DSP Library User Guide for PIC32A Microcontrollers

Introduction

This document outlines the DSP library for PIC32A MCU devices, which offers a suite of pre-optimized software routines tailored to streamline digital signal processing tasks. This library delivers efficient implementations of intricate mathematical functions and algorithms, empowering developers to boost the performance and capabilities of PIC32A MCU devices in applications, such as advanced sensing and control, power conversion, audio processing, telecommunications and motor control. The DSP Library currently supports a total of 52 functions.

Table of Contents

1. DSP Library for PIC32A MCU

1.1. C Code Applications

The DSP library for PIC32A devices is included with the XC32 compiler. Within the compiler install directory, the library and its related files can be found at:

- *pic32a\lib\libdsp-elf.a* – DSP library/archive file
- *pic32a\src\libdsp.zip* – zipped source code for library functions, including a batch file to rebuild the library
- *pic32a\support\generic\h\dsp.h* – header file for DSP library

1.2. Using the DSP Library

Read the following document sections before using the DSP library in specific user or conventional applications.

- Building with the DSP Library
- Memory Models
- DSP Library Function Calling Convention
- Data Types
- Data Memory Usage
- CORCON Register Usage
- Overflow and Saturation Handling
- Integrating with Interrupts and an RTOS
- Rebuilding the DSP Library
- DSP Library Functions

1.2.1. Building with the DSP Library

Building an application which utilizes the DSP Library requires only two files: ***dsp.h*** and ***libdsp-elf.a***. *dsp.h* is a header file which provides all the function prototypes, *#defines* and *typedefs* used by the library. *libdsp-elf.a* is the archived library file which contains all the individual object files for each library function. (See the *"MPLAB® XC32 Libraries Reference Manual for PIC32A MCU"*, DS50003838, for more on ELF-specific libraries.)

When compiling an application, ***dsp.h*** must be referenced (using #include) by all source files which call a function in the DSP Library or use its symbols or typedefs. When linking an application, ***libdsp-elf.a*** must be provided as an input to the linker (using the ***--library dsp*** or ***-ldsp*** linker switch), so the functions used by the application be linked to the project.

The linker will place the functions of the DSP library into a special text section named *.libdsp*. This can be seen in the map file generated by the linker.

1.2.2. Memory Models

The DSP Library is built using the small code and small data memory models to minimize the library footprint. Some functions within the DSP library are written in C and utilize the compiler's floating-point library. Consequently, the compiler linker script files position the .libm and .libdsp text sections adjacent to each other. This ensures that the DSP library can safely call the `RCALL` instruction to invoke the necessary floating-point routines from the floating-point library.

1.2.3. DSP Library Function Calling Convention

All the object modules within the DSP Library are compliant with the C compatibility guidelines for the PIC32A devices. The guidelines follow the function call conventions documented in the Function call convention section of "*MPLAB® XC32 C Compiler User's Guide for PIC32A MCU*", DS50003831.

1.2.4. Data Types

The operations provided by the DSP Library have been designed to take advantage of the DSP instruction set and architectural features of the PIC32A devices. Therefore, most operations are computed using 1.31 fixed-point arithmetic.

The DSP Library defines a fractional type from an integer type:

```
#ifndef fractional

     typedef int fractional;

#endif
```

The 1.31 fixed-point data are represented with one sign bit and 31 fractional bits, which is commonly known as the 1.31 fixed-point format. The results of functions which use the multiplier are computed using the 72-bit accumulator with 9.63 arithmetic. This format comprises nine sign/magnitude bits and 63 fractional bits, providing space for extra computation beyond the range of -1.00 to approximately +1.00 offered by the 1.31 format. When these functions return a result, they revert to a fractional data type in the 1.31 format.

The use of fractional arithmetic has some constraints on the allowable set of values to be input to a particular function. However, several functions perform implicit scaling to the input data and/or output results, which may decrease the resolution of the output values (when compared to a floating-point implementation).

A subset of operations in the DSP Library, which requires a higher degree of numerical resolution, does operate in floating-point arithmetic. Nevertheless, the results of these operations are transformed into fractional values for integration with the application. The only exception to this is the *MatrixInvert* function which computes the inversion of a floating-point matrix in floating-point arithmetic and provides the results in a floating-point format.

1.2.5. Data Memory Usage

The DSP Library performs no allocation of RAM and leaves this task to the users. If the users do not allocate and align the data memory properly, undesired results will occur during the function execution. In addition, to minimize execution time, the DSP Library will do no checking on the provided function arguments (including pointers to data memory) to determine if they are valid.

Most functions accept data pointers as function arguments, which contain the data to be operated on and, typically, the location to store the result. For convenience, most functions in the DSP Library expect their input arguments to be allocated in the default RAM memory space (X-Data or Y-Data) and the output to be stored back into the default RAM memory space. However, the computational intensive functions require that some operands reside in X-Data and Y-Data, so the operation can take advantage of the dual data fetch capability of the PIC32A architecture.

However, the architecture does not mandate that specific operands be located in either the X or Y-data space. For a few computationally intensive functions, placing them in the same data space (whether X or Y) will result in sequential operand access, therefore, affecting data fetch efficiency.

1.2.6. CORCON Register Usage

Many functions of the DSP Library place the device into a special operating mode by modifying the CORCON register. On the entry of these functions, the CORCON register is pushed to the stack. It is then modified to correctly perform the desired operation, and lastly, the CORCON register is popped

from the stack to preserve its original value. This mechanism allows the library to execute without disrupting the CORCON setting.

The CORCON register is configured to place the target device into the following operational mode.

- DSP multipliers are configured to utilize signed fractional data.
- Accumulator saturation is enabled for Accumulator A and Accumulator B.
- Saturation mode is set to 9.63 saturation (Super saturation).
- Data Space Write saturation is enabled.
- Convergent (unbiased) rounding is enabled.

For a detailed explanation of the CORCON register and its effects, refer to the respective device data sheets.

1.2.7. Overflow and Saturation Handling

The DSP Library performs most computations using 9.63 saturation but must store the output of the function in 1.31 format. If the accumulator in use saturates during the course of operation, the corresponding saturation bit (SA or SB) in the STATUS register will be set. This bit will stay set until it is cleared. This allows the inspection of SA or SB after the function executes and to determine if action should be taken to scale the input data to the function.

Similarly, if a computation performed with the accumulator results in an overflow, the corresponding overflow bit (OA or OB) in the STATUS register will be set. Unlike the SA and SB status bits, OA and OB will not stay set until they are cleared. These bits are updated each time an operation is executed by using the accumulator. If exceeding this specified range marks an important event, it is recommended to enable the Accumulator Overflow Trap via the OVATE, OVBTE and COVTE bits in the INTCON4 register. This will have the effect of generating an Arithmetic Math Error Trap as soon as the Overflow condition occurs to then take the required action.

1.2.8. Integrating with Interrupts and an RTOS

The DSP Library may easily be integrated into an application which utilizes interrupts or an RTOS with certain guidelines. To minimize execution time, the DSP Library utilizes REPEAT loops, Modulo Addressing and Bit-Reversed Addressing. Each of these components is a finite hardware resource on the PIC32A device, and the background code must consider the use of each resource when disrupting execution of a DSP Library function.

When integrating with the DSP Library, make sure to examine the Function Profile of each function description to determine which resources are used. If a library function is to be interrupted, it shall be the user's responsibility to save and restore the contents of all registers used by the function, including the state of the REPEAT and special addressing hardware. This also includes saving and restoring the contents of the CORCON and Status registers.

1.2.9. Rebuilding the DSP Library

To build the DSP library, extract the source files in the *(Compiler Install Path)\src\libdsp.zip* path; this creates a sub-folder called *libdsp*. Execute the *makedsplib.bat* file within this folder to rebuild the library archives. This creates two folders called "*lib*" and "*obj*". Once the batch file execution is completed, the rebuild *libdsp.a* file can be found in the "*lib*" folder.

The MPLAB XC32 compiler is required to rebuild the DSP library, and its installation path must be added to Windows® or Linux® environment variables.

1.2.10. DSP Library Functions

The DSP library supports the functional modules listed below:

- Vector Functions
- Window Functions

- Matrix Functions
- Filtering Functions
- Transform Functions
- Control Functions
- Conversion Functions

2. Vector Functions

Function	Description
VectorAdd	Adds the value of each element in the source one vector with its counterpart in the source two vector and places the result in the destination vector.
VectorConvolve	Computes the convolution between two source vectors and stores the result in a destination vector.
VectorCopy	Copies the elements of the source vector into the beginning of an (already existing) destination vector.
VectorCorrelate	Computes the correlation between two source vectors and stores the result in a destination vector.
VectorDotProduct	Computes the sum of the products between corresponding elements of the source one and source two vectors.
VectorMax	Returns the value and index of the last element in the source vector whose value is greater than or equal to any previous vector element.
VectorMin	Returns the value and index of the last element in the source vector whose value is lesser than or equal to any previous vector element.
VectorMultiply	Multiplies the value of each element in the source one vector with its counterpart in the source two vector and places the result in the corresponding element of destination vector.
VectorNegate	Negates (changes the sign of) the values of the elements in the source vector and places them in the destination vector.
VectorPower	Computes the power of a source vector as the sum of the squares of its elements.
VectorScale	Scales (multiplies) the values of all the elements in the source vector by a scale value and places the result in the destination vector.
VectorSubtract	Subtracts the value of each element in the source one vector with its counterpart in the source two vector and places the result in the destination vector.
VectorZeroPad	Copies the contents of the source vector into the beginning of the destination vector, and then fills the remaining positions—specified by *numZeros* — with zeros to complete the destination vector.

2.1. Fractional Vector Operations

A fractional vector is a collection of numerical values, the vector elements, allocated contiguously in memory, with the first element at the lowest memory address. One word of memory is used to store the value of each element, and this quantity must be interpreted as a fractional number represented in 1.31 data format.

The pointer addressing the first element of the vector serves as a handle, granting access to each value within the vector. This address of the first element is known as the base address (BA) of the vector. Given that each vector element is 32-bits (for 1.31 fractional data), the last 2-bits of the base address *must* be zero.

The one-dimensional arrangement of a vector accommodates to the memory storage model of the device, so that the n^{th} element of an N-element vector can be accessed from the vector's base address as:

$BA + 4*(n - 1)$, for $1 \leq n \leq N$.

Unary and binary fractional vector operations are implemented in this library. The operand vector in an unary operation is called the source vector. In a binary operation, the first operand is referred to as the source one vector and the second as the source two vector. Each operation applies some computation to one or several elements of the source vector(s). Some operations produce a result which is a scalar value (also to be interpreted as a 1.31 fractional number), while other operations

produce a result which is a vector. When the result is also a vector, this is referred to as the destination vector.

Some operations, that result in a vector, allow computation in place. This means the results of the operation are placed back into the source vector (or the source one vector for binary operations). In this case, the destination vector is said to (physically) replace the source (one) vector. If an operation can be computed in place, it is indicated as such in the comments provided with the function description.

For some binary operations, the two operands can be the same (physical) source vector, which means the operation is applied to the source vector and itself. If this type of computation is possible for a given operation, it is indicated as self-applicable in the comments provided with the function description.

Some operations can be both self-applicable and computed in place.

All the fractional vector operations in this library take as an argument the cardinality (number of elements) of the operand vector(s). Based on the value of this argument, the following assumptions are made:

1. The sum of sizes of all the vectors involved in a particular operation falls within the range of available data memory for the target device.
2. In the case of binary operations, the cardinalities of both operand vectors *must* obey the rules of vector algebra (specifically for the *VectorConvolve* and *VectorCorrelate* functions).
3. The destination vector *must* be large enough to accept the results of an operation.

2.2. Additional Remarks

The description of the functions limits its scope of what could be considered the regular usage of these operations. However, since no boundary checking is performed during computation of these functions, operations and its results are interpreted to fit specific needs.

For instance, while computing the *VectorMax* function, the length of the source vector could be greater than *numElems*. In this case, the function would be used to find the maximum value only among the first *numElems* elements of the source vector.

Similarly, if the requirement is to replace *numElems* elements of a destination vector located between N and N+*numElems*-1, with *numElems* elements from a source vector located between elements M and M+*numElems*-1, then, the *VectorCopy* function could be used as follows:

```
fractional* dstV[DST_ELEMS] = {...};
fractional* srcV[SRC_ELEMS] = {...};

int n = NUM_ELEMS;
int N = N_PLACE;   /* NUM_ELEMS+N ≤ DST_ELEMS */
int M = M_PLACE;   /* NUM_ELEMS+M ≤ SRC_ELEMS */

fractional* dstVector = dstV+N;
fractional* srcVector = srcV+M;

dstVector = VectorCopy (n, dstVector, srcVector);
```

Also in this context, the *VectorZeroPad* function can operate in place, where *dstV = srcV, numElems* is the number of elements at the beginning of the source vector to preserve, and *numZeros* is the number of elements at the vector tail to set to zero. Other possibilities can be exploited from the fact that no boundary checking is performed.

2.3. Functions

2.3.1. VectorAdd

Description

`VectorAdd` adds the value of each element in the source one vector with its counterpart in the source two vector and places the result in the destination vector.

Prototype

```
fractional* VectorAdd (int numElems, fractional* dstV, fractional* srcV1,
fractional* srcV2);
```

Arguments

Parameters	Description
numElems	Number of elements in the source vectors
dstV	Pointer to the destination vector
srcV1	Pointer to the source one vector
srcV2	Pointer to the source two vector

Returns

Pointer to the base address of the destination vector.

Remarks

If the absolute value of *srcV1[n]* + *srcV2[n]* is larger than the maximum value of 1.31 fractional datatype, this operation results in saturation for the n^{th} element.

This function can be computed in place.

This function can be self-applicable.

Source File

- `vadd_aa.s`

Function Profile

Device	Program Words	Cycles
PIC32A	11	36 + 4*(*numElems*)

System resource usage

- *W0...W4* - used, not restored
- *ACCA* - used, not restored
- *CORCON* - saved, used, restored
- *REPEAT* instruction(s) usage – None

2.3.2. VectorConvolve

Description

`VectorConvolve` computes the convolution between two source vectors and stores the result in a destination vector. The result is computed as follows:

$$y[n] = \sum_{k=0}^{n} x[k] * h[n-k] \quad , for\ 0 \le n \le N + M - 1$$

Where,

x[k] = source one vector of size *N*,

h[k] = source two vector of size *M* (with *M* < *N*).

Prototype

```
fractional* VectorConvolve (int numElems1,int numElems2,fractional* dstV,
fractional* srcV1,fractional* srcV2);
```

Arguments

Parameters	Description
numElems1	Number of elements in source one vector
numElems2	Number of elements in source two vector
dstV	Pointer to the destination vector
srcV1	Pointer to the source one vector
srcV2	Pointer to the source two vector

Returns

Pointer to the base address of the destination vector.

Remarks

The number of elements in the source two vector *must* be less than or equal to the number of elements in the source one vector.

The destination vector *must* already exist, with exactly *numElems1*+*numElems2-1* number of elements.

This function can be self-applicable.

Source File

- vcon_aa.s

Function Profile

Device	Program Words	Cycles
PIC32A	37	For N = *numElems1*, and M = *numElems2*, $32 + 17M + 2\sum_{m=1}^{M} m + (N - M)(7 + M)$ *for* $M < N$ $32 + 17M + 2\sum_{m=1}^{M} m$ *for* $M = N$

System resource usage

- *W0... W7* - used, not restored
- *W8...W9* - saved, used, restored
- *ACCA* - used, not restored
- *CORCON* - saved, used, restored
- Two *REPEAT* instruction(s)

2.3.3. VectorCopy

Description

VectorCopy copies the elements of the source vector into the beginning of an (already existing) destination vector, so that:

dstV[n] = srcV[n], 0 ≤ n < numElems

Prototype

```
fractional* VectorCopy (int numElems, fractional* dstV, fractional* srcV);
```

Arguments

Parameters	Description
numElems	Number of elements in the source vectors
dstV	Pointer to the destination vector
srcV1	Pointer to the source one vector

Returns

Pointer to the base address of the destination vector.

Remarks

The destination vector must already exist. The size of the destination vector must be greater than or equal to *numElems*.

This function can be computed in place. See Additional Remarks for comments on this mode of operation.

Source File

- vcopy_aa.s

Function Profile

Cycle counts for PIC32A	
Source Vector Size	**Cycles**
32	68
64	122
128	228
256	442
512	868
1024	1722
2048	3428

System resource usage

- *W0...W3* - used, not restored
- One *REPEAT* instruction(s)

2.3.4. VectorCorrelate

Description

VectorCorrelate computes the correlation between two source vectors and stores the result in a destination vector. The result is computed as follows:

$$y[n] = \sum_{k=0}^{n} x[k] * h[k-n] \quad , for\ 0 \le n \le N + M - 1$$

Prototype

```
fractional* VectorCorrelate (int numElems1, int numElems2, fractional* dstV,
fractional* srcV1, fractional* srcV2);
```

Arguments

Parameters	Description
numElems1	Number of elements in the source one vector (*N*)
numElems2	Number of elements in the source two vector (*M*, with $M \leq N$)
dstV	Pointer to the destination vector (of size *N+M-1*)
srcV1	Pointer to the source one vector
srcV2	Pointer to the source two vector

Returns

Pointer to the base address of the destination vector.

Remarks

The number of elements in the source two vector must be less than or equal to the number of elements in the source one vector.

The destination vector must already exist, with exactly *numElems1+numElems2-1* number of elements.

This function can be self-applicable.

This function uses *VectorConvolve*.

Source File

- vcor_aa.s

Function Profile

Device	Program Words	Cycles
PIC32A	12	$28 + \lfloor M/2 \rfloor \times 3$

Note :

1. The above-mentioned program word and cycle counts pertain solely to *VectorCorrelate*. However, as this function inherently utilizes *VectorConvolve*, the respective counts for *VectorConvolve* must also be considered.
2. In the description of *VectorConvolve*, the number of cycles reported includes four cycles of C-function call overhead. Thus, the number of actual cycles from *VectorConvolve* to add to *VectorCorrelate* is four less than whatever number is reported for a stand-alone *VectorConvolve*.

System resource usage

The below system resource usages are excluded from those of the *VectorConvolve* function.

- *W0...W7* - used, not restored
- *REPEAT* instruction(s) usage – None

2.3.5. VectorDotProduct

Description

VectorDotProduct computes the sum of the products between the corresponding elements of the source one and source two vectors.

Prototype

```
fractional VectorDotProduct (int numElems, fractional* srcV1,
fractional* srcV2);
```

Arguments

Parameters	Description
numElems	Number of elements in the source vectors
srcV1	Pointer to the source one vector
srcV2	Pointer to the source two vector

Returns

Value of the sum of products.

Remarks

If the absolute value of *sum-of-products* is larger than the maximum value of 1.31 fractional data type, the operation results in saturation.

This function can be computed in place.

This function can be self-applicable.

Source File

- `vdot_aa.s`

Function Profile

Device	Program Words	Cycles
PIC32A	11	`32 + numElems` - If *srcV2* in y-memory and *srcV1* in x-memory. `32 + 2*numElems` - If both *srcV2* and *srcV1* in x-memory.

System resource usage

- *W0...W4* - used, not restored
- *ACCA* - used, not restored
- *CORCON* - saved, used, restored
- *REPEAT* instruction(s) usage – 1

2.3.6. VectorMax

Description

`VectorMax` returns the value and index of the last element in the source vector whose value is greater than or equal to any previous vector element.

Prototype

```
fractional VectorMax (int numElems, fractional* srcV, int* maxIndex);
```

Arguments

Parameters	Description
numElems	Number of elements in the source vector
srcV	Pointer to the source vector

VectorMax (continued)	
Parameters	**Description**
maxIndex	Pointer to the holder for the index of (last) maximum element

Returns

Maximum value in the vector.

Remarks

If *srcV*[i] = *srcV*[j] = *maxVal*, and i < j, then **maxIndex* = j.

Source File

- vmax_aa.s

Function Profile

Device	Program Words	Cycles
PIC32A	10	32 + 4*(*numElems* - 1),

System resource usage

- *W0...W6* - used, not restored
- *REPEAT* instruction(s) usage – None

2.3.7. VectorMin

Description

VectorMin returns the value and index of the last element in the source vector whose value is lesser than or equal to any previous vector element.

Prototype

```
fractional VectorMin (int numElems, fractional* srcV, int* minIndex);
```

Arguments

Parameters	Description
numElems	Number of elements in the source vector
srcV	Pointer to the source vector
minIndex	Pointer to the holder for the index of the (last) minimum element

Returns

Minimum value in the vector.

Remarks

If *srcV*[i] = *srcV*[j] = *minVal*, and i < j, then **minIndex* = j.

Source File

- vmin_aa.s

Function Profile

Device	Program Words	Cycles
PIC32A	10	32 + 4*(*numElems* - 1),

System resource usage

- *W0...W6* - used, not restored
- *REPEAT* instruction/s usage – None

2.3.8. VectorMultiply

Description

`VectorMultiply` multiplies the value of each element in the source one vector with its counterpart in the source two vector and places the result in the destination vector.

Prototype

```
fractional* VectorMultiply (int numElems, fractional* dstV,
fractional* srcV1, fractional* srcV2);
```

Arguments

Parameters	Description
numElems	Number of elements in the source vectors
dstV	Pointer to the destination vector
srcV1	Pointer to the source one vector
srcV2	Pointer to the source two vector

Returns

Pointer to the base address of the destination vector.

Remarks

This operation is also known as the vector element-by-element multiplication.

This function can be computed in place.

This function can be self-applicable.

Source File

- `vmul_aa.s`

Function Profile

Device	Program Words	Cycles
PIC32A	20	36 + ⌈2.5 x *numElems*⌉

System resource usage

- *W0...W4* - used, not restored
- *W13* - saved, used, restored
- *ACCA* - used, not restored
- *CORCON* - saved, used, restored
- *REPEAT* instruction(s) usage – None

2.3.9. VectorNegate

Description

`VectorNegate` negates (changes the sign of) the values of the elements in the source vector and places them in the destination vector.

Prototype

```
fractional* VectorNegate (int numElems, fractional* dstV, fractional* srcV1);
```

Arguments

Parameters	Description
numElems	Number of elements in the source vectors
dstV	Pointer to the destination vector
srcV1	Pointer to the source vector

Returns

Pointer to the base address of the destination vector.

Remarks

The negated value of 0x80000000 is set to 0x7FFFFFFF.

This function can be computed in place.

Source File

- `vneg_aa.s`

Function Profile

Device	Program Words	Cycles
PIC32A	10	32 + (*numElems*)

System resource usage

- *W0...W3* - used, not restored
- *CORCON* - saved, used, restored
- *REPEAT* instruction(s) usage – None

2.3.10. VectorPower

Description

`VectorPower` computes the power of a source vector as the sum of the squares of its elements.

Prototype

```
fractional VectorPower(int numElems, fractional* srcV);
```

Arguments

Parameters	Description
numElems	Number of elements in the source vectors
srcV	Pointer to the source vector

Returns

Value of the vector's power (sum of squares).

Remarks

If the absolute value of *sum-of-squares* is larger than the maximum value of 1.31 fractional data type, the operation results in saturation.

This function can be self-applicable.

Source File

- `vpow_aa.s`

Function Profile

Device	Program Words	Cycles
PIC32A	9	36 + (*numElems*)

System resource usage

- *W0...W4* - used, not restored
- *ACCA* - used, not restored
- *CORCON* - saved, used, restored
- One *REPEAT* instruction

2.3.11. VectorScale

Description

`VectorScale` multiplies the values of all the elements in the source vector by a scale value and places the result in the destination vector.

Prototype

```
fractional* VectorScale (int numElems, fractional* dstV, fractional* srcV,
fractional sclVal);
```

Arguments

Parameters	Description
numElems	Number of elements in the source vectors
dstV	Pointer to the destination vector
srcV	Pointer to the source vector
sclVal	Value by which to scale the vector elements

Returns

Pointer to the base address of the destination vector.

Remarks

sclVal must be a fractional number in 1.31 format.

This function can be computed in place.

This function can be self-applicable.

Source File

- `vscl_aa.s`

Function Profile

Device	Program Words	Cycles
PIC32A	20	40 + ⌈2.5 x *numElems*⌉

System resource usage

- *W0...W4* - used, not restored
- *W13* - saved, used, restored
- *ACCA* - used, not restored

- *CORCON* - saved, used, restored
- *REPEAT* instruction(s) usage – None

2.3.12. VectorSubtract

Description

`VectorSubtract` subtracts the value of each element in the source one vector with its counterpart in the source two vector and places the result in the destination vector.

Prototype

```
fractional* VectorSubtract (int numElems, fractional* dstV,
fractional* srcV1, fractional* srcV2);
```

Arguments

Parameters	Description
numElems	Number of elements in the source vectors
dstV	Pointer to the destination vector
srcV1	Pointer to the source one vector
srcV2	Pointer to the source two vector

Returns

Pointer to the base address of the destination vector.

Remarks

If the absolute value of *srcV1[n] - srcV2[n]* is larger than the maximum value of 1.31 fractional data type, the operation results in saturation for the n^{th} element.

This function can be computed in place.

This function can be self-applicable.

Source File

- `vsub_aa.s`

Function Profile

Device	Program Words	Cycles
PIC32A	11	36 + 4*(*numElems*)

System resource usage

- *W0...W4* - used, not restored
- *ACCA* - used, not restored
- *CORCON* - saved, used, restored
- *REPEAT* instruction(s) usage – None

2.3.13. VectorZeroPad

Description

`VectorZeroPad` copies the contents of the source vector into the beginning of the destination vector, and then fills the remaining positions—specified by *numZeros* — with zeros to complete the destination vector.

dstV[n] = srcV[n], 0 ≤ n < numElems

dstV[n] = 0, numElems ≤ n < numElems + numZeros

Prototype

```
fractional* VectorZeroPad (int numElems, int numZeros, fractional* dstV,
fractional* srcV1);
```

Arguments

Parameters	Description
numElems	Number of elements in the source vectors
numZeros	Number of zeros
dstV	Pointer to the destination vector (of size *numElems* + *numZeros*)
*srcV*1	Pointer to the source vector

Returns

Pointer to the base address of the destination vector.

Remarks

The destination vector *must* already exist with exactly *numElems* + *numZeros* number of elements.

This function can be computed in place.

This function uses *VectorCopy*.

Source File

- vzpad_aa.s

Function Profile

Device	Program Words	Cycles
PIC32A	9	12 + (*numZeros*)

Notes:

1. The above-mentioned program word and cycle counts pertain solely to *VectorZeroPad*. However, as this function inherently utilizes *VectorCopy*, the respective counts for *VectorCopy* must also be considered.
2. In the description of *VectorCopy*, the number of cycles reported includes four cycles of C-function call overhead. Thus, the number of actual cycles from *VectorCopy* to add to *VectorZeroPad* is four less than whatever number is reported for a stand-alone *VectorCopy*.

System resource usage

- *W0...W5* - used, not restored
- One *REPEAT* instruction(s)

3. Window Functions

Function	Description
BartlettInit	Initializes a Barlett window of length *numElems*.
BlackmanInit	Initializes a Blackman (three terms) window of length *numElems*.
HammingInit	Initializes a Hamming window of length *numElems*.
HanningInit	Initializes a Hanning window of length *numElems*.
KaiserInit	Initializes a Kaiser window with shape determined by argument *betaVal* and of length *numElems*.
VectorWindow	Applies a window to a given source vector and stores the resulting windowed vector in a destination vector.

3.1. Window Operations

A window is a vector with a specific value distribution within its domain (0 ≤ n < *numElems*). The particular value distribution depends on the characteristics of the window being generated.

Given a vector, its value distribution may be modified by applying a window to it. In these cases, the window must have the same number of elements as the vector to modify.

Before a vector can be windowed, the window must be created. Window initialization operations are provided which generate the values of the window elements. For higher numerical precision, these values are computed in floating-point arithmetic, and the resulting quantities are stored as 1.31 fractionals.

To avoid excessive overhead when applying a window operation, a particular window could be generated once and used many times during the execution of the program. Thus, it is advisable to store the window returned by any of the initialization operations in a permanent (static) vector.

3.2. User Considerations

When using window functions, consider the following:

1. All the window initialization functions have been designed to generate window vectors allocated in the default RAM memory space (X-Data or Y-Data).
2. The window function is designed to operate on vectors allocated in the default RAM memory space (X-Data or Y-Data).
3. It is recommended that the STATUS Register (SR) be examined after completion of each function call.

3.3. Functions

3.3.1. BartlettInit

Description

`BartlettInit` initializes a Barlett window of length *numElems*.

Prototype

```
fractional* BartlettInit(int numElems, fractional* window);
```

Arguments

Parameters	Description
numElems	Number of rows in the source matrices.

BartlettInit (continued)	
Parameters	**Description**
window	Number of columns in the source matrices.

Returns

Pointer to the base address of the initialized window.

Remarks

The window vector must already exist, with exactly *numElems* number of elements.

Source File

- `initbart.c (c-code implementation)`

3.3.2. BlackmanInit

Description

`BlackmanInit` initializes a Blackman (three terms) window of length *numElems*.

Prototype

```
fractional* BlackmanInit(int numElems, fractional* window);
```

Arguments

Parameters	Description
numElems	Number of rows in the source matrices.
window	Number of columns in the source matrices.

Returns

Pointer to the base address of the initialized window.

Remarks

The window vector must already exist, with exactly *numElems* number of elements.

Source File

- `initblck.c (c-code implementation)`

3.3.3. HammingInit

Description

`HammingInit` initializes a Hamming window of length *numElems*.

Prototype

```
fractional* HammingInit(int numElems, fractional* window);
```

Arguments

Parameters	Description
numElems	Number of rows in the source matrices.
window	Number of columns in the source matrices.

Returns

Pointer to the base address of the initialized window.

Remarks

The window vector must already exist, with exactly *numElems* number of elements.

Source File

- `inithamm.c (c-code implementation)`

3.3.4. HanningInit

Description

`HanningInit` initializes a Hanning window of length *numElems*.

Prototype

```
fractional* HanningInit(int numElems, fractional* window);
```

Arguments

Parameters	Description
numElems	Number of rows in the source matrices.
window	Number of columns in the source matrices.

Returns

Pointer to the base address of the initialized window.

Remarks

The window vector must already exist, with exactly *numElems* number of elements.

Source File

- `inithann.c (c-code implementation)`

3.3.5. KaiserInit

Description

`KaiserInit` initializes a Kaiser window of length *numElems*.

Prototype

```
fractional* KaiserInit(int numElems, fractional* window, float betaVal);
```

Arguments

Parameters	Description
numElems	Number of rows in the source matrices.
window	Number of columns in the source matrices.
betaVal	Window shaping parameter.

Returns

Pointer to the base address of the initialized window.

Remarks

The window vector must already exist, with exactly *numElems* number of elements.

Source File

- `initkais.c (c-code implementation)`

3.3.6. VectorWindow

Description

`VectorWindow` applies a window to a given source vector and stores the resulting windowed vector in a destination vector.

Prototype

```
fractional* VectorWindow (int numElems, fractional* dstV, fractional* srcV,
fractional* window);
```

Arguments

Parameters	Description
numElems	Number of elements in the source vectors.
dstV	Pointer to the destination vector.
srcV	Pointer to the source vector.
window	Pointer to the initialized window.

Returns

Pointer to the base address of the destination vector.

Remarks

The window vector must have already been initialized, with exactly *numElems* number of elements.

This function can be computed in place.

This function can be self-applicable.

This function uses *VectorMultiply*.

Source File

- `dowindow_aa.s`

Function Profile

Device	Program Words	Cycles
PIC32A	2	11

Notes:

1. The above-mentioned program word and cycle counts pertain solely to *VectorWindow*. However, as this function inherently utilizes *VectorMultiply*, the respective counts for *VectorMultiply* must also be considered.
2. In the description of *VectorMultiply*, the number of cycles reported includes four cycles of C-function call overhead. Thus, the number of actual cycles from *VectorMultiply* to add to *VectorWindow* is four less than whatever number is reported for a stand-alone *VectorMultiply*.

System resource usage

- None. (*VectorWindow* just includes a function to call to *VectorMultiply*.)

4. Matrix Functions

Function	Description
MatrixAdd	Adds the value of each element in the source one matrix with its counterpart in the source two matrix and places the result in the destination matrix.
MatrixMultiply	Performs the matrix multiplication between the source one and source two matrices and places the result in the destination matrix.
MatrixScale	Scales (multiplies) the values of all elements in the source matrix by a scale value and places the result in the destination matrix.
MatrixSubtract	Subtracts the value of each element in the source two matrix from its counterpart in the source one matrix and places the result in the destination matrix.
MatrixTranspose	Transposes the rows by the columns in the source matrix and places the result in the destination matrix.
MatrixInvert	Computes the inverse of the source matrix and places the result in the destination matrix.

4.1. Fractional Matrix Operations

A fractional matrix is a collection of numerical values, the matrix elements allocated contiguously in memory, with the first element at the lowest memory address. One word of memory is used to store the value of each element, and this quantity must be interpreted as a fractional number represented in the 1.31 format.

A pointer addressing the first element of the matrix is used as a handle which provides access to each of the matrix values. The address of the first element is referred to as the base address (*BA*) of the matrix. Because each element of the matrix is 32-bits, the base address must be aligned to two or four, respectively.

The two-dimensional arrangement of a matrix is emulated in the memory storage area by placing its elements organized in row major order. Thus, the first value in memory is the first element of the first row. It is followed by the rest of the elements of the first row. Then, the elements of the second row are stored, and so on, until all the rows are in memory. This way, the element at row *r* and column *c* of a matrix with *R* rows and *C* columns is located from the matrix base address (BA) at:

$BA + 4 * (C(r - 1) + c - 1)$, for $1 \le r \le R$, $1 \le c \le C$.

Unary and binary fractional matrix operations are implemented in this library. The operand matrix in a unary operation is called the source matrix. In a binary operation the first operand is referred to as the source one matrix and the second matrix as the source two matrix. Each operation applies some computation to one or several elements of the source matrix(ces). The operations result in a matrix, referred to as the destination matrix.

Some operations resulting in a matrix allow computation in place. This means the results of the operation are placed back into the source matrix (or the source one matrix for a binary operation). In this case, the destination matrix is said to (physically) replace the source (one) matrix. If an operation can be computed in place, it is indicated as such in the comments provided with the function description.

For some binary operations, the two operands can be the same (physical) source matrix, which means the operation is applied to the source matrix and itself. If this type of computation is possible for a given operation, it is indicated as such in the comments provided with the function description.

Some operations can be self-applicable and computed in place.

All fractional matrix operations in this library take as arguments to the number of rows and the number of columns of the operand matrix(ces). Based on the values of these arguments, the following assumptions are made:

1. The sum of sizes of all the matrices involved in a particular operation falls within the range of available data memory for the target device.
2. In the case of binary operations, the number of rows and columns of the operand matrices must obey the rules of vector algebra (i.e., For matrix addition and subtraction, the two matrices must have the same number of rows and columns, while for matrix multiplication, the number of columns of the first operand must be the same as the number of rows of the second operand.). The source matrix to the inversion operation must be square (the same number of rows and columns) and non-singular (its determinant different than zero).

The destination matrix must be large enough to accept the results of an operation.

4.2. User Considerations

When using matrix functions, consider the following:

1. No boundary checking is performed by these functions. Out of range dimensions (including zero row and/or zero column matrices) as well as nonconforming use of source matrix sizes in binary operations may produce unexpected results.
2. The matrix addition and subtraction operations could lead to saturation if the sum of corresponding elements in the source(s) matrix(ces) is greater than $1-2^{-31}$ for 1.31 fractional or smaller than -1.
3. The matrix multiplication operation could lead to saturation if the sum of products of corresponding row and column sets results in a value greater than $1-2^{-31}$ for 1.31 fractional or smaller than -1.
4. It is recommended that the STATUS Register (SR) is examined after completion of each function call. In particular, users can inspect the SA, SB and SAB flags after the function returns to determine if saturation occurred.
5. Operations which return a destination matrix can be nested. For instance, if:
   a = Op1 (b, c), with b = Op2 (d), and c = Op3 (e, f), then

   a = Op1 (Op2 (d), Op3 (e, f))
6. All cycle count values for PIC32A are considered with the PBU cache enabled and may differ depending on the status of the PBU cache or on the placement of vectors and code.

4.3. Additional Remarks

The description of the functions limits the scope to the regular usage of these operations. However, since no boundary checking is performed during computation of these functions, operations and the results are interpreted to fit specific needs.

For instance, while computing the *MatrixMultiply* function, the dimensions of the intervening matrices do not necessarily need to be {*numRows1, numCos1Rows2*} for the source one matrix, {*numCols1Rows2, numCols2*} for the source two matrix and {*numRows1, numCols2*} for the destination matrix. In fact, all that is needed is that their sizes are large enough so during computation, the pointers do not exceed their memory range.

As another example, when a source matrix of dimension {*numRows, numCols*} is transposed, the destination matrix has dimensions {*numCols, numRows*}. Therefore, the operation can be computed in place only if the source matrix is square. Nevertheless, the operation can be successfully applied in place to non-square matrices; all that needs to be kept in mind is the implicit change of dimensions.

Other possibilities can be exploited from the fact that no boundary checking is performed.

4.4. Functions

4.4.1. MatrixAdd

Description

`MatrixAdd` adds the value of each element in the source one matrix with its counterpart in the source two matrix and places the result in the destination matrix.

Prototype

```
fractional* MatrixAdd (int numRows, int numCols, fractional* dstM,
fractional* srcM1, fractional* srcM2);
```

Arguments

Parameters	Description
numRows	Number of rows in the source matrices.
numCols	Number of columns in the source matrices.
dstM	Pointer to the destination matrix
srcM1	Pointer to the source one matrix
srcM2	Pointer to the source two matrix

Returns

Pointer to the base address of the destination matrix.

Remarks

If the absolute value of *srcM1[r][c]* + *srcM2[r][c]* is larger than the maximum value of the 1.31 fractional data type, this operation results in saturation for the n^{th} element.

This function can be computed in place.

This function can be self applicable.

Source File

- `madd_aa.s`

Function Profile

Device	Program Words	Cycles
PIC32A	11	38 + 4*(*numRows* * *numCols*)

System resource usage

- *W0...W5* - used, not restored
- *ACCA* - used, not restored
- *CORCON* - saved, used, restored
- *REPEAT* instruction(s) usage – None

4.4.2. MatrixMultiply

Description

`MatrixMultiply` performs the matrix multiplication between the source one and source two matrices and places the result in the destination matrix.

Symbolically:

$$dstM[i][j] = \sum_{k=0}^{numCols1Rows2-1} src1M[i][k] * src2M[k][j]$$

where:

$0 \leq i < numRows1$

$0 \leq j < numCols2$

$0 \leq k < numCols1Rows2$

Prototype

```
fractional* MatrixMultiply (int numRows1, int numCols1Rows2, int numCols2,
fractional* dstM, fractional* srcM1, fractional* srcM2);
```

Arguments

Parameters	Description
numRows1	Number of rows in the source one matrix.
numCols1Rows2	Number of columns in the source one matrix which is same as number of rows in the source two matrix.
numCols2	Number of columns in the source two matrix.
dstM	Pointer to the destination matrix
srcM1	Pointer to the source one matrix
srcM2	Pointer to the source two matrix

Returns

Pointer to the base address of the destination matrix.

Remarks

If the absolute value of result in *dstM[r][c]* is larger than the maximum value of the 1.31 fractional data type, this operation results in saturation for the (r,c)th element.

If the source one matrix is squared, then this function can be computed in place and can be self-applicable.

Source File

- `mmul_aa.s`

Function Profile

Device	Program Words	Cycles
PIC32A	27	45+*numRows1**(6+*numCols2**(9+4*(*numCols1Rows2* - 1)))

System resource usage

- *W0...W7* - used, not restored
- *W8...W12* - saved, used, restored
- *ACCA* - used, not restored
- *CORCON* - saved, used, restored
- *REPEAT* instruction(s) usage – None

4.4.3. MatrixScale

Description

`MatrixScale` scales (multiplies) the values of all elements in the source matrix by a scale value and places the result in the destination matrix.

Prototype

```
fractional* MatrixScale (int numRows, int numCols, fractional* dstM,
fractional* srcM, fractional sclVal);
```

Arguments

Parameters	Description
numRows	Number of rows in the source matrices
numCols	Number of columns in the source matrices
dstM	Pointer to the destination matrix
srcM1	Pointer to the source one matrix
sclVal	Value used to scale the matrix elements

Returns

Pointer to the base address of the destination matrix.

Remarks

This function can be computed in place.

Source File

- `mscl_aa.s`

Function Profile

Device	Program Words	Cycles
PIC32A	20	44 + ⌈2.5 x *numElems*⌉

System resource usage

- *W0...W5* - used, not restored
- *ACCA* - used, not restored
- *CORCON* - saved, used, restored
- *REPEAT* instruction(s) usage – None

4.4.4. MatrixSubtract

Description

`MatrixSubtract` subtracts the value of each element in the source two matrix from its counterpart in the source one matrix and places the result in the destination matrix.

Prototype

```
fractional* MatrixAdd (int numRows, int numCols, fractional* dstM,
fractional* srcM1, fractional* srcM2);
```

Arguments

Parameters	Description
numRows	Number of rows in the source matrices
numCols	Number of columns in the source matrices
dstM	Pointer to the destination matrix
srcM1	Pointer to the source one matrix
srcM2	Pointer to the source two matrix

Returns

Pointer to the base address of the destination matrix.

Remarks

If the absolute value of *srcM1[r][c] - srcM2[r][c]* is larger than the maximum value of the 1.31 fractional data type, this operation results in saturation for the n^{th} element.

This function can be computed in place.

This function can be self-applicable.

Source File

- `msub_aa.s`

Function Profile

Device	Program Words	Cycles
PIC32A	11	38 + 4*(*numRows* * *numCols*)

System resource usage

- *W0...W5* - used, not restored
- *ACCA* - used, not restored
- *CORCON* - saved, used, restored
- *REPEAT* instruction(s) usage – None

4.4.5. MatrixTranspose

Description

`MatrixTranspose` transposes the rows by the columns in the source matrix and places the result in the destination matrix.

In effect:

dstM[i][j] = srcM[j][i],

0 ≤ i < numRows, 0 ≤ j < numCols.

Prototype

```
fractional* MatrixTranspose (int numRows, int numCols, fractional* dstM,
fractional* srcM);
```

Arguments

Parameters	Description
numRows	Number of rows in the source matrices
numCols	Number of columns in the source matrices
dstM	Pointer to the destination matrix
srcM	Pointer to the source matrix

Returns

Pointer to the base address of the destination matrix.

Remarks

If the source matrix is square, this function can be computed in place.

Source File

- `mtrp_aa.s`

Function Profile

Device	Program Words	Cycles
PIC32A	9	`23 + numCols *(7 + (numRows-1)*4)`

System resource usage

- *W0...W5* - used, not restored
- *REPEAT* instruction(s) usage – None

4.4.6. MatrixInvert

The result of inverting a non-singular, square, fractional matrix is another square matrix (of the same dimension) whose element values are not necessarily constrained to the discrete fractional set {-1, ..., ~1}. Thus, no matrix inversion operation is provided for fractional matrices.

However, since matrix inversion is a very useful operation, an implementation based on floating-point number representation and arithmetic is provided within the DSP Library.

Description

`MatrixInverse` computes the inverse of the source matrix and places the result in the destination matrix.

Prototype

```
float* MatrixInvert (int numRowsCols, float* dstM, float* srcM,
float* pivotFlag, int* swappedRows, int* swappedCols);
```

Arguments

Parameters	Description
numRowsCols	Number of rows/columns in the source (square) matrix
dstM	Pointer to the destination matrix
srcM	Pointer to the source matrix
pivotFlag	Pointer to a length *numRowsCols* vector
swappedRows	Pointer to a length *numRowsCols* vector
swappedCols	Pointer to a length *numRowsCols* vector

Returns

Pointer to the base address of the destination matrix, or NULL if the source matrix is singular.

Remarks

Even though the vectors *pivotFlag*, *swappedRows* and *swappedCols* are for internal use only, they must be allocated prior to calling this function.

If the source matrix is singular (determinant equal to zero), the matrix does not have an inverse. In this case, the function returns NULL.

This function can be computed in place.

Source File

- `minv.c (assembled from C-code)`

5. Filtering Functions

Function	Description
FIRStruct	Describes the filter structure for any of the FIR filters.
FIR	Applies an FIR filter to the sequence of the source samples, places the results in the sequence of destination samples and updates the delay values.
FIRDecimate	Decimates the sequence of the source samples at a rate of R to 1, or equivalently, it downsamples the signal by a factor of R.
FIRDelayInit	Initializes to zero the delay values in an *FIRStruct* filter structure.
FIRInterpolate	Interpolates the sequence of the source samples at a rate of 1 to R, or equivalently, it up-samples the signal by a factor of R.
FIRInterpDelayInit	Initializes to zero the delay values in an *FIRStruct* filter structure, optimized for use with an FIR interpolating filter.
FIRLattice	Uses a lattice structure implementation to apply an FIR filter to the sequence of the source samples. It then places the results in the sequence of the destination samples and updates the delay values.
FIRLMS	Applies an adaptive FIR filter to the sequence of the source samples, stores the results in the sequence of the destination samples and updates the delay values.
FIRLMSNorm	Applies an adaptive FIR filter to the sequence of the source samples, stores the results in the sequence of the destination samples and updates the delay values. The filter coefficients are also updated, at a sample-per-sample basis, using a Normalized Least Mean Square algorithm applied according to the values of the reference samples.
FIRStructInit	Initializes the values of the parameters in an *FIRStruct* FIR Filter structure.
IIRCanonic	Applies an IIR filter, using a cascade of canonic (direct form II) biquadratic sections, to the sequence of the source samples. It places the results in the sequence of the destination samples and updates the delay values.
IIRCanonicInit	Initializes to zero the delay values in an *IIRCanonicStruct* filter structure.
IIRLattice	Uses a lattice structure implementation to apply an IIR filter to the sequence of the source samples. It then places the results in the sequence of the destination samples and updates the delay values.
IIRLatticeInit	Initializes to zero the delay values in an *IIRLatticeStruct* filter structure.
IIRTransposed	Applies an IIR filter, using a cascade of transposed (direct form II) biquadratic sections, to the sequence of the source samples.
IIRTransposedInit	Initializes to zero the delay values in an *IIRTransposedStruct* filter structure.

5.1. Fractional Filter Operations

Filtering the data sequence represented by fractional vector *x[n] (0 ≤ n < N)* is equivalent to solving the following difference equation for every n^{th} sample, which results into the filtered data sequence *y[n].*

$$y[n] + \sum_{p=1}^{P-1} -a[p] \times y[n-p] = \sum_{m=1}^{M-1} b[m] \times x[n-m]$$

In this sense, the fractional filter is characterized by the fractional vectors *a[p] (0 ≤ p < P)* and *b[m] (0 ≤ m < M),* referred to as the set of filter coefficients, which are designed to induce some pre-specified changes in the signal represented by the input data sequence.

When filtering, it is important to know and manage the past history of the input and output data sequences (*x[n], -M + 1 ≤ n < 0*, and *y[n], -P + 1 ≤ n < 0),* which represent the initial conditions of the filtering operation. Also, when repeatedly applying the filter to contiguous sections of the input data

sequence, it is necessary to remember the final state of the last filtering operation (*x[n], N – M + 1 ≤ n < N – 1*, and *y[n], N – P + 1 ≤ n < N – 1*). This final state is then taken into consideration for the calculations of the next filtering stage. Accounting for the past history and current state is required in order to perform a correct filtering operation.

The management of the past state and current state of a filtering operation is commonly implemented via additional sequences (also fractional vectors), referred to as the filter delay line. Prior to applying the filter operation, the delay describes the past state of the filter. After performing the filtering operation, the delay contains a set of the most recently filtered data samples and of the most recent output samples.

Note: To ensure correct operation of a particular filter implementation, it is advisable to initialize the delay values to zero by calling the corresponding initialization function.

In the filter implementations provided with the DSP Library, the input data sequence is referred to as the sequence of source samples, while the resulting filtered sequence is called the destination samples. The filter coefficients (*a,b*) and delay are usually thought of as making up a filter structure. In all filter implementations, the input and output data samples may be allocated in default RAM memory space (X-Data or Y-Data). Filter coefficients may reside either in X-Data memory or program memory, and filter delay values must be accessed only from Y-Data.

Note: The PIC32A architecture does not mandate filter delay values to be located in Y data space. However, not placing it in Y data space will result in sequential operand access, thereby affecting data fetch efficiency.

5.2. FIR and IIR Filter Implementations

The properties of a filter depend on the value distribution of its coefficients. In particular, two types of filters are of special interest: Finite Impulse Response (FIR) filters, for which *a[m] = 0* when *1 ≤ m < M*, and Infinite Impulse Response (IIR) filters, those such that *a[0] ≠ 0* and *a[m] ≠ 0* for some *m* in {1, ..., M}. Other classifications within the FIR and IIR filter families account for the effects that the operation induces on input data sequences.

Furthermore, even though filtering consists on solving the difference equation stated above, several implementations are available which are more efficient than direct computation of the difference equation. Also, some other implementations are designed to execute the filtering operation under the constraints imposed by fractional arithmetic.

All these considerations lead to a proliferation of filtering operations, of which a subset is provided by the DSP Library.

5.3. Single Sample Filtering

The filtering functions provided in the DSP Library are designed for block processing. Each filter function accepts an argument named *numSamps* which indicates the number of words of input data (block size) to operate on. If single sample filtering is desired, set *numSamps* to 1. This will have the effect of filtering one input sample, and the function will compute a single output sample from the filter.

5.4. User Considerations

All the fractional filtering operations in this library rely on the values of either input parameters or data structure elements to specify the number of samples to process and the sizes of the coefficients and delay vectors. Based on these values, the following assumptions are made:

1. The sum of sizes of all the vectors (sample sequences) involved in a particular operation falls within the range of available data memory for the target device.
2. The destination vector must be large enough to accept the results of an operation.
3. No boundary checking is performed by these functions. Out of range sizes (including zero length vectors) as well as nonconforming use of source vectors and coefficient sets may produce unexpected results.

4. It is recommended that the STATUS Register (SR) is examined after completion of each function call. In particular, users can inspect the SA, SB and SAB flags after the function returns to determine if saturation occurred.
5. Operations which return a destination vector can be nested. For instance, if:
   a = Op1 (b, c), with b = Op2 (d), and c = Op3 (e, f), then
   a = Op1 (Op2 (d), Op3 (e, f))
6. All cycle count values for PIC32A are measured with PBU cache enabled and may differ depending on the status of PBU cache or on the placement of vectors and code.

5.5. Functions

This section describes the individual functions for implementing filtering operations. For more information on digital filters refer to Alan Oppenheim and Ronald Schafer's "Discrete-time Signal Processing", Prentice Hall, 1989. For implementation details of Least Mean Square FIR filters, please refer to T. Hsia's "Convergence Analysis of LMS and NLMS Adaptive Algorithms", Proc. ICASSP, pp. 667-670, 1983, as well as Sangil Park and Garth Hillman's "On Acoustic-Echo Cancellation Implementation with Multiple Cascadable Adaptive FIR Filter Chips", Proc. ICASSP, 1989.

5.5.1. FIRStruct

Description

`FIRStruct` describes the filter structure for any of the FIR filters.

Declaration

```
typedef struct {
  int numCoeffs;
  fractional* coeffsBase;
  fractional* coeffsEnd;
  fractional* delayBase;
  fractional* delayEnd;
  fractional* delay;
} FIRStruct;
```

Parameters

Parameters	Description
numCoeffs	Number of filter coefficients (also M)
coeffsBase	Base address for filter coefficients (also h)
coeffsEnd	End address for filter coefficients
delayBase	Base address for delay buffer
delayEnd	End address for delay buffer
delay	Current value of delay pointer (also d)

Remarks

The number of filter coefficients is M.

Coefficients, h[m], defined in 0 ≤ m < M, either within X-Data space or program memory.

Delay buffer d[m], defined in 0 ≤ m < M, only in Y-Data space.

delayBase points to the actual address where the delay buffer is allocated.

delayEnd is the address of the last byte of the filter delay buffer.

When the coefficients and delay buffers are implemented as circular increasing modulo buffers, both *coeffsBase* and *delayBase* must be aligned to a zero power of two for each address (*coeffsEnd* and *delayEnd* are odd addresses). Whether these buffers are implemented as circular increasing modulo buffers or not is indicated in the remarks section of each FIR filter function description.

When the coefficients and delay buffers are not implemented as circular (increasing) modulo buffers, *coeffsBase* and *delayBase* do not need to be aligned to a zero power of two address, and the values of *coeffsEnd* and *delayEnd* are ignored within the particular FIR Filter function implementation.

5.5.2. FIR

Description

`FIR` applies an FIR filter to the sequence of source samples, places the results in the sequence of destination samples and updates the delay values.

Prototype

```
fractional* FIR (int numSamps, fractional* dstSamps, fractional* srcSamps,
FIRStruct* filter);
```

Arguments

Parameters	Description
numSamps	Number of the input samples to filter (also *N)*
dstSamps	Pointer to the destination samples (also *y*)
srcSamps	Pointer to the source samples (also *x*)
filter	Pointer to the *FIRStruct* filter structure

Return

Pointer to the base address of the destination sample.

Remarks

Number of filter coefficients is M.

Coefficients, h[m], defined in 0 ≤ m < M, implemented as a circular increasing modulo buffer.

Delay, d[m], defined in 0 ≤ m < M, implemented as a circular increasing modulo buffer. The Delay vector must be placed in Y-Data space.

Source samples, x[n], defined in 0 ≤ n < N.

Destination samples, y[n], defined in 0 ≤ n < N.

(See also *FIRStruct*, *FIRStructInit* and *FIRDelayInit*.)

Source File

- `fir_aa.s`

Cycle counts for PIC32A:

Source Vector Size	Cycles if coefficients in X-mem	Cycles if coefficients in P-mem
32	1222	3112
64	2376	6152
128	4676	12232
256	9288	24392
512	18500	48712
1024	36936	97352

FIR (continued)		
Source Vector Size	Cycles if coefficients in X-mem	Cycles if coefficients in P-mem
2048	73796	194632

*(*All values with `numCoeffs = 32`)*

System resource usage

- *W0...W7* - used, not restored
- *ACCA* - used, not restored
- *CORCON* - saved, used, restored
- *MODCON* - saved, used, restored
- *XMODSTRT* - saved, used, restored
- *XMODEND* - saved, used, restored
- *YMODSTRT* - saved, used, restored
- *YMODEND* - saved, used, restored
- *REPEAT* instruction(s) usage – 1

5.5.3. FIRDecimate

Description

`FIRDecimate` decimates the sequence of the source samples at a rate of R to 1, or equivalently, it downsamples the signal by a factor of R.

Effectively,

y[n] = x[Rn].

To diminish the effect of aliasing, the source samples are first filtered and then downsampled. The decimated results are stored in the sequence of destination samples, and the delay values are updated.

Prototype

```
fractional* FIRDecimate (int numSamps, fractional* dstSamps,
fractional* srcSamps, FIRStruct* filter, int rate);
```

Arguments

Parameters	Description
numSamps	Number of the output samples to filter (also *N; with N being multiple of R)*
dstSamps	Pointer to the destination samples (also *y*)
srcSamps	Pointer to the source samples (also *x*)
filter	Pointer to the *FIRStruct* filter structure
rate	Rate of decimation (downsampling factor, also R)

Return

Pointer to the base address of the destination sample.

Remarks

Number of filter coefficients is M, with M being an integer multiple of R.

Coefficients, h[m], defined in 0 ≤ m < M, are not implemented as a circular modulo buffer.

Delay, d[m], defined in 0 ≤ m < M, are not implemented as a circular modulo buffer.

Source samples, x[n], defined in 0 ≤ n < NR.

Destination samples, y[n], defined in 0 ≤ n < N.

(See also *FIRStruct*, *FIRStructInit* and *FIRDelayInit*.)

Source File

- `firdecim_aa.s`

Cycle counts for PIC32A:

Source Vector Size	Cycles if coefficients in X-mem	Cycles if coefficients in P-mem
32	2352	4240
64	4660	8432
128	9258	16816
256	18484	33584
512	36916	67120
1024	73780	134192
2048	147508	268336

*(*All values with `numCoeffs = 32; rate = 2`)*

System resource usage

- *W0...W7* - used, not restored
- *W8...W11* - saved, used, restored
- *ACCA* - used, not restored
- *CORCON* - saved, used, restored
- *REPEAT* instruction(s) usage – 2

5.5.4. FIRDelayInit

Description

`FIRDelayInit` initializes to zero the delay values in an *`FIRStruct`* filter structure.

Prototype

```
void FIRDelayInit (FIRStruct* filter);
```

Arguments

Parameters	Description
`filter`	Pointer to *FIRStruct* filter structure

Returns

None.

Remarks

See description of *FIRStruct* structure above.

Note: FIR interpolator's delay is initialized by function *FIRInterpDelayInit*.

Source File

- `firdelay_aa.s`

Function Profile

Device	Program Words	Cycles
PIC32A	6	22 + M

System resource usage

- *W0...W2* - used, not restored
- *REPEAT* instruction(s) usage – 1

5.5.5. FIRInterpolate

Description

`FIRInterpolate` interpolates the sequence of the source samples at a rate of 1 to R, or equivalently, it upsamples the signal by a factor of R.

Effectively,

y[n] = x[n/R].

To diminish the effect of aliasing, the source samples are first upsampled and then filtered. The interpolated results are stored in the sequence of destination samples, and the delay values are updated.

Prototype

```
fractional* FIRInterpolate (int numSamps, fractional* dstSamps,
fractional* srcSamps, FIRStruct* filter, int rate);
```

Arguments

Parameters	Description
numSamps	Number of the input samples to filter (also *N; with N being multiple of R)*
dstSamps	Pointer to the destination samples (also *y*)
srcSamps	Pointer to the source samples (also *x*)
filter	Pointer to the *FIRStruct* filter structure
rate	Rate of the interpolation (upsampling factor, also R)

Return

Pointer to the base address of the destination sample.

Remarks

Number of filter coefficients is M, with M being an integer multiple of R.

Coefficients, h[m], defined in 0 ≤ m < M, are not implemented as a circular modulo buffer.

Delay, d[m], defined in 0 ≤ m < M/R, are not implemented as a circular modulo buffer.

Source samples, x[n], defined in 0 ≤ n < N.

Destination samples, y[n], defined in 0 ≤ n < NR.

(See also *FIRStruct*, *FIRStructInit* and *FIRInterpDelayInit*.)

Source File

- `firinter_aa.s`

Cycle counts for PIC32A:

Source Vector Size	Cycles if coefficients in X-mem	Cycles if coefficients in P-mem
32	5344	9002

FIRInterpolate (continued)		
Source Vector Size	**Cycles if coefficients in X-mem**	**Cycles if coefficients in P-mem**
64	10624	17930
128	21184	35786
256	42304	71498
512	84544	142922
1024	169024	285770
2048	337984	571466

*(*All values with `numCoeffs = 32; rate = 2`)*

System resource usage

- *W0...W7* - used, not restored
- *W8...W12* - saved, used, restored
- *ACCA* - used, not restored
- *CORCON* - saved, used, restored
- *REPEAT* instruction(s) usage – 2

5.5.6. FIRInterpDelayInit

Description

`FIRInterpDelayInit` initializes to zero the delay values in an *FIRStruct* filter structure, optimized for use with an FIR interpolating filter.

Prototype

```
void FIRInterpDelayInit (FIRStruct* filter, int rate);
```

Arguments

Parameters	Description
filter	Pointer to the *FIRStruct* filter structure
rate	Rate of interpolation (upsampling factor, also R)

Returns

None.

Remarks

Delay, d[m], defined in 0 ≤ m < M/R, with M being the number of filter coefficients in the interpolator. See the description of the *FIRStruct* structure above.

Source File

- `firinterpdelay_aa.s`

Function Profile

Device	Program Words	Cycles
PIC32A	7	32 + M/R

System resource usage

- *W0...W3* - used, not restored

- *REPEAT* instruction(s) usage – 2

5.5.7. FIRLattice

Description

`FIRLattice` uses a lattice structure implementation to apply an FIR filter to the sequence of source samples. It then places the results in the sequence of the destination samples and updates the delay values.

Prototype

```
fractional* FIRLattice (int numSamps, fractional* dstSamps,
fractional* srcSamps, FIRStruct* filter);
```

Arguments

Parameters	Description
numSamps	Number of input samples to filter (also *N; with N being multiple of R)*
dstSamps	Pointer to the destination samples (also *y*)
srcSamps	Pointer to the source samples (also *x*)
filter	Pointer to the *FIRStruct* filter structure

Return

Pointer to the base address of the destination sample.

Remarks

Number of the filter coefficients is M.

Coefficients, h[m], defined in 0 ≤ m < M, are not implemented as a circular modulo buffer.

Delay, d[m], defined in 0 ≤ m < M, are not implemented as a circular modulo buffer.

Source samples, x[n], defined in 0 ≤ n < N.

Destination samples, y[n], defined in 0 ≤ n < N.

(See also *FIRStruct*, *FIRStructInit* and *FIRDelayInit*.)

Source File

- `firlatt_aa.s`

Cycle counts for PIC32A:

Source Vector Size	Cycles if coefficients in X-mem	Cycles if coefficients in P-mem
32	6410	10154
64	12782	20266
128	25518	40490
256	50990	80938
512	101934	161834
1024	203822	323626
2048	407597	647210

*(*All values with `numCoeffs = 32`)*

System resource usage

- *W0...W7* - used, not restored

- *W8...W10* - saved, used, restored
- *ACCA* - used, not restored
- *CORCON* - saved, used, restored
- *REPEAT* instruction(s) usage – None

5.5.8. FIRLMS

Description

`FIRLMS` applies an adaptive FIR filter to the sequence of source samples, stores the results in the sequence of destination samples and updates the delay values.

The filter coefficients are also updated, at a sample-per-sample basis, using a Least Mean Square algorithm applied according to the values of the reference samples.

Prototype

```
fractional* FIRLMS (int numSamps, fractional* dstSamps, fractional* srcSamps,
FIRStruct* filter, fractional* refSamps, fractional muVal);
```

Arguments

Parameters	Description
numSamps	Number of the input samples to filter (also *N; with N being multiple of R)*
dstSamps	Pointer to the destination samples (also *y*)
srcSamps	Pointer to the source samples (also *x*)
filter	Pointer to the *FIRStruct* filter structure
refSamps	Pointer to the reference samples (also r)
muVal	Adapting factor (also mu)

Return

Pointer to the base address of the destination sample.

Remarks

Number of the filter coefficients is M.

Coefficients, h[m], defined in 0 ≤ m < M, are implemented as a circular modulo buffer.

Delay, d[m], defined in 0 ≤ m < M, is implemented as a circular modulo buffer, placed in Y-data space.

Source samples, x[n], defined in 0 ≤ n < N.

Reference samples, r[n], defined in 0 ≤ n < N.

Destination samples, y[n], defined in 0 ≤ n < N.

Adaptation:

$h_m[n] = h_m[n - 1] + mu * (r[n] - y[n]) * x[n - m]$ *for* $0 \le n < N, 0 \le m < M$.

The operation could result in saturation if the absolute value of (r[n] - y[n]) is greater than or equal to 1.

Filter coefficients must not be allocated in the program memory, because in that case, their values could not be adapted. If filter coefficients are detected as allocated in program memory, the function returns NULL.

(See also *FIRStruct*, *FIRStructInit* and *FIRInterpDelayInit*.)

Source File

- `firlms_aa.s`

Cycle counts for PIC32A:

Source Vector Size	Cycles if coefficients in X-mem
32	5586
64	11098
128	22102
256	44122
512	88150
1024	176218
2048	352342

*(*All values with `numCoeffs = 32`)*

System resource usage

- *W0...W7* - used, not restored
- *W8...W12* - saved, used, restored
- *ACCA* - used, not restored
- *CORCON* - saved, used, restored
- *MODCON* - saved, used, restored
- *XMODSTRT* - saved, used, restored
- *XMODEND* - saved, used, restored
- *YMODSTRT* - saved, used, restored
- *YMODEND* - saved, used, restored
- *REPEAT* instruction(s) usage – 1

5.5.9. FIRLMSNorm

Description

`FIRLMSNorm` applies an adaptive FIR filter to the sequence of the source samples, stores the results in the sequence of destination samples and updates the delay values.

The filter coefficients are also updated, at a sample-per-sample basis, using a Normalized Least Mean Square algorithm applied according to the values of the reference samples.

Prototype

```
fractional* FIRLMSNorm (int numSamps, fractional* dstSamps,
fractional* srcSamps, FIRStruct* filter, fractional* refSamps,
fractional muVal, fractional* energyEstimate);
```

Arguments

Parameters	Description
numSamps	Number of the input samples to filter (also *N; with N being multiple of R)*
dstSamps	Pointer to the destination samples (also *y*)
srcSamps	Pointer to the source samples (also *x*)
filter	Pointer to the *FIRStruct* filter structure
refSamps	Pointer to the reference samples (also r)

FIRLMSNorm (continued)	
Parameters	**Description**
`muVal`	Adapting factor (also mu)
`energyEstimate`	Pointer to the estimated energy (*e[N-1]*) value for the last *M* input samples

Return

Pointer to the base address of the destination sample.

Remarks

Number of the filter coefficients is M.

Coefficients, h[m], defined in 0 ≤ m < M, are implemented as a circular modulo buffer.

Delay, d[m], defined in 0 ≤ m < M, is implemented as a circular modulo buffer.

Source samples, x[n], defined in 0 ≤ n < N.

Reference samples, r[n], defined in 0 ≤ n < N.

Destination samples, y[n], defined in 0 ≤ n < N.

Adaptation:

$h_m[n] = h_m[n - 1] + nu * (r[n] - y[n]) * x[n - m]$ for 0 ≤ n < N, 0 ≤ m < M.

Where,

$$nu = \frac{mu}{mu + E[n]}$$

With,

$E[n] = E[n - 1] + (x[n])^2 - (x[n - M - 1])^2$ an estimate of input signal energy.

On start-up, *energyEstimate* should be initialized to the value of E[-1] (zero the first time the filter is invoked). Upon return, *energyEstimate* is updated to the value E[N – 1] (which may be used as the start-up value for a subsequent function call, if filtering an extension of the input signal).

The operation could result in saturation, if the absolute value of *(r[n] – y[n])* is greater than or equal to 1.

Note: Another expression for the energy estimate is: $E[n] = (x[n])^2 + (x[n - 1])^2 + ... + (x[n - M + 2])^2$

Thus, to avoid saturation while computing the estimate, the input sample values should be bound so that:

$$\sum_{m=0}^{-M+2} (x[m+n])^2 < 1$$

Filter coefficients must not be allocated in program memory, because in that case, their values could not be adapted. If filter coefficients are detected as allocated in program memory, the function returns NULL.

(See also *FIRStruct*, *FIRStructInit* and *FIRInterpDelayInit*.)

Source File

- `firlms_aa.s`

Function Profile

Device	Program Words	Cycles
PIC32A	74	See **Cycle counts for PIC32A**

Cycle counts for PIC32A:

Source Vector Size	Cycles if coefficients in X-mem
32	6360
64	12636
128	25176
256	50268
512	100440
1024	200796
2048	401496

*(*All values with `numCoeffs = 32`)*

System resource usage

- *W0...W7* - used, not restored
- *W8...W13* - saved, used, restored
- *ACCA* - used, not restored
- *CORCON* - saved, used, restored
- *MODCON* - saved, used, restored
- *XMODSTRT* - saved, used, restored
- *XMODEND* - saved, used, restored
- *YMODSTRT* - saved, used, restored
- *YMODEND* - saved, used, restored
- *REPEAT* instruction(s) usage – 2

5.5.10. FIRStructInit

Description

`FIRStructInit` initializes the values of the parameters in an *FIRStruct* FIR Filter structure.

Prototype

```
void FIRStructInit (FIRStruct* filter, int numCoeffs, fractional* coeffsBase
fractional* delayBase);
```

Arguments

Parameters	Description
filter	Pointer to the *FIRStruct* filter structure
numCoeffs	Number of the filter coefficients (also M)
coeffsBase	Base address for the filter coefficients (also h)
delayBase	Base address for delay buffer

Returns

None.

Remarks

See description of *FIRStruct* structure above.

Upon completion, *FIRStructInit* initializes the *coeffsEnd* and *delayEnd* pointers, accordingly. Also, the delay is set equal to *delayBase*.

Source File

- `firinit_aa.s`

Function Profile

Device	Program Words	Cycles
PIC32A	8	26

System resource usage

- *W0...W5* - used, not restored
- *REPEAT* instruction(s) usage – None

5.5.11. IIRCanonic

Description

`IIRCanonic` applies an IIR filter, using a cascade of canonic (direct form II) biquadratic sections, to the sequence of the source samples. It places the results in the sequence of the destination samples and updates the delay values.

Prototype

```
typedef struct {
 int numSectionsLess1;
 fractional* coeffsBase;
 fractional* delayBase;
 int initialGain;
 int finalShift;
} IIRCanonicStruct;
```

```
fractional* IIRCanonic (int numSamps, fractional* dstSamps,
fractional* srcSamps, IIRCanonicStruct* filter);
```

Arguments

Filter Structure:

Parameters	Description
numSectionsLess1	One less than the number of cascaded second order (biquadratic) sections (also S-1)
coeffsBase	Pointer to filter coefficients (also {a, b}), either within X-Data or program memory
delayBase	Pointer to filter delay (also d), only in Y-Data
initialGain	Initial gain value
finalShift	Output scaling (shift left)

Filter Description:

Parameters	Description
numSamps	The number of input samples to filter (also *N; with N being multiple of R)*
dstSamps	Pointer to the destination samples (also *y*)
srcSamps	Pointer to the source samples (also *x*)
filter	Pointer to the `IIRCanonicStruct` filter structure

Return

Pointer to the base address of the destination sample.

Remarks

There are five coefficients per second order (biquadratic) sections (generated externally) arranged in the ordered set - **{$a_2[s]$, $a_1[s]$, $a_0[s]$, $b_1[s]$, $b_0[s]$}, 0 ≤ s < S.**

The delay is made up of two words of filter state per section {$d_1[s]$, $d_2[s]$}, 0 ≤ s < S.

Source samples, x[n], defined in 0 ≤ n < N.

Destination samples, y[n], defined in 0 ≤ n < N.

The initial gain value is applied to each input sample prior to entering the filter structure.

The output scale is applied as a shift to the output of the filter structure prior to storing the result in the output sequence. It is used to restore the filter gain to 0 dB. Shift count may be zero; if not zero, it represents the number of bits to shift: negative indicates shift left, positive is shift right.

Source File

- `iircan_aa.s`

Function Profile

Device	Program Words	Cycles
PIC32A	30	See **Cycle counts for PIC32A**

Cycle counts for PIC32A:

Source Vector Size	Cycles if coefficients in X-mem	Cycles if coefficients in P-mem
32	1920	3538
64	3818	7026
128	7594	14002
256	15146	27954
512	30250	55858
1024	60458	111666
2048	120874	223282

*(*All values with `S = 5`)*

System resource usage

- *W0...W7* - used, not restored
- *W8...W9* - saved, used, restored
- *ACCA* - used, not restored
- *CORCON* - saved, used, restored
- *REPEAT* instruction(s) usage – None

5.5.12. IIRCanonicInit

Description

`IIRCanonicInit` initializes to zero the delay values in an *IIRCanonicStruct* filter structure.

Prototype

```
void IIRCanonicInit(IIRCanonicStruct* filter);
```

Arguments

Parameters	Description
filter	Pointer to the *IIRCanonicStruct* filter structure

Returns

None.

Remarks

See description of *IIRCanonic function* above.

Two words of the filter state per second order section $\{d_1[s], d_2[s]\}$, $0 \le s < S$.

Source File

- iircan_aa.s

Function Profile

Device	Program Words	Cycles
PIC32A	6	28+S

System resource usage

- *W0...W2* - used, not restored
- *REPEAT* instruction(s) usage – 1

5.5.13. IIRLattice

Description

IIRLattice uses a lattice structure implementation to apply an IIR filter to the sequence of the source samples. It then places the results in the sequence of the destination samples and updates the delay values.

Prototype

```
typedef struct {
 int order;
  fractional* kappaVals;
  fractional* gammaVals;
  fractional* delay;
} IIRLatticeStruct;
```

```
fractional* IIRLattice (int numSamps, fractional* dstSamps,
fractional* srcSamps, IIRLatticeStruct* filter);
```

Arguments

Filter Structure:

Parameters	Description
order	Filter order (also M, M ≤ N; see FIRLattice for N)
kappaVals	Base address for lattice coefficients (also k), either in X-Data or program memory
gammaVals	Base address for ladder coefficients (also g), either in X-Data or program memory. If NULL, the function will implement an all-pole filter.
delay	Base address for delay (also d), only in Y-Data

Filter Description:

Parameters	Description
numSamps	Number of input samples to filter (also *N; with N being multiple of R)*
dstSamps	Pointer to the destination samples (also *y*)
srcSamps	Pointer to the source samples (also *x*)
filter	Pointer to the `IIRLatticeStruct` filter structure

Return

Pointer to the base address of the destination sample.

Remarks

Lattice coefficients, k[m], defined in 0 ≤ m ≤ M.

Ladder coefficients, g[m], defined in 0 ≤ m ≤ M (unless if implementing an all-pole filter).

Delay, d[m], defined in 0 ≤ m ≤ M.

Source samples, x[n], defined in 0 ≤ n < N.

Destination samples, y[n], defined in 0 ≤ n < N.

Note: The fractional implementation provided with this library is prone to saturation.

Appropriately scaling the input signal, prior to applying the filter, should prevent the fractional implementation from saturating.

Source File

- `iirlatt_aa.s`

Cycle counts for PIC32A:

Source Vector Size	Cycles if coefficients in X-mem	Cycles if coefficients in P-mem
32	9586	15290
64	19122	30522
128	38194	60986
256	76338	121914
512	152626	243770
1024	305202	487482
2048	610254	774906

*(*All values with `M = 32`)*

System resource usage

- *W0...W7* - used, not restored
- *W8...W14* - saved, used, restored
- *ACCA* - used, not restored
- *CORCON* - saved, used, restored
- *REPEAT* instruction(s) usage – None

5.5.14. IIRLatticeInit

Description

`IIRLatticeInit` initializes to zero the delay values in an *IIRLatticeStruct* filter structure.

Prototype

```
void IIRLatticeInit(IIRLatticeStruct* filter);
```

Arguments

Parameters	Description
filter	Pointer to the *IIRLatticeStruct* filter structure

Returns

None.

Remarks

See description of *IIRLattice function* above.

Source File

- iirlatt_aa.s

Function Profile

Device	Program Words	Cycles
PIC32A	5	24 + M

System resource usage

- *W0...W2* - used, not restored
- *REPEAT* instruction(s) usage – 1

5.5.15. IIRTransposed

Description

`IIRTransposed` applies an IIR filter, using a cascade of transposed (direct form II) biquadratic sections, to the sequence of the source samples. It places the results in the sequence of the destination samples and updates the delay values.

Prototype

```
typedef struct {
 int numSectionsLess1;
 fractional* coeffsBase;
 fractional* delayBase1;
 fractional* delayBase2;
 int finalShift;
} IIRTransposedStruct;
```

```
fractional* IIRTransposed (int numSamps, fractional* dstSamps,
fractional* srcSamps, IIRTransposedStruct* filter);
```

Arguments

Filter Structure:

Parameters	Description
numSectionsLess1	One less than the number of cascaded second order (biquadratic) sections (also S-1)
coeffsBase	Pointer to filter coefficients (also {a, b}), either in X-Data or program memory
delayBase1	Pointer to filter state 1, with one word of delay per second order section (also d_1), only in Y-Data
delayBase2	Pointer to filter state 2, with one word of delay per second order section (also d_2), only in Y-Data
finalShift	Output scaling (shift left)

Filter Description:

Parameters	Description
numSamps	Number of input samples to filter (also *N; with N being multiple of R)*
dstSamps	Pointer to the destination samples (also *y*)
srcSamps	Pointer to the source samples (also *x*)
filter	Pointer to the `IIRTransposedStruct` filter structure

Return

Pointer to the base address of the destination sample.

Remarks

There are five coefficients per second order (biquadratic) sections (generated externally) arranged in the ordered set - **{$b_0[s]$, $b_1[s]$, $a_1[s]$, $b_2[s]$, $a_2[s]$}, 0 ≤ s < S.**

The delay is made up of two words of the filter state per section {$d_1[s]$, $d_2[s]$}, 0 ≤ s < S.

Source samples, x[n], defined in 0 ≤ n < N.

Destination samples, y[n], defined in 0 ≤ n < N.

The output scale is applied as a shift to the output of the filter structure prior to storing the result in the output sequence. It is used to restore the filter gain to 0 dB. The shift count may be zero; if not zero, it represents the number of bits to shift: negative indicates shift left and positive is shift right.

Source File

- `iirtrans_aa.s`

Cycle counts for PIC32A:

Source Vector Size	Cycles if coefficients in X-mem	Cycles if coefficients in P-mem
32	2508	4112
64	4972	8176
128	9900	16304
256	19756	32560
512	39468	65072
1024	78892	130096
2048	157740	260144

*(*All values with `S = 5`)*

System resource usage

- *W0...W7* - used, not restored
- *W8...W10* - saved, used, restored
- *ACCA* - used, not restored
- *CORCON* - saved, used, restored
- *REPEAT* instruction(s) usage – None

5.5.16. IIRTransposedInit

Description

`IIRTransposedInit` initializes to zero the delay values in an *IIRTransposedStruct* filter structure.

Prototype

```
void IIRTransposedInit (IIRTransposedStruct* filter);
```

Arguments

Parameters	Description
filter	Pointer to the *IIRTransposedStruct* filter structure

Returns

None.

Remarks

The delay is made up of two independent buffers, each buffer containing one word of filter state per section $\{d_2[s], d_1[s]\}$, $0 \le s < S$.

See the description of *IIRTransposed function*.

Source File

- `iirtrans_aa.s`

Function Profile

Device	Program Words	Cycles
PIC32A	8	18 + 2*S

System resource usage

- *W0...W3* - used, not restored
- *REPEAT* instruction(s) usage – 2

6. Transform Functions

Function	Description
BitReverseComplex	Reorganizes, in place, the elements of a complex vector in bit-reverse order.
CosFactorInit	Generates the first half of the set of cosine factors required by a Type II Discrete Cosine Transform and places the result in the complex destination vector.
DCT	Computes the Discrete Cosine Transform of a source vector and stores the results in the destination vector.
DCTIP	Computes the Discrete Cosine Transform of a source vector in place.
FFTComplex	Computes the Fast Fourier Transform of a source complex vector and stores the results in the destination complex vector.
FFTComplexIP	Computes the Fast Fourier Transform of a source complex vector in place.
IFFTComplex	Computes the Inverse Fast Fourier Transform of a source complex vector and stores the results in the destination complex vector.
IFFTComplexIP	Computes the Inverse Fast Fourier Transform of a source complex vector in place.
FFTRealIP	Performs an in-place Fast Fourier Transform (FFT) on a real-valued source vector. It utilizes an efficient algorithm that computes the FFT of a 2N-point real vector by leveraging an N-point complex FFT, supplemented with additional operations known as split functions.
FFTReal	Performs Fast Fourier Transform (FFT) on a real-valued source vector and stores the result in a complex destination vector. It utilizes an efficient algorithm that computes the FFT of a 2N-point real vector by leveraging an N-point complex FFT, supplemented with additional operations known as split functions.
IFFTRealIP	Computes, in place, the inverse Fast Fourier Transform of a source complex vector, which was derived out a real-vector using the `FFTReal` function. The algorithm to compute IFFT is based on the efficient computation of IFFT of a 2N point complex vector using the N point complex FFT with additional computations called split functions.
IFFTReal	Computes the inverse Fast Fourier Transform of a source complex vector, which was derived out of a real-vector using the `FFTReal` function. The algorithm to compute IFFT is based on the efficient computation of IFFT of a 2N point complex vector using the N point complex FFT with additional computations called split functions.
SquareMagnitudeComplex	Computes the squared magnitude of each element in a complex source vector.
TwidFactorInit	Generates the first half of the set of twiddle factors required by a Discrete Fourier Transform or Discrete Cosine Transform, and places the result in the complex destination vector.

6.1. Fractional Transform Operations

A fractional transform is a linear, time invariant, discrete operation that when applied to a fractional time domain sample sequence, results in a fractional frequency in the frequency domain. Conversely, an inverse fractional transform operation, when applied to frequency domain data, results in its time domain representation.

A set of transforms (and a subset of inverse transforms) are provided by the DSP Library. The first set applies a Discrete Fourier Transform (or its inverse) to a complex/real data set. The second set applies a Type II Discrete Cosine Transform (DCT) to a real valued sequence. These transforms have been designed to either operate out-of-place or in-place. The former type populates an output sequence with the results of the transformation. In the latter, the input sequence is (physically) replaced by the transformed sequence. For out-of-place operations, enough memory to accept the results of the computation must be provided.

The transforms make use of factors (or constants) which must be supplied to the transforming function during its initialization. These transform factors, which are complex data sets, are computed in floating-point arithmetic and then transformed into fractionals for use by the operations. To avoid excessive computational overhead when applying a transformation, a

particular set of transform factors could be generated once and used many times during the execution of the program. Thus, it is advisable to store the factors returned by any of the initialization operations in a permanent (static) complex vector. It is also advantageous to generate the factors off-line, place them in program memory and use them when the program is later executing. This way, not only cycles, but also RAM memory are saved when designing an application which involves transformations.

6.2. Fractional Complex Vectors

A complex data vector is represented by a data set in which every pair of values represent an element of the vector. The first value in the pair is the real part of the element, and the second its imaginary part. Both the real and imaginary parts are stored in memory using one word for each and must be interpreted as 1.31 fractionals. As with the fractional vector, the fractional complex vector stores its elements consecutively in memory.

The organization of data in a fractional complex vector may be addressed by the following data structure:

```
#ifdef fractional
   #ifndef fractcomplex
       typedef struct {
          fractional real;
          fractional imag;
       } fractcomplex;
   #endif
#endif
```

6.3. User Considerations

When using transform functions, consider the following:

1. No boundary checking is performed by these functions. Out of range sizes (including zero length vectors) as well as nonconforming use of source complex vectors and factor sets may produce unexpected results.
2. It is recommended that the STATUS Register (SR) is examined after the completion of each function call. In particular, users can inspect the SA, SB and SAB flags after the function returns to determine if saturation occurred.
3. The input and output complex vectors involved in the family of transformations must be allocated in Y-Data memory. Transform factors may be allocated either in X-Data or program memory.
4. Because Bit-Reverse Addressing requires the vector set to be modulo aligned, the input and output complex vectors in operations using either explicitly or implicitly the *BitReverseComplex* function must be properly allocated.
5. Operations which return a destination complex vector can be nested. For instance, if:
   a = Op1 (b, c), with b = Op2 (d), and c = Op3 (e, f), then
   a = Op1 (Op2 (d), Op3 (e, f)).
6. All cycle count values for PIC32A are measured with PBU cache enabled and may differ depending on the status of PBU cache or on the placement of vectors and code.

6.4. Functions

6.4.1. BitReverseComplex

Description

`BitReverseComplex` reorganizes, in place, the elements of a complex vector in bit-reverse order.

Prototype

```
fractcomplex* BitReverseComplex (int log2N, fractcomplex* srcCV);
```

Arguments

Parameters	Description
log2N	Base 2 logarithm of N (N = number of complex elements in source vector)
srcCV	Pointer to source complex vector

Returns

Pointer to the base address of the source complex vector.

Remarks

N must be an integer power of 2.

The srcCV vector must be allocated at a modulo alignment of N.

This function operates in place.

Source File

- `bitrev_aa.s`

Function Profile

	PIC32A
Program Words	18

	Transform Size	PIC32A
Cycle count	32	306
	64	594
	128	1154
	256	2306
	512	4578
	1024	9186
	2048	18338

System resource usage

- *W0..W6* - used, not restored
- *XBREV* - saved, used, restored
- *MODCON* - saved, used, restored
- *REPEAT* instruction usage – None

6.4.2. CosFactorInit

Description

`CosFactorInit` generates the first half of the set of cosine factors required by a Type II Discrete Cosine Transform and places the result in the complex destination vector.

Effectively, the set contains the values:

$$CN[k] = e^{\frac{k\pi i}{2N}}, \quad for\ 0 \le k < \frac{N}{2}$$

Prototype

```
fractcomplex* CosFactorInit (int log2N, fractcomplex* cosFactors);
```

Arguments

Parameters	Description
log2N	Based 2 logarithm of N (N = number of complex factors needed by a DCT)
cosFactors	Pointer to complex cosine factors

Returns

Pointer to the base address of the cosine factors.

Remarks

N must be an integer power of 2.

Only the first N/2 cosine factors are generated.

A complex vector of size N/2 must have already been allocated and assigned to cosFactors prior to invoking the function. The complex vector should reside in X-Data memory.

Factors are computed in floating-point arithmetic and converted to 1.31 complex fractionals.

Source File

- initcosf_aa.c

6.4.3. DCT

Description

DCT computes the Discrete Cosine Transform of a source vector.

Prototype

```
fractcomplex* DCT (int log2N, fractional* dstCV, fractcomplex* srcCV,
fractcomplex* cosFactors, fractcomplex* twidFactors);
```

Arguments

Parameters	Description
log2N	Base 2 logarithm of N (number of complex elements in source vector)
dstCV	Pointer to the destination vector
srcCV	Pointer to the source vector
cosFactors	Pointer to the cosine factors
twidFactors	Pointer to the twiddle factors

Return

Pointer to the base address of the destination sample.

Remarks

This function internally uses *DCTIP* and the *VectorZeroPad* function.

N must be an integer power of 2.

A vector of size 2N elements must already have been allocated and assigned to *dstCV*.

The *dstCV* vector must be allocated in the Y-Data space with address alignment to a modulo of N.

The results of the computation are stored in the first N elements of destination vector.

To avoid saturation (overflow) during computation, the values of the source vector should be in the range [-0.5, 0.5].

Only the first N/2 cosine factors are needed.

Only the first N/2 twiddle factors are needed.

The twiddle factors must be initialized with *conjFlag* set to a value different than zero.

Output is scaled by the factor of N.

Source File

- `dctoop_aa.s`

Function Profile

Program Words	64
Cycle Count	33

Notes:

1. The above-mentioned program word and cycle counts pertains solely to *DCT*. However, as this function inherently utilizes *DCTIP* and *VectorZeroPad*, the respective counts for *DCTIP* and *VectorZeroPad* must also be considered.
2. In the description of *DCTIP* and *VectorZeroPad*, the number of cycles reported includes four cycles of C-function call overhead. Thus, the number of actual cycles from *DCTIP* and *VectorZeroPad* to add to *DCT* is 2x4 less than whatever number is reported for a stand-alone *DCTIP/VectorZeroPad*.

System resource usage

The below system resource usages exclude that of *DCTIP* and *VectorZeroPad.*

- *W0...W5* - used, not restored
- *REPEAT* instruction usage - None

6.4.4. DCTIP

Description

`DCTIP` computes the Discrete Cosine Transform of a source vector in place.

Prototype

```
fractcomplex* DCTIP (int log2N, fractcomplex* srcCV, fractcomplex*
cosFactors, fractcomplex* twidFactors);
```

Arguments

Parameters	Description
`log2N`	Base 2 logarithm of N (number of complex elements in source vector)
`srcCV`	Pointer to the source vector
`cosFactors`	Pointer to the cosine factors
`twidFactors`	Pointer to the twiddle factors

Return

Pointer to the base address of the destination sample.

Remarks

N must be an integer power of 2.

This function expects that the source vector has been zero padded to length 2N.

The *srcCV* vector must be allocated in the Y-Data space with address alignment to a modulo of N.

The results of computation are stored in the first N elements of the source vector.

To avoid saturation (overflow) during computation, the values of the source vector should be in the range [-0.5, 0.5].

Only the first N/2 cosine factors are needed.

Only the first N/2 twiddle factors are needed.

The twiddle factors must be initialized with *conjFlag* set to a value different than zero.

Output is scaled by the factor of N.

Source File

- `dct_aa.s`

Function Profile

Program Words	PIC32A
	64

	Transform Size	PIC32A	
		Cycles if Twiddle Factors in X-mem	Cycles if Twiddle Factors in P-mem
Cycle count	32	2,468	3,556
	64	5,350	7,926
	128	11,668	17,636
	256	25,478	39,062
	512	55,364	85,844
	1024	119,798	187,398
	2048	257,812	406,308

System resource usage

- *W0...W7* - used, not restored
- *W8...W11* - saved, used, restored
- *ACCA* - used, not restored
- *CORCON* - saved, used, restored
- *REPEAT* instruction(s) usage – 1

6.4.5. FFTComplex

Description

`FFTComplex` computes the Fast Fourier Transform of a source complex vector and stores the results in the destination complex vector.

Prototype

```
fractcomplex* FFTComplex(int log2N, fractcomplex* dstCV, fractcomplex* srcCV,
fractcomplex* twidFactors);
```

Arguments

Parameters	Description
log2N	Base 2 logarithm of N (number of complex elements in source vector)
dstCV	Pointer to the destination vector
srcCV	Pointer to the source vector
twidFactors	Pointer to twiddle factors

Return

Pointer to the base address of the destination sample.

Remarks

N must be an integer power of 2.

This function internally calls *VectorCopy*, *FFTComplexIP* and *BitReversal* functions.

This function operates out of place. A complex vector, large enough to receive the results of the operation, must already have been allocated and assigned to *dstCV*.

The elements in the source complex vector are expected in a natural order, and the resulting transform in the destination vector is stored back in a natural order.

The *dstCV* vector must be allocated in the Y-Data space with address alignment to a modulo of N.

To avoid saturation (overflow) during computation, the values of the source vector should be in the range [-0.5, 0.5].

Only the first N/2 twiddle factors are needed.

The twiddle factors must be initialized with the *conjFlag* set to zero.

The output is scaled by the factor of N.

Source File

- `fftoop_aa.s`

Function Profile

Program Words	9
Cycle Count	36

Notes:

1. The above-mentioned program word and cycle counts pertain solely to *FFTComplex*. However, as this function inherently utilizes *VectorCopy, FFTComplexIP* and *BitReversalComplex,* the respective counts for these functions must also be considered.
2. In the description of *VectorCopy, FFTComplexIP* and *BitReversalComplex*, the number of cycles reported includes four cycles of C-function call overhead. Thus, the number of actual cycles from these function to add to *FFTComplex* is 3x4 less than whatever number is reported for a stand-alone *FFTComplex.*

System resource usage

The below system resource usages exclude that of *VectorCopy, FFTComplexIP* and *BitReversalComplex.*

- *W0...W4* - used, not restored
- *REPEAT* instruction usage - None

6.4.6. FFTComplexIP

Description

`FFTComplexIP` computes the Fast Fourier Transform of a source complex vector in place.

Prototype

```
fractcomplex* FFTComplexIP (int log2N, fractcomplex* srcCV, fractcomplex*
twidFactors);
```

Arguments

Parameters	Description
`log2N`	Base 2 logarithm of N (number of complex elements in source vector)
`srcCV`	Pointer to the source vector
`twidFactors`	Pointer to twiddle factors

Return

Pointer to the base address of the destination sample.

Remarks

N must be an integer power of 2.

The elements in the source complex vector are expected in a natural order. The resulting transform is stored in bit-reverse order.

The *srcCV* vector must be allocated in the Y-Data space with address alignment to a modulo of N.

To avoid saturation (overflow) during computation, the values of the source vector should be in the range [-0.5, 0.5].

Only the first N/2 twiddle factors are needed.

The twiddle factors must be initialized with *conjFlag* set to zero.

Output is scaled by the factor of N.

Source File

- `fft_aa.s`

Function Profile

Program Words	PIC32A
	46

	Transform Size (N)	PIC32A	
		Cycles if Twiddle Factors in X-mem	Cycles if Twiddle Factors in P-mem
Cycle count	32	1,780	2,318
	64	4,060	5,356
	128	9,188	12,228
	256	20,588	27,564
	512	45,684	61,428
	1024	100,476	135,548
	2048	219,268	296,580

System resource usage

- *W0...W7* - used, not restored

- *W8...W14* - saved, used, restored
- *ACCA/ACCB* - used, not restored
- *CORCON* - saved, used, restored
- *REPEAT* instruction(s) usage – 1

6.4.7. IFFTComplex

Description

`IFFTComplex` computes the inverse Fast Fourier Transform of a source complex vector and stores the results in the destination complex vector.

Prototype

```
fractcomplex* IFFTComplex(int log2N, fractcomplex* dstCV, fractcomplex*
srcCV, fractcomplex* twidFactors);
```

Arguments

Parameters	Description
`log2N`	Base 2 logarithm of N (number of complex elements in source vector)
`dstCV`	Pointer to the destination vector
`srcCV`	Pointer to the source vector
`twidFactors`	Pointer to twiddle factors

Return

Pointer to the base address of the destination sample.

Remarks

N must be an integer power of 2.

This function internally calls *VectorCopy* and *IFFTComplexIP* functions.

This function operates out of place. A complex vector, large enough to receive the results of the operation, must already have been allocated and assigned to *dstCV*.

The source array is expected to be in a natural order. Similarly, the resultant array will be stored back in a natural order as well.

The *dstCV* vector must be allocated in the Y-Data space with address alignment to a modulo of N.

To avoid saturation (overflow) during computation, the values of the source vector should be in the range [-0.5, 0.5].

Only the first N/2 twiddle factors are needed.

The twiddle factors must be initialized with *conjFlag* set to a value other than zero.

Output is scaled by the factor of N.

Source File

- `ifftoop_aa.s`

Function Profile

Program Words	10
Cycle Count	30

Notes:

1. The above-mentioned program word and cycle counts pertain solely to *IFFTComplex*. However, as this function inherently utilizes *VectorCopy* and *IFFTComplexIP*, the respective counts for these functions must also be considered.
2. In the description of *VectorCopy* and *IFFTComplexIP*, the number of cycles reported includes four cycles of C-function call overhead. Thus, the number of actual cycles from these functions to add to *IFFTComplex* is 2x4 less than whatever number is reported for a stand-alone *IFFTComplex*.

System resource usage

The following system resource usages exclude that of *VectorCopy* and IFFTComplexIP.

- *W0...W4* - used, not restored
- *REPEAT* instruction usage - None

6.4.8. IFFTComplexIP

Description

`IFFTComplexIP` computes the inverse Fast Fourier Transform of a source complex vector in place.

Prototype

```
fractcomplex* IFFTComplexIP(int log2N, fractcomplex* srcV, fractcomplex*
twidFactors);
```

Arguments

Parameters	Description
`log2N`	Base 2 logarithm of N (number of complex elements in source vector)
`srcV`	Pointer to the source vector
`twidFactors`	Pointer to twiddle factors

Return

Pointer to the base address of the destination sample.

Remarks

N must be an integer power of 2.

This function internally calls *BitReversalComplex* and *FFTComplexIP* functions.

This function operates in place.

The elements in the source complex vector are expected in a natural order, and the resulting transform will be stored back in natural order.

The *dstV* vector must be allocated in the Y-Data space with address alignment to a modulo of N.

To avoid saturation (overflow) during computation, the values of the source vector should be in the range [-0.5, 0.5].

Only the first N/2 twiddle factors are needed.

The twiddle factors must be initialized with the *conjFlag* set to a value other than zero.

Output is scaled by the factor of N.

Source File

- `ifft_aa.s`

Function Profile

Program Words	6
Cycle Count	27

Notes:

1. The above-mentioned program word and cycle counts pertains solely to `IFFTComplexIP`. However, as this function inherently utilizes `BitReversalComplex` and `FFTComplexIP`, the respective counts for these functions must also be considered.
2. In the description of `BitReversalComplex` and `FFTComplexIP`, the number of cycles reported includes four cycles of C-function call overhead. Thus, the number of actual cycles from these function to add to `IFFTComplexIP` is 2x4 less than whatever number is reported for a stand-alone `IFFTComplexIP`.

System resource usage

The below system resource usages exclude that of *BitReversalComplex* and *FFTComplexIP.*

- *W0, W1* - used, not restored
- *REPEAT* instruction usage - None

6.4.9. FFTRealIP

Description

`FFTRealIP` performs an in-place Fast Fourier Transform (FFT) on a real-valued source vector. It utilizes an efficient algorithm that computes the FFT of a 2N-point real vector by leveraging an N-point complex FFT, supplemented with additional operations known as split functions.

Prototype

```
fractcomplex* FFTRealIP (int log2N, fractional* srcV, fractcomplex*
twidFactors);
```

Arguments

Parameters	Description
`log2N`	Base 2 logarithm of N (number of complex elements in source vector)
`srcV`	Pointer to the real source vector
`twidFactors`	Pointer to the complex twiddle factors

Return

Pointer to the base address of the complex destination sample.

Remarks

N must be an integer power of 2.

The elements in the source complex vector are expected in a natural order, and the resultant vector, likewise, returned in a natural order. The resulting transform is a complex vector of size N/2 + 1 stored in the same location as that of the source vector. Since the second half of the resulting transform is a conjugate of first half, only N/2 complex elements are returned.

This function operates in-place. *srcV* must be a non-complex vector with N elements. The additional space of two words must be allocated to *srcV* to hold the $(N/2)^{th}$ element of the destination complex vector.

The *srcV* vector must be allocated in the Y-Data space with address alignment to a modulo of N.

To avoid saturation (overflow) during computation, the values of the source vector should be in the range [-0.5, 0.5].

Only the first N/2 twiddle factors are needed.

This function internally utilizes the *BitReversal* function.

The twiddle factors must be initialized with the *conjFlag* set to zero.

Output is scaled by the factor of N.

Source File

- `rfft_aa.s`

Function Profile

	PIC32A
Program Words	`115 + program word counts of BitReversalComplex function`

	Transform Size (N)	PIC32A	
		Cycles if Twiddle Factors in X-mem	Cycles if Twiddle Factors in P-mem
Cycle count	32	1,198	1,692
	64	2,551	3,726
	128	5,559	8,328
	256	12,122	18,587
	512	26,418	40,824
	1024	57,319	89,418
	2048	123,684	194,600

Note: The above cycle count values include cycles of the *BitReversal* function that is internally used.

System resource usage

- *W0...W7* - used, not restored
- *W8...W14* - saved, used, restored
- *ACCA/ACCB* - used, not restored
- *CORCON* - saved, used, restored
- *REPEAT* instruction(s) usage – 1
- Plus resources used by the *BitReversalComplex* function

6.4.10. FFTReal

Description

`FFTReal` performs Fast Fourier Transform (FFT) on a real-valued source vector and stores the result in a complex destination vector. It utilizes an efficient algorithm that computes the FFT of a 2N-point real vector by leveraging an N-point complex FFT, supplemented with additional operations known as split functions.

Prototype

```
fractcomplex* FFTReal (int log2N, fractional* srcV, fractcomplex* dstCV,
fractcomplex* twidFactors);
```

Arguments

Parameters	Description
log2N	Base 2 logarithm of N (number of complex elements in source vector)
srcV	Pointer to the real source vector
dstCV	Pointer to the destination complex vector
twidFactors	Pointer to complex twiddle factors

Return

Pointer to the base address of the complex destination sample.

Remarks

N must be an integer power of 2.

srcV must be a non-complex vector with N elements.

dstCV must be a complex vector of size N/2 + 1.

The elements in the source complex vector are expected in a natural order, and the resultant vector, likewise, returned in a natural order. The resulting transform is a complex vector of size N/2 + 1. Since the second half of the resulting transform is a conjugate of the first half, only N/2 complex elements are returned.

The *srcV* vector must be allocated at a modulo alignment of N in y-memory space.

To avoid saturation (overflow) during computation, the values of the source vector should be in the range [-0.5, 0.5].

Only the first N/2 twiddle factors are needed.

This function internally utilizes the *VectorCopy* and *FFTRealIP* functions.

The twiddle factors must be initialized with the *conjFlag* set to zero.

Output is scaled by the factor of N.

Source File

- rfft_aa.s

Function Profile

	Program Words	Cycle count
PIC32A	10 + program word counts of *FFTRealIP* and *VectorCopy* functions.	Cycle counts of *VectorCopy* + *FFTRealIP* + 16

Note: The cycle counts of *VectorCopy* and *FFTRealIP*, in their respective sections, include function call and return overheads. Hence, ~4 cycle from each of these will have to be subtracted while calculating the total cycle counts of the *FFTReal* function.

System resource usage

- *W0...W3* - used, not restored
- Plus resources used by *VectorCopy* and *FFTRealIP* functions

6.4.11. IFFTRealIP

Description

IFFTRealIP computes, in place, the inverse Fast Fourier Transform of a source complex vector, which was derived out of a real-vector using the FFTReal function. The algorithm to compute IFFT is

based on the efficient computation of IFFT of a 2N point complex vector using the N point complex FFT with additional computations called split functions.

Prototype

```
fractional* IFFTRealIP (int log2N, fractcomplex* srcCV, fractcomplex*
twidFactors);
```

Arguments

Parameters	Description
log2N	Base 2 logarithm of N (number of complex elements in source vector)
srcCV	Pointer to the complex source vector
twidFactors	Pointer to complex twiddle factors

Return

Pointer to the base address of the real destination sample.

Remarks

N must be an integer power of 2.

The complex srcCV vector must be of size N/2 + 1, holding zero to N/2 elements.

The elements in the source complex vector are expected in a natural order, and the resultant vector, likewise, returned in a natural order. The resulting transform is a real vector of size N stored in the same location as that of the source complex vector.

This function operates in-place. *srcV* must be a complex vector with N/2 + 1 elements. The resultant vector will be a real-vector of size *N* stored in same location as that of complex *srcCV* vector.

The *dstCV* vector must be allocated at a modulo alignment of N in y-memory space.

To avoid saturation (overflow) during computation, the values of the source vector should be in the range [-0.5, 0.5].

Only the first N/2 twiddle factors are needed.

This function internally utilizes the *BitReversal* function and parts of the *FFTReal* function.

The twiddle factors must be initialized with *conjFlag* set to a value other than zero.

Output is scaled by the factor of N.

Source File

- irfft_aa.s

Function Profile

	PIC32A
Program Words	68 + program word counts of *BitReversalComplex* functions

	Transform Size (N)	PIC32A	
		Cycles if Twiddle Factors in X-mem	Cycles if Twiddle Factors in P-mem
Cycle count	32	1,229	1,712
	64	2,599	3,770
	128	5,629	8,390
	256	12,250	18,618
	512	26,768	41,160
	1024	57,970	90,095
	2048	125,050	195,958

Note: The above cycle count values include cycles of *BitReversalComplex* functions that are internally used.

System resource usage

- *W0...W7* - used, not restored
- *W8...W9* - saved, used, restored
- *ACCA/ACCB* - used, not restored
- *CORCON* - saved, used, restored
- *REPEAT* instruction(s) usage – 1
- Plus resources used by the *BitReversalComplex* function.

6.4.12. IFFTReal

Description

`IFFTReal` computes the inverse Fast Fourier Transform of a source complex vector, which was derived out of a real-vector using the FFTReal function. The algorithm to compute IFFT is based on the efficient computation of IFFT of a 2N point complex vector using the N point complex FFT with additional computations called split functions.

Prototype

```
fractional * IFFTReal (int log2N, fractcomplex* srcCV, fractional* dstV,
fractcomplex* twidFactors);
```

Arguments

Parameters	Description
`log2N`	Base 2 logarithm of N (number of complex elements in the source vector)
`srcCV`	Pointer to the complex source vector of size N/2 + 1
`dstV`	Pointer to the destination real vector of size N
`twidFactors`	Pointer to complex twiddle factors.

Return

Pointer to the base address of the complex destination sample.

Remarks

N must be an integer power of 2.

srcCV must be a complex vector with N/2 + 1 elements.

dstV must be a real vector of size N.

The elements in the source complex vector are expected in a natural order and the resultant vector likewise returned in a natural order. The resulting transform is a real vector of size N.

The *dstV* vector must be allocated at a modulo alignment of N in y-memory space.

To avoid saturation (overflow) during computation, the values of the source vector should be in the range [-0.5, 0.5].

Only the first N/2 twiddle factors are needed.

This function internally utilizes *VectorCopy* and *IFFTRealIP* functions.

The twiddle factors must be initialized with *conjFlag* set to a value other than zero.

Output is scaled by the factor of N.

Source File

- `irfft_aa.s`

Function Profile

PIC32A	Program Words	Cycle count
	11	Cycle counts of *VectorCopy* + *IFFTRealIP* + 17

Note: The cycle counts of *VectorCopy* and *FFTRealIP*, in their respective sections, include function call and return overheads. Hence, ~4 cycle from each of these will have to be subtracted while calculating the total cycle counts of the *FFTReal* function.

System resource usage

- *W0...W3* - used, not restored
- Plus resources used by *VectorCopy* and *IFFTRealIP* functions.

6.4.13. SquareMagnitudeComplex

Description

`SquareMagnitudeCplx` computes the squared magnitude of each element in a complex source vector.

Prototype

```
fractional* SquareMagnitudeCplx (numElems, fractcomplex* srcV,
fractional* dstV);
```

Arguments

Parameters	Description
`numElems`	Number of complex elements in the source vector
`srcV`	Pointer to the fractcomplex source vector
`dstV`	Pointer to the fractional destination vector

Return

Pointer to the base address of the destination sample.

Remarks

If the sum of squares of the real and imaginary parts of a complex element in the source vector is larger than the max value of supported fractional data ($1 - 2^{-31}$), this operation results in saturation.

This function can be used to operate in-place on a source data set.

Source File

- `cplxsqrmag_aa.s`

Function Profile

	PIC32A
Program Words	15
Cycle Count	34 + 3.5(*numElems*)

System resource usage

- *W0...W4* - used, not restored
- *W13* - saved, used, restored
- *ACCA/ACCB* - used, not restored
- *REPEAT* instruction usage – None

6.4.14. TwidFactorInit

Description

`TwidFactorInit` generates the first half of the set of twiddle factors required by a Discrete Fourier Transform or Discrete Cosine Transform, and places the result in the complex destination vector.

Effectively, the set contains the values:

- For `conjFlag` = 0:

$$w[k] = e^{\frac{-k \times 2\pi}{N}}, \quad for\ 0 \le k < \frac{N}{2}$$

- For `conjFlag` != 0:

$$w[k] = e^{\frac{k \times 2\pi}{N}}, \quad for\ 0 \le k < \frac{N}{2}$$

Prototype

```
fractcomplex* TwidFactorInit (int log2N, fractcomplex* twidFactors, int
conjFlag);
```

Arguments

Parameters	Description
log2N	Based 2 logarithm of N (N = number of complex factors needed by a FFT)
`twidFactors`	Pointer to the complex twiddle factors
conjFlag	Flag to indicate whether or not conjugate values are to be generated

Returns

Pointer to the base address of twiddle factors.

Remarks

N must be an integer power of 2.

Only the first N/2 twiddle factors are generated.

The value of *conjFlag* determines the sign of the exponent in the Fourier transform's exponential function. For Forward Fourier transforms, *conjFlag* should be set to 0. For inverse Fourier transforms and Discrete Cosine Transforms (DCT), *conjFlag* must be set to 1 to ensure a correct computation.

A complex vector of size N/2 must have already been allocated and assigned to *twidFactors* prior to invoking the function. The complex vector should be allocated in X-Data memory.

Factors computed in floating-point arithmetic and converted to 1.31 complex fractionals.

Source File

- `inittwid_aa.c`

7. Control Functions

Function	Description
PIDInit	Clears the delay line elements in the 3-element array located in Y-Data space and pointed to by `controlHistory`. It also clears the current PID output element, `controlOutput`.
PIDCoeffCalc	`PIDCoeffCalc` computes the PID coefficients based on values of Kp, Ki and Kd provided by the user.
PID	Computes the `controlOutput` element of the structure `tPID`.

7.1. Proportional Integral Derivative (PID) Control

This section describes functions provided in the DSP library that aid the implementation of closed-loop control systems. A complete discussion of Proportional Integral Derivative (PID) controllers is beyond the scope of this documentation, but this section provides some guidelines for tuning PID controllers.

7.1.1. PID Controller Background

A PID controller responds to an error signal in a closed control loop and attempts to adjust the controlled quantity in order to achieve the desired system response. The controlled parameter can be any measurable system quantity, such as speed, voltage or current. The output of the PID controller can control one or more system parameters that will affect the controlled system quantity. For example, a speed control loop in a Sensorless Brushless DC motor application can control the PWM duty cycle directly, or it can set the current demand for an inner control loop that regulates the motor currents. The benefit of the PID controller is that it can be adjusted empirically by adjusting one or more gain values and observing the change in system response.

A digital PID controller is executed at a periodic sampling interval, and it is assumed that the controller is executed frequently enough so that the system can be properly controlled. For example, the current controller in the Sensorless Brushless DC motor application is executed in every PWM cycle, since the motor can change very rapidly. The speed controller in such an application is executed at the medium event rate (100 Hz) because motor speed changes will occur relatively slowly due to mechanical time constants.

The error signal is formed by subtracting the desired setting of the parameter to be controlled from the actual measured value of that parameter. This sign of the error indicates the direction of change required by the control input.

The Proportional (P) term of the controller is formed by multiplying the error signal by a P gain. This will cause the PID controller to produce a control response that is a function of the error magnitude. As the error signal becomes larger, the P term of the controller becomes larger to provide more correction.

The effect of the P term will tend to reduce the overall error as time elapses. However, the effect of the P term will reduce as the error approaches zero. In most systems, the error of the controlled parameter will get very close to zero but will not converge. The result is a small remaining steady state error. The Integral (I) term of the controller is used to fix small steady state errors. The I term takes a continuous running total of the error signal. Therefore, a small steady state error will accumulate into a large error value over time. This accumulated error signal is multiplied by an I gain factor and becomes the I output term of the PID controller.

The Differential (D) term of the PID controller is used to enhance the speed of the controller and responds to the rate of change of the error signal. The D term input is calculated by subtracting the present error value from a prior value. This delta error value is multiplied by a D gain factor that becomes the D output term of the PID controller. The D term of the controller produces more control output the faster the system error is changing.

It should be noted that not all PID controllers will implement the D or, less commonly, the I terms. For example, the speed controller in a Brushless DC motor application described in Microchip Application Note AN901 does not have a D term due to the relatively slow response time of motor speed changes. In this case, the D term could cause excessive changes in PWM duty cycle that could affect the operation of the sensorless algorithm and produce overcurrent trips.

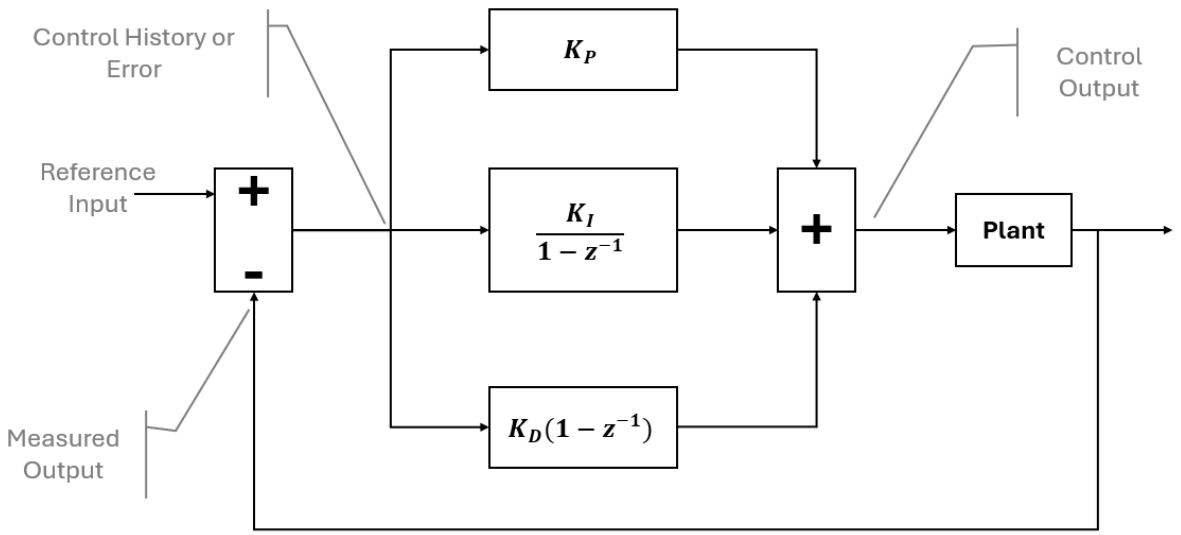
7.1.2. Adjusting PID Gains

The P gain of a PID controller will set the overall system response. When first tuning a controller, the I and D gains should be set to zero. The P gain can then be increased until the system responds well to set point changes without excessive overshoot or oscillations. Using lower values of P gain will loosely control the system, while higher values will give tighter control. At this point, the system will probably not converge to the set point.

After a reasonable P gain is selected, the I gain can be slowly increased to force the system error to zero. Only a small amount of I gain is required in most systems. Note that the effect of the I gain, if large enough, can overcome the action of the P gain, slow the overall control response and cause the system to oscillate around the set point. If this occurs, reducing the I gain and increasing the P gain will usually solve the problem.

After the P and I gains are set, the D gain can be set. The D term will speed up the response of control changes, but it should be used sparingly because it can cause very rapid changes in the controller output. This behavior is called set point kick. The set point kick occurs because the difference in system error becomes instantaneously very large when the control set point is changed. In some cases, damage to system hardware can occur. If the system response is acceptable with the D gain set to zero, then omit the D gain.

Figure 7-1. PID Control System



7.1.3. *PID Library Functions and Data Stuctures*

The DSP library provides a PID Controller function, `PID ( tPID* )`, to perform a PID operation. The function uses a data structure defined in the header file `dsp.h`, which has the following form:

```
typedef struct {
 fractional* abcCoefficients;
 fractional* controlHistory;
 fractional controlOutput;
 fractional measuredOutput;
```

```
  fractional controlReference;
} tPID;
```

Prior to invoking the `PID()` function, the application should initialize the data structure of type `tPID`. This is done in the following steps:

1. Calculate Coefficients from PID Gain Values. The element `abcCoefficients` in the data structure of type tPID is a pointer to A, B and C coefficients located in X-Data space. These coefficients are derived from the PID gain values, Kp, Ki and Kd, shown in Figure 7-1, as follows: `A = Kp + Ki + Kd B = -(Kp + 2*Kd) C = Kd` . To derive the `A, B` and `C` coefficients, the DSP library provides a function, `PIDCoeffCalc`.
2. Clear the PID State Variables. The structural element `controlHistory` is a pointer to a history of three samples located in Y-Data space, with the first sample being the most recent (current). These samples constitute a history of current and past differences between the Reference Input and the Measured Output of the plant function. The `PIDInit` function clears the elements pointed to by `controlHistory`. It also clears the `controlOutput` element in the `tPID` data structure.

7.2. Functions

7.2.1. PIDInit

Description

`PIDInit` clears the delay line elements in the three-element array located in Y-Data space and pointed to by *controlHistory*. It also clears the current PID output element, *`controlOutput`*.

Prototype

`void PIDInit ( tPID *fooPIDStruct);`

Arguments

Parameters	Description
`fooPIDStruct`	A pointer to a PID data structure of type `tPID`

Returns

None

Remarks

None

Source File

- `pid_aa.s`

Function Profile

Device	Program Words	Cycles
PIC32A	6	10

System resource usage

- – *W0* - used, not restored.
  – *REPEAT* instruction usage – None

7.2.2. PIDCoeffCalc

Description

`PIDCoeffCalc` computes the PID coefficients based on values of `Kp, Ki` and `Kd` provided by the user.

```
abcCoefficients[0] = Kp + Ki + Kd
abcCoefficients[1] = -(Kp + 2*Kd)
abcCoefficients[2] = Kd
```

This routine also clears the delay line elements in the array `ControlDifference` as well as clears the current PID output element, `ControlOutput`.

Prototype

```
void PIDCoeffCalc (fractional *PIDGainCoeff, tPID *PIDStruct);
```

Arguments

Parameters	Description
`PIDGainCoeff`	Pointer to an input array containing `Kp, Ki, Kd` coefficients in order `[Kp, Ki, Kd]`
`PIDStruct`	Pointer to a PID data structure of type `tPID`

Returns

None

Remarks

PID coefficient array elements may be subject to saturation depending on values of `Kp, Ki, Kd`.

Source File

- `pid_aa.s`

Function Profile

Device	Program Words	Cycles
PIC32A	15	28

System resource usage

- *W0* - used, not restored
- *ACCA, ACCB* - used, not restored
- *CORCON* - saved, used, restored
- *REPEAT* instruction usage – None

7.2.3. PID

Description

`PID` computes the `controlOutput` element of the data structure `tPID`:

```
controlOutput[n] = controlOutput[n-1]
                   + controlHistory[n] * abcCoefficient[0]
                   + controlHistory[n-1] * abcCoefficient[1]
                   + controlHistory[n-2] * abcCoefficient[2]
//where
abcCoefficient[0] = Kp + Ki + Kd
abcCoefficient[1] = -(Kp + 2*Kd)
abcCoefficient[2] = Kd
ControlHistory[n] =  MeasuredOutput[n] - ReferenceInput[n]
```

Prototype

```
void PID ( tPID* PIDStruct );
```

Arguments

Parameters	Description
PIDStruct	A pointer to a PID data structure of type `tPID`

Returns

Pointer to *PIDStruct*

Remarks

`controlOutput` element is updated by the `PID()` routine. The `controlOutput` will be subject to saturation.

Source File

- `pid_aa.s`

Function Profile

Device	Program Words	Cycles
PIC32A	22	34

System resource usage

- *W0...W7* - used, not restored.
- *ACCA, ACCB* - used, not restored
- *CORCON* - saved, used, restored
- *REPEAT* - instruction usage – None

8. Conversion Functions

Function	Description
Fract2Float	Converts a 1.31 fractional value to an IEEE floating-point single-precision value.
Float2Fract	Converts a IEEE floating-point single-precision value to 1.31 fractional value.

8.1. Fract2Float

Description

`Fract2Float` converts a 1.31 fractional value to an IEEE floating-point single-precision value.

Prototype

```
float Fract2Float (fractional aVal);
```

Arguments

Parameters	Description
aVal	1.31 fractional number in the implicit range [-1, $1 - 2^{-31}$]

Returns

IEEE floating-point single-precision value in range [-1, 4.656613×10^{-10}].

Remarks

The conversion is performed using a hardware floating-point unit.

Source File

- `flt2frct_aa.s`

System resource usage

- *W0* – used, not restored
- *F0 – F1* used, not restored
- *REPEAT* instruction usage – None

8.2. Float2Fract

Description

`Float2Fract` converts a IEEE floating-point single-precision value to a 1.31 fractional value.

Prototype

```
fractional Float2Fract (float aVal);
```

Arguments

Parameters	Description
aVal	IEEE floating-point single-precision value in range • [-1, 4.656613×10^{-10}] for PIC32A.

Returns

- 1.15/1.31 fractional numbers depending on the device family
- 1.31 fractional number in the implicit range [-1, $1 - 2^{-31}$]

Remarks

The conversion is performed using a hardware floating-point unit in case of PIC32A and convergent rounding and saturation mechanisms for other devices.

Source File

- `frct2flt_aa.s`

System resource usage

- *W0* – used, not restored
- *F0* – *F2* used, not restored
- *REPEAT* instruction usage – None

Microchip Information

Trademarks

The “Microchip” name and logo, the “M” logo, and other names, logos, and brands are registered and unregistered trademarks of Microchip Technology Incorporated or its affiliates and/or subsidiaries in the United States and/or other countries (“Microchip Trademarks”). Information regarding Microchip Trademarks can be found at https://www.microchip.com/en-us/about/legal-information/microchip-trademarks.

ISBN: 979-8-3371-1744-7

Legal Notice

This publication and the information herein may be used only with Microchip products, including to design, test, and integrate Microchip products with your application. Use of this information in any other manner violates these terms. Information regarding device applications is provided only for your convenience and may be superseded by updates. It is your responsibility to ensure that your application meets with your specifications. Contact your local Microchip sales office for additional support or, obtain additional support at www.microchip.com/en-us/support/design-help/client-support-services.

THIS INFORMATION IS PROVIDED BY MICROCHIP “AS IS”. MICROCHIP MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND WHETHER EXPRESS OR IMPLIED, WRITTEN OR ORAL, STATUTORY OR OTHERWISE, RELATED TO THE INFORMATION INCLUDING BUT NOT LIMITED TO ANY IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE, OR WARRANTIES RELATED TO ITS CONDITION, QUALITY, OR PERFORMANCE.

IN NO EVENT WILL MICROCHIP BE LIABLE FOR ANY INDIRECT, SPECIAL, PUNITIVE, INCIDENTAL, OR CONSEQUENTIAL LOSS, DAMAGE, COST, OR EXPENSE OF ANY KIND WHATSOEVER RELATED TO THE INFORMATION OR ITS USE, HOWEVER CAUSED, EVEN IF MICROCHIP HAS BEEN ADVISED OF THE POSSIBILITY OR THE DAMAGES ARE FORESEEABLE. TO THE FULLEST EXTENT ALLOWED BY LAW, MICROCHIP’S TOTAL LIABILITY ON ALL CLAIMS IN ANY WAY RELATED TO THE INFORMATION OR ITS USE WILL NOT EXCEED THE AMOUNT OF FEES, IF ANY, THAT YOU HAVE PAID DIRECTLY TO MICROCHIP FOR THE INFORMATION.

Use of Microchip devices in life support and/or safety applications is entirely at the buyer’s risk, and the buyer agrees to defend, indemnify and hold harmless Microchip from any and all damages, claims, suits, or expenses resulting from such use. No licenses are conveyed, implicitly or otherwise, under any Microchip intellectual property rights unless otherwise stated.

Microchip Devices Code Protection Feature

Note the following details of the code protection feature on Microchip products:

- Microchip products meet the specifications contained in their particular Microchip Data Sheet.
- Microchip believes that its family of products is secure when used in the intended manner, within operating specifications, and under normal conditions.
- Microchip values and aggressively protects its intellectual property rights. Attempts to breach the code protection features of Microchip products are strictly prohibited and may violate the Digital Millennium Copyright Act.
- Neither Microchip nor any other semiconductor manufacturer can guarantee the security of its code. Code protection does not mean that we are guaranteeing the product is “unbreakable”. Code protection is constantly evolving. Microchip is committed to continuously improving the code protection features of our products.