PM0014
Programming manual

General-purpose DSP library for ST10

Introduction

STMicroelectronics' ST10-DSP microcontroller family offers an attractive set of digital signal processing (DSP) features. The 16-bit multiply-accumulate unit (MAC) of the ST10-DSP microcontrollers allows all data acquisition (average, Max/Min), control oriented signal processing (PID, PD) and filtering (FIR, IIR) widely used in numerous applications.

In addition, the combination of ST10 real time CPU performances with the MAC unit makes ST10-DSP microcontrollers superior in a lot of applications like electronic steering, suspension, engine control, airbag, etc.

The ST10-DSP devices are identified by the number 2 in the first of the three digits of the variant code (e.g.: ST10F269).

This document provides a ST10-DSP library. The ST10-DSP library is a set of optimized routines made for C programmers using ST10F2xx devices. This library is made of arithmetic and signal processing functions callable from C and fully compatible with the TASKING C compiler. It facilitates the evaluation and the use of the most common digital filtering routines with the ST10-DSP microcontrollers.

Contents

1 ST10-DSP features

ST10 is a combined CPU and DSP.

As a CPU, it is a powerful real time oriented 16-bit CPU.

As a DSP, it is a single MAC 16 by 16-bit multiplier with a 40-bit accumulator.

1.1 Harvard architecture

ST10-DSP has an Harvard architecture to allow on every instruction cycle:

- 1 opcode fetch,
- 2 operand reads,
- 1 optional operand write.

1.2 Multiply and accumulate

ST10 supports different multiply and accumulate instructions with different addressing modes.

With the CoMAC [Rx+], [Ry+], in a single instruction cycle, ST10 is able to:

- To fetch 2 operands addressed by Rx and Ry,
- To update 2 pointers (increment, decrement, add an index)

CoMacM is similar to CoMac except that ST10 also moves 1 operand to the bottom of the table (useful in data acquisition routines).

1.3 Minimum and maximum

ST10 has 2 instructions for minimum and maximum detection (CoMin and CoMax). Combined with auto-incrementation, those instructions allow ST10 to scan a table of samples and detect either the minimum or the maximum in one instruction cycle per sample.

1.4 Instruction repeat unit

Each instruction can be repeated either an immediate number of times or a variable number of times.

Register MRW is used to repeat a variable number of times.

Repeat sequences can be interrupted.

This allows ST10-DSP to compute a FIR (16 by 16; real) at a rate of 1 TAP per instruction cycle.

1.5 Barrel shifter

ST10-DSP has a barrel shifter connected to the accumulator: any result or value loaded into the accumulator can be shifted. The shift value must be between 0 and 8 (included).

1.6 Real-time aspects

The ST10-DSP is both a real time CPU and a DSP. Any DSP code developed for ST10-DSP can be interrupted at any time (including during repeat sequences) and execution resumed after the interrupt routine. During the interrupt, bit MR remains set to indicate that a repeated instruction has been interrupted.

- **Latency:** there is no added latency on interrupts when DSP functions are used.
- **Interrupt routine requirements:** the only requirement for interrupt routines that are using the DSP and that are interrupting a DSP function is to save and restore the MAC registers at the entry point and exit point of the routine. This control can be automatically done by Tasking tool chain by using **"#pragma savemac"** on each task using DSP functions (for details, refer to Tasking user's manual).

1.7 ST10 intrinsic benchmarks

Table 1. ST10 intrinsic benchmarks

Benchmark	Execution time (no. of cycles)	Comments
FIR 16*16 real	N	
IIR real DF2	2N	
Proportional integrator differentiator (PID)	6N	u(n) = u(n-1)+k0*e(n)+k1*e(n-1)+k2*e(n-2)
Array search: index of the highest value (signed or unsigned)	1.5N	1N to search for the maximum + 0.5N to retrieve its address. (N: number of elements)

2 ST10-DSP library benchmarks

2.1 Performance considerations

The performance figures reported underneath have been obtained with all operands mapped into ST10 Dual Port RAM (DPRAM). Please note that different memory mapping will lead to longer execution time and this could explain differences between the times reported here and the times on other applications. (for details see "ST10-DSP programming" application note).

2.2 Filter functions

The following table illustrates the ST10-DSP capabilities in filtering functions:

Table 2. ST10-DSP capabilities

Function	Code size (byte)	Number of Instruction cycles
FIR 16 *16 real	88	30 + samp *(12 + coef)
FIR 16* 32 real	130	34 + samp *(16 + 2*coef)
FIR 32* 32 real	194	34 + samp *(20 + 4*coef)
IIR direct 2	130	36 + samp *(17 + coef)
IIR cascaded biquad cells	136	34 + samp *(16 + 15*coef)

3 Using the ST10-DSP library

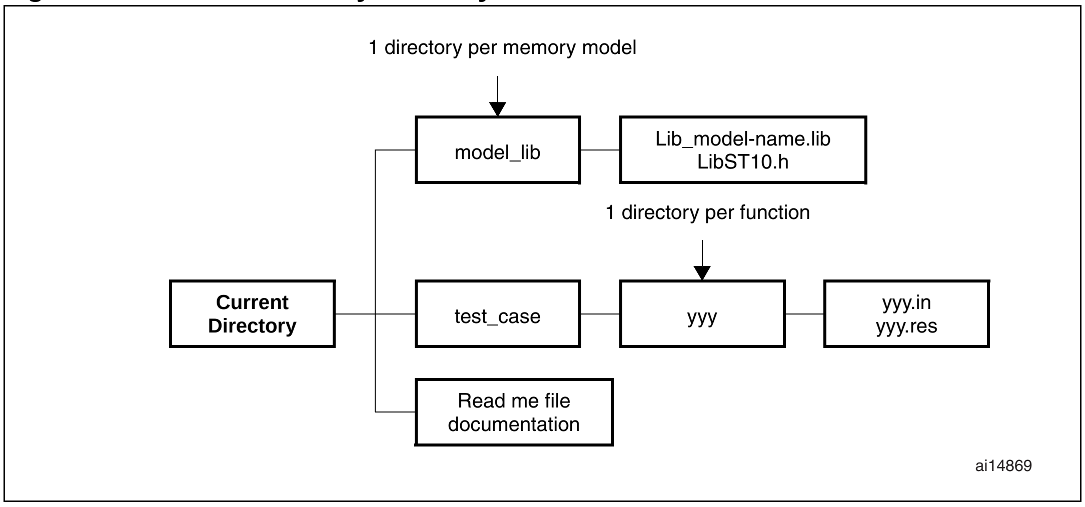
3.1 Library content

The library package consists of 5 main directories:

- 4 directories for the libraries (binary and include file) for each ST10 memory model: tiny, small, medium, large.
- 1 directory for all test cases: 1 subdirectory per library function.

The file structure is the following:

Figure 1. ST10-DSP library directory structure



3.2 How to Install ST10-DSP library

The ST10-DSP library is delivered as an archive file with .zip extension (see directory structure description above).To install the ST10-DSP library, you need to unzip the file in the directory where you want the library to be copied into.

Note: Please, read the README.txt file in the archive file for specific details on the release.

3.3 Calling a function

The functions have been written to be called by a C language program. Calling from assembly language program is possible with respect to parameter passing.

Library users have to save their different used registers before calling a function. See Section 1.6 Real-time aspects on page 6 in previous chapter for more details.

To include a function in a C language program, it is needed to:

- Include the "LibST10.h",
- Link the code with the object library file "Lib_yyyy.lib" of the relevant memory model.

3.4 Tool chain compatibility

As ST10 has different memory models, a library file has been generated for each memory model: tiny, small, medium, large.

ST10-DSP library is compatible with Tasking tool chain (V7.0r1 and upward).

3.5 ST10 device compatibility

The ST10-DSP library is defined to be compatible with all ST10-DSP variants, including the bond-out.

As a consequence, it implements the needed workarounds for the functional features that may be on silicon devices (see respective product errata-sheets).

3.6 ST10 MAC configuration

As this library has be done primarily for filtering functions, **it assumes fractional variables are used**. As a consequence, **users shall set bit "MP" in the MAC control word** either at the reset sequence or before calling any function.

3.7 Interrupts

As explained before, any DSP code developed for ST10 can be interrupted at any time and execution resumed after the interrupt routine. There is no added latency when the DSP library is used.

Interrupt routine requirements: the only requirements are only when the DSP unit is used by task that have different priorities: the interrupting task that may interrupt another task using the DSP should save and restore the MAC registers at the entry point and exit point of the routine. (**use #pragma savemac** in Tasking tool chain).

3.8 Naming convention and parameter passing

The name of a function begins with a lowercase. Each uppercase starts a new word in the name. Underscore is used to separate type of data. Types after the name indicate the types of the parameters passed to the function.

Example: fir_q15_q31_q15

The first q15 indicates the inputs format. q31 corresponds to the type of coefficients. The last q15 indicates the output format.

Variable formats:

- q31 means a signed fractional variable with two's complement fractional format using the 1.31 format (1 signed, 31 fractional bits).
- q15 means a signed fractional variable with two's complement fractional format using the 1.31 format (1 signed, 15 fractional bits).

3.8.1 Arithmetic functions

When applicable, the parameters are passed in the following order:

- Left operator imaginary part MSB
- Left operator imaginary part LSB
- Left operator real part MSB
- Left operator real part LSB
- Right operator imaginary part MSB
- Right operator imaginary part LSB
- Right operator real part MSB
- Right operator real part LSB
- Pointer to an output array.

3.8.2 Filter functions

There is only one parameter to be passed:

- Pointer on a filter structure. This structure is defined in the filter functions chapter.

3.9 Testing a function

All functions of the DSP library are delivered with their test case.

3.9.1 Test environment

For each functions, test files have been created in the testMain directory:

- **X.in**: File of input data on hexadecimal format.
- **X.res**: This file contains outputs obtained by running the Asm source. Inputs are those of the file X.in. Results are on hexadecimal format in X.res.

3.9.2 Using crossview debugger for testing

Crossview debugger has been used for the verification of the results thanks to the provided test vectors. A Tasking project can contain the source asm or C function and a main function. The output result can be displayed on a virtual IO or a hyperterminal.

3.10 Memory reservations

3.10.1 Data Alignment

As the library doesn't use 8-bit values, there's no data alignment restraint, even for Q1.31 variables.

3.10.2 Memory reservations

Filtering functions and spectral functions have to allocate data in DP-RAM (See Section 2.1 Performance considerations on page 7). There are several means to do that.

- The easiest is certainly to declare theses variables as global and to use *iram* directive:

  Example:

  ```
  iram short DelayLine[NumberCoeffs]
  ...
  void main()
  {...}
  ```

- If you want this variable to be local, you have to declare the variable as a static one and use the iram directive.

  Example:

  ```
  static _iram Delayline[NumberCoeffs];
  ```

3.10.3 Memory model aspects

TINY and SMALL models

Library function’s argument can not be stored as far data (use only near data).

MEDIUM and LARGE models

The arguments of the function must be in the same memory page. In fact, the same Data Page Pointer is used for all the arguments of the function.

E.g.: add_16(a,b,&c)

a and b are located in the same data page

An argument that is a pointer must be an address contained in the same Data Page than the other arguments.

E.g.1: c is located in the same data page than a and b.

E.g. 2: firq15_q15_q15(&fir) where:

- fir.CoeffPtr is a pointer
- fir.DelayLinePtr is a pointer

fir.CoeffPtr and fir.DelayLinePtr must address data in the same page. As fir.DelayLinePtr must be located in DPRAM, it could be a solution to map RAM in the same page than DPRAM (generally page number 3).

4 Developing new functions

This library has been done for filtering. Other general purpose functions or application segment specific functions can be developed.

4.1 Examples of functions for data acquisition

4.1.1 Detection of the minimum or maximum in a collection of samples

ST10 instructions (CoMin and CoMax) allow to detect the maximum or the minimum between the accumulator and an external operand. Using a DSP loop with pointer auto-modification, the minimum or the maximum of a parameter in a collection of samples can be detected at a rate of one instruction cycle per sample for 16-bit operands and for 32-bit operands.

4.1.2 Computing the sum of a collection of samples

Using a DSP loop with ST10 instruction CoAdd allows to compute the sum of a collection of samples at a rate of one instruction cycle per sample for both 16-bit and 32-bit operands.

4.1.3 Search for an element within a collection of samples

Using a DSP loop with ST10 instruction CoCMP allow to compare an external operand with the accumulator at a rate of one instruction cycle per sample for both 16-bit and 32-bit operands.

4.2 Developing your own DSP functions

Please refer to the application note AN1442 - Signal Processing with ST10-DSP for more information.

5 Arithmetic library functions

5.1 Overview

In this part, principal arithmetic functions for C programmers on ST10 are presented. For each operation, four function types have been created according to parameters:

- Real parameters in single precision,
- Real parameters in double precision,
- Complex parameters in single precision,
- Complex parameters in double precision.

Code size results will be given in bytes and performance ones in instruction cycles (1 instruction cycle is equal to two ST10 CPU clock cycles).

5.2 Addition

5.2.1 add_16

add_16 (short Op1, short Op2, short *Output)

Description:	Addition of two real 16-bit inputs.	
Argument:	Op1	Left operand
	Op2	Right operand
	Output	Output pointer
Algorithm:	Op1 + Op2 = Output	
Assembly source code:	add_16.asm	
Code size and cycles:	Size: 6 bytes	Instruction cycles: 10
Test:	Two files are provided to test the function: an input vector file (Add_16.in) and an output vector file (Add_16.res):	

Add_16.in is made as follows:

```
Left operand1, Right operand1,
Left operand2, Right operand2,
```

...

Add_16.res is made as follows:

```
Output operand1,
Output operand2,
```

...

5.2.2 add_32

add_32 (short LeftOpMsb, short LeftOpLsb, short RightOpMsb, short RightOpLsb, long *Output)

Description: Addition of two real 32-bit operands.

Argument:

LeftOpMsb	Left operand most significant bits.
LeftOpLsb	Left operand least significant bits.
RightOpMsb	Right operand most significant bits.
RightOpLsb	Right operand least significant bits.
Output	Output pointer.

Algorithm: LeftOp + RightOp = Output

Note: Set the MS bit field of the MCW register to have a saturated result (signed arithmetic)

Assembly source code: `add_32.asm`

Code size and cycles: Size: 20 bytes
Instruction cycles: 14

Test: Two files are provided to test the function: an input vector file (Add_32.in) and an output vector file (Add_32.res):

Add_32.in is made as follows:

```
LeftOpMsb1, LeftOpLsb1, RightOpMsb1,
RightOpLsb1

LeftOpMsb2, LeftOpLsb2, RightOpMsb2,
RightOpLsb2

...
```

Add_32.res is made as follows:

```
OutputLsb1, OutputMsb1,

OutputLsb2, OutputMsb2,

...
```

5.2.3 cAdd_16

cAdd_16 (short LeftOp_I, short LeftOp_R, short RightOp_I, short RightOp_R, long *Output)

Description: Addition of two complex 16-bit operands.

Argument:

LeftOp_I	Left operand imaginary part.
LeftOp_R	Left operand real part.
RightOp_I	Right operand imaginary part.
RightOp_R	Right operand real part.
Output	Output pointer.

Algorithm: LeftOp_R + RightOp_R = Output_R

LeftOp_I + RightOp_I = Output_I

Note: Output imaginary part is contained in the highest address (pointed by Output + 2). Output real part is contained in the lowest address (pointed by Output).

Assembly source code: `cAdd_16.asm`

Code size and cycles: Size: 14 bytes

Instruction cycles: 18

Test: Two files are provided to test the function: an input vector file (cAdd_16.in) and an output vector file (cAdd_16.res):

cAdd_16.in is made as follows:

```
LeftOp_I1, LeftOp_R1, RightOp_I1, RightOp_R1
LeftOp_I2, LeftOp_R2, RightOp_I2, RightOp_R2
...
```

cAdd_16.res is made as follows:

```
Output_I1,Output_R1
Output_I2,Output_R2
...
```

5.2.4 cAdd_32

cAdd_32 (short LeftOpMsb_I, short LeftOpLsb_I, short LeftOpMsb_R, short LeftOpLsb_R, short RightOpMsb_I, short RightOpLsb_I, short RightOpMsb_R, short RightOpMsb_I, long *Output)

Description: Addition of two complex 32-bit operands.

Argument:

LeftOpMsb_I	Left operand imaginary part most significant bits.
LeftOpLsb_I	Left operand imaginary part least significant bits.
LeftOpMsb_R	Left operand real part most significant bits.
LeftOpLsb_R	Left operand real part least significant bits.
RightOpMsb_I	Right operand imaginary part most significant bits.
RightOpLsb_I	Right operand imaginary part least significant bits.
RightOpMsb_R	Right Operand real part most significant bits.
RightOpLsb_R	Right Operand real part least significant bits.
Output	Output pointer

Algorithm: LeftOp_R + RightOp_R = Output_R

LeftOp_I + RightOp_I = Output_I

Note: Output imaginary part is contained in the highest addresses (pointed by Output + 4). Output real part is contained in the lowest addresses (pointed by Output).

Set the MS bit-field of the MCW register to have a saturated result (signed arithmetic).

Assembly source code: `cAdd_32.asm`

Code size and cycles: Size: 52 bytes
Instruction cycles: 25

Test: Two files are provided to test the function: an input vector file (cAdd_32.in) and an output vector file (cAdd_32.res):

cAdd_32.in is made as follows:

```
LeftOpMsb_I1, LeftOpLsb_I1, LeftOpMsb_R1, LeftOpLsb_R1,
RightOpMsb_I1, RightOpLsb_I1, RightOpMsb_R1, RightOpLsb_R1,

LeftOpMsb_I2, LeftOpLsb_I2, LeftOpMsb_R2, LeftOpLsb_R2,
RightOpMsb_I2, RightOpLsb_I2, RightOpMsb_R2, RightOpLsb_R2,

...
```

cAdd_32.res is made as follows:

```
OutputLsb_R1, OutputMsb_R1, OutputLsb_I1, OutputMsb_I1

OutputLsb_R2, OutputMsb_R2, OutputLsb_I2, OutputMsb_I2,

...
```

5.3 Subtraction

5.3.1 sub_16

sub_16 (short Op1, short Op2, short *Output)

Description:	Subtraction of two real 16-bit inputs.	
Argument:	Op1	Left operand.
	Op2	Right operand.
	Output	Output pointer.
Algorithm:	Op1 - Op2 = Output	
Note:		
Assembly source code:	`sub_16.asm`	
Code size and cycles:	Size: 6 bytes	
	Instruction cycles: 10	

Test: Two files are provided to test the function: an input vector file (Sub_16.in) and an output vector file (Sub_16.res):

Sub_16.in is made as follows:

```
Left operand1, Right operand1,
...
```

Sub_16.res is made as follows:

```
Output operand1,
...
```

5.3.2 sub_32

sub_32 (short LeftOpMsb, short LeftOpLsb, short RightOpMsb, short RightOpLsb, long *Output)

Description: Subtraction of two real 32-bit inputs.

Argument:

LeftOpMsb	Left operand most significant bits.
LeftOpLsb	Left operand least significant bits.
RightOpMsb	Right operand most significant bits.
RightOpLsb	Right operand least significant bits.
Output	Output pointer.

Algorithm: LeftOp - RightOp = Output

Note: Set the MS bit-field of the MCW register to have a saturated result (signed arithmetic).

Assembly source code: `sub_32.asm`

Code size and cycles: Size: 20 bytes

Instruction cycles: 14

Test: Two files are provided to test the function: an input vector file (Sub_32.in) and an output vector file (Sub_32.res):

Sub_32.in is made as follows:

```
LeftOpMsb1, LeftOpLsb1, RightOpMsb1, RightOpLsb1
...
```

Sub_32.res is made as follows:

```
OutputMsb1, OutputLsb1,...
```

5.3.3 cSub_16

cSub_16 (short LeftOp_I, short LeftOp_R, short RightOp_I, short RightOp_R, long *Output)

Description: Subtraction of two complex 16-bit operands.

Argument:

LeftOp_I	Left operand imaginary part.
LeftOp_R	Left operand real part.
RightOp_I	Right operand imaginary part.
RightOp_R	Right operand real part.
Output	Output pointer.

Algorithm: LeftOp_R - RightOp_R = Output_R

LeftOp_I - RightOp_I = Output_I

Note: Output imaginary part is contained in the highest address (pointed by Output + 2). Output real part is contained in the lowest address (pointed by Output).

Assembly source code: `cSub_16.asm`

Code size and cycles: Size: 14 bytes

Instruction cycles: 18

Test: Two files are provided to test the function: an input vector file (cSub_16.in) and an output vector file (cSub_16.res):

cSub_16.in is made as follows:

```
LeftOp_I1, LeftOp_R1, RightOp_I1, RightOp_R1
LeftOp_I2, LeftOp_R2, RightOp_I2, RightOp_R2
...
```

cSub_16.res is made as follows:

```
Output_R1,Output_I1
...
```

5.3.4 cSub_32

csub_32 (short LeftOpMsb_I, short LeftOpLsb_I, short LeftOpMsb_R, short LeftOpLsb_R, short RightOpMsb_I, short RightOpLsb_I, short RightOpMsb_R, short RightOpMsb_I, long *Output)

Description: Subtraction of two complex 32-bit operands.

Argument:

LeftOpMsb_I	Left operand imaginary part most significant bits.
LeftOpLsb_I	Left operand imaginary part least significant bits.
LeftOpMsb_R	Left operand real part most significant bits.
LeftOpLsb_R	Left operand real part least significant bits.
RightOpMsb_I	Right operand imaginary part most significant bits.
RightOpLsb_I	Right operand imaginary part least significant bits.
RightOpMsb_R	Right Operand real part most significant bits.
RightOpLsb_R	Right Operand real part least significant bits.
Output	Output pointer.

Algorithm: LeftOp_R - RightOp_R = Output_R

LeftOp_I - RightOp_I = Output_I

Note: Output imaginary part is contained in the highest addresses (pointed by Output + 4). Output real part is contained in the lowest addresses (pointed by Output).

Set the MS bit-field of the MCW register to have a saturated result (signed arithmetic).

Assembly source code: `cSub_32.asm`

Code size and cycles: Size: 52 bytes

Instruction cycles: 25

Test: Two files are provided to test the function: an input vector file (cSub_32.in) and an output vector file (cSub_32.res):

cSub_32.in is made as follows:

```
LeftOpMsb_I1, LeftOpLsb_I1, LeftOpMsb_R1, LeftOpLsb_R1,
RightOpMsb_I1, RightOpLsb_I1, RightOpMsb_R1, RightOpLsb_R1,
...
```

cSub_32.res is made as follows:

```
OutputMsb_R1, OutputLsb_R1,
...
```

5.4 Multiplication

5.4.1 mul_q15_q15_q31

mul_q15_q15_q31 (short LeftOp, short RightOp, long *Output)

Description: Multiplication of two real 16-bit fractional operands.

Argument:

LeftOp	Left operand.
RightOp	Right operand.
Output	Output pointer.

Algorithm: LeftOp * RightOp = Output

Note: None

Assembly source code: `mul_q15_q15_q31.asm`

Code size and cycles: Size: 22 bytes

Instruction cycles: 10

Test: Two files are provided to test the function: an input vector file (mul_q15_q15_q31.in) and an output vector file (mul_q15_q15_q31.res):

mul_q15_q15_q31.in is made as follows:

```
LeftOp1, RightOp1,
...
```

mul_q15_q15_q31.res is made as follows:

```
Output1,
...
```

5.4.2 mul_q31_q31_q31

mul_q31_q31_q31 (short LeftOpMsb, short LeftOpLsb, short RightOpMsb, short RightOpLsb, long *Output)

Description:	Multiplication of two real 32-bit fractional operands. Output is on 32 bits.	
Argument:	LeftOpMsb	Left operand most significant bits.
	LeftOpLsb	Left operand least significant bits.
	RightOpMsb	Right operand most significant bits.
	RightOpLsb	Right operand least significant bits.
	Output	Output pointer.
Algorithm:	LeftOp * RightOp = Output	
Note:	None	
Assembly source code:	`mul_q31_q31_q31.asm`	
Code size and cycles:	Size: 54 bytes	
	Instruction cycles: 19	

Test: Two files are provided to test the function: an input vector file (mul_q31_q31_q31.in) and an output vector file (mul_q31_q31_q31.res):

mul_q31_q31_q31.in is made as follows:

```
LeftOpMsb1, LeftOpLsb1, RightOpMsb1, RightOpLsb1,
...
```

mul_q31_q31_q31.res is made as follows:

```
Output1,...
```

5.4.3 cMul_q15_q15_q15

cMul_q15_q15_q15 (short LeftOp_I, short LeftOp_R, short RightOp_I, short RightOp_R, short *Output).

Description: Multiplication of two complex 32-bit (16 bits for each part) inputs. Output is on 32 bits (16 bits for each part).

Argument:

LeftOp_I	Left operand imaginary part.
LeftOp_R	Left operand real part.
RightOp_I	Right operand imaginary part.
RightOp_R	Right operand real part.
Output	Output pointer.

Algorithm: LeftOp_R * RightOp_R - LeftOp_I * RightOp_I = Output_R

LeftOp_R * RightOp_I + LeftOp_R * RightOp_I = Output_I

Note: Output imaginary part is contained in the highest address (pointed by Output + 2). Output real part is in the lowest address (pointed by Output).

Assembly source code: `cMul_q15_q15_q15.asm`

Code size and cycles: Size: 36 bytes

Instruction cycles: 14

Test: Two files are provided to test the function: an input vector file (cMul_q15_q15_q15.in) and an output vector file (cMul_q15_q15_q15.res):

cMul_q15_q15_q15.in is made as follows:

```
LeftOp_I1, LeftOp_R1, RightOp_I1, RightOp_R1,
...
```

cMul_q15_q15_q15.res is made as follows:

```
Output_I1,
Output_R1,
...
```

5.4.4 cMul_q31_q31_q31

cMul_q31_q31_q31 (short Op1_I_Msb, short Op1_I_Lsb, short Op1_R_Msb, short Op1_R_Lsb, short Op2_I_Msb, short Op2_I_Lsb, short Op2_R_Msb, short Op2_R_Lsb, long *Output).

Description: Multiplication of two complex 64-bit (32 bits for each part) inputs. Output is on 64 bits (32 bits for each part).

Argument:

Op1_I_Msb	Left operand imaginary part most significant bits.
Op1_I_Lsb	Left operand real part least significant bits.
Op1_R_Msb	Right operand imaginary part most significant bits.
Op1_R_Lsb	Right operand real part least significant bits.
Op2_I_Msb	Left operand imaginary part most significant bits.
Op2_I_Lsb	Left operand real part least significant bits.
Op2_R_Msb	Right operand imaginary part most significant bits.
Op2_R_Lsb	Right operand real part least significant bits.
Output	Output pointer.

Algorithm: LeftOp_R * RightOp_R - LeftOp_I * RightOp_I = Output_R

LeftOp_R * RightOp_I + LeftOp_R * RightOp_I = Output_I

Note: Output imaginary part is contained in the highest address (pointed by Output + 4). Output real part is in the lowest address (pointed by Output).

Assembly source code: `cMul_q31_q31_q31.asm`

Code size and cycles: Size: 214 bytes

Instruction cycles: 75

Test: Two files are provided to test the function: an input vector file (cMul_q31_q31_q31.in) and an output vector file (cMul_q31_q31_q31.res):

cMul_q31_q31_q31.in is made as follows:

```
LeftOpMsb_I1, LeftOpLsb_I1, LeftOpMsb_R1, LeftOpLsb_R1,
RightOpMsb_I1, RightOpLsb_I1, RightOpMsb_R1, RightOpLsb_R1,
...
```

cMul_q31_q31_q31.res is made as follows:

```
OutputMsb_I1, OutputLsb_I1,
...
```

5.5 Division

5.5.1 div_q31_q15_q15

div_q31_q15_q15 (short LeftOpMsb, short LeftOpLsb, short RightOp, short *Output).

Description:	Division of two real fractional inputs. Dividend is on 32 bits, divisor on 16 bits and quotient on 16 bits.	
Argument:	LeftOpMsb	Left operand most significant bits.
	LeftOpLs	Left operand least significant bits.
	RightOp	Right operand.
	Output	Output pointer.

Algorithm: LeftOp / RightOp = Output

Note: The dividend must be smaller than the divisor for a valid result.

Divisions on limits (for example, division of 1 by 1 or of -1 by -1) are not computed by this routine.

The algorithm implemented does not allow integer division. For dividing two integers (dividend in 32.0 format and divisor in 16.0), shift the dividend one bit to the left (into 31.1 format) before dividing.

Assembly source code: `div_q31_q15_q15.asm`

Code size and cycles: Size: 98 bytes

Instruction cycles: 216

Test: Two files are provided to test the function: an input vector file (div_q31_q15_q15.in) and an output vector file (div_q31_q15_q15.res):

div_q31_q15_q15.in is made as follows:

```
LeftOpMsb1, LeftOpLsb1, RightOp1
...
```

div_q31_q15_q15.res is made as follows:

```
Output1,
...
```

5.5.2 div_q63_q31_q31

div_q63_q31_q31 (short LeftOpMsb, short LeftOp2b, short LeftOp3b,
short LeftOpLsb,short RightOpMsb, short RightOpLsb, long *Output)

Description: Division of two real fractional inputs. Dividend is on 64 bits, divisor on 32 bits and quotient on 32 bits.

Argument:

LeftOpMsb	Left operand most significant bits high.
LeftOp2b	Left operand most significant bits low.
LeftOp3b	Left operand least significant bits high.
LeftOpLsb	Left operand least significant bits low.
RightOpMsb	Right operand most significant bits.
RightOpLsb	Right operand least significant bits.
Output	Output pointer.

Algorithm: LeftOp / RightOp = Output

Note: The dividend must be smaller than the divisor for a valid result.

Divisions on limits (for example, division of 1 by 1 or of -1 by -1) are not made by this routine.

The algorithm implemented does not allow integer division. For dividing two integers (dividend in 64.0 format and divisor in 32.0), shift the dividend one bit to the left (into 63.1 format) before dividing.

Assembly source code: `div_q63_q31_q31.asm`

Code size and cycles: Size: 204 bytes

Instruction cycles: 892

Test: Two files are provided to test the function: an input vector file (div_q63_q31_q31.in) and an output vector file (div_q63_q31_q31.res):

div_q63_q31_q31.in is made as follows:

```
LeftOpMsb1, LeftOp2b1, LeftOp3b1, LeftOpLsb1,
RightOpMsb1, RightOpLsb1
...
```

div_q63_q31_q31.res is made as follows:

```
Output1,
...
```

5.5.3 cDiv_q31_q15_q15

cDiv_q31_q15_q15 (short LeftOpIMsb, short LeftOpILsb, short LeftOpRMsb, short LeftOpRLsb, short RightOpI, short RightOpR, short *Output).

Description: Division of two complex fractional inputs. Dividend is on 64 bits (32 bits for each part), divisor and quotient on 32 bits (16 bits for each part).

Argument:

LeftOpIMsb	Left operand imaginary part most significant bits.
LeftOpILsb	Left operand imaginary part least significant bits.
LeftOpRMsb	Left operand real part most significant bits.
LeftOpRLsb	Left operand real part least significant bits.
RightOpI	Right operand imaginary part.
RightOpR	Right operand real part.
Output	Output pointer.

Algorithm:

$$\frac{a+ib}{c+id}=\frac{ac+bd}{c^2+d^2}+i\times\frac{bc-ad}{c^2+d^2}$$

Note: Output imaginary part is contained in the lowest address (pointed by Output). Output real part is in the highest address (pointed by Output + 2).

If $c^2 + d^2$ is equal to 0, there is a result saturation to the maximum positive value.

$c^2 + d^2$ must be smaller than the dividend real part and than the dividend imaginary part for a valid result. Else, there is a result saturation to the maximum positive or negative value.

If $c^2 + d^2$, ac + bd or bc - ad is greater than 1 or smaller than -1, it saturates to the maximum positive or negative value.

If $c^2 + d^2$ and ac + bd or bc - ad are both a maximum value. The real part of the result is not valid (respectively the imaginary part of the result).

The implemented algorithm does not allow integer division. For dividing integers (dividend in 32.0 format and divisor in 16.0), shift the dividend one bit to the left (into 31.1 format) before dividing.

Assembly source code: `cDiv_q31_q15_q15.asm`

Code size and cycles: Size: 262 bytes
Instruction cycles: 470

Test:

Two files are provided to test the function: an input vector file (cDiv_q31_q15_q15.in) and an output vector file (cDiv_q31_q15_q15.res):

cDiv_q31_q15_q15.in is made as follows:

```
LeftOpMsb_I1, LeftOpLsb_I1, LeftOpMsb_R1,
LeftOpLsb_R1, RightOp_I1, RightOp_R1,

...
```

cDiv_q31_q15_q15.res is made as follows:

```
Output_R1, Output_I1

...
```

5.5.4 cDiv_q31

cDiv_q31 (short LeftOpIMsb, short LeftOpILsb, short LeftOpRMsb, short LeftOpRLsb, short RightOpIMsb, short RightOpILsb, short RightOpRMsb, short RightOpRLsb, long *Output).

Description: Division of two complex fractional inputs. Dividend, divisor and quotient are on 64 bits (32 bits for each part).

Argument:

LeftOpIMsb	Left operand imaginary part most significant bits.
LeftOpILsb	Left operand imaginary part least significant bits.
LeftOpRMsb	Left operand real part most significant bits.
LeftOpRLsb	Left operand real part least significant bits.
RightOpIMsb	Right operand imaginary part most significant bits.
RightOpILsb	Right operand imaginary part least significant bits.
RightOpRMsb	Right operand real part most significant bits.
RightOpRLsb	Right operand real part least significant bits.
Output	Output pointer.

Algorithm:

$$\frac{a+ib}{c+id} = \frac{ac+bd}{c^2+d^2} + i \times \frac{bc-ad}{c^2+d^2}$$

Note: Output imaginary part is contained in the lowest addresses (pointed by Output). Output real part is in the highest addresses (pointed by Output + 4).

If $c^2 + d^2$ is equal to 0, result saturates to the maximum positive value.

$c^2 + d^2$ must be smaller than the dividend real part and than the dividend imaginary part for a valid result. Else, there is a result saturation to the maximum positive or negative value.

If $c^2 + d^2$ and ac + bd (respectively $c^2 + d^2$ and bc - ad) are both a maximum value. The result real part is not valid (respectively the result imaginary part).

The implemented algorithm doesn’t allow integer division. For dividing integers (dividend in 32.0 format and divisor in 16.0), shift the dividend one bit to the left (into 31.1 format) before dividing.

Assembly source code: `cDiv_q31.asm`

Code size and cycles: Size: 442 bytes
Instruction cycles: 1148

Test:

Two files are provided to test the function: an input vector file (cDiv_q31.in) and an output vector file (cDiv_q31.res):

cDiv_q31.in is made as follows:

```
LeftOpIMsb1, LeftOpILsb1, LeftOpRMsb1,
LeftOpRLsb1, RightOpIMsb1, RightOpILsb1,
RightOpRMsb1, RightOpRLsb1,

...
```

cDiv_q31.res is made as follows:

```
Output_R1,

Output_I1,

...
```

5.6 Matrix multiply

5.6.1 mMul_q15_q15_q15

void mMul_q15_q15_q15 (short Mat1[][], int row1, int column1,
short Mat2[][], int column2, short Output[][]).

Description: Real 16-bit fractional matrix multiply. The result matrix is on 1.15 format.

Argument:

Mat1	Pointer to matrix 1.
row1	Number of rows in matrix 1.
column1	Number of columns in matrix 1.
Mat2	Pointer to matrix 2.
column2	Number of columns in matrix 2.
Output	Pointer to result matrix

Algorithm:

$$\text{Output}(i, j) = \sum_{k=0}^{\text{column1}} \text{Mat1}(1, k) \times \text{Mat2}(k, j)$$

Note: Matrix 1 and Matrix 2 should be stored in the DPRAM.

The output matrix saturates to 1 or -1 if the result is greater than 1 or respectively lower than -1.

Assembly source code: `mMul_q15_q15_q15.asm`

Code size and cycles: Size: 122 bytes
Instruction cycles: 40 + row1 * (6 + column2 * (8 + column1))

Test: To test this function, two test cases are provided. Each test case has two input vector files that contain the matrix to multiply and one output vector file that contains the output matrix. For instance, Mat1_4_3.in and Mat2_3_2.in contains respectively a Matrix[4][3] and a Matrix [3][2]. Be aware that it is possible to multiply a Matrix[4][3] by a Matrix [3][2] and not the inverse. An input file as well as the output file contains the elements of the matrix sorted as follows:

```
Op_row1_col1, Op_row1_col2, Op_row1_col3...
...Op_row2_col1,Op_row2_col2...
```

5.6.2 mVM_q15_q15_q15

void mVM_q15_q15_q15 (short Mat[][], short *Vectin, int N, short *Output).

Description: Real 16-bit fractional [N*N] [N*1] matrix multiply. The result vector is on 1.15 format.

Argument:

Mat	Pointer to matrix.
Vectin	Pointer to input vector.
N	Size of the squared matrix and size of vectors.
Output	Pointer to result vector.

Algorithm:

$$\text{Output}(i) = \sum_{k=0}^{N} \text{Mat}(i, k) \times \text{Vectin}(k)$$

Note: Matrix and Vector should be stored in the DPRAM.

The output vector saturates to 1 or -1 if the result is greater than 1 or respectively lower than -1.

Assembly source code: `mVM_q15_q15_q15.asm`

Code size and cycles: Size: 60 bytes

Instruction cycles: $N^2 + 10N - 10$

Test: To test this function, two test cases are provided. Each test case has two input vector files that contain the matrix to multiply and one output vector file that contains the output matrix. For instance, mat_10_10.in and vect_10_1.in contains respectively a Matrix[10][10] and a vector [10][1]. The input file as well as the output file contains the elements of the matrix sorted as follows:

```
Op_row1_col1, Op_row1_col2, Op_row1_col3...
....Op_row2_col1,Op_row2_col2...
```

5.7 Function approximation

5.7.1 sqrt_p16q16_p8q8

void sqrt_p16q16_p8q8 (short xmsb, short xlsb, short *Output).

Description: The square root function is approximated by the Hoerner polynomial. The input is in 16.16 format and the output is in 8.8 format with a precision to within three least significant bits.

Argument:

	xmsb	Input most significant bits.
	xlsb	Input least significant bits.
	Output	Output pointer.

Algorithm: $x^{1/2} \approx 0.2075806 + 1.454895x - 1.34491x^2 + 1.106812x^3 - 0.536499x^4 + 0.11212116x^5$

Note: The input value can range from 0 to 2^{15} - $1/2^{15}$.

In fact the previous approximation is valid only between 0.5 and 1 but all inputs are scaled into this range.

Assembly source code: `sqrt_p16q16_p8q8.asm`

Code size and cycles: Size: 330 bytes

Instruction cycles: 164

Test: Two files are provided to test the function: an input vector file (sqrt_p16q16_p8q8.in) and an output vector file (sqrt_p16q16_p8q8.res):

sqrt_p16q16_p8q8.in is made as follows:

```
OpLsb1,
OpMsb1,
OpLsb2,
OpMsb2,
...
```

sqrt_p16q16_p8q8.res is made as follows:

```
Output1,
Output2,
...
```

6 Signal processing library functions

6.1 Overview

In this part, a set of signal processing functions for C programmers on ST10 are presented.

6.2 Data type

LibST10.h (C compiler package) defines the data types used for ST10 programming.

The following types are used in the library:

- Filter: structure containing all the elements needed for filtering.

Filt.NumberCoeffs : number of filter coefficients.

Filt.NumberOutputSamples : number of output samples.

Filt.CoeffPtr : pointer on filter coefficients.

Filt.InputPtr : pointer on input samples.

Filt.OutputPtr : pointer on output samples.

Filt.DelayLinePtr : pointer on delay line.

Library's users have to fill the different fields of this structure before calling any filtering function.

According to the filter type, the number of coefficients definition changes.

Examples:

```
  .FIR:
y(n)=a0*x(n)+a1*x(n-1)+...+a15*x(n-15)
Store NumberCoeffs=16

.IIR direct form
y(n)=b0*x(n)+b1*x(n-1)+...+b15*x(n-15)+a1*y(n-1)+...+a15*y(n-15)
Store NumberCoeffs=32   (a0=0 is stored in memory)

.IIR cascaded biquad form
y(n)=b00*x(n)+...+a02*y(n-2)+.....+a92*y(n-2)
Be careful: store NumberCoeffs = NumberCells = 10 !
```

The delay line is a buffer in memory used by filtering functions to store previous samples needed to perform calculation of the next output sample. The delay line size depends on the number of coefficients.

6.3 Finite impulse response filters

6.3.1 fir_q15_q15_q15

fir_q15_q15_q15 (struct Filter *FirPtr).

Description: Real FIR filter of 16-bit fractional inputs, coefficients and outputs.

Argument: FirPtr Filter pointer.

Algorithm:

$$y(n) = \sum_{i=0}^{nbcoeff-1} b_i * x(n - i)$$

Note: The delay line must be stored in DPRAM. Its size is 2*nb_coeff. Before the first call to the function, the delay line must be initialized to 0.

There is a saturation to 1 (respectively -1) if the computed output sample is greater than 1 (respectively lower than -1).

```
Example of the use for fir_q15_q15_q15:

   #define FirNumberCoeffs NumberOfCoefficients
   #define NbOutput NumberOfOutputSamples
   iram short Delayline [FirNumberCoeffs];
   iram short Coeff [FirNumberCoeffs];
   void main ()
   {
   // Data definition
   short Input[NbOutput];
   short Out[NbOutput];
   // Here should be the coef[] and input[] initialization.
See the Test part...
   // Declare a filter structure
   struct Filter *Filt;

   // Initialisation of filt
   Filt.NumberCoeffs = FirNumberCoeffs;
   Filt.NumberOutputSamples = NbOutput;
   Filt.CoeffPtr = Coeff;
   Filt.InputPtr = Input;
   Filt.OutputPtr = Out;
   Filt.DelayLinePtr = Delayline;

   //Filter processing
   fir_q15_q15_q15(&Filt);
   }

Coefficients must be stored in memory as follow:
   addr Filt.CoeffPtr   ----> a0
   addr Filt.CoeffPtr + 2 ----> a1
   ...
   addr Filt.CoeffPtr + 2 * (FirNumberCoeffs -1)---->
aFirNumberCoeffs-1
```

Assembly source code: `fir_q15_q15_q15.asm`

Code size and cycles:	Size: 88 bytes
	Instruction Cycles: 30 + NbOutput * (12 + FirNumberCoeffs)
Test:	Two test cases are provided.

For each case, two files allow to test the function:

- an input vector file (filterX_q15_q15_q15.in)
- an output vector file (filterX_q15_q15_q15.res).

The configuration is the following:

- number of coefficients: 64
- number of output samples: 201

To perform the test, initialize the Delay line[] to zero, then Coeff[] and Input[] with the filterX_q15_q15_q15.in file. The first values are the coefficients of the filter and the rest of the values are the input values. (See the example for declarations).

6.3.2 fir_q15_q31_q15

fir_q15_q31_q15 (struct Filter *FirPtr).

Description: Real FIR filter of 16-bit fractional inputs and outputs. Fractional coefficients are on 32 bits.

Argument: FirPtr Filter pointer.

Algorithm:

$$y(n) = \sum_{i=0}^{nbcoeff-1} b_i * x(n-i)$$

Note: The delay line must be stored in DPRAM. Its size is 2*2*nb_coeff. Before the first call to the function, the delay line must be initialized to 0.

There is a saturation to 1 (respectively -1) if the computed output sample is greater than 1 (respectively lower than -1).

Example of the use for fir_q15_q31_q15:

```
#define FirNumberCoeffs NumberOfCoefficients
   #define NbOutput NumberOfOutputSamples
   iram short Delayline[FirNumberCoeffs];
   iram short Coeff[2*FirNumberCoeffs];//coef are 32-bit wide
   void main ()
   {
   // Data definition
   short Input[NbOutput];
   short Out[NbOutput];
   // Here should be the coef[] and input[] initialization.
See the Test part...
   // Declare a filter structure
   struct Filter *Filt;

   // Initialisation of filt
   Filt.NumberCoeffs = FirNumberCoeffs;
   Filt.NumberOutputSamples = NbOutput;
   Filt.CoeffPtr = Coeff;
   Filt.InputPtr = Input;
   Filt.OutputPtr = Out;
   Filt.DelayLinePtr = Delayline;

   //Filter processing
   fir_q15_q31_q15(&Filt);
   }
Coefficients must be stored in memory as follow:
addr Filt.CoeffPtr   ----> a0  lsb
addr Filt.CoeffPtr + 2 ----> a0  msb
...
addr Filt.CoeffPtr + 4 *(FirNumberCoeffs -2)---->
aFirNumberCoeffs-1 msb
```

Assembly source code: `fir_q15_q31_q15.asm`

Code size and cycles: Size: 130 bytes

Instruction Cycles: 34 + NbOutput * (16 + 2 * FirNumberCoeffs)

Test:

Two test cases are provided. For each test case, two files allow to test the function:

- an input vector file (filterX_q15_q31_q15.in)
- an output vector file (filterX_q15_q31_q15.res).

The configuration is the following:

- number of coefficients: 32
- number of output samples: 201

To perform the test, initialize the Delay line[] to zero, then Coeff[] and Input[] with the filterX_q15_q31_q15.in file. The first values are the coefficients (lsb first) of the filter and the rest of the values are the input values. (See the example for declarations).

6.3.3 fir_q31_q31_q31

fir_q31_q31_q31(struct Filter *FirPtr).

Description: Real FIR filter of 32-bit fractional inputs, coefficients and outputs.

Argument: FirPtr Filter pointer.

Algorithm:

$$y(n) = \sum_{i=0}^{nbcoeff-1} b_i * x(n - i)$$

Note: The delay line must be stored in DPRAM. Its size is 2*2*nb_coeff. Before the first call to the function, the delay line must be initialized to 0.

There is a saturation to 1 (respectively -1) if the computed output sample is greater than 1 (respectively lower than -1).

Example of the use for fir_q31_q31_q31:

```
   #define FirNumberCoeffs NumberOfCoefficients
   #define NbOutput NumberOfOutputSamples
   iram short Delayline[FirNumberCoeffs];
   iram short Coeff[2*FirNumberCoeffs];//coef are 32-bit wide
   void main ()
   {
   // Data definition
   short Input[2*NbOutput];//Input are 32-bit wide
   short Out[2* NbOutput];//Output are 32-bit wide
   // Here should be the coef[] and input[] initialization.
See the Test part...
   // Declare a filter structure
   struct Filter *Filt;
   // Initialisation of filt
   Filt.NumberCoeffs = FirNumberCoeffs;
   Filt.NumberOutputSamples = NbOutput;
   Filt.CoeffPtr = Coeff;
   Filt.InputPtr = Input;
   Filt.OutputPtr = Out;
   Filt.DelayLinePtr = Delayline;
   //Filter processing
   fir_q31_q31_q31(&Filt);
   }
```

Coefficients must be stored in memory as follows:

```
   addr Filt.CoeffPtr   ----> a0  lsb
   addr Filt.CoeffPtr + 2 ----> a0  msb
   ...
   addr Filt.CoeffPtr + 4 *(FirNumberCoeffs -2)---->
aFirNumberCoeffs-1 msb
```

Inputs and Outputs are stored in memory as follows:

```
   addr Filt.InputPtr   ----> x0  lsb
   addr Filt.InputPtr + 2 ----> x0  msb
   ...
   addr Filt.Input + 4 *(NbOutput -2)----> aNbOutput-1 msb

   addr Filt.OutputPtr  ----> y0  lsb
   addr Filt.OutputPtr + 2 ----> y0  msb
   ...
   addr Filt.OutputPtr + 4 *(NbOutput -2)----> aNbOutput-1 msb
```

Assembly source code:	`fir_q31_q31_q31.asm`
Code size and cycles:	Size: 194 bytes
	Instruction Cycles: 34 + NbOutput * (20 + 4 * FirNumberCoeffs)

Test:

Two test cases are provided. For each test case, two files allow to test the function:

- an input vector file (filterX_q31_q31_q31.in)
- an output vector file (filterX_q31_q31_q31.res).

The configuration is the following:

- number of coefficients: 32
- number of output samples: 201

To perform the test, initialize the Delay line[] to zero, then Coeff[] and Input[] with the filterX_q31_q31_q31.in file./ The first values are the coefficients (lsb first) of the filter and the rest of the values are the input values (lsb first). The output vector file contains the result value (lsb first). (See the example for declarations).

6.4 Infinite impulse response filter

6.4.1 iir2_q15_q15_q15

iir2_q15_q15_q15 (struct Filter *IirPtr).

Description: Real IIR filter of 16-bit fractional inputs, coefficients and outputs. The filter implemented is a direct form II.

Argument: IirPtr Filter pointer.

Algorithm:

$$w(n) = x(n) + \sum_{k=1}^{\frac{\text{IirNumberCoeff}}{2} - 1} a(k) * w(n-k)$$

Notes: The w(i) are the filter internal states and are saved in the delay line during computation.

The delay line must be stored in DPRAM. Before the first call to the function, the delay line must be initialized to 0. The length of the delay line must be equal to the number of b(i) coefficients. As the number of b(i) coefficients is supposed to be equal to the number of a(i) coefficients, the length of the delay line is equal to IirNumberCoeffs/2.

Library users must care about the a(i) coefficients. If their modulus are too great, the filter internal states w(i) can saturate. The output samples are in this condition false.

The coefficient a_0 must be initialized to 0.

There is a saturation to 1 (respectively -1) if the computed output sample is greater than 1 (respectively lower than -1).

Example of the use for iir2_q15_q15_q15:

```
   #define IirNumberCoeffs NumberOfCoefficients
   #define NbOutput NumberOfOutput
   iram short Delayline[IirNumberCoeffs/2];
   iram short Coeff[IirNumberCoeffs];
   void main()
   {
   // Data definition
   short Input[NbOutput];
   short Out[NbOutput];
   // Here should be the coef[] and input[] initialization.
See the Test part...
   // Declare a filter structure
   struct Filter Filt;

   //Filter initialisation
   Filt.NumberCoeffs = IirNumberCoeffs;
   Filt.NumberOutputSamples = NbOutput;
   Filt.CoeffPtr = Coeff;
   Filt.InputPtr = Input;
   Filt.OutputPtr = Out;
   Filt.DelayLinePtr = Delayline;
   //Filter processing
   iir2_q15_q15_q15(&Filt);
   }
```

Coefficients must be stored in memory as follow:

```
   addr Iir.CoeffPtr     ----> b0
   addr Iir.CoeffPtr + 2 ----> b1
   ...
   addr Iir.CoeffPtr + 2 * (IirNumberCoeffs/2 - 1)---->
b(IirNumberCoeff/2)-1
   addr Iir.CoeffPtr + 2 * (IirNumberCoeffs/2)----> a0 = 0
   addr Iir.CoeffPtr + 2 * (IirNumberCoeffs/2 + 1) ----> a1
   ...
   addr Iir.CoeffPtr + 2 * (IirNumberCoeffs - 1)---->
a(IirNumberCoeff/2)-1
```

Assembly source code: `iir2_q15_q15_q15.asm`

Code size and cycles: Size: 130 bytes

Instruction Cycles: 36 + NbOutput * (17 + IirNumberCoeffs)

Test: Two test cases are provided. For each test case, two files allow to test the function: an input vector file (dirFilterX_q15_q15_q15.in) and an output vector file (dirfilterX_q15_q15_q15.res). The configuration is the following:

number of coefficients: 32

number of output samples: 201

To perform the test, initialize the Delay line[] to zero, then Coeff[] and Input[] with the dirfilterX_q15_q15_q15.in file: the first values are the coefficients of the filter and the rest of the values are the input values. (See the example for declarations).

6.4.2 iirBiquad5_q15_q15_q15

iirBiquad5_q15_q15_q15 (struct Filter *IirPtr).

Description: Real IIR filter of 16-bit fractional inputs, coefficients and outputs. The filter implemented is an IIR cascaded biquad cells form.

Argument: IirPtr Filter pointer.

Algorithm:

$$H(z) = \prod_{k=0}^{Q-1} \frac{b_{0k} + b_{1k} \times z^{-1} + b_{2k} \times z^{-2}}{1 - a_{1k} \times z^{-1} - a_{2k} \times z^{-2}}$$

with Q = filter order/2, the number of cells

Notes: The filter order must be an even number. Each biquad cells is implemented in the direct form II.

The delay line must be stored in DPRAM. Its size is 2*3*Q. Before the first call to the function, the delay line must be initialized to 0. The coefficient a_0 is fixed to 1.

Library users must care about the a(i) coefficients. If they are too big in modulus, the filter internal states w(i) can saturate. The output samples are in this condition false. This problem is less important in iirBiquad5_q15_q15_q15 than in iir2_q15_q15_q15 because there are less a(i) coefficients used to compute biquad cells. There can be some error bits because of the internal rounding.

There is a saturation to 1 (respectively -1) if the computed output sample is greater than 1 (respectively lower than -1).

Example of the use for iirBiquad5_q15_q15_q15:

```
   #define IirNumberCells NumberOfCells
   #define NbOutput NumberOfOutput
   iram short Delayline[IirNumberCells*3];
   iram short Coeff[IirNumberCells*5];
   void main()
   {
   // Data definition
   short Input[NbOutput];
   short Out[NbOutput];
   // Here should be the coef[] and input[] initialization.
See the Test part...
   // Declare a filter structure
   struct Filter Filt;
   //Filter initialisation
   Filt.NumberCoeffs = IirNumberCoeffs;
   Filt.NumberOutputSamples = NbOutput;
   Filt.CoeffPtr = Coeff;
   Filt.InputPtr = Input;
   Filt.OutputPtr = Out;
   Filt.DelayLinePtr = Delayline;
   //Filter processing
   iirBiquad5_q15_q15_q15(&Filt);
   }
```

```
Coefficients must be stored in memory as follow:
   addr Iir.CoeffPtr ----> $b_{00}$
   addr Iir.CoeffPtr + 2 ----> $b_{10}$
   addr Iir.CoeffPtr + 4----> $b_{20}$
addr Iir.CoeffPtr +6----> $a_{10}$
   addr Iir.CoeffPtr + 8 ----> $a_{20}$
   ...
   addr Iir.CoeffPtr + 2 * 5 * (Q - 1)----> $b_{0(Q-1)}$
   addr Iir.CoeffPtr + 2 * 5 *(Q-1) + 2----> $b_{1(Q-1)}$
   addr Iir.CoeffPtr + 2 * 5 * (Q - 1) + 4 ----> $b_{2(Q-1)}$
   addr Iir.CoeffPtr + 2 * 5 * (Q - 1) + 6 ----> $a_{1(Q-1)}$
   addr Iir.CoeffPtr + 2 * 5 *(Q-1) + 8----> $a_{2(Q-1)}$
```

Assembly source code: `iirBiquad5_q15_q15_q15.asm`

Code size and cycles: Size: 136 bytes

Instruction Cycles: 34 + NbOutput * (12 + 15 * IirNumberCells)

Test: Two test cases are provided. For each test case, two files allow to test the function: an input vector file (BiqFilterX_q15_q15_q15.in) and an output vector file (BiqfilterX_q15_q15_q15.res). The configuration is the following:

number of cells: 3

number of output samples: 201

To perform the test, initialize the Delay line[] to zero, then Coeff[] and Input[] with the BiqfilterX_q15_q15_q15.in file. The first values are the coefficients of the filter and the rest of the values are the input values. (See the example for declarations).

7 Revision history

The ST10-DSP library may be periodically updated, incorporating new functions and fixes as they are available. Read the README.TXT available in the root directory of every release.

Table 3. Document revision history

Date	Revision	Changes
28-May-2002	1	Initial release.
04-Mar-2008	2	Document reformatted. Reference to ST10F280 removed. End-user license agreement section removed.
18-Sep-2013	3	Updated Disclaimer