tr>
<td>Italic</td>
<td>Calls, functions, statements, procedures, routines, arguments, file names and applications</td>
<td>&quot;the pConfig argument...&quot; &quot;defined in the C header file, aec.h...&quot; &quot;makes a call to the Callback procedure...&quot;</td>
</tr>
<tr>
<td>Bold</td>
<td>Reference sources, paths, emphasis</td>
<td>&quot;refer to the Targeting DSP56824 Platform manual...&quot; &quot;see: C:\Program Files\Motorola\Embedded SDK\help\tutorials&quot;</td>
</tr>
<tr>
<td>Bold/Italic</td>
<td>Directory name, project name</td>
<td>&quot;and contains these core directories: applications contains applications software...&quot; &quot;CodeWarrior project, 3des.mcp, is...&quot;</td>
</tr>
<tr>
<td>Blue Text</td>
<td>Linkable on-line</td>
<td>&quot;refer to Chapter 7, License...&quot;</td>
</tr>
<tr>
<td>Number</td>
<td>Any number is considered a positive value, unless preceded by a minus symbol to signify a negative value</td>
<td>3V -10 DES^-1</td>
</tr>
<tr>
<td>Hexadecimal Values</td>
<td>Begin with a &quot;$&quot; symbol</td>
<td>$0FF0 $80</td>
</tr>
<tr>
<td>ALL CAPITAL LETTERS</td>
<td>Variables, directives, defined constants, files libraries</td>
<td>INCLUDE_DSPFUNC #define INCLUDE_STACK_CHECK</td>
</tr>
<tr>
<td>Brackets [...]</td>
<td>Function keys</td>
<td>&quot;by pressing function key [F7]...&quot;</td>
</tr>
</tbody>
</table>
Definitions, Acronyms, and Abbreviations

The following list defines the acronyms and abbreviations used in this document. As this template develops, this list will be generated from the document. As we develop more group resources, these acronyms will be easily defined from a common acronym dictionary. Please note that while the acronyms are in solid caps, terms in the definition should be initial capped ONLY IF they are trademarked names or proper nouns.

<table>
<thead>
<tr>
<th>Typeface, Symbol or Term</th>
<th>Meaning</th>
<th>Examples</th>
</tr>
</thead>
</table>
| Quotation marks “...” | Returned messages | ...the message, “Test Passed” is displayed....
|                         |         | ...if unsuccessful for any reason, it will return “NULL”.... |

A Accumulator Register A
A2 4-bit Accumulator A Extension Register
ABS Absolute Value
afr Single-Dimensioned Array, Fractional
B Accumulator Register B
B2 4-bit Accumulator B Extension Register
dfr Digital Signal Processing, Fractional
DSP Digital Signal Processor or Digital Signal Processing
FFT Fast Fourier Transforms
FIR Finite Impulse Response
I/O Input/Output
IDE Integrated Development Environment
IIR Infinite Impulse Response
JTAG Joint Test Action Group
L Limit Bit
LSB Least Significant Bit
MAC Multiply/Accumulate
mfr Basic Math, Fractional
MIPS Million Instructions Per Second
MSB Most Significant Bit
OnCE™ On-Chip Emulation
OMR Operating Mode Register
PC Personal Computer
pX Pointer to variable X
pY Pointer to variable Y
pZ Pointer to variable Z
Rounding Bit

16-bit Address Registers
Saturation Bit
Software Development Kit
Stack Pointer
Serial Peripheral Interface
Status Register
Source
Synchronous Serial Interface
Trigonometric, fractional
Overflow bit
Vector, fractional
Input variable
Matrix, fractional
Input register X0
Input variable
Input register Y0
Input register Y1
Output variable

References

The following sources were used to produce this book:

1. *DSP56800 Family Manual*, DSP56800FM/AD
2. *DSP56824 User’s Manual*, DSP56824UM/AD
3. Embedded SDK Programmer’s Guide
Chapter 1
Introduction

Welcome to Motorola’s Family of Digital Signal Processors (DSPs). This document describes the Digital Signal Processing Function Library, which is a part of Motorola’s comprehensive Software Development Kit (SDK), for its DSPs. In this document, you will find all the information required to use and maintain the DSP Function Library’s algorithms.

Motorola provides these algorithms for use on the Motorola DSPs to expedite your application development and reduce the time it takes to bring your own products to market.

Motorola’s Digital Signal Processing Function Library is licensed for your use at no charge on Motorola processors. Please refer to the standard Software License Agreement in Chapter 7 for license terms and conditions; please consult with your Motorola representative for premium product licensing.

1.1 Digital Signal Processing Algorithms

The Digital Signal Processing Function Library provides mathematical algorithms for digital signal processing applications using fractional data types. The fractional data type is a fixed-point representation encompassing the range $-1 \leq \text{fractional range} < 1$. The fractional data type is one of the strengths of the Motorola architecture, in that it can be used efficiently for computationally-intensive algorithms such as digital filters, speech coders, vector and array processing, digital control, and other signal processing tasks.

The Function Library described in this document includes over 50 general-purpose algorithms. These algorithms share a standard interface definition across many different processors, yet have been highly optimized in their implementation on each target architecture. These algorithms cover the following functional categories:

- Basic fractional math primitives
- Trigonometric functions
- Sine wave generation
- Single-dimensional array functions
- Vector processing functions
- Matrix operation functions
- Fast Fourier Transforms (FFT)
- Finite Impulse Response (FIR) filters
- Infinite Impulse Response (IIR) filters
- Correlation
1.2 Advantages

There are many advantages in using the Digital Signal Processing Function Library, which include:

- Portable interfaces
- Implementation on multiple processors in Motorola DSP Family
- Highly-optimized routines
- Proven algorithm implementations
- Tested for correctness
- Published performance data
- Callable from both C and assembler
- Test cases provided as ‘how to use’ examples
- Immediately available to expedite application development

1.3 Quick Start

The Digital Signal Processing Function Library is distributed as part of Motorola’s Embedded SDK. Please refer to the Embedded SDK Programmer’s Guide for information on the SDK.

The Digital Signal Processing Function Library is intended for use with the Metrowerks’ CodeWarrior for Motorola Embedded DSP. You can learn about CodeWarrior for the Motorola Embedded DSPs by contacting Metrowerks at 1-800-377-5416, or via the web at URL:

http://www.metrowerks.com

The first target for the DSP Function Library is the Motorola DSP56824 part. A free trial copy of CodeWarrior for the DSP56824 is provided with Motorola’s DSP56824 Evaluation Module (EVM) Kit. For information about the DSP56824 EVM Kit, visit Motorola SPS’s web, URL:

http://www.motorola.com/sps/dsp/docs/evm-824.html

To begin using the DSP Function Library, create a CodeWarrior project to use the Embedded SDK. Using CodeWarrior stationery for the Embedded SDK makes it easy; see Section 1.3.1, Creating a New Project, for step-by-step instructions.

Once the CodeWarrior project is created, initialize the DSP Function Library; for details, refer to Section 1.3.2, Initializing the DSP Function Library. You need include only `#define INCLUDE_DSPFUNC` in the `appconfig.h` file.

To use DSP Function Library routines, include this line in your C application code:

```c
#include "dspfunc.h"
```
Finally, in your C or Assembly language program, use the interfaces specified in this document. As you build your application using the Project/Make menu command, CodeWarrior will extract the routines, and only those routines, that you actually use in your application, while linking to create your executable image. See Section 1.3.3, Using the DSP Function Library Routines for details.

To see explicit examples using the DSP Function Library interfaces, review the test files in this directory C:\Program Files\Motorola\Embedded SDK\src\dsp56824evm\nos\signal\dspfunc\test

Now let’s walk through the process of creating a CodeWarrior application which uses the DSP Function Library:

### 1.3.1 Creating a New Project

To create a new project with the SDK, follow these steps:

**Step 1:** Under the File menu, select *New*, as shown in Figure 1-1.

![Figure 1-1. Select File/New](image)

In your C or Assembly language program, use the interfaces specified in this document. As you build your application using the Project/Make menu command, CodeWarrior will extract the routines, and only those routines, that you actually use in your application, while linking to create your executable image. See Section 1.3.3, Using the DSP Function Library Routines for details.

To see explicit examples using the DSP Function Library interfaces, review the test files in this directory C:\Program Files\Motorola\Embedded SDK\src\dsp56824evm\nos\signal\dspfunc\test

Now let’s walk through the process of creating a CodeWarrior application which uses the DSP Function Library:

### 1.3.1 Creating a New Project

To create a new project with the SDK, follow these steps:

**Step 1:** Under the File menu, select *New*, as shown in Figure 1-1.

![Figure 1-1. Select File/New](image)
Introduction

Step 2: Under the Project tab, select Motorola Embedded SDK stationery and name the application in the Project name box on the right side of the window. See Figure 1-2.

Figure 1-2. Select Stationery

Step 3: Select target platform as shown in Figure 1-3.

Figure 1-3. Select Target Platform
Step 4: Select build type; see Figure 1-4.

Figure 1-4. Select Build Type

As shown in Figure 1-5, a new project, *application*, has been created:

Figure 1-5. A New Project
1.3.2 Initializing the DSP Function Library

Before using the DSP Function Library routines in an application, you’ll initialize the DSP Function Library by performing these two steps:

**Step 1:** Open the `appconfig.h` file by double-clicking it, as shown in **Figure 1-6**.

![Figure 1-6. Opening the appconfig.h file](image)
Step 2: Initialize the DSP Function Library by changing `#undef INCLUDE_DSPFUNC` to `#define INCLUDE_DSPFUNC`; see Figure 1-7.

```c
/* Refer to config/config.h for complete list of all components and
component default initialization */

/***************************************************************************/
* Include needed SDX components
***************************************************************************/

#define INCLUDE_BSP  /* BSP support */
#undef INCLUDE_CODEC  /* codec driver */
#undef INCLUDE_IO  /* I/O support */
#undef INCLUDE_LED  /* led support for target board */
#undef INCLUDE_SERIAL  /* serial support */
#undef INCLUDE_SPI  /* spi support */
#undef INCLUDE_TIMER  /* timer support */

#undef INCLUDE_MEMORY  /* memory support */
#define INCLUDE_DSPFUNC  /* dsp functional library */

/***************************************************************************/
* Overwrite default component initialization from config/config.h
***************************************************************************/
```

Figure 1-7. Initializing the DSP Function Library

Save this file as `appconfig.h`.

### 1.3.3 Using the DSP Function Library Routines

To use the DSP Function Library routines, you need only `#include dspfunc.h` in your application. With this file, you may then utilize any DSP Function Library routine in your application. The following steps demonstrate how to create and execute a new program which uses the DSP Function Library:
Step 1: Create a new text file as shown in Figure 1-8.

Figure 1-8. Create a New File
Step 2: Enter your application code into the new text file; see Figure 1-9.

```c
#include "dspfunc.h"

#define ARRAY_LEN 10

void main (void)
{
    Frac16 a[ARRAY_LEN];
    Frac16 b[ARRAY_LEN];
    Frac16 c[ARRAY_LEN];
    int i;

    /* Initialize two arrays */
    for (i=0; i<ARRAY_LEN; i++)
    {
        a[i] = 0x4000 /* 0.5 */;
        b[i] = 0x2000 /* 0.25 */;
    }

    /* Fractional add - single-dimensioned array */
    afx16Add(a, b, c, ARRAY_LEN);
}
```

Figure 1-9. Enter Application Code
Step 3: Save the text file under another name, shown as *main.c* in Figure 1-10:

![Image of Metrowerks CodeWarrior interface showing file save dialog]

**Figure 1-10.** Save the Text File
Step 4: As shown in Figure 1-11, add the file name main.c to the project by selecting Add Files from the Project menu.

![Add File](image1)

To add the file, either highlight the file name main.c and double click, or click the Add button in the lower right corner of the window; see Figure 1-12.

![Name File](image2)
Step 5: Compile the code and then execute the program under control of the debugger as shown in Figure 1-13.
Step 6: By default, upon clicking *Debug*, your application runs and breakpoints at *Main*.

![Debugging Interface](image)

Figure 1-14. Debug the Code

```c
#include "dspfunc.h"

#define ARRAY_LEN 10

void main (void)
{
    Frac16 a[ARRAY_LEN];
    Frac16 b[ARRAY_LEN];
    Frac16 c[ARRAY_LEN];
```
Chapter 2
Target Architectures

This section describes details of the Motorola DSP architecture relating to fractional data types and fractional algorithms. These details are presented so that the implementation of the Digital Signal Processing Function Library can be properly characterized on each target on which it is implemented.

In particular, this section describes those DSP architectures on which the DSP Function Library has been implemented. For each target architecture, the following topics are covered:

- Fractional type representation
- Fractional value ranges
- Data limiting moves
- Saturation
- Rounding
- Internal vs external memory
- Implementation of the DSP Function Library with regard to the above issues

2.1 DSP56800 Family

For more detailed information about the Motorola DSP56800 family, please refer to the following manuals:

DSP 56800 Family Manual
DSP 56824 User’s Manual

2.1.1 Fractional Data Representation

The DSP56800 architecture represents fractional values in two’s complement representation in the range of -1 \leq \text{fractional range} < 1. The main difference between fractional and integer representations is the location of the decimal point. While an integer representation locates the decimal point to the right of the value’s least significant bit, the fractional representation locates the decimal point immediately to the right of the most significant bit.
Signed fractional numbers lie in the range \(-1.0 \leq \text{fractional range} \leq +1.0 \, 2^{-N-1}\), where N is either 16 or 32. Thus, the most negative fractional number that can be represented is -1.0, whose internal representation is \$8000\ (16-bit) and \$80000000\ (32-bit). The most positive fractional number that can be represented is \(1.0 \cdot 2^{-15}\), or \$7FFF, in 16 bits and \(1.0 \cdot 2^{-31}\), or \$7FFFFFFF, in 32 bits.

The DSP56800 architecture represents and operates on both integer and fractional data. Data in a memory location or register can be interpreted either as fractional or integer because the decimal point is not explicitly represented. Instead, the decimal point is implied by the program or machine instructions being executed.

<table>
<thead>
<tr>
<th>Hex Representation</th>
<th>Integer Value (decimal)</th>
<th>Fractional Value (decimal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$4000</td>
<td>16384</td>
<td>0.5</td>
</tr>
<tr>
<td>$2000</td>
<td>8192</td>
<td>0.25</td>
</tr>
<tr>
<td>$1000</td>
<td>4096</td>
<td>0.125</td>
</tr>
<tr>
<td>$7000</td>
<td>28672</td>
<td>0.875</td>
</tr>
<tr>
<td>$0000</td>
<td>0</td>
<td>0.0</td>
</tr>
<tr>
<td>$C000</td>
<td>-16384</td>
<td>-0.5</td>
</tr>
<tr>
<td>$E000</td>
<td>-8192</td>
<td>-0.25</td>
</tr>
<tr>
<td>$F000</td>
<td>-4096</td>
<td>-0.125</td>
</tr>
<tr>
<td>$9000</td>
<td>-28672</td>
<td>-0.875</td>
</tr>
</tbody>
</table>
The following equation shows the relationship between a 16-bit integer and a fractional value:

\[
\text{Fractional Value} = \frac{\text{Integer Value}}{2^{15}}
\]

There is a similar equation relating 32-bit integers and fractional values:

\[
\text{Fractional Value} = \frac{\text{Integer Value}}{2^{31}}
\]

### 2.1.2 Fractional Arithmetic

The similarities between fractional and integer arithmetic permit the DSP56800 architecture to use the same machine instructions for add, subtract, compare, and logical operations. However, the machine instructions for multiplication differ between fractional and integer operations. The division operation requires multiple machine instructions on the DSP56800, and also differs between fractional and integer operations.

Please refer to the [DSP56800 Family Manual](#), Section 3.3.6, Multiplication, and Section 3.3.7, Division, for more details on the recommended sequences of machine instructions for multiplication and division of fractional and integer data.

### 2.1.3 Accumulators

The DSP56800 is a 16-bit, register-based architecture. Major registers include:

- Three 16-bit input registers (X0, Y0, Y1)
- Two 32-bit accumulator registers (A and B)
- Two 4-bit accumulator extension registers (A2 and B2)
- Four 16-bit address registers (R0 - R3)
- Stack Pointer Register (SP)
- Status Register (SR)
- Operating Mode Register (OMR)

The 32-bit accumulator registers can be accessed as either 32-bit registers (A or B) or two 16-bit sub-registers (A0 and A1, B0 and B1). The 32-bit accumulator registers can also be combined with the 4-bit accumulator extension registers to form 36-bit accumulator registers (A and B).
The 36-bit accumulator registers are useful for DSP algorithms because they can extend the range of intermediate calculations. Using the 36-bit accumulator registers, intermediate calculations can use a fractional range of \(-16 \leq \text{extended fractional range} < 16\). The 36-bit accumulators permit signal processing functions to exceed the normal fractional range in the middle of an algorithm as long as the final result stored to memory is in the range of \(-1 \leq \text{normal fractional range} < 1\).

### 2.1.4 Data Limiting

The DSP56800 architecture automatically limits extended fractional values to the normal fractional range as data is moved out of a 36-bit accumulator. When a MOVE instruction specifies an accumulator (A or B) as a source, and if the contents of that accumulator cannot be represented without overflow in the destination register or memory, the data limiter will substitute a “limited” data value. This limited data value will be maximum magnitude, with the same sign, as the source accumulator.

In other words, the data limiting function of the DSP56800 will store into the 16 bits of the destination register or memory either the maximum positive value, \(1 - 2^{-(N-1)}\), or $7FFF, when the 36-bit accumulator is greater or equal to 1, or the maximum negative value, \(-1\), or $8000, when the 36-bit accumulator is less than or equal to \(-1\).

It is also possible to bypass the data limiting feature of the DSP56800 by utilizing the A1 or B1 16-bit registers, rather than the 36-bit A or B accumulator registers, as the source register in a MOVE instruction. It is also possible to limit all intermediate results to 32 bits using the DSP56800’s saturation mode of operation, as described in Section 2.1.5.

### 2.1.5 Saturation

The DSP56800 provides a mode of operation, called saturation mode, for DSP algorithms that do not recognize or cannot take advantage of the extension accumulator. Saturation mode is enabled when the Saturation (SA) bit (bit 4 or $0010) is set in the Operating Mode Register (OMR).

Saturation mode is required for certain algorithms, such as bit-exact voice codecs, published as ITU specifications.

When the SA bit is set, all arithmetic results, including intermediate results in the accumulators, are limited to their maximum or minimum values supported by the 16-bit or 32-bit destination. The accumulator extension registers are not used.
Certain machine instructions on the DSP56800 operate independently from the saturation mode. Shifting operations, logical operations, and integer multiply instructions all operate independently from saturation mode, so that the expected and proper results are generated whether saturation mode is on or not.

Note that the saturation mode of operation may affect integer values as well as fractional values. When any arithmetic overflow or underflow occurs during an integer calculation, the integer result will be limited to $7FFF$ or $8000$ rather than wrapping around zero (0). Because C programmers have traditionally expected wrap-around to occur on overflow or underflow, saturation mode may cause C programs which inadvertently relied on the wrap-around feature to misbehave.

### 2.1.6 Limit Bit

The DSP56800 sets a Limit Bit (bit 6 or $0040$) in the Status Register, (SR), if any overflow occurs or if the data limiters perform a limiting operation. The Limit Bit, (L), is cleared only by an instruction that specifically clears it, which means that it is, in effect, a latching indicator bit. The Limit Bit can thus be used as an indicator for both data limiting and saturation functions during execution.

### 2.1.7 Rounding

The DSP56800 implements two types of rounding: convergent rounding and two’s complement rounding. Within the DSP56800, rounding occurs when a 32-bit result is converted to a 16-bit result, so that the value can be stored in memory or in a 16-bit register.

Convergent rounding rounds to the nearest even number. Two’s complement rounding always rounds up. A full presentation of the two types of rounding is presented in the [DSP56800 Family Manual](#), Section 3.5.1.

Upon reset, the DSP56800 selects convergent rounding by default. However, you may select the particular mode for rounding through the Rounding Bit (bit 5 or $0020$) of the OMR register.

### 2.1.8 Internal vs External Memory

The DSP56800 family of chips uses a Harvard memory architecture, which includes two independent memory spaces: data memory and program memory. Both data and program memory spaces can be mapped to internal (on-chip) ROM and RAM, and to external (off-chip) ROM and RAM. Typically, the amount of internal ROM and RAM is limited. External ROM and RAM is used to extend the address spaces to accommodate larger applications.
The DSP56800 family of chips supports dual-memory move instructions, in which two different data accesses can be made concurrently. These dual-memory access instructions, when properly utilized, greatly optimize the speed of various DSP algorithms. However, the DSP56800 architecture requires that one of the memory accesses in a dual-memory access instruction be to internal memory. Furthermore, certain external memory accesses must be made sequentially, whereas internal program and data memory can be accessed in parallel. For these efficiency reasons, a combination of internal and external ROM and RAM is typically configured.

2.1.9 DSP Function Library Implementation

As seen from previous sections, the DSP56800 architecture allows a programmer to select several different modes of operation with regard to data limiting, saturation, and rounding. This section relates these different modes of operation to the implementation of the DSP Function Library algorithms.

Data limiting is selected by using particular machine code sequences which use the A or B accumulator as a source register. The DSP Function Library has used the A and B accumulators with data limiting instructions for coding most algorithms. This means that, in general, intermediate results are placed in the A and B accumulators, which permits an extended fractional range. Final results are data limited when the accumulator is stored in memory.

Note that automatic data limiting, as described in Section 2.1.4, is not affected by the state of the Saturation (SA) bit. However, when saturation mode is enabled, the extended range of the A and B accumulators is not used, because intermediate results are limited-saturated at the time of each calculation. Thus, data limiting is never invoked, because no extended range values are ever produced as a result of intermediate calculations in an accumulator.

For many DSP applications, the extended precision in the accumulators is useful. Nevertheless, some DSP standards have been defined in a manner which requires disabling this extended precision capability. For example, the ETSI and ITU reference implementations of bit-exact standards for telecommunication applications limit the results of each arithmetic operation, rather than using extended intermediate ranges. Therefore, for these standards, saturation mode is required.

Control of the rounding modes and saturation modes has been given to the application programmer, who can control these modes by setting the appropriate bits in the OMR register. Alternatively, the following C routines are defined in the arch.h include file of the Motorola Embedded SDK for the DSP56800:

```c
/* Turn saturation mode off */
void archSetNoStat (void);

/* Turn saturation mode on */
void archSetStat32 (void);

/* Get, then set saturation mode */
bool archGetSetSaturationMode (bool bSatMode);

/* Turn on Two's Complement Rounding / turn off Convergent Rounding */
void archSet2CompRound (void);

/* Turn on Convergent Rounding / turn off Two's Complement Rounding */
void archSetConvRound (void);
```

The default values of these modes are detailed in Table 2-4.
Note that saturation mode is enabled by default in the Motorola Embedded SDK. Defaulting saturation mode on ensures that code developed for the DSP56800 is compatible with both the ETSI/ITU reference code and other Motorola DSP implementations of intrinsics for fractional operations.

Certain algorithms in the DSP56800 Signal Processing Function Library may require particular saturation and rounding modes. In these cases, the standard implementation has been to save the state of the saturation and rounding bits upon entry into a function, set the modes required by the algorithm, and reset the bits to the user’s configuration upon exit from the function. If particular modes are required, the special semantics and particular implementation of the algorithm will be noted in this document.

In general, the DSP Function Library algorithms do not explicitly manipulate the Limit Bit. However, the hardware will automatically set the Limit Bit to indicate that an overflow, data limiting, or saturation operation has occurred. Exceptions are noted in the semantics of each library function in this manual.

The application programmer can therefore manipulate the Limit Bit, (L), in the Status Register (SR), to determine if overflow, data limiting, or saturation has occurred within a code sequence. The following C routines are defined in the arch.h include file of the Motorola Embedded SDK for the DSP56800 for use by application programmers:

```c
/* Retrieve the Limit Bit */
Flag archGetLimitBit (void);

/* Reset the Limit Bit */
void archResetLimitBit (void);
```

These C routines can then be used to determine whether any calculations have exceeded the range of the fractional data type, as shown in the following code sequence. If saturation is enabled, having the Limit Bit set would indicate that a calculated value had been limited to its maximum or minimum fractional value.

```c
/* Reset the Limit Bit */
archResetLimitBit (void);

/* Perform some calculation */
dfr16IIR (pIIR, x, y, n);
if (archGetLimitBit() == 1)
{
    /* Saturation has occurred in the previous calculation */
    /* so take corrective action such as rescaling the input values */
    ...
    archResetLimitBit();
}
```

### Table 2-4. Default Saturation and Rounding Modes

<table>
<thead>
<tr>
<th></th>
<th>Power Up Reset Default</th>
<th>CodeWarrior Runtime Initialization</th>
<th>SDK appconfig.h #include DSP_FUNC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Saturation mode</td>
<td>Off</td>
<td>Off</td>
<td>On</td>
</tr>
<tr>
<td>Rounding</td>
<td>Convergent</td>
<td>Convergent</td>
<td>Two’s Complement</td>
</tr>
</tbody>
</table>
To obtain the best performance on the DSP56800, many DSP Function Library routines must take advantage of the dual-parallel move instructions. To do so in all cases, however, would force the programmer to allocate most data structures in internal data memory. For substantive applications, the constraint to place all data structures in internal data memory is burdensome. The amount of internal data memory is limited (3.5K words on the DSP56824, for example), so that not all data structures will fit in internal data memory.

Therefore, we have coded most of the DSP Function Library routines to be resilient in their use of dual-parallel move instructions. If data structures passed to the DSP Function Library routines are located in internal memory, each function will execute optimal code, when possible, using dual-parallel move instructions. However, if the data structures passed to the routine are located in external data memory, the dual-parallel move instructions will not be used. Therefore, whether or not the data structures are allocated in internal memory, the DSP Function Library routines will work correctly.

The description of each DSP Function Library routine in this manual details the optimal conditions for data allocation necessary to achieve the best algorithm performance. As a DSP application exhausts internal memory resources, the application programmer must make the appropriate performance tradeoffs by allocating selected data structures in external memory.
Chapter 3
CodeWarrior C

This chapter discusses how to use the Metrowerks’ CodeWarrior for Motorola Embedded DSPs with the DSP Function Library and emphasizes the use of fractional types. For a more in-depth discussion of CodeWarrior features, please refer to the Metrowerks’ documentation.

3.1 Frac16 vs __fixed__

The Metrowerks’ CodeWarrior C compiler provides pre-defined fractional types for both 16-bit and 32-bit values, detailed in Table 3-1.

| __fixed__  | 16 bits |
| __shortfixed__ | 16 bits |
| __longfixed__ | 32 bits |

Here is a sample program illustrating use of these fixed types:

```c
#include <stdio.h>

void setvals (__fixed__ *pX, __shortfixed__ *pY, __longfixed__ *pZ) {
    *pX = 0.5;
    *pY = 0.25;
    *pZ = 0.125;
}

void main() {
    __fixed__ x16;
    __shortfixed__ xs16;
    __longfixed__ xl32;
    setvals (&x16, &xs16, &xl32);
```
The fractional type \_\_fixed\_\_ is specific to the Metrowerks’ CodeWarrior implementation. Code written to take advantage of the \_\_fixed\_\_ types has proven not to be portable to other DSPs and other DSP tools.

Therefore, with Release 3.0 of the CodeWarrior for Motorola Embedded DSP, Metrowerks has implemented a set of fractional intrinsics that are compatible with both the ETSI/ITU bit-exact standards and the StarCore DSP C implementation. These intrinsics rely on underlying types of \_short\_ and \_long\_, rather than \_\_fixed\_. For example, the fractional add intrinsic is defined as:

```c
short add(short,short);
```

A complete list and definition of these fractional intrinsics is defined in Chapter 6, Basic Fractional Math Library.

For portability reasons, the DSP Function Library has defined two typedefs for the basic fractional types rather than using short and long directly. These two typedefs, which can be found in \_port.h\_, are:

```c
typedef short          Frac16;
typedef long           Frac32;
```

\_Frac16\_ and \_Frac32\_ have been used consistently throughout the DSP Function Library. We recommend that you use these types within your application as well to produce portable code using all the tools supplied with the DSP Function Library and the Software Development Kit. This approach to portability will help you take advantage of upcoming advances in Motorola’s rapidly-expanding DSP solutions.

### 3.2 Basic Types in \_port.h\_

The \_port.h\_ include file defines a set of basic types to support code portability between different DSPs and DSP tools. \_port.h\_ is shown in \_Code Example 3-1\_. Note in particular the fractional types \_Frac16\_ and \_Frac32\_, as well as the complex fractional structures, \_CFrac16\_ and \_CFrac32\_; these types are used throughout the interface definitions for the DSP Function Library.

#### Code Example 3-1. \_port.h\_

```c
/* File: port.h */

#ifndef __PORT_H
#define __PORT_H

/*******************************************************
* Target designation
 *******************************************************/

/* Target designation
 *******************************************************/

/** MetroWerks defines __m56800__ */

#endif

/***********************************************************/
```
* C Constructs
*******************************************************************************/
#define EXPORT extern
#define ITU_INTRINSICS

/******************************************************************************
* Basic Types
*******************************************************************************/

/* Generic word types for ITU compatibility */
typedef short          Word16;
typedef unsigned short UWord16;
typedef long           Word32;
typedef unsigned long  UWord32;
typedef int            Int16;
typedef unsigned int   UInt16;
typedef long           Int32;
typedef unsigned long  UInt32;

/* Fractional data types for portability */
typedef short          Frac16;
typedef long           Frac32;
tydef struct {
    Frac16     real;
    Frac16     imag;
} CFrac16;
typedef struct {
    Frac32     real;
    Frac32     imag;
} CFrac32;

/* Useful definitions */

/* Convert int/float to Frac16; constant x generates compile time constant */
#define FRAC16(x) (x < 1 ? (x >= -1 ? x*0x8000 : 0x8000) : 0x7FFF)

/* Miscellaneous types */
typedef int            Flag;
typedef int            Result;
#ifndef COMPILER_HAS_BOOL
typedef int            bool;
#endif

/******************************************************************************
* Constants
*******************************************************************************/

/* Function Result Values */
#define PASS           0
#define FAIL           -1
#ifndef COMPILER_HAS_BOOL
#define true           1
#define false          0
#endif

/******************************************************************************
* Implementation Limits
*******************************************************************************/
#define MAX_VECTOR_LEN  8192

#endif
3.3 Constants

With the CodeWarrior __fixed__ types, you can use real constants in assignment statements and arithmetic expressions. For example, CodeWarrior C accepts the following statements:

```
__fixed__ x16, y16;

x16 = 0.5;
y16 = x16 + 0.25;
```

However, CodeWarrior Release 3.0 has changed the emphasis on fractional types to use intrinsics defined with `short (Frac16)` and `long (Frac32)`. With these intrinsics, we recommend that programmers use hex representations of the fractional value for all constants. For example, the previous code sequence using these intrinsics now becomes:

```
Frac16 x16, y16;

x16 = 0x4000;
y16 = add ( x16, 0x2000);
```

This approach produces portable code compatible with the ETSI/ITU bit-exact reference standards used in the telecommunications industry and with the compilers supporting Motorola’s StarCore product line.

You may find it useful to define the following macro defined in `port.h` to convert between floating point representations and `Frac` constants:

```
#define FRAC16(x) (x < 1 ? (x >= -1 ? x*0x8000 : 0x8000) : 0x7FFF)
```

As long as `x` is a floating point constant, CodeWarrior will evaluate the expression at compile time to produce the fractional constant.

3.4 Data Storage

Most of the CodeWarrior data types have obvious storage requirements and layout. However, the storage layout for the long and unsigned long (32-bit) types is noteworthy. CodeWarrior stores these 32-bit types in a little endian-like format, as portrayed in Table 3-2:

```
long x32 = 0x12345678;
```

**Table 3-2. Data Storage for Long and Unsigned Long**

<table>
<thead>
<tr>
<th>Address n</th>
<th>Address n+1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x5678</td>
<td>0x1234</td>
</tr>
</tbody>
</table>

Other 32-bit types derived from long and unsigned long, including Frac32, Int32, UInt32, Word32, and UWord32, have the same little endian-like storage layout.
3.5 Intrinsics

Metrowerks’ CodeWarrior for Embedded DSP product provides these intrinsics for fractional operations:

- absolute value
- addition
- subtraction
- multiplication
- division
- multiply and accumulate
- multiple and subtract
- negation
- round
- shift left and right

Refer to Chapter 6, Basic Fractional Math Library, for complete details on these intrinsics and additional library functions.

3.6 Calling Conventions

This DSP Function Library has been implemented using both C and assembler. All interfaces have been implemented using the CodeWarrior C calling conventions so that the routines can be called from both C and assembler regardless of what language has been used for the implementation. The C calling conventions are described in the following paragraphs, organized by target architecture.

3.6.1 DSP56800 Calling Conventions

The registers $A$, $R2$, $R3$, $Y0$, and $Y1$ are used to pass parameters to functions. When a function is called, the parameter list is scanned from left to right. The parameters are passed as follows:

1. The first long fixed-point value is placed in $A$
2. The first two 16-bit fixed-point values are placed in $Y0$ and $Y1$
3. The first two 16-bit addresses are placed in $R2$ and $R3$
4. If there are no long fixed-point parameters, the first non-fixed-point 32-bit value or 19-bit address is placed in $A$
5. If there are no 16-bit fixed-point parameters, the first two 16-bit, non-fixed-point, non-address values are placed in $Y0$ and $Y1$, and $Y0$ receives the single value when only one is passed

All remaining parameters are pushed onto the stack, beginning with the right most parameter. Multiple-word parameters (19- and 32-bit values) have their least significant word pushed onto the stack first. When calling a routine that returns a structure, the caller passes an address in $R0$ which specifies where to copy the structure.
The registers $A$, $R0$, $R2$, and $Y0$ are used to return function results as follows:

1. Long fixed-point values are returned in $A$
2. All 19-bit addresses and 32-bit values are returned in $A$
3. 16-bit addresses are returned in $R2$
4. All 16-bit non-address values are returned in $Y0$
Chapter 4
Software Development Kit

The DSP Function Library is distributed as a part of Motorola’s Embedded Software Development Kit, SDK. This chapter describes the basic directory structure, configuration file, and make facility for the SDK, as it applies to the DSP Function Library. For more complete documentation on the SDK, refer to the SDK Programmer’s Guide.

4.1 SDK Directory Structure

The DSP Function Library is shown in Figure 4-1, and can be found in the SDK directory structure: src\...\os\signal\dsfunc
Within the `src\..\nos\signal\dspfunc` directory are sub-directories which contain test cases and CodeWarrior compilation files. These directories are described below:

`signal\dspfunc` - Interfaces for the DSP56800 Signal Processing Function Library described by this manual.

`signal\dspfunc\dspfunc_Data` - This directory is created by the Metrowerks’ CodeWarrior IDE when the project is built to hold object files and library files resulting from the make.

`signal\dspfunc\test` - Directory of test cases for the DSP Function Library.

`signal\dspfunc\test\cfft` - Directory of test cases for the complex FFT routines of the DSP Function Library.

`signal\dspfunc\test\rfft` - Directory of test cases for the real FFT routines of the DSP Function Library.

`signal\dspfunc\test\mathandfilters` - Directory of test cases for the mathematical and filter functions of the DSP Function Library.
4.2 *appconfig.h*

The include file, *appconfig.h*, found in your application’s *config* subdirectory, provides a common method to tailor the SDK for a specific application. By defining, or undefining, various C preprocessor variables, you can include or exclude different capabilities provided by the SDK. In addition, by including a library, *appconfig.h* invokes standard initialization code for the included library as well as consistency checks to make sure that all included capabilities are compatible.

Typically, *appconfig.h* is used to include and initialize the DSP Function Library. To include the DSP Function Library, modify *appconfig.h* so that it defines the preprocessor variable “INCLUDE_DSPFUNC” as in the following code sequence. Because some of the DSP Function Library routines use capabilities of the SDK memory management routines, it may be necessary to define the preprocessor variable “INCLUDE_MEMORY” as well.

```c
/* Refer to config/config.h for complete list of all components and component default initialization */

/*****************************************************************************/
* Include needed SDK components
/*****************************************************************************/

#define INCLUDE_BSP          /* BSP support */
#undef INCLUDE_MEMORY       /* Memory support */
#define INCLUDE_DSPFUNC      /* DSP Function Library */
```

Note that CodeWarrior linker pulls code from the DSP Function Library only for those functions used within an application. In this way, the aggregate amount of application code is minimized to that actually referenced within the application. When you define the preprocessor INCLUDE_DSPFUNC in the *appconfig.h* file, you are preparing the library for use, but including only those DSP Function Library routines used in the application.

As part of the initialization of the DSP Function Library, the SDK configuration mechanism calls the procedure *dspfuncInitialize* found in the C file *dspfunc.c*. The code for *dspfuncInitialize* sets the saturation mode of the processor, enables two’s complement rounding, and resets the limit bit of the processor. Before the application function main is called, *dspfuncInitialize* is called. If you desire a different processor configuration than that set by *dspfuncInitialize*, you may explicitly establish that configuration at the start of the main procedure.

```c
void dspfuncInitialize(void)
{
    /* Set saturation mode, set 2’s complement rounding, and reset limit bit */
    archSetSat32();
    archSet2CompRound();
    archResetLimitBit();
}
```
4.3 CodeWarrior Project

In order for CodeWarrior to pull functions from the DSP Function Library, CodeWarrior must be told where to find the DSP Function Library \textit{dspfunc.lib} file. This is typically done by including the DSP Function Library, \textit{dspfunc.lib}, file in your CodeWarrior project.

A sample CodeWarrior project, which includes the \textit{dspfunc.lib}, is shown in Figure 4-2.

![Figure 4-2. Contents of an Example CodeWarrior Project](image)

The \textit{dspfunc.lib} is included in the CodeWarrior project as part of the Embedded SDK stationery. Normally you need not add the \textit{dspfunc.lib} file to your application project explicitly, although this is certainly possible. To automatically access the DSP Function Library, you need only use the Embedded SDK stationery when you establish a new project.

Note that the DSP Function Library project, \textit{dspfunc.mcp}, is also normally included in your application project. CodeWarrior uses this sub-project reference to ensure that all files are consistently compiled and linked, as in a traditional UNIX \textit{make} capability.
Chapter 5
Include and Library Files

This chapter describes the include files comprising the DSP Function Library.

5.1 Include Files

The DSP Function Library interfaces have been partitioned into C include files declaring like functions. These include files are:

- **port.h**: Basic types defined for portability
- **arch.h**: Architectural specific routines
- **prototype.h**: Intrinsics declared for compatibility with StarCore DSP
- **mfr16.h**: Basic 16-bit fractional math interfaces
- **mfr32.h**: Basic 32-bit fractional math interfaces
- **afr16.h**: Operations on single-dimensioned arrays of 16-bit fractional data
- **afr32.h**: Operations on single-dimensioned arrays of 32-bit fractional data
- **tfr16.h**: Trigonometric functions for 16-bit fractional data
- **vfr16.h**: Vector functions for 16-bit fractional data
- **vfr32.h**: Vector functions for 32-bit fractional data
- **xfr16.h**: Matrix functions for 16-bit fractional data
- **xfr32.h**: Matrix functions for 32-bit fractional data
- **dfr16.h**: Digital signal processing functions for 16-bit fractional data

To locate the source code for any particular DSP Function Library routine, the naming convention introduced above in the *.h file names is used for most individual function names. For example, the function **afr16Add** refers to the add operation on single-dimensioned arrays of 16-bit fractional data. The declaration for the **afr16Add** function is found in the file **afr16.h**. The source code for the routine can be found in the files **afr16.c** and **afr16.asm**.

Rather than including the individual *.h files listed above, we recommend that applications include only the **dspfunc.h** file, which is described in **Section 5.2**.
5.2 **dspfunc.h**

The file *dspfunc.h*, shown in **Code Example 5-1**, serves as a repository for all of the include files comprising the DSP Function Library interfaces. Include this file in your C program to gain access to all functions documented in this manual.

**Code Example 5-1. dspfunc.h**

```c
/* File: dspfunc.h */

#ifndef __DSPFUNC_H
#define __DSPFUNC_H

#include "appconfig.h"
#ifndef INCLUDE_DSPFUNC
#error INCLUDE_DSPFUNC must be defined in appconfig.h to initialize the
DSP Function Library
#endif

/* This include file is the master include file for the
DSP568xx Digital Signal Processing Function Library -
Fractional Algorithms for C and Assembly Programmers.*/

/***************************
Foundational Include Files
***************************/
#include "port.h"
#include "arch.h"
#include "prototype.h"

/***************************
Basic Fractional Math
***************************/
#include "mfr16.h"
#include "mfr32.h"

/***************************
Trigonometric Functions
***************************/
#include "tfr16.h"

/***************************
Single Dimension Array Functions
***************************/
#include "afr16.h"
#include "afr32.h"

/***************************
Vector Functions
***************************/
#include "vfr16.h"
```
Matrix Functions

#include "xfr16.h"

Signal Processing Functions

#include "dfrl6.h"

void dspfuncInitialize(void);
#endif

5.3 dspfunc.lib

The DSP Function Library routines are compiled and linked into a Software Development Kit library called *dspfunc.lib*, which can be found in this directory:
sdk\src\dsp56824evm\nos\signal\dspfunc\dspfunc_Data

If you add the *dspfunc.lib* file to your CodeWarrior project, CodeWarrior will extract the functions that you have used in your application from the DSP Function Library. To set up this extraction, highlight your CodeWarrior project window and then use the Project/Add Files menu command to add this file:
sdk\src\dsp56824evm\nos\signal\dspfunc\dspfunc_Data\dspfunc.lib

CodeWarrior includes in your executable image only those routines which you have actually used from the DSP Function Library, making your application’s footprint as small as possible in both RAM and ROM.

5.4 Rebuilding dspfunc.lib

Due to the powerful *make* facility inherent in CodeWarrior projects, the DSP Function Library is normally rebuilt automatically, as necessary, whenever it is referenced. To rebuild the library manually, however, open the DSP Function Library project file explicitly from CodeWarrior:
sdk\src\dsp56824evm\nos\signal\dspfunc\dspfunc.mcp

Then, make the library by clicking *Make* from the Project menu.

5.5 Sample Code

The DSP Function Library is delivered with test cases covering all functions and interfaces within the library. These test cases can be found in the SDK directory:
sdk\src\dsp56824evm\nos\signal\dspfunc\test

Please review these test cases for detailed coding examples which show how to use DSP Function Library interfaces.
Chapter 6
Basic Fractional Math Library

This section describes the basic math primitives for 16-bit and 32-bit fractional values.

6.1 Introduction

Basic math primitives, such as add, subtract, and multiply, for fractional types are implemented by the CodeWarrior C compiler as intrinsics. Intrinsic means that efficient code is generated by the C compiler because knowledge of the basic math operation for fractional types has been built into the compiler.

CodeWarrior provides basic math intrinsics for both 16-bit (short) and 32-bit (long) fractional types. For portability, we have defined a short 16-bit fractional type as \texttt{Frac16} and a long 32-bit fractional type as \texttt{Frac32}.

In addition, the Digital Signal Processing Function Library augments the C compiler intrinsics with such basic math functions as random number generation and square root.

The next two sections present the C interfaces for 16-bit fractional math primitives, \texttt{mfr16.h}, and 32-bit fractional math primitives, \texttt{mfr32.h}.

6.2 16-bit Fractional Math Interface - \texttt{mfr16.h}

The C interface for the 16-bit fractional math primitives is defined in the C header file \texttt{mfr16.h}, shown in Code Example 6-1.

\begin{verbatim}
Code Example 6-1. \texttt{mfr16.h}

/* File: mfr16.h */

#ifndef __MFR16_H
#define __MFR16_H

#include "port.h"
#include "prototype.h"
#include "mfr32.h"

#ifdef __cplusplus
extern "C" {
#endif

#endif

#endif

#endif

\end{verbatim}
6.3 32-bit Fractional Math Interface - *mfr32.h*

The C interface for the 32-bit fractional math primitives is defined in the C header file *mfr32.h*, which is shown in Code Example 6-2.
/* File: mfr32.h */

#ifndef __MFR32_H
#define __MFR32_H

#include "port.h"
#include "prototype.h"

#ifdef __cplusplus
extern "C" {
#endif

/*******************************************************
* Fractional Math Intrinsics
* These math intrinsics are provided by the MetroWerks
* CodeWarrior C compiler for fractional types. These
* declarations are included here as documentation only.
*************************************************************/

* Frac32 L_abs          (Frac32 x);
* Frac32 L_add          (Frac32 x, Frac32 y);
* Frac32 div_ls         (Frac32 x, Frac16 y);
* Frac32 L_deposit_h    (Frac16 x);
* Frac32 L_deposit_l    (Frac16 x);
* Frac16 extract_l      (Frac32 x);
* Frac16 extract_h      (Frac32 x);
* Frac16 mac_r          (Frac32 w, Frac16 x, Frac16 y);
* Frac32 L_mac          (Frac32 w, Frac16 x, Frac16 y);
* Frac16 msu_r          (Frac32 w, Frac16 x, Frac16 y);
* Frac32 L_msu          (Frac32 w, Frac16 x, Frac16 y);
* Frac32 L_mult         (Frac16 x, Frac16 y);
* Frac32 L_mult_ls      (Frac32 x, Frac16 y);
* Frac32 L_negate       (Frac32 x);
* Frac16 norm_l         (Frac32 x);
* Frac16 round          (Frac32 x);
* Frac32 L_shl          (Frac32 x, Int16 n);
* Frac32 L_shr          (Frac32 x, Int16 n);
* Frac32 L_shr_r        (Frac32 x, Int16 n);
* Frac32 L_sub          (Frac32 x, Frac32 y);
*/

/*******************************************************
* Misc Fractional Math
*************************************************************/

EXPORT Frac16  mfr32Sqrt (Frac32 x);

#ifdef __cplusplus
}
#endif
#endif

Freescale Semiconductor, Inc.
ARCHIVED BY FREESCALE SEMICONDUCTOR, INC. 2005
Freescale Semiconductor, Inc.
ARCHIVED BY FREESCALE SEMICONDUCTOR, INC. 2005
For More Information On This Product,
Go to: www.freescale.com
ARCHIVED BY FREESCALE SEMICONDUCTOR, INC. 2005
Freescale Semiconductor, Inc.
ARCHIVED BY FREESCALE SEMICONDUCTOR, INC. 2005
6.4 Basic Fractional Math Function Specifications

The following pages describe the basic math functions for both 16-bit and 32-bit fractional types.

Each page utilizes a short name to generically describe its function. In many cases, however, the actual function name that must be used in C or assembler may be different from the generic function name. Therefore, refer to the C function declaration to find the specific function name to put in your program. For example, the generic function name may be \textit{abs}, but the actual function name that one would use for a 32-bit fractional value is \textit{L_abs}.

Function arguments for each routine are described as \textit{in}, \textit{out}, or \textit{inout}. An \textit{in} argument means that the parameter value is an input only to the function. An \textit{out} argument means that the parameter value is an output only from the function. An \textit{inout} argument means that a parameter value is an input to the function, but the same parameter is also an output from the function.

Typically, \textit{inout} parameters are input pointer variables in which the caller passes the address of a pre-allocated data structure to a function. The function stores its results within that data structure. The actual value of the \textit{inout} pointer parameter is not changed.
6.4.1 abs

Call(s):

Frac16 abs_s (Frac16 x);
Frac32 L_abs (Frac32 x);

Arguments:

Table 6-1. abs arguments

<table>
<thead>
<tr>
<th>x</th>
<th>in</th>
<th>The input data value</th>
</tr>
</thead>
</table>

Description: The abs function calculates the absolute value of the input value.

Equation:

\[ |x| = y \] \hspace{1cm} Eqn. 6-1

Returns: The abs function returns the result of the absolute value calculation.

Range Issues: If saturation is enabled, abs will return the maximum fractional value if overflow occurs. It is not possible to have an underflow with the abs function.

Special Issues: None

Design/Implementation: The abs function is implemented as a CodeWarrior C compiler-intrinsic.

Code Example 6-3. abs

```
{  Frac16 x16, z16;
   Frac32 x32, z32;

   ...  
   z16 = abs_s (x16);
   z32 = L_abs (x32);
   ...
}
```
6.4.2 add

Call(s):

Frac16 add (Frac16 x, Frac16 y);
Frac32 L_add (Frac32 x, Frac32 y);

Arguments:

<table>
<thead>
<tr>
<th>x</th>
<th>in</th>
<th>The first input value</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>in</td>
<td>The second input value</td>
</tr>
</tbody>
</table>

Description: The add function adds two fractional numbers \((x + y)\).

Equation:

\[
(x + y) = z \quad \text{Eqn. 6-2}
\]

Returns: The add function returns the results of the addition.

Range Issues: If saturation is enabled, the add function will return the maximum or minimum fractional values if overflow or underflow occurs, respectively.

Special Issues: None

Design/Implementation: The add function is implemented as a CodeWarrior C compiler-intrinsic.

Code Example 6-4. add

```c
{
  Frac16 x16, y16, z16;
  Frac32 x32, y32, z32;
  ...
  z16 = add (x16, y16);
  z32 = L_add (x32, y32);
  ...
}
```
6.4.3 div

Call(s):

Frac16 div_s (Frac16 x, Frac16 y);
Frac16 div_ls (Frac32 x, Frac16 y);

Arguments:

| x  | in | The dividend of the division |
| y  | in | The divisor for the division |

Description: The div function divides two fractional numbers (x / y); x and y must be positive and y must be greater than or equal to x.

Equation:

\[ \frac{y}{x} = z \]  \quad Eqn. 6-3

Returns: The div function returns the quotient of the division calculation.

Range Issues: The division calculation relies on the assumption that the quotient is within the valid fractional result range, -1 <= result < 1. However, if this assumption is violated, the result of the calculation is undefined and the resulting quotient may be erroneous. If saturation is enabled, and if x is equal to y, the maximum fractional value is returned.

Special Issues: None

Design/Implementation: The div function is implemented as a CodeWarrior C compiler-intrinsic.

Code Example 6-5. div

```c
{  Frac16 x16, y16, z16;
  Frac32 x32;
  ...
  z16 = div_s (x16, y16);
  z16 = div_ls(x32, y16);
  ...
}
```
6.4.4 deposit

Call(s):

\[
\text{Frac32 } L\text{-deposit}_h \quad (\text{Frac16 } x); \\
\text{Frac32 } L\text{-deposit}_l \quad (\text{Frac16 } x);
\]

Arguments:

<table>
<thead>
<tr>
<th>x</th>
<th>in</th>
<th>The input data value</th>
</tr>
</thead>
</table>

Description: The \( L\text{-deposit}_h \) function places a 16-bit fractional value into the most significant 16-bits of the 32-bit fractional output value and zeroes the least significant 16 bits. The \( L\text{-deposit}_l \) function places a 16-bit fractional value into the least significant 16 bits of the 32-bit fractional output value and sign extends the most significant 16 bits.

Equation: None

Returns: The function returns a 32-bit fractional value.

Range Issues: No overflow or underflow is possible with this function.

Special Issues: None

Design/Implementation: The deposit function is implemented as an inlined assembly function.

Code Example 6-6. deposit

```c
{
    Frac16 x16;
    Frac32 z32;

    ... 
    z16 = L\text{-deposit}_l \ (x16);
    z32 = L\text{-deposit}_h \ (x16);
    ...
}
```
6.4.5 extract

Call(s):

\[
\begin{align*}
\text{Frac16 extract\_l} & \quad (\text{Frac32} \ x); \\
\text{Frac16 extract\_h} & \quad (\text{Frac32} \ x);
\end{align*}
\]

Arguments:

<table>
<thead>
<tr>
<th>(x)</th>
<th>in</th>
<th>The input data value</th>
</tr>
</thead>
</table>

Description: The \textit{extract\_l} function extracts the least significant 16 bits from the 32-bit fractional value. The \textit{extract\_h} function extracts the most significant 16 bits from the 32-bit fractional value.

Equation: None

Returns: Both the \textit{extract\_l} and \textit{extract\_h} functions return the extracted values as a 16-bit fractional number.

Range Issues: Neither overflow nor underflow is possible with \textit{extract}.

Special Issues: None

Design/Implementation: The \textit{extract} function is implemented as a CodeWarrior C compiler-intrinsic.

Code Example 6-7. \textit{extract}

\[
\begin{align*}
\{ \\
\text{Frac32} \ x32; \\
\text{Frac16} \ z16; \\
\ldots \\
z16 = \text{extract\_l} \ (x32); \\
z16 = \text{extract\_h} \ (x32); \\
\ldots
\}
\end{align*}
\]
6.4.6 \textit{mac}

Call(s):
\begin{verbatim}
Frac16 mac_r        (Frac32 w, Frac16 x, Frac16 y);
Frac32 L_mac        (Frac32 w, Frac16 x, Frac16 y);
\end{verbatim}

Arguments:

\begin{table}[h]
\centering
\caption{\textit{mac} arguments}
\begin{tabular}{|c|c|}
\hline
\text{Argument} & \text{Description} \\
\hline
\text{x} & \text{in} \quad \text{The first input data value for the multiplication} \\
\text{y} & \text{in} \quad \text{The second input data value for the multiplication} \\
\text{w} & \text{in} \quad \text{The input value used with the addition} \\
\hline
\end{tabular}
\end{table}

Description: The \textit{mac} function multiplies two 16-bit fractional input values \((x \times y)\) and adds the 32-bit result to \(w\).

Equation:
\begin{equation}
(x \times y) + w = z \quad \text{Eqn. 6-4}
\end{equation}

Returns: The function returns the result of the calculation \((x \times y) + w\).

Range Issues: If saturation is enabled, the \textit{mac} function will return the maximum or minimum fractional values if overflow or underflow occurs, respectively.

Special Issues: None

Design/Implementation: The \textit{mac} function is implemented as a CodeWarrior C compiler-intrinsic.

Code Example 6-8. \textit{mac}
\begin{verbatim}
{
  Frac16 x16, y16, z16;
  Frac32 x32, z32;

  ...
  z16 = mac_r (x32, x16, y16);
  z32 = L_mac (x32, x16, y16);
  ...
}
\end{verbatim}
6.4.7 *msu*

Call(s):

- Frac16 msu_r = (Frac32 w, Frac16 x, Frac16 y);
- Frac32 L_msu = (Frac32 w, Frac16 x, Frac16 y);

Arguments:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>in</td>
</tr>
<tr>
<td>y</td>
<td>in</td>
</tr>
<tr>
<td>w</td>
<td>in</td>
</tr>
</tbody>
</table>

Table 6-7. *msu* arguments

- **x**: The first input data value for the multiplication
- **y**: The second input data value for the multiplication
- **w**: The input value used with the subtraction

Description: The *msu* function multiplies *x* by *y* and subtracts the 32-bit result from *w*.

Equation:

\[ w - (x + y) = z \]  \hspace{1cm} Eqn. 6-5

Returns: The function returns the result of the calculation \( w - (x + y) \).

Range Issues: If saturation is enabled, the *msu* function will return the maximum or minimum fractional values if overflow or underflow occurs, respectively.

Special Issues: None

Design/Implementation: The *msu* function is implemented as a CodeWarrior C compiler-intrinsic.

Code Example 6-9. *msu*

```c
{
    Frac16 x16, y16, z16;
    Frac32 x32, z32;
    ...
    z16 = msu_r (x32, x16, y16);
    z32 = L_msu (x32, x16, y16);
    ...
}
```
6.4.8 mult

Call(s):

- Frac16 mult (Frac16 x, Frac16 y);
- Frac16 mult_r (Frac16 x, Frac16 y);
- Frac32 L_mult (Frac16 x, Frac16 y);
- Frac32 L_mult_ls (Frac32 x, Frac16 y);

Arguments:

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>in</th>
<th>Description</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>y</td>
<td>in</td>
<td>The first input data value</td>
<td></td>
</tr>
</tbody>
</table>

Description: These multiplication functions multiply two fractional input values and return the result. The mult_r function differs from mult in that mult_r rounds the result, whereas mult truncates the result to 16bits.

Equation:

\[ xy = z \quad \text{Eqn. 6-6} \]

Returns: The mult function returns returns the result of the fractional multiplication.

Range Issues: If saturation is enabled, the mult function will return the maximum or minimum fractional values if overflow or underflow occurs, respectively.

Special Issues: None

Design/Implementation: The mult function is implemented as a CodeWarrior C compiler-intrinsic.

Code Example 6-10. mult

```c
{
    Frac16 x16, y16, z16;
    Frac32 x32, z32;

    ... z16 = mult (x16, y16);
    z16 = mult_r (x16, y16);
    z32 = L_mult (x16, y16);
    z32 = L_mult_ls (x32, y16);
    ...
}
```
6.4.9 negate

Call(s):

Frac16 negate (Frac16 x);
Frac32 L_negate (Frac32 x);

Arguments:

Table 6-9. negate arguments

<table>
<thead>
<tr>
<th>x</th>
<th>in</th>
<th>The input data value</th>
</tr>
</thead>
</table>

Description: The negate function negates the fractional input value.

Equation:

\[ x = -x \]  \hspace{1cm} Eqn. 6-7

Returns: The negate function returns the result of the negation.

Range Issues: If saturation is enabled, the negate function will return the maximum value if overflow occurs. It is not possible to have an underflow with the negate function.

Special Issues: None

Design/Implementation: The negate function is implemented as a CodeWarrior C compiler-intrinsic.

Code Example 6-11. negate

```c
{
    Frac16 x16, z16;
    Frac32 x32, z32;

    ...  
    z16 = negate (x16);
    z32 = L_negate (x32);
    ... 
}
```
6.4.10 **norm**

**Call(s):**

```
Int16 norm_s (Frac16 x);
Int16 norm_l (Frac32 x);
```

**Arguments:**

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>
| x | in | The input data value

**Description:** The *norm* function calculates the number of left shifts needed to normalize a fractional value.

**Equation:** None

**Returns:** The *norm* function returns the number of left shifts needed to normalize the fractional input value.

**Range Issues:** It is not possible to have an overflow or underflow with the *norm* function.

**Special Issues:** To normalize a value, rather than just calculating the number of left shift to normalize the value, the following operation must be done:

```
z = shl (x, norm_s(x))
```

**Design/Implementation:** The *norm* function is implemented as a CodeWarrior C compiler-intrinsic.

**Code Example 6-12. norm**

```
{
  Int16 i;
  Frac16 x16;
  Frac32 x32;
  ...
  i = norm_s (x16);
  i = norm_l (x32);
  ...
}
```
6.4.11  rand

Call(s):

Frac16 mfr16Rand (void);

Arguments:

<table>
<thead>
<tr>
<th>Table 6-11.  rand arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>rand takes no arguments</td>
</tr>
</tbody>
</table>

Description: The rand function calculates a pseudo-random number.

Equation: None

Returns: The rand function returns the pseudo-random number calculated.

Range Issues: There are no issues involving saturation with the rand function.

Special Issues: The randseed function may be used to set the seed used by the random number calculation; see Section 6.4.12, randseed.

Design/Implementation: The rand function is implemented as a library function call.

Code Example 6-13.  rand

```c
{
    Frac16 x16;
    x16 = mfx16Rand ();
}
```
6.4.12 randseed

Call(s):
void mfr16SetRandSeed (Frac16 x);

Arguments:

<table>
<thead>
<tr>
<th>x</th>
<th>in</th>
<th>The input data value to be used as the random number seed</th>
</tr>
</thead>
</table>

Description: The randseed function sets the random number seed used by the pseudo-random number generation algorithm.

Equation: None

Returns: The randseed function does not return any result.

Range Issues: There are no issues involving saturation with the randseed function.

Special Issues: None

Design/Implementation: The randseed function is implemented as a library function call.

Code Example 6-14. randseed

```c
{
    Frac16 x16;
    mfr16SetRandSeed (x16);
}
```
6.4.13  *round*

Call(s):

\[
\text{Frac16} \text{ round} \quad (\text{Frac32} \ x);
\]

Arguments:

Table 6-13.  *round* arguments

<table>
<thead>
<tr>
<th>x</th>
<th>in</th>
<th>The input data value</th>
</tr>
</thead>
</table>

Description: The *round* function rounds the 32-bit fractional input value to 16 bits.

Equation: None

Returns: The *round* returns the 16-bit representation of the input value, rounded to the least significant bit.

Range Issues: If saturation is enabled, the *round* function will return the maximum or minimum fractional values if overflow or underflow occurs, respectively.

Special Issues: None

Design/Implementation: The *round* function is implemented as a CodeWarrior C compiler-intrinsic.

Code Example 6-15.  *round*

```
{  Frac16 z16;
    Frac32 x32;
    ...
    z16 = round (x32);
    ...
}
```
6.4.14 \textit{shl}

\textbf{Call(s):}

\begin{verbatim}
Frac16 shl (Frac16 x, Int16 n);
Frac32 L_shl (Frac32 x, Int16 n);
\end{verbatim}

\textbf{Arguments:}

\begin{table}[h]
\centering
\begin{tabular}{|c|c|}
\hline
\textbf{x} & The input data value \\
\hline
\textbf{n} & The number of bits to left shift \(x\); \(n\) is signed and, if negative, implies a right shift \\
\hline
\end{tabular}
\end{table}

\textbf{Description:} The \textit{shl} function arithmetically shifts the input variable \(x\) left \(n\) positions, zero filling the least significant bits of the result. If \(n\) is negative, the \textit{shl} function arithmetically shifts \(x\) right by \(-n\) bits with sign extension.

\textbf{Equation:}

\[ x \leftarrow n = y \quad Eqn. \; 6-8 \]

\textbf{Returns:} The \textit{shl} function returns the value \(x\) arithmetically shifted by \(n\) bits.

\textbf{Range Issues:} If saturation is enabled, the \textit{shl} function will return the maximum or minimum fractional values if overflow or underflow occurs, respectively.

\textbf{Special Issues:} None

\textbf{Design/Implementation:} The \textit{shl} function is implemented as a CodeWarrior C compiler-intrinsic.

\textbf{Code Example 6-16. \textit{shl}}

\begin{verbatim}
{
  Frac16 x16, z16;
  Frac32 x32, z32;
  Int16  n;

  ...
  z16 = shl (x16, n);
  z32 = L_shl (x32, n);
}
\end{verbatim}
6.4.15 shr

Call(s):

\[ \text{Frac16 shr} \quad (\text{Frac16 } x, \text{ Int16 } n); \]
\[ \text{Frac16 shr_r} \quad (\text{Frac16 } x, \text{ Int16 } n); \]
\[ \text{Frac32 L_shr} \quad (\text{Frac32 } x, \text{ Int16 } n); \]
\[ \text{Frac32 L_shr_r} \quad (\text{Frac32 } x, \text{ Int16 } n); \]

Arguments:

<table>
<thead>
<tr>
<th>( x )</th>
<th>in</th>
<th>The input data value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n )</td>
<td>in</td>
<td>The number of bits to right shift ( x ); ( n ) is signed and, if negative, implies a left shift</td>
</tr>
</tbody>
</table>

Description: The \textit{shr} function arithmetically shifts the input variable \( x \) right \( n \) positions, sign extending the result. If \( n \) is negative, the \textit{shr} function arithmetically shifts \( x \) left by \(-n\) bits, zero filling the least significant bits. \textit{L_shr} \_\textit{r} differs from \textit{L_shr} in that \textit{L_shr} \_\textit{r} rounds the 32-bit fractional result.

Equation:

\[ x » n = y \quad \text{Eqn. 6-9} \]

Returns: The \textit{shr} function returns the value \( x \) arithmetically shifted by \( n \) bits.

Range Issues: If saturation is enabled, the \textit{shr} function will return the maximum or minimum fractional values if overflow or underflow occurs, respectively.

Special Issues: None

Design/Implementation: The \textit{shr} function is implemented as a CodeWarrior C compiler-intrinsic.

Code Example 6-17. \textit{shr}

```c
{  
    Frac16 x16, z16;
    Frac32 x32, z32;
    Int16 n;

    ...
    z16 = shr (x16, n);
    z16 = shr_r (x16, n);
    z32 = L_shr (x32, n);
    z32 = L_shr_r (x32, n);
    ...
}
```
6.4.16 \textit{sqrt}

**Call(s):**

\begin{verbatim}
Frac16 mfr16Sqrt (Frac16 x);
Frac16 mfr32Sqrt (Frac32 x);
\end{verbatim}

**Arguments:**

<table>
<thead>
<tr>
<th>( x )</th>
<th>\text{in}</th>
<th>\text{The input data value}</th>
</tr>
</thead>
</table>

**Description:** The \textit{sqrt} function calculates the square root of the fractional input data value.

**Equation:**

\[ \sqrt{x} = y \quad \text{Eqn. 6-10} \]

**Returns:** The \textit{sqrt} function returns the result of the square root calculation.

**Range Issues:** If input value \( x \) is negative, results of \textit{sqrt} are undefined.

To guarantee accuracy in its results, the function \textit{sqrt} saves the state of the saturation bit upon entry to the function, enables the saturation bit locally, and then restores the bit to its original setting upon return from the function.

**Special Issues:** None

**Design/Implementation:** The \textit{sqrt} function is implemented as a library function call.

**Code Example 6-18. \textit{sqrt}**

\begin{verbatim}
{    Frac16 x16, z16;
    Frac32 x32;
    ...
    z16 = mfx16Sqrt (x16);
    z16 = mfx32Sqrt (x32);
    ...
}
\end{verbatim}
6.4.17  sub

Call(s):

\[
\begin{align*}
\text{Frac16 sub} & \quad (\text{Frac16 } x, \text{ Frac16 } y); \\
\text{Frac32 L_sub} & \quad (\text{Frac32 } x, \text{ Frac32 } y);
\end{align*}
\]

Arguments:

<table>
<thead>
<tr>
<th>( x )</th>
<th>in</th>
<th>The input data value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( y )</td>
<td>in</td>
<td>The second input data value to subtract</td>
</tr>
</tbody>
</table>

Description: The sub function calculates \((x - y)\).

Equation:

\[
x - y = z
\]

Eqn. 6-11

Returns: The function returns the results of the subtraction \((x - y)\).

Range Issues: If saturation is enabled, the sub function will return the maximum or minimum fractional values if overflow or underflow occurs, respectively.

Special Issues: None

Design/Implementation: The sub function is implemented as a CodeWarrior C compiler-intrinsic.

Code Example 6-19. sub

```c
{
    Frac16 x16, y16, z16;
    Frac32 x32, y32, z32;
    ...
    z16 = sub (x16, y16);
    z32 = L_sub (x32, y32);
    ...
}
```
Chapter 7
Trigonometric Math Library

This section describes trigonometric functions for 16-bit fractional values.

7.1 Introduction

The Digital Signal Processing Function Library implements trigonometric functions, such as sine and cosine, for fractional types as library routines. The function library provides such trigonometric functions for both 16-bit (short) fractional types. For portability, we have defined a short 16-bit fractional type as Frac16.

The next two sections present the C interface for 16-bit fractional trigonometric functions, tfr16.h.

7.2 16-bit Fractional Interface

The C interface for the 16-bit fractional trigonometric functions is defined in the C header file tfr16.h, shown in Code Example 7-1.

Code Example 7-1. C Header File tfr16.h

```c
/* File: tfr16.h */

#ifndef __TFR16_H
#define __TFR16_H

#include "port.h"

#ifdef __cplusplus
extern "C" {
#endif

/*******************************************************
* Trigonometric Functions for 16-bit Fractional
****************************************************/

EXPORT Frac16 tfr16SinPIx      (Frac16 x);
```

For More Information On This Product, Go to: www.freescale.com
Trigonometric Math Library

EXPORT Frac16 tfr16CosPIx (Frac16 x);
EXPORT Frac16 tfr16AsinOverPI (Frac16 x);
EXPORT Frac16 tfr16AcosOverPI (Frac16 x);
EXPORT Frac16 tfr16Atan (Frac16 x);
EXPORT Frac16 tfr16Atan2OverPI (Frac16 y, Frac16 x);

/******* Trigonometric Math Library Functions *******

// Sine Wave Generation Functions for 16-bit Fractional
*******************************************************/

/* Table lookup method via integer delta */
typedef struct tfr16_sSineWaveGenIDTL
{
   Word16 PrivateData[5]; /* Private data for the IDTL sine generation function */
}tfr16_tSineWaveGenIDTL;

EXPORT tfr16_tSineWaveGenIDTL * tfr16SineWaveGenIDTLCreate(Frac16 * pSineTable,
                                                            Uint16   SineTableLength,
                                                            Int16 SineFreq,
                                                            Int16 SampleFreq,
                                                            Frac16   InitialPhasePIx);

EXPORT void tfr16SineWaveGenIDTLDestroy(tfr16_tSineWaveGenIDTL * pSWG);

EXPORT void tfr16SineWaveGenIDTLInit( tfr16_tSineWaveGenIDTL * pSWG,
                                       Frac16                 * pSineTable,
                                       Uint16                   SineTableLength,
                                       Int16                  SineFreq,
                                       Int16                  SampleFreq,
                                       Frac16                  InitialPhasePIx);

EXPORT void tfr16SineWaveGenIDTL(tfr16_tSineWaveGenIDTL * pSWG,
                                  Frac16 * pValues,
                                  Uint16 Nsamples);

/* Table lookup method via real delta */
typedef struct tfr16_sSineWaveGenRDTL
{
   Word16 PrivateData[4]; /* Private data for the RDTL sine generation function */
}tfr16_tSineWaveGenRDTL;

EXPORT tfr16_tSineWaveGenRDTL * tfr16SineWaveGenRDTLCreat(Frac16 * pSineTable,
                                                            Uint16   SineTableLength,
                                                            Int16 SineFreq,
                                                            Int16 SampleFreq,
                                                            Frac16   InitialPhasePIx);

EXPORT void tfr16SineWaveGenRDTLDestroy(tfr16_tSineWaveGenRDTL * pSWG);

EXPORT void tfr16SineWaveGenRDTL(tfr16_tSineWaveGenRDTL * pSWG,
                                  Frac16 * pValues,
                                  Uint16 Nsamples);
MOTOROLA

Trigonometric Math Library

For More Information On This Product,
Go to: www.freescale.com

16-bit Fractional Interface

EXPORT void tfr16SineWaveGenRDTLInit( tfr16_tSineWaveGenRDTL * pSWG,
Frac16 * pSineTable,
UInt16 SineTableLength,
Int16 SineFreq,
Int16 SampleFreq,
Frac16 InitialPhasePIx);

EXPORT void tfr16SineWaveGenRDTL(tfr16_tSineWaveGenRDTL * pSWG,
Frac16 * pValues,
UInt16 Nsamples);

/* Table lookup method via real delta with interpolation */
typedef struct tfr16_sSineWaveGenRDITL
{
    Word16 PrivateData[4]; /* Private data for the RDITL sine generation function */
}tfr16_tSineWaveGenRDITL;

EXPORT tfr16_tSineWaveGenRDITL * tfr16SineWaveGenRDITLCreate(Frac16 * pSineTable,
UInt16 SineTableLength,
Int16 SineFreq,
Int16 SampleFreq,
Frac16 InitialPhasePIx);

EXPORT void tfr16SineWaveGenRDITLDestroy(tfr16_tSineWaveGenRDITL * pSWG);

EXPORT void tfr16SineWaveGenRDITLInit( tfr16_tSineWaveGenRDITL * pSWG,
Frac16 * pSineTable,
UInt16 SineTableLength,
Int16 SineFreq,
Int16 SampleFreq,
Frac16 InitialPhasePIx);

EXPORT void tfr16SineWaveGenRDITL( tfr16_tSineWaveGenRDITL * pSWG,
Frac16 * pValues,
UInt16 Nsamples);

/* Table lookup method via real delta with interpolation quarter of a table */
typedef struct tfr16_sSineWaveGenRDITLQ
{
    Word16 PrivateData[5]; /* Private data for the RDITLQ sine generation function */
}tfr16_tSineWaveGenRDITLQ;

EXPORT tfr16_tSineWaveGenRDITLQ * tfr16SineWaveGenRDITLQCreate(Frac16 * pSineTable,
UInt16 SineTableLength,
Int16 SineFreq,
Int16 SampleFreq,
Frac16 InitialPhasePIx);
EXPORT void tfr16SineWaveGenRDITLQDestroy(tfr16_tSineWaveGenRDITLQ * pSWG);

EXPORT void tfr16SineWaveGenRDITLQInit(tfr16_tSineWaveGenRDITLQ * pSWG, Frac16 * pSineTable, UInt16 SineTableLength, Int16 SineFreq, Int16 SampleFreq, Frac16 InitialPhasePIx);

EXPORT void tfr16SineWaveGenRDITLQ(tfr16_tSineWaveGenRDITLQ * pSWG, Frac16 * pValues, UInt16 Nsamples);

/* Digital oscillator method */
typedef struct tfr16_sSineWaveGenDOM
{
    Word16 PrivateData[3]; /* Private data for the Digital Oscillator method */
}tfr16_tSineWaveGenDOM;

EXPORT tfr16_tSineWaveGenDOM * tfr16SineWaveGenDOMCreate( Int16   SineFreq, Int16   SampleFreq, Frac16   InitialPhasePIx, Frac16   Amplitude);

EXPORT void tfr16SineWaveGenDOMDestroy(tfr16_tSineWaveGenDOM * pSWG);

EXPORT void tfr16SineWaveGenDOMInit( tfr16_tSineWaveGenDOM * pSWG, Int16                  SineFreq, Int16                  SampleFreq, Frac16                  InitialPhasePIx, Frac16                  Amplitude);

EXPORT void tfr16SineWaveGenDOM( tfr16_tSineWaveGenDOM * pSWG, Frac16 * pValues, UInt16 Nsamples);

/* Polynomial approximation method */
typedef struct tfr16_sSineWaveGenPAM
{
    Word16 PrivateData[4]; /* Private data for the polynomial approximation method */
}tfr16_tSineWaveGenPAM;

EXPORT tfr16_tSineWaveGenPAM * tfr16SineWaveGenPAMCreate( Int16   SineFreq, Int16   SampleFreq, Frac16   InitialPhasePIx, Frac16   Amplitude);

EXPORT void tfr16SineWaveGenPAMDestroy(tfr16_tSineWaveGenPAM * pSWG);

EXPORT void tfr16SineWaveGenPAMInit( tfr16_tSineWaveGenPAM * pSWG, Int16                  SineFreq, Int16                  SampleFreq,};
EXPORT void tfr16SineWaveGenPAM(tfr16_tSineWaveGenPAM * pSWG, Frac16 * pValues, UInt16 Nsamples);

/* Table lookup method via real delta with interpolation, quarter of a sine LUT */
typedef struct tfr16_sWaveGenRDITLQ {
    Word16 PrivateData[4]; /* Private data for the RDITLQ wave generation function */
} tfr16_tWaveGenRDITLQ;

EXPORT tfr16_tWaveGenRDITLQ * tfr16WaveGenRDITLQCreate(Frac16 * pSineTable, UInt16 SineTableLength, Frac16 InitialPhasePIx);

EXPORT void tfr16WaveGenRDITLQDestroy(tfr16_tWaveGenRDITLQ * pSWG);

EXPORT void tfr16WaveGenRDITLQInit(tfr16_tWaveGenRDITLQ * pSWG, Frac16 * pSineTable, UInt16 SineTableLength, Frac16 InitialPhasePIx);

EXPORT Frac16 tfr16WaveGenRDITLQ(tfr16_tWaveGenRDITLQ * pSWG, Frac16 PhaseIncrement);

/* Table lookup method via real delta with interpolation, quarter of a sine LUT */
typedef struct tfr16_sSinPIxLUT {
    Word16 PrivateData[3]; /* Private data for the SinPIxLUT function */
} tfr16_tSinPIxLUT;

EXPORT tfr16_tSinPIxLUT * tfr16SinPIxLUTCreate(Frac16 * pSineTable, UInt16 SineTableLength);

EXPORT void tfr16SinPIxLUTDestroy(tfr16_tSinPIxLUT * pSWG);

EXPORT void tfr16SinPIxLUTInit(tfr16_tSinPIxLUT * pSWG, Frac16 * pSineTable, UInt16 SineTableLength);

EXPORT Frac16 tfr16SinPIxLUT(tfr16_tSinPIxLUT * pSWG, Frac16 PhasePIx);

#ifdef __cplusplus
}
#endif
#endif
7.3 Fractional Trigonometric Function Specs

The following pages describe the trigonometric functions for the 16-bit fractional type.

Each page utilizes a short name to generically describe its function. In many cases, however, the actual function name that must be used in C or assembler may be different from the generic function name. Therefore, refer to the C function declaration to find the specific function name to put in your program. For example, the generic function name may be $AcosOverPI$, but the actual function name that one would use for a 16-bit fractional value is $tfr16AcosOverPI$.

Function arguments for each routine are described as in, out, or inout. An in argument means that the parameter value is an input only to the function. An out argument means that the parameter value is an output only from the function. An inout argument means that a parameter value is an input to the function, but the same parameter is also an output from the function.

Typically, inout parameters are input pointer variables in which the caller passes the address of a pre-allocated data structure to a function. The function stores its results within that data structure. The actual value of the inout pointer parameter is not changed.
7.3.1 **AcosOverPI**

Call(s):

```c
Frac16 tfr16AcosOverPI (Frac16 x);
```

Arguments:

<table>
<thead>
<tr>
<th>x</th>
<th>in</th>
<th>The input data value</th>
</tr>
</thead>
</table>

**Description:** The `AcosOverPI` function calculates the Arccos function of the fractional input value `x`, and divides that result by \( \pi \); i.e., \( (\text{Arccos} \ x) / \pi \).

**Equation:**

\[
\frac{\text{acos}(x)}{\pi} = y \quad \text{Eqn. 7-1}
\]

**Returns:** The function returns result of the \( (\text{Arccos} \ x) / \pi \) calculation. This result represents the angle in radians whose cosine is `x`, divided by \( \pi \).

**Range Issues:** The result of the Arccos calculation is scaled by \( \pi \) to keep the result in the valid fractional range, \( 0 \leq (\text{Arccos} \ x) / \pi < 1 \).

**Special Issues:** None

**Design/Implementation:** The `AcosOverPI` function is implemented as a library function call.

**Code Example 7-2. ** `AcosOverPI`

```c
{
    Frac16 x16, z16;
    z16 = tfr16AcosOverPI (x16);
    ...
}
```
7.3.2 *AsinOverPI*

Call(s):

```c
Frac16 tfr16AsinOverPI (Frac16 x);
```

Arguments:

<table>
<thead>
<tr>
<th><code>x</code></th>
<th>in</th>
<th>The input data value</th>
</tr>
</thead>
</table>

Description: The *AsinOverPI* function calculates the Arcsin function of the fractional input value `x`, and divides that result by \( \pi \); i.e., \( \frac{\text{Arcsin } x}{\pi} \).

Equation:

\[
\frac{\text{asin}(x)}{\pi} = y \quad \text{Eqn. 7-2}
\]

Returns: The function returns result of the \( \frac{\text{Arcsin } x}{\pi} \) calculation. This result represents the angle in radians whose sine is \( x \), divided by \( \pi \).

Range Issues: The result of the Arcsin calculation is scaled by \( \pi \) to keep the result in the valid fractional range, \( -1/2 \leq \frac{\text{Arcsin } x}{\pi} \leq 1/2 \).

Special Issues: None

Design/Implementation: The *AsinOverPI* function is implemented as a library function call.

Code Example 7-3. *AsinOverPI*

```c
{
    Frac16 x16, z16;
    z16 = tfr16AsinOverPI (x16);
    ...
}
```
7.3.3 *AtanOverPI*

**Call(s):**

\[ \text{Frac16 tfr16AtanOverPI (Frac16 x);} \]

**Arguments:**

<table>
<thead>
<tr>
<th>( x )</th>
<th>( \text{in} )</th>
<th>The input data value</th>
</tr>
</thead>
</table>

**Description:** The *AtanOverPI* function calculates the Arctan function of the fractional input value \( x \), and divides that result by \( \pi \); i.e., \( \frac{\text{Arctan}(x)}{\pi} \).

**Equation:**

\[ \frac{\text{atan}(x)}{\pi} = y \]  

**Returns:** The function returns the result of the \( \frac{\text{Arctan}(x)}{\pi} \) calculation. This result represents the angle in radians whose tangent is \( x \), divided by \( \pi \).

**Range Issues:** The result of the Arctan calculation is divided by \( \pi \) in order to keep the result in the valid fractional range, \(-1/2 \leq \frac{\text{Arctan}(x)}{\pi} < 1/2\).

No scaling is performed on input or output values.

**Special Issues:** None

**Design/Implementation:** The *AtanOverPI* function is implemented as a library function call.

**Code Example 7-4. *AtanOverPI***

```
{ 
    Frac16 x16, z16;
    z16 = tfr16AtanOverPI (x16);
    ...
}
```
Trigonometric Math Library

7.3.4 Atan2OverPI

Call(s):

\[ \text{Frac16 tfr16Atan2OverPI (Frac16 y, Frac16 x);} \]

Arguments:

<table>
<thead>
<tr>
<th>y</th>
<th>in</th>
<th>The first input data value</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>in</td>
<td>The second input data value</td>
</tr>
</tbody>
</table>

Description: The Atan2OverPI function calculates the Arctan \( \left( \frac{y}{x} \right) \), and divides that result by \( \pi \); i.e., \( \frac{(\text{Arctan} \left( \frac{y}{x} \right))}{\pi} \).

Equation:

\[ \frac{\text{atan} \left( \frac{y}{x} \right)}{\pi} = z \]

Eqn. 7-4

Returns: The function returns result of the \( \frac{(\text{Arctan} \left( \frac{y}{x} \right))}{\pi} \) calculation.

Range Issues: The result of the Arctan2 calculation is divided by \( \pi \) in order to keep the result in the valid fractional range, \(-1 \leq \frac{(\text{Arctan} \left( \frac{y}{x} \right))}{\pi} < 1\).

Special Issues: None

Design/Implementation: The Atan2OverPI function is implemented as a library function call.

Code Example 7-5. Atan2OverPI

```c
{ 
    Frac16 x16, y16, z16;
    ...
    z16 = tfr16Atan2OverPI (y16, x16);
    ...
}
```
7.3.5 CosPIx

Call(s):

Frac16 tfr16CosPIx (Frac16 x);

Arguments:

<table>
<thead>
<tr>
<th>Table 7-5. CosPIx arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
</tr>
</tbody>
</table>

Description: The CosPIx function calculates cosine (πx).

Equation:

\[ \cos(\pi \times x) = y \]  \hspace{1cm} Eqn. 7-5

Returns: The CosPIx function returns result of the cosine (πx) calculation.

Range Issues: The range of x is -1 <= x < 1 and, therefore, the input to the cosine calculation will be within the range of -π <= πx < π.

Special Issues: None

Design/Implementation: The CosPIx function is implemented as a library function call.

Code Example 7-6. CosPIx

```c
{  
  Frac16 x16, z16;
  z16 = tfr16CosPIx (x16);
  ...  
}
```
7.3.6 \textit{SinPlx}

Call(s):

\begin{verbatim}
Frac16 tfr16SinPlx (Frac16 x);
\end{verbatim}

Arguments:

<table>
<thead>
<tr>
<th>$x$</th>
<th>in</th>
<th>The input data value</th>
</tr>
</thead>
</table>

Description: The \textit{SinPlx} function calculates sine ($\pi x$).

Equation:

\[
\sin(\pi x) = y
\]

Eqn. 7-6

Returns: The \textit{SinPlx} function returns result of the sine ($\pi x$) calculation.

Range Issues: The range of $x$ is $-1 \leq x < 1$ and, therefore, the input to the sine calculation will be within the range of $-\pi \leq \pi x < \pi$.

Special Issues: None

Design/Implementation: The \textit{SinPlx} function is implemented as a library function call.

Code Example 7-7. \textit{SinPlx}

\begin{verbatim}
{
    Frac16 x16, z16;
    z16 = tfr16SinPlx (x16);
    \ldots
}
\end{verbatim}
7.3.7 SineWaveGenIDTL

Call(s):

tfr16_tSineWaveGenIDTL * tfr16SineWaveGenIDTLCreate(Frac16 * pSineTable,
    Uint16 SineTableLength,
    Int16 SineFreq,
    Int16 SampleFreq,
    Frac16 InitialPhasePIx);

void tfr16SineWaveGenIDTLDestroy(tfr16_tSineWaveGenIDTL * pSWG);

void tfr16SineWaveGenIDTLInit(tfr16_tSineWaveGenIDTL * pSWG,
    Frac16 * pSineTable,
    Uint16 SineTableLength,
    Int16 SineFreq,
    Int16 SampleFreq,
    Frac16 InitialPhasePIx);

void tfr16SineWaveGenIDTL(tfr16_tSineWaveGenIDTL * pSWG,
    Frac16 * pValues,
    Uint16 Nsamples);

Arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pSWG</td>
<td>in</td>
<td>Pointer to tfr16_SineWaveGenIDTL type defined in tfr16.h header file. It points to private data for the IDTL sine wave generation function.</td>
</tr>
<tr>
<td>pSineTable</td>
<td>in</td>
<td>Pointer to a sine look-up table which is allocated and filled by the application</td>
</tr>
<tr>
<td>SineTableLength</td>
<td>in</td>
<td>The length of the sine look-up table</td>
</tr>
<tr>
<td>SineFreq</td>
<td>in</td>
<td>The frequency of the sine wave to be generated</td>
</tr>
<tr>
<td>SampleFreq</td>
<td>in</td>
<td>The frequency to sample the sine wave</td>
</tr>
<tr>
<td>InitialPhasePIx</td>
<td>in</td>
<td>The starting phase of the sine wave to be generated</td>
</tr>
<tr>
<td>pValues</td>
<td>in</td>
<td>A pointer to memory allocated by the application to store sine samples</td>
</tr>
<tr>
<td>Nsamples</td>
<td>in</td>
<td>The number of sine samples to generate</td>
</tr>
</tbody>
</table>

Description: This method of sine wave generation uses “Integer Delta Direct Look-up”, which is done by stepping through a predefined sine table. The sine table has a full cycle of sine values, stored at a constant rate.

This method calculates an index into a sine look-up table and retrieves a value. Successive values are obtained by adding a delta to the previous index and modulo the length of the sine table.
This technique of sine wave generation requires the sample frequency to be devisable by the sine look-up table length.

**tfr16SineWaveGenIDTLcreate:**

The `tfr16SineWaveGenIDTLCreate` function allocates the structure of type `tfr16_tSineWaveGenIDTL`.

The function `tfr16SineWaveGenIDTLCreate` returns a pointer to the `tfr16_tSineWaveGenIDTL` structure, which is used by all other `tfr16SineWaveGenIDTL` functions.

In general, the data structure `tfr16_tSineWaveGenIDTL` allocated by `tfr16SineWaveGenIDTLCreate` is treated as private data to better encapsulate the details of the implementation and thus provide portability.

**tfr16SineWaveGenIDTLInit:**

The `tfr16SineWaveGenIDTLInit` function initializes the `tfr16_tSineWaveGenIDTL` data structure used by all `tfr16SineWaveGenIDTL` functions. The `tfr16_tSineWaveGenIDTL` data structure pointed to by `pSWG` must have been allocated prior to calling `tfr16SineWaveGenIDTLInit`.

The primary reason to use `tfr16SineWaveGenIDTLInit` directly is to statically allocate the `tfr16_tSineWaveGenIDTL` data structure, rather than dynamically allocating it with `tfr16SineWaveGenIDTLCreate`. The data structure `tfr16_tSineWaveGenIDTL`, utilized in the `tfr16.h` interface, has been declared as private data in order to better encapsulate the details of the implementation and thus provide portability. However, to utilize the `tfr16SineWaveGenIDTLInit` routine directly, the application must allocate storage space for the `tfr16_tSineWaveGenIDTL` data structure before calling this function.

**tfr16SineWaveGenIDTL:***

The `tfr16SineWaveGenIDTL` function generates `Nsamples` and stores them in `pValues`.

**tfr16SineWaveGenIDTLDestroy:**

The `tfr16SineWaveGenIDTLDestroy` function frees the `tfr16_tSineWaveGenIDTL` data structures(s) previously allocated by `tfr16SineWaveGenIDTLCreate`.

**Equation:**

\[ \sin(x[1\ldots n] \times \pi) = y[1\ldots n] \]  

**Eqn. 7-7**

**Returns:** None

**Algorithm:**

\[ \text{SineValue} = \text{SineTable}[\text{Index}] \]

\[ \text{Index} = (\text{Index} + \text{Delta}) \mod \text{SineTableLength} \]

**Range Issues:** The range of InitialPhasePIx is \(-1 \leq \text{InitialPhasePIx} < 1\) and, therefore, the input will be within the range of \(-\pi \leq \pi(\text{InitialPhasePIx}) < \pi\).
Special Issues: The advantages of this implementation are high speed and low distortion for the allowed frequencies. The disadvantage of this implementation is that the sample frequency must be divisible by the length of the sine look-up table.

Design/Implementation: The \texttt{tfr16SineWaveGenIDTL} functions are implemented as library function calls.

Code Example 7-8. \textit{SineWaveGenIDTL}

```c
#include "tfr16.h"

#define LENGTH 1000

extern Frac16 SineTable[LENGTH];
extern Frac16 Values[LENGTH];

{ tfr16_tSineWaveGenIDTL * pIDTLHandle;
  UInt16 SineTableLength = LENGTH;
  Int16 SineFreq = 3200;
  Int16 SampleFreq = 32000;
  UInt16 Nsamples = LENGTH;
  Frac16 InitialPhasePIx = 16384; /*.5*\pi radians */

  pIDTLHandle = tfr16SineWaveGenIDTLCreate( &SineTable[0],
                                           SineTableLength,
                                           SineFreq,
                                           SampleFreq,
                                           Nsamples,
                                           InitialPhasePIx);

  tfr16SineWaveGenIDTL(pIDTLHandle, &Values[0], Nsamples);
  tfr16SineWaveGenIDTL_Destroy(pIDTLHandle);

  ...
}
```
7.3.8 **SineWaveGenRDTL**

**Call(s):**

```c
void tfr16SineWaveGenRDTLDestroy(tfr16_tSineWaveGenRDTL * pSWG);
void tfr16SineWaveGenRDTLInit(tfr16_tSineWaveGenRDTL * pSWG,
                               Frac16 * pSineTable,
                               UInt16 SineTableLength,
                               Int16 SineFreq,
                               Int16 SampleFreq,
                               Frac16 InitialPhasePIx);
void tfr16SineWaveGenRDTL(tfr16_tSineWaveGenRDTL * pSWG,
                          Frac16 * pValues,
                          UInt16 Nsamples);
```

**Arguments:**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pSWG</td>
<td>Pointer to <code>tfr16_SineWaveGenRDTL</code> type defined in <code>tfr16.h</code> header file. It points to private data for the RDTL sine wave generation function.</td>
</tr>
<tr>
<td>pSineTable</td>
<td>Pointer to a sine look-up table which is allocated and filled by the application</td>
</tr>
<tr>
<td>SineTableLength</td>
<td>The length of the sine look-up table</td>
</tr>
<tr>
<td>SineFreq</td>
<td>The frequency of the sine wave to be generated</td>
</tr>
<tr>
<td>SampleFreq</td>
<td>The frequency to sample the sine wave</td>
</tr>
<tr>
<td>InitialPhasePIx</td>
<td>The starting phase of the sine wave to be generated</td>
</tr>
<tr>
<td>pValues</td>
<td>A pointer to memory allocated by the application to store sine samples</td>
</tr>
<tr>
<td>Nsamples</td>
<td>The number of sine samples to generate</td>
</tr>
</tbody>
</table>

**Description:** This method of sine wave generation uses “Real Delta Direct Look-up”. In this technique, sine waves are generated by stepping through a sine table. The sine table has a full cycle of sine values, stored at a constant rate.

This method calculates an index into a sine look-up table and retrieves a value. Successive values are obtained by adding a delta to the previous index.
This technique of sine wave generation allows the delta and the index into the sine table to be fractional. If the index is not an integer, the algorithm takes the integer part of the index to obtain a sine value, synthesizing a continuous range of frequencies.

**tfr16SineWaveGenRDTLCreaten**:  
The tfr16SineWaveGenRDTLCreaten function allocates the structure of type tfr16_tSineWaveGenRDTL.  
The function tfr16SineWaveGenRDTLCreaten returns a pointer to the tfr16_tSineWaveGenRDTL structure, which is used by all other tfr16SineWaveGenRDTL functions.  
In general, the data structure tfr16_tSineWaveGenRDTL allocated by tfr16SineWaveGenRDTLCreaten is treated as private data to better encapsulate the details of the implementation and to provide portability.

**tfr16SineWaveGenRDTLInit**:  
The tfr16SineWaveGenRDTLInit function initializes the tfr16_tSineWaveGenRDTL data structure used by all tfr16SineWaveGenRDTL functions. The tfr16_tSineWaveGenRDTL data structure pointed to by pSWG must have been allocated prior to calling tfr16SineWaveGenRDTLInit.  
The primary reason to use tfr16SineWaveGenRDTLInit directly is to statically allocate the tfr16_tSineWaveGenRDTL data structure, rather than allocating it dynamically with tfr16SineWaveGenRDTLCreaten. The data structure tfr16_tSineWaveGenRDTL, utilized in the tfr16.h interface, has been declared as private data to better encapsulate the details of the implementation and to provide portability. However, to utilize the tfr16SineWaveGenRDTLInit routine directly, the application must allocate storage space for the tfr16_tSineWaveGenRDTL data structure before calling this function.

**tfr16SineWaveGenRDTL**:  
The tfr16SineWaveGenRDTL function generates Nsamples and stores them in pValues.

**tfr16SineWaveGenRDTLDestroy**:  
The tfr16SineWaveGenRDTLDestroy function frees the tfr16_tSineWaveGenRDTL data structures(s) previously allocated by tfr16SineWaveGenRDTLCreaten.

**Equation**:  
\[
\sin(x[1...n] \times \pi) = y[1...n]
\]

**Eqn. 7-8**

**Returns**: None

**Algorithm**:  
\[
\text{Index} = \text{integer}_\text{part}_\text{of}(\text{Index})
\]
\[
\text{SineValue} = \text{SineTable}[\text{Index}]
\]
\[
\text{Index} = (\text{Index} + \text{Delta}) \mod \text{SineTableLength}
\]

**Range Issues**: The range of InitialPhasePIx is \(-1 \leq \text{InitialPhasePIx} < 1\) and, therefore, the input will be within the range of \(-\pi \leq \pi(\text{InitialPhasePIx}) < \pi\).
Special Issues: The advantage of this implementation is the ability to synthesize a continuous range of frequencies. The disadvantages of this implementation are that there will be some distortion in the sine wave and increased processing time, compared to the SineWaveGenIDTL.

Design/Implementation: The tfr16SineWaveGenRDTL functions are implemented as library function calls.

Code Example 7-9. SineWaveGenRDTL

```c
#include "tfr16.h"

#define LENGTH 1000

extern Frac16 SineTable[LENGTH];
extern Frac16 Values[LENGTH];

{ tfr16_tSineWaveGenRDTL * pRDTLHandle;
  UInt16 SineTableLength = LENGTH;
  Int16 SineFreq = 3200;
  Int16 SampleFreq = 32000;
  UInt16 Nsamples = LENGTH;
  Frac16 InitialPhasePIx = 16384; /* .5 * π radians */

  pRDTLHandle = tfr16SineWaveGenRDTLCreate(
    &SineTable[0],
    SineTableLength,
    SineFreq,
    SampleFreq,
    InitialPhasePIx);

  tfr16SineWaveGenRDTL(pRDTLHandle, &Values[0], Nsamples);
  tfr16SineWaveGenRDTLDestroy(pRDTLHandle);
}
```
7.3.9 SineWaveGenRDITL

Call(s):

\begin{verbatim}
tfr16_tSineWaveGenRDITL * tfr16SineWaveGenRDITLCreate(Frac16 * pSineTable, Uint16 SineTableLength, Int16 SineFreq, Int16 SampleFreq, Frac16 InitialPhasePIx);
void tfr16SineWaveGenRDITLDestroy(tfr16_tSineWaveGenRDITL * pSWG);
void tfr16SineWaveGenRDITLInit(tfr16_tSineWaveGenRDITL * pSWG, Frac16 * pSineTable, Uint16 SineTableLength, Int16 SineFreq, Int16 SampleFreq, Frac16 InitialPhasePIx);
void tfr16SineWaveGenRDITL(tfr16_tSineWaveGenRDITL * pSWG, Frac16 * pValues, Uint16 Nsamples);
\end{verbatim}

Arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pSWG</td>
<td>Pointer to tfr16_SineWaveGenRDITL type defined in tfr16.h header file. It points to private data for the RDITL sine wave generation function.</td>
</tr>
<tr>
<td>pSineTable</td>
<td>Pointer to a sine look-up table which is allocated and filled by the application.</td>
</tr>
<tr>
<td>SineTableLength</td>
<td>The length of the sine look-up table</td>
</tr>
<tr>
<td>SineFreq</td>
<td>The frequency of the sine wave to be generated</td>
</tr>
<tr>
<td>SampleFreq</td>
<td>The frequency to sample the sine wave</td>
</tr>
<tr>
<td>InitialPhasePIx</td>
<td>The starting phase of the sine wave to be generated</td>
</tr>
<tr>
<td>pValues</td>
<td>A pointer to memory allocated by the application to store sine samples</td>
</tr>
<tr>
<td>Nsamples</td>
<td>The number of sine samples to generate</td>
</tr>
</tbody>
</table>

Description: This method of sine wave generation uses “Real Delta Loop-up with Interpolation”. In this technique, sine waves are generated by stepping through a sine table. The sine table has a full cycle of sine values, stored at a constant rate.

This method calculates an index into a sine look-up table and retrieves a value. Successive values are obtained by adding a delta to the previous index.
This technique of sine wave generation allows the delta and the index into the sine table to be fractional. To get a sine value, linear interpolation is done between the two table entries between which the index falls, synthesizing a continuous range of frequencies.

**tfr16SineWaveGenRDITLCreate:**

The `tfr16SineWaveGenRDITLCreate` function allocates the structure of type `tfr16_tSineWaveGenRDITL`.

The function `tfr16SineWaveGenRDITLCreate` returns a pointer to the `tfr16_tSineWaveGenRDITL` structure, which is used by all other `tfr16SineWaveGenRDITL` functions.

In general, the data structure `tfr16_tSineWaveGenRDITL` allocated by `tfr16SineWaveGenRDITLCreate` is treated as private data to better encapsulate the details of the implementation and to provide portability.

**tfr16SineWaveGenRDITLInit:**

The `tfr16SineWaveGenRDITLInit` function initializes the `tfr16_tSineWaveGenRDITL` data structure used by all `tfr16SineWaveGenRDITL` functions. The `tfr16_tSineWaveGenRDITL` data structure pointed to by `pSWG` must have been allocated prior to calling `tfr16SineWaveGenRDITLInit`.

The primary reason to use `tfr16SineWaveGenRDITLInit` directly is to statically allocate the `tfr16_tSineWaveGenRDITL` data structure, rather than allocating it dynamically with `tfr16SineWaveGenRDITLCreate`. The data structure `tfr16_tSineWaveGenRDITL`, utilized in the `tfr16.h` interface, has been declared as private data to better encapsulate the details of the implementation and to provide portability. However, to utilize the `tfr16SineWaveGenRDITLInit` routine directly, the application must allocate storage space for the `tfr16_tSineWaveGenRDITL` data structure before calling this function.

**tfr16SineWaveGenRDITL:**

The `tfr16SineWaveGenRDITL` function generates `Nsamples` and stores them in `pValues`.

**tfr16SineWaveGenRDITLDestroy:**

The `tfr16SineWaveGenRDITLDestroy` function frees the `tfr16_tSineWaveGenRDITL` data structures(s) previously allocated by `tfr16SineWaveGenRDITLCreate`.

**Equation:**

\[
\sin(x[1...n] \times \pi) = y[1...n]
\]

**Eqn. 7-9**

**Returns:** None

**Algorithm:**

\[
\text{Index} = \text{integer\_part\_of(Index)}
\]

\[
\text{Index2} = (\text{Index} + 1) \mod \text{SineTableLength}
\]

\[
\text{Delta} = \text{fractional\_part\_of(Index)}
\]

\[
\text{SineDelta} = \text{SineTable[Index]} - \text{SineTable[Index2]}
\]
SineValue = SineTable[Index] + Delta * SineDelta

Index = (Index + Delta) mod SineTableLength

**Range Issues:** The range of InitialPhasePIx is -1 <= InitialPhasePIx < 1 and, therefore, the input will be within the range of -π <= π(InitialPhasePIx) < π.

**Special Issues:** The size of the sine look-up table must be on the order of $Size = ((2^n) + 1) <= 8192$. Due to interpolation comparing between K and K+1 entries in the table, the last two values in the table must be equivalent. For example, if the size of your table is 257, the 257th value should equal the 256th. When passing in the size of the sine look-up table in the `tfr16WaveGenRDITLQCreate` function, pass in $Size - 1$, as illustrated in **Code Example 7-10**.

**Design/Implementation:** The `tfr16SineWaveGenRDITL` functions are implemented as library function calls.

---

**Code Example 7-10. SineWaveGenRDITL**

```c
#include "tfr16.h"

#define LENGTH 257

extern Frac16 SineTable[LENGTH];
extern Frac16 Values[LENGTH];

{ tfr16_tSineWaveGenRDITL * pRDITLHandle;
  UInt16 SineTableLength = LENGTH - 1;
  Int16 SineFreq = 3200;
  Int16 SampleFreq = 32000;
  UInt16 Nsamples = LENGTH;
  Frac16 InitialPhasePIx = 16384; /* .5 * π radians */

  ... pRDITLHandle = tfr16SineWaveGenRDITLCreate( &SineTable[0],
    SineTableLength,
    SineFreq,
    SampleFreq,
    InitialPhasePIx);

  tfr16SineWaveGenRDITL(pRDITLHandle, &Values[0], Nsamples);
  tfr16SineWaveGenRDITLDestroy(pRDITLHandle);

  ...}
```
7.3.10 SineWaveGenRDITLQ

Call(s):

```c
trf16_tSineWaveGenRDITLQ * tfr16SineWaveGenRDITLQCreate( Frac16 * pSineTable, UInt16 SineTableLength, Int16 SineFreq, Int16 SampleFreq, Frac16 InitialPhasePIx);

void tfr16SineWaveGenRDITLQDestroy(trf16_tSineWaveGenRDITLQ * pSWG);

void tfr16SineWaveGenRDITLQInit(trf16_tSineWaveGenRDITLQ * pSWG, Frac16 * pSineTable, UInt16 SineTableLength, Int16 SineFreq, Int16 SampleFreq, Frac16 InitialPhasePIx);

void tfr16SineWaveGenRDITLQ(trf16_tSineWaveGenRDITLQ * pSWG, Frac16 * pValues, UInt16 Nsamples);
```

Arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pSWG</td>
<td>Pointer to <code>trf16_SineWaveGenRDITLQ</code> type defined in <code>trf16.h</code> header file.</td>
</tr>
<tr>
<td></td>
<td>It points to private data for the RDITL sine wave generation function.</td>
</tr>
<tr>
<td>pSineTable</td>
<td>Pointer to a sine look-up table which is allocated and filled by the application.</td>
</tr>
<tr>
<td>SineTableLength</td>
<td>The length of the sine look-up table</td>
</tr>
<tr>
<td>SineFreq</td>
<td>The frequency of the sine wave to be generated</td>
</tr>
<tr>
<td>SampleFreq</td>
<td>The frequency to sample the sine wave</td>
</tr>
<tr>
<td>InitialPhasePIx</td>
<td>The starting phase of the sine wave to be generated</td>
</tr>
<tr>
<td>pValues</td>
<td>A pointer to memory allocated by the application to store sine samples</td>
</tr>
<tr>
<td>Nsamples</td>
<td>The number of sine samples to generate</td>
</tr>
</tbody>
</table>

Description: This method of sine wave generation uses “Real Delta Loop-up with Interpolation”. In this technique, sine waves are generated by stepping through a sine table. The sine table has a one-quarter of a full cycle of sine values, stored at a constant rate.

This method calculates an index into a sine look-up table and retrieves a value. Successive values are obtained by adding a delta to the previous index.
This technique of sine wave generation allows the delta and the index into the sine table to be fractional. To get a sine value, linear interpolation is done between the two table entries between which the index falls, synthesizing a continuous range of frequencies.

**tfr16SineWaveGenRDITLQCreate:**

The `tfr16SineWaveGenRDITLQCreate` function allocates the structure of type `tfr16_tSineWaveGenRDITLQ`. The function `tfr16SineWaveGenRDITLQCreate` returns a pointer to the `tfr16_tSineWaveGenRDITLQ` structure, which is used by all other `tfr16SineWaveGenRDITLQ` functions. In general, the data structure `tfr16_tSineWaveGenRDITLQ` allocated by `tfr16SineWaveGenRDITLQCreate` is treated as private data to better encapsulate the details of the implementation and to provide portability.

**tfr16SineWaveGenRDITLQInit:**

The `tfr16SineWaveGenRDITLQInit` function initializes the `tfr16_tSineWaveGenRDITLQ` data structure used by all `tfr16SineWaveGenRDITLQ` functions. The `tfr16_tSineWaveGenRDITLQ` data structure pointed to by `pSWG` must have been allocated prior to calling `tfr16SineWaveGenRDITLQInit`. The primary reason to use `tfr16SineWaveGenRDITLQInit` directly is to statically allocate the `tfr16_tSineWaveGenRDITLQ` data structure, rather than allocating it dynamically with `tfr16SineWaveGenRDITLQCreate`. The data structure `tfr16_tSineWaveGenRDITLQ`, utilized in the `tfr16.h` interface, has been declared as private data to better encapsulate the details of the implementation and to provide portability. However, to utilize the `tfr16SineWaveGenRDITLQInit` routine directly, the application must allocate storage space for the `tfr16_tSineWaveGenRDITLQ` data structure before calling this function.

**tfr16SineWaveGenRDITLQ:**

The `tfr16SineWaveGenRDITLQ` function generates `Nsamples` and stores them in `pValues`.

**tfr16SineWaveGenRDITLQDestroy:**

The `tfr16SineWaveGenRDITLQDestroy` function frees the `tfr16_tSineWaveGenRDITLQ` data structures(s) previously allocated by `tfr16SineWaveGenRDITLQCreate`.

**Equation:**

\[
\sin(x[1\ldots n] \times \pi) = y[1\ldots n]
\]

**Eqn. 7-10**

**Returns:** None

**Algorithm:**

\[
\text{Index} = \text{integer\_part\_of}(\text{Index})
\]

\[
\text{Index2} = (\text{Index} + 1) \mod \text{SineTableLength}
\]

\[
\text{Delta} = \text{fractional\_part\_of}(\text{Index})
\]

\[
\text{SineDelta} = \text{SineTable}[\text{Index}] - \text{SineTable}[\text{Index2}]
\]
SineValue = SineTable[Index] + Delta * SineDelta

Index = (Index + Delta) \text{mod} \ SineTableLength

**Range Issues:** The range of InitialPhasePIx is -1 <= InitialPhasePIx < 1 and therefore the input will be within the range of -\pi <= \pi(InitialPhasePIx) < \pi. The size of the sine look-up table must be on the order of \( Size = ((2^n) + 1) < 8192 \) and contain only a quarter of a full sine table. Due to interpolation comparing between K and K+1 entries in the table, the last two values in the table must be equivalent. For example, if the size of your table is 257, the 257th value should equal the 256th. When passing in the size of the sine look-up table in the \textit{tfr16WaveGenRDITLQCreate} function, pass in \( Size - 1 \). This is illustrated in \textbf{Code Example 7-11}.

**Special Issues:** The advantage of this implementation is the ability to synthesize a continuous range of frequencies. There is also less distortion by using this method than in using \textit{SineWaveGenRDLT}. Less memory is used compared to \textit{SineWaveGenRDITL} by using only a quarter of a full sine wave cycle in the table. The disadvantage of this implementation is increased processing time compared to \textit{SineWaveGenRDLT} and \textit{SineWaveGenRDITL}.

**Design/Implementation:** The \textit{tfr16SineWaveGenRDITL} functions are implemented as library function calls.

**Code Example 7-11. SineWaveGenRDITLQ**

```c
#include "tfr16.h"

#define LENGTH 257

extern Frac16 SineTable[LENGTH];
extern Frac16 Values[LENGTH];

{ tfr16_tSineWaveGenRDITLQ * pRDITLQHandle;
  Uint16   SineTableLength = LENGTH - 1;
  Int16    SineFreq = 3200;
  Int16    SampleFreq = 32000;
  Uint16   Nsamples = LENGTH;
  Frac16   InitialPhasePIx = 16384; /*.5*\pi radians */

  pRDITLQHandle = tfr16SineWaveGenRDITLQCreate( &SineTable[0],
                                                  SineTableLength,
                                                  SineFreq,
                                                  SampleFreq,
                                                  InitialPhasePIx);

  tfr16SineWaveGenRDITLQ(pRDITLQHandle, &Values[0], Nsamples);
  tfr16SineWaveGenRDITLQDestroy(pRDITLQHandle);

... ```
7.3.11 *SineWaveGenPAM*

**Call(s):**

```c
void tfr16SineWaveGenPAMInit( tfr16_tSineWaveGenPAM * pSWG,
Int16  SineFreq,
Int16  SampleFreq,
Frac16 InitialPhasePIx,
Frac16 Amplitude);
```

```c
void tfr16SineWaveGenPAM( tfr16_tSineWaveGenPAM * pSWG,
Frac16 * pValues,
Uint16 Nsamples);
```

```c
void tfr16SineWaveGenPAMDestroy(tfr16_tSineWaveGenPAM * pSWG);
```

**Arguments:**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pSWG</td>
<td>Pointer to tfr16_SineWaveGenPAM type defined in tfr16.h header file. It points to private data for the PAM sine wave generation function.</td>
</tr>
<tr>
<td>SineFreq</td>
<td>The frequency of the sine wave to be generated</td>
</tr>
<tr>
<td>SampleFreq</td>
<td>The frequency to sample the sine wave</td>
</tr>
<tr>
<td>InitialPhasePIx</td>
<td>The starting phase of the sine wave to be generated</td>
</tr>
<tr>
<td>Amplitude</td>
<td>A scaling factor of the generated sine wave</td>
</tr>
<tr>
<td>pValues</td>
<td>A pointer to memory allocated by the application to store sine samples</td>
</tr>
<tr>
<td>Nsamples</td>
<td>The number of sine samples to generate</td>
</tr>
</tbody>
</table>

**Description:** This method of sine wave generation uses a polynomial approximation. The sine wave is created by first generating angles for each sample, then calculating the corresponding sine value using a polynomial approximation.

**tfr16SineWaveGenPAMCreate:**

The `tfr16SineWaveGenPAMCreate` function allocates the structure of type `tfr16_SineWaveGenPAM`.

The function `tfr16SineWaveGenPAMCreate` returns a pointer to the `tfr16_tSineWaveGenPAM` structure, which is used by all other `tfr16SineWaveGenPAM` functions.
In general, the data structure `tfr16_tSineWaveGenPAM` allocated by `tfr16SineWaveGenPAMCreate` is treated as private data to better encapsulate the details of the implementation and to provide portability.

### `tfr16SineWaveGenPAMInit`:

The `tfr16SineWaveGenPAMInit` function initializes the `tfr16_tSineWaveGenPAM` data structure used by all `tfr16SineWaveGenPAM` functions. The `tfr16_tSineWaveGenPAM` data structure pointed to by `pSWG` must have been allocated prior to calling `tfr16SineWaveGenPAMInit`.

The primary reason to use `tfr16SineWaveGenPAMInit` directly is to statically allocate the `tfr16_tSineWaveGenPAM` data structure, rather than allocating it dynamically with `tfr16SineWaveGenPAMCreate`. The data structure `tfr16_tSineWaveGenPAM`, utilized in the `tfr16.h` interface, has been declared as private data to better encapsulate the details of the implementation and to provide portability. However, to utilize the `tfr16SineWaveGenPAMInit` routine directly, the application must allocate storage space for the `tfr16_tSineWaveGenPAM` data structure before calling this function.

### `tfr16SineWaveGenPAM`:

The `tfr16SineWaveGenPAM` function generates `Nsamples` and stores them in `pValues`.

### `tfr16SineWaveGenPAMDestroy`:

The `tfr16SineWaveGenPAMDestroy` function frees the `tfr16_tSineWaveGenPAM` data structures(s) previously allocated by `tfr16SineWaveGenPAMCreate`.

### Equation:

\[
\sin(x[1\ldots n] \times \pi) = y[1\ldots n]
\]

*Eqn. 7-11*

### Returns:

None

### Algorithm:

\[
\text{Sin}(x) = C_1 x + C_3 x^3 + C_5 x^5
\]

### Range Issues:

The range of `InitialPhasePIx` is `-1 \leq \text{InitialPhasePIx} < 1` and, therefore, the input will be within the range of `-\pi \leq \pi(\text{InitialPhasePIx}) < \pi`. The range of Amplitude is `0 < \text{Amplitude} \leq 1`.

### Special Issues:

The advantages of this implementation are low storage requirements and low distortion. The disadvantage of this method is greater processing time than the other methods discussed.

### Design/Implementation:

The `tfr16SineWaveGenPAM` functions are implemented as library function calls.
Code Example 7-12.  SineWaveGenPAM

```c
#include "tfr16.h"

#define ONEHALF 16384
#define LENGTH 1000

extern Frac16 Values[LENGTH];

{
    tfr16_tSineWaveGenPAM * pPAMHandle;
    Int16 SineFreq = 3200;
    Int16 SampleFreq = 32000;
    UInt16 Nsamples = LENGTH;
    Frac16 InitialPhasePIx = ONEHALF; /* .5*π */
    Frac16 Amplitude = ONEHALF; /* .5 */

    pPAMHandle = tfr16SineWaveGenPAMCreate(SineFreq,
                                           SampleFreq,
                                           InitialPhasePIx,
                                           Amplitude);

    tfr16SineWaveGenPAM(pPAMHandle, &Values[0], Nsamples);
    tfr16SineWaveGenPAMDestroy(pPAMHandle);
}
```
7.3.12 **SineWaveGenDOM**

**Call(s):**

```c
trfr16_tSineWaveGenDOM * trfr16SineWaveGenDOMCreate(Int16 SineFreq,
                                                      Int16 SampleFreq,
                                                      Frac16 InitialPhasePIx,
                                                      Frac16 Amplitude);
```

```c
void trfr16SineWaveGenDOMDestroy(trfr16_tSineWaveGenDOM * pSWG);
```

```c
void trfr16SineWaveGenDOMInit(trfr16_tSineWaveGenDOM * pSWG,
                               Int16 SineFreq,
                               Int16 SampleFreq,
                               Frac16 InitialPhasePIx,
                               Frac16 Amplitude);
```

```c
void trfr16SineWaveGenDOM(trfr16_tSineWaveGenDOM * pSWG,
                          Frac16 * pValues,
                          UInt16 Nsamples);
```

**Arguments:**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pSWG</td>
<td>in</td>
<td>Pointer to <code>trfr16_SineWaveGenDOM</code> type defined in <code>trfr16.h</code> header file. It points to private data for the DOM sine wave generation function.</td>
</tr>
<tr>
<td>SineFreq</td>
<td>in</td>
<td>The frequency of the sine wave to be generated</td>
</tr>
<tr>
<td>SampleFreq</td>
<td>in</td>
<td>The frequency to sample the sine wave</td>
</tr>
<tr>
<td>InitialPhasePIx</td>
<td>in</td>
<td>The starting phase of the sine wave to be generated</td>
</tr>
<tr>
<td>Amplitude</td>
<td>in</td>
<td>A scaling factor of the generated sine wave</td>
</tr>
<tr>
<td>pValues</td>
<td>in</td>
<td>A pointer to memory allocated by the application to store sine samples</td>
</tr>
<tr>
<td>Nsamples</td>
<td>in</td>
<td>The number of sine samples to generate</td>
</tr>
</tbody>
</table>

**Description:** This method of sine wave generation uses a second-order IIR filter with its pole on the unit circle. The pole angle gives the frequency of the synthesized sine wave.

**trfr16SineWaveGenDOMCreate:**

The `trfr16SineWaveGenDOMCreate` function allocates the structure of type `trfr16_tSineWaveGenPAM`.

The function `trfr16SineWaveGenDOMCreate` returns a pointer to the `trfr16_tSineWaveGenPAM` structure, which is used by all other `trfr16SineWaveGenDOM` functions.
In general, the data structure `tfr16_tSineWaveGenPAM` allocated by `tfr16SineWaveGenDOMCreate` is treated as private data to better encapsulate the details of the implementation and to provide portability.

### `tfr16SineWaveGenDOMInit`:

The `tfr16SineWaveGenDOMInit` function initializes the `tfr16_tSineWaveGenPAM` data structure used by all `tfr16SineWaveGenDOM` functions. The `tfr16_tSineWaveGenPAM` data structure pointed to by `pSWG` must have been allocated prior to calling `tfr16SineWaveGenDOMInit`.

The primary reason to use `tfr16SineWaveGenDOMInit` directly is to statically allocate the `tfr16_tSineWaveGenPAM` data structure, rather than allocating it dynamically with `tfr16SineWaveGenDOMCreate`. The data structure `tfr16_tSineWaveGenPAM`, utilized in the `tfr16.h` interface, has been declared as private data to better encapsulate the details of the implementation and to provide portability. However, to utilize the `tfr16SineWaveGenDOMInit` routine directly, the application must allocate storage space for the `tfr16_tSineWaveGenPAM` data structure before calling this function.

### `tfr16SineWaveGenDOM`:

The `tfr16SineWaveGenDOM` function generates `Nsamples` and stores them in `pValues`.

### `tfr16SineWaveGenPAMDestroy`:

The `tfr16SineWaveGenPAMDestroy` function frees the `tfr16_tSineWaveGenPAM` data structures(s) previously allocated by `tfr16SineWaveGenPAMCreate`.

### Equation:

\[
\sin(x[1\ldots n] \times \pi) = y[1\ldots n]
\]

### Eqn. 7-12

### Returns: None

### Algorithm:

The difference equation for the oscillator is:

\[
x(n) = 2\cos(2\pi f)(x(n-1)) - x(n-2)
\]

where “x(n)” is the sine wave signal and “f” is the required frequency (normalized with respect to the sampling frequency). The phase and amplitude of the sine wave are determined by the initial states x(-1) and x(-2).

### Range Issues:

The range of `InitialPhasePIx` is `-1 <= InitialPhasePIx < 1` and, therefore, the input will be within the range of `-\pi <= \pi(InitialPhasePIx) < \pi`. The range of `Amplitude` is `0 < Amplitude <= 1`.

### Special Issues:

The advantages of this implementation are low storage requirements and high speed. The disadvantage of this method is that the quality of the generated sine-wave greatly depends on the exact value of the frequency to be generated. Distortion is greater than that of the table look-up methods `tfr16SineWaveGenRDTL` or `tfr16SineWaveGenRDITL` for most frequencies, but for some frequencies, distortion is less than can accurately be represented.

### Design/Implementation:

The `tfr16SineWaveGenDOM` functions are implemented as library function calls.
Code Example 7-13. SineWaveGenDOM

```c
{
    #include "tfr16.h"

    #define ONEHALF 16384
    #define LENGTH 1000

    extern Frac16 Values[LENGTH];

    tfr16_tSineWaveGenDOM * pDOMHandle;
    Int16 SineFreq = 3200;
    Int16 SampleFreq = 32000;
    UInt16 Nsamples = LENGTH;
    Frac16 InitialPhasePIx = ONEHALF; /* .5*π */
    Frac16 Amplitude = ONEHALF; /* .5 */

    pDOMHandle = tfr16SineWaveGenDOMCreate(SineFreq,
                                            SampleFreq,
                                            InitialPhasePIx,
                                            Amplitude);
    tfr16SineWaveGenDOM(pDOMHandle, &Values[0], Nsamples);
    tfr16SineWaveGenDOMDestroy(pDOMHandle);
    ...
}
```
### 7.3.13 WaveGenRDITLQ

**Call(s):**

```c
void tfr16WaveGenRDITLQDestroy(tfr16_tWaveGenRDITLQ * pSWG);
void tfr16WaveGenRDITLQInit(tfr16_tWaveGenRDITLQ * pSWG,
                                Frac16 * pSineTable,
                                UInt16 SineTableLength,
                                Frac16 InitialPhasePIx);
Frac16 tfr16WaveGenRDITLQ(tfr16_tWaveGenRDITLQ * pSWG,
                                Frac16 PhaseIncrement);
```

**Arguments:**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pSWG</code></td>
<td>Pointer to <code>tfr16_WaveGenRDITLQ</code> type defined in <code>tfr16.h</code> header file. It points to private data for the RDITLQ wave generation function.</td>
</tr>
<tr>
<td><code>pSineTable</code></td>
<td>Pointer to a sine look-up table which is allocated and filled by the application.</td>
</tr>
<tr>
<td><code>SineTableLength</code></td>
<td>The length of the sine look-up table</td>
</tr>
<tr>
<td><code>InitialPhasePIx</code></td>
<td>The starting phase of the sine wave to be generated</td>
</tr>
<tr>
<td><code>PhaseIncrement</code></td>
<td>The value to increment/decrement the phase by -1 period to 1 period</td>
</tr>
</tbody>
</table>

**Description:** This method of wave generation uses “Real Delta Loop-up with Interpolation”. This technique of wave generation is done by stepping through a sine table. The sine table has a one-quarter of a full cycle of sine values, stored at a constant rate.

This method calculates an index into a sine look-up table and retrieves a value. Successive values are obtained by adding a `PhaseIncrement` to the previous index.

This technique of sine wave generation allows the `PhaseIncrement` and the index into the sine table to be fractional. A continuous range of frequencies can be synthesized because a sine value is generated via linear interpolation done between the two table entries where the index falls.

This method produces one sample per call; successive calls are needed to generate a sine-wave.

**tfr16WaveGenRDITLQCreate:**

The `tfr16WaveGenRDITLQCreate` function allocates the structure of type `tfr16_tWaveGenRDITLQ`.

*For More Information On This Product, Go to: www.freescale.com*
The function `tfr16WaveGenRDTLQCreate` returns a pointer to the `tfr16_tWaveGenRDTLQ` structure, which is used by all other `tfr16WaveGenRDTLQ` functions.

In general, the data structure `tfr16_tWaveGenRDTLQ` allocated by `tfr16WaveGenRDTLQCreate` is treated as private data to better encapsulate the details of the implementation and to provide portability.

**tfr16WaveGenRDTLQInit:**

The `tfr16WaveGenRDTLQInit` function initializes the `tfr16_tWaveGenRDTLQ` data structure used by all `tfr16WaveGenRDTLQ` functions. The `tfr16_tWaveGenRDTLQ` data structure pointed to by `pSWG` must have been allocated prior to calling `tfr16WaveGenRDTLQInit`.

The primary reason to use `tfr16WaveGenRDTLQInit` directly is to statically allocate the `tfr16_tWaveGenRDTLQ` data structure, rather than allocating it dynamically with `tfr16WaveGenRDTLQCreate`. The data structure `tfr16_tWaveGenRDTLQ`, utilized in the `tfr16.h` interface, has been declared as private data to better encapsulate the details of the implementation and to provide portability. However, to utilize the `tfr16WaveGenRDTLQInit` routine directly, the application must allocate storage space for the `tfr16_tWaveGenRDTLQ` data structure before calling this function.

**tfr16WaveGenRDTLQ:**

The `tfr16WaveGenRDTLQ` function generates one sample and returns it.

**tfr16WaveGenRDTLQDestroy:**

The `tfr16WaveGenRDTLQDestroy` function frees the `tfr16_tWaveGenRDTLQ` data structures(s) previously allocated by `tfr16WaveGenRDTLQCreate`.

**Equation:**

\[ \sin(\pi \times x) = y \] 

**Eqn. 7-13**

**Returns:** The `tfr16WaveGenRDTLQ` function returns one sample.

**Algorithm:**

1. Index = integer_part_of(Index)
2. Index2 = (Index + 1) mod SineTableLength
3. Delta = fractional_part_of(Index)
4. SineDelta = SineTable[Index] - SineTable[Index2]
5. SineValue = SineTable[Index] + Delta * SineDelta
6. Index = (Index + Delta) mod SineTableLength

**Range Issues:** The range of `InitialPhasePIx` is \(-1 \leq \text{InitialPhasePIx} < 1\) and, therefore, the input will be within the range of \(-\pi \leq \pi(\text{InitialPhasePIx}) < \pi\). The size of the sine look-up table must be on the order of \(\text{Size} = (2^n) + 1 \leq 8192\) and contain only a quarter of a full sine table. Due to interpolation, comparing between K and K+1 entries in the table, the last two values in the table must be equivalent. For example, if
the size of your table is 257, the 257th value should equal the 256th. When passing in the size of the sine
look-up table in the \texttt{tfr16WaveGenRDITLQCreate} function, pass in \texttt{Size - 1}, as illustrated in \textbf{Code
Example 7-14}. The \texttt{PhaseIncrement} input parameter has a valid range from $-\pi \leq \texttt{PhaseIncrement} \leq \pi$.

\textbf{Special Issues:} The advantage of this implementation is the ability to synthesize a continuous range of
frequencies. There is also less distortion by using this method than in using \texttt{SineWaveGenRDITL}.
Compared to \texttt{SineWaveGenRDITL}, less memory is used by using only a quarter of a full sine wave cycle in
the table. Phase incrementation is done on the fly. One disadvantage of this implementation is increased
processing time compared to \texttt{SineWaveGenRDTL} and \texttt{SineWaveGenRDITL}. Another disadvantage is that
\texttt{tfr16WaveGenRDITLQ} returns only one sample per call.

\textbf{Design/Implementation:} The \texttt{tfr16WaveGenRDITL} functions are implemented as library function calls.

\textbf{Code Example 7-14. WaveGenRDITLQ}

\begin{verbatim}
#include "tfr16.h"

#define LENGTH 257

extern Frac16 SineTable[LENGTH];

{ tfr16_tWaveGenRDITLQ * pHandle;
  UInt16 SineTableLength = LENGTH - 1;
  Frac16 InitialPhasePIx = 16384; /*.5*\pi radians */
  Frac16 PhaseIncrement = 1024;
  Frac16 Value;

  ...
  pHandle = tfr16WaveGenRDITLQCreate( &SineTable[0], SineTableLength,
                                        InitialPhasePIx);

  for(I = 0; I < 100; I++)
  { Value = tfr16WaveGenRDITLQ(pHandle, PhaseIncrement);
    printf("%d\n", Value);
  }
  tfr16SineWaveGenRDITLQDestroy(pHandle);

  ...
}
\end{verbatim}
Trigonometric Math Library

7.3.14 SinPlxLUT

Call(s):

tfr16_tSinPlxLUT * tfr16SinPlxLUTCreate(Frac16 * pSineTable,
  UInt16 SineTableLength);

void tfr16SinPlxLUTDestroy(tfr16_tSinPlxLUT * pSWG);

void tfr16SinPlxLUTInit(tfr16_tSinPlxLUT * pSWG,
  Frac16 * pSineTable,
  UInt16 SineTableLength);

Frac16 tfr16SinPlxLUT(tfr16_tSinPlxLUT * pSWG,
  Frac16 PhasePIx);

Arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pSWG</td>
<td>in Frac16</td>
<td>Pointer to tfr16_tSinPlxLUT type defined in tfr16.h header file. It points to private data for the SinPlxLUT function.</td>
</tr>
<tr>
<td>pSineTable</td>
<td>in Pointer to a sine look-up table, which is allocated and filled by the application.</td>
<td></td>
</tr>
<tr>
<td>SineTableLength</td>
<td>in The length of the sine look-up table</td>
<td></td>
</tr>
<tr>
<td>PhasePIx</td>
<td>in The phase, $-\pi$ to $\pi$, from which the sine value is calculated</td>
<td></td>
</tr>
</tbody>
</table>

Description: This function returns a sine value for a specific phase. This technique of generating a sine value for a specific phase is done by stepping through a sine table. The sine table has a one quarter of a full cycle of sine values, stored at a constant rate. To get a sine value, linear interpolation is done between the two table entries between which the index into the table falls.

**tfr16SinPlxLUTCreate:**

The tfr16SinPlxLUTCreate function allocates the structure of type tfr16_tSinPlxLUT.

The function tfr16SinPlxLUTCreate returns a pointer to the tfr16_tSinPlxLUT structure which is used by all other tfr16SinPlxLUT functions.

In general, the data structure tfr16_tSinPlxLUT allocated by tfr16SinPlxLUTCreate is treated as private data to better encapsulate the details of the implementation and to provide portability.

**tfr16SinPlxLUTInit:**

The tfr16SinPlxLUTInit function initializes the tfr16_tSinPlxLUT data structure used by all tfr16SinPlxLUT functions. The tfr16_tSinPlxLUT data structure pointed to by pSWG must have been allocated prior to calling tfr16SinPlxLUTInit.

The primary reason to use tfr16SinPlxLUTInit directly is to statically allocate the tfr16_tSinPlxLUT data structure, rather than allocating it dynamically with tfr16SinPlxLUTCreate. The data structure tfr16_tSinPlxLUT, utilized in the tfr16.h
Fractional Trigonometric Function Specs

interface, has been declared as private data to better encapsulate the details of the implementation and to provide portability. However, to utilize the tfr16SinPIxLUTInit routine directly, the application must allocate storage space for the tfr16_tSinPIxLUT data structure before calling this function.

**tfr16SinPIxLUT:**

The tfr16SinPIxLUT function generates one sine value and returns it.

**tfr16SinPIxLUTDestroy:**

The tfr16SinPIxLUTDestroy function frees the tfr16_tSinPIxLUT data structures(s) previously allocated by tfr16SinPIxLUTCreate.

**Equation:**

\[ \sin(\pi \times x) = y \]  

*Eqn. 7-14*

**Returns:** The function tfr16SinPIxLUT returns one sine value.

**Algorithm:**

Index = integer_part_of(Index)  
Index2 = (Index + 1) mod SineTableLength  
Delta = fractional_part_of(Index)  
SineDelta = SineTable[Index] - SineTable[Index2]  
SineValue = SineTable[Index] + Delta * SineDelta  
Index = (Index + Delta) mod SineTableLength

**Range Issues:** The range of PhasePIx is \(-1 \leq \text{InitialPhasePIx} < 1\) and, therefore, the input will be within the range of \(-\pi \leq (\text{PhasePIx}) \leq \pi\). The size of the sine look-up table must be on the order of \(\text{Size} = (2^n) + 1\) \(\leq 8192\) and contain only a quarter of a full sine table. Due to interpolation, comparing between \(K\) and \(K+1\) entries in the table, the last two values in the table must be equivalent. For example, if the size of your table is 257, the 257th value should equal the 256th. When passing in the size of the sine look-up table in the tfr16SinPIxLUTCreate function, pass in \(\text{Size} - 1\); this is illustrated in Code Example 7-15.

**Special Issues:** None

**Design/Implementation:** The tfr16SinPIxLUT functions are implemented as library function calls.

---

**Code Example 7-15. SinPIxLUT**

```c
#include "tfr16.h"

#define LENGTH 257

extern Frac16 SineTable[LENGTH];

{  
tfr16_tSinPIxLUT * pHandle;
  UInt16 SineTableLength = LENGTH - 1;
```

MOTOROLA Trigonometric Math Library For More Information On This Product, Go to: www.freescale.com 7-35
Frac16 PhasePIx = 16384; /*.5*π radians */
Frac16 Value;

... pHandle = tfr16SinPIxLUTCreate(&SineTable[0],
    SineTableLength,
    InitialPhasePIx);

for(I = 0; I < 100; I++)
{
    Value = tfr16SinPIxLUT(pHandle, PhasePIx);
    printf("%d\n", Value);
}

tfr16SinPIxLUTDestroy(pHandle);
...
7.3.15 CosPIxLUT

Call(s):

```c
    tfr16_tCosPIxLUT * tfr16CosPIxLUTCreate(Frac16 * pSineTable,
                                           UInt16   SineTableLength);

    void tfr16CosPIxLUTDestroy(tfr16_tCosPIxLUT * pSWG);

    void tfr16CosPIxLUTInit(tfr16_tCosPIxLUT * pSWG,
                             Frac16   * pSineTable,
                             UInt16   SineTableLength);

    Frac16 tfr16CosPIxLUT(tfr16_tCosPIxLUT * pSWG,
                          Frac16   PhasePIx);
```

Arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pSWG</td>
<td>Pointer to <code>tfr16_tCosPIxLUT</code> type defined in <code>tfr16.h</code> header file. It points to private data for the CosPIxLUT function.</td>
</tr>
<tr>
<td>pSineTable</td>
<td>Pointer to a sine look-up table, which is allocated and filled by the application.</td>
</tr>
<tr>
<td>SineTableLength</td>
<td>The length of the sine look-up table</td>
</tr>
<tr>
<td>PhasePIx</td>
<td>The phase, $-\pi$ to $\pi$, from which the cosine value is calculated.</td>
</tr>
</tbody>
</table>

Description: This function returns a cosine value for a specific phase. This technique of generating a cosine value for a specific phase is done by stepping through a sine table. The sine table has a one-quarter of a full cycle of sine values, stored at a constant rate. A cosine value is generated via linear interpolation done between the two table entries between which the index falls.

`tfr16CosPIxLUTCreate`:

The `tfr16CosPIxLUTCreate` function allocates the structure of type `tfr16_tCosPIxLUT`.

The function `tfr16CosPIxLUTCreate` returns a pointer to the `tfr16_tCosPIxLUT` structure, which is used by all other `tfr16CosPIxLUT` functions.

In general, the data structure `tfr16_tCosPIxLUT` allocated by `tfr16CosPIxLUTCreate` is treated as private data to better encapsulate the details of the implementation and to provide portability.

`tfr16CosPIxLUTInit`:

The `tfr16CosPIxLUTInit` function initializes the `tfr16_tCosPIxLUT` data structure used by all `tfr16CosPIxLUT` functions. The `tfr16_tCosPIxLUT` data structure pointed to by `pSWG` must have been allocated prior to calling `tfr16CosPIxLUTInit`.

The primary reason to use `tfr16CosPIxLUTInit` directly is to statically allocate the `tfr16_tCosPIxLUT` data structure, rather than allocating it dynamically with `tfr16CosPIxLUTCreate`. The data structure `tfr16_tCosPIxLUT`, utilized in the `tfr16.h`
interface, has been declared as private data to better encapsulate the details of the implementation and to provide portability. However, to utilize the `tfr16CosPIxLUTInit` routine directly, the application must allocate storage space for the `tfr16_tCosPIxLUT` data structure before calling this function.

**tfr16CosPIxLUT:**
The `tfr16CosPIxLUT` function generates one cosine value and returns it.

**tfr16CosPIxLUTDestroy:**
The `tfr16CosPIxLUTDestroy` function frees the `tfr16_tCosPIxLUT` data structures(s) previously allocated by `tfr16CosPIxLUTCreate`.

**Equation:**
\[ \cos(\pi \times x) = y \]  
*Eqn. 7-15*

**Returns:** The function `tfr16CosPIxLUT` returns one cosine value.

**Algorithm:**

1. Index = integer_part_of(Index)
2. Index2 = (Index + 1) mod SineTableLength
3. Delta = fractional_part_of(Index)
4. SineDelta = SineTable[Index] - SineTable[Index2]
5. SineValue = SineTable[Index] + Delta * SineDelta
6. Index = (Index + Delta) mod SineTableLength

**Range Issues:** The range of `PhasePIx` is \(-1 \leq \text{InitialPhasePIx} < 1\) and, therefore, the input will be within the range of \(-\pi \leq (\text{PhasePIx}) \leq \pi\). The size of the sine look-up table must be on the order of \(\text{Size} = (2^n) + 1 \leq 8192\) and contain only a quarter of a full sine table. Due to interpolation, comparing between \(K\) and \(K+1\) entries in the table, the last two values in the table must be equivalent. For example, if the size of your table is 257, the 257th value should equal the 256th. When passing in the size of the sine look-up table in the `tfr16CosPIxLUTCreate` function, pass in `Size - 1`, as illustrated in Code Example 7-16.

**Special Issues:** None

**Design/Implementation:** The `tfr16CosPIxLUT` functions are implemented as library function calls.

---

**Code Example 7-16. CosPIxLUT**

```c
#include "tfr16.h"
#define LENGTH 257
extern Frac16 SineTable[LENGTH];
{
    tfr16_tCosPIxLUT * pHandle;
    Uint16 SineTableLength = LENGTH - 1;
}
```
Frac16 PhasePIx = 16384; /* .5*π radians */
Frac16 Value;

...

pHandle = tfr16CosPIxLUTCreate(&SineTable[0],
    SineTableLength,
    InitialPhasePIx);

for(I = 0; I < 100; I++)
{
    Value = tfr16CosPIxLUT(pHandle, PhasePIx);
    printf("%d\n", Value);
}

tfr16CosPIxLUTDestroy(pHandle);
...

Chapter 8
Array Library

This section describes functions for single-dimensioned arrays of 16-bit and 32-bit fractional values.

8.1 Introduction

The Digital Signal Processing Function Library implements many functions for single-dimensioned arrays of fractional types as library routines. Most basic math primitives, such as add, multiply, and subtract, have corresponding array operations. The array operations provide highly optimized loops on the elements of each array to provide the best performance.

The function library provides such array functions for both 16-bit (short) and 32-bit (long) fractional types. For portability, we have defined a short 16-bit fractional type as Frac16 and a long 32-bit fractional type as Frac32.

Section 8.2 presents the C interfaces for 16-bit fractional single-dimensioned array functions, afr16.h, and the C interfaces for the 32-bit fractional array functions, afr32.h, are presented in Section 8.3.

8.2 16-bit Fractional Interface

The C interface for the 16-bit fractional array functions is defined in the C header file afr16.h, shown in Code Example 8-1.

Code Example 8-1. C Header File afr16.h

```c
/* File: afr16.h */

#ifndef __AFR16_H
#define __AFR16_H

#include "port.h"

#ifdef __cplusplus
extern "C" {
#endif

/******************************************************
* Single dimension array operations - 16 bit fractional
*******************************************************/

EXPORT void afr16Abs (Frac16 *pX, Frac16 *pZ, UInt16 n);

#ifdef __cplusplus
}
#endif
#endif
```

For More Information On This Product, Go to: www.freescale.com
EXPORT void afr16Add        (Frac16 *pX, Frac16 *pY, Frac16 *pZ, UInt16 n);
EXPORT void afr16Div        (Frac16 *pX, Frac16 *pY, Frac16 *pZ, UInt16 n);
EXPORT bool  afr16Equal     (Frac16 *pX, Frac16 *pY, UInt16 n);
EXPORT void  afr16Mac_r     (Frac16 *pW, Frac16 *pX, Frac16 *pY, Frac16 *pZ, UInt16 n);
EXPORT Frac16 afr16Max      (Frac16 *pX, UInt16 n, UInt16 *pMaxIndex);
EXPORT Frac16 afr16Min      (Frac16 *pX, UInt16 n, UInt16 *pMinIndex);
EXPORT void  afr16Msu_r     (Frac16 *pW, Frac16 *pX, Frac16 *pY, Frac16 *pZ, UInt16 n);
EXPORT void  afr16Mult      (Frac16 *pX, Frac16 *pY, Frac16 *pZ, UInt16 n);
EXPORT void  afr16Mult_r    (Frac16 *pX, Frac16 *pY, Frac16 *pZ, UInt16 n);
EXPORT void  afr16Negate    (Frac16 *pX, Frac16 *pZ, UInt16 n);
EXPORT void  afr16Rand      (Frac16 *pZ, UInt16 n);
EXPORT void  afr16Sqrt      (Frac16 *pX, Frac16 *pZ, UInt16 n);
EXPORT void  afr16Sub       (Frac16 *pX, Frac16 *pY, Frac16 *pZ, UInt16 n);

#ifdef __cplusplus
}
#endif
#endif

8.3 32-bit Fractional Interface

The C interface for the 32-bit fractional array functions is defined in the C header file afr32.h, shown in Code Example 8-2.

Code Example 8-2. C Header File afr32.h

/* File: afr32.h */

#ifndef __AFR32_H
#define __AFR32_H
#include "port.h"
#ifdef __cplusplus
extern "C" {
#endif

/* Single dimension array operations - 16 bit fractional
*******************************************************************************
EXPORT void afr32Abs        (Frac32 *pX, Frac32 *pZ, UInt16 n);
EXPORT void afr32Add        (Frac32 *pX, Frac32 *pY, Frac32 *pZ, UInt16 n);
EXPORT void afr32Div        (Frac32 *pX, Frac32 *pY, Frac32 *pZ, UInt16 n);
EXPORT bool  afr32Equal     (Frac32 *pX, Frac32 *pY, UInt16 n);
EXPORT void  afr32Mac_r     (Frac32 *pW, Frac32 *pX, Frac32 *pY, Frac32 *pZ, UInt16 n);
EXPORT Frac32 afr32Max      (Frac32 *pX, UInt16 n, UInt16 *pMaxIndex);
EXPORT Frac32 afr32Min      (Frac32 *pX, UInt16 n, UInt16 *pMinIndex);
EXPORT void  afr32Msu_r     (Frac32 *pW, Frac32 *pX, Frac32 *pY, Frac32 *pZ, UInt16 n);
EXPORT void  afr32Mult      (Frac32 *pX, Frac32 *pY, Frac32 *pZ, UInt16 n);
EXPORT void  afr32Mult_r    (Frac32 *pX, Frac32 *pY, Frac32 *pZ, UInt16 n);
EXPORT void  afr32Negate    (Frac32 *pX, Frac32 *pZ, UInt16 n);
EXPORT void  afr32Rand      (Frac32 *pZ, UInt16 n);
EXPORT void  afr32Sqrt      (Frac32 *pX, Frac32 *pZ, UInt16 n);
EXPORT void  afr32Sub       (Frac32 *pX, Frac32 *pY, Frac32 *pZ, UInt16 n);

#endif
}}
#endif
EXPORT void afr32Mac (Frac32 *pW, Frac16 *pX, Frac16 *pY, Frac32 *pZ, UInt16 n);
EXPORT void afr32Mac_r (Frac32 *pW, Frac16 *pX, Frac16 *pY, Frac32 *pZ, UInt16 n);

EXPORT Frac32 afr32Max (Frac32 *pX, UInt16 n, UInt16 *pMaxIndex);
EXPORT Frac32 afr32Min (Frac32 *pX, UInt16 n, UInt16 *pMinIndex);

EXPORT void afr32Msu (Frac32 *pW, Frac16 *pX, Frac16 *pY, Frac32 *pZ, UInt16 n);
EXPORT void afr32Msu_r (Frac32 *pW, Frac16 *pX, Frac16 *pY, Frac16 *pZ, UInt16 n);

EXPORT void afr32Mult (Frac16 *pX, Frac16 *pY, Frac32 *pZ, UInt16 n);
EXPORT void afr32Mult_ls (Frac32 *pX, Frac16 *pY, Frac32 *pZ, UInt16 n);

EXPORT void afr32Negate (Frac32 *pX, Frac32 *pZ, UInt16 n);
EXPORT void afr32Round (Frac32 *pX, Frac16 *pZ, UInt16 n);
EXPORT void afr32Sqrt (Frac32 *pX, Frac16 *pZ, UInt16 n);
EXPORT void afr32Sub (Frac32 *pX, Frac32 *pY, Frac32 *pZ, UInt16 n);

#ifdef __cplusplus
}
#endif
#endif
8.4 Array Function Specifications

The following pages describe the single-dimensioned array functions for both 16-bit and 32-bit fractional types.

Each page utilizes a short name to generically describe its function. In many cases, however, the actual function name that must be used in C or assembler may be different from the generic function name. Therefore, refer to the C function declaration to find the specific function name to put in your program. For example, the generic function name for an absolute value function may be abs, but the actual function name that must be used for a 32-bit fractional array is afr32Abs.

Function arguments for each routine are described as in, out, or inout. An in argument means that the parameter value is an input only to the function. An out argument means that the parameter value is an output only from the function. An inout argument means that a parameter value is an input to the function, but the same parameter is also an output from the function.

Typically, inout parameters are input pointer variables in which the caller passes the address of a preallocated data structure to a function. The function stores its results within that data structure. The actual value of the inout pointer parameter is not changed.
8.4.1 abs

Call(s):

void afr16Abs (Frac16 *pX, Frac16 *pZ, UInt16 n);
void afr32Abs (Frac32 *pX, Frac32 *pZ, UInt16 n);

Arguments:

<table>
<thead>
<tr>
<th>pX</th>
<th>in</th>
<th>Pointer to an input array of fractional data values</th>
</tr>
</thead>
<tbody>
<tr>
<td>pZ</td>
<td>inout</td>
<td>Pointer to an output array of fractional data values</td>
</tr>
<tr>
<td>n</td>
<td>in</td>
<td>The length of the input and output arrays</td>
</tr>
</tbody>
</table>

Description: The function abs calculates the absolute value of each fractional element of the input array pointed to by pX; i.e., pZ[i] = pX[i], 0 <= i < n.

Equation:

\[ |x[1...n]| = z[1...n] \quad \text{Eqn. 8-1} \]

Returns: The function abs stores the results of the absolute value calculation in the array pointed to by pZ.

Range Issues: If saturation is enabled, abs will store the maximum fractional value if overflow occurs during the calculation of any array element. It is not possible to have an underflow with the abs function.

The length of the input and output arrays, n, must be in the range 0 <= n < MAX VECTOR LEN.

Special Issues: In place computation is allowed; i.e., pX can be equal to pZ.

Design/Implementation: The abs array function is implemented as a library function.

Code Example 8-3. abs

```c
{
    Frac16 x16[10];
    Frac16 z16[10];
    Frac32 x32[10];
    Frac32 z32[10];
    ...
    afr16Abs (x16, z16, sizeof(x16)/sizeof(x16[0]));
    afr32Abs (x32, z32, sizeof(x32)/sizeof(x32[0]));
    ...
}
```
8.4.2 *add*

Call(s):

- void afr16Add (Frac16 *pX, Frac16 *pY, Frac16 *pZ, UInt16 n);
- void afr32Add (Frac32 *pX, Frac32 *pY, Frac32 *pZ, UInt16 n);

Arguments:

<table>
<thead>
<tr>
<th>pX</th>
<th>in</th>
<th>Pointer to the first input array of fractional data values</th>
</tr>
</thead>
<tbody>
<tr>
<td>pY</td>
<td>in</td>
<td>Pointer to the second input array of fractional data values</td>
</tr>
<tr>
<td>pZ</td>
<td>inout</td>
<td>Pointer to the output array of fractional results</td>
</tr>
<tr>
<td>n</td>
<td>in</td>
<td>The length of all input and output arrays</td>
</tr>
</tbody>
</table>

Description: The function *add* totals the corresponding elements of each input array; i.e.,

\[ pZ[i] = pX[i] + pY[i], \quad 0 \leq i < n. \]

Equation: \[ x[1...n] + y[1...n] = z[1...n] \] \hspace{1cm} Eqn. 8-2

Returns: The function *add* stores the results of the addition calculation in the array pointed to by *pZ*, where \[ pZ[i] = pX[i] + pY[i], \quad 0 \leq i < n. \]

Range Issues: If saturation is enabled, the array function *add* will return the maximum or minimum fractional values if overflow or underflow occurs during the calculation of each output element.

The length of the input and output arrays, *n*, must be in the range \( 0 \leq n < \text{MAX\_VECTOR\_LEN} \).

Special Issues: In place computation is allowed; i.e., *pX* and/or *pY* can be equal to *pZ*.

Design/Implementation: The array function *add* is implemented as a library function.

Code Example 8-4. *add*

```c
{ 
    Frac16 x16[10];
    Frac16 y16[10];
    Frac16 z16[10];
    Frac32 x32[10];
    Frac32 y32[10];
    Frac32 z32[10];
    ...
    afr16Add (x16, y16, z16, sizeof(x16)/sizeof(x16[0]));
    afr32Add (x32, y32, z32, sizeof(x32)/sizeof(x32[0]));
    ...
}
```
8.4.3 \textit{div}

Call(s):

\begin{verbatim}
void afr16Div (Frac16 *pX, Frac16 *pY, Frac16 *pZ, UInt16 n);
void afr32Div (Frac32 *pX, Frac16 *pY, Frac16 *pZ, UInt16 n);
\end{verbatim}

Arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(pX)</td>
<td>Pointer to the input array of dividends</td>
</tr>
<tr>
<td>(pY)</td>
<td>Pointer to the input array of divisors</td>
</tr>
<tr>
<td>(pZ)</td>
<td>Pointer to the output array of quotients</td>
</tr>
<tr>
<td>(n)</td>
<td>The length of all input and output arrays</td>
</tr>
</tbody>
</table>

Description: The function \textit{div} divides the corresponding elements of each input array; i.e.,
\(pZ[i] = pX[i] / pY[i], 0 \leq i < n\). In this equation, \(pX[i]\) and \(pY[i]\) must be positive, and \(pY[i]\) must be greater than or equal to \(pX[i]\).

Equation:

\[
y[1...n] \quad x[1...n] = z[1...n] 
\]

Returns: The function \textit{div} stores the quotients resulting from the division calculations in the array pointed to by \(pZ\), where \(pZ[i] = pX[i] / pY[i], 0 \leq i < n\).

Range Issues: The division calculation relies on the assumption that the quotient is within the valid fractional result range, \(-1 \leq pZ[i] < 1\). If this assumption is violated, the result of the calculation is undefined and the resulting quotient may be erroneous.

If saturation is enabled, and if \(pX[i]\) is equal to \(pY[i]\), the maximum fractional value is returned for that quotient.

The length of the input and output arrays, \(n\), must be in the range \(0 \leq n < \text{MAX\_VECTOR\_LEN}\).

Special Issues: In place computation is allowed; i.e., \(pX\) and/or \(pY\) can be equal to \(pZ\).

Design/Implementation: The \textit{div} array function is implemented as a library function.
Code Example 8-5.  \texttt{div}

\begin{verbatim}
{
  Frac16 x16[10];
  Frac16 y16[10];
  Frac16 z16[10];

  Frac32 x32[10];

  ...

  afr16Div (x16, y16, z16, sizeof(x16)/sizeof(x16[0]));
  afr32Div (x32, y16, z16, sizeof(x32)/sizeof(x32[0]));

  ...
}
\end{verbatim}
8.4.4 equal

Call(s):

```c
bool afr16Equal (Frac16 *pX, Frac16 *pY, UInt16 n);
bool afr32Equal (Frac32 *pX, Frac32 *pY, UInt16 n);
```

Arguments:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pX</td>
<td>pointer</td>
<td>Pointer to an input array of fractional data values</td>
</tr>
<tr>
<td>pY</td>
<td>pointer</td>
<td>Pointer to an input array of fractional data values</td>
</tr>
<tr>
<td>n</td>
<td>integer</td>
<td>The length of both input arrays</td>
</tr>
</tbody>
</table>

Description: The function equal compares the corresponding elements of each fractional array for equality, i.e., \( pX[i] == pY[i], 0 \leq i < n \).

Equation: None

Returns: The function equal returns true if all corresponding elements of the two input arrays are equal. If any element of the two input arrays is not equal, equal returns false.

Range Issues: The length of the input and output arrays, n, must be in the range \( 0 \leq n < MAX\_VECTOR\_LEN \).

Special Issues: None

Design/Implementation: The equal array function is implemented as a library function.

Code Example 8-6. equal

```c
{
    Frac16 x16[10];
    Frac16 y16[10];
    Frac32 x32[10];
    Frac32 y32[10];
    bool bIsEqual;
    ...
    bIsEqual = afr16Equal (x16, y16, sizeof(x16)/sizeof(x16[0]));
    bIsEqual = afr32Equal (x32, y32, sizeof(x32)/sizeof(x32[0]));
    ...
}
```
8.4.5 \textit{mac}

\textbf{Call(s):}

\begin{verbatim}
void afr16Mac_r (Frac16 *pW, Frac16 *pX, Frac16 *pY, Frac16 *pZ, UInt16 n);
void afr32Mac    (Frac32 *pW, Frac16 *pX, Frac16 *pY, Frac32 *pZ, UInt16 n);
void afr32Mac_r  (Frac32 *pW, Frac16 *pX, Frac16 *pY, Frac32 *pZ, UInt16 n);
\end{verbatim}

\textbf{Arguments:}

\begin{table}[h]
\centering
\begin{tabular}{|l|l|}
\hline
\textbf{\textit{pW}} & in & Pointer to the input array of fractional data values to be added \\
\textbf{\textit{pX}} & in & Pointer to the input array of fractional data values to be multiplied \\
\textbf{\textit{pY}} & in & Pointer to the input array of fractional data values to be multiplied \\
\textbf{\textit{pZ}} & inout & Pointer to the output array \\
\textbf{\textit{n}} & in & The length of all input and output arrays \\
\hline
\end{tabular}
\caption{\textit{mac} (array) arguments}
\end{table}

\textbf{Description:} The function \textit{mac} multiplies the corresponding elements of each X and Y input array and adds the result to the corresponding element of the W input array; for example:

\[ pZ[i] = (pX[i] \times pY[i]) + pW[i], \ 0 \leq i < n \]

\textbf{Equation:}

\[ (x[1\ldots n] \times y[1\ldots n]) + w[1\ldots n] = z[1\ldots n] \quad \text{Eqn. 8-4} \]

\textbf{Returns:} The function \textit{mac} stores the results of the calculation in the array pointed to by \textit{pZ}, where \( pZ[i] = (pX[i] \times pY[i]) + pW[i], \ 0 \leq i < n \).

\textbf{Range Issues:} If saturation is enabled, the array function \textit{mac} will return the maximum or minimum fractional values if overflow or underflow occurs during the calculation of each output element.

The length of the input and output arrays, \textit{n}, must be in the range \( 0 \leq n < \text{MAX\_VECTOR\_LEN} \).

\textbf{Special Issues:} For the array functions \textit{afr16Mac\_r} and \textit{afr32Mac\_r}, in place computation is allowed; i.e., \textit{pX}, \textit{pY}, and \textit{pW} can be equal to \textit{pZ}.

For the array function \textit{afr32Mac}, in place computation is allowed only between \textit{pW} and \textit{pZ}; \textit{pX} and \textit{pY} must not be equal to \textit{pZ}.

\textbf{Design/Implementation:} The array function \textit{mac} is implemented as a library function.
Code Example 8-7.  \textit{mac}

\begin{verbatim}
\{
    Frac16 w16[10];
    Frac16 x16[10];
    Frac16 y16[10];
    Frac16 z16[10];

    Frac32 w32[10];
    Frac32 z32[10];

    afr16Mac_r (w16, x16, y16, z16, sizeof(x16)/sizeof(x16[0]));
    afr32Mac   (w32, x16, y16, z32, sizeof(x16)/sizeof(x16[0]));
    afr32Mac_r (w32, x16, y16, z16, sizeof(x16)/sizeof(x16[0]));
    ...
\}
\end{verbatim}
8.4.6 max

Call(s):

Frac16 afr16Max (Frac16 *pX, U16 n, U16 *pMaxIndex);
Frac32 afr32Max (Frac32 *pX, U16 n, U16 *pMaxIndex);

Arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pX</td>
<td>Pointer to the input array of fractional data values</td>
</tr>
<tr>
<td>n</td>
<td>The length of the array</td>
</tr>
<tr>
<td>pMaxIndex</td>
<td>A pointer to a variable in which to store the index of</td>
</tr>
<tr>
<td></td>
<td>the maximum value of the input array; may be NULL</td>
</tr>
</tbody>
</table>

Description: The function max searches the input array for the maximum fractional value.

Equation: None

Returns: The function max returns the value of the maximum array element as the return value of the function. In addition, the array index of the maximum array element may be stored in *pMaxIndex, in the range 0 <= index < n, if pMaxIndex is not NULL.

Range Issues: No overflow or underflow is possible with this function.

The length of the input array, n, must be in the range 0 <= n < MAX_VECTOR_LEN.

Special Issues: None

Design/Implementation: The array function max is implemented as a library function.

Code Example 8-8. max

```c
{
    Frac16 x16[10];
    Frac16 MaxValue16;
    U16 MaxIndex16;

    Frac32 x32[10];
    Frac32 MaxValue32;
    U16 MaxIndex32;

    ...
    MaxValue16 = afr16Max (x16, sizeof(x16)/sizeof(x16[0]), &MaxIndex16);
    MaxValue32 = afr32Max (x32, sizeof(x32)/sizeof(x32[0]), &MaxIndex32);
    ...
}
```
8.4.7 \textit{min} 

Call(s):

\begin{verbatim}
Frac16 afr16Min (Frac16 *pX, UInt16 n, UInt16 *pMinIndex);
Frac32 afr32Min (Frac32 *pX, UInt16 n, UInt16 *pMinIndex);
\end{verbatim}

Arguments:

<table>
<thead>
<tr>
<th>\textbf{Table 8-7.} \textit{min} (array) arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textbf{\textit{pX}}</td>
</tr>
<tr>
<td>\textbf{n}</td>
</tr>
<tr>
<td>\textbf{pMinIndex}</td>
</tr>
</tbody>
</table>

\textbf{Description}: The function \textit{min} searches the input array for the minimum fractional value.

\textbf{Equation}: None

\textbf{Returns}: The function \textit{min} returns the value of the minimum array element as the return value of the function. In addition, the array index of the minimum array element may be stored in \textit{*pMinIndex}, in the range \(0 \leq \text{index} < n\), if \textit{pMinIndex} is not NULL.

\textbf{Range Issues}: No overflow or underflow is possible with this function.

The length of the input array, \(n\), must be in the range \(0 \leq n < \text{MAX\_VECTOR\_LEN}\).

\textbf{Special Issues}: None

\textbf{Design/Implementation}: The array function \textit{min} is implemented as a library function.

\textbf{Code Example 8-9.} \textit{min}

\begin{verbatim}
{
    Frac16 x16[10];
    Frac16 MinValue16;
    UInt16 MinIndex16;

    Frac32 x32[10];
    Frac32 MinValue32;
    UInt16 MinIndex32;

    ...  

    MinValue16 = afr16Min (x16, sizeof(x16)/sizeof(x16[0]), &MinIndex16);
    MinValue32 = afr32Min (x32, sizeof(x32)/sizeof(x32[0]), &MinIndex32);
    ...
}
\end{verbatim}
**msu**

Call(s):

```c
void afr16Msu_r (Frac16 *pW, Frac16 *pX, Frac16 *pY, Frac16 *pZ, UInt16 n);
void afr32Msu   (Frac32 *pW, Frac16 *pX, Frac16 *pY, Frac32 *pZ, UInt16 n);
void afr32Msu_r (Frac32 *pW, Frac16 *pX, Frac16 *pY, Frac16 *pZ, UInt16 n);
```

Arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pW</code></td>
<td>Pointer to the input array of fractional data values against which <code>(pX[i] * pY[i])</code> will be subtracted</td>
</tr>
<tr>
<td><code>pX</code></td>
<td>Pointer to the input array of fractional data values to be multiplied</td>
</tr>
<tr>
<td><code>pY</code></td>
<td>Pointer to the input array of fractional data values to be multiplied</td>
</tr>
<tr>
<td><code>pZ</code></td>
<td>Pointer to the output array</td>
</tr>
<tr>
<td><code>n</code></td>
<td>The length of all input and output arrays</td>
</tr>
</tbody>
</table>

**Description:** The function `msu` multiplies the corresponding elements of each X and Y input array and subtracts the result from the corresponding element of the W input array; for example:

```
pZ[i] = pW[i] - (pX[i] * pY[i]), 0 <= i < n
```

**Equation:**

```
w[1...n] - (x[1...n] \times y[1...n]) = z[1...n] \quad Eqn. 8-5
```

**Returns:** The function `msu` stores the results of the calculation in the array pointed to by `pZ`, where `pZ[i] = pW[i] - (pX[i] * pY[i]), 0 <= i < n`.

**Range Issues:** If saturation is enabled, the array function `msu` will return the maximum or minimum fractional values if overflow or underflow occurs during the calculation of each output element.

The length of the input and output arrays, `n`, must be in the range `0 <= n < MAX_VECTOR_LEN`.

**Special Issues:** For the array function `afr16Msu_r` and `afr32Msu_r`, in place computation is allowed; i.e., `pX`, `pY`, and `pW` can be equal to `pZ`.

For the array function `afr32Msu`, in place computation is allowed only between `pW` and `pZ`; `pX` and `pY` must not be equal to `pZ`.

**Design/Implementation:** The array function `msu` is implemented as a library function.
Code Example 8-10.  *msu*

```c
{
    Frac16 w16[10];
    Frac16 x16[10];
    Frac16 y16[10];
    Frac16 z16[10];

    Frac32 w32[10];
    Frac32 z32[10];

    afr16Msu_r (w16, x16, y16, z16, sizeof(x16)/sizeof(x16[0]));
    afr32Msu (w32, x16, y16, z32, sizeof(x16)/sizeof(x16[0]));
    afr32Msu_r (w32, x16, y16, z16, sizeof(x16)/sizeof(x16[0]));
    ...
}
```
8.4.9 \textit{mult}

\textbf{Call(s):}

\begin{verbatim}
void afr16Mult (Frac16 *pX, Frac16 *pY, Frac16 *pZ, UInt16 n);
void afr16Mult_r (Frac16 *pX, Frac16 *pY, Frac16 *pZ, UInt16 n);
void afr32Mult (Frac16 *pX, Frac16 *pY, Frac32 *pZ, UInt16 n);
void afr32Mult_ls (Frac32 *pX, Frac16 *pY, Frac32 *pZ, UInt16 n);
\end{verbatim}

\textbf{Arguments:}

\begin{tabular}{|c|c|}
\hline
\textbf{pX} & \text{in} \hspace{1cm} \text{Pointer to the first input array} \\
\hline
\textbf{pY} & \text{in} \hspace{1cm} \text{Pointer to the second input array} \\
\hline
\textbf{pZ} & \text{inout} \hspace{1cm} \text{Pointer to the output array} \\
\hline
\textbf{n} & \text{in} \hspace{1cm} \text{The length of all input and output arrays} \\
\hline
\end{tabular}

\textbf{Description:} The array function \textit{mult} multiplies the corresponding elements of each input array; i.e., 
pZ[i] = pX[i] * pY[i], 0 \leq i < n. The \textit{mult} function differs from \textit{mult} in that \textit{mult}_r rounds the result, but \textit{mult} truncates the result.

\textbf{Equation:}

\begin{equation}
x[1...n] \times y[1...n] = z[1...n] \hspace{1cm} \text{Eqn. 8-6}
\end{equation}

\textbf{Returns:} The array function \textit{mult} returns the results of the multiplication in the array pointed to by \textit{pZ}, where \( pZ[i] = pX[i] \times pY[i], 0 \leq i < n \).

\textbf{Range Issues:} If saturation is enabled, the array function \textit{mult} will return the maximum or minimum fractional values if overflow or underflow occurs during the calculation of each output element.

The length of the input and output arrays, \textit{n}, must be in the range \( 0 \leq n < \text{MAX\_VECTOR\_LEN} \).

\textbf{Special Issues:} In place computation is allowed; i.e., \textit{pX} and/or \textit{pY} can be equal to \textit{pZ}.

\textbf{Design/Implementation:} The array function \textit{mult} is implemented as a library function.

\textbf{Code Example 8-11.} \textit{mult}

\begin{verbatim}
{
    Frac16 x16[10];
    Frac16 y16[10];
    Frac16 z16[10];
    Frac32 x32[10];
    Frac32 z32[10];

    ...
    afr16Mult   (x16, y16, z16, sizeof(x16)/sizeof(x16[0]));
    afr16Mult_r (x16, y16, z16, sizeof(x16)/sizeof(x16[0]));
    afr32Mult   (x16, y16, z32, sizeof(x16)/sizeof(x16[0]));
    afr32Mult_ls(x16, y16, z32, sizeof(x32)/sizeof(x32[0]));
    ...
}
\end{verbatim}
### 8.4.10 negate

**Call(s):**

```c
void afr16Negate (Frac16 *pX, Frac16 *pZ, UInt16 n);
void afr32Negate (Frac32 *pX, Frac32 *pZ, UInt16 n);
```

**Arguments:**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pX</code></td>
<td>Pointer to an input array of fractional data values</td>
</tr>
<tr>
<td><code>pZ</code></td>
<td>Pointer to an output array of fractional data values</td>
</tr>
<tr>
<td><code>n</code></td>
<td>The length of the input and output arrays</td>
</tr>
</tbody>
</table>

**Description:** The array function `negate` negates each fractional element of the input array pointed to by `pX`; for example: `pZ[i] = - pX[i], 0 <= i < n`

**Equation:**

\[ x[1...n] = -x[1...n] \]

**Eqn. 8-7**

**Returns:** The array function `negate` returns the results of the negations in the array pointed to by `pZ`.

**Range Issues:** If saturation is enabled, `negate` will store the maximum fractional value if overflow occurs during the calculation of any array element. It is not possible to have an underflow with the `negate` function.

The length of the input and output arrays, `n`, must be in the range `0 <= n < MAX_VECTOR_LEN`.

**Special Issues:** In place computation is allowed; i.e., `pX` can be equal to `pZ`.

**Design/Implementation:** The array function `negate` is implemented as a library function.

**Code Example 8-12. negate**

```c
{  
  Frac16 x16[10];
  Frac16 z16[10];
  
  Frac32 x32[10];
  Frac32 z32[10];
  
  ...
  afr16Negate (x16, z16, sizeof(x16)/sizeof(x16[0]));
  afr32Negate (x32, z32, sizeof(x32)/sizeof(x32[0]));
  ...
}  
```
8.4.11  Rand

Call(s):

```c
void afr16Rand (Frac16 *pZ, UInt16 n);
```

Arguments:

<table>
<thead>
<tr>
<th>pZ</th>
<th>inout</th>
<th>Pointer to an output array of fractional data values</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>in</td>
<td>The length of the output array</td>
</tr>
</tbody>
</table>

**Description:** The function `rand` calculates n random 16-bit fractional numbers and stores them in the output array pointed to by `pZ`; i.e., `pZ[i] = rand()`, `0 <= i < n`.

**Equation:** None

**Returns:** The array function `rand` stores n random numbers in the array pointed to by `pZ`.

**Range Issues:** The length of the output array, `n`, must be in the range `0 <= n < MAX_VECTOR_LEN`.

**Special Issues:** None

**Design/Implementation:** The array function `rand` is implemented as a library function.

**Code Example 8-13.  Rand**

```c
{  
  Frac16 z16[10];
  ...
  afr16Rand (z16, sizeof(x32)/sizeof(x32[0]));
  ...
}
```
### 8.4.12 `round`

**Call(s):**

```c
void afr32Round (Frac32 *pX, Frac16 *pZ, UInt16 n);
```

**Arguments:**

<table>
<thead>
<tr>
<th><code>pX</code></th>
<th>in</th>
<th>Pointer to an input array of fractional data values</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pZ</code></td>
<td>inout</td>
<td>Pointer to an output array of fractional data values</td>
</tr>
<tr>
<td><code>n</code></td>
<td>in</td>
<td>The length of the input and output arrays</td>
</tr>
</tbody>
</table>

**Description:** The array function `round` converts each 32-bit fractional element of the input array pointed to by `pX` into a 16-bit fractional value, rounding the least significant bit; for example:

\[
pZ[i] = \text{round}(pX[i]), \quad 0 \leq i < n
\]

**Equation:** None

**Returns:** The array function `round` stores the results of the conversion with rounding in the array pointed to by `pZ`.

**Range Issues:** If saturation is enabled, `round` will store the maximum or minimum fractional value if overflow or underflow occurs during the calculation of any array element.

The length of the input and output arrays, `n`, must be in the range \(0 \leq n < \text{MAX VECTOR LEN}\).

**Special Issues:** In place computation is allowed; i.e., `pX` may be equal to `pZ`.

**Design/Implementation:** The array function `round` is implemented as a library function.

**Code Example 8-14. `round`**

```c
{
    Frac16 z16[10];
    Frac32 x32[10];

    ...
    afr32Round (x32, z16, sizeof(x32)/sizeof(x32[0]));
    ...
}
```
8.4.13 \textit{sqrt}

Call(s):

\begin{verbatim}
void afr16Sqrt (Frac16 *pX, Frac16 *pZ, Uint16 n);
void afr32Sqrt (Frac32 *pX, Frac16 *pZ, Uint16 n);
\end{verbatim}

Arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( pX )</td>
<td>in Pointer to an input array of fractional data values</td>
</tr>
<tr>
<td>( pZ )</td>
<td>inout Pointer to an output array of fractional data values</td>
</tr>
<tr>
<td>( n )</td>
<td>in The length of the input and output arrays</td>
</tr>
</tbody>
</table>

\textbf{Description:} The array function \textit{sqrt} calculates the square root of each fractional element of the input array pointed to by \( pX \); i.e., \( pZ[i] = \sqrt{pX[i]} \), \( 0 \leq i < n \)

\textbf{Equation:}

\[
\sqrt{x[1...n]} = y[1...n]
\] \hspace{1cm} \textit{Eqn. 8-8}

\textbf{Returns:} The array function \textit{sqrt} stores the results of the square root calculation in the array pointed to by \( pZ \).

\textbf{Range Issues:} All elements of the input array must be greater or equal to zero; otherwise, the result of the \textit{sqrt} calculation is undefined for that array element.

The length of the input and output arrays, \( n \), must be in the range \( 0 \leq n < \text{MAX\_VECTOR\_LEN} \).

\textbf{Special Issues:} In place computation is allowed; i.e., \( pX \) may be equal to \( pZ \).

\textbf{Design/Implementation:} The array function \textit{sqrt} is implemented as a library function.

\textbf{Code Example 8-15.} \textit{sqrt}

\begin{verbatim}
{
  Frac16 x16[10];
  Frac32 x32[10];
  Frac16 z16[10];
  ...
  afr16Sqrt (x16, z16, sizeof(x16)/sizeof(x16[0]));
  afr32Sqrt (x32, z16, sizeof(x32)/sizeof(x32[0]));
  ...
}
\end{verbatim}
8.4.14 sub

Call(s):

```c
void afr16Sub (Frac16 *pX, Frac16 *pY, Frac16 *pZ, UInt16 n);
void afr32Sub (Frac32 *pX, Frac32 *pY, Frac32 *pZ, UInt16 n);
```

Arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pX</td>
<td>in Pointer to the first input array of fractional data values</td>
</tr>
<tr>
<td>pY</td>
<td>in Pointer to the second input array of fractional data values</td>
</tr>
<tr>
<td>pZ</td>
<td>inout Pointer to the output array of fractional results</td>
</tr>
<tr>
<td>n</td>
<td>in The length of all input and output arrays</td>
</tr>
</tbody>
</table>

Description: The array function `sub` subtracts the corresponding elements of the `pY` input array from the `pX` input array; i.e., `pZ[i] = pX[i] - pY[i]`, 0 <= i < n

Equation:

```
x[1...n] - y[1...n] = z[1...n]
```

Eqn. 8-9

Returns: The array function `sub` returns the results of the subtraction calculation in the array pointed to by `pZ`, where `pZ[i] = pX[i] - pY[i]`, 0 <= i < n.

Range Issues: If saturation is enabled, the array function `sub` will return the maximum or minimum fractional values if overflow or underflow occurs during the calculation of each output element.

The length of the input and output arrays, n, must be in the range 0 <= n < MAX_VECTOR_LEN.

Special Issues: In place computation is allowed; i.e., `pX` and/or `pY` can be equal to `pZ`.

Design/Implementation: The array function `sub` is implemented as a library function.

Code Example 8-16. sub

```c
{
    Frac16 x16[10];
    Frac16 y16[10];
    Frac16 z16[10];

    Frac32 x32[10];
    Frac32 y32[10];
    Frac32 z32[10];

    ...  
    afr16Sub (x16, y16, z16, sizeof(x16)/sizeof(x16[0]));
    afr32Sub (x32, y32, z32, sizeof(x32)/sizeof(x32[0]));
    ...
}
```
Chapter 9
Vector Library

This section describes functions for vectors of 16-bit fractional values.

9.1 Introduction

The Digital Signal Processing Function Library implements several mathematical functions for vectors of fractional types. These vector functions offer highly optimized code for each operation to provide the best performance.

The function library provides such vector functions for the 16-bit (short), fractional type. For portability, we have defined a short 16-bit fractional type as Frac16.

The vector functions include:

- addition
- dot product
- equality
- length
- multiplication
- scaling
- subtraction

Section 9.2 presents the C interfaces for 16-bit fractional vector functions, vfr16.h; Section 9.3 details specifications for these functions.

9.2 16-bit Fractional Interface

As shown in Code Example 9-1, the C interface for the 16-bit fractional vector functions is defined in the C header file vfr16.h.
#ifndef __VFR16_H
#define __VFR16_H

#include "port.h"

#ifdef __cplusplus
extern "C" {
#endif

/*******************************************************************************
 * Vector Math - 16 bit fractional
 *******************************************************************************/

EXPORT void    vfr16Add     (Frac16 *pX, Frac16 *pY, Frac16 *pZ, UInt16 n);
EXPORT Frac32  vfr16DotProd (Frac16 *pX, Frac16 *pY, UInt16 n);
EXPORT bool    vfr16Equal   (Frac16 *pX, Frac16 *pY, UInt16 n);
EXPORT Frac16  vfr16Length  (Frac16 *pX, UInt16 n);
EXPORT void    vfr16Mult    (Frac16 c, Frac16 *pX, Frac16 *pZ, UInt16 n);
EXPORT void    vfr16Scale   (Int16  k, Frac16 *pX, Frac16 *pZ, UInt16 n);
EXPORT void    vfr16Sub     (Frac16 *pX, Frac16 *pY, Frac16 *pZ, UInt16 n);

#ifdef __cplusplus
}
#endif
#endif
9.3 Vector Function Specifications

The following pages describe the vector functions for the 16-bit fractional type.

Each page utilizes a short name to generically describe its function. In many cases, however, the actual function name that must be used in C or assembler may be different from the generic function name. Therefore, please refer to the C function declaration to find the specific function name to put in your program. For example, the generic function name for an absolute value function may be `add`, but the actual function name that must be used for a 16-bit fractional vector is `vfr16Add`.

Function arguments for each routine are described as `in`, `out`, or `inout`. An `in` argument means that the parameter value is an input only to the function. An `out` argument means that the parameter value is an output only from the function. An `inout` argument means that a parameter value is an input to the function, but the same parameter is also an output from the function.

Typically, `inout` parameters are input pointer variables in which the caller passes the address of a preallocated data structure to a function. The function stores its results within that data structure. The actual value of the `inout` pointer parameter is not changed.
9.3.1 add

Call(s):

```c
void vfr16Add (Frac16 *pX, Frac16 *pY, Frac16 *pZ, UInt16 n);
```

Arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pX</code></td>
<td>in</td>
</tr>
<tr>
<td><code>pY</code></td>
<td>in</td>
</tr>
<tr>
<td><code>pZ</code></td>
<td>inout</td>
</tr>
<tr>
<td><code>n</code></td>
<td>in</td>
</tr>
</tbody>
</table>

Description: The vector function `add` totals two vectors; i.e., $Z = X + Y$. The actual implementation adds each element of the two input vectors, $pZ[i] = pX[i] + pY[i]$, $0 \leq i < n$.

Equation:

$$x[1...n] + y[1...n] = z[1...n] \quad \text{Eqn. 9-1}$$

Returns: The vector function `add` stores the resulting vector pointed to by `pZ`, where $pZ[i] = pX[i] + pY[i]$, $0 \leq i < n$.

Range Issues: If saturation is enabled, the vector function `add` will return the maximum or minimum fractional values if overflow or underflow occurs during the calculation of each output element.

The length of the input and output vectors, `n`, must be in the range $0 \leq n < \text{PORT_MAX_VECTOR_LEN}$.

Special Issues: In place computation is allowed; i.e., `pX` and/or `pY` can be equal to `pZ`.

Design/Implementation: The vector function `add` is implemented as a library function.

Code Example 9-2. `add`

```c
{
    Frac16 x16[10];
    Frac16 y16[10];
    Frac16 z16[10];

    vfr16Add (x16, y16, z16, sizeof(x16)/sizeof(x16[0]));
}
```
9.3.2 *dotProd*

Call(s):

```
Frac32 vfr16DotProd (Frac16 *pX, Frac16 *pY, UInt16 n);
```

Arguments:

<table>
<thead>
<tr>
<th>pX</th>
<th>in</th>
<th>Pointer to the first input vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>pY</td>
<td>in</td>
<td>Pointer to the second input vector</td>
</tr>
<tr>
<td>n</td>
<td>in</td>
<td>The number of elements in both input vectors</td>
</tr>
</tbody>
</table>

Description: The vector function *dotProd* calculates the dot product of two input vectors; i.e.,

\[
c = X \cdot Y.
\]

Equation:

\[
c = \sum_{i=0}^{n-1} x[i] \times y[i]
\]

*Eqn. 9-2*

Returns: The vector function *dotProd* returns the 32-bit fractional result of the dot product \((X \cdot Y)\) operation.

Range Issues: If saturation is enabled, any overflow or underflow of the dot product will return the maximum/minimum fractional value.

The length of the input and output vectors, \(n\), must be in the range \(0 \leq n < \text{PORT_MAXVECTORLEN}\).

Special Issues: None

Design/Implementation: The *dotProd* vector function is implemented as a library function.

Code Example 9-3. *dotProd*

```c
{
    Frac16 x16[2];
    Frac16 y16[2];

    Frac32 z;
    ...
    z = vfr16DotProd (x16, y16, 2);
    ...
}
```
9.3.3  

**equal**

**Call(s):**

```cpp
bool vfr16Equal (Frac16 *pX, Frac16 *pY, UInt16 n);
```

**Arguments:**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pX</code></td>
<td>in</td>
</tr>
<tr>
<td><code>pY</code></td>
<td>inout</td>
</tr>
<tr>
<td><code>n</code></td>
<td>in</td>
</tr>
</tbody>
</table>

**Description:** The vector function `equal` tests two vectors for equality; i.e., \( X = Y \); the actual implementation tests each corresponding element, \( pX[i] == pY[i], 0 <= i < n \). 

**Equation:** None

**Returns:** The function `equal` returns `true` if two vectors are equal; otherwise, `equal` returns `false`.

**Range Issues:** The length of the input and output arrays, \( n \), must be in the range \( 0 <= n < \text{PORT\_MAX\_VECTOR\_LEN} \).

**Special Issues:** None

**Design/Implementation:** The `equal` vector function is implemented as a library function.

**Code Example 9-4.  equal**

```cpp
{
    Frac16 x16[3];
    Frac16 y16[3];

    bool bIsEqual;

    ...

    bIsEqual = vfr16Equal (x16, y16, 3);
}
```
## 9.3.4 length

**Call(s):**

```c
Frac16 vfr16Length (Frac16 *pX, UInt16 n);
```

**Arguments:**

<table>
<thead>
<tr>
<th>pX</th>
<th>in</th>
<th>Pointer to the input vector of fractional data values</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td></td>
<td>The number of elements in the input vector</td>
</tr>
</tbody>
</table>

**Description:** The vector function length calculates the length, or amplitude, of the input vector.

**Equation:**

\[
\text{length} = \sqrt{\sum_{i=0}^{n-1} x[i]^2}
\]

**Returns:** The vector function length returns the length of the input vector.

**Range Issues:** No overflow or underflow is possible with this function.

The length of the input vector, \( n \), must be in the range \( 0 \leq n < \text{PORT_MAX_VECTOR_LEN} \).

**Special Issues:** None

**Design/Implementation:** The vector function length is implemented as a library function.

**Code Example 9-5. length**

```c
{
    Frac16 x16[4];
    Frac16 length;
    ...
    length = vfr16Length (x16, 4);
}
```
9.3.5 *mult*

Call(s):

```c
void vfl6Mult (Frac16 c, Frac16 *pX, Frac16 *pZ, UInt16 n);
```

Arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>c</code></td>
<td>A fractional value to multiply the input vector</td>
</tr>
<tr>
<td><code>pX</code></td>
<td>Pointer to the input vector</td>
</tr>
<tr>
<td><code>pZ</code></td>
<td>Pointer to the output vector</td>
</tr>
<tr>
<td><code>n</code></td>
<td>The number of elements in the input and output vectors</td>
</tr>
</tbody>
</table>

Description: The vector function *mult* multiplies an input vector by a fractional constant; i.e., \( Z = cX \). The actual calculation multiplies the constant \( c \) by each of the elements of the vector, i.e., \( pZ[i] = c \times pX[i] \), \( 0 \leq i < n \).

Equation:

\[
\begin{bmatrix}
  x_1 \\
  x_2 \\
  \vdots
\end{bmatrix} \times 
\begin{bmatrix}
  c \\
  c \\
  \vdots
\end{bmatrix} = 
\begin{bmatrix}
  c \times x_1 \\
  c \times x_2 \\
  \vdots
\end{bmatrix}
\]

Eqn. 9-4

Returns: The vector function *mult* returns the output vector pointed to by `pZ`, where \( pZ[i] = c \times pY[i] \), \( 0 \leq i < n \).

Range Issues: The length of the input and output vectors, `n`, must be in the range \( 0 \leq n < \text{PORT_MAX_VECTOR_LEN} \).

Special Issues: In place computation is allowed; i.e., `pX` can be equal to `pZ`.

Design/Implementation: The vector function *mult* is implemented as a library function.

Code Example 9-6. *mult*

```c
{
    Frac16 x16[3];
    Frac16 c;
    Frac16 z16[3];
    ...
    vfl6Mult (c, x16, z16, 3);
    ...
}
```
9.3.6 scale

Call(s):

```c
void vfr16Scale (UInt16 k, Frac16 *pX, Frac16 *pZ, UInt16 n);
```

Arguments:

<table>
<thead>
<tr>
<th>k</th>
<th>in</th>
<th>An unsigned, integer value with which to scale the input vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>pX</td>
<td>in</td>
<td>Pointer to the input vector</td>
</tr>
<tr>
<td>pZ</td>
<td>inout</td>
<td>Pointer to the output vector</td>
</tr>
<tr>
<td>n</td>
<td>in</td>
<td>The number of elements in the input and output vectors</td>
</tr>
</tbody>
</table>

Description: The vector function `scale` multiplies an input vector by an unsigned integer constant; i.e., \( Z = kX \). The actual calculation multiplies the constant \( k \) by each of the elements of the vector; i.e., \( pZ[i] = k \times pX[i], 0 \leq i < n \).

Equation:

\[
k \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} k \times x_1 \\ k \times x_2 \\ \vdots \\ k \times x_n \end{bmatrix} \quad \text{Eqn. 9-5}
\]

Returns: The vector function `scale` returns the output vector pointed to by \( pZ \), where \( pZ[i] = k \times pY[i], 0 \leq i < n \).

Range Issues: The length of the input and output vectors, \( n \), must be in the range \( 0 \leq n < \text{PORT_MAX_VECTOR_LEN} \).

Special Issues: In place computation is allowed; i.e., \( pX \) can be equal to \( pZ \).

Design/Implementation: The vector function `scale` is implemented as a library function.

Code Example 9-7. scale

```c
{
    Frac16 x16[3];
    UInt16 k;
    Frac16 z16[3];
    ...
    vfr16Scale (k, x16, z16, 3);
    ...
}
```
9.3.7 sub

Call(s):

```c
void vfr16Sub (Frac16 *pX, Frac16 *pY, Frac16 *pZ, UInt16 n);
```

Arguments:

<table>
<thead>
<tr>
<th>pX</th>
<th>in</th>
<th>Pointer to the first input vector of fractional data values</th>
</tr>
</thead>
<tbody>
<tr>
<td>pY</td>
<td>in</td>
<td>Pointer to the second input vector of fractional data values</td>
</tr>
<tr>
<td>pZ</td>
<td>inout</td>
<td>Pointer to the output vector of fractional results</td>
</tr>
<tr>
<td>n</td>
<td>in</td>
<td>The length of all input and output vectors</td>
</tr>
</tbody>
</table>

**Description:** The vector function `sub` provides a vector subtraction operation; i.e., \( Z = X - Y \); the actual calculation subtracts each corresponding element, \( pZ[i] = pX[i] - pY[i] \), \( 0 \leq i < n \).

**Equation:**

\[
x[1...n] - y[1...n] = z[1...n]
\]

**Eqn. 9-6**

**Returns:** The vector function `sub` returns the vector resulting from the vector subtraction, \( Z = X - Y \).

**Range Issues:** If saturation is enabled, the vector function `sub` will return the maximum or minimum fractional values if overflow or underflow occurs during the calculation of each output element.

The length of the input and output vectors, \( n \), must be in the range \( 0 \leq n < \text{PORT_MAX VECTOR LEN} \).

**Special Issues:** In place computation is allowed; i.e., \( pX \) and/or \( pY \) can be equal to \( pZ \).

**Design/Implementation:** The vector function `sub` is implemented as a library function.

**Code Example 9-8. sub**

```c
{
    Frac16 x16[10];
    Frac16 y16[10];
    Frac16 z16[10];

    vfr16Sub (x16, y16, z16, 10);
}
```
Chapter 10
Matrix Library

This section describes functions for matrices of 16-bit and 32-bit fractional values.

10.1 Introduction

The Digital Signal Processing Function Library implements functions for matrices of fractional types as library routines. The matrix operations provide highly optimized loops on the elements of each matrix in order to provide the best performance.

The function library provides such matrix functions for the 16-bit, \((\text{short})\), fractional type. For portability, we have defined the short 16-bit fractional type as \(\text{Frac16}\).

Section 10.2 presents the C interfaces for 16-bit fractional matrix functions, \(\text{xfr16.h}\); specifications for matrix functions are detailed in Section 10.3.

10.2 16-bit Fractional Interface

As shown in Code Example 10-1, the C interface for the 16-bit fractional matrix functions is defined in the C header file \(\text{xfr16.h}\).
/* File: xfr16.h */

#ifndef __XFR16_H
#define __XFR16_H

#include "port.h"

/*******************************************************
* Matrix Math - 16 bit fractional
*******************************************************/

void xfr16Add ( Frac16 *pX, int rows, int cols,
             Frac16 *pY,
             Frac16 *pZ);

void xfr16Sub  ( Frac16 *pX, int rows, int cols,
             Frac16 *pY,
             Frac16 *pZ);

void xfr16Mult ( Frac16 *pX, int xrows, int xcols,
             Frac16 *pY, int ycols,
             Frac16 *pZ);

bool xfr16Equal( Frac16 *pX, int rows, int cols,
              Frac16 *pY);

void xfr16Trans( Frac16 *pX, int xrows, int xcols,
              Frac16 *pZ);

Frac32 xfr16Inv ( Frac16 *pX, int rowscols,
             Frac16 *pZ);

Frac32 xfr16Det( Frac16 *pX, int rowscols);

#endif
10.3 Matrix Function Specifications

The following pages describe the matrix functions for the 16-bit fractional type.

Each page utilizes a short name to generically describe its function. In many cases, however, the actual function name that must be used in C or assembler may be different from the generic function name. Therefore, please refer to the C function declaration to find the specific function name to put in your program. For example, the generic function name for an absolute value function may be `inv`, but the actual function name that must be used for a 16-bit fractional matrix is `xfr16Inv`.

Function arguments for each routine are described as `in`, `out`, or `inout`. An `in` argument means that the parameter value is an input only to the function. An `out` argument means that the parameter value is an output only from the function. An `inout` argument means that a parameter value is an input to the function, but the same parameter is also an output from the function.

Typically, `inout` parameters are input pointer variables in which the caller passes the address of a preallocated data structure to a function. The function stores its results within that data structure. The actual value of the `inout` pointer parameter is not changed.
10.3.1 add

Call(s):

```c
void xfr16Add ( Frac16 *pX, int rows, int cols,
                Frac16 *pY, Frac16 *pZ);
```

Arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pX</td>
<td>in Pointer to an input matrix (X) of fractional data values</td>
</tr>
<tr>
<td>rows</td>
<td>in The number of rows in the matrices X, Y, and Z</td>
</tr>
<tr>
<td>cols</td>
<td>in The number of columns in the matrices X, Y, and Z</td>
</tr>
<tr>
<td>pY</td>
<td>in Pointer to an input matrix (Y) of fractional data values</td>
</tr>
<tr>
<td>pZ</td>
<td>inout Pointer to an output matrix (Z) of fractional data values</td>
</tr>
</tbody>
</table>

Description: The matrix function add totals two matrices, X and Y, to produce Z: \( Z = X + Y \).

Equation:

\[
\begin{bmatrix}
  x_{11} & x_{12} & x_{1n} \\
  x_{21} & x_{22} & x_{2n} \\
  \vdots & \vdots & \vdots \\
  x_{n1} & x_{n2} & x_{nn}
\end{bmatrix} + \begin{bmatrix}
  y_{11} & y_{12} & y_{1n} \\
  y_{21} & y_{22} & y_{2n} \\
  \vdots & \vdots & \vdots \\
  y_{n1} & y_{n2} & y_{nn}
\end{bmatrix} = \begin{bmatrix}
  z_{11} & z_{12} & z_{1n} \\
  z_{21} & z_{22} & z_{2n} \\
  \vdots & \vdots & \vdots \\
  z_{n1} & z_{n2} & z_{nn}
\end{bmatrix}
\]

Eqn. 10-1

Returns: The matrix function add stores the results of the matrix addition in the matrix pointed to by pZ.

Range Issues: If saturation is enabled, the matrix function add will store the maximum or minimum fractional values if overflow or underflow occurs during the calculation of any output element; if saturation is disabled, the result of add is undefined if overflow or underflow occurs during the calculation of any output element.

The maximum length of a row (rows) or column (cols), must be in the range \( 1 \leq n \leq \text{PORT_MAXVECTOR}_\text{LEN} \).

Special Issues: In place computation is allowed; i.e., pX and pY can be equal to pZ.

Design/Implementation: The add matrix function is implemented as a library function.

Code Example 10-2. add

```c
{
    Frac16 x16[10][10];
    Frac16 y16[10][10];
    Frac16 z16[10][10];

    xfr16Add (x16, 10, 10, y16, z16);
}
```
10.3.2 \textit{det} (determinant)

Call(s):

\[ \text{Frac32 xfr16Det( Frac16 *pX, int rowscols);} \]

Arguments:

<table>
<thead>
<tr>
<th>\textbf{Table 10-2. \textit{det} (matrix) arguments}</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{pX}</td>
</tr>
<tr>
<td>\textit{rowscols}</td>
</tr>
</tbody>
</table>

Description: The matrix function \textit{det} calculates the determinant of the square input matrix \( X \).

Equation:

\[ \text{det} \begin{bmatrix} x_{11} & x_{1n} \\ x_{2n} & x_{nn} \end{bmatrix} = z \quad \text{Eqn. 10-2} \]

Returns: The matrix function \textit{det} returns the results of the determinant calculation as the return value for the function.

Range Issues: If saturation is enabled, the matrix function \textit{det} will return the maximum or minimum fractional value if overflow or underflow occurs during the determinant calculation; if saturation is disabled, the result of \textit{det} is undefined if overflow or underflow occurs during the calculation of any output element.

The number of rows and columns of the input matrix must be in the range \( 1 <= n <= \text{PORT\_MAX\_VECTOR\_LEN} \).

Special Issues: The input matrix \( X \) must be a square matrix. \textit{Det} is currently implemented only for square matrices of two and three elements.

Design/Implementation: The matrix function \textit{det} is implemented as a library function.

Code Example 10-3. \textit{det}

\[
\begin{array}{l}
\{ \\
\quad \text{Frac16 x16[10][10];} \\
\quad \text{Frac32 Determinant;} \\
\quad \ldots \\
\quad \text{Determinant = xfr16Det (x16, 10);} \\
\}
\end{array}
\]
10.3.3 equal

Call(s):

```cpp
bool xfr16Equal(Frac16 *pX, int rows, int cols, Frac16 *pY);
```

Arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pX</code></td>
<td>Pointer to the first input matrix X of fractional data values</td>
</tr>
<tr>
<td><code>rows</code></td>
<td>The number of rows of the input matrices X and Y</td>
</tr>
<tr>
<td><code>cols</code></td>
<td>The number of columns of the input matrices X and Y</td>
</tr>
<tr>
<td><code>pY</code></td>
<td>Pointer to the second input matrix Y of fractional data values</td>
</tr>
</tbody>
</table>

**Description:** The matrix function `equal` determines whether two matrices are equal.

**Equation:** None

**Returns:** The matrix function `equal` returns `true` if all corresponding elements of the two input matrices are equal. If any element of X differs in value from the corresponding element of Y, `equal` will return `false`.

**Range Issues:** The maximum length of a row, `rows`, or column, `cols`, must be in the range `1 <= n <= PORT_MAX_VECTOR_LEN`.

**Special Issues:** None

**Design/Implementation:** The matrix function `equal` is implemented as a library function.

**Code Example 10-4. equal**

```cpp
{
    Frac16 x16[10][10];
    Frac16 y16[10][10];
    bool bEqual;

    ...
    bEqual = xfr16Equal(x16, 10, 10, y16);
}
```
10.3.4 inv (inverse)

Call(s):

Frac32 xfr16Inv ( Frac16 *pX, int rowscols,
                   Frac16 *pZ);

Arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pX</td>
<td>in</td>
<td>Pointer to a square input matrix X of fractional data values</td>
</tr>
<tr>
<td>rowscols</td>
<td>in</td>
<td>The number of rows and columns of the square input matrix X</td>
</tr>
<tr>
<td>pZ</td>
<td>inout</td>
<td>Pointer to a square output matrix Z of fractional data values</td>
</tr>
</tbody>
</table>

Description: The matrix function inv calculates the inverse matrix Z of the square input matrix X.

Equation:

\[
\begin{bmatrix} x_{11} & x_{1n} \\ x_{2n} & x_{nn} \end{bmatrix}^{-1} = \begin{bmatrix} z_{11} & z_{1n} \\ z_{2n} & z_{nn} \end{bmatrix}
\]

Eqn. 10-3

Returns: The matrix function inv stores the inverse matrix in the matrix pointed to by pZ. The inv function returns the value of the determinant of the matrix X.

Range Issues: If saturation is enabled, the matrix function inv will store the maximum or minimum fractional values if overflow or underflow occurs during the calculation of any output element; if saturation is disabled, the result of inv is undefined if overflow or underflow occurs during the calculation of any output element.

The number of rows and columns of the input matrix, (rowscols), must be in the range 1 <= n <= PORT_MAX_VECTOR_LEN.

Special Issues: The input matrix X must be a square matrix. Inv is currently implemented only for square matrices of two and three elements.

Design/Implementation: The matrix function inv is implemented as a library function.

Code Example 10-5. inv

```c
{
    Frac16 x16[3][3];
    Frac16 z16[3][3];
    Frac32 Determinant;

    ... 
    Determinant = xfr16Inv (x16, 3, z16);
}
```
10.3.5  **mult** (multiply)

Call(s):

```c
void xfr16Mult ( Frac16 *pX, int xrows, int xcols,
                Frac16 *pY, int ycols,
                Frac16 *pZ);
```

Arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pX</td>
<td>in Pointer to the first input matrix X of fractional data values</td>
</tr>
<tr>
<td>xrows</td>
<td>in The number of rows of the input matrix X</td>
</tr>
<tr>
<td>xcols</td>
<td>in The number of columns of the input matrix X; this is also the number of rows of the input matrix Y</td>
</tr>
<tr>
<td>pY</td>
<td>inout Pointer to the second input matrix Y of fractional data values</td>
</tr>
<tr>
<td>ycols</td>
<td>in The number of columns of the input matrix Y</td>
</tr>
<tr>
<td>pZ</td>
<td>in Pointer to the output matrix Z of fractional data values</td>
</tr>
</tbody>
</table>

Description: The matrix function *mult* multiplies the input matrix X by the input matrix Y to produce output matrix Z.

Equation:

\[
\begin{bmatrix}
  x_{11} & x_{1n} \\
  x_{2n} & x_{nn}
\end{bmatrix}
\begin{bmatrix}
  y_{11} & y_{1n} \\
  y_{2n} & y_{nn}
\end{bmatrix} =
\begin{bmatrix}
  z_{11} & z_{1n} \\
  z_{2n} & z_{nn}
\end{bmatrix}
\]

Eqn. 10-4

Returns: The matrix function *mult* stores the results of the matrix multiplication in the matrix pointed to by pZ.

Range Issues: If saturation is enabled, the matrix function *mult* will store the maximum or minimum fractional values if overflow or underflow occurs during the calculation of any output element; if saturation is disabled, the result of *mult* is undefined if overflow or underflow occurs during the calculation of any output element.

The number of rows (xrows), and columns (xcols, ycols), of both input matrices must be in the range 1 <= n <= PORT_MAX_VECTOR_LEN.

Special Issues: The size of the output matrix Z is xrows, rows, and ycols, columns.

In place computation is not allowed; i.e., pX and pY must not be equal to pZ.

Design/Implementation: The matrix function *mult* is implemented as a library function.
Code Example 10-6. *mult*

```c
{
    Frac16 x16[2][4];
    Frac16 y16[4][6];
    Frac16 z16[2][6];

    ...
    xfr16Mult (x16, 2, 4, y16, 6, z16);
}
```
10.3.6 sub (subtract)

Call(s):

```c
void xfr16Sub ( Frac16 *pX, int rows, int cols,
               Frac16 *pY,
               Frac16 *pZ);
```

Arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pX</td>
<td>Pointer to the first input matrix X of fractional data values</td>
</tr>
<tr>
<td>rows</td>
<td>The number of rows of the input matrix X; this is also the number of rows of the input matrix Y</td>
</tr>
<tr>
<td>cols</td>
<td>The number of columns of the input matrix X; this is also the number of columns of the input matrix Y</td>
</tr>
<tr>
<td>pY</td>
<td>Pointer to the second input matrix Y of fractional data values</td>
</tr>
<tr>
<td>pZ</td>
<td>Pointer to the output matrix Z of fractional data values</td>
</tr>
</tbody>
</table>

**Description:** The matrix function `sub` subtracts the input matrix `Y` from the input matrix `X`, producing the output matrix `Z`; i.e., \( Z = X - Y \).

**Equation:**

\[
\begin{bmatrix}
  x_{11} & x_{1n} \\
  x_{2n} & x_{nn}
\end{bmatrix} - 
\begin{bmatrix}
  y_{11} & y_{1n} \\
  y_{2n} & y_{nn}
\end{bmatrix} = 
\begin{bmatrix}
  z_{11} & z_{1n} \\
  z_{2n} & z_{nn}
\end{bmatrix}
\]

**Eqn. 10-5**

**Returns:** The matrix function `sub` stores the output matrix in the matrix pointed to by `pZ`.

**Range Issues:** If saturation is enabled, the matrix function `sub` will store the maximum or minimum fractional values if overflow or underflow occurs during the calculation of any output element; if saturation is disabled, the result of `sub` is undefined if overflow or underflow occurs during the calculation of any output element.

The number of rows (`rows`), and the number of columns (`cols`), of the input matrices `X` and `Y` must be in the range \( 1 <= n <= \text{PORT_MAX_VECTOR_LEN} \).

**Special Issues:** The size of the matrix `X` must be equal to the size of the matrix `Y`.

**Design/Implementation:** The matrix function `sub` is implemented as a library function.
Code Example 10-7.  \textit{sub}

\begin{verbatim}
{
    Frac16 x16[10][10];
    Frac16 y16[10][10];
    Frac16 z16[10][10];

    ...
    xfr16Sub (x16, 10, 10, y16, z16);
}
\end{verbatim}
10.3.7  
**trans** (transpose)

Call(s):

```c
void xfr16Trans( Frac16 *pX, int xrows, int xcols,
                   Frac16 *pZ);
```

Arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pX</td>
<td>in</td>
<td>Pointer to the input matrix ( X ) of fractional data values</td>
</tr>
<tr>
<td>xrows</td>
<td>in</td>
<td>The number of rows of the input matrix ( X ); this is also the number of columns of the output matrix ( Z )</td>
</tr>
<tr>
<td>xcols</td>
<td>in</td>
<td>The number of columns of the input matrix ( Y ); this is also the number of rows of the output matrix ( Z )</td>
</tr>
<tr>
<td>pZ</td>
<td>inout</td>
<td>Pointer to the output matrix ( Z ) of fractional data values</td>
</tr>
</tbody>
</table>

Description: The matrix function `trans` transposes the input matrix \( X \) and produces the output matrix \( Z \).

Equation:

\[
\begin{pmatrix}
x_{11} & x_{1n} \\
x_{2n} & x_{nn}
\end{pmatrix}^T =
\begin{pmatrix}
z_{11} & z_{1n} \\
z_{2n} & z_{nn}
\end{pmatrix}
\]  

Returns: The matrix function `trans` stores the transpose of the input matrix \( X \) in the matrix pointed to by \( pZ \).

Range Issues: The number of rows \( (xrows) \), and the number of columns \( (xcols) \), of the input matrix must be in the range \( 1 \leq n \leq \text{PORT_MAX_VECTOR_LEN} \).

Special Issues: The size of the output matrix \( Z \) is \( xrows \), columns, and \( xcols \), rows.

In place computation is not allowed; i.e., \( pX \) must not be equal to \( pZ \).

Design/Implementation: The matrix function `trans` is implemented as a library function.

Code Example 10-8.  `trans`

```c
{
    Frac16 x16[5][10];
    Frac16 z16[10][5];

    ...  
    xfr16Trans (x16, 5, 10, z16);
}
```
Chapter 11
Signal Processing Library

This section describes signal processing functions for 16-bit fractional values.

11.1 Introduction

The DSP568xx Digital Signal Processing Function Library provides many different types of signal processing algorithms. Most of these signal processing algorithms, such as FFT, FIR, IIR, and correlation, are general purpose. These algorithms can be used for digital filters, speech coders, vector and array processing, and motor control.

The function library provides such signal processing functions for 16-bit, (short), fractional types. For portability, we have defined a short 16-bit fractional type as Frac16 in the header file port.h.

11.2 16-bit Fractional Interface

As shown in Code Example 11-1, the C interface for the 16-bit fractional DSP functions is defined in the C header file dfr16.h.

Code Example 11-1.  C Header File dfr16.h

/* File: dfr16.h */

#ifndef __DFR16_H
#define __DFR16_H

#include "port.h"
#include "dfr16priv.h"    /* MIEL_PRASAD */

#endif

Code Example 11-1.  C Header File dfr16.h
/* To switch between C and assembly implementations
 *       #if 0  => assembly
 *       #if 1  => C
 */

#if 0
#define dfr16FIR      dfr16FIRC
#define dfr16FIRInt   dfr16FIRIntC
#define dfr16AutoCorr dfr16AutoCorrC
#define dfr16Corr     dfr16CorrC
#endif

/*********************************************************
* Flags for Fast Fourier Transform (FFT) Functions
**********************************************************/

#ifndef FFT_DEFAULT_OPTIONS
#define FFT_DEFAULT_OPTIONS               0  /* Default all options bits to 0     */
#define FFT_SCALE_RESULTS_BY_N            1  /* Unconditionally scale by N        */
#define FFT_SCALE_RESULTS_BY_DATA_SIZE    2  /* Scale according to data sizes     */
#define FFT_INPUT_IS_BITREVERSED          4  /* Default to normal (linear) input  */
#define FFT_OUTPUT_IS_BITREVERSED         8  /* Default to normal (linear) output */
#endif

/********************************************************
* Data Structure for Real FFT and Real Inverse FFT
*********************************************************/

typedef struct {
    Frac16 z0;
    Frac16 zNDiv2;
    CFrac16 cz[1];
} dfr16_sInplaceCRFFT;

/*********************************************************
* 16-bit Complex Fractional Forward Fast Fourier Transform
**********************************************************/
#define dfr16CFFTCreate(n,o) dfr16CFFTCreateFFT_SIZE_##n(o)
#define dfr16CFFTInit(p,n,o)   dfr16CFFTInitFFT_SIZE_##n(p,o)

EXPORT dfr16_tCFFTStruct *dfr16CFFTCreate   (8,   UInt16 options);
EXPORT dfr16_tCFFTStruct *dfr16CFFTCreate   (16,  UInt16 options);
EXPORT dfr16_tCFFTStruct *dfr16CFFTCreate   (32,  UInt16 options);
EXPORT dfr16_tCFFTStruct *dfr16CFFTCreate   (64,  UInt16 options);
EXPORT dfr16_tCFFTStruct *dfr16CFFTCreate   (128, UInt16 options);
EXPORT dfr16_tCFFTStruct *dfr16CFFTCreate   (256, UInt16 options);
EXPORT dfr16_tCFFTStruct *dfr16CFFTCreate   (512, UInt16 options);
EXPORT dfr16_tCFFTStruct *dfr16CFFTCreate   (1024,UInt16 options);
EXPORT dfr16_tCFFTStruct *dfr16CFFTCreate   (2048,UInt16 options);
EXPORT void dfr16CFFTInit (dfr16_tCFFTStruct * pCFFT, 8, Uint16 options);
EXPORT void dfr16CFFTInit (dfr16_tCFFTStruct * pCFFT, 16, Uint16 options);
EXPORT void dfr16CFFTInit (dfr16_tCFFTStruct * pCFFT, 32, Uint16 options);
EXPORT void dfr16CFFTInit (dfr16_tCFFTStruct * pCFFT, 64, Uint16 options);
EXPORT void dfr16CFFTInit (dfr16_tCFFTStruct * pCFFT, 128, Uint16 options);
EXPORT void dfr16CFFTInit (dfr16_tCFFTStruct * pCFFT, 256, Uint16 options);
EXPORT void dfr16CFFTInit (dfr16_tCFFTStruct * pCFFT, 512, Uint16 options);
EXPORT void dfr16CFFTInit (dfr16_tCFFTStruct * pCFFT, 1024, Uint16 options);
EXPORT void dfr16CFFTInit (dfr16_tCFFTStruct * pCFFT, 2048, Uint16 options);

EXPORT void dfr16CFFTDestroy (dfr16_tCFFTStruct *pCFFT);

EXPORT Result dfr16CFFT (dfr16_tCFFTStruct *pCFFT, CFrac16 *pX, CFrac16 *pZ);

/*********************************************************
* 16-bit Complex Fractional Inverse Fast Fourier Transform
**********************************************************/
#define dfr16CIFFTCreate(n,o) dfr16CFFTCreateFFT_SIZE_##n(o)
#define dfr16CIFFTInit(p,n,o) dfr16CFFTInitFFT_SIZE_##n(p,o)

EXPORT void dfr16CIFFTDestroy (dfr16_tCFFTStruct * pCIFFT);
EXPORT Result dfr16CIFFT (dfr16_tCFFTStruct *pCIFFT, CFrac16 *pX, CFrac16 *pZ);

/**********************************************************
* 16-bit In-Place Fractional Forward Fast Fourier Transform
**********************************************************/
#define dfr16RFFTCreate(n,o) dfr16RFFTCreateFFT_SIZE_##n(o)
#define dfr16RFFTInit(p,n,o) dfr16RFFTInitFFT_SIZE_##n(p,o)

EXPORT dfr16_tRFFTStruct *dfr16RFFTCreate (8, Uint16 options);
EXPORT dfr16_tRFFTStruct *dfr16RFFTCreate (16, Uint16 options);
EXPORT dfr16_tRFFTStruct *dfr16RFFTCreate (32, Uint16 options);
EXPORT dfr16_tRFFTStruct *dfr16RFFTCreate (64, Uint16 options);
EXPORT dfr16_tRFFTStruct *dfr16RFFTCreate (128, Uint16 options);
EXPORT dfr16_tRFFTStruct *dfr16RFFTCreate (256, Uint16 options);
EXPORT dfr16_tRFFTStruct *dfr16RFFTCreate (512, Uint16 options);
EXPORT dfr16_tRFFTStruct *dfr16RFFTCreate (1024, Uint16 options);
EXPORT dfr16_tRFFTStruct *dfr16RFFTCreate (2048, Uint16 options);

EXPORT void dfr16RFFTInit   (dfr16_tRFFTStruct * pRFFT, 8,    Uint16 options);
EXPORT void dfr16RFFTInit   (dfr16_tRFFTStruct * pRFFT, 16,   Uint16 options);
EXPORT void dfr16RFFTInit   (dfr16_tRFFTStruct * pRFFT, 32,   Uint16 options);
EXPORT void dfr16RFFTInit   (dfr16_tRFFTStruct * pRFFT, 64,   Uint16 options);
EXPORT void dfr16RFFTInit   (dfr16_tRFFTStruct * pRFFT, 128,  Uint16 options);
EXPORT void dfr16RFFTInit   (dfr16_tRFFTStruct * pRFFT, 256,  Uint16 options);
EXPORT void dfr16RFFTInit   (dfr16_tRFFTStruct * pRFFT, 512,  Uint16 options);
EXPORT void dfr16RFFTInit   (dfr16_tRFFTStruct * pRFFT, 1024, Uint16 options);
EXPORT void dfr16RFFTInit   (dfr16_tRFFTStruct * pRFFT, 2048, Uint16 options);

EXPORT void    dfr16RFFTDestroy  (dfr16_tRFFTStruct * pRFFT);
EXPORT Result  dfr16RFFT (dfr16_tRFFTStruct * pRFFT, Frac16 *pX, dfr16_sInplaceCRFFT *pZ);

16-bit In-Place Inverse Fractional Fast Fourier Transform

#define dfr16RIFFTCreate(n,o) dfr16RFFTCreateFFT_SIZE_##n(o)
#define dfr16RIFFTInit(p,n,o) dfr16RFFTInitFFT_SIZE_##n(p,o)

EXPORT void dfr16RIFFTDestroy (dfr16_tRFFTStruct * pRIFFT);

EXPORT Result dfr16RIFFT  (dfr16_tRFFTStruct * pRIFFT, dfr16_sInplaceCRFFT *pX, Frac16 *p2);

16-bit Fractional Fast Fourier Transform Utilities

EXPORT Result dfr16Cbitrev (CFrac16 *pX, CFrac16 *pZ, UInt16 n);

16-bit Fractional FIR Filters

typedef struct dfr16_sFirStruct {
  Frac16   * pC;                 /* Coefficients for the filter */
  Frac16   * pHistory;           /* Memory for the filter history buffer */
  UWord16    Private[6];
} dfr16_tFirStruct;

EXPORT dfr16_tFirStruct * dfr16FIRCreate     (Frac16 *pC, UInt16 n);
EXPORT void               dfr16FIRDestroy    (dfr16_tFirStruct *pFIR);
EXPORT void               dfr16FIRInit       (dfr16_tFirStruct *pFIR, Frac16 *pC,
UInt16 n);
EXPORT void               dfr16FIRHistory    (dfr16_tFirStruct *pFIR, Frac16 *pX);
EXPORT void               dfr16FIR           (dfr16_tFirStruct *pFIR, Frac16 *pX, Frac16 *pZ, UInt16 n);
EXPORT Frac16             dfr16FIRs          (dfr16_tFirStruct *pFIR, Frac16 x);

#define dfr16_tFirDecStruct dfr16_tFirStruct
EXPORT dfr16_tFirDecStruct * dfr16FIRDecCreate  (Frac16 *pC, UInt16 n, UInt16 f);
/* EXPORT void               dfr16FIRDecDestroy (dfr16_tFirDecStruct *pFIRDec); */
#define dfr16FIRDecDestroy dfr16FIRDestroy
EXPORT void               dfr16FIRDecInit       (dfr16_tFirDecStruct *pFIRDec, Frac16 *pC,
UInt16 n,     UInt16 f);
EXPORT UInt16              dfr16FIRDec        (dfr16_tFirDecStruct *pFIRDec, Frac16 *pX, Frac16 *p2, UInt16 nx);

#define dfr16_tFirIntStruct dfr16_tFirStruct

Freescale Semiconductor, Inc.
16-bit Fractional Interface

**MOTOROLA Signal Processing Library**

11-5

```c
EXPORT dfr16_tFirIntStruct * dfr16FIRIntCreate (Frac16 *pC, UInt16 n, UInt16 f);
/* EXPORT void               dfr16FIRIntDestroy (dfr16_tFirIntStruct *pFIRInt); */
#define dfr16FIRIntDestroy dfr16FIRDestroy

EXPORT void                  dfr16FIRIntInit    (dfr16_tFirIntStruct *pFIRInt, Frac16 *pC, UInt16 n, UInt16 f);

EXPORT UInt16                dfr16FIRInt        (dfr16_tFirIntStruct *pFIRInt, Frac16 *pX, Frac16 *pZ, UInt16 n);

/****************************************************************************
* 16-bit Fractional IIR Filters
****************************************************************************/
#define FILT_STATES_PER_BIQ 2
#define FILT_COEF_PER_BIQ   5
typedef struct dfr16_sIirStruct dfr16_tIirStruct;

EXPORT dfr16_tIirStruct * dfr16IIRCreate  (Frac16 *pC, UInt16 nbiq);
EXPORT void               dfr16IIRDestroy (dfr16_tIirStruct *pIIR);

EXPORT Result             dfr16IIR        (dfr16_tIirStruct *pIIR, Frac16 *pX, Frac16 *pZ, UInt16 n);

/****************************************************************************
* 16-bit Fractional Correlations
****************************************************************************/
#ifndef CORR_RAW
#define CORR_RAW      0  /* Select Raw correlation     */
#define CORR_BIAS     1  /* Select Bias correlation    */
#define CORR_UNBIAS   2  /* Select Unbias correlation  */
#endif

EXPORT Result  dfr16AutoCorr (UInt16 options, Frac16 *pX, Frac16 *pZ, UInt16 nx, UInt16 nz);
EXPORT Result  dfr16Corr     (UInt16 options, Frac16 *pX, Frac16 *pY, Frac16 *pZ, UInt16 nx, UInt16 ny);
#endif
#endif
```
11.3 Signal Processing Function Specifications

The following pages describe the digital signal processing functions for 16-bit fractional types.

Each page utilizes a short name to generically describe its function. In many cases, however, the actual function name that must be used in C or assembler may be different from the generic function name. Therefore, please refer to the C function declaration to find the specific function name to put in your program. For example, the generic function name for an auto-correlation algorithm may be `autoCorr`, but the actual function name that must be used for a 16-bit fractional vector is `dfrl6AutoCorr`.

Function arguments for each routine are described as `in`, `out`, or `inout`. An `in` argument means that the parameter value is an input only to the function. An `out` argument means that the parameter value is an output only from the function. An `inout` argument means that a parameter value is an input to the function, but the same parameter is also an output from the function.

Typically, `inout` parameters are input pointer variables in which the caller passes the address of a preallocated data structure to a function. The function stores its results within that data structure. The actual value of the `inout` pointer parameter is not changed.
11.3.1 autoCorr - Auto-Correlation

Call(s):

```c
#define CORR_RAW 0  /* Select Raw correlation */
#define CORR_BIAS 1  /* Select Bias correlation */
#define CORR_UNBIAS 2  /* Select Unbias correlation */
```

Result: `dfr16AutoCorr` (UInt16 options, Frac16 *pX, Frac16 *pZ, UInt16 nx, UInt16 nz);

Arguments:

<table>
<thead>
<tr>
<th>options</th>
<th>in</th>
<th>Selects raw, biased, or unbiased auto correlation</th>
</tr>
</thead>
<tbody>
<tr>
<td>pX</td>
<td>in</td>
<td>Pointer to the input vector of <code>nx</code> data elements</td>
</tr>
<tr>
<td>pZ</td>
<td>in/out</td>
<td>Pointer to the output vector of <code>nz</code> data elements</td>
</tr>
<tr>
<td><code>nx</code></td>
<td>in</td>
<td>Length of the input vector pointed to by <code>pX</code></td>
</tr>
<tr>
<td><code>nz</code></td>
<td>in</td>
<td>Length of the output vector pointed to by <code>pZ</code></td>
</tr>
</tbody>
</table>

Description: Computes the first `nz` points of auto-correlation of a vector of fractional data values.

Three different types of auto-correlation may be selected via the `options` parameter: raw, biased, and unbiased. One of the three following values must be passed in the `options` parameter to `autoCorr`, as shown in Table 11-2.

Table 11-2. `autoCorr` options

<table>
<thead>
<tr>
<th>Options</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>CORR_RAW</td>
<td>If set, the auto-correlation type will be raw; i.e., bias = 1 in the following auto-correlation algorithm</td>
</tr>
<tr>
<td>CORR_BIAS</td>
<td>If set, the auto-correlation type will be biased; i.e., bias = (1 / <code>nx</code>) in the following auto-correlation algorithm</td>
</tr>
<tr>
<td>CORR_UNBIAS</td>
<td>If set, the auto-correlation type will be unbiased; i.e., bias = (1 / (nz - j)) in the following auto-correlation algorithm</td>
</tr>
</tbody>
</table>

Equation: See the Algorithm in this section.

Returns: The `autoCorr` computation generates `nz` output values, which are stored in the vector pointed to by `pZ`. The function `autoCorr` returns:

FAIL ("-1"): if the length of the output vector, `nz`, is outside the range

0 <= `nz` <= min(2nx-1, 8192), or if the `options` parameter is invalid

PASS ("0"): otherwise
Algorithm:

\[
\begin{align*}
z[j] &= \text{bias} \sum_{k=0}^{nx-1} x[j+k]x[k] & 0 \leq j < nx - 1 \\
&= \text{bias} \sum_{k=0}^{nx} x[j+k]x[k] & 1 \leq j < nx
\end{align*}
\]

**Range Issues:** If saturation is enabled, the \textit{autoCorr} function will return the maximum or minimum fractional values if overflow or underflow occurs during the calculation of each output element.

The length of the input and output vectors, \( nx \) and \( nz \), must be within the range \( 0 \leq nx, nz \leq 8192 \).

**Special Issues:** In place computation is not allowed; i.e., the address of the input vector, \( pX \), must be different than the address of the output vector, \( pZ \).

**Design/Implementation:** None

---

**Code Example 11-2. autoCorr**

```c
{ 
    Result        res;
    UInt16        options;
    Frac16        *pX;
    Frac16        *pZ;
    UInt16        nx;
    UInt16        nz;

    res = dfr16AutoCorr (options, pX, pZ, nx, nz);

    ...
}
```
11.3.2 cbirev - Complex Bit Reverse

Call(s):

Result dfr16Cbitrev (CFrac16 *pX, CFrac16 *pZ, UInt16 n);

Arguments:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>pX</td>
<td>in/out</td>
<td>Pointer to the input vector of ( n ) complex data elements</td>
</tr>
<tr>
<td>pZ</td>
<td>in/out</td>
<td>Pointer to the output vector of ( n ) complex data elements</td>
</tr>
<tr>
<td>n</td>
<td>in</td>
<td>Length of the input and output vectors pointed to by ( pX ) and ( pZ )</td>
</tr>
</tbody>
</table>

Description: The \textit{cbirev} function bit-reverses the position of \( n \) complex data elements in the input vector, pointed to by \( pX \). The \textit{cbirev} function is used to convert normal, (linear), order complex vectors to bit-reversed complex vectors, and vice-versa, for use in Fast Fourier Transform functions.

Equation: None

Returns: The \textit{cbirev} computation generates \( n \) output values, which are stored in the vector pointed to by \( pZ \).

The function \textit{cbirev} returns:

\textbf{FAIL ("-1")}: if the length of the input and output vectors, \( n \), is outside the range \( 0 < n \leq 8192 \)

\textbf{PASS ("0")}: otherwise

Range Issues: The length of the input and output vectors, \( n \), must be within the range \( 0 \leq n \leq 8192 \).

Special Issues: In place computation is allowed; i.e., the address of the input vector, \( pX \), may be equal to the address of the output vector, \( pZ \).

Design/Implementation: None
11.3.3 **cfft - Complex Fast Fourier Transform**

**Call(s):**

```c
typedef struct {
    UInt16 options; /*Option flags to control cfft algorithm operation */
    UInt16 n; /* Length of the complex FFT input and output vectors:
                8, 16, 32, 64, 128, 256, 512, 1024, 2048 */
    CFrac16 *Twiddle; /* Pointer to twiddle factor table */
    Int16 No_of_Stages; /* The number of FFT stages */
} dfr16_tCFFTStruct;

Result dfr16CFFT (dfr16_tCFFTStruct *pCFFT, CFrac16 *pX, CFrac16 *pZ);
```

**Arguments:**

<table>
<thead>
<tr>
<th>pCFFT</th>
<th>in</th>
<th>Pointer to a data structure containing private data for the cfft function. This pointer is created by a call to cfftCreate</th>
</tr>
</thead>
<tbody>
<tr>
<td>pX</td>
<td>in/out</td>
<td>Pointer to the input vector of n complex fractional data</td>
</tr>
<tr>
<td>pZ</td>
<td>in/out</td>
<td>Pointer to the output structure of complex values</td>
</tr>
</tbody>
</table>

**Description:** Computes an in place, radix-2, complex, decimation-in-time, (DIT), forward Fast Fourier Transform function for a vector of complex fractional data values. Prior to any call to cfft, the function must be initialized via a call to cfftCreate. The function cfft uses the private data structure established by cfftCreate to maintain its state between function calls.

This algorithm can perform its operations in place/non-in place. If pX is equal to pZ, (in place computation), the contents of this input vector are destroyed as a result of the cfft function; the input vector is overlaid with the output results. If pX is not equal to pZ, (non-in place computation), the cfft algorithm is then performed in place using the output structure’s memory, leaving the input vector intact.

Options valid in the options parameter direct the cfft algorithm operation; see Table 11-5 for details.

<table>
<thead>
<tr>
<th>Options</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFT_DEFAULTOPTIONS</td>
<td>If selected, cfft performs no scaling, assumes input to be in normal order, and writes output in normal order</td>
</tr>
<tr>
<td>FFT_SCALE_RESULTS_BY_N</td>
<td>If set, cfft will unconditionally scale down the results by n</td>
</tr>
<tr>
<td>FFT_SCALE_RESULTS_BY_DATA_SIZE</td>
<td>If set, cfft will conditionally scale down the results, depending upon the data sizes</td>
</tr>
<tr>
<td>FFT_INPUT_IS_BITREVERSED</td>
<td>If set, cfft assumes that the input vector is bit-reversed; otherwise, cfft assumes that the input vector is normal</td>
</tr>
<tr>
<td>FFT_OUTPUT_IS_BITREVERSED</td>
<td>If set, cfft will leave the output vector bit-reversed; otherwise, cfft leaves the output vector in normal order</td>
</tr>
</tbody>
</table>

For more information, see Section 11.3.4
Equation: See the Algorithm in this section

Returns: The function `cfft` returns either “FAIL” or the amount of scaling done:

Note: Scaling/Scale down implies Right Shift; Scale up implies Left Shift.

-1: Indicates FAIL

(0 to \(\log_2 n\), both inclusive): Indicates that the result was scaled down by so many times.

Algorithm:

\[
z[k] = scale \sum_{i=0}^{n-1} x[i](\cos(2 \pi \cdot i \cdot k / n) - jsin(2 \pi \cdot k / n)) \quad 0 \leq k < n
\]

Range Issues:

- The complex input data values satisfy the following constraint in order to get proper results:

  \[
  |x[i]| = ? |a_r + a_i| < 1
  \]

  where \(a_r\) and \(a_i\) are the real and imaginary parts of input complex data point \(x[i]\).

- Scaling can be controlled as an input to the `cfftCreate` call. If the `options` parameter to `cfftCreate` includes `FFT_DEFAULT_OPTIONS`, no scaling is performed (i.e., `scale = 1`). If `FFT_SCALE_RESULTS_BY_N`, (Auto scaling), is selected, `cfft` always returns:

  \[
  \log_2 n
  \]

  indicating that the results were scaled by \(1/n\), (n must be a power of 2). If `FFT_SCALE_RESULTS_BY_DATA_SIZE`, (block floating point scaling,) is selected, `cfft` returns “S”, where:

  \[
  0 \leq S \leq \log_2 n
  \]

  indicating that the results are scaled down by \(2^S\).

- Enabling or disabling the saturation bit before calling `cfft` will not have any effect on the results. The `cfft` function always disables it internally, performs the calculations, and restores the saturation bit before returning. This is done to prevent intermediate saturation of results, which otherwise would give inaccurate results at the end.

Special Issues:

- The assembly implementation of FFT uses two-level nesting of the **do loop**. If the user application (either a calling routine or an ISR) must call FFT in a **loop**, it is advised to use a software loop because the third do loop overhead is higher than that of the software loop.

- See Special Issues in Section 11.3.7.

Design/Implementation: For details on the data structure allocation utilized for the `cfft` function, see the description of `cfftCreate`, Section 11.3.4.
{  
dfr16_tCFFTStruct *pCFFT;
Result res;
CFrac16 *pX, *pZ;
UInt16 options = FFT_SCALE_RESULTS_BY_N;

...  
pCFFT = dfr16CFFTCreate (8, options); /* N = 8 point CFFT */
...  
res = dfr16CFFT (pCFFT, pX, pZ);
...  
dfr16CFFTDestroy (pCFFT);
}

11.3.4 cfftCreate - Create Complex FFT

Call(s):

dfr16_tCFFTStruct *dfr16CFFTCreate (UInt16 n, UInt16 options);

Arguments:

<table>
<thead>
<tr>
<th>n</th>
<th>in</th>
<th>Length of the complex FFT input and output vectors: 8, 16, 32, 64, 128, 256, 512, 1024, 2048</th>
</tr>
</thead>
<tbody>
<tr>
<td>options</td>
<td>in</td>
<td>Option flags to control cfft algorithm operation</td>
</tr>
</tbody>
</table>

Description: Performs the initialization for an in place, radix-2, complex, decimation-in-time (DIT), forward Fast Fourier Transform (FFT) function, cfft, using complex fractional data values. The function cfftCreate allocates and initializes a data structure, which is used by cfft to preserve the FFT function's state between calls.

Options valid in the options parameter direct cfft algorithm operation; see Table 11-7 for details.

<table>
<thead>
<tr>
<th>Options</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFT_DEFAULT_OPTIONS</td>
<td>If selected, cfft performs no scaling, assumes input to be in normal order, and writes output in normal order.</td>
</tr>
<tr>
<td>FFT_SCALE_RESULTS_BY_N</td>
<td>If set, cfft will unconditionally scale down the results by n.</td>
</tr>
<tr>
<td>FFT_SCALE_RESULTS_BY_DATA_SIZE</td>
<td>If set, cfft will conditionally scale down the results depending upon the data sizes.</td>
</tr>
<tr>
<td>FFT_INPUT_IS_BITREVERSED</td>
<td>If set, cfft assumes that the input vector is bit-reversed; otherwise, cfft assumes that the input vector is normal.</td>
</tr>
<tr>
<td>FFT_OUTPUT_IS_BITREVERSED</td>
<td>If set, cfft will leave the output vector bit-reversed; otherwise, cfft leaves the output vector in normal order.</td>
</tr>
</tbody>
</table>

Equation: None

Returns: The cfftCreate function returns a pointer to a private data structure, which must then be passed to subsequent calls of the cfft function.

Note: Because cfftCreate does not return FAIL status which may be due to a fail in the memory allocation for private data structure, the user must check the returned pointer for NULL.

Range Issues: The length of the complex FFT input vector, n, must be set to one of the following values: 8, 16, 32, 64, 128, 256, 512, 1024, 2048.

Special Issues:

- INCLUDE_MEMORY must be defined in appconfig.h in order to initialize the Memory Manager which is used by the cfftCreate function.
The `cfftCreate` function must be called before any call to `cfft` or `cfftDestroy`. Multiple calls may be made to `cfftCreate`, however, to initialize different `cfft` functions which could be used concurrently. Call `cfftDestroy` to deallocate the data structure allocated by `cfftCreate`.

The parameter `n` in the `cfftCreate` function must be passed as a number, and not through a variable or #define.

Options `FFT_SCALE_RESULTS_BY_N` and `FFT_SCALE_RESULTS_BY_DATA_SIZE` are not allowed simultaneously. If both are selected, the `cfft` function returns inaccurate results without any indication. The following is a list of valid options:

- `FFT_DEFAULT_OPTIONS`
- `FFT_DEFAULT_OPTIONS+FFT_INPUT_IS_BITREVERSED`
- `FFT_DEFAULT_OPTIONS+FFT_OUTPUT_IS_BITREVERSED`
- `FFT_DEFAULT_OPTIONS+FFT_INPUT_IS_BITREVERSED+FFT_OUTPUT_IS_BITREVERSED`
- `FFT_SCALE_RESULTS_BY_N`
- `FFT_SCALE_RESULTS_BY_N+FFT_INPUT_IS_BITREVERSED`
- `FFT_SCALE_RESULTS_BY_N+FFT_OUTPUT_IS_BITREVERSED`
- `FFT_SCALE_RESULTS_BY_N+FFT_INPUT_IS_BITREVERSED+FFT_OUTPUT_IS_BITREVERSED`
- `FFT_SCALE_RESULTS_BY_DATA_SIZE`
- `FFT_SCALE_RESULTS_BY_DATA_SIZE+FFT_INPUT_IS_BITREVERSED`
- `FFT_SCALE_RESULTS_BY_DATA_SIZE+FFT_OUTPUT_IS_BITREVERSED`
- `FFT_SCALE_RESULTS_BY_DATA_SIZE+FFT_INPUT_IS_BITREVERSED+FFT_OUTPUT_IS_BITREVERSED`

**Design/Implementation:** In general, the data structure allocated by `cfftCreate` is treated as private data to better encapsulate the details of the implementation. To review the private data structure, refer to the actual implementation of `cfftCreate`.
11.3.5 **cfftDestroy** - Destroy Complex FFT

Call(s):

```c
void dfr16CFFTDestroy (dfr16_tCFFTStruct *pCFFT);
```

Arguments:

<table>
<thead>
<tr>
<th>pCFFT</th>
<th>in</th>
</tr>
</thead>
<tbody>
<tr>
<td>in Pointer to a data structure created by the <code>cfftCreate</code> function</td>
<td></td>
</tr>
</tbody>
</table>

**Description:** The function `cfftDestroy` deallocates the data structure(s) initially allocated by `cfftCreate`.

**Equation:** None

**Returns:** Void

**Range Issues:** None

**Special Issues:** The function `cfftDestroy` may be called only once to deallocate a data structure created by `cfftCreate`.

**Design/Implementation:** None
11.3.6 *cfftInit* - Initialize Complex FFT

Call(s):

```c
dfrl6CFFTInit (dfrl6_tCFFTStruct * pCFFT, n, UInt16 options);
```

Arguments:

**Table 11-9. cfftInit arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>pCFFT</em></td>
<td>in</td>
<td>Pointer to a data structure containing private data for the <em>cfft</em> function; this pointer is statically allocated by caller prior to the <em>dfrl6CFFTInit</em> function.</td>
</tr>
<tr>
<td><em>n</em></td>
<td>in</td>
<td>Length of the complex FFT input and output vectors: 8, 16, 32, 64, 128, 256, 512, 1024, 2048</td>
</tr>
<tr>
<td><em>options</em></td>
<td>in</td>
<td>Option flags to control <em>cfft</em> algorithm operation</td>
</tr>
</tbody>
</table>

**Description:** Performs the initialization for an in-place, radix-2, complex, decimation-in-time (DIT), forward Fast Fourier Transform (FFT) function, *cfft*, using complex fractional data values. Initializes the function from a previously-allocated *dfrl6_tCFFTStruct* data structure. Typically, the *dfrl6_tCFFTStruct* data structure is allocated statically by the caller prior to the *cfftInit* call. The *cfftCreate* function also uses *cfftInit* to initialize the *dfrl6_tCFFTStruct* that it dynamically allocates.

Options valid in the *options* parameter direct *cfft* algorithm operation; see **Table 11-10** for details.

**Table 11-10. cfftCreate options**

<table>
<thead>
<tr>
<th>Options</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFT_DEFAULT_OPTIONS</td>
<td>If selected, <em>cfft</em> performs no scaling, assumes input to be in normal order, and writes output in normal order</td>
</tr>
<tr>
<td>FFT_SCALE_RESULTS_BY_N</td>
<td>If set, <em>cfft</em> will unconditionally scale down the results by <em>n</em></td>
</tr>
<tr>
<td>FFT_SCALE_RESULTS_BY_DATA_SIZE</td>
<td>If set, <em>cfft</em> will conditionally scale down the results, depending upon the data sizes</td>
</tr>
<tr>
<td>FFT_INPUT_IS_BITREVERSED</td>
<td>If set, <em>cfft</em> assumes that the input vector is bit-reversed; otherwise, <em>cfft</em> assumes that the input vector is normal</td>
</tr>
<tr>
<td>FFT_OUTPUT_IS_BITREVERSED</td>
<td>If set, <em>cfft</em> will leave the output vector bit-reversed; otherwise, <em>cfft</em> leaves the output vector in normal order</td>
</tr>
</tbody>
</table>

**Equation:** None

**Returns:** None

**Note:** Because *cfftInit* does not return FAIL status, which may be due to a fail in the memory allocation for private data structure, the user must check the returned pointer for “NULL”.

**Range Issues:** The length of the complex FFT input vector, *n*, must be set to one of the following values: 8, 16, 32, 64, 128, 256, 512, 1024, 2048.
Special Issues: The Twiddle Factor Table used by `cfft` function is allocated in `const.h` and `const.c` files. These files contain SDK-initialized constant data. For more information, see the SDK Programmers Guide, Chapter 6, Target Configuration.

- The `cfftInit` function must be called before any call to `cfft`. Multiple calls may be made to `cfftInit`, however, to initialize different `cfft` functions which could be used concurrently.
- The parameter `n` in the `cfftInit` function must be passed as a number, not through a variable or `#define`.
- Options `FFT_SCALE_RESULTS_BY_N` and `FFT_SCALE_RESULTS_BY_DATA_SIZE` are not allowed simultaneously. If both are selected, the `cfft` function returns inaccurate results without any indication. The following is a list of valid options:
  - FFT_DEFAULT_OPTIONS
  - FFT_DEFAULT_OPTIONS+FFT_INPUT_IS_BITREVERSED
  - FFT_DEFAULT_OPTIONS+FFT_OUTPUT_IS_BITREVERSED
  - FFT_DEFAULT_OPTIONS+FFT_INPUT_IS_BITREVERSED+FFT_OUTPUT_IS_BITREVERSED
  - FFT_SCALE_RESULTS_BY_N
  - FFT_SCALE_RESULTS_BY_N+FFT_INPUT_IS_BITREVERSED
  - FFT_SCALE_RESULTS_BY_N+FFT_OUTPUT_IS_BITREVERSED
  - FFT_SCALE_RESULTS_BY_N+FFT_INPUT_IS_BITREVERSED+FFT_OUTPUT_IS_BITREVERSED
  - FFT_SCALE_RESULTS_BY_DATA_SIZE
  - FFT_SCALE_RESULTS_BY_DATA_SIZE+FFT_INPUT_IS_BITREVERSED
  - FFT_SCALE_RESULTS_BY_DATA_SIZE+FFT_OUTPUT_IS_BITREVERSED
  - FFT_SCALE_RESULTS_BY_DATA_SIZE+FFT_INPUT_IS_BITREVERSED+FFT_OUTPUT_IS_BITREVERSED

Design/Implementation: None
Code Example 11-4. cfftInit

.....

#define MAX_CFFT_LEN 32

Cfrac16 FFTBuf[MAX_CFFT_LEN * sizeof(Cfrac16)];
Cfrac16 zCFFTBuf[32 * sizeof(Cfrac16)];

Int16 res;
(UInt16 options = FFT_SCALE_RESULTS_BY_N;
Cfrac16 *pX, *pZ;

/*Allocate cfft structure statically */
dfr16_tCFFTStruct CFFT;
dfr16_tCFFTStruct *pCFFT = &CFFT;

/* Call FFT init function */
dfr16CFFTInit (pCFFT, 32, options);

pX = FFTBuf;
pZ = zCFFTBuf;

........

/* here read fft input complex array(real/imaginary) into pX */

........
/* Call FFT C version */
res = dfr16CFFTC (pCFFT, &pX[0], &pZ[0]);
11.3.7 *cifft* - Complex Inverse Fast Fourier Transform

Call(s):

```
Result dfrl6CIFFT (dfrl6_tCFFTStruct *pCIFFT, CFrac16 *pX, CFrac16 *pZ);
```

Arguments:

<table>
<thead>
<tr>
<th>Table 11-11. <em>cifft</em> arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>pCIFFT</strong></td>
</tr>
<tr>
<td><strong>pX</strong></td>
</tr>
<tr>
<td><strong>pZ</strong></td>
</tr>
</tbody>
</table>

Description: Computes an in place, radix-2, complex, decimation-in-time (DIT), inverse Fast Fourier Transform function for a vector of complex fractional data values. Prior to any call to *cifft*, the function must be initialized via a call to *cifftCreate*. The function *cifft* uses the private data structure established by *cifftCreate* to maintain its state between function calls.

This algorithm can perform its operations in place/non-in place. If *pX* is equal to *pZ*, (in place computation), the contents of this input vector are destroyed as a result of the *cifft* function; the input vector is overlaid with the output results described in the next paragraph. If *pX* is not equal to *pZ*, (non-in place computation), the *cifft* algorithm is then performed in place using the output structure’s memory, leaving the input vector intact.

Options valid in the *options* parameter direct *cifft* algorithm operation; see Table 11-12 for details.

<table>
<thead>
<tr>
<th>Table 11-12. <em>cifft</em> options</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Options</strong></td>
</tr>
<tr>
<td>FFT_DEFAULT_OPTIONS</td>
</tr>
<tr>
<td>FFT_SCALE_RESULTS_BY_N</td>
</tr>
<tr>
<td>FFT_SCALE_RESULTS_BY_DATA_SIZE</td>
</tr>
<tr>
<td>FFT_INPUT_IS_BITREVERSED</td>
</tr>
<tr>
<td>FFT_OUTPUT_IS_BITREVERSED</td>
</tr>
</tbody>
</table>

Equation: See Algorithm section below

Returns: The function *cifft* returns either “FAIL” or the amount of scaling done:

-1: Indicates FAIL

(0 to \( \log_2 n \), both inclusive): Indicates that the result was scaled down by so many times

Note: Scaling/Scale down implies Right Shift; Scale up implies Left Shift.
Algorithm:
\[ z[i] = \text{scale} \sum_{k=0}^{n-1} x[k] \cdot (\cos(2 \cdot \pi \cdot i \cdot k / n) + j\sin(2 \cdot \pi \cdot i \cdot k / n)) \quad 0 \leq i < n \]

Range Issues:
- The complex input data values satisfy the following constraint in order to get proper results:
  \[ |x[k]| = ? |a_r + a_i| < 1 \]
  where \( a_r \) and \( a_i \) are the real and imaginary parts of input complex data point \( x[k] \).
- Scaling can be controlled as an input to the \textit{cifftCreate} call. If the \textit{options} parameter to \textit{cifftCreate} includes \texttt{FFT_DEFAULT_OPTIONS}, no scaling is performed; (i.e., \( \text{scale} = 1 \)). If \texttt{FFT_SCALE_RESULTS_BY_N}, (Auto scaling,) is selected, \textit{cifft} always returns:
  \[ \log_2 n \]
  indicating that the results are scaled by \( 1/n \); (n must be a power of 2). If \texttt{FFT_SCALE_RESULTS_BY_DATA_SIZE}, (block floating point scaling), is selected, \textit{cifft} returns \( S \), where:
  \[ 0 \leq S \leq \log_2 n \]
  indicating that the results are scaled down by \( 2^S \).
- Enabling or disabling the saturation bit before calling \textit{cifft} will not have any effect on the results. The \textit{cifft} function always disables the saturation bit internally, performs the calculations, and restores the saturation bit before returning. This is done to prevent intermediate saturation of results, which otherwise would give incorrect results at the end.

Special Issues:
- The assembly implementation of inverse FFT uses two-level nesting of the \textit{do loop}. If the user application (either a calling routine or an ISR) must call inverse FFT in a \textit{loop}, it is advised to use a software loop because the third do loop overhead is higher than that of the software loop.
- If \( S_1 \) and \( S_2 \) are the return values of \textit{cfft} and \textit{cifft} functions respectively, then the results after \textit{cifft} function must be:
  - scaled \textbf{up} by \( (S_1 + S_2) - n \) times, if \( S_1 + S_2 < n \), or
  - scaled \textbf{down} by \( n - (S_1 + S_2) \) times, if \( S_1 + S_2 < n \), where \( n \) is FFT/IFFT size

Design/Implementation: See the description of \textit{cifftCreate}, \textit{Section 11.3.8}, for details on the data structure allocation utilized for the \textit{cifft} function.
Code Example 11-5.  
cifft

{
    dfr16_tCFFTStruct *pCIFFT;
    Result        res;
    CFrac16       *pX, *pZ;
    UInt16        options = FFT_SCALE_RESULTS_BY_N;

    ...
    pCIFFT = dfr16CIFFTCreate (8, options); /* N = 8 point CIFFT */
    ...

    res = dfr16CIFFT (pCIFFT, pX, pZ);

    ...
    dfr16CIFFTDestroy (pCIFFT);
}
11.3.8  **cifftCreate** - Create Complex Inverse FFT

Call(s):

```c
dfrl6_tCFFTStruct *dfrl6CIFFTCreate (UInt16 n, UInt16 options);
```

Arguments:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>in</td>
<td>Length of the inverse complex FFT input and output vectors 8, 16, 32, 64, 128, 256, 512, 1024, or 2048</td>
</tr>
<tr>
<td>options</td>
<td>in</td>
<td>Option flags to control cifft algorithm operation</td>
</tr>
</tbody>
</table>

**Description:** Performs the initialization for an in place, radix-2, complex, decimation-in-time (DIT), inverse Fast Fourier Transform, (IFFT), using complex fractional data values. The function `cifftCreate` allocates and initializes a data structure, which is used by `cifft` to preserve the IFFT function’s state between calls.

Options valid in the `options` parameter direct the `cifft` algorithm operation; see **Table 11-14** for details.

<table>
<thead>
<tr>
<th>Options</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFT_DEFAULT_OPTIONS</td>
<td>If selected, <code>cifft</code> performs no scaling, assumes input to be in normal order, and writes output in normal order</td>
</tr>
<tr>
<td>FFT_SCALE_RESULTS_BY_N</td>
<td>If set, <code>cifft</code> will unconditionally scale down the results by n</td>
</tr>
<tr>
<td>FFT_SCALE_RESULTS_BY_DATA_SIZE</td>
<td>If set, <code>cifft</code> will conditionally scale down the results, depending upon the data sizes</td>
</tr>
<tr>
<td>FFT_INPUT_IS_BITREVERSED</td>
<td>If set, <code>cifft</code> assumes that the input vector is bit-reversed; otherwise, <code>cifft</code> assumes that the input vector is normal</td>
</tr>
<tr>
<td>FFT_OUTPUT_IS_BITREVERSED</td>
<td>If set, <code>cifft</code> will leave the output vector bit-reversed; otherwise, <code>cifft</code> leaves the output vector in normal order</td>
</tr>
</tbody>
</table>

**Equation:** None

**Returns:** The `cifftCreate` function returns a pointer to a private data structure, which must then be passed to subsequent calls of the `cifft` function.

**Note:** Because `cifftCreate` does not return FAIL status which may be due to a fail in the memory allocation for private data structure, the user must check the returned pointer for NULL.

**Range Issues:** The length of the real FFT output vector, `n`, must be set to one of the following values: 8, 16, 32, 64, 128, 256, 512, 1024, 2048.

**Special Issues:**

- `INCLUDE_MEMORY` must be defined in `appconfig.h` in order to initialize the Memory Manager which is used by the `cifftCreate` function.
- The function `cifftCreate` must be called before any call to `cifft` or `cifftDestroy`. Multiple calls may be made to `cifftCreate`, however, to initialize different `cifft` functions which could be used concurrently. Call `cifftDestroy` to deallocate the data structure allocated by `cifftCreate`.
The parameter \( n \) in the \texttt{cifftCreate} function must be passed as a number, and not through a variable or \#define.

Options \texttt{FFT\_SCALE\_RESULTS\_BY\_N} and \texttt{FFT\_SCALE\_RESULTS\_BY\_DATA\_SIZE} are not allowed simultaneously. If both are selected, the \texttt{cifft} function returns inaccurate results without any indication. The following is a list of \textbf{valid} options:

- \texttt{FFT\_DEFAULT\_OPTIONS}
- \texttt{FFT\_DEFAULT\_OPTIONS+FFT\_INPUT\_IS\_BITREVERSED}
- \texttt{FFT\_DEFAULT\_OPTIONS+FFT\_OUTPUT\_IS\_BITREVERSED}
- \texttt{FFT\_DEFAULT\_OPTIONS+FFT\_INPUT\_IS\_BITREVERSED+FFT\_OUTPUT\_IS\_BITREVERSED}
- \texttt{FFT\_SCALE\_RESULTS\_BY\_N}
- \texttt{FFT\_SCALE\_RESULTS\_BY\_N+FFT\_INPUT\_IS\_BITREVERSED}
- \texttt{FFT\_SCALE\_RESULTS\_BY\_N+FFT\_OUTPUT\_IS\_BITREVERSED}
- \texttt{FFT\_SCALE\_RESULTS\_BY\_N+FFT\_INPUT\_IS\_BITREVERSED+FFT\_OUTPUT\_IS\_BITREVERSED}
- \texttt{FFT\_SCALE\_RESULTS\_BY\_DATA\_SIZE}
- \texttt{FFT\_SCALE\_RESULTS\_BY\_DATA\_SIZE+FFT\_INPUT\_IS\_BITREVERSED}
- \texttt{FFT\_SCALE\_RESULTS\_BY\_DATA\_SIZE+FFT\_OUTPUT\_IS\_BITREVERSED}
- \texttt{FFT\_SCALE\_RESULTS\_BY\_DATA\_SIZE+FFT\_INPUT\_IS\_BITREVERSED+FFT\_OUTPUT\_IS\_BITREVERSED}

Design/Implementation: In general, the data structure allocated by \texttt{cifftCreate} is treated as private data to better encapsulate the details of the implementation. If you need to view the private data structure, refer to the actual implementation of \texttt{cifftCreate}. 

11.3.9  

**cifftDestroy - Destroy Complex Inverse FFT**

Call(s):

```
void dfr16CIFFTDestroy (dfr16_tCFFTStruct *pCIFFT);
```

Arguments:

<table>
<thead>
<tr>
<th>pCIFFT</th>
<th>in</th>
<th>Pointer to a data structure created by the <code>cifftCreate</code> function</th>
</tr>
</thead>
</table>

**Description:** The function `cifftDestroy` deallocates the data structure(s) initially allocated by `cifftCreate`.

**Equation:** None

**Returns:** void

**Range Issues:** None

**Special Issues:** The function `cifftDestroy` may be called only once to deallocate a data structure created by `cifftCreate`.

**Design/Implementation:** None
11.3.10 **cifftInit** - Initialize Complex Inverse FFT

Call(s):

```c
dfrl6CIFFTInit (dfrl6_tCFFTStruct * pCIFFT, n, UINT16 options);
```

Arguments:

<table>
<thead>
<tr>
<th>pCIFFT</th>
<th>in</th>
<th>Pointer to a data structure containing private data for the <code>cfft</code> function; this pointer is statically allocated by caller prior to the <code>dfrl6CIFFTInit</code> function</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>in</td>
<td>Length of the complex FFT input and output vectors: 8, 16, 32, 64, 128, 256, 512, 1024, 2048</td>
</tr>
<tr>
<td>options</td>
<td>in</td>
<td>Option flags to control <code>cfft</code> algorithm operation</td>
</tr>
</tbody>
</table>

**Description**: Performs the initialization for an in place, radix-2, complex, decimation-in-time (DIT), inverse Fast Fourier Transform (IFFT), using complex fractional data values. Initializes the function from a previously-allocated `dfrl6_tCFFTStruct` data structure. Typically, the `dfrl6_tCFFTStruct` data structure is allocated statically by the caller prior to the `cifftInit` call. The `cifftCreate` function also uses `cifftInit` to initialize allocate the `dfrl6_tCFFTStruct` dynamically.

Options valid in the `options` parameter direct `cfft` algorithm operation; see Table 11-17 for details.

<table>
<thead>
<tr>
<th>Options</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFT_DEFAULT_OPTIONS</td>
<td>If selected, <code>cfft</code> performs no scaling, assumes input to be in normal order, and writes output in normal order</td>
</tr>
<tr>
<td>FFT_SCALE_RESULTS_BY_N</td>
<td>If set, <code>cfft</code> will unconditionally scale down the results by <code>n</code></td>
</tr>
<tr>
<td>FFT_SCALE_RESULTS_BY_DATA_SIZE</td>
<td>If set, <code>cfft</code> will conditionally scale down the results, depending upon the data sizes</td>
</tr>
<tr>
<td>FFT_INPUT_IS_BITREVERSED</td>
<td>If set, <code>cfft</code> assumes that the input vector is bit-reversed; otherwise, <code>cfft</code> assumes that the input vector is normal</td>
</tr>
<tr>
<td>FFT_OUTPUT_IS_BITREVERSED</td>
<td>If set, <code>cfft</code> will cifft leave the output vector bit-reversed; otherwise, <code>cfft</code> leaves the output vector in normal order</td>
</tr>
</tbody>
</table>

**Equation**: None

**Returns**: None

**Note**: Because `cifftInit` does **not** return FAIL status which may be due to a fail in the memory allocation for private data structure, the user must check the returned pointer for “NULL”.

**Range Issues**: The length of the complex FFT input vector, `n`, must be set to one of the following values: 8, 16, 32, 64, 128, 256, 512, 1024, 2048.
Special Issues: The Twiddle Factor Table used by the `cifft` function is allocated in the `const.h` and `const.c` files, which contain SDK-initialized constant data. For more information, see the SDK Programmers Guide, Chapter 6, Target Configuration.

- The `cifftInit` function must be called before any call to `cifft`. Multiple calls may be made to `cifftInit`, however, to initialize different `cifft` functions which could be used concurrently.
- The parameter `n` in the `cifftInit` function must be passed as a number, and not through a variable or `#define`.
- Options `FFT_SCALE_RESULTS_BY_N` and `FFT_SCALE_RESULTS_BY_DATA_SIZE` are not allowed simultaneously. If both are selected, the `cifft` function returns inaccurate results without any indication. The following is a list of valid options:
  - `FFT_DEFAULT_OPTIONS`
  - `FFT_DEFAULT_OPTIONS+FFT_INPUT_IS_BITREVERSED`
  - `FFT_DEFAULT_OPTIONS+FFT_OUTPUT_IS_BITREVERSED`
  - `FFT_DEFAULT_OPTIONS+FFT_INPUT_IS_BITREVERSED+FFT_OUTPUT_IS_BITREVERSED`
  - `FFT_SCALE_RESULTS_BY_N`
  - `FFT_SCALE_RESULTS_BY_N+FFT_INPUT_IS_BITREVERSED`
  - `FFT_SCALE_RESULTS_BY_N+FFT_OUTPUT_IS_BITREVERSED`
  - `FFT_SCALE_RESULTS_BY_N+FFT_INPUT_IS_BITREVERSED+FFT_OUTPUT_IS_BITREVERSED`
  - Options `FFT_SCALE_RESULTS_BY_DATA_SIZE` and `FFT_SCALE_RESULTS_BY_DATA_SIZE+FFT_INPUT_IS_BITREVERSED` are not allowed simultaneously. If both are selected, the `cifft` function returns inaccurate results without any indication. The following is a list of valid options:
    - `FFT_SCALE_RESULTS_BY_DATA_SIZE`
    - `FFT_SCALE_RESULTS_BY_DATA_SIZE+FFT_INPUT_IS_BITREVERSED`
    - `FFT_SCALE_RESULTS_BY_DATA_SIZE+FFT_OUTPUT_IS_BITREVERSED`
    - `FFT_SCALE_RESULTS_BY_DATA_SIZE+FFT_INPUT_IS_BITREVERSED+FFT_OUTPUT_IS_BITREVERSED`

Design/Implementation: None

Code Example 11-6. `cifftInit`

```c
#define MAX_CFFT_LEN 32

CFrac16 CIFFTBuf[MAX_CFFT_LEN * sizeof(CFrac16)];
CFrac16 zCIFFTBuf[32 * sizeof(CFrac16)];

Int16 res;

UInt16 options = FFT_SCALE_RESULTS_BY_N;
CFrac16 *pX, *pZ;
```
/* Allocate cifft structure statically */
dfr16_tCFFTStruct CIFFT;
dfr16_tCFFTStruct *pCIFFT = &CIFFT;

/* Call FFT init function */
dfr16CIFFTInit (pCIFFT, 32, options);

pX = CIFFTBuf;
pZ = zCIFFTBuf;

........

/* here read Inverset fft input complex array into pX */
........

/* Call Inverset FFT C version */
res = dfr16CIFFTC (pCIFFT, &pX[0], &pZ[0]);
11.3.11 rfft - Real Fast Fourier Transform

Call(s):

typedef struct {
    Frac16   z0; /* z[0]               (real 16-bit fractional) */
    Frac16   zNDiv2; /* z[n/2]            (real 16-bit fractional) */
    CFrac16  cz[1]; /* z[1] .. z[n/2 - 1] (complex 16-bit fractional) */
} dfr16_sInplaceCRFFT;

Note: The remaining (n/2-2) locations for cz[] buffer must be allocated by the user.

Result dfr16RFFT (dfr16_tRFFTStruct * pRFFT, Frac16 *pX, dfr16_sInplaceCRFFT *pZ);

Arguments:

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pRFFT</td>
<td>Pointer to a data structure containing private data for the rfft function; created by a call to rfftCreate</td>
</tr>
<tr>
<td>pX</td>
<td>Pointer to the input vector of n fractional data elements, stored in normal, as opposed to bit-reversed, order</td>
</tr>
<tr>
<td>pZ</td>
<td>Pointer to the output structure of complex values</td>
</tr>
</tbody>
</table>

Description: Computes an in-place, radix-2, real, decimation-in-time (DIT) forward Fast Fourier Transform function for a vector of fractional data values. Prior to any call to rfft, the function must be initialized via a call to rfftCreate. The function rfft uses the private data structure established by rfftCreate to maintain its state between function calls.

This algorithm performs its operations in place/non-in place. If pX is equal to pZ, (in place computation), the contents of this input vector are destroyed as a result of the rfft function; the input vector is overlaid with the output results described in the next paragraph. If pX is not equal to pZ, (non-inplace computation), the rfft algorithm is then performed in place, using the output structure’s memory without affecting the contents of pX.

The elements of the input vector, pointed to by pX, must always be stored in normal order.

Valid options in the options parameter direct rfft algorithm operation; see Table 11-19 for details.

<table>
<thead>
<tr>
<th>Options</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFT_DEFAULT_OPTIONS</td>
<td>If selected, rfft performs no scaling, assumes input to be in normal order, and writes output in normal order</td>
</tr>
<tr>
<td>FFT_SCALE_RESULTS_BY_N</td>
<td>If set, rfft will unconditionally scale down the results by n</td>
</tr>
<tr>
<td>FFT_SCALE_RESULTS_BY_DATA_SIZE</td>
<td>If set, rfft will conditionally scale down the results, depending upon the data sizes</td>
</tr>
</tbody>
</table>

Note: The options FFT_INPUT_IS_BITREVERSED and FFT_OUTPUT_IS_BITREVERSED are invalid. Also, the combination of options FFT_SCALE_RESULTS_BY_N and FFT_SCALE_RESULTS_BY_DATA_SIZE is invalid, and setting these options will lead to inaccurate results.
**Equation:** See the following Algorithm.

**Returns:** The function `rfft` returns either “FAIL”, or the number of stages of FFT scaled.

- **-1:** Indicates FAIL

  **(0 to log₂n, both inclusive):** Indicates that the result was scaled down by so many times

**Note:** Scaling/Scale down implies Right Shift; Scale up implies Left Shift.

**Algorithm:**

\[
z[k] = \text{scale} \sum_{i=0}^{n-1} x[i] \left( \cos\left(2 \cdot \pi \cdot i \cdot k / n\right) \right) - j \sin\left(2 \cdot \pi \cdot i \cdot k / n\right))
\]

**Range Issues:**

- The real input data values satisfy the following constraint in order to get proper results:

  \[
  (x[i])^2 + (x[i + 1])^2 < 1
  \]

- Scaling can be controlled as an input to the `rfftCreate` call. If the `options` parameter to `rfftCreate` includes `FFT_DEFAULT_OPTIONS`, no scaling is performed; (i.e., `scale = 1`). If `FFT_SCALE_RESULTS_BY_N`, (Auto scaling), is selected, `rfft` always returns `log₂n`, indicating that the results are scaled by `1/n`; (n must be a power of 2). If `FFT_SCALE_RESULTS_BY_DATA_SIZE`, (block floating point scaling) is selected, `rfft` returns `S`, where `0 < S < log₂n`, indicating that the results are scaled down by `2^S`.

- Enabling or disabling the saturation bit before calling `rfft` will not have any effect on the results. The `rfft` function always disables the saturation bit internally, performs the calculations, and restores the saturation bit before returning. This is done to prevent intermediate saturation of results, which otherwise would give inaccurate results at the end.

**Special Issues:**

- Note that the `rfft` function relies on the fact that an FFT of a real sequence has hermitian symmetry around the fold-over frequency. Therefore, the `rfft` function only returns `n / 2` complex output values.

- The assembly implementation of real FFT uses two-level nesting of `do loop`. If the user application (either a calling routine or an ISR) must call real FFT in a `loop`, it is advised to use a software loop because the third do loop overhead is higher than that of the software loop.

- See Special Issues, Section 11.3.32.

**Design/Implementation:** See the description of `rfftCreate` for details on the data structure allocation utilized for the `rfft` function.

**Code Example 11-7. rfft**

```c
{
    dfr16_tRFFTStruct *pRFFT;
    Result res;
    Frac16 *pX;
    dfr16_sInplaceCRFFT *pZ;
    Uint16 options = FFT_SCALE_RESULTS_BY_N;
}```
... 
pRFFT = dfr16RFFTCreate (8, options); /* N = 8 point RFFT */ 
...

res = dfr16RFFT (pRFFT, pX, pZ);

...
dfr16RFTTDestroy (pRFFT);
11.3.12 \textit{rfftCreate} - Create Real FFT

\textbf{Call(s):}

\begin{verbatim}
dfr16_tRFFTStruct * dfr16RFFTCreate   (UInt16 n, UInt16 options);
\end{verbatim}

\textbf{Arguments:}

\begin{table}[h!]
\centering
\begin{tabular}{|c|c|}
\hline
\textbf{n} & in \text{Length of the real FFT input vector} \tabularnewline
8,16, 32, 64, 128, 256, 512, 1024, or 2048 \tabularnewline
\hline
\textbf{options} & in \text{Option flags to control rfft algorithm operation} \tabularnewline
\hline
\end{tabular}
\caption{\textit{rfftCreate}}
\end{table}

\textbf{Description:} Performs the initialization for an in place, radix-2, real, decimation-in-time, (DIT), forward Fast Fourier Transform, (FFT), function using fractional data values. The function \textit{rfftCreate} allocates and initializes a data structure, which is used by \textit{rfft} to preserve the FFT function’s state between calls.

Options valid in the \textit{options} parameter direct \textit{rfft} algorithm operation; see Table 11-21 for details.

\begin{table}[h!]
\centering
\begin{tabular}{|c|c|}
\hline
\textbf{Options} & \textbf{Function} \tabularnewline
\hline
FFT_DEFAULT_OPTIONS & If selected, \textit{rfft} performs no scaling, assumes input to be in normal order, and writes output in normal order \tabularnewline
\hline
FFT_SCALE_RESULTS_BY_N & If set, \textit{rfft} will unconditionally scale down the results by n \tabularnewline
\hline
FFT_SCALE_RESULTS_BY_DATA_SIZE & If set, \textit{rfft} will conditionally scale down the results, depending upon the data sizes \tabularnewline
\hline
\end{tabular}
\caption{\textit{rfftCreate} options}
\end{table}

\textbf{Equation:} None

\textbf{Returns:} The \textit{rfftCreate} function returns a pointer to a private data structure, which must then be passed to subsequent calls of the \textit{rfft} function.

\textbf{Note:} Because the \textit{rfftCreate} function does not return “FAIL” status, which may be due to a fail in the memory allocation for private data structure, the user must check the returned pointer for “NULL”.

\textbf{Range Issues:} The length of the real FFT input vector, \textit{n}, must be set to one of the following values: 8,16, 32, 64, 128, 256, 512, 1024, 2048.

\textbf{Special Issues:}

\begin{itemize}
\item \textit{INCLUDE_MEMORY} must be defined in \textit{appconfig.h} in order to initialize the Memory Manager, which is used by the \textit{rfftCreate} function.
\item The function \textit{rfftCreate} must be called before any call to \textit{rfft} or \textit{rfftDestroy}. Multiple calls may be made to \textit{rfftCreate}, however, to initialize different \textit{rfft} functions which could be used concurrently. Call \textit{rfftDestroy} to deallocate the data structure allocated by \textit{rfftCreate}.
\item The parameter \textit{n} in the \textit{rfftCreate} function must be passed as a number, and not through a variable or \#define.
\end{itemize}
Design/Implementation: In general, the data structure allocated by `rfftCreate` is treated as private data to better encapsulate the details of the implementation. However, please refer to the actual implementation of `rfftCreate` to review the private data structure.
11.3.13  \textit{rfftDestroy} - Destroy Real FFT

Call(s):

\begin{verbatim}
void dfrl6RFFTDestroy (dfrl6_tRFFTStruct * pRFFT);
\end{verbatim}

Arguments:

\begin{table}[h]
\centering
\begin{tabular}{|c|c|}
\hline
\textit{pRFFT} & in & Pointer to a data structure created by the \textit{rfftCreate} function \\
\hline
\end{tabular}
\caption{\textit{rfftDestroy} arguments}
\end{table}

Description: The \textit{rfftDestroy} function deallocates the data structure(s) initially allocated by \textit{rfftCreate}.

Equation: None

Returns: Void

Range Issues: None

Special Issues: The function \textit{rfftDestroy} may be called only once to deallocate a data structure created by \textit{rfftCreate}.

Design/Implementation: None
11.3.14  

**rfftInit - Initialize Real FFT**

**Call(s):**

```c
df16rFFTInit (df16_tRFFTStruct * pRFFT, n, UInt16 options);
```

**Arguments:**

<table>
<thead>
<tr>
<th>pRFFT</th>
<th>in</th>
<th>Pointer to a data structure containing private data for the cfft function; this pointer is statically created by caller prior to the df16CFFTInit function</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>in</td>
<td>Length of the complex FFT input and output vectors: 8, 16, 32, 64, 128, 256, 512, 1024, 2048</td>
</tr>
<tr>
<td>options</td>
<td>in</td>
<td>Option flags to control rfft algorithm operation</td>
</tr>
</tbody>
</table>

**Description:** Performs the initialization for an in place, radix-2, real, decimation-in-time (DIT), forward Fast Fourier Transform (FFT) function, rfft, using complex fractional data values. Initializes the function from a previously allocated df16_tRFFTStruct data structure. Typically, the df16_tRFFTStruct data structure is allocated statically by the caller prior to the rfftInit call. The rfftCreate function also uses rfftInit to initialize the df16_tRFFTStruct that it dynamically allocates.

Options valid in the options parameter direct rfft algorithm operation; see Table 11-24 for details.

**Table 11-24.  rfftInit options**

<table>
<thead>
<tr>
<th>Options</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFT_DEFAULT_OPTIONS</td>
<td>If selected, rfft performs no scaling, assumes input to be in normal order, and writes output in normal order</td>
</tr>
<tr>
<td>FFT_SCALE_RESULTS_BY_N</td>
<td>If set, rfft will unconditionally scale down the results by n</td>
</tr>
<tr>
<td>FFT_SCALE_RESULTS_BY_DATA_SIZE</td>
<td>If set, rfft will conditionally scale down the results, depending upon the data sizes</td>
</tr>
</tbody>
</table>

**Equation:** None

**Returns:** None

**Note:** Because rfftInit does not return FAIL status which may be due to a fail in the memory allocation for private data structure, the user must check the returned pointer for “NULL”.

**Range Issues:** The length of the complex FFT input vector, n, must be set to one of the following values: 8, 16, 32, 64, 128, 256, 512, 1024, 2048.

**Special Issues:** The Twiddle Factor Table used by the rfft function is allocated in the *const.h and const.c* files, which contain SDK-initialized constant data. For more information, see the SDK Programmers Guide, Chapter 6, Target Configuration.

- The rfftInit function must be called before any call to rfft. Multiple calls may be made to rfftInit, however, to initialize different rfft functions which could be used concurrently.
- The parameter n in the rfftInit function must be passed as a number, and not through a variable or #define.
Options FFT_SCALE_RESULTS_BY_N and FFT_SCALE_RESULTS_BY_DATA_SIZE are not allowed simultaneously. If both are selected, the rfft function returns inaccurate results without any indication. The following is a list of valid options:

- FFT_DEFAULT_OPTIONS
- FFT_DEFAULT_OPTIONS+FFT_INPUT_IS_BITREVERSED
- FFT_DEFAULT_OPTIONS+FFT_OUTPUT_IS_BITREVERSED
- FFT_DEFAULT_OPTIONS+FFT_INPUT_IS_BITREVERSED+FFT_OUTPUT_IS_BITREVERSED
- FFT_SCALE_RESULTS_BY_N
- FFT_SCALE_RESULTS_BY_N+FFT_INPUT_IS_BITREVERSED
- FFT_SCALE_RESULTS_BY_N+FFT_OUTPUT_IS_BITREVERSED
- FFT_SCALE_RESULTS_BY_N+FFT_INPUT_IS_BITREVERSED+FFT_OUTPUT_IS_BITREVERSED
- FFT_SCALE_RESULTS_BY_DATA_SIZE
- FFT_SCALE_RESULTS_BY_DATA_SIZE+FFT_INPUT_IS_BITREVERSED
- FFT_SCALE_RESULTS_BY_DATA_SIZE+FFT_OUTPUT_IS_BITREVERSED
- FFT_SCALE_RESULTS_BY_DATA_SIZE+FFT_INPUT_IS_BITREVERSED+FFT_OUTPUT_IS_BITREVERSED

Design/Implementation: None
11.3.15 rifft - Real Inverse Fast Fourier Transform

Call(s):

```c
typedef struct {
    Frac16 z0; /* z[0] (real 16-bit fractional) */
    Frac16 zNDiv2; /* z[n/2] (real 16-bit fractional) */
    CFrac16 cz[1]; /* z[1] .. z[n/2 - 1] (complex 16-bit fractional)*/
} dfr16_sInplaceCRFFT;
```

Note: The remaining (n/2-2) locations for cz[] buffer must be allocated by the user.

Result: dfr16RIFFT (dfr16_tRFFTStruct * pRIFFT, dfr16_sInplaceCRFFT *pX, Frac16 *pZ);

Arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pRIFFT</td>
<td>Pointer to a data structure containing private data for the rifft function; created by a call to rifftCreate</td>
</tr>
<tr>
<td>pX</td>
<td>Pointer to the input structure of complex values stored in normal, rather than bit-reversed, order</td>
</tr>
<tr>
<td>pZ</td>
<td>Pointer to the output vector of real values stored in normal order</td>
</tr>
</tbody>
</table>

Description: Computes an in place, radix-2, decimation-in-time, (DIT), inverse Fast Fourier Transform function for a structure of complex data values. Prior to any call to rifft, the function must be initialized via a call to rifftCreate. The function rifft uses the private data structure established by rifftCreate to maintain its state between function calls.

This algorithm performs its operations in place/non-in place. If pX is equal to pZ, (in place computation), the contents of this input structure are destroyed as a result of the rifft function; the input structure is overlaid with the output results. If pX is not equal to pZ, (non-in place computation), the rifft algorithm is then performed in place, using the output structure’s memory without affecting the contents of pX.

Options valid in the options parameter direct rifft algorithm operation; see Table 11-26 for details.

<table>
<thead>
<tr>
<th>Options</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFT_DEFAULT_OPTIONS</td>
<td>If selected, rifft performs no scaling, assumes input to be in normal order, and writes output in normal order</td>
</tr>
<tr>
<td>FFT_SCALE_RESULTS_BY_N</td>
<td>If set, rifft will unconditionally scale down the results by n</td>
</tr>
<tr>
<td>FFT_SCALE_RESULTS_BY_DATA_SIZE</td>
<td>If set, rifft will conditionally scale down the results, depending upon the data sizes</td>
</tr>
</tbody>
</table>

Note: The options FFT_INPUT_IS_BITREVERSED and FFT_OUTPUT_IS_BITREVERSED are invalid. Also, the combination of options FFT_SCALE_RESULTS_BY_N and...
FFT_SCALE_RESULTS_BY_DATA_SIZE is invalid, and setting these options will lead to inaccurate results.

**Equation:** See Algorithm in this section

**Returns:** The function rifft returns either “FAIL” or the number of stages of FFT scaled.

-1: Indicates “FAIL”

(0 to log₂n, both inclusive): Indicates that the result was scaled down by so many times

**Note:** Scaling/Scale down implies Right Shift; Scale up implies Left Shift.

**Algorithm:**

\[
z[i] = \text{scale} \sum_{k=0}^{n-1} x[k] \cdot (\cos(2 \pi i k / n) + jsin(2 \pi i k / n)) \quad 0 \leq i < n
\]

**Range Issues:** The real input data values satisfy the following constraint in order to get proper results:

\[|x[k]| < 1\] , where \(x[k]\) is complex

- Scaling can be controlled as an input to the rifftCreate call. If the options parameter to rifftCreate includes FFT_DEFAULT_OPTIONS, no scaling is performed; (i.e., scale = 1). If FFT_SCALE_RESULTS_BY_N, (Auto scaling), is selected, rifft always returns log₂n, indicating that the results are scaled down by n; (n must be a power of 2). If FFT_SCALE_RESULTS_BY_DATA_SIZE, (block floating point scaling), is selected, rifft returns \(S\), where \(0 \leq S \leq \log_2 n\), indicating that the results are scaled down by \(2^S\).

- Enabling or disabling the saturation bit before calling rifft will not have any effect on the results. The rifft function always disables the saturation bit internally, performs the calculations, and restores the saturation bit before returning. This is done to prevent intermediate saturation of results, which otherwise would yield inaccurate results at the end.

**Special Issues:**

- Note that the rifft function relies on the fact that an FFT of a real sequence has even symmetry around its center. Therefore, the rifft function uses only \(n / 2\), complex input values, to generate \(n\), real output values.

- The assembly implementation of inverse real FFT uses two-level nesting of the do loop. If the user application (either a calling routine or an ISR) must call inverse real FFT in a loop, it is advised to use a software loop because the third do loop overhead is higher than that of the software loop.

- If \(S_1\) and \(S_2\) are the return values of the rfft and rifft functions respectively, then the results after the rifft function must be:
  - scaled up by \((S_1 + S_2) - n\) times if \(S_1 + S_2 > n\), or
  - scaled down by \(n - (S_1 + S_2)\) times if \(S_1 + S_2 < n\), where \(n\) is FFT/IFFT size

**Design/Implementation:** See the description of rifftCreate, Section 11.3.16, for details on the data structure allocation utilized for the rifft function.
Code Example 11-8.  *rifft*

```
{
    dfr16_tRFFTStruct *pRFFT;
    Result        res;
    Frac16       *pZ;
    dfr16_sInplaceCRFFT *pX;
    UInt16      options = FFT_SCALE_RESULTS_BY_N;

    ...
    pRFFT = dfr16RFFTCreate (8, options); /* N = 8 point RFFT */
    ...
    res = dfr16RFFT (pRFFT, pX, pZ);
    ...
    dfr16RFFTDestroy (pRFFT);
}
```
11.3.16  **riffCreate** - Create Real Inverse FFT

**Call(s):**

```c
dfrl6_tRFFTStruct * dfrl6RIFFTCreate (UInt16 n, UInt16 options);
```

**Arguments:**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>in</td>
<td>Length of the inverse real FFT output vector 8,16, 32, 64, 128, 256, 512,</td>
</tr>
<tr>
<td>options</td>
<td>in</td>
<td>Option flags to control <strong>riff</strong> algorithm operation</td>
</tr>
</tbody>
</table>

**Description:** Performs the initialization for an in place, inverse real Fast Fourier Transform (IFFT), function using fractional data values. The function **riffCreate** allocates and initializes a data structure, which is used by **riff** to preserve the FFT function’s state between calls.

Options valid in the **options** parameter direct **riff** algorithm operation as seen in **Table 11-28**.

**Table 11-28.  **riffCreate** options**

<table>
<thead>
<tr>
<th>Options</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFT_DEFAULT_OPTIONS</td>
<td>If selected, <strong>riff</strong> performs no scaling, assumes input to be in normal order, and writes output in normal order</td>
</tr>
<tr>
<td>FFT_SCALE_RESULTS_BY_N</td>
<td>If set, <strong>riff</strong> will unconditionally scale down the results by <em>n</em></td>
</tr>
<tr>
<td>FFT_SCALE_RESULTS_BY_DATA_SIZE</td>
<td>If set, <strong>riff</strong> will conditionally scale down the results, depending upon the data sizes</td>
</tr>
</tbody>
</table>

**Equation:** None

**Returns:** The **riffCreate** function returns a pointer to a private data structure, which must then be passed to subsequent calls of the **riff** function.

**Note:** Because **riffCreate** does not return “FAIL” status which may be due to a fail in the memory allocation for private data structure, the user must check the returned pointer for “NULL”.

**Range Issues:** The length of the real FFT output vector, *n*, must be set to one of the following values: 8,16, 32, 64, 128, 256, 512, 1024, 2048.

**Special Issues:**

- **INCLUDE_MEMORY** must be defined in **appconfig.h** in order to initialize the Memory Manager, which is used by the **riffCreate** function.
- The function **riffCreate** must be called before any call to **riff** or **riffDestroy**. Multiple calls may be made to **riffCreate**, however, to initialize different **riff** functions which could be used concurrently. Call **riffDestroy** to deallocate the data structure allocated by **riffCreate**.
- The parameter *n* in the **riffCreate** function must be passed as a number, and not through a variable or #define.

**Design/Implementation:** In general, the data structure allocated by **riffCreate** is treated as private data to better encapsulate the details of the implementation. However, please refer to the actual implementation of **riffCreate** to review the private data structure.
11.3.17 rifftDestroy - Destroy Real Inverse FFT

Call(s):

```c
void dfr16RIFFTDestroy (dfr16_tRFFTStruct * pRIFFT);
```

Arguments:

<table>
<thead>
<tr>
<th>pRIFFT</th>
<th>in</th>
<th>Pointer to a data structure created by the rifftCreate function</th>
</tr>
</thead>
</table>

Description: The function rifftDestroy deallocates the data structure(s) initially allocated by rifftCreate.

Equation: None

Returns: Void

Range Issues: None

Special Issues: The rifftDestroy function may be called only once to deallocate a data structure created by rifftCreate.

Design/Implementation: None
11.3.18 corr - Correlation

Call(s):

```c
#define CORR_RAW 0  /* Select Raw correlation  */
#define CORR_BIAS 1 /* Select Bias correlation */
#define CORR_UNBIAS 2 /* Select Unbias correlation */
```

Result: `dfr16Corr (UInt16 options, Frac16 *pX, Frac16 *pY, Frac16 *pZ, UInt16 nx, UInt16 ny);`

Arguments:

<table>
<thead>
<tr>
<th>options</th>
<th>in</th>
<th>Selects between raw, biased, and unbiased correlation</th>
</tr>
</thead>
<tbody>
<tr>
<td>pX</td>
<td>in</td>
<td>Pointer to the first input vector of nx data elements</td>
</tr>
<tr>
<td>pY</td>
<td>in</td>
<td>Pointer to the second input vector of ny data elements</td>
</tr>
<tr>
<td>pZ</td>
<td>in/out</td>
<td>Pointer to the output vector of (nx + ny -1) data elements</td>
</tr>
<tr>
<td>nx</td>
<td>in</td>
<td>Length of the input vector pointed to by pX</td>
</tr>
<tr>
<td>ny</td>
<td>in</td>
<td>Length of the input vector pointed to by pY</td>
</tr>
</tbody>
</table>

Description: Computes the full length correlation of two vectors of fractional data values.

Three different types of full correlation may be selected via the `options` parameter: raw, biased, and unbiased. One of the three values shown in Table 11-31 must be passed in the `options` parameter to `corr`.

Equation: See Algorithm section below.

<table>
<thead>
<tr>
<th>Options</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>CORR_RAW</td>
<td>If set, the correlation type will be raw; i.e., bias = 1 in the following correlation algorithm</td>
</tr>
<tr>
<td>CORR_BIAS</td>
<td>If set, the correlation type will be biased; i.e., bias = (1 / (nx + ny -1)) in the following correlation algorithm</td>
</tr>
<tr>
<td>CORR_UNBIAS</td>
<td>If set, the correlation type will be unbiased; i.e., bias = (1 / (nx + ny -1 - j)) in the following correlation algorithm</td>
</tr>
</tbody>
</table>

Returns: The `corr` computation generates (nx + ny - 1) output values which are stored in the vector pointed to by pZ.

The function `corr` returns:

- **FAIL (“-1”)**: if the length of the output vector, nx + ny - 1, exceeds 8192.
- **PASS (“0”)**: otherwise
Algorithm:

\[ z[j] = \text{bias} \sum_{k=0}^{lpcnt} x[j + k] y[k], \text{nx} - 1 \geq j > 0 \]

where, \( lpcnt = \min ((\text{nx}-(j + 1)), (\min (\text{nx}, \text{ny}) - 1)) \)

\[ z[j] = \text{bias} \sum_{k=0}^{lpcnt} y[j + k] x[k], 0 \leq j < \text{ny} \]

where, \( lpcnt = \min ((\text{nx}-(j + 1)), (\min (\text{nx}, \text{ny}) - 1)) \)

**Range Issues:** If saturation is enabled, the \( \text{corr} \) function will return the maximum or minimum fractional values if overflow or underflow occurs during the calculation of each output element.

The length of the output vector, \( \text{nx} + \text{ny} - 1 \), must be within the range \( 0 \leq \text{nx} + \text{ny} - 1 \leq 8192 \).

**Special Issues:** In place computation is not allowed; i.e., the addresses of the input vectors, \( pX \) and \( pY \), must be different from the address of the output vector, \( pZ \).

**Design/Implementation:** None

**Code Example 11-9. \( \text{corr} \)**

```c
{
    Result res;
    UInt16 options;
    Frac16 *pX;
    Frac16 *pY;
    Frac16 *pZ;
    UInt16 nx;
    UInt16 ny;
    ...

    res = dfr16Corr (options, pX, pY, pZ, nx, ny);
    ...
}
```
11.3.19  \textit{fir} - Finite Impulse Response Filter

Call(s):

```c
void dfir16FIR (dfir16_tFirStruct *pFIR, Frac16 *pX, Frac16 *pZ, U16 n);
```

Arguments:

<table>
<thead>
<tr>
<th>pFIR</th>
<th>in</th>
<th>Pointer to a data structure containing private data for the \textit{fir} filter; created by a call to \textit{firCreate}</th>
</tr>
</thead>
<tbody>
<tr>
<td>pX</td>
<td>in</td>
<td>Pointer to the input vector of (n) data elements</td>
</tr>
<tr>
<td>pZ</td>
<td>inout</td>
<td>Pointer to the output vector of (n) data elements</td>
</tr>
<tr>
<td>(n)</td>
<td>in</td>
<td>Length of the input and output vectors</td>
</tr>
</tbody>
</table>

\textbf{Description:} Computes a Finite Impulse Response, (FIR), filter for a vector of fractional data values. Prior to any call to \textit{fir}, the FIR filter must be initialized via a call to \textit{firCreate}; the FIR filter uses coefficients passed to that \textit{firCreate} call. The function \textit{fir} uses the private data structure established by \textit{firCreate} to maintain the past history of data elements required by the FIR filter computation.

\textbf{Equation:} See Algorithm in this section.

\textbf{Returns:} The FIR filter computation generates \(n\) output values, which are stored in the vector pointed to by \(pZ\).

\textbf{Algorithm:}

\[
z[j] = \sum_{k=0}^{nc-1} c[k] \times [j - k] \quad 0 \leq j < n
\]

\textbf{Range Issues:} If saturation is enabled, the \textit{fir} function will return the maximum or minimum fractional values if overflow or underflow occurs during the calculation of each output element.

The length of the input and output vectors, \(n\), must be within the range \(0 \leq n \leq \text{MAX VECTOR LEN}\).

\textbf{Special Issues:} The \textit{fir} function can be used for block filtering where the length of the input vector, \(n\), is greater than 1, or for single samples where \(n = 1\). (Note, however, that the function \textit{firs} is provided for filtering single samples; see \textbf{Section 11.3.20}.) In place computation is allowed; i.e., the addresses of the output vector, \(pZ\) can be equal to the address of the input vector, \(pX\).

\textbf{Design/Implementation:} Modulo addressing is utilized in the implementation to optimize the performance of the FIR filter. See the description of \textit{firCreate} for details on the data structure allocation utilized for modulo addressing.

\textbf{Performance:} The call to \textit{firCreate} attempts to allocate buffers such that dual-parallel moves using modulo addressing can be used. However, if insufficient internal memory is available to optimally allocate the buffers, degraded \textit{fir} performance for block filtering may result.
Code Example 11-10 shows the usage of `dfr16FIR` function. The example implements a 37 tap low-pass filter with the following specifications:

PassBand Edge Frequency, \(W_p = 0.4\pi\) rad/sample

StopBand Edge Frequency, \(W_s = 0.6\pi\) rad/sample

Peak Error in PassBand = Peak Error in StopBand = 0.001

A Kaiser window is used with Beta = 5.653

---

**Code Example 11-10. fir**

```
#define FIR_COEF_LENGTH 37  /*Number of filter taps */

#define NUM_SAMPLES 100 /*Number of incoming samples */

/* FIR filter specifications
PassBand Edge Frequency, \(W_p = 0.4\pi\)
StopBand Edge Frequency, \(W_s = 0.6\pi\)
Peak Error in PassBand = Peak Error in StopBand = 0.001
Kaiser window is used
Beta = 5.653
M = 37
Alpha = M/2 = 18.5*/

const Frac16 FirCoefs[] =
{
    0xffff8,
    0x0012,
    0x0020,
    0xffcc,
    0xffb0,
    0x0075,
    0x00a4,
    0xff20,
    0xfed5,
    0x0189,
    0x01fe,
    0xfd6e,
    0xfcb1,
    0x0449,
    0x05a2,
    0xf856,
    0xf4d6,
    0x130e,
    0x399f,
    0x392a,
    0x129b,
    0xf546,
    0xf8c2,
    0x053c,
    0x03e9,
    0xfd09,
};```
0xfdbe,
0x01b7,
0x014a,
0xff0b,
0xff4e,
0x007e,
0x0056,
0xffc8,
0xffde,
0x0013,
0x0009);

dfr16_tFirStruct * pFir;
    Frac16        * pFirCoefs
    Frac16        x[NUM_SAMPLES];
    Frac16        z[NUM_SAMPLES];

    pFirCoefs=(Frac16 *)(&FirCoefs[0])

    pFir = dfr16FIRCreate (pFirCoefs, FIR_COEF_LENGTH);

    dfr16FIR (pFir, x, z, NUM_SAMPLES);
    ...

    dfr16FIRDestroy (pFir);

}
11.3.20  

`srs` - FIR Filter for a Single Input

Call(s):

```c
Frac16 dfr16FIRs (dfr16_tFirStruct *pFIR, Frac16 x);
```

Arguments:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pFIR</code></td>
<td>in</td>
<td>Pointer to a data structure containing private data for the fir filter; created by a call to <code>firCreate</code></td>
</tr>
<tr>
<td><code>x</code></td>
<td>in</td>
<td>Input data value</td>
</tr>
</tbody>
</table>

**Description:** Computes a Finite Impulse Response (FIR) filter for a single fractional data value. Prior to any call to `firs`, the FIR filter must be initialized via a call to `firCreate`; the FIR filter uses coefficients passed to that `firCreate` call. The function `firs` uses the private data structure established by `firCreate` to maintain the past history of data elements required by the FIR filter computation.

**Equation:** See Algorithm in this section.

**Returns:** The FIR filter computation returns the single output value which results from the FIR calculation on the single input value.

**Algorithm:**

```
z[j] = \sum_{k=0}^{nc-1} c[k] x[j-k] \quad 0 \leq j < n
```

**Range Issues:** If saturation is enabled, the `firs` function will return the maximum or minimum fractional values if overflow or underflow occurs during the calculation of the output element.

**Special Issues:** None

**Design/Implementation:** None

**Code Example 11-11** shows the usage of `dfr16FIRs`. The filter specifications for this example are the same as those in **Code Example 11-10**.

**Code Example 11-11.  `firs`**

```c
#define FIR_COEF_LENGTH 37 /*Number of filter taps */
#define NUM_SAMPLES 100 /*Number of incoming samples */
const Frac16 FirCoefs[] = {
  0xfff8,
  0x0012,
  ....,
};

dfr16_tFirStruct * pFir;
Frac16 * pFirCoefs
Frac16 x[NUM_SAMPLES];
Frac16 z[NUM_SAMPLES];
```
UInt16

tempIndex;

pFir = dfr16FIRCreate (pFirCoefs, FIR_COEF_LENGTH);
...

z[tempIndex+1] = dfr16FIRs (pFir, x[tempIndex+1]);
...

dfr16FIRDestroy (pFir);
11.3.21  

**firCreate** - Create Finite Impulse Response Filter

**Call(s):**

```c
dfri6_tFirStruct * dfri6FIRCreate  (Frac16 *pC, UInt16 n);
```

**Arguments:**

<table>
<thead>
<tr>
<th>pC</th>
<th>in</th>
<th>Pointer to a vector of FIR filter coefficients</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>in</td>
<td>Length of the vector of FIR filter coefficients pointed to by pC</td>
</tr>
</tbody>
</table>

**Description:** Performs memory allocation and initialization for the fir filter function.

The function `firCreate` first dynamically allocates the `dfri6_tFirStruct` data structure which is used by `fir` to preserve the filter’s state between calls. To optimize the performance of `fir`, `firCreate` then allocates the `dfri6_tFirStruct.pC` buffer to hold the FIR filter coefficients.

Next, `firCreate` allocates the buffer `dfri6_tFirStruct.pHistory` to save the past history of `n` data elements required by the FIR filter computation. The `firCreate` function tries to allocate this history buffer aligned on a `k`-bit boundary where `k=log2(n)`, so that `fir` can achieve maximum efficiency using modulo addressing of `pFIR`’s history buffer.

After the `dfri6_tFirStruct` structure and its component buffers have been allocated, `firCreate` calls the `firInit` function to initialize the `dfri6_tFirStruct` structure.

**Equation:** None

**Returns:** The `firCreate` function returns a pointer to the `dfri6_tFirStruct` data structure that it allocated if all allocations and initializations succeed. This pointer must then be passed to subsequent calls of the `fir` function.

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

**Range Issues:** None

**Special Issues:**

- `INCLUDE_MEMORY` must be defined in `appconfig.h` in order to initialize the Memory Manager, which is used by the `firCreate` function.
- Either `firCreate` or `firInit` must be called before any call to `fir`, `fis`, `firHistory`, or `firDestroy`.
- Multiple calls may be made to `firCreate` to initialize different `fir` filters, which can be used concurrently. Separate data streams must use different `dfri6_tFirStruct` structures returned by separate calls to `firCreate`.
- Call `firDestroy` to deallocate the `dfri6_tFirStruct` data structure allocated by `firCreate`.
- The `firCreate` provides a very portable function that allocates all data structures required by `fir`. To eliminate use of dynamic memory, you may allocate the `dfri6_tFirStruct` data structure statically in the application and call `firInit` directly, rather than calling the `firCreate` function. However, you should use either `firCreate` or `firInit`, but not both, since `firCreate` itself calls `firInit`.

**Design/Implementation:** In general, the data structure allocated by `firCreate` is treated as private data to better encapsulate the details of the implementation. However, refer to the file `dfri6priv.h` to see the `dfri6_tFirStruct` data structure.
Performance: If `firCreate` cannot optimally allocate the buffers within the `dfr16_tFirStruct` data structure, non-optimal buffers will be allocated. In this case, the `fir` function will operate correctly, but not at optimal performance. For example, if the buffer for the copy of the filter coefficients cannot be allocated in internal memory, `firCreate` will allocate the buffer out of external memory; if the history buffer cannot be aligned, a non-aligned buffer will be allocated. In either case, `fir` will work, but more slowly.

**Code Example 11-12.  `firCreate`**

```c
#define FIR_COEF_LENGTH 37 /*Number of filter taps*/
#define NUM_SAMPLES 100 /*Number of incoming samples*/

const Frac16 FirCoefs[] = {
  0xfff8,
  0x0012,
  ...
};

dfr16_tFirStruct * pFir;
Frac16       * pFirCoefs
Frac16       x[NUM_SAMPLES];
Frac16       z[NUM_SAMPLES];
UInt16       tempIndex;

pFirCoefs=(Frac16 *)(&FirCoefs[0])

pFir = dfr16FIRCreate (pFirCoefs, FIR_LENGTH);
...
tempIndex = NUM_SAMPLES - (sizeof(FirCoefs)/sizeof(Frac16)) - 1;

z[tempIndex+1] = dfr16FIRs (pFir, x[tempIndex+1]);
...
dfr16FIRDestroy (pFir);
```
11.3.22  

**firDestroy - Destroy Finite Impulse Response Filter**

Call(s):

```c
void dfr16FIRDestroy (dfr16_tFirStruct *pFIR);
```

Arguments:

<table>
<thead>
<tr>
<th><strong>Table 11-35. firDestroy arguments</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>pFIR</strong></td>
</tr>
<tr>
<td><strong>in</strong></td>
</tr>
<tr>
<td>Pointer to a data structure created by the firCreate function;</td>
</tr>
</tbody>
</table>

Description: The function *firDestroy* deallocates the data structure(s) initially allocated by *firCreate*.

Equation: None

Returns: Void

Range Issues: None

Special Issues: To deallocate a data structure created by *firCreate*, *firDestroy* may be called only once.

Design/Implementation: None


```c
#define FIR_COEF_LENGTH 37 /*Number of filter taps */
#define NUM_SAMPLES 100 /*Number of incoming samples */

const Frac16 FirCoefs[] = {
    0xfff8,
    0x0012,
    ...
};

{  
    dfr16_tFirStruct * pFir;
    Frac16       * pFirCoefs
    Frac16        x[NUM_SAMPLES];
    Frac16        z[NUM_SAMPLES];
    UInt16        tempIndex;

    pFirCoefs=(Frac16 *)(&FirCoefs[0])

    pFir = dfr16FIRCreate (pFirCoefs, FIR_COEF_LENGTH);
    ...
    z[tempIndex+1] = dfr16FIRs (pFir, x[tempIndex+1]);
    ...
    dfr16FIRDestroy (pFir);
}
```
11.3.23 firHistory - Reinitialize FIR History Buffer

Call(s):

```c
void dfr16FIRHistory (df16_tFirStruct *pFIR, Frac16 *pX);
```

Arguments:

<table>
<thead>
<tr>
<th>Argument</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; created by a call to firCreate</td>
</tr>
<tr>
<td>pX</td>
<td>in</td>
<td>Pointer to the input vector of ( n ) data elements</td>
</tr>
</tbody>
</table>

Description: The function `firHistory` reinitializes the Finite Impulse Response, (FIR), filter history buffer data values. Prior to any call to `firHistory`, the FIR filter must be initialized via a call to `firCreate`. The `firHistory` function uses the private data structure established by `firCreate` to establish the past history of data elements required by the FIR filter computation. The argument `pX` must point to \( n \) fractional data elements, where \( n \) is the number of coefficients used to establish the FIR filter in the call to `firCreate`.

Equation: None

Returns: The `firHistory` function returns “PASS”.

Range Issues: None

Special Issues: None

Design/Implementation: None
#define FIR_COEF_LENGTH 37 /*Number of filter taps */

#define NUM_SAMPLES 100 /*Number of incoming samples */

const Frac16 FirCoefs[] = {
    0xfff8,
    0x0012,
    ...
};

{  
dfr16_tFirStruct * pFir;
    Frac16 * pFirCoefs
    Frac16 x[NUM_SAMPLES];
    Frac16 z;

    pFirCoefs=(Frac16 *)&FirCoefs[0]

    pFir = dfr16FIRCreate (pFirCoefs, FIR_COEF_LENGTH);
    ...
    dfr16FIRHistory (pFir, &x[0]);

    z = dfr16FIRs (pFir, x[FIR_COEF_LENGTH+1]);
    ...
    dfr16FIRDestroy (pFir);
}
11.3.24 firInit - Initialize Finite Impulse Response Filter

Call(s):

```c
typedef struct dfr16_sFirStruct {
    Frac16   * pC;                 /* Coefficients for the filter */
    Frac16   * pHistory;           /* Memory for the filter history buffer */
    UWord16    Private[6];
} dfr16_tFirStruct;

void dfr16FIRInit (dfr16_tFirStruct *pFIR, Frac16 *pC, UInt16 n);
```

Arguments:

<table>
<thead>
<tr>
<th>pFir</th>
<th>in</th>
<th>Pointer to a previously allocated dfr16_tFirStruct data structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>pC</td>
<td>in</td>
<td>Pointer to a vector of FIR filter coefficients</td>
</tr>
<tr>
<td>n</td>
<td>in</td>
<td>Length of the vector of FIR filter coefficients pointed to by pC</td>
</tr>
</tbody>
</table>

**Description:** Initializes the fir filter function from a previously allocated dfr16_tFirStruct data structure. Typically, this dfr16_tFirStruct data structure is allocated statically by the caller prior to the firInit call. However, firCreate also uses firInit to initialize the dfr16_tFirStruct that it dynamically allocates.

**Equation:** None

**Returns:** The firInit function initializes the dfr16_tFirStruct data structure pointed to by pFir. The firInit function itself returns void.

**Range Issues:** None

**Special Issues:** The function firInit must be called before any call to fir, firs, or firHistory. However, note that firInit is called by firCreate.

Multiple calls may be made to firInit to initialize different fir filters, which can be used concurrently. Separate data streams must use different dfr16_tFirStruct structures initialized by separate calls to firInit.

You may allocate the dfr16_tFirStruct data structure statically in the application and call firInit directly, rather than call the firCreate function to eliminate use of dynamic memory. However, you should use either firCreate or firInit, but not both, since firCreate itself calls firInit.

**Design/Implementation:** The dfr16_tFirStruct data structure contains two single-dimensioned arrays of length n. The function firInit copies the array of coefficients pointed to by pC into the array dfr16_tFirStruct.pC used in the FIR filter. The firInit function also initializes the dfr16_tFirStruct.pHistory array used in the FIR filter.

The rest of the dfr16_tFirStruct data structure initialized by firInit is treated as private data to better encapsulate the details of the implementation. However, please review the file dfr16priv.h to see the dfr16_tFirStruct data structure.

**Performance:** The performance of the FIR filter varies greatly, depending on where/how the dfr16_tFirStruct.pC array and dfr16_tFirStruct.pHistory arrays are allocated. For optimal performance, prior to the call to firInit, allocate the dfr16_tFirStruct.pC array in internal data memory without regard to alignment and allocate the dfr16_tFirStruct.pHistory in internal data memory aligned for modulo n addressing. Use linker.cmd file directives to achieve this alignment.
Code Example 11-15.  

```
#define FIR_COEF_LENGTH 37 /*Number of filter taps */
#define NUM_SAMPLES 100 /*Number of incoming samples */

const Frac16 FirCoefs[] = {
  0xfff8,
  0x0012,
  ...
};

.....

{...
  Frac16 z1[NUM_SAMPLES];
  Frac16 x [NUM_SAMPLES];
  dfr16_tFirStruct fir;
  dfr16_tFirStruct *pFir=&fir;
  Frac16 *pFirCoefs;
  Frac16 firCoefArray[FIR_COEF_LENGTH * sizeof(Frac16)];
  Frac16 history[NUM_SAMPLES]; /*unaligned */

  /* Initialize FIR filter */
  pFir->pC =(Frac16 *)&firCoefArray[0];
  pFir -> pHistory =(Frac16 *)&history[0];

  pFirCoefs =(Frac16 *)&FirCoefs[0];

  dfr16FIRInit (pFir, pFirCoefs,sizeof(FirCoefs)/sizeof(Frac16));

  /* Run the Fir filter (C version)*/
  dfr16FIRC (pFir, &x[0], z1, NUM_SAMPLES);

  ...
}
```
11.3.25 *firdec* - Decimating FIR Filter

Call(s):

```c
UInt16 dfr16FIRDec(dfr16_tFirDecStruct *pFIRDEC, Frac16 *pX, Frac16 *pZ, UInt16 nx);
```

Arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pFIRDEC</code></td>
<td>Pointer to a data structure containing private data for the <em>firdec</em> filter; created by a call to <em>firdecCreate</em></td>
</tr>
<tr>
<td><code>pX</code></td>
<td>Pointer to the input vector of <code>nx</code> data elements</td>
</tr>
<tr>
<td><code>pZ</code></td>
<td>Pointer to the output vector of <code>*pNZ</code> data elements</td>
</tr>
<tr>
<td><code>nx</code></td>
<td>Length of the input data vector pointed to by <code>pX</code></td>
</tr>
</tbody>
</table>

**Description:** Computes a decimating Finite Impulse Response (FIR) filter for a vector of fractional data values. Prior to any call to *firdec*, the decimating FIR filter must be initialized via a call to *firdecCreate*; the FIR filter uses coefficients passed to that *firdecCreate* call. The function *firdec* uses the private data structure established by *firdecCreate* to maintain the past history of data elements required by the decimating FIR filter computation.

**Equation:** See Algorithm in this section.

**Returns:** The *firdec* filter computation generates output values which are stored in the vector pointed to by `pZ`. The number of output values generated is returned by the *firdec* function. In general, the length of the output vector is equal to `nx / f`, where `f` is the decimation factor established in the call to *firdecCreate*.

However, if `nx` is not an integral multiple of `f`, the length of the output vector may vary; see Special Issues in this section for more information on the length of the output vector.

**Algorithm:**

\[
z[j] = \sum_{k=0}^{nc - 1} c[k] x [j x f - k]
\]

**Range Issues:** If saturation is enabled, the *firdec* function will return the maximum or minimum fractional values if overflow or underflow occurs during the calculation of each output element.

The length of the input vector, `nx`, must be within the range `0 <= n <= 8192`.

**Special Issues:** The function *firdec* can be used for block filtering where the length of the input vector, `nx`, is greater than 1, or for single samples where `nx = 1`. In place computation is allowed; i.e., the address of the output vector, `pZ`, can be equal to the address of the input vector, `pX`.

The length of the input vector, `nx`, does not have to be an integral multiple of the decimation factor `f`. The *firdec* function tracks the number of inputs to properly generate the decimated number of outputs. This implies that for any particular *firdec* call, the number of outputs returned by the function may differ by plus or minus 1 from the number of inputs, `nx`, divided by the decimation factor, `f`.
Design/Implementation: Modulo addressing is utilized in the implementation of firdec to optimize the performance of the FIR filter. See the description of firdecCreate, Section 11.3.26, for details on the data structure allocation utilized for modulo addressing.
**11.3.26 firdecCreate - Create Decimating FIR Filter**

Call(s):

\[ \text{dfrl6_tFirDecStruct * dfrl6FIRDecCreate (Frac16 *pC, UInt16 n, UInt16 f)} \]

Arguments:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( pC )</td>
<td>in</td>
<td>Pointer to a vector of decimating FIR filter coefficients</td>
</tr>
<tr>
<td>( n )</td>
<td>in</td>
<td>Length of the vector of decimating FIR filter coefficients pointed to by ( pC )</td>
</tr>
<tr>
<td>( f )</td>
<td>in</td>
<td>The decimation factor; the firdec filter will drop every ( f-1 ) output</td>
</tr>
</tbody>
</table>

**Description:** Performs the initialization for the firdec filter function. The function firdecCreate allocates and initializes a data structure, `dfrl6_tFirDecStruct`, which is used by firdec to preserve the filter’s state between calls. The data structure preserves the \( pC \) pointer to the vector of decimating FIR filter coefficients. The firdecCreate function also allocates a buffer to save the past history of \( n \) data elements required by the decimating FIR filter computation; a pointer to this buffer is stored in the data structure pointed to by \( pFIRDEC \). So firdec can achieve maximum efficiency using modulo addressing, firdecCreate allocates the history buffer from the system heap such that its address starts on a \( k \)-bit boundary, where \( k = \log_2(n) \).

**Equation:** None

**Returns:** The firdecCreate function returns a pointer to the `dfrl6_tFirDecStruct` data structure if all allocations and initializations succeed. This pointer must then be passed to subsequent calls of the firdec function.

If insufficient system heap is available, or if the history buffer cannot be aligned properly due to insufficient memory in the system heap, the firdecCreate function returns NULL.

**Range Issues:** None

**Special Issues:**

- \textit{INCLUDE\_MEMORY} must be defined in \textit{appconfig.h} in order to initialize the Memory Manager, which is used by the firdecCreate function.
- The function firdecCreate must be called before any call to firdec or firdecDestroy. Multiple calls may be made to firdecCreate, however, to initialize different firdec filters, which can be used concurrently. The vector of coefficients, pointed to by \( pC \) as input to firdecCreate, must exist for the entire duration in which firdec is called. The firdecCreate function saves the \( pC \) pointer in its allocated data structure so that firdec can use it. No copy of the coefficient vector is made. Call firdecDestroy to deallocate the data structure allocated by firdecCreate.

**Design/Implementation:** In general, the data structure allocated by firdecCreate is treated as private data to better encapsulate the details of the implementation. However, please review the actual implementation of firdecCreate to see the private data structure.
11.3.27 **firdecDestroy** - Destroy Decimating FIR Filter

Call(s):

```c
void dfr16FIRDecDestroy (void *pFIRDEC);
```

Arguments:

<table>
<thead>
<tr>
<th>pFIRDEC</th>
<th>in</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pointer to a data structure created by the <em>firdecCreate</em> function</td>
<td></td>
</tr>
</tbody>
</table>

**Description:** The function *firdecDestroy* deallocates the data structure initially allocated by *firdecCreate*.

**Equation:** None

**Returns:** Void

**Range Issues:** None

**Special Issues:** The *firdecDestroy* function may be called only once to deallocate a data structure created by *firdecCreate*.

**Design/Implementation:** None
11.3.28 **firdecInit** - Initialize Decimating FIR Filter

Call(s):

```c
typedef struct dfr16_sFirdecStruct {
    Frac16  * pC;                 /* Coefficients for the filter */
    Frac16  * pHistory;           /* Memory for the filter history buffer */
    UWord16 Private[6];
} dfr16_tFirDecStruct;

void dfr16FIRDecInit (dfr16_tFirDecStruct *pFIRDec, Frac16 *pC, UInt16 n);
```

**Arguments:**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pFIRDec</code></td>
<td>Pointer to a previously allocated <code>dfr16_tFirDecStruct</code> data structure</td>
</tr>
<tr>
<td><code>pC</code></td>
<td>Pointer to a vector of <code>firdec</code> filter coefficients</td>
</tr>
<tr>
<td><code>n</code></td>
<td>Length of the vector of <code>firdec</code> filter coefficients pointed to by <code>pC</code></td>
</tr>
</tbody>
</table>

**Description:** Initializes the `firdec` filter function from a previously allocated `dfr16_tFirDecStruct` data structure. Typically, this `dfr16_tFirDecStruct` data structure is allocated statically by the caller prior to the `firdecInit` call. However, `firdecCreate` also uses `firdecInit` to initialize the `dfr16_tFirDecStruct` that it dynamically allocates.

**Equation:** None

**Returns:** The `firdecInit` function initializes the `dfr16_tFirDecStruct` data structure pointed to by `pFIRDec`. The `firdecInit` function itself returns “void”.

**Range Issues:** None

**Special Issues:** The function `firdecInit` must be called before any call to `firdec`. However, note that `firdecCreate` calls `firdecInit`.

Multiple calls may be made to `firdecInit` to initialize different `firdec` filters, which can be used concurrently. Separate data streams must use different `dfr16_tFirDecStruct` structures initialized by separate calls to `firdecInit`.

You may allocate the `dfr16_tFirDecStruct` data structure statically in the application and call `firdecInit` directly, rather than call the `firdecCreate` function to eliminate use of dynamic memory. However, you should use either `firdecCreate` or `firdecInit`, but not both, since `firdecCreate` itself calls `firdecInit`.

**Design/Implementation:** The `dfr16_tFirDecStruct` data structure contains two single-dimensioned arrays of length `n`. The function `firdecInit` copies the array of coefficients pointed to by `pC` into the array `dfr16_tFirDecStruct.pC` used in the `firdec` filter. The `firdecInit` function also initializes the `dfr16_tFirDecStruct.pHistory` array used by the `firdec` filter.

The rest of the `dfr16_tFirDecStruct` data structure initialized by `firdecInit` is treated as private data to better encapsulate the details of the implementation. However, please review the file `dfr16priv.h` to see the `dfr16_tFirDecStruct` data structure.
Performance: The performance of the firdec filter varies greatly depending on where/how the dfr16_tFirDecStruct.pC array and dfr16_tFirDecStruct.pHistory arrays are allocated. For optimal performance, prior to the call to firdecInit, allocate the dfr16_tFirDecStruct.pC array in internal data memory without regard to alignment and allocate the dfr16_tFirDecStruct.pHistory in internal data memory aligned for modulo n addressing. Use linker.cmd file directives to achieve this alignment.
11.3.29 firint - Interpolating FIR Filter

Call(s):

```c
UInt16 dfrl16FIRInt(dfrl16_tFirIntStruct *pFIRInt, Frac16 *pX, Frac16 *pZ, UInt16 n);
```

Arguments:

<table>
<thead>
<tr>
<th>pFIRInt</th>
<th>in</th>
<th>Pointer to a data structure containing private data for the firint filter; created by a call to firintCreate</th>
</tr>
</thead>
<tbody>
<tr>
<td>pX</td>
<td>in</td>
<td>Pointer to the input vector of n data elements</td>
</tr>
<tr>
<td>pZ</td>
<td>inout</td>
<td>Pointer to the output vector of (n * f) data elements</td>
</tr>
<tr>
<td>n</td>
<td>in</td>
<td>Length of the input data vector pointed to by pX</td>
</tr>
</tbody>
</table>

Description: Computes an interpolating Finite Impulse Response (FIR) filter for a vector of fractional data values. Prior to any call to firint, the interpolating FIR filter must be initialized via a call to firintCreate; the FIR filter uses coefficients passed to that firintCreate call. The function firint uses the private data structure established by firintCreate to maintain the past history of data elements required by the interpolating FIR filter computation.

Equation: See Algorithm section below

Returns: The firint filter computation generates (n * f) output values, which are stored in the vector pointed to by pZ, where f is the interpolation factor established in the call to firintCreate.

The function firint returns the number of output values generated in the array pointed to by pZ.

Algorithm:

\[
z[j] = \sum_{k=0}^{nc-1} c[k] \times \lfloor j / f \rfloor - k \quad 0 \leq j < nf\]

Range Issues: If saturation is enabled, the firint function will return the maximum or minimum fractional values if overflow or underflow occurs during the calculation of each output element. The length of the input vector, n, must be within the range 0 <= n <= 8192.

Special Issues: The function firint can be used for block filtering where the length of the input vector, n, is greater than 1, or for single samples where n = 1. In place computation is not allowed; i.e., the address of the output vector, pZ, may not equal the address of the input vector, pX.

Design/Implementation: Modulo addressing is utilized in the implementation of firint to optimize the performance of the FIR filter. See the description of firintCreate, Section 11.3.30, for details on the data structure allocation utilized for modulo addressing.
Call(s):

```c
dfrl6_tFirIntStruct * dfrl6FIRIntCreate (Frac16 *pC, UInt16 n, UInt16 f);
```

Arguments:

<table>
<thead>
<tr>
<th>pC</th>
<th>in</th>
<th>Pointer to a vector of interpolating FIR filter coefficients</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>in</td>
<td>Length of the vector of interpolating FIR filter coefficients pointed to by pC</td>
</tr>
<tr>
<td>f</td>
<td>in</td>
<td>The interpolation factor; the <code>firint</code> filter will produce f outputs for every input</td>
</tr>
</tbody>
</table>

**Description:** Performs the initialization for the `firint` filter function. The `firintCreate` function allocates and initializes a data structure, `dfrl6_tFirIntStruct`, which is used by `firint` to preserve the filter’s state between calls. The data structure preserves the pC pointer to the vector of interpolating FIR filter coefficients. The function `firintCreate` also allocates a buffer to save the past history of n data elements required by the interpolating FIR filter computation; a pointer to this buffer is stored in the data structure returned by `firint`. So that `firint` can achieve maximum efficiency using modulo addressing, `firintCreate` allocates the history buffer from the system heap such that its address starts on a k-bit boundary, where k=log2(n).

**Equation:** None

**Returns:** If all allocations and initializations succeed, `firintCreate` returns the pointer to the allocated data structure `dfrl6_tFirIntStruct`. This pointer must then be passed to subsequent calls of the `firint` function.

If insufficient system heap is available, or if the history buffer cannot be aligned properly due to insufficient memory in the system heap, the `firintCreate` function returns “NULL”.

**Range Issues:** None

**Special Issues:**

- `INCLUDE_MEMORY` must be defined in `appconfig.h` in order to initialize the Memory Manager, which is used by the `firintCreate` function.
- The function `firintCreate` must be called before any call to `firint` or `firintDestroy`. Multiple calls may be made to `firintCreate`, however, to initialize different `firint` filters, which can be used concurrently. The vector of coefficients, pointed to by pC as input to `firintCreate`, must exist for the entire duration in which `firint` is called. The `firintCreate` function saves the pC pointer in its allocated data structure so that `firint` can use it. No copy of the coefficient vector is made. Call `firintDestroy` to deallocate the data structure allocated by `firintCreate`.

**Design/Implementation:** In general, the data structure allocated by `firintCreate` is treated as private data to better encapsulate the details of the implementation. However, please refer to the actual implementation of `firintCreate` to view the private data structure.
11.3.31  *firintDestroy* - Destroy Interpolating FIR Filter

Call(s):

```c
void dfr16FIRIntDestroy (void *pFIRInt);
```

Arguments:

| pFIRInt | in | Pointer to a data structure created by the *firintCreate* function; |

Description: The function *firintDestroy* deallocates the data structure initially allocated by *firintCreate*.

Equation: None

Returns: Void

Range Issues: None

Special Issues: The *firintDestroy* function may be called only once to deallocate a data structure created by *firintCreate*.

Design/Implementation: None
11.3.32 **firintInit** - Initialize Interpolating FIR Filter

**Call(s):**

```c
typedef struct dfr16_sFirIntStruct {
    Frac16  * pC;                 /* Coefficients for the filter */
    Frac16  * pHistory;           /* Memory for the filter history buffer */
    UWord16 Private[6];          
} dfr16_tFirIntStruct;

void dfr16FIRIntInit (dfr16_tFirIntStruct *pFIRInt, Frac16 *pC, UInt16 n);
```

**Arguments:**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pFIRInt</td>
<td>Pointer to a previously allocated dfr16_tFirIntStruct data structure</td>
</tr>
<tr>
<td>pC</td>
<td>Pointer to a vector of firint filter coefficients</td>
</tr>
<tr>
<td>n</td>
<td>Length of the vector of firint filter coefficients pointed to by pC</td>
</tr>
</tbody>
</table>

**Description:** Initializes the firint filter function from a previously allocated dfr16_tFirIntStruct data structure. Typically, this dfr16_tFirIntStruct data structure is allocated statically by the caller prior to the firintInit call. However, firintCreate also uses firintInit to initialize the dfr16_tFirIntStruct that it dynamically allocates.

**Equation:** None

**Returns:** The firintInit function initializes the dfr16_tFirIntStruct data structure pointed to by pFIRInt. The firintInit function itself returns void.

**Range Issues:** None

**Special Issues:** The function firintInit must be called before any call to firint. However, note that firintCreate calls firintInit.

Multiple calls may be made to firintInit to initialize different firint filters which can be used concurrently. Separate data streams must use different dfr16_tFirIntStruct structures initialized by separate calls to firintInit.

You may allocate the dfr16_tFirIntStruct data structure statically in the application and call firintInit directly, rather than call the firintCreate function to eliminate use of dynamic memory. However, you should use either firintCreate or firintInit, but not both, since firintCreate itself calls firintInit.

**Design/Implementation:** The dfr16_tFirIntStruct data structure contains two single-dimensioned arrays used by firint. The array dfr16_tFirIntStruct.pHistory must be of length int((n + f - 1) / f). The array dfr16_tFirIntStruct.pC must be of length f * int((n + f - 1) / f). The function firintInit copies the array of coefficients pointed to by pC into the array dfr16_tFirIntStruct.pC used in the fiintc filter. The firintInit function also initializes the dfr16_tFirIntStruct.pHistory array used by the firint filter.

The rest of the dfr16_tFirIntStruct data structure initialized by firintInit is treated as private data to better encapsulate the details of the implementation. However, please refer to the file dfr16priv.h to review the dfr16_tFirIntStruct data structure.
Performance: The performance of the `firint` filter varies greatly, depending on where/how the `dfr16_tFirIntStruct.pC` array and `dfr16_tFirIntStruct.pHistory` arrays are allocated. For optimal performance, prior to the call to `firintInit`, allocate the `dfr16_tFirIntStruct.pC` array in internal data memory without regard to alignment and allocate the `dfr16_tFirIntStruct.pHistory` in internal data memory aligned for modulo `\( \text{int}(n + f - 1) / f \)` addressing. Use `linker.cmd` file directives to achieve this alignment.
**11.3.33 iir - Infinite Impulse Response Filter**

**Call(s):**

```
Result dfr16IIR (dfr16_tIirStruct *pIIR, Frac16 *pX, Frac16 *pZ, UInt16 n);
```

**Arguments:**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pIIR</code> in Pointer to a data structure containing private data for the <code>iir</code> filter; created by a call to <code>iirCreate</code></td>
<td></td>
</tr>
<tr>
<td><code>pX</code> in Pointer to the input vector of <code>n</code> data elements</td>
<td></td>
</tr>
<tr>
<td><code>pZ</code> inout Pointer to the output vector of <code>n</code> data elements</td>
<td></td>
</tr>
<tr>
<td><code>n</code> in Length of the input and output vectors</td>
<td></td>
</tr>
</tbody>
</table>

**Description:** Computes an Infinite Impulse Response, (IIR), filter for a vector of fractional data values using a cascade filter of biquad coefficients. Prior to any call to `iir`, the IIR filter must be initialized via a call to `iirCreate`; the IIR filter uses biquad coefficients passed to that `iirCreate` call. The function `iir` uses the private data structure established by `iirCreate` to maintain the past history of data elements (w) required by the cascaded IIR filter computation.

**Equation:** See Algorithm section below

**Returns:** The IIR filter computation generates `n` output values, which are stored in the vector pointed to by `pZ`. The function `iir` returns:

- *FAIL* (“-1”): if the length of the input and output vectors, `n`, is greater than 8192
- *PASS* (“0”): otherwise

**Algorithm:** The biquad structure that is implemented is shown in **Figure 11-1.**

![Figure 11-1. Biquad Structure with a 1 Scaled](image-url)
For each biquad calculation:

\[
\begin{align*}
   w(n) &= 2 \left[ \frac{x(n) + a_2 w(n-2)}{2} + \frac{a_1}{2} w(n-1) \right] \\
   y(n) &= b_0 w(n) + b_1 w(n-1) + b_2 w(n-2)
\end{align*}
\]

where “x(n)” is the input and “y(n)” is the output.

**Range Issues:** If saturation is enabled, the `iir` function will return the maximum or minimum fractional values if overflow or underflow occurs during the calculation of each output element.

The length of the input and output vectors, \( n \), must be within the range \( 0 \leq n \leq 8192 \).

The coefficients \( b_0, b_1 \) and \( b_2 \) must all be less than 1. If any of these coefficients is higher than or equal to 1, scale them to fall in the valid range (which is \( >-1 \) and \( <1 \)), then multiply the final output by the scaling factor.

**Special Issues:** The `iir` function can be used for block filtering, where the length of the input vector, \( n \), is greater than 1, or for single samples, where \( n = 1 \). In-place computation is allowed; i.e., the addresses of the output vector, \( pZ \), can equal the address of the input vector, \( pX \).

Refer to the specification of `iirCreate` for a description of the coefficients used by the `iir` filter.

**Design/Implementation:** Modulo addressing is utilized in the implementation to optimize the performance of the IIR filter. See the description of `iirCreate`, Section 11.3.34, for details on the data structure allocation utilized for modulo addressing.

**Code Example 11-16** shows the usage of `iir` function. In this example a filter with the following specifications is implemented:

- **Type:** Chebychev2
- **Order:** 4
- **Passband edge:** 1000Hz
- **StopBand Edge:** 2000Hz
- **Maximum Ripple in PassBand:** 1db
- **Attenuation in Stopband:** 30db
- **Sampling Frequency:** 8000Hz

Note that the order of initialization of coefficients is \( a_2, a_1/2, b_0, b_1, b_2 \) for each biquad.
Code Example 11-16.  

```c
{
    dfr16_tIirStruct *pIIR;
    Frac16 *pC;
    Result res;
    Frac16 *pX, *pZ;
    UInt16 n, nbiq;
    /* The order of coefficients: a2, a1/2, b0, b1, b2 for each biquad
    Note that |b0|, |b1| and |b2| should all be lesser than 1.
    
    Test Case Filter Specifications:
    Type: Chebychev2
    Order: 4
    Passband edge: 1000Hz
    StopBand edge: 2000Hz
    Maximum Ripple in PassBand: 1db
    Attenuation in Stopband: 30db
    Sampling Frequency: 8000Hz */

    const Frac16 IirCoefs[] =
    {
        FRAC16(-0.1310),    /* a2 */
        FRAC16(0.27805),   /* a1/2 */
        FRAC16(0.1808),    /* b0 */
        FRAC16(0.2133),    /* b1 */
        FRAC16(0.1808),    /* b2 */
        FRAC16(-0.6107),   /* a2 */
        FRAC16(0.4944),    /* a1/2 */
        FRAC16(0.3892),    /* b0 */
        FRAC16(-0.1566),   /* b1 */
        FRAC16(0.3892)     /* b2 */
    };

    ...
    pIIR = dfr16IIRCreate (pC, nbiq);
    ...
    res = dfr16IIR (pIIR, pX, pZ, n);
    ...
    dfr16IIRDestroy (pIIR);
}
```
11.3.34  *iirCreate* - Create IIR Filter

Call(s):

```
dfr16_tIirStruct *dfr16IIRCreate (Frac16 *pC, UInt16 nbiq);
```

Arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pC</td>
<td>Pointer to a vector of cascaded IIR filter coefficients, organized as a linear vector of ( n ) biquads, 5 coefficients per biquad</td>
</tr>
<tr>
<td>n</td>
<td>The number of biquads for the cascaded ( iir ) filter</td>
</tr>
</tbody>
</table>

**Description:** The function *iirCreate* initializes the cascade \( iir \) filter function and allocates and initializes a data structure, which is used by \( iir \) to preserve the filter’s state between calls. The data structure preserves the \( pC \) pointer to the linearized vector of \( n \) biquad IIR filter coefficients. The *iirCreate* function also allocates a buffer to save the past history of two data elements (as shown in the following formula using variable “\( w \)”) per biquad required by the cascade IIR filter computation; a pointer to this buffer is stored in the private data structure. So that \( iir \) can achieve maximum efficiency using modulo addressing, \( iirCreate \) allocates the history buffer from the system heap such that its address starts on a \( k \)-bit boundary, where:

\[
k = \log_2 2n
\]

The biquad coefficients, \( a \) & \( b \), are shown below in the cascade \( iir \) filter algorithm for a biquad:

\[
w(n) = 2 \left[ \frac{x(n) + a_2 w(n-2)}{2} + \frac{a_1}{2} w(n-1) \right]
\]

\[
y(n) = b_0 w(n) + b_1 w(n-1) + b_2 w(n-2)
\]

Five coefficients comprise each biquad: \( a_1, a_2, b_0, b_1, \) and \( b_2 \). The function \( pC \) must be initialized with the biquad coefficients in the following order: \( a_2, a_1/2, b_0, b_1 \) and \( b_2 \). If there is more than one biquad to be used in cascade, the coefficients of the second biquad follow those of the first, and so on.

**Equation:** See Description section.

**Returns:** The *iirCreate* function returns a pointer to a private data structure, which must then be passed to subsequent calls of the \( iir \) function.

**Note:** Because *iirCreate* does not return “FAIL” status which may be due to a fail in the memory allocation for private data structure, the user must check the returned pointer for “NULL”.

**Range Issues:** None

**Special Issues:**

- *INCLUDE_MEMORY* must be defined in \textit{appconfig.h} in order to initialize the Memory Manager, which is used by the *iirCreate* function.
The function `iirCreate` must be called before any call to `iir` or `iirDestroy`. Multiple calls may be made to `iirCreate`, however, to initialize different `iir` filters which could be used concurrently. The vector of biquad coefficients, pointed to by `pC` as input to `iirCreate`, must exist for the entire duration in which `iir` is called. The `iirCreate` function saves the `pC` pointer in its allocated data structure so that `iir` can use it. No copy of the coefficient vector is made. Call `iirDestroy` to deallocate the data structure allocated by `iirCreate`.

**Design/Implementation:** In general, the data structure allocated by `iirCreate` is treated as private data to better encapsulate the details of the implementation. However, please refer to the actual implementation of `iirCreate` to review the private data structure.
11.3.35  \textit{iirDestroy} - Destroy IIR Filter

Call(s):

\begin{verbatim}
void dfr16IIRDestroy (dfr16_tIirStruct *pIIR);
\end{verbatim}

Arguments:

\begin{table}[h]
\centering
\begin{tabular}{|c|c|}
\hline
\textit{pIIR} & in  \\
\hline
\end{tabular}
\caption{\textit{iirDestroy} arguments}
\end{table}

Description: The function \textit{iirDestroy} deallocates the data structure(s) initially allocated by \textit{iirCreate}.

Equation: None

Returns: Void

Range Issues: None

Special Issues: The \textit{iirDestroy} function may be called only once to deallocate a data structure created by \textit{iirCreate}.

Design/Implementation: None
11.3.36  \textit{iirInit} - Initialize IIR Filter

Call(s):

\begin{verbatim}
void dfr16IIRInit (dfr16_tIirStruct * pIir, Frac16 *pC, UIInt16 nbiq)
\end{verbatim}

Arguments:

\begin{table}[h]
\centering
\begin{tabular}{|l|l|p{10cm}|}
\hline
\textit{plir} & in/out & Pointer to a data structure allocated statically by the caller before \textit{iirInit} is called \\
\hline
\textit{pC} & in & Pointer to a vector of cascaded IIR filter coefficients, organized as a linear vector of \textit{n} biquads, 5 coefficients per biquad \\
\hline
\textit{niq} & in & The number of biquads for the cascaded \textit{iir} filter \\
\hline
\end{tabular}
\caption{\textit{iirCreate} arguments}
\end{table}

\textbf{Description:} The function \textit{iirInit} performs the initialization for the cascade \textit{iir} filter function and initializes the function from a previously-allocated \textit{dfr16_tIirStruct} data structure. Typically, the \textit{dfr16_tIirStruct} data structure is allocated statically by the caller prior to the \textit{iirInit} call. The \textit{iirCreate} function also uses \textit{iirInit} to initialize the \textit{dfr16_tIirStruct} that it dynamically allocates.

\textbf{Equation:} See Description, Section 11.3.34.

\textbf{Returns:} The \textit{iirInit} function returns a pointer to the data structure \textit{plir}.

\textbf{Range Issues:} None

\textbf{Special Issues:} Design/Implementation: The \textit{plir} data structure must be statically allocated prior to calling the \textit{iirInit} function.

\textbf{Code Example 11-17.} \textit{iirInit}

\begin{verbatim}
#define NUM_SAMPLES_IIR 100 /*Number of incoming samples */

const Frac16 IirCoefs[] = {
    \ldots,
};

Frac16 zIIR[NUM_SAMPLES];
Frac16 xIIR[NUM_SAMPLES];
Frac16 history[HISTORY_BUFFER_SIZE * sizeof(Frac16)];
Frac16 iirCoefArray[IIR_COEF_LENGTH* sizeof(Frac16)];

/* IIR Structures */

dfr16_tIirStruct Iir;
dfr16_tIirStruct *pIir = &Iir;

pIir -> pC = (Frac16 *)&iirCoefArray[0];
pIir-> Coefs = (Frac16 *)&IirCoefs[0];
pIir -> pHistory = (Frac16 *)&history[0];

/* Initialize the IIR filter */
\end{verbatim}
dfr16IIRInit (pIir,  
    pIirCoefs,  
    sizeof(IirCoefs)/(sizeof(Frac16)*FILT_COEF_PER_BIQ));

    /*Run the IIR filter (C version)*/  
    dfr16IIRC (pIir, xIIR, zIIR, NUM_SAMPLES_IIR);  
}
Chapter 12
Performance

12.1 DSP56824 Performance

This chapter details the performance of each of the DSP Function Library functions on the target DSP56824.

12.2 Trigonometric Math Function Performance

This section documents the performance of the trigonometric functions on the DSP56824. Table 12-1 shows the average counts taken over the input range [-1, 1].

<table>
<thead>
<tr>
<th>Trigonometric Function</th>
<th>Oscillator Cycles</th>
<th>Machine Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>sinPIx</td>
<td>900</td>
<td>195</td>
</tr>
<tr>
<td>cosPIx</td>
<td>1078</td>
<td>244</td>
</tr>
<tr>
<td>AsinOverPI</td>
<td>1237</td>
<td>300</td>
</tr>
<tr>
<td>AcosOverPI</td>
<td>1375</td>
<td>335</td>
</tr>
<tr>
<td>AtanOverPI</td>
<td>2337</td>
<td>597</td>
</tr>
<tr>
<td>Atan2OverPI</td>
<td>2606</td>
<td>665</td>
</tr>
</tbody>
</table>

12.3 Signal Processing Function Performance

This section documents the performance of the Digital Signal Processing functions on the DSP56824.
12.3.1 autoCorr

Table 12-2 shows the approximate worst-case cycle counts and Memory details. In the table, nx = input vector length

<table>
<thead>
<tr>
<th>Options</th>
<th>Instruction Cycle Count</th>
<th>Program Memory (words)</th>
<th>Data Memory (words)</th>
<th>Data ROM (words)</th>
<th>Stack Depth (words)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CORR_RAW</td>
<td>$(nx)^2 + 33nx + 52$</td>
<td>96</td>
<td>0</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>CORR_BIAS</td>
<td>$(nx)^2 + 33nx + 75$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CORR_UNBIAS</td>
<td>$(nx)^2 + 62nx + 52$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

12.3.2 cbitrev

Table 12-3 details the approximate worst-case instruction cycle count (= oscillator clocks/2) for Complex Bit-Reverse ASM Code.

<table>
<thead>
<tr>
<th>Size of Complex FFT, N</th>
<th>Instruction Cycle Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>271</td>
</tr>
<tr>
<td>16</td>
<td>521</td>
</tr>
<tr>
<td>32</td>
<td>1063</td>
</tr>
<tr>
<td>64</td>
<td>2081</td>
</tr>
<tr>
<td>128</td>
<td>4195</td>
</tr>
<tr>
<td>256</td>
<td>8285</td>
</tr>
<tr>
<td>512</td>
<td>16615</td>
</tr>
<tr>
<td>1024</td>
<td>32993</td>
</tr>
<tr>
<td>2048</td>
<td>66043</td>
</tr>
</tbody>
</table>

12.3.3 cfft

Table 12-4 shows the approximate worst-case instruction cycle counts (= oscillator clocks/2) for the ASM code. For more details on options for FFT, see Section 11.3.4.
### 12.3.4 cifft

Table 12-5 details the approximate worst-case instruction cycle counts (= oscillator clocks/2) for the ASM code. For information on options for IFFT, see Section 11.3.8.

<table>
<thead>
<tr>
<th>Size of Complex FFT, N</th>
<th>Core Complex FFT</th>
<th>Core Complex FFT + API</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>AS</td>
<td>BFP</td>
</tr>
<tr>
<td>8</td>
<td>438</td>
<td>736</td>
</tr>
<tr>
<td>16</td>
<td>1062</td>
<td>1635</td>
</tr>
<tr>
<td>32</td>
<td>2454</td>
<td>3917</td>
</tr>
<tr>
<td>64</td>
<td>5558</td>
<td>8231</td>
</tr>
<tr>
<td>128</td>
<td>12438</td>
<td>18393</td>
</tr>
<tr>
<td>256</td>
<td>27574</td>
<td>40843</td>
</tr>
<tr>
<td>512</td>
<td>60630</td>
<td>90045</td>
</tr>
<tr>
<td>1024</td>
<td>132342</td>
<td>197103</td>
</tr>
<tr>
<td>2048</td>
<td>286998</td>
<td>428577</td>
</tr>
</tbody>
</table>

---

**Table 12-5. cifft Performance**

<table>
<thead>
<tr>
<th>Size of Complex IFFT, N</th>
<th>Core Complex IFFT</th>
<th>Core Complex IFFT + API</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>AS</td>
<td>BFP</td>
</tr>
<tr>
<td>8</td>
<td>455</td>
<td>723</td>
</tr>
<tr>
<td>16</td>
<td>1084</td>
<td>1626</td>
</tr>
<tr>
<td>32</td>
<td>2500</td>
<td>3963</td>
</tr>
<tr>
<td>64</td>
<td>5652</td>
<td>8342</td>
</tr>
<tr>
<td>128</td>
<td>12628</td>
<td>18792</td>
</tr>
<tr>
<td>256</td>
<td>27956</td>
<td>42010</td>
</tr>
<tr>
<td>512</td>
<td>61396</td>
<td>93132</td>
</tr>
<tr>
<td>1024</td>
<td>133876</td>
<td>204798</td>
</tr>
<tr>
<td>2048</td>
<td>290068</td>
<td>447024</td>
</tr>
</tbody>
</table>
12.3.5 **corr**

Table 12-6 shows the approximate worst-case cycle counts and machine instructions. In the table, \(nx\) = length of the first input vector, and \(ny\) = length of the second input vector.

<table>
<thead>
<tr>
<th>Options</th>
<th>Oscillator Cycles</th>
<th>Machine Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>CORR_RAW</td>
<td>(140 + 144(nx - 1) + 132ny)</td>
<td>(38 + 36(nx - 1) + 38ny)</td>
</tr>
<tr>
<td>CORR_BIAS</td>
<td>(196 + 144(nx - 1) + 132ny)</td>
<td>(45 + 36(nx - 1) + 38ny)</td>
</tr>
<tr>
<td>CORR_UNBIAS</td>
<td>(140 + 208(nx - 1) + 196ny)</td>
<td>(38 + 46(nx - 1) + 48ny)</td>
</tr>
</tbody>
</table>

12.3.6 **fir**

Performance of the \(fir\) functions vary, depending on whether \(firCreate\) was able to align the history buffer in memory to perform modulo addressing and to allocate the coefficient buffer within internal memory.

For the performance formulas below, let \(n\) be the number of input samples to be processed by the filter and let \(f\) be the number of filter coefficients.

**Case 1:** History buffer aligned for modulo addressing; coefficients in internal memory

- Number of machine instructions (uses REP): \(29 + 13n\)
- Number of oscillator cycles: \(132 + n(2f + 50)\)
- Number of oscillator cycles interrupts blocked: \(2f\) (due to REP instruction)

**Case 2:** History buffer aligned for modulo addressing; coefficients in external memory

- Number of machine instructions: \(31 + n(2f + 11)\)
- Number of oscillator cycles: \(140 + n(6f + 50)\)
- Number of oscillator cycles interrupts blocked: \(0\)

**Case 3:** History buffer not aligned for modulo addressing; coefficients in external memory

- Number of machine instructions: \(29 + n(6f + 22)\)
- Number of oscillator cycles: \(144 + n(22f + 86)\)
- Number of oscillator cycles interrupts blocked: \(0\)

12.3.7 **iir**

Performance of the \(iir\) functions vary, depending on whether \(iirCreate\) was able to align the history buffer in memory to perform modulo addressing and to allocate the coefficient buffer within internal memory.
For the performance formulas in Table 12-7, let \( n \) be the number of input samples to be processed by the filter and let \( nbiq \) be the number of filter biquads. Each biquad uses 5 coefficients. The following cases are defined:

Case 1: History buffer aligned for modulo addressing; coefficients in internal memory
Case 2: History buffer aligned for modulo addressing; coefficients in external memory
Case 3: History buffer not aligned for modulo addressing; coefficients in external memory

<table>
<thead>
<tr>
<th>Case</th>
<th>Oscillator Cycles</th>
<th>Machine Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Case 1</td>
<td>( 100 + n(80 + 32*nbiq) )</td>
<td>( 26 + n(13 + 11*nbiq) )</td>
</tr>
<tr>
<td>Case 2</td>
<td>( 116 + n(68 + 28*nbiq) )</td>
<td>( 29 + n(9 + 13*nbiq) )</td>
</tr>
<tr>
<td>Case 3</td>
<td>( 120 + n(56 + 32*nbiq) )</td>
<td>( 28 + n(10 + 13*nbiq) )</td>
</tr>
</tbody>
</table>

### 12.3.8 rfft

Table 12-8 details the approximate worst-case instruction cycle counts (= oscillator clocks/2) for the ASM code. For more information on options for FFT, see Section 11.3.12.

<table>
<thead>
<tr>
<th>Size of Real FFT, ( N )</th>
<th>Core Real FFT</th>
<th>Core Real FFT + API</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>AS</td>
<td>BFP</td>
</tr>
<tr>
<td>8</td>
<td>485</td>
<td>729</td>
</tr>
<tr>
<td>16</td>
<td>1007</td>
<td>1435</td>
</tr>
<tr>
<td>32</td>
<td>2081</td>
<td>2914</td>
</tr>
<tr>
<td>64</td>
<td>4415</td>
<td>6103</td>
</tr>
<tr>
<td>128</td>
<td>9337</td>
<td>12931</td>
</tr>
<tr>
<td>256</td>
<td>19931</td>
<td>27703</td>
</tr>
<tr>
<td>512</td>
<td>42357</td>
<td>59235</td>
</tr>
<tr>
<td>1024</td>
<td>90143</td>
<td>126751</td>
</tr>
<tr>
<td>2048</td>
<td>191033</td>
<td>270155</td>
</tr>
</tbody>
</table>
12.3.9 rifft

Table 12-9 shows the approximate worst-case instruction cycle counts (= oscillator clocks/2) for the ASM code. For details on options for FFT, see Section 11.3.15.

<table>
<thead>
<tr>
<th>Size of Real IFFT, N</th>
<th>Core Real IFFT</th>
<th>Core Real IFFT + API</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>AS</td>
<td>BFP</td>
</tr>
<tr>
<td>8</td>
<td>447</td>
<td>638</td>
</tr>
<tr>
<td>16</td>
<td>980</td>
<td>1317</td>
</tr>
<tr>
<td>32</td>
<td>2081</td>
<td>2791</td>
</tr>
<tr>
<td>64</td>
<td>4448</td>
<td>5401</td>
</tr>
<tr>
<td>128</td>
<td>9450</td>
<td>12681</td>
</tr>
<tr>
<td>256</td>
<td>20204</td>
<td>27453</td>
</tr>
<tr>
<td>512</td>
<td>42950</td>
<td>59177</td>
</tr>
<tr>
<td>1024</td>
<td>91376</td>
<td>127461</td>
</tr>
<tr>
<td>2048</td>
<td>193546</td>
<td>273169</td>
</tr>
</tbody>
</table>
Chapter 13
License

13.1 Limited Use License Agreement

LIMITED USE LICENSE AGREEMENT

PLEASE READ THIS AGREEMENT CAREFULLY BEFORE USING THIS SOFTWARE. BY USING OR COPYING THE SOFTWARE, YOU AGREE TO THE TERMS OF THIS AGREEMENT.

The software in either source code form ("Source") or object code form ("Object") (cumulatively hereinafter "Software") is provided under a license agreement ("Agreement") as described herein. Any use of the Software including copying, modifying, or installing the Software so that it is usable by or accessible by a central processing unit constitutes acceptance of the terms of the Agreement by the person or persons making such use or, if employed, the employer thereof ("Licensee") and if employed, the person(s) making such use hereby warrants that he has the authority of their employer to enter this license agreement,. If Licensee does not agree with and accept the terms of this Agreement, Licensee must return or destroy any media containing the Software or materials related thereto, and destroy all copies of the Software.

The Software is licensed to Licensee by Motorola Incorporated ("Motorola") for use under the terms of this Agreement. Motorola retains ownership of the Software. Motorola grants only the rights specifically granted in this Agreement and grants no other rights. Title to the Software, all copies thereof and all rights therein, including all rights in any intellectual property including patents, copyrights, and trade secrets applicable thereto, shall remain vested in Motorola.

For the Source, Motorola grants Licensee a personal, non-exclusive, non-assignable, revocable, royalty-free right to use, copy, and make derivatives of the Source solely in a development system environment in order to produce object code solely for operating on a Motorola semiconductor device having a central processing unit ("Derivative Object").

For the Object and Derivative Object, Motorola grants Licensee a personal, non-exclusive, non-assignable, revocable, royalty-free right to copy, use, and distribute the Object and the Derivative Object solely for operating on a Motorola semiconductor device having a central processing unit.

Licensee agrees to: (a) not use, modify, or copy the Software except as expressly provided herein, (b) not distribute, disclose, transfer, sell, assign, rent, lease, or otherwise make available the Software, any derivatives thereof, or this license to a third party except as expressly provided herein, (c) not remove obliterate, or otherwise defeat any copyright, trademark, patent or proprietary notices, related to the
Software (d) not in any form export, re-export, resell, ship or divert or cause to be exported, re-exported, resold, shipped, or diverted, directly or indirectly, the Software or a direct product thereof to any country which the United States government or any agency thereof at the time of export or re-export requires an export license or other government approval without first obtaining such license or approval.

THE SOFTWARE IS PROVIDED ON AN "AS IS" BASIS AND WITHOUT WARRANTY OF ANY KIND INCLUDING (WITHOUT LIMITATION) ANY WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. IN NO EVENT SHALL MOTOROLA BE LIABLE FOR ANY LIABILITY OR DAMAGES OF ANY KIND INCLUDING, WITHOUT LIMITATION, DIRECT OR INDIRECT OR INCIDENTAL OR CONSEQUENTIAL OR PUNITIVE DAMAGES OR LOST PROFITS OR LOSS OF USE ARISING FROM USE OF THE SOFTWARE OR THE PRODUCT REGARDLESS OF THE FORM OF ACTION OR THEORY OF LIABILITY (INCLUDING WITHOUT LIMITATION, ACTION IN CONTRACT, NEGLIGENCE, OR PRODUCT LIABILITY) EVEN IF MOTOROLA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE. THIS DISCLAIMER OF WARRANTY Extends to LICENSEE or USERS OF PRODUCTS AND IS IN LIEU OF ALL WARRANTIES WHETHER EXPRESS, IMPLIED, OR STATUTORY, INCLUDING IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR PARTICULAR PURPOSE.

Motorola does not represent or warrant that the Software is free of infringement of any third party patents, copyrights, trade secrets, or other intellectual property rights or that Motorola has the right to grant the licenses contained herein. Motorola does not represent or warrant that the Software is free of defect, or that it meets any particular requirements or need of the Licensee, or that it conforms to any documentation, or that it meets any standards.

Motorola shall not be responsible to maintain the Software, provide upgrades to the Software, or provide any field service of the Software. Motorola reserves the right to make changes to the Software without further notice to Licensee.

The Software is 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 Software could create a situation where personal injury or death may occur. Should Licensee purchase or use the Software for any such unintended or unauthorized application, Licensee shall indemnify and hold Motorola 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 Motorola was negligent regarding the design or manufacture of the Software.

The term of this Agreement is for as long as Licensee uses the Software for its intended purpose and is not in default of any provisions of this Agreement. Motorola may terminate this Agreement if Licensee is in default of any of the terms and conditions of this Agreement.

This Agreement shall be governed by and construed in accordance with the laws of the State of Arizona and can only be modified in a writing signed by both parties. Licensee agrees to jurisdiction and venue in the State of Arizona.

By using, modifying, installing, compiling, or copying the Software, Licensee acknowledges that this Agreement has been read and understood and agrees to be bound by its terms and conditions. Licensee agrees that this Agreement is the complete and exclusive statement of the agreement between Licensee and Motorola and supersedes any earlier proposal or prior arrangement, whether oral or written, and any other communications relative to the subject matter of this Agreement.
Symbols
__fixed__ 3-1
__longfixed__ 3-1
__shortfixed__ 3-1

A
A xix
  Accumulator Register A xix
A2 xix
  4-bit accumulator A extension register xix

ABS xix
  Absolute Value xix
abs_s 6-5
absolute value 8-5

afr xix
  Single-dimensional array, fractional xix
afr16.h 5-1, 8-1
afr16Abs 8-5
afr16Add 8-6
afr16Div 8-7
afr16Equal 8-9, 9-6
afr16Mac_r 8-10
afr16Max 8-12
afr16Min 8-13
afr16Msu_r 8-14
afr16Mult 8-16
afr16Mult_r 8-16
afr16Negate 8-17
afr16Rand 8-18
afr16Sqr 8-20
afr16Sub 8-21
afr32.h 5-1, 8-1, 8-2
afr32Abs 8-5
afr32Add 8-6
afr32Div 8-7
afr32Equal 8-9
afr32Mac 8-10
afr32Mac_r 8-10
afr32Max 8-12
afr32Min 8-13
afr32Msu 8-14
afr32Msu_r 8-14
afr32Mult 8-16
afr32Mult_ls 8-16
afr32Negate 8-17
afr32Round 8-19
afr32Sqrt 8-20
afr32Sub 8-21
arch.h 5-1

B
B xix
  Accumulator Register B xix
B2 xix
  4-bit accumulator B extension register xix
Bit Reverse 11-9

C
CFrac16 3-2
CFrac32 3-2
Complex Fast Fourier Transform 11-10
Complex Inverse Fast Fourier Transform 11-19
Complex Inverse FFT 11-22, 11-24
Correlation 11-41
Create Complex FFT 11-13, 11-16
Create Finite Impulse Response Filter 11-48
Create IIR Filter 11-69
Create Interpolating FIR Filter 11-62
Create Real FFT 11-31
Create Real Inverse FFT 11-39
Creating Decimating FIR Filter 11-57

D
Decimating FIR Filter 11-55
Destroy Complex FFT 11-15
Destroy Decimating FIR Filter 11-58
Destroy IIR Filter 11-71
Destroy Interpolating FIR Filter 11-63
Destroy Real FFT 11-33
Destroy Real Inverse FFT 11-40
determinant 10-5
dfr xix
  Digital signal processing, fractional xix
dfr16.h 5-1, 11-1
dfr16AutoCorr 11-7
dfr16Cbitrev 11-9
dfr16CFFT 11-10
dfr16CFFTCreate 11-13
dfr16CFFTDestroy 11-15
dfr16CIFFT 11-19
dfr16CIFFTCreate 11-22
dfr16CIFFTDestroy 11-24
dfr16Corr 11-41
dfr16FIR 11-43
<table>
<thead>
<tr>
<th>Function</th>
<th>Page Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>dfr16FIRCreate</td>
<td>11-48</td>
</tr>
<tr>
<td>dfr16FIRDec</td>
<td>11-55</td>
</tr>
<tr>
<td>dfr16FIRDecCreate</td>
<td>11-57</td>
</tr>
<tr>
<td>dfr16FIRDecDestroy</td>
<td>11-58</td>
</tr>
<tr>
<td>dfr16FIRDecInit</td>
<td>11-59</td>
</tr>
<tr>
<td>dfr16FIRDestroy</td>
<td>11-50</td>
</tr>
<tr>
<td>dfr16FIRHistory</td>
<td>11-51</td>
</tr>
<tr>
<td>dfr16FIRInit</td>
<td>11-53</td>
</tr>
<tr>
<td>dfr16FIRInt</td>
<td>11-61</td>
</tr>
<tr>
<td>dfr16FIRIntCreate</td>
<td>11-62</td>
</tr>
<tr>
<td>dfr16FIRIntDestroy</td>
<td>11-63</td>
</tr>
<tr>
<td>dfr16FIRIntInit</td>
<td>11-64</td>
</tr>
<tr>
<td>dfr16FIRs</td>
<td>11-46</td>
</tr>
<tr>
<td>dfr16IIR</td>
<td>11-66</td>
</tr>
<tr>
<td>dfr16IIRCreate</td>
<td>11-69</td>
</tr>
<tr>
<td>dfr16IIRDestory</td>
<td>11-71</td>
</tr>
<tr>
<td>dfr16RFFT</td>
<td>11-28</td>
</tr>
<tr>
<td>dfr16RFFTCreate</td>
<td>11-31</td>
</tr>
<tr>
<td>dfr16RFFTDestory</td>
<td>11-33</td>
</tr>
<tr>
<td>dfr16RIFFT</td>
<td>11-36</td>
</tr>
<tr>
<td>dfr16RIFFTCreate</td>
<td>11-39</td>
</tr>
<tr>
<td>dfr16RIFFTDestory</td>
<td>11-40</td>
</tr>
<tr>
<td>div_LS</td>
<td>6-7</td>
</tr>
<tr>
<td>div_s</td>
<td>6-7</td>
</tr>
<tr>
<td>divide</td>
<td>8-7</td>
</tr>
<tr>
<td>DSP</td>
<td>xix</td>
</tr>
</tbody>
</table>

Digital Signal Processor xix
DSP56800 Family Manual xx
DSP56824 User’s Manual xx
dspfunc.h 1-2, 5-1
dspfunc.lib 4-4
dspfunc.mcp 4-4

E
Embedded SDK Programmer’s Guide xx
extract.h 6-9
extract_l 6-9

F
FFT xix
FFT xix
Fast Fourier Transforms xix
Finite Impulse Response Filter 11-43
FIR xix
Finite Impulse Response xix
FIR Filter for a Single Input 11-46
FIR History Buffer 11-51
Frac16 3-2
Frac32 3-2

I
I/O xix
Input/Output xix
IDE xix
Integrated Development Environment xix

IIR xix
IIR xix
Infinite Impulse Response xix
Infinite Impulse Response Filter 11-66
Initialize Complex FFT 11-34
Initialize Complex Inverse FFT 11-25
Initialize Decimating FIR Filter 11-59
Initialize Finite Impulse Response Filter 11-53
Initialize IIR Filter 11-72
Initialize Interpolating FIR Filter 11-64
Integer Delta Direct Look-up 7-13
Interpolating FIR Filter 11-61
inverse 10-7

J
JTAG xix
Joint Test Action Group xix

L
L xix
Limit Bit xix
L_abs 6-5
L_add 6-6
L_deposit_h 6-8
L_deposit_l 6-8
L_mac 6-10
L_msu 6-11
L_mult 6-12
L_mult_ls 6-12
L_negate 6-13
L_shl 6-18
L_shr 6-19
L_shr_r 6-19
L_sub 6-21
LSB xix
Least Significant Bit xix

M
MAC xix
MAC xix
Multiply/Accumulate xix
mac_r 6-10
maximum 8-12
Memory 2-5
MetroWerks CodeWarrior 1-2
mfr xix
Basic Math, Fractional xix
mfr16.h 5-1, 6-1
mfr16Rand 6-15
mfr16SetRandSeed 6-16
mfr16Sqrt 6-20
mfr32.h 5-1, 6-2
mfr32Sqrt 6-20
minimum 8-13
MIPS xix
Million Instructions Per Second xix
MSB xix
  Most Significant Bit xix
msu_r 6-11
mult 6-12
mult_r 6-12
multiple and accumulate 8-10
multiply 8-16, 10-8
multiply and subtract 8-14

N
naming convention 5-1
norm_1 6-14
norm_s 6-14

O
OMR xix
  Operating Mode Register xix
OnCE xix
  On-Chip Emulation xix

P
PC xix
  Personal Computer xix
Polynomial Approximation 7-25
port.h 3-2, 5-1
prototype.h 5-1
pX xix
  Pointer to variable X xix
pY xix
  Pointer to variable Y xix
pZ xix
  Pointer to variable Z xix

R
R xx
  Rounding Bit xx
R0, R1, R2, R3 xx
  16-bit Address Registers xx
Real Delta Direct Look-up 7-16
Real Delta Loop-up with Interpolation 7-19, 7-22, 7-31
Real Fast Fourier Transform 11-28
Real Inverse Fast Fourier Transform 11-36

S
SA xx
  Saturation Bit xx
saturation mode 2-6
SDK xx, 4-1
  Software Development Kit xx
shl 6-18
shr 6-19
shr_r 6-19
sine wave generation 7-13, 7-16, 7-19, 7-22, 7-25, 7-28, 7-31
single-dimensional arrays 8-1
Software Development Kit 1-2
SP xx
  Stack Pointer xx
SPI xx
  Serial Peripheral Interface xx
SR xx
  Status Register xx
SRC xx
  Source xx
SSI xx
  Synchronous Serial Interface xx
subtract 10-10

T
tfr xx
  Trigonometric, fractional xx
tfr16.h 5-1, 7-1
tfr16AcosOverPI 7-7
tfr16AsinOverPI 7-8
tfr16Atan2OverPI 7-10
tfr16AtanOverPI 7-9
tfr16CosPIx 7-11
tfr16SineWaveGenDOMinit 7-28
tfr16SineWaveGenDTLinit 7-13
tfr16SineWaveGenPAMinit 7-25
tfr16SineWaveGenRDITLinit 7-19, 7-22, 7-31
tfr16SineWaveGenRDTLinit 7-16
tfr16SinPIx 7-12
transpose 10-12
trigonometric 7-1

V
V xx
  Overflow bit xx
vfr xx
  Vector, fractional xx
vfr16.h 5-1, 9-2
vfr16Add 9-4
vfr16DotProd 9-5
vfr16Equal 9-6
vfr16Length 9-7
vfr16Mult 9-8
vfr16Scale 9-9
vfr16Sub 9-10
vfr32.h 5-1

X
x xx
  Input variable xx
X0 xx
  Input register X0 xx
xfr xx
  Matrix, fractional xx
xfr16.h 5-1, 10-1
xfr16Add 10-4
xfr16Det 10-5
xfr16Equal 10-6
xfr16Inv 10-7
xfr16Mult 10-8
xfr16Sub 10-10
xfr16Trans 10-12
xfr32.h 5-1

Y

y xx
   Input variable xx
Y0 xx
   Input register Y0 xx
Y1 xx
   Input register Y1 xx

Z

z xx
   Output variable xx