

---

# GFLIB User's Guide

ARM® Cortex® M0+

Document Number: CM0GFLIBUG  
Rev. 3, 04/2016

Review Draft





# Contents

Section number	Title	Page
<b>Chapter 1</b>		
<b>Library</b>		
1.1	Introduction.....	5
1.2	Library integration into project (Kinetis Design Studio) .....	7
1.3	Library integration into project (Keil $\mu$ Vision) .....	16
1.4	Library integration into project (IAR Embedded Workbench) .....	24
<b>Chapter 2</b>		
<b>Algorithms in detail</b>		
2.1	GFLIB_Sin.....	33
2.2	GFLIB_Cos.....	35
2.3	GFLIB_Atan.....	36
2.4	GFLIB_AtanYX.....	38
2.5	GFLIB_Sqrt.....	40
2.6	GFLIB_Limit.....	42
2.7	GFLIB_LowerLimit.....	43
2.8	GFLIB_UpperLimit.....	44
2.9	GFLIB_VectorLimit1.....	45
2.10	GFLIB_Hyst.....	48
2.11	GFLIB_Lut1D.....	50
2.12	GFLIB_LutPer1D.....	52
2.13	GFLIB_Ramp.....	55
2.14	GFLIB_DRamp.....	58
2.15	GFLIB_FlexRamp.....	62
2.16	GFLIB_DFlexRamp.....	66
2.17	GFLIB_Integrator.....	71
2.18	GFLIB_CtrlPIpAW.....	74



# Chapter 1

## Library

### 1.1 Introduction

#### 1.1.1 Overview

This user's guide describes the General Functions Library (GFLIB) for the family of ARM Cortex M0+ core-based microcontrollers. 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** — $\langle -256.0 ; 256.0 - 2^{-7} \rangle$  with the minimum resolution of  $2^{-7}$
- **Fixed-point 32-bit accumulator** — $\langle -65536.0 ; 65536.0 - 2^{-15} \rangle$  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
<a href="#">frac16_t</a>	F16	s
<a href="#">frac32_t</a>	F32	l
<a href="#">acc32_t</a>	A32	a

### 1.1.4 Supported compilers

GFLIB for the ARM Cortex M0+ core is written in C language. The library is built and tested using the following compilers:

- Kinetis Design Studio
- IAR Embedded Workbench
- Keil  $\mu$ Vision

For the Kinetis Design Studio, the library is delivered in the *gflib.a* file.

For the IAR Embedded Workbench, the library is delivered in the *gflib.a* file.

For the Keil  $\mu$ Vision, 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 that round the result (the API contains Rnd) round to nearest (half up).

## 1.2 Library integration into project (Kinetis Design Studio)

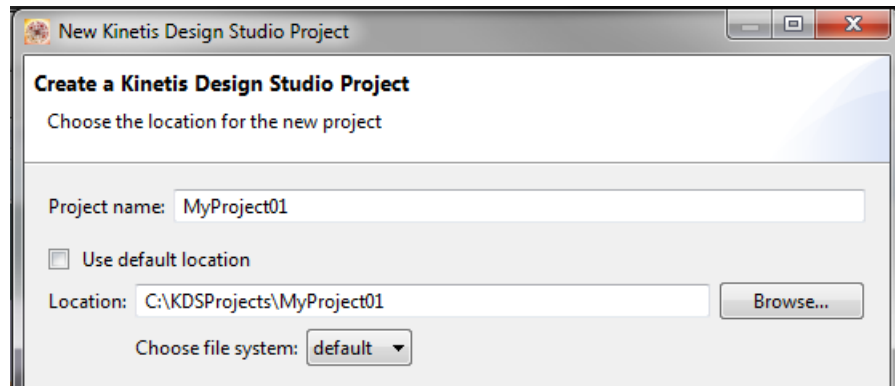
This section provides a step-by-step guide on how to quickly and easily include GFLIB into an empty project or any SDK example or demo application projects using Kinetis Design Studio. This example uses the default installation path (C:\NXP\RTCESL\CM0\_RTCESL\_4.3\_KDS). If you have a different installation path, use that path instead. If you want to use an existing SDK project (for example the hello\_world project) see [Memory-mapped divide and square root support](#) . If not, continue with the next section.

### 1.2.1 New project (without SDK)

This example uses the NXP MKV10Z32xxx7 MCU, and the default installation path (C:\NXP\RTCESL\CM0\_RTCESL\_4.3\_KDS) is supposed. To start working on an application, create a new project. If the project already exists and is opened, skip to the next section. Follow these steps to create a new project:

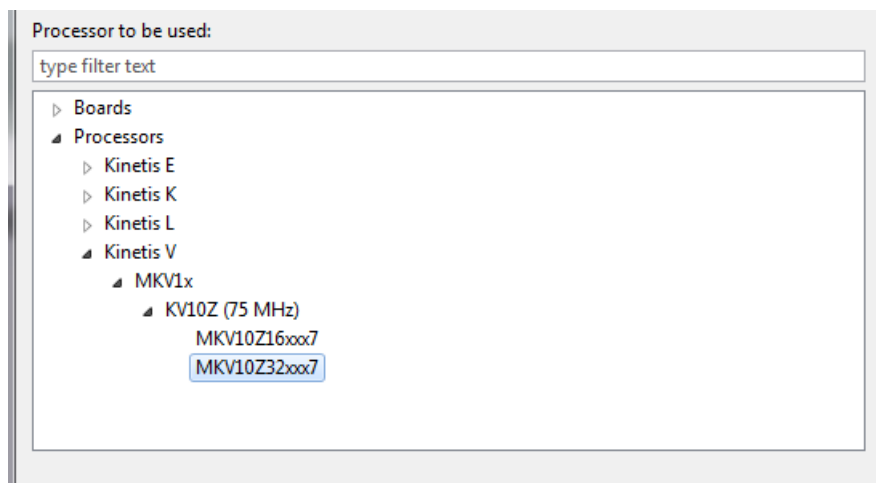
1. Launch Kinetis Design Studio.
2. Select File > New > Kinetis Design Studio Project so that the New Kinetis Design Studio Project dialog appears.
3. Type the name of the project, for example, MyProject01.

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:\KDSProjects\MyProject01) and click Next. See [Figure 1-1](#).



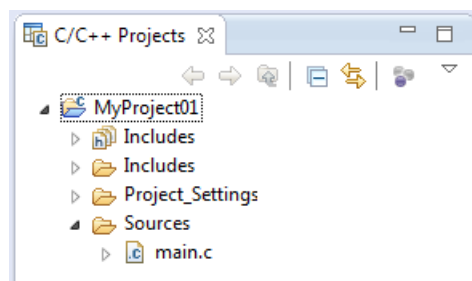
**Figure 1-1. Project name and location**

5. Expand the tree by clicking Processors, then Kinetis V > MKV1x > KV10Z, and click MKV10Z32xxx7. Click Finish. See [Figure 1-2](#).



**Figure 1-2. Processor selection**

6. The newly-created project is now visible in the left-hand part of Kinetis Design Studio. See [Figure 1-3](#).



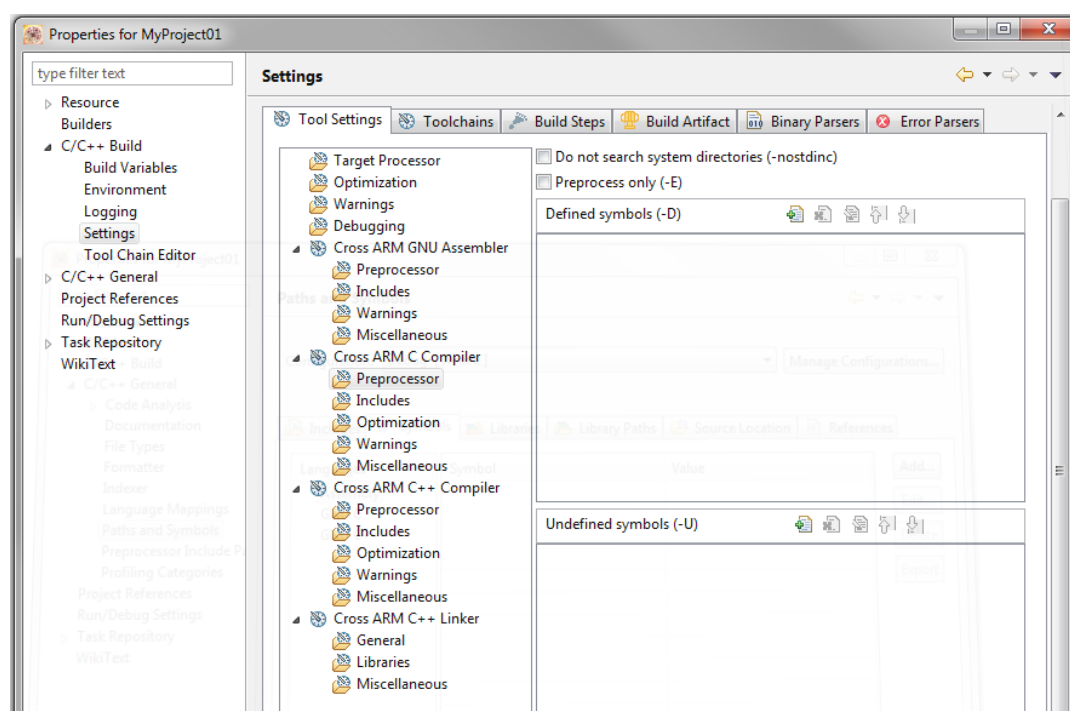
**Figure 1-3. Project folder**



## 1.2.2 Memory-mapped divide and square root support

Some Kinetis platforms contain a peripheral module dedicated for division and square root. This section shows how to turn the memory-mapped divide and square root (MMDVQS) support on and off.

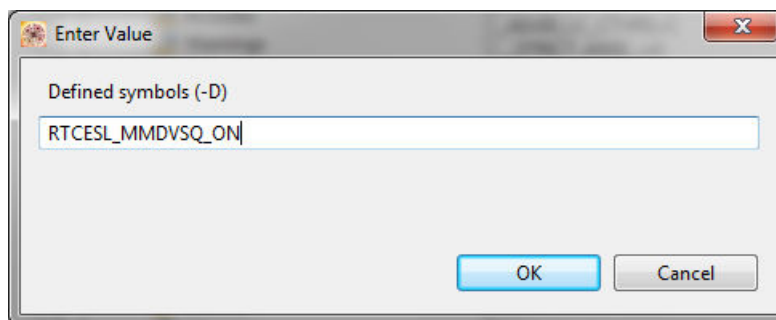
1. Right-click the MyProject01 or SDK project name node or in the left-hand part and click Properties, or select Project > Properties from the menu. A project properties dialog appears.
2. Expand the C/C++ Build node and select Settings. See [Figure 1-4](#).
3. In the right-hand part, under the Cross ARM C compiler node, click the Preprocessor node. See [Figure 1-4](#).



**Figure 1-4. Defined symbols**

4. In the right-hand part of the dialog, click the Add... icon located next to the Defined symbols (-D) title.
5. In the dialog that appears (see [Figure 1-5](#)), type the following:
  - RTCESL\_MMDVQS\_ON—to turn the hardware division and square root support on
  - RTCESL\_MMDVQS\_OFF—to turn the hardware division and square root support off

If neither of these two defines is defined, the hardware division and square root support is turned off by default.



**Figure 1-5. Symbol definition**

6. Click OK in the dialog.
7. Click OK in the main dialog.

See the device reference manual to verify whether the device contains the MMDVSQ module.

### 1.2.3 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 or SDK project name node in the left-hand part and click Properties, or select Project > Properties from the menu. A project properties dialog appears.
2. Expand the Resource node and click Linked Resources. See [Figure 1-6](#).

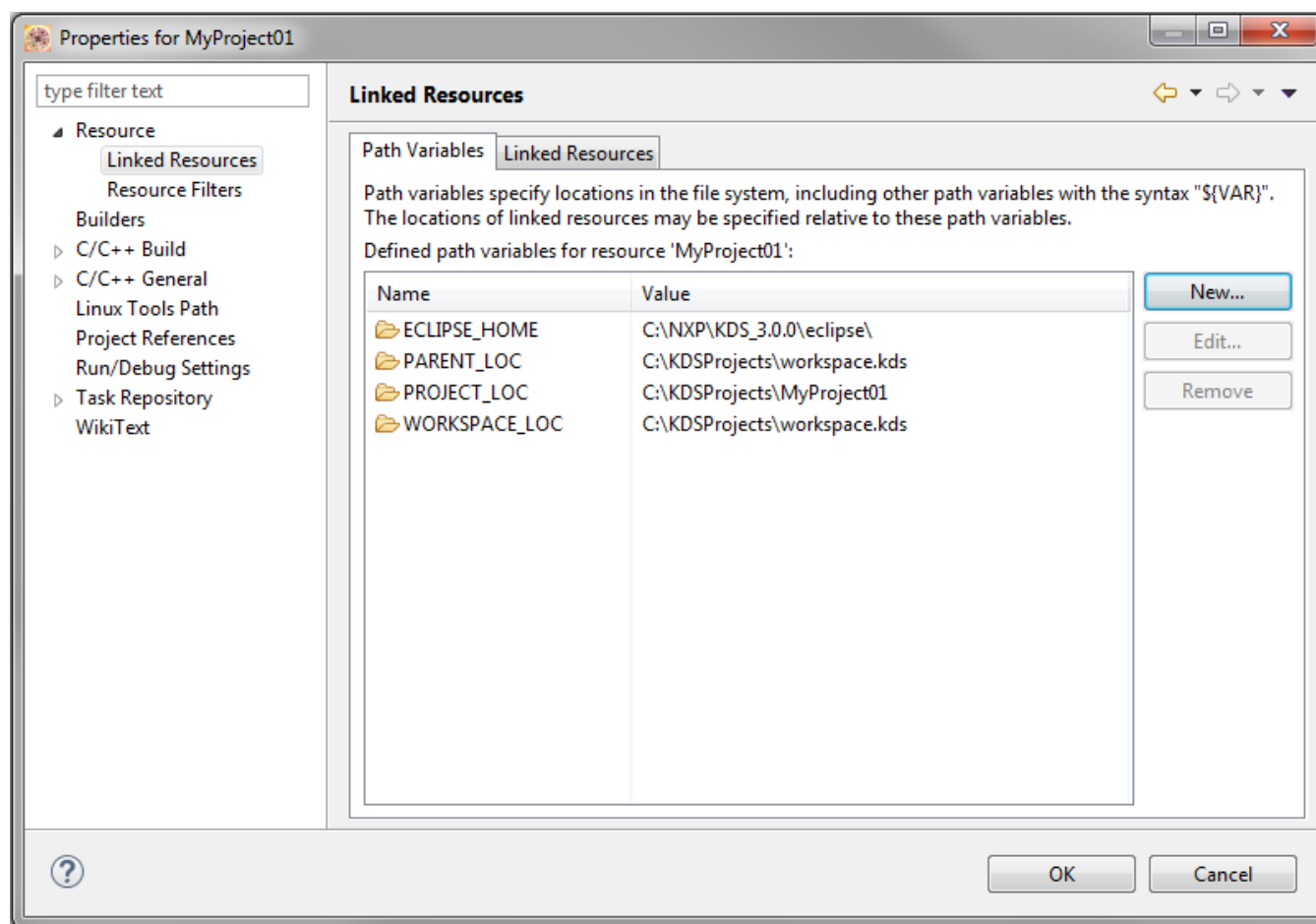


Figure 1-6. Project properties

3. Click the New... button in the right-hand side.
4. In the dialog that appears (see Figure 1-7), type this variable name into the Name box: RTCESL\_LOC.
5. Select the library parent folder by clicking Folder..., or just type the following path into the Location box: C:\NXP\RTCESL\CM0\_RTCESEL\_4.3\_KDS. Click OK.

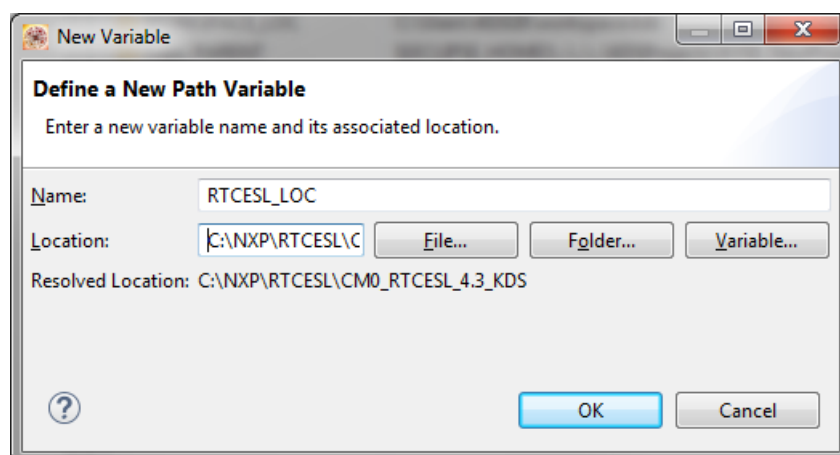


Figure 1-7. New variable

6. Create such variable for the environment. Expand the C/C++ Build node and click Environment.
7. Click the Add... button in the right-hand side.
8. In the dialog that appears (see [Figure 1-8](#)), type this variable name into the Name box: RTCESL\_LOC.
9. Type the library parent folder path into the Value box: C:\NXP\RTCESL\CM0\_RTCESL\_4.3\_KDS.
10. Tick the Add to all configurations box to use this variable in all configurations. See [Figure 1-8](#).
11. Click OK.
12. In the previous dialog, click OK.

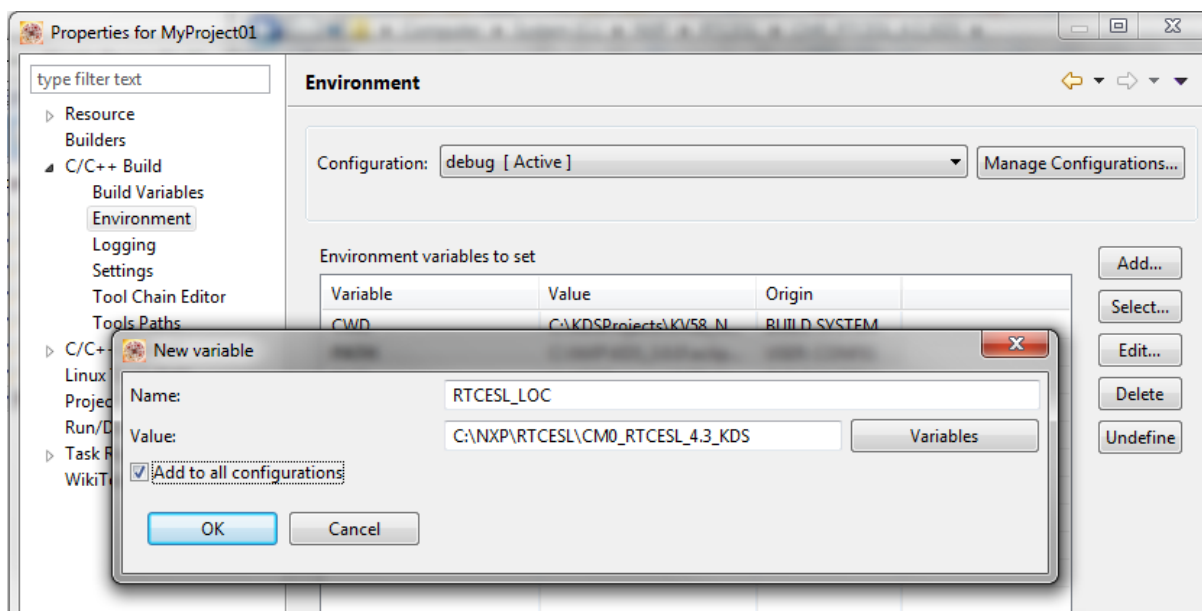


Figure 1-8. Environment variable

## 1.2.4 Library folder addition

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

1. Right-click the MyProject01 or SDK project name 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 option Link to alternate location (Linked Folder).
4. Click Variables..., select the RTCESL\_LOC variable in the dialog, click OK, and/or type the variable name into the box. See [Figure 1-9](#).
5. Click Finish, and you will see the library folder linked in the project. See [Figure 1-10](#).

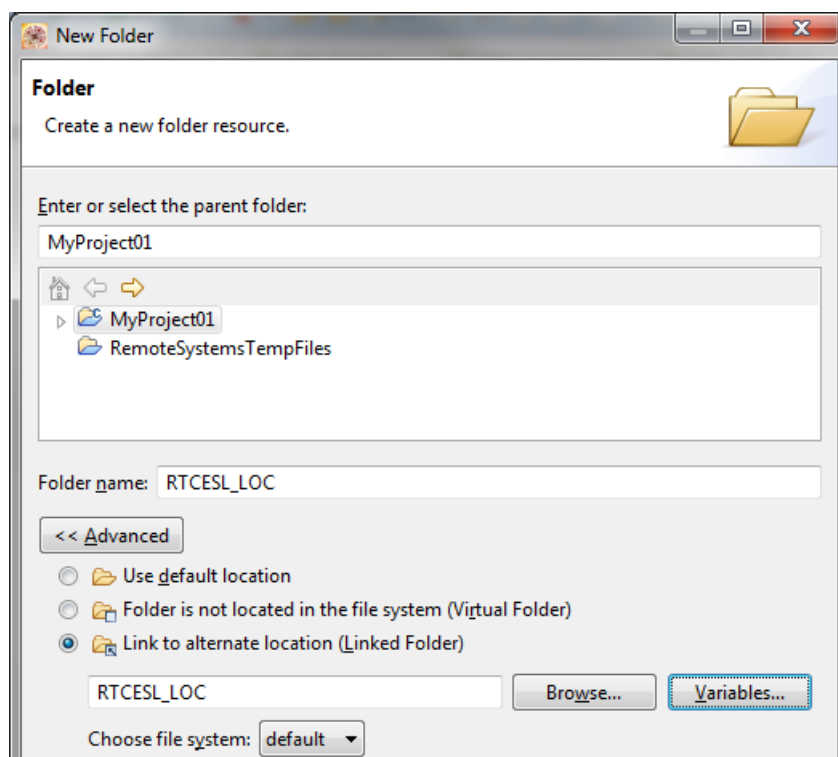


Figure 1-9. Folder link

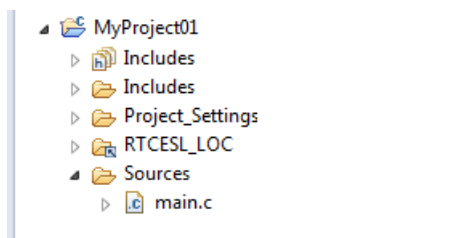


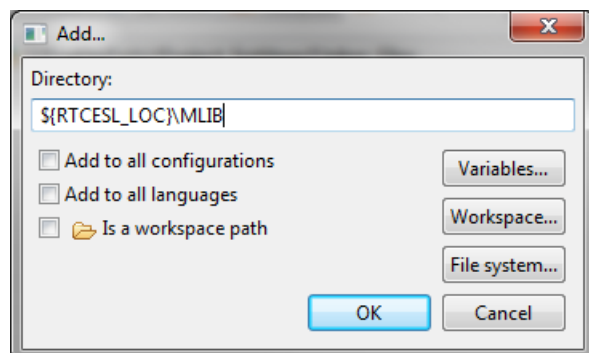
Figure 1-10. Projects libraries paths

## 1.2.5 Library path setup

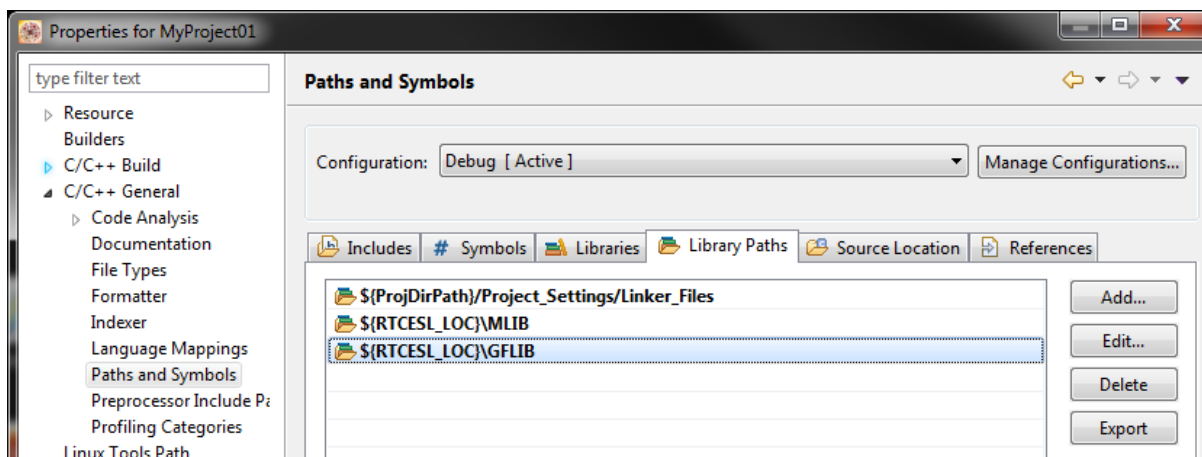
GFLIB requires MLIB to be included too. These steps show how to include all dependent modules:

1. Right-click the MyProject01 or SDK project name node in the left-hand part and click Properties, or select Project > Properties from the menu. A project properties dialog appears.
2. Expand the C/C++ General node, and click Paths and Symbols.
3. In the right-hand dialog, select the Library Paths tab. See [Figure 1-12](#).
4. Click the Add... button on the right, and a dialog appears.
5. Look for the RTCESL\_LOC variable by clicking Variables..., and then finish the path in the box by adding the following (see [Figure 1-11](#)): `${RTCESL_LOC}\MLIB`.

6. Click OK, and then click the Add... button.
7. Look for the RTCESL\_LOC variable by clicking Variables..., and then finish the path in the box by adding the following: \${RTCESL\_LOC}\GFLIB.
8. Click OK, and the paths will be visible in the list. See [Figure 1-12](#).



**Figure 1-11. Library path inclusion**



**Figure 1-12. Library paths**

9. After adding the library paths, add the library files. Click the Libraries tab. See [Figure 1-14](#).
10. Click the Add... button on the right, and a dialog appears.
11. Type the following into the File text box (see [Figure 1-13](#)): :mlib.a
12. Click OK, and then click the Add... button.
13. Type the following into the File text box: :gflib.a
14. Click OK, and you will see the libraries added in the list. See [Figure 1-14](#).

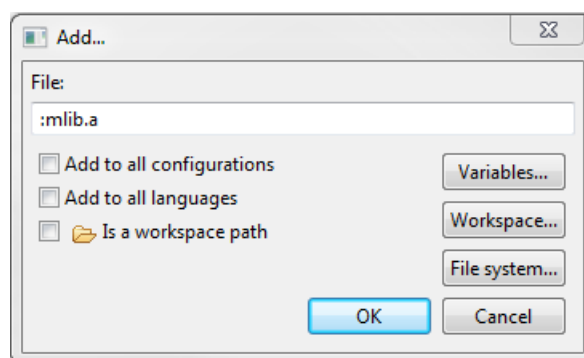


Figure 1-13. Library file inclusion

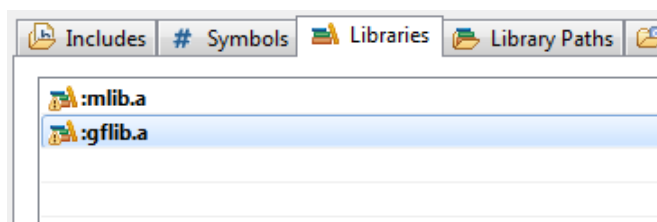


Figure 1-14. Libraries

15. In the right-hand dialog, select the Includes tab, and click GNU C in the Languages list. See [Figure 1-16](#).
16. Click the Add... button on the right, and a dialog appears. See [Figure 1-15](#).
17. Look for the RTCESL\_LOC variable by clicking Variables..., and then finish the path in the box to be: `${RTCESL_LOC}\MLIB\Include`
18. Click OK, and then click the Add... button.
19. Look for the RTCESL\_LOC variable by clicking Variables..., and then finish the path in the box to be: `${RTCESL_LOC}\GFLIB\Include`
20. Click OK, and you will see the paths added in the list. See [Figure 1-16](#). Click OK.

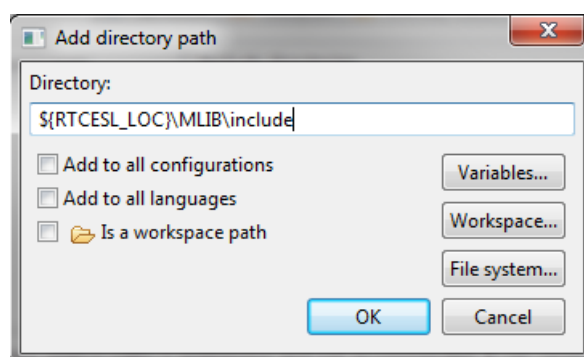


Figure 1-15. Library include path addition

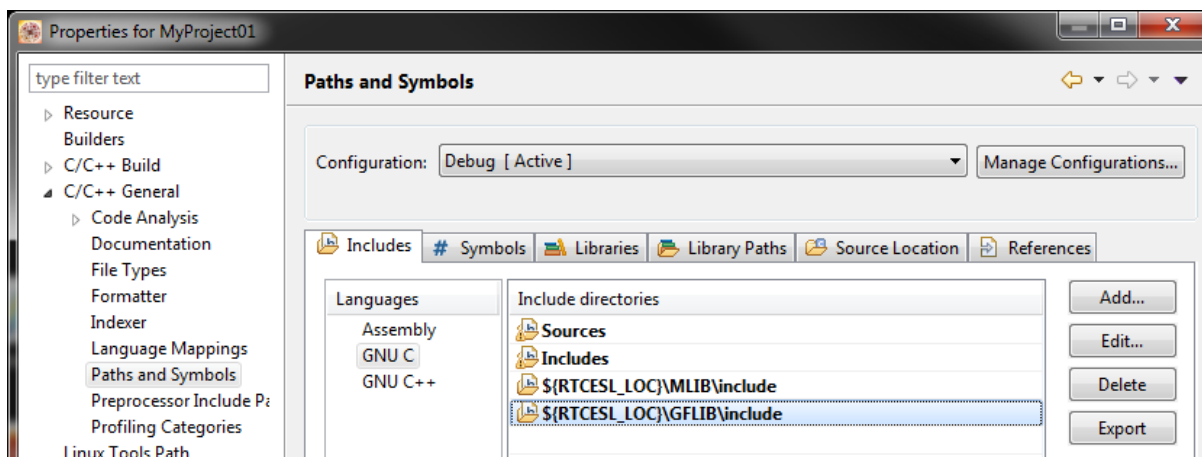


Figure 1-16. Compiler setting

Type 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 in the `#include` section:

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

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

## 1.3 Library integration into project (Keil µVision)

This section provides a step-by-step guide on how to quickly and easily include GFLIB into an empty project or any SDK example or demo application projects using Keil µVision. This example uses the default installation path (C:\NXP\RTCESL\CM0\_RTCESL\_4.3\_KEIL). If you have a different installation path, use that path instead. If any SDK project is intended to use (for example hello\_world project) go to [Memory-mapped divide and square root support](#) chapter otherwise read next chapter.

### 1.3.1 NXP pack installation for new project (without SDK)

This example uses the NXP MKV10Z32xxx7 part, and the default installation path (C:\NXP\RTCESL\CM0\_RTCESL\_4.3\_KEIL) is supposed. If the compiler has never been used to create any NXP MCU-based projects before, check whether the NXP MCU pack for the particular device is installed. Follow these steps:

1. Launch Keil µVision.
2. In the main menu, go to Project > Manage > Pack Installer....



3. In the left-hand dialog (under the Devices tab), expand the All Devices > Freescale (NXP) node.
4. Look for a line called "KVxx Series" and click it.
5. In the right-hand dialog (under the Packs tab), expand the Device Specific node.
6. Look for a node called "Keil::Kinetis\_KVxx\_DFP." If there are the Install or Update options, click the button to install/update the package. See [Figure 1-17](#).
7. When installed, the button has the "Up to date" title. Now close the Pack Installer.

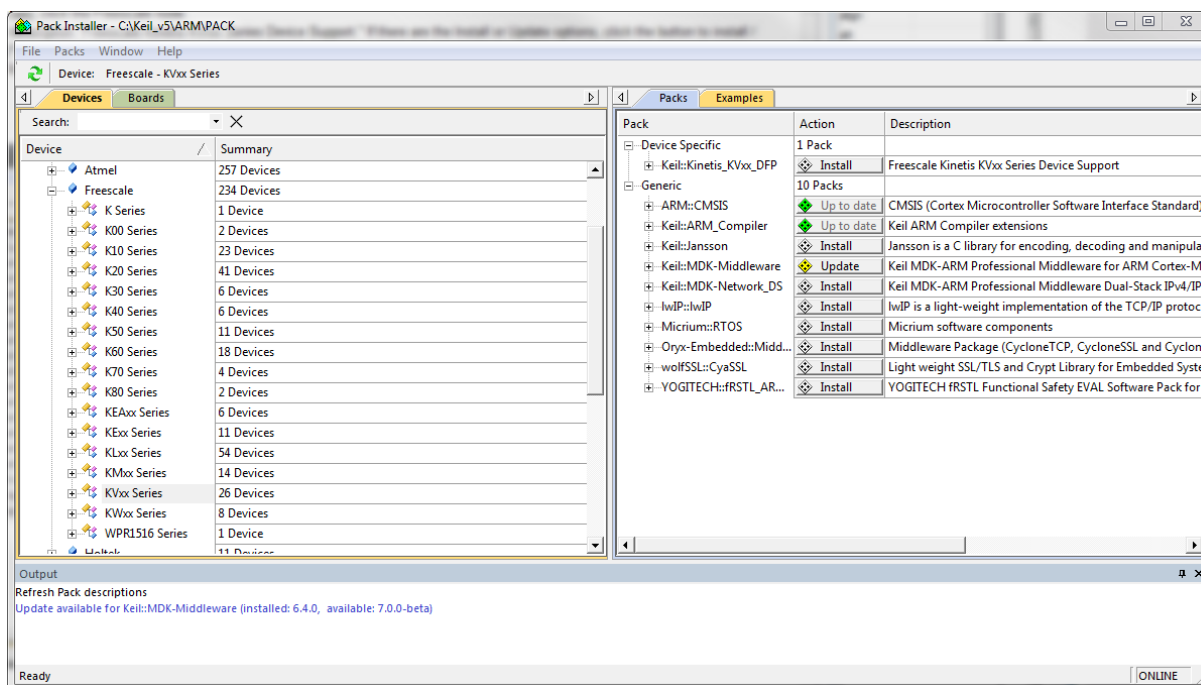
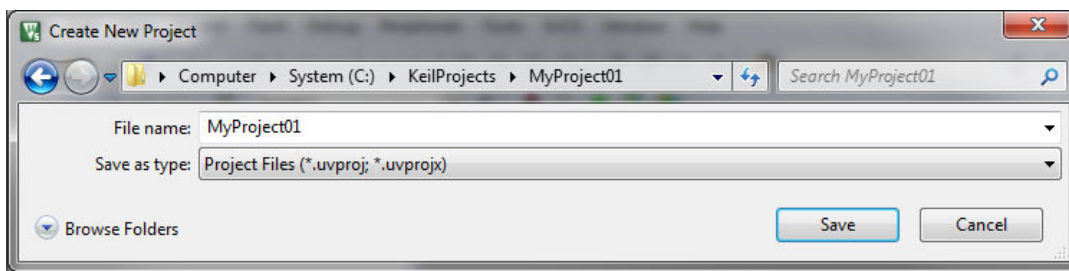


Figure 1-17. Pack Installer

### 1.3.2 New project (without SDK)

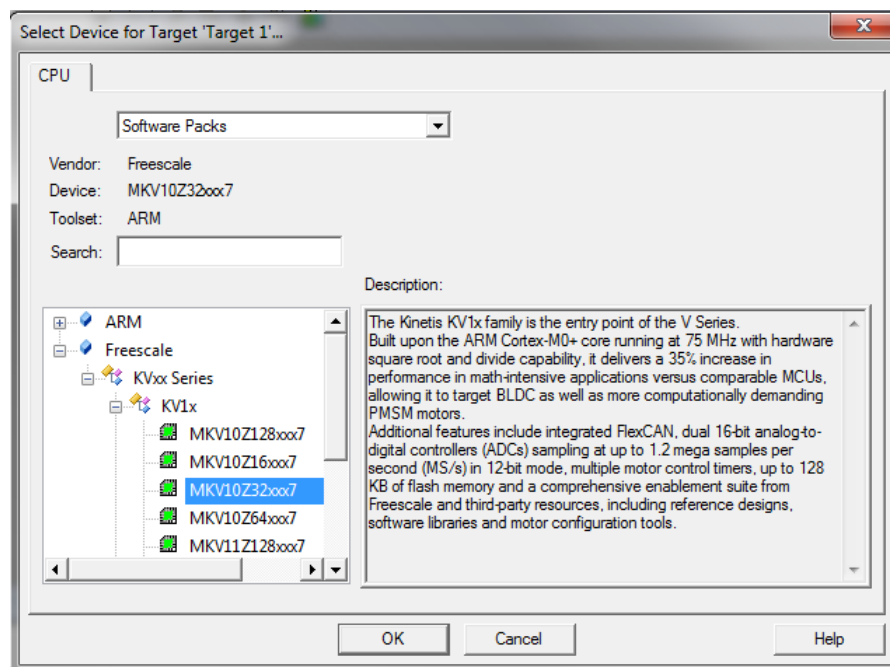
To start working on an application, create a new project. If the project already exists and is opened, skip to the next section. Follow these steps to create a new project:

1. Launch Keil  $\mu$ Vision.
2. In the main menu, select Project > New  $\mu$ Vision Project..., and the Create New Project dialog appears.
3. Navigate to the folder where you want to create the project, for example C:\KeilProjects\MyProject01. Type the name of the project, for example MyProject01. Click Save. See [Figure 1-18](#).



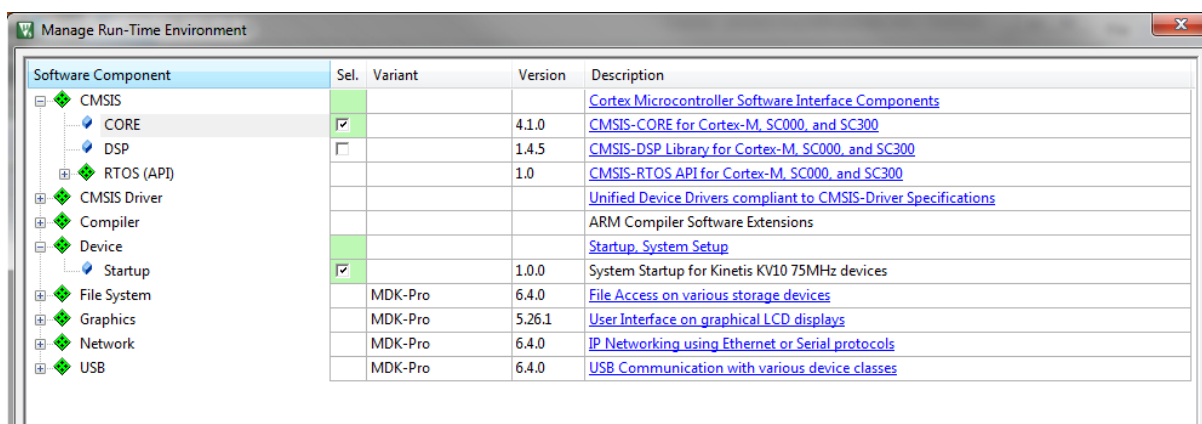
**Figure 1-18. Create New Project dialog**

4. In the next dialog, select the Software Packs in the very first box.
5. Type 'kv10' into the Search box, so that the device list is reduced to the KV10 devices.
6. Expand the KV10 node.
7. Click the MKV10Z32xxx7 node, and then click OK. See [Figure 1-19](#).



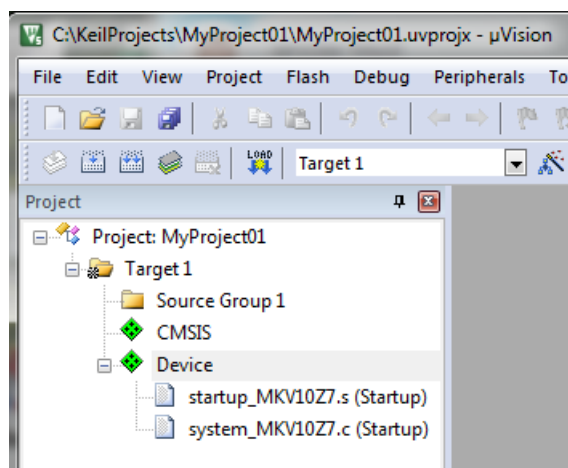
**Figure 1-19. Select Device dialog**

8. In the next dialog, expand the Device node, and tick the box next to the Startup node. See [Figure 1-20](#).
9. Expand the CMSIS node, and tick the box next to the CORE node.



**Figure 1-20. Manage Run-Time Environment dialog**

- Click OK, and a new project is created. The new project is now visible in the left-hand part of Keil  $\mu$ Vision. See [Figure 1-21](#).



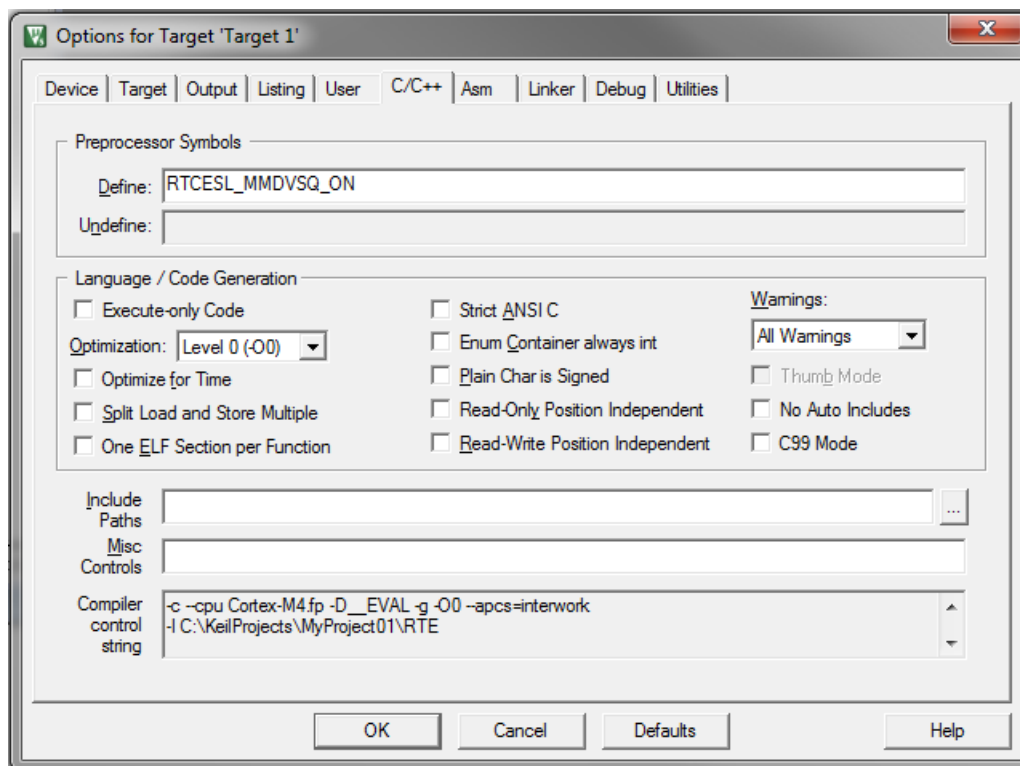
**Figure 1-21. Project**

### 1.3.3 Memory-mapped divide and square root support

Some Kinetis platforms contain a peripheral module dedicated for division and square root. This section shows how to turn the memory-mapped divide and square root (MMDVQS) support on and off.

- In the main menu, go to Project > Options for Target 'Target1'..., and a dialog appears.
- Select the C/C++ tab. See [Figure 1-22](#).
- In the Include Preprocessor Symbols text box, type the following:
  - RTCESL\_MMDVQS\_ON—to turn the hardware division and square root support on
  - RTCESL\_MMDVQS\_OFF—to turn the hardware division and square root support off

If neither of these two defines is defined, the hardware division and square root support is turned off by default.



**Figure 1-22. Preprocessor symbols**

4. Click OK in the main dialog.

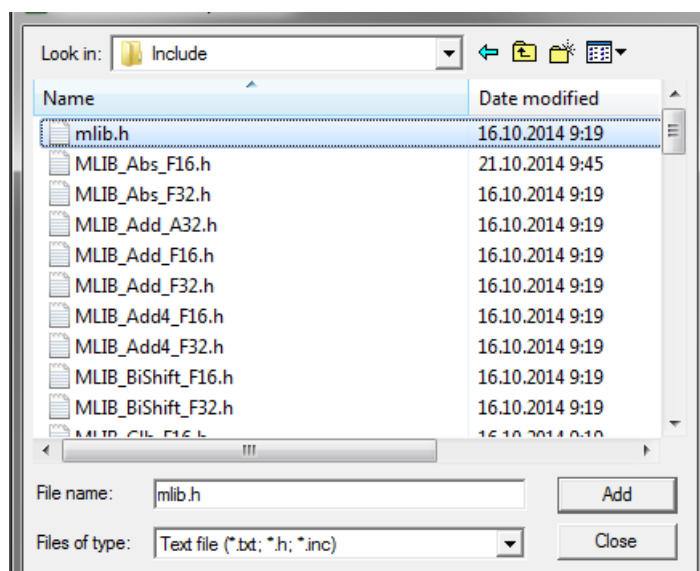
See the device reference manual to verify whether the device contains the MMDVSQ module.

### 1.3.4 Linking the files into the project

GFLIB requires MLIB to be included too. The following steps show how to include all dependent modules.

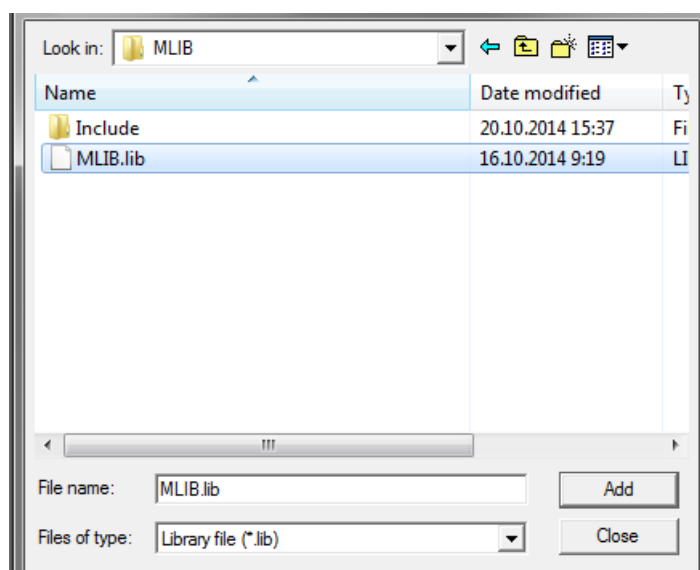
To include the library files in the project, create groups and add them.

1. Right-click the Target 1 node in the left-hand part of the Project tree, and select Add Group... from the menu. A new group with the name New Group is added.
2. Click the newly created group, and press F2 to rename it to RTCESL.
3. Right-click the RTCESL node, and select Add Existing Files to Group 'RTCESL'... from the menu.
4. Navigate into the library installation folder C:\NXP\RTCESL\CM0\_RTCESL\_4.3\_KEIL\MLIB\Include, and select the *mllib.h* file. If the file does not appear, set the Files of type filter to Text file. Click Add. See [Figure 1-23](#).



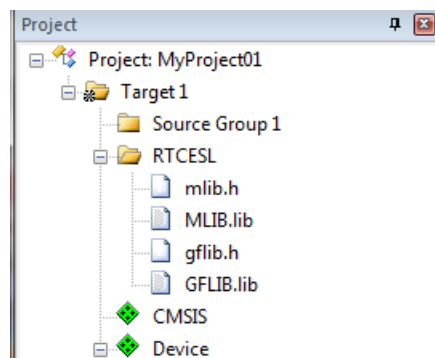
**Figure 1-23. Adding .h files dialog**

5. Navigate to the parent folder C:\NXP\RTCESL\CM0\_RTCESL\_4.3\_KEIL\MLIB, and select the *mlib.lib* file. If the file does not appear, set the Files of type filter to Library file. Click Add. See [Figure 1-24](#).



**Figure 1-24. Adding .lib files dialog**

6. Navigate into the library installation folder C:\NXP\RTCESL\CM0\_RTCESL\_4.3\_KEIL\GFLIB\Include, and select the *gflib.h* file. If the file does not appear, set the Files of type filter to Text file. Click Add.
7. Navigate to the parent folder C:\NXP\RTCESL\CM0\_RTCESL\_4.3\_KEIL\GFLIB, and select the *gflib.lib* file. If the file does not appear, set the Files of type filter to Library file. Click Add.
8. Now, all necessary files are in the project tree; see [Figure 1-25](#). Click Close.

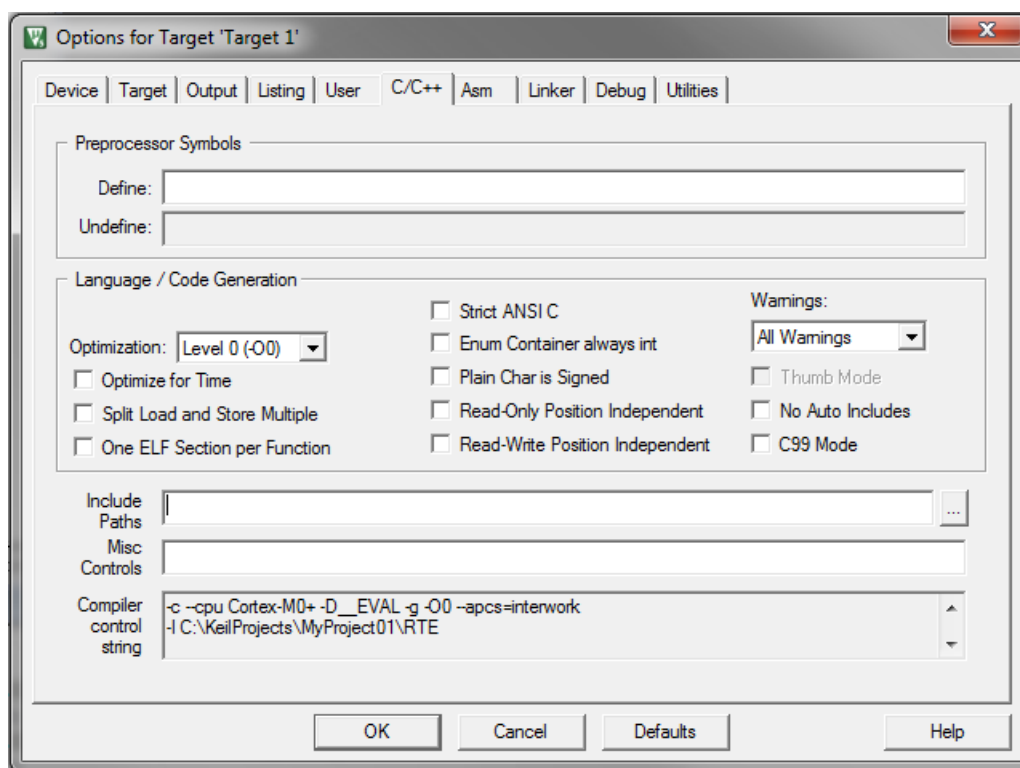


**Figure 1-25. Project workspace**

### 1.3.5 Library path setup

The following steps show the inclusion of all dependent modules.

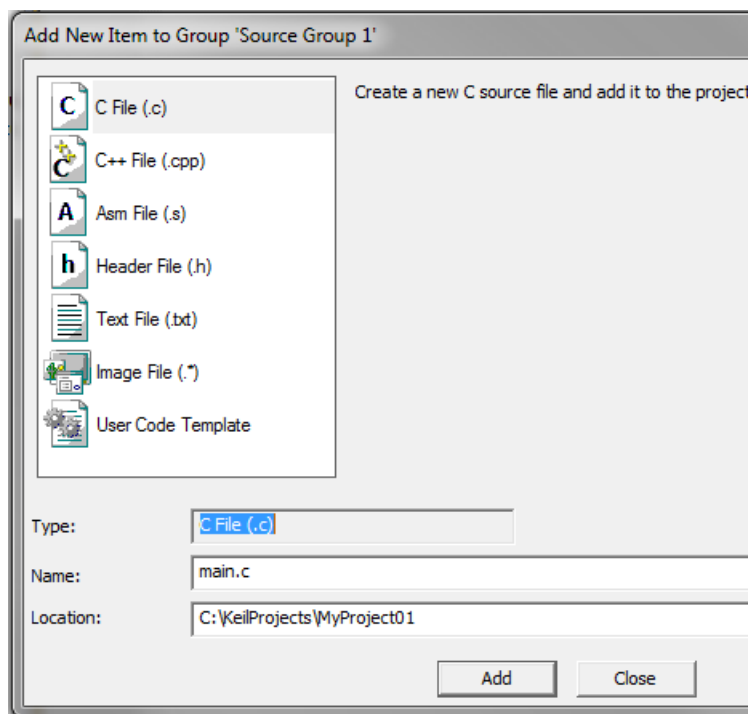
1. In the main menu, go to Project > Options for Target 'Target1'..., and a dialog appears.
2. Select the C/C++ tab. See [Figure 1-26](#).
3. In the Include Paths text box, type the following paths (if there are more paths, they must be separated by ';') or add them by clicking the ... button next to the text box:
  - "C:\NXP\RTCESL\CM0\_RTCESEL\_4.3\_KEIL\MLIB\Include"
  - "C:\NXP\RTCESL\CM0\_RTCESEL\_4.3\_KEIL\GFLIB\Include"
4. Click OK.
5. Click OK in the main dialog.



**Figure 1-26. Library path addition**

Type the `#include` syntax into the code. Include the library into a source file. In the new project, it is necessary to create a source file:

1. Right-click the Source Group 1 node, and Add New Item to Group 'Source Group 1'... from the menu.
2. Select the C File (.c) option, and type a name of the file into the Name box, for example '*main.c*'. See [Figure 1-27](#).



**Figure 1-27. Adding new source file dialog**

3. Click Add, and a new source file is created and opened up.
4. In the opened source file, include the following lines into the #include section, and create a main function:

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

int main(void)
{
    while(1);
}
```

When you click the Build (F7) icon, the project will be compiled without errors.

## 1.4 Library integration into project (IAR Embedded Workbench)

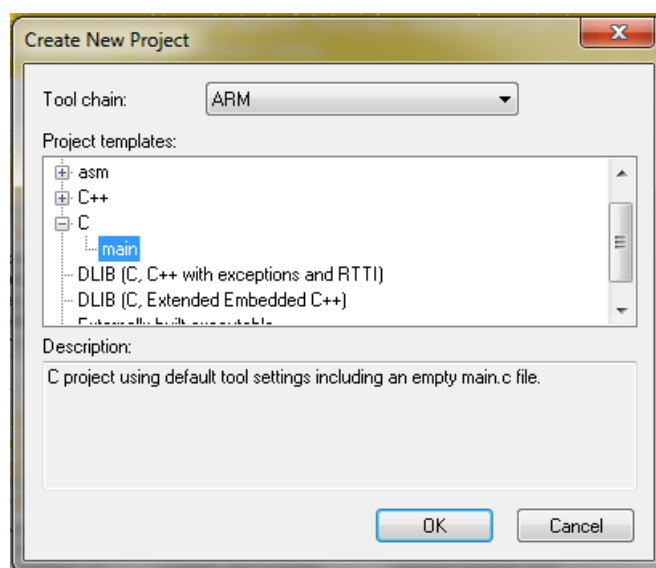
This section provides a step-by-step guide on how to quickly and easily include the GFLIB into an empty project or any SDK example or demo application projects using IAR Embedded Workbench. This example uses the default installation path (C:\NXP\RTCESL\CM0\_RTCESL\_4.3\_IAR). If you have a different installation path, use that path instead. If any SDK project is intended to use (for example hello\_world project) go to [Memory-mapped divide and square root support](#) chapter otherwise read next chapter.



### 1.4.1 New project (without SDK)

This example uses the NXP MKV10Z32xxx7 part, and the default installation path (C:\NXP\RTCESL\CM0\_RTCESL\_4.3\_IAR) is supposed. To start working on an application, create a new project. If the project already exists and is opened, skip to the next section. Perform these steps to create a new project:

1. Launch IAR Embedded Workbench.
2. In the main menu, select Project > Create New Project... so that the "Create New Project" dialog appears. See [Figure 1-28](#).



**Figure 1-28. Create New Project dialog**

3. Expand the C node in the tree, and select the "main" node. Click OK.
4. Navigate to the folder where you want to create the project, for example, C:\IARProjects\MyProject01. Type the name of the project, for example, MyProject01. Click Save, and a new project is created. The new project is now visible in the left-hand part of IAR Embedded Workbench. See [Figure 1-29](#).

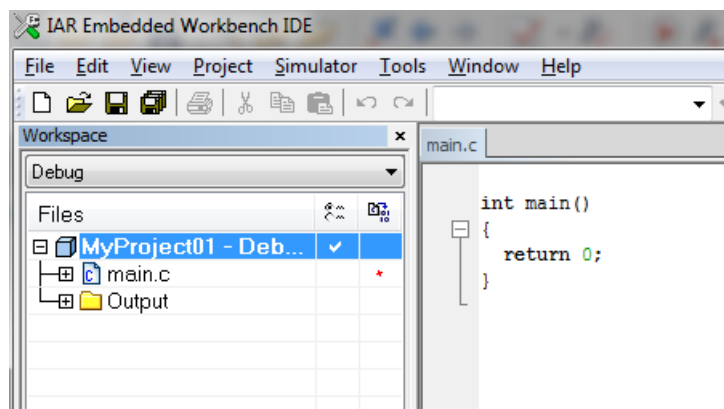


Figure 1-29. New project

5. In the main menu, go to Project > Options..., and a dialog appears.
6. In the Target tab, select the Device option, and click the button next to the dialog to select the MCU. In this example, select NXP > KV1x > NXP MKV10Z32xxx7 Click OK. See Figure 1-30.

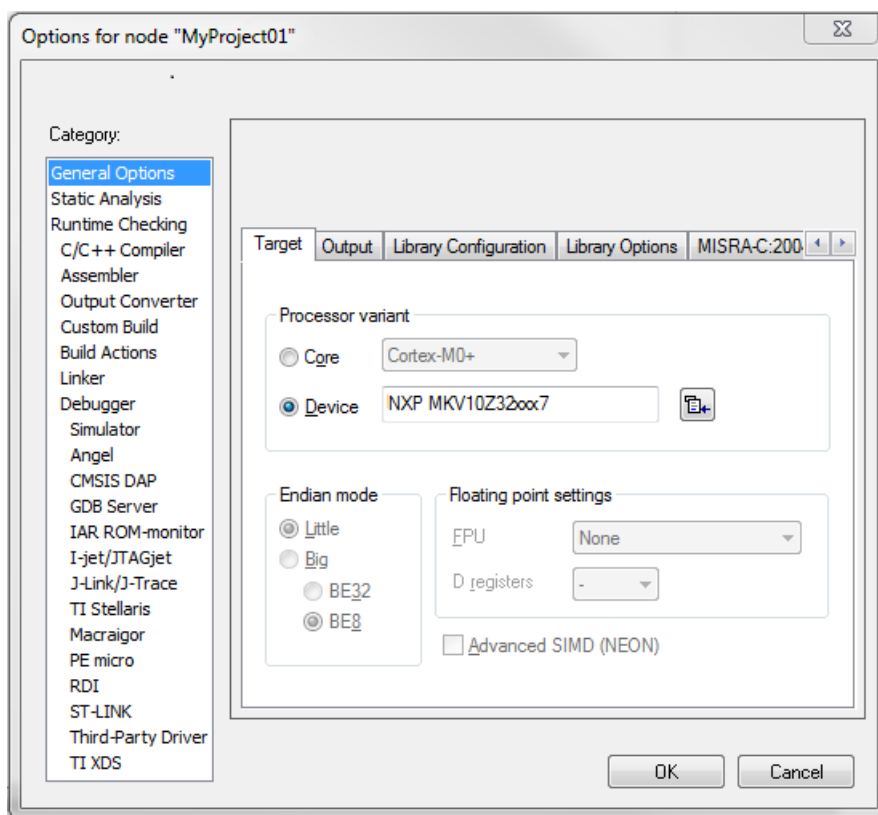


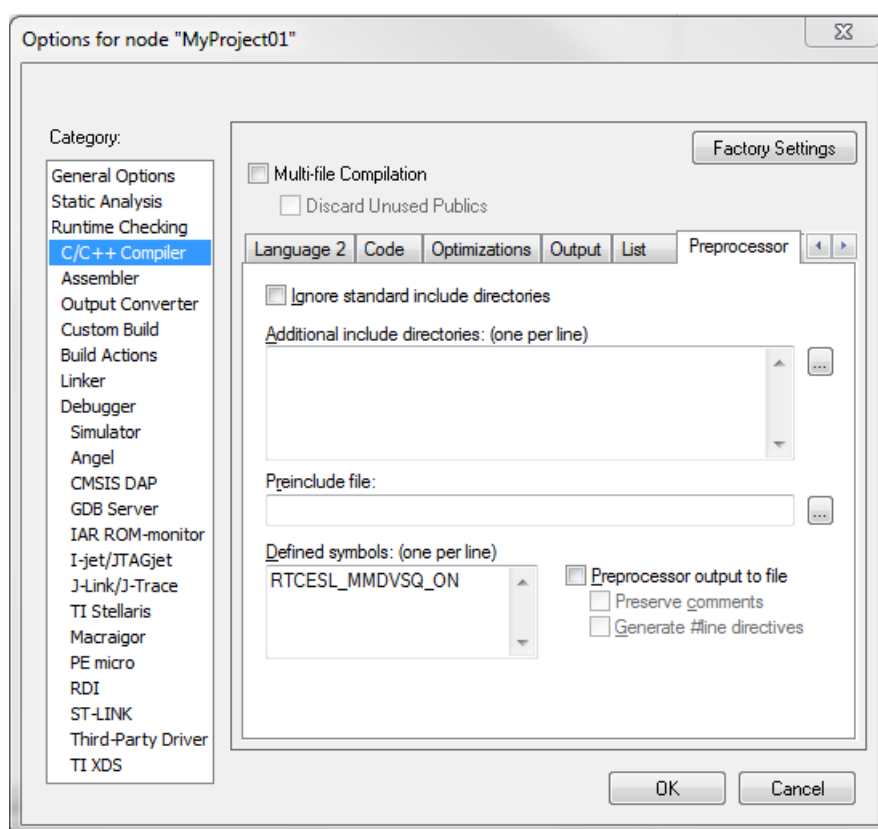
Figure 1-30. Options dialog

## 1.4.2 Memory-mapped divide and square root support

Some Kinetis platforms contain a peripheral module dedicated to division and square root. This section shows how to turn the memory-mapped divide and square root (MMDVSQ) support on and off.

1. In the main menu, go to Project > Options..., and a dialog appears.
2. In the left-hand column, select C/C++ Compiler.
3. In the right-hand part of the dialog, click the Preprocessor tab (it can be hidden in the right; use the arrow icons for navigation).
4. In the text box (at the Defined symbols: (one per line)), type the following (See [Figure 1-31](#)):
  - RTCESL\_MMDVSQ\_ON—to turn the hardware division and square root support on
  - RTCESL\_MMDVSQ\_OFF—to turn the hardware division and square root support off

If neither of these two defines is defined, the hardware division and square root support is turned off by default.



**Figure 1-31. Defined symbols**

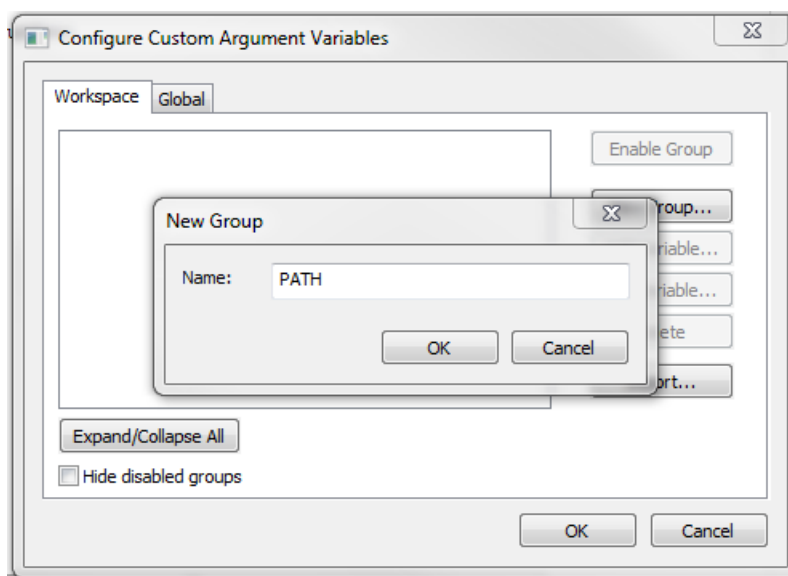
5. Click OK in the main dialog.

See the device reference manual to verify whether the device contains the MMDVSQ module.

### 1.4.3 Library path variable

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

1. In the main menu, go to Tools > Configure Custom Argument Variables..., and a dialog appears.
2. Click the New Group button, and another dialog appears. In this dialog, type the name of the group PATH, and click OK. See [Figure 1-32](#).



**Figure 1-32. New Group**

3. Click on the newly created group, and click the Add Variable button. A dialog appears.
4. Type this name: RTCESL\_LOC
5. To set up the value, look for the library by clicking the '...' button, or just type the installation path into the box: C:\NXP\RTCESL\CM0\_RTCESL\_4.3\_IAR. Click OK.
6. In the main dialog, click OK. See [Figure 1-33](#).

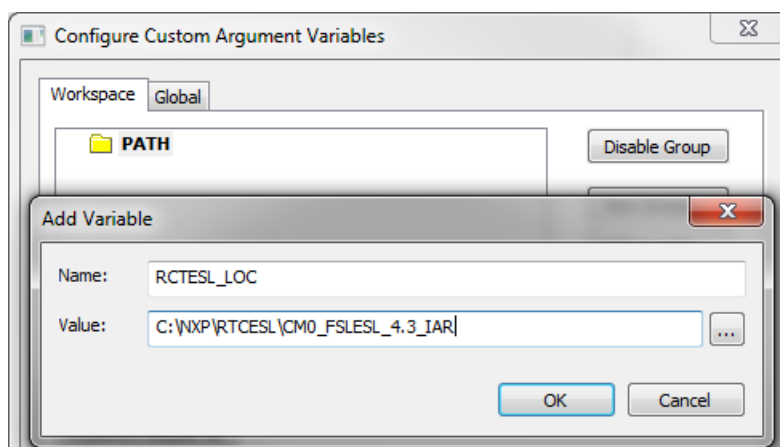


Figure 1-33. New variable

### 1.4.4 Linking the files into the project

GFLIB requires MLIB to be included too. The following steps show the inclusion of all dependent modules.

To include the library files into the project, create groups and add them.

1. Go to the main menu Project > Add Group...
2. Type RTCESL, and click OK.
3. Click on the newly created node RTCESL, go to Project > Add Group..., and create a MLIB subgroup.
4. Click on the newly created node MLIB, and go to the main menu Project > Add Files... See [Figure 1-35](#).
5. Navigate into the library installation folder C:\NXP\RTCESL\CM0\_RTCESEL\_4.3\_IAR\MLIB\Include, and select the *mlib.h* file. (If the file does not appear, set the file-type filter to Source Files.) Click Open. See [Figure 1-34](#).
6. Navigate into the library installation folder C:\NXP\RTCESL\CM0\_RTCESEL\_4.3\_IAR\MLIB, and select the *mlib.a* file. If the file does not appear, set the file-type filter to Library / Object files. Click Open.

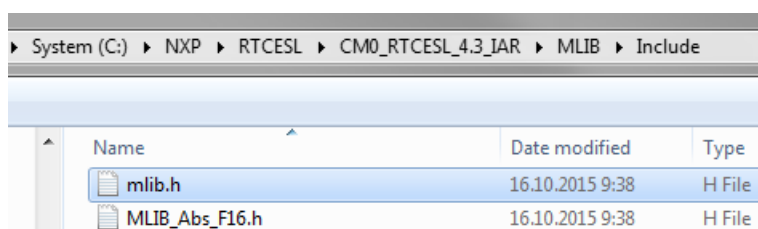


Figure 1-34. Add Files dialog

7. Click on the RTCESL node, go to Project > Add Group..., and create a GFLIB subgroup.

8. Click on the newly created node GFLIB, and go to the main menu Project > Add Files....
9. Navigate into the library installation folder C:\NXP\RTCESL\CM0\_RTCESL\_4.3\_IAR\GFLIB\Include, and select the *gflib.h* file. (If the file does not appear, set the file-type filter to Source Files.) Click Open.
10. Navigate into the library installation folder C:\NXP\RTCESL\CM0\_RTCESL\_4.3\_IAR\GFLIB, and select the *gflib.a* file. If the file does not appear, set the file-type filter to Library / Object files. Click Open.
11. Now you will see the files added in the workspace. See [Figure 1-35](#).

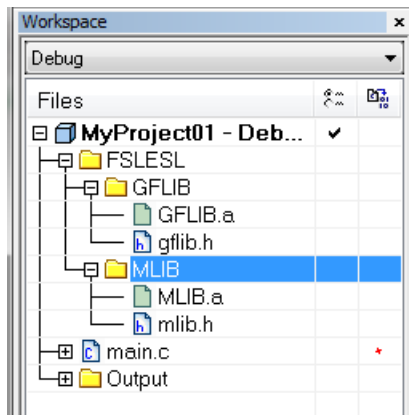
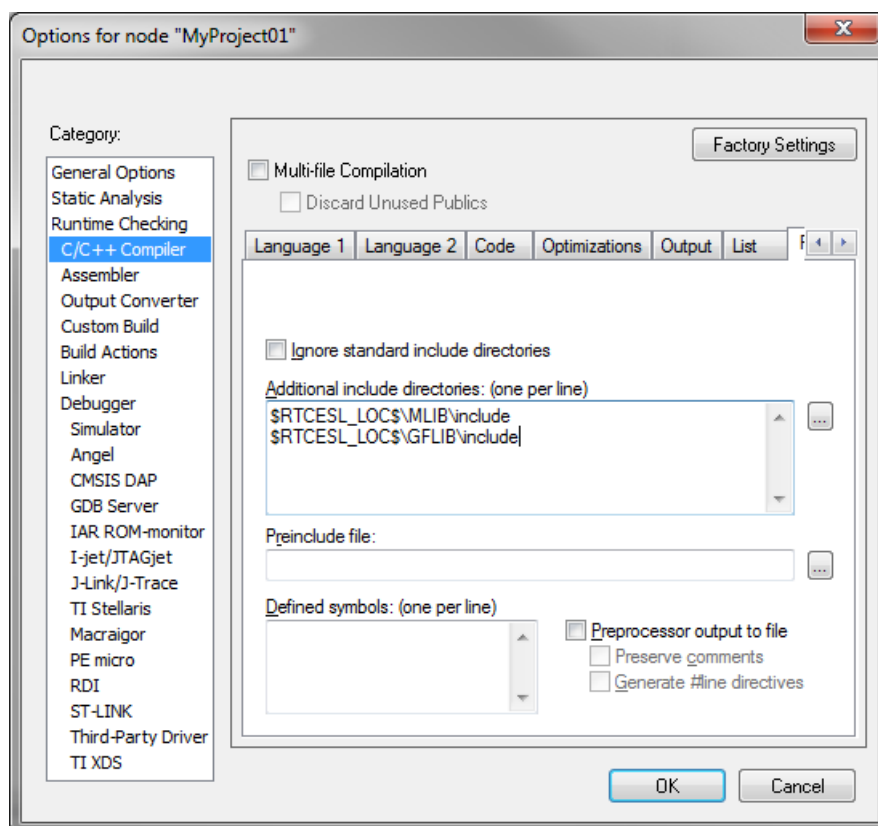


Figure 1-35. Project workspace

### 1.4.5 Library path setup

The following steps show the inclusion of all dependent modules:

1. In the main menu, go to Project > Options..., and a dialog appears.
2. In the left-hand column, select C/C++ Compiler.
3. In the right-hand part of the dialog, click on the Preprocessor tab (it can be hidden in the right; use the arrow icons for navigation).
4. In the text box (at the Additional include directories title), type the following folder (using the created variable):
  - \$RTCESL\_LOC\$\MLIB\Include
  - \$RTCESL\_LOC\$\GFLIB\Include
5. Click OK in the main dialog. See [Figure 1-36](#).



**Figure 1-36. Library path addition**

Type the `#include` syntax into the code. Include the library included into the *main.c* file. In the workspace tree, 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 Make icon, 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<sup>th</sup> 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.**

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

**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  $\pi$ :  $\sin(\pi \cdot x)$ , where the user passes the  $x$  argument. Example: if the input is  $-0.5$ , it corresponds to  $-0.5\pi$ .

The fractional function  $\sin(\pi \cdot x)$  is expressed using the 9<sup>th</sup> order Taylor polynomial as follows:

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

**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	<a href="#">frac16_t</a>	<a href="#">frac16_t</a>	Calculation of the $\sin(\pi \cdot x)$ , 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 <- $\pi$ ; $\pi$ ). 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); ( $\pi$  * f16Angle[rad]) = deg * ( $\pi$  / 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  $\pi$ :  $\cos(\pi \cdot x)$ , where the user passes the  $x$  argument. For example, if the input is  $-0.5$ , it corresponds to  $-0.5\pi$ .

### 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	<a href="#">frac16_t</a>	<a href="#">frac16_t</a>	Calculation of $\cos(\pi \cdot x)$ , 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 $<-\pi ; \pi)$ . 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"

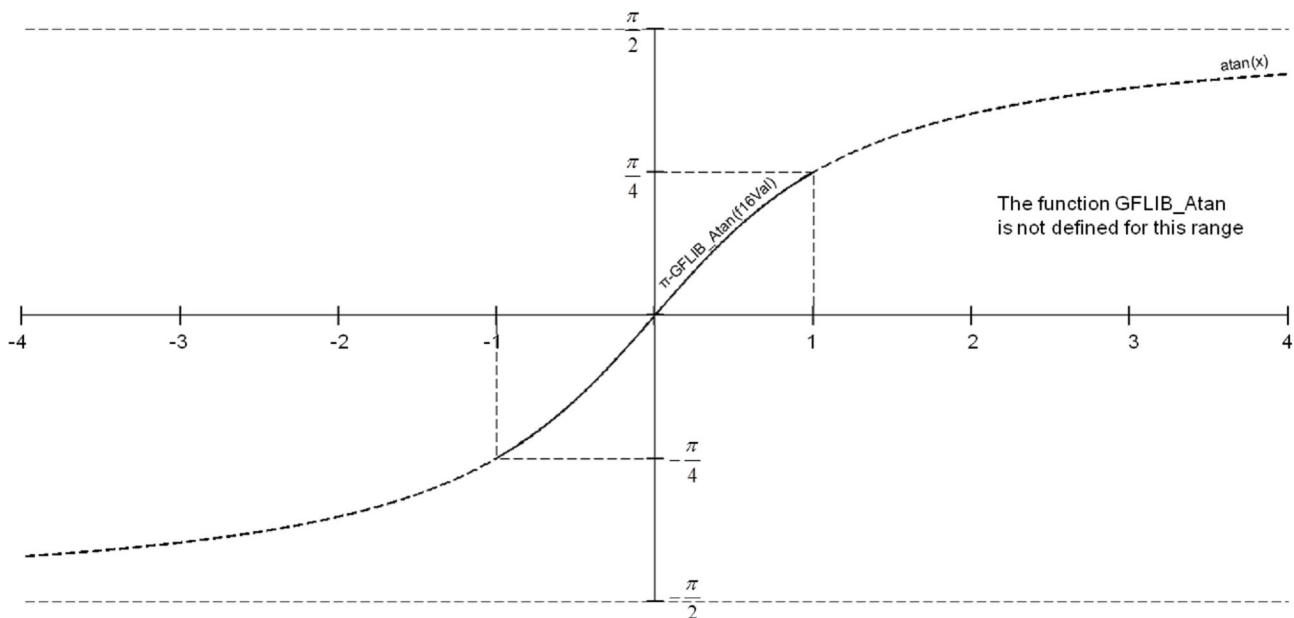
static frac16_t f16Result;
static frac16_t f16Angle;

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\_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 2-1. 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.

## 2.3.1 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  $<-0.25 ; 0.25)$ , which corresponds to the angle  $<-\pi / 4 ; \pi / 4)$ .

The available versions of the [GFLIB\\_Atan](#) function are shown in the following table:

**Table 2-3. Function versions**

Function name	Input type	Result type	Description
GFLIB_Atan_F16	<a href="#">frac16_t</a>	<a href="#">frac16_t</a>	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 $<-\pi / 4 ; \pi / 4) <-45^\circ ; 45^\circ)$ .

## 2.3.2 Declaration

The available [GFLIB\\_Atan](#) functions have the following declarations:

```
frac16\_t GFLIB_Atan_F16(frac16\_t f16Val)
```

## 2.3.3 Function use

The use of the [GFLIB\\_Atan](#) function is shown in the following example:

```
#include "gflib.h"

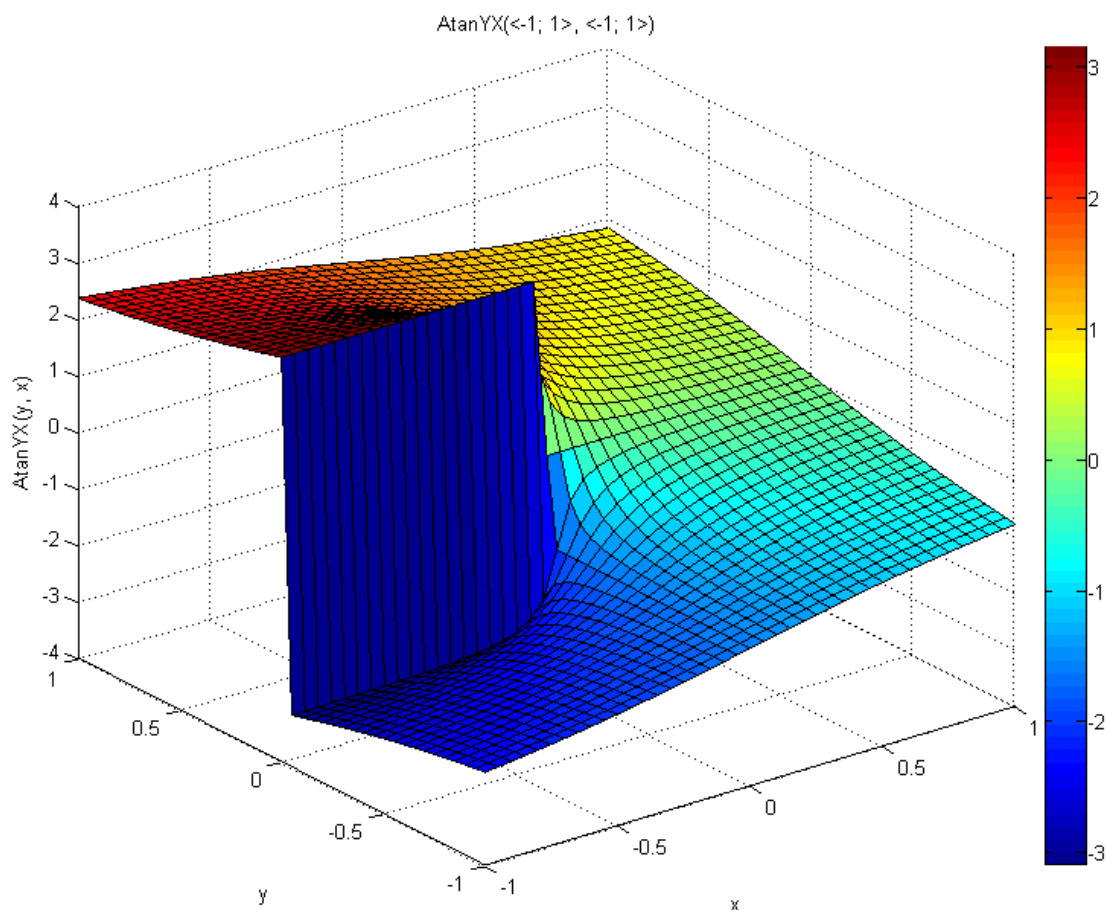
static frac16\_t f16Result;
static frac16\_t f16Val;

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

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

## 2.4 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 2-2. 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  $45^\circ$  (half-quarter), and the remaining offset within the  $45^\circ$  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  $<-\pi ; \pi>$ .

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  $<-\pi ; \pi>$ .

## 2.4.1 Available versions

This function is available in the following versions:

- Fractional output - the output is the fractional portion of the result; the result is within the range  $<-1 ; 1>$ , which corresponds to the angle  $<-\pi ; \pi>$ .

The available versions of the [GFLIB\\_AtanYX](#) function are shown in the following table:

**Table 2-4. Function versions**

Function name	Input type		Output type	Result type
	Y	X	Error flag	
GFLIB_AtanYX_F16	<a href="#">frac16_t</a>	<a href="#">frac16_t</a>	<a href="#">bool_t</a> *	<a href="#">frac16_t</a>
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 $<-\pi ; \pi>$ . 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.				

### NOTE

This algorithm can use the MMDVSQ peripheral module. See the following chapters for more details:

- [Memory-mapped divide and square root support](#) in Kinetis Design Studio
- [Memory-mapped divide and square root support](#) in Keil  $\mu$ Vision
- [Memory-mapped divide and square root support](#) in IAR Embedded Workbench

## 2.4.2 Declaration

The available [GFLIB\\_AtanYX](#) functions have the following declarations:

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

## 2.4.3 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);
}
```

## 2.5 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:

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

**Equation 5. Algorithm formula**

### 2.5.1 Available versions

This function is available in the following versions:

- Fractional output - the output is the fractional portion of the result; the result is within the range <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 2-5. Function versions**

Function name	Input type	Result type	Description
GFLIB_Sqrt_F16	<a href="#">frac16_t</a>	<a href="#">frac16_t</a>	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	<a href="#">frac32_t</a>	<a href="#">frac16_t</a>	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).

### NOTE

This algorithm can use the MMDVSQ peripheral module. See the following chapters for more details:

- [Memory-mapped divide and square root support](#) in Kinetis Design Studio
- [Memory-mapped divide and square root support](#) in Keil  $\mu$ Vision
- [Memory-mapped divide and square root support](#) in IAR Embedded Workbench

## 2.5.2 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)
```

## 2.5.3 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);
}
```

## 2.6 GFLIB\_Limit

The [GFLIB\\_Limit](#) function returns the value limited by the upper and lower limits. See the following equation:

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

**Equation 6. Algorithm formula**

### 2.6.1 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 2-6. Function versions**

Function name	Input type			Result type	Description
	Input	Lower limit	Upper limit		
GFLIB_Limit_F16	<a href="#">frac16_t</a>	<a href="#">frac16_t</a>	<a href="#">frac16_t</a>	<a href="#">frac16_t</a>	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	<a href="#">frac32_t</a>	<a href="#">frac32_t</a>	<a href="#">frac32_t</a>	<a href="#">frac32_t</a>	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>.

### 2.6.2 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)
```

### 2.6.3 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);
}
```

## 2.7 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 7. Algorithm formula**

### 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). The result may saturate.

The available versions of the [GFLIB\\_LowerLimit](#) functions are shown in the following table:

**Table 2-7. Function versions**

Function name	Input type		Result type	Description
	Input	Lower limit		
GFLIB_LowerLimit_F16	<a href="#">frac16_t</a>	<a href="#">frac16_t</a>	<a href="#">frac16_t</a>	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	<a href="#">frac32_t</a>	<a href="#">frac32_t</a>	<a href="#">frac32_t</a>	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).

## 2.7.2 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)
```

## 2.7.3 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);
}
```

## 2.8 GFLIB\_UpperLimit

The [GFLIB\\_UpperLimit](#) function returns the value limited by the upper limit. See the following equation:

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

**Equation 8. 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 <-1 ; 1). The result may saturate.

The available versions of the [GFLIB\\_UpperLimit](#) functions are shown in the following table:

**Table 2-8. Function versions**

Function name	Input type		Result type	Description
	Input	Upper limit		
GFLIB_UpperLimit_F16	<a href="#">frac16_t</a>	<a href="#">frac16_t</a>	<a href="#">frac16_t</a>	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	<a href="#">frac32_t</a>	<a href="#">frac32_t</a>	<a href="#">frac32_t</a>	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>.

## 2.8.2 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)
```

## 2.8.3 Function use

The use of the [GFLIB\\_UpperLimit](#) function is shown in the following example:

```
#include "gflib.h"

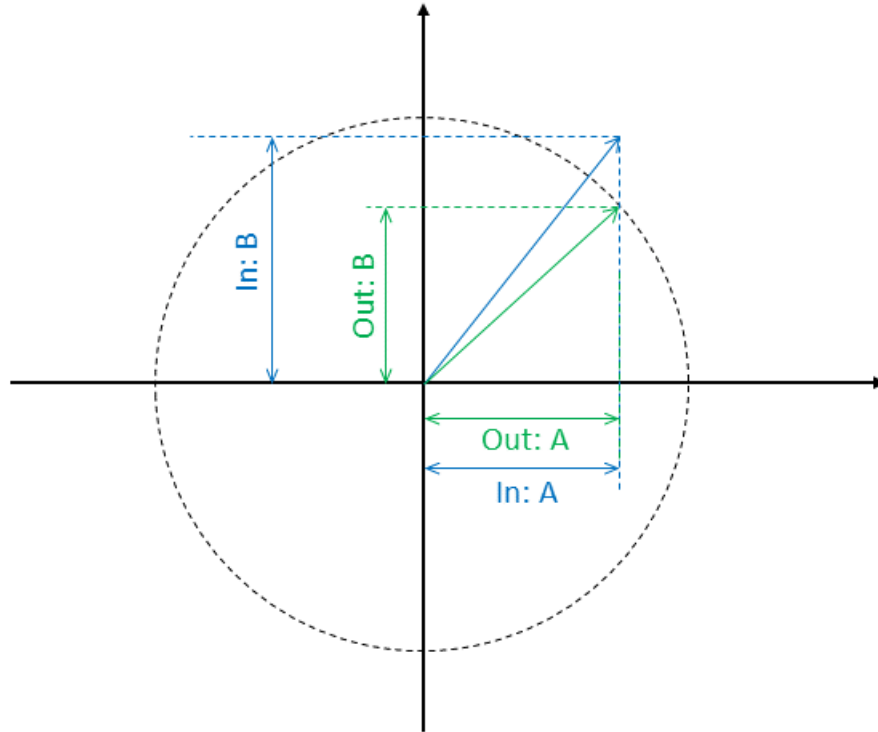
static frac16\_t f16Val, f16ULim, f16Result;

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

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

## 2.9 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 2-3. Input and related 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:

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

**Equation 9**

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

**Equation 10**

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 2-3](#).

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.

## 2.9.1 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 2-9. Function versions**

Function name	Input type		Output type	Result type
	Input	Limit		
GFLIB_VectorLimit1_F16	<a href="#">GFLIB_VECTORLIMIT_T_F16</a> *	<a href="#">frac16_t</a>	<a href="#">GFLIB_VECTORLIMIT_T_F16</a> *	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.				

### NOTE

This algorithm can use the MMDVSQ peripheral module. See the following sections for more details:

- [Memory-mapped divide and square root support](#) in Kinetis Design Studio
- [Memory-mapped divide and square root support](#) in Keil  $\mu$ Vision
- [Memory-mapped divide and square root support](#) in IAR Embedded Workbench

## 2.9.2 GFLIB\_VECTORLIMIT\_T\_F16 type description

Variable name	Input type	Description
f16A	<a href="#">frac16_t</a>	A-component; 16-bit fractional type.
f16B	<a href="#">frac16_t</a>	B-component; 16-bit fractional type.

### 2.9.3 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)
```

### 2.9.4 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;

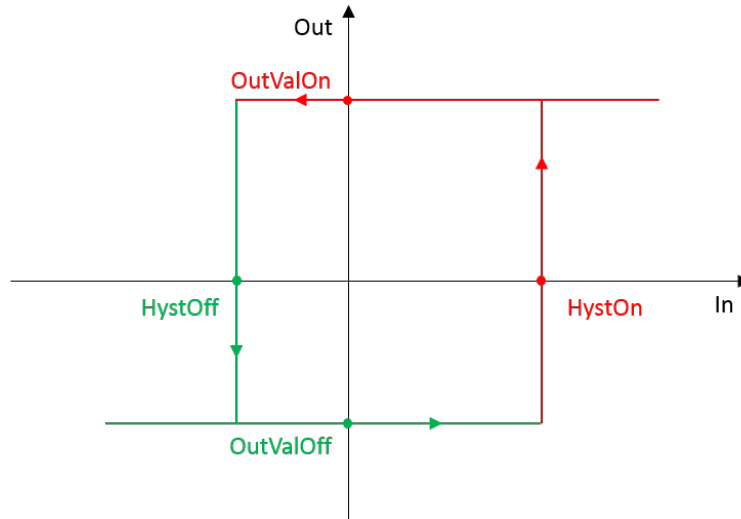
void main(void)
{
    f16MaxAmpl = FRAC16(0.5);
    sVector.f16A = FRAC16(-0.4);
    sVector.f16B = FRAC16(0.2);

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

## 2.10 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 2-4. 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.

### 2.10.1 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 2-10. Function versions**

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

### 2.10.2 GFLIB\_HYST\_T\_F16

Variable name	Input type	Description
f16HystOn	<a href="#">frac16_t</a>	The point where the output sets the output to the f16OutValOn value when the input rises. Set by the user.

*Table continues on the next page...*

Variable name	Input type	Description
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.

### 2.10.3 Declaration

The available [GFLIB\\_Hyst](#) functions have the following declarations:

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

### 2.10.4 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);
}
```

## 2.11 GFLIB\_Lut1D

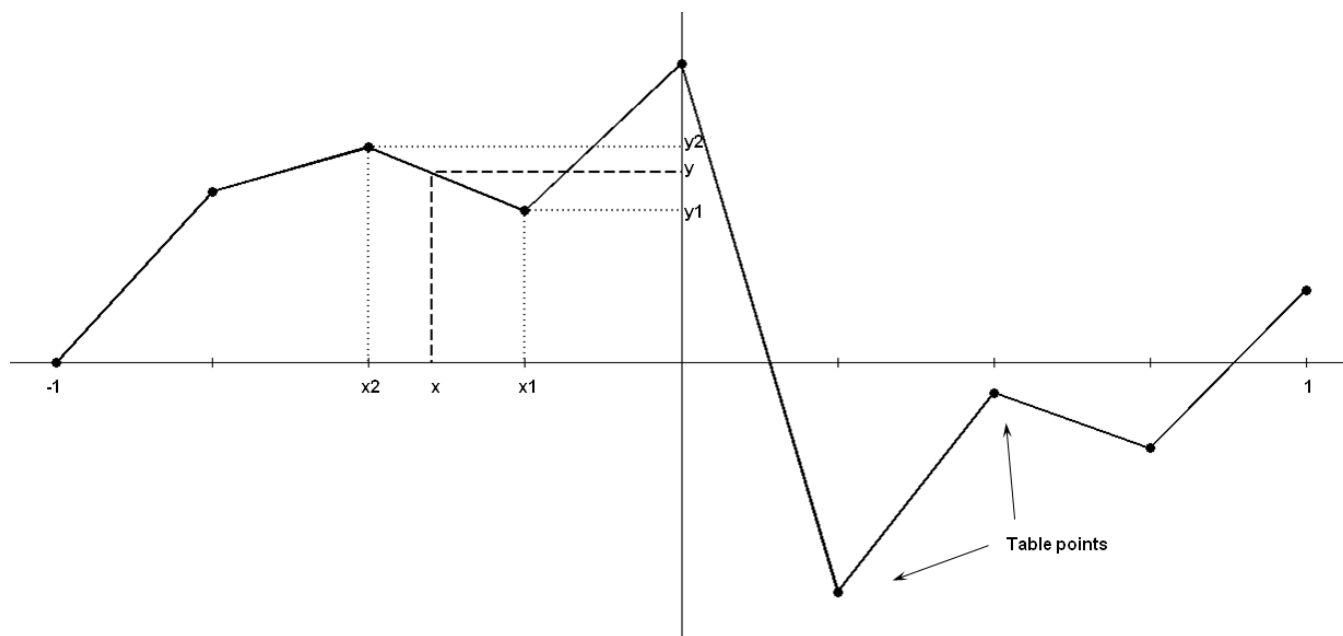
The [GFLIB\\_Lut1D](#) function implements the one-dimensional look-up table.

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

**Equation 11.**

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 2-5. 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.

### 2.11.1 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 2-11. Function versions**

Function name	Input type	Parameters		Result type
		Table	Table size	
GFLIB_Lut1D_F16	<a href="#">frac16_t</a>	<a href="#">frac16_t</a> *	<a href="#">uint16_t</a>	<a href="#">frac16_t</a>
	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.			

## 2.11.2 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)
```

## 2.11.3 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);
}
```

## 2.12 GFLIB\_LutPer1D

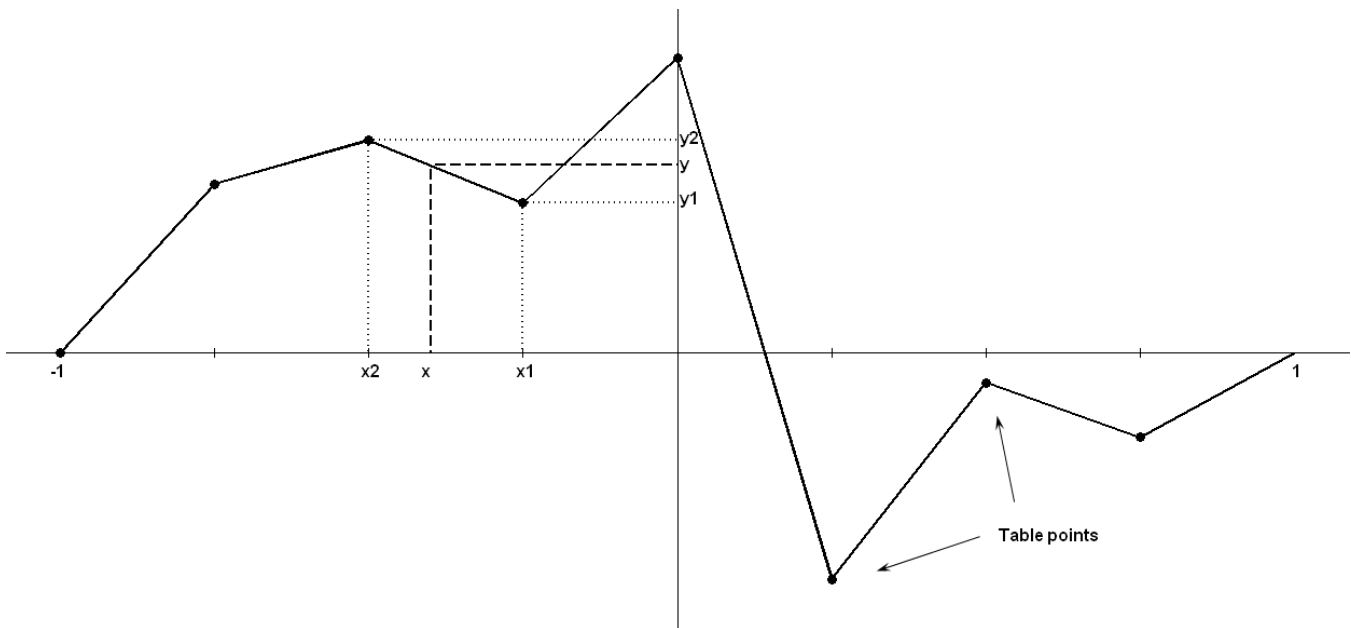
The [GFLIB\\_LutPer1D](#) function approximates the one-dimensional arbitrary user function using the interpolation look-up method. It is periodic.

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

**Equation 12.**

where:

- y is the interpolated value
- y<sub>1</sub> and y<sub>2</sub> are the ordinate values at the beginning and end of the interpolating interval, respectively
- x<sub>1</sub> and x<sub>2</sub> 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 2-6. 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.

## 2.12.1 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 2-12. Function versions**

Function name	Input type	Parameters		Result type
		Table	Table size	
GFLIB_LutPer1D_F16	<a href="#">frac16_t</a>	<a href="#">frac16_t*</a>	<a href="#">uint16_t</a>	<a href="#">frac16_t</a>
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.				

## 2.12.2 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)
```

## 2.12.3 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);
}
```

## 2.13 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:

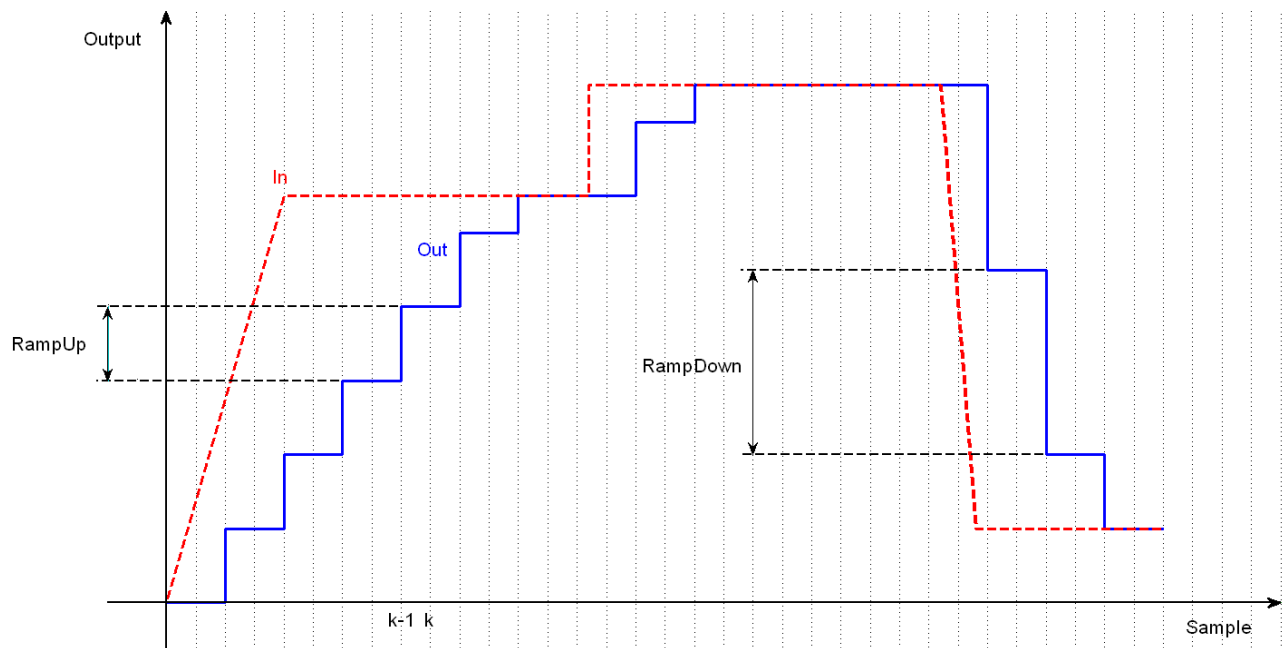


Figure 2-7. GFLIB\_Ramp functionality

## 2.13.1 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 2-13. Init function versions**

Function name	Input type	Parameters	Result type	Description
GFLIB_RampInit_F16	<a href="#">frac16_t</a>	<a href="#">GFLIB_RAMP_T_F16</a> *	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	<a href="#">frac32_t</a>	<a href="#">GFLIB_RAMP_T_F32</a> *	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 2-14. Function versions**

Function name	Input type	Parameters	Result type	Description
GFLIB_Ramp_F16	<a href="#">frac16_t</a>	<a href="#">GFLIB_RAMP_T_F16</a> *	<a href="#">frac16_t</a>	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	<a href="#">frac32_t</a>	<a href="#">GFLIB_RAMP_T_F32</a> *	<a href="#">frac32_t</a>	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 $-1 ; 1$ ).



## 2.13.2 GFLIB\_RAMP\_T\_F16

Variable name	Type	Description
f16State	<a href="#">frac16_t</a>	Actual value - controlled by the algorithm.
f16RampUp	<a href="#">frac16_t</a>	Value of the ramp-up increment. The data value is in the range <0 ; 1). Set by the user.
f16RampDown	<a href="#">frac16_t</a>	Value of the ramp-down increment. The data value is in the range <0 ; 1). Set by the user.

## 2.13.3 GFLIB\_RAMP\_T\_F32

Variable name	Type	Description
f32State	<a href="#">frac32_t</a>	Actual value - controlled by the algorithm.
f32RampUp	<a href="#">frac32_t</a>	Value of the ramp-up increment. The data value is in the range <0 ; 1). Set by the user.
f32RampDown	<a href="#">frac32_t</a>	Value of the ramp-down increment. The data value is in the range <0 ; 1). Set by the user.

## 2.13.4 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)
```

## 2.13.5 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);

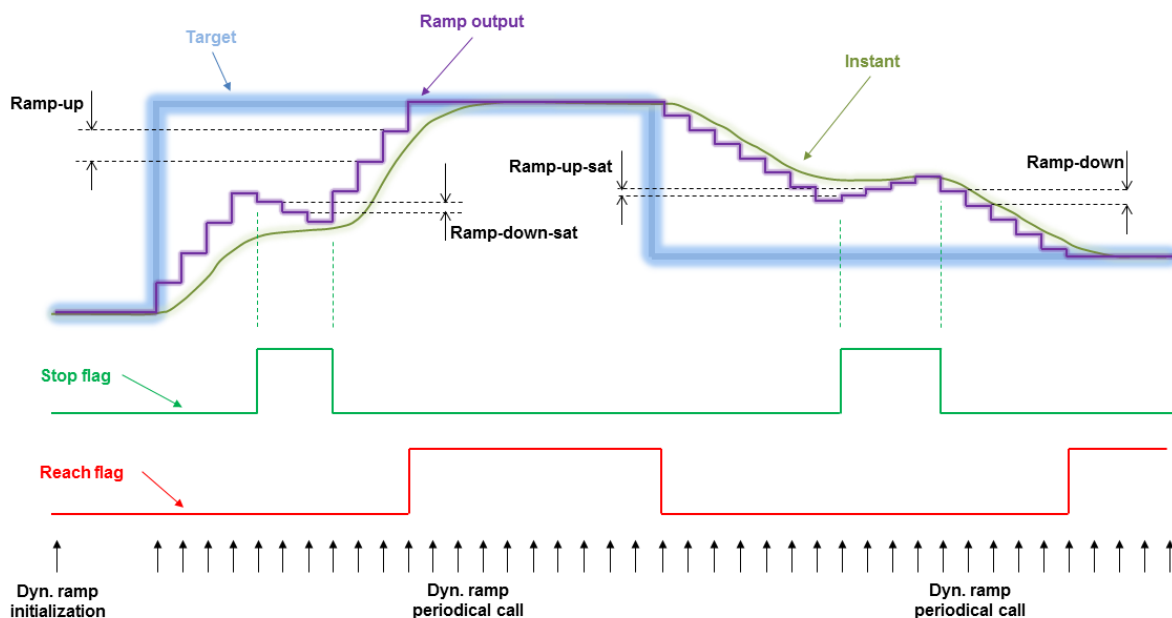
void main(void)
{
    sParam.f16RampUp = FRAC16(0.1);
    sParam.f16RampDown = FRAC16(0.02);
```

## GFLIB\_DRamp

```
f16Target = FRAC16(0.75);  
f16InitVal = FRAC16(0.9);  
GFLIB_RampInit_F16(f16InitVal, &sParam);  
}  
  
/* periodically called function */  
void Isr()  
{  
    f16Result = GFLIB_Ramp_F16(f16Target, &sParam);  
}
```

## 2.14 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 2-8. 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.

## 2.14.1 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 2-15. Init function versions**

Function name	Input type	Parameters	Result type	Description
GFLIB_DRampInit_F16	<a href="#">frac16_t</a>	<a href="#">GFLIB_DRAMP_T_F16</a> *	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	<a href="#">frac32_t</a>	<a href="#">GFLIB_DRAMP_T_F32</a> *	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 2-16. Function versions**

Function name	Input type			Parameters	Result type
	Target	Instant	Stop flag		
GFLIB_DRamp_F16	<a href="#">frac16_t</a>	<a href="#">frac16_t</a>	<a href="#">bool_t</a> *	<a href="#">GFLIB_DRAMP_T_F16</a> *	<a href="#">frac16_t</a>
	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 $<-1 ; 1$ ), the Stop flag parameter is a pointer to a boolean value, and the output data value is in the range $<-1 ; 1$ ).				
GFLIB_DRamp_F32	<a href="#">frac32_t</a>	<a href="#">frac32_t</a>	<a href="#">bool_t</a> *	<a href="#">GFLIB_DRAMP_T_F32</a> *	<a href="#">frac32_t</a>
	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 $<-1 ; 1$ ).				

## 2.14.2 GFLIB\_DRAMP\_T\_F16

Variable name	Type	Description
f16State	<a href="#">frac16_t</a>	Actual value - controlled by the algorithm.
f16RampUp	<a href="#">frac16_t</a>	Value of non-saturation ramp-up increment. The data value is in the range $<0 ; 1$ ). Set by the user.
f16RampDown	<a href="#">frac16_t</a>	Value of non-saturation ramp-down increment. The data value is in the range $<0 ; 1$ ). Set by the user.
f16RampUpSat	<a href="#">frac16_t</a>	Value of saturation ramp-up increment. The data value is in the range $<0 ; 1$ ). Set by the user.
f16RampDownSat	<a href="#">frac16_t</a>	Value of saturation ramp-down increment. The data value is in the range $<0 ; 1$ ). Set by the user.
bReachFlag	<a href="#">bool_t</a>	If the actual state value reaches the target value, this flag is set, otherwise, it is cleared. Set by the algorithm.

## 2.14.3 GFLIB\_DRAMP\_T\_F32

Variable name	Type	Description
f32State	<a href="#">frac32_t</a>	Actual value - controlled by the algorithm.
f32RampUp	<a href="#">frac32_t</a>	Value of non-saturation ramp-up increment. The data value is in the range $<0 ; 1$ ). Set by the user.
f32RampDown	<a href="#">frac32_t</a>	Value of non-saturation ramp-down increment. The data value is in the range $<0 ; 1$ ). Set by the user.

*Table continues on the next page...*

Variable name	Type	Description
f32RampUpSat	<a href="#">frac32_t</a>	Value of saturation ramp-up increment. The data value is in the range <0 ; 1). Set by the user.
f32RampDownSat	<a href="#">frac32_t</a>	Value of saturation ramp-down increment. The data value is in the range <0 ; 1). Set by the user.
bReachFlag	<a href="#">bool_t</a>	If the actual state value reaches the target value, this flag is set, otherwise, it is cleared. Set by the algorithm.

## 2.14.4 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)
```

## 2.14.5 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);
}
```

## 2.15 GFLIB\_FlexRamp

The [GFLIB\\_FlexRamp](#) function calculates the up/down ramp with a fixed-step increment that is calculated according to 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 to initialize the ramp.

To calculate the increment, 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 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](#) function 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 13.**

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 this figure:

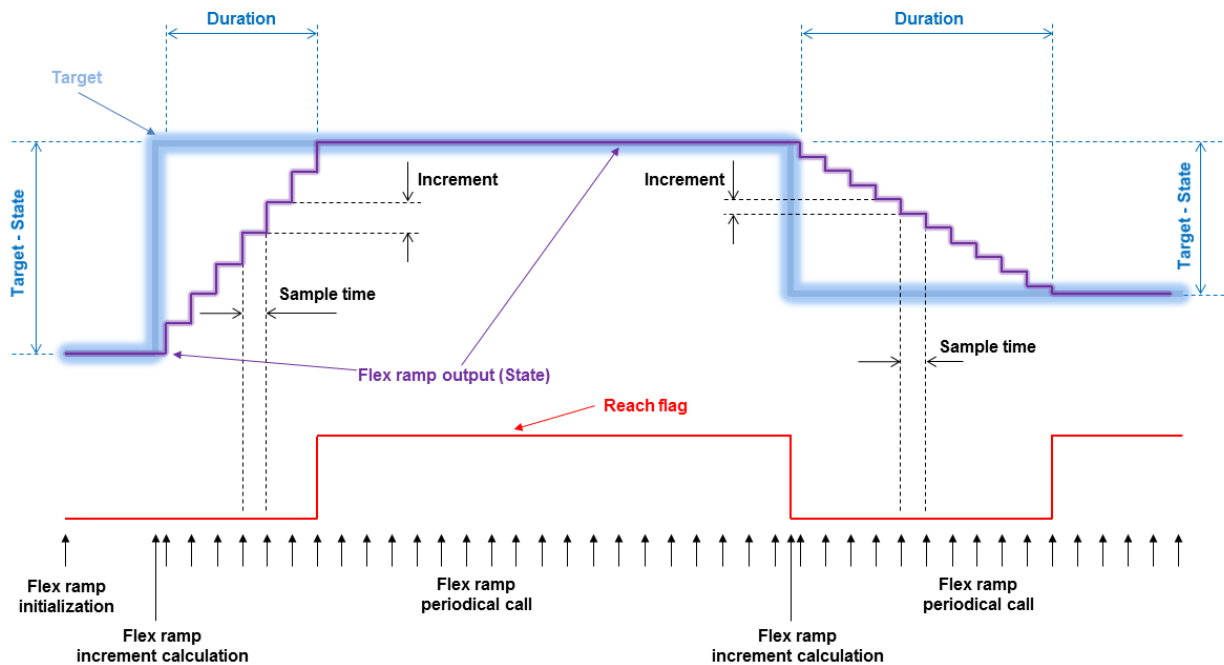


Figure 2-9. GFLIB\_FlexRamp functionality

### 2.15.1 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\_FlexRampInit function are shown in the following table:

**Table 2-17. Init function versions**

Function name	Input type	Parameters	Result type	Description
GFLIB_FlexRampInit_F16	frac16_t	GFLIB_FLEXRAMP_T_F32 *	void	The 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 2-18. 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
	The 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 of the ramp (in seconds) to reach the target value. The parameters' structure is pointed to by a pointer.			

**Table 2-19. 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).

## 2.15.2 GFLIB\_FLEXRAMP\_T\_F32

Variable name	Type	Description
f32State	frac32_t	The actual value. Controlled by the GFLIB_FlexRampInit_F16 and GFLIB_FlexRamp_F16 algorithms.
f32Incr	frac32_t	The value of the flex ramp increment. Controlled by the GFLIB_FlexRampCalcIncr_F16 algorithm.

Table continues on the next page...



Variable name	Type	Description
f32Target	<a href="#">frac32_t</a>	The target value of the flex ramp algorithm. Controlled by the GFLIB_FlexRampCalcIncr_F16 algorithm.
f32Ts	<a href="#">frac32_t</a>	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	<a href="#">frac32_t</a>	The maximum value of the flex ramp increment. The data value is in the range (0 ; 1). Set by the user.
bReachFlag	<a href="#">bool_t</a>	The 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.

### 2.15.3 Declaration

The available GFLIB\_FlexRampInit functions have the following declarations:

```
void GFLIB_FlexRampInit_F16(frac16\_t f16InitVal, GFLIB\_FLEXRAMP\_T\_F32 *psParam)
```

The available GFLIB\_FlexRampCalcIncr functions have the following declarations:

```
void GFLIB_FlexRampCalcIncr_F16(frac16\_t f16Target, acc32\_t a32Duration,
GFLIB\_FLEXRAMP\_T\_F32 *psParam)
```

The available [GFLIB\\_FlexRamp](#) functions have the following declarations:

```
frac16\_t GFLIB_FlexRamp_F16(GFLIB\_FLEXRAMP\_T\_F32 *psParam)
```

### 2.15.4 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\_F32 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);
}

```

## 2.16 GFLIB\_DFlexRamp

The [GFLIB\\_DFlexRamp](#) function calculates the up/down ramp with a fixed-step increment that is calculated according to 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 functionality 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, 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 to initialize the ramp.

To calculate the increment, use the [GFLIB\\_DFlexRampCalcIncr](#) function. Call this function when you want to change the ramp output value. This function's inputs are the target value and 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 calculate the ramp increment, fill the control structure with the sample time, that means the period of the loop where the [GFLIB\\_DFlexRamp](#) function 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 14.**

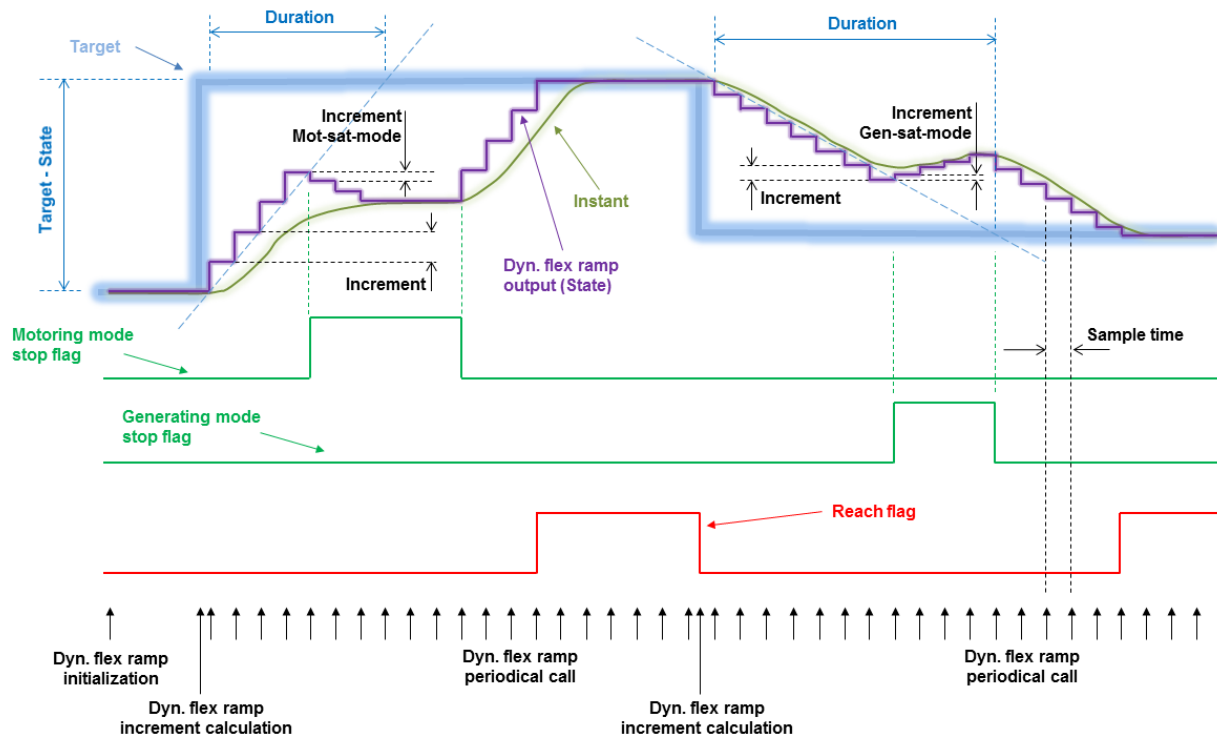
where:

- 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.

The state, target, and instant values must have the same sign, otherwise the saturation modes don't work properly.

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 2-10. 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 the state value, the function 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 dissipate the energy of the capacitor.

### 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 input parameters are the fractional and accumulator types.

The available versions of the GFLIB\_DFLEXRampInit functions are shown in the following table:

**Table 2-20. Init function versions**

Function name	Input type	Parameters	Result type	Description
GFLIB_FlexRampInit_F16	frac16_t	GFLIB_DFLEXRAMP_T_F32 *	void	The 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 2-21. 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
The input arguments are 16-bit fractional values in the range <-1 ; 1) that represent 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 2-22. 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
The 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).					

## 2.16.2 GFLIB\_DFLEXRAMP\_T\_F32

Variable name	Type	Description
f32State	<a href="#">frac32_t</a>	The actual value. Controlled by the GFLIB_FlexRampInit_F16 and GFLIB_FlexRamp_F16 algorithms.
f32Incr	<a href="#">frac32_t</a>	The value of the dyn. flex ramp increment. Controlled by the GFLIB_FlexRampCalcIncr_F16 algorithm.
f32IncrSatMot	<a href="#">frac32_t</a>	The value of the dyn. flex ramp increment when in the motoring saturation mode. Controlled by the GFLIB_DFlexRampCalcIncr_F16 algorithm.
f32IncrSatGen	<a href="#">frac32_t</a>	The value of the dyn. flex ramp increment when in the generating saturation mode. Controlled by the GFLIB_DFlexRampCalcIncr_F16 algorithm.
f32Target	<a href="#">frac32_t</a>	The target value of the flex ramp algorithm. Controlled by the GFLIB_DFlexRampCalcIncr_F16 algorithm.
f32Ts	<a href="#">frac32_t</a>	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	<a href="#">frac32_t</a>	The maximum value of the flex ramp increment. The data value is in the range (0 ; 1). Set by the user.
bReachFlag	<a href="#">bool_t</a>	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.

## 2.16.3 Declaration

The available GFLIB\_DFlexRampInit functions have the following declarations:

```
void GFLIB_DFlexRampInit_F16(frac16\_t f16InitVal, GFLIB\_DFLEXRAMP\_T\_F32 *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\_F32 *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\_F32 *psParam)
```

## 2.16.4 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\_F32 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_DFflexRampCalcIncr_F16(f16Target, a32RampDuration, f32IncrSatMotMode,
    f32IncrSatGenMode, &sDFlexRamp);
}

/* periodically called control loop with a period of 2 ms */
void Isr()
{
    f16RampResult = GFLIB_DFflexRamp_F16(f16Instant, &bSatMot, &bSatGen, &sDFlexRamp);
}

```

## 2.17 GFLIB\_Integrator

The [GFLIB\\_Integrator](#) function calculates a discrete implementation of the integrator (sum), discretized using a trapezoidal rule in Tustin's method (bi-linear transformation).

The continuous time domain representation of the integrator is defined as follows:

$$u(t) = \int e(t) dt$$

**Equation 15.**

In a continuous time domain, the transfer function for this integrator is described using the Laplace transformation as follows:

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

**Equation 16.**

Transforming the above equation into a digital time domain using the bi-linear transformation leads to the following transfer function:

$$Z\{H(s)\} = \frac{U(z)}{E(z)} = \frac{T_s + T_s z^{-1}}{2 - 2z^{-1}}$$

**Equation 17.**

where  $T_s$  is the sampling period of the system. The discrete implementation of the digital transfer function in the above equation is expressed as follows:

$$u(k) = u(k-1) + e(k) \frac{T_s}{2} + e(k-1) \frac{T_s}{2}$$

**Equation 18.**

Considering integrator gain  $K_I$ , the transfer function leads to the following equation:

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

**Equation 19.**

where:

- $u_I(k)$  is the integrator's output in the actual step
- $u_I(k-1)$  is the integrator's output from the previous step
- $e(k)$  is the integrator's input in the actual step
- $e(k-1)$  is the integrator's input from the previous step
- $K_I$  is the integrator's gain coefficient
- $T_s$  is the sampling period of the system

[Equation 19 on page 72](#) 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 20.**

where:

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



For a proper use of this function, it is recommended to initialize the function's data by the GFLIB\_IntegratorInit functions, before using the [GFLIB\\_Integrator](#) function. You must call the init function when you want the integrator to be initialized.

## 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), and it may overflow. The parameters use the accumulator types.

The available versions of the GFLIB\_IntegratorInit function are shown in the following table:

**Table 2-23. Init function versions**

Function name	Input type	Parameters	Result type	Description
GFLIB_IntegratorInit_F16	<a href="#">frac16_t</a>	<a href="#">GFLIB_INTEGRATOR_T_A32</a> *	void	The inputs are a 16-bit fractional initial value and a pointer to the integrator parameters' structure.

The available versions of the [GFLIB\\_Integrator](#) function are shown in the following table:

**Table 2-24. Function versions**

Function name	Input type	Parameters	Result type	Description
GFLIB_Integrator_F16	<a href="#">frac16_t</a>	<a href="#">GFLIB_INTEGRATOR_T_A32</a> *	<a href="#">frac16_t</a>	The inputs are a 16-bit fractional value to be integrated and a pointer to the integrator parameters' structure. The output is limited to range <-1 ; 1>. When the integrator reaches the limit, it overflows to the other limit.

## 2.17.2 GFLIB\_INTEGRATOR\_T\_A32

Variable name	Input type	Description
a32Gain	<a href="#">acc32_t</a>	Integrator gain is set up according to <a href="#">Equation 20 on page 72</a> as follows: $K_I T_s \cdot \frac{e_{max}}{u_{max}}$

*Table continues on the next page...*

Variable name	Input type	Description
		The parameter is a 32-bit accumulator type within the range <-65536.0 ; 65536.0). Set by the user.
f32IAccK_1	<a href="#">frac32_t</a>	Integral portion in the step k - 1. Controlled by the algorithm.
f16InValK_1	<a href="#">frac16_t</a>	Input value in the step k - 1. Controlled by the algorithm.

### 2.17.3 Declaration

The available GFLIB\_IntegratorInit functions have the following declarations:

```
void GFLIB_IntegratorInit_F16(frac16\_t f16InitVal, GFLIB\_INTEGRATOR\_T\_A32 *psParam)
```

The available [GFLIB\\_Integrator](#) functions have the following declarations:

```
frac16\_t GFLIB_Integrator_F16(frac16\_t f16InVal, GFLIB\_INTEGRATOR\_T\_A32 *psParam)
```

### 2.17.4 Function use

The use of the GFLIB\_IntegratorInit and [GFLIB\\_Integrator](#) functions is shown in the following example:

```
#include "gflib.h"

static frac16\_t f16Result, f16InVal, f16InitVal;
static GFLIB\_INTEGRATOR\_T\_A32 sParam;

void Isr(void);

void main(void)
{
    f16InVal = FRAC16(-0.4);
    sParam.a32Gain = ACC32(0.1);

    f16InitVal = FRAC16(0.1);

    GFLIB_IntegratorInit_F16(f16InitVal, &sParam);
}

/* periodically called function */
void Isr()
{
    f16Result = GFLIB_Integrator_F16(f16InVal, &sParam);
}
```

## 2.18 GFLIB\_CtrlIpAW

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:

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

**Equation 21.**

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 21 on page 75](#) can be expressed using the Laplace transformation as follows:

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

**Equation 22.**

The proportional part ( $u_P$ ) of [Equation 21 on page 75](#) is transformed into the discrete time domain as follows:

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

**Equation 23.**

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 23 on page 75 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 24.**

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 21 on page 75 into a discrete time domain using the bi-linear method, also known as the trapezoidal approximation, is as follows:

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

**Equation 25.**

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 25 on page 76 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 26.**

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.

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

**Equation 27.**

The bounds are described by a limitation element, as shown in [Equation 27 on page 77](#). 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.

## 2.18.1 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 2-25. Init function versions**

Function name	Input type	Parameters	Result type	Description
<code>GFLIB_CtrlPIpAWInit_F16</code>	<code>frac16_t</code>	<code>GFLIB_CTRL_PI_P_AW_T_A32</code> *	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\\_CtrlIpAW](#) function are shown in the following table:

**Table 2-26. Function versions**

Function name	Input type		Parameters	Result type
	Error	Stop flag		
GFLIB_CtrlIpAW_F16	<a href="#">frac16_t</a>	<a href="#">bool_t</a> *	<a href="#">GFLIB_CTRL_PI_P_AW_T_A32</a> *	<a href="#">frac16_t</a>
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>.				

## 2.18.2 GFLIB\_CTRL\_PI\_P\_AW\_T\_A32

Variable name	Input type	Description
a32PGain	<a href="#">acc32_t</a>	Proportional gain is set up according to <a href="#">Equation 24 on page 76</a> 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	<a href="#">acc32_t</a>	Integral gain is set up according to <a href="#">Equation 26 on page 76</a> 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	<a href="#">frac32_t</a>	State variable of the internal accumulator (integrator). Controlled by the algorithm.
f16InErrK_1	<a href="#">frac16_t</a>	Input error at the step k - 1. Controlled by the algorithm.
f16UpperLim	<a href="#">frac16_t</a>	Upper limit of the controller's output and the internal accumulator (integrator). This parameter must be greater than f16LowerLim. Set by the user.
f16LowerLim	<a href="#">frac16_t</a>	Lower limit of the controller's output and the internal accumulator (integrator). This parameter must be lower than f16UpperLim. Set by the user.
bLimFlag	<a href="#">bool_t</a>	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.

## 2.18.3 Declaration

The available GFLIB\_CtrlIpAWInit functions have the following declarations:

```
void GFLIB_CtrlIpAWInit_F16(frac16\_t f16InitVal, GFLIB\_CTRL\_PI\_P\_AW\_T\_A32 *psParam)
```

The available [GFLIB\\_CtrlIpAW](#) functions have the following declarations:

```
frac16\_t GFLIB_CtrlIpAW_F16(frac16\_t f16InErr, const bool\_t *pbStopIntegFlag,  
GFLIB\_CTRL\_PI\_P\_AW\_T\_A32 *psParam)
```

## 2.18.4 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);
}
```





# 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 uint8_t;
```

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

**Table A-2. Data storage**

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

## A.3 uint16\_t

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

```
typedef unsigned short uint16_t;
```

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

**Table A-3. Data storage**

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

## A.4 uint32\_t

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

```
typedef unsigned long uint32_t;
```

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

**Table A-4. Data storage**

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

## A.5 int8\_t

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

```
typedef char int8_t;
```

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

**Table A-5. Data storage**

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

## A.6 int16\_t

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

```
typedef short int16_t;
```

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

**Table A-6. Data storage**

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

## A.7 int32\_t

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

```
typedef long int32_t;
```

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

**Table A-7. Data storage**

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

## A.8 frac8\_t

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

```
typedef char frac8_t;
```

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

**Table A-8. Data storage**

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

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

## A.9 frac16\_t

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

```
typedef short frac16_t;
```

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

**Table A-9. Data storage**

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

*Table continues on the next page...*

**Table A-9. Data storage (continued)**

0.47357  -0.75586	8				0				0				0			
	0	0	1	1	1	1	0	0	1	0	0	1	1	1	1	0
	3				C				9				E			
	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  $\langle 0x80 ; 0x7F \rangle$ , which corresponds to  $\langle -1.0 ; 1.0 \cdot 2^{-7} \rangle$ .



```
#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](http://freescale.com)**Web Support:**[freescale.com/support](http://freescale.com/support)

Information in this document is provided solely to enable system and software implementers to use Freescale products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits based on the information in this document.

Freescale reserves the right to make changes without further notice to any products herein.

Freescale makes no warranty, representation, or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages.

“Typical” parameters that may be provided in Freescale data sheets and/or specifications can and do vary in different applications, and actual performance may vary over time. All operating parameters, including “typicals,” must be validated for each customer application by customer's technical experts. Freescale does not convey any license under its patent rights nor the rights of others. Freescale sells products pursuant to standard terms and conditions of sale, which can be found at the following address: [www.freescale.com/salestermsandconditions](http://www.freescale.com/salestermsandconditions).

Freescale and the Freescale logo are trademarks of Freescale Semiconductor, Inc. ARM and Cortex are the registered trademarks of ARM Limited, in EU and/or elsewhere. ARM logo is the trademark of ARM Limited. All rights reserved. All other product or service names are the property of their respective owners.

© 2016 Freescale Semiconductor, Inc.

Document Number CM0GFLIBUG  
Revision 3, 04/2016

