Application Note: AN00209

xCORE-200 DSP Elements Library

The application note gives an overview of using the xCORE-200 DSP Elements Library.

Required tools and libraries

- xTIMEcomposer Tools - Version 14.1.2 and above
- XMOS DSP library module - Version 2.0.0 and above

Required hardware

This application note is designed to run on any XMOS xCORE-200 multicore microcontroller or the XMOS simulator.

Prerequisites

- This document assumes familiarity with the XMOS xCORE architecture, xCONNECT interconnect communication, the XMOS tool chain and the xC language. Documentation related to these aspects are linked to in the appendix §A.
- For descriptions of XMOS related terms found in this document please see the XMOS Glossary[1].

[1] http://www.xmos.com/published/glossary

1 Introduction

The XMOS xCORE-200 DSP Elements Library provides foundation Digital Signal Processing functions.

The library is split into the following modules :

- Adaptive Filters - lib_dsp_adaptive.h
- Filtering - lib_dsp_filters.h
- Basic Math - lib_dsp_math.h
- Matrix Math - lib_dsp_matrix.h
- Statistics - lib_dsp_statistics.h
- Vectors - lib_dsp_vector.h
- FFT and inverse FFT - lib_dsp_fft.h

The library supports variable Q formats from Q8 to Q31, using the macros in the following header file : lib_dsp_qformat.h.

2 Getting Started

Ensure you are in the xTIMEcomposer edit perspective by clicking on the Edit perspective button on the left hand side toolbar.

In the Project Explorer view you will see the following applications :

- Adaptive Filters - app_adaptive
- Filtering - app_filters
- Basic Math - app_math
- Matrix Math - app_matrix
- Statistics - app_statistics
- Vectors - app_vector
- FFT and inverse FFT - app_fft
- FFT Processing of signals received through a double buffer - app_fft_double_buf

The applications contain code to generate the simulation data and call all of the functions in each module and print the results in the xTIMEcomposer console.

2.1 Build the applications

To build the application, select 'Project -> Build Project' in the menu, or click the 'Build' button on the toolbar. The output from the compilation process will be visible on the console. Note that some applications offer different configurations. Particular Build Configurations can be selected by clicking the small arrow next to the Build Icon (Hammer).

2.2 Create and launch a run configuration

To run or debug the application, you have to initially create a run or debug configuration. The xTIMEcomposer allows multiple configurations to exist, thus allowing you to store configurations for running on different targets/with different runtime options and arguments. Right-click on the generated binary in the *Project Explorer* view, and select *Run As -> Run Configurations*. To debug, select *Run As -> Debug Configurations*.

In the resulting dialog, double click on *xCORE Application*, then perform the following operations:

- On the *Main* tab select the desired Hardware Target, or check the *simulator* option.
- Select the Run button to launch the application.

The results will be displayed in the xTIMEcomposer *console* tab.

3 Using The DSP Library In Other Applications

3.1 Makefile Additions For These Examples

To start using the DSP Library, you need to add lib_dsp to your Makefile:

```
USED_MODULES = ... lib_dsp ...
```

3.2 Including The Library Into Your Source Code

In order to use any of these modules and the Q formats it is only necessary to include the following header file:

```
#include "lib_dsp.h"
```

APPENDIX A - References

XMOS Tools User Guide

http://www.xmos.com/published/xtimecomposer-user-guide

XMOS xCORE Programming Guide

http://www.xmos.com/published/xmos-programming-guide

XMOS DSP Library

http://www.xmos.com/support/libraries/lib_dsp

xCORE-200: The XMOS XS2 Architecture (ISA)

https://www.xmos.com/published/xs2-isa-specification

APPENDIX B - Full Source Code Listings

This section includes the source code for all of the example programs.

B.1 Adaptive Filtering Functions

```
// Copyright (c) 2016, XMOS Ltd, All rights reserved
// XMOS DSP Library - Adaptive Filtering Function Test Program
// Uses Q24 format

// Include files
#include <stdio.h>
#include <xs1.h>
#include <print.h>
#include <lib_dsp.h>

// Define constants

#define Q_M               1
#define Q_N               31

#define FIR_FILTER_LENGTH     160

void print31( int32_t x ) {if(x >=0) printf("+%f ",F31(x)); else printf("%f ",F31(x));}


// Declare global variables and arrays
int32_t fir_coeffs[] = // 161 taps
{
  Q31(+0.0391607),Q31(+0.0783215),Q31(+0.0191607),Q31(+0.01531791),Q31(-0.03098222),
  Q31(+0.0391607),Q31(+0.0783215),Q31(+0.0191607),Q31(+0.01531791),Q31(-0.03098222),
  Q31(+0.0391607),Q31(+0.0783215),Q31(+0.0191607),Q31(+0.01531791),Q31(-0.03098222),
  Q31(+0.0391607),Q31(+0.0783215),Q31(+0.0191607),Q31(+0.01531791),Q31(-0.03098222),
  Q31(+0.0391607),Q31(+0.0783215),Q31(+0.0191607),Q31(+0.01531791),Q31(-0.03098222),
  Q31(+0.0391607),Q31(+0.0783215),Q31(+0.0191607),Q31(+0.01531791),Q31(-0.03098222),
  Q31(+0.0391607),Q31(+0.0783215),Q31(+0.0191607),Q31(+0.01531791),Q31(-0.03098222),
  Q31(+0.0391607),Q31(+0.0783215),Q31(+0.0191607),Q31(+0.01531791),Q31(-0.03098222),
  Q31(+0.0391607),Q31(+0.0783215),Q31(+0.0191607),Q31(+0.01531791),Q31(-0.03098222),
  Q31(+0.0391607),Q31(+0.0783215),Q31(+0.0191607),Q31(+0.01531791),Q31(-0.03098222),
  Q31(+0.0391607),Q31(+0.0783215),Q31(+0.0191607),Q31(+0.01531791),Q31(-0.03098222),
  Q31(+0.0391607),Q31(+0.0783215),Q31(+0.0191607),Q31(+0.01531791),Q31(-0.03098222),
  Q31(+0.0391607),Q31(+0.0783215),Q31(+0.0191607),Q31(+0.01531791),Q31(-0.03098222),
  Q31(+0.0391607),Q31(+0.0783215),Q31(+0.0191607),Q31(+0.01531791),Q31(-0.03098222),
  Q31(+0.0391607),Q31(+0.0783215),Q31(+0.0191607),Q31(+0.01531791),Q31(-0.03098222),
  Q31(+0.0391607),Q31(+0.0783215),Q31(+0.0191607),Q31(+0.01531791),Q31(-0.03098222),
  Q31(+0.0391607),Q31(+0.0783215),Q31(+0.0191607),Q31(+0.01531791),Q31(-0.03098222),
  Q31(+0.0391607),Q31(+0.0783215),Q31(+0.0191607),Q31(+0.01531791),Q31(-0.03098222),
  Q31(+0.0391607),Q31(+0.0783215),Q31(+0.0191607),Q31(+0.01531791),Q31(-0.03098222),
  Q31(+0.0391607),Q31(+0.0783215),Q31(+0.0191607),Q31(+0.01531791),Q31(-0.03098222),
  Q31(+0.0391607),Q31(+0.0783215),Q31(+0.0191607),Q31(+0.01531791),Q31(-0.03098222),
  Q31(+0.0391607),Q31(+0.0783215),Q31(+0.0191607),Q31(+0.01531791),Q31(-0.03098222),
  Q31(+0.0391607),Q31(+0.0783215),Q31(+0.0191607),Q31(+0.01531791),Q31(-0.03098222),
  Q31(+0.0391607),Q31(+0.0783215),Q31(+0.0191607),Q31(+0.01531791),Q31(-0.03098222),
  Q31(+0.0391607),Q31(+0.0783215),Q31(+0.0191607),Q31(+0.01531791),Q31(-0.03098222),
  Q31(+0.0391607),Q31(+0.0783215),Q31(+0.0191607),Q31(+0.01531791),Q31(-0.03098222),
  Q31(+0.0391607),Q31(+0.0783215),Q31(+0.0191607),Q31(+0.01531791),Q31(-0.03098222),
  Q31(+0.0391607),Q31(+0.0783215),Q31(+0.0191607),Q31(+0.01531791),Q31(-0.03098222),
  Q31(+0.0391607),Q31(+0.0783215),Q31(+0.0191607),Q31(+0.01531791),Q31(-0.03098222),
  Q31(+0.0391607),Q31(+0.0783215),Q31(+0.0191607),Q31(+0.01531791),Q31(-0.03098222),
  Q31(+0.0391607),Q31(+0.0783215),Q31(+0.0191607),Q31(+0.01531791),Q31(-0.03098222),
  Q31(+0.0391607),Q31(+0.0783215),Q31(+0.0191607),Q31(+0.01531791),Q31(-0.03098222),
  Q31(+0.0391607),Q31(+0.0783215),Q31(+0.0191607),Q31(+0.01531791),Q31(-0.03098222),
  Q31(+0.0391607),Q31(+0.0783215),Q31(+0.0191607),Q31(+0.01531791),Q31(-0.03098222),
  Q31(+0.0391607),Q31(+0.0783215),Q31(+0.0191607),Q31(+0.01531791),Q31(-0.03098222),
  Q31(+0.0391607),Q31(+0.0783215),Q31(+0.0191607),Q31(+0.01531791),Q31(-0.03098222),
  Q31(+0.0391607),Q31(+0.0783215),Q31(+0.0191607),Q31(+0.01531791),Q31(-0.03098222),
  Q31(+0.0391607),Q31(+0.0783215),Q31(+0.0191607),Q31(+0.01531791),Q31(-0.03098222),
  Q31(+0.0391607),Q31(+0.0783215),Q31(+0.0191607),Q31(+0.01531791),Q31(-0.03098222),
  Q31(+0.0391607),Q31(+0.0783215),Q31(+0.0191607),Q31(+0.01531791),Q31(-0.03098222),
};


int32_t fir_state[FIR_FILTER_LENGTH];
```

```
int32_t lms_coeffs[FIR_FILTER_LENGTH];
int32_t nlms_coeffs[FIR_FILTER_LENGTH];

int main(void)
{
  int32_t c;
  int32_t x;
  int32_t err;

              // Apply LMS Filter
  for( c = 5; c <= 5; c *= 2 )
  {
    printf( "LMS %u\n", c );
    for( int32_t i = 0; i < FIR_FILTER_LENGTH; ++i ) lms_coeffs[i] = fir_coeffs[i];
    for( int32_t i = 0; i < FIR_FILTER_LENGTH; ++i ) fir_state[i] = 0;
    for( int32_t i = 0; i < c+30; ++i )
    {
      x = lib_dsp_adaptive_lms( Q31(0.08), Q31(0.10), &err, lms_coeffs, fir_state, c, Q31(0.01), Q_N );
      print31( x ); print31( err ); printf( "\n" );
    }
  }

              // Apply Normalized LMS Filter
  for( c = 5; c <= 5; c *= 2 )
  {
    printf( "\nNormalized LMS %u\n", c );
    for( int32_t i = 0; i < FIR_FILTER_LENGTH; ++i ) nlms_coeffs[i] = fir_coeffs[i];
    for( int32_t i = 0; i < FIR_FILTER_LENGTH; ++i ) fir_state[i] = 0;
    for( int32_t i = 0; i < c+30; ++i )
    {
      x = lib_dsp_adaptive_nlms( Q31(0.08), Q31(0.10), &err, nlms_coeffs, fir_state, c, Q31(0.01), Q_N );
      print31( x ); print31( err ); printf( "\n" );
    }
  }

  return (0);
}
```

B.2 Fixed Coefficient Filtering Functions

```
// Copyright (c) 2016, XMOS Ltd, All rights reserved
// XMOS DSP Library - Filtering Functions Test Program
// Uses Q24 format

// Include files
#include <stdio.h>
#include <xs1.h>
#include <lib_dsp.h>

// Define constants

#define Q_M               8
#define Q_N               24

#define FIR_FILTER_LENGTH     30
#define IIR_STATE_LENGTH      4
#define IIR_CASCADE_DEPTH     3
#define SAMPLE_LENGTH         50

#define INTERP_FILTER_LENGTH  160

void print31( int32_t x ) {if(x >=0) printf("+%f ",F31(x)); else printf("%f ",F31(x));}

// Declare global variables and arrays
int32_t  Src[] = { Q24(.11), Q24(.12), Q24(.13), Q24(.14), Q24(.15), Q24(.16), Q24(.17), Q24(.18), Q24(.19),
   ↪ Q24(.20),
               Q24(.21), Q24(.22), Q24(.23), Q24(.24), Q24(.25), Q24(.26), Q24(.27), Q24(.28), Q24(.29), Q24
                 ↪ (.30),
               Q24(.31), Q24(.32), Q24(.33), Q24(.34), Q24(.35), Q24(.36), Q24(.37), Q24(.38), Q24(.39), Q24
                 ↪ (.40),
               Q24(.41), Q24(.42), Q24(.43), Q24(.44), Q24(.45), Q24(.46), Q24(.47), Q24(.48), Q24(.49), Q24
                 ↪ (.50),
               Q24(.51), Q24(.52), Q24(.53), Q24(.54), Q24(.55), Q24(.56), Q24(.57), Q24(.58), Q24(.59), Q24
                 ↪ (.60)};

int32_t             Dst[INTERP_FILTER_LENGTH];

int32_t firCoeffs[] = { Q24(.11), Q24(.12), Q24(.13), Q24(.14), Q24(.15), Q24(.16), Q24(.17), Q24(.18), Q24
   ↪ (.19), Q24(.20),
                    Q24(.21), Q24(.22), Q24(.23), Q24(.24), Q24(.25), Q24(.26), Q24(.27), Q24(.28), Q24(.29),
                      ↪ Q24(.30),
                    Q24(.31), Q24(.32), Q24(.33), Q24(.34), Q24(.35), Q24(.36), Q24(.37), Q24(.38), Q24(.39),
                      ↪ Q24(.40)};

int32_t firCoeffsInt[] =
{
    Q31(+0.0391607),Q31(+0.0783215),Q31(+0.0191607),Q31(+0.01531791),Q31(-0.03098222),
    Q31(+0.0391607),Q31(+0.0783215),Q31(+0.0191607),Q31(+0.01531791),Q31(-0.03098222),
    Q31(+0.0391607),Q31(+0.0783215),Q31(+0.0191607),Q31(+0.01531791),Q31(-0.03098222),
    Q31(+0.0391607),Q31(+0.0783215),Q31(+0.0191607),Q31(+0.01531791),Q31(-0.03098222),
    Q31(+0.0391607),Q31(+0.0783215),Q31(+0.0191607),Q31(+0.01531791),Q31(-0.03098222),
    Q31(+0.0391607),Q31(+0.0783215),Q31(+0.0191607),Q31(+0.01531791),Q31(-0.03098222),
    Q31(+0.0391607),Q31(+0.0783215),Q31(+0.0191607),Q31(+0.01531791),Q31(-0.03098222),
    Q31(+0.0391607),Q31(+0.0783215),Q31(+0.0191607),Q31(+0.01531791),Q31(-0.03098222),
    Q31(+0.0391607),Q31(+0.0783215),Q31(+0.0191607),Q31(+0.01531791),Q31(-0.03098222),
    Q31(+0.0391607),Q31(+0.0783215),Q31(+0.0191607),Q31(+0.01531791),Q31(-0.03098222),
    Q31(+0.0391607),Q31(+0.0783215),Q31(+0.0191607),Q31(+0.01531791),Q31(-0.03098222),
    Q31(+0.0391607),Q31(+0.0783215),Q31(+0.0191607),Q31(+0.01531791),Q31(-0.03098222),
    Q31(+0.0391607),Q31(+0.0783215),Q31(+0.0191607),Q31(+0.01531791),Q31(-0.03098222),
    Q31(+0.0391607),Q31(+0.0783215),Q31(+0.0191607),Q31(+0.01531791),Q31(-0.03098222),
    Q31(+0.0391607),Q31(+0.0783215),Q31(+0.0191607),Q31(+0.01531791),Q31(-0.03098222),
    Q31(+0.0391607),Q31(+0.0783215),Q31(+0.0191607),Q31(+0.01531791),Q31(-0.03098222),
    Q31(+0.0391607),Q31(+0.0783215),Q31(+0.0191607),Q31(+0.01531791),Q31(-0.03098222),
    Q31(+0.0391607),Q31(+0.0783215),Q31(+0.0191607),Q31(+0.01531791),Q31(-0.03098222),
    Q31(+0.0391607),Q31(+0.0783215),Q31(+0.0191607),Q31(+0.01531791),Q31(-0.03098222),
    Q31(+0.0391607),Q31(+0.0783215),Q31(+0.0191607),Q31(+0.01531791),Q31(-0.03098222),
    Q31(+0.0391607),Q31(+0.0783215),Q31(+0.0191607),Q31(+0.01531791),Q31(-0.03098222),
    Q31(+0.0391607),Q31(+0.0783215),Q31(+0.0191607),Q31(+0.01531791),Q31(-0.03098222),
    Q31(+0.0391607),Q31(+0.0783215),Q31(+0.0191607),Q31(+0.01531791),Q31(-0.03098222),
    Q31(+0.0391607),Q31(+0.0783215),Q31(+0.0191607),Q31(+0.01531791),Q31(-0.03098222),
    Q31(+0.0391607),Q31(+0.0783215),Q31(+0.0191607),Q31(+0.01531791),Q31(-0.03098222),
    Q31(+0.0391607),Q31(+0.0783215),Q31(+0.0191607),Q31(+0.01531791),Q31(-0.03098222),
    Q31(+0.0391607),Q31(+0.0783215),Q31(+0.0191607),Q31(+0.01531791),Q31(-0.03098222),
    Q31(+0.0391607),Q31(+0.0783215),Q31(+0.0191607),Q31(+0.01531791),Q31(-0.03098222),
    Q31(+0.0391607),Q31(+0.0783215),Q31(+0.0191607),Q31(+0.01531791),Q31(-0.03098222),
    Q31(+0.0391607),Q31(+0.0783215),Q31(+0.0191607),Q31(+0.01531791),Q31(-0.03098222),
    Q31(+0.0391607),Q31(+0.0783215),Q31(+0.0191607),Q31(+0.01531791),Q31(-0.03098222),
    Q31(+0.0391607),Q31(+0.0783215),Q31(+0.0191607),Q31(+0.01531791),Q31(-0.03098222),
    Q31(+0.0391607),Q31(+0.0783215),Q31(+0.0191607),Q31(+0.01531791),Q31(-0.03098222),
    Q31(+0.0391607),Q31(+0.0783215),Q31(+0.0191607),Q31(+0.01531791),Q31(-0.03098222),
    Q31(+0.0391607),Q31(+0.0783215),Q31(+0.0191607),Q31(+0.01531791),Q31(-0.03098222),
    Q31(+0.0391607),Q31(+0.0783215),Q31(+0.0191607),Q31(+0.01531791),Q31(-0.03098222),
    Q31(+0.0391607),Q31(+0.0783215),Q31(+0.0191607),Q31(+0.01531791),Q31(-0.03098222),
    Q31(+0.0391607),Q31(+0.0783215),Q31(+0.0191607),Q31(+0.01531791),Q31(-0.03098222),
    Q31(+0.0391607),Q31(+0.0783215),Q31(+0.0191607),Q31(+0.01531791),Q31(-0.03098222),
    Q31(+0.0391607),Q31(+0.0783215),Q31(+0.0191607),Q31(+0.01531791),Q31(-0.03098222),
};

int32_t iirCoeffs[] = { Q24(.11), Q24(.12), Q24(.13), Q24(.14), Q24(.15),
                    Q24(.21), Q24(.22), Q24(.23), Q24(.24), Q24(.25),
                    Q24(.31), Q24(.32), Q24(.33), Q24(.34), Q24(.35)};

//int filterState[FIR_FILTER_LENGTH];
int32_t filterState[INTERP_FILTER_LENGTH];

int32_t inter_coeff[INTERP_FILTER_LENGTH];
int32_t decim_coeff[INTERP_FILTER_LENGTH];
int32_t decim_input[16];

int main(void)
{
  int32_t i, j, c, r, x, y;


                 // Initiaize FIR filter state array
  for (i = 0; i < FIR_FILTER_LENGTH; i++)
  {
    filterState[i] = 0;
  }

                // Apply FIR filter and store filtered data
  for (i = 0; i < SAMPLE_LENGTH; i++)
  {
    Dst[i] =
      lib_dsp_filters_fir (Src[i],                 // Input data sample to be filtered
                           firCoeffs,              // Pointer to filter coefficients
                           filterState,            // Pointer to filter state array
                           FIR_FILTER_LENGTH,      // Filter length
                           Q_N);                   // Q Format N
  }

  printf ("FIR Filter Results\n");
  for (i = 0; i < SAMPLE_LENGTH; i++)
  {
      printf ("Dst[%d] = %lf\n", i, F24 (Dst[i]));
  }

                 // Initiaize IIR filter state array
  for (i = 0; i < IIR_STATE_LENGTH; i++)
  {
    filterState[i] = 0;
  }

                // Apply IIR filter and store filtered data
  for (i = 0; i < SAMPLE_LENGTH; i++)
  {
    Dst[i] =
      lib_dsp_filters_biquad (Src[i],              // Input data sample to be filtered
                              iirCoeffs,           // Pointer to filter coefficients
                              filterState,         // Pointer to filter state array
                              Q_N);                // Q Format N
  }

  printf ("\nIIR Biquad Filter Results\n");
  for (i = 0; i < SAMPLE_LENGTH; i++)
  {
      printf ("Dst[%d] = %lf\n", i, F24 (Dst[i]));
  }
```

```
               // Initiaize IIR filter state array
  for (i = 0; i < (IIR_CASCADE_DEPTH * IIR_STATE_LENGTH); i++)
  {
    filterState[i] = 0;
  }

              // Apply IIR filter and store filtered data
  for (i = 0; i < SAMPLE_LENGTH; i++)
  {
    Dst[i] =
      lib_dsp_filters_biquads (Src[i],             // Input data sample to be filtered
                               iirCoeffs,          // Pointer to filter coefficients
                               filterState,        // Pointer to filter state array
                               IIR_CASCADE_DEPTH,  // Number of cascaded sections
                               Q_N);               // Q Format N
  }

  printf ("\nCascaded IIR Biquad Filter Results\n");
  for (i = 0; i < SAMPLE_LENGTH; i++)
  {
      printf ("Dst[%d] = %lf\n", i, F24 (Dst[i]));
  }



  printf ("\nInterpolation\n");
  for( r = 2; r <= 8; ++r )
  {
    c = 8;
    printf( "INTERP taps=%u L=%u\n", c*r, r );

    i = 0;
    for( y = 0; y < c; ++y )
      for( x = 0; x < r; ++x )
        inter_coeff[x*c+y] = firCoeffsInt[i++];

    for( i = 0; i < 160; ++i )
      filterState[i] = 0;

    for( i = 0; i < c; ++i )
    {
      lib_dsp_filters_interpolate( Q31(0.1), inter_coeff, filterState, c*r, r, Dst, 31 );
      for( j = 0; j < r; ++j )
        print31( Dst[j] );
      printf( "\n" );
    }
  }


  printf ("\nDecimation\n");
  for( int32_t r = 2; r <= 8; ++r )
  {
    for( i = 0; i < 16; ++i )
      decim_input[i] = Q31(0.1);
    printf( "DECIM taps=%02u M=%02u\n", 32, r );
    for( i = 0; i < 160; ++i )
      filterState[i] = 0;
    for( i = 0; i < 32/r; ++i )
    {
      x = lib_dsp_filters_decimate( decim_input, firCoeffsInt, filterState, 32, r, 31 );
      print31( x );
      if( (i&7) == 7 )
        printf( "\n" );
    }
    if( (--i&7) != 7 )
      printf( "\n" );
  }

  return (0);
}
```

B.3 Math Functions

```
// Copyright (c) 2016, XMOS Ltd, All rights reserved
// XMOS DSP Library - Math Functions Test Program
// Uses Q24 format

#include <stdio.h>
#include <xs1.h>
#include <lib_dsp.h>
#include <math.h>
#include <stdint.h>
#include <stdlib.h>

#define CHECK_RESULTS 1
#define PRINT_ERROR_TOO_BIG 1
#define EXIT_ON_ERROR_TOO_BIG 0
#define TEST_ALL_INPUTS 0
#define NO_Q 0

#if TEST_ALL_INPUTS
#define PRINT_CYCLE_COUNT 0
#define PRINT_INPUTS_AND_OUTPUTS 0
#define RAD_INCR 1
#define X_INCR 1
#define EXPONENTIAL_INPUT 0
#else
#define PRINT_CYCLE_COUNT 0
#define PRINT_INPUTS_AND_OUTPUTS 1
#define RAD_INCR PI2_Q8_24/100
#define X_INCR MAX_Q8_24/100
#define EXPONENTIAL_INPUT 1
#endif

#if PRINT_CYCLE_COUNT
#define DIVIDE_STRESS 1 // execute divide on other cores to get worst case performance.
#else
#define DIVIDE_STRESS 0
#endif

// errors from -3..+3
#define ERROR_RANGE 7
typedef struct {
    int32_t errors[ERROR_RANGE];
    int32_t max_positive_error;
    int32_t max_negative_error;
    int32_t num_checked;
} error_s;

/*
 * Report Errors within the Error Range. Top and Bottom or range contain saturated values
 */
int32_t report_errors(uint32_t max_abs_error, error_s *e) {
    int32_t result = 1; // PASS
    int32_t half_range = ERROR_RANGE/2;
    for(int32_t i=0; i<ERROR_RANGE; i++) {
        int32_t error = -half_range+i;
        if(i == 0) {
           printf("Cases Error <= %d: %d\n",error, e->errors[i]);
        } else if (i == ERROR_RANGE-1) {
           printf("Cases Error >= %d: %d\n",error, e->errors[i]);
        } else {
           printf("Cases Error == %d: %d\n",error, e->errors[i]);
        }
        if (error > max_abs_error || error < -max_abs_error) {
            result = 0;
        }
    }
    printf("Maximum Positive Error: %d\n",e->max_positive_error);
    printf("Maximum Negative Error: %d\n",e->max_negative_error);
    printf("Number of values checked: %d\n", e->num_checked);
    printf("Percentage of 0 Error: %5.2f%%\n", 100.0*e->errors[half_range]/e->num_checked);
    return result;
}
void reset_errors(error_s *e) {
    for(int32_t i=0; i<ERROR_RANGE; i++) {
        e->errors[i] = 0;
    }
    e->max_positive_error = 0;
    e->max_negative_error = 0;
    e->num_checked = 0;
}

/* Function to check a result against a expected result and store error statistics
 *
 * \param[in]      result
 * \param[in]      expected result
 * \param[in]      max absolute error
 * \param[in,out]  pointer to error struct object
 * \returns        true if check passed
*/

int32_t check_result(int32_t result, int32_t expected, uint32_t max_abs_error, error_s *e) {
    static int32_t half_range = ERROR_RANGE/2;
    uint32_t error_found=0;

    int32_t error = result - expected;

    // Save max positive and negative error
    if (error < e->max_negative_error) e->max_negative_error = error;
    if (error > e->max_positive_error) e->max_positive_error = error;

    if (error > (int) max_abs_error || error < -((int) max_abs_error)) {
        error_found=1;
    }

    if (error_found) {
    //if (error > max_error) {
#if PRINT_ERROR_TOO_BIG
        printf("ERROR: absolute error > %d is a failure criteria. Error found is %d\n",max_abs_error, error);
        printf("result is 0x%x, Expected is 0x%x\n",result, expected);
        printf("\n");
#if EXIT_ON_ERROR_TOO_BIG
        report_errors(max_abs_error, e);
        exit (0);
#endif
#endif
    }
    // saturate Errors
    if (error < -half_range) error = -half_range;
    if (error > half_range) error = half_range;

    e->errors[error+half_range]++; // increment the error counter

    e->num_checked++;

    return error_found;
}

uint32_t overhead_time;

//Todo: Test if this performs as well as the conversion macros in lib_dsp_qformat.h
inline int32_t qs(double d, const int32_t q_format) {
  return (int)((signed long long)((d) * ((unsigned long long)1 << (q_format+20)) + (1<<19)) >> 20);
}

void test_roots() {
    int32_t start_time, end_time;
    uint32_t cycles_taken; // absolute positive values
    timer tmr;
    error_s err;

    reset_errors(&err);

    printf("Test Roots\n");
    printf("----------\n");

    uint32_t result, expected;

#if EXPONENTIAL_INPUT
    for(uint32_t i=1; i<=32; i++) {
        uint32_t x = (unsigned long long) (1<<i)-1; // 2^x - 1 (x in 1..31)
#else
    for(uint32_t x=0; x<=MAX_UINT; x+=X_INCR) {
#endif
        double d_sqrt;

        TIME_FUNCTION(result = lib_dsp_math_squareroot(x););
#if PRINT_CYCLE_COUNT
    printf("Cycles taken for lib_dsp_math_squareroot function: %d\n", cycles_taken);
#endif

#if PRINT_INPUTS_AND_OUTPUTS
        printf ("Square Root (%.8f) : %.8f\n", F24(x), F24(result));
#endif

        TIME_FUNCTION(d_sqrt = sqrt(F24(x)));
#if PRINT_CYCLE_COUNT
    printf("Cycles taken for sqrt function: %d\n", cycles_taken);
#endif
        expected =  Q24(d_sqrt);
        check_result(result, expected, 1, &err);

    }

    printf("Error report: lib_dsp_math_squareroot vs sqrt:\n");
    report_errors(1, &err);
    printf("\n");

}

void test_multipliation_and_division() {
    int32_t q_format = 24; // location of the decimal point. Gives 8 digits of precision after conversion to
       ↪ floating point.
    q8_24 result, expected;
    error_s err;
    reset_errors(&err);

    printf("Test Multiplication and Division\n");
    printf("--------------------------------\n");
    printf("Note: All calculations are done in Q8.24 format. That gives 7 digits of precision after the
       ↪ decimal point\n");
    printf("Note: Maximum double representation of Q8.24 format: %.8f\n\n", F24(0x7FFFFFFF));

    double f0, f1;
    f0 = 11.3137085;
    f1 = 11.3137085;
    // Multiply the square root of 128 (maximum double representation of Q8.24)
    printf ("Multiplication (%.8f x %.8f): %.8f\n\n",f0, f1, F24(lib_dsp_math_multiply(Q24(f0), Q24(f1),
       ↪ q_format)));

    printf ("Multiplication (11.4 x 11.4). Will overflow!: %.8f\n\n", F24(lib_dsp_math_multiply(Q24(11.4), Q24
       ↪ (11.4), q_format)));;

    printf ("Saturated Multiplication (11.4 x 11.4): %.8f\n\n", F24(lib_dsp_math_multiply_sat(Q24(11.4), Q24
       ↪ (11.4), q_format)));;

    /*
    The result of 0.0005 x 0.0005 is 0.00000025. But this number is not representable as a binary.
    The closest representation in Q8.24 format is (4/2^24) = 0.000000238418579
    printf rounds this to 0.0000002 because the formatting string to printf specifies 8 digits of precision
       ↪ after the decimal point.
    This is the maximum precision that can be achieved with the 24 fractional bits of the Q8.24 format.
    */
    printf ("Multiplication of small numbers (0.0005 x 0.0005): %.8f\n\n", F24(lib_dsp_math_multiply(Q24
       ↪ (0.0005), Q24(0.0005), q_format)));;


    double dividend, divisor;

    dividend = 1.123456; divisor = -128;
    result = lib_dsp_math_divide(Q24(dividend), Q24(divisor), q_format);
    printf ("Signed Division %.8f / %.8f): %.8f\n\n",dividend, divisor, F24(result));
    expected = Q24(dividend/divisor);
    check_result(result, expected, 1, &err);
    dividend = -1.123456; divisor = -128;
    result = lib_dsp_math_divide(Q24(dividend), Q24(divisor), q_format);
    printf ("Signed Division %.8f / %.8f): %.8f\n\n",dividend, divisor, F24(result));
    expected = Q24(dividend/divisor);
    check_result(result, expected, 1, &err);

    dividend = -1.123456; divisor = 127.9999999;
    result = lib_dsp_math_divide(Q24(dividend), Q24(divisor), q_format);
    printf ("Signed Division %.8f / %.8f): %.8f\n\n",dividend, divisor, F24(result));
    expected = Q24(dividend/divisor);
    check_result(result, expected, 1, &err);

    dividend = 1.123456; divisor = 127.9999999;
    result = lib_dsp_math_divide(Q24(dividend), Q24(divisor), q_format);
    printf ("Signed Division %.8f / %.8f): %.8f\n\n",dividend, divisor, F24(result));
    expected = Q24(dividend/divisor);
    check_result(result, expected, 1, &err);

    result = lib_dsp_math_divide_unsigned(Q24(dividend), Q24(divisor), q_format);
    printf ("Division %.8f / %.8f): %.8f\n\n",dividend, divisor, F24(result));
    expected = Q24(dividend/divisor);
    check_result(result, expected, 1, &err);

    printf("Error report from test_multipliation_and_division:\n");
    report_errors(1, &err);
    printf("\n");

}

void test_trigonometric() {
    int32_t start_time, end_time;
    uint32_t cycles_taken; // absolute positive values
    timer tmr;

    error_s err;
    reset_errors(&err);

    printf("Test Trigonometric Functions\n");
    printf("----------------------------\n");

    /*
     * Testing lib_dsp_math_sin
     */
    printf ("Sine wave (one cycle from -Pi to +Pi) :\n");

    for(q8_24 rad = -PI_Q8_24; rad <= PI_Q8_24; rad += RAD_INCR) {
        q8_24 sine;
        TIME_FUNCTION(sine = lib_dsp_math_sin(rad));

#if PRINT_INPUTS_AND_OUTPUTS
        printf("sin(%.8f) = %.8f\n",F24(rad), F24(sine));
#endif

#if CHECK_RESULTS
        // check the fixed point result vs the floating point result from math.h
        double d_rad = F24(rad);
        double d_sine_ref = sin(d_rad);
        q8_24 expected = Q24(d_sine_ref);
        check_result(sine, expected, 1, &err);
#endif

    }
#if CHECK_RESULTS
    printf("Error report: lib_dsp_math_sin vs sin:\n");
    report_errors(1, &err);
#endif

#if PRINT_CYCLE_COUNT
    printf("Cycles taken for lib_dsp_math_sin function: %d\n", cycles_taken);
    // just to measure cycles
    tmr :> start_time;
    double sine_float = sin(3.141592653589793/4);
    tmr :> end_time;
    cycles_taken = end_time-start_time-overhead_time;
    printf("math.h sin(%.8f) = %.8f\n",3.141592653589793/4, sine_float);
    printf("Cycles taken for math.h sine function: %d\n", cycles_taken);
#endif
    printf("\n");

    /*
     * Testing lib_dsp_math_sin
     */
    printf("Cosine wave (one cycle from -Pi to +Pi) :\n");
    reset_errors(&err);

    for(q8_24 rad = -PI_Q8_24; rad <= PI_Q8_24; rad += RAD_INCR) {
        q8_24 cosine;
        TIME_FUNCTION(cosine=lib_dsp_math_cos(rad));
#if PRINT_INPUTS_AND_OUTPUTS
        printf("cos(%.8f) = %.8f\n",F24(rad), F24(cosine));
#endif
#if CHECK_RESULTS
        // check the fixed point result vs the floating point result from math.h
        double d_rad = F24(rad);
        double d_cosine_ref = cos(d_rad);
        q8_24 expected = Q24(d_cosine_ref);
        check_result(cosine, expected, 1, &err);
#endif
    }

#if CHECK_RESULTS
    printf("Error report: lib_dsp_math_cos vs cos:\n");
    report_errors(1, &err);
#endif

#if PRINT_CYCLE_COUNT
    printf("Cycles taken for cosine function: %d\n", cycles_taken);
    // just to measure cycles
    tmr :> start_time;
    double cosine_float = cos(3.141592653589793/4);
    tmr :> end_time;
    cycles_taken = end_time-start_time-overhead_time;
    printf("math.h cos(%.8f) = %.8f\n",3.141592653589793/4, cosine_float);
    printf("Cycles taken for math.h cosine function: %d\n", cycles_taken);
#endif
    printf("\n");

    /*
     * Testing lib_dsp_math_atan
     */
    printf("Test lib_dsp_math_atan\n");
    reset_errors(&err);

    int32_t worst_cycles=0;
    int32_t worst_cycles_input;

    /*
    * Test result in terms of Errors:
    * num calculations:  1000226; Errors >=1:    44561 ( 4.46%); Errors >=2:     0 ( 0.00%)
    * Max absolute error: 1
    */

#if EXPONENTIAL_INPUT
    for(int32_t i=-31; i<=31; i++) {
        int32_t x;
        if(i<0) {
            // create negative numbers
            //x = sext((1<<i), i+1); // -2^x (x in 31..1)
            x = -((1<<-i)-1); //-(2^x-1) (x in 31..1)
        } else if(i>0) {
            x = (1<<i)-1; // 2^x-1 (x in 1..31)
        } else {
            x = 0;
        }
#else
    for(uint32_t x=0; x <= MAX_Q8_24; x+= X_INCR) {
#endif


        q8_24 arctan;
        TIME_FUNCTION(arctan=lib_dsp_math_atan(x));

        if(cycles_taken>worst_cycles) {
            worst_cycles = cycles_taken;
            worst_cycles_input = x;
        }
#if PRINT_CYCLE_COUNT
        printf("Cycles taken for lib_dsp_math_atan function: %d\n", cycles_taken);
#endif
#if PRINT_INPUTS_AND_OUTPUTS
        printf("atan(%.8f) = %.8f\n",F24(x),F24(arctan));
#endif
#if CHECK_RESULTS
        double d_x = F24(x);
        double d_arctan_ref = atan(d_x);
        q8_24 expected = Q24(d_arctan_ref);
        check_result(arctan, expected, 1, &err);
#endif
    }

#if CHECK_RESULTS
    printf("Error report from lib_dsp_math_atan:\n");
    report_errors(1, &err);
#endif

#if PRINT_CYCLE_COUNT
    printf("max cyles taken for lib_dsp_math_atan function: %d, input value was %.8f\n", worst_cycles, F24(
       ↪ worst_cycles_input));
    // just to measure cycles
    double d_x = F24(worst_cycles_input);
    tmr :> start_time;
    double d_arctan = atan(d_x);
    tmr :> end_time;
    cycles_taken = end_time-start_time-overhead_time;
    printf("math.h atan(%.8f) = %.8f\n",d_x, d_arctan);
    printf("Cycles taken for math.h atan function: %u\n", cycles_taken);
#endif
    printf("\n");
}

void test_math(void)
{

    int32_t start_time, end_time;
    timer tmr;
    tmr :> start_time;
    tmr :> end_time;
    overhead_time = end_time - start_time;

    printf("Test example for Math functions\n");
    printf("===============================\n");

    test_multipliation_and_division();

    test_roots();

    test_trigonometric();

    exit (0);
}

void divide() {
    int32_t divisor = 3;
    int32_t result = 0x7FFFFFFF;;
    while(1) {
        result = lib_dsp_math_divide(result, divisor, 24);
        if(result==0) result = 0x7FFFFFFF;
    }
}

int main(void) {
    par {
        test_math();
#if DIVIDE_STRESS
        divide();
        divide();
        divide();
        divide();
#endif
    }
    return 0;
}
```

B.4 Matrix Functions

```
// Copyright (c) 2016, XMOS Ltd, All rights reserved
// XMOS DSP Library - Matrix Functions Test Program
// Uses Q24 format

// Include files
#include <stdio.h>
#include <xs1.h>
#include <lib_dsp.h>

// Define constants

#define Q_M               8
#define Q_N               24

#define MATRIX_NUM_ROWS   3
#define MATRIX_NUM_COLS   3

// Declare global variables and arrays
int32_t  Src1[] = { Q24(.11), Q24(.12), Q24(.13),
                Q24(.21), Q24(.22), Q24(.23),
                Q24(.31), Q24(.32), Q24(.33)};
int32_t  Src2[] = { Q24(.41), Q24(.42), Q24(.43),
                Q24(.51), Q24(.52), Q24(.53),
                Q24(.61), Q24(.62), Q24(.63)};
int32_t           Dst[MATRIX_NUM_ROWS*MATRIX_NUM_COLS];

int main(void)
{
                                                  // Matrix negation: R = -X
  lib_dsp_matrix_negate (Src1,                    // 'input_matrix_X':  Pointer/reference to source data
                         Dst,                     // 'result_matrix_R': Pointer to the resulting 2-dimensional
                           ↪  data array
                         MATRIX_NUM_ROWS,         // 'row_count':       Number of rows in input and output
                           ↪ matrices
                         MATRIX_NUM_COLS);        // 'column_count':    Number of columns in input and output
                           ↪ matrices

  printf ("Matrix negation: R = -X\n");
  printf ("%lf, %lf, %lf\n", F24 (Dst[0]), F24 (Dst[1]), F24 (Dst[2]));
  printf ("%lf, %lf, %lf\n", F24 (Dst[3]), F24 (Dst[4]), F24 (Dst[5]));
  printf ("%lf, %lf, %lf\n", F24 (Dst[6]), F24 (Dst[7]), F24 (Dst[8]));

                                                  // Matrix / scalar addition: R = X + a
  lib_dsp_matrix_adds (Src1,                      // 'input_matrix_X':  Pointer/reference to source data
                       Q24(2.),                   // 'scalar_value_A':  Scalar value to add to each 'input'
                         ↪ element
                       Dst,                       // 'result_matrix_R': Pointer to the resulting 2-dimensional
                         ↪  data array
                       MATRIX_NUM_ROWS,           // 'row_count':       Number of rows in input and output
                         ↪ matrices
                       MATRIX_NUM_COLS);          // 'column_count':    Number of columns in input and output
                         ↪ matrices

  printf ("Matrix / scalar addition: R = X + a\n");
  printf ("%lf, %lf, %lf\n", F24 (Dst[0]), F24 (Dst[1]), F24 (Dst[2]));
  printf ("%lf, %lf, %lf\n", F24 (Dst[3]), F24 (Dst[4]), F24 (Dst[5]));
  printf ("%lf, %lf, %lf\n", F24 (Dst[6]), F24 (Dst[7]), F24 (Dst[8]));

                                                  // Matrix / scalar multiplication: R = X * a
  lib_dsp_matrix_muls (Src1,                      // 'input_matrix_X':  Pointer/reference to source data
                       Q24(2.),                   // 'scalar_value_A':  Scalar value to multiply each 'input'
                         ↪ element by
                       Dst,                       // 'result_matrix_R': Pointer to the resulting 2-dimensional
                         ↪  data array
                       MATRIX_NUM_ROWS,           // 'row_count':       Number of rows in input and output
                         ↪ matrices
                       MATRIX_NUM_COLS,           // 'column_count':    Number of columns in input and output
                         ↪ matrices
                       Q_N);                      // 'q_format':        Fixed point format, the number of bits
                         ↪  making up fractional part

  printf ("Matrix / scalar multiplication: R = X + a\n");
  printf ("%lf, %lf, %lf\n", F24 (Dst[0]), F24 (Dst[1]), F24 (Dst[2]));
  printf ("%lf, %lf, %lf\n", F24 (Dst[3]), F24 (Dst[4]), F24 (Dst[5]));
  printf ("%lf, %lf, %lf\n", F24 (Dst[6]), F24 (Dst[7]), F24 (Dst[8]));

                                               // Matrix / matrix addition: R = X + Y
  lib_dsp_matrix_addm (Src1,                   // 'input_matrix_X':  Pointer to source data array X
                       Src2,                   // 'input_matrix_Y':  Pointer to source data array Y
                       Dst,                    // 'result_matrix_R': Pointer to the resulting 2-dimensional
                         ↪  data array
                       MATRIX_NUM_ROWS,        // 'row_count':       Number of rows in input and output
                         ↪ matrices
                       MATRIX_NUM_COLS);       // 'column_count':    Number of columns in input and output
                         ↪ matrices

  printf ("Matrix / matrix addition: R = X + Y\n");
  printf ("%lf, %lf, %lf\n", F24 (Dst[0]), F24 (Dst[1]), F24 (Dst[2]));
  printf ("%lf, %lf, %lf\n", F24 (Dst[3]), F24 (Dst[4]), F24 (Dst[5]));
  printf ("%lf, %lf, %lf\n", F24 (Dst[6]), F24 (Dst[7]), F24 (Dst[8]));

                                               // Matrix / matrix subtraction: R = X - Y
  lib_dsp_matrix_subm (Src1,                   // 'input_matrix_X':  Pointer to source data array X
                       Src2,                   // 'input_matrix_Y':  Pointer to source data array Y
                       Dst,                    // 'result_matrix_R': Pointer to the resulting 2-dimensional
                         ↪  data array
                       MATRIX_NUM_ROWS,        // 'row_count':       Number of rows in input and output
                         ↪ matrices
                       MATRIX_NUM_COLS);       // 'column_count':    Number of columns in input and output
                         ↪ matrices

  printf ("Matrix / matrix subtraction: R = X - Y\n");
  printf ("%lf, %lf, %lf\n", F24 (Dst[0]), F24 (Dst[1]), F24 (Dst[2]));
  printf ("%lf, %lf, %lf\n", F24 (Dst[3]), F24 (Dst[4]), F24 (Dst[5]));
  printf ("%lf, %lf, %lf\n", F24 (Dst[6]), F24 (Dst[7]), F24 (Dst[8]));

                                               // Matrix / matrix multiplication: R = X * Y
  lib_dsp_matrix_mulm (Src1,                   // 'input_matrix_X':  Pointer to source data array X
                       Src2,                   // 'input_matrix_Y':  Pointer to source data array Y
                       Dst,                    // 'result_matrix_R': Pointer to the resulting 2-dimensional
                         ↪  data array
                       MATRIX_NUM_ROWS,        // 'row_count':       Number of rows in input and output
                         ↪ matrices
                       MATRIX_NUM_COLS,        // 'column_count':    Number of columns in input and output
                         ↪ matrices
                       Q_N);                   // 'q_format':        Fixed point format, the number of bits
                         ↪  making up fractional part

  printf ("Matrix / matrix multiplication: R = X * Y\n");
  printf ("%lf, %lf, %lf\n", F24 (Dst[0]), F24 (Dst[1]), F24 (Dst[2]));
  printf ("%lf, %lf, %lf\n", F24 (Dst[3]), F24 (Dst[4]), F24 (Dst[5]));
  printf ("%lf, %lf, %lf\n", F24 (Dst[6]), F24 (Dst[7]), F24 (Dst[8]));

                                               // Matrix transposition
  lib_dsp_matrix_transpose (Src1,              // 'input_matrix_X':  Pointer to source data array
                            Dst,               // 'result_matrix_R': Pointer to the resulting 2-dimensional
                              ↪  data array
                            MATRIX_NUM_ROWS,   // 'row_count':       Number of rows in input and output
                              ↪ matrices
                            MATRIX_NUM_COLS,   // 'column_count':    Number of columns in input and output
                              ↪ matrices
                            Q_N);              // 'q_format':        Fixed point format, the number of bits
                              ↪  making up fractional part

  printf ("Matrix transposition\n");
  printf ("%lf, %lf, %lf\n", F24 (Dst[0]), F24 (Dst[1]), F24 (Dst[2]));
  printf ("%lf, %lf, %lf\n", F24 (Dst[3]), F24 (Dst[4]), F24 (Dst[5]));
  printf ("%lf, %lf, %lf\n", F24 (Dst[6]), F24 (Dst[7]), F24 (Dst[8]));

  return (0);
}
```

B.5 Statistics Functions

```
// Copyright (c) 2016, XMOS Ltd, All rights reserved
// XMOS DSP Library - Statistics Functions Test Program
// Uses Q24 format

// Include files
#include <stdio.h>
#include <xs1.h>
#include <lib_dsp.h>

// Define constants

#define Q_M               8
#define Q_N               24

#define SAMPLE_LENGTH         50
#define SHORT_SAMPLE_LENGTH   5

// Declare global variables and arrays
int32_t  Src[] = { Q24(.11), Q24(.12), Q24(.13), Q24(.14), Q24(.15), Q24(.16), Q24(.17), Q24(.18), Q24(.19),
  ↪ Q24(.20),
               Q24(.21), Q24(.22), Q24(.23), Q24(.24), Q24(.25), Q24(.26), Q24(.27), Q24(.28), Q24(.29), Q24
                 ↪ (.30),
               Q24(.31), Q24(.32), Q24(.33), Q24(.34), Q24(.35), Q24(.36), Q24(.37), Q24(.38), Q24(.39), Q24
                 ↪ (.40),
               Q24(.41), Q24(.42), Q24(.43), Q24(.44), Q24(.45), Q24(.46), Q24(.47), Q24(.48), Q24(.49), Q24
                 ↪ (.50),
               Q24(.51), Q24(.52), Q24(.53), Q24(.54), Q24(.55), Q24(.56), Q24(.57), Q24(.58), Q24(.59), Q24
                 ↪ (.60)};
int32_t  Src2[] = { Q24(.51), Q24(.52), Q24(.53), Q24(.54), Q24(.55)};
int32_t           Dst[SAMPLE_LENGTH];

int main(void)
{
  int32_t result;

  result =
    lib_dsp_vector_mean (Src,                     // Input vector
                         SAMPLE_LENGTH,           // Vector length
                         Q_N);                    // Q Format N

  printf ("Vector Mean = %lf\n", F24 (result));

  result =
    lib_dsp_vector_power (Src,                    // Input vector
                          SAMPLE_LENGTH,          // Vector length
                          Q_N);                   // Q Format N

  printf ("Vector Power (sum of squares) = %lf\n", F24 (result));

  result =
    lib_dsp_vector_rms (Src,                      // Input vector
                        SAMPLE_LENGTH,            // Vector length
                        Q_N);                     // Q Format N

  printf ("Vector Root Mean Square = %lf\n", F24 (result));

  result =
    lib_dsp_vector_dotprod (Src,                  // Input vector 1
                            Src2,                 // Input vector 2
                            SHORT_SAMPLE_LENGTH,  // Vector length
                            Q_N);                 // Q Format N

  printf ("Vector Dot Product = %lf\n", F24 (result));

  return (0);
}
```

B.6 Vector Functions

```
// Copyright (c) 2016, XMOS Ltd, All rights reserved
// XMOS DSP Library - Vector Functions Test Program
// Uses Q24 format

#include <stdio.h>
#include <xs1.h>
#include <lib_dsp.h>

#define Q_M               8
#define Q_N               24

#define SAMPLE_LENGTH         50
#define SHORT_SAMPLE_LENGTH   5

int32_t  Src[] = { Q24(.11), Q24(.12), Q24(.13), Q24(.14), Q24(.15), Q24(.16), Q24(.17), Q24(.18), Q24(.19),
   ↪ Q24(.20),
               Q24(.21), Q24(.22), Q24(.23), Q24(.24), Q24(.25), Q24(.26), Q24(.27), Q24(.28), Q24(.29), Q24
                  ↪ (.30),
               Q24(.31), Q24(.32), Q24(.33), Q24(.34), Q24(.35), Q24(.36), Q24(.37), Q24(.38), Q24(.39), Q24
                  ↪ (.40),
               Q24(.41), Q24(.42), Q24(.43), Q24(.44), Q24(.45), Q24(.46), Q24(.47), Q24(.48), Q24(.49), Q24
                  ↪ (.50),
               Q24(.51), Q24(.52), Q24(.53), Q24(.54), Q24(.55), Q24(.56), Q24(.57), Q24(.58), Q24(.59), Q24
                  ↪ (.60)};
int32_t  Src2[] = { Q24(.51), Q24(.52), Q24(.53), Q24(.54), Q24(.55), Q24(.56), Q24(.57), Q24(.58), Q24(.59),
   ↪ Q24(.60)};
int32_t  Src3[] = { Q24(.61), Q24(.62), Q24(.63), Q24(.64), Q24(.65), Q24(.66), Q24(.67), Q24(.68), Q24(.69),
   ↪ Q24(.70)};
int32_t           Dst[SAMPLE_LENGTH];

int main(void)
{
  int32_t result;
  int32_t i;

  result =
    lib_dsp_vector_minimum (Src,                    // Input vector
                            SAMPLE_LENGTH);         // Vector length

  printf ("Minimum location = %d\n", result);
  printf ("Minimum = %lf\n", F24 (Src[result]));

  result =
    lib_dsp_vector_maximum (Src,                    // Input vector
                            SAMPLE_LENGTH);         // Vector length

  printf ("Maximum location = %d\n", result);
  printf ("Maximum = %lf\n", F24 (Src[result]));

  lib_dsp_vector_negate (Src,                       // Input vector
                         Dst,                       // Output vector
                         SHORT_SAMPLE_LENGTH);      // Vector length

  printf ("Vector Negate Result\n");
  for (i = 0; i < SHORT_SAMPLE_LENGTH; i++)
  {
    printf ("Dst[%d] = %lf\n", i, F24 (Dst[i]));
  }

  lib_dsp_vector_abs (Src,                          // Input vector
                      Dst,                          // Output vector
                      SHORT_SAMPLE_LENGTH);         // Vector length

  printf ("Vector Absolute Result\n");
  for (i = 0; i < SHORT_SAMPLE_LENGTH; i++)
  {
    printf ("Dst[%d] = %lf\n", i, F24 (Dst[i]));
  }

  lib_dsp_vector_adds (Src,                         // Input vector
                       Q24(2.),                     // Input scalar
                       Dst,                         // Output vector
                       SHORT_SAMPLE_LENGTH);        // Vector length
```

```
printf ("Vector / scalar addition Result\n");
for (i = 0; i < SHORT_SAMPLE_LENGTH; i++)
{
  printf ("Dst[%d] = %lf\n", i, F24 (Dst[i]));
}

lib_dsp_vector_muls (Src,                        // Input vector
                     Q24(2.),                     // Input scalar
                     Dst,                         // Output vector
                     SHORT_SAMPLE_LENGTH,         // Vector length
                     Q_N);                        // Q Format N

printf ("Vector / scalar multiplication Result\n");
for (i = 0; i < SHORT_SAMPLE_LENGTH; i++)
{
  printf ("Dst[%d] = %lf\n", i, F24 (Dst[i]));
}

lib_dsp_vector_addv (Src,                        // Input vector
                     Src2,                        // Input vector 2
                     Dst,                         // Output vector
                     SHORT_SAMPLE_LENGTH);        // Vector length

printf ("Vector / vector addition Result\n");
for (i = 0; i < SHORT_SAMPLE_LENGTH; i++)
{
  printf ("Dst[%d] = %lf\n", i, F24 (Dst[i]));
}

lib_dsp_vector_subv (Src,                        // Input vector
                     Src2,                        // Input vector 2
                     Dst,                         // Output vector
                     SHORT_SAMPLE_LENGTH);        // Vector length

printf ("Vector / vector subtraction Result\n");
for (i = 0; i < SHORT_SAMPLE_LENGTH; i++)
{
  printf ("Dst[%d] = %lf\n", i, F24 (Dst[i]));
}

lib_dsp_vector_mulv (Src,                        // Input vector
                     Src2,                        // Input vector 2
                     Dst,                         // Output vector
                     SHORT_SAMPLE_LENGTH,         // Vector length
                     Q_N);                        // Q Format N

printf ("Vector / vector multiplication Result\n");
for (i = 0; i < SHORT_SAMPLE_LENGTH; i++)
{
  printf ("Dst[%d] = %lf\n", i, F24 (Dst[i]));
}

lib_dsp_vector_mulv_adds (Src,                   // Input vector
                          Src2,                   // Input vector 2
                          Q24(2.),                // Input scalar
                          Dst,                    // Output vector
                          SHORT_SAMPLE_LENGTH,    // Vector length
                          Q_N);                   // Q Format N

printf ("Vector multiplication and scalar addition Result\n");
for (i = 0; i < SHORT_SAMPLE_LENGTH; i++)
{
  printf ("Dst[%d] = %lf\n", i, F24 (Dst[i]));
}

lib_dsp_vector_muls_addv (Src,                   // Input vector
                          Q24(2.),                // Input scalar
                          Src2,                   // Input vector 2
                          Dst,                    // Output vector
                          SHORT_SAMPLE_LENGTH,    // Vector length
                          Q_N);                   // Q Format N

printf ("Vector / Scalar multiplication and vector addition Result\n");
for (i = 0; i < SHORT_SAMPLE_LENGTH; i++)
```

```
  {
    printf ("Dst[%d] = %lf\n", i, F24 (Dst[i]));
  }

  lib_dsp_vector_muls_subv (Src,                        // Input vector
                            Q24(2.),                    // Input scalar
                            Src2,                       // Input vector 2
                            Dst,                        // Output vector
                            SHORT_SAMPLE_LENGTH,        // Vector length
                            Q_N);                       // Q Format N

  printf ("Vector / Scalar multiplication and vector subtraction Result\n");
  for (i = 0; i < SHORT_SAMPLE_LENGTH; i++)
  {
    printf ("Dst[%d] = %lf\n", i, F24 (Dst[i]));
  }

  lib_dsp_vector_mulv_addv (Src,                        // Input vector
                            Src2,                       // Input vector 2
                            Src3,                       // Input vector 2
                            Dst,                        // Output vector
                            SHORT_SAMPLE_LENGTH,        // Vector length
                            Q_N);                       // Q Format N

  printf ("Vector / Vector multiplication and vector addition Result\n");
  for (i = 0; i < SHORT_SAMPLE_LENGTH; i++)
  {
    printf ("Dst[%d] = %lf\n", i, F24 (Dst[i]));
  }

  lib_dsp_vector_mulv_subv (Src,                        // Input vector
                            Src2,                       // Input vector 2
                            Src3,                       // Input vector 2
                            Dst,                        // Output vector
                            SHORT_SAMPLE_LENGTH,        // Vector length
                            Q_N);                       // Q Format N

  printf ("Vector / Vector multiplication and vector subtraction Result\n");
  for (i = 0; i < SHORT_SAMPLE_LENGTH; i++)
  {
    printf ("Dst[%d] = %lf\n", i, F24 (Dst[i]));
  }

  return (0);
}
```

B.7 FFT and inverse FFT

Note: The method for processing two real signals with a single complex FFT was improved. It now requires only half the memory. See Build Configurations tworeals and tworeals_int16_buf.

```
// Copyright (c) 2016, XMOS Ltd, All rights reserved
// XMOS DSP Library - Example to use FFT and inverse FFT

#include <stdio.h>
#include <xs1.h>
#include <xclib.h>
#include <lib_dsp.h>
#include <stdint.h>

#define TRACE_VALUES 1
#if TRACE_VALUES
#define PRINT_FFT_INPUT 1
#define PRINT_FFT_OUTPUT 1
#define PRINT_IFFT_OUTPUT 1
#define N_FFT_POINTS 32
#define PRINT_CYCLE_COUNT 0
#else
#define PRINT_FFT_INPUT 0
#define PRINT_FFT_OUTPUT 0
#define PRINT_IFFT_OUTPUT 0
#define N_FFT_POINTS 4096
#define PRINT_CYCLE_COUNT 1
#endif

#define INPUT_FREQ N_FFT_POINTS/8

#ifndef INT16_BUFFERS
#define INT16_BUFFERS 0 // disabled by default. int32_t buffers is default
#endif

#ifndef TWOREALS
#define TWOREALS 0 // Processing two real signals with a single complex FFT. Disabled by default
#endif

#if TWOREALS
int32_t do_tworeals_fft_and_ifft();
#else
int32_t do_complex_fft_and_ifft();
#endif

// Array holding one complex signal or two real signals
#if INT16_BUFFERS
lib_dsp_fft_complex_short_t data[N_FFT_POINTS];
#else
lib_dsp_fft_complex_t  data[N_FFT_POINTS];
#endif


/**
 * Experiments with functions that generate sine and cosine signals with a defined number of points
 **/
// Macros to ease the use of the sin_N and cos_N functions
// Note: 31-clz(N) == log2(N) when N is power of two
#define SIN(M, N) sin_N(M, 31-clz(N), lib_dsp_sine_ ## N)
#define COS(M, N) cos_N(M, 31-clz(N), lib_dsp_sine_ ## N)

int32_t sin_N(int32_t x, int32_t log2_points_per_cycle, const int32_t sine[]);
int32_t cos_N(int32_t x, int32_t log2_points_per_cycle, const int32_t sine[]);

// generate sine signal with a configurable number of samples per cycle
#pragma unsafe arrays
int32_t sin_N(int32_t x, int32_t log2_points_per_cycle, const int32_t sine[]) {
    // size of sine[] must be equal to num_points!
    int32_t num_points = (1<<log2_points_per_cycle);
    int32_t half_num_points = num_points>>1;

    x = x & (num_points-1); // mask off the index

    switch (x >> (log2_points_per_cycle-2)) { // switch on upper two bits
```

```
        // upper two bits determine the quadrant.
        case 0: return sine[x];
        case 1: return sine[half_num_points-x];
        case 2: return -sine[x-half_num_points];
        case 3: return -sine[num_points-x];
    }
    return 0; // unreachable
}

// generate cosine signal with a configurable number of samples per cycle
#pragma unsafe arrays
int32_t cos_N(int32_t x, int32_t log2_points_per_cycle, const int32_t sine[]) {
    int32_t quarter_num_points = (1<<(log2_points_per_cycle-2));
    return sin_N(x+(quarter_num_points), log2_points_per_cycle, sine); // cos a = sin(a + 2pi/4)
}

int main( void )
{

#if TWOREALS
    do_tworeals_fft_and_ifft();
#else
    do_complex_fft_and_ifft();
#endif
    return 0;
};

#if INT16_BUFFERS
#define RIGHT_SHIFT 17  // shift down to Q14. Q15 would cause overflow.
#else
#define RIGHT_SHIFT 1   // shift down to Q30. Q31 would cause overflow
#endif

void print_data_array() {
#if INT16_BUFFERS
    printf("re,      im         \n");
    for(int32_t i=0; i<N_FFT_POINTS; i++) {
        printf( "%.5f, %.5f\n", F14(data[i].re), F14(data[i].im));
    }
#else
    printf("re,           im         \n");
    for(int32_t i=0; i<N_FFT_POINTS; i++) {
        printf( "%.10f, %.10f\n", F30(data[i].re), F30(data[i].im));
    }
#endif
}

#if TWOREALS
void generate_tworeals_test_signal(int32_t N, int32_t test) {
    switch(test) {
     case 0: {
         printf("++++ Test %d: %d point FFT of two real signals:: re0: %d Hz cosine, re1: %d Hz cosine\n"
                ,test,N,INPUT_FREQ,INPUT_FREQ);
         for(int32_t i=0; i<N; i++) {
             data[i].re = COS(i, 8) >> RIGHT_SHIFT;
             data[i].im = COS(i, 8) >> RIGHT_SHIFT;
         }
         break;
     }
     case 1: {
         printf("++++ Test %d: %d point FFT of two real signals:: re0: %d Hz sine, re1: %d Hz sine\n"
                ,test,N,INPUT_FREQ,INPUT_FREQ);
         for(int32_t i=0; i<N; i++) {
             data[i].re = SIN(i, 8) >> RIGHT_SHIFT;
             data[i].im = SIN(i, 8) >> RIGHT_SHIFT;
         }
         break;
     }
     case 2: {
         printf("++++ Test %d: %d point FFT of two real signals:: re0: %d Hz sine, re1: %d Hz cosine\n"
                ,test,N,INPUT_FREQ,INPUT_FREQ);
         for(int32_t i=0; i<N; i++) {
             data[i].re = SIN(i, 8) >> RIGHT_SHIFT;
             data[i].im = COS(i, 8) >> RIGHT_SHIFT;
         }
```

```
            break;
        }
        case 3: {
            printf("++++ Test %d: %d point FFT of two real signals:: re0: %d Hz cosine, re1: %d Hz sine\n"
                    ,test,N,INPUT_FREQ,INPUT_FREQ);
            for(int32_t i=0; i<N; i++) {
                data[i].re = COS(i, 8) >> RIGHT_SHIFT;
                data[i].im = SIN(i, 8) >> RIGHT_SHIFT;
            }
            break;
        }
        }
#if PRINT_FFT_INPUT
    printf("Generated Two Real Input Signals:\n");
    print_data_array();
#endif
}

int32_t do_tworeals_fft_and_ifft() {
    timer tmr;
    uint32_t start_time, end_time, overhead_time, cycles_taken;
    tmr :> start_time;
    tmr :> end_time;
    overhead_time = end_time - start_time;
#if INT16_BUFFERS
    printf("FFT/iFFT of two real signals of type int16_t\n");
#else
    printf("FFT/iFFT of two real signals of type int32_t\n");
#endif
    printf("============================================\n");

    for(int32_t test=0; test<4; test++) {
        generate_tworeals_test_signal(N_FFT_POINTS, test);

        tmr :> start_time;
#if INT16_BUFFERS
        lib_dsp_fft_complex_t tmp_data[N_FFT_POINTS]; // tmp buffer to enable 32-bit FFT/iFFT
        lib_dsp_fft_short_to_long(tmp_data, data, N_FFT_POINTS); // convert into tmp buffer
        lib_dsp_fft_bit_reverse(tmp_data, N_FFT_POINTS);
        lib_dsp_fft_forward(tmp_data, N_FFT_POINTS, FFT_SINE(N_FFT_POINTS));
        lib_dsp_fft_split_spectrum(tmp_data, N_FFT_POINTS);
        lib_dsp_fft_long_to_short(data, tmp_data, N_FFT_POINTS); // convert from tmp buffer
#else
        lib_dsp_fft_bit_reverse(data, N_FFT_POINTS);
        lib_dsp_fft_forward(data, N_FFT_POINTS, FFT_SINE(N_FFT_POINTS));
        lib_dsp_fft_split_spectrum(data, N_FFT_POINTS);
#endif
        tmr :> end_time;
        cycles_taken = end_time-start_time-overhead_time;
#if PRINT_CYCLE_COUNT
        printf("Cycles taken for %d point FFT of two purely real signals: %d\n", N_FFT_POINTS, cycles_taken);
#endif

#if PRINT_FFT_OUTPUT
        // Print forward complex FFT results
        //printf( "First half of Complex FFT output spectrum of Real signal 0 (cosine):\n");
        printf( "FFT output of two half spectra. Second half could be discarded due to symmetry without
           ↪ loosing information\n");
        printf( "spectrum of first signal in data[0..N/2-1]. spectrum of second signa in data[N/2..N-1]:\n");

        print_data_array();
#if 0
        printf( "First half of Complex FFT output spectrum of Real signal 1 (sine):\n");
        printf("re,           im         \n");
        for(int32_t i=0; i<N_FFT_POINTS; i++) {
            printf( "%.10f, %.10f\n", F30(data[i].re), F30(data[i].im));
        }
#endif
#endif

        tmr :> start_time;
#if INT16_BUFFERS
        lib_dsp_fft_short_to_long(tmp_data, data, N_FFT_POINTS); // convert into tmp buffer
        lib_dsp_fft_merge_spectra(tmp_data, N_FFT_POINTS);
        lib_dsp_fft_bit_reverse(tmp_data, N_FFT_POINTS);
        lib_dsp_fft_inverse(tmp_data, N_FFT_POINTS, FFT_SINE(N_FFT_POINTS));
        lib_dsp_fft_long_to_short(data, tmp_data, N_FFT_POINTS); // convert from tmp buffer
#else
        lib_dsp_fft_merge_spectra(data, N_FFT_POINTS);
        lib_dsp_fft_bit_reverse(data, N_FFT_POINTS);
        lib_dsp_fft_inverse(data, N_FFT_POINTS, FFT_SINE(N_FFT_POINTS));
#endif

        tmr :> end_time;
        cycles_taken = end_time-start_time-overhead_time;
#if PRINT_CYCLE_COUNT
        printf("Cycles taken for iFFT: %d\n", cycles_taken);
#endif

#if PRINT_IFFT_OUTPUT
        printf( "////// Time domain signal after lib_dsp_fft_inverse\n");
        print_data_array();
#endif
        printf("\n" ); // Test delimiter
    }

    printf( "DONE.\n" );
    return 0;
}
# else  // Complex Signals
void generate_complex_test_signal(int32_t N, int32_t test) {
    switch(test) {
    case 0: {
        printf("++++ Test 0: %d point FFT/iFFT of complex signal:: Real: %d Hz cosine, Imag: 0\n"
                ,N_FFT_POINTS,N_FFT_POINTS/8);
        for(int32_t i=0; i<N; i++) {
            data[i].re = COS(i, 8) >> RIGHT_SHIFT;
            data[i].im = 0;
        }
        break;
    }
    case 1: {
        printf("++++ Test 1: %d point FFT/iFFT of complex signal:: Real: %d Hz sine, Imag: 0\n"
                ,N_FFT_POINTS,N_FFT_POINTS/8);
        for(int32_t i=0; i<N; i++) {
            data[i].re = SIN(i, 8) >> RIGHT_SHIFT;
            data[i].im = 0;
        }
        break;
    }
    case 2: {
        printf("++++ Test 2: %d point FFT/iFFT of complex signal: Real: 0, Imag: %d Hz cosine\n"
                ,N_FFT_POINTS,N_FFT_POINTS/8);
        for(int32_t i=0; i<N; i++) {
            data[i].re = 0;
            data[i].im = COS(i, 8) >> RIGHT_SHIFT;
        }
        break;
    }
    case 3: {
        printf("++++ Test 3: %d point FFT/iFFT of complex signal: Real: 0, Imag: %d Hz sine\n"
                ,N_FFT_POINTS,N_FFT_POINTS/8);
        for(int32_t i=0; i<N; i++) {
            data[i].re = 0;
            data[i].im = SIN(i, 8) >> RIGHT_SHIFT;
        }
        break;
    }
    }
#if PRINT_FFT_INPUT
    printf("Generated Signal:\n");
    print_data_array();
#endif
}

int32_t do_complex_fft_and_ifft() {
    timer tmr;
    uint32_t start_time, end_time, overhead_time, cycles_taken;
    tmr :> start_time;
    tmr :> end_time;
    overhead_time = end_time - start_time;

#if INT16_BUFFERS
    printf("FFT/iFFT of a complex signal of type int16_t\n");
#else
    printf("FFT/iFFT of a complex signal of type int32_t\n");
#endif
    printf("============================================\n");

    for(int32_t test=0; test<4; test++) {
        generate_complex_test_signal(N_FFT_POINTS, test);

        tmr :> start_time;

#if INT16_BUFFERS
        lib_dsp_fft_complex_t tmp_data[N_FFT_POINTS]; // tmp buffer to enable 32-bit FFT/iFFT
        // convert into int32_t temporary buffer
        lib_dsp_fft_short_to_long(tmp_data, data, N_FFT_POINTS);
        // 32 bit FFT
        lib_dsp_fft_bit_reverse(tmp_data, N_FFT_POINTS);
        lib_dsp_fft_forward(tmp_data, N_FFT_POINTS, FFT_SINE(N_FFT_POINTS));
        // convert back into int16_t buffer
        lib_dsp_fft_long_to_short(data, tmp_data, N_FFT_POINTS);
#else
        // 32 bit FFT
        lib_dsp_fft_bit_reverse(data, N_FFT_POINTS);
        lib_dsp_fft_forward(data, N_FFT_POINTS, FFT_SINE(N_FFT_POINTS));
#endif
        tmr :> end_time;
        cycles_taken = end_time-start_time-overhead_time;


#if PRINT_CYCLE_COUNT
        printf("Cycles taken for %d point FFT of complex signal: %d\n", N_FFT_POINTS, cycles_taken);
#endif


#if PRINT_FFT_OUTPUT
        printf( "FFT output:\n");
        print_data_array();
#endif

        tmr :> start_time;
#if INT16_BUFFERS
        // convert into int32_t temporary buffer
        lib_dsp_fft_short_to_long(tmp_data, data, N_FFT_POINTS);
        // 32 bit iFFT
        lib_dsp_fft_bit_reverse(tmp_data, N_FFT_POINTS);
        lib_dsp_fft_inverse(tmp_data, N_FFT_POINTS, FFT_SINE(N_FFT_POINTS));
        // convert back into int16_t buffer
        lib_dsp_fft_long_to_short(data, tmp_data, N_FFT_POINTS);
#else
        // 32 bit iFFT
        lib_dsp_fft_bit_reverse(data, N_FFT_POINTS);
        lib_dsp_fft_inverse(data, N_FFT_POINTS, FFT_SINE(N_FFT_POINTS));
#endif
        tmr :> end_time;
        cycles_taken = end_time-start_time-overhead_time;
#if PRINT_CYCLE_COUNT
        printf("Cycles taken for iFFT: %d\n", cycles_taken);
#endif

#if PRINT_IFFT_OUTPUT
        printf( "////// Time domain signal after lib_dsp_fft_inverse\n");
        print_data_array();
#endif
        printf("\n" ); // Test delimiter
    }

    printf( "DONE.\n" );
    return 0;

}
#endif
```

B.8 FFT Processing of signals received through a double buffer

```
// Copyright (c) 2016, XMOS Ltd, All rights reserved

#include <stdio.h>
#include <xs1.h>
#include <lib_dsp.h>
#include <stdlib.h>
#include <math.h>

/**
Example showing FFT processing of a configurable number of input signals received through a double buffer.
----------------------------------------------------------------------------------------------------------


This example shows how two tasks can implement a double buffering
mechanism accessing a shared memory area.

The task produce_samples fills one buffer with a stereo signal
whilst the do_fft task processes the other buffer
"in place" which means the buffer is also used to store the real part of the result
Tasks access these buffers via *movable* pointers. These pointers can
be safely transferred between tasks without any race conditions
between the tasks using them.

When the the produce_samples taks has finished filling the buffer, it calls the
swap function of the interface to synchronise with the do_fft task and swap pointers (buffers)

.. aafig::

   +---------------+                     +------------------+
   |  `do_fft`     |   `bufswap_i`       | `produce_samples`|
   |               +<------------------->|                  |
   |   +---------+ |   `interface`       |   +---------+    |
   |   |`movable`| |                     |   |`movable`|    |
   |   |`pointer`| |                     |   |`pointer`|    |
   |   +----+----+ |                     |   +----+----+    |
   |        |      |                     |        |         |
   +--------|------+                     +--------|---------+
            |             `MEMORY`                |
            |          +------+------+            |
            |          |`buf1`|`buf2`|            |
            +--------->+      |      +<-----------+
                       |      |      |
                       |      |      |
                       +------+------+
**/

/** Global configuration defines **/

#ifndef NUM_CHANS
// Number of input channels
#define NUM_CHANS 4
#endif


#ifndef N_FFT_POINTS
// FFT Points
#define N_FFT_POINTS 256
#endif

#ifndef INT16_BUFFERS
#define INT16_BUFFERS 0 // Disabled by default
#endif

#ifndef TWOREALS
#define TWOREALS 0 // Processing two real signals with a single complex FFT. Disabled by default
#endif

#define SAMPLE_FREQ 48000
#define SAMPLE_PERIOD_CYCLES XS1_TIMER_HZ/SAMPLE_FREQ

#ifndef CHECK_TIMING
#define CHECK_TIMING 0
#endif
```

```
#ifndef PRINT_INPUTS_AND_OUTPUTS
#define PRINT_INPUTS_AND_OUTPUTS 0
#endif

/****/

/** Declaration of Data Types and Memory Buffers **/

// Array holding one complex signal or two real signals
#if INT16_BUFFERS
#define SIGNAL_ARRAY_TYPE lib_dsp_fft_complex_short_t
#define OUTPUT_SUM_TYPE int32_t // double precision variable to avoid overflow in addition
#else
#define SIGNAL_ARRAY_TYPE lib_dsp_fft_complex_t
#define OUTPUT_SUM_TYPE int64_t // double precision variable to avoid overflow in addition
#endif

#if TWOREALS
#define NUM_SIGNAL_ARRAYS NUM_CHANS/2
#else
#define NUM_SIGNAL_ARRAYS NUM_CHANS
#endif

/** Union to store blocks of samples for multiple digital signals
*/
typedef union {
    SIGNAL_ARRAY_TYPE data[NUM_SIGNAL_ARRAYS][N_FFT_POINTS];  // time domain or frequency domain signals
    SIGNAL_ARRAY_TYPE half_spectra[NUM_SIGNAL_ARRAYS*2][N_FFT_POINTS/2];  // frequency domain half spectra
} multichannel_sample_block_s ;
/****/

/** Print Functions **/
void print_signal(SIGNAL_ARRAY_TYPE signal[N_FFT_POINTS]) {
#if INT16_BUFFERS
       printf("re,      im         \n");
       for(int32_t i=0; i<N_FFT_POINTS; i++) {
           //printf( "0x%x, 0x%x\n", signal[i].re, signal[i].im);
           printf( "%.5f, %.5f\n", F14(signal[i].re), F14(signal[i].im));
       }

#else
       printf("re,      im         \n");
       for(int32_t i=0; i<N_FFT_POINTS; i++) {
           printf( "%.8f, %.8f\n", F24(signal[i].re), F24(signal[i].im));
       }
#endif
}

void print_buffer(multichannel_sample_block_s *buffer) {
    for(int32_t c=0; c<NUM_SIGNAL_ARRAYS; c++) {
        print_signal(buffer->data[c]);
    }
}
/****/


/**
 The interface between the two tasks is a single transaction that swaps
 the movable pointers. It has an argument that is a reference to a
 movable pointer. Since it is a reference the server side of the
 connection can update the argument.
*/

interface bufswap_i {
  void swap(multichannel_sample_block_s * movable &x);
};

/**
 The do_fft task takes as arguments the server end of an interface
 connection and the initial buffer it is going to process. It is
 initialized by creating a movable pointer to this buffer and then
 processing it.
*/
void do_fft(server interface bufswap_i input,
        multichannel_sample_block_s * initial_buffer)
{
  multichannel_sample_block_s * movable buffer = initial_buffer;

  timer tmr; uint32_t start_time, end_time, overhead_time;
  tmr :> start_time;
  tmr :> end_time;
  overhead_time = end_time - start_time;

#if INT16_BUFFERS
  printf("%d Point FFT Processing of %d int16_t signals received through a double buffer\n"
                ,N_FFT_POINTS,NUM_CHANS);
#else
  printf("%d Point FFT Processing of %d int32_t signals received through a double buffer\n"
                ,N_FFT_POINTS,NUM_CHANS);
#endif

  /** The main loop of the filling task waits for a swap transaction with
      the other task and implements the swap of pointers. After that it
      fills the new buffer it has been given:
  */
  while (1) {
    // swap buffers
    select {
      case input.swap(multichannel_sample_block_s * movable &input_buf):
        // Swapping uses the 'move' operator. This operator transfers the
        // pointer to a new variable, setting the original variable to null.
        // The 'display_buffer' variable is a reference, so updating it will
        // update the pointer passed in by the other task.
        multichannel_sample_block_s * movable tmp;
        tmp = move(input_buf);
        input_buf = move(buffer);
        buffer = move(tmp);

#if PRINT_INPUTS_AND_OUTPUTS
        print_buffer(buffer);
#endif

        tmr :> start_time;

        // Do FFTs
        for(int32_t a=0; a<NUM_SIGNAL_ARRAYS; a++) {
            // process the new buffer "in place"
    #if INT16_BUFFERS
            lib_dsp_fft_complex_t tmp_buffer[N_FFT_POINTS];
            lib_dsp_fft_short_to_long(tmp_buffer, buffer->data[a], N_FFT_POINTS); // convert into tmp buffer
            lib_dsp_fft_bit_reverse(tmp_buffer, N_FFT_POINTS);
            lib_dsp_fft_forward(tmp_buffer, N_FFT_POINTS, FFT_SINE(N_FFT_POINTS));
    #if TWOREALS
            lib_dsp_fft_split_spectrum(tmp_buffer, N_FFT_POINTS);

    #endif
            lib_dsp_fft_long_to_short(buffer->data[a], tmp_buffer, N_FFT_POINTS); // convert from tmp buffer
    ////// 32 bit buffers
    #else
            lib_dsp_fft_bit_reverse(buffer->data[a], N_FFT_POINTS);
            lib_dsp_fft_forward(buffer->data[a], N_FFT_POINTS, FFT_SINE(N_FFT_POINTS));
    #if TWOREALS
            lib_dsp_fft_split_spectrum(buffer->data[a], N_FFT_POINTS);
    #endif

    #endif

        }

        // Process the frequency domain of all NUM_CHANS channels.
        // 1. Lowpass
        // cut off frequency: (N_FFT_POINTS/4 * Fs/N_FFT_POINTS) Hz = (48000/4) Hz = 12 kHz
        uint32_t cutoff_idx = N_FFT_POINTS/4;
        // 2. Calculate average per frequency bin into the output signal array.

        // To calculate the average over all channels.
        // To divide by NUM_CHANS, shift down throughout the loop log2(NUM_CHANS) times to avoid overflow.
        uint32_t log_num_chan = log2(NUM_CHANS);
        uint32_t step = NUM_CHANS/log_num_chan;
        uint shift_idx = step;

        static SIGNAL_ARRAY_TYPE output[N_FFT_POINTS];

    #if TWOREALS
        for(unsigned i = 0; i < N_FFT_POINTS/2; i++) {
          OUTPUT_SUM_TYPE output_re = 0;
          OUTPUT_SUM_TYPE output_im = 0;
          for(int32_t c=0; c<NUM_CHANS; c++) {
             if(i>=cutoff_idx) {
               buffer->half_spectra[c][i].re = 0;
               buffer->half_spectra[c][i].im = 0;
             }
             output_re += buffer->half_spectra[c][i].re;
             output_im += buffer->half_spectra[c][i].im;
          }
          output[i].re = (OUTPUT_SUM_TYPE) output_re/NUM_CHANS; // average
          output[i].im = (OUTPUT_SUM_TYPE) output_im/NUM_CHANS; // average
        }
    #else // Complex
        for(unsigned i = 0; i < N_FFT_POINTS/2; i++) {
          OUTPUT_SUM_TYPE output_re = 0, output_re_ri = 0;
          OUTPUT_SUM_TYPE output_im = 0, output_im_ri = 0;
          uint32_t ri = N_FFT_POINTS-i; // reverse index
          for(int32_t c=0; c<NUM_CHANS; c++) {
            if(i>=cutoff_idx) {
              buffer->data[c][i].re = 0;
              buffer->data[c][i].im = 0;
              if(i>0) {
                buffer->data[c][N_FFT_POINTS-i].re = 0;
                buffer->data[c][N_FFT_POINTS-i].im = 0;
              }
            }
            output_re += buffer->data[c][i].re;
            output_im += buffer->data[c][i].im;
            if(i>0) {
              output_re_ri += buffer->data[c][ri].re;
              output_im_ri += buffer->data[c][ri].im;
            }
          }

          output[i].re = (OUTPUT_SUM_TYPE) output_re/NUM_CHANS; // average
          output[i].im = (OUTPUT_SUM_TYPE) output_im/NUM_CHANS; // average
          if(i>0) {
            output[ri].re = (OUTPUT_SUM_TYPE) output_re_ri/NUM_CHANS; // average
            output[ri].im = (OUTPUT_SUM_TYPE) output_im_ri/NUM_CHANS; // average
          }
        }
    #endif

        // Todo: Add iFFT

        tmr :> end_time;
        int32_t cycles_taken = end_time-start_time-overhead_time;

#if CHECK_TIMING
#if TWOREALS
        printf("%d Point FFT processing of %d real sequences 'in place' in double buffer %x took %d cycles\n"
                ,N_FFT_POINTS,NUM_CHANS,buffer,cycles_taken);
#else
        printf("%d Point FFT processing of %d complex sequences 'in place' in double buffer %x took %d cycles\
           ↪ n"
                ,N_FFT_POINTS,NUM_CHANS,buffer,cycles_taken);
#endif
        if(cycles_taken>SAMPLE_PERIOD_CYCLES*N_FFT_POINTS) {
            printf("Timing Check ERROR: Max allowed cycles at Fs %d Hz is %d\n",SAMPLE_FREQ,
              ↪ SAMPLE_PERIOD_CYCLES*N_FFT_POINTS);
        } else {
            printf("Timing Check PASS: Max allowed cycles at Fs %d Hz is %d\n",SAMPLE_FREQ,
              ↪ SAMPLE_PERIOD_CYCLES*N_FFT_POINTS);
        }
#endif

#if PRINT_INPUTS_AND_OUTPUTS
        print_buffer(buffer);
```

```
#endif
        printf("Processed output signal\n");
        print_signal(output);

        break;
    }
    // fill the buffer with data
  }
}

/** Utility functions for signal generation **/
int32_t scaled_sin(q8_24 x) {
   q8_24 y = lib_dsp_math_sin(x);
#if INT16_BUFFERS
   y >>= 10; // convert to Q14
#else
#endif
   return y;
}
int32_t scaled_cos(q8_24 x) {
   q8_24 y = lib_dsp_math_cos(x);
#if INT16_BUFFERS
   y >>= 10; // convert to Q14
#else
#endif
   return y;
}
/****/

/**
 The displaying task takes the other end of the interface connection
 and its initial buffer as arguments. It also creates a movable pointer
 to that buffer.
**/

#define MAX_SAMPLE_PERIODS 1

void produce_samples(client interface bufswap_i filler,
        multichannel_sample_block_s * initial_buffer) {
  multichannel_sample_block_s * movable buffer = initial_buffer;
  timer tmr;
  int32_t t;
  static int32_t counter;
  int32_t done = 0;

  tmr :> t;
  /** The main loop of the display task first calls the 'swap' transaction,
      which synchronizes with the fill task and updates the 'buffer' pointer
      to the new swapped memory location. After that it calls an auxiliary
      'display' function to do the actual displaying. This function is
      application dependent and not defined here.
   **/
  while(!done) {
    //fill the next buffer
    for(int32_t a=0; a<NUM_SIGNAL_ARRAYS; a++) {
      // points per cycle. divide by power of two to ensure signals fit into the FFT window
      int32_t ppc = N_FFT_POINTS/(1<<a);
      printf("Points Per Cycle is %d\n", ppc);

      for(int32_t i=0; i<N_FFT_POINTS; i++) {
        // generate input signals

        // Equation: x = 2pi * i/ppc = 2pi * ((i%ppc) / ppc))
        q8_24 factor = ((i%ppc) << 24) / ppc; // factor is always < Q24(1)
        q8_24 x = lib_dsp_math_multiply(PI2_Q8_24, factor, 24);

#if TWOREALS
        buffer->data[a][i].re = scaled_sin(x);
        buffer->data[a][i].im = scaled_cos(x);
#else
        buffer->data[a][i].re = scaled_sin(x);
        buffer->data[a][i].im = 0;
#endif

      }
      // wait until next sample period
      tmr when timerafter(t+SAMPLE_PERIOD_CYCLES) :> t;
    }

    // swap buffers
    filler.swap(buffer);
    counter++;

    if(counter == MAX_SAMPLE_PERIODS) {
        exit(0);
    }

  }
}

/**
 The application runs both of these tasks in parallel using a 'par'
 statement. The two global buffers are passed into
 the two tasks:
**/

// make global to enforce 64 bit alignment
multichannel_sample_block_s buffer0;
multichannel_sample_block_s buffer1;

int main() {

  interface bufswap_i bufswap;
  par {
      produce_samples(bufswap, &buffer1);
      do_fft(bufswap, &buffer0);
  }
  return 0;
}
```

APPENDIX C - Correct Results Listings

This section includes the source code for all of the example programs.

C.1 Adaptive Filtering Functions

```
LMS 5
+0.003133 +0.096867
+0.009405 +0.090595
+0.010949 +0.089051
+0.012192 +0.087808
+0.009736 +0.090264
+0.009765 +0.090235
+0.009793 +0.090207
+0.009822 +0.090178
+0.009851 +0.090149
+0.009880 +0.090120
+0.009909 +0.090091
+0.009938 +0.090062
+0.009966 +0.090034
+0.009995 +0.090005
+0.010024 +0.089976
+0.010053 +0.089947
+0.010082 +0.089918
+0.010110 +0.089890
+0.010139 +0.089861
+0.010168 +0.089832
+0.010197 +0.089803
+0.010225 +0.089775
+0.010254 +0.089746
+0.010283 +0.089717
+0.010312 +0.089688
+0.010340 +0.089660
+0.010369 +0.089631
+0.010398 +0.089602
+0.010426 +0.089574
+0.010455 +0.089545
+0.010484 +0.089516
+0.010512 +0.089488
+0.010541 +0.089459
+0.010570 +0.089430
+0.010598 +0.089402

Normalized LMS 5
+0.003133 +0.096867
+0.010367 +0.089633
+0.012796 +0.087204
+0.014894 +0.085106
+0.013266 +0.086734
+0.014134 +0.085866
+0.014992 +0.085008
+0.015842 +0.084158
+0.016684 +0.083316
+0.017517 +0.082483
+0.018342 +0.081658
+0.019159 +0.080841
+0.019967 +0.080033
```

```
+0.020767 +0.079233
+0.021560 +0.078440
+0.022344 +0.077656
+0.023121 +0.076879
+0.023889 +0.076111
+0.024651 +0.075349
+0.025404 +0.074596
+0.026150 +0.073850
+0.026888 +0.073112
+0.027620 +0.072380
+0.028343 +0.071657
+0.029060 +0.070940
+0.029769 +0.070231
+0.030472 +0.069528
+0.031167 +0.068833
+0.031855 +0.068145
+0.032537 +0.067463
+0.033211 +0.066789
+0.033879 +0.066121
+0.034540 +0.065460
+0.035195 +0.064805
+0.035843 +0.064157
```

C.2 Fixed Coefficient Filtering Functions

```
FIR Filter Results
Dst[0] = 0.012100
Dst[1] = 0.026400
Dst[2] = 0.043000
Dst[3] = 0.062000
Dst[4] = 0.083500
Dst[5] = 0.107600
Dst[6] = 0.134400
Dst[7] = 0.164000
Dst[8] = 0.196500
Dst[9] = 0.232000
Dst[10] = 0.270600
Dst[11] = 0.312400
Dst[12] = 0.357500
Dst[13] = 0.406000
Dst[14] = 0.458000
Dst[15] = 0.513600
Dst[16] = 0.572900
Dst[17] = 0.636000
Dst[18] = 0.703000
Dst[19] = 0.774000
Dst[20] = 0.849100
Dst[21] = 0.928400
Dst[22] = 1.012000
Dst[23] = 1.100000
Dst[24] = 1.192500
Dst[25] = 1.289600
Dst[26] = 1.391400
Dst[27] = 1.498000
Dst[28] = 1.609500
Dst[29] = 1.726000
Dst[30] = 1.802500
Dst[31] = 1.879000
Dst[32] = 1.955500
Dst[33] = 2.032000
Dst[34] = 2.108500
Dst[35] = 2.185000
Dst[36] = 2.261500
Dst[37] = 2.338000
Dst[38] = 2.414500
Dst[39] = 2.491000
Dst[40] = 2.567500
Dst[41] = 2.644000
Dst[42] = 2.720500
Dst[43] = 2.797000
Dst[44] = 2.873500
Dst[45] = 2.950000
Dst[46] = 3.026500
Dst[47] = 3.103000
Dst[48] = 3.179500
Dst[49] = 3.256000

IIR Biquad Filter Results
Dst[0] = 0.012100
Dst[1] = 0.028094
Dst[2] = 0.048748
```

```
Dst[3] = 0.057639
Dst[4] = 0.065582
Dst[5] = 0.071627
Dst[6] = 0.077265
Dst[7] = 0.082561
Dst[8] = 0.087748
Dst[9] = 0.092869
Dst[10] = 0.097964
Dst[11] = 0.103045
Dst[12] = 0.108121
Dst[13] = 0.113194
Dst[14] = 0.118265
Dst[15] = 0.123336
Dst[16] = 0.128407
Dst[17] = 0.133477
Dst[18] = 0.138548
Dst[19] = 0.143618
Dst[20] = 0.148689
Dst[21] = 0.153759
Dst[22] = 0.158830
Dst[23] = 0.163900
Dst[24] = 0.168970
Dst[25] = 0.174041
Dst[26] = 0.179111
Dst[27] = 0.184182
Dst[28] = 0.189252
Dst[29] = 0.194323
Dst[30] = 0.199393
Dst[31] = 0.204463
Dst[32] = 0.209534
Dst[33] = 0.214604
Dst[34] = 0.219675
Dst[35] = 0.224745
Dst[36] = 0.229815
Dst[37] = 0.234886
Dst[38] = 0.239956
Dst[39] = 0.245027
Dst[40] = 0.250097
Dst[41] = 0.255168
Dst[42] = 0.260238
Dst[43] = 0.265308
Dst[44] = 0.270379
Dst[45] = 0.275449
Dst[46] = 0.280520
Dst[47] = 0.285590
Dst[48] = 0.290661
Dst[49] = 0.295731

Cascaded IIR Biquad Filter Results
Dst[0] = 0.000788
Dst[1] = 0.003924
Dst[2] = 0.012215
Dst[3] = 0.027035
Dst[4] = 0.048666
Dst[5] = 0.074798
Dst[6] = 0.103666
Dst[7] = 0.133224
Dst[8] = 0.162755
```

```
Dst[9] = 0.191477
Dst[10] = 0.219245
Dst[11] = 0.245924
Dst[12] = 0.271591
Dst[13] = 0.296325
Dst[14] = 0.320256
Dst[15] = 0.343501
Dst[16] = 0.366176
Dst[17] = 0.388383
Dst[18] = 0.410208
Dst[19] = 0.431725
Dst[20] = 0.452995
Dst[21] = 0.474067
Dst[22] = 0.494982
Dst[23] = 0.515771
Dst[24] = 0.536461
Dst[25] = 0.557073
Dst[26] = 0.577623
Dst[27] = 0.598124
Dst[28] = 0.618587
Dst[29] = 0.639019
Dst[30] = 0.659427
Dst[31] = 0.679817
Dst[32] = 0.700191
Dst[33] = 0.720554
Dst[34] = 0.740908
Dst[35] = 0.761255
Dst[36] = 0.781596
Dst[37] = 0.801932
Dst[38] = 0.822265
Dst[39] = 0.842595
Dst[40] = 0.862924
Dst[41] = 0.883250
Dst[42] = 0.903575
Dst[43] = 0.923899
Dst[44] = 0.944222
Dst[45] = 0.964545
Dst[46] = 0.984867
Dst[47] = 1.005188
Dst[48] = 1.025510
Dst[49] = 1.045831

Interpolation
INTERP taps=16 L=2
+0.003916 +0.007832
+0.005832 +0.009364
+0.002734 +0.013280
+0.010566 +0.015196
+0.012098 +0.012098
+0.016014 +0.019930
+0.017930 +0.021462
+0.014832 +0.025378
INTERP taps=24 L=3
+0.003916 +0.007832 +0.001916
+0.005448 +0.004734 +0.005832
+0.013280 +0.006650 +0.007364
+0.010182 +0.010566 +0.015196
+0.012098 +0.012098 +0.012098
```

```
+0.016014 +0.019930 +0.014014
+0.017546 +0.016832 +0.017930
+0.025378 +0.018748 +0.019462
INTERP taps=32 L=4
+0.003916 +0.007832 +0.001916 +0.001532
+0.000818 +0.011748 +0.009748 +0.003448
+0.002350 +0.008650 +0.013664 +0.011280
+0.004266 +0.010182 +0.010566 +0.015196
+0.012098 +0.012098 +0.012098 +0.012098
+0.016014 +0.019930 +0.014014 +0.013630
+0.012916 +0.023846 +0.021846 +0.015546
+0.014447 +0.020748 +0.025762 +0.023378
INTERP taps=40 L=5
+0.003916 +0.007832 +0.001916 +0.001532 -0.003098
+0.007832 +0.015664 +0.003832 +0.003064 -0.006196
+0.011748 +0.023496 +0.005748 +0.004595 -0.009295
+0.015664 +0.031329 +0.007664 +0.006127 -0.012393
+0.019580 +0.039161 +0.009580 +0.007659 -0.015491
+0.023496 +0.046993 +0.011496 +0.009191 -0.018589
+0.027412 +0.054825 +0.013412 +0.010723 -0.021688
+0.031329 +0.062657 +0.015329 +0.012254 -0.024786
INTERP taps=48 L=6
+0.003916 +0.007832 +0.001916 +0.001532 -0.003098 +0.003916
+0.011748 +0.009748 +0.003448 -0.001566 +0.000818 +0.011748
+0.013664 +0.011280 +0.000350 +0.002350 +0.008650 +0.013664
+0.015196 +0.008182 +0.004266 +0.010182 +0.010566 +0.015196
+0.012098 +0.012098 +0.012098 +0.012098 +0.012098 +0.012098
+0.016014 +0.019930 +0.014014 +0.013630 +0.009000 +0.016014
+0.023846 +0.021846 +0.015546 +0.010531 +0.012916 +0.023846
+0.025762 +0.023378 +0.012447 +0.014447 +0.020748 +0.025762
INTERP taps=56 L=7
+0.003916 +0.007832 +0.001916 +0.001532 -0.003098 +0.003916 +0.007832
+0.005832 +0.009364 -0.001182 +0.005448 +0.004734 +0.005832 +0.009364
+0.002734 +0.013280 +0.006650 +0.007364 +0.006266 +0.002734 +0.013280
+0.010566 +0.015196 +0.008182 +0.004266 +0.010182 +0.010566 +0.015196
+0.012098 +0.012098 +0.012098 +0.012098 +0.012098 +0.012098 +0.012098
+0.016014 +0.019930 +0.014014 +0.013630 +0.009000 +0.016014 +0.019930
+0.017930 +0.021462 +0.010916 +0.017546 +0.016832 +0.017930 +0.021462
+0.014832 +0.025378 +0.018748 +0.019462 +0.018364 +0.014832 +0.025378
INTERP taps=64 L=8
+0.003916 +0.007832 +0.001916 +0.001532 -0.003098 +0.003916 +0.007832 +0.001916
+0.005448 +0.004734 +0.005832 +0.009364 -0.001182 +0.005448 +0.004734 +0.005832
+0.013280 +0.006650 +0.007364 +0.006266 +0.002734 +0.013280 +0.006650 +0.007364
+0.010182 +0.010566 +0.015196 +0.008182 +0.004266 +0.010182 +0.010566 +0.015196
+0.012098 +0.012098 +0.012098 +0.012098 +0.012098 +0.012098 +0.012098 +0.012098
+0.016014 +0.019930 +0.014014 +0.013630 +0.009000 +0.016014 +0.019930 +0.014014
+0.017546 +0.016832 +0.017930 +0.021462 +0.010916 +0.017546 +0.016832 +0.017930
+0.025378 +0.018748 +0.019462 +0.018364 +0.014832 +0.025378 +0.018748 +0.019462

Decimation
DECIM taps=32 M=02
+0.003916 +0.013664 +0.012098 +0.023846 +0.027294 +0.028112 +0.037860 +0.036294
+0.048042 +0.051490 +0.052308 +0.062056 +0.060489 +0.072238 +0.075685 +0.076503
DECIM taps=32 M=03
+0.003916 +0.015196 +0.023846 +0.024196 +0.037860 +0.040210 +0.051490 +0.060140
+0.060489 +0.074154
DECIM taps=32 M=04
+0.003916 +0.012098 +0.027294 +0.037860 +0.048042 +0.052308 +0.060489 +0.075685
```

```
DECIM taps=32 M=05
+0.003916 +0.016014 +0.028112 +0.040210 +0.052308 +0.064405
DECIM taps=32 M=06
+0.003916 +0.023846 +0.037860 +0.051490 +0.060489
DECIM taps=32 M=07
+0.003916 +0.025762 +0.036294 +0.060140
DECIM taps=32 M=08
+0.003916 +0.027294 +0.048042 +0.060489
```

C.3 Math Functions

```
Test example for Math functions
===============================
Test Multiplication and Division
--------------------------------
Note: All calculations are done in Q8.24 format. That gives 7 digits of precision after the decimal point
Note: Maximum double representation of Q8.24 format: 127.99999994

Multiplication (11.31370850 x 11.31370850): 127.99999964

Multiplication (11.4 x 11.4). Will overflow!: -126.04000056

Saturated Multiplication (11.4 x 11.4): 127.99999994

Multiplication of small numbers (0.0005 x 0.0005): 0.00000024

Signed Division 1.12345600 / -128.00000000): -0.00877702

Signed Division -1.12345600 / -128.00000000): 0.00877702

Signed Division -1.12345600 / 127.99999990): -0.00877702

Signed Division 1.12345600 / 127.99999990): 0.00877702

Division 1.12345600 / 127.99999990): 0.00877696

Error report from test_multipliation_and_division:
Cases Error <= -3: 0
Cases Error == -2: 0
Cases Error == -1: 1
Cases Error == 0: 4
Cases Error == 1: 0
Cases Error == 2: 0
Cases Error >= 3: 0
Maximum Positive Error: 0
Maximum Negative Error: -1
Number of values checked: 5
Percentage of 0 Error: 80.00%

Test Roots
----------
Square Root (0.00000006) : 0.00024414
Square Root (0.00000018) : 0.00042284
Square Root (0.00000042) : 0.00064588
Square Root (0.00000089) : 0.00094551
Square Root (0.00000185) : 0.00135928
Square Root (0.00000376) : 0.00193775
Square Root (0.00000757) : 0.00275129
Square Root (0.00001520) : 0.00389856
Square Root (0.00003046) : 0.00551885
Square Root (0.00006098) : 0.00780863
Square Root (0.00012201) : 0.01104581
Square Root (0.00024408) : 0.01562303
Square Root (0.00048822) : 0.02209568
Square Root (0.00097650) : 0.03124899
Square Root (0.00195307) : 0.04419345
Square Root (0.00390619) : 0.06249946
Square Root (0.00781244) : 0.08838797
Square Root (0.01562494) : 0.12499970
Square Root (0.03124994) : 0.17677647
Square Root (0.06249994) : 0.24999982
Square Root (0.12499994) : 0.35355330
Square Root (0.24999994) : 0.49999988
Square Root (0.49999994) : 0.70710671
Square Root (0.99999994) : 0.99999994
Square Root (1.99999994) : 1.41421354
Square Root (3.99999994) : 1.99999994
Square Root (7.99999994) : 2.82842708
Square Root (15.99999994) : 3.99999994
Square Root (31.99999994) : 5.65685421
Square Root (63.99999994) : 7.99999994
Square Root (127.99999994) : 11.31370848
Square Root (255.99999994) : 15.99999994
Error report: lib_dsp_math_squareroot vs sqrt:
```

```
Cases Error <= -3: 0
Cases Error == -2: 0
Cases Error == -1: 24
Cases Error == 0: 8
Cases Error == 1: 0
Cases Error == 2: 0
Cases Error >= 3: 0
Maximum Positive Error: 0
Maximum Negative Error: -1
Number of values checked: 32
Percentage of 0 Error: 25.00%

Test Trigonometric Functions
----------------------------
Sine wave (one cycle from -Pi to +Pi) :
sin(-3.14159262) = -0.00000006
sin(-3.07876080) = -0.06279051
sin(-3.01592898) = -0.12533319
sin(-2.95309716) = -0.18738127
sin(-2.89026535) = -0.24868983
sin(-2.82743353) = -0.30901688
sin(-2.76460171) = -0.36812443
sin(-2.70176989) = -0.42577910
sin(-2.63893807) = -0.48175347
sin(-2.57610625) = -0.53582656
sin(-2.51327443) = -0.58778501
sin(-2.45044261) = -0.63742375
sin(-2.38761079) = -0.68454683
sin(-2.32477897) = -0.72896838
sin(-2.26194715) = -0.77051294
sin(-2.19911534) = -0.80901670
sin(-2.13628352) = -0.84432763
sin(-2.07345170) = -0.87630641
sin(-2.01061988) = -0.90482682
sin(-1.94778806) = -0.92977625
sin(-1.88495624) = -0.95105630
sin(-1.82212442) = -0.96858299
sin(-1.75929260) = -0.98228711
sin(-1.69646078) = -0.99211460
sin(-1.63362896) = -0.99802667
sin(-1.57079715) = -1.00000000
sin(-1.50796533) = -0.99802679
sin(-1.44513351) = -0.99211478
sin(-1.38230169) = -0.98228741
sin(-1.31946987) = -0.96858341
sin(-1.25663805) = -0.95105684
sin(-1.19380623) = -0.92977685
sin(-1.13097441) = -0.90482748
sin(-1.06814259) = -0.87630719
sin(-1.00531077) = -0.84432852
sin(-0.94247895) = -0.80901766
sin(-0.87964714) = -0.77051401
sin(-0.81681532) = -0.72896945
sin(-0.75398350) = -0.68454802
sin(-0.69115168) = -0.63742501
sin(-0.62831986) = -0.58778632
sin(-0.56548804) = -0.53582793
sin(-0.50265622) = -0.48175490
sin(-0.43982440) = -0.42578059
sin(-0.37699258) = -0.36812592
sin(-0.31416076) = -0.30901843
sin(-0.25132895) = -0.24869138
sin(-0.18849713) = -0.18738288
sin(-0.12566531) = -0.12533480
sin(-0.06283349) = -0.06279212
sin(-0.00000167) = -0.00000167
sin(0.06283015) = 0.06278884
sin(0.12566197) = 0.12533152
sin(0.18849379) = 0.18737954
sin(0.25132561) = 0.24868816
sin(0.31415743) = 0.30901521
sin(0.37698925) = 0.36812282
sin(0.43982106) = 0.42577755
sin(0.50265288) = 0.48175198
sin(0.56548470) = 0.53582513
sin(0.62831652) = 0.58778363
sin(0.69114834) = 0.63742238
sin(0.75398016) = 0.68454558
sin(0.81681198) = 0.72896719
sin(0.87964380) = 0.77051187
sin(0.94247562) = 0.80901569
sin(1.00530744) = 0.84432673
sin(1.06813926) = 0.87630558
sin(1.13097107) = 0.90482605
sin(1.19380289) = 0.92977560
sin(1.25663471) = 0.95105577
sin(1.31946653) = 0.96858257
sin(1.38229835) = 0.98228681
sin(1.44513017) = 0.99211437
sin(1.50796199) = 0.99802655
sin(1.57079381) = 1.00000000
sin(1.63362563) = 0.99802685
sin(1.69645745) = 0.99211502
sin(1.75928926) = 0.98228770
sin(1.82212108) = 0.96858382
sin(1.88495290) = 0.95105737
sin(1.94778472) = 0.92977750
sin(2.01061654) = 0.90482825
sin(2.07344836) = 0.87630802
sin(2.13628018) = 0.84432942
sin(2.19911200) = 0.80901867
sin(2.26194382) = 0.77051508
sin(2.32477564) = 0.72897065
sin(2.38760746) = 0.68454927
sin(2.45043927) = 0.63742632
sin(2.51327109) = 0.58778775
sin(2.57610291) = 0.53582942
sin(2.63893473) = 0.48175639
sin(2.70176655) = 0.42578214
sin(2.76459837) = 0.36812752
sin(2.82743019) = 0.30902004
sin(2.89026201) = 0.24869305
sin(2.95309383) = 0.18738455
sin(3.01592565) = 0.12533653
sin(3.07875746) = 0.06279385
sin(3.14158928) = 0.00000340
Error report: lib_dsp_math_sin vs sin:
Cases Error <= -3: 0
Cases Error == -2: 0
Cases Error == -1: 10
Cases Error == 0: 88
Cases Error == 1: 3
Cases Error == 2: 0
Cases Error >= 3: 0
Maximum Positive Error: 1
Maximum Negative Error: -1
Number of values checked: 101
Percentage of 0 Error: 87.13%

Cosine wave (one cycle from -Pi to +Pi) :
cos(-3.14159262) = -1.00000000
cos(-3.07876080) = -0.99802673
cos(-3.01592898) = -0.99211466
cos(-2.95309716) = -0.98228729
cos(-2.89026535) = -0.96858317
cos(-2.82743353) = -0.95105654
cos(-2.76460171) = -0.92977655
cos(-2.70176989) = -0.90482712
cos(-2.63893807) = -0.87630677
cos(-2.57610625) = -0.84432805
cos(-2.51327443) = -0.80901718
cos(-2.45044261) = -0.77051347
cos(-2.38761079) = -0.72896892
cos(-2.32477897) = -0.68454742
cos(-2.26194715) = -0.63742435
cos(-2.19911534) = -0.58778566
cos(-2.13628352) = -0.53582722
cos(-2.07345170) = -0.48175418
cos(-2.01061988) = -0.42577982
cos(-1.94778806) = -0.36812514
cos(-1.88495624) = -0.30901760
cos(-1.82212442) = -0.24869055
cos(-1.75929260) = -0.18738204
cos(-1.69646078) = -0.12533396
cos(-1.63362896) = -0.06279129
cos(-1.57079715) = -0.00000083
cos(-1.50796533) = 0.06278962
cos(-1.44513351) = 0.12533236
cos(-1.38230169) = 0.18738037
cos(-1.31946987) = 0.24868894
cos(-1.25663805) = 0.30901605
cos(-1.19380623) = 0.36812359
cos(-1.13097441) = 0.42577833
cos(-1.06814259) = 0.48175269
cos(-1.00531077) = 0.53582585
cos(-0.94247895) = 0.58778429
cos(-0.87964714) = 0.63742304
cos(-0.81681532) = 0.68454617
cos(-0.75398350) = 0.72896773
cos(-0.69115168) = 0.77051240
cos(-0.62831986) = 0.80901617
cos(-0.56548804) = 0.84432715
cos(-0.50265622) = 0.87630600
cos(-0.43982440) = 0.90482646
cos(-0.37699258) = 0.92977595
cos(-0.31416076) = 0.95105606
cos(-0.25132895) = 0.96858275
cos(-0.18849713) = 0.98228693
cos(-0.12566531) = 0.99211448
cos(-0.06283349) = 0.99802661
cos(-0.00000167) = 1.00000000
cos(0.06283015) = 0.99802679
cos(0.12566197) = 0.99211490
cos(0.18849379) = 0.98228759
cos(0.25132561) = 0.96858358
cos(0.31415743) = 0.95105708
cos(0.37698925) = 0.92977720
cos(0.43982106) = 0.90482789
cos(0.50265288) = 0.87630761
cos(0.56548470) = 0.84432900
cos(0.62831652) = 0.80901819
cos(0.69114834) = 0.77051455
cos(0.75398016) = 0.72897005
cos(0.81681198) = 0.68454868
cos(0.87964380) = 0.63742566
cos(0.94247562) = 0.58778703
cos(1.00530744) = 0.53582871
cos(1.06813926) = 0.48175567
cos(1.13097107) = 0.42578137
cos(1.19380289) = 0.36812675
cos(1.25663471) = 0.30901927
cos(1.31946653) = 0.24869221
cos(1.38229835) = 0.18738371
cos(1.44513017) = 0.12533569
cos(1.50796199) = 0.06279302
cos(1.57079381) = 0.00000256
cos(1.63362563) = -0.06278795
cos(1.69645745) = -0.12533063
cos(1.75928926) = -0.18737870
cos(1.82212108) = -0.24868727
cos(1.88495290) = -0.30901438
cos(1.94778472) = -0.36812198
cos(2.01061654) = -0.42577672
cos(2.07344836) = -0.48175120
cos(2.13628018) = -0.53582436
cos(2.19911200) = -0.58778292
cos(2.26194382) = -0.63742173
cos(2.32477564) = -0.68454492
cos(2.38760746) = -0.72896653
cos(2.45043927) = -0.77051127
cos(2.51327109) = -0.80901521
cos(2.57610291) = -0.84432626
cos(2.63893473) = -0.87630516
cos(2.70176655) = -0.90482569
cos(2.76459837) = -0.92977530
cos(2.82743019) = -0.95105553
cos(2.89026201) = -0.96858233
cos(2.95309383) = -0.98228663
cos(3.01592565) = -0.99211425
cos(3.07875746) = -0.99802649
cos(3.14158928) = -1.00000000
Error report: lib_dsp_math_cos vs cos:
Cases Error <= -3: 0
Cases Error == -2: 0
Cases Error == -1: 11
Cases Error == 0: 70
Cases Error == 1: 20
Cases Error == 2: 0
Cases Error >= 3: 0
Maximum Positive Error: 1
Maximum Negative Error: -1
Number of values checked: 101
Percentage of 0 Error: 69.31%

Test lib_dsp_math_atan
atan(-127.99999994) = -1.56298399
atan(-63.99999994) = -1.55517262
atan(-31.99999994) = -1.53955650
atan(-15.99999994) = -1.50837749
atan(-7.99999994) = -1.44644135
atan(-3.99999994) = -1.32581764
atan(-1.99999994) = -1.10714871
atan(-0.99999994) = -0.78539813
atan(-0.49999994) = -0.46364754
atan(-0.24999994) = -0.24497861
atan(-0.12499994) = -0.12435496
atan(-0.06249994) = -0.06241876
atan(-0.03124994) = -0.03123975
atan(-0.01562494) = -0.01562369
atan(-0.00781244) = -0.00781226
atan(-0.00390619) = -0.00390619
atan(-0.00195307) = -0.00195307
atan(-0.00097650) = -0.00097650
atan(-0.00048822) = -0.00048822
atan(-0.00024408) = -0.00024408
atan(-0.00012201) = -0.00012201
atan(-0.00006098) = -0.00006098
atan(-0.00003046) = -0.00003046
atan(-0.00001520) = -0.00001520
atan(-0.00000757) = -0.00000757
atan(-0.00000376) = -0.00000376
atan(-0.00000185) = -0.00000185
atan(-0.00000089) = -0.00000089
atan(-0.00000042) = -0.00000042
atan(-0.00000018) = -0.00000018
atan(-0.00000006) = -0.00000006
atan(0.00000000) = 0.00000000
atan(0.00000006) = 0.00000006
atan(0.00000018) = 0.00000018
atan(0.00000042) = 0.00000042
atan(0.00000089) = 0.00000089
atan(0.00000185) = 0.00000185
atan(0.00000376) = 0.00000376
atan(0.00000757) = 0.00000757
atan(0.00001520) = 0.00001520
atan(0.00003046) = 0.00003046
atan(0.00006098) = 0.00006098
atan(0.00012201) = 0.00012201
atan(0.00024408) = 0.00024408
atan(0.00048822) = 0.00048822
atan(0.00097650) = 0.00097650
atan(0.00195307) = 0.00195307
atan(0.00390619) = 0.00390619
atan(0.00781244) = 0.00781226
atan(0.01562494) = 0.01562369
atan(0.03124994) = 0.03123975
atan(0.06249994) = 0.06241876
atan(0.12499994) = 0.12435496
atan(0.24999994) = 0.24497861
atan(0.49999994) = 0.46364754
atan(0.99999994) = 0.78539813
atan(1.99999994) = 1.10714871
atan(3.99999994) = 1.32581764
atan(7.99999994) = 1.44644135
atan(15.99999994) = 1.50837749
atan(31.99999994) = 1.53955650
atan(63.99999994) = 1.55517262
atan(127.99999994) = 1.56298399
Error report from lib_dsp_math_atan:
Cases Error <= -3: 0
Cases Error == -2: 0
Cases Error == -1: 0
Cases Error == 0: 63
Cases Error == 1: 0
Cases Error == 2: 0
Cases Error >= 3: 0
Maximum Positive Error: 0
Maximum Negative Error: 0
Number of values checked: 63
Percentage of 0 Error: 100.00%
```

C.4 Matrix Functions

```
Matrix negation: R = -X
-0.110000, -0.120000, -0.130000
-0.210000, -0.220000, -0.230000
-0.310000, -0.320000, -0.330000
Matrix / scalar addition: R = X + a
2.110000, 2.120000, 2.130000
2.210000, 2.220000, 2.230000
2.310000, 2.320000, 2.330000
Matrix / scalar multiplication: R = X + a
0.220000, 0.240000, 0.260000
0.420000, 0.440000, 0.460000
0.620000, 0.640000, 0.660000
Matrix / matrix addition: R = X + Y
0.520000, 0.540000, 0.560000
0.720000, 0.740000, 0.760000
0.920000, 0.940000, 0.960000
Matrix / matrix subtraction: R = X - Y
-0.300000, -0.300000, -0.300000
-0.300000, -0.300000, -0.300000
-0.300000, -0.300000, -0.300000
Matrix / matrix multiplication: R = X * Y
0.185600, 47.702803, -75.889630
0.338600, 87.026818, 7.217119
0.491600, 126.350802, 90.316039
Matrix transposition
0.110000, 0.210000, 0.310000
0.120000, 0.220000, 0.320000
0.130000, 0.230000, 0.330000
```

C.5 Statistics Functions

```
Vector Mean = 0.355000
Vector Power (sum of squares) = 7.342500
Vector Root Mean Square = 0.383210
Vector Dot Product = 0.345500
```

C.6 Vector Functions

```
Minimum location = 0
Minimum = 0.110000
Maximum location = 49
Maximum = 0.600000
Vector Negate Result
Dst[0] = -0.110000
Dst[1] = -0.120000
Dst[2] = -0.130000
Dst[3] = -0.140000
Dst[4] = -0.150000
Vector Absolute Result
Dst[0] = 0.110000
Dst[1] = 0.120000
Dst[2] = 0.130000
Dst[3] = 0.140000
Dst[4] = -0.150000
Vector / scalar addition Result
Dst[0] = 2.110000
Dst[1] = 2.120000
Dst[2] = 2.130000
Dst[3] = 2.140000
Dst[4] = 2.150000
Vector / scalar multiplication Result
Dst[0] = 0.220000
Dst[1] = 0.240000
Dst[2] = 0.260000
Dst[3] = 0.280000
Dst[4] = 0.300000
Vector / vector addition Result
Dst[0] = 0.620000
Dst[1] = 0.640000
Dst[2] = 0.660000
Dst[3] = 0.680000
Dst[4] = 0.700000
Vector / vector subtraction Result
Dst[0] = -0.400000
Dst[1] = -0.400000
Dst[2] = -0.400000
Dst[3] = -0.400000
Dst[4] = -0.400000
Vector / vector multiplication Result
Dst[0] = 0.056100
Dst[1] = 0.062400
Dst[2] = 0.068900
Dst[3] = 0.075600
Dst[4] = 0.082500
Vector multiplication and scalar addition Result
Dst[0] = 2.056100
Dst[1] = 2.062400
Dst[2] = 2.068900
Dst[3] = 2.075600
Dst[4] = 2.082500
Vector / Scalar multiplication and vector addition Result
Dst[0] = 0.730000
Dst[1] = 0.760000
Dst[2] = 0.790000
```

```
Dst[3] = 0.820000
Dst[4] = 0.850000
Vector / Scalar multiplication and vector subtraction Result
Dst[0] = -0.453900
Dst[1] = -0.457600
Dst[2] = -0.461100
Dst[3] = -0.464400
Dst[4] = -0.467500
Vector / Vector multiplication and vector addition Result
Dst[0] = 0.610370
Dst[1] = 0.620370
Dst[2] = 0.630370
Dst[3] = 0.640370
Dst[4] = 0.650370
Vector / Vector multiplication and vector subtraction Result
Dst[0] = -0.553900
Dst[1] = -0.557600
Dst[2] = -0.561100
Dst[3] = -0.564400
Dst[4] = -0.567500
```