GDFLIB User's Guide

DSP56800EX

Document Number: DSP56800EXGDFLIBUG
Rev. 2, 10/2015

Review Draft

Contents

Section number | **Title** | **Page**

Chapter 1
Library

1.1 Introduction

1.1.1 Overview

This user's guide describes the General Digital Filters Library (GDFLIB) for the family of DSP56800EX core-based digital signal controllers. This library contains optimized functions.

1.1.2 Data types

GDFLIB supports several data types: (un)signed integer, fractional, and accumulator. The integer data types are useful for general-purpose computation; they are familiar to the MPU and MCU programmers. The fractional data types enable powerful numeric and digital-signal-processing algorithms to be implemented. The accumulator data type is a combination of both; that means it has the integer and fractional portions.

The following list shows the integer types defined in the libraries:

- Unsigned 16-bit integer —<0 ; 65535> with the minimum resolution of 1
- Signed 16-bit integer —<-32768 ; 32767> with the minimum resolution of 1
- Unsigned 32-bit integer —<0 ; 4294967295> with the minimum resolution of 1
- Signed 32-bit integer —<-2147483648 ; 2147483647> with the minimum resolution of 1

The following list shows the fractional types defined in the libraries:

- Fixed-point 16-bit fractional —$<-1 ; 1 - 2^{-15}>$ with the minimum resolution of 2^{-15}
- Fixed-point 32-bit fractional —$<-1 ; 1 - 2^{-31}>$ with the minimum resolution of 2^{-31}

The following list shows the accumulator types defined in the libraries:

- Fixed-point 16-bit accumulator —<-256.0 ; 256.0 - 2^{-7}> with the minimum resolution of 2^{-7}
- Fixed-point 32-bit accumulator —<-65536.0 ; 65536.0 - 2^{-15}> with the minimum resolution of 2^{-15}

1.1.3 API definition

GDFLIB uses the types mentioned in the previous section. To enable simple usage of the algorithms, their names use set prefixes and postfixes to distinguish the functions' versions. See the following example:

```
f32Result = MLIB_Mac_F32lss(f32Accum, f16Mult1, f16Mult2);
```

where the function is compiled from four parts:

- MLIB—this is the library prefix
- Mac—the function name—Multiply-Accumulate
- F32—the function output type
- lss—the types of the function inputs; if all the inputs have the same type as the output, the inputs are not marked

The input and output types are described in the following table:

Table 1-1. Input/output types

Type	Output	Input
frac16_t	F16	s
frac32_t	F32	l
acc32_t	A32	a

1.1.4 Supported compilers

GDFLIB for the DSP56800EX core is written in assembly language with C-callable interface. The library is built and tested using the following compilers:

- CodeWarrior™ Development Studio

For the CodeWarrior™ Development Studio, the library is delivered in the *gdflib.lib* file.

The interfaces to the algorithms included in this library are combined into a single public interface include file, *gdflib.h*. This is done to lower the number of files required to be included in your application.

1.1.5 Special issues

1. The equations describing the algorithms are symbolic. If there is positive 1, the number is the closest number to 1 that the resolution of the used fractional type allows. If there are maximum or minimum values mentioned, check the range allowed by the type of the particular function version.
2. The library functions require the core saturation mode to be turned off, otherwise the results can be incorrect. Several specific library functions are immune to the setting of the saturation mode.
3. The library functions round the result (the API contains Rnd) to the nearest (two's complement rounding) or to the nearest even number (convergent round). The mode used depends on the core option mode register (OMR) setting. See the core manual for details.
4. All non-inline functions are implemented without storing any of the volatile registers (refer to the compiler manual) used by the respective routine. Only the non-volatile registers (C10, D10, R5) are saved by pushing the registers on the stack. Therefore, if the particular registers initialized before the library function call are to be used after the function call, it is necessary to save them manually.

1.2 Library integration into project (CodeWarrior™ Development Studio)

This section provides a step-by-step guide to quickly and easily integrate the GDFLIB into an empty project using CodeWarrior™ Development Studio. This example uses the MC56F84789 part, and the default installation path (C:\Freescale\FSLESL \DSP56800EX_FSLESL_4.2) is supposed. If you have a different installation path, you must use that path instead.

1.2.1 New project

To start working on an application, create a new project. If the project already exists and is open, skip to the next section. Follow the steps given below to create a new project.

1. Launch CodeWarrior™ Development Studio.

2. Choose File > New > Bareboard Project, so that the "New Bareboard Project" dialog appears.
3. Type a name of the project, for example, MyProject01.
4. If you don't use the default location, untick the “Use default location” checkbox, and type the path where you want to create the project folder; for example, C:\CWProjects\MyProject01, and click Next. See Figure 1-1.

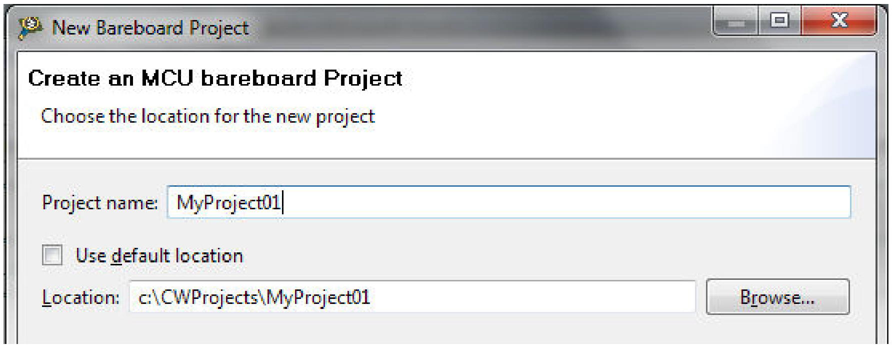

Figure 1-1. Project name and location

5. Expand the tree by clicking the 56800/E (DSC) and MC56F84789. Select the Application option and click Next. See Figure 1-2.

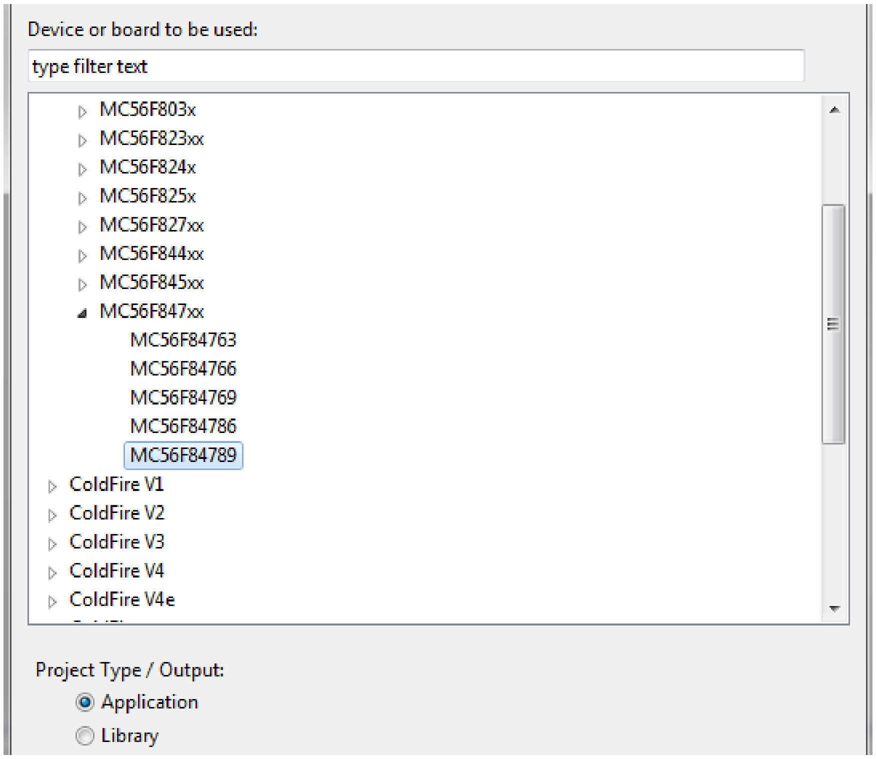

Figure 1-2. Processor selection

6. Now select the connection that will be used to download and debug the application. In this case, select the option P&E USB MultiLink Universal[FX] / USB MultiLink and Freescale USB TAP, and click Next. See Figure 1-3.

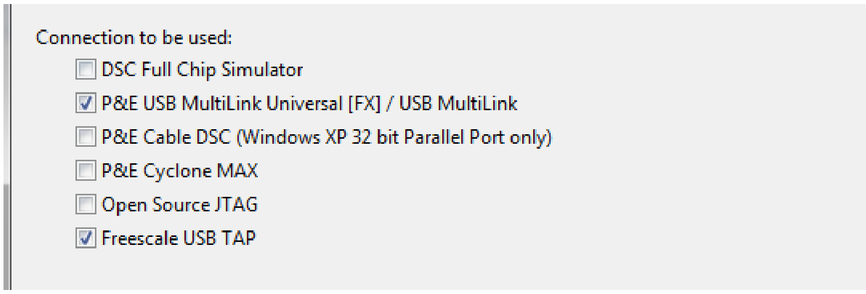

Figure 1-3. Connection selection

7. From the options given, select the Simple Mixed Assembly and C language, and click Finish. See Figure 1-4.

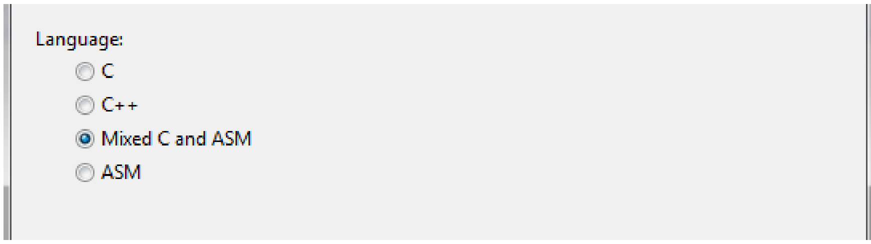

Figure 1-4. Language choice

The new project is now visible in the left-hand part of CodeWarrior™ Development Studio. See Figure 1-5.

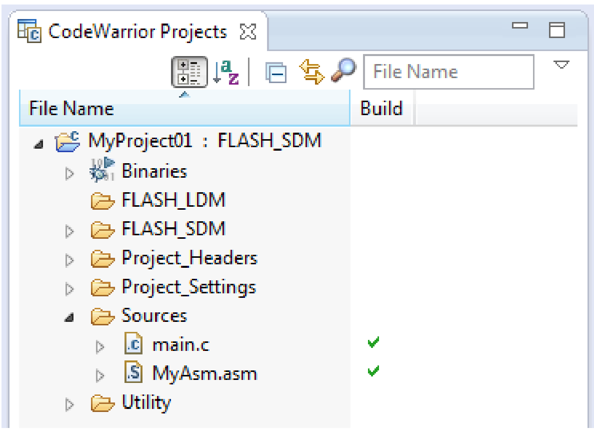

Figure 1-5. Project folder

1.2.2 Library path variable

To make the library integration easier, create a variable that will hold the information about the library path.

1. Right-click the MyProject01 node in the left-hand part and click Properties, or select Project > Properties from the menu. The project properties dialog appears.

2. Expand the Resource node and click Linked Resources. See Figure 1-6.

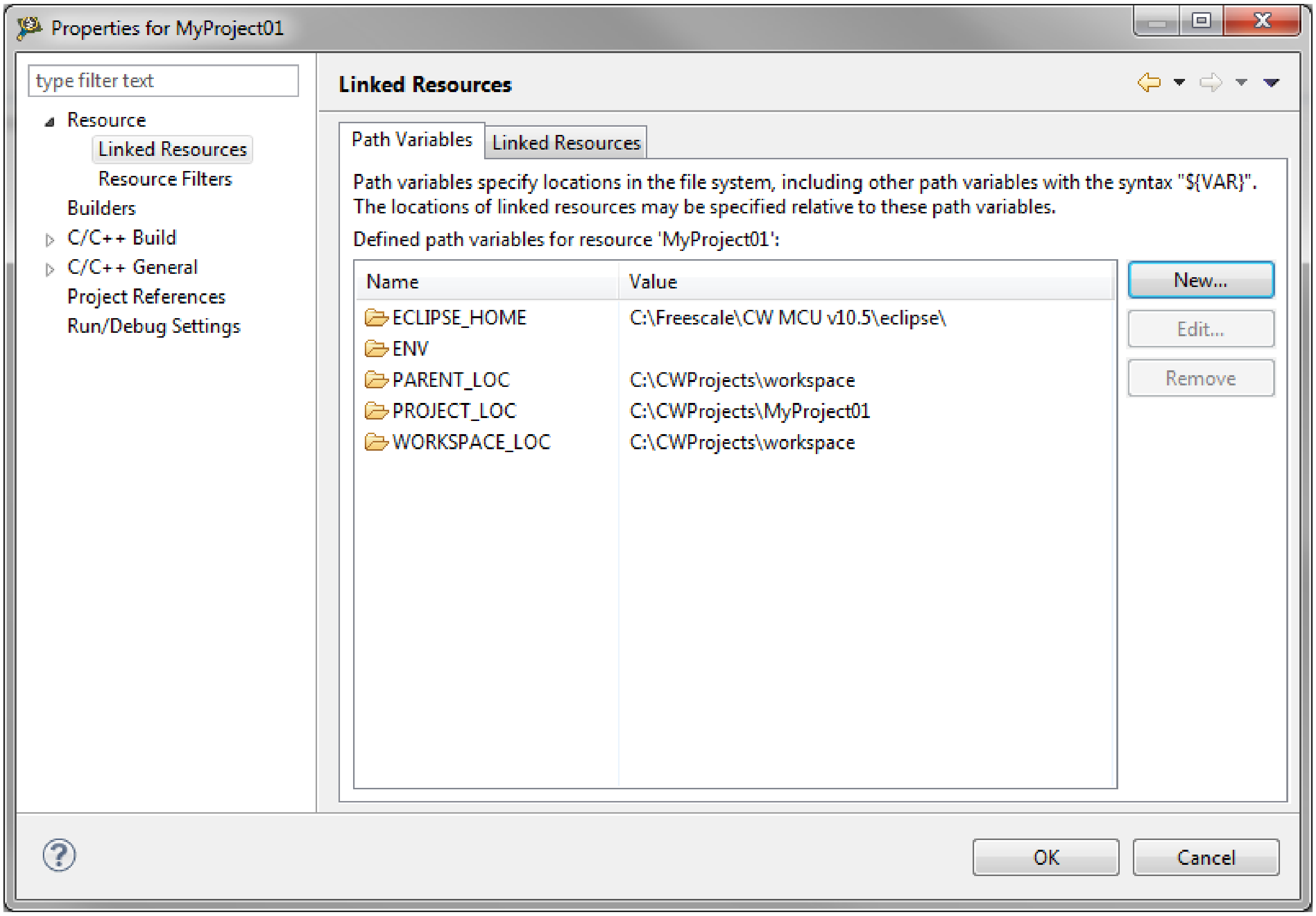

Figure 1-6. Project properties

3. Click the 'New…' button on the right-hand side.
4. In the dialog that appears (see Figure 1-7), type this variable name into the Name box: FSLESL_LOC
5. Select the library parent folder by clicking 'Folder…' or just typing the following path into the Location box: C:\Freescale\FSLESL\DSP56800EX_FSLESL_4.2_CW and click OK.
6. Click OK in the previous dialog.

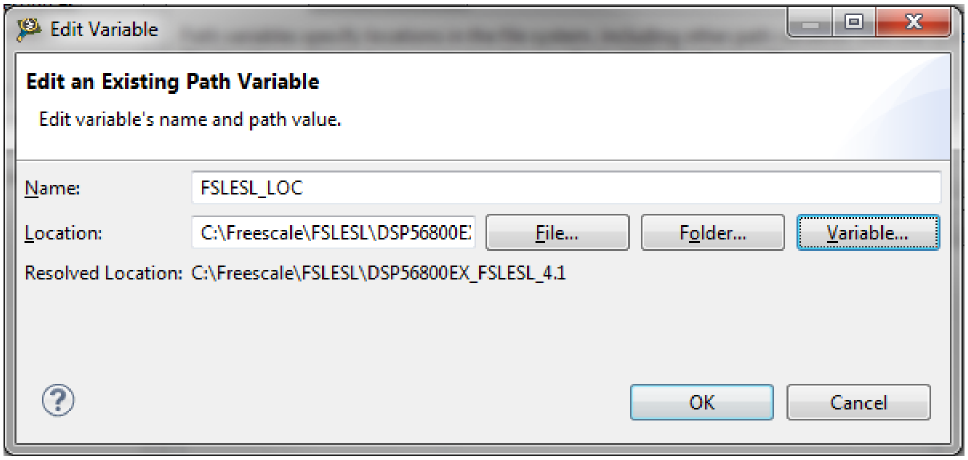

Figure 1-7. New variable

1.2.3 Library folder addition

To use the library, add it into the CodeWarrior Project tree dialog.

1. Right-click the MyProject01 node in the left-hand part and click New > Folder, or select File > New > Folder from the menu. A dialog appears.
2. Click Advanced to show the advanced options.
3. To link the library source, select the third option—Link to alternate location (Linked Folder).
4. Click Variables…, and select the FSLESL_LOC variable in the dialog that appears, click OK, and/or type the variable name into the box. See Figure 1-8.
5. Click Finish, and you will see the library folder linked in the project. See Figure 1-9

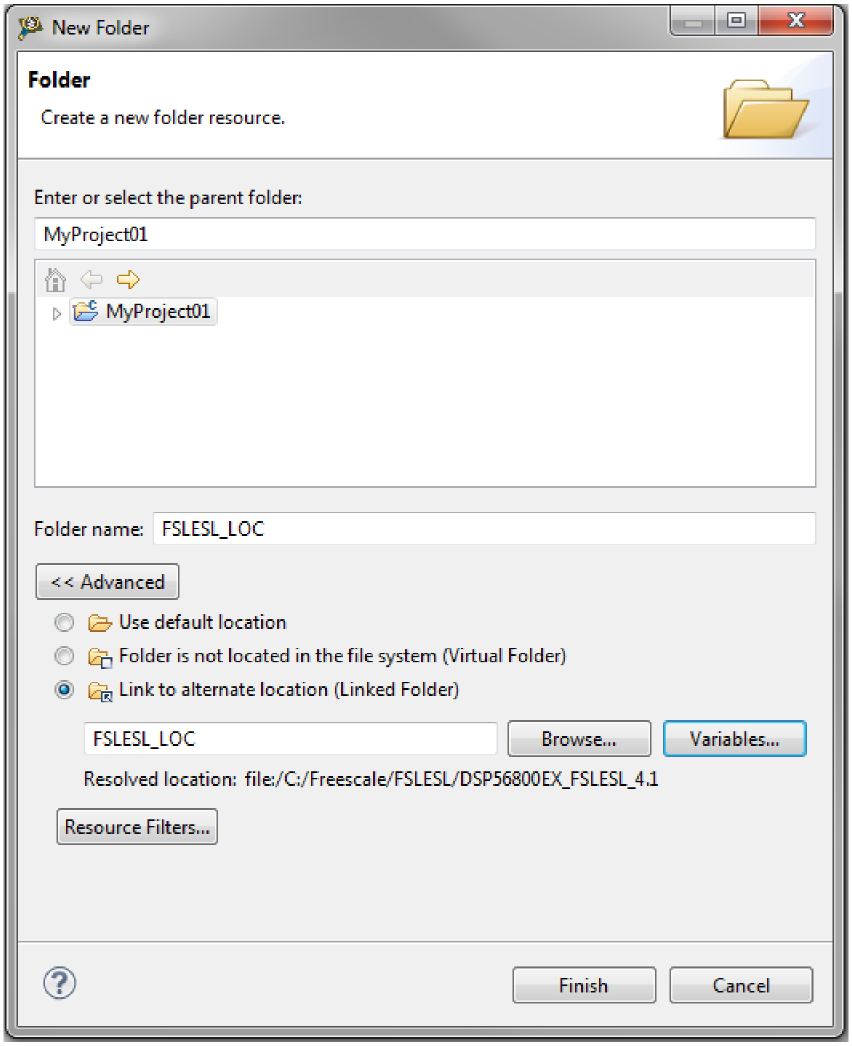

Figure 1-8. Folder link

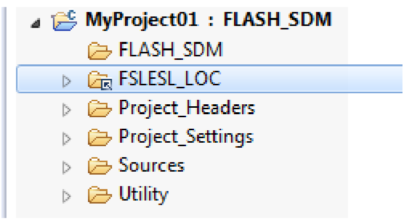

Figure 1-9. Projects libraries paths

1.2.4 Library path setup

GDFLIB requires MLIB to be included too. Therefore, the following steps show the inclusion of all dependent modules.

1. Right-click the MyProject01 node in the left-hand part and click Properties, or select Project > Properties from the menu. A dialog with the project properties appears.

2. Expand the C/C++ Build node, and click Settings.
3. In the right-hand tree, expand the DSC Linker node, and click Input. See Figure 1-11.
4. In the third dialog Additional Libraries, click the 'Add…' icon, and a dialog appears.
5. Look for the FSLESL_LOC variable by clicking Variables…, and then finish the path in the box by adding one of the following:
   - ${FSLESL_LOC}\MLIB\mlib_SDM.lib—for small data model projects
   - ${FSLESL_LOC}\MLIB\mlib_LDM.lib—for large data model projects
6. Tick the box Relative To, and select FSLESL_LOC next to the box. See Figure 1-9. Click OK.
7. Click the 'Add…' icon in the third dialog Additional Libraries.
8. Look for the FSLESL_LOC variable by clicking Variables…, and then finish the path in the box by adding one of the following:
   - ${FSLESL_LOC}\GDFLIB\gdflib_SDM.lib—for small data model projects
   - ${FSLESL_LOC}\GDFLIB\gdflib_LDM.lib—for large data model projects
9. Tick the box Relative To, and select FSLESL_LOC next to the box. Click OK.
10. Now, you will see the libraries added in the box. See Figure 1-11.

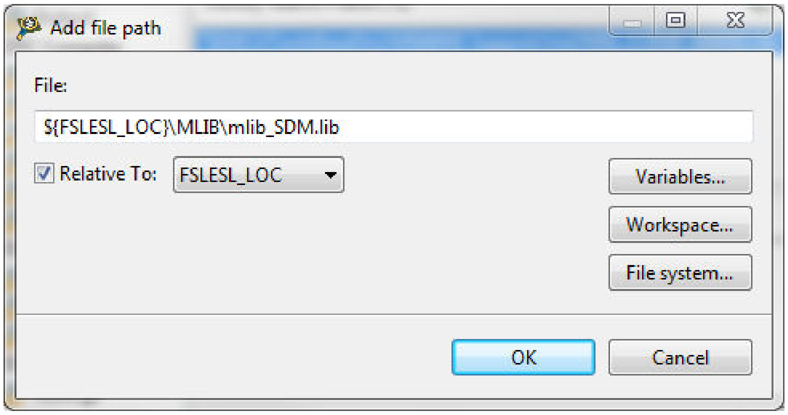

Figure 1-10. Library file inclusion

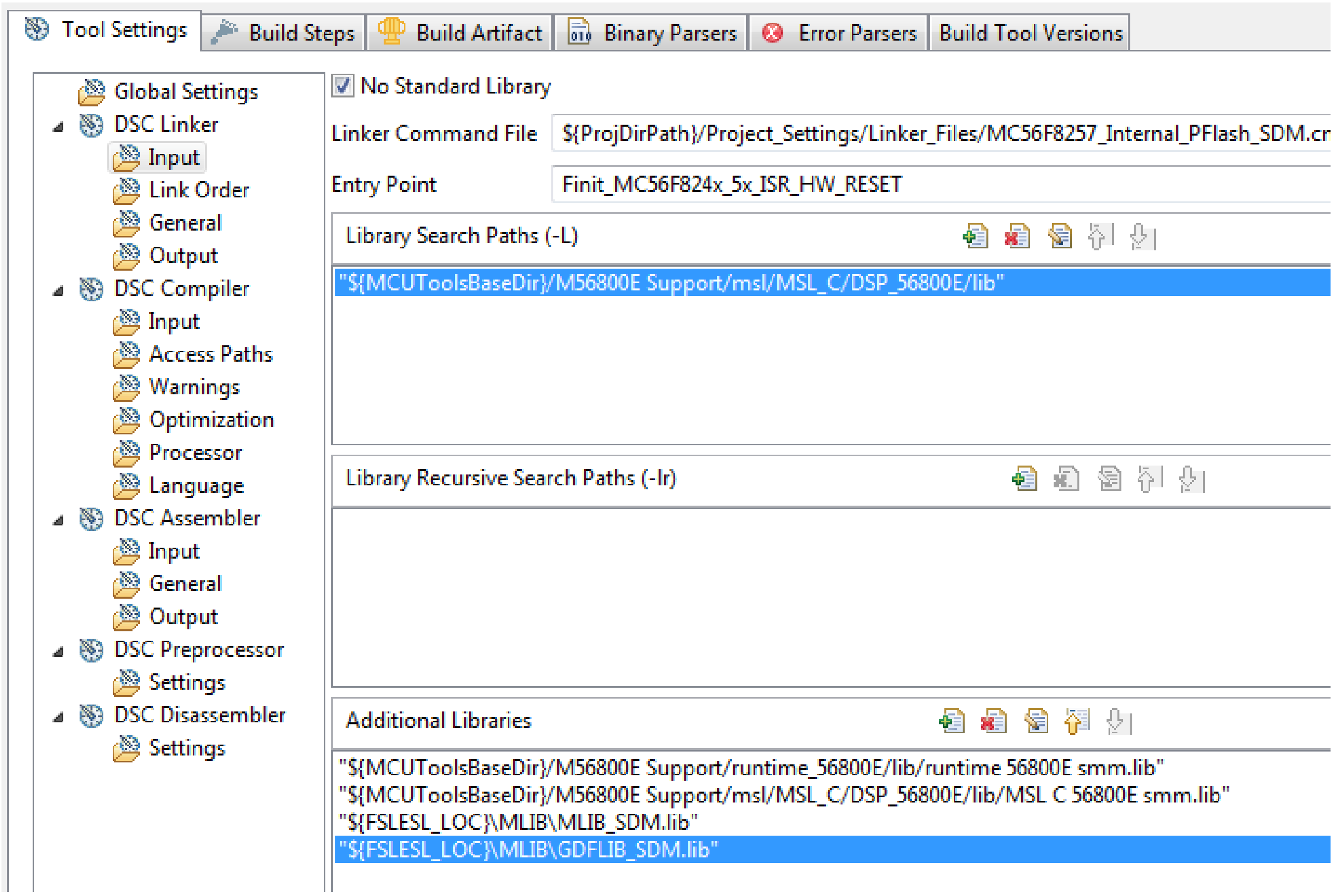

Figure 1-11. Linker setting

11. In the tree under the DSC Compiler node, click Access Paths.
12. In the Search User Paths dialog (#include "…"), click the 'Add…' icon, and a dialog will appear.
13. Look for the FSLESL_LOC variable by clicking Variables…, and then finish the path in the box to be: ${FSLESL_LOC}\MLIB\include.
14. Tick the box Relative To, and select FSLESL_LOC next to the box. See Figure 1-12. Click OK.
15. Click the 'Add…' icon in the Search User Paths dialog (#include "…").
16. Look for the FSLESL_LOC variable by clicking Variables…, and then finish the path in the box to be: ${FSLESL_LOC}\GDFLIB\include.
17. Tick the box Relative To, and select FSLESL_LOC next to the box. Click OK.
18. Now you will see the paths added in the box. See Figure 1-13. Click OK.

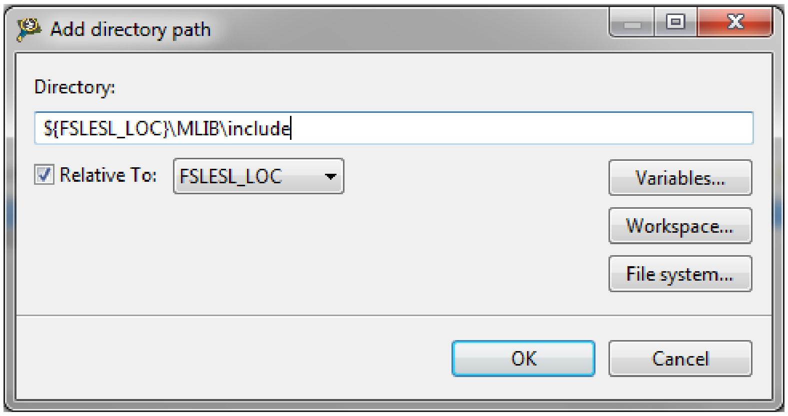

Figure 1-12. Library include path addition

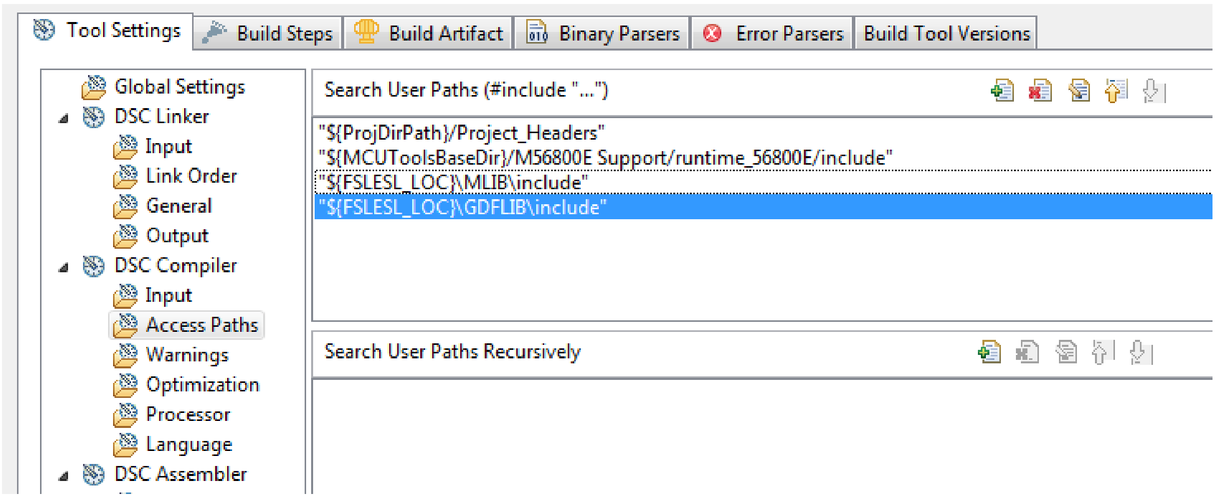

Figure 1-13. Compiler setting

The final step is typing the #include syntax into the code. Include the library into the *main.c* file. In the left-hand dialog, open the Sources folder of the project, and double-click the *main.c* file. After the *main.c* file opens up, include the following lines into the #include section:

```
#include "mlib.h"
#include "gdflib.h"
```

When you click the Build icon (hammer), the project will be compiled without errors.

Chapter 2
Algorithms in detail

2.1 GDFLIB_FilterIIR1

This function calculates the first-order direct form 1 IIR filter.

For a proper use, it is recommended that the algorithm is initialized by the GDFLIB_FilterIIR1Init function, before using the GDFLIB_FilterIIR1 function. The GDFLIB_FilterIIR1Init function initializes the buffer and coefficients of the first-order IIR filter.

The GDFLIB_FilterIIR1 function calculates the first-order infinite impulse response (IIR) filter. The IIR filters are also called recursive filters, because both the input and the previously calculated output values are used for calculation. This form of feedback enables the transfer of energy from the output to the input, which leads to an infinitely long impulse response (IIR). A general form of the IIR filter, expressed as a transfer function in the Z-domain, is described as follows:

$$H(z) = \frac{B(z)}{A(z)} = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2} + \ldots + b_N z^{-N}}{1 + a_1 z^{-1} + a_2 z^{-2} + \ldots + a_N z^{-N}}$$

Equation 1.

where N denotes the filter order. The first-order IIR filter in the Z-domain is expressed as follows:

$$H(z) = \frac{B(z)}{A(z)} = \frac{b_0 + b_1 z^{-1}}{1 + a_1 z^{-1}}$$

Equation 2.

which is transformed into a time-domain difference equation as follows:

$$y(k) = b_0 x(k) + b_1 x(k-1) - a_1 y(k-1)$$

Equation 3.

The filter difference equation is implemented in the digital signal controller directly, as given in Equation 3 on page 17; this equation represents a direct-form 1 first-order IIR filter, as shown in Figure 2-1.

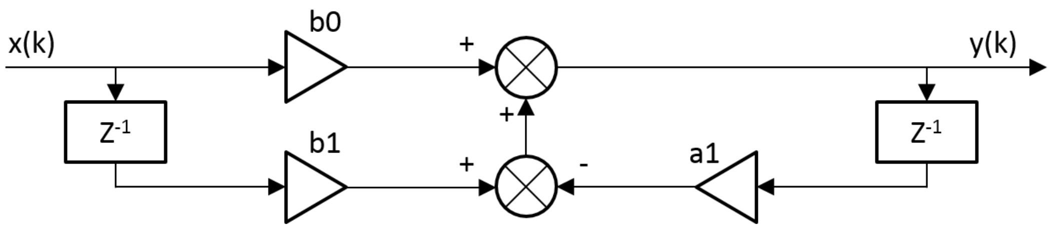

Figure 2-1. Direct form 1 first-order IIR filter

The coefficients of the filter shown in Figure 2-1 can be designed to meet the requirements for the first-order low-pass filter (LPF) or high-pass filter (HPF). The coefficient quantization error is not important in the case of a first-order filter due to a finite precision arithmetic. A higher-order LPF or HPF can be obtained by connecting a number of first-order filters in series. The number of connections gives the order of the resulting filter.

The filter coefficients must be defined before calling this function. As some coefficients can be greater than 1 (and lesser than 2), the coefficients are scaled down (divided) by 2.0 for the fractional version of the algorithm. For faster calculation, the A coefficient is sign-inverted. The function returns the filtered value of the input in the step k, and stores the input and the output values in the step k into the filter buffer.

2.1.1 Available versions

This function is available in the following versions:

- Fractional output - the output is the fractional portion of the result; the result is within the range <-1 ; 1).

The available versions of the GDFLIB_FilterIIR1Init function are shown in the following table:

Table 2-1. Init function versions

Function name	Parameters	Result type	Description
GDFLIB_FilterIIR1Init_F16	GDFLIB_FILTER_IIR1_T_F32 *	void	Filter initialization (reset) function. The parameters' structure is pointed to by a pointer.

The available versions of the GDFLIB_FilterIIR1 function are shown in the following table:

Table 2-2. Function versions

Function name	Input type	Parameters	Result type	Description
GDFLIB_FilterIIR1_F16	frac16_t	GDFLIB_FILTER_IIR1_T_F32 *	frac16_t	The input argument is a 16-bit fractional value of the input signal to be filtered within the range <-1 ; 1). The parameters' structure is pointed to by a pointer. The function returns a 16-bit fractional value within the range <-1 ; 1).

2.1.2 GDFLIB_FILTER_IIR1_T_F32

Variable name	Input type	Description
sFltCoeff	GDFLIB_FILTER_IIR1_COEFF_T_F32 *	Substructure containing filter coefficients.
f32FltBfrY[1]	frac32_t	Internal buffer of y-history. Controlled by the algorithm.
f16FltBfrX[1]	frac16_t	Internal buffer of x-history. Controlled by the algorithm.

2.1.3 GDFLIB_FILTER_IIR1_COEFF_T_F32

Variable name	Type	Description
f32B0	frac32_t	B0 coefficient of the IIR1 filter. Set by the user, and must be divided by 2.
f32B1	frac32_t	B1 coefficient of the IIR1 filter. Set by the user, and must be divided by 2.
f32A1	frac32_t	A1 (sign-inverted) coefficient of the IIR1 filter. Set by the user, and must be divided by -2.

2.1.4 Declaration

The available GDFLIB_FilterIIR1Init functions have the following declarations:

```
void GDFLIB_FilterIIR1Init_F16(GDFLIB_FILTER_IIR1_T_F32 *psParam)
```

The available GDFLIB_FilterIIR1 functions have the following declarations:

```
frac16_t GDFLIB_FilterIIR1_F16(frac16_t f16InX, GDFLIB_FILTER_IIR1_T_F32 *psParam)
```

2.1.5 Calculation of filter coefficients

There are plenty of methods for calculating the coefficients. The following example shows the use of Matlab to set up a low-pass filter with the 500 Hz sampling frequency, and 240 Hz stopped frequency with a 20 dB attenutation. Maximum passband ripple is 3 dB at the cut-off frequency of 50 Hz.

```
% sampling frequency 500 Hz, low pass
Ts = 1 / 500

% cut-off frequency 50 Hz
Fc = 50

% max. passband ripple 3 dB
Rp = 3

% stopped frequency 240Hz
Fs = 240

% attenuation 20 dB
Rs = 20

% checking order of the filter
n = buttord(2 * Ts * Fc, 2 * Ts * Fs, Rp, Rs)
% n = 1, i.e. the filter is achievable with the 1st order

% getting the filter coefficients
[b, a] = butter(n, 2 * Ts * Fc, 'low');

% the coefs are:
% b0 = 0.245237275252786, b1 = 0.245237275252786
% a0 = 1.0000, a1 = -0.509525449494429
```

The filter response is shown in Figure 2-2.

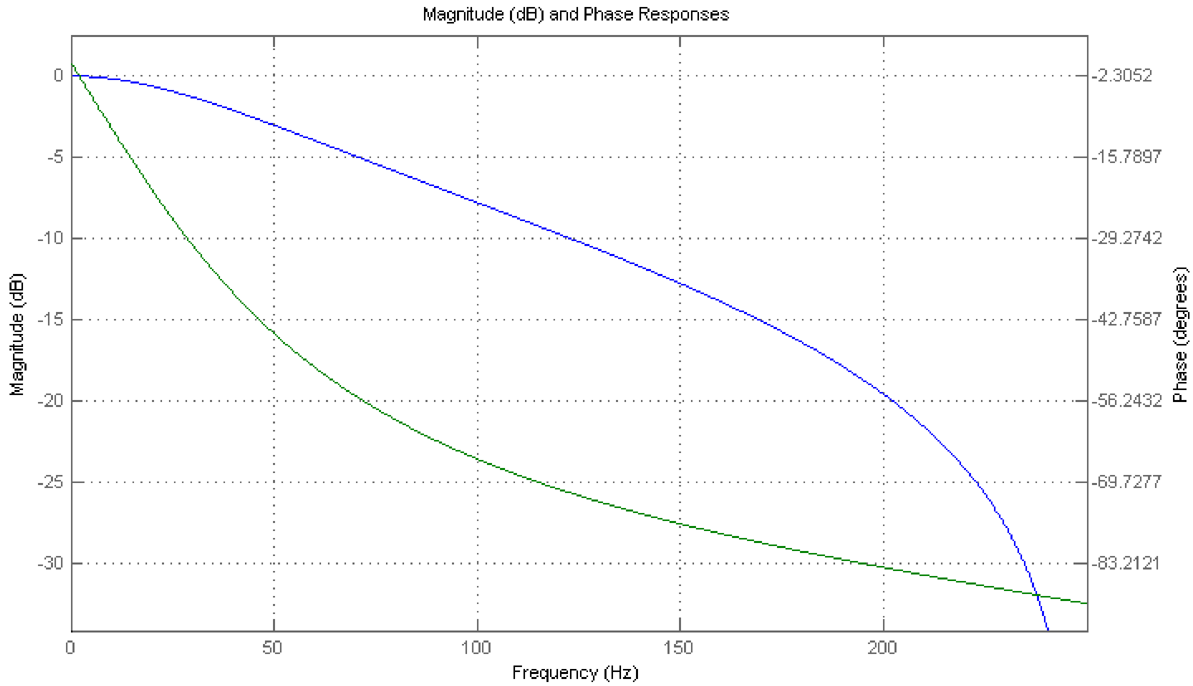

Figure 2-2. Filter response

2.1.6 Function use

The use of the GDFLIB_FilterIIR1Init and GDFLIB_FilterIIR1 functions is shown in the following example. The filter uses the above-calculated coefficients:

```
#include "gdflib.h"

static frac16_t f16Result;
static frac16_t f16InX;
static GDFLIB_FILTER_IIR1_T_F32 sFilterParam;

void Isr(void);

void main(void)
{
   sFilterParam.sFltCoeff.f32B0 = FRAC32(0.245237275252786 / 2.0);
   sFilterParam.sFltCoeff.f32B1 = FRAC32(0.245237275252786 / 2.0);
   sFilterParam.sFltCoeff.f32A1 = FRAC32(-0.509525449494429 / -2.0);

   GDFLIB_FilterIIR1Init_F16(&sFilterParam);

   f16InX = FRAC16(0.1);
}

/* periodically called function */
void Isr(void)
{
     f16Result = GDFLIB_FilterIIR1_F16(f16InX, &sFilterParam);
}
```

2.2 GDFLIB_FilterIIR2

This function calculates the second-order direct-form 1 IIR filter.

For a proper use, it is recommended that the algorithm is initialized by the GDFLIB_FilterIIR2Init function, before using the GDFLIB_FilterIIR2 function. The GDFLIB_FilterIIR2Init function initializes the buffer and coefficients of the second-order IIR filter.

The GDFLIB_FilterIIR2 function calculates the second-order infinite impulse response (IIR) filter. The IIR filters are also called recursive filters, because both the input and the previously calculated output values are used for calculation. This form of feedback enables the transfer of energy from the output to the input, which leads to an infinitely long impulse response (IIR). A general form of the IIR filter, expressed as a transfer function in the Z-domain, is described as follows:

$$H(z) = \frac{B(z)}{A(z)} = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2} + \ldots + b_N z^{-N}}{1 + a_1 z^{-1} + a_2 z^{-2} + \ldots + a_N z^{-N}}$$

Equation 4.

where N denotes the filter order. The second-order IIR filter in the Z-domain is expressed as follows:

$$H(z) = \frac{B(z)}{A(z)} = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2}}{1 + a_1 z^{-1} + a_2 z^{-2}}$$

Equation 5.

which is transformed into a time-domain difference equation as follows:

$$y(k) = b_0 x(k) + b_1 x(k-1) + b_2 x(k-2) - a_1 y(k-1) - a_2 y(k-2)$$

Equation 6.

The filter difference equation is implemented in the digital signal controller directly, as given in Equation 6 on page 22; this equation represents a direct-form 1 second-order IIR filter, as depicted in Figure 2-3.

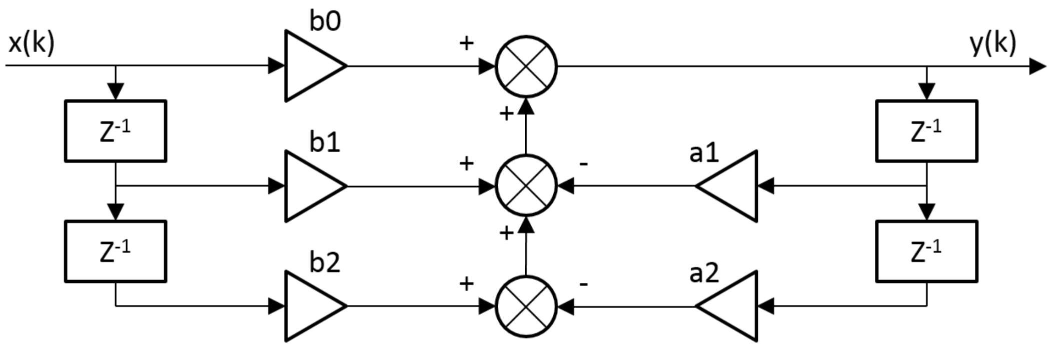

Figure 2-3. Direct-form 1 second-order IIR filter

The coefficients of the filter depicted in Figure 2-3 can be designed to meet the requirements for the second-order low-pass filter (LPF), high-pass filter (HPF), band-pass filter (BPF) or band-stop filter (BSF). The coefficient quantization error can be neglected in the case of a second-order filter due to a finite precision arithmetic. A higher-order LPF or HPF can be obtained by connecting a number of second-order filters in series. The number of connections gives the order of the resulting filter.

The filter coefficients must be defined before calling this function. As some coefficients can be greater than 1 (and lesser than 2), the coefficients are scaled down (divided) by 2.0 for the fractional version of the algorithm. For faster calculation, the A coefficients are sign-inverted. The function returns the filtered value of the input in the step k, and stores the input and output values in the step k into the filter buffer.

2.2.1 Available versions

This function is available in the following versions:

- Fractional output - the output is the fractional portion of the result; the result is within the range <-1 ; 1).

The available versions of the GDFLIB_FilterIIR2Init function are shown in the following table:

Table 2-3. Init function versions

Function name	Parameters	Result type	Description
GDFLIB_FilterIIR2Init_F16	GDFLIB_FILTER_IIR2_T_F32 *	void	Filter initialization (reset) function. The parameters' structure is pointed to by a pointer.

The available versions of the GDFLIB_FilterIIR2 function are shown in the following table:

Table 2-4. Function versions

Function name	Input type	Parameters	Result type	Description
GDFLIB_FilterIIR2_F16	frac16_t	GDFLIB_FILTER_IIR2_T_F32 *	frac16_t	Input argument is a 16-bit fractional value of the input signal to be filtered within the range <-1 ; 1). The parameters' structure is pointed to by a pointer. The function returns a 16-bit fractional value within the range <-1 ; 1).

2.2.2 GDFLIB_FILTER_IIR2_T_F32

Variable name	Input type	Description
sFltCoeff	GDFLIB_FILTER_IIR2_COEFF_T_F32 *	Substructure containing filter coefficients.
f32FltBfrY[2]	frac32_t	Internal buffer of y-history. Controlled by the algorithm.
f16FltBfrX[2]	frac16_t	Internal buffer of x-history. Controlled by the algorithm.

2.2.3 GDFLIB_FILTER_IIR2_COEFF_T_F32

Variable name	Type	Description
f32B0	frac32_t	B0 coefficient of the IIR2 filter. Set by the user, and must be divided by 2.
f32B1	frac32_t	B1 coefficient of the IIR2 filter. Set by the user, and must be divided by 2.
f32B2	frac32_t	B2 coefficient of the IIR2 filter. Set by the user, and must be divided by 2.
f32A1	frac32_t	A1 (sign-inverted) coefficient of the IIR2 filter. Set by the user, and must be divided by -2.
f32A2	frac32_t	A2 (sign-inverted) coefficient of the IIR2 filter. Set by the user, and must be divided by -2.

2.2.4 Declaration

The available GDFLIB_FilterIIR2Init functions have the following declarations:

```
void GDFLIB_FilterIIR2Init_F16(GDFLIB_FILTER_IIR2_T_F32 *psParam)
```

The available GDFLIB_FilterIIR2 functions have the following declarations:

```
frac16_t GDFLIB_FilterIIR2_F16(frac16_t f16InX, GDFLIB_FILTER_IIR2_T_F32 *psParam)
```

2.2.5 Calculation of filter coefficients

There are plenty of methods for calculating the coefficients. The following example shows the use of Matlab to set up a stopband filter with the 1000 Hz sampling frequency, 100 Hz stop frequency with 10 dB attenuation, and 30 Hz bandwidth. Maximum passband ripple is 3 dB.

```
% sampling frequency 1000 Hz, stop band
Ts = 1 / 1000

% center stop frequency 100 Hz
Fc = 50

% attenuation 10 dB
Rs = 10

% bandwidth 30 Hz
Fbw = 30

% max. passband ripple 3 dB
Rp = 3

% checking order of the filter
n = buttord(2 * Ts * [Fc - Fbw /2 Fc + Fbw / 2], 2 * Ts * [Fc - Fbw Fc + Fbw], Rp, Rs)
% n = 2, i.e. the filter is achievable with the 2nd order

% getting the filter coefficients
[b, a] = butter(n / 2, 2 * Ts * [Fc - Fbw /2 Fc + Fbw / 2], 'stop')

% the coefs are:
% b0 = 0.913635972986238, b1 = -1.745585863109291, b2 = 0.913635972986238
% a0 = 1.0000, a1 = -1.745585863109291, a2 = 0.827271945972476
```

The filter response is shown in Figure 2-4.

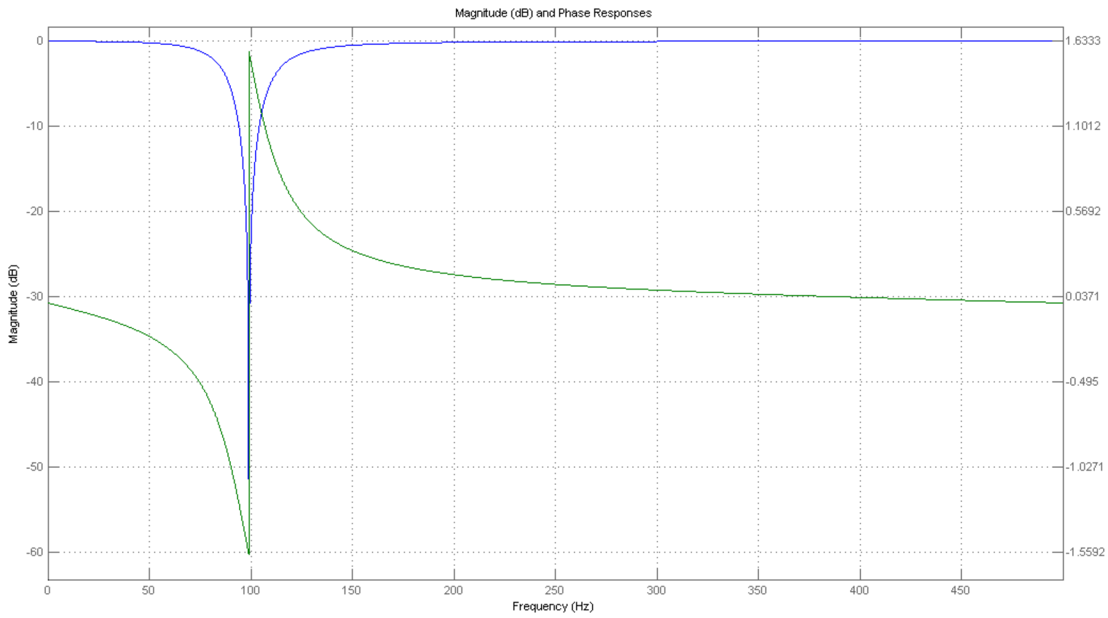

Figure 2-4. Filter response

2.2.6 Function use

The use of the GDFLIB_FilterIIR2Init and GDFLIB_FilterIIR2 functions is shown in the following example. The filter uses the above-calculated coefficients:

```
#include "gdflib.h"

static frac16_t f16Result;
static frac16_t f16InX;
static GDFLIB_FILTER_IIR2_T_F32 sFilterParam;

void Isr(void);

void main(void)
{
   sFilterParam.sFltCoeff.f32B0 = FRAC32(0.913635972986238 / 2.0);
   sFilterParam.sFltCoeff.f32B1 = FRAC32(-1.745585863109291 / 2.0);
   sFilterParam.sFltCoeff.f32B2 = FRAC32(0.913635972986238 / 2.0);
   sFilterParam.sFltCoeff.f32A1 = FRAC32(-1.745585863109291 / -2.0);
   sFilterParam.sFltCoeff.f32A2 = FRAC32(0.827271945972476 / -2.0);

   GDFLIB_FilterIIR2Init_F16(&sFilterParam);

   f16InX = FRAC16(0.1);
}

/* periodically called function */
void Isr(void)
{
     f16Result = GDFLIB_FilterIIR2_F16(f16InX, &sFilterParam);
}
```

2.3 GDFLIB_FilterIIR3

This function calculates the third-order direct-form 1 IIR filter.

For a proper use, it is recommended to initialize the algorithm by the GDFLIB_FilterIIR3Init function before using the GDFLIB_FilterIIR3 function. The GDFLIB_FilterIIR3Init function initializes the buffer and coefficients of the third-order IIR filter.

The GDFLIB_FilterIIR3 function calculates the third-order infinite impulse response (IIR) filter. The IIR filters are also called recursive filters because both the input and the previously calculated output values are used for calculation. This form of feedback enables the transfer of energy from the output to the input, which leads to an infinitely long impulse response (IIR). A general form of the IIR filter (expressed as a transfer function in the Z-domain) is described as follows:

$$H(z) = \frac{B(z)}{A(z)} = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2} + \ldots + b_N z^{-N}}{1 + a_1 z^{-1} + a_2 z^{-2} + \ldots + a_N z^{-N}}$$

Equation 7.

where N denotes the filter order. The third-order IIR filter in the Z-domain is expressed as follows:

$$H(z) = \frac{B(z)}{A(z)} = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2} + b_3 z^{-3}}{1 + a_1 z^{-1} + a_2 z^{-2} + a_3 z^{-3}}$$

Equation 8.

which is transformed into a time-domain difference equation as follows:

$$y(k) = b_0 x(k) + b_1 x(k-1) + b_2 x(k-2) + b_3 x(k-3) - a_1 y(k-1) - a_2 y(k-2) - a_3 y(k-3)$$

Equation 9.

The filter difference equation is implemented in the digital signal controller directly, as given in Equation 9 on page 27. This equation represents a direct-form 1 third-order IIR filter, as depicted in Figure 2-5.

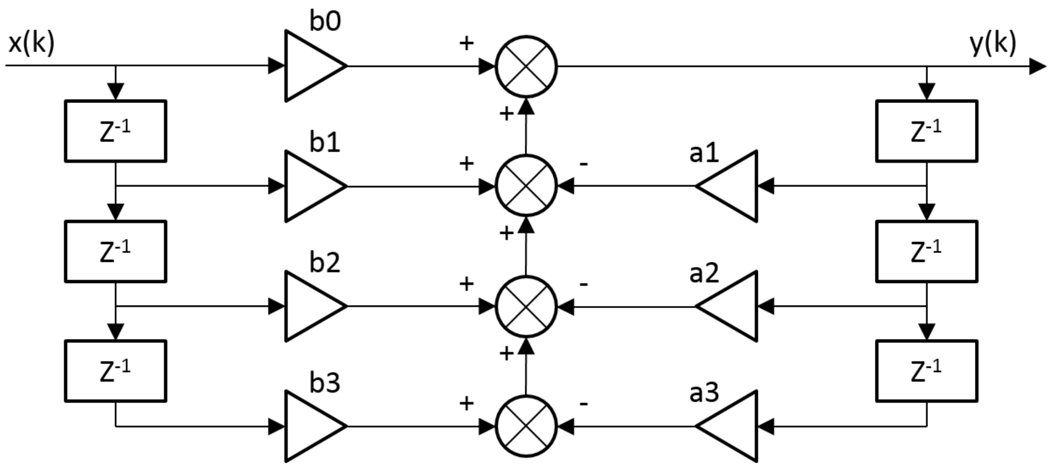

Figure 2-5. Direct-form 1 third-order IIR filter

The coefficients of the filter depicted in Figure 2-5 can be designed to meet the requirements for the third-order low-pass filter (LPF) or high-pass filter (HPF). The coefficient quantization error can be neglected in the case of a third-order filter due to a finite precision arithmetic. A higher-order LPF or HPF can be obtained by connecting a number of third-order filters in series. The number of connections gives the order of the resulting filter.

Define the filter coefficients before calling this function. As some coefficients can be greater than 1 (and lesser than 4), the coefficients are scaled down (divided) by 4.0 for the fractional version of the algorithm. For a faster calculation, the A coefficients are sign-inverted. The function returns the filtered value of the input in the step k, and stores the input and output values in the step k into the filter buffer.

2.3.1 Available versions

This function is available in the following versions:

- Fractional output - the output is the fractional portion of the result; the result is within the range <-1 ; 1).

The available versions of the GDFLIB_FilterIIR3Init function are shown in the following table:

Table 2-5. Init function versions

Function name	Parameters	Result type	Description
GDFLIB_FilterIIR3Init_F16	GDFLIB_FILTER_IIR3_T_F32 *	void	Filter initialization (reset) function. The parameters' structure is pointed to by a pointer.

The available versions of the GDFLIB_FilterIIR3 function are shown in the following table:

Table 2-6. Function versions

Function name	Input type	Parameters	Result type	Description
GDFLIB_FilterIIR3_F16	frac16_t	GDFLIB_FILTER_IIR3_T_F32 *	frac16_t	Input argument is a 16-bit fractional value of the input signal to be filtered within the range <-1 ; 1). The parameters' structure is pointed to by a pointer. The function returns a 16-bit fractional value within the range <-1 ; 1).

2.3.2 GDFLIB_FILTER_IIR3_T_F32

Variable name	Input type	Description
sFltCoeff	GDFLIB_FILTER_IIR3_COEFF_T_F32 *	Substructure containing filter coefficients.
f32FltBfrY[3]	frac32_t	Internal buffer of y-history. Controlled by the algorithm.
f16FltBfrX[3]	frac16_t	Internal buffer of x-history. Controlled by the algorithm.

2.3.3 GDFLIB_FILTER_IIR3_COEFF_T_F32

Variable name	Type	Description
f32B0	frac32_t	B0 coefficient of the IIR3 filter. Set by the user, and must be divided by 4.
f32B1	frac32_t	B1 coefficient of the IIR3 filter. Set by the user, and must be divided by 4.
f32B2	frac32_t	B2 coefficient of the IIR3 filter. Set by the user, and must be divided by 4.
f32B3	frac32_t	B3 coefficient of the IIR3 filter. Set by the user, and must be divided by 4.
f32A1	frac32_t	A1 (sign-inverted) coefficient of the IIR3 filter. Set by the user. Must be divided by -4.
f32A2	frac32_t	A2 (sign-inverted) coefficient of the IIR3 filter. Set by the user. Must be divided by -4.
f32A3	frac32_t	A3 (sign-inverted) coefficient of the IIR3 filter. Set by the user. Must be divided by -4.

2.3.4 Declaration

The available GDFLIB_FilterIIR3Init functions have the following declarations:

```
void GDFLIB_FilterIIR3Init_F16(GDFLIB_FILTER_IIR3_T_F32 *psParam)
```

The available GDFLIB_FilterIIR3 functions have the following declarations:

```
frac16_t GDFLIB_FilterIIR3_F16(frac16_t f16InX, GDFLIB_FILTER_IIR3_T_F32 *psParam)
```

2.3.5 Calculation of filter coefficients

There are plenty of methods for calculating the coefficients. The following example shows the use of Matlab to set up a high-pass filter with the 10000 Hz sampling frequency and 200 Hz stop frequency with 60 dB attenuation. The ripple is 3 dB at the cut-off frequency of 2000 Hz.

```
% sampling frequency 10000 Hz, high pass
Ts = 1 / 10000

% cut-off frequency 2 KHz
Fc = 2000

% attenuation 60 dB
Rs = 60

% stop frequency 200 Hz
Fs = 200

% max. passband ripple 3 dB
Rp = 3

% checking order of the filter
n = buttord(2 * Ts * Fc, 2 * Ts * Fs, Rp, Rs)
% n = 3, i.e. the filter is achievable with the 3rd order

% getting the filter coefficients
[b, a] = butter(n, 2* Ts * Fc, 'high')

% the coefs are:
% b0 = 0.256915601248463, b1 = -0.770746803745390, b2 = 0.770746803745390,
% b3 = -0.256915601248463
% a0 = 1.0000, a1 = -0.577240524806303, a2 = 0.421787048689562, a3 = -0.056297236491843
```

The filter response is shown in Figure 2-6.

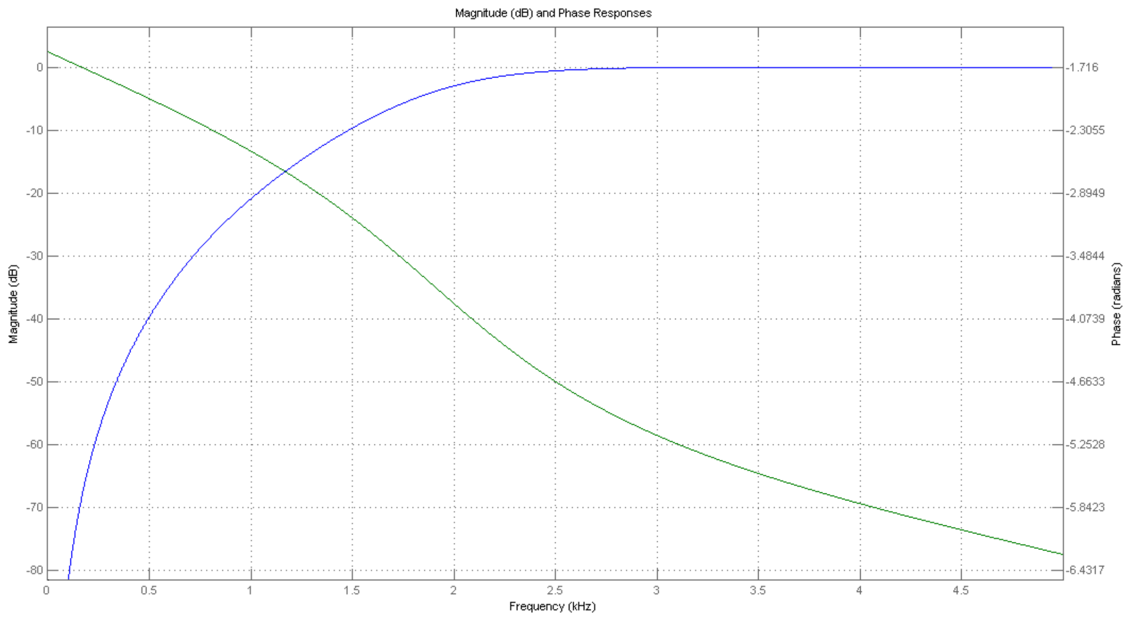

Figure 2-6. Filter response

2.3.6 Function use

The use of the GDFLIB_FilterIIR3Init and GDFLIB_FilterIIR3 functions is shown in the following example. The filter uses the above-calculated coefficients:

```
#include "gdflib.h"

static frac16_t f16Result;
static frac16_t f16InX;
static GDFLIB_FILTER_IIR3_T_F32 sFilterParam;

void Isr(void);

void main(void)
{
   sFilterParam.sFltCoeff.f32B0 = FRAC32(0.256915601248463 / 4.0);
   sFilterParam.sFltCoeff.f32B1 = FRAC32(-0.770746803745390 / 4.0);
   sFilterParam.sFltCoeff.f32B2 = FRAC32(0.770746803745390 / 4.0);
   sFilterParam.sFltCoeff.f32B3 = FRAC32(-0.256915601248463 / 4.0);
   sFilterParam.sFltCoeff.f32A1 = FRAC32(-0.577240524806303 / -4.0);
   sFilterParam.sFltCoeff.f32A2 = FRAC32(0.421787048689562 / -4.0);
   sFilterParam.sFltCoeff.f32A3 = FRAC32(-0.056297236491843 / -4.0);

   GDFLIB_FilterIIR3Init_F16(&sFilterParam);

   f16InX = FRAC16(0.1);
}

/* periodically called function */
void Isr(void)
{
```

```
    f16Result = GDFLIB_FilterIIR3_F16(f16InX, &sFilterParam);
}
```

2.4 GDFLIB_FilterIIR4

This function calculates the fourth-order direct-form 1 IIR filter.

For a proper use, it is recommended to initialize the algorithm by the GDFLIB_FilterIIR4Init function, before using the GDFLIB_FilterIIR4 function. The GDFLIB_FilterIIR4Init function initializes the buffer and coefficients of the fourth-order IIR filter.

The GDFLIB_FilterIIR4 function calculates the fourth-order infinite impulse response (IIR) filter. The IIR filters are also called recursive filters, because both the input and the previously calculated output values are used for calculation. This form of feedback enables the transfer of energy from the output to the input, which leads to an infinitely long impulse response (IIR). A general form of the IIR filter (expressed as a transfer function in the Z-domain) is described as follows:

$$H(z) = \frac{B(z)}{A(z)} = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2} + \ldots + b_N z^{-N}}{1 + a_1 z^{-1} + a_2 z^{-2} + \ldots + a_N z^{-N}}$$

Equation 10.

where N denotes the filter order. The fourth-order IIR filter in the Z-domain is expressed as follows:

$$H(z) = \frac{B(z)}{A(z)} = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2} + b_3 z^{-3} + b_4 z^{-4}}{1 + a_1 z^{-1} + a_2 z^{-2} + a_3 z^{-3} + a_4 z^{-4}}$$

Equation 11.

which is transformed into a time-domain difference equation as follows:

$$y(k) = b_0 x(k) + b_1 x(k-1) + b_2 x(k-2) + b_3 x(k-3) + b_4 x(k-4) - a_1 y(k-1) - a_2 y(k-2) - a_3 y(k-3) - a_4 y(k-4)$$

Equation 12.

The filter difference equation is implemented directly in the digital signal controller, as given in Equation 12 on page 32; this equation represents a direct-form 1 fourth-order IIR filter, as shown in Figure 2-7.

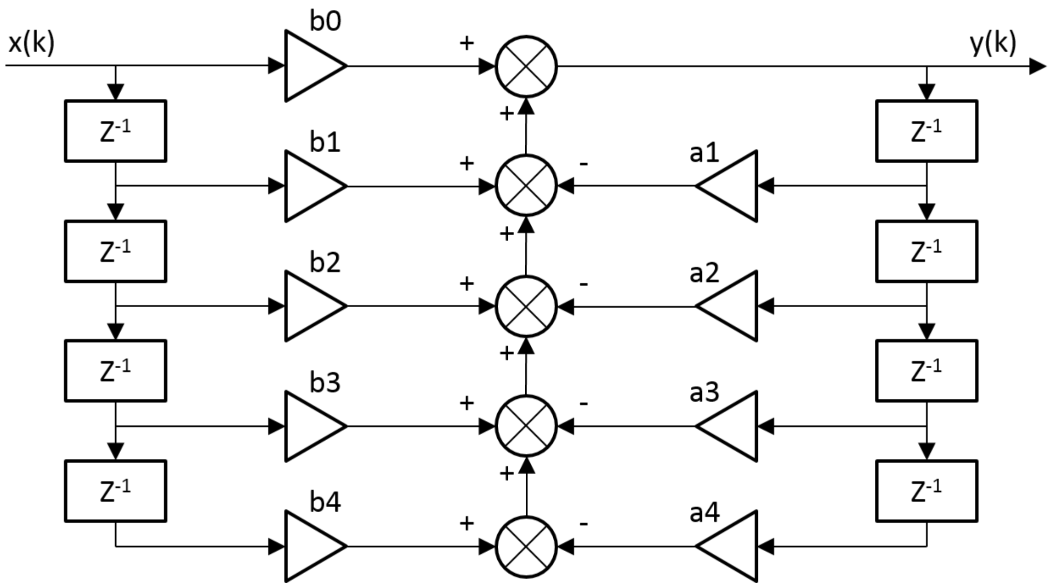

Figure 2-7. Direct-form 1 fourth-order IIR filter

The coefficients of the filter shown in Figure 2-7 can be designed to meet the requirements for the fourth-order low-pass filter (LPF), high-pass filter (HPF), band-pass filter (BPF), or band-stop filter (BSF). The coefficient quantization error can be ignored in the case of a fourth-order filter due to a finite precision arithmetic. A higher-order LPF or HPF can be obtained by connecting a number of fourth-order filters in series. The number of connections gives the order of the resulting filter.

Define the filter coefficients before calling this function. As some coefficients can be greater than 1 (and lesser than 8), the coefficients are scaled down (divided) by 8.0 for the fractional version of the algorithm. For a faster calculation, the A coefficients are sign-inverted. The function returns the filtered value of the input in step k, and stores the input and output values in the step k into the filter buffer.

2.4.1 Available versions

This function is available in the following versions:

- Fractional output - the output is the fractional portion of the result; the result is within the range <-1 ; 1).

The available versions of the GDFLIB_FilterIIR4Init function are shown in the following table:

Table 2-7. Init function versions

Function name	Parameters	Result type	Description
GDFLIB_FilterIIR4Init_F16	GDFLIB_FILTER_IIR4_T_F32 *	void	Filter initialization (reset) function. The parameters' structure is pointed to by a pointer.

The available versions of the GDFLIB_FilterIIR4 function are shown in the following table:

Table 2-8. Function versions

Function name	Input type	Parameters	Result type	Description
GDFLIB_FilterIIR4_F16	frac16_t	GDFLIB_FILTER_IIR4_T_F32 *	frac16_t	Input argument is a 16-bit fractional value of the input signal to be filtered within the range <-1 ; 1). The parameters' structure is pointed to by a pointer. The function returns a 16-bit fractional value within the range <-1 ; 1).

2.4.2 GDFLIB_FILTER_IIR4_T_F32

Variable name	Input type	Description
sFltCoeff	GDFLIB_FILTER_IIR4_COEFF_T_F32 *	Substructure containing filter coefficients.
f32FltBfrY[4]	frac32_t	Internal buffer of y-history. Controlled by the algorithm.
f16FltBfrX[4]	frac16_t	Internal buffer of x-history. Controlled by the algorithm.

2.4.3 GDFLIB_FILTER_IIR4_COEFF_T_F32

Variable name	Type	Description
f32B0	frac32_t	B0 coefficient of the IIR4 filter. Set by the user, and must be divided by 8.
f32B1	frac32_t	B1 coefficient of the IIR4 filter. Set by the user, and must be divided by 8.
f32B2	frac32_t	B2 coefficient of the IIR4 filter. Set by the user, and must be divided by 8.

Table continues on the next page...

Variable name	Type	Description
f32B3	frac32_t	B3 coefficient of the IIR4 filter. Set by the user, and must be divided by 8.
f32B4	frac32_t	B4 coefficient of the IIR4 filter. Set by the user, and must be divided by 8.
f32A1	frac32_t	A1 (sign-inverted) coefficient of the IIR4 filter. Set by the user, and must be divided by -8.
f32A2	frac32_t	A2 (sign-inverted) coefficient of the IIR4 filter. Set by the user, and must be divided by -8.
f32A3	frac32_t	A3 (sign-inverted) coefficient of the IIR4 filter. Set by the user, and must be divided by -8.
f32A4	frac32_t	A4 (sign-inverted) coefficient of the IIR4 filter. Set by the user, and must be divided by -8.

2.4.4 Declaration

The available GDFLIB_FilterIIR4Init functions have the following declarations:

```
void GDFLIB_FilterIIR4Init_F16(GDFLIB_FILTER_IIR4_T_F32 *psParam)
```

The available GDFLIB_FilterIIR4 functions have the following declarations:

```
frac16_t GDFLIB_FilterIIR4_F16(frac16_t f16InX, GDFLIB_FILTER_IIR4_T_F32 *psParam)
```

2.4.5 Calculation of filter coefficients

There are plenty of methods for the coefficients calculation. The following example shows the use of Matlab to set up a band-pass filter with the 10000 Hz sampling frequency, 1000 Hz pass frequency, and 250 Hz bandwidth. The maximum passband ripple is 3 dB, and the attenuation is 20 dB.

```
% sampling frequency 10000 Hz, band pass
Ts = 1 / 10000

% center pass frequency 2000 Hz
Fc = 2000

% attenuation 20 dB
Rs = 20

% bandwidth 250 Hz
Fbw = 250

% max. passband ripple 3 dB
Rp = 3

% checking order of the filter
n = buttord(2 * Ts * [Fc - Fbw /2 Fc + Fbw / 2], 2 * Ts * [Fc - Fbw Fc + Fbw], Rp, Rs)
% n = 4, i.e. the filter is achievable with the 4th order

% getting the filter coefficients
[b, a] = butter(n / 2, 2 * Ts * [Fc - Fbw /2 Fc + Fbw / 2])

% the coefs are:
% b0 = 0.005542717210281, b1 = 0, b2 = -0.011085434420561, b3 = 0, b4 = 0.005542717210281
```

```
% a0 = 1.0000, a1 = -1.171272075750262, a2 = 2.122554479822350, a3 = -1.047780658093187,
% a4 = 0.800802646665706
```

The filter response is shown in Figure 2-8.

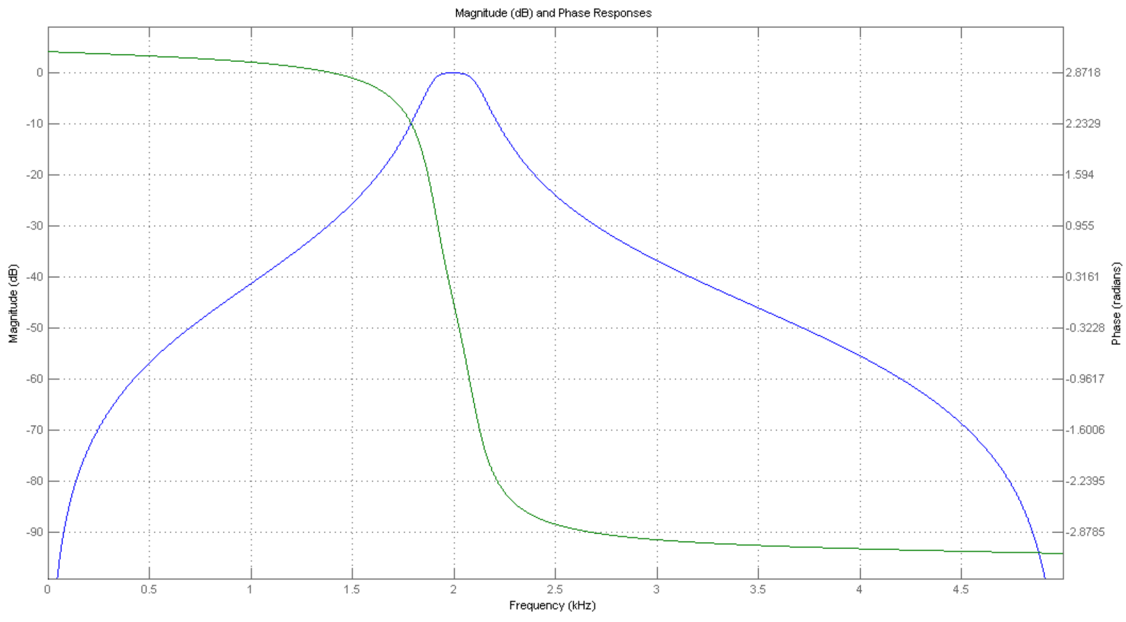

Figure 2-8. Filter response

2.4.6 Function use

The use of the GDFLIB_FilterIIR4Init and GDFLIB_FilterIIR4 functions is shown in the following example. The filter uses the above-calculated coefficients:

```
#include "gdflib.h"

static frac16_t f16Result;
static frac16_t f16InX;
static GDFLIB_FILTER_IIR4_T_F32 sFilterParam;

void Isr(void);

void main(void)
{
   sFilterParam.sFltCoeff.f32B0 = FRAC32(0.005542717210281 / 8.0);
   sFilterParam.sFltCoeff.f32B1 = FRAC32(0.0 / 8.0);
   sFilterParam.sFltCoeff.f32B2 = FRAC32(-0.011085434420561 / 8.0);
   sFilterParam.sFltCoeff.f32B3 = FRAC32(0.0 / 8.0);
   sFilterParam.sFltCoeff.f32B4 = FRAC32(0.005542717210281 / 8.0);
   sFilterParam.sFltCoeff.f32A1 = FRAC32(-1.171272075750262 / -8.0);
   sFilterParam.sFltCoeff.f32A2 = FRAC32(2.122554479822350 / -8.0);
   sFilterParam.sFltCoeff.f32A3 = FRAC32(-1.047780658093187 / -8.0);
   sFilterParam.sFltCoeff.f32A4 = FRAC32(0.800802646665706 / -8.0);

   GDFLIB_FilterIIR4Init_F16(&sFilterParam);
```

```
    f16InX = FRAC16(0.1);
}

/* periodically called function */
void Isr(void)
{
     f16Result = GDFLIB_FilterIIR4_F16(f16InX, &sFilterParam);
}
```

2.5 GDFLIB_FilterMA

The GDFLIB_FilterMA function calculates a recursive form of a moving average filter. For a proper use, it is recommended that the algorithm is initialized by the GDFLIB_FilterMAInit function, before using the GDFLIB_FilterMA function.

The filter calculation consists of the following equations:

$$acc(k) = acc(k-1) + x(k)$$

Equation 13.

$$y(k) = \frac{acc(k)}{n_p}$$

Equation 14.

$$acc(k) \leftarrow acc(k) - y(k)$$

Equation 15.

where:

- x(k) is the actual value of the input signal
- acc(k) is the internal filter accumulator
- y(k) is the actual filter output
- n_p is the number of points in the filter window

The size of the filter window (number of filtered points) must be defined before calling this function, and must be equal to or greater than 1.

The function returns the filtered value of the input at step k, and stores the difference between the filter accumulator and the output at step k into the filter accumulator.

2.5.1 Available versions

This function is available in the following versions:

- Fractional output - the output is the fractional portion of the result; the result is within the range <-1 ; 1). The parameters use the accumulator types.

The available versions of the GDFLIB_FilterMAInit function are shown in the following table:

Table 2-9. Function versions

Function name	Input type	Parameters	Result type	Description
GDFLIB_FilterMAInit_F16	frac16_t	GDFLIB_FILTER_MA_T_A32 *	void	Input argument is a 16-bit fractional value that represents the initial value of the filter at the current step. The input is within the range <-1 ; 1). The parameters' structure is pointed to by a pointer.

The available versions of the GDFLIB_FilterMA function are shown in the following table:

Table 2-10. Function versions

Function name	Input type		Result type	Description
	Value	Parameter		
GDFLIB_FilterMA_F16	frac16_t	GDFLIB_FILTER_MA_T_A32 *	frac16_t	Input argument is a 16-bit fractional value of the input signal to be filtered within the range <-1 ; 1). The parameters' structure is pointed to by a pointer. The function returns a 16-bit fractional value within the range <-1 ; 1).

2.5.2 GDFLIB_FILTER_MA_T_A32

Variable name	Input type	Description
a32Acc	acc32_t	Filter accumulator. The parameter is a 32-bit accumulator type within the range <-65536.0 ; 65536.0). Controlled by the algorithm.
u16Sh	uint16_t	Number of samples for averaging filtered points (size of the window) defined as a number of shifts: $n_p = 2^{u16Sh}$ $u16Sh = \log_2 n_p$ The parameter is a 16-bit unsigned integer type within the range <0 ; 15>. Set by the user.

2.5.3 Declaration

The available GDFLIB_FilterMAInit functions have the following declarations:

```
void GDFLIB_FilterMAInit_F16(frac16_t f16InitVal, GDFLIB_FILTER_MA_T_A32 *psParam)
```

The available GDFLIB_FilterMA functions have the following declarations:

```
frac16_t GDFLIB_FilterMA_F16(frac16_t f16InX, GDFLIB_FILTER_MA_T_A32 *psParam)
```

2.5.4 Function use

The use of GDFLIB_FilterMAInit and GDFLIB_FilterMA functions is shown in the following example:

```
#include "gdflib.h"

static frac16_t f16Result;
static frac16_t f16InitVal, f16InX;
static GDFLIB_FILTER_MA_T_A32 sFilterParam;

void Isr(void);

void main(void)
{
   f16InitVal = FRAC16(0.0);                /* f16InitVal = 0.0 */

   /* Filter window = 2 ^ 2 = 4 points */
   sFilterParam.u16Sh = 2;

   GDFLIB_FilterMAInit_F16(f16InitVal, &sFilterParam);

   f16InX =  FRAC16(0.8);
}

/* periodically called function */
void Isr(void)
{
   f16Result = GDFLIB_FilterMA_F16(f16InX, &sFilterParam);
}
```

Appendix A

A.1 bool_t

The bool_t type is a logical 16-bit type. It is able to store the boolean variables with two states: TRUE (1) or FALSE (0). Its definition is as follows:

```
typedef unsigned short bool_t;
```

The following figure shows the way in which the data is stored by this type:

Table A-1. Data storage

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Value	Unused															Logical
TRUE	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
	0				0				0				1			
FALSE	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	0				0				0				0			

To store a logical value as bool_t, use the FALSE or TRUE macros.

A.2 uint8_t

The uint8_t type is an unsigned 8-bit integer type. It is able to store the variables within the range <0 ; 255>. Its definition is as follows:

```
typedef unsigned char int8_t;
```

The following figure shows the way in which the data is stored by this type:

Table A-2. Data storage

Table continues on the next page...

Table A-2. Data storage (continued)

	7	6	5	4	3	2	1	0
Value	Integer							
255	1	1	1	1	1	1	1	1
	F				F			
11	0	0	0	0	1	0	1	1
	0				B			
124	0	1	1	1	1	1	0	0
	7				C			
159	1	0	0	1	1	1	1	1
	9				F			

A.3 uint16_t

The uint16_t type is an unsigned 16-bit integer type. It is able to store the variables within the range <0 ; 65535>. Its definition is as follows:

```
typedef unsigned short uint16_t;
```

The following figure shows the way in which the data is stored by this type:

Table A-3. Data storage

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Value	Integer															
65535	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	F				F				F				F			
5	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1
	0				0				0				5			
15518	0	0	1	1	1	1	0	0	1	0	0	1	1	1	1	0
	3				C				9				E			
40768	1	0	0	1	1	1	1	1	0	1	0	0	0	0	0	0
	9				F				4				0			

A.4 uint32_t

The uint32_t type is an unsigned 32-bit integer type. It is able to store the variables within the range <0 ; 4294967295>. Its definition is as follows:

```
typedef unsigned long uint32_t;
```

The following figure shows the way in which the data is stored by this type:

Table A-4. Data storage

	31	24	23	16	15	8	7	0
Value	Integer							
4294967295	F	F	F	F	F	F	F	F
2147483648	8	0	0	0	0	0	0	0
55977296	0	3	5	6	2	5	5	0
3451051828	C	D	B	2	D	F	3	4

A.5 int8_t

The int8_t type is a signed 8-bit integer type. It is able to store the variables within the range <-128 ; 127>. Its definition is as follows:

```
typedef char int8_t;
```

The following figure shows the way in which the data is stored by this type:

Table A-5. Data storage

	7	6	5	4	3	2	1	0
Value	Sign	Integer						
127	0	1	1	1	1	1	1	1
	7				F			
-128	1	0	0	0	0	0	0	0
	8				0			
60	0	0	1	1	1	1	0	0
	3				C			
-97	1	0	0	1	1	1	1	1
	9				F			

A.6 int16_t

The int16_t type is a signed 16-bit integer type. It is able to store the variables within the range <-32768 ; 32767>. Its definition is as follows:

```
typedef short int16_t;
```

The following figure shows the way in which the data is stored by this type:

Table A-6. Data storage

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Value	Sign	Integer														
32767	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	7				F				F				F			
-32768	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	8				0				0				0			
15518	0	0	1	1	1	1	0	0	1	0	0	1	1	1	1	0
	3				C				9				E			
-24768	1	0	0	1	1	1	1	1	0	1	0	0	0	0	0	0
	9				F				4				0			

A.7 int32_t

The int32_t type is a signed 32-bit integer type. It is able to store the variables within the range <-2147483648 ; 2147483647>. Its definition is as follows:

```
typedef long int32_t;
```

The following figure shows the way in which the data is stored by this type:

Table A-7. Data storage

	31	24	23	16	15	8	7	0
Value	S	Integer						
2147483647	7	F	F	F	F	F	F	F
-2147483648	8	0	0	0	0	0	0	0
55977296	0	3	5	6	2	5	5	0
-843915468	C	D	B	2	D	F	3	4

A.8 frac8_t

The frac8_t type is a signed 8-bit fractional type. It is able to store the variables within the range <-1 ; 1). Its definition is as follows:

```
typedef char frac8_t;
```

The following figure shows the way in which the data is stored by this type:

Table A-8. Data storage

	7	6	5	4	3	2	1	0
Value	Sign	Fractional						
0.99219	0	1	1	1	1	1	1	1
	7				F			
-1.0	1	0	0	0	0	0	0	0
	8				0			
0.46875	0	0	1	1	1	1	0	0
	3				C			
-0.75781	1	0	0	1	1	1	1	1
	9				F			

To store a real number as frac8_t, use the FRAC8 macro.

A.9 frac16_t

The frac16_t type is a signed 16-bit fractional type. It is able to store the variables within the range <-1 ; 1). Its definition is as follows:

```
typedef short frac16_t;
```

The following figure shows the way in which the data is stored by this type:

Table A-9. Data storage

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Value	Sign	Fractional														
0.99997	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	7				F				F				F			
-1.0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table continues on the next page...

Table A-9. Data storage (continued)

	8				0				0				0			
0.47357	0	0	1	1	1	1	0	0	1	0	0	1	1	1	1	0
	3				C				9				E			
-0.75586	1	0	0	1	1	1	1	1	0	1	0	0	0	0	0	0
	9				F				4				0			

To store a real number as frac16_t, use the FRAC16 macro.

A.10 frac32_t

The frac32_t type is a signed 32-bit fractional type. It is able to store the variables within the range <-1 ; 1). Its definition is as follows:

```
typedef long frac32_t;
```

The following figure shows the way in which the data is stored by this type:

Table A-10. Data storage

	31	24	23	16	15	8	7	0
Value	S	Fractional						
0.9999999995	7	F	F	F	F	F	F	F
-1.0	8	0	0	0	0	0	0	0
0.02606645970	0	3	5	6	2	5	5	0
-0.3929787632	C	D	B	2	D	F	3	4

To store a real number as frac32_t, use the FRAC32 macro.

A.11 acc16_t

The acc16_t type is a signed 16-bit fractional type. It is able to store the variables within the range <-256 ; 256). Its definition is as follows:

```
typedef short acc16_t;
```

The following figure shows the way in which the data is stored by this type:

Table A-11. Data storage

Value	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	Sign	Integer								Fractional						
255.9921875	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	7				F				F				F			
-256.0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	8				0				0				0			
1.0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
	0				0				8				0			
-1.0	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0
	F				F				8				0			
13.7890625	0	0	0	0	0	1	1	0	1	1	1	0	0	1	0	1
	0				6				E				5			
-89.71875	1	1	0	1	0	0	1	1	0	0	1	0	0	1	0	0
	D				3				2				4			

To store a real number as acc16_t, use the ACC16 macro.

A.12 acc32_t

The acc32_t type is a signed 32-bit accumulator type. It is able to store the variables within the range <-65536 ; 65536). Its definition is as follows:

```
typedef long acc32_t;
```

The following figure shows the way in which the data is stored by this type:

Table A-12. Data storage

Value	31	24	23	16	15	8	7	0
	S	Integer			Fractional			
65535.999969	7	F	F	F	F	F	F	F
-65536.0	8	0	0	0	0	0	0	0
1.0	0	0	0	0	8	0	0	0
-1.0	F	F	F	F	8	0	0	0
23.789734	0	0	0	B	E	5	1	6
-1171.306793	F	D	B	6	5	8	B	C

To store a real number as acc32_t, use the ACC32 macro.

A.13 FALSE

The FALSE macro serves to write a correct value standing for the logical FALSE value of the bool_t type. Its definition is as follows:

```
#define FALSE    ((bool_t)0)
```

```
#include "mlib.h"

static bool_t bVal;

void main(void)
{
  bVal = FALSE;               /* bVal = FALSE */
}
```

A.14 TRUE

The TRUE macro serves to write a correct value standing for the logical TRUE value of the bool_t type. Its definition is as follows:

```
#define TRUE     ((bool_t)1)
```

```
#include "mlib.h"

static bool_t bVal;

void main(void)
{
  bVal = TRUE;               /* bVal = TRUE */
}
```

A.15 FRAC8

The FRAC8 macro serves to convert a real number to the frac8_t type. Its definition is as follows:

```
#define FRAC8(x) ((frac8_t)((x) < 0.9921875 ? ((x) >= -1 ? (x)*0x80 : 0x80) : 0x7F))
```

The input is multiplied by 128 ($=2^7$). The output is limited to the range <0x80 ; 0x7F>, which corresponds to $<-1.0 ; 1.0-2^{-7}>$.

```
#include "mlib.h"

static frac8_t f8Val;

void main(void)
{
  f8Val = FRAC8(0.187);              /* f8Val = 0.187 */
}
```

A.16 FRAC16

The FRAC16 macro serves to convert a real number to the frac16_t type. Its definition is as follows:

```
#define FRAC16(x) ((frac16_t)((x) < 0.999969482421875 ? ((x) >= -1 ? (x)*0x8000 : 0x8000) :
0x7FFF))
```

The input is multiplied by 32768 (=2^{15}). The output is limited to the range <0x8000 ; 0x7FFF>, which corresponds to <-1.0 ; 1.0-2^{-15}>.

```
#include "mlib.h"

static frac16_t f16Val;

void main(void)
{
  f16Val = FRAC16(0.736);              /* f16Val = 0.736 */
}
```

A.17 FRAC32

The FRAC32 macro serves to convert a real number to the frac32_t type. Its definition is as follows:

```
#define FRAC32(x) ((frac32_t)((x) < 1 ? ((x) >= -1 ? (x)*0x80000000 : 0x80000000) :
0x7FFFFFFF))
```

The input is multiplied by 2147483648 (=2^{31}). The output is limited to the range <0x80000000 ; 0x7FFFFFFF>, which corresponds to <-1.0 ; 1.0-2^{-31}>.

```
#include "mlib.h"

static frac32_t f32Val;

void main(void)
{
  f32Val = FRAC32(-0.1735667);              /* f32Val = -0.1735667 */
}
```

A.18 ACC16

The ACC16 macro serves to convert a real number to the acc16_t type. Its definition is as follows:

```
#define ACC16(x) ((acc16_t)((x) < 255.9921875 ? ((x) >= -256 ? (x)*0x80 : 0x8000) : 0x7FFF))
```

The input is multiplied by 128 (=2^7). The output is limited to the range <0x8000 ; 0x7FFF> that corresponds to <-256.0 ; 255.9921875>.

```
#include "mlib.h"

static acc16_t a16Val;

void main(void)
{
  a16Val = ACC16(19.45627);              /* a16Val = 19.45627 */
}
```

A.19 ACC32

The ACC32 macro serves to convert a real number to the acc32_t type. Its definition is as follows:

```
#define ACC32(x) ((acc32_t)((x) < 65535.999969482421875 ? ((x) >= -65536 ? (x)*0x8000 :
0x80000000) : 0x7FFFFFFF))
```

The input is multiplied by 32768 (=2^{15}). The output is limited to the range <0x80000000 ; 0x7FFFFFFF>, which corresponds to <-65536.0 ; 65536.0-2^{-15}>.

```
#include "mlib.h"

static acc32_t a32Val;

void main(void)
{
  a32Val = ACC32(-13.654437);              /* a32Val = -13.654437 */
}
```

How to Reach Us:

Home Page:
freescale.com

Web Support:
freescale.com/support

Document Number DSP56800EXGDFLIBUG
Revision 2, 10/2015

freescale™