LEGAL DISCLAIMER

THE INFORMATION GIVEN IN THIS APPLICATION NOTE IS GIVEN AS A HINT FOR THE IMPLEMENTATION OF THE INFINEON TECHNOLOGIES COMPONENT ONLY AND SHALL NOT BE REGARDED AS ANY DESCRIPTION OR WARRANTY OF A CERTAIN FUNCTIONALITY, CONDITION OR QUALITY OF THE INFINEON TECHNOLOGIES COMPONENT. THE RECIPIENT OF THIS APPLICATION NOTE MUST VERIFY ANY FUNCTION DESCRIBED HEREIN IN THE REAL APPLICATION. INFINEON TECHNOLOGIES HEREBY DISCLAIMS ANY AND ALL WARRANTIES AND LIABILITIES OF ANY KIND (INCLUDING WITHOUT LIMITATION WARRANTIES OF NON-INFRINGEMENT OF INTELLECTUAL PROPERTY RIGHTS OF ANY THIRD PARTY) WITH RESPECT TO ANY AND ALL INFORMATION GIVEN IN THIS APPLICATION NOTE.

Information

For further information on technology, delivery terms and conditions and prices please contact your nearest Infineon Technologies Office (www.infineon.com).

Warnings

Due to technical requirements components may contain dangerous substances. For information on the types in question please contact your nearest Infineon Technologies Office.

Infineon Technologies Components may only be used in life-support devices or systems with the express written approval of Infineon Technologies, if a failure of such components can reasonably be expected to cause the failure of that life-support device or system, or to affect the safety or effectiveness of that device or system. Life support devices or systems are intended to be implanted in the human body, or to support and/or maintain and sustain and/or protect human life. If they fail, it is reasonable to assume that the health of the user or other persons may be endangered.
XC166Lib
A DSP Library for XC16x Family

Guangyu Wang

AIM MC ATV AT (Munich, Germany)
Revision History: 2003-06 V 1.1
Previous Version: 2002-02 V 1.0, 2002-09 V 1.1

Page | Subjects (major changes since last revision)
--- | ---
--- | ---
--- | ---
--- | ---
--- | ---

We Listen to Your Comments
Any information within this document that you feel is wrong, unclear or missing at all? Your feedback will help us to continuously improve the quality of this document. Please send your proposal (including a reference to this document) to:

C166Lib-support@infineon.com
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1</strong> Introduction</td>
<td>13</td>
</tr>
<tr>
<td>1.1 Introduction to XC166Lib, a DSP Library for XC16x</td>
<td>13</td>
</tr>
<tr>
<td>1.2 Features</td>
<td>13</td>
</tr>
<tr>
<td>1.3 New Features of Version 1.1</td>
<td>13</td>
</tr>
<tr>
<td>1.4 Future of the XC166Lib</td>
<td>13</td>
</tr>
<tr>
<td>1.5 Support Information</td>
<td>14</td>
</tr>
<tr>
<td><strong>2</strong> Installation and Build</td>
<td>15</td>
</tr>
<tr>
<td>2.1 XC166Lib Content</td>
<td>15</td>
</tr>
<tr>
<td>2.2 Installing XC166Lib</td>
<td>15</td>
</tr>
<tr>
<td>2.3 Building XC166Lib</td>
<td>15</td>
</tr>
<tr>
<td>2.4 Source Files List</td>
<td>16</td>
</tr>
<tr>
<td><strong>3</strong> DSP Library Notations</td>
<td>18</td>
</tr>
<tr>
<td>3.1 XC166Lib Data Types</td>
<td>18</td>
</tr>
<tr>
<td>3.2 Calling a DSP Library Function from C Code</td>
<td>18</td>
</tr>
<tr>
<td>3.3 Calling a DSP Library Function from Assembly Code</td>
<td>18</td>
</tr>
<tr>
<td>3.4 XC166Lib Example Implementation</td>
<td>18</td>
</tr>
<tr>
<td>3.5 XC166Lib Implementation - A Technical Note</td>
<td>19</td>
</tr>
<tr>
<td>3.5.1 Memory Issues</td>
<td>19</td>
</tr>
<tr>
<td>3.5.2 Memory Models for C Compiler</td>
<td>20</td>
</tr>
<tr>
<td>3.5.3 Optimization Techniques</td>
<td>21</td>
</tr>
<tr>
<td>3.5.4 Cycle Count</td>
<td>23</td>
</tr>
<tr>
<td>3.5.5 Testing Methodology</td>
<td>23</td>
</tr>
<tr>
<td><strong>4</strong> Function Descriptions</td>
<td>24</td>
</tr>
<tr>
<td>4.1 Conventions</td>
<td>24</td>
</tr>
<tr>
<td>4.1.1 Argument Conventions</td>
<td>24</td>
</tr>
<tr>
<td>4.2 Arithmetic Functions</td>
<td>26</td>
</tr>
<tr>
<td>4.2.1 Complex Numbers</td>
<td>26</td>
</tr>
<tr>
<td>4.2.2 Complex Number Representation</td>
<td>26</td>
</tr>
<tr>
<td>4.2.3 Complex Plane</td>
<td>26</td>
</tr>
<tr>
<td>4.2.4 Complex Arithmetic</td>
<td>27</td>
</tr>
<tr>
<td>4.2.4.1 Addition</td>
<td>27</td>
</tr>
<tr>
<td>4.2.4.2 Subtraction</td>
<td>27</td>
</tr>
<tr>
<td>4.2.4.3 Multiplication</td>
<td>27</td>
</tr>
<tr>
<td>4.2.4.4 Conjugate</td>
<td>28</td>
</tr>
<tr>
<td>4.2.5 Complex Number Schematic</td>
<td>28</td>
</tr>
<tr>
<td>4.2.6 Descriptions</td>
<td>28</td>
</tr>
<tr>
<td>4.3 FIR Filters</td>
<td>40</td>
</tr>
<tr>
<td>4.3.1 Descriptions</td>
<td>41</td>
</tr>
<tr>
<td>4.4 IIR Filters</td>
<td>49</td>
</tr>
<tr>
<td>4.4.1 Descriptions</td>
<td>52</td>
</tr>
<tr>
<td>4.5 Adaptive Digital Filters</td>
<td>71</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
</tr>
<tr>
<td>4.5.1</td>
<td>Delayed LMS algorithm for an adaptive filter</td>
</tr>
<tr>
<td>4.5.2</td>
<td>Descriptions</td>
</tr>
<tr>
<td>4.6</td>
<td>Fast Fourier Transforms</td>
</tr>
<tr>
<td>4.6.1</td>
<td>Radix-2 Decimation-In-Time FFT Algorithm</td>
</tr>
<tr>
<td>4.6.2</td>
<td>Radix-2 Decimation-In-Frequency FFT Algorithm</td>
</tr>
<tr>
<td>4.6.3</td>
<td>Complex FFT Algorithm</td>
</tr>
<tr>
<td>4.6.4</td>
<td>Calculation of Real Forward FFT from Complex FFT</td>
</tr>
<tr>
<td>4.6.5</td>
<td>Calculation of Real Inverse FFT from Complex FFT</td>
</tr>
<tr>
<td>4.7</td>
<td>XC16x Implementation Note</td>
</tr>
<tr>
<td>4.7.1</td>
<td>Organization of FFT functions</td>
</tr>
<tr>
<td>4.7.2</td>
<td>Implementation of Real Forward FFT</td>
</tr>
<tr>
<td>4.7.3</td>
<td>Implementation of Real Inverse FFT</td>
</tr>
<tr>
<td>4.7.4</td>
<td>Description</td>
</tr>
<tr>
<td>4.8</td>
<td>Matrix Operations</td>
</tr>
<tr>
<td>4.8.1</td>
<td>Descriptions</td>
</tr>
<tr>
<td>4.9</td>
<td>Mathematical Operations</td>
</tr>
<tr>
<td>4.10</td>
<td>Statistical Functions</td>
</tr>
<tr>
<td>4.10.1</td>
<td>Correlation</td>
</tr>
<tr>
<td>4.10.1.1</td>
<td>Definitions of Correlation</td>
</tr>
<tr>
<td>4.10.1.2</td>
<td>Implementation Note</td>
</tr>
<tr>
<td>4.10.2</td>
<td>Implementation Description</td>
</tr>
<tr>
<td>5</td>
<td>Integrated Test Process</td>
</tr>
<tr>
<td>5.1</td>
<td>Structure of The Test Process</td>
</tr>
<tr>
<td>5.2</td>
<td>Building The Integrated Test Process</td>
</tr>
<tr>
<td>6</td>
<td>References</td>
</tr>
<tr>
<td>7</td>
<td></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 4-1</td>
<td>The Complex Plane</td>
<td>28</td>
</tr>
<tr>
<td>Figure 4-2</td>
<td>16-bit Complex number representation</td>
<td>29</td>
</tr>
<tr>
<td>Figure 4-3</td>
<td>Complex Number addition for 16 bits</td>
<td>31</td>
</tr>
<tr>
<td>Figure 4-4</td>
<td>Complex number subtraction for 16 bits</td>
<td>34</td>
</tr>
<tr>
<td>Figure 4-5</td>
<td>Complex number multiplication for 16 bits</td>
<td>37</td>
</tr>
<tr>
<td>Figure 4-6</td>
<td>32 bit real number multiplication</td>
<td>40</td>
</tr>
<tr>
<td>Figure 4-7</td>
<td>Block Diagram of the FIR Filter</td>
<td>42</td>
</tr>
<tr>
<td>Figure 4-8</td>
<td>FIr_16</td>
<td>44</td>
</tr>
<tr>
<td>Figure 4-9</td>
<td>FIr_32</td>
<td>48</td>
</tr>
<tr>
<td>Figure 4-10</td>
<td>Canonical Form (Direct Form 2)</td>
<td>51</td>
</tr>
<tr>
<td>Figure 4-11</td>
<td>Cascaded Biquad IIR Filter with Direct Form 2 Implementation</td>
<td>52</td>
</tr>
<tr>
<td>Figure 4-12</td>
<td>Cascaded Biquad IIR Filter with Direct Form 1 Implementation</td>
<td>53</td>
</tr>
<tr>
<td>Figure 4-13</td>
<td>IIR_1</td>
<td>56</td>
</tr>
<tr>
<td>Figure 4-14</td>
<td>IIR_2</td>
<td>60</td>
</tr>
<tr>
<td>Figure 4-15</td>
<td>IIR_bi_1</td>
<td>65</td>
</tr>
<tr>
<td>Figure 4-16</td>
<td>IIR_bi_2</td>
<td>70</td>
</tr>
<tr>
<td>Figure 4-17</td>
<td>Adaptive filter with LMS algorithm</td>
<td>72</td>
</tr>
<tr>
<td>Figure 4-18</td>
<td>Adap_filter_16</td>
<td>78</td>
</tr>
<tr>
<td>Figure 4-19</td>
<td>Adp_filter_32</td>
<td>83</td>
</tr>
<tr>
<td>Figure 4-20</td>
<td>Complexity Graph</td>
<td>88</td>
</tr>
<tr>
<td>Figure 4-21</td>
<td>8-point DIT FFT</td>
<td>90</td>
</tr>
<tr>
<td>Figure 4-22</td>
<td>Alternate Form of 8-point DIT FFT</td>
<td>91</td>
</tr>
<tr>
<td>Figure 4-23</td>
<td>8-point DIF FFT</td>
<td>92</td>
</tr>
<tr>
<td>Figure 4-24</td>
<td>Alternate Form of 8-point DIF FFT</td>
<td>93</td>
</tr>
<tr>
<td>Figure 4-25</td>
<td>8-point DIT complex FFT</td>
<td>94</td>
</tr>
<tr>
<td>Figure 4-26</td>
<td>Butterfly of Radix-2 DIT complex FFT</td>
<td>94</td>
</tr>
<tr>
<td>Figure 4-27</td>
<td>8-point DIF complex FFT</td>
<td>95</td>
</tr>
<tr>
<td>Figure 4-28</td>
<td>Butterfly of Radix-2 DIF complex FFT</td>
<td>96</td>
</tr>
<tr>
<td>Figure 4-29</td>
<td>Bit_reverse</td>
<td>103</td>
</tr>
<tr>
<td>Figure 4-30</td>
<td>FloatTo1Q15</td>
<td>106</td>
</tr>
<tr>
<td>Figure 4-31</td>
<td>Real_DIT_FFT</td>
<td>110</td>
</tr>
<tr>
<td>Figure 4-32</td>
<td>Real_DIF_FFT</td>
<td>116</td>
</tr>
<tr>
<td>Figure 4-33</td>
<td>Matrix</td>
<td>122</td>
</tr>
<tr>
<td>Figure 4-34</td>
<td>Sine</td>
<td>127</td>
</tr>
<tr>
<td>Figure 4-35</td>
<td>P_series</td>
<td>130</td>
</tr>
<tr>
<td>Figure 4-36</td>
<td>Windowing</td>
<td>134</td>
</tr>
<tr>
<td>Figure 4-37</td>
<td>Raw autocorrelation</td>
<td>142</td>
</tr>
<tr>
<td>Figure 4-38</td>
<td>Biased autocorrelation</td>
<td>146</td>
</tr>
<tr>
<td>Figure 4-39</td>
<td>Unbiased autocorrelation</td>
<td>150</td>
</tr>
<tr>
<td>Figure 4-40</td>
<td>Raw cross-correlation</td>
<td>155</td>
</tr>
<tr>
<td>Figure 4-41</td>
<td>Biased cross-correlation</td>
<td>161</td>
</tr>
<tr>
<td>Figure 4-42</td>
<td>Unbiased cross-correlation</td>
<td>167</td>
</tr>
</tbody>
</table>
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 2-1</td>
<td>Directory Structure</td>
<td>15</td>
</tr>
<tr>
<td>Table 2-2</td>
<td>Source files</td>
<td>16</td>
</tr>
<tr>
<td>Table 3-1</td>
<td>XC166Lib Data Types</td>
<td>18</td>
</tr>
<tr>
<td>Table 4-1</td>
<td>Argument Conventions</td>
<td>24</td>
</tr>
<tr>
<td>Table 5-1</td>
<td>Directory structure of the test process</td>
<td>169</td>
</tr>
<tr>
<td>Table 5-2</td>
<td>List of source files of the integrated test process</td>
<td>169</td>
</tr>
</tbody>
</table>
Preface

This is the User Manual of version 1.1 for XC166Lib - a DSP library for Infineon XC16x microcontroller. The version 1.0 was released on April 2002.

XC16x microcontroller is the most recent generation of the popular C166 microcontroller families. It combines high performance with enhanced modular architecture. Impressive DSP performance and advanced interrupt handling with fast context switching make XC16x microcontroller the instrument of choice for powerful applications.

This manual describes the implementation of essential algorithms for general digital signal processing applications on the XC16x microcontroller.

The source codes are assembly files and this library can be used as a library of basic functions for developing bigger DSP applications on XC16x microcontroller. The library serves as a user guide and a reference for XC16x microcontroller DSP programmers. It demonstrates how the processor’s architecture can be exploited for achieving high performance.

The various functions and algorithms implemented and described are:

- Arithmetic Functions
- Filters
  - FIR
  - IIR
- Adaptive filters
- Transforms
  - FFT
  - IFFT
- Matrix Operations
- Mathematical Operations
- Statistical Functions

Each function is described in detail under the following headings:

**Signature:**
This gives the function interface.

**Inputs:**
Lists the inputs to the function.

**Outputs:**
Lists the output of the function if any.

**Return:**
Gives the return value of the function if any.

**Implementation Description:**
Gives a brief note on the implementation, the size of the inputs and the outputs, alignment requirements etc.

**Pseudocode:**
The implementation is expressed as a pseudocode using C conventions.

**Techniques:**
The techniques employed for optimization are listed here.

**Register Usage:**
Lists the registers that are used for parameter transfer from C to Assembly or inverse.

**Assumptions:**
Lists the assumptions made for an optimal implementation such as constraint on DPRAM. The input output formats are also given here.

**Memory Note:**
A detailed sketch showing how the arrays are stored in memory, the alignment requirements of the different memories, the nature of the arithmetic performed on them. The diagrams give a great insight into the actual implementation.

Further, the path of an Example calling program, the Cycle Count and Code Size are given for each function.
Organization

Chapter 1, Introduction, gives a brief introduction of the XC166Lib and its features.

Chapter 2, Installation and Build, describes the XC166Lib content, how to install and build the XC166Lib.

Chapter 3, DSP Library Notations, describes the DSP Library data types, arguments, calling a function from the C code and the assembly code, and the implementation notes.

Chapter 4, Function Descriptions, describes the arithmetic functions, FIR filters, IIR filters, Adaptive filters, Fast Fourier Transforms, Matrix operations and Mathematical operations. Each function is described with its signature, inputs, outputs, return, implementation description, pseudocode, techniques used, registers used for parameter transfer, assumptions made, memory note, example, cycle count and code size.

Chapter 5, References, gives the list of related references.

Acknowledgements

The user’s manual for XC166Lib is developed based on the user's manual for TriLib-A DSP library for TriCore. All source codes in XC166Lib have been designed, developed and tested using Tasking Tool chains. We in advance would like to acknowledge users for their feedback and suggestions to improve this product.

Guangyu Wang
XC166Lib Developer - Infineon
Acronyms and Definitions

<table>
<thead>
<tr>
<th>Acronyms</th>
<th>Definitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIT</td>
<td>Decimation-In-Time</td>
</tr>
<tr>
<td>DIF</td>
<td>Decimation-In-Frequency</td>
</tr>
<tr>
<td>LMS</td>
<td>Least Mean Square</td>
</tr>
<tr>
<td>DSP</td>
<td>Digital Signal Processing</td>
</tr>
<tr>
<td>XC166Lib</td>
<td>DSP Library functions for XC16x microcontroller</td>
</tr>
<tr>
<td>FFT</td>
<td>Fast Fourier Transform</td>
</tr>
<tr>
<td>FIR</td>
<td>Finite Impulse Response</td>
</tr>
<tr>
<td>IIR</td>
<td>Infinite Impulse Response</td>
</tr>
</tbody>
</table>

Documentation/Symbol Conventions

The following is the list of documentation/symbol conventions used in this manual.

<table>
<thead>
<tr>
<th>Documentation/Symbol Conventions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Courier</td>
<td>Pseudocode</td>
</tr>
<tr>
<td><em>Times-italic</em></td>
<td>File name</td>
</tr>
</tbody>
</table>
1 Introduction

1.1 Introduction to XC166Lib, a DSP Library for XC16x microcontroller family

The XC166Lib, a DSP Library for XC16x microcontroller is C-callable, hand-coded assembly, general purpose signal processing routines. The XC166Lib includes commonly used DSP routines. The throughput of the system using the XC166Lib routines is considerably better than those achieved using the equivalent code written in ANSI C language. The XC166Lib significantly helps in understanding the general purpose signal processing routines, its implementation on XC16x microcontroller. It also reduces the DSP application development time. Furthermore, The XC166Lib is also a good reference for XC16x microcontroller DSP programmer.

The routines are broadly classified into the following functional categories:

- Arithmetic functions
- FIR filters
- IIR filters
- Adaptive filters
- Fast Fourier Transforms
- Matrix operations
- Mathematical operations
- Statistical functions

1.2 Features

- Common DSP algorithms with source codes
- Hand-coded and optimized assembly modules with CoMAC instructions
- C-callable functions on Tasking compiler
- Multi platform support - Win 95, Win 98, Win NT
- Examples to demonstrate the usage of functions
- Complete User’s manual covering many aspects of implementation

1.3 New Features of Version 1.1

- Extension to large memory model
- Improvement of the source code structure
- Provision of an integrated test process to test XC166Lib together or separately
- 8 new functions in the areas of FFT, mathematical functions and statistical functions

1.4 Future of the XC166Lib

The planned future releases will have the following improvements.
• Expansion of the library by adding more functions in the domain of generic core routines of DSP.
• Upgrading the existing 16 bit functions to 32 bit

1.5 Support Information

Any suggestions for improvement, bug report if any, can be sent via e-mail to C166Lib-support@infineon.com.

Visit www.infineon.com /C166DSPLIB for update on XC166Lib releases.
2 Installation and Build

2.1 XC166Lib Content

The following table depicts the XC166Lib content with its directory structure.

<table>
<thead>
<tr>
<th>Directory name</th>
<th>Contents</th>
<th>Files</th>
</tr>
</thead>
<tbody>
<tr>
<td>C166Lib</td>
<td>Directories which has all the files related to the XC166Lib</td>
<td>None</td>
</tr>
<tr>
<td>Source</td>
<td>Directories of source files. Each directory has respective assembly language implementation files of the library functions</td>
<td>*.asm</td>
</tr>
<tr>
<td>Include</td>
<td>Directory and common include files for 'C' of the Tasking compiler</td>
<td>DspLib.h</td>
</tr>
<tr>
<td>Examples</td>
<td>Example directories. Each directory contains example &quot;c&quot; functions to depict the usage of XC166Lib.</td>
<td>*.c</td>
</tr>
</tbody>
</table>

2.2 Installing XC166Lib

XC166Lib is distributed as a ZIP file. To install the XC166Lib on the system, unzip the ZIP file and extract them to the defined directory.

The directory structure is as given in “XC166Lib Content” on Page 2-11

2.3 Building XC166Lib

Include the `DspLib.h` into your project and also include the same into the files that need to call the library function like:

```c
#include "DspLib.h"
```

Now include the respective source files for the required functionality into your project. Refer the functionality table, Table 2-2.

Build the system and start using the library.
2.4 Source Files List

<table>
<thead>
<tr>
<th>Table 2-2</th>
<th>Source files</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complier:</td>
<td>Tasking</td>
</tr>
<tr>
<td>Arithmetic Functions</td>
<td></td>
</tr>
<tr>
<td>CplxAdd.asm</td>
<td></td>
</tr>
<tr>
<td>CplxSub.asm</td>
<td></td>
</tr>
<tr>
<td>CplxMul.asm</td>
<td></td>
</tr>
<tr>
<td>Mul_32.asm</td>
<td></td>
</tr>
<tr>
<td>FIR Filters</td>
<td></td>
</tr>
<tr>
<td>Fir_16.asm</td>
<td></td>
</tr>
<tr>
<td>Fir_32.asm</td>
<td></td>
</tr>
<tr>
<td>IIR Filters</td>
<td></td>
</tr>
<tr>
<td>IIR_1.asm</td>
<td></td>
</tr>
<tr>
<td>IIR_2.asm</td>
<td></td>
</tr>
<tr>
<td>IIR_bi_1.asm</td>
<td></td>
</tr>
<tr>
<td>IIR_bi_2.asm</td>
<td></td>
</tr>
<tr>
<td>Adaptive Filters</td>
<td></td>
</tr>
<tr>
<td>Adap_filter_16.asm</td>
<td></td>
</tr>
<tr>
<td>Adap_filter_32.asm</td>
<td></td>
</tr>
<tr>
<td>FFT</td>
<td></td>
</tr>
<tr>
<td>Bit_reverse.asm</td>
<td></td>
</tr>
<tr>
<td>FloatTo1Q15.asm</td>
<td></td>
</tr>
<tr>
<td>real_DIT_FFT.asm</td>
<td></td>
</tr>
<tr>
<td>real_DIF_IFFT.asm</td>
<td></td>
</tr>
<tr>
<td>Matrix Operations</td>
<td></td>
</tr>
<tr>
<td>Matrix_mul.asm</td>
<td></td>
</tr>
<tr>
<td>Mathematical Operations</td>
<td></td>
</tr>
<tr>
<td>Power_series.asm</td>
<td></td>
</tr>
<tr>
<td>Windowing.asm</td>
<td></td>
</tr>
<tr>
<td>Sine.asm</td>
<td></td>
</tr>
<tr>
<td>Table 2-2</td>
<td>Source files</td>
</tr>
<tr>
<td>-------------</td>
<td>--------------</td>
</tr>
<tr>
<td></td>
<td>Statistical Functions</td>
</tr>
<tr>
<td></td>
<td>Auto_raw.asm</td>
</tr>
<tr>
<td></td>
<td>Auto_bias.asm</td>
</tr>
<tr>
<td></td>
<td>Auto_unbias.asm</td>
</tr>
<tr>
<td></td>
<td>Cross_raw.asm</td>
</tr>
<tr>
<td></td>
<td>Cross_bias.asm</td>
</tr>
<tr>
<td></td>
<td>Cross_unbias.asm</td>
</tr>
</tbody>
</table>
3 DSP Library Notations

3.1 XC166Lib Data Types
The XC166Lib handles the following fractional data types.

<table>
<thead>
<tr>
<th>Table 3-1</th>
<th>XC166Lib Data Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>1Q15 (DataS)</td>
<td>1Q15 operand is represented by a short data type that is predefined as DataS in header files DspLib.h.</td>
</tr>
<tr>
<td>1Q31 (DataL)</td>
<td>1Q31 operand is represented by a long data type that is predefined as DataL in header files DspLib.h.</td>
</tr>
<tr>
<td>CplxS</td>
<td>Complex data type that contains the two 1Q15 data arranged in Re-Im format.</td>
</tr>
<tr>
<td>CplxL</td>
<td>Complex data type that contains the two 1Q31 data arranged in Re-Im format.</td>
</tr>
</tbody>
</table>

3.2 Calling a DSP Library Function from C Code
After installing the XC166Lib, include a XC166Lib function in the source code as follows:
1. Choose the memory model, small or large for C compiler
2. Include the header file DspLib.h
3. Include the source file that contains required DSP functions into the project along with the other source files
4. Set the Project Options-CPU to select an MCU with XC16x microcontroller
5. If necessary, define a DPRAM class under EDE option Linker/Locator-Memory, e.g. DPRAM_CLASS(0f600h-0fe00h)
6. Build the system

3.3 Calling a DSP Library Function from Assembly Code
The XC166Lib functions are written to be used from C. Calling the functions from assembly language source code is possible as long as the calling function conforms to the DSP calling conventions. Refer Chapter 4, Function Descriptions, for more details.

3.4 XC166Lib Example Implementation
The examples of how to use the XC166Lib functions are implemented and are placed in examples subdirectory. This subdirectory contains a subdirectory for set of functions.
3.5 XC166Lib Implementation - A Technical Note

3.5.1 Memory Issues

The XC16x microcontroller is a 16-bit microcontroller core, with impressive DSP performance. There are two sets of instructions for XC16x microcontroller, namely Normal Instruction Set and DSP Instruction Set (MAC-instructions). Normal instruction set is compatible with the microcontroller family C166, while the DSP instruction set is especially designed for implementing DSP algorithms. XC166Lib was developed mainly using DSP instruction set. But the normal instruction set has been also often used in the routines, in particular, for initializing memories and registers.

For each instruction set there is a different addressing mode. DSP instructions use some standard C166 addressing modes such as GPR direct and #data for immediate shift value. To supply the MAC instructions with up to 2 new operands in one CPU cycle, new MAC instruction addressing models have been added in XC16x microcontroller. These allow indirect addressing with address pointer post-modification. Double indirect addressing requires 2 pointers, one of which can be supplied by any GPR, the other is provided by one of two Specific Function Registers (SFRs) IDX0 and IDX1. Two pairs of offset registers QR0/QR1 and QX0/QX1 are associated with each pointer (GPR or IDX). The GPR pointer gives access to the entire memory space, whereas IDX are limited to the internal Dual-Port RAM (DPRAM), except for the CoMOV instruction.

The XC166Lib is implemented with the XC16x microcontroller memory addressing architecture. The following information gives memory conditions in order to work properly.

Because the specific function register IDX is limited to the internal DPRAM, in order to use MAC instructions properly we must first initialize IDXi with the address pointed to DPRAM space from 00’F200 to 00’FE00 (3kBytes) before using MAC instructions. This means that we must locate one of operands in MAC-instructions with double indirect addressing modes in range from 00’F200 to 00’FE00. Using Tasking Compiler we have two possibilities to realize it:

- The simple way to locate a variable at a special address with Tasking tool chain is to use the _at() attribute provided by Tasking Compiler. For example, if we want to locate the vector x[N] with the size N into DPRAM area, we can define the vector in C program as

  ```c
  _near short x[n] _at(0x0f600)
  or _far short x[n] _at(0x0f600)
  ```

  After compiling the vector x will be located at the address 0x0f600 in the DPRAM area.

- Another way to realize it is to use the compiler control "pragma nb=DPRAM_CLASS" or "pragma fb=DPRAM_CLASS" to define a special memory class, then locate this memory class in the area 00’F200 to 00’FE00 with linker/locator control
"classes(DPRAM_CLASS (0f200h-0fE00h))". After doing this all variables with definition "_near" or "_far" will be located in the DPRAM area.

Note that XC16x microcontroller has defined 3 KBytes of DPRAM. However, the microcontroller XC161 and XC164 consist of a XC16x microcontroller with only 2 KBytes of DPRAM in the range from 00'F600H to 00'FE00H. The limited DPRAM area can make difficulty by executing DSP algorithms in such microcontrollers, if the size of the vector is larger than 2 KBytes, e.g. a 1024-point Fir filter.

When using pointer post-modification addressing models, the address pointed to must be a legal address, even if its content is not modified. An odd value will trigger the class-B hardware Trap (Illegal Word Operand Access Trap (ILLOPA)).

3.5.2 Memory Models for C Compiler

Just as we said, the DSP library is developed mainly for calling from a C program. Therefore, the memory modes selected by C and assembly modules must be same in order to avoid memory model conflicts. Tasking tool chain supports four memory models: tiny, small, medium and large. The basic differences between them are:

Tiny and small memory models use nonsegmented data, which means that the normal data in both memory models must be located in first segment with the size of less than 64K, and the assembler control "&SEGMENTED" is not allowed to be used in source file. In this case, when calling an assembly routine from C program only the 16 bit page offset of the parameter needs to be translated to assembly routine. The difference of tiny and small memory models is that the code size in tiny model is limited to first segment of 64K, while the small model allows code to locate anywhere in the space.

Medium and large memory models use segmented data and allow the assembler control "&SEGMENTED" to be used in the assembly program. In this case, calling an assembly routine from C program not only 16 bit page offset, but also page number DPP of the parameters have to be translated to assembly routine, which means that the address information of one parameter needs two registers to translate from C to assembly routine. This is different from the tiny and small memory models. The difference of the medium and large memory models is that the code size in medium model is limited to first segment of 64K, while the large model allows code to locate anywhere in the space.

The first version of XC166Lib v1.0 supports only small memory model. Now we have extended small to large memory model. The current version of XC166Lib v1.1 supports both small and large memory models.

There are only 16 registers R0-R15 in XC16x microcontroller that can be used for programming. If the XC166Lib routines are called from C, according to Tasking Compiler conventions the registers from R12 to R15 will be used to translate parameters from C to the assembly code, and the rest parameters will be translated through using stack. R4 and R5 are used to store the output values after executing the XC166Lib routines.
R0 is used by the Tasking Compiler as a pointer to stack. If it is possible, do not use R0 when programming.

### 3.5.3 Optimization Techniques

DSP optimization techniques depend strongly on the core architecture. So, different cores have different optimization techniques. Furthermore, the number of tricks that can be played by a DSP programmer are endless. In the development of XC166Lib the following optimization techniques have been used.

- **data dependencies removing**
  
  Due to the pipeline requirement of the C166S V2 CPU there are a lot of possible data dependencies between instructions using GPRs. In the XC16x microcontroller the dedicated hardware is added to detect and resolve the data dependencies. However, in the XC16x microcontroller none of the instructions using indirect addressing modes are capable of using a GPR, which is to be updated by one of the two immediately preceding instructions. This means that the instruction using indirect addressing modes will lead two cycles stall. To use these two cycles for the optimization we can insert before this instruction a multicycle or two single cycle instructions that must not update the GPR used for indirect addressing.

Example:

---

**Assembly without optimization** (6 cycles)

```
............
ADD     R1, R2
MOV    R8, [R1]    ; instruction using indirect addressing mode
ADD     R5, R1
ADD     R6, R1
............
```

**Assembly with optimization** (4 cycles)

```
............
ADD     R1, R2
ADD     R5, R1      ; inserted one cycle instruction
ADD     R6, R1      ; inserted one cycle instruction
MOV    R8, [R1]    ; instruction using indirect addressing mode
............
```

- **memory bandwidth conflicts removing**
  
  Memory bandwidth conflicts can occur if instructions in the pipeline access the same memory areas at the same time. The CoXXX instructions are specially designed for DSP implementation. To avoid the memory bandwidth conflicts in the DPRAM areas, one of the
operands should be located in the internal SRAM to guarantee a single cycle execution time of the CoXXX instructions.

- instruction re-ordering

By writing DSP routines with CoXXX instructions it is often needed to change and update the Special Function Registers (SFRs), such as IDX0, IDX1, QX0, QX1, QR0, QR1, and so on. CPU-SFRs control the CPU functionality and behavior. Therefore, special care is required to ensure that instructions in the pipeline always work with the correct SFRs values. With instruction re-ordering the flow of instructions through the pipeline can be improved to optimize the routines.

Example:

Assembly code without optimization (7 cycles)

............
EXTR #1
MOV IDX1, #12 ; initialize IDX1 with 12
CoMUL [IDX1], [R1]
MOV R6, R1
ADD R2, R1
............

Assembly code with optimization (5 cycles)

............
EXTR #1
MOV IDX1, #12 ; initialize IDX1 with 12
MOV R6, R1 ; instruction re-ordering
ADD R2, R1 ; instruction re-ordering
CoMUL [IDX1], [R1]
............

- loop unrolling

The equation is written twice or more inside a loop.

Example (unrolling factor 2):

Assembly code without loop unrolling (17 cycles)

............
MOV R3, #3
loop:
CoMAC [IDX0+], [R1+]
CoMAC [IDX0+], [R1+]
CMPD1 R3,#0h
JMPR cc_NZ,loop

Assembly code with loop unrolling (13 cycles)

MOV R3, #1

loop:
CoMAC [IDX0+], [R1+]
CoMAC [IDX0+], [R1+]
CoMAC [IDX0+], [R1+]
CoMAC [IDX0+], [R1+]

CMPD1 R3,#0h
JMPR cc_NZ,loop

3.5.4 Cycle Count
The cycle count given for each function in this User’s Manual represents the cycles to be needed for executing the assembly instructions in the function. They have been verified on XC161Board. To some degree one can understand as the theoretical cycle count. The given values of cycle count can only be achieved on conditions that all data and assembly codes are located in the internal memory area and no pipeline conflicts occur in the program. Note that the real cycle count may be much larger than the given values, if the data or source code are located in the external memory.

3.5.5 Testing Methodology
The XC166Lib is tested on XC16Board with XC161CJ microcontroller. The test is believed to be accurate and reliable. However, the developer assumes no responsibility for the consequences of use of the XC166Lib.

From version 1.1 we include an integrated test process in XC166Lib to show the test results. With the given test process the user can test the functions in XC166Lib integrated or separately. Chapter 5 will describe this integrated test process and its implementation in details.
4 Function Descriptions

Each function is described with its signature, inputs, outputs, return, implementation description, pseudocode, techniques used, assumptions made, register usage, memory note, example, cycle count and code size.

Functions are classified into the following categories.

- Arithmetic functions
- FIR filters
- IIR filters
- Adaptive filters
- Fast Fourier Transforms
- Matrix operations
- Mathematical operations
- Statistical functions

4.1 Conventions

4.1.1 Argument Conventions

The following conventions have been followed while describing the arguments for each individual function.

Table 4-1 Argument Conventions

<table>
<thead>
<tr>
<th>Argument</th>
<th>Convention</th>
</tr>
</thead>
<tbody>
<tr>
<td>X, x</td>
<td>Input data or input data vector beside in FFT functions, where X representing the FFT spectra of the input vector x</td>
</tr>
<tr>
<td>Y, y</td>
<td>Output data or output data vector</td>
</tr>
<tr>
<td>D_buffer, d_buffer</td>
<td>Delay buffer</td>
</tr>
<tr>
<td>N_x</td>
<td>The size of input vectors</td>
</tr>
<tr>
<td>H, h</td>
<td>Filter coefficient vector</td>
</tr>
<tr>
<td>N_h</td>
<td>The size of coefficient vector H</td>
</tr>
<tr>
<td>DataS</td>
<td>Data type definition equating a short, a 16-bit value representing a 1Q15 number</td>
</tr>
<tr>
<td>DataL</td>
<td>Data type definition equating a long, a 32-bit value representing a 1Q31 number</td>
</tr>
<tr>
<td>Argument</td>
<td>Convention</td>
</tr>
<tr>
<td>----------</td>
<td>------------</td>
</tr>
<tr>
<td>CplxS</td>
<td>Data type definition equating a short, a 16-bit value representing a 1Q15 complex number</td>
</tr>
<tr>
<td>CplxL</td>
<td>Data type definition equating a long, a 32-bit value representing a 1Q31 complex number</td>
</tr>
</tbody>
</table>
4.2 Arithmetic Functions

4.2.1 Complex Numbers
A complex number $z$ is an ordered pair $(x,y)$ of real numbers $x$ and $y$, written as
$$ z = (x,y) $$
where $x$ is called the real part and $y$ the imaginary part of $z$.

4.2.2 Complex Number Representation
A complex number can be represented in different ways, such as

Rectangular form : $C = R + iP$  \[4.1\]
Trigonometric form : $C = M[\cos(\phi) + j\sin(\phi)]$ \[4.2\]
Exponential form : $C = Me^{i\phi}$ \[4.3\]
Magnitude and angle form : $C = M\angle\phi$ \[4.4\]

In the complex functions implementation, the rectangular form is considered.

4.2.3 Complex Plane
To geometrically represent complex numbers as points in the plane two perpendicular coordinate axis in the Cartesian coordinate system are chosen. The horizontal $x$-axis is called the real axis, and the vertical $y$-axis is called the imaginary axis. Plot a given complex number $z= x + iy$ as the point $P$ with coordinates $(x, y)$. The $xy$-plane in which the complex numbers are represented in this way is called the Complex Plane.
4.2.4 Complex Arithmetic

**Addition**

if $z_1$ and $z_2$ are two complex numbers given by $z_1 = x_1 + iy_1$ and $z_2 = x_2 + iy_2$,

$$z_1 + z_2 = (x_1 + iy_1) + (x_2 + iy_2) = (x_1 + x_2) + i(y_1 + y_2)$$  \[4.5\]

**Subtraction**

if $z_1$ and $z_2$ are two complex numbers given by $z_1 = x_1 + iy_1$ and $z_2 = x_2 + iy_2$,

$$z_1 - z_2 = (x_1 - x_2) + i(y_1 - y_2)$$  \[4.6\]

**Multiplication**

if $z_1$ and $z_2$ are two complex numbers given by $z_1 = x_1 + iy_1$ and $z_2 = x_2 + iy_2$,

$$z_1z_2 = (x_1 + iy_1)(x_2 + iy_2) = x_1x_2 + i(x_1y_2 + y_1x_2) + i^2y_1y_2$$

$$= (x_1x_2 - y_1y_2) + i(x_1y_2 + x_2y_1)$$  \[4.7\]
Conjugate

The complex conjugate, $\bar{z}$ of a complex number $z = x + iy$ is given by

$$\bar{z} = x - iy$$

and is obtained by geometrically reflecting the point $z$ in the real axis.

4.2.5 Complex Number Schematic

![Complex Number Schematic Diagram]

Figure 4-2 16-bit Complex number representation

4.2.6 Descriptions

The following arithmetic functions for 16 bit and 32 bit are described:

- 16 bit complex addition
- 16 bit complex subtraction
- 16 bit complex multiplication
- 32 bit real multiplication
CplxAdd_16  16 bit Complex Number Addition

Signature  void  CplxAdd_16 (CplxS*  X, CplxS*  Y, ClpxS*  R)

Inputs  

X  :  Pointer to 16 bit Complex input value in 1Q15 format

Y  :  Pointer to 16 bit Complex input value in 1Q15 format

Output  None

Return  Pointer to the sum of two complex numbers as a 16 bit complex number in 1Q15 format

Implementation Description

This function computes the sum of two 16 bit complex numbers. Wraps around the result in case of overflow. The algorithm is as follows

\[
R_r = x_r + y_r \\
R_i = x_i + y_i
\]

Pseudo code

{ 
  R.real = X.real + Y.real;  
    //add the real part
  R.imag = X.imag + Y.imag;  
    //add the imaginary part
  return R;
}

Techniques  None

Assumption

Register Usage

•  From .c file to .asm file:
  R12 contains the pointer to complex input X.
  R13 contains the pointer to complex input Y.
  R14 contains the pointer to complex output R.
DSP Library for
XC16x Family

Function Descriptions

CplxAdd_16  16 bit Complex Number Addition  (cont’d)

Memory Note

![Diagram of complex number addition for 16 bits]

Figure 4-3  Complex Number addition for 16 bits

Example  

*C166Lib\Examples\Arith_16\Arith_16.c*

Cycle Count

- Store state: 3
- Initialization and read input values: 6
- Real Addition: 3
- Imaginary Addition: 3
- Restore state: 3
- Return: 1

Total: 19

Code Size

- Store state: 6 bytes
- Initialization and read input values: 12 bytes
### CplxAdd_16 16 bit Complex Number Addition (cont’d)

<table>
<thead>
<tr>
<th>Function</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real Addition</td>
<td>10</td>
</tr>
<tr>
<td>Imaginary Addition</td>
<td>10</td>
</tr>
<tr>
<td>Restore state</td>
<td>6</td>
</tr>
<tr>
<td>Return</td>
<td>2</td>
</tr>
</tbody>
</table>

**Total**: 46 bytes
CplxSub_16  16 bit Complex Number Subtraction

Signature  void CplxSub_16 (CplxS* X, CplxS* Y, CplxS* R )

Inputs  X : Pointer to 16 bit complex input value in 1Q15 format
        Y : Pointer to 16 bit complex input value in 1Q15 format

Output  None

Return  Pointer to the difference of two complex numbers as a 16 bit complex number

Implementation Description  This function computes the difference of two 16 bit complex numbers. Wraps around the result in case of underflow. The algorithm is as follows.

\[
R_r = x_r - y_r
\]
\[
R_i = x_i - y_i
\]

Pseudo code

```
{ 
    R.real = X.real - Y.real;  //subtract the real part
    R.imag = X.imag - Y.imag;  //subtract the imaginary part
    return R;
}
```

Techniques  None

Assumption  Register Usage

- From .c file to .asm file:
  - R12 contains the pointer to complex input X.
  - R13 contains the pointer to complex input Y.
  - R14 contains the pointer to complex output R.
CplxSub_16  16 bit Complex Number Subtraction  (cont’d)

Memory Note

![Diagram showing complex number subtraction for 16 bits]

Figure 4-4  Complex number subtraction for 16 bits

Example  
`C166Lib\Examples\Arith_16\Arith_16.c`

Cycle Count

<table>
<thead>
<tr>
<th>Operation</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Store state</td>
<td>3</td>
</tr>
<tr>
<td>Initialization and read input</td>
<td>6</td>
</tr>
<tr>
<td>Real Subtraction</td>
<td>3</td>
</tr>
<tr>
<td>Imaginary Subtraction</td>
<td>3</td>
</tr>
<tr>
<td>Restore state</td>
<td>3</td>
</tr>
<tr>
<td>Return</td>
<td>1</td>
</tr>
</tbody>
</table>

Total  19

Code Size

<table>
<thead>
<tr>
<th>Operation</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Store state</td>
<td>6</td>
</tr>
</tbody>
</table>
**CplxSub_16**

16 bit Complex Number Subtraction (cont’d)

<table>
<thead>
<tr>
<th>Description</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialization and read input</td>
<td>12</td>
</tr>
<tr>
<td>values</td>
<td></td>
</tr>
<tr>
<td>Real Subtraction</td>
<td>10</td>
</tr>
<tr>
<td>Imaginary Subtraction</td>
<td>10</td>
</tr>
<tr>
<td>Restore state</td>
<td>6</td>
</tr>
<tr>
<td>Return</td>
<td>2</td>
</tr>
</tbody>
</table>

**Total** 46 bytes
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CplxMul_16</td>
<td>16 bit Complex Number Multiplication</td>
</tr>
</tbody>
</table>

**Signature**

```c
void CplxMul_16 (CplxS* X, CplxS* Y, CplxS* R )
```

**Inputs**

- **X**: Pointer to 16 bit Complex input value in 1Q15 format
- **Y**: Pointer to 16 bit Complex input value in 1Q15 format

**Return**

Pointer to the multiplication result in 1Q15 format

**Implementation Description**

This function computes the product of the two 16 bit complex numbers. Wraps around the result in case of overflow. The complex multiplication is computed as follows.

\[
R_r = x_r \times y_r - x_i \times y_i \\
R_i = x_i \times y_r + x_r \times y_i
\]

**Pseudo code**

```c
{
    R->real = X.real*Y.real - Y.imag*X.imag;
    R->imag = X.real*Y.imag + Y.real*X.imag;
}
```

**Techniques**

None

**Assumption**

**Register Usage (Small)**

- From `.c` file to `.asm` file:
  - R12 contains the pointer to complex input X.
  - R13 contains the pointer to complex input Y.
  - R14 contains the pointer to complex output R.
CplxMul_16  16 bit Complex Number Multiplication (cont’d)

Memory Note

![Diagram of complex number multiplication for 16 bits]

Figure 4-5  Complex number multiplication for 16 bits

Example  

```
C166Lib\Examples\Arith_16\Arith_16.c
```

Cycle Count

<table>
<thead>
<tr>
<th>Operation</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Store state</td>
<td>3</td>
</tr>
<tr>
<td>Initialization and read input values</td>
<td>5</td>
</tr>
<tr>
<td>Real multiplication</td>
<td>3</td>
</tr>
<tr>
<td>Imaginary multiplication</td>
<td>3</td>
</tr>
<tr>
<td>Restore state</td>
<td>3</td>
</tr>
<tr>
<td>Return</td>
<td>1</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>18</strong></td>
</tr>
</tbody>
</table>

User's Manual for Tasking Compiler  4-32  V 1.1, 2003-06
### CplxMul_16

**16 bit Complex Number Multiplication** (cont’d)

#### Code Size

<table>
<thead>
<tr>
<th>Description</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Store state</td>
<td>6</td>
</tr>
<tr>
<td>Initialization and read input values</td>
<td>10</td>
</tr>
<tr>
<td>Real multiplication</td>
<td>12</td>
</tr>
<tr>
<td>Imaginary multiplication</td>
<td>12</td>
</tr>
<tr>
<td>Restore state</td>
<td>6</td>
</tr>
<tr>
<td>Return</td>
<td>2</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>48</strong></td>
</tr>
</tbody>
</table>
Mul_32 32 bit Real Multiplication

Signature DataL Mul_32 (DataL X, DataL Y)

Inputs
X : 32 bit real input value in 1Q31
Y : 32 bit real input value in 1Q31

Return Multiplication result in 1Q31 format

Implementation Description This function computes the product of the two 32 bit real numbers. Wraps around the result in case of overflow. The multiplication is computed as follows.

\[ R = x_L \times y_L + x_L \times y_H + (x_H \times y_L + x_H \times y_H) \gg 16 \]

Pseudo code
{ 
  \[ R = x_L \times y_L + x_L \times y_H + (x_H \times y_L + x_H \times y_H) \gg 16 \]
}

Techniques None

Assumption Register Usage
- From .c file to .asm file:
  \( X_L(LSW) \) and \( X_H(MSW) \) are stored in R14 and R15, respectively.
  \( Y_L(LSW) \) and \( Y_H(MSW) \) are stored in R12 and R13, respectively.
- From .asm file to .c file:
  \( R_L(LSW) \) is stored in R4.
  \( R_H(MSW) \) is stored in R5.
**Mul_32**  
32 bit Real Multiplication (cont’d)

**Memory Note**

---

**Figure 4-6** 32 bit real number multiplication

---

**Example**  
*C166Lib\Examples\Arith_16\Arith_16.c*

---

**Cycle Count**

<table>
<thead>
<tr>
<th></th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialization</td>
<td>1</td>
</tr>
<tr>
<td>Multiplication</td>
<td>10</td>
</tr>
<tr>
<td>Return</td>
<td>1</td>
</tr>
</tbody>
</table>

**Total**  
**12**

---

**Code Size**

<table>
<thead>
<tr>
<th></th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialization</td>
<td>2 bytes</td>
</tr>
<tr>
<td>Multiplication</td>
<td>40 bytes</td>
</tr>
<tr>
<td>Return</td>
<td>2</td>
</tr>
</tbody>
</table>

**Total**  
**44 bytes**
4.3 FIR Filters

The FIR (Finite Impulse Response) filter, as its name suggests, will always have a finite duration of non-zero output values for given finite duration of non-zero input values. FIR filters use only current and past input samples, and none of the filter's previous output samples, to obtain a current output sample value.

For causal FIR systems, the system function has only zeros (except for poles at z=0). The FIR filter can be realized in transversal, cascade and lattice forms. The implemented structure is of transversal type, which is realized by a tapped delay line. In case of FIR, delay line stores the past input values. The input x(n) for the current calculation will become x(n-1) for the next calculation. The output from each tap is summed to generate the filter output. For a general N tap FIR filter, the difference equation is

\[ y(n) = \sum_{i=0}^{N-1} h_i \cdot x(n-i) \]  

where,

\[ x(n) \]: the filter input for \( n^{th} \) sample \\
\[ y(n) \]: output of the filter for \( n^{th} \) sample \\
\[ h_i \]: filter coefficients \\
\[ N \]: filter order

The filter coefficients, which decide the scaling of current and past input samples stored in the delay line, define the filter response.

The transfer function of the filter in Z-transform is

\[ H[z] = \frac{Y[z]}{X[z]} = \sum_{i=0}^{N-1} h_i \cdot z^{-i} \]  

\[ \frac{Y[z]}{X[z]} = \sum_{i=0}^{N-1} h_i \cdot z^{-i} \]  

[4.12]
4.3.1 Descriptions

The following FIR filter routines are described.

- 16 bit filter coefficients, sample processing
- 32 bit filter coefficients, sample processing
### Fir_16

**FIR Filter, 16 bit coefficients, Sample processing**

#### Signature

```
DataS Fir_16 ( DataS* H, DataS* IN, DataS N_h, DataS* D_buffer)
```

#### Inputs

- **H**: Pointer to filter coefficients in 1Q15 format
- **IN**: Pointer to the new input sample in 1Q15 format
- **L**: Filter order
- **D_buffer**: Pointer to delay buffer

#### Output

- **Return Y**: Filter output in 1Q15 format

#### Implementation Description

The implementation of FIR filter uses transversal structure (direct form). A single input is processed at a time and output for every sample is returned. The filter operates on 16-bit real input, 16-bit coefficients and gives 16-bit real output. The number of coefficients given by the user is arbitrary. The delay line is implemented in parallel to the multiply-accumulate operation using instructions CoMACM. Delay buffer will be located in the DPRAM area.

#### Pseudo code

```
{
    short x(N_h)={0,...};       //Input vector
    short Y;                    //Filter result
    short i;

    //Update the input vector with the new input value
    for(i=0; i<N_h-1; i++)
        x(i) = x(i+1);
    x(N_h-1) = IN;             //move the new input unto X[N_h-1]

    //Calculate the current FIR output
    Y = 0;
    for(i=0; i<N_h; i++)
        Y = Y + h(i)*x(N_h-1-i)       //FIR filter output

    return Y;                  //Filter output returned
}
```
Fir_16  

FIR Filter, 16 bit coefficients, Sample processing (cont’d)

**Techniques**  
- Memory bandwidth conflicts removing

**Assumptions**  
- Delay buffer must be located in DPRAM area

**Register Usage**  
- From .c file to .asm file:  
  R12 contains the pointer of coefficient vector H 
  \((R13)) = \text{IN} \  
  (R14) = N_h 
  R15 contains the pointer of delay buffer
- From .asm file to .c file:  
  Y is stored in R4

**Memory Note**

---

**Figure 4-8 Fir_16**
Fir_16  Fir Filter, 16 bit coefficients, Sample processing (cont’d)

Example  C166Lib\Examples\Filters\Fir\Fir16.c

Cycle Count

<table>
<thead>
<tr>
<th></th>
<th>Cycle Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory Initialization</td>
<td>8</td>
</tr>
<tr>
<td>Read new input into DPRAM</td>
<td>2</td>
</tr>
<tr>
<td>FIR loop</td>
<td>N_h</td>
</tr>
<tr>
<td>Write output</td>
<td>2</td>
</tr>
<tr>
<td>Return</td>
<td>1</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>N_h + 13</td>
</tr>
</tbody>
</table>

Example:

N_h = 12

cycle = 25

Code Size

<table>
<thead>
<tr>
<th></th>
<th>Code Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory initialization</td>
<td>16 bytes</td>
</tr>
<tr>
<td>Read new input into DPRAM</td>
<td>8 bytes</td>
</tr>
<tr>
<td>FIR loop</td>
<td>10 bytes</td>
</tr>
<tr>
<td>Write output</td>
<td>8 bytes</td>
</tr>
<tr>
<td>Restore state</td>
<td>8 bytes</td>
</tr>
<tr>
<td>Return</td>
<td>2 bytes</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>44 bytes</td>
</tr>
</tbody>
</table>
Fir_32  
FIR Filter, 32 bit coefficients, Sample processing

Signature  
DataS  Fir_32 ( DataL*  H,  DataS*  IN,  DataS  N_h,  
DataS*  D_buffer)

Inputs  
H : Pointer to filter coefficients in 1Q31 format
IN : Pointer to the new input sample in 1Q15 format
N_h : Filter order
D_buffer : Pointer to delay buffer located in DPRAM area from 0xf200 to 0xfe00 (3 KBytes)

Output  
Return : Filter output in 1Q15 format

Implementation Description  
The implementation of FIR filter uses transversal structure (direct form). A single input is processed at a time and output for every sample is returned. The filter operates on 16-bit real input, 32-bit coefficients and gives 16-bit real output. The number of coefficients given by the user is arbitrary. The delay line is implemented in parallel to the multiply-accumulate operation using instructions CoMACM. Delay buffer will be located in the DPRAM area.
Fir_32  
FIR Filter, 32 bit coefficients, Sample processing  
(cont’d)

Pseudo code
{
    short x(N_h)={0,...};     //Input vector
    short Y;                  //Filter result
    short i;
    long temp;

    //Update the input vector with the new input value
    for(i=0; i<N_h-1; i++)
    {
        x(i) = x(i+1);
        x(N_h-1) = IN;            //move the new input unto X[N_h-1]
    }

    //Calculate the current FIR output
    temp = 0;
    for(i=0; i<N_h; i++)
    {
        temp = temp + h(i)*x(N_h-1-i)
    }
    Y = (short)temp;        //FIR filter output

    return Y;          //Filter output returned
}

Techniques
• Memory bandwidth conflicts removing
• Instruction re-ordering
• Loop unrolling

Assumptions
• Delay buffer must be located in DPRAM area

Register Usage
• From .c file to .asm file:
  R12 contains the pointer of coefficient vector H
  ((R13)) = IN
  (R14) = N_h
  R15 contains the pointer of delay buffer
• From .asm file to .c file:
  Y is stored in R4
DSP Library for
XC16x Family

Function Descriptions

Fir_32
FIR Filter, 32 bit coefficients, Sample processing (cont’d)

Memory Note

![Memory Diagram]

Figure 4-9   Fir_32

Example

C166Lib\Examples\Filters\Fir\Fir32.c

Cycle Count

<table>
<thead>
<tr>
<th>Operation</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory Initialization</td>
<td>15</td>
</tr>
<tr>
<td>Read new input into DPRAM</td>
<td>2</td>
</tr>
<tr>
<td>FIR loop (LSW)</td>
<td>N_h + 4</td>
</tr>
<tr>
<td>FIR loop (MSW)</td>
<td>N_h + 1</td>
</tr>
<tr>
<td>Write output</td>
<td>1</td>
</tr>
</tbody>
</table>
**Fir_32**  
FIR Filter, 32 bit coefficients, Sample processing (cont’d)

<table>
<thead>
<tr>
<th>Description</th>
<th>Code Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return</td>
<td>1</td>
</tr>
<tr>
<td>Total</td>
<td>(2 \times N_h + 24)</td>
</tr>
</tbody>
</table>

Example:

\[N_h = 12\]
\[cycle = 48\]

**Code Size**

<table>
<thead>
<tr>
<th>Description</th>
<th>Code Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory initialization</td>
<td>30 bytes</td>
</tr>
<tr>
<td>Read new input into DPRAM</td>
<td>8 bytes</td>
</tr>
<tr>
<td>FIR loop (LSW)</td>
<td>28 bytes</td>
</tr>
<tr>
<td>FIR loop (MSW)</td>
<td>14 bytes</td>
</tr>
<tr>
<td>Write output</td>
<td>4 bytes</td>
</tr>
<tr>
<td>Return</td>
<td>2 bytes</td>
</tr>
</tbody>
</table>

**Total**

\[86\] bytes
4.4 IIR Filters

Infinite Impulse Response (IIR) filters have infinite duration of non-zero output values for a given finite duration of non-zero impulse input. Infinite duration of output is due to the feedback used in IIR filters.

Recursive structures of IIR filters make them computational efficient but because of feedback not all IIR structures are realizable (stable). The $N^{th}$ order difference equation for the direct form 1 of the IIR filter is given by

$$y(n) = \sum_{i=1}^{N} a(i-1) \cdot y(n-i) + \sum_{i=0}^{M} b(i) \cdot x(n-i)$$  \[4.13\]

where, $x(n)$ is the $n^{th}$ input and $y(n)$ is the corresponding output.

If $M=N=2$, we have the biquad (second order) IIR filter as

$$y(n) = b(0) \cdot x(n) + b(1) \cdot x(n-1) + b(2) \cdot x(n-2)$$
$$a(0) \cdot y(n-1) + a(1) \cdot y(n-2)$$  \[4.14\]

where $a(0), a(1)$ correspond to the poles and $b(0), b(1), b(2)$ correspond to the zeroes of the filter.

The equivalent transform function is

$$H[z] = \frac{Y[z]}{X[z]} = \frac{b(0) + b(1)z^{-1} + b(2)z^{-2}}{1 - a(0)z^{-1} - a(1)z^{-2}}.$$  \[4.15\]

In the case of a linear shift-invariant system, the overall input-output relationship of a cascade is independent of the order in which systems are cascaded. This property suggests a second direct form realization. Breaking Equation [4.13] into two parts in terms of zeroes and poles of transfer function ($M=N$), we have

$$u(n) = x(n) + \sum_{i=1}^{N} a(i-1) \cdot u(n-i)$$  \[4.16\]

$$y(n) = \sum_{i=0}^{N} b(i) \cdot u(n-i)$$  \[4.17\]
where intermediate state variable \( u(n) \) is used to calculate the filter output \( y(n) \). This representation is called "Direct Form 2" implementation of an IIR filter and is illustrated in Figure 4-10. Direct Form 2 has an advantage over Direct Form 1 as it requires less data memory.

If \( N=2 \), Equation [4.16] and Equation [4.17] are reduced to the biquad (second order) IIR filter with direct form 2 implementation:

\[
\begin{align*}
\dot{u}(n) &= x(n) + a(0) \cdot u(n - 1) + a(1) \cdot u(n - 2) \\
y(n) &= b(0) \cdot u(n) + b(1) \cdot u(n - 1) + b(2) \cdot u(n - 2)
\end{align*}
\]
Any higher order IIR filter can be constructed by cascading several biquad stages together. A cascaded realization of a fourth order system using direct form 2 realization of each biquad subsystem would be as shown in the following diagram.

![Cascaded Biquad IIR Filter with Direct Form 2 Implementation](image)

Figure 4-11  Cascaded Biquad IIR Filter with Direct Form 2 Implementation

In Figure 4-11 each biquad subsystem is realized with direct form 2 structure. Similarly, the biquad subsystem can also be implemented with direct form 1 structure. Rewriting Equation [4.14] we have

\[ y(n) = b(0) \cdot x(n) + u(n - 1) \]  \[ u(n) = a(1) \cdot y(n) + b(1) \cdot x(n) + w(n - 1) \]  \[ w(n) = a(2) \cdot y(n) + b(2) \cdot x(n) \]

where \( u(n) \) and \( w(n) \) are state variables at time \( n \). According to Equation [4.20], Equation [4.21] and Equation [4.22] we have another cascaded biquads IIR filter implementation showed in Figure 4-12.
A Comparison between FIR and IIR filters:

- IIR filters are computational efficient than FIR filters i.e., IIR filters require less memory and fewer instruction when compared to FIR to implement a specific transfer function.
- The number of necessary multiplications are least in IIR while it is most in FIR.
- IIR filters are made up of poles and zeroes. The poles give IIR filter an ability to realize transfer functions that FIR filters cannot do.
- IIR filters are not necessarily stable, because of their recursive nature it is designer’s task to ensure stability, while FIR filters are guaranteed to be stable.
- IIR filters can simulate prototype analog filter while FIR filters cannot.
- Probability of overflow errors is quite high in IIR filters in comparison to FIR filters.
- FIR filters are linear phase as long as $H(z) = H(z^{-1})$ but all stable, realizable IIR filters are not linear phase except for the special cases where all poles of the transfer function lie on the unit circle.

4.4.1 Descriptions

The following IIR filter routines are described.

- 16 bit filter coefficients, direct form 1, sample processing
- 16 bit filter coefficients, direct form 2, sample processing
- 16 bit filter coefficients, N-cascaded real biquads, direct form 1, sample processing
- 16 bit filter coefficients, N-cascaded real biquads, direct form 2, sample processing
IIR_1  
16 bit filter coefficients, Direct form 1, Sample processing

**Signature**

DataS IIR_1(DataS* h, DataS* IN, DataS N, DataS* x_y)

**Inputs**

h : Pointer to filter coefficients in 2Q14
IN : Pointer to new input sample in 1Q15
N : Filter order
x_y : Pointer to delay buffer containing the past input and output samples

**Output**

: 

**Return**

Y : Filter output in 1Q15 format

**Implementation Description**

The routine is implemented according to the [Equation 4.13](Equation), which is called direct form 1 implementation. The vector h contains IIR filter coefficient a(i) and b(i), and is stored in memory. Before starting the routine the delay buffer x_y containing the past input and output signals should be located in DPRAM area. This IIR filter routine processes one sample at a time and returns the output for that sample in 1Q15 format.
DSP Library for
XC16x Family

Function Descriptions

IIR_1 16 bit filter coefficients, Direct form 1, Sample processing (cont’d)

Pseudo code
{
; x(n) = input signal at time n
; y(n) = output signal at time n
; a(k), b(k) = IIR filter coefficients
; N refer to the filter order in Equation [4.13]

DataS  a[N], b[N+1];     //Filter vectors
DataS  y[N], x[N+1];     //input and output signal vectors
DataS  i, temp;

;Move the new input sample into input vector in DPRAM
for(i=N; i>0; i--)
  x(n-i-1) = x(n-i);

x(n) = IN;               //New input sample

;y(n) = 0;
for(i=0 to N)
  y(n) = y(n)+b(i)*x(n-i);
for(i=1 to N)
  y(n) = y(n)+a(i)*y(n-i);

Y = y(n)

return Y;         //Filter Output returned
}

Techniques
• Memory bandwidth conflicts removing
• Instruction re-ordering

Assumptions
• Delay buffer must be located in DPRAM area

Register Usage
• From .c file to .asm file:
  R12  points to the coefficient space containing the
       coefficient vector B and coefficient vector A.
  R13 points to new input sample
       (R14) = N
  R15 points to delay buffer in DPRAM area
• From .asm file to .c file:
  Y is stored in R4
IIR_1  16 bit filter coefficients, Direct form 1, Sample processing (cont’d)

Memory Note

Example  
\textit{C166Lib\Examples\Filters\IIR\IIRform1.c}

Cycle Count

Store state 1
IIR_1

16 bit filter coefficients, Direct form 1, Sample processing (cont’d)

Memory 8
Initialization
Read the new sample into DPRAM 2
First IIR loop N+1
Repeat count re-initialization 2
Second IIR loop N
Write the output 6
Restore state 1
Return 1

Total 2N + 22

Example:
N=4
cycle = 30

Code Size

Store state 2 bytes
Memory initialization 16 bytes
Read the new sample into DPRAM 8 bytes
First IIR loop 10 bytes
Repeat count re-initialization 4 bytes
Second IIR loop 10 bytes
Write the output 22 bytes
Restore state 2 bytes
Return 2 bytes

Total 76 bytes
IIR_2

16 bit filter coefficients, Direct form 2, Sample processing

Signature

DataS IIR_2 ( DataS* h, DataS* IN, DataS N, DataS* u)

Inputs

h : Pointer to filter coefficients in 2Q14
IN : Pointer to new input sample in 1Q15
N : Filter order
u : Pointer to state variable vector

Output

Y : Filter output in 1Q15 format

Return

Implementation Description

The IIR filter routine is implemented based on direct form II (Nth Order) shown in Figure 4-10. The filter operates on 16-bit real input, 16-bit real coefficients and returns 16-bit real output. The number of inputs is arbitrary.

Pseudo code

{ 
  ; x(n) = input signal at time n
  ; u(n) = state variable at time n
  ; y(n) = output signal at time n
  ; a(k), b(k) = IIR filter coefficients
  ; N = M refer to the filter order in Equation [4.13]

  DataS a[N], b[N+1];  //Filter vectors
  DataS u[N+1];        //State variable vector
  DataS i;

  ;Calculate the state variable at time n, u(n)
  u(n) = x(n);
  for(i=1; i<N; i++)
    u(n) = u(n) + a(i-1)*u(n-i);

  ;Calculate the output at time n
  y(n) = 0;
  for(i=0; i<=N; i++)
    y(n) = y(n)+b(i)*u(n-i);

  Y = y(n)

  return Y;          //Filter Output returned
}
IIR_2 16 bit filter coefficients, Direct form 2, Sample processing (cont’d)

Techniques
- Memory bandwidth conflicts removing
- Use of CoMAC and CoMACM instructions
- Filter output converted to 16-bit with saturation

Assumptions
- The state variable vector must be located in DPRAM area.

Register Usage
- From .c file to .asm file:
  R12 points to the coefficient space containing the coefficient vector B and coefficient vector A.
  R13 points to new input sample
  (R14) = N
  R15 points to the state variable vector
- From .asm file to .c file:
  Y is stored in R4
IIR_2  16 bit filter coefficients, Direct form 2, Sample processing (cont’d)

Memory Note

![Diagram of Memory Layout]

Before

<table>
<thead>
<tr>
<th>High Addr.</th>
<th>Low Addr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>DPRAM</td>
<td>Memory</td>
</tr>
<tr>
<td>u(n-1)</td>
<td>b(0)</td>
</tr>
<tr>
<td>u(n-1)</td>
<td>b(1)</td>
</tr>
<tr>
<td>u(n-2)</td>
<td>b(2)</td>
</tr>
<tr>
<td>u(n-3)</td>
<td>b(3)</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>u(n-N+2)</td>
<td>b(N-1)</td>
</tr>
<tr>
<td>u(n-N+1)</td>
<td>b(N)</td>
</tr>
<tr>
<td>u(n-N)</td>
<td>a(0)</td>
</tr>
<tr>
<td></td>
<td>a(1)</td>
</tr>
<tr>
<td></td>
<td>a(2)</td>
</tr>
<tr>
<td></td>
<td>a(3)</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>a(N-1)</td>
</tr>
<tr>
<td></td>
<td>a(N)</td>
</tr>
</tbody>
</table>

After

<table>
<thead>
<tr>
<th>High Addr.</th>
<th>Low Addr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>DPRAM</td>
<td>Memory</td>
</tr>
<tr>
<td>u(n)</td>
<td>b(0)</td>
</tr>
<tr>
<td>u(n)</td>
<td>b(1)</td>
</tr>
<tr>
<td>u(n-1)</td>
<td>b(2)</td>
</tr>
<tr>
<td>u(n-2)</td>
<td>b(3)</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>u(n-N+2)</td>
<td>b(N-1)</td>
</tr>
<tr>
<td>u(n-N+1)</td>
<td>b(N)</td>
</tr>
<tr>
<td>u(n-N)</td>
<td>a(0)</td>
</tr>
<tr>
<td></td>
<td>a(1)</td>
</tr>
<tr>
<td></td>
<td>a(2)</td>
</tr>
<tr>
<td></td>
<td>a(3)</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>a(N-1)</td>
</tr>
<tr>
<td></td>
<td>a(N)</td>
</tr>
</tbody>
</table>

Figure 4-14  IIR_2

Example  

C166Lib\Examples\Filters\IIR\IIRform2.c

Cycle Count

Store state  2
IIR_2  

16 bit filter coefficients, Direct form 2, Sample processing (cont’d)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Code Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory Initialization</td>
<td>9</td>
</tr>
<tr>
<td>Read the new input sample</td>
<td>2</td>
</tr>
<tr>
<td>First IIR loop</td>
<td>(N + 4)</td>
</tr>
<tr>
<td>Repeat count re-initialization</td>
<td>1</td>
</tr>
<tr>
<td>Second IIR loop</td>
<td>(N+1)</td>
</tr>
<tr>
<td>Write the output</td>
<td>3</td>
</tr>
<tr>
<td>Restore state</td>
<td>2</td>
</tr>
<tr>
<td>Return</td>
<td>1</td>
</tr>
</tbody>
</table>

**Total**  

\(2N + 25\) bytes

Example:

\[N = 4\]

\[\text{cycle} = 34\]

**Code Size**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Code Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Store state</td>
<td>4 bytes</td>
</tr>
<tr>
<td>Memory initialization</td>
<td>18 bytes</td>
</tr>
<tr>
<td>Read the new input sample</td>
<td>6 bytes</td>
</tr>
<tr>
<td>First IIR loop</td>
<td>22 bytes</td>
</tr>
<tr>
<td>Repeat count re-initialization</td>
<td>2 bytes</td>
</tr>
<tr>
<td>Second IIR loop</td>
<td>10 bytes</td>
</tr>
<tr>
<td>Write the output</td>
<td>12 bytes</td>
</tr>
<tr>
<td>Restore state</td>
<td>4 bytes</td>
</tr>
<tr>
<td>Return</td>
<td>2 bytes</td>
</tr>
</tbody>
</table>

**Total**  

80 bytes
IIR bi_1

IIR N-cascaded real biquads, Direct form 1, 16 bit coefficients, Sample processing

Signature

DataS IIR_bi_1(DataS* h, DataS* IN, DataS N, DataS* u_w)

Inputs

h : Pointer to filter coefficients in 1Q15
IN : Pointer to new input sample in 1Q15
N : Number of the biquads
u_w : Pointer to the state variable vector

Output

Return

Y : Filter output in 1Q15 format

Implementation

The IIR filter is implemented as a cascade of direct form 1 biquads according to Figure 4-11. If the number of biquads is 'N', the filter order is 2*N. A single sample is processed at a time and the output for that sample is returned. The filter operates on 16-bit real input, 16-bit real coefficients and returns 16-bit real output. The number of inputs is arbitrary. Length of delay line is 2*N.
Pseudo code

{  
  ; x_i(n) = input signal at time n of biquad number i  
  ; u_i(n), w_i(n) = state variables at time n of biquad number i  
  ; y_i(n) = output signal at time n of biquad number i  
  ; a_i(k), b_i(k) = Filter coefficients of biquad number i  
  ; N = number of biquads  

  DataS a_i(n), b_i(n);  //Filter vectors  
  DataS u_i(n), w_i(n);  //state variable vector  
  DataS y_i(n), Y;  //output  
  DataS i;  

  for(i=1; i<=N; i++)  
  {  
    ;Calculate the output at time n  
    y_i(n) = b_i(0)*x_i(n) + u_i(n-1);  
    
    ;Update the state variable u_i(n) at time n  
    u_i(n) = b_i(1)*x_i(n) + a_i(0)*y_i(n) + w_i(n-1);  
    
    ;Update the state variable w_i(n) at time n  
    w_i(n) = b_i(2)*x_i(n) + a_i(1)*y_i(n);  
    
    ;Set i-th biquad output to (i+1)-th biquad input  
    x_{i+1}(n) = y_i(n);  
  }  

  ;Output of the last biquad  
  Y = y_{N-1}(n);  

  return Y;  //Filter Output returned  
}

Techniques

- Memory bandwidth conflicts removing  
- Use of CoMAC instructions  
- Instruction re-ordering  
- Filter output converted to 16-bit with saturation

Assumptions

- The state variable vector must be located in DPRAM area.
IIR bi_1  IIR N-cascaded real biquads, Direct form 1, 16 bit coefficients, Sample processing (cont’d)

Register Usage

- From .c file to .asm file:
  R12 points to the coefficient space containing the coefficient vector B and coefficient vector A.
  R13 points to new input sample
  (R14) = N
  R15 points to the state variable vector

- From .asm file to .c file:
  Y is stored in R4

Memory Note
IIR bi_1

IIR N-cascaded real biquads, Direct form 1, 16 bit coefficients, Sample processing (cont’d)

Figure 4-15  IIR_bi_1

Example  
C166Lib\Examples\Filters\IIR\IIRbi_1.c

Cycle Count

Store state  2
IIR _bi_1

IIR N-cascaded real biquads, Direct form 1, 16 bit coefficients, Sample processing (cont’d)

<table>
<thead>
<tr>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory Initialization</td>
<td>5</td>
</tr>
<tr>
<td>First to (N-1)-th biquad (biquad loop)</td>
<td>19(N-1)</td>
</tr>
<tr>
<td>Last biquad</td>
<td>17</td>
</tr>
<tr>
<td>Restore state</td>
<td>2</td>
</tr>
<tr>
<td>Return</td>
<td>1</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>19(N-1) + 27</td>
</tr>
</tbody>
</table>

Example (Small):
N = 2
cycle = 46

Code Size

<table>
<thead>
<tr>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Store state</td>
<td>4</td>
</tr>
<tr>
<td>Memory initialization</td>
<td>10</td>
</tr>
<tr>
<td>First to (N-1)-th biquad (biquad loop)</td>
<td>42</td>
</tr>
<tr>
<td>Last biquad</td>
<td>46</td>
</tr>
<tr>
<td>Restore state</td>
<td>4</td>
</tr>
<tr>
<td>Return</td>
<td>2</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>108</td>
</tr>
</tbody>
</table>
IIR_bi_2  IIR N-cascaded real biquads, Direct form 2, 16 bit coefficients, Sample processing

Signature  DataS IIR_bi_2 ( DataS* h, DataS* IN, DataS N, DataS* u )

Inputs  h  :  Pointer to filter coefficients in 1Q15
IN  :  Pointer to new input sample
N  :  Number of the biquads
u  :  Pointer to state variable vector

Output  

Return  Y  :  Filter output in 1Q15 format

Implementation Description  The IIR filter is implemented as a cascade of direct form 2 biquads according to Figure 4-11. If number of biquads is 'N', the filter order is 2*N. A single sample is processed at a time and output for that sample is returned. The filter operates on 16-bit real input, 16-bit real coefficients and returns 16-bit real output. The number of inputs is arbitrary. Length of delay line is 2*N.
DSP Library for
XC16x Family
Function Descriptions

IIR_bi_2  IIR N-cascaded real biquads, Direct form 2, 16 bit
coefficients, Sample processing (cont’d)

Pseudo code

```
{  ; x_i(n) = input signal at time n of biquad number i
  ; u_i(n) = state variables at time n of biquad number i
  ; y_i(n) = output signal at time n of biquad number i
  ; a_i(k), b_i(k) = Filter coefficients of biquad number i
  ; N = number of biquads

  DataS  a_i(n), b_i(n);    //Filter vectors
  DataS  u_i(n);            //state variable vector
  DataS  y_i(n), Y;         //output
  DataS  i;

  for(i=1; i<=N; i++)
  {
    ;Update the state variable u_i(n)at time n
    u_i(n) = x_i(n) + a_i(0)*u_i(n-1) + a_i(1)*u_i(n-2);

    ;Calculate the output at time n
    y_i(n) = b_i(0)*u_i(n) + b_i(1)*u_i(n-1) + b_i(2)*u_i(n-2);

    ;Set i-th biquad output to (i+1)-th biquad input
    x_{i+1}(n) = y_i(n);
  }

  ;Output of the last biquad
  Y = y_{N-1}(n);

  return Y;       //Filter Output returned
}
```

Techniques
- Memory bandwidth conflicts removing
- Use of CoMAC instructions
- Instruction re-ordering
- Filter output converted to 16-bit with saturation

Assumptions
- The state variable vector must be located in DPRAM area.
IIR_Bi_2

IIR N-cascaded real biquads, Direct form 2, 16 bit coefficients, Sample processing (cont’d)

Register Usage

- From .c file to .asm file:
  R12 points to the coefficient space containing the coefficient vector B and coefficient vector A.
  R13 points to new input sample
  \((R14) = N\)
  R15 points to state variable vector
- From .asm file to .c file:
  Y is stored in R4
IIR_bi_2  IIR N-cascaded real biquads, Direct form 2, 16 bit coefficients, Sample processing (cont’d)

Memory Note

![Memory Diagram]

Figure 4-16  IIR_bi_2

Example  
C166Lib\Examples\Filters\IIR\IIRbi_2.c

Cycle Count

| Store state | 1 |

User’s Manual for Tasking Compiler 4-65 V 1.1, 2003-06
### IIR_bi_2

**IIR N-cascaded real biquads, Direct form 2, 16 bit coefficients, Sample processing** (cont’d)

<table>
<thead>
<tr>
<th>Step</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory Initialization</td>
<td>8</td>
</tr>
<tr>
<td>First to (N-1)-th biquad (biquad loop)</td>
<td>12(N-1)</td>
</tr>
<tr>
<td>Last biquad</td>
<td>12</td>
</tr>
<tr>
<td>Restore state</td>
<td>1</td>
</tr>
<tr>
<td>Return</td>
<td>1</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>12(N-1) + 23</td>
</tr>
</tbody>
</table>

#### Example:
- N = 2
- cycle = 35

### Code Size

<table>
<thead>
<tr>
<th>Step</th>
<th>Code Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Store state</td>
<td>2 bytes</td>
</tr>
<tr>
<td>memory initialization</td>
<td>16 bytes</td>
</tr>
<tr>
<td>First to (N-1)-th biquad (biquad loop)</td>
<td>40 bytes</td>
</tr>
<tr>
<td>Last biquad</td>
<td>36 bytes</td>
</tr>
<tr>
<td>Restore state</td>
<td>2 bytes</td>
</tr>
<tr>
<td>Return</td>
<td>2 bytes</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>98 bytes</td>
</tr>
</tbody>
</table>
4.5 Adaptive Digital Filters

An adaptive filter adapts to changes in its input signals automatically.

Conventional linear filters are those with fixed coefficients. These can extract signals where the signal and noise occupy fixed and separate frequency bands. Adaptive filters are useful when there is a spectral overlap between the signal and noise or if the band occupied by the noise is unknown or varies with time. In an adaptive filter, the filter characteristics are variable and they adapt to changes in signal characteristics. The coefficients of these filters vary and cannot be specified in advance.

The self-adjusting nature of adaptive filters is largely used in applications like telephone echo cancelling, radar signal processing, equalization of communication channels etc.

Adaptive filters with the LMS (Least Mean Square) algorithm are the most popular kind. The basic concept of an LMS adaptive filter is as follows.

![Figure 4-17 Adaptive filter with LMS algorithm](image)

The filter part is an N-tap FIR filter with coefficients h(0,n), h(1,n), ..., h(N-1,n), whose input signal is x(n) and output is y(n). The difference between the actual output y(n) and a desired output d(n), gives an error signal

\[ e(n) = d(n) - y(n) \] [4.23]
The algorithm uses the input signal \( x(n) \) and the error signal \( e(n) \) to adjust the filter coefficients \( h(0,n), h(1,n), \ldots, h(N-1,n) \), such that the difference, \( e(n) \) is minimized on a criterion. The LMS algorithm uses the minimum mean square error criterion

\[
\min_{h(0,n), h(1,n), \ldots, h(N-1,n)} E(e^2(n)) \quad [4.24]
\]

Where \( E \) denotes statistical expectation. The algorithm of a regular (non-delayed) LMS adaptive filter is mathematically expressed as follows.

\[
y(n) = h(0, n) \cdot x(n) + h(1, n) \cdot x(n - 1) + h(2, n) \cdot x(n - 2) + \ldots + h(N - 1, n) \cdot x(n - N + 1) \quad [4.25]
\]

\[
e(n) = d(n) - y(n) \quad [4.26]
\]

\[
h(k, n + 1) = h(k, n) + x(n - k) \times \mu \times e(n) \quad [4.27]
\]

where \( \mu > 0 \) is a constant called step-size. Note that the filter coefficients are time varying. \( h(i,n) \) denotes the value of the \( i \)-th coefficient at time \( n \). The algorithm has three stages:

1. calculate the current filter output \( y(n) \).
2. calculate the current error value \( e(n) \) using the current expected value \( d(n) \) and currently calculated output \( y(n) \).
3. update the filter coefficients for next iteration using current error \( e(n) \) and samples \( x(n-k) \).

Step-size \( \mu \) controls the convergence of the filter coefficients to the optimal (or stationary) state. The larger the \( \mu \) value, the faster the convergence of the adaptation. On the other hand, a large value of \( \mu \) also leads to a large variation of \( h(i,n) \) (a bad accuracy) and thus a large variation of the output error (a large residual error). Therefore, the choice of \( \mu \) is always a trade-off between fast convergence and high accuracy. \( \mu \) must not be larger than a certain threshold. Otherwise, the LMS algorithm diverges.

### 4.5.1 Delayed LMS algorithm for an adaptive filter

In the regular LMS adaptive filter the update of filter coefficients makes use of current error value and input value. This makes the choice of step-size \( \mu \) more difficult due to the effect of \( \mu \) on convergence of adaptive filter. To minimize the effect of \( \mu \) on the filter convergence a delayed LMS algorithm for an adaptive filter is introduced. The algorithm of a delayed LMS adaptive filter can be represented by the following mathematical equations.
The algorithm has three stages:
1. calculate the current filter output $y(n)$.
2. update filter coefficients for the next iteration using previous error value $e(n-1)$ and the delayed input sample $x(n-k-1)$.
3. calculate the current error value $e(n)$ and store it in memory.

### 4.5.2 Descriptions

In the DSP library, the delayed LMS adaptive filters are implemented. The following are the implemented delayed LMS adaptive FIR filter functions with 16 bit input and 16 bit or 32 bit filter coefficients.

- 16 bit real coefficients, delayed LMS, Sample processing
- 32 bit real coefficients, delayed LMS, Sample processing
Adp_filter_16

16 bit real coefficients, Delayed LMS, Sample Processing

Signature

DataS Adap_filter_16 ( DataS* h,
DataS* IN,
DataS* D,
DataS* error,
DataS N_h,
DataS Step,
DataS* d_buffer)

Inputs

h : Pointer to filter coefficients in 1Q15
IN : Pointer to new input value
D : Pointer to the expected signal at time n
error : Pointer to error signal
N_h : Filter size
Step : Adaptive gain
d_buffer : Pointer to delay buffer

Return

Y : Output value of the filter

Implementation Description

Delayed LMS algorithm has been used to realize an adaptive FIR filter. That is, the update of coefficients in the current instant is done using the error in the previous output.

The FIR filter is implemented using transversal structure and is realized as a tapped delay line. In this routine, both of signals and filter coefficients have 16 bit precision.

This routine processes one sample at a time and returns output of that sample. The input for which the output is to be calculated is sent as an argument to the function.
Adp_filter_16 16 bit real coefficients, Delayed LMS, Sample Processing (cont’d)

Pseudo code
{
    ; x(n) = input signal at time n
    ; d(n) = desired signal at time n
    ; y(n) = output signal at time n
    ; h(k,n) = k-th coefficient at time n
    ; gain = adaptive gain
    ; N = number of coefficient taps in the filter

    DataS x(n);       //input signal
    DataS d(n);       //desired signal
    DataS h(k,n);     //filter coefficient
    DataS y(n), Y;    //output
    DataS k;

    ;Calculate the output at time n
    y(n) = 0;
    for(k=0; k<N_h; k++)
        y(n) = y(n) + h(k,n)*x(n-k);

    ;Update the filter coefficients
    for(k=0; k<N_h; k++)
        h(k,n+1) = h(k,n) + gain*e(n-1)*x(n-k-1);

    ;Error signal at time n
    e(n) = d(n) - y(n);

    Y = y(n);
    return Y;       //Filter Output returned
}

Techniques
- Memory bandwidth conflicts removing
- Use of CoMACM instructions
- Instruction re-ordering
- Filter output converted to 16-bit with saturation
- Intermediate result stored in [R13]

Assumptions
- Delay buffer must be located in DPRAM area.
Adp_filter_16 16 bit real coefficients, Delayed LMS, Sample Processing (cont’d)

Register Usage

• From .c file to .asm file:
  R12 points to the vector h.
  ((R13)) = IN
  ((R14)) = D
  ((R15)) = error
  (R0) = N_h
  (R0+2) = Step
  (R0+4) points to delay buffer

• From .asm file to .c file:
  (R4) = Y

Memory Note
Adp_filter_16 16 bit real coefficients, Delayed LMS, Sample Processing (cont’d)

**Example**:  
C166Lib\Examples\Adaptive_filter\Adaptive_filter_16.c

**Cycle Count**
- Store state: 8
- Read parameters from stack: 5
- Memory Initialization: 11
- Read the new input sample: 3

---

Figure 4-18  Adap_filter_16

DPRAM

Memory

Before

After

1Q16

R12

IDX0

CoMACM

DPRAM

Memory
Adp_filter_16  16 bit real coefficients, Delayed LMS, Sample Processing (cont’d)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Code Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Calculate the current output</td>
<td>N_h + 3</td>
</tr>
<tr>
<td>Filter coefficient adaptation</td>
<td>7N_h + 1</td>
</tr>
<tr>
<td>Calculation of the error signal</td>
<td>4</td>
</tr>
<tr>
<td>Restore state</td>
<td>8</td>
</tr>
<tr>
<td>Return</td>
<td>1</td>
</tr>
</tbody>
</table>

**Total** 8*N_h + 44

Example:
N_h = 12
cycle = 140

**Code Size**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Code Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Store state</td>
<td>16 bytes</td>
</tr>
<tr>
<td>Read parameters from stack</td>
<td>10 bytes</td>
</tr>
<tr>
<td>Memory Initialization</td>
<td>22 bytes</td>
</tr>
<tr>
<td>Read the new input sample</td>
<td>12 bytes</td>
</tr>
<tr>
<td>Calculate the current output</td>
<td>28 bytes</td>
</tr>
<tr>
<td>Filter coefficient adaptation</td>
<td>52 bytes</td>
</tr>
<tr>
<td>Calculation of the error signal</td>
<td>10 bytes</td>
</tr>
<tr>
<td>Restore state</td>
<td>16 bytes</td>
</tr>
<tr>
<td>Return</td>
<td>2 bytes</td>
</tr>
</tbody>
</table>

**Total** 168 bytes
Adp_filter_32  32 bit real coefficients, Delayed LMS, Sample Processing

**Signature**

```c
DataS Adap_filter_32( DataL* h,
DataS* IN,
DataS* D,
DataS* error,
DataS N_h,
DataS Step,
DataS* d_buffer
)
```

**Inputs**

- **h**: Pointer to filter coefficients in 1Q31
- **IN**: Pointer to new input value
- **D**: Pointer to the expected signal at time n
- **error**: Pointer to error signal
- **N_h**: Filter size
- **Step**: Adaptive gain
- **d_buffer**: Pointer to delay buffer

**Return**

- **Y**: Output value of the filter

**Implementation Description**

LMS algorithm has been used to realize an adaptive FIR filter. That is, the update of coefficients in the current instant is done using the error in the previous output.

The FIR filter is implemented using transversal structure and is realized as a tapped delay line. Here, the filter coefficients have 32 bit precision, while input and output signals have 16 bit precision.

This routine processes one sample at a time and returns output of that sample. The input for which the output is to be calculated is sent as an argument to the function.
Adp_filter_32  32 bit real coefficients, Delayed LMS, Sample Processing (cont’d)

Pseudo code

{  
;  x(n) = input signal at time n  
;  d(n) = desired signal at time n  
;  y(n) = output signal at time n  
;  h(k,n) = k-th coefficient at time n  
;  gain = adaptive gain  
;  N = number of coefficient taps in the filter

DataS  x(n);       //input signal  
DataS  d(n);       //desired signal  
DataL  h(k,n);     //filter coefficient  
DataS  y(n), Y;    //output  
DataS  k;

;Calculate the output at time n  
y(n) = 0;  
for(k=0; k<N_h; k++)  
  y(n) = y(n) + (short)h(k,n)*x(n-k);

;Update the filter coefficients  
for(k=0; k<N_h; k++)  
  h(k,n+1) = h(k,n) + (long)gain*e(n)*x(n-k);

;Error signal at time n  
e(n) = d(n) - y(n);

Y = y(n);  
return Y;   //Filter Output returned
}

Techniques
• Memory bandwidth conflicts removing
• Use of CoMAC instructions
• Instruction re-ordering
• Filter output converted to 16-bit with saturation
• Intermediate result stored in [R13]

Assumptions
• Delay buffer must be located in DPRAM area.
Adp_filter_32  32 bit real coefficients, Delayed LMS, Sample Processing (cont’d)

Register Usage

• From .c file to .asm file:
  R12 points to the vector h.
  ((R13)) = IN
  ((R14)) = D
  ((R15)) = error
  (R0) = N_h
  (R0+2) = Step
  (R0+4) points to delay buffer

• From .asm file to .c file:
  (R4) = Y

Memory Note
**Adp_filter_32**

32 bit real coefficients, Delayed LMS, Sample Processing (cont’d)

![Diagram](image.png)

**Figure 4-19 Adp_filter_32**

**Example**

*C166Lib\Examples\Adaptive_filter\Adaptive_filter_32.c*

**Cycle Count**

- Store state: 7
- Read parameters: 5
- Memory: 18
- Initialization
- Read the new input: 3
### Adp_filter_32

32 bit real coefficients, Delayed LMS, Sample Processing (cont’d)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Code Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Calculation of the current output y(n) (LSW )</td>
<td>$N_h + 3$</td>
</tr>
<tr>
<td>Calculation of the current output y(n) (MSW )</td>
<td>$N_h + 3$</td>
</tr>
<tr>
<td>Filter coefficient adaptation</td>
<td>$6N_h + 1$</td>
</tr>
<tr>
<td>Calculation of the error signal</td>
<td>4</td>
</tr>
<tr>
<td>Restore state</td>
<td>7</td>
</tr>
<tr>
<td>Return</td>
<td>1</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>$8*N_h + 52$</td>
</tr>
</tbody>
</table>

Example:

$N_h = 12$

cycle = 150

### Code Size

<table>
<thead>
<tr>
<th>Operation</th>
<th>Code Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Store state</td>
<td>14 bytes</td>
</tr>
<tr>
<td>Read parameters</td>
<td>10 bytes</td>
</tr>
<tr>
<td>Memory Initialization</td>
<td>36 bytes</td>
</tr>
<tr>
<td>Read the new input</td>
<td>12 bytes</td>
</tr>
<tr>
<td>Calculation of the current output y(n) (LSW )</td>
<td>24 bytes</td>
</tr>
<tr>
<td>Calculation of the current output y(n) (MSW )</td>
<td>26 bytes</td>
</tr>
<tr>
<td>Filter coefficient adaptation</td>
<td>48 bytes</td>
</tr>
<tr>
<td>Calculation of the error signal</td>
<td>10 bytes</td>
</tr>
</tbody>
</table>
Adp_filter_32 32 bit real coefficients, Delayed LMS, Sample Processing (cont’d)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Restore state</td>
<td>14 bytes</td>
</tr>
<tr>
<td>Return</td>
<td>2 bytes</td>
</tr>
<tr>
<td>Total</td>
<td>196 bytes</td>
</tr>
</tbody>
</table>

Adp_filter_32 32 bit real coefficients, Delayed LMS, Sample Processing (cont’d)
4.6 Fast Fourier Transforms

Spectrum (Spectral) analysis is a very important methodology in Digital Signal Processing. Many applications have a requirement of spectrum analysis. The spectrum analysis is a process of determining the correct frequency domain representation of the sequence. The analysis gives rise to the frequency content of the sampled waveform such as bandwidth and centre frequency.

One of the method of doing the spectrum analysis in Digital Signal Processing is by employing the Discrete Fourier Transform (DFT).

The DFT is used to analyze, manipulate and synthesize signals in ways not possible with continuous (analog) signal processing. It is a mathematical procedure that helps in determining the harmonic, frequency content of a discrete signal sequence. The DFT is defined by

\[
X(k) = \sum_{n=0}^{N-1} x(n) W_N^{nk} \tag{4.31}
\]

where

\[
W_N = e^{-j2\pi/N} = \cos(2\pi nk/N) - j\sin(2\pi nk/N) \tag{4.32}
\]

Using Equation [4.32] in Equation [4.31] we can rewrite \(X(k)\) as

\[
X(k) = \sum_{n=0}^{N-1} x(n)\left[\cos(2\pi nk/N) - j\sin(2\pi nk/N)\right] \tag{4.33}
\]

\(X(k)\) is the \(k^{th}\) DFT output component for \(k=0,1,2,\ldots,N-1\)
\(x(n)\) is the sequence of discrete sample for \(n=0,1,2,\ldots,N-1\)
\(j\) is imaginary unit \(\sqrt{-1}\)

\(N\) is the number of samples of the input sequence (and number of frequency points of DFT output).

While the DFT is used to convert the signal from time domain to frequency domain. The complementary function for DFT is the IDFT, which is used to convert a signal from frequency to time domain. The IDFT is given by

\[
x(n) = \frac{1}{N} \sum_{k=0}^{N-1} X(k) e^{j2\pi nk/N} \quad \text{[exponential form]} \tag{4.34}
\]
Notice the difference between DFT in Equation [4.33] and Equation [4.35], the IDFT Kernel is the complex conjugate of the DFT and the output is scaled by \( N \).

\[ W_N^{nk} \]

The Kernel of the DFT and IDFT is called the Twiddle-Factor and is given by,

In exponential form,

\[
\begin{align*}
\text{for DFT} & : e^{-j\frac{2\pi nk}{N}} \\
\text{for IDFT} & : e^{j\frac{2\pi nk}{N}}
\end{align*}
\]

In rectangular form,

\[
\begin{align*}
\cos\left(\frac{2\pi nk}{N}\right) - j\sin\left(\frac{2\pi nk}{N}\right) & \quad \text{for DFT} \\
\cos\left(\frac{2\pi nk}{N}\right) + j\sin\left(\frac{2\pi nk}{N}\right) & \quad \text{for IDFT}
\end{align*}
\]

While calculating DFT, a complex summation of \( N \) complex multiplications is required for each of \( N \) output samples. \( N^2 \) complex multiplications and \( N(N-1) \) complex additions compute an \( N \)-point DFT. The processing time required by large number of calculation limits the usefulness of DFT. This drawback of DFT is overcome by a more efficient and fast algorithm called Fast Fourier Transform (FFT). The radix-2 FFT computes the DFT in \( N\log_2(N) \) complex operations instead of \( N^2 \) complex operations for that of the DFT.

The FFT has the following preconditions to operate at a faster rate.

- The radix-2 FFT works only on the sequences with lengths that are power of two.
- The FFT has a certain amount of overhead that is unavoidable, called bit reversed ordering. The output is scrambled for the ordered input or the input has to be arranged in a predefined order to get output properly arranged. This makes the straight DFT better suited for short length computation than FFT. The graph shows the algorithm complexity of both on a typical processor like pentium.
The Fourier transform plays an important role in a variety of signal processing applications. Anytime, if it is more comfortable to work with a signal in the frequency domain than in the original time or space domain, we need to compute Fourier transform. FFT is an incredibly efficient algorithm for computing DFT. The main idea of FFT is to exploit the periodic and symmetric properties of the DFT Kernel $W_N^{nk}$. The resulting algorithm depends strongly on the transform length N. The basic Cooley-Tukey algorithm assumes that N is a power of two. Hence it is called radix-2 algorithm. Depending on how the input samples $x(n)$ and the output data $X(k)$ are grouped, either a decimation-in-time (DIT) or a decimation-in-frequency (DIF) algorithm is obtained. The technique used by Cooley and Tukey can also be applied to DFTs, where N is a power of r. The resulting algorithms are referred as radix-r FFT. It turns out that radix-4, radix-8, and radix-16 are especially interesting. In cases where N cannot be represented in terms of powers of single number, mixed-radix algorithms must be used. For example for 28 point input, since 28 cannot be represented in terms of powers of 2 and 4 we use radix-7 and radix-4 FFT to get the frequency spectrum. In this library the basic radix-2 decimation-in-frequency FFT algorithm is implemented.
4.6.1 Radix-2 Decimation-In-Time FFT Algorithm

The decimation-in-time (DIT) FFT divides the input (time) sequence into two groups, one of even samples and the other of odd samples. N/2-point DFTs are performed on these sub-sequences and their outputs are combined to form the N-point DFT.

First, \( x(n) \) the input sequence in the Equation [4.31] is divided into even and odd sub-sequences.

\[
X(k) = \sum_{n=0}^{\frac{N}{2}-1} x(2n)W_N^{2nk} + \sum_{n=0}^{\frac{N}{2}-1} x(2n+1)W_N^{(2n+1)k} \quad \text{for } k=0 \text{ to } N-1 \quad [4.36]
\]

\[
= \sum_{n=0}^{\frac{N}{2}-1} x(2n)W_N^{2nk} + W_N^k \sum_{n=0}^{\frac{N}{2}-1} x(2n+1)W_N^{2nk}
\]

But, \( W_N^{2nk} = (e^{-j2\pi/N})^{2nk} = (e^{-j2\pi/(N/2)})^{nk} = W_N^{nk} \)

By substituting the following in Equation [4.36]

\( h(n) = x(2n) \)
\( g(n) = x(2n+1) \),

Equation [4.36] becomes

\[
X(k) = \sum_{n=0}^{\frac{N}{2}-1} h(n)W_N^{nk/2} + W_N^k \sum_{n=0}^{\frac{N}{2}-1} g(n)W_N^{nk/2} \quad \text{for } k=0 \text{ to } N-1 \quad [4.37]
\]

Equation [4.37] is the radix-2 DIT FFT equation. It consists of two N/2-point DFTs \( H(k) \) and \( G(k) \) performed on the subsequences of even and odd samples of the input sequence \( x(n) \), respectively. Multiples of \( W_N \), the Twiddle-Factors are the coefficients in the FFT calculation.

Further,

\[
W_N^{k+N/2} = (e^{-j2\pi/N})^k \times (e^{-j2\pi/N})^{N/2} = -W_N^k . \quad [4.38]
\]
**Equation [4.37]** can be expressed in two equations

\[ X(k) = H(k) + W_N^k G(k) \] \[ X(k + N/2) = H(k) - W_N^k G(k) \] \[ k=0 \text{ to } N/2-1 \] \[ [4.39] \]

The decomposition procedure can be continued until two-point DFTs are reached. **Figure 4-22** illustrates the flow graph of a real 8-point DIT FFT.

**Figure 4-21** 8-point DIT FFT

Note that the input sequence x(n) in **Figure 4-21** is in the scrabbled order. The same procedure can be repeated with the linear input order. Then we have the alternate form of the DIT FFT shown in **Figure 4-22**, where the output sequence X(n) is rearranged to appear in bit-reversed order. The only difference between **Figure 4-21** and **Figure 4-22** is a rearrangement of the butterflies and the computational load and final results are identical. In the library the implementation of real-valued forward FFT is based on **Figure 4-22**.
4.6.2 Radix-2 Decimation-In-Frequency FFT Algorithm

A second variant of the radix-2 FFT is the decimation-in-frequency algorithm. In order to get this algorithm, we split the input sequence into the first and second halves and write the DFT as

\[
X(k) = \sum_{n=0}^{N/2-1} x(n)W_N^{nk} + \sum_{n=0}^{N/2-1} x(n + N/2)W_N^{(n + N/2)k}
\]

\[
= \sum_{n=0}^{N/2-1} [x(n) + (-1)^k x(n + N/2)]W_N^{nk}, \quad \text{for } k=0 \text{ to } N. \quad [4.41]
\]

For the even and odd numbered DFT points we get...
As with the decimation-in-time algorithm, the N-point DFT is decomposed into two N/2-point DFTs. Using the principle repeatedly results in an FFT algorithm where the input values appear in their natural order, but where the output values appear in bit reversed order. The complexity is the same as for the decimation-in-time FFT. Figure 4-23 shows the flow graph of an 8-point DIF FFT. The comparison of Figure 4-21 and Figure 4-23 shows that the two graphs can be viewed as transposed versions of one another.

![8-point DIF FFT](image)

**Figure 4-23 8-point DIF FFT**

Similar to FFT with decimation-in-time, if the inputs in Figure 4-23 are rearranged in bit-inverse order and the outputs in linear order, we have an alternate implementation for DIF FFT showed in Figure 4-25.
4.6.3 Complex FFT Algorithm

If the input sequence is complex, according to Equation [4.39] and Equation [4.40], we can write the complex DFT as

\[ X(k) = P(k) + W_N^k Q(k) \]
\[ X(k + N/2) = P(k) - W_N^k Q(k) \quad \text{for } k=0 \text{ to } N/2-1, \]  

where \( P(k) \) and \( Q(k) \) are the complex even and odd partial DFTs. As same as real FFT, the complex FFT has also two implementations, i.e. decimation-in-time and decimation-in-frequency complex FFT. Figure 4-25 shows an 8-point DIT complex FFT implementation.

In Figure 4-25, each pair of arrows represents a Butterfly. The whole of the complex FFT is computed by different patterns of Butterflies. These are called groups and stages.
For 8-point DIT FFT the first stage consists of one groups of four Butterfly, second consists of two groups of two butterflies and third stage has four group of one Butterflies. Each Butterfly is represented as in Figure 4-26.
The output is derived as follows

\[ P_{n+1} = P_n + Q_n \cdot W_N^k \]  \[4.45\]

\[ Q_{n+1} = P_n - Q_n \cdot W_N^k \]  \[4.46\]

where \(n\) represents the number of stages.

Of course, the complex FFT can also be implemented with decimation-in-frequency FFT showed in Figure 4-24. In this case, an 8-point complex FFT has the implementation structure depicted in Figure 4-27.

![8-point DIF complex FFT](image)

**Figure 4-27 8-point DIF complex FFT**

The corresponding butterfly is illustrated in Figure 4-28, in that the input-output relationship writes

\[ P_{n+1} = P_n + Q_n \]  \[4.47\]

\[ Q_{n+1} = (P_n - Q_n) \cdot W_N^k \]  \[4.48\]
4.6.4 Calculation of Real Forward FFT from Complex FFT

Having only real valued input data \( x(n) \), the computational effort of a \( N \) point real FFT can be reduced to a \( N/2 \) point complex FFT. Firstly, even indexed data \( h(n)=x(2n) \) and odd indexed data \( g(n)=x(2n+1) \) are separated. According to Equation [4.39] and Equation [4.40], the spectrum \( X(k) \) can be decomposed into the spectra \( H(k) \) and \( G(k) \) as follows:

\[
X(k) = H(k) + \left( \cos \frac{2\pi k}{N} - j \sin \frac{2\pi k}{N} \right) G(k) \quad \text{for } k=0 \text{ to } N/2-1. \tag{4.49}
\]

In order to cut the \( N \)-point real FFT into \( N/2 \)-point complex transformation a complex input vector \( y(n) = h(n) + jg(n) \) is formed with the index \( n \) running from 0 to \( N/2-1 \). The real part values are formed by the even indexed input data \( h(n) \). The imaginary part is formed by the odd indexed input data \( g(n) \). Then \( y(n) \) is transformed into the frequency domain resulting in a spectrum consisting of the spectra \( H(k) \) and \( G(k) \).

\[
Y(k) = H(k) + jG(k) = \text{Re}\{Y(k)\} + j\text{Im}\{Y(k)\} \tag{4.50}
\]

Now the complex spectra \( H(k) \) and \( G(k) \) have to be extracted out of the complex spectrum \( Y(k) \). By employing symmetry relations, the spectra \( H(k) \) and \( G(k) \) can be derived from the spectrum \( Y(k) \) as follows:

\[
\begin{align*}
\text{Re}\{H(k)\} &= \frac{\text{Re}\{Y(N/2-k)\} + \text{Re}\{Y(k)\}}{2} \\
\text{Im}\{H(k)\} &= \frac{\text{Im}\{Y(k)\} - \text{Im}\{Y(N/2-k)\}}{2} \\
\text{Re}\{G(k)\} &= \frac{\text{Im}\{Y(k)\} + \text{Im}\{Y(N/2-k)\}}{2} \\
\text{Im}\{G(k)\} &= \frac{\text{Re}\{Y(N/2-k)\} - \text{Re}\{Y(k)\}}{2}
\end{align*} \tag{4.51}
\]
Therefore, computing an N-point real forward FFT has the following steps:
1. Generating the N/2 point complex sequence \( y(n) = x(2n) + jx(2n+1) \),
2. Computing the complex spectra \( Y(k) \) using complex FFT,
3. Extracting \( H(k) \) and \( G(k) \) from computed \( Y(k) \) according to Equation [4.51],
4. Calculating the first N/2+1 points of \( X(k) \) based on Equation [4.49] using \( H(k) \) and \( G(k) \),
5. Using the symmetry relation of the spectra of the real sequence to get the complete N-point \( X(k) \).

### 4.6.5 Calculation of Real Inverse FFT from Complex FFT

Using the algorithm in Section 4.6.4 we can calculate a N-point real-valued FFT through N/2-point complex FFT. Now we present the corresponding inverse FFT algorithm to repeat the original real-valued input sequences based on the FFT spectrum. Usually a N-point real-valued forward FFT produces N-point complex spectrum. To get the original real-valued sequences one can simply input this N-point complex spectrum to an inverse FFT. But it needs N-point real inverse FFT. With the symmetry properties of real-valued FFT a N-point real inverse FFT can be reduced to N/2-point complex inverse FFT.

This can be realized through two steps. In the first step the real FFT spectrum will be unpacked to the corresponding complex spectrum similar with the unpack stage in the forward FFT. Then, the results are inputted into a N/2-point inverse complex FFT to get real-valued sequences.

In the unpack stage only the first \( (N/2+1) \)-point spectrum are needed because the first \( (N/2+1) \)-point spectrum of a N-point real-valued FFT contain all information. According to Equation [4.49] and the symmetry properties of \( H(k) \) and \( G(k) \), we have

\[
\begin{align*}
\text{Re}\{H(k)\} &= \frac{\text{Re}\{X(k)\} + \text{Re}\{X(N/2 - k)\}}{2} \quad \text{for } k = 0 \text{ to } N/2 - 1. \quad [4.52] \\
\text{Im}\{H(k)\} &= \frac{\text{Im}\{X(k)\} - \text{Im}\{X(N/2 - k)\}}{2}
\end{align*}
\]

Defining
\[
G'(k) = X(k) - H(k),
\]

and replacing Equation [4.52] into Equation [4.53] we get
Finally we have the expression of the complex spectrum $Y(k)$ from Equation [4.50]

\[
\begin{align*}
\text{Re}\{G'(k)\} &= \frac{\text{Re}\{X(k)\} - \text{Re}\{X(N/2 - k)\}}{2} \\
\text{Im}\{G'(k)\} &= \frac{\text{Im}\{X(k)\} + \text{Im}\{X(N/2 - k)\}}{2} \\
\end{align*}
\]

where $K=0$ to $N/2-1$.

Summarily, computing an $N$-point real inverse FFT has the following steps:

1. Extracting $H(k)$ according to Equation [4.52],
2. Extracting $G'(k)$ according to Equation [4.54],
3. Computing the complex spectrum $Y(k)$ based on $H(k)$ and $G'(k)$ using Equation [4.55],
4. Calculating $N/2$-point inverse complex FFT with input $Y(k)$. 

\[
\begin{align*}
\text{Re}\{Y(k)\} &= \text{Re}\{H(k)\} - \sin \frac{2\pi k}{N} \cdot \text{Re}\{G'(k)\} - \cos \frac{2\pi k}{N} \cdot \text{Im}\{G'(k)\} \\
\text{Im}\{Y(k)\} &= \text{Im}\{H(k)\} + \cos \frac{2\pi k}{N} \cdot \text{Re}\{G'(k)\} - \sin \frac{2\pi k}{N} \cdot \text{Im}\{G'(k)\} \\
\end{align*}
\]
4.7 C166S V2 Core Implementation Note

4.7.1 Organization of FFT functions

In the library the radix-2 FFT is implemented. Basically there are following three kinds of functions:

- Bit reverse function (Bit_reverse.asm)
- Floating point to 1Q15 format function (FloatToQ15.asm)
- FFT kernel function

The first two functions will be as subroutines called by FFT kernel function. The subroutine Bit_reverse.asm is used for bit reversing the binary presentation of the input indices to get the output data indices. FloatToQ15.asm is used to change the floating point format to 1Q15 fractional format. The kernel FFT realizes the kernel FFT algorithm that may be complex and real FFT. Butterflies are implemented in the form of macros.

The kernel FFT implementation in the C166S V2 Core Library is based on an application note performed early by Siemens HL MCB AT, in which an implementation of a real-valued 1024-point radix-2 decimation-in-time forward FFT for C166 microcontroller family is described. However, there are many basic differences between the two implementations. Firstly, the FFT implementation for C166S V2 Core is not only C-callable but also Assembler-callable, while the early implementation for C166 is only Assembler-callable. Secondly, the FFT implementation for C166S V2 Core is performed with the new DSP MAC instruction set, while the early implementation for C166 has only used the normal instruction set. Therefore, the FFT implementation for C166S V2 Core is a more optimal implementation in comparison with early implementation. Thirdly, the size of input data of the new FFT implementation for C166S V2 Core can be changed from 2 to 2048 points, while the early implementation is only suitable for 1024-point FFT.

This library provides not only forward FFT but also inverse FFT implementation as well as their implementation theory basis.

4.7.2 Implementation of Real Forward FFT

A real valued N point FFT can be reduced to a N/2 point complex FFT followed by an unweave phase in order to compute the N/2 point spectrum. The N/2 complex FFT consists of log₂(N/2) stages and each stage calculates N/4 butterflies.

The input data is stored in the vector FFT_in that consists of N 16 bit words and is defined in C main function. Since we perform an in-place FFT, the input data field will be overwritten by the output data. For providing the trigonometric functions, the precomputed sinus and cosines values are stored in memory. Because the input data and the trigonometric function values are represented by a 15 bit fixed-point fraction in two’s complement (1Q15 format), the floating-point sinus and cosines values have to be changed into 1Q15 format with the routine FloatToQ15. To rearrange the bit reversed
output of the complex FFT and to calculate the twiddle factor $W_k$, a bit reversal index table has been precomputed.

Regarding the $N/2$ point complex FFT, the number of twiddle factors amounts $N/2$. However, due to the symmetry only the first $N/4$ are used. The implementation consists of $\log_2(N/2)$ stages. Each stage contains basically three loops, i.e. Outloop, Mitloop and Inloop. One stage has one Outloop and $N/2$ Inloops, but different Mitloops due to different twiddle factors. Each Inloop implements one butterfly with a twiddle factor. For example, an 8-point DIT complex FFT in Figure 4-25 has the following structure:

Stage 1 (Outloop_1): 4 butterflies with $W_0$
Stage 2 (Outloop_2): 2 butterflies with $W_0$
Stage 3 (Outloop_3): 1 butterfly with $W_0$

2 butterflies with $W_2$
1 butterfly with $W_1$
1 butterfly with $W_2$
1 butterfly with $W_3$

The output of the decimation-in-time FFT shows a bit reversed order that has to be ordered to calculate the final frequency spectrum. Supposing the input data has been in a sequential order, the indices of the output data can be easily computed by bit reversing the binary presentation of the input indices. The table below gives an example of the bit reversal for an 8-point FFT.

<table>
<thead>
<tr>
<th>Order of input data</th>
<th>Order of output data</th>
</tr>
</thead>
<tbody>
<tr>
<td>index</td>
<td>binary</td>
</tr>
<tr>
<td>0</td>
<td>000</td>
</tr>
<tr>
<td>1</td>
<td>001</td>
</tr>
<tr>
<td>2</td>
<td>010</td>
</tr>
<tr>
<td>3</td>
<td>011</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
</tr>
<tr>
<td>5</td>
<td>101</td>
</tr>
<tr>
<td>6</td>
<td>110</td>
</tr>
<tr>
<td>7</td>
<td>111</td>
</tr>
</tbody>
</table>

The second part of the program unweaves the bit reversed output of the $N/2$ point FFT to extract the $N$ point real valued FFT.
4.7.3 Implementation of Real Inverse FFT

From Section 4.6.5 we know that the N-point real inverse FFT can be realized by N/2 point complex inverse FFT. Different from forward FFT the unpack stage will be performed at beginning to get the N/2 point complex input spectra according to N point real FFT spectra.

The input data is the first (N/2+1)-point real FFT spectra and stored in the vector with size N/2+1 that consists of N+2 16 bit words and is defined in C main function. After unpacking the N/2 point complex spectra are strode in vector FFT_out. Note that the input data should be by 1/N scaled real FFT spectra. All data has 1Q15 format. Here we use the name real inverse FFT, which doesn’t mean that the input data is real value. Actually here the input data is complex value coming from a real-valued FFT. The word real indicates that the input value come from a real-valued forward FFT instead of a complex forward FFT.

The butterfly of inverse FFT has the same structure as forward FFT beside the twiddle factor. The twiddle factor of the inverse FFT butterfly has the form \( W_{nk}^{-1} \) instead of \( W_N^{nk} \). Similar with forward FFT the inverse FFT can be also implemented with DIT and DIF. For example, for an 8-point real inverse FFT, if the DIF implementation in Figure 4-27 is used, there are following structure:

Stage 1 (Outloop_1): 1 butterfly with \( W_0 \)
  1 butterfly with \( W_1 \)
  1 butterfly with \( W_2 \)
  1 butterfly with \( W_3 \)

Stage 2 (Outloop_2): 2 butterflies with \( W_0 \)
  2 butterflies with \( W_2 \)

Stage 3 (Outloop_3): 4 butterflies with \( W_0 \)

In this case the output shows the linear order while the input has bit-reversed order. The corresponding butterfly has the structure showed in Figure 4-28.

4.7.4 Description

The following functions are described.

- Bit_reverse.asm
- FloatToQ15.asm
- real_DIT_FFT.asm
- real_DIF_IFFT.asm
Bit_reverse

Reverse the binary bit of the input data

Signature

`DataS Bit_reverse(DataS* X, DataS N)`

Inputs

- **X**: 16 bit Input data vector
- **N**: Size of the vector, N = 2^n (n<12)

Output

- **X**: Bit reversed data vector

Return

- **n**: Exponent of N

Implementation Description

This routine is a subroutine in the DIT FFT implementation packet and used to obtain a bit reversed data in respect to the input data. The bit reversed data is also stored in the vector X.

For an 8 bit input data there is following algorithm:

before bit reverse: b7.b6.b5.b4.b3.b2.b1.b0
after bit reverse: b0.b1.b2.b3.b4.b5.b6.b7

Pseudo code

```c
{
    ; X = input/output data vector
    ; N = the size of the vector, N = 2^n (n<12)

    DataS* X;       //input/output vector
    DataS N;        //size of vector N
    DataS i, k, n;

    for(k=0; k<N; k++)
    {
        temp = 0;
        for(i=15; i>0; i--)
        {
            temp = temp + (X(k)<<i)>>(15-i);
        }
        //write the output into X
        X(k) = temp;
    }

    n = (DataS)log10(N);

    Y = n;
    return Y;  //Filter Output returned
}
```

Techniques
Bit_reverse

Reverse the binary bit of the input data (cont’d)

Assumptions

- 16 bit Input data vector X
- \( N = 2^n \ (n < 12) \)

Register Usage

- From .c file to .asm file:
  - R12 points to input vector X.
  - \((R13) = N\)
- From .asm file to .c file:
  - \((R4) = n\) (exponent of N)

Memory Note

![Diagram showing bit_reverse operation](image)

Figure 4-29 Bit_reverse

Example

*C166Lib\Examples\Fft\bit_rev.c*

Cycle Count

Store state 3
### Bit_reverse

Reverse the binary bit of the input data (cont’d)

<table>
<thead>
<tr>
<th>Description</th>
<th>Code Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exponent determination</td>
<td>6 bytes</td>
</tr>
<tr>
<td>Bit reverse:</td>
<td></td>
</tr>
<tr>
<td>if $N=2^1$</td>
<td>3 bytes</td>
</tr>
<tr>
<td>if $N=2^2$, $N=2^3$</td>
<td>10 bytes</td>
</tr>
<tr>
<td>if $N=2^4$, $N=2^5$</td>
<td>12 bytes</td>
</tr>
<tr>
<td>if $N=2^6$, $N=2^7$</td>
<td>14 bytes</td>
</tr>
<tr>
<td>if $N=2^8$, $N=2^9$</td>
<td>16 bytes</td>
</tr>
<tr>
<td>if $N=2^{10}$, $N=2^{11}$</td>
<td>18 bytes</td>
</tr>
<tr>
<td>Restore state</td>
<td>3 bytes</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>310 bytes</strong></td>
</tr>
</tbody>
</table>

### Code Size

<table>
<thead>
<tr>
<th>Description</th>
<th>Code Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Store state</td>
<td>6 bytes</td>
</tr>
<tr>
<td>Exponent determination</td>
<td>12 bytes</td>
</tr>
<tr>
<td>Bit reverse:</td>
<td></td>
</tr>
<tr>
<td>if $N=2^1$</td>
<td>6 bytes</td>
</tr>
<tr>
<td>if $N=2^2$, $N=2^3$</td>
<td>20 bytes</td>
</tr>
<tr>
<td>if $N=2^4$, $N=2^5$</td>
<td>24 bytes</td>
</tr>
<tr>
<td>if $N=2^6$, $N=2^7$</td>
<td>28 bytes</td>
</tr>
<tr>
<td>if $N=2^8$, $N=2^9$</td>
<td>32 bytes</td>
</tr>
<tr>
<td>if $N=2^{10}$, $N=2^{11}$</td>
<td>36 bytes</td>
</tr>
<tr>
<td>Restore state</td>
<td>6 bytes</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>310 bytes</strong></td>
</tr>
</tbody>
</table>
FloatTo1Q15  

**Change the floating point format to fixed point format 1Q15**

**Signature**  
DataS FloatTo1Q15 ( float x)

**Inputs**  
x : Input in float format

**Output**  
:

**Return**  
: Output in 1Q15 format

**Implementation Description**  
This routine is a subroutine in the DIT FFT implementation packet and used to change the floating point data into a 1Q15 fixed point format.

The floating point format has the structure:

1. word: s e e e e e e e e mmmmmm
2. word: mmmmmm

where s = sign, e = exponent, m = mantissa. After format change we have the 1Q15 fixed point data with the structure:

s. b1 b2 b3 b4 ...... b15

sig. 2^{-1} 2^{-2} 2^{-3} 2^{-4} ...... 2^{-15}

For example:

<table>
<thead>
<tr>
<th>binary</th>
<th>hex</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0111 1111 1111 1111</td>
<td>7FFF</td>
<td>+1</td>
</tr>
<tr>
<td>0110 0000 0000 0000</td>
<td>6000</td>
<td>+0.75</td>
</tr>
<tr>
<td>1010 0000 0000 0000</td>
<td>A000</td>
<td>-0.75</td>
</tr>
<tr>
<td>1000 0000 0000 0000</td>
<td>8000</td>
<td>-1</td>
</tr>
</tbody>
</table>

**Pseudo code**

**Assumption**  
DPRAM begins with the address # f200h (it can be changed)

**Register Usage**

- From .c file to .asm file:
  
  (R12) = s e e e e e e e e mmmmmm
  
  (R13) = mmmmmm

- From .asm file to .c file:
  
  The output is stored in R4.
FloatTo1Q15  Change the floating point format to fixed point format
1Q15 (cont’d)

Memory Note

![Diagram of floating point format to fixed point format (1Q15)]

Figure 4-30  FloatTo1Q15

Example

*C166Lib\Examples\Fft\Float_1Q15.c*

Cycle Count

<table>
<thead>
<tr>
<th>Step</th>
<th>Cycle Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Store state</td>
<td>3</td>
</tr>
<tr>
<td>Read exponent</td>
<td>10</td>
</tr>
<tr>
<td>Read mantissa</td>
<td>7</td>
</tr>
<tr>
<td>Restore state</td>
<td>2</td>
</tr>
<tr>
<td>Output</td>
<td>12</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>31</strong></td>
</tr>
</tbody>
</table>

Code Size

<table>
<thead>
<tr>
<th>Step</th>
<th>Code Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Store state</td>
<td>4 bytes</td>
</tr>
<tr>
<td>Read exponent</td>
<td>20 bytes</td>
</tr>
<tr>
<td>Read mantissa</td>
<td>14 bytes</td>
</tr>
<tr>
<td>Output</td>
<td>20 bytes</td>
</tr>
<tr>
<td>Restore state</td>
<td>4 bytes</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>62</strong> bytes</td>
</tr>
</tbody>
</table>
real_DIT_FFT Real Forward Radix-2 Decimation-in-Time Fast Fourier Transformation

Signature
void real_DIT_FFT (    DataS*        x,
                        DataS*       index,
                        DataS        exp,
                        DataS*       table,
                        DataS*       X
                    )

Inputs
x : 16 bit real input vector
index : Bit reversed input index vector
exp : Exponent of the input block size
table : The precalculated trigonometric function (sinus and cosine) table

Output
X : The FFT output vector in 1Q15 format

Return
:

Implementation Description
This function computes the N-point real forward radix-2 decimation-in-time Fast Fourier Transform on the given N-point real input array. The detailed implementation is given in the Section 4.7.2.

The function is implemented as a complex FFT of size N/2 followed by a unpack stage to unpack the real FFT results. Normally an FFT of a real sequence of size N produces a complex sequence of size N or 2N real numbers that will not fit in the input sequence. To accommodate all the results without requiring extra memory locations, the output reflects only half of the complex spectrum plus the spectrum at the nyquist point (N/2). This still provides the full information because an FFT of a real sequence has even symmetry around the nyquist point.
real_DIT_FFT
Real Forward Radix-2 Decimation-in-Time Fast
Fourier Transformation (cont’d)

Pseudo code
{
    DataS* table; // sinus and cosine table
    DataS* x; // 16 bit real input vector
    CplxS* X; // FFT output vector
    CplxS P(n), P(n+1), Q(n), Q(n+1), Y(N/2), H(N/2), G(N/2);
    DataS k, i, n;

    // building the complex sequences P(n) and Q(n)
    Q(n) = x(2n) + jx(2n+1);
    P(n) = x(2n+N/4) + jx(2n+N/4+1);

    // Outloop = 1 to exp=log₂(N/2)
    for(k=0; k++; k<exp)
    {
        // Midloop = 1 to N/4 according to which stage
        for(i=0; i<Midloop; i++)
        {
            // Inloop = N/2 to 1 (number of butterflies)
            for(n=0; n<Inloop; n++)
            {
                P(n+1) = Re(P(n)) + Re(Q(n)) * cos(X) + Im(Q(n)) * sin(X)
                + j[Im(P(n)) - Im(Q(n)) * cos(X) - Re(Q(n)) * sin(X)];
                Q(n+1) = Re(P(n)) - Re(Q(n)) * cos(X) - Im(Q(n)) * sin(X)
                + j[Im(P(n)) - Im(Q(n)) * cos(X) - Re(Q(n)) * sin(X)];
            }
            // if all elements processed, jump out of the Midloop
        }
    }

    // output the first half of N/2 point complex FFT values Y,
    // Extracting H and G from computed Y according to Equation [4.51],
    // Calculating the first N/2 points of X based on Equation [4.49] using H and G
}

Assumption
    • Scale factor = 1/N (N = 2^exp), preventing overflow
    • FFT spectra = N * X(k)

Techniques
    • Memory bandwidth conflicts removing
    • Use of CoMAC instructions
    • Instruction re-ordering
    • output converted to 16-bit with saturation
real_DIT_FFT

Real Forward Radix-2 Decimation-in-Time Fast Fourier Transformation (cont’d)

Register Usage

- From .c file to .asm file:
  - R12 points to input vector x
  - R13 points to index vector
  - (R14) = exp
  - R15 points to the trigonometric function table
  - (R0) contains the pointer of output vector X
real_DIT_FFT Real Forward Radix-2 Decimation-in-Time Fast Fourier Transformation (cont'd)

Memory Note

Figure 4-31 Real_DIT_FFT
real_DIT_FFT Real Forward Radix-2 Decimation-in-Time Fast Fourier Transformation (cont’d)

Example

Example: \( C166Lib\Examples\real_FFT\ real_FFT.c \)

**Cycle Count**

- **Store state**: 11
- **Read parameters and memory initialization**: 16
- **Butterfly**: 31
- **FFT kernel**:
  \[
  (FFT\_cycle) = \sum_{n=0}^{\text{exp}-2} \left\{ 6 + 2^n \cdot 29 + \frac{N}{4} \cdot 36 \right\}
  \]
  where \( \text{exp} = \log_2 N \).
- **Unpack stage**:
  \( (U\_cycle) = 71 + (N/4-1)\times44 \)
- **Restore state**: 11
- **Return**: 1

**Total**

\( 39 + FFT\_cycle + U\_cycle \)

Examples:
- \( N = 8 \): cycle = 380
- \( N = 16 \): cycle = 896
- \( N = 1024 \): cycle = 109131

**Code Size**

- **Store state**: 22 bytes
- **Read parameters and memory initialization**: 32 bytes
- **Butterfly**: 96 bytes
- **FFT kernel**: 88 bytes
real_DIT_FFT

Real Forward Radix-2 Decimation-in-Time Fast Fourier Transformation (cont’d)

<table>
<thead>
<tr>
<th>Description</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unpack Stage</td>
<td>178</td>
</tr>
<tr>
<td>Restore state</td>
<td>22</td>
</tr>
<tr>
<td>Return</td>
<td>2</td>
</tr>
</tbody>
</table>

Total: 442 bytes
real_DIF_IFFT  Real Inverse Radix-2 Decimation-in-Frequency Fast Fourier Transformation

Signature
void real_DIF_IFFT ( DataS* x,
DataS* index,
DataS exp,
DataS* table,
DataS* X )

Inputs
index : Bit reversed input index vector
exp : Exponent of the input block size
table : The precalculated trigonometric function (sinus and cosine) table
X : Input vector, FFT spectra in 1Q15 format

Output
x : 16 bit output of inverse FFT, real sequences

Return

Implementation Description
This function computes the N-point real inverse radix-2 Fast Fourier Transform with decimation-in-frequency on the given N-point FFT spectra. The detailed implementation is given in the Section 4.7.3.

The function is implemented as a unpack stage followed by a complex inverse FFT of size N/2. The unpack stage aims to unpack the N-point real FFT as N/2-point complex FFT. The N/2-point complex inverse FFT is implemented with decimation-in-frequency structure showed in Figure 4-27, where the butterfly $W_N^{nk}$ should be replaced by $W_N^{-nk}$. 
Pseudo code

{  
  CplxS*  X;            //input FFT spectra  
  DataS*  x;            //16 bit real output vector  
  CplxS   P(n), P(n+1), Q(n), Q(n+1), Y(N/2), H(N/2), G'(N/2);  
  DataS   k,i,n;  
  
  //extracting H(k) according to Equation [4.52]  
  Re{H(n)} = (Re{X(n)} + Re{X(N/2-n)})/2;  
  Im{H(n)} = (Im{X(n)} - Im{X(N/2-n)})/2;  
  
  //extracting G’(k) according to Equation [4.54]  
  Re{G’(n)} = (Re{X(n)} - Re{X(N/2-n)})/2;  
  Im{G’(n)} = (Im{X(n)} + Im{X(N/2-n)})/2;  
  
  //computing the complex spectrum Y according to Equation [4.55]  
  Re{Y(n)} = Re{H(n)}-sin(2*pi*n/N)*Re{G’(n)}-cos(2*pi*n/N)*Im{G’(k)};  
  Im{Y(n)} = Im{H(n)}+cos(2*pi*n/N)*Re{G’(n)}-sin(2*pi*n/N)*Im{G’(k)};  
  
  //define  
  P(n) = Y(2n); Q(n) = Y(2n+1);  
  
  //N/2-point inverse complex FFT  
  //Outloop = 1 to exp=log₂(N/2)  
  for(k=0; k++; k<exp)  
  {  
    //Midloop = 1 to N/4 according to which stage  
    for(i=0; i<Midloop; i++)  
    {  
      //Inloop = 1 to N/4 (number of butterflies)  
      for(n=0; n<Inloop; n++)  
      {  
        P(n+1) = Re{P(n)}+Re{Q(n)}+j[Im{P(n)}+Im{Q(n)}];  
        Re{Q(n+1)} = (Re{P(n)}-Re{Q(n)})*cos(x)-  
          (Im{Q(n)}-Im{Q(n)})*sin(X);  
        Im{Q(n+1)} = (Im{P(n)}-Im{Q(n)})*cos(X)+  
          (Re{Q(n)}-Re{Q(n)})*sin/X;  
      }  
    }  
  }  
}  

real_DIF_IFFT Real Inverse Radix-2 Decimation-in-Frequency Fast Fourier Transformation (cont’d)
real_DIF_IFFT Real Inverse Radix-2 Decimation-in-Frequency Fast Fourier Transformation (cont’d)

Assumption
- The inputs are the scaled FFT spectra with factor = 1/N (N = 2^{exp})
- Output without scale

Techniques
- Memory bandwidth conflicts removing
- Use of CoMAC instructions
- Instruction re-ordering
- Output converted to 16-bit with saturation

Register Usage
- From .c file to .asm file:
  - R12  pointer of output vector x
  - R13  pointer of index vector
  - (R14) = exp
  - R15  pointer of trigonomic function table
  - (R0)  contains the pointer of input vector X
Function Descriptions

real_DIF_IFFT Real Inverse Radix-2 Decimation-in-Frequency Fast Fourier Transformation (cont’d)

Memory Note

Input-Buffer | Output-Buffer | Stage 1 -- Stage exp-1 | Output-Buffer
---|---|---|---
X(N/2) | Y(N/2-1) | x(N-1) | x(N-2)
X(N/2-1) | Y(N/2-2) | . | .
. | . | . | .
X(3) | Y(3) | x(3) | .
X(2) | Y(2) | x(2) | .
X(1) | Y(1) | x(1) | .
X(0) | Y(0) | x(0) | .

32 bit | 32 bit | 16 bit

N/2+1 point complex input data in 1Q15 in linear order
rearranged N/2 point complex data in 1Q15 in bit-reversed order

First Stage:
The N/2+1 point complex data is rearranged as N/2 point complex data

Figure 4-32 Real_DIF_IFFT
real_DIF_IFFT

Real Inverse Radix-2 Decimation-in-Frequency Fast
Fourier Transformation (cont’d)

Example

\[ C166Lib\ Examples\ \text{real\_IFFT}\ \text{real\_IFFT.}\ c \]

Cycle Count

- Store state: 11
- Read parameters: 9
- Initialization:
  - Butterfly: 28
  - Unpack stage: \(6 + 48\cdot N/4\)
- IFFT kernel:
  \[ \sum_{n=0}^{\exp-2} \left\{ 7 + 2^n \cdot 28 + \frac{N}{4} \cdot 33 \right\} \]
  where \( \exp = \log_2 N \).
- Store state: 11
- Return: 1

Total: \(32 + \text{IFFT}\_\text{cycle} + \text{U}\_\text{cycle}\)

Examples:
- \(N = 8\) : cycle = 354
- \(N = 16\) : cycle = 833
- \(N = 1024\) : cycle = 77670

Code Size

- Store state: 22 bytes
- Read parameters: 18 bytes
- Initialization:
  - Butterfly: 78 bytes
  - Unpack Stage: 122 bytes
  - IFFT kernel: 102 bytes
real_DIF_IFFT  Real Inverse Radix-2 Decimation-in-Frequency Fast Fourier Transformation (cont’d)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Restore state</td>
<td>22</td>
</tr>
<tr>
<td>Return</td>
<td>2</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>366</strong></td>
</tr>
</tbody>
</table>
4.8 Matrix Operations

A matrix is a rectangular array of numbers (or functions) enclosed in brackets. These numbers (or functions) are called entries or elements of the matrix. The number of entries in the matrix is product of number of rows and columns. An $m \times n$ matrix means matrix with $m$ rows and $n$ columns. In the double-subscript notation for the entries, the first subscript always denotes the row and the second the column.

4.8.1 Descriptions

The following Matrix Operations are described.

- Multiplication
Matrix [MxN][NxP] Matrix Multiplication

Signature
void Matrix(
    DataS* x_1,
    DataS* x_2,
    DataS* y,
    DataS row1,
    DataS col1,
    DataS row2,
    DataS col2
)

Inputs
x_1 : Pointer to the first matrix in 1Q15 format
x_2 : Pointer to the second matrix in 1Q15 format
row1 : M, the number of rows in the first matrix
col1, row2 : N, the number of columns in the first matrix, the number of rows in the second matrix
col2 : P, M, the number of rows in the second matrix

Output
y : Pointer to the output in 1Q15 format

Return

Implementation Description
The multiplication of two matrices A and B is done. The multiplication results will be stored in the [MxP] matrix y. Both the input matrices and output matrix are 16-bit. All the elements of the matrix are stored row-by-row in the buffer.

\[
\begin{bmatrix}
  y_{11} & y_{12} & \ldots & y_{1P} \\
  y_{21} & y_{22} & \ldots & y_{2P} \\
  \vdots & \vdots & \ddots & \vdots \\
  y_{M1} & y_{M2} & \ldots & y_{MP}
\end{bmatrix}
= \begin{bmatrix}
  a_{11} & a_{12} & \ldots & a_{1N} \\
  a_{21} & a_{22} & \ldots & a_{2N} \\
  \vdots & \vdots & \ddots & \vdots \\
  a_{M1} & a_{M2} & \ldots & a_{MN}
\end{bmatrix}
\times
\begin{bmatrix}
  b_{11} & b_{12} & \ldots & b_{1P} \\
  b_{21} & b_{22} & \ldots & b_{2P} \\
  \vdots & \vdots & \ddots & \vdots \\
  b_{N1} & b_{N2} & \ldots & b_{NP}
\end{bmatrix}
\]
Matrix

**[MxN][NxP] Matrix Multiplication** (cont'd)

**Pseudo code**

```c
{
  DataS x_1[M*N];          //first input matrix
  DataS x_2[N*P];          //second input matrix
  DataS y[M*P];            //output matrix
  DataS i,j;

  for (i=0 to M-1)
    for (k=0 to P-1)
    {
      y(i,k) = 0;
      for (j=0 to N-1)
        y(i,k) = y(i,k) + x_1(i,j)*x_2(j,k);
    }
}
```

**Assumption**

**Techniques**
- Memory bandwidth conflicts removing
- Use of CoMAC instructions
- Instruction re-ordering

**Register Usage**
- From .c file to .asm file:
  - R12 points to the first input matrix x_1.
  - R13 points to the second input matrix x_2.
  - R14 points to the output matrix y
  - (R15) = row1
  - ((R0+2)) = col1
  - ((R0+4)) = row2
  - ((R0+4)) = col2
Matrix

**[MxN][NxP] Matrix Multiplication** (cont’d)

**Memory Note**

```
<table>
<thead>
<tr>
<th>High Addr.</th>
<th>Low Addr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>a[M-1,N-1]</td>
<td>a[0,0]</td>
</tr>
<tr>
<td>.</td>
<td>a[0,1]</td>
</tr>
<tr>
<td>a[1,1]</td>
<td>.</td>
</tr>
<tr>
<td>a[1,0]</td>
<td>.</td>
</tr>
<tr>
<td>a[0,N-1]</td>
<td>.</td>
</tr>
<tr>
<td>.</td>
<td>a[0,1]</td>
</tr>
<tr>
<td>.</td>
<td>a[1,1]</td>
</tr>
</tbody>
</table>

DPRAM
```

```
<table>
<thead>
<tr>
<th>High Addr.</th>
<th>Low Addr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>b[N-1,P-1]</td>
<td>b[0,0]</td>
</tr>
<tr>
<td>b[N-1,P-2]</td>
<td>b[0,1]</td>
</tr>
<tr>
<td>.</td>
<td>b[0,2]</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
</tr>
</tbody>
</table>

Memory
```

```
<table>
<thead>
<tr>
<th>High Addr.</th>
<th>Low Addr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>y[M-1,P-1]</td>
<td>y[0,0]</td>
</tr>
<tr>
<td>y[M-1,P-2]</td>
<td>y[0,1]</td>
</tr>
<tr>
<td>.</td>
<td>y[0,2]</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
</tr>
</tbody>
</table>

Memory
```

**Figure 4-33 Matrix**

**Example**  
*C166Lib\Examples\Matrix\Matrix_N.c*

**Cycle Count**
### Matrix

**[MxN][NxP] Matrix Multiplication** (cont’d)

<table>
<thead>
<tr>
<th>Step</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Store state</td>
<td>5</td>
</tr>
<tr>
<td>Read parameters and memory</td>
<td>14</td>
</tr>
<tr>
<td>Initialization</td>
<td></td>
</tr>
<tr>
<td>Matrix loop</td>
<td>( M(4+P(6+3N)) )</td>
</tr>
<tr>
<td>Restore state</td>
<td>5</td>
</tr>
<tr>
<td>Return</td>
<td>1</td>
</tr>
</tbody>
</table>

**Total** \( 25 + M(4 + P(6 + 3N)) \)

Example: \( M = 4, N = 3, P = 5 \)

cycle = 341

### Code Size

<table>
<thead>
<tr>
<th>Step</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Store state</td>
<td>10</td>
</tr>
<tr>
<td>Read parameters and memory</td>
<td>32</td>
</tr>
<tr>
<td>initialization</td>
<td></td>
</tr>
<tr>
<td>Matrix loop</td>
<td>40</td>
</tr>
<tr>
<td>Restore state</td>
<td>10</td>
</tr>
<tr>
<td>Return</td>
<td>2</td>
</tr>
</tbody>
</table>

**Total** 94 bytes
4.9 Mathematical Operations

Here we provide some assembly routines of basic mathematical operations which can be used in many DSP algorithms, such as speech codecs and spectrum analysis.

The following routines are described.

- Sine
- N-th order power series
- Windowing with N coefficients
Sine function

**Signature**

DataS Sine( DataS* x)

**Inputs**

x : By pi normalized input value between [-1,1] in 1Q15 format, 
x=xrad/pi, where xrad contains the angle in radians [-p1,p1].

**Output**

Return y : output in 1Q15 format

**Implementation Description**

For the input value in 1th quadrant (0, π/2) the Sin(x) can be approximated using the polynomial expansion.

\[
\sin(x) = 3,140625(x) + 0,02026367(x)^2 - 5,325196(x)^3 + 0,5446778(x)^4 + 1,800293(x)^5 \quad [4.56]
\]

\[0 \leq x \leq 0.5 \, .\]

The values of Sine function in other quadrants are computed by using the relations,

\[\sin(-x) = -\sin(x) \quad \text{and} \quad \sin(180 - x) = \sin x \, .\]

The function takes 16 bit input in 1Q15 format to accommodate the range (-1, 1) . The output is 16 bits in 1Q15 format. Coefficients are stored in 4Q12 format.

The absolute value of the input is calculated. If the input is negative (III/IV Quadrant), then sign=1. If absolute value of the input is greater than 1/2 (II/III Quadrant), it is subtracted from 1. If sign=1, the result is negated to give the final sine result.

To have an optimal implementation with zero overhead load store, the polynomial in **Equation [4.56]** is rearranged as below.
Hence, 4 multiply-accumulate and 1 multiply instruction will compute the expression Equation [4.57].

**Pseudo code**

```c
{
    DataS a[5];          //coefficient vector
    DataS x;             //input value
    DataS y;             //output value
    DataS i;
    DataS sign;          //sign of input

    //determine the sign of input
    sign = 0;
    if(x<0)
        sign = 1

    //if x in III/IV quadrant
    if(abs(x)>0.5)
        x = 1-abs(x);

    //polynomial
    y = a(0)*x + a(1);
    y = y*x + a(2);
    y = y*x + a(3);
    y = y*x + a(4);
    y = y*x;

    //x in I/II quadrant
    if(sign == 0)
        y = y;
    //x in III/IV quadrant
    else if(sign == 1)
        y = -y;
}
```

**Assumption**
Sine

Sine function (cont’d)

Techniques
- Use of CoMAC instructions
- Instruction re-ordering

Register Usage
- From .c file to .asm file:
  (R12) = x.
- From .asm file to .c file
  y is stored in R4

Memory Note

Figure 4-34 Sine

Example

\[ \sum_{i=0}^{4} a(i) x^i \]

C166Lib\Examples\Math\Sine.c

Cycle Count

Store state 2
## Sine

### Sine function (cont’d)

<table>
<thead>
<tr>
<th>Component</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory Initialization</td>
<td>1</td>
</tr>
<tr>
<td>Polynomial</td>
<td>28</td>
</tr>
<tr>
<td>Restore state</td>
<td>2</td>
</tr>
<tr>
<td>Return</td>
<td>1</td>
</tr>
</tbody>
</table>

**Total** 34 bytes

### Code Size

<table>
<thead>
<tr>
<th>Component</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Store state</td>
<td>4</td>
</tr>
<tr>
<td>Memory initialization</td>
<td>2</td>
</tr>
<tr>
<td>Polynomial</td>
<td>90</td>
</tr>
<tr>
<td>Restore state</td>
<td>4</td>
</tr>
<tr>
<td>Return</td>
<td>2</td>
</tr>
</tbody>
</table>

**Total** 102 bytes
P_series       N-th order power series

Signature       DataS P_series(DataS* a, DataS* IN, DataS N)

Inputs
  a : Pointer to the coefficient vector in 1Q15 format
  IN : Input value
  N : Size of series that must be even number

Output

Return
  Y : Series output in 1Q15 format

Implementation
Description
The routine is implemented with CoMUL and CoADD instructions. To optimize the routine, the implementation uses "loop unrolling" technique and assumes that N is even. The implementation formula is:

\[ y = \sum_{i=0}^{N} a(i) \cdot x^i = [[[a(N) \cdot x + a(N-1)]x + a(N-2)]x + a(N-3)]x + ...] \]

Pseudo code
{
  DataS   a[N];          //coefficient vector
  DataS   X;             //input value
  DataS   Y;             //output value
  DataS   i;

  Y = a(N);
  for(i=0; i<N; i++)
    Y = Y*X + a(N-i);
}

Assumption
  • N = even number

Techniques
  • Use of CoMAC instructions
  • Loop unrolling
  • Instruction re-ordering
P_series  N-th order power series (cont'd)

Register Usage
- From .c file to .asm file:
  R12 points to the coefficient vector.
  \(((\text{R13})) = \text{IN} \)
  \((\text{R14}) = \text{N} \)
- From .asm file to .c file
  Y is stored in R4

Memory Note

Figure 4-35 P_series

Example

\textit{C166Lib Examples Math Power_Series.c}

Cycle Count

\begin{array}{|c|}
\hline
\text{Store state} & 2 \\
\hline
\end{array}
P_series  

N-th order power series (cont’d)

- Memory: 11
- Initialization
- Series loop: $7N/2$
- Restore state: 2
- Return: 1

Total: $16 + 7N/2$

Example:
N = 10
cycle = 51

Code Size

- Store state: 4 bytes
- Memory initialization: 22 bytes
- Series loop: 28 bytes
- Restore state: 4 bytes
- Return: 2 bytes

Total: 60 bytes
## Windowing with N coefficients

### Signature

```c
void Windowing(DataS* h, DataS* x, DataS* y, DataS N)
```

### Inputs
- **h**: Pointer to the coefficient vector in 1Q15 format
- **x**: Pointer to the input vector in 1Q15 format
- **N**: The length of the window

### Output
- **y**: Output vector in 1Q15 format

### Implementation Description

To optimize the routine, the implementation uses "loop unrolling" technique and assumes that N is an even number. The implementation formula is:

\[ y(i) = x(i) \cdot w(i), \quad \text{for} \quad i = 0, 1, \ldots, N - 1 \]

### Pseudo code

```c
{  
    DataS w[N]; //coefficient vector
    DataS x[N]; //input value
    DataS y[N]; //output value
    DataS i;

    for(i=0; i<N; i++)
        y(i) = x(i) * w(i);
}
```

### Assumption
- N should be an even number.

### Techniques
- Use of CoMUL instructions
- Loop unrolling with the factor 2
Windowing with N coefficients (cont’d)

Register Usage

- From .c file to .asm file:
  R12 points to the coefficient vector.
  R13 points to the input vector.
  R14 points to the output vector.
  (R15) = N
- From .asm file to .c file
Windowing

Windowing with N coefficients (cont’d)

Memory Note

![Diagram of windowing with N coefficients]

**Figure 4-36 Windowing**

**Example**

*C166Lib\Examples\Math\Window.c*

**Cycle Count**
## Windowing

<table>
<thead>
<tr>
<th>Description</th>
<th>Code Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory Initialization</td>
<td>4 bytes</td>
</tr>
<tr>
<td>Loop</td>
<td>7N/2 bytes</td>
</tr>
<tr>
<td>Return</td>
<td>1 byte</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>5 + 7N/2 bytes</td>
</tr>
</tbody>
</table>

**Example:**
- N = 128
- cycle = 453

## Code Size

<table>
<thead>
<tr>
<th>Description</th>
<th>Code Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory initialization</td>
<td>8 bytes</td>
</tr>
<tr>
<td>Loop</td>
<td>24 bytes</td>
</tr>
<tr>
<td>Return</td>
<td>2 bytes</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>34 bytes</td>
</tr>
</tbody>
</table>
4.10 Statistical Functions

The following statistical functions are described.

- Correlation
  - Cross-correlation
  - Autocorrelation

4.10.1 Correlation

4.10.1.1 Definitions of Correlation

Correlation determines the degree of similarity between two signals. If two signals are identical their correlation coefficient is 1, and if they are completely different it is 0. If they are identical by 180 phase shift between them, then the correlation coefficient is -1.

There are two types of correlation, Cross-Correlation and Autocorrelation.

When two independent signals are compared, the procedure is cross-correlation. When the same signal is compared to phase shifted copies of itself, the procedure is autocorrelation. Autocorrelation is used to extract the fundamental frequency of a signal. The distance between correlation peaks is the fundamental period of the signal.

Suppose that N1 and N2 represent the size of input signals x1 and x2, respectively, and N=N1+N2 and $N1 \geq N2$. Extending x1 to N-point vector through adding N2-points of zero at the beginning, and x2 to N-point vector through adding N1-points of zero in the end, we can define the discrete cross-correlation as follows.

Rau Cross-correlation:

$$r(j) = \sum_{i=0}^{N1+j-1} x1(i)x2(i+j), \quad -N2 + 1 \leq j \leq N1 - 1$$  \hfill [4.58]

Biased Cross-correlation:

$$r(j) = \frac{1}{N1} \times \sum_{i=0}^{N1+j-1} x1(i)x2(i+j), \quad -N2 + 1 \leq j \leq N1 - 1$$  \hfill [4.59]

Unbiased Cross-correlation:

$$r(j) = \frac{1}{N - \text{abs}(j)} \times \sum_{i=0}^{N1+j-1} x1(i)x2(i+j), \quad -N2 + 1 \leq j \leq N1 - 1$$  \hfill [4.60]
The above definitions contain the full-length cross-correlation of the real input signals \( x_1 \) and \( x_2 \) with \( N \) points output, which consists of \( N_2 \) points of the negative-side and \( N_1 \) points of the positive-side.

If the input vectors \( x_1 \) and \( x_2 \) are same and equal to \( x \) with the size of \( N \), we have the following definitions of the positive-side of the autocorrelation.

**Rau Autocorrelation:**

\[
 r(j) = \sum_{i=0}^{N-j-1} x(i)x(i+j) \quad \text{for } j = 0 \text{ to } Nr \leq N - 1 \quad [4.61]
\]

**Biased Autocorrelation:**

\[
 r(j) = \frac{1}{N} \times \sum_{i=0}^{N-j-1} x(i)x(i+j) \quad \text{for } j = 0 \text{ to } Nr \leq N - 1 \quad [4.62]
\]

**Unbiased Autocorrelation:**

\[
 r(j) = \frac{1}{N-j} \times \sum_{i=0}^{N-j-1} x(i)x(i+j) \quad \text{for } j = 0 \text{ to } Nr \leq N - 1 , \quad [4.63]
\]

where \( j \) is the lag value, as it indicates the shift/lag considered for the \( r(j) \) autocorrelation. \( Nr \) is the correlation length and it determines how much data is used for each autocorrelation result. Note that the full-length autocorrelation of vector \( x \) will have \( 2N-1 \) points with even symmetry around the lag 0 point \( r(0) \). The above definitions define only the positive half for memory and computational savings.

### 4.10.1.2 Implementation Note

Directly implementing the cross-correlation according to definitions in Equation [4.56], Equation [4.57] and Equation [4.58] has difficulty due to the negative index. To make the algorithms realizable in assembly we need to rewrite the definitions.

**Rau Cross-correlation:**

The negative-side can be rewritten as with positive index

\[
 r(j) = \sum_{i=0}^{j} x_1(i)x_2(N2 - j - 1 + i) \quad , \quad 0 \leq j \leq N2 - 1 \quad [4.64]
\]
and the positive-side as

\[
\begin{align*}
\text{Biased Cross-correlation:} \\
\text{The negative-side:} \\
&\quad r(j + N2) = \frac{1}{N1} \times \sum_{i=0}^{j} x1(i) x2(N2 - j - 1 + i), \quad 0 \leq j \leq N1 - N2 - 1 \tag{4.67} \\
&\text{The positive-side:} \\
&\quad r(j + N2) = \frac{1}{N1} \times \sum_{i=0}^{N2-1} x1(i + j) x2(i), \quad 0 \leq j \leq N1 - N2 - 1 \tag{4.68} \\
&\quad r(j + N2) = \frac{1}{N1} \times \sum_{i=0}^{N1 - j - 1} x1(i) x2(N2 - j - 1 + i), \quad N1 - N2 < j \leq N1 - 1 \tag{4.69} \\
\text{Unbiased Cross-correlation:} \\
\text{The negative-side:} \\
&\quad r(j) = \frac{1}{\text{abs}(j + 1)} \times \sum_{i=0}^{j} x1(i) x2(N2 - j - 1 + i), \quad 0 \leq j \leq N2 - 1 \tag{4.70} \\
&\text{The positive-side:}
\end{align*}
\]
4.10.2 Implementation Description

The following routines are described.

- Raw autocorrelation
- Biased autocorrelation
- Unbiased autocorrelation
- Raw cross-correlation
- Biased cross-correlation
- Unbiased cross-correlation

\[
\begin{align*}
\hat{r}(j + N2) & = \frac{1}{N2} \times \sum_{i=0}^{N2 - 1} x1(i + j)x2(i), \quad 0 \leq j \leq N1 - N2 - 1 \quad [4.71] \\
\hat{r}(j + N2) & = \frac{1}{N1 - \text{abs}(j)} \times \sum_{i=0}^{N1 - j - 1} x1(i)x2(N2 - j - 1 + i), \quad N1 - N2 < j \leq N1 - 1 \quad [4.72]
\end{align*}
\]
Auto_raw  Raw autocorrelation

Signature  void Auto_raw(    DataS*       x,
                         DataS*       y,
                         DataS        N_x,
                         DataS        N_y
                      )

Inputs  x  :  Pointer to input vector in 1Q15 format
N_x  :  The length of input vector

Output  y  :  Pointer to output vector in 1Q15 format
N_y (Nr)  :  The length of output vector

Return  :

Implementation
Description  The function performs the positive side of the raw autocorrelation function of real vector x according to the definition in Equation [4.59]. The arguments to the function are pointer to the input vector, pointer to output buffer to store autocorrelation result, size of input buffer and number of auto correlated outputs desired. The input values and output values are both in 16 bit fractional format.

Pseudo code
{
    DataS *x;    //Ptr to input vector
    DataS *y;    //Ptr to output vector
    DataS N_x;   //size of input
    DataS N_y;   //size of autocorrelation result
    DataS i,j;

    for(j=0; j<N_y; j++)
    {
        y[j] = 0;
        for(i=0; i<N_x-j; i++)
            y[j] = y[j] + x[i+j]*x[i];
    }
}

Assumption  •  N_y<=N_x

Techniques  •  Use of CoMUL instructions
Auto_raw

Register Usage

- From .c file to .asm file:
  R12 points to the input vector.
  R13 points to the output vector.
  (R14) = N_x
  (R15) = N_y
### Auto_raw

**Raw autocorrelation (cont’d)**

#### Memory Note

![Diagram](image.png)

**Figure 4-37 Raw autocorrelation**

**Example**

`C166Lib\Examples\Static_funcs\Auto_correlation\Auto.c`
Auto_raw  Raw autocorrelation (cont’d)

Cycle Count

<table>
<thead>
<tr>
<th>Operation</th>
<th>Cycle Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Store state</td>
<td>5</td>
</tr>
<tr>
<td>Memory Initialization</td>
<td>4</td>
</tr>
<tr>
<td>Loop_cyc</td>
<td>Ny – 1 [2 \cdot \sum_{j=0}^{Ny-1} (Nx - j + 8)]</td>
</tr>
<tr>
<td>Restore state</td>
<td>5</td>
</tr>
<tr>
<td>Return</td>
<td>1</td>
</tr>
</tbody>
</table>

**Total** 15 + loop_cyc

Example: 
Nx = 10, Ny = 3 
cycle = 117

Code Size

<table>
<thead>
<tr>
<th>Operation</th>
<th>Code Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Store state</td>
<td>10 bytes</td>
</tr>
<tr>
<td>Memory Initialization</td>
<td>8 bytes</td>
</tr>
<tr>
<td>Loop_cyc</td>
<td>46 bytes</td>
</tr>
<tr>
<td>Restore state</td>
<td>10 bytes</td>
</tr>
<tr>
<td>Return</td>
<td>2 bytes</td>
</tr>
</tbody>
</table>

**Total** 76 bytes
DSP Library for
XC16x Family

Function Descriptions

Auto_bias

Biased autocorrelation

Signature

void Auto_bias( DataS* x,
    DataS* y,
    DataS N_x,
    DataS N_y
  )

Inputs

x : Pointer to input vector in 1Q15 format
N_x : The length of input vector
N_y (Nr) : The length of output vector

Output

y : Pointer to output vector in 1Q15 format

Return

Implementation Description

The function performs the positive side of the biased autocorrelation function of real vector x according to the definition in Equation [4.60]. The arguments to the function are pointer to the input vector, pointer to output buffer to store autocorrelation result, size of input buffer and number of autocorrelated outputs desired. The input values and output values are both in 16 bit fractional format.

Pseudo code

{
  DataS *x;       //Ptr to input vector
  DataS *y;       //Ptr to output vector
  DataS N_x;      //size of input
  DataS N_y;      //size of autocorrelation result
  DataS i,j;

  for(j=0; j<N_y; j++)
  {
    y[j] = 0;
    for(i=0; i<N_x-j; i++)
      y[j] = (y[j] + x[i+j]*x[i])/N_x;
  }
}

Assumption

- N_y<=N_x

Techniques

- Use of CoMUL instructions
Register Usage

- From .c file to .asm file:
  - R12 points to the input vector.
  - R13 points to the output vector.
  - (R14) = N_x
  - (R15) = N_y
**Auto_bias**

**Biased autocorrelation (cont’d)**

**Memory Note**

![Diagram of biased autocorrelation](image)

**Figure 4-38 Biased autocorrelation**

**Example**

`C166Lib\Examples\Static_funcs\Auto_correlation\Auto.c`
Auto_bias

Biased autocorrelation (cont’d)

Cycle Count

<table>
<thead>
<tr>
<th>Operation</th>
<th>Cycle Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Store state</td>
<td>5</td>
</tr>
<tr>
<td>Memory Initialization</td>
<td>4</td>
</tr>
<tr>
<td>Loop_cyc</td>
<td>Ny – 1 (\sum_{j = 0}^{2N - 1} (Nx - j + 17))</td>
</tr>
<tr>
<td>Restore state</td>
<td>5</td>
</tr>
<tr>
<td>Return</td>
<td>1</td>
</tr>
</tbody>
</table>

Total \(15 + \text{loop}_\text{cyc}\)

Example:
\(Nx = 10, Ny = 3\)
\(\text{cycle} = 171\)

Code Size

<table>
<thead>
<tr>
<th>Operation</th>
<th>Code Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Store state</td>
<td>10 bytes</td>
</tr>
<tr>
<td>Memory Initialization</td>
<td>8 bytes</td>
</tr>
<tr>
<td>Loop_cyc</td>
<td>54 bytes</td>
</tr>
<tr>
<td>Restore state</td>
<td>10 bytes</td>
</tr>
<tr>
<td>Return</td>
<td>2 bytes</td>
</tr>
</tbody>
</table>

Total \(84\) bytes
Auto_unbias | Unbiased autocorrelation

**Signature**

```c
void Auto_unbias(   DataS*       x,
                    DataS*       y,
                    DataS        N_x,
                    DataS        N_y
                );
```

**Inputs**

- **x**: Pointer to input vector in 1Q15 format
- **N_x**: The length of input vector
- **N_y (Nr)**: The length of output vector

**Output**

- **y**: Pointer to output vector in 1Q15 format

**Return**


**Implementation Description**

The function performs the positive side of the unbiased autocorrelation function of real vector `x` according to the definition in Equation [4.61]. The arguments to the function are pointer to the input vector, pointer to output buffer to store autocorrelation result, size of input buffer and number of auto correlated outputs desired. The input values and output values are both in 16 bit fractional format.

**Pseudo code**

```c
{ 
    DataS *x;        //Ptr to input vector
    DataS *y;        //Ptr to output vector
    DataS N_x;       //size of input
    DataS N_y;       //size of autocorrelation result
    DataS i,j;

    for(j=0; j<N_y; j++)
    {
        y[j] = 0;
        for(i=0; i<N_x-j; i++)
            y[j] = (y[j] + x[i+j]*x[i])/(N_x-j);
    }
}
```

**Assumption**

- **N_y<=N_x**

**Techniques**

- Use of CoMUL instructions
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Auto_unbias</td>
<td><strong>Unbiased autocorrelation</strong> (cont’d)</td>
</tr>
</tbody>
</table>

**Register Usage**

- From .c file to .asm file:
  - R12 points to the input vector.
  - R13 points to the output vector.
  - \((R14) = N_x\)
  - \((R15) = N_y\)
**Auto_unbias**  

**Unbiased autocorrelation (cont’d)**

**Memory Note**

![Diagram of memory layout and data flow](image)

**Figure 4-39 Unbiased autocorrelation**

**Example**  

*C166Lib\Examples\Static_funcs\Auto_correlation\Auto.c*
**Auto_unbias**

**Unbiased autocorrelation (cont’d)**

**Cycle Count**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Cycle Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Store state</td>
<td>6</td>
</tr>
<tr>
<td>Memory</td>
<td>4</td>
</tr>
<tr>
<td>Initialization</td>
<td></td>
</tr>
<tr>
<td>Loop_cyc</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Ny – 1</td>
</tr>
<tr>
<td></td>
<td>2 \cdot \sum_{j=0}^{N_y} (N_x – j + 18)</td>
</tr>
<tr>
<td>Restore state</td>
<td>6</td>
</tr>
<tr>
<td>Return</td>
<td>1</td>
</tr>
</tbody>
</table>

**Total** \(17 + \text{loop}_\text{cyc}\)

Example:

\(N_x = 10, N_y = 3\)

\(\text{cycle} = 179\)

**Code Size**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Code Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Store state</td>
<td>12 bytes</td>
</tr>
<tr>
<td>Memory</td>
<td>8 bytes</td>
</tr>
<tr>
<td>Initialization</td>
<td></td>
</tr>
<tr>
<td>Loop_cyc</td>
<td>56 bytes</td>
</tr>
<tr>
<td>Restore state</td>
<td>12 bytes</td>
</tr>
<tr>
<td>Return</td>
<td>2 bytes</td>
</tr>
</tbody>
</table>

**Total** 90 bytes
### Cross_raw

**Raw cross-correlation**

**Signature**

```c
void Cross_raw(
  DataTypes* x1,
  DataTypes* x2,
  DataTypes* y,
  DataTypes N_x1,
  DataTypes N_x2
);
```

**Inputs**

- **x1**: Pointer to first input vector in 1Q15 format
- **x2**: Pointer to second input vector in 1Q15 format
- **N_x1 (N1)**: The length of the first input vector
- **N_x2 (N2)**: The length of the second input vector

**Output**

- **y**: Pointer to output vector in 1Q15 format

**Implementation Description**

The function performs the full-length raw cross-correlation function of real vector x1 and x2 according to *Equation [4.62]*, *Equation [4.63]* and *Equation [4.64]*. The arguments to the function are pointers to the input vectors, pointer to output buffer to store autocorrelation result and sizes of input buffers. The input values and output values are both in 16 bit fractional format.
Cross_raw       Raw cross-correlation (cont’d)

Pseudo code

{  
    DataS *x1;       //Ptr to the first input vector  
    DataS *x2;       //Ptr to the second input vector  
    DataS *y;        //Ptr to output vector  
    DataS N1;       //size of the first input  
    DataS N2;       //size of the second input  
    DataS i,j;

    //negative side
    for(j=0; j<N2; j++)
    {
        y[j] = 0;
        for(i=0; i<j; i++)
            y[j] = y[j] + x1[i]*x2[N2-j-1+i];
    }

    //positive side
    if(0<=j<=(N1-N2-1))
    {
        y[j+N2] = 0;
        for(i=0; i<N2; i++)
            y[j+N2] = y[j+N2] + x1[i+j]*x2[i];
    }
    else if((N1-N2)<=j<=(N1-1))
    {
        y[j+N2] = 0;
        for(i=0; i<N1-j; i++)
            y[j+N2] = y[j+N2] + x1[i]*x2[N2-j-1+i];
    }
}

Assumption

- N2<=N1
- x1 must be stored in DPRAM area.

Techniques

- Use of CoMUL instructions
Cross_raw Raw cross-correlation (cont’d)

Register Usage

- From .c file to .asm file:
  - R12 points to x1.
  - R13 points to x2
  - R14 points to the output vector.
  - (R15) = N1
  - ((R0)) = N2
Cross_raw

**Raw cross-correlation (cont’d)**

**Memory Note**

```
Input vector x1
High Addr.
x1(N1-1)
. 
x1(N1-2)
. 
. 
. 
x1(2)
x1(1)x1(0)

Low Addr.
DPRAM

Input vector x2
High Addr.
x2(N2-1)
. 
x2(N2-2)
. 
. 
. 
x2(2)x2(1)x2(0)

Low Addr.
CoMAC

Memory

Output y
High Addr.
y(N-1)
. 
y(N-2)
. 
. 
y(2)
y(1)y(0)

Low Addr.

Memory

DPRAM

R13

IDX0

R14

Figure 4-40 Raw cross-correlation

**Example**

`C166Lib\Examples\Static_funcs\Cross_correlation\Cross.c`

User's Manual for Tasking Compiler 4-150 V 1.1, 2003-06
Cross_raw  Raw cross-correlation (cont'd)

Cycle Count

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Store state</td>
<td>7</td>
</tr>
<tr>
<td>Memory</td>
<td>5</td>
</tr>
<tr>
<td>Neg_cyc</td>
<td></td>
</tr>
<tr>
<td>[ N2-1 \sum_{j=0} (j+18) ]</td>
<td></td>
</tr>
<tr>
<td>Positive-side:</td>
<td></td>
</tr>
<tr>
<td>Initialzation</td>
<td>3</td>
</tr>
<tr>
<td>Pos_cyc1</td>
<td>(N1-N2)*(N2+19)</td>
</tr>
<tr>
<td>Pos_cyc2</td>
<td>[ N2-1 \sum_{j=0} (N1-j+19) ]</td>
</tr>
<tr>
<td>Restore state</td>
<td>7</td>
</tr>
<tr>
<td>Return</td>
<td>1</td>
</tr>
</tbody>
</table>

Total 23 + Neg_cyc + Pos_cyc1 + Pos_cyc2

Example:
N1 = 3, N2 = 3
cycle = 143

Code Size

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Store state</td>
<td>14  bytes</td>
</tr>
<tr>
<td>Memory</td>
<td>10  bytes</td>
</tr>
<tr>
<td>Neg_cyc</td>
<td>50  bytes</td>
</tr>
<tr>
<td>Positive-side:</td>
<td></td>
</tr>
<tr>
<td>Initialzation</td>
<td>6   bytes</td>
</tr>
</tbody>
</table>
### Cross_raw  Raw cross-correlation (cont’d)

<table>
<thead>
<tr>
<th>Function</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pos_cyc1</td>
<td>36 bytes</td>
</tr>
<tr>
<td>Pos_cyc2</td>
<td>44 bytes</td>
</tr>
<tr>
<td>Restore state</td>
<td>14 bytes</td>
</tr>
<tr>
<td>Return</td>
<td>2 bytes</td>
</tr>
</tbody>
</table>

**Total** 176 bytes
## Cross_bias

### Biased cross-correlation

**Signature**

```c
void Cross_bias(     DataS*       x1,
                     DataS*       x2,
                     DataS*       y,
                     DataS        N_x1,
                     DataS        N_x2
)
```

**Inputs**

- **x1**: Pointer to first input vector in 1Q15 format
- **x2**: Pointer to second input vector in 1Q15 format
- **N_x1 (N1)**: The length of the first input vector
- **N_x2 (N2)**: The length of the second input vector

**Output**

- **y**: Pointer to output vector in 1Q15 format

**Return**

- 

**Implementation Description**

The function performs the full-length biased cross-correlation function of real vector `x1` and `x2` according to Equation [4.65], Equation [4.66] and Equation [4.67]. The arguments to the function are pointers to the input vectors, pointer to output buffer to store autocorrelation result and sizes of input buffers. The input values and output values are both in 16 bit fractional format.
Cross_bias  Biased cross-correlation (cont’d)

Pseudo code
{
DataS *x1;       //Ptr to the first input vector
DataS *x2;       //Ptr to the second input vector
DataS *y;        //Ptr to output vector
DataS N1;        //size of the first input
DataS N2;        //size of the second input
DataS i,j;

//negative side
for(j=0; j<N2; j++)
{
    y[j] = 0;
    for(i=0; i<j; i++)
        y[j] = (y[j] + x1[i]*x2[N2-j-1+i])/N1;
}

//positive side
if(0<=j<=(N1-N2-1))
{
    y[j+N2] = 0;
    for(i=0; i<N2; i++)
        y[j+N2] = (y[j+N2] + x1[i+j]*x2[i])/N1;
}
else if((N1-N2)<=j<=(N1-1))
{
    y[j+N2] = 0;
    for(i=0; i<N1-j; i++)
        y[j+N2] = (y[j+N2] + x1[i]*x2[N2-j-1+i])/N1;
}
}

Assumption
• N2<=N1
• x1 must be stored in DPRAM area.

Techniques
• Use of CoMUL instructions
Cross_bias  Biased cross-correlation (cont’d)

Register Usage

• From .c file to .asm file:
  R12 points to x1.
  R13 points to x2
  R14 points to the output vector.
  (R15) = N1
  ((R0)) = N2
Cross_bias  Biased cross-correlation (cont’d)

Memory Note

Figure 4-41 Biased cross-correlation

Example  
C166Lib\Examples\Static funcs\ Cross_correlation\Cross.c
Cross_bias

Biased cross-correlation (cont’d)

Cycle Count

<table>
<thead>
<tr>
<th>Description</th>
<th>Cycle Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Store state</td>
<td>7</td>
</tr>
<tr>
<td>Memory Initialization</td>
<td>5</td>
</tr>
<tr>
<td>Neg_cyc</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$\sum_{j=0}^{N2-1} (j + 36)$</td>
</tr>
<tr>
<td>Positive-side:</td>
<td></td>
</tr>
<tr>
<td>Initialization</td>
<td>3</td>
</tr>
<tr>
<td>Pos_cyc1</td>
<td>(N1-N2)*(N2+37)</td>
</tr>
<tr>
<td>Pos_cyc2</td>
<td>$\sum_{j=0}^{N2-1} (N1 - j + 34)$</td>
</tr>
<tr>
<td>Restore state</td>
<td>7</td>
</tr>
<tr>
<td>Return</td>
<td>1</td>
</tr>
</tbody>
</table>

Total  $23 + \text{Neg}_\text{cyc} + \text{Pos}_\text{cyc1} + \text{Pos}_\text{cyc2}$

Example:
N1 = 3, N2 = 3
cycle = 171

Code Size

<table>
<thead>
<tr>
<th>Description</th>
<th>Code Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Store state</td>
<td>14 bytes</td>
</tr>
<tr>
<td>Memory Initialization</td>
<td>10 bytes</td>
</tr>
<tr>
<td>Neg_cyc</td>
<td>56 bytes</td>
</tr>
<tr>
<td>Positive-side:</td>
<td></td>
</tr>
<tr>
<td>Initialization</td>
<td>6 bytes</td>
</tr>
</tbody>
</table>

Cross_bias               Biased cross-correlation (cont’d)

j3 6
(+)

j0

= N2 1 – \sum

j = 0

N1 j – 34

= N2 – \sum

j = 0

(N1 – j + 34)

Cross_bias               Biased cross-correlation (cont’d)
<table>
<thead>
<tr>
<th>Cross_bias</th>
<th>Biased cross-correlation (cont'd)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pos_cyc1</td>
<td>32 bytes</td>
</tr>
<tr>
<td>Pos_cyc2</td>
<td>52 bytes</td>
</tr>
<tr>
<td>Restore state</td>
<td>14 bytes</td>
</tr>
<tr>
<td>Return</td>
<td>2 bytes</td>
</tr>
</tbody>
</table>

Total 186 bytes
Cross_unbias | Unbiased cross-correlation

Signature

```c
void Cross_unbias( DataS* x1,
                   DataS* x2,
                   DataS* y,
                   DataS N_x1,
                   DataS N_x2)
```

Inputs

- **x1**: Pointer to first input vector in 1Q15 format
- **x2**: Pointer to second input vector in 1Q15 format
- **N_x1 (N1)**: The length of the first input vector
- **N_x2 (N2)**: The length of the second input vector

Output

- **y**: Pointer to output vector in 1Q15 format

Return

Implementation Description

The function performs the full-length unbiased cross-correlation function of real vector x1 and x2 according to \textbf{Equation [4.68]}, \textbf{Equation [4.69]} and \textbf{Equation [4.70]}. The arguments to the function are pointers to the input vectors, pointer to output buffer to store autocorrelation result and sizes of input buffers. The input values and output values are both in 16 bit fractional format.
Cross_unbias

Unbiased cross-correlation (cont’d)

Pseudo code

```c
{ 
    DataS *x1;   //Ptr to the first input vector
    DataS *x2;   //Ptr to the second input vector
    DataS *y;    //Ptr to output vector
    DataS N1;    //size of the first input
    DataS N2;    //size of the second input
    DataS i,j;

    //negative side
    for(j=0; j<N2; j++)
    {
        y[j] = 0;
        for(i=0; i<j; i++)
            y[j] = (y[j] + x1[i]*x2[N2-j-1+i])/abs(j+1);
    }

    //positive side
    if(0<=j<=(N1-N2-1))
    {
        y[j+N2] = 0;
        for(i=0; i<N2; i++)
            y[j+N2] = (y[j+N2] + x1[i+j]*x2[i])/N2;
    }
    else if((N1-N2)<=j<=(N1-1))
    {
        y[j+N2] = 0;
        for(i=0; i<N1-j; i++)
            y[j+N2] = (y[j+N2] + x1[i]*x2[N2-j-1+i])/(N1-abs(j));
    }
}
```

Assumption
- N2<=N1
- x1 must be stored in DPRAM area.

Techniques
- Use of CoMUL instructions
Cross_unbias

Unbiased cross-correlation (cont’d)

Register Usage

- From .c file to .asm file:
  - R12 points to x1.
  - R13 points to x2
  - R14 points to the output vector.
  - (R15) = N1
  - ((R0)) = N2
Cross_unbias Unbiased cross-correlation (cont’d)

Memory Note

<table>
<thead>
<tr>
<th>Input vector x1</th>
<th>input vector x2</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x1(N1-1)</code></td>
<td><code>x2(N2-1)</code></td>
</tr>
<tr>
<td><code>x1(N1-2)</code></td>
<td><code>x2(N2-2)</code></td>
</tr>
<tr>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td><code>x1(2)</code></td>
<td><code>x2(2)</code></td>
</tr>
<tr>
<td><code>x1(1)</code></td>
<td><code>x2(1)</code></td>
</tr>
<tr>
<td><code>x1(0)</code></td>
<td><code>x2(0)</code></td>
</tr>
</tbody>
</table>

CoMAC

<table>
<thead>
<tr>
<th>Low Addr.</th>
<th>High Addr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>DPRAM</td>
<td>IDX0</td>
</tr>
<tr>
<td>Low Addr.</td>
<td>High Addr.</td>
</tr>
<tr>
<td>Memory</td>
<td>CoMAC</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Low Addr.</th>
<th>High Addr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td>R13</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Low Addr.</th>
<th>High Addr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td>R14</td>
</tr>
</tbody>
</table>

Figure 4-42 Unbiased cross-correlation

Example

`C166Lib\Examples\Static_funcs\Cross_correlation\Cross.c`
Cross_unbias

Unbiased cross-correlation (cont’d)

Cycle Count

<table>
<thead>
<tr>
<th>Operation</th>
<th>Cycle Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Store state</td>
<td>7</td>
</tr>
<tr>
<td>Memory</td>
<td>5</td>
</tr>
<tr>
<td>Initialization</td>
<td></td>
</tr>
<tr>
<td>Neg_cyc</td>
<td></td>
</tr>
</tbody>
</table>

\[
\text{Neg}_\text{cyc} = \sum_{j=0}^{N2-1} (j + 38)
\]

 Positive-side:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Cycle Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialization</td>
<td>3</td>
</tr>
<tr>
<td>Pos_cyc1</td>
<td>(N1-N2)*(N2+39)</td>
</tr>
<tr>
<td>Pos_cyc2</td>
<td></td>
</tr>
</tbody>
</table>

\[
\text{Pos}_\text{cyc}2 = \sum_{j=0}^{N2-1} (N1 - j + 41)
\]

<table>
<thead>
<tr>
<th>Operation</th>
<th>Cycle Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Restore state</td>
<td>7</td>
</tr>
<tr>
<td>Return</td>
<td>1</td>
</tr>
</tbody>
</table>

Total

\[23 + \text{Neg}_\text{cyc} + \text{Pos}_\text{cyc}1 + \text{Pos}_\text{cyc}2\]

Example:

\[N1 = 3, N2 = 3\]

\[\text{cycle} = 227\]

Code Size

<table>
<thead>
<tr>
<th>Operation</th>
<th>Code Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Store state</td>
<td>14 bytes</td>
</tr>
<tr>
<td>Memory</td>
<td>10 bytes</td>
</tr>
<tr>
<td>Initialization</td>
<td></td>
</tr>
<tr>
<td>Neg_cyc</td>
<td>60 bytes</td>
</tr>
</tbody>
</table>

 Positive-side:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Code Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialization</td>
<td>6 bytes</td>
</tr>
<tr>
<td>Cross_unbias</td>
<td>Unbiased cross-correlation (cont’d)</td>
</tr>
<tr>
<td>----------------------------</td>
<td>-----------------------------------</td>
</tr>
<tr>
<td>Pos_cyc1</td>
<td>36 bytes</td>
</tr>
<tr>
<td>Pos_cyc2</td>
<td>64 bytes</td>
</tr>
<tr>
<td>Restore state</td>
<td>14 bytes</td>
</tr>
<tr>
<td>Return</td>
<td>2 bytes</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>206 bytes</strong></td>
</tr>
</tbody>
</table>
5 Integrated Test Process

5.1 Structure of The Test Process

The reason for building an integrated test process is to make the verification of library functions for the new tool chains easier. For example, if we want to test the library functions with the new release of Tasking tool chain, we need to integrate all source files of the test process included test vectors into one project and build it, then run it to test all library functions throughout. Otherwise we must test each function separately, which requires a great deal of time.

The provided test process consists of source files and test vectors of different functions. The source files are written in C. The DSP library functions have been compiled and linked as a lib-file with the name “dsplib.lib” along with the other source files. Table 5-1 gives the directory structure of the implemented integrated test process and Table 5-1 lists the names of source files used in the test process.

<table>
<thead>
<tr>
<th>Directory name</th>
<th>Contents</th>
<th>Files</th>
</tr>
</thead>
<tbody>
<tr>
<td>C166Lib/Test</td>
<td>Directories which have all the files related to the integrated test process</td>
<td>None</td>
</tr>
<tr>
<td>C166Lib/Test/Source</td>
<td>Source files including C implementations and header files</td>
<td>*.c</td>
</tr>
<tr>
<td></td>
<td></td>
<td>*.h</td>
</tr>
<tr>
<td></td>
<td></td>
<td>dsplib.lib</td>
</tr>
<tr>
<td>C166Lib/Test/Testvector</td>
<td>Directories of test vectors. Each directory contains respective test vector in ASCII and binary format. The text file “FunctionList.txt” lists the names of the test vectors used in the test process</td>
<td>FunctionList.txt</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>File name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>main.c</td>
<td>Main function</td>
</tr>
<tr>
<td>DataIO.c</td>
<td>Subfunction for data I/O</td>
</tr>
<tr>
<td>Output.c</td>
<td>Subfunction for output</td>
</tr>
<tr>
<td>Test_all.c</td>
<td>Subfunction for implementation of the integrated test</td>
</tr>
<tr>
<td>Test_each.c</td>
<td>Subfunction for implementation of the separated test</td>
</tr>
</tbody>
</table>
5.2 Building The Integrated Test Process

Do the following steps to build the test process using Tasking tool chains:

1. Choose small memory model for C compiler
2. Include all files under the directory C166Lib/Test/Source/ into the project, including C files, header files and lib-file
3. Set the Project Options-CPU to select an MCU with XC16x microcontroller
4. Define Heap size under the Project Options-Linker/Locator-Specials
5. Build the system

After building the project correctly, the test process can be run with simulator or hardware. At beginning of the test the directory that the test vectors are stored in must be entered. If the test vectors have been already stored in the default directory C166Lib/Test/Testvector/, press “Enter”. After that the choice of the test approach determines how to test the DSP library. With choice of number 1, the functions will be tested integrated and the test process runs until all DSP functions are tested. With the choice 2 we have possibility to test a single function in the library. If number 2 is chosen, the terminal window shows a list of the functions. In this case the further choice of which function should be tested is required. Type in the number corresponding to the function, this function is then tested.

Table 5-2 List of source files of the integrated test process

<table>
<thead>
<tr>
<th>File name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test_Adap_filter.c</td>
<td>Subfunction for testing adaptive filters</td>
</tr>
<tr>
<td>Test_Arith.c</td>
<td>Subfunction for testing arithmetical functions</td>
</tr>
<tr>
<td>Test_FFT.c</td>
<td>Subfunction for testing FFT transform</td>
</tr>
<tr>
<td>Test_Fir.c</td>
<td>Subfunction for testing Fir filters</td>
</tr>
<tr>
<td>Test_IIR.c</td>
<td>Subfunction for testing IIR filters</td>
</tr>
<tr>
<td>Test_math.c</td>
<td>Subfunction for testing mathematical functions</td>
</tr>
<tr>
<td>Test_Matrix.c</td>
<td>Subfunction for testing matrix operations</td>
</tr>
<tr>
<td>Test_Static.c</td>
<td>Subfunction for testing statistical functions</td>
</tr>
<tr>
<td>DataIO.h</td>
<td>Header file</td>
</tr>
<tr>
<td>DspLib.h</td>
<td>Header file</td>
</tr>
<tr>
<td>dsplib.lib</td>
<td>Library file including all DSP functions in C166Lib</td>
</tr>
</tbody>
</table>
6 References

Total Quality Management

Qualität hat für uns eine umfassende Bedeutung. Wir wollen allen Ihren Ansprüchen in der bestmöglichen Weise gerecht werden. Es geht uns also nicht nur um die Produktqualität – unsere Anstrengungen gelten gleichermaßen der Lieferqualität und Logistik, dem Service und Support sowie allen sonstigen Beratungs- und Betreuungsleistungen.


Unternehmensweit orientieren wir uns dabei auch an „top“ (Time Optimized Processes), um Ihnen durch größere Schnelligkeit den entscheidenden Wettbewerbsvorsprung zu verschaffen.

Geben Sie uns die Chance, hohe Leistung durch umfassende Qualität zu beweisen.
Wir werden Sie überzeugen.

http://www.infineon.com

Published by Infineon Technologies AG