GFLIB User's Guide

DSP56800EX

Document Number: DSP56800EXGFLIBUG
Rev. 2, 10/2015

Review Draft

Contents

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

Chapter 1
Library

Chapter 2
Algorithms in detail

Section number	Title	Page

Section number	Title	Page

Section number	Title	Page

Section number	Title	Page

Chapter 1
Library

1.1 Introduction

1.1.1 Overview

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

1.1.2 Data types

GFLIB 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

GFLIB 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

GFLIB 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 *gflib.lib* file.

The interfaces to the algorithms included in this library are combined into a single public interface include file, *gflib.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 GFLIB 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.

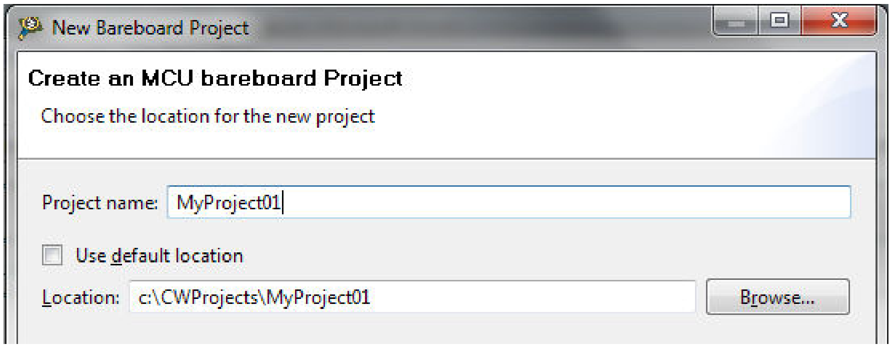

Figure 1-1. Project name and location

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

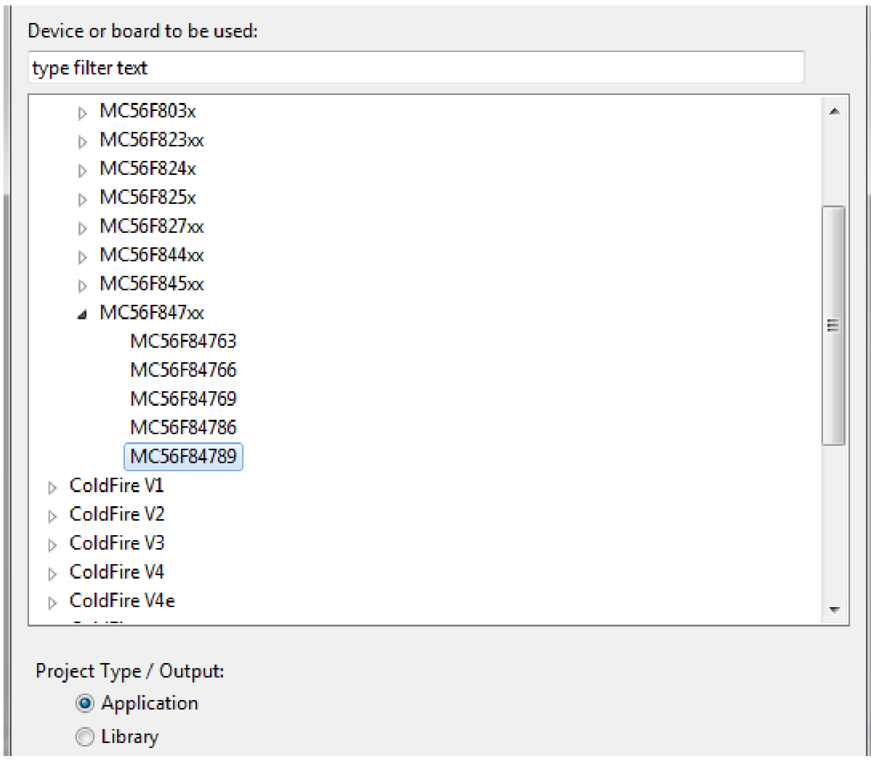

Figure 1-2. Processor selection

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.

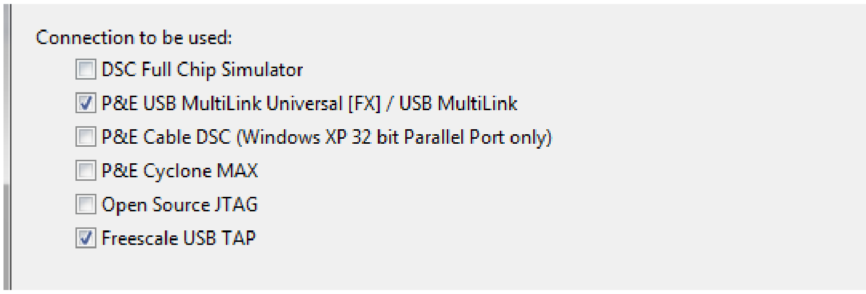

Figure 1-3. Connection selection

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

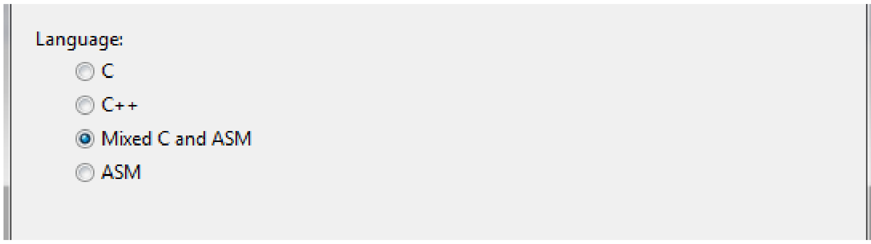

Figure 1-4. Language choice

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

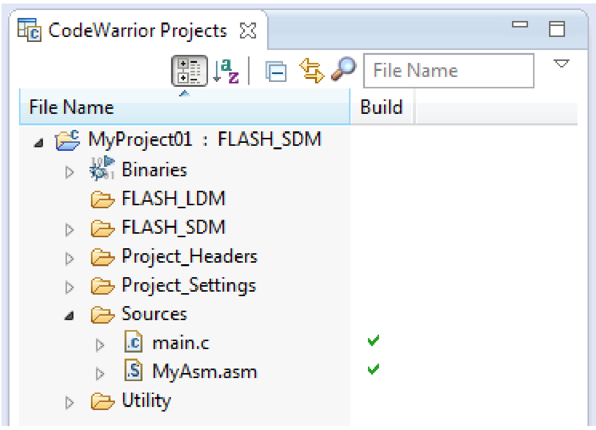

Figure 1-5. Project folder

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.

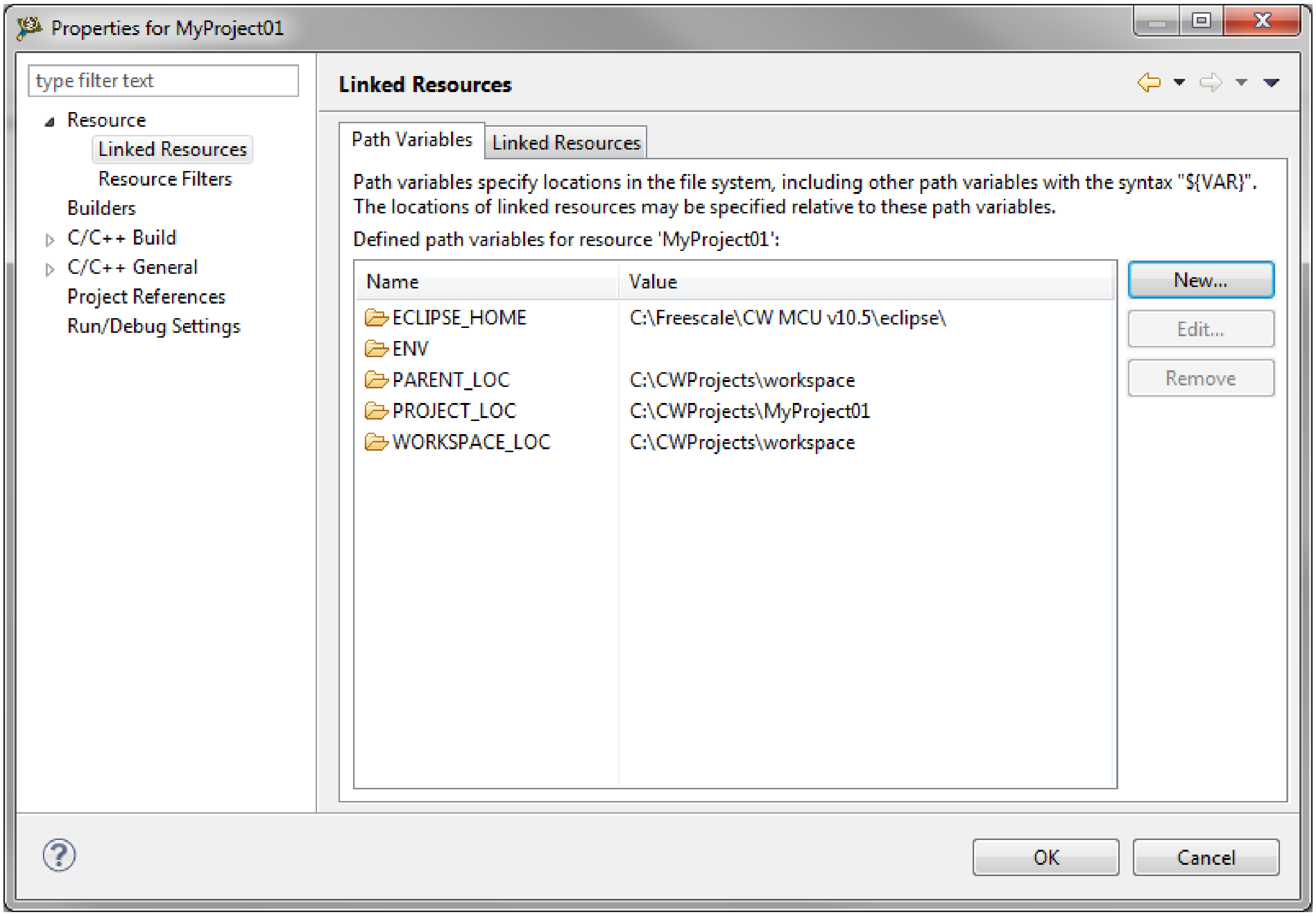

Figure 1-6. Project properties

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.

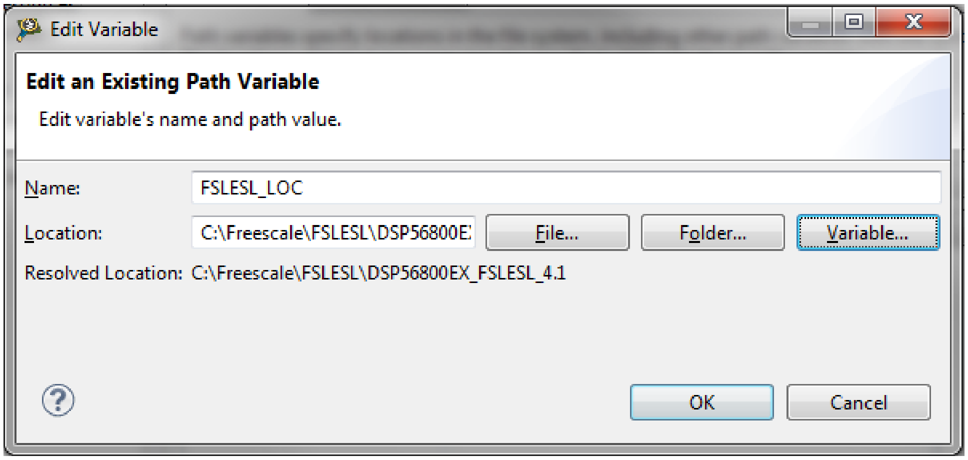

Figure 1-7. New variable

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

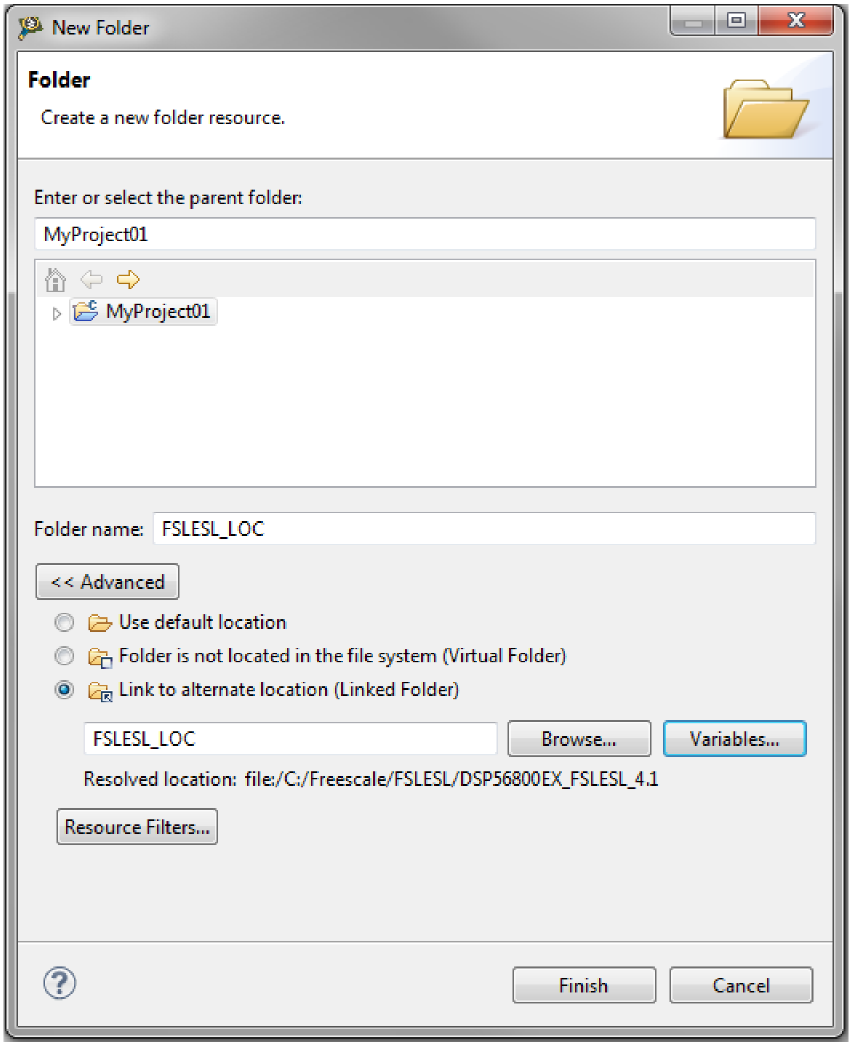

Figure 1-8. Folder link

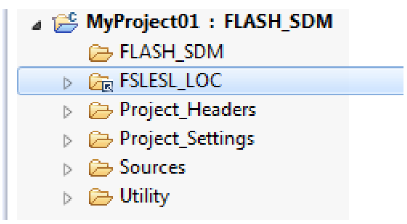

Figure 1-9. Projects libraries paths

1.2.4 Library path setup

GFLIB 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}\GFLIB\gflib_SDM.lib—for small data model projects
   - ${FSLESL_LOC}\GFLIB\gflib_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.

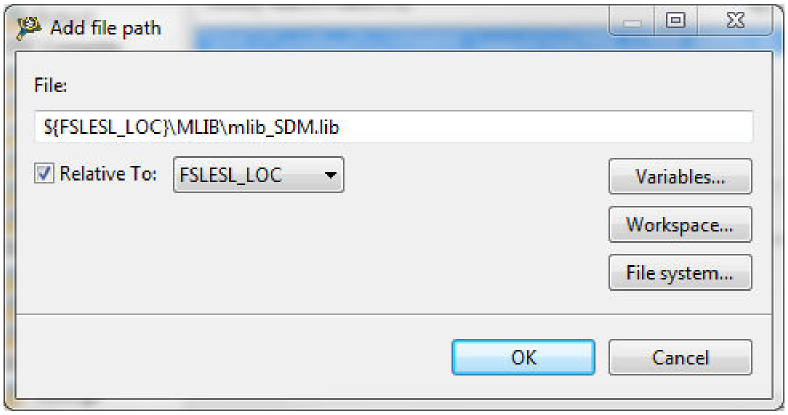

Figure 1-10. Library file inclusion

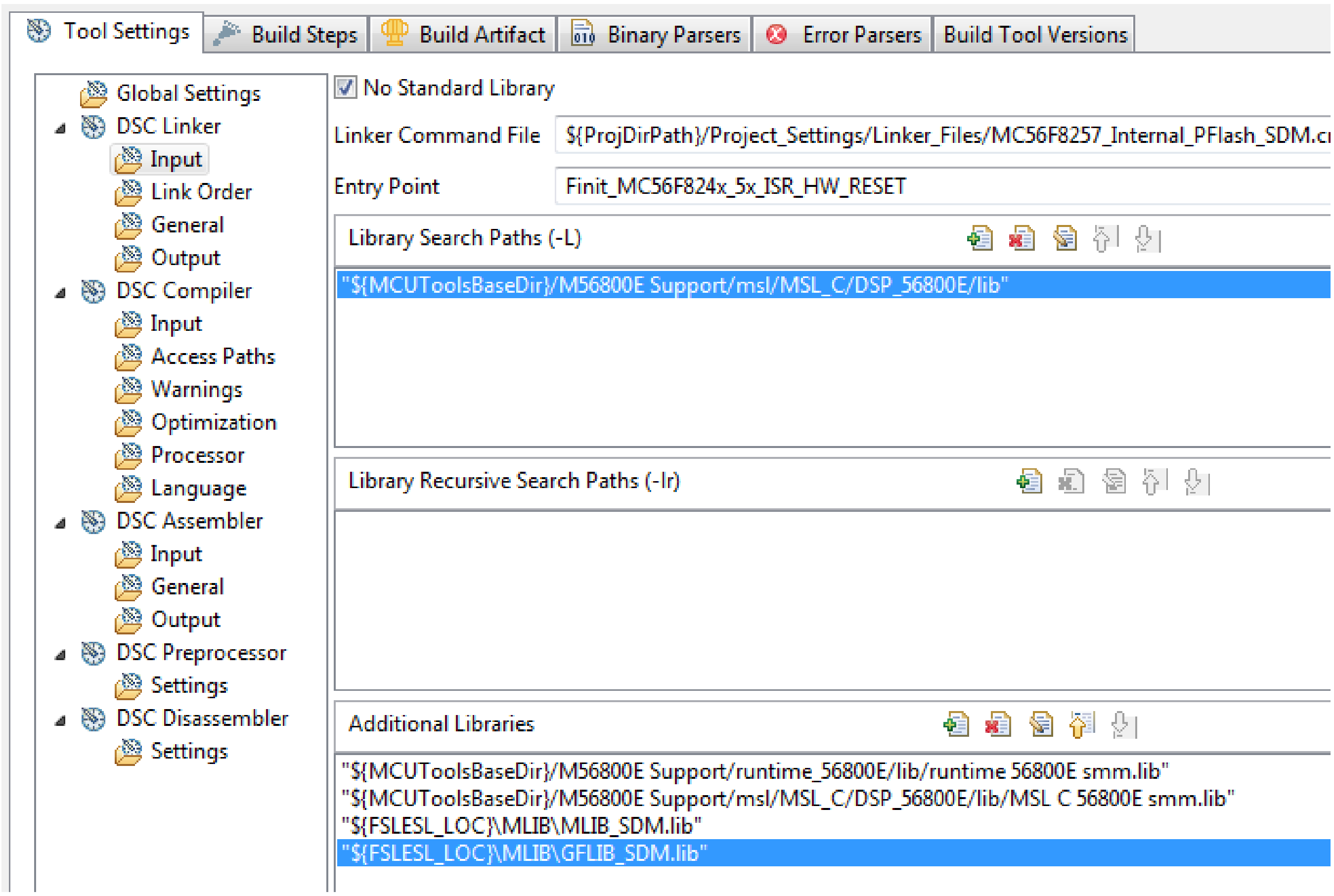

Figure 1-11. Linker setting

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}\GFLIB\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.

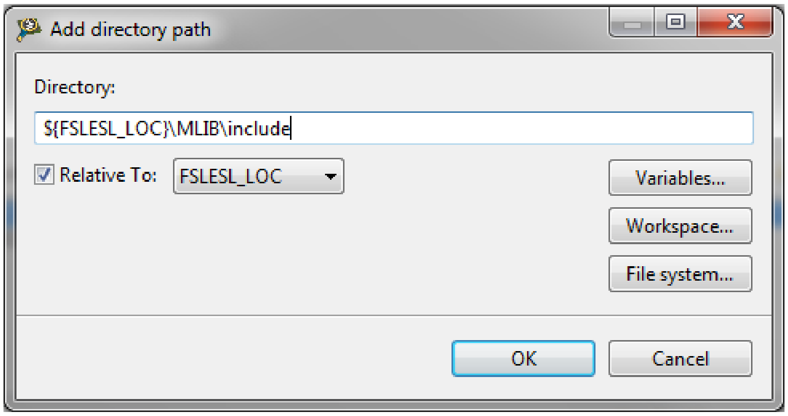

Figure 1-12. Library include path addition

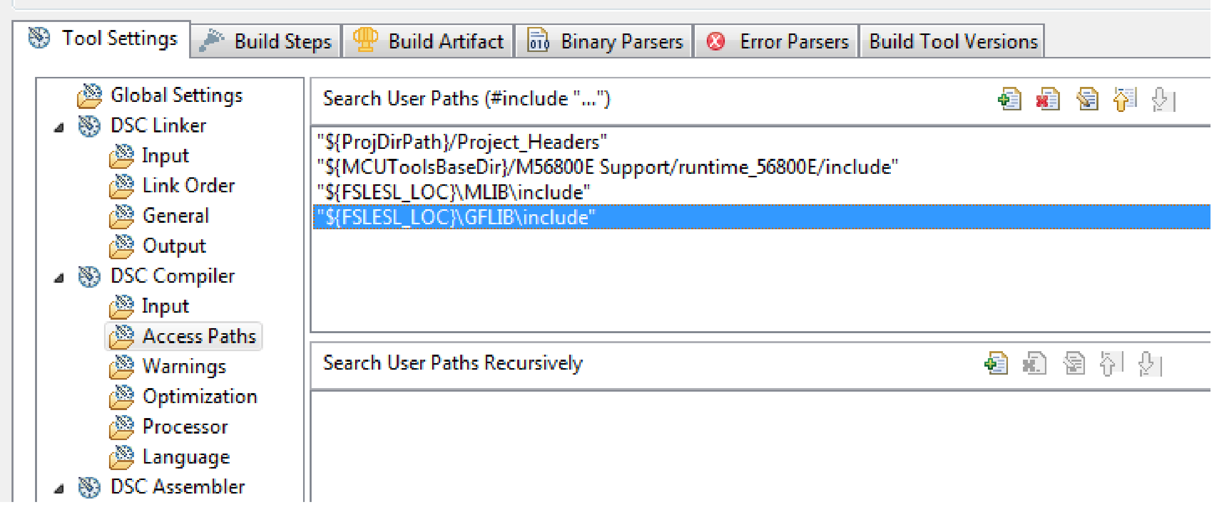

Figure 1-13. Compiler setting

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 "gflib.h"
```

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

Chapter 2
Algorithms in detail

2.1 GFLIB_Sin

The GFLIB_Sin function implements the polynomial approximation of the sine function. It provides a computational method for the calculation of a standard trigonometric sine function sin(x), using the 9^{th} order Taylor polynomial approximation. The Taylor polynomial approximation of a sine function is expressed as follows:

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!}$$

Equation 1.

$$\sin(x) = x(d_1 + x^2(d_3 + x^2(d_5 + x^2(d_7 + x^2 d_9))))$$

Equation 2.

where the constants are:

$d_1 = 1$

$d_3 = -\frac{1}{3!}$

$d_5 = \frac{1}{5!}$

$d_7 = -\frac{1}{7!}$

$d_9 = \frac{1}{9!}$

The fractional arithmetic is limited to the range <-1 ; 1), so the input argument can only be within this range. The input argument is the multiplier of π: sin(π · x), where the user passes the x argument. Example: if the input is -0.5, it corresponds to -0.5π.

The fractional function sin(π · x) is expressed using the 9^{th} order Taylor polynomial as follows:

$$\sin(\pi x) = x(c_1 + x^2(c_3 + x^2(c_5 + x^2(c_7 + x^2 c_9))))$$

Equation 3.

where:

$c_1 = d_1\pi^1 = \pi$

$c_3 = d_3\pi^3 = -\frac{\pi^3}{3!}$

$c_5 = d_5\pi^5 = \frac{\pi^5}{5!}$

$c_7 = d_7\pi^7 = -\frac{\pi^7}{7!}$

$c_9 = d_9\pi^9 = \frac{\pi^9}{9!}$

2.1.1 Available versions

The 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 result may saturate.

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

Table 2-1. Function versions

Function name	Input type	Result type	Description
GFLIB_Sin_F16	frac16_t	frac16_t	Calculation of the sin(π · x), where the input argument is a 16-bit fractional value normalized to the range <-1 ; 1) that represents an angle in radians within the range <-π; π). The output is a 16-bit fractional value within the range <-1 ; 1).

2.1.2 Declaration

The available GFLIB_Sin functions have the following declarations:

```
frac16_t GFLIB_Sin_F16(frac16_t f16Angle)
```

2.1.3 Function use

The use of the GFLIB_Sin function is shown in the following example:

```
#include "gflib.h"

static frac16_t f16Result;
static frac16_t f16Angle;

void main(void)
{
   f16Angle = FRAC16(0.333333);          /* f16Angle = 0.333333 [60°] */

   /* f16Result = sin(f16Angle); (π * f16Angle[rad]) = deg * (π / 180) */
   f16Result = GFLIB_Sin_F16(f16Angle);
}
```

2.2 GFLIB_Cos

The GFLIB_Cos function implements the polynomial approximation of the cosine function. This function computes the cos(x) using the ninth-order Taylor polynomial approximation of the sine function, and its equation is as follows:

$$\cos(x) = \sin\left[\frac{\pi}{2} + |x|\right]$$

Equation 4.

Because the fractional arithmetic is limited to the range <-1 ; 1), the input argument can only be within this range. The input argument is the multiplier of π: cos(π · x), where the user passes the x argument. For example, if the input is -0.5, it corresponds to -0.5π.

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 result may saturate.

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

Table 2-2. Function versions

Function name	Input type	Result type	Description
GFLIB_Cos_F16	frac16_t	frac16_t	Calculation of cos(π · x), where the input argument is a 16-bit fractional value, normalized to the range <-1 ; 1) that represents an angle in radians within the range <- π; π). The output is a 16-bit fractional value within the range <-1 ; 1).

2.2.2 Declaration

The available GFLIB_Cos functions have the following declarations:

```
frac16_t GFLIB_Cos_F16(frac16_t f16Angle)
```

2.2.3 Function use

The use of the GFLIB_Cos function is shown in the following example:

```
#include "gflib.h"

static frac16_t f16Result;
static frac16_t f16Angle;

void main(void)
{
   f16Angle = FRAC16(0.333333);          /* f16Angle = 0.333333 [60°] */

   /* f16Result = cos(f16Angle); (π * f16Angle[rad]) = deg * (π / 180) */
   f16Result = GFLIB_Cos_F16(f16Angle);
}
```

2.3 GFLIB_Tan

The GFLIB_Tan function provides a computational method for calculation of a standard trigonometric tangent function tan(x), using the piece-wise polynomial approximation. Function tan(x) takes an angle and returns the ratio of two sides of a right-angled triangle. The ratio is the length of the side opposite the angle divided by the length of the side adjacent to the angle.

$$\tan(x) = \frac{\sin(x)}{\cos(x)}$$

Equation 5.

Because both sin(x) and cos(x) are defined in interval <-π ; π>, the function tan(x) is equal to zero when sin(x)=0 and is equal to infinity when cos(x)=0. The graph of tan(x) is shown in the following figure:

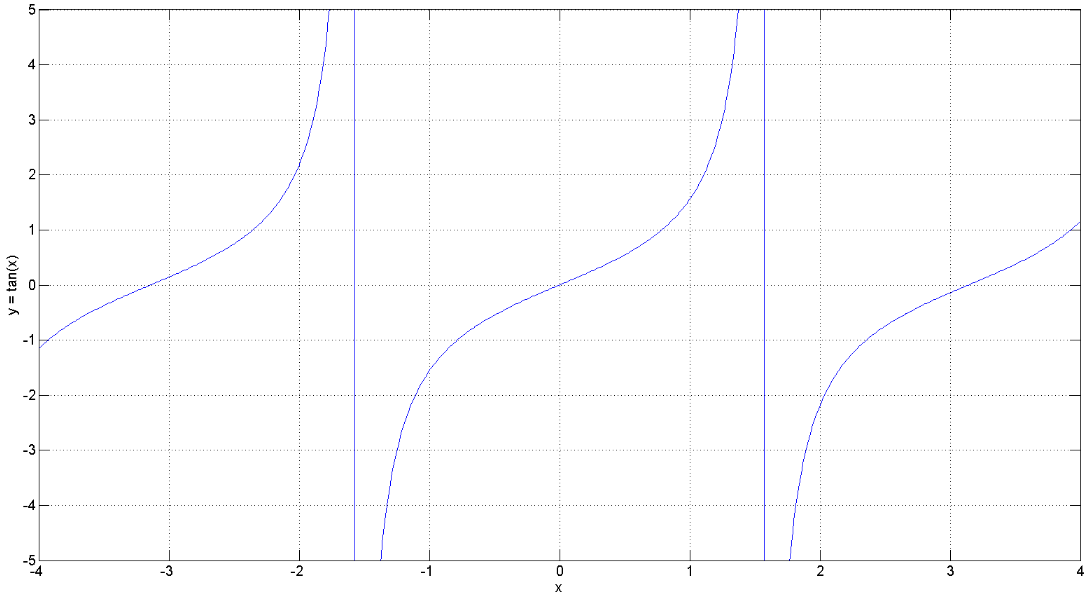

Figure 2-1. Course of the function GFLIB_Tan

The fractional arithmetic is limited to the range <-1 ; 1) so the input argument can only be within this range. The input argument is the multiplier of π: tan(π · x) where you pass the x argument. Example: if the input is -0.5, it corresponds to -0.5π. The output of the function is limited to the range <-1 ; 1) for the fractional arithmetic. For the points where the function is not defined, the output is fractional -1.

2.3.1 Available versions

The 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 result may saturate.

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

Table 2-3. Function versions

Function name	Input type	Result type	Description
GFLIB_Tan_F16	frac16_t	frac16_t	Calculation of the tan(π · x) where the input argument is a 16-bit fractional value normalized to the range <-1 ; 1) that represents an angle in radians within the range <-π ; π). The output is a 16-bit fractional value within the range <-1 ; 1).

2.3.2 Declaration

The available GFLIB_Tan functions have the following declarations:

```
frac16_t GFLIB_Tan_F16(frac16_t f16Angle)
```

2.3.3 Function use

The use of the GFLIB_Tan function is shown in the following example:

```
#include "gflib.h"

static frac16_t f16Result;
static frac16_t f16Angle;

void main(void)
{
   f16Angle = FRAC16(0.1);         /* f16Angle = 0.1 [18°] */

   /* f16Result = tan(f16Angle); (π * f16Angle[rad]) = deg * (π / 180) */
   f16Result = GFLIB_Tan_F16(f16Angle);
}
```

2.4 GFLIB_Asin

The GFLIB_Asin function provides a computational method for calculation of a standard inverse trigonometric arcsine function arcsin(x), using the piece-wise polynomial approximation. Function arcsin(x) takes the ratio of the length of the opposite side to the length of the hypotenuse and returns the angle.

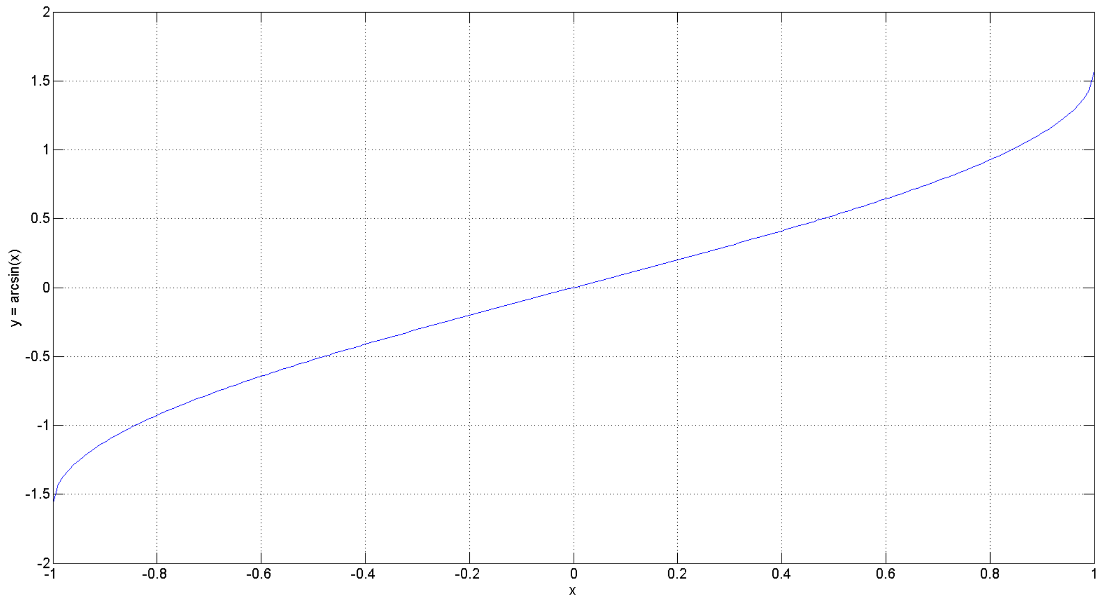

Figure 2-2. Course of the function GFLIB_Asin

The fractional arithmetic is limited by the range <-1;1) so the output can only be within this range. This range corresponds to the angle <-1;1). Example: if the output is -0.5 it corresponds to -0.5π.

2.4.1 Available versions

The 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 result may saturate.

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

Table 2-4. Function versions

Function name	Input type	Result type	Description
GFLIB_Asin_F16	frac16_t	frac16_t	Calculation of the arcsin(x) / π where the input argument is a 16-bit fractional within the range <-1;1). The output is a 16-bit fractional value within the range <-1;1) that represents an angle in radians within the range <-π;π).

2.4.2 Declaration

The available GFLIB_Asin functions have the following declarations:

```
frac16_t GFLIB_Asin_F16(frac16_t f16Val)
```

2.4.3 Function use

The use of the GFLIB_Asin function is shown in the following example:

```
#include "gflib.h"

static frac16_t f16Result;
static frac16_t f16Value;

void main(void)
{
   f16Value = FRAC16(0.5);         /* f16Value = 0.5 */

   /* f16Result = arcsin(f16Value); */
   f16Result = GFLIB_Asin_F16(f16Value);
}
```

2.5 GFLIB_Acos

The GFLIB_Acos function provides a computational method for calculation of a standard inverse trigonometric arccosine function arccos(x), using the piece-wise polynomial approximation. Function arccos(x) takes the ratio of the length of the adjacent side to the length of the hypotenuse and returns the angle.

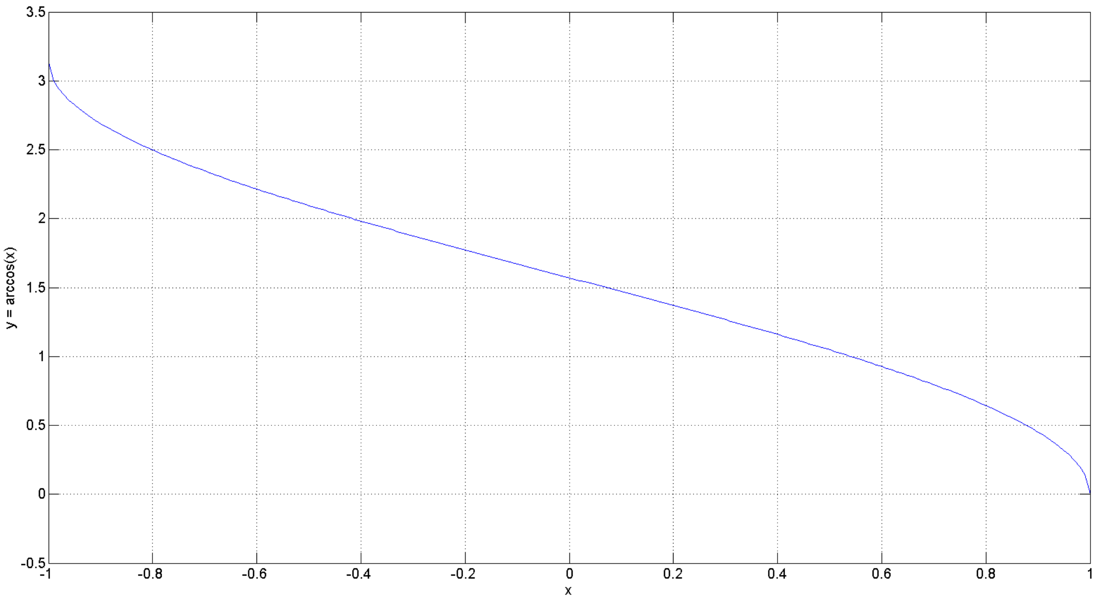

Figure 2-3. Course of the function GFLIB_Acos

The fractional arithmetic is limited by the range <-1;1) so the output can only be within this range. This range corresponds to the angle <-1;1). Example: if the output is -0.5 it corresponds to -0.5π.

2.5.1 Available versions

The 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 result may saturate.

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

Table 2-5. Function versions

Function name	Input type	Result type	Description
GFLIB_Acos_F16	frac16_t	frac16_t	Calculation of the arccos(x) / π where the input argument is a 16-bit fractional within the range <-1;1). The output is a 16-bit fractional value within the range <-1;1) that represents an angle in radians within the range <-π;π).

2.5.2 Declaration

The available GFLIB_Acos functions have the following declarations:

```
frac16_t GFLIB_Acos_F16(frac16_t f16Val)
```

2.5.3 Function use

The use of the GFLIB_Acos function is shown in the following example:

```
#include "gflib.h"

static frac16_t f16Result;
static frac16_t f16Value;

void main(void)
{
   f16Value = FRAC16(0.5);         /* f16Value = 0.5 */

   /* f16Result = arcscos(f16Value); */
   f16Result = GFLIB_Acos_F16(f16Value);
}
```

2.6 GFLIB_Atan

The GFLIB_Atan function implements the polynomial approximation of the arctangent function. It provides a computational method for calculating the standard trigonometric arctangent function arctan(x), using the piece-wise minimax polynomial approximation. Function arctan(x) takes a ratio, and returns the angle of two sides of a right-angled triangle. The ratio is the length of the side opposite to the angle divided by the length of the side adjacent to the angle. The graph of the arctan(x) is shown in the following figure:

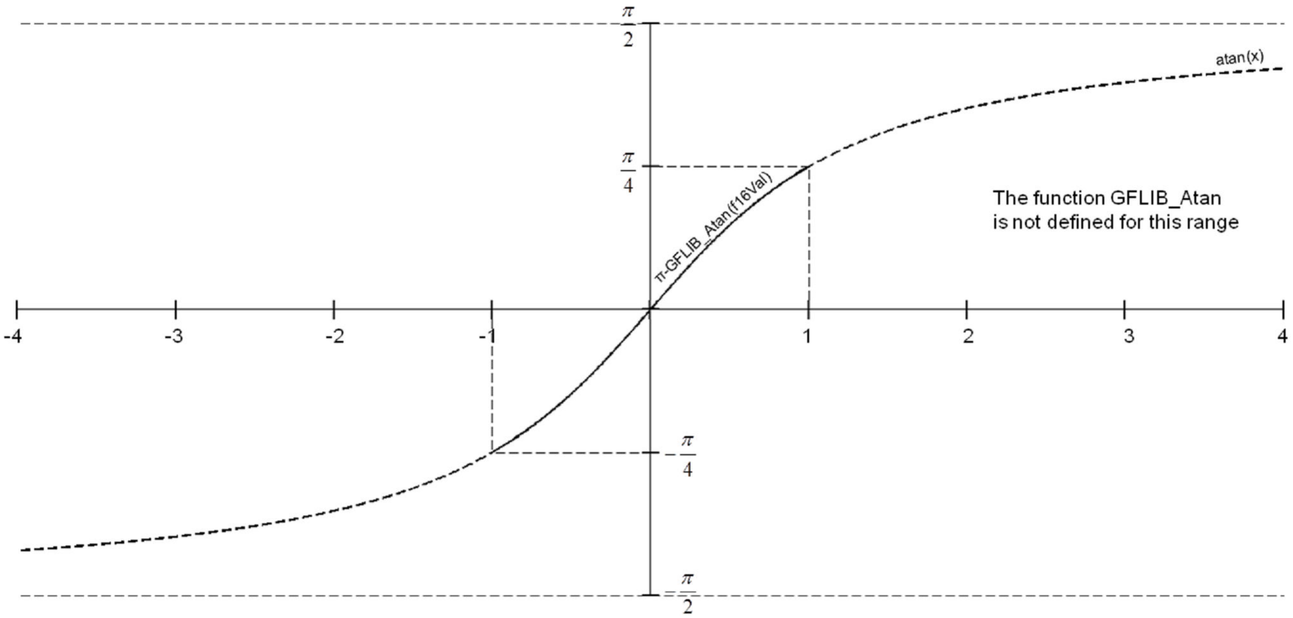

Figure 2-4. Course of the GFLIB_Atan function

The fractional arithmetic version of the GFLIB_Atan function is limited to a certain range of inputs <-1 ; 1). Because the arctangent values are the same, with just an opposite sign for the input ranges <-1 ; 0) and <0 ; 1), the approximation of the arctangent function over the entire defined range of input ratios can be simplified to the approximation for a ratio in the range <0 ; 1). After that, the result will be negated, depending on the input ratio.

2.6.1 Available versions

The function is available in the following versions:

- Fractional output - the output is the fractional portion of the result; the result is within the range <-0.25 ; 0.25), which corresponds to the angle <-π / 4 ; π / 4).

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

Table 2-6. Function versions

Function name	Input type	Result type	Description
GFLIB_Atan_F16	frac16_t	frac16_t	Input argument is a 16-bit fractional value within the range <-1 ; 1). The output is the arctangent of the input as a 16-bit fractional value, normalized within the range <-0.25 ; 0.25), which represents an angle (in radians) in the range <-π / 4 ; π / 4) <-45° ; 45°).

2.6.2 Declaration

The available GFLIB_Atan functions have the following declarations:

```
frac16_t GFLIB_Atan_F16(frac16_t f16Val)
```

2.6.3 Function use

The use of the GFLIB_Atan function is shown in the following example:

```
#include "gflib.h"

static frac16_t f16Result;
static frac16_t f16Val;

void main(void)
{
   f16Val = FRAC16(0.57735026918962576450914878050196);   /* f16Val = tan(30°)  */

   /* f16Result = atan(f16Val); f16Result * 180 => angle[degree] */
   f16Result = GFLIB_Atan_F16(f16Val);
}
```

2.7 GFLIB_AtanYX

The GFLIB_AtanYX function computes the angle, where its tangent is y / x (see the figure below). This calculation is based on the input argument division (y divided by x), and the piece-wise polynomial approximation.

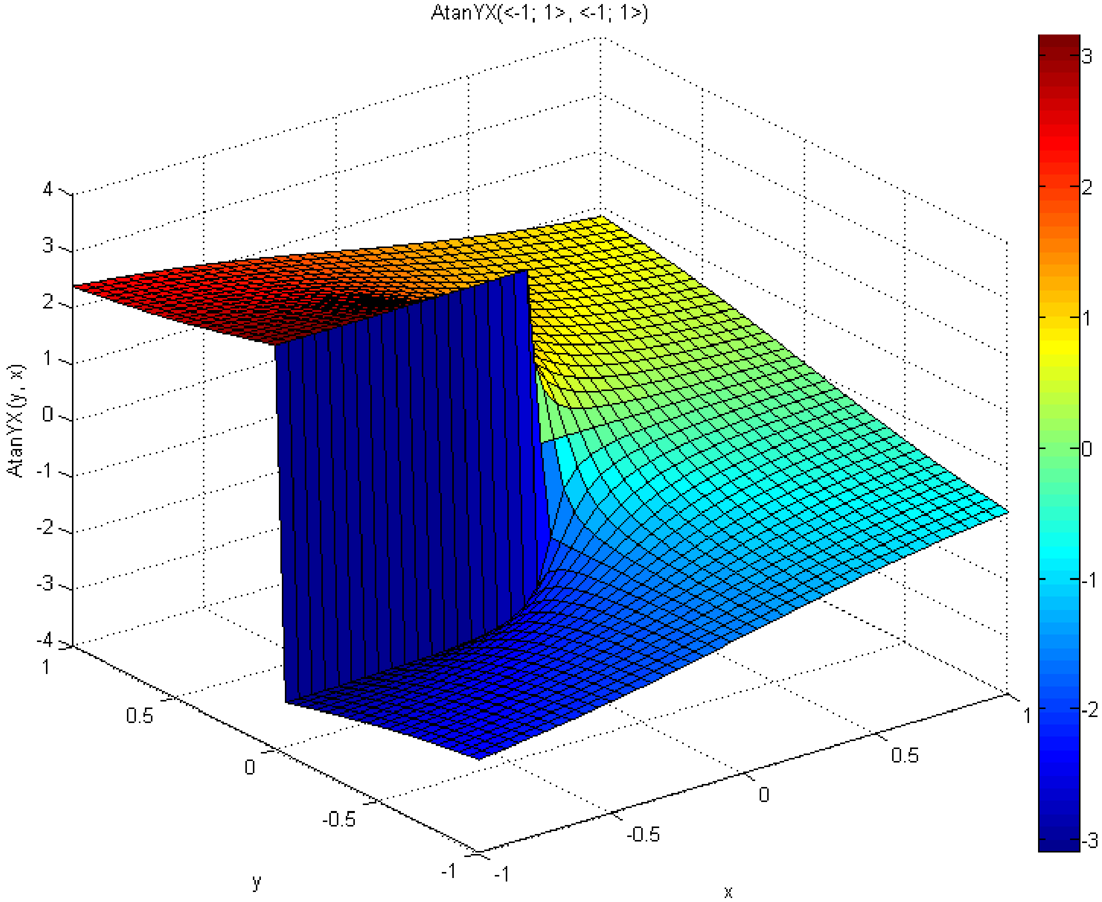

Figure 2-5. Course of the GFLIB_AtanYX function

The first parameter Y is the ordinate (the x coordinate), and the second parameter X is the abscissa (the x coordinate). The counter-clockwise direction is assumed to be positive, and thus a positive angle is computed if the provided ordinate (Y) is positive. Similarly, a negative angle is computed for the negative ordinate. The calculations are performed in several steps. In the first step, the angle is positioned within the correct half-quarter of the circumference of a circle by dividing the angle into two parts: the integral multiple of 45^{o} (half-quarter), and the remaining offset within the 45^{o} range. Simple geometric properties of the Cartesian coordinate system are used to calculate the coordinates of the vector with the calculated angle offset. In the second step, the vector ordinate is divided by the vector abscissa (y / x) to obtain the tangent value of the angle offset. The angle offset is computed by applying the GFLIB_Atan function. The sum of the integral multiple of half-quarters and the angle offset within a single halfquarter form the angle is computed.

The function returns 0 if both input arguments equal 0, and sets the output error flag; in other cases, the output flag is cleared. When compared to the GFLIB_Atan function, the GFLIB_AtanYX function places the calculated angle correctly within the fractional range <-π ; π>.

In the fractional arithmetic, both input parameters are assumed to be in the fractional range <-1 ; 1). The output is within the range <-1 ; 1), which corresponds to the real range <-π ; π).

2.7.1 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), which corresponds to the angle <-π ; π).

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

Table 2-7. Function versions

Function name	Input type		Output type	Result type
	Y	X	Error flag	
GFLIB_AtanYX_F16	frac16_t	frac16_t	bool_t *	frac16_t
	The first input argument is a 16-bit fractional value that contains the ordinate of the input vector (y coordinate). The second input argument is a 16-bit fractional value that contains the abscissa of the input vector (x coordinate). The result is the arctangent of the input arguments as a 16-bit fractional value within the range <-1 ; 1), which corresponds to the real angle range <- π; π). The function sets the boolean error flag pointed to by the output parameter if both inputs are zero; in other cases, the output flag is cleared.			

2.7.2 Declaration

The available GFLIB_AtanYX functions have the following declarations:

```
frac16_t GFLIB_AtanYX_F16(frac16_t f16Y, frac16_t f16X, bool_t *pbErrFlag)
```

2.7.3 Function use

The use of the GFLIB_AtanYX function is shown in the following example:

```
#include "gflib.h"
```

```
static frac16_t f16Result;
static frac16_t f16Y, f16X;
static bool_t bErrFlag;

void main(void)
{
   f16Y = FRAC16(0.9);          /* f16Y = 0.9 */
   f16X = FRAC16(0.3);          /* f16X = 0.3 */

   /* f16Result =  atan(f16Y / f16X); f16Result * 180 => angle [degree] */
   f16Result = GFLIB_AtanYX_F16(f16Y, f16X, &bErrFlag);
}
```

2.8 GFLIB_Sqrt

The GFLIB_Sqrt function returns the square root of the input value. The input must be a non-negative number, otherwise the function returns undefined results. See the following equation:

$$\mathrm{GFLIB\_Sqrt}(x) = \begin{cases} \sqrt{x}, & x \geq 0 \\ \text{undefined}, & x < 0 \end{cases}$$

Equation 6. Algorithm formula

2.8.1 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 <0 ; 1). The function is only defined for non-negative inputs. The function returns undefined results out of this condition.

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

Table 2-8. Function versions

Function name	Input type	Result type	Description
GFLIB_Sqrt_F16	frac16_t	frac16_t	The input value is a 16-bit fractional value, limited to the range <0 ; 1). The function is not defined out of this range. The output is a 16-bit fractional value within the range <0 ; 1).
GFLIB_Sqrt_F16l	frac32_t	frac16_t	The input value is a 32-bit fractional value, limited to the range <0 ; 1). The function is not defined out of this range. The output is a 16-bit fractional value within the range <0 ; 1).

2.8.2 Declaration

The available GFLIB_Sqrt functions have the following declarations:

```
frac16_t GFLIB_Sqrt_F16(frac16_t f16Val)
frac16_t GFLIB_Sqrt_F16l(frac32_t f32Val)
```

2.8.3 Function use

The use of the GFLIB_Sqrt function is shown in the following example:

```
#include "gflib.h"

static frac16_t f16Result;
static frac16_t f16Val;

void main(void)
{
   f16Val = FRAC16(0.5);          /* f16Val = 0.5 */

   /* f16Result = sqrt(f16Val) */
   f16Result = GFLIB_Sqrt_F16(f16Val);
}
```

2.9 GFLIB_Limit

The GFLIB_Limit function returns the value limited by the upper and lower limits. See the following equation:

$$\mathrm{GFLIB\_Limit}(x, min, max) = \begin{cases} min, & x < min \\ max, & x > max \\ x, & \text{else} \end{cases}$$

Equation 7. Algorithm formula

2.9.1 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 result may saturate.

The available versions of the GFLIB_Limit functions are shown in the following table:

Table 2-9. Function versions

Function name	Input type			Result type	Description
	Input	Lower limit	Upper limit		
GFLIB_Limit_F16	frac16_t	frac16_t	frac16_t	frac16_t	The inputs are 16-bit fractional values within the range <-1 ; 1). The function returns a 16-bit fractional value in the range <f16LLim ; f16ULim>.
GFLIB_Limit_F32	frac32_t	frac32_t	frac32_t	frac32_t	The inputs are 32-bit fractional values within the range <-1 ; 1). The function returns a 32-bit fractional value in the range <f32LLim ; f32ULim>.

2.9.2 Declaration

The available GFLIB_Limit functions have the following declarations:

```
frac16_t GFLIB_Limit_F16(frac16_t f16Val, frac16_t f16LLim, frac16_t f16ULim)
frac32_t GFLIB_Limit_F32(frac32_t f32Val, frac32_t f32LLim, frac32_t f32ULim)
```

2.9.3 Function use

The use of the GFLIB_Limit function is shown in the following example:

```
#include "gflib.h"

static frac16_t f16Val, f16ULim, f16LLim, f16Result;

void main(void)
{
   f16ULim = FRAC16(0.8);
   f16LLim = FRAC16(-0.3);
   f16Val = FRAC16(0.9);

   f16Result = GFLIB_Limit_F16(f16Val, f16LLim, f16ULim);
}
```

2.10 GFLIB_LowerLimit

The GFLIB_LowerLimit function returns the value limited by the lower limit. See the following equation:

$$\text{GFLIB\_LowerLimit}(x, min) = \begin{cases} min, & x < min \\ x, & \text{else} \end{cases}$$

Equation 8. Algorithm formula

2.10.1 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 result may saturate.

The available versions of the GFLIB_LowerLimit functions are shown in the following table:

Table 2-10. Function versions

Function name	Input type		Result type	Description
	Input	Lower limit		
GFLIB_LowerLimit_F16	frac16_t	frac16_t	frac16_t	The inputs are 16-bit fractional values within the range <-1 ; 1). The function returns a 16-bit fractional value in the range <f16LLim ; 1).
GFLIB_LowerLimit_F32	frac32_t	frac32_t	frac32_t	The inputs are 32-bit fractional values within the range <-1 ; 1). The function returns a 32-bit fractional value in the range <f32LLim ; 1).

2.10.2 Declaration

The available GFLIB_LowerLimit functions have the following declarations:

```
frac16_t GFLIB_LowerLimit_F16(frac16_t f16Val, frac16_t f16LLim)
frac32_t GFLIB_LowerLimit_F32(frac32_t f32Val, frac32_t f32LLim)
```

2.10.3 Function use

The use of the GFLIB_LowerLimit function is shown in the following example:

```
#include "gflib.h"

static frac16_t f16Val, f16LLim, f16Result;

void main(void)
{
   f16LLim = FRAC16(0.3);
```

```
    f16Val = FRAC16(0.1);

    f16Result = GFLIB_LowerLimit_F16(f16Val, f16LLim);
}
```

2.11 GFLIB_UpperLimit

The GFLIB_UpperLimit function returns the value limited by the upper limit. See the following equation:

$$\text{GFLIB\_UpperLimit}(x, max) = \begin{cases} max, & x > max \\ x, & \text{else} \end{cases}$$

Equation 9. Algorithm formula

2.11.1 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 result may saturate.

The available versions of the GFLIB_UpperLimit functions are shown in the following table:

Table 2-11. Function versions

Function name	Input type		Result type	Description
	Input	Upper limit		
GFLIB_UpperLimit_F16	frac16_t	frac16_t	frac16_t	The inputs are 16-bit fractional values within the range <-1 ; 1). The function returns a 16-bit fractional value in the range <-1 ; f16ULim>.
GFLIB_UpperLimit_F32	frac32_t	frac32_t	frac32_t	The inputs are 32-bit fractional values within the range <-1 ; 1). The function returns a 32-bit fractional value in the range <-1 ; f32ULim>.

2.11.2 Declaration

The available GFLIB_UpperLimit functions have the following declarations:

```
frac16_t GFLIB_UpperLimit_F16(frac16_t f16Val, frac16_t f16ULim)
frac32_t GFLIB_UpperLimit_F32(frac32_t f32Val, frac32_t f32ULim)
```

2.11.3 Function use

The use of the GFLIB_UpperLimit function is shown in the following example:

```
#include "gflib.h"

static frac16_t f16Val, f16ULim, f16Result;

void main(void)
{
   f16ULim = FRAC16(0.3);
   f16Val = FRAC16(0.9);

   f16Result = GFLIB_UpperLimit_F16(f16Val, f16ULim);
}
```

2.12 GFLIB_VectorLimit

The GFLIB_VectorLimit function returns the limited vector by an amplitude. This limitation is calculated to achieve the zero angle error.

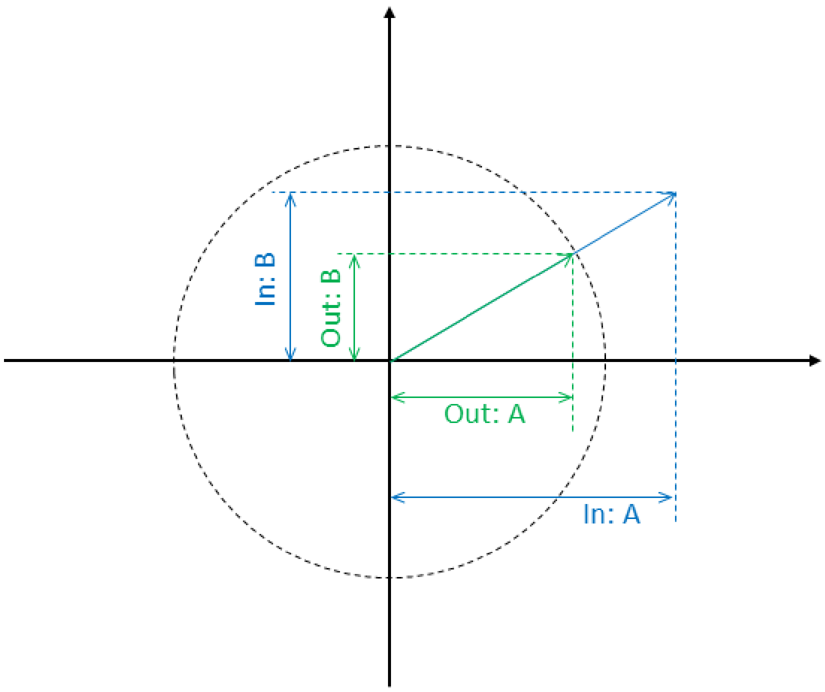

Figure 2-6. Input and releated output

The GFLIB_VectorLimit function limits the amplitude of the input vector. The input vector *a, b* components, are passed into the function as the input arguments. The resulting limited vector is transformed back into the *a, b* components. The limitation is performed according to the following equations:

$$a^* = \begin{cases} a, & \sqrt{a^2+b^2} \le lim \\ a \bullet \frac{lim}{\sqrt{a^2+b^2}}, & \text{else} \end{cases}$$

Equation 10. Algorithm formulas

$$b^* = \begin{cases} b, & \sqrt{a^2+b^2} \le lim \\ b \bullet \frac{lim}{\sqrt{a^2+b^2}}, & \text{else} \end{cases}$$

Equation 11

where:

- a, b are the vector coordinates
- a*, b* are the vector coordinates after limitation
- lim is the maximum amplitude

The relationship between the input and limited output vectors is obvious from Figure 2-6.

If the amplitude of the input vector is greater than the input Lim value, the function calculates the new coordinates from the Lim value; otherwise the function copies the input values to the output.

2.12.1 Available versions

The 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 result may saturate.

The available versions of the GFLIB_VectorLimit functions are shown in the following table:

Table 2-12. Function versions

Function name	Input type		Output type	Result type
	Input	Limit		
GFLIB_VectorLimit_F16	GFLIB_VECTORLIMIT_T_F16 *	frac16_t	GFLIB_VECTORLIMIT_T_F16 *	void
	Limitation of a two-component 16-bit fractional vector within the range <-1;1) with a 16-bit fractional limitation amplitude. The function returns a two-component 16-bit fractional vector.			

2.12.2 GFLIB_VECTORLIMIT_T_F16 type description

Variable name	Input type	Description
f16A	frac16_t	A-component; 16-bit fractional type.
f16B	frac16_t	B-component; 16-bit fractional type

2.12.3 Declaration

The available GFLIB_VectorLimit functions have the following declarations:

```
frac16_t GFLIB_VectorLimit_F16(const GFLIB_VECTORLIMIT_T_F16 *psVectorIn, frac16_t f16Lim,
GFLIB_VECTORLIMIT_T_F16 *psVectorOut)
```

2.12.4 Function use

The use of the GFLIB_VectorLimit function is shown in the following example:

```
#include "gflib.h"

static GFLIB_VECTORLIMIT_T_F16 sVector, sResult;
static frac16_t f16MaxAmpl;


void main(void)
{
   f16MaxAmpl = FRAC16(0.8);
   sVector.f16A = FRAC16(-0.79);
   sVector.f16B = FRAC16(0.86);

   GFLIB_VectorLimit_F16(&sVector, f16MaxAmpl, &sResult);
}
```

2.13 GFLIB_VectorLimit1

The GFLIB_VectorLimit1 function returns the limited vector by an amplitude. This limitation is calculated to achieve that the first component remains unchanged (if the limitation factor allows).

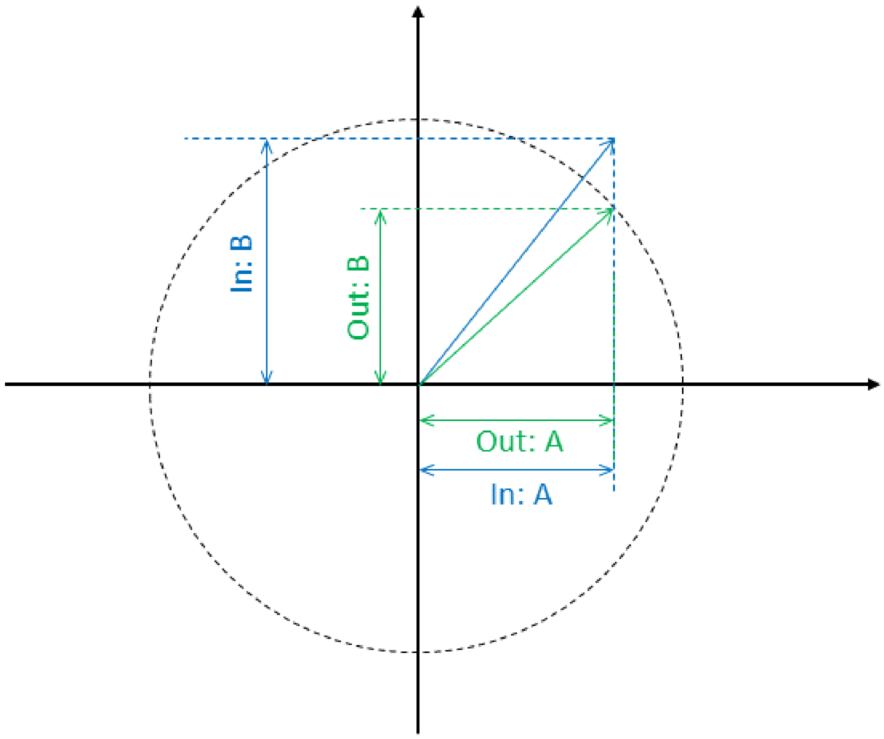

Figure 2-7. Input and releated output

The GFLIB_VectorLimit1 function limits the amplitude of the input vector. The input vector *a, b* components are passed to the function as the input arguments. The resulting limited vector is transformed back into the *a, b* components. The limitation is performed according to the following equations:

$$a^* = \begin{cases} a, & |a| \le lim \\ lim \cdot \operatorname{sgn}(a), & \text{else} \end{cases}$$

Equation 12

$$b^* = \begin{cases} b, & |b| \le \sqrt{lim^2 - a^{*2}} \\ \sqrt{lim^2 - a^{*2}} \cdot \operatorname{sgn}(b), & \text{else} \end{cases}$$

Equation 13

where:

- a, b are the vector coordinates
- a*, b* are the vector coordinates after limitation
- lim is the maximum amplitude

The relationship between the input and limited output vectors is shown in Figure 2-7.

If the amplitude of the input vector is greater than the input Lim value, the function calculates the new coordinates from the Lim value; otherwise the function copies the input values to the output.

2.13.1 Available versions

The 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 result may saturate.

The available versions of the GFLIB_VectorLimit1 function are shown in the following table:

Table 2-13. Function versions

Function name	Input type		Output type	Result type
	Input	Limit		
GFLIB_VectorLimit1_F16	GFLIB_VECTORLIMIT_T_F16 *	frac16_t	GFLIB_VECTORLIMIT_T_F16 *	void
	Limitation of a two-component 16-bit fractional vector within the range <-1 ; 1) with a 16-bit fractional limitation amplitude. The function returns a two-component 16-bit fractional vector.			

2.13.2 GFLIB_VECTORLIMIT_T_F16 type description

Variable name	Input type	Description
f16A	frac16_t	A-component; 16-bit fractional type.
f16B	frac16_t	B-component; 16-bit fractional type.

2.13.3 Declaration

The available GFLIB_VectorLimit1 functions have the following declarations:

```
frac16_t GFLIB_VectorLimit1_F16(const GFLIB_VECTORLIMIT_T_F16 *psVectorIn, frac16_t f16Lim,
GFLIB_VECTORLIMIT_T_F16 *psVectorOut)
```

2.13.4 Function use

The use of the GFLIB_VectorLimit1 function is shown in the following example:

```
#include "gflib.h"

static GFLIB_VECTORLIMIT_T_F16 sVector, sResult;
static frac16_t f16MaxAmpl;


void main(void)
{
   f16MaxAmpl = FRAC16(0.5);
   sVector.f16A = FRAC16(-0.4);
   sVector.f16B = FRAC16(0.2);

   GFLIB_VectorLimit1_F16(&sVector, f16MaxAmpl, &sResult);
}
```

2.14 GFLIB_Hyst

The GFLIB_Hyst function represents a hysteresis (relay) function. The function switches the output between two predefined values. When the input is higher than the upper threshold, the output is high; when the input is lower than the lower threshold, the output is low. When the input is between the two thresholds, the output retains its value. See the following figure:

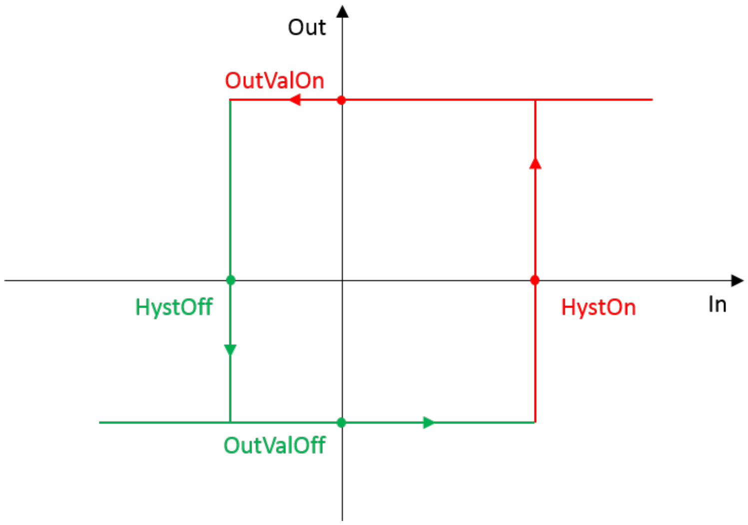

Figure 2-8. GFLIB_Hyst functionality

The four points in the figure are to be set up in the parameters structure of the function. For a proper functionality, the HystOn point must be greater than the HystOff point.

2.14.1 Available versions

This function is available in the following versions:

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

The available versions of the GFLIB_Hyst function are shown in the following table.

Table 2-14. Function versions

Function name	Input type	Parameters	Result type	Description
GFLIB_Hyst_F16	frac16_t	GFLIB_HYST_T_F16 *	frac16_t	The input is a 16-bit fractional value within the range <-1 ; 1). The output is a two-state 16-bit fractional value.

2.14.2 GFLIB_HYST_T_F16

Variable name	Input type	Description
f16HystOn	frac16_t	The point where the output sets the output to the f16OutValOn value when the input rises. Set by the user.
f16HystOff	frac16_t	The point where the output sets the output to the f16OutValOff value when the input falls. Set by the user.
f16OutValOn	frac16_t	The ON value. Set by the user.
f16OutValOff	frac16_t	The OFF value. Set by the user.
f16OutState	frac16_t	The output state. Set by the algorithm. Must be initialized by the user.

2.14.3 Declaration

The available GFLIB_Hyst functions have the following declarations:

```
frac16_t GFLIB_Hyst_F16(frac16_t f16Val, GFLIB_HYST_T_F16 *psParam)
```

2.14.4 Function use

The use of the GFLIB_Hyst function is shown in the following example:

```
#include "gflib.h"
```

```
static frac16_t f16Result, f16InVal;
static GFLIB_HYST_T_F16 sParam;

void main(void)
{
    f16InVal = FRAC16(-0.11);
    sParam.f16HystOn = FRAC16(0.5);
    sParam.f16HystOff = FRAC16(-0.1);
    sParam.f16OutValOn = FRAC16(0.7);
    sParam.f16OutValOff = FRAC16(0.3);
    sParam.f16OutState = FRAC16(0.0);

    f16Result = GFLIB_Hyst_F16(f16InVal, &sParam);
}
```

2.15 GFLIB_Lut1D

The GFLIB_Lut1D function implements the one-dimensional look-up table.

$$y = y_1 + \frac{y_2 - y_1}{x_2 - x_1}(x - x_1)$$

Equation 14.

where:

- y is the interpolated value
- y_1 and y_2 are the ordinate values at the beginning and end of the interpolating interval, respectively
- x_1 and x_2 are the abscissa values at the beginning and end of the interpolating interval, respectively
- x is the input value provided to the function in the X input argument

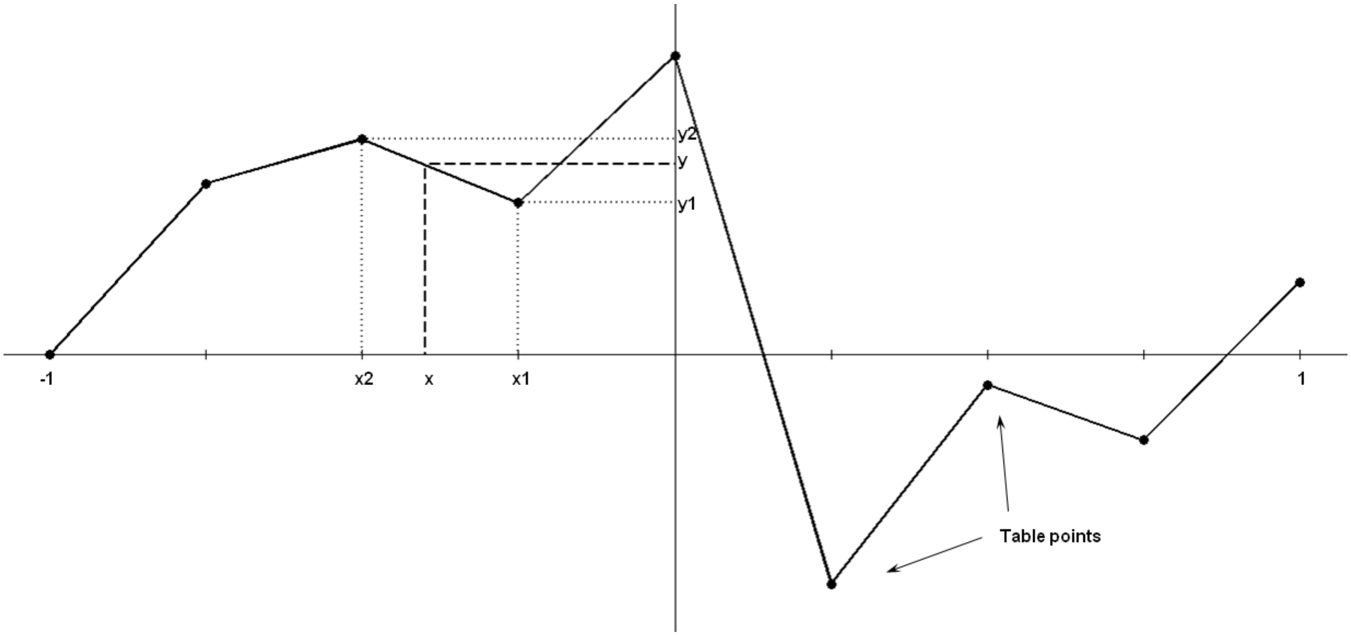

Figure 2-9. Algorithm diagram - fractional version

The GFLIB_Lut1D fuses a table of the precalculated function points. These points are selected with a fixed step.

The fractional version of the algorithm has a defined interval of inputs within the range <-1 ; 1). The number of points must be $2^n + 1$, where n can range from 1 through to 15.

The function finds two nearest precalculated points of the input argument, and calculates the output value using the linear interpolation between these two points.

2.15.1 Available versions

The 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 GFLIB_Lut1D function are shown in the following table:

Table 2-15. Function versions

Function name	Input type	Parameters		Result type
		Table	Table size	
GFLIB_Lut1D_F16	frac16_t	frac16_t *	uint16_t	frac16_t

Table continues on the next page...

Table 2-15. Function versions (continued)

Function name	Input type	Parameters		Result type
		Table	Table size	
	The input arguments are the 16-bit fractional value that contains the abscissa for which the 1-D interpolation is performed, the pointer to a structure which contains the 16-bit fractional values of the look-up table, and the size of the look-up table. The table size parameter can be in the range <1 ; 15> (that means the parameter is $\log_2$ of the number of points - 1). The output is the interpolated 16-bit fractional value computed from the look-up table.			

2.15.2 Declaration

The available GFLIB_Lut1D functions have the following declarations:

```
frac16_t GFLIB_Lut1D_F16(frac16_t f16X, const frac16_t *pf16Table, uint16_t u16TableSize)
```

2.15.3 Function use

The use of the GFLIB_Lut1D function is shown in the following example:

```
#include "gflib.h"

static frac16_t f16Result, f16X;
static uint16_t u16TableSize;
static frac16_t f16Table[9] = {FRAC16(0.8), FRAC16(0.1), FRAC16(-0.2), FRAC16(0.7),
FRAC16(0.2), FRAC16(-0.3), FRAC16(-0.8), FRAC16(0.91), FRAC16(0.99)};

void main(void)
{
   u16TableSize = 3;                           /* size of table = 2 ^ 3 + 1 */
   f16X = FRAC16(0.625);                       /* f16X = 0.625 */

   /* f16Result = value from look-up table between 7th and 8th position */
   f16Result = GFLIB_Lut1D_F16(f16X, f16Table, u16TableSize);
}
```

2.16 GFLIB_LutPer1D

The GFLIB_LutPer1D function approximates the one-dimensional arbitrary user function using the interpolation look-up method. It is periodic.

$$y = y_1 + \frac{y_2 - y_1}{x_2 - x_1}(x - x_1)$$

Equation 15.

where:

- y is the interpolated value
- y_1 and y_2 are the ordinate values at the beginning and end of the interpolating interval, respectively
- x_1 and x_2 are the abscissa values at the beginning and end of the interpolating interval, respectively
- x is the input value provided to the function in the X input argument

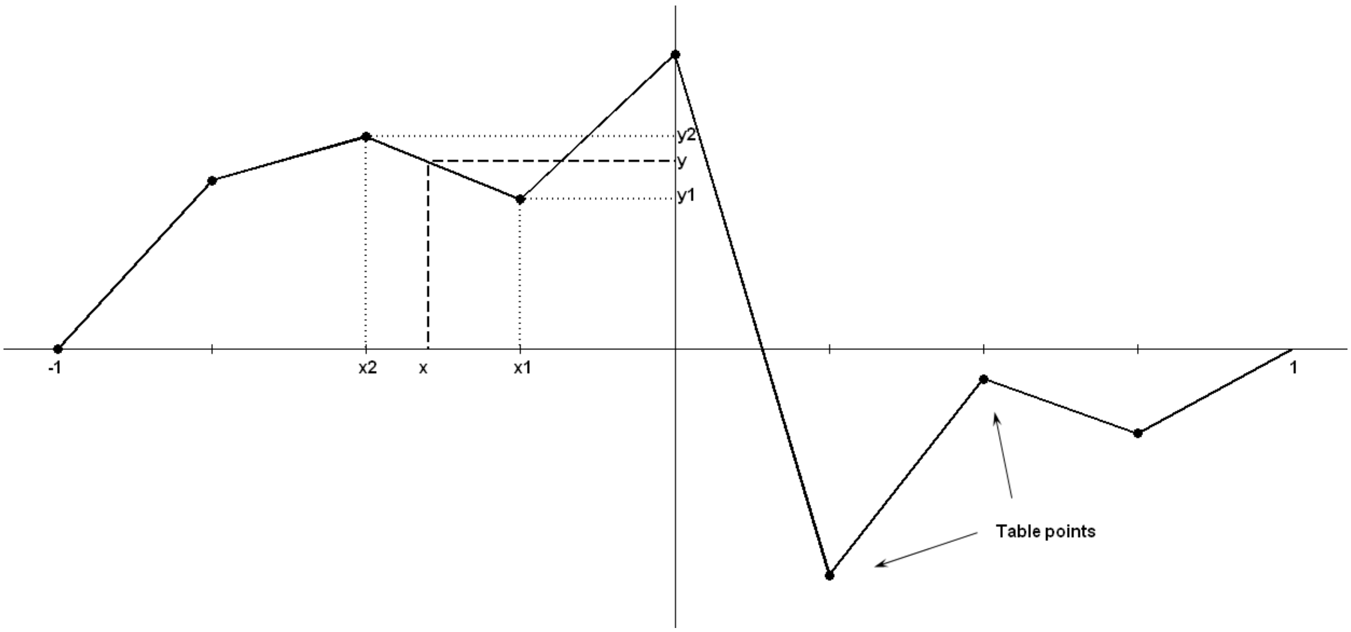

Figure 2-10. Algorithm diagram - fractional version

The GFLIB_LutPer1D fuses a table of the precalculated function points. These points are selected with a fixed step.

The fractional version of the algorithm has a defined interval of inputs within the range <-1 ; 1). The number of points must be 2^n, where n can range from 1 through to 15.

The function finds two nearest precalculated points of the input argument, and calculates the output value using the linear interpolation between these two points. This algorithm serves for periodical functions, that means if the input argument lies behind the last precalculated point of the function, the interpolation is calculated between the last and first points of the table.

2.16.1 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 GFLIB_LutPer1D function are shown in the following table:

Table 2-16. Function versions

Function name	Input type	Parameters		Result type
		Table	Table size	
GFLIB_LutPer1D_F16	frac16_t	frac16_t *	uint16_t	frac16_t
	The input arguments are the 16-bit fractional value that contains the abscissa for which the 1-D interpolation is performed, the pointer to a structure which contains the 16-bit fractional values of the look-up table, and the size of the look-up table. The table size parameter can be in the range <1 ; 15> (that means the parameter is $\log_2$ of the number of points). The output is the interpolated 16-bit fractional value computed from the look-up table.			

2.16.2 Declaration

The available GFLIB_LutPer1D functions have the following declarations:

```
frac16_t GFLIB_LutPer1D_F16(frac16_t f16X, const frac16_t *pf16Table, uint16_t u16TableSize)
```

2.16.3 Function use

The use of the GFLIB_LutPer1D function is shown in the following example:

```
#include "gflib.h"

static frac16_t f16Result, f16X;
static uint16_t u16TableSize;
static frac16_t f16Table[8] = {FRAC16(0.8), FRAC16(0.1), FRAC16(-0.2), FRAC16(0.7),
FRAC16(0.2), FRAC16(-0.3), FRAC16(-0.8), FRAC16(0.91)};

void main(void)
{
   u16TableSize = 3;                    /* size of table = 2 ^ 3 */
   f16X = FRAC16(0.25);                  /* f16X = 0.25 */

   /* f16Result = value from look-up table at 6th position */
   f16Result = GFLIB_LutPer1D_F16(f16X, f16Table, u16TableSize);
}
```

2.17 GFLIB_Ramp

The GFLIB_Ramp function calculates the up / down ramp with the defined fixed-step increment / decrement. These two parameters must be set by the user.

For a proper use, it is recommended that the algorithm is initialized by the GFLIB_RampInit function, before using the GFLIB_Ramp function. The GFLIB_RampInit function initializes the internal state variable of the GFLIB_Ramp algorithm with a defined value. You must call the init function when you want the ramp to be initialized.

The use of the GFLIB_Ramp function is as follows: If the target value is greater than the ramp state value, the function adds the ramp-up value to the state output value. The output will not trespass the target value, that means it will stop at the target value. If the target value is lower than the state value, the function subtracts the ramp-down value from the state value. The output is limited to the target value, that means it will stop at the target value. This function returns the actual ramp output value. As time passes, it is approaching the target value by step increments defined in the algorithm parameters' structure. The functionality of the implemented ramp algorithm is explained in the next figure:

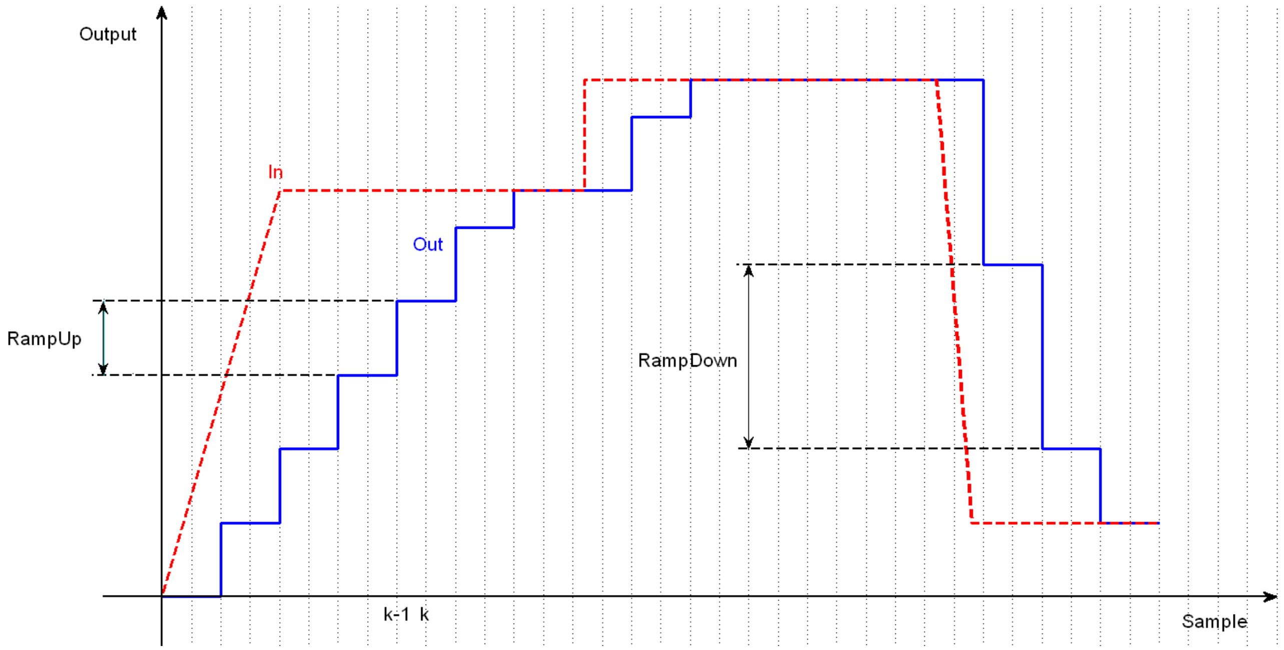

Figure 2-11. GFLIB_Ramp functionality

2.17.1 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 result may saturate.

The available versions of the GFLIB_RampInit functions are shown in the following table:

Table 2-17. Init function versions

Function name	Input type	Parameters	Result type	Description
GFLIB_RampInit_F16	frac16_t	GFLIB_RAMP_T_F16 *	void	Input argument is a 16-bit fractional value that represents the initialization value. The parameters' structure is pointed to by a pointer. The input data value is in the range <-1 ; 1).
GFLIB_RampInit_F32	frac32_t	GFLIB_RAMP_T_F32 *	void	Input argument is a 32-bit fractional value that represents the initialization value. The parameters' structure is pointed to by a pointer. The input data value is in the range <-1 ; 1).

The available versions of the GFLIB_Ramp functions are shown in the following table:

Table 2-18. Function versions

Function name	Input type	Parameters	Result type	Description
GFLIB_Ramp_F16	frac16_t	GFLIB_RAMP_T_F16 *	frac16_t	Input argument is a 16-bit fractional value that represents the target output value. The parameters' structure is pointed to by a pointer. The function returns a 16-bit fractional value, which represents the actual ramp output value. The input data value is in the range <-1 ; 1), and the output data value is in the range <-1 ; 1).
GFLIB_Ramp_F32	frac32_t	GFLIB_RAMP_T_F32 *	frac32_t	Input argument is a 32-bit fractional value that represents the target output value. The parameters' structure is pointed to by a pointer. The function returns a 32-bit fractional value, which represents the actual ramp output value. The input data value is in the range <-1 ; 1), and the output data value is in the range <-1 ; 1).

2.17.2 GFLIB_RAMP_T_F16

Variable name	Type	Description
f16State	frac16_t	Actual value - controlled by the algorithm.
f16RampUp	frac16_t	Value of the ramp-up increment. The data value is in the range <-1 ; 1). Set by the user.
f16RampDown	frac16_t	Value of the ramp-down increment. The data value is in the range <-1 ; 1). Set by the user.

2.17.3 GFLIB_RAMP_T_F32

Variable name	Type	Description
f32State	frac32_t	Actual value - controlled by the algorithm.
f32RampUp	frac32_t	Value of the ramp-up increment. The data value is in the range <-1 ; 1). Set by the user.
f32RampDown	frac32_t	Value of the ramp-down increment. The data value is in the range <-1 ; 1). Set by the user.

2.17.4 Declaration

The available GFLIB_RampInit functions have the following declarations:

```
void GFLIB_RampInit_F16(frac16_t f16InitVal, GFLIB_RAMP_T_F16 *psParam)
void GFLIB_RampInit_F32(frac32_t f32InitVal, GFLIB_RAMP_T_F32 *psParam)
```

The available GFLIB_Ramp functions have the following declarations:

```
frac16_t GFLIB_Ramp_F16(frac16_t f16Target, GFLIB_RAMP_T_F16 *psParam)
frac32_t GFLIB_Ramp_F32(frac32_t f32Target, GFLIB_RAMP_T_F32 *psParam)
```

2.17.5 Function use

The use of the GFLIB_RampInit and GFLIB_Ramp functions is shown in the following example:

```
#include "gflib.h"

static frac16_t f16InitVal;
static GFLIB_RAMP_T_F16 sParam;
static frac16_t f16Target, f16Result;

void Isr(void);

void main(void)
{
   sParam.f16RampUp = FRAC16(0.1);
   sParam.f16RampDown = FRAC16(0.02);
   f16Target = FRAC16(0.75);
   f16InitVal = FRAC16(0.9);
   GFLIB_RampInit_F16(f16InitVal, &sParam);
}

/* periodically called function */
void Isr()
{
   f16Result = GFLIB_Ramp_F16(f16Target, &sParam);
}
```

2.18 GFLIB_DRamp

The GFLIB_DRamp function calculates the up / down ramp with the defined step increment / decrement. The algorithm approaches the target value when the stop flag is not set, and/or returns to the instant value when the stop flag is set.

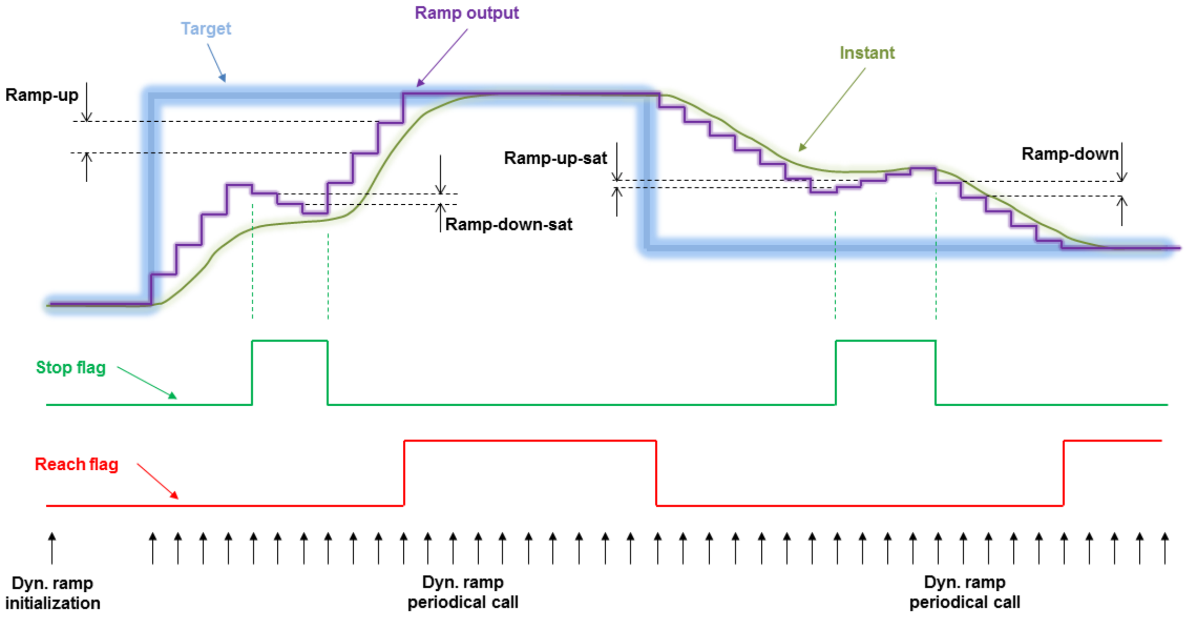

Figure 2-12. GFLIB_DRamp functionality

For a proper use, it is recommended that the algorithm is initialized by the GFLIB_DRampInit function, before using the GFLIB_DRamp function. This function initializes the internal state variable of GFLIB_DRamp algorithm with the defined value. You must call this function when you want the ramp to be initialized.

The GFLIB_DRamp function calculates a ramp with a different set of up / down parameters, depending on the state of the stop flag. If the stop flag is cleared, the function calculates the ramp of the actual state value towards the target value, using the up or down increments contained in the parameters' structure. If the stop flag is set, the function calculates the ramp towards the instant value, using the up or down saturation increments.

If the target value is greater than the state value, the function adds the ramp-up value to the state value. The output cannot be greater than the target value (case of the stop flag being cleared), nor lower than the instant value (case of the stop flag being set).

If the target value is lower than the state value, the function subtracts the ramp-down value from the state value. The output cannot be lower than the target value (case of the stop flag being cleared), nor greater than the instant value (case of the stop flag being set).

If the actual internal state reaches the target value, the reach flag is set.

2.18.1 Available versions

The 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 result may saturate.

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

Table 2-19. Init function versions

Function name	Input type	Parameters	Result type	Description
GFLIB_DRampInit_F16	frac16_t	GFLIB_DRAMP_T_F16 *	void	Input argument is a 16-bit fractional value that represents the initialization value. The parameters' structure is pointed to by a pointer. The input data value is in the range <-1 ; 1).
GFLIB_DRampInit_F32	frac32_t	GFLIB_DRAMP_T_F32 *	void	Input argument is a 32-bit fractional value that represents the initialization value. The parameters' structure is pointed to by a pointer. The input data value is in the range <-1 ; 1).

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

Table 2-20. Function versions

<table>
<tr><th rowspan="2">Function name</th><th colspan="3">Input type</th><th rowspan="2">Parameters</th><th rowspan="2">Result type</th></tr>
<tr><th>Target</th><th>Instant</th><th>Stop flag</th></tr>
<tr><td rowspan="2">GFLIB_DRamp_F16</td><td>frac16_t</td><td>frac16_t</td><td>bool_t *</td><td>GFLIB_DRAMP_T_F16 *</td><td>frac16_t</td></tr>
<tr><td colspan="5">The target and instant arguments are 16-bit fractional values. The parameters' structure is pointed to by a pointer. The function returns a 16-bit fractional value, which represents the actual ramp output value. The input data values are in the range of <-1 ; 1), the Stop flag parameter is a pointer to a boolean value, and the output data value is in the range <-1 ; 1).</td></tr>
<tr><td rowspan="2">GFLIB_DRamp_F32</td><td>frac32_t</td><td>frac32_t</td><td>bool_t *</td><td>GFLIB_DRAMP_T_F32 *</td><td>frac32_t</td></tr>
<tr><td colspan="5">The target and instant arguments are 32-bit fractional values. The parameters' structure is pointed to by a pointer. The function returns a 32-bit fractional value, which represents the actual ramp output value. The input data values are in the range <-1 ; 1), the Stop flag parameter is a pointer to a boolean value, and the output data value is in the range <-1 ; 1).</td></tr>
</table>

2.18.2 GFLIB_DRAMP_T_F16

Variable name	Type	Description
f16State	frac16_t	Actual value - controlled by the algorithm.
f16RampUp	frac16_t	Value of non-saturation ramp-up increment. The data value is in the range <-1 ; 1). Set by the user.
f16RampDown	frac16_t	Value of non-saturation ramp-down increment. The data value is in the range <-1 ; 1). Set by the user.
f16RampUpSat	frac16_t	Value of saturation ramp-up increment. The data value is in the range <-1 ; 1). Set by the user.
f16RampDownSat	frac16_t	Value of saturation ramp-down increment. The data value is in the range <-1 ; 1). Set by the user.
bReachFlag	bool_t	If the actual state value reaches the target value, this flag is set, otherwise, it is cleared. Set by the algorithm.

2.18.3 GFLIB_DRAMP_T_F32

Variable name	Type	Description
f32State	frac32_t	Actual value - controlled by the algorithm.
f32RampUp	frac32_t	Value of non-saturation ramp-up increment. The data value is in the range <-1 ; 1). Set by the user.
f32RampDown	frac32_t	Value of non-saturation ramp-down increment. The data value is in the range <-1 ; 1). Set by the user.
f32RampUpSat	frac32_t	Value of saturation ramp-up increment. The data value is in the range <-1 ; 1). Set by the user.
f32RampDownSat	frac32_t	Value of saturation ramp-down increment. The data value is in the range <-1 ; 1). Set by the user.
bReachFlag	bool_t	If the actual state value reaches the target value, this flag is set, otherwise, it is cleared. Set by the algorithm.

2.18.4 Declaration

The available GFLIB_DRampInit functions have the following declarations:

```
void GFLIB_DRampInit_F16(frac16_t f16InitVal, GFLIB_DRAMP_T_F16 *psParam)
void GFLIB_DRampInit_F32(frac32_t f32InitVal, GFLIB_DRAMP_T_F32 *psParam)
```

The available GFLIB_DRamp functions have the following declarations:

```
frac16_t GFLIB_DRamp_F16(frac16_t f16Target, frac16_t f16Instant, const bool_t *pbStopFlag,
GFLIB_DRAMP_T_F16 *psParam)
frac32_t GFLIB_DRamp_F32(frac32_t f32Target, frac32_t f32Instant, const bool_t *pbStopFlag,
GFLIB_DRAMP_T_F32 *psParam)
```

2.18.5 Function use

The use of the GFLIB_DRampInit and GFLIB_DRamp functions is shown in the following example:

```
#include "gflib.h"

static frac16_t f16InitVal, f16Target, f16Instant, f16Result;
static GFLIB_DRAMP_T_F16 sParam;
static bool_t bStopFlag;

void Isr(void);

void main(void)
{
   sParam.f16RampUp = FRAC16(0.05);
   sParam.f16RampDown = FRAC16(0.02);
   sParam.f16RampUpSat = FRAC16(0.025);
   sParam.f16RampDownSat = FRAC16(0.01);
   f16Target = FRAC16(0.7);
   f16InitVal = FRAC16(0.3);
   f16Instant = FRAC16(0.6);
   bStopFlag = FALSE;

   GFLIB_DRampInit_F16(f16InitVal, &sParam);
}

/* periodically called function */
void Isr()
{
    f16Result = GFLIB_DRamp_F16(f16Target, f16Instant, &bStopFlag, &sParam);
}
```

2.19 GFLIB_FlexRamp

The GFLIB_FlexRamp function calculates the up/down ramp with a fixed-step increment that is calculated based on the required speed change per a defined duration. These parameters must be set by the user.

The GFLIB_FlexRamp algorithm consists of three functions that must be used for a proper functionality of the algorithm:

- GFLIB_FlexRampInit - this function initializes the state variable with a defined value and clears the reach flag

- GFLIB_FlexRampCalcIncr - this function calculates the increment and clears the reach flag
- GFLIB_FlexRamp - this function calculates the ramp in the periodically called loop

For a proper use, it is recommended to initialize the algorithm by the GFLIB_FlexRampInit function. The GFLIB_FlexRampInit function initializes the internal state variable of the algorithm with a defined value and clears the reach flag. Call the init function when you want the ramp to be initialized.

To calculate the increment, it is necessary to use the GFLIB_FlexRampCalcIncr function. This function is called at the point when you want to change the ramp output value. This function's inputs are the target value and the duration. The target value is the destination value that you want to get to. The duration is the time required to change the ramp output from the actual state to the target value. To be able to calculate the ramp increment, fill the control structure with the sample time, that means the period of the loop where the GFLIB_FlexRamp funciton is called. The structure also contains a variable which determines the maximum value of the increment. It is necessary to set it up too. The equation for the increment calculation is as follows:

$$I = \frac{V_t - V_s}{T} \cdot T_s$$

Equation 16.

where:

- I is the increment
- V_t is the target value
- V_s is the state (actual) value (in the structure)
- T is the duration of the ramp (to reach the target value starting at the state value)
- T_s is the sample time, that means the period of the loop where the ramp algorithm is called (set in the structure)

If the increment is greater than the maximum increment (set in the structure), the increment uses the maximum increment value.

As soon as the new increment is calculated, call the GFLIB_FlexRamp algorithm in the periodical control loop. The function works as follows: The function adds the increment to the state value (from the previous step), which results in a new state. The new state is returned by the function. As the time passes, the algorithm is approaching the target value. If the new state trespasses the target value, that new state is limited to the target value and the reach flag is set. The functionality of the implemented algorithm is shown in the next figure:

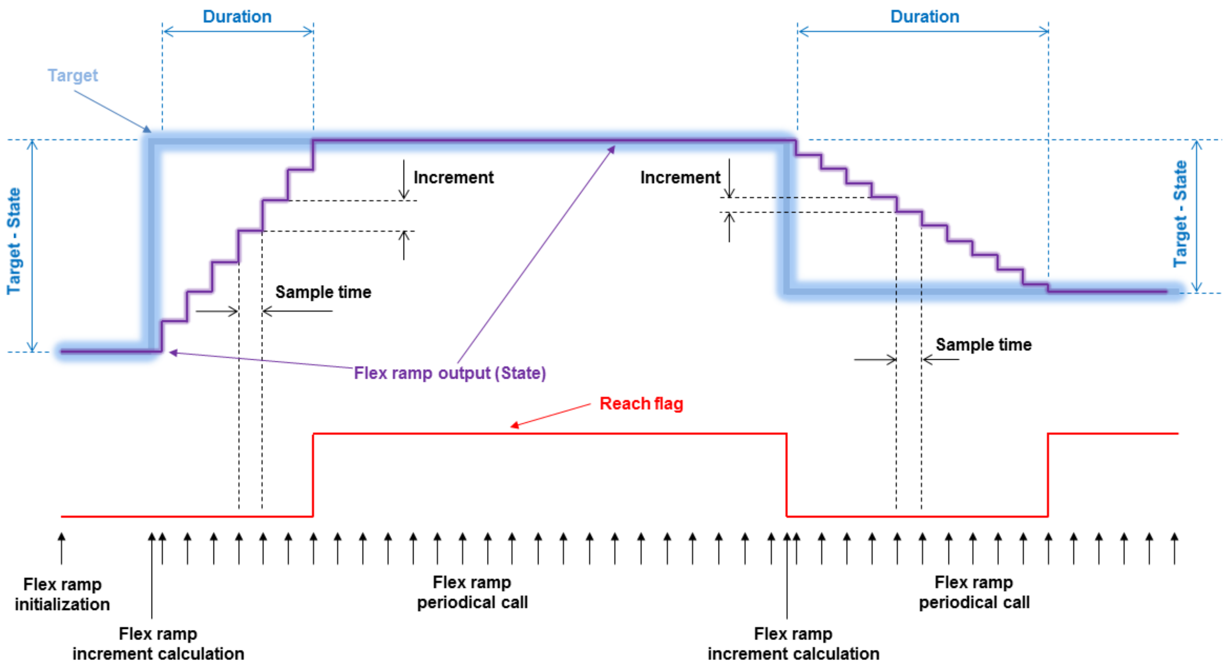

Figure 2-13. GFLIB_Ramp functionality

2.19.1 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 input parameters are the fractional and accumulator types.

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

Table 2-21. Init function versions

Function name	Input type	Parameters	Result type	Description
GFLIB_FlexRampInit_F16	frac16_t	GFLIB_FLEXRAMP_T_F32 *	void	Input argument is a 16-bit fractional value that represents the initialization value. The parameters' structure is pointed to by a pointer. The input data value is in the range <-1 ; 1).

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

Table 2-22. Increment calculation function versions

Function name	Input type		Parameters	Result type
	Target	Duration		
GFLIB_FlexRampCalcIncr_F16	frac16_t	acc32_t	GFLIB_FLEXRAMP_T_F32 *	void
	Input arguments are a 16-bit fractional value in the range <-1 ; 1) that represents the target output value and a 32-bit accumulator value in the range <0 ; 65536.0) that represents the duration (in seconds) of the ramp to reach the target value. The parameters' structure is pointed to by a pointer.			

Table 2-23. Function versions

Function name	Parameters	Result type	Description
GFLIB_FlexRamp_F16	GFLIB_FLEXRAMP_T_F32 *	frac16_t	The parameters' structure is pointed to by a pointer. The function returns a 16-bit fractional value, which represents the actual ramp output value. The output data value is in the range <-1 ; 1).

2.19.2 GFLIB_FLEXRAMP_T_F32

Variable name	Type	Description
f32State	frac32_t	Actual value. Controlled by the GFLIB_FlexRampInit_F16 and the GFLIB_FlexRamp_F16 algorithms.
f32Incr	frac32_t	Value of the flex ramp increment. Controlled by the GFLIB_FlexRampCalcIncr_F16 algorithm.
f32Target	frac32_t	The target value of the flex ramp algorithm. Controlled by the GFLIB_FlexRampCalcIncr_F16 algorithm.
f32Ts	frac32_t	The sample time, that means the period of the loop where the GFLIB_FlexRamp_F16 algorithms are periodically called. The data value (in seconds) is in the range <0 ; 1). Set by the user.
f32IncrMax	frac32_t	Maximum value of the flex ramp increment. The data value is in the range <-1 ; 1). Set by the user.
bReachFlag	bool_t	Reach flag. This flag is controlled by the GFLIB_FlexRamp_F16 algorithm. It is cleared by the GFLIB_FlexRampInit_F16 and GFLIB_FlexRampCalcIncr_F16 algorithms.

2.19.3 Declaration

The available GFLIB_FlexRampInit functions have the following declarations:

```
void GFLIB_FlexRampInit_F16(frac16_t f16InitVal, GFLIB_FLEXRAMP_T_F16 *psParam)
```

The available GFLIB_FlexRampCalcIncr functions have the following declarations:

```
void GFLIB_FlexRampCalcIncr_F16(frac16_t f16Target, acc32_t a32Duration,
GFLIB_FLEXRAMP_T_F16 *psParam)
```

The available GFLIB_FlexRamp functions have the following declarations:

```
frac16_t GFLIB_FlexRamp_F16(GFLIB_FLEXRAMP_T_F16 *psParam)
```

2.19.4 Function use

The use of the GFLIB_FlexRampInit, GFLIB_FlexRampCalcIncr and GFLIB_FlexRamp functions is shown in the following example:

```
#include "gflib.h"

static frac16_t f16InitVal;
static GFLIB_FLEXRAMP_T_F16 sFlexRamp;
static frac16_t f16Target, f16RampResult;
static acc32_t a32RampDuration

void Isr(void);

void main(void)
{
   /* Control loop period is 0.002 s; maximum increment value is 0.15 */
   sFlexRamp.f32Ts = FRAC32(0.002);
   sFlexRamp.f32IncrMax = FRAC32(0.15);

   /* Initial value to 0 */
   f16InitVal = FRAC16(0.0);

   /* Flex ramp initialization */
   GFLIB_FlexRampInit_F16(f16InitVal, &sFlexRamp);

   /* Target value is 0.7 in duration of 5.3 s */
   f16Target = FRAC16(0.7);
   a32RampDuration = ACC32(5.3);;

   /* Flex ramp increment calculation */
   GFLIB_FlexRampCalcIncr_F16(f16Target, a32RampDuration, &sFlexRamp);
}

/* periodically called control loop with a period of 2 ms */
void Isr()
{
   f16RampResult = GFLIB_FlexRamp_F16(&sFlexRamp);
}
```

2.20 GFLIB_DFlexRamp

The GFLIB_DFlexRamp function calculates the up/down ramp with a fixed-step increment that is calculated based on the required speed change per a defined duration.These parameters must be set by the user. The algorithm has stop flags. If none of them is set, the ramp behaves normally. If one of them is set, the ramp can run in the opposite direction.

The GFLIB_DFlexRamp algorithm consists of three functions that must be used for a proper function of the algorithm:

- GFLIB_DFlexRampInit - this function initializes the state variable with a defined value and clears the reach flag
- GFLIB_DFlexRampCalcIncr - this function calculates the increment and clears the reach flag
- GFLIB_DFlexRamp - this function calculates the ramp in the periodically called loop

For a proper use, it is recommended to initialize the algorithm by the GFLIB_DFlexRampInit function. The GFLIB_DFlexRampInit function initializes the internal state variable of the algorithm with a defined value and clears the reach flag. Call the init function when you want the ramp to be initialized.

To calculate the increment, use the GFLIB_DFlexRampCalcIncr function. Call this function at the point when you want to change the ramp output value. This function's inputs are the target value and the duration, and the ramp increments for motoring and generating saturation modes. The target value is the destination value that you want to get to. The duration is the time required to change the ramp output from the actual state to the target value. To be able to calculate the ramp increment, fill the control structure with the sample time, that means the period of the loop where the GFLIB_DFlexRamp funciton is called. The structure also contains a variable which determines the maximum value of the increment. It is necessary to set it up too. The equation for the increment calculation is as follows:

$$I = \frac{V_t - V_s}{T} \cdot T_s$$

Equation 17.

where:

- I is the increment
- V_t is the target value
- V_s is the state (actual) value (in the structure)
- T is the duration of the ramp (to reach the target value starting at the state value)
- T_s is the sample time, that means the period of the loop where the ramp algorithm is called (set in the structure)

If the increment is greater than the maximum increment (set in the structure), the increment uses the maximum increment value.

As soon as the new increment is calculated, you can call the GFLIB_DFlexRamp algorithm in the periodical control loop. If none of the stop flags is set, the function works as follows: The function adds the increment to the state value (from the previous step), which results in a new state. The new state is returned by the function. As time passes, the algorithm is approaching the target value. If the new state trespasses the target value that new state is limited to, the target value and the reach flag are set. The functionality of the implemented algorithm is shown in the following figure:

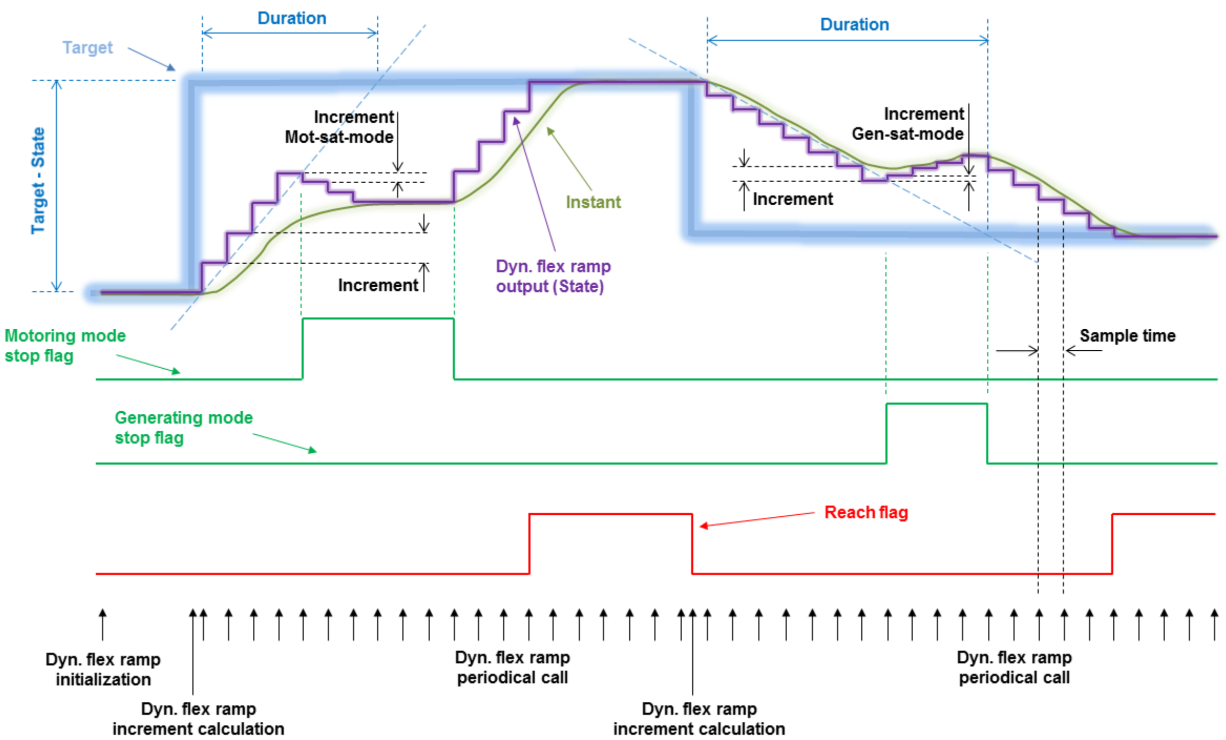

Figure 2-14. GFLIB_DFlexRamp functionality

If the motoring mode stop flag is set and the absolute value of the target value is greater than the absolute value of the state value, the function uses the increment for the motoring saturation mode to return to the instant value. Use case: when the application is in the saturation mode and cannot supply more power to increase the speed, then a saturation (motoring mode) flag is generated. To get out of the saturation, the ramp output value is being reduced.

If the generating mode stop flag is set and the absolute value of the target value is lower than the absolute value of state value, the funcion uses the increment for the generating saturation mode to return to the instant value. Use case: when the application is braking a motor and voltage increases on the DC-bus capacitor, then a saturation (generating mode) flag is generated. To avoid trespassing the DC-bus safe voltage limit, the speed requirement is increasing to disipate the energy of the capacitor.

2.20.1 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 input parameters are the fractional and accumulator types.

The available versions of the GFLIB_RampInit functions are shown in the following table:

Table 2-24. Init function versions

Function name	Input type	Parameters	Result type	Description
GFLIB_FlexRampInit_F16	frac16_t	GFLIB_DFLEXRAMP_T_F32 *	void	Input argument is a 16-bit fractional value that represents the initialization value. The parameters' structure is pointed to by a pointer. The input data value is in the range <-1 ; 1).

The available versions of the GFLIB_DFlexRamp functions are shown in the following table:

Table 2-25. Increment calculation function versions

Function name	Input type				Parameters	Result type
	Target	Duration	Incr. sat-mot	Incr. sat-gen		
GFLIB_DFlexRampCalcIncr_F16	frac16_t	acc32_t	frac32_t	frac32_t	GFLIB_DFLEXRAMP_T_F32 *	void
	Input arguments are 16-bit fractional values in the range <-1 ; 1) that represents the target output value and a 32-bit accumulator value in the range <0 ; 65536.0) that represents the duration (in seconds) of the ramp to reach the target value. The other two arguments are increments for the saturation mode when in the motoring and generating modes. The parameters' structure is pointed to by a pointer.					

Table 2-26. Function versions

Function name	Input type			Parameters	Result type
	Instant	Stop flag-mot	Stop flag-gen		
GFLIB_DFlexRamp_F16	frac16_t	bool_t *	bool_t *	GFLIB_DFLEXRAMP_T_F32 *	frac16_t
	Input argument is a 16-bit fractional value in the range <-1 ; 1) that represents the measured instant value. The stop flags are pointers to the bool_t types. The parameters' structure is pointed to by a pointer. The function returns a 16-bit fractional value, which represents the actual ramp output value. The output data value is in the range <-1 ; 1).				

2.20.2 GFLIB_DFLEXRAMP_T_F32

Variable name	Type	Description
f32State	frac32_t	Actual value. Controlled by the GFLIB_FlexRampInit_F16 and the GFLIB_FlexRamp_F16 algorithms.
f32Incr	frac32_t	Value of the dyn. flex ramp increment. Controlled by the GFLIB_FlexRampCalcIncr_F16 algorithm.
f32IncrSatMot	frac32_t	Value of the dyn. flex ramp increment when in the motoring saturation mode. Controlled by the GFLIB_DFlexRampCalcIncr_F16 algorithms.
f32IncrSatGen	frac32_t	Value of the dyn. flex ramp increment when in the generating saturation mode. Controlled by the GFLIB_DFlexRampCalcIncr_F16 algorithms.
f32Target	frac32_t	The target value of the flex ramp algorithm. Controlled by the GFLIB_DFlexRampCalcIncr_F16 algorithm.
f32Ts	frac32_t	The sample time, that means the period of the loop where the GFLIB_DFlexRamp_F16 algorithm is periodically called. The data value (in seconds) is in the range <0 ; 1). Set by the user.
f32IncrMax	frac32_t	The maximum value of the flex ramp increment. The data value is in the range <-1 ; 1). Set by the user.
bReachFlag	bool_t	Reach flag. This flag is controlled by the GFLIB_DFlexRamp_F16 algorithm. It is cleared by the GFLIB_DFlexRampInit_F16 and GFLIB_DFlexRampCalcIncr_F16 algorithms.

2.20.3 Declaration

The available GFLIB_DFlexRampInit functions have the following declarations:

```
void GFLIB_DFlexRampInit_F16(frac16_t f16InitVal, GFLIB_DFLEXRAMP_T_F16 *psParam)
```

The available GFLIB_DFlexRampCalcIncr functions have the following declarations:

```
void GFLIB_DFlexRampCalcIncr_F16(frac16_t f16Target, acc32_t a32Duration, frac32_t
f32IncrSatMot, frac32_t f32IncrSatGen, GFLIB_DFLEXRAMP_T_F16 *psParam)
```

The available GFLIB_DFlexRamp functions have the following declarations:

```
frac16_t GFLIB_DFlexRamp_F16(frac16_t f16Instant, const bool_t *pbStopFlagMot, const bool_t
*pbStopFlagGen, GFLIB_DFLEXRAMP_T_F16 *psParam)
```

2.20.4 Function use

The use of the GFLIB_DFlexRampInit, GFLIB_DFlexRampCalcIncr and GFLIB_DFlexRamp functions is shown in the following example:

```
#include "gflib.h"

static frac16_t f16InitVal;
static GFLIB_DFLEXRAMP_T_F16 sDFlexRamp;
static frac16_t f16Target, f16RampResult, f16Instant;
static acc32_t a32RampDuration;
static frac32_t f32IncrSatMotMode, f32IncrSatGenMode;
static bool_t bSatMot, bSatGen;

void Isr(void);

void main(void)
{
   /* Control loop period is 0.002 s; maximum increment value is 0.15 */
   sDFlexRamp.f32Ts = FRAC32(0.002);
   sDFlexRamp.f32IncrMax = FRAC32(0.15);

   /* Initial value to 0 */
   f16InitVal = FRAC16(0.0);

   /* Dyn. flex ramp initialization */
   GFLIB_FlexRampInit_F16(f16InitVal, &sDFlexRamp);

   /* Target value is 0.7 in duration of 5.3 s */
   f16Target = FRAC16(0.7);
   a32RampDuration = ACC32(5.3);;

   /* Saturation increments */
   f32IncrSatMotMode = FRAC32(0.000015);
   f32IncrSatGenMode = FRAC32(0.00002);

   /* Saturation flags init */
   bSatMot = FALSE;
   bSatGen = FALSE;

   /* Dyn. flex ramp increment calculation */
   GFLIB_DFlexRampCalcIncr_F16(f16Target, a32RampDuration, f32IncrSatMotMode,
f32IncrSatGenMode, &sDFlexRamp);
}

/* periodically called control loop with a period of 2 ms */
void Isr()
{
   f16RampResult = GFLIB_DFlexRamp_F16(f16Instant, &bSatMot, &bSatGen, &sDFlexRamp);
}
```

2.21 GFLIB_CtrlPIpAW

The GFLIB_CtrlPIpAW function calculates the parallel form of the Proportional-Integral (PI) controller with implemented integral anti-windup functionality.

The PI controller attempts to correct the error between the measured process variable and the desired set-point by calculating a corrective action that can adjust the process accordingly. The GFLIB_CtrlPIpAW function calculates the PI algorithm according to the equations below. The PI algorithm is implemented in the parallel (non-interacting) form, allowing the user to define the P and I parameters independently and without interaction. The controller output is limited and the limit values (upper limit and lower limit) are defined by the user.

The PI controller algorithm also returns a limitation flag, which indicates that the controller's output is at the limit. If the PI controller output reaches the upper or lower limit, then the limit flag is set to 1, otherwise it is 0 (integer values).

An anti-windup strategy is implemented by limiting the integral portion. The integral state is limited by the controller limits in the same way as the controller output. The integration can be stopped by a flag that is pointed to by the function's API.

The PI algorithm in the continuous time domain can be expressed as follows:

$$u(t) = e(t) \cdot K_P + K_I \int_0^t e(t) dt$$

Equation 18.

where:

- u(t) is the controller output in the continuous time domain
- e(t) is the input error in the continuous time domain
- K_P is the proportional gain
- K_I is the integral gain

Equation 18 on page 68 can be expressed using the Laplace transformation as follows:

$$H(s) = \frac{U(s)}{E(s)} = K_P + K_I \frac{1}{s}$$

Equation 19.

The proportional part (u_P) of Equation 18 on page 68 is transformed into the discrete time domain as follows:

$$u_P(k) = K_P \cdot e(k)$$

Equation 20.

where:

- $u_P(k)$ is the proportional action in the actual step
- $e(k)$ is the error in the actual step
- K_P is the proportional gain coefficient

Equation 20 on page 69 can be used in the fractional arithmetic as follows:

$$u_{Psc}(k) \cdot u_{max} = K_P \cdot e_{sc}(k) \cdot e_{max}$$

Equation 21.

where:

- u_{max} is the action output scale
- $u_{Psc}(k)$ is the scaled proportional action in the actual step
- e_{max} is the error input scale
- $e_{sc}(k)$ is the scale error in the actual step

Transforming the integral part (u_I) of Equation 18 on page 68 into a discrete time domain using the bi-linear method, also known as the trapezoidal approximation, is as follows:

$$u_I(k) = u_I(k-1) + e(k) \cdot \frac{K_I T_s}{2} + e(k-1)\frac{K_I T_s}{2}$$

Equation 22.

where:

- $u_I(k)$ is the integral action in the actual step
- $u_I(k - 1)$ is the integral action from the previous step
- $e(k)$ is the error in the actual step
- $e(k - 1)$ is the error in the previous step
- T_s is the sampling period of the system
- K_I is the integral gain coefficient

Equation 22 on page 69 can be used in the fractional arithmetic as follows:

$$u_{Isc}(k) \cdot u_{max} = u_{Isc}(k-1) \cdot u_{max} + K_I T_s \cdot \frac{e_{sc}(k) + e_{sc}(k-1)}{2} \cdot e_{max}$$

Equation 23.

where:

- u_{max} is the action output scale
- $u_{Isc}(k)$ is the scaled integral action in the actual step

- $u_{Isc}(k - 1)$ is the scaled integral action from the previous step
- e_{max} is the error input scale
- $e_{sc}(k)$ is the scaled error in the actual step
- $e_{sc}(k - 1)$ is the scaled error in the previous step

The output signal limitation is implemented in this controller. The actual output u(k) is bounded not to exceed the given limit values UpperLimit and LowerLimit. This is due to either the bounded power of the actuator or due to the physical constraints of the plant.

$$u(k) = \begin{cases} UpperLimit & u(k) \geq UpperLimit \\ LowerLimit & u(k) \leq LowerLimit \\ u(k) & else \end{cases}$$

Equation 24.

The bounds are described by a limitation element, as shown in Equation 24 on page 70. When the bounds are exceeded, the nonlinear saturation characteristic will take effect and influence the dynamic behavior. The described limitation is implemented on the integral part accumulator (limitation during the calculation) and on the overall controller output. Therefore, if the limitation occurs, the controller output is clipped to its bounds, and the wind-up occurrence of the accumulator portion is avoided by saturating the actual sum.

For a proper use of this function, it is recommended to initialize the function data by the GFLIB_CtrlPIpAWInit functions, before using the GFLIB_CtrlPIpAW function. You must call this function when you want the PI controller to be initialized.

2.21.1 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 GFLIB_CtrlPIpAWInit function are shown in the following table:

Table 2-27. Init function versions

Function name	Input type	Parameters	Result type	Description
GFLIB_CtrlPIpAWInit_F16	frac16_t	GFLIB_CTRL_PI_P_AW_T_A32 *	void	The inputs are a 16-bit fractional initial value and a pointer to the controller's parameters structure.

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

Table 2-28. Function versions

Function name	Input type		Parameters	Result type
	Error	Stop flag		
GFLIB_CtrlPIpAW_F16	frac16_t	bool_t *	GFLIB_CTRL_PI_P_AW_T_A32 *	frac16_t
	The error input is a 16-bit fractional value within the range <-1 ; 1). The integration of the PI controller is suspended if the stop flag is set. When it is cleared, the integration continues. The parameters are pointed to by an input pointer. The function returns a 16-bit fractional value in the range <f16LowerLim ; f16UpperLim>.			

2.21.2 GFLIB_CTRL_PI_P_AW_T_A32

Variable name	Input type	Description
a32PGain	acc32_t	Proportional gain is set up according to Equation 21 on page 69 as follows: $K_P \cdot \frac{e_{max}}{u_{max}}$ The parameter is a 32-bit accumulator type within the range <0 ; 65536.0). Set by the user.
a32IGain	acc32_t	Integral gain is set up according to Equation 23 on page 69 as follows: $K_I T_s \cdot \frac{e_{max}}{u_{max}}$ The parameter is a 32-bit accumulator type within the range <0 ; 65536.0). Set by the user.
f32IAccK_1	frac32_t	State variable of the internal accumulator (integrator). Controlled by the algorithm.
f16InErrK_1	frac16_t	Input error at the step k - 1. Controlled by the algorithm.
f16UpperLim	frac16_t	Upper limit of the controller's output and the internal accumulator (integrator). This parameter must be greater than f16LowerLim. Set by the user.
f16LowerLim	frac16_t	Lower limit of the controller's output and the internal accumulator (integrator). This parameter must be lower than f16UpperLim. Set by the user.
bLimFlag	bool_t	Limitation flag, which identifies that the controller's output reached the limits. 1 - the limit is reached; 0 - the output is within the limits. Controlled by the application.

2.21.3 Declaration

The available GFLIB_CtrlPIpAWInit functions have the following declarations:

```
void GFLIB_CtrlPIpAWInit_F16(frac16_t f16InitVal, GFLIB_CTRL_PI_P_AW_T_A32 *psParam)
```

The available GFLIB_CtrlPIpAW functions have the following declarations:

```
frac16_t GFLIB_CtrlPIpAW_F16(frac16_t f16InErr, const bool_t *pbStopIntegFlag,
GFLIB_CTRL_PI_P_AW_T_A32 *psParam)
```

2.21.4 Function use

The use of the GFLIB_CtrlPIpAWInit and GFLIB_CtrlPIpAW functions is shown in the following example:

```
#include "gflib.h"

static frac16_t f16Result, f16InitVal, f16InErr;
static bool_t bStopIntegFlag;
static GFLIB_CTRL_PI_P_AW_T_A32 sParam;

void Isr(void);

void main(void)
{
    f16InErr = FRAC16(-0.4);
    sParam.a32PGain = ACC32(0.1);
    sParam.a32IGain = ACC32(0.2);
    sParam.f16UpperLim = FRAC16(0.9);
    sParam.f16LowerLim = FRAC16(-0.9);
    bStopIntegFlag = FALSE;

    f16InitVal = FRAC16(0.0);

    GFLIB_CtrlPIpAWInit_F16(f16InitVal, &sParam);
}

/* periodically called function */
void Isr()
{
   f16Result = GFLIB_CtrlPIpAW_F16(f16InErr, &bStopIntegFlag, &sParam);
}
```

2.22 GFLIB_CtrlPIDpAW

The GFLIB_CtrlPIDpAW function calculates the parallel form of the Proportional-Integral-Derivative (PID) controller with implemented integral anti-windup functionality.

The PID controller attempts to correct the error between the measured process variable and the desired set-point by calculating a corrective action that can adjust the process accordingly. The GFLIB_CtrlPIDpAW function calculates the PID algorithm according to the equations below. The PID algorithm is implemented in the parallel (non-interacting) form, allowing the user to define the P, I, and D parameters independently and without interaction. The controller output is limited, and the limit values (upper limit and lower limit) are defined by the user.

The algorithm has two error inputs: one for the P and I calculation, and the other for the D calculation. This allows the user to apply different filters on both inputs.

The PID controller algorithm also returns a limitation flag, which indicates that the controller's output is at the limit. If the PID controller output reaches the upper or lower limit, then the limit flag is set to 1, otherwise it is 0 (integer values).

An anti-windup strategy is implemented by limiting the integral portion. The integral state is limited by the controller limits in the same way as the controller output. The integration can be stopped by a flag, which is pointed to by the function's API.

The PID algorithm in the continuous time domain can be expressed as follows:

$$u(t) = e(t) \cdot K_P + K_I \int_0^t e(t)dt + K_D \frac{d}{dt} e_D(t)$$

Equation 25.

where

- u(t) is the controller output in the continuous time domain
- e(t) is the input error for the proportional and integral calculation in the continuous time domain
- $e_D(t)$ is the input error for the derivative calculation in the continuous time domain
- K_P is the proportional gain
- K_I is the integral gain
- K_D is the derivative gain

Equation 25 on page 73 can be expressed using the Laplace transformation as follows:

$$H(s) = \frac{U(s)}{E(s)} = K_P + K_I \frac{1}{s} + K_D s$$

Equation 26.

The proportional part (u_P) of Equation 26 on page 73 is transformed into the discrete time domain as follows:

$$u_P(k) = K_P \cdot e(k)$$

Equation 27.

where:

- $u_P(k)$ is the proportional action in the actual step
- e(k) is the error in the actual step
- K_P is the proportional gain coefficient

Equation 27 on page 73 can be used in the fractional arithmetic as follows:

$$u_{Psc}(k) \cdot u_{max} = K_P \cdot e_{sc}(k) \cdot e_{max}$$

Equation 28.

where:

- u_{max} is the action output scale
- $u_{Psc}(k)$ is the scaled proportional action in the actual step
- e_{max} is the error input scale
- $e_{sc}(k)$ is the scale error in the actual step

Transforming the integral part (u_I) of Equation 26 on page 73 into a discrete time domain using the bi-linear method, also known as the trapezoidal approximation, is as follows:

$$u_I(k) = u_I(k-1) + e(k)\cdot\frac{K_I T_s}{2} + e(k-1)\frac{K_I T_s}{2}$$

Equation 29.

where:

- $u_I(k)$ is the integral action in the actual step
- $u_I(k - 1)$ is the integral action from the previous step
- $e(k)$ is the error in the actual step
- $e(k - 1)$ is the error in the previous step
- T_s is the sampling period of the system
- K_I is the integral gain coefficient

Equation 29 on page 74 can be used in the fractional arithmetic as follows:

$$u_{Isc}(k)\cdot u_{max} = u_{Isc}(k-1)\cdot u_{max} + K_I T_s\cdot\frac{e_{sc}(k) + e_{sc}(k-1)}{2}\cdot e_{max}$$

Equation 30.

where:

- u_{max} is the action output scale
- $u_{Isc}(k)$ is the scaled integral action in the actual step
- $u_{Isc}(k - 1)$ is the scaled integral action from the previous step
- e_{max} is the error input scale
- $e_{sc}(k)$ is the scaled error in the actual step
- $e_{sc}(k - 1)$ is the scaled error in the previous step

The derivative part (u_D) of Equation 25 on page 73 is transformed into the discrete time domain as follows:

$$u_D(k) = \frac{K_D}{T_s}\cdot[e_D(k) - e_D(k-1)]$$

Equation 31.

where:

- $u_D(k)$ is the proportional action in the actual step
- $e_D(k)$ is the error used for the derivative input in the actual step
- $e_D(k - 1)$ is the error used for the derivative input in the previous step
- K_D is the proportional gain coefficient

Equation 27 on page 73 can be used in the fractional arithmetic as follows:

$$u_{Dsc}(k) \cdot u_{max} = \frac{K_D}{T_s} \cdot [e_{Dsc}(k) - e_{Dsc}(k-1)] \cdot e_{max}$$

Equation 32.

where:

- u_{max} is the action output scale
- $u_{Dsc}(k)$ is the scaled derivative action in the actual step
- e_{max} is the error input scale
- $e_{Dsc}(k)$ is the scaled error for the derivative input in the actual step
- $e_{Dsc}(k - 1)$ is the scaled error for the derivative input in the previous step

The output signal limitation is implemented in this controller. The actual output u(k) is bounded to not exceed the given limit values - UpperLimit and LowerLimit. This is due to either the bounded power of the actuator, or due to the physical constraints of the plant.

$$u(k) = \begin{cases} UpperLimit & u(k) \geq UpperLimit \\ LowerLimit & u(k) \leq LowerLimit \\ u(k) & else \end{cases}$$

Equation 33.

The bounds are described by a limitation element, as shown in Equation 33 on page 75. When the bounds are exceeded, the non-linear saturation characteristic will take effect, and influence the dynamic behavior. The described limitation is implemented in the integral part accumulator (limitation during the calculation) and in the overall controller output. Therefore, if the limitation occurs, the controller output is clipped to its bounds, and the wind-up occurrence of the accumulator portion is avoided by saturating the actual sum.

For a proper use of this function, it is recommended to initialize the function data by the GFLIB_CtrlPIDpAWInit functions, before using the GFLIB_CtrlPIDpAW function. You must call this function, when you want the PID controller to be initialized.

2.22.1 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 GFLIB_CtrlPIDpAWInit function are shown in the following table:

Table 2-29. Init function versions

Function name	Input type	Parameters	Result type	Description
GFLIB_CtrlPIDpAWInit_F16	frac16_t	GFLIB_CTRL_PID_P_AW_T_A32 *	void	The inputs are a 16-bit fractional initial value and a pointer to the controller's parameters structure.

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

Table 2-30. Function versions

Function name	Input type			Parameters	Result type
	PI error	D error	Stop flag		
GFLIB_CtrlPIDpAW_F16	frac16_t	frac16_t	bool_t *	GFLIB_CTRL_PID_P_AW_T_A32 *	frac16_t
	The error inputs are 16-bit fractional values within the range <-1 ; 1). The integration of the PID controller is suspended if the stop flag is set. When it is cleared, the integration continues. The parameters are pointed to by an input pointer. The function returns a 16-bit fractional value in the range <f16LowerLim ; f16UpperLim>.				

2.22.2 GFLIB_CTRL_PID_P_AW_T_A32

Variable name	Input type	Description
a32PGain	acc32_t	Proportional gain is set up according to Equation 28 on page 73 as follows: $K_P \cdot \frac{e_{max}}{u_{max}}$ The parameter is a 32-bit accumulator type within the range <0 ; 65536.0). Set by the user.
a32IGain	acc32_t	Integral gain is set up according to Equation 30 on page 74 as follows: $K_I T_s \cdot \frac{e_{max}}{u_{max}}$ The parameter is a 32-bit accumulator type within the range <0 ; 65536.0). Set by the user.
a32DGain	acc32_t	Derivative gain is set up according to Equation 32 on page 75 as follows: $\frac{K_D}{T_s} \cdot \frac{e_{max}}{u_{max}}$ The parameter is a 32-bit accumulator type within the range <0 ; 65536.0). Set by the user.

Table continues on the next page...

Variable name	Input type	Description
f32IAccK_1	frac32_t	State variable of the internal accumulator (integrator). Controlled by the algorithm.
f16InErrK_1	frac16_t	Input error in the step k - 1. Controlled by the algorithm.
f16UpperLim	frac16_t	Upper limit of the controller's output and the internal accumulator (integrator). This parameter must be greater than f16LowerLim. Set by the user.
f16LowerLim	frac16_t	Lower limit of the controller's output and the internal accumulator (integrator). This parameter must be lower than f16UpperLim. Set by the user.
f16InErrDK_1	frac16_t	Input error for the derivative calculation in the step k - 1. Controlled by the algorithm.
bLimFlag	bool_t	Limitation flag, which identifies that the controller's output reached the limits. 1 - the limit is reached; 0 - the output is within the limits. Controlled by the application.

2.22.3 Declaration

The available GFLIB_CtrlPIDpAWInit functions have the following declarations:

```
void GFLIB_CtrlPIDpAWInit_F16(frac16_t f16InitVal, GFLIB_CTRL_PID_P_AW_T_A32 *psParam)
```

The available GFLIB_CtrlPIDpAW functions have the following declarations:

```
frac16_t GFLIB_CtrlPIDpAW_F16(frac16_t f16InErr, frac16_t f16InErrD, const bool_t
*pbStopIntegFlag, GFLIB_CTRL_PID_P_AW_T_A32 *psParam)
```

2.22.4 Function use

The use of the GFLIB_CtrlPIDpAWInit and GFLIB_CtrlPIDpAW functions is shown in the following example:

```
#include "gflib.h"

static frac16_t f16Result, f16InitVal, f16InErr, f16InErrD;
static bool_t bStopIntegFlag;
static GFLIB_CTRL_PID_P_AW_T_A32 sParam;

void Isr(void);

void main(void)
{
    f16InErr = FRAC16(-0.4);
    f16InErr = f16InErrD;
    sParam.a32PGain = ACC32(0.1);
    sParam.a32IGain = ACC32(0.2);
    sParam.a32DGain = ACC32(0.001);
    sParam.f16UpperLim = FRAC16(0.9);
    sParam.f16LowerLim = FRAC16(-0.9);
    bStopIntegFlag = FALSE;

    f16InitVal = FRAC16(0.0);

    GFLIB_CtrlPIDpAWInit_F16(f16InitVal, &sParam);
```

```
}

/* periodically called function */
void Isr()
{
   f16Result = GFLIB_CtrlPIDpAW_F16(f16InErr, f16InErrD, &bStopIntegFlag, &sParam);
}
```

Appendix A
Library types

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

Value	7	6	5	4	3	2	1	0
	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

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

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

	31	24	23	16	15	8	7	0
Value	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 DSP56800EXGFLIBUG
Revision 2, 10/2015