

# Automotive Math and Motor Control Library Set for NXP S32K14x devices

User's Guide

Rev. 17 — 26 June 2020

[User guide](#)



**Revision History**

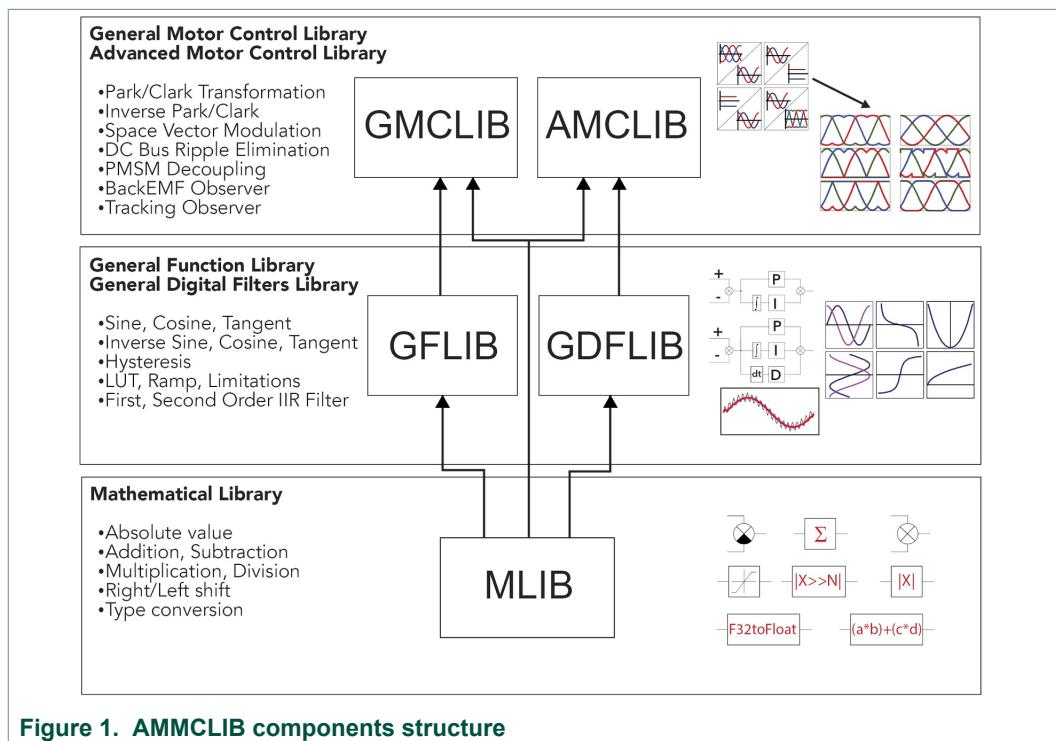
Revision	Date	Description
1.0	6-March-2015	Initial version.
2.0	31-December-2015	The Service Release v1.1.3. The example code for GFLIB_Lut1D_FLT function was corrected. The GreenHills compiler version was increased to the latest v2015.1.4.
3.0	31-March-2016	Change from Freescale to NXP entity, Service release 1.1.4.
4.0	30-June-2016	The Service Release v1.1.5. Added new MLIB_RndSat_F16F32 function. Corrected examples for GFLIB_ControllerPIrAW_FLT/GFLIB_ControllerPIr_FLT functions. Added chapter about possible exceptions.
5.0	31-December-2016	The Service Release v1.1.7. Added AMCLIB_BemfObsrvDQ and AMCLIB_TrackObsrv functions. Software License Agreement updated.
6.0	31-March-2017	The Service Release v1.1.8. Repeated sections were merged into a common chapters in the User Guide. This approach was applied to all AMMCLIB functions.
7.0	31-December-2017	The Service Release v1.1.11. Added new functions AMCLIB_CurrentLoop, AMCLIB_FW, AMCLIB_FWSpeedLoop, AMCLIB_SpeedLoop.
8.0	31-March-2018	The Service Release v1.1.12. Added new function GFLIB_VMin.
9.0	30-June-2018	The Service Release v1.1.13. Added new SetState and Init functions.
10.0	30-September-2018	The Service Release v1.1.14. Changed implementation of GFLIB_Ramp.
11.0	31-December-2018	The Service Release v1.1.15. See release notes for list of changes.
12.0	31-March-2019	The Service Release v1.1.16. See release notes for list of changes.
13.0	30-June-2019	The Service Release v1.1.17. See release notes for list of changes.
14.0	30-September-2019	The Service Release v1.1.18. See release notes for list of changes.
15.0	31-December-2019	The Service Release v1.1.19. See release notes for list of changes.
16.0	31-March-2020	The Service Release v1.1.20. See release notes for list of changes.
17.0	30-June-2020	The Service Release v1.1.21. See release notes for list of changes.

## 1 Introduction

The aim of this document is to describe the Automotive Math and Motor Control Library Set for NXP S32K14x devices. It describes the components of the library, its behavior and interaction, the API and steps needed to integrate the library into the customer project.

### 1.1 Architecture Overview

The Automotive Math and Motor Control Library Set for NXP S32K14x devices consists of several sub-libraries, functionally connected as depicted in [Figure 1](#).



**Figure 1. AMMCLIB components structure**

The Automotive Math and Motor Control Library Set for NXP S32K14x devices sub-libraries are as follows:

- **Mathematical Function Library (MLIB)** - comprising basic mathematical operations such as addition, multiplication, etc.
- **General Function Library (GFLIB)** - comprising basic trigonometric and general math functions such as sine, cosine, tan, hysteresis, limit, etc.
- **General Digital Filters Library (GDFLIB)** - comprising digital IIR and FIR filters designed to be used in a motor control application
- **General Motor Control Library (GMCLIB)** - comprising standard algorithms used for motor control such as Clarke/Park transformations, Space Vector Modulation, etc.
- **Advanced Motor Control Function Library (AMCLIB)** - comprising advanced algorithms used for motor control purposes

As can be seen in [Figure 1](#), the Automotive Math and Motor Control Library Set for NXP S32K14x devices libraries form the layer architecture where all upper libraries utilize the

functions from MLIB library. This concept is a key factor for mathematical operations abstractions allowing to support the highly target-optimized variants.

## 1.2 General Information

The Automotive Math and Motor Control Library Set for NXP S32K14x devices was developed to support these major implementations:

- Fixed-point 32-bit fractional
- Fixed-point 16-bit fractional
- Single precision floating point

With exception of those functions where the mathematical principle limits the input or output values, these values are considered to be in the following limits:

- **Fixed-point 32-bit fractional:**  $<-1; 1-2^{-31}>$  in Q1.31 format and with minimum positive normalized value  $2^{-31}$ .
- **Fixed-point 16-bit fractional:**  $<-1; 1-2^{-15}>$  in Q1.15 format and with minimum positive normalized value  $2^{-15}$ .
- **Single precision floating point:**  $<-2^{128}; 2^{128}>$  and with minimum positive normalized value  $2^{-128}$ .

Also those functions which are not relevant for particular implementation, e.g. saturated functions or shifting for single precision floating point implementation, are not delivered with this package. For detailed information about available functions please refer to [Section 2.1](#).

**Note:** The fixed-point 32-bit fractional and fixed-point 16-bit fractional functions are implemented based on the unity model. Which means that the input and output numbers are normalized to fit between the  $<-1; 1-2^{-31}>$  or  $<-1; 1-2^{-15}>$  range representing the Q1.31 or Q1.15 format.

The Automotive Math and Motor Control Library Set for NXP S32K14x devices was tested using two different test methods. To test the precision of each function implementation, the testing based on MATLAB reference models was used. This release was tested using the MATLAB R2019a version. To test the implementation on the embedded side, the target-in-loop testing was performed on the Automotive Math and Motor Control Library Set for NXP S32K14x devices. This release was tested using MATLAB R2019a version.

## 1.3 Multiple Implementation Support

In order to allow the user to utilize arbitrary implementation within one user application without any limitations, three different function call methods are supported in the Automotive Math and Motor Control Library Set for NXP S32K14x devices:

- Global configuration option
- Additional parameter option
- API postfix option

Each of these method calls the API postfix function. Thus, for each implementation (32-bit fixed-point, 16-bit fixed point and single precision floating point) only one function is available within the package. This approach is based on ANSI-C99 ISO/IEC 9899:1999 function overloading.

## Global Configuration Option

This function call supports the user legacy applications, which were based on older version of Motor Control Library. Prior to any Automotive Math and Motor Control Library Set for NXP S32K14x devices function call using the Global configuration option, the `SWLIBS_DEFAULT_IMPLEMENTATION` macro definition has to be setup in the file `SWLIBS_Config.h`.

The `SWLIBS_DEFAULT_IMPLEMENTATION` macro is defined in the `SWLIBS_Config.h` file located in Common directory of the Automotive Math and Motor Control Library Set for NXP S32K14x devices installation destination. The `SWLIBS_DEFAULT_IMPLEMENTATION` can be defined as the one of the following supported implementations:

- `SWLIBS_DEFAULT_IMPLEMENTATION_F32` for 32-bit fixed-point implementation
- `SWLIBS_DEFAULT_IMPLEMENTATION_F16` for 16-bit fixed-point implementation
- `SWLIBS_DEFAULT_IMPLEMENTATION_FLT` for single precision floating point implementation

After proper definition of `SWLIBS_DEFAULT_IMPLEMENTATION` macro the Automotive Math and Motor Control Library Set for NXP S32K14x devices functions can be called using standard legacy API convention, e.g. `GFLIB_Sin(x)`.

For example if the `SWLIBS_DEFAULT_IMPLEMENTATION` macro definition is set to `SWLIBS_DEFAULT_IMPLEMENTATION_F32`, the 32-bit fixed-point implementation of sine function is invoked after the `GFLIB_Sin(x)` API call. Note that all standard legacy API calls will invoke the 32-bit fixed-point implementation in this example.

**Note:** As the Automotive Math and Motor Control Library Set for NXP S32K14x devices supports the global configuration option, it is highly recommended to copy the `SWLIBS_Config.h` file to your local structure and refer the configuration to this local copy. This approach will prevent the incorrect setup of default configuration option, in case multiple projects with different default configuration are used.

## Additional Parameter Option

In order to support the free selection of used implementation in the user application while keeping the function name same as in standard legacy API approach, the additional parameter option is implemented in the Automotive Math and Motor Control Library Set for NXP S32K14x devices. In this option the additional parameter is used to distinguish which implementation shall be invoked. There are the following possible switches selecting the implementation:

- `F32` for 32-bit fixed-point implementation
- `F16` for 16-bit fixed-point implementation
- `FLT` for single precision floating point implementation

For example, if the user application needs to invoke the 16-bit fixed-point implementation of sine function, the `GFLIB_Sin(x, F16)` API call needs to be used. Note that there is a possibility to call any implementation of the functions in user application without any limitation.

## API Postfix Option

In order to support the free selection of used implementation in the user application while keeping the number of parameters same as in standard legacy API approach, the API

postfix option is implemented in the Automotive Math and Motor Control Library Set for NXP S32K14x devices. In this option the implementation postfix is used to distinguish which implementation shall be invoked. There are the following possible API postfixes selecting the implementation:

- **F32** for 32-bit fixed-point implementation
- **F16** for 16-bit fixed-point implementation
- **FLT** for single precision floating point implementation

For example, if the user application needs to invoke the 32-bit implementation of sine function, the *GFLIB\_Sin\_F32* API call needs to be used. Note that there is a possibility to call any implementation of the functions in user application without any limitation.

## 1.4 Supported Compilers

The Automotive Math and Motor Control Library Set for NXP S32K14x devices is written in ANSI-C99 ISO/IEC 9899:1999 standard language with critical parts implemented in assembly. The library was built and tested using the following compilers:

1. Green Hills MULTI v2017.1.4
2. IAR Embedded Workbench for ARM V8.11.2.13589
3. S32 Design Studio for ARM based MCUs 2018.R1 (GCC compiler version 6.3.1 20170509)

The library is delivered in a library module "S32K14x\_AMMCLIB.a" for the Green Hills compiler. The library module is located in "C:\NXPI\AMMCLIB\S32K14x\_AMMCLIB\_v1.1.21\lib\lhs" folder (considering the default installation path).

The library is delivered in a library module "S32K14x\_AMMCLIB.a" for the IAR compiler. The library module is located in "C:\NXPI\AMMCLIB\S32K14x\_AMMCLIB\_v1.1.21\lib\iar" folder (considering the default installation path).

The library is delivered in a library module "S32K14x\_AMMCLIB.a" for the S32 Design Studio IDE for ARM based MCUs. The library module is located in "C:\NXPI\AMMCLIB\S32K14x\_AMMCLIB\_v1.1.21\lib\s32ds\_arm32" folder (considering the default standalone installation path). The library is also pre-installed with the S32 Design Studio IDE for ARM based MCUs. The library module is located in "<S32DS installation directory>\S32DS\S32K14x\_AMMCLIB\_v1.1.21\lib\s32ds\_arm32" folder.

Together with the pre-compiled library modules, these are all the necessary header files. The interfaces to the algorithms included in this library have been combined into a single public interface header file for each respective sub-library, i.e. *mlib.h*, *gplib.h*, *gdplib.h* and *gmplib.h*. This was done to simplify the number of files required for inclusion by application programs. Refer to the specific algorithm sections of this document for details on the software Application Programming Interface (API), definitions and functionality provided for the individual algorithms.

## 1.5 MATLAB Integration

Automotive Math and Motor Control Library Set for NXP S32K14x devices provides Bit Accurate Models (BAM) for all library functions. These models can be used in MATLAB Simulink to simulate the behavior of each function in real implementation as well as generate target code using the Embedded Coder®. The BAMs come in a common library file *ammclib\_bam\_S32K14x.slx* located in the "bam" folder in the library installation path. In order to get the BAMs working in Simulink, it is necessary to add the "bam" folder and all of its subfolders into the MATLAB Search Path. The Bit Accurate Models

were compiled and tested using MATLAB R2019a and MinGW 6.3 C compiler. Only C language is supported for model-based code generation.

BAMs of functions with internal state can be configured to make the state variables accessible as input/output signals for debugging purposes. Note that in this configuration the performance is limited and the user must manually connect all feedback loops, i.e. add a "Unit delay" block between a state signal input (e.g. *f32Acc*) and the corresponding state signal output (e.g. *f32Acc\_Out*). The internal state is automatically reset during model initialization. Some BAMs allow setting the internal state to a predefined nonzero value during the reset. This feature can be used for seamless transition from an open loop control to a closed loop control in applications such as the sensorless field-oriented control (FOC).

Several BAMs use nonvirtual buses for efficient representation of struct signal types. The definitions of the data types required by Simulink are automatically loaded from the file *SWLIBS\_Typedefs.mat* located in the "bam" folder. A nonvirtual bus can be created from individual signals using the "Bus Creator" block with appropriate setting of the "Output data type" (e.g. *Bus: SWLIBS\_2Syst\_F16*) and checking the option "Output as nonvirtual bus". Individual signals can be extracted from a nonvirtual bus using the "Bus Selector" block. Refer to the description of functions API in the following chapters for definitions of the input/output data types of individual library functions.

BAMs of matrix/vector functions expect native Simulink matrix/vector signals in column-major arrangement on their signal inputs. The matrix/vector size defined for these signals in Simulink is automatically propagated in the underlying C function as an API parameter. Functions with complex-valued inputs/outputs use complex signal type in the Simulink interface. The underlying C code recasts the complex signals into an array of interleaved real/imaginary pairs.

The BAMs contain sources auto-generated by MATLAB®. © 1984 - 2019 The MathWorks, Inc.

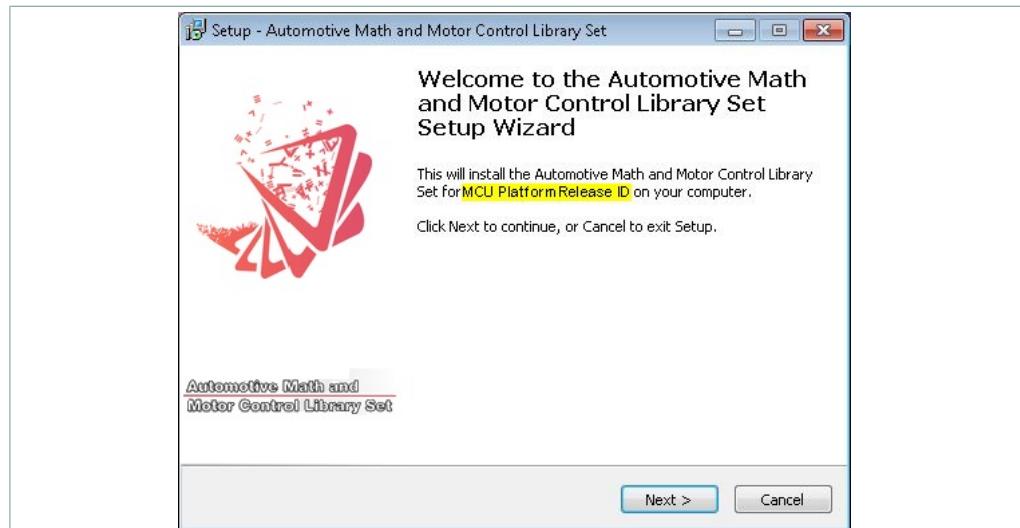
## 1.6 Installation

The Automotive Math and Motor Control Library Set for NXP S32K14x devices is delivered as a single executable file.

**Note:** *The Automotive Math and Motor Control Library Set for NXP S32K14x devices is preinstalled with S32 Design Studio IDE for Arm based MCUs, therefore installation step can be skipped when using this toolchain.*

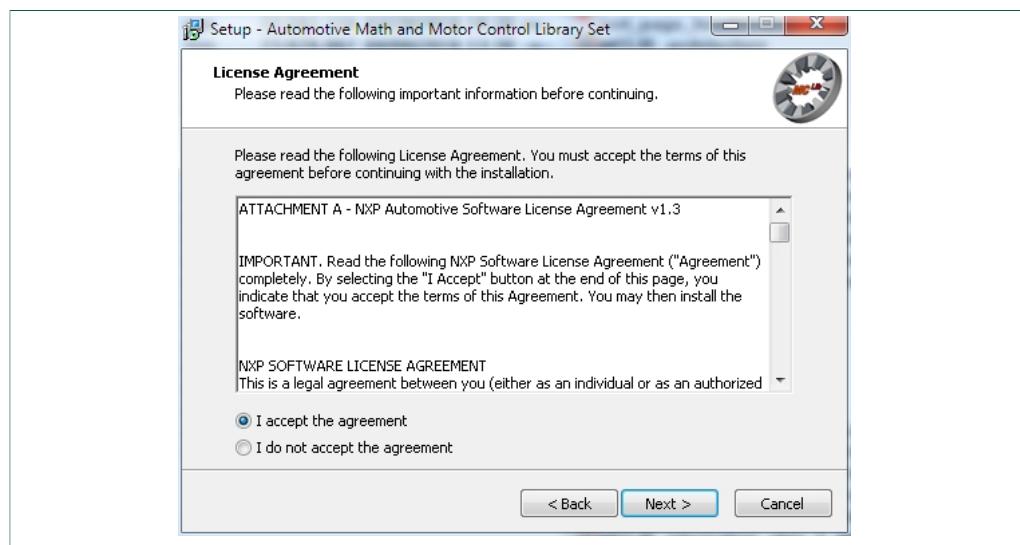
To install the Automotive Math and Motor Control Library Set for NXP S32K14x devices on a user computer, it is necessary to run the installation file and follow these steps:

1. On welcome page select the Next to start the installation



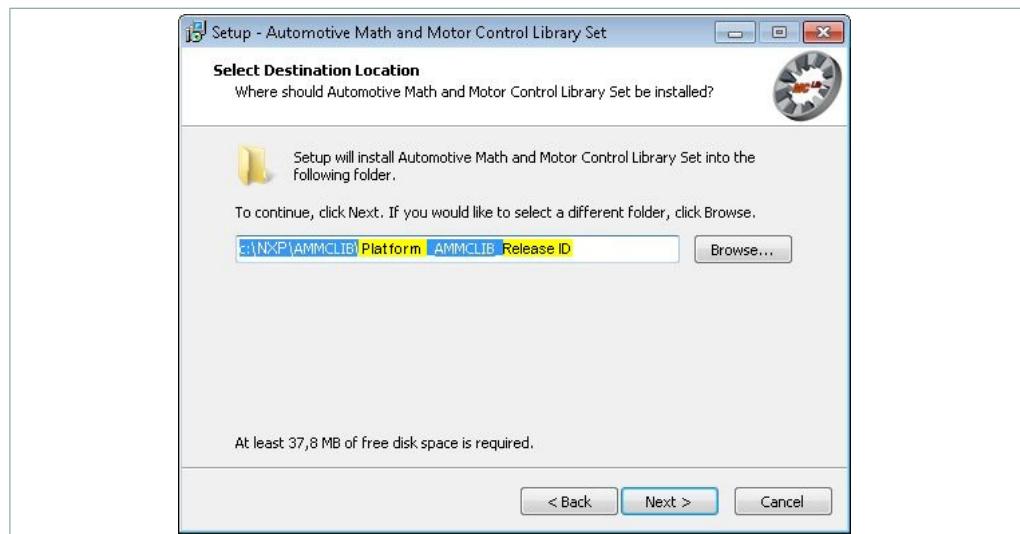
**Figure 2. AMMCLib installation - step 1. Highlighted "MCU Platform" and "ReleaseID" identifies the actual release, which is the [S32K14x\\_AMMCLIB\\_v1.1.21](#)**

2. Accept the license agreement



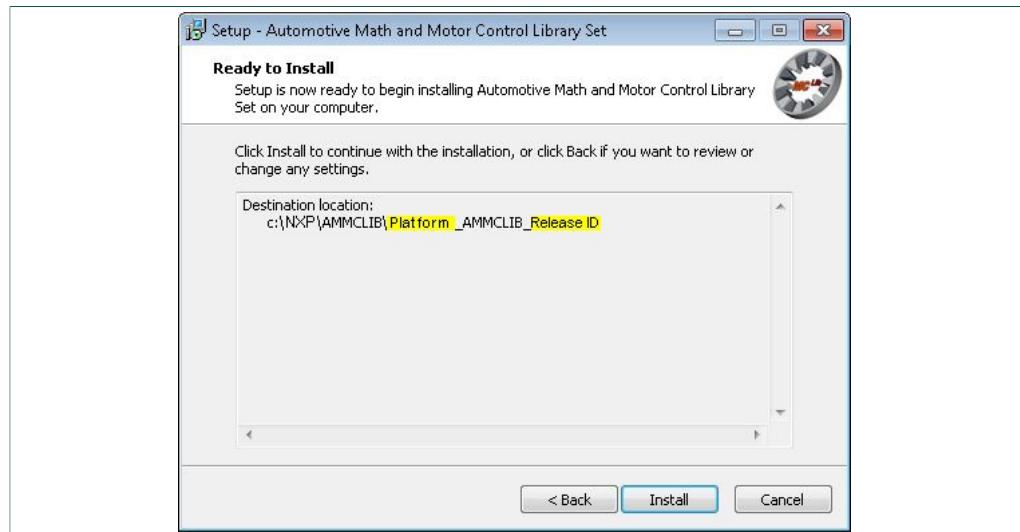
**Figure 3. AMMCLib installation - step 2**

3. Select the destination directory



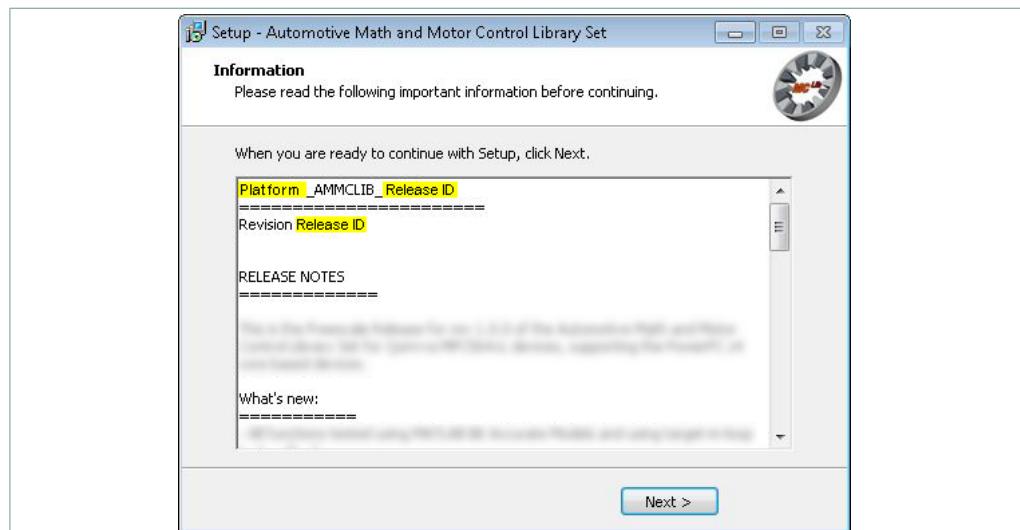
**Figure 4. AMMCLib installation - step 3. Highlighted "Platform" and "ReleaseID" identifies the actual release installation path, which is the **S32K14x\_AMMCLIB\_v1.1.21****

4. Check the destination directory and confirm the installation



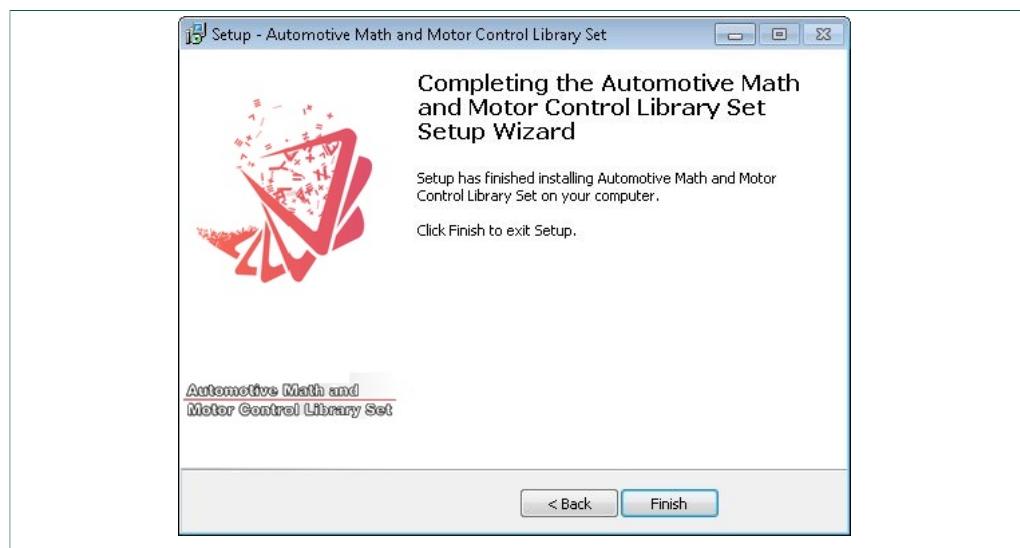
**Figure 5. AMMCLib installation - step 4. Highlighted "Platform" and "ReleaseID" identifies the actual release installation path, which is the **S32K14x\_AMMCLIB\_v1.1.21****

5. After installation carefully read the Release notes with important additional information about the release



**Figure 6. AMMCLib installation - step 5. Highlighted "Platform" and "ReleaseID" identifies the actual release, which is the S32K14x\_AMMCLIB\_v1.1.21**

6. Select Finish to end the installation



**Figure 7. MCLib installation - step 6**

## 1.7 Library File Structure

After a successful installation, the Automotive Math and Motor Control Library Set for NXP S32K14x devices is added by default into the "C:\NXPIAMMCLIB\|S32K14x\_AMMCLIB\_v1.1.21" subfolder. This folder will contain other nested subfolders and files required by the Automotive Math and Motor Control Library Set for NXP S32K14x devices, as shown in [Figure 8](#).

Name	Ext	Size
[..]	<DIR>	
[bam]	<DIR>	
[doc]	<DIR>	
[include]	<DIR>	
[lib]	<DIR>	
license	txt	14,522

Figure 8. AMMCLIB directory structure

A list of the installed directories/files, and their brief description, is given below:

- **bam** - contains Bit Accurate Models of all the functions for Matlab/Simulink
- **doc** - contains the User Manual
- **include** - contains all the header files, including the master header files of each library to be included in the user application
- **lib** - contains the compiled library file to be included in the user application
- **src** - contains all source files

In order to integrate the Automotive Math and Motor Control Library Set for NXP S32K14x devices into a new Green Hills compiler based project, the steps described in [Section 1.9](#) section must be performed.

For integration of the Automotive Math and Motor Control Library Set for NXP S32K14x devices into a new IAR based project, the steps described in section [Section 1.10](#) must be performed.

The header files structure of the Automotive Math and Motor Control Library Set for NXP S32K14x devices is depicted in [Figure 9](#).

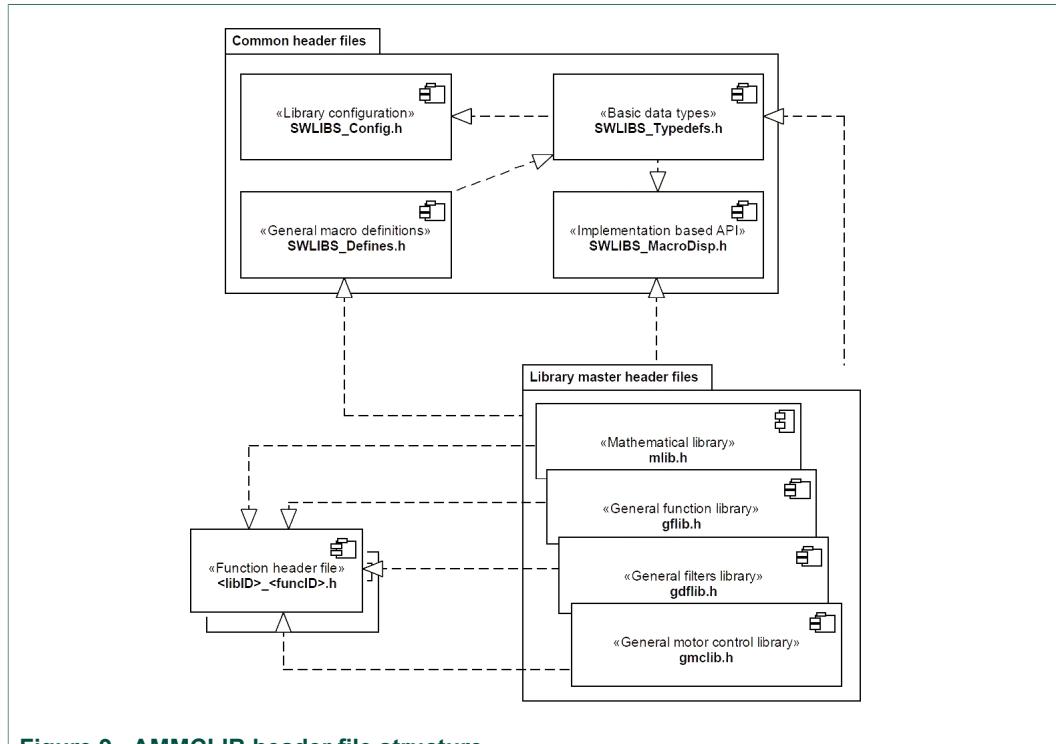


Figure 9. AMMCLIB header file structure

## 1.8 Integration Assumption

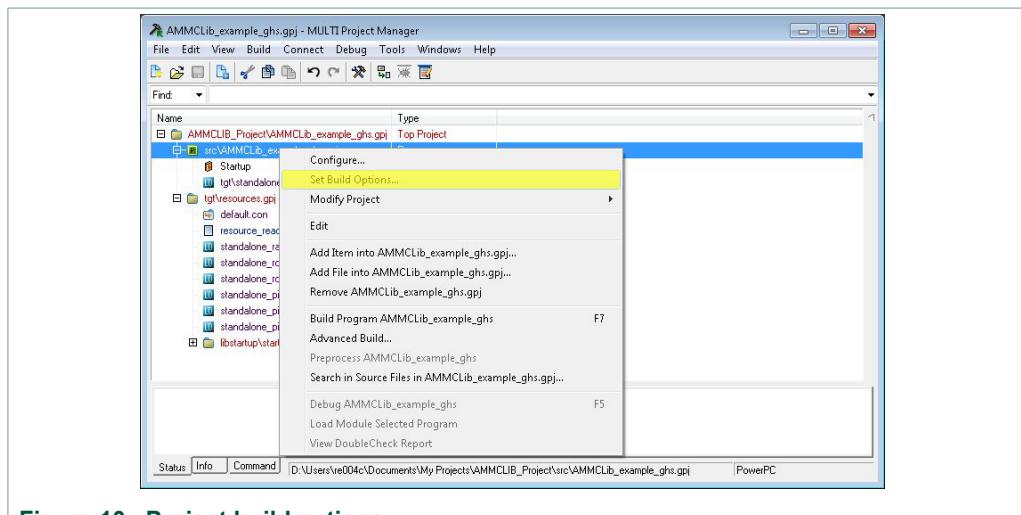
In order to successfully integrate the Automotive Math and Motor Control Library Set for NXP S32K14x devices to the customer application, the following assumptions need to be considered:

1. The C-99 language has to be enabled in the user application (please refer to User manual of your compiler to enable this feature).
2. In case the floating point unit is available on target MCU, the single precision floating point HW support has to be switched on (please refer to User manual of your compiler to enable this feature).

## 1.9 Library Integration into a Green Hills Multi Development Environment

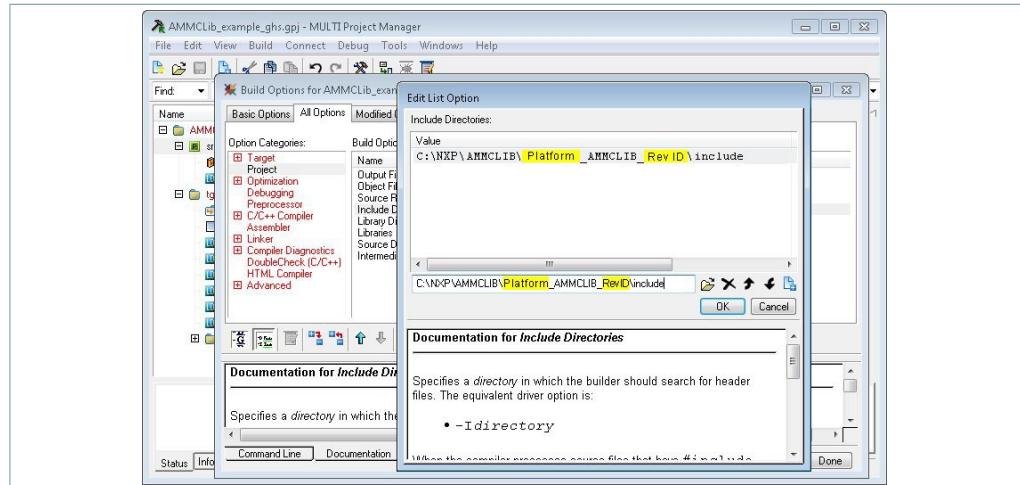
The Automotive Math and Motor Control Library Set for NXP S32K14x devices is added into a new Green Hills Multi project using the following steps:

1. Open a new empty C project in the Green Hills Multi IDE. See the Green Hills Multi user manual for instructions.
2. Once you have successfully created and opened a new C project, right click on the project file \*.gpj in the GHS Multi Project Manager. Select <Set Build Options...> from the pop-up menu, as shown in [Figure 10](#)



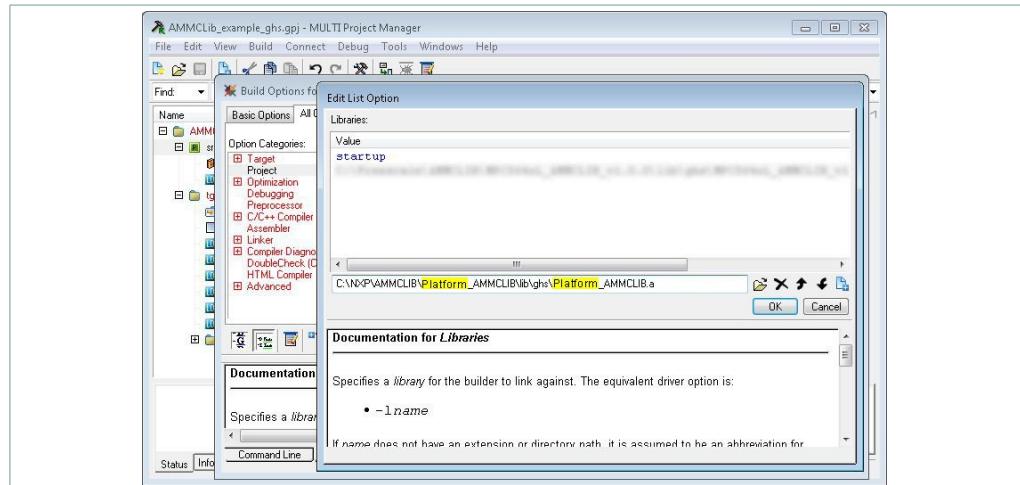
**Figure 10. Project build options**

3. In the <Basic Options> tab of the <Build Options> window, expand section <Project> and double click on the item <Include Directories (-I)>. Here, the directory where the builder should look for all the project header files, including the library header files, shall be specified as shown in [Figure 11](#). Considering default settings, the following path shall be added: "C:\NXP\AMMCLIB\S32K14x\_AMMCLIB\_v1.1.21\include."



**Figure 11. Adding a path to the library header files**

4. While still in the <Project> section, double click on <Libraries (-l)> and enter a path and a pre-compiled library object file into the dialogue box, as shown in [Figure 12](#).



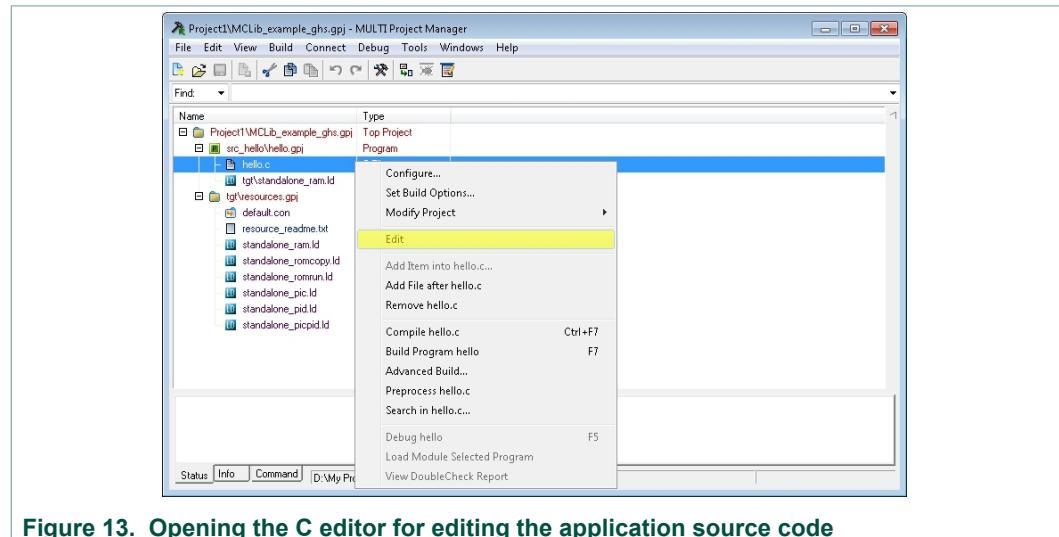
**Figure 12. Adding a path to the library object files**

Considering default settings, you should add "C:\NXP\AMMCLIB\IS32K14x\_AMMCLIB\_v1.1.21\lib\ghs\S32K14x\_AMMCLIB.a"

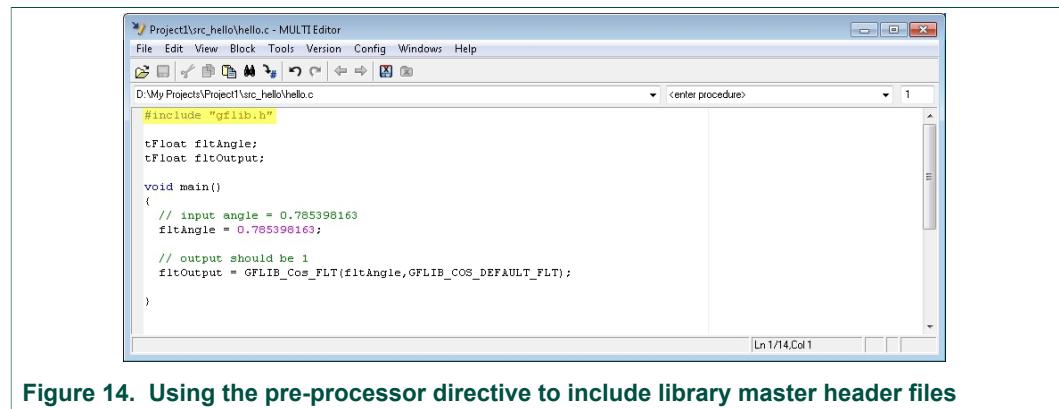
5. In order to use the library functions, the library master header files must be included into the application source code. This is done using the pre-processor directive `\#include "<libID>.h"`, where <libID> can be `amclib`, `gdflib`, `gplib`, `gmclib` depending on which library is to be employed.

The master header files contain several additional header files that are needed for the Automotive Math and Motor Control Library Set for NXP S32K14x devices integration into any user application. They include the `"SWLIBS_Typedefs.h"` header file which contains all general purpose data type definitions, the `"mlib.h"` header file containing all general math functions, the `"SWLIBS_Defines.h"` file containing common macro definitions and the `"SWLIBS_MacroDisp.h"` allowing the implementation based API call.

**Note:** Remember that by default there is no default implementation selected in the "SWLIBS\_Config.h" thus the error message will be displayed during the compilation requesting the default implementation selection.



**Figure 13. Opening the C editor for editing the application source code**



**Figure 14. Using the pre-processor directive to include library master header files**

At this point, the Automotive Math and Motor Control Library Set for NXP S32K14x devices is linked with the user project file, and hence the library functions can be exploited and flawlessly compiled/linked with the user application.

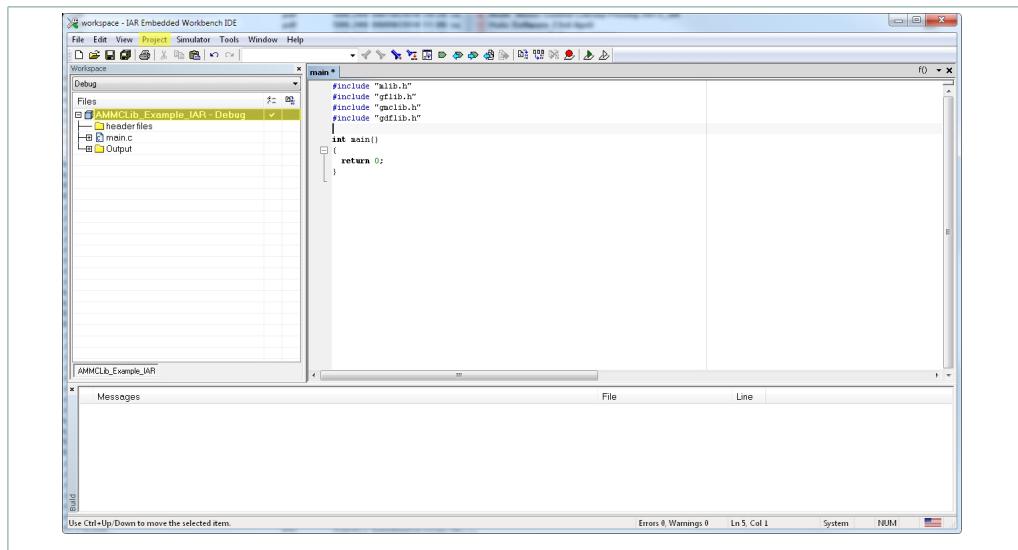
## 1.10 Library Integration into a IAR Project

The Automotive Math and Motor Control Library Set for NXP S32K14x devices is added into a IAR project using the following steps:

1. Open a new empty C project in the IAR Embedded Workbench IDE. See the IAR Embedded Workbench Getting Started manual for instructions.
2. Once having the new project successfully created and opened, the Automotive Math and Motor Control Library Set for NXP S32K14x devices files needs to be added to the project.
3. In order to integrate the Automotive Math and Motor Control Library Set for NXP S32K14x devices in the IAR project, it is necessary to provide the IAR Embedded

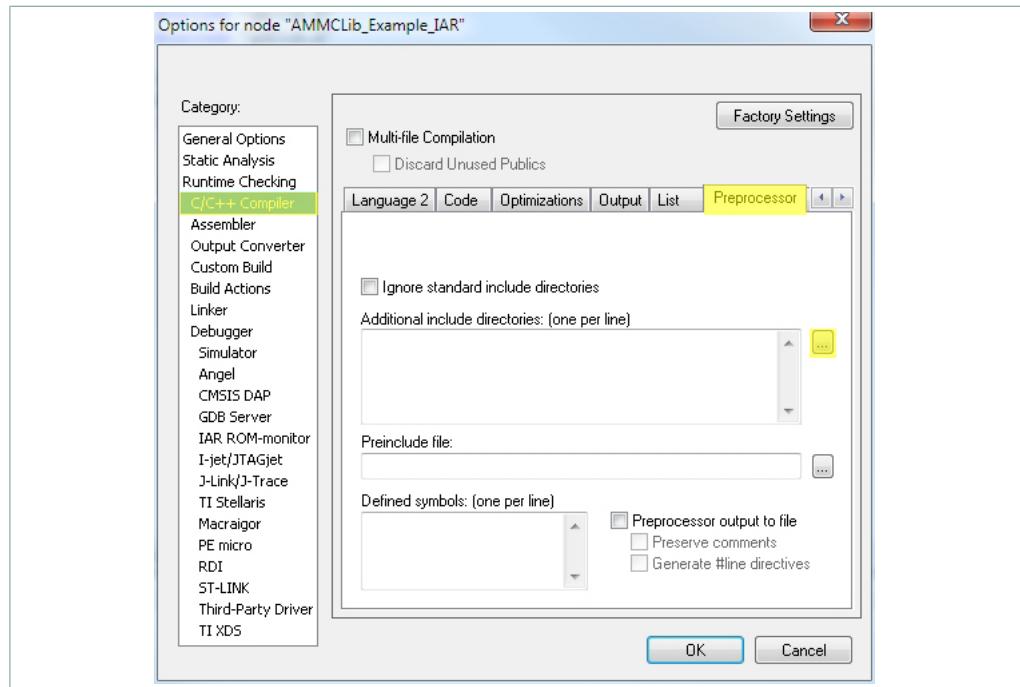
Workbench IDE with the access paths to the Automotive Math and Motor Control Library Set for NXP S32K14x devices files. Two access paths needs to be added:

- a. Access path to the Automotive Math and Motor Control Library Set for NXP S32K14x devices header files.
- b. Access path to the Automotive Math and Motor Control Library Set for NXP S32K14x devices binary file.
4. To add the necessary access paths, select the project from the workspace list (**AMMCLib\_Example\_IAR** in the example shown in [Figure 15](#)) and from the main menu select <Project> - <Options> item.

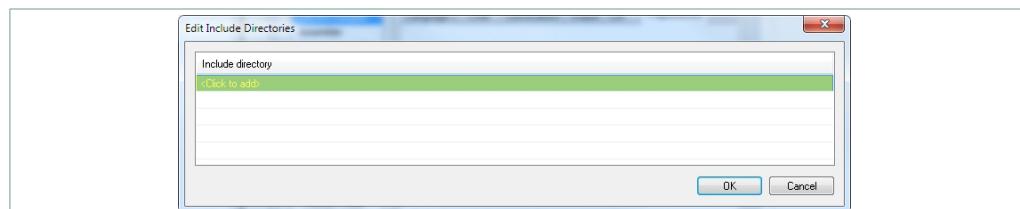


**Figure 15. Selecting the project properties**

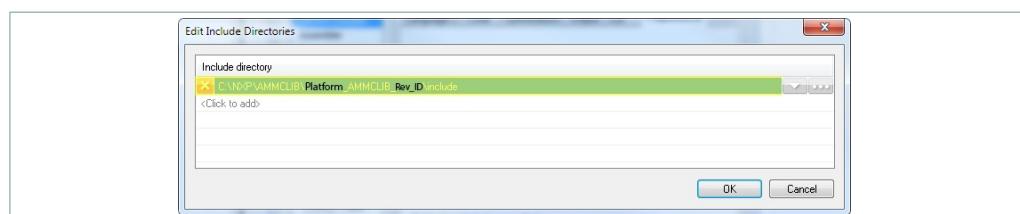
5. To add the Automotive Math and Motor Control Library Set for NXP S32K14x devices header files, in the <Options for node> window select the <C/C++ Compiler> category. Here, choose the <Preprocessor> tab and on the right side of <Additional include directories:> click on the dot symbols, as shown in [Figure 16](#).

**Figure 16. Adding the include directory**

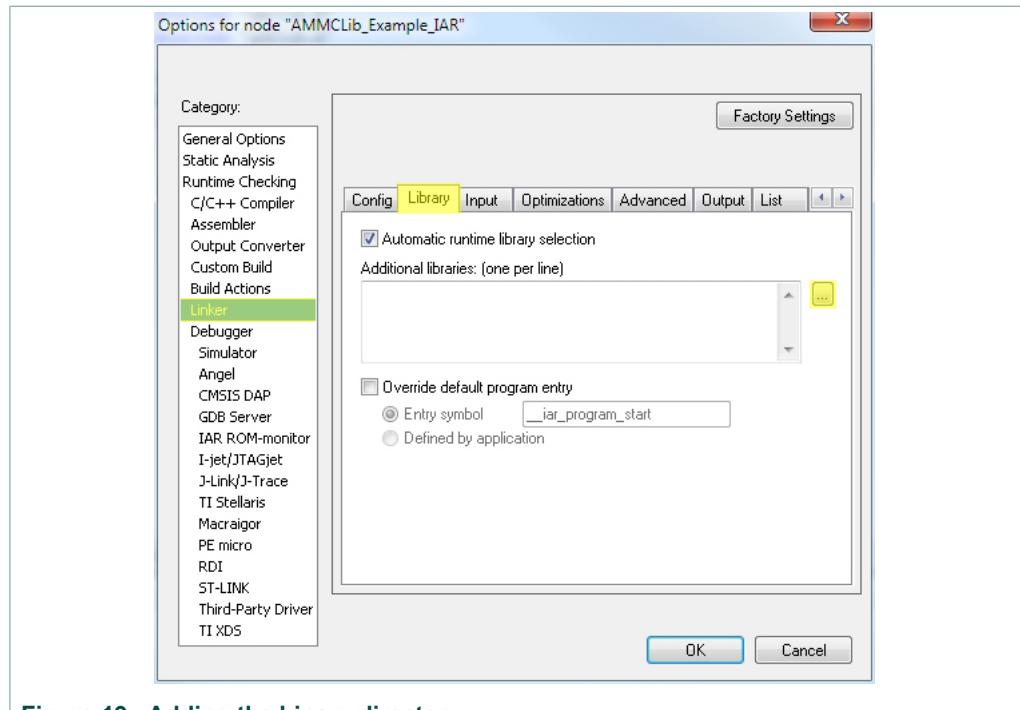
6. In the <Edit Include Directories> click on the <Click to add> as shown in [Figure 17](#)

**Figure 17. Adding the include directory to the list**

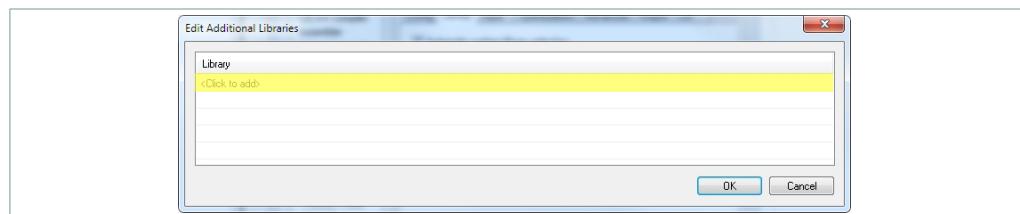
7. Considering the default paths, the <Edit Include Directories> should look as shown in [Figure 18](#)

**Figure 18. Adding the default include path**

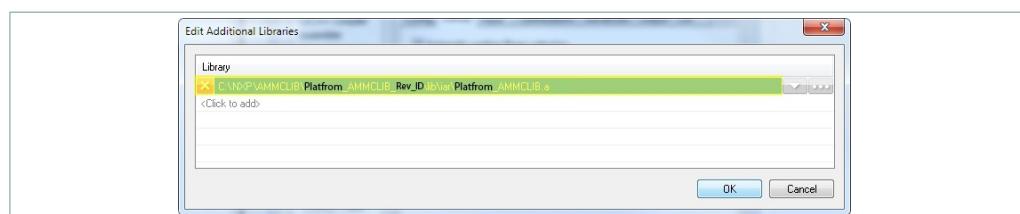
8. While still in the <Options for node> window, add the Automotive Math and Motor Control Library Set for NXP S32K14x devices binary file. In the <Options for node> window select the <Linker> category. Here, choose the <Library> tab and on the right side of <Additional libraries:> click on the dot symbols, as shown in [Figure 19](#).

**Figure 19. Adding the binary directory**

9. In the <Edit Additional Libraries> click on the <Click to add> as shown in [Figure 20](#)

**Figure 20. Adding the binary directory to the list**

10. Considering the default paths, the <Edit Additional Libraries> should look as shown in [Figure 21](#)

**Figure 21. Adding the default binary path**

11. In order to use the library functions, the library master header files must be included into the application source code. This is done using the pre-processor directive `#include "<libID>.h"`, where <libID> can be `amclib`, `gdflib`, `gflib`, `gmclib` depending on which library is to be employed.

The master header files contain several additional header files that are needed for the Automotive Math and Motor Control Library Set for NXP S32K14x devices integration into any user application. They include the "SWLIBS\_Typedefs.h" header file which contains all general purpose data type definitions, the "mlib.h" header file containing all general

math functions, the "SWLIBS\_Defines.h" file containing common macro definitions and the "SWLIBS\_MacroDisp.h" allowing the implementation based API call.

**Note:** Remember that by default there is no default implementation selected in the "SWLIBS\_Config.h" thus the error message will be displayed during the compilation requesting the default implementation selection.

At this point, the Automotive Math and Motor Control Library Set for NXP S32K14x devices is linked with the user project file, and hence the library functions can be exploited and flawlessly compiled/linked with the user application.

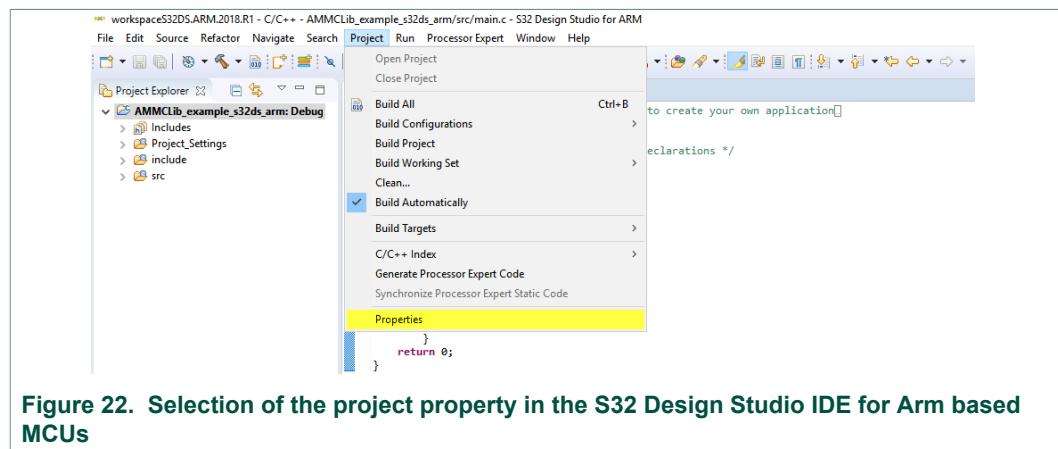
## 1.11 Library Integration into a S32 Design Studio IDE for Arm based MCUs

In order to use the Automotive Math and Motor Control Library Set for NXP S32K14x devices within a S32 Design Studio for Arm based MCUs GUI environment, it is necessary to provide the S32 Design Studio GUI with access paths to the Automotive Math and Motor Control Library Set for NXP S32K14x devices files. The following files shall be added to the user project:

- Library binary files located in the directory "C:\NXP\AMMCLIB\|S32K14x\_AMMCLIB\_v1.1.21\lib\s32ds\_arm32" (note: this is the default location and may be modified during library installation)
- Header files located in the directory "C:\NXP\AMMCLIB\|S32K14x\_AMMCLIB\_v1.1.21\include" (note: this is the default location and may be modified during library installation)

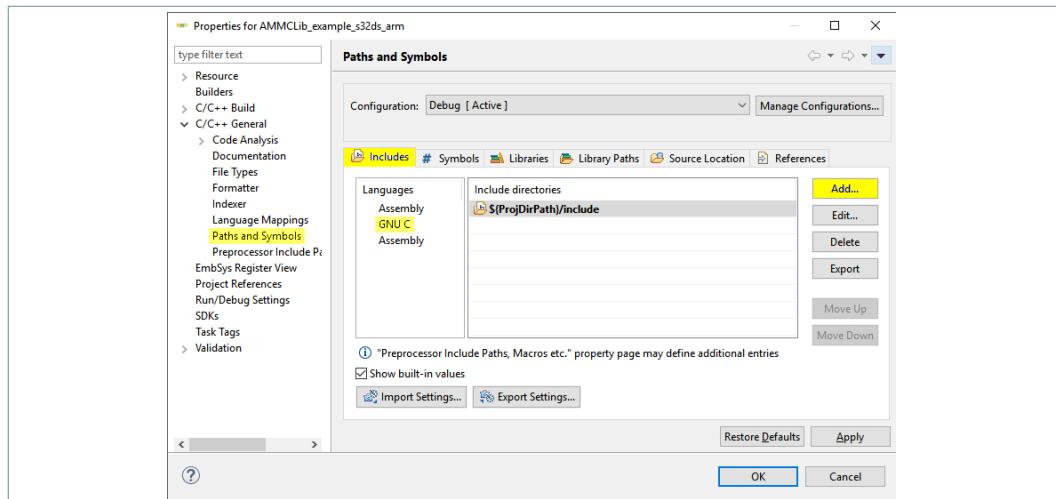
**Note:** When creating new S32 Design Studio project make sure the ARM Bare-Metal 32-bit Target Binary Toolchain is selected as well as the NewLib standard library.

To add these files, select your project in the <Project Explorer> in the left tab, and from the <Project> menu select the <Properties> option as described in [Figure 22](#).



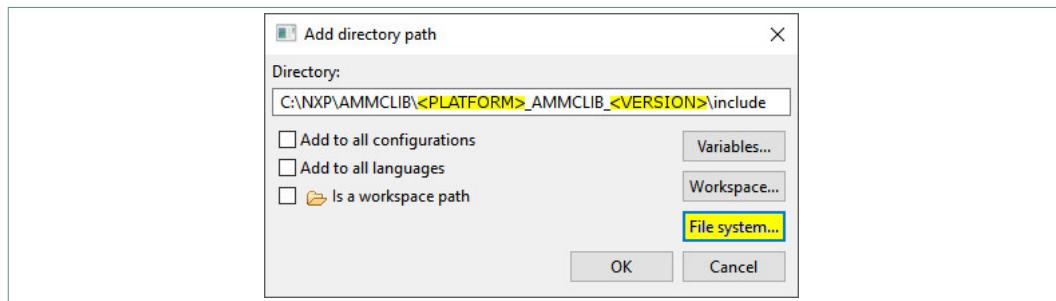
**Figure 22. Selection of the project property in the S32 Design Studio IDE for Arm based MCUs**

In the <Properties> window, select the <C/C++ General><Paths and Symbols> from the left-hand list and choose the <Includes> tab and <GNU C> under the <Languages> section, as described in [Figure 23](#).



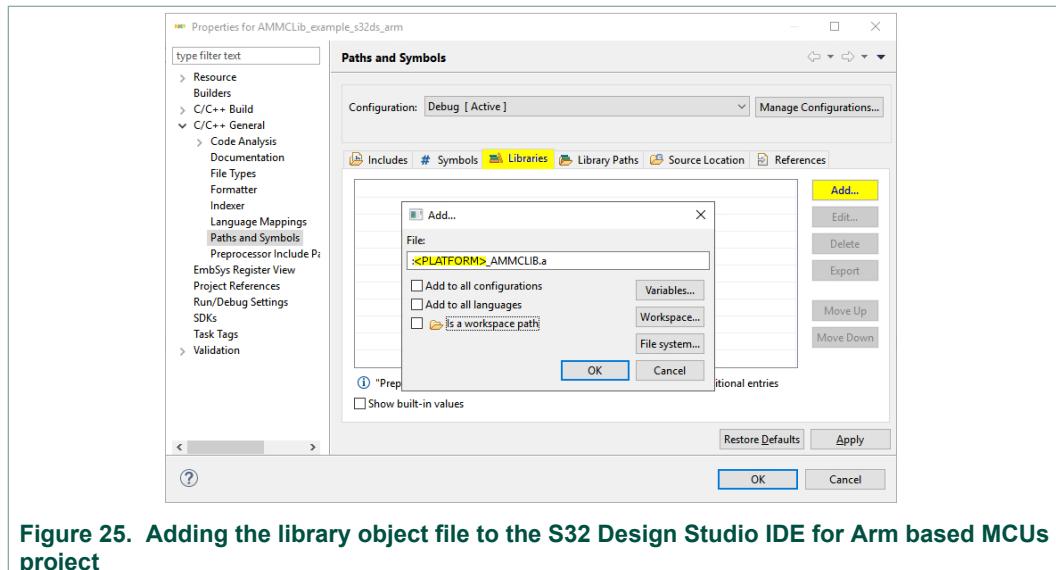
**Figure 23. Selecting the include paths in the S32 Design Studio IDE for Arm based MCUs**

By selecting the <Add..> button, add the directories, where the builder should look for all the header files. Considering the default settings, the path "C:\NXP\AMMCLIB\S32K14x\_AMMCLIB\_v1.1.21\include" shall be added.



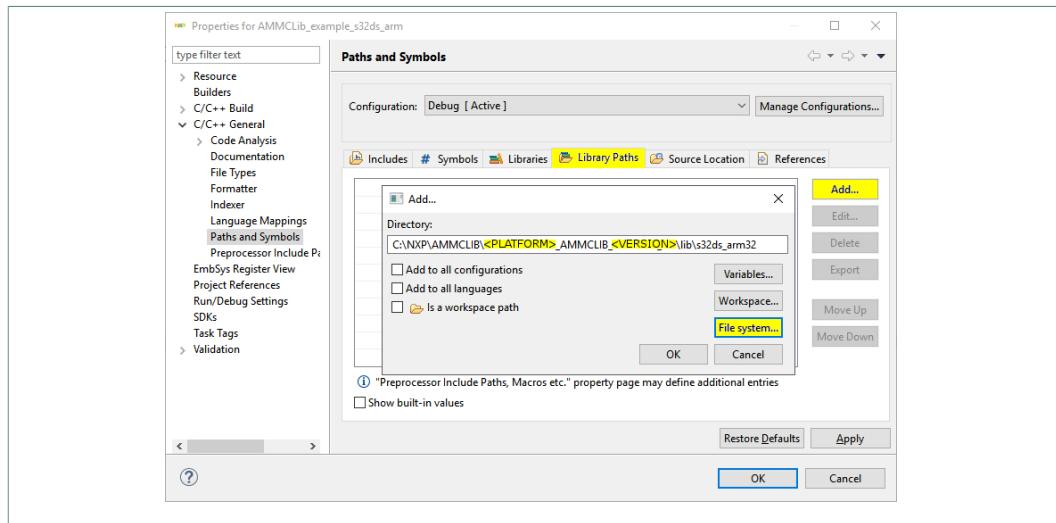
**Figure 24. Adding the header files path to the S32 Design Studio IDE for Arm based MCUs project**

After adding of the Automotive Math and Motor Control Library Set for NXP S32K14x devices header files path, the library object file shall be add to the S32 Design Studio for Arm based MCUs project. In the <Libraries> tab, select the <Add..> button and add the library object file ":S32K14x\_AMMCLIB.a", as described in [Figure 25](#). Be aware, that colon must be added as a prefix of the library name.



**Figure 25.** Adding the library object file to the S32 Design Studio IDE for Arm based MCUs project

In the next tab <Library Paths> select the <Add..> and add the directory, where the builder should look for the object file from previous step. Considering the default settings, the path "C:\NXP\AMMCLIB\S32K14x\_AMMCLIB\_v1.1.21\lib\s32ds\_arm32" shall be added.



**Figure 26.** Adding the library object file path to the S32 Design Studio IDE for Arm based MCUs project

**Note:** In S32 Design Studio IDE for Arm based MCUs it is possible to include Automotive Math and Motor Control Library Set for NXP S32K14x devices as internal SDK, but version of the library installed together with S32 Design Studio IDE can be obsolete. So it is recommended to always use standalone installation of the latest version of the Automotive Math and Motor Control Library Set for NXP S32K14x devices and add this library to the S32 Design Studio project by procedure described above.

In order to use the library functions, the library master header files must be included into the application source code. This is done using the pre-processor directive `\#include "<libID>.h"`, where `<libID>` can be `amclib`, `gdflib`, `gflib`, `gmclib` `mlib`, depending on which library is to be employed.

The master header files contain several additional header files that are needed for the Automotive Math and Motor Control Library Set for NXP S32K14x devices integration into any user application. They include the "SWLIBS\_Typedefs.h" header file which contains all general purpose data type definitions, the "mlib.h" header file containing all general math functions, the "SWLIBS\_Defines.h" file containing common macro definitions and the "SWLIBS\_MacroDisp.h" allowing the implementation based API call.

*Note: Remember that, there is no default implementation selected in the "SWLIBS\_Config.h" by default, thus the error message will be displayed during the compilation requesting the default implementation selection.*

At this point, the Automotive Math and Motor Control Library Set for NXP S32K14x devices is linked with the user project file, and hence the library functions can be exploited and flawlessly compiled/linked with the user application.

## 1.12 Library Testing

In order to validate the implementation of the Automotive Math and Motor Control Library Set for NXP S32K14x devices, the comparison of results from the Matlab Reference Model and outputs from the tested library function is used. To ensure the Automotive Math and Motor Control Library Set for NXP S32K14x devices precision, two test methods are used:

- Matlab Simulink Toolbox based testing (refer to [Section "AMMCLIB Testing based on the Matlab Simulink Toolbox"](#) for more details).
- Target-in-loop based testing (refer to [Section "AMMCLIB target-in-loop Testing based on the SFIO Toolbox"](#) for detailed information).

The [Figure 27](#) shows the testing principle:

- **Input vector** represents the test vector which enters simultaneously into the Reference Model and into the Unit Under Test (UUT).
- **Reference Model (RM)** implements the real model of the UUT. For simple functions, the models are a part of the Matlab Simulink Toolbox. Advanced functions such as filters or controllers had been designed separately.
- By the type of test method used, the **Unit Under Test (UUT)** may be:
  - **Bit Accurate Model (BAM)** - the "C" implementation of the tested function compiled in the Matlab environment. The compilation result, called the binary MEX-file, is a dynamically-linked subroutine that the Matlab interpreter can load and execute.
  - **SFIO Model** represents the tested function running directly on the target MCU.

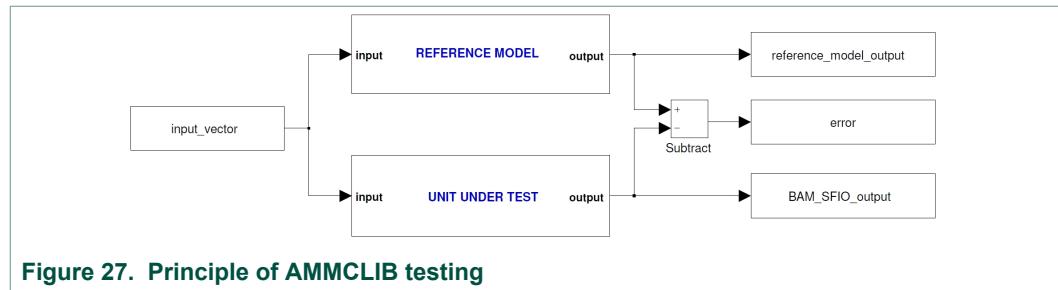
Results from the UUT and Reference Model are saved in the final report, together with the calculated error which is simply the difference between the output value from the Reference Model and the output value from the UUT, recalculated to an equivalent precision.

The output value of the function is determined by the following expressions:

- **32-bit fixed point:** <PRECISE\_VALUE> +/- e\*2^-15
- **16-bit fixed point:** <PRECISE\_VALUE> +/- e\*2^-15

where e is the allowed approximation error (see [Section 1.13](#) for specific values).

For single precision floating point implementations, the output accuracy is measured in ULP (units in the last place) and is specified in a separate document S32K14XMCFLTACC.pdf.

**Figure 27. Principle of AMMCLIB testing**

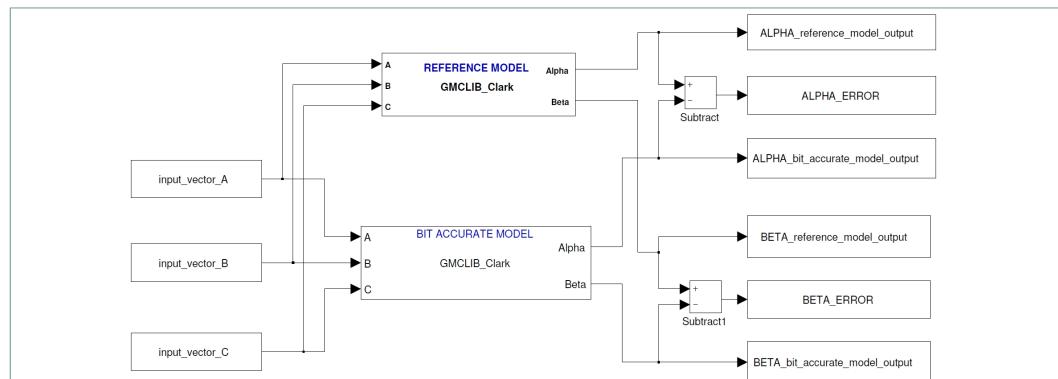
In order to test the UUT under all conditions, three types of test vector sets are used:

- Deterministic vectors - a specifically defined set of input values over the entire input range.
- Stochastic vectors - a pseudo-randomly generated set of values (non-deterministic values fully covering the input range).
- Boundary vectors - a set of input values for which the potential weaknesses of the tested function are expected. This test is performed only on functions where these limit conditions might occur.

Each function is considered tested if the required accuracy during deterministic, stochastic and boundary tests has been achieved. The following two subchapters describe the differences between AMMCLIB testing based on BAM models and target-in-loop testing based on SFIO models.

### AMMCLIB Testing based on the Matlab Simulink Toolbox

An example of the testing principle based on the BAM is depicted in the Clark transformation function ([Figure 28](#)). The Bit Accurate Model contains the binary MEX-file built from the GMCLIB\_Clark function using the Matlab compiler. This file is called inside the BAM model, see [Figure 29](#). The Reference Model of the Clark transformation is not included in the Matlab Simulink Toolbox and hence its mathematical representation had to be created. A detailed scheme of the Clark RM is in [Figure 30](#).

**Figure 28. Testing of the GMCLIB\_Clark function based on the Matlab Simulink Toolbox**

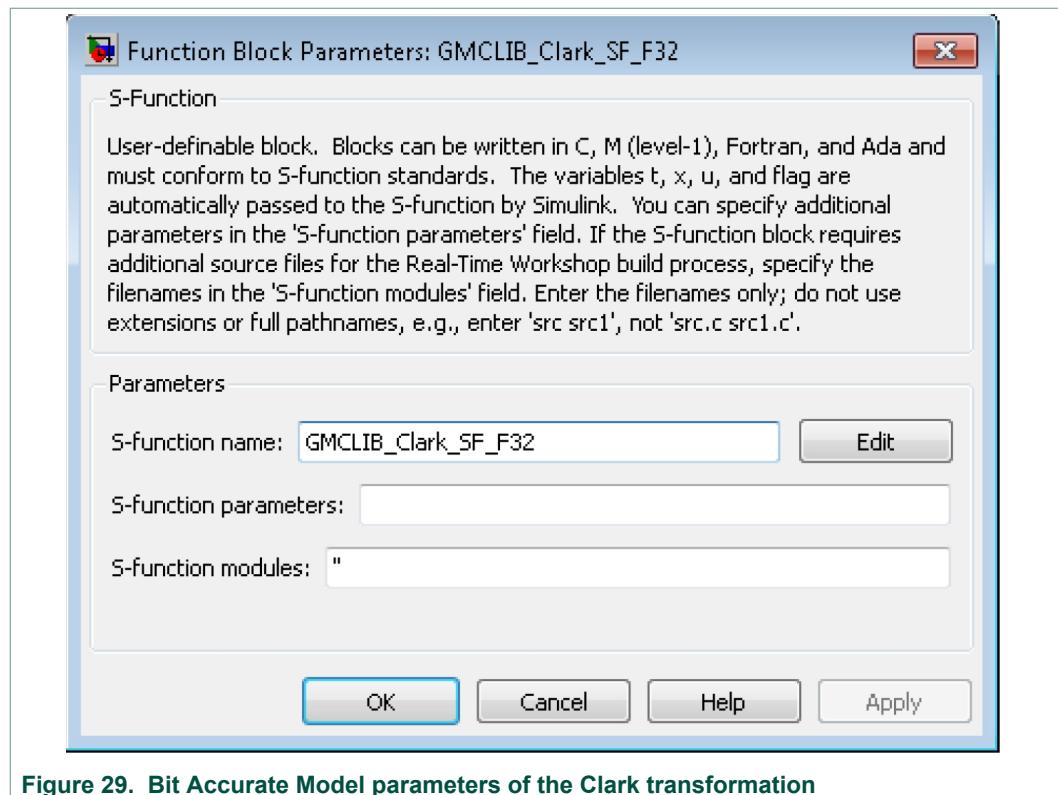


Figure 29. Bit Accurate Model parameters of the Clark transformation

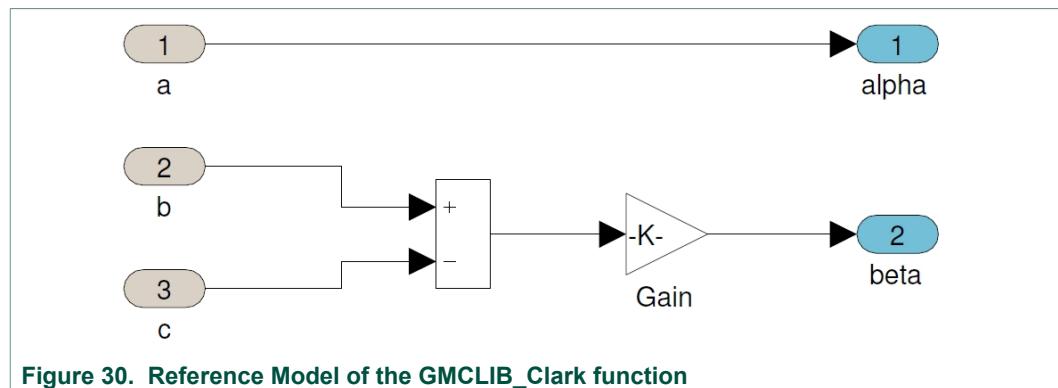


Figure 30. Reference Model of the GMCLIB\_Clark function

### AMMCLIB target-in-loop Testing based on the SFIO Toolbox

The testing method in [Figure 31](#) is similar to that described in the previous chapter with exception that the BAM model is replaced by the SFIO model. The SFIO Toolbox realizes the bridge between Matlab and the Embedded target. During testing, the function GMCLIB\_Clark is called directly from the application running on the target MCU. Unlike testing based on Matlab, the target-in-loop method verifies that the implementation of the Automotive Math and Motor Control Library Set for NXP S32K14x devices functions works correctly on the target MCU. Moreover, the SFIO application running on the processor is used to measure performance of the functions.

The SFIO block Set-up allows the setting of communication parameters which are common to the whole scheme.

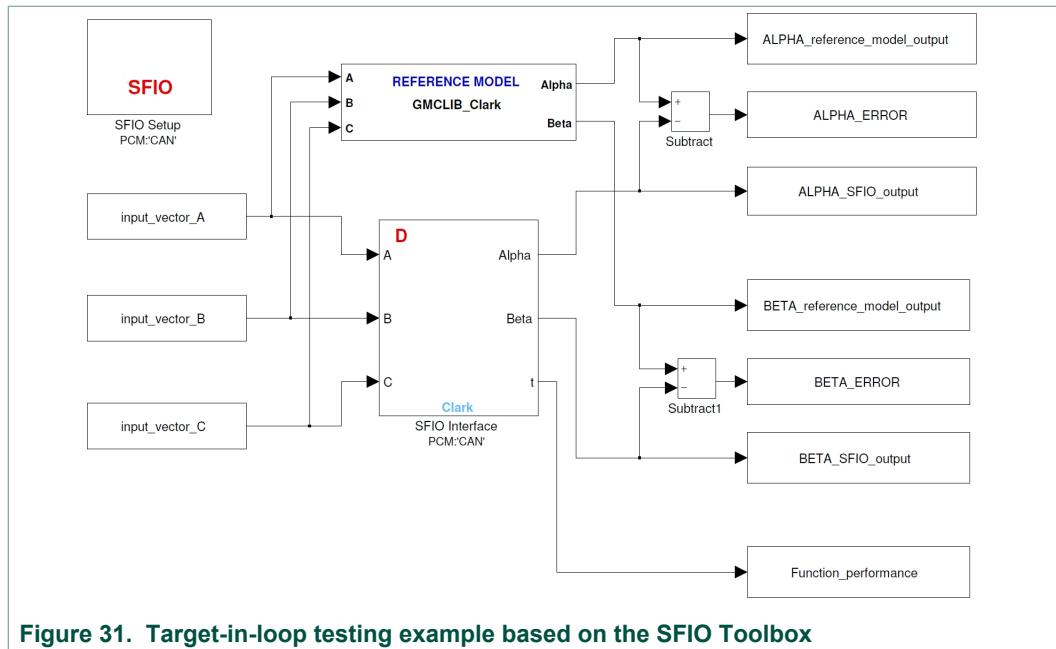


Figure 31. Target-in-loop testing example based on the SFIO Toolbox

## 1.13 Functions Accuracy

The maximum allowed error of the library functions vary based on the implementation, and based on the character of the function. The output error is calculated as the difference between the implemented function and a double-precision reference model, as described in the [Section 1.12](#) section. The actual output error of the function for current input values is smaller or equal to the maximum allowed error described in the table below.

The output error of the 32-bit and 16-bit fixed-point implementations is calculated as an absolute error.

Accuracy of the floating-point functions is described in a separate document S32K14XMCFLTACC.pdf.

The following table describes the maximum absolute error of the fixed-point functions measured in the 16-bit/32-bit LSB multiples (LSB16/LSB32):

**Table 1. Maximum absolute error**

Function name	F32 variant	F16 variant
GDFLIB_FilterFIR	u32Order+1 LSB32	u16Order+1 LSB16
GDFLIB_FilterIIR1	1 LSB16	1 LSB16
GDFLIB_FilterIIR2	1 LSB16	1 LSB16
GDFLIB_FilterMA	1 LSB16	1 LSB16
GFLIB_Acos	3 LSB16	3 LSB16
GFLIB_Asin	3 LSB16	3 LSB16
GFLIB_Atan	3 LSB16	3 LSB16
GFLIB_AtanYX	3 LSB16	3 LSB16
GFLIB_AtanYXShifted	3 LSB16	3 LSB16
GFLIB_ControllerPlp	3 LSB16	3 LSB16
GFLIB_ControllerPlpAW	3 LSB16	3 LSB16
GFLIB_ControllerPlr	3 LSB16	3 LSB16

Function name	F32 variant	F16 variant
GFLIB_ControllerPIrAW	3 LSB16	3 LSB16
GFLIB_Cos	3 LSB16	3 LSB16
GFLIB_Hyst	1 LSB16	1 LSB16
GFLIB_IntegratorTR	3 LSB16	3 LSB16
GFLIB_Limit	1 LSB16	1 LSB16
GFLIB_LowerLimit	1 LSB16	1 LSB16
GFLIB_Lut1D	3 LSB16	3 LSB16
GFLIB_Lut2D	3 LSB16	3 LSB16
GFLIB_Ramp	1 LSB16	1 LSB16
GFLIB_Sign	1 LSB16	1 LSB16
GFLIB_Sin	3 LSB16	3 LSB16
GFLIB_SinCos	3 LSB16	3 LSB16
GFLIB_Sqrt	1 LSB16	3 LSB16
GFLIB_Tan	3 LSB16	3 LSB16
GFLIB_UpperLimit	1 LSB16	1 LSB16
GFLIB_VectorLimit	3 LSB16	3 LSB16
GMCLIB_BetaProjection	5 LSB32	4 LSB16
GMCLIB_BetaProjection3Ph	5 LSB32	4 LSB16
GMCLIB_Clark	1 LSB16	3 LSB16
GMCLIB_ClarkInv	1 LSB16	3 LSB16
GMCLIB_DecouplingPMSM	1 LSB16	3 LSB16
GMCLIB_ElimDcBusRip	3 LSB16	3 LSB16
GMCLIB_Park	2 LSB16	2 LSB16
GMCLIB_ParkInv	1 LSB16	1 LSB16
GMCLIB_PwmIct	1 LSB16	3 LSB16
GMCLIB_SvmStd	1 LSB16	3 LSB16
GMCLIB_SvmU0n	2 LSB32	2 LSB16
MLIB_Abs	1 LSB16	1 LSB16
MLIB_AbsSat	1 LSB16	1 LSB16
MLIB_Add	1 LSB16	1 LSB16
MLIB_AddSat	1 LSB16	1 LSB16
MLIB_Convert_F16F32	N/A	1 LSB16
MLIB_Convert_F16FLT	N/A	1 LSB16
MLIB_Convert_F32F16	1 LSB16	N/A
MLIB_Convert_F32FLT	1 LSB16	N/A
MLIB_Convert_FLTF16	N/A	N/A
MLIB_Convert_FLTF32	N/A	N/A
MLIB_ConvertPU_F16F32	N/A	1 LSB16
MLIB_ConvertPU_F16FLT	N/A	1 LSB16
MLIB_ConvertPU_F32F16	1 LSB16	N/A
MLIB_ConvertPU_F32FLT	1 LSB16	N/A
MLIB_ConvertPU_FLTF16	N/A	N/A
MLIB_ConvertPU_FLTF32	N/A	N/A
MLIB_Div	3 LSB16	1 LSB16
MLIB_DivSat	3 LSB16	1 LSB16
MLIB_Mac	1 LSB16	1 LSB16
MLIB_MacSat	1 LSB16	1 LSB16

Function name	F32 variant	F16 variant
MLIB_Mnac	1 LSB16	1 LSB16
MLIB_Msu	1 LSB16	1 LSB16
MLIB_Mul	1 LSB16	1 LSB16
MLIB_MulSat	1 LSB16	1 LSB16
MLIB_Neg	1 LSB16	1 LSB16
MLIB_NegSat	1 LSB16	1 LSB16
MLIB_Norm	1 LSB16	1 LSB16
MLIB_RndSat	1 LSB16	1 LSB16
MLIB_Round	1 LSB16	1 LSB16
MLIB_ShBi	1 LSB16	1 LSB16
MLIB_ShBiSat	1 LSB16	1 LSB16
MLIB_ShL	1 LSB16	1 LSB16
MLIB_ShLSat	1 LSB16	1 LSB16
MLIB_ShR	1 LSB16	1 LSB16
MLIB_Sub	1 LSB16	1 LSB16
MLIB_SubSat	1 LSB16	1 LSB16
MLIB_VMac	1 LSB16	2 LSB16
AMCLIB_BemfObsrvDQ	24 LSB16	30 LSB16
AMCLIB_TrackObsrv	3 LSB16	3 LSB16
AMCLIB_CurrentLoop	94 LSB16	232 LSB16
AMCLIB_FW	29 LSB16	28 LSB16
AMCLIB_FWSpeedLoop	29 LSB16	28 LSB16
AMCLIB_SpeedLoop	29 LSB16	28 LSB16
AMCLIB_Windmilling	9 LSB16	23 LSB16

## 2 Functions

This section describes the Application Interface and implementation details for all functions available in Automotive Math and Motor Control Library Set for NXP S32K14x devices.

### 2.1 Function index

Table 2. Quick function reference

Type	Name	Arguments
void	<a href="#">AMCLIB_BemfObsrvDQInit_F16</a>	AMCLIB_BEMF_OBSRV_DQ_T_F16 *const pCtrl
void	<a href="#">AMCLIB_BemfObsrvDQInit_F32</a>	AMCLIB_BEMF_OBSRV_DQ_T_F32 *const pCtrl
void	<a href="#">AMCLIB_BemfObsrvDQInit_FLT</a>	AMCLIB_BEMF_OBSRV_DQ_T_FLT *const pCtrl
void	<a href="#">AMCLIB_BemfObsrvDQSetState_F16</a>	const SWLIBS_2Syst_F16 *const pIAB const SWLIBS_2Syst_F16 *const pUAB <a href="#"><i>tFrac16</i></a> f16Velocity <a href="#"><i>tFrac16</i></a> f16Phase AMCLIB_BEMF_OBSRV_DQ_T_F16 *const pCtrl

Type	Name	Arguments
void	<a href="#">AMCLIB_BemfObsrvDQSetState_F32</a>	const SWLIBS_2Syst_F32 *const pIAB const SWLIBS_2Syst_F32 *const pUAB <a href="#">tFrac32</a> f32Velocity <a href="#">tFrac32</a> f32Phase AMCLIB_BEMF_OBSRV_DQ_T_F32 *const pCtrl
void	<a href="#">AMCLIB_BemfObsrvDQSetState_FLT</a>	const SWLIBS_2Syst_FLT *const pIAB const SWLIBS_2Syst_FLT *const pUAB <a href="#">tFloat</a> fltVelocity <a href="#">tFloat</a> fltPhase AMCLIB_BEMF_OBSRV_DQ_T_FLT *const pCtrl
<a href="#">tFrac16</a>	<a href="#">AMCLIB_BemfObsrvDQ_F16</a>	const SWLIBS_2Syst_F16 *const pIAB const SWLIBS_2Syst_F16 *const pUAB <a href="#">tFrac16</a> f16Velocity <a href="#">tFrac16</a> f16Phase AMCLIB_BEMF_OBSRV_DQ_T_F16 *const pCtrl
<a href="#">tFrac32</a>	<a href="#">AMCLIB_BemfObsrvDQ_F32</a>	const SWLIBS_2Syst_F32 *const pIAB const SWLIBS_2Syst_F32 *const pUAB <a href="#">tFrac32</a> f32Velocity <a href="#">tFrac32</a> f32Phase AMCLIB_BEMF_OBSRV_DQ_T_F32 *const pCtrl
<a href="#">tFloat</a>	<a href="#">AMCLIB_BemfObsrvDQ_FLT</a>	const SWLIBS_2Syst_FLT *const pIAB const SWLIBS_2Syst_FLT *const pUAB <a href="#">tFloat</a> fltVelocity <a href="#">tFloat</a> fltPhase AMCLIB_BEMF_OBSRV_DQ_T_FLT *const pCtrl
void	<a href="#">AMCLIB_CurrentLoopInit_F16</a>	AMCLIB_CURRENT_LOOP_T_F16 *const pCtrl
void	<a href="#">AMCLIB_CurrentLoopInit_F32</a>	AMCLIB_CURRENT_LOOP_T_F32 *const pCtrl
void	<a href="#">AMCLIB_CurrentLoopInit_FLT</a>	AMCLIB_CURRENT_LOOP_T_FLT *const pCtrl
void	<a href="#">AMCLIB_CurrentLoopSetState_F16</a>	<a href="#">tFrac16</a> f16ControllerPIrAWDOut <a href="#">tFrac16</a> f16ControllerPIrAWQOut AMCLIB_CURRENT_LOOP_T_F16 * pCtrl
void	<a href="#">AMCLIB_CurrentLoopSetState_F32</a>	<a href="#">tFrac32</a> f32ControllerPIrAWDOut <a href="#">tFrac32</a> f32ControllerPIrAWQOut AMCLIB_CURRENT_LOOP_T_F32 * pCtrl
void	<a href="#">AMCLIB_CurrentLoopSetState_FLT</a>	<a href="#">tFloat</a> fltControllerPIrAWDOut <a href="#">tFloat</a> fltControllerPIrAWQOut AMCLIB_CURRENT_LOOP_T_FLT * pCtrl
void	<a href="#">AMCLIB_CurrentLoop_F16</a>	<a href="#">tFrac16</a> f16UDcBus SWLIBS_2Syst_F16 *const pUDQReq AMCLIB_CURRENT_LOOP_T_F16 * pCtrl
void	<a href="#">AMCLIB_CurrentLoop_F32</a>	<a href="#">tFrac32</a> f32UDcBus SWLIBS_2Syst_F32 *const pUDQReq AMCLIB_CURRENT_LOOP_T_F32 * pCtrl
void	<a href="#">AMCLIB_CurrentLoop_FLT</a>	<a href="#">tFloat</a> fltUDcBus SWLIBS_2Syst_FLT *const pUDQReq AMCLIB_CURRENT_LOOP_T_FLT * pCtrl

Type	Name	Arguments
void	<a href="#">AMCLIB_FWDebug_F16</a>	<a href="#">tFrac16</a> f16IDQReqAmp <a href="#">tFrac16</a> f16VelocityFbck SWLIBS_2Syst_F16 *const pIDQReq AMCLIB_FW_DEBUG_T_F16 * pCtrl
void	<a href="#">AMCLIB_FWDebug_F32</a>	<a href="#">tFrac32</a> f32IDQReqAmp <a href="#">tFrac32</a> f32VelocityFbck SWLIBS_2Syst_F32 *const pIDQReq AMCLIB_FW_DEBUG_T_F32 * pCtrl
void	<a href="#">AMCLIB_FWDebug_FLT</a>	<a href="#">tFloat</a> fltIDQReqAmp <a href="#">tFloat</a> fltVelocityFbck SWLIBS_2Syst_FLT *const pIDQReq AMCLIB_FW_DEBUG_T_FLT * pCtrl
void	<a href="#">AMCLIB_FWInit_F16</a>	AMCLIB_FW_T_F16 *const pCtrl
void	<a href="#">AMCLIB_FWInit_F32</a>	AMCLIB_FW_T_F32 *const pCtrl
void	<a href="#">AMCLIB_FWInit_FLT</a>	AMCLIB_FW_T_FLT *const pCtrl
void	<a href="#">AMCLIB_FWSetState_F16</a>	<a href="#">tFrac16</a> f16FilterMAFWOut <a href="#">tFrac16</a> f16ControllerPipAWFWOut AMCLIB_FW_T_F16 * pCtrl
void	<a href="#">AMCLIB_FWSetState_F32</a>	<a href="#">tFrac32</a> f32FilterMAFWOut <a href="#">tFrac32</a> f32ControllerPipAWFWOut AMCLIB_FW_T_F32 * pCtrl
void	<a href="#">AMCLIB_FWSetState_FLT</a>	<a href="#">tFloat</a> fltFilterMAFWOut <a href="#">tFloat</a> fltControllerPipAWFWOut AMCLIB_FW_T_FLT * pCtrl
void	<a href="#">AMCLIB_FWSpeedLoopDebug_F16</a>	<a href="#">tFrac16</a> f16VelocityReq <a href="#">tFrac16</a> f16VelocityFbck SWLIBS_2Syst_F16 *const pIDQReq AMCLIB_FW_SPEED_LOOP_DEBUG_T_F16 * pCtrl
void	<a href="#">AMCLIB_FWSpeedLoopDebug_F32</a>	<a href="#">tFrac32</a> f32VelocityReq <a href="#">tFrac32</a> f32VelocityFbck SWLIBS_2Syst_F32 *const pIDQReq AMCLIB_FW_SPEED_LOOP_DEBUG_T_F32 * pCtrl
void	<a href="#">AMCLIB_FWSpeedLoopDebug_FLT</a>	<a href="#">tFloat</a> fltVelocityReq <a href="#">tFloat</a> fltVelocityFbck SWLIBS_2Syst_FLT *const pIDQReq AMCLIB_FW_SPEED_LOOP_DEBUG_T_FLT * pCtrl
void	<a href="#">AMCLIB_FWSpeedLoopInit_F16</a>	AMCLIB_FW_SPEED_LOOP_T_F16 *const pCtrl
void	<a href="#">AMCLIB_FWSpeedLoopInit_F32</a>	AMCLIB_FW_SPEED_LOOP_T_F32 *const pCtrl
void	<a href="#">AMCLIB_FWSpeedLoopInit_FLT</a>	AMCLIB_FW_SPEED_LOOP_T_FLT *const pCtrl
void	<a href="#">AMCLIB_FWSpeedLoopSetState_F16</a>	<a href="#">tFrac16</a> f16FilterMAWOut <a href="#">tFrac16</a> f16FilterMAFWOut <a href="#">tFrac16</a> f16ControllerPipAWQOut <a href="#">tFrac16</a> f16ControllerPipAWFWOut <a href="#">tFrac32</a> f32RampOut AMCLIB_FW_SPEED_LOOP_T_F16 * pCtrl

Type	Name	Arguments
void	<a href="#">AMCLIB_FWSpeedLoopSetState_F32</a>	<a href="#">tFrac32</a> f32FilterMAWOut <a href="#">tFrac32</a> f32FilterMAFWOut <a href="#">tFrac32</a> f32ControllerPlpAWQOut <a href="#">tFrac32</a> f32ControllerPlpAWFWOut <a href="#">tFrac32</a> f32RampOut AMCLIB_FW_SPEED_LOOP_T_F32 * pCtrl
void	<a href="#">AMCLIB_FWSpeedLoopSetState_FLT</a>	<a href="#">tFloat</a> fltFilterMAWOut <a href="#">tFloat</a> fltFilterMAFWOut <a href="#">tFloat</a> fltControllerPlpAWQOut <a href="#">tFloat</a> fltControllerPlpAWFWOut <a href="#">tFloat</a> fltRampOut AMCLIB_FW_SPEED_LOOP_T_FLT * pCtrl
void	<a href="#">AMCLIB_FWSpeedLoop_F16</a>	<a href="#">tFrac16</a> f16VelocityReq <a href="#">tFrac16</a> f16VelocityFbck SWLIBS_2Syst_F16 *const pIDQReq AMCLIB_FW_SPEED_LOOP_T_F16 * pCtrl
void	<a href="#">AMCLIB_FWSpeedLoop_F32</a>	<a href="#">tFrac32</a> f32VelocityReq <a href="#">tFrac32</a> f32VelocityFbck SWLIBS_2Syst_F32 *const pIDQReq AMCLIB_FW_SPEED_LOOP_T_F32 * pCtrl
void	<a href="#">AMCLIB_FWSpeedLoop_FLT</a>	<a href="#">tFloat</a> fltVelocityReq <a href="#">tFloat</a> fltVelocityFbck SWLIBS_2Syst_FLT *const pIDQReq AMCLIB_FW_SPEED_LOOP_T_FLT * pCtrl
void	<a href="#">AMCLIB_FW_F16</a>	<a href="#">tFrac16</a> f16IDQReqAmp <a href="#">tFrac16</a> f16VelocityFbck SWLIBS_2Syst_F16 *const pIDQReq AMCLIB_FW_T_F16 * pCtrl
void	<a href="#">AMCLIB_FW_F32</a>	<a href="#">tFrac32</a> f32IDQReqAmp <a href="#">tFrac32</a> f32VelocityFbck SWLIBS_2Syst_F32 *const pIDQReq AMCLIB_FW_T_F32 * pCtrl
void	<a href="#">AMCLIB_FW_FLT</a>	<a href="#">tFloat</a> fltIDQReqAmp <a href="#">tFloat</a> fltVelocityFbck SWLIBS_2Syst_FLT *const pIDQReq AMCLIB_FW_T_FLT * pCtrl
void	<a href="#">AMCLIB_SpeedLoopDebug_F16</a>	<a href="#">tFrac16</a> f16VelocityReq <a href="#">tFrac16</a> f16VelocityFbck SWLIBS_2Syst_F16 *const pIDQReq AMCLIB_SPEED_LOOP_DEBUG_T_F16 * pCtrl
void	<a href="#">AMCLIB_SpeedLoopDebug_F32</a>	<a href="#">tFrac32</a> f32VelocityReq <a href="#">tFrac32</a> f32VelocityFbck SWLIBS_2Syst_F32 *const pIDQReq AMCLIB_SPEED_LOOP_DEBUG_T_F32 * pCtrl

Type	Name	Arguments
void	<a href="#">AMCLIB_SpeedLoopDebug_FLT</a>	<code>tFloat</code> fltVelocityReq <code>tFloat</code> fltVelocityFbck SWLIBS_2Syst_FLT *const pIDQReq AMCLIB_SPEED_LOOP_DEBUG_T_FLT * pCtrl
void	<a href="#">AMCLIB_SpeedLoopInit_F16</a>	AMCLIB_SPEED_LOOP_T_F16 *const pCtrl
void	<a href="#">AMCLIB_SpeedLoopInit_F32</a>	AMCLIB_SPEED_LOOP_T_F32 *const pCtrl
void	<a href="#">AMCLIB_SpeedLoopInit_FLT</a>	AMCLIB_SPEED_LOOP_T_FLT *const pCtrl
void	<a href="#">AMCLIB_SpeedLoopSetState_F16</a>	<code>tFrac16</code> f16FilterMAWOut <code>tFrac16</code> f16ControllerPlpAWQOut <code>tFrac32</code> f32RampOut AMCLIB_SPEED_LOOP_T_F16 * pCtrl
void	<a href="#">AMCLIB_SpeedLoopSetState_F32</a>	<code>tFrac32</code> f32FilterMAWOut <code>tFrac32</code> f32ControllerPlpAWQOut <code>tFrac32</code> f32RampOut AMCLIB_SPEED_LOOP_T_F32 * pCtrl
void	<a href="#">AMCLIB_SpeedLoopSetState_FLT</a>	<code>tFloat</code> fltFilterMAWOut <code>tFloat</code> fltControllerPlpAWQOut <code>tFloat</code> fltRampOut AMCLIB_SPEED_LOOP_T_FLT * pCtrl
void	<a href="#">AMCLIB_SpeedLoop_F16</a>	<code>tFrac16</code> f16VelocityReq <code>tFrac16</code> f16VelocityFbck SWLIBS_2Syst_F16 *const pIDQReq AMCLIB_SPEED_LOOP_T_F16 * pCtrl
void	<a href="#">AMCLIB_SpeedLoop_F32</a>	<code>tFrac32</code> f32VelocityReq <code>tFrac32</code> f32VelocityFbck SWLIBS_2Syst_F32 *const pIDQReq AMCLIB_SPEED_LOOP_T_F32 * pCtrl
void	<a href="#">AMCLIB_SpeedLoop_FLT</a>	<code>tFloat</code> fltVelocityReq <code>tFloat</code> fltVelocityFbck SWLIBS_2Syst_FLT *const pIDQReq AMCLIB_SPEED_LOOP_T_FLT * pCtrl
void	<a href="#">AMCLIB_TrackObsrvInit_F16</a>	AMCLIB_TRACK_OBSRV_T_F16 * pCtrl
void	<a href="#">AMCLIB_TrackObsrvInit_F32</a>	AMCLIB_TRACK_OBSRV_T_F32 * pCtrl
void	<a href="#">AMCLIB_TrackObsrvInit_FLT</a>	AMCLIB_TRACK_OBSRV_T_FLT * pCtrl
void	<a href="#">AMCLIB_TrackObsrvSetState_F16</a>	<code>tFrac16</code> f16PosOut <code>tFrac16</code> f16VelocityOut AMCLIB_TRACK_OBSRV_T_F16 * pCtrl
void	<a href="#">AMCLIB_TrackObsrvSetState_F32</a>	<code>tFrac32</code> f32PosOut <code>tFrac32</code> f32VelocityOut AMCLIB_TRACK_OBSRV_T_F32 * pCtrl
void	<a href="#">AMCLIB_TrackObsrvSetState_FLT</a>	<code>tFloat</code> fltPosOut <code>tFloat</code> fltVelocityOut AMCLIB_TRACK_OBSRV_T_FLT * pCtrl

Type	Name	Arguments
void	<a href="#">AMCLIB_TrackObsrv_F16</a>	<code>tFrac16</code> f16PhaseErr <code>tFrac16</code> * pPosEst <code>tFrac16</code> * pVelocityEst AMCLIB_TRACK_OBSRV_T_F16 * pCtrl
void	<a href="#">AMCLIB_TrackObsrv_F32</a>	<code>tFrac32</code> f32PhaseErr <code>tFrac32</code> * pPosEst <code>tFrac32</code> * pVelocityEst AMCLIB_TRACK_OBSRV_T_F32 * pCtrl
void	<a href="#">AMCLIB_TrackObsrv_FLT</a>	<code>tFloat</code> fltPhaseErr <code>tFloat</code> * pPosEst <code>tFloat</code> * pVelocityEst AMCLIB_TRACK_OBSRV_T_FLT * pCtrl
void	<a href="#">AMCLIB_WindmillingInit_F16</a>	<code>tFrac16</code> f16ADCMaxError AMCLIB_WINDMILLING_T_F16 *const pCtrl
void	<a href="#">AMCLIB_WindmillingInit_F32</a>	<code>tFrac32</code> f32ADCMaxError AMCLIB_WINDMILLING_T_F32 *const pCtrl
void	<a href="#">AMCLIB_WindmillingInit_FLT</a>	<code>tFloat</code> fltADCMaxError AMCLIB_WINDMILLING_T_FLT *const pCtrl
AMCLIB_WINDMILLING_RET_T	<a href="#">AMCLIB_Windmilling_F16</a>	const SWLIBS_3Syst_F16 * pUabcln <code>tFrac16</code> * pPosEst <code>tFrac16</code> * pVelocityEst AMCLIB_WINDMILLING_T_F16 *const pCtrl
AMCLIB_WINDMILLING_RET_T	<a href="#">AMCLIB_Windmilling_F32</a>	const SWLIBS_3Syst_F32 * pUabcln <code>tFrac32</code> * pPosEst <code>tFrac32</code> * pVelocityEst AMCLIB_WINDMILLING_T_F32 *const pCtrl
AMCLIB_WINDMILLING_RET_T	<a href="#">AMCLIB_Windmilling_FLT</a>	const SWLIBS_3Syst_FLT * pUabcln <code>tFloat</code> * pPosEst <code>tFloat</code> * pVelocityEst AMCLIB_WINDMILLING_T_FLT *const pCtrl
void	<a href="#">GDFLIB_FilterFIRInit_F16</a>	const GDFLIB_FILTERFIR_PARAM_T_F16 *const pParam GDFLIB_FILTERFIR_STATE_T_F16 *const pState <code>tFrac16</code> * pInBuf
void	<a href="#">GDFLIB_FilterFIRInit_F32</a>	const GDFLIB_FILTERFIR_PARAM_T_F32 *const pParam GDFLIB_FILTERFIR_STATE_T_F32 *const pState <code>tFrac32</code> * pInBuf
void	<a href="#">GDFLIB_FilterFIRInit_FLT</a>	const GDFLIB_FILTERFIR_PARAM_T_FLT *const pParam GDFLIB_FILTERFIR_STATE_T_FLT *const pState <code>tFloat</code> * pInBuf
<code>tFrac16</code>	<a href="#">GDFLIB_FilterFIR_F16</a>	<code>tFrac16</code> f16In const GDFLIB_FILTERFIR_PARAM_T_F16 *const pParam GDFLIB_FILTERFIR_STATE_T_F16 *const pState

Type	Name	Arguments
tFrac32	<a href="#">GDFLIB_FilterFIR_F32</a>	<code>tFrac32 f32In</code> <code>const GDFLIB_FILTERFIR_PARAM_T_F32 *const pParam</code> <code>GDFLIB_FILTERFIR_STATE_T_F32 *const pState</code>
tFloat	<a href="#">GDFLIB_FilterFIR_FLT</a>	<code>tFloat fltIn</code> <code>const GDFLIB_FILTERFIR_PARAM_T_FLT *const pParam</code> <code>GDFLIB_FILTERFIR_STATE_T_FLT *const pState</code>
void	<a href="#">GDFLIB_FilterIIR1Init_F16</a>	<code>GDFLIB_FILTER_IIR1_T_F16 *const pParam</code>
void	<a href="#">GDFLIB_FilterIIR1Init_F32</a>	<code>GDFLIB_FILTER_IIR1_T_F32 *const pParam</code>
void	<a href="#">GDFLIB_FilterIIR1Init_FLT</a>	<code>GDFLIB_FILTER_IIR1_T_FLT *const pParam</code>
tFrac16	<a href="#">GDFLIB_FilterIIR1_F16</a>	<code>tFrac16 f16In</code> <code>GDFLIB_FILTER_IIR1_T_F16 *const pParam</code>
tFrac32	<a href="#">GDFLIB_FilterIIR1_F32</a>	<code>tFrac32 f32In</code> <code>GDFLIB_FILTER_IIR1_T_F32 *const pParam</code>
tFloat	<a href="#">GDFLIB_FilterIIR1_FLT</a>	<code>tFloat fltIn</code> <code>GDFLIB_FILTER_IIR1_T_FLT *const pParam</code>
void	<a href="#">GDFLIB_FilterIIR2Init_F16</a>	<code>GDFLIB_FILTER_IIR2_T_F16 *const pParam</code>
void	<a href="#">GDFLIB_FilterIIR2Init_F32</a>	<code>GDFLIB_FILTER_IIR2_T_F32 *const pParam</code>
void	<a href="#">GDFLIB_FilterIIR2Init_FLT</a>	<code>GDFLIB_FILTER_IIR2_T_FLT *const pParam</code>
tFrac16	<a href="#">GDFLIB_FilterIIR2_F16</a>	<code>tFrac16 f16In</code> <code>GDFLIB_FILTER_IIR2_T_F16 *const pParam</code>
tFrac32	<a href="#">GDFLIB_FilterIIR2_F32</a>	<code>tFrac32 f32In</code> <code>GDFLIB_FILTER_IIR2_T_F32 *const pParam</code>
tFloat	<a href="#">GDFLIB_FilterIIR2_FLT</a>	<code>tFloat fltIn</code> <code>GDFLIB_FILTER_IIR2_T_FLT *const pParam</code>
void	<a href="#">GDFLIB_FilterMAInit_F16</a>	<code>GDFLIB_FILTER_MA_T_F16 * pParam</code>
void	<a href="#">GDFLIB_FilterMAInit_F32</a>	<code>GDFLIB_FILTER_MA_T_F32 * pParam</code>
void	<a href="#">GDFLIB_FilterMAInit_FLT</a>	<code>GDFLIB_FILTER_MA_T_FLT * pParam</code>
void	<a href="#">GDFLIB_FilterMASetState_F16</a>	<code>tFrac16 f16FilterMAOut</code> <code>GDFLIB_FILTER_MA_T_F16 * pParam</code>
void	<a href="#">GDFLIB_FilterMASetState_F32</a>	<code>tFrac32 f32FilterMAOut</code> <code>GDFLIB_FILTER_MA_T_F32 * pParam</code>
void	<a href="#">GDFLIB_FilterMASetState_FLT</a>	<code>tFloat fltFilterMAOut</code> <code>GDFLIB_FILTER_MA_T_FLT * pParam</code>
tFrac16	<a href="#">GDFLIB_FilterMA_F16</a>	<code>tFrac16 f16In</code> <code>GDFLIB_FILTER_MA_T_F16 * pParam</code>
tFrac32	<a href="#">GDFLIB_FilterMA_F32</a>	<code>tFrac32 f32In</code> <code>GDFLIB_FILTER_MA_T_F32 * pParam</code>
tFloat	<a href="#">GDFLIB_FilterMA_FLT</a>	<code>tFloat fltIn</code> <code>GDFLIB_FILTER_MA_T_FLT * pParam</code>

Type	Name	Arguments
tFrac16	GFLIB_Acos_F16	tFrac16 f16In const GFLIB_ACOS_T_F16 *const pParam
tFrac32	GFLIB_Acos_F32	tFrac32 f32In const GFLIB_ACOS_T_F32 *const pParam
tFloat	GFLIB_Acos_FLT	tFloat fltIn const GFLIB_ACOS_T_FLT *const pParam
tFrac16	GFLIB_Asin_F16	tFrac16 f16In const GFLIB_ASIN_T_F16 *const pParam
tFrac32	GFLIB_Asin_F32	tFrac32 f32In const GFLIB_ASIN_T_F32 *const pParam
tFloat	GFLIB_Asin_FLT	tFloat fltIn const GFLIB_ASIN_T_FLT *const pParam
tFrac16	GFLIB_AtanYXShifted_F16	tFrac16 f16InY tFrac16 f16InX const GFLIB_ATANYXSHIFTED_T_F16 * pParam
tFrac32	GFLIB_AtanYXShifted_F32	tFrac32 f32InY tFrac32 f32InX const GFLIB_ATANYXSHIFTED_T_F32 * pParam
tFloat	GFLIB_AtanYXShifted_FLT	tFloat fltInY tFloat fltInX const GFLIB_ATANYXSHIFTED_T_FLT * pParam
tFrac16	GFLIB_AtanYX_F16	tFrac16 f16InY tFrac16 f16InX
tFrac32	GFLIB_AtanYX_F32	tFrac32 f32InY tFrac32 f32InX
tFloat	GFLIB_AtanYX_FLT	tFloat fltInY tFloat fltInX
tFrac16	GFLIB_Atan_F16	tFrac16 f16In const GFLIB_ATAN_T_F16 *const pParam
tFrac32	GFLIB_Atan_F32	tFrac32 f32In const GFLIB_ATAN_T_F32 *const pParam
tFloat	GFLIB_Atan_FLT	tFloat fltIn const GFLIB_ATAN_T_FLT *const pParam
void	GFLIB_ControllerPIDpAWInit_F16	GFLIB_CONTROLLER_PID_P_AW_T_F16 *const pParam
void	GFLIB_ControllerPIDpAWInit_F32	GFLIB_CONTROLLER_PID_P_AW_T_F32 *const pParam
void	GFLIB_ControllerPIDpAWInit_FLT	GFLIB_CONTROLLER_PID_P_AW_T_FLT *const pParam
void	GFLIB_ControllerPIDpAWSetState_F16	tFrac16 f16ControllerPIDpAWOut GFLIB_CONTROLLER_PID_P_AW_T_F16 *const pParam

Type	Name	Arguments
void	<a href="#">GFLIB_ControllerPIDpAWSetState_F32</a>	<code>tFrac32</code> f32ControllerPIDpAWOut <code>GFLIB_CONTROLLER_PID_P_AW_T_F32 *const pParam</code>
void	<a href="#">GFLIB_ControllerPIDpAWSetState_FLT</a>	<code>tFloat</code> fltControllerPIDpAWOut <code>GFLIB_CONTROLLER_PID_P_AW_T_FLT *const pParam</code>
<code>tFrac16</code>	<a href="#">GFLIB_ControllerPIDpAW_F16</a>	<code>tFrac16</code> f16InErr <code>GFLIB_CONTROLLER_PID_P_AW_T_F16 *const pParam</code>
<code>tFrac32</code>	<a href="#">GFLIB_ControllerPIDpAW_F32</a>	<code>tFrac32</code> f32InErr <code>GFLIB_CONTROLLER_PID_P_AW_T_F32 *const pParam</code>
<code>tFloat</code>	<a href="#">GFLIB_ControllerPIDpAW_FLT</a>	<code>tFloat</code> fltInErr <code>GFLIB_CONTROLLER_PID_P_AW_T_FLT *const pParam</code>
void	<a href="#">GFLIB_ControllerPipAWInit_F16</a>	<code>GFLIB_CONTROLLER_PIAW_P_T_F16 *const pParam</code>
void	<a href="#">GFLIB_ControllerPipAWInit_F32</a>	<code>GFLIB_CONTROLLER_PIAW_P_T_F32 *const pParam</code>
void	<a href="#">GFLIB_ControllerPipAWInit_FLT</a>	<code>GFLIB_CONTROLLER_PIAW_P_T_FLT *const pParam</code>
void	<a href="#">GFLIB_ControllerPipAWSetState_F16</a>	<code>tFrac16</code> f16ControllerPipAWOut <code>GFLIB_CONTROLLER_PIAW_P_T_F16 *const pParam</code>
void	<a href="#">GFLIB_ControllerPipAWSetState_F32</a>	<code>tFrac32</code> f32ControllerPipAWOut <code>GFLIB_CONTROLLER_PIAW_P_T_F32 *const pParam</code>
void	<a href="#">GFLIB_ControllerPipAWSetState_FLT</a>	<code>tFloat</code> fltControllerPipAWOut <code>GFLIB_CONTROLLER_PIAW_P_T_FLT *const pParam</code>
<code>tFrac16</code>	<a href="#">GFLIB_ControllerPipAW_F16</a>	<code>tFrac16</code> f16InErr <code>GFLIB_CONTROLLER_PIAW_P_T_F16 *const pParam</code>
<code>tFrac32</code>	<a href="#">GFLIB_ControllerPipAW_F32</a>	<code>tFrac32</code> f32InErr <code>GFLIB_CONTROLLER_PIAW_P_T_F32 *const pParam</code>
<code>tFloat</code>	<a href="#">GFLIB_ControllerPipAW_FLT</a>	<code>tFloat</code> fltInErr <code>GFLIB_CONTROLLER_PIAW_P_T_FLT *const pParam</code>
void	<a href="#">GFLIB_ControllerPipInit_F16</a>	<code>GFLIB_CONTROLLER_PI_P_T_F16 *const pParam</code>
void	<a href="#">GFLIB_ControllerPipInit_F32</a>	<code>GFLIB_CONTROLLER_PI_P_T_F32 *const pParam</code>
void	<a href="#">GFLIB_ControllerPipInit_FLT</a>	<code>GFLIB_CONTROLLER_PI_P_T_FLT *const pParam</code>
void	<a href="#">GFLIB_ControllerPipSetState_F16</a>	<code>tFrac16</code> f16ControllerPipOut <code>GFLIB_CONTROLLER_PI_P_T_F16 *const pParam</code>
void	<a href="#">GFLIB_ControllerPipSetState_F32</a>	<code>tFrac32</code> f32ControllerPipOut <code>GFLIB_CONTROLLER_PI_P_T_F32 *const pParam</code>

Type	Name	Arguments
void	<a href="#">GFLIB_ControllerPlpSetState_FLT</a>	<a href="#">tFloat</a> fltControllerPlpOut GFLIB_CONTROLLER_PI_P_T_FLT *const pParam
tFrac16	<a href="#">GFLIB_ControllerPlp_F16</a>	<a href="#">tFrac16</a> f16InErr GFLIB_CONTROLLER_PI_P_T_F16 *const pParam
tFrac32	<a href="#">GFLIB_ControllerPlp_F32</a>	<a href="#">tFrac32</a> f32InErr GFLIB_CONTROLLER_PI_P_T_F32 *const pParam
tFloat	<a href="#">GFLIB_ControllerPlp_FLT</a>	<a href="#">tFloat</a> fltInErr GFLIB_CONTROLLER_PI_P_T_FLT *const pParam
void	<a href="#">GFLIB_ControllerPlrAWInit_F16</a>	GFLIB_CONTROLLER_PIAW_R_T_F16 *const pParam
void	<a href="#">GFLIB_ControllerPlrAWInit_F32</a>	GFLIB_CONTROLLER_PIAW_R_T_F32 *const pParam
void	<a href="#">GFLIB_ControllerPlrAWInit_FLT</a>	GFLIB_CONTROLLER_PIAW_R_T_FLT *const pParam
void	<a href="#">GFLIB_ControllerPlrAWSetState_F16</a>	<a href="#">tFrac16</a> f16ControllerPlrAWOut GFLIB_CONTROLLER_PIAW_R_T_F16 *const pParam
void	<a href="#">GFLIB_ControllerPlrAWSetState_F32</a>	<a href="#">tFrac32</a> f32ControllerPlrAWOut GFLIB_CONTROLLER_PIAW_R_T_F32 *const pParam
void	<a href="#">GFLIB_ControllerPlrAWSetState_FLT</a>	<a href="#">tFloat</a> fltControllerPlrAWOut GFLIB_CONTROLLER_PIAW_R_T_FLT *const pParam
tFrac16	<a href="#">GFLIB_ControllerPlrAW_F16</a>	<a href="#">tFrac16</a> f16InErr GFLIB_CONTROLLER_PIAW_R_T_F16 *const pParam
tFrac32	<a href="#">GFLIB_ControllerPlrAW_F32</a>	<a href="#">tFrac32</a> f32InErr GFLIB_CONTROLLER_PIAW_R_T_F32 *const pParam
tFloat	<a href="#">GFLIB_ControllerPlrAW_FLT</a>	<a href="#">tFloat</a> fltInErr GFLIB_CONTROLLER_PIAW_R_T_FLT *const pParam
void	<a href="#">GFLIB_ControllerPlrInit_F16</a>	GFLIB_CONTROLLER_PI_R_T_F16 *const pParam
void	<a href="#">GFLIB_ControllerPlrInit_F32</a>	GFLIB_CONTROLLER_PI_R_T_F32 *const pParam
void	<a href="#">GFLIB_ControllerPlrInit_FLT</a>	GFLIB_CONTROLLER_PI_R_T_FLT *const pParam
void	<a href="#">GFLIB_ControllerPlrSetState_F16</a>	<a href="#">tFrac16</a> f16ControllerPlrOut GFLIB_CONTROLLER_PI_R_T_F16 *const pParam
void	<a href="#">GFLIB_ControllerPlrSetState_F32</a>	<a href="#">tFrac32</a> f32ControllerPlrOut GFLIB_CONTROLLER_PI_R_T_F32 *const pParam
void	<a href="#">GFLIB_ControllerPlrSetState_FLT</a>	<a href="#">tFloat</a> fltControllerPlrOut GFLIB_CONTROLLER_PI_R_T_FLT *const pParam
tFrac16	<a href="#">GFLIB_ControllerPlr_F16</a>	<a href="#">tFrac16</a> f16InErr GFLIB_CONTROLLER_PI_R_T_F16 *const pParam

Type	Name	Arguments
tFrac32	<a href="#">GFLIB_ControllerPIr_F32</a>	<a href="#">tFrac32</a> f32InErr GFLIB_CONTROLLER_PI_R_T_F32 *const pParam
tFloat	<a href="#">GFLIB_ControllerPIr_FLT</a>	<a href="#">tFloat</a> fltInErr GFLIB_CONTROLLER_PI_R_T_FLT *const pParam
tFrac16	<a href="#">GFLIB_Cos_F16</a>	<a href="#">tFrac16</a> f16In const GFLIB_COS_T_F16 *const pParam
tFrac32	<a href="#">GFLIB_Cos_F32</a>	<a href="#">tFrac32</a> f32In const GFLIB_COS_T_F32 *const pParam
tFloat	<a href="#">GFLIB_Cos_FLT</a>	<a href="#">tFloat</a> fltIn const GFLIB_COS_T_FLT *const pParam
tFrac16	<a href="#">GFLIB_Hyst_F16</a>	<a href="#">tFrac16</a> f16In GFLIB_HYST_T_F16 *const pParam
tFrac32	<a href="#">GFLIB_Hyst_F32</a>	<a href="#">tFrac32</a> f32In GFLIB_HYST_T_F32 *const pParam
tFloat	<a href="#">GFLIB_Hyst_FLT</a>	<a href="#">tFloat</a> fltIn GFLIB_HYST_T_FLT *const pParam
void	<a href="#">GFLIB_IntegratorTRSetState_F16</a>	<a href="#">tFrac16</a> f16IntegratorTROut GFLIB_INTEGRATOR_TR_T_F16 *const pParam
void	<a href="#">GFLIB_IntegratorTRSetState_F32</a>	<a href="#">tFrac32</a> f32IntegratorTROut GFLIB_INTEGRATOR_TR_T_F32 *const pParam
void	<a href="#">GFLIB_IntegratorTRSetState_FLT</a>	<a href="#">tFloat</a> fltIntegratorTROut GFLIB_INTEGRATOR_TR_T_FLT *const pParam
tFrac16	<a href="#">GFLIB_IntegratorTR_F16</a>	<a href="#">tFrac16</a> f16In GFLIB_INTEGRATOR_TR_T_F16 *const pParam
tFrac32	<a href="#">GFLIB_IntegratorTR_F32</a>	<a href="#">tFrac32</a> f32In GFLIB_INTEGRATOR_TR_T_F32 *const pParam
tFloat	<a href="#">GFLIB_IntegratorTR_FLT</a>	<a href="#">tFloat</a> fltIn GFLIB_INTEGRATOR_TR_T_FLT *const pParam
tFrac16	<a href="#">GFLIB_Limit_F16</a>	<a href="#">tFrac16</a> f16In const GFLIB_LIMIT_T_F16 *const pParam
tFrac32	<a href="#">GFLIB_Limit_F32</a>	<a href="#">tFrac32</a> f32In const GFLIB_LIMIT_T_F32 *const pParam
tFloat	<a href="#">GFLIB_Limit_FLT</a>	<a href="#">tFloat</a> fltIn const GFLIB_LIMIT_T_FLT *const pParam
tFloat	<a href="#">GFLIB_Log10_FLT</a>	<a href="#">tFloat</a> fltIn const GFLIB_LOG10_T_FLT *const pParam
tFrac16	<a href="#">GFLIB_LowerLimit_F16</a>	<a href="#">tFrac16</a> f16In const GFLIB_LOWERLIMIT_T_F16 *const pParam
tFrac32	<a href="#">GFLIB_LowerLimit_F32</a>	<a href="#">tFrac32</a> f32In const GFLIB_LOWERLIMIT_T_F32 *const pParam
tFloat	<a href="#">GFLIB_LowerLimit_FLT</a>	<a href="#">tFloat</a> fltIn const GFLIB_LOWERLIMIT_T_FLT *const pParam

Type	Name	Arguments
tFrac16	<a href="#">GFLIB_Lut1D_F16</a>	<code>tFrac16 f16In</code> <code>const GFLIB_LUT1D_T_F16 *const pParam</code>
tFrac32	<a href="#">GFLIB_Lut1D_F32</a>	<code>tFrac32 f32In</code> <code>const GFLIB_LUT1D_T_F32 *const pParam</code>
tFloat	<a href="#">GFLIB_Lut1D_FLT</a>	<code>tFloat fltIn</code> <code>const GFLIB_LUT1D_T_FLT *const pParam</code>
tFrac16	<a href="#">GFLIB_Lut2D_F16</a>	<code>tFrac16 f16In1</code> <code>tFrac16 f16In2</code> <code>const GFLIB_LUT2D_T_F16 *const pParam</code>
tFrac32	<a href="#">GFLIB_Lut2D_F32</a>	<code>tFrac32 f32In1</code> <code>tFrac32 f32In2</code> <code>const GFLIB_LUT2D_T_F32 *const pParam</code>
tFloat	<a href="#">GFLIB_Lut2D_FLT</a>	<code>tFloat fltIn1</code> <code>tFloat fltIn2</code> <code>const GFLIB_LUT2D_T_FLT *const pParam</code>
tFrac16	<a href="#">GFLIB_Ramp_F16</a>	<code>tFrac16 f16In</code> <code>GFLIB_RAMP_T_F16 *const pParam</code>
tFrac32	<a href="#">GFLIB_Ramp_F32</a>	<code>tFrac32 f32In</code> <code>GFLIB_RAMP_T_F32 *const pParam</code>
tFloat	<a href="#">GFLIB_Ramp_FLT</a>	<code>tFloat fltIn</code> <code>GFLIB_RAMP_T_FLT *const pParam</code>
tFrac16	<a href="#">GFLIB_Sign_F16</a>	<code>tFrac16 f16In</code>
tFrac32	<a href="#">GFLIB_Sign_F32</a>	<code>tFrac32 f32In</code>
tFloat	<a href="#">GFLIB_Sign_FLT</a>	<code>tFloat fltIn</code>
void	<a href="#">GFLIB_SinCos_F16</a>	<code>tFrac16 f16In</code> <code>SWLIBS_2Syst_F16 * pOut</code> <code>const GFLIB_SINCOS_T_F16 *const pParam</code>
void	<a href="#">GFLIB_SinCos_F32</a>	<code>tFrac32 f32In</code> <code>SWLIBS_2Syst_F32 * pOut</code> <code>const GFLIB_SINCOS_T_F32 *const pParam</code>
void	<a href="#">GFLIB_SinCos_FLT</a>	<code>tFloat fltIn</code> <code>SWLIBS_2Syst_FLT * pOut</code> <code>const GFLIB_SINCOS_T_FLT *const pParam</code>
tFrac16	<a href="#">GFLIB_Sin_F16</a>	<code>tFrac16 f16In</code> <code>const GFLIB_SIN_T_F16 *const pParam</code>
tFrac32	<a href="#">GFLIB_Sin_F32</a>	<code>tFrac32 f32In</code> <code>const GFLIB_SIN_T_F32 *const pParam</code>
tFloat	<a href="#">GFLIB_Sin_FLT</a>	<code>tFloat fltIn</code> <code>const GFLIB_SIN_T_FLT *const pParam</code>
tFrac16	<a href="#">GFLIB_Sqrt_F16</a>	<code>tFrac16 f16In</code>
tFrac32	<a href="#">GFLIB_Sqrt_F32</a>	<code>tFrac32 f32In</code>
tFloat	<a href="#">GFLIB_Sqrt_FLT</a>	<code>tFloat fltIn</code>

Type	Name	Arguments
tFrac16	<a href="#">GFLIB_Tan_F16</a>	<code>tFrac16 f16In</code> <code>const GFLIB_TAN_T_F16 *const pParam</code>
tFrac32	<a href="#">GFLIB_Tan_F32</a>	<code>tFrac32 f32In</code> <code>const GFLIB_TAN_T_F32 *const pParam</code>
tFloat	<a href="#">GFLIB_Tan_FLT</a>	<code>tFloat fltIn</code> <code>const GFLIB_TAN_T_FLT *const pParam</code>
tFrac16	<a href="#">GFLIB_UpperLimit_F16</a>	<code>tFrac16 f16In</code> <code>const GFLIB_UPPERLIMIT_T_F16 *const pParam</code>
tFrac32	<a href="#">GFLIB_UpperLimit_F32</a>	<code>tFrac32 f32In</code> <code>const GFLIB_UPPERLIMIT_T_F32 *const pParam</code>
tFloat	<a href="#">GFLIB_UpperLimit_FLT</a>	<code>tFloat fltIn</code> <code>const GFLIB_UPPERLIMIT_T_FLT *const pParam</code>
void	<a href="#">GFLIB_VLog10_FLT</a>	<code>tFloat * plnOut</code> <code>tU32 u32N</code> <code>const GFLIB_VLOG10_T_FLT *const pParam</code>
INLINE tU16	<a href="#">GFLIB_VMin10_F16</a>	<code>const tFrac16 * pln</code>
INLINE tU16	<a href="#">GFLIB_VMin11_F16</a>	<code>const tFrac16 * pln</code>
INLINE tU16	<a href="#">GFLIB_VMin12_F16</a>	<code>const tFrac16 * pln</code>
INLINE tU16	<a href="#">GFLIB_VMin13_F16</a>	<code>const tFrac16 * pln</code>
INLINE tU16	<a href="#">GFLIB_VMin14_F16</a>	<code>const tFrac16 * pln</code>
INLINE tU16	<a href="#">GFLIB_VMin15_F16</a>	<code>const tFrac16 * pln</code>
INLINE tU16	<a href="#">GFLIB_VMin16_F16</a>	<code>const tFrac16 * pln</code>
INLINE tU16	<a href="#">GFLIB_VMin4_F16</a>	<code>const tFrac16 * pln</code>
INLINE tU16	<a href="#">GFLIB_VMin5_F16</a>	<code>const tFrac16 * pln</code>
INLINE tU16	<a href="#">GFLIB_VMin6_F16</a>	<code>const tFrac16 * pln</code>
INLINE tU16	<a href="#">GFLIB_VMin7_F16</a>	<code>const tFrac16 * pln</code>
INLINE tU16	<a href="#">GFLIB_VMin8_F16</a>	<code>const tFrac16 * pln</code>
INLINE tU16	<a href="#">GFLIB_VMin9_F16</a>	<code>const tFrac16 * pln</code>
tU16	<a href="#">GFLIB_VMin_F16</a>	<code>const tFrac16 * pln</code> <code>tU16 u16N</code>
tU32	<a href="#">GFLIB_VMin_F32</a>	<code>const tFrac32 * pln</code> <code>tU32 u32N</code>
tU32	<a href="#">GFLIB_VMin_FLT</a>	<code>const tFloat * pln</code> <code>tU32 u32N</code>
tBool	<a href="#">GFLIB_VectorLimit_F16</a>	<code>const SWLIBS_2Syst_F16 *const pln</code> <code>SWLIBS_2Syst_F16 *const pOut</code> <code>const GFLIB_VECTORLIMIT_T_F16 *const pParam</code>
tBool	<a href="#">GFLIB_VectorLimit_F32</a>	<code>const SWLIBS_2Syst_F32 *const pln</code> <code>SWLIBS_2Syst_F32 *const pOut</code> <code>const GFLIB_VECTORLIMIT_T_F32 *const pParam</code>

Type	Name	Arguments
tBool	<a href="#">GFLIB_VectorLimit_FLT</a>	const SWLIBS_2Syst_FLT *const pln SWLIBS_2Syst_FLT *const pOut const GFLIB_VECTORLIMIT_T_FLT *const pParam
void	<a href="#">GMCLIB_BetaProjection3Ph_F16</a>	const SWLIBS_3Syst_F16 *const pln SWLIBS_3Syst_F16 *const pOut
void	<a href="#">GMCLIB_BetaProjection3Ph_F32</a>	const SWLIBS_3Syst_F32 *const pln SWLIBS_3Syst_F32 *const pOut
void	<a href="#">GMCLIB_BetaProjection3Ph_FLT</a>	const SWLIBS_3Syst_FLT *const pln SWLIBS_3Syst_FLT *const pOut
void	<a href="#">GMCLIB_BetaProjection_F16</a>	const SWLIBS_3Syst_F16 *const pln SWLIBS_2Syst_F16 *const pOut
void	<a href="#">GMCLIB_BetaProjection_F32</a>	const SWLIBS_3Syst_F32 *const pln SWLIBS_2Syst_F32 *const pOut
void	<a href="#">GMCLIB_BetaProjection_FLT</a>	const SWLIBS_3Syst_FLT *const pln SWLIBS_2Syst_FLT *const pOut
void	<a href="#">GMCLIB_ClarkInv_F16</a>	const SWLIBS_2Syst_F16 *const pln SWLIBS_3Syst_F16 *const pOut
void	<a href="#">GMCLIB_ClarkInv_F32</a>	const SWLIBS_2Syst_F32 *const pln SWLIBS_3Syst_F32 *const pOut
void	<a href="#">GMCLIB_ClarkInv_FLT</a>	const SWLIBS_2Syst_FLT *const pln SWLIBS_3Syst_FLT *const pOut
void	<a href="#">GMCLIB_Clark_F16</a>	const SWLIBS_3Syst_F16 *const pln SWLIBS_2Syst_F16 *const pOut
void	<a href="#">GMCLIB_Clark_F32</a>	const SWLIBS_3Syst_F32 *const pln SWLIBS_2Syst_F32 *const pOut
void	<a href="#">GMCLIB_Clark_FLT</a>	const SWLIBS_3Syst_FLT *const pln SWLIBS_2Syst_FLT *const pOut
void	<a href="#">GMCLIB_DecouplingPMSM_F16</a>	SWLIBS_2Syst_F16 *const pUdqDec const SWLIBS_2Syst_F16 *const pUdq const SWLIBS_2Syst_F16 *const pldq <a href="#">tFrac16</a> f16AngularVel const GMCLIB_DECOUPLINGPMSM_T_F16 *const pParam
void	<a href="#">GMCLIB_DecouplingPMSM_F32</a>	SWLIBS_2Syst_F32 *const pUdqDec const SWLIBS_2Syst_F32 *const pUdq const SWLIBS_2Syst_F32 *const pldq <a href="#">tFrac32</a> f32AngularVel const GMCLIB_DECOUPLINGPMSM_T_F32 *const pParam
void	<a href="#">GMCLIB_DecouplingPMSM_FLT</a>	SWLIBS_2Syst_FLT *const pUdqDec const SWLIBS_2Syst_FLT *const pUdq const SWLIBS_2Syst_FLT *const pldq <a href="#">tFloat</a> fltAngularVel const GMCLIB_DECOUPLINGPMSM_T_FLT *const pParam

Type	Name	Arguments
void	<a href="#">GMCLIB_ElimDcBusRip_F16</a>	SWLIBS_2Syst_F16 *const pOut const SWLIBS_2Syst_F16 *const pln const GMCLIB_ELIMDCBUSRIP_T_F16 *const pParam
void	<a href="#">GMCLIB_ElimDcBusRip_F32</a>	SWLIBS_2Syst_F32 *const pOut const SWLIBS_2Syst_F32 *const pln const GMCLIB_ELIMDCBUSRIP_T_F32 *const pParam
void	<a href="#">GMCLIB_ElimDcBusRip_FLT</a>	SWLIBS_2Syst_FLT *const pOut const SWLIBS_2Syst_FLT *const pln const GMCLIB_ELIMDCBUSRIP_T_FLT *const pParam
void	<a href="#">GMCLIB_ParkInv_F16</a>	SWLIBS_2Syst_F16 *const pOut const SWLIBS_2Syst_F16 *const plnAngle const SWLIBS_2Syst_F16 *const pln
void	<a href="#">GMCLIB_ParkInv_F32</a>	SWLIBS_2Syst_F32 *const pOut const SWLIBS_2Syst_F32 *const plnAngle const SWLIBS_2Syst_F32 *const pln
void	<a href="#">GMCLIB_ParkInv_FLT</a>	SWLIBS_2Syst_FLT *const pOut const SWLIBS_2Syst_FLT *const plnAngle const SWLIBS_2Syst_FLT *const pln
void	<a href="#">GMCLIB_Park_F16</a>	SWLIBS_2Syst_F16 * pOut const SWLIBS_2Syst_F16 *const plnAngle const SWLIBS_2Syst_F16 *const pln
void	<a href="#">GMCLIB_Park_F32</a>	SWLIBS_2Syst_F32 * pOut const SWLIBS_2Syst_F32 *const plnAngle const SWLIBS_2Syst_F32 *const pln
void	<a href="#">GMCLIB_Park_FLT</a>	SWLIBS_2Syst_FLT * pOut const SWLIBS_2Syst_FLT *const plnAngle const SWLIBS_2Syst_FLT *const pln
tU16	<a href="#">GMCLIB_Pwmlct_F16</a>	SWLIBS_3Syst_F16 * pOut const SWLIBS_2Syst_F16 *const pln
tU32	<a href="#">GMCLIB_Pwmlct_F32</a>	SWLIBS_3Syst_F32 * pOut const SWLIBS_2Syst_F32 *const pln
tU32	<a href="#">GMCLIB_Pwmlct_FLT</a>	SWLIBS_3Syst_FLT * pOut const SWLIBS_2Syst_FLT *const pln
tU16	<a href="#">GMCLIB_SvmSci_F16</a>	SWLIBS_3Syst_F16 * pOut const SWLIBS_2Syst_F16 *const pln
tU32	<a href="#">GMCLIB_SvmSci_F32</a>	SWLIBS_3Syst_F32 * pOut const SWLIBS_2Syst_F32 *const pln
tU32	<a href="#">GMCLIB_SvmSci_FLT</a>	SWLIBS_3Syst_FLT * pOut const SWLIBS_2Syst_FLT *const pln
tU16	<a href="#">GMCLIB_SvmStd_F16</a>	SWLIBS_3Syst_F16 * pOut const SWLIBS_2Syst_F16 *const pln

Type	Name	Arguments
tU32	<a href="#">GMCLIB_SvmStd_F32</a>	SWLIBS_3Syst_F32 * pOut const SWLIBS_2Syst_F32 *const pln
tU32	<a href="#">GMCLIB_SvmStd_FLT</a>	SWLIBS_3Syst_FLT * pOut const SWLIBS_2Syst_FLT *const pln
tU16	<a href="#">GMCLIB_SvmU0n_F16</a>	SWLIBS_3Syst_F16 * pOut const SWLIBS_2Syst_F16 *const pln
tU32	<a href="#">GMCLIB_SvmU0n_F32</a>	SWLIBS_3Syst_F32 * pOut const SWLIBS_2Syst_F32 *const pln
tU32	<a href="#">GMCLIB_SvmU0n_FLT</a>	SWLIBS_3Syst_FLT * pOut const SWLIBS_2Syst_FLT *const pln
tU16	<a href="#">GMCLIB_SvmU7n_F16</a>	SWLIBS_3Syst_F16 * pOut const SWLIBS_2Syst_F16 *const pln
tU32	<a href="#">GMCLIB_SvmU7n_F32</a>	SWLIBS_3Syst_F32 * pOut const SWLIBS_2Syst_F32 *const pln
tU32	<a href="#">GMCLIB_SvmU7n_FLT</a>	SWLIBS_3Syst_FLT * pOut const SWLIBS_2Syst_FLT *const pln
void	<a href="#">GMCLIB_VRot_F16</a>	SWLIBS_2Syst_F16 *const f16OutVec const SWLIBS_2Syst_F16 *const f16InVec <a href="#">tFrac16</a> f16Angle
void	<a href="#">GMCLIB_VRot_F32</a>	SWLIBS_2Syst_F32 *const f32OutVec const SWLIBS_2Syst_F32 *const f32InVec <a href="#">tFrac32</a> f32Angle
void	<a href="#">GMCLIB_VRot_FLT</a>	SWLIBS_2Syst_FLT *const fltOutVec const SWLIBS_2Syst_FLT *const fltInVec <a href="#">tFloat</a> fltAngle
void	<a href="#">GMCLIB_VUnit_F16</a>	SWLIBS_2Syst_F16 *const f16OutVec const SWLIBS_2Syst_F16 *const f16InVec
void	<a href="#">GMCLIB_VUnit_F32</a>	SWLIBS_2Syst_F32 *const f32OutVec const SWLIBS_2Syst_F32 *const f32InVec
void	<a href="#">GMCLIB_VUnit_FLT</a>	SWLIBS_2Syst_FLT *const fltOutVec const SWLIBS_2Syst_FLT *const fltInVec
INLINE tFrac16	<a href="#">MLIB_AbsSat_F16</a>	register <a href="#">tFrac16</a> f16In
INLINE tFrac32	<a href="#">MLIB_AbsSat_F32</a>	register <a href="#">tFrac32</a> f32In
INLINE tFrac16	<a href="#">MLIB_Abs_F16</a>	register <a href="#">tFrac16</a> f16In
INLINE tFrac32	<a href="#">MLIB_Abs_F32</a>	register <a href="#">tFrac32</a> f32In
INLINE tFloat	<a href="#">MLIB_Abs_FLT</a>	register <a href="#">tFloat</a> fltIn
INLINE tFrac16	<a href="#">MLIB_AddSat_F16</a>	register <a href="#">tFrac16</a> f16In1 register <a href="#">tFrac16</a> f16In2
INLINE tFrac32	<a href="#">MLIB_AddSat_F32</a>	register <a href="#">tFrac32</a> f32In1 register <a href="#">tFrac32</a> f32In2
INLINE tFrac16	<a href="#">MLIB_Add_F16</a>	register <a href="#">tFrac16</a> f16In1 register <a href="#">tFrac16</a> f16In2

Type	Name	Arguments
INLINE tFrac32	<a href="#">MLIB_Add_F32</a>	register <a href="#">tFrac32</a> f32In1 register <a href="#">tFrac32</a> f32In2
INLINE tFloat	<a href="#">MLIB_Add_FLT</a>	register <a href="#">tFloat</a> fltIn1 register <a href="#">tFloat</a> fltIn2
INLINE tFrac16	<a href="#">MLIB_ConvertPU_F16F32</a>	register <a href="#">tFrac32</a> f32In
INLINE tFrac16	<a href="#">MLIB_ConvertPU_F16FLT</a>	register <a href="#">tFloat</a> fltIn
INLINE tFrac32	<a href="#">MLIB_ConvertPU_F32F16</a>	register <a href="#">tFrac16</a> f16In
INLINE tFrac32	<a href="#">MLIB_ConvertPU_F32FLT</a>	register <a href="#">tFloat</a> fltIn
INLINE tFloat	<a href="#">MLIB_ConvertPU_FLTF16</a>	register <a href="#">tFrac16</a> f16In
INLINE tFloat	<a href="#">MLIB_ConvertPU_FLTF32</a>	register <a href="#">tFrac32</a> f32In
INLINE tFrac16	<a href="#">MLIB_Convert_F16F32</a>	register <a href="#">tFrac32</a> f32In1 register <a href="#">tFrac32</a> f32In2
INLINE tFrac16	<a href="#">MLIB_Convert_F16FLT</a>	register <a href="#">tFloat</a> fltIn1 register <a href="#">tFloat</a> fltIn2
INLINE tFrac32	<a href="#">MLIB_Convert_F32F16</a>	register <a href="#">tFrac16</a> f16In1 register <a href="#">tFrac16</a> f16In2
INLINE tFrac32	<a href="#">MLIB_Convert_F32FLT</a>	register <a href="#">tFloat</a> fltIn1 register <a href="#">tFloat</a> fltIn2
INLINE tFloat	<a href="#">MLIB_Convert_FLTF16</a>	register <a href="#">tFrac16</a> f16In1 register <a href="#">tFrac16</a> f16In2
INLINE tFloat	<a href="#">MLIB_Convert_FLTF32</a>	register <a href="#">tFrac32</a> f32In1 register <a href="#">tFrac32</a> f32In2
INLINE tFrac16	<a href="#">MLIB_DivSat_F16</a>	register <a href="#">tFrac16</a> f16In1 register <a href="#">tFrac16</a> f16In2
INLINE tFrac32	<a href="#">MLIB_DivSat_F32</a>	register <a href="#">tFrac32</a> f32In1 register <a href="#">tFrac32</a> f32In2
INLINE tFrac16	<a href="#">MLIB_Div_F16</a>	register <a href="#">tFrac16</a> f16In1 register <a href="#">tFrac16</a> f16In2
INLINE tFrac32	<a href="#">MLIB_Div_F32</a>	register <a href="#">tFrac32</a> f32In1 register <a href="#">tFrac32</a> f32In2
INLINE tFloat	<a href="#">MLIB_Div_FLT</a>	register <a href="#">tFloat</a> fltIn1 register <a href="#">tFloat</a> fltIn2
INLINE tFrac16	<a href="#">MLIB_MacSat_F16</a>	register <a href="#">tFrac16</a> f16In1 register <a href="#">tFrac16</a> f16In2 register <a href="#">tFrac16</a> f16In3
INLINE tFrac32	<a href="#">MLIB_MacSat_F32</a>	register <a href="#">tFrac32</a> f32In1 register <a href="#">tFrac32</a> f32In2 register <a href="#">tFrac32</a> f32In3
INLINE tFrac32	<a href="#">MLIB_MacSat_F32F16F16</a>	register <a href="#">tFrac32</a> f32In1 register <a href="#">tFrac16</a> f16In2 register <a href="#">tFrac16</a> f16In3

Type	Name	Arguments
INLINE tFrac16	<a href="#">MLIB_Mac_F16</a>	register <a href="#">tFrac16</a> f16In1 register <a href="#">tFrac16</a> f16In2 register <a href="#">tFrac16</a> f16In3
INLINE tFrac32	<a href="#">MLIB_Mac_F32</a>	register <a href="#">tFrac32</a> f32In1 register <a href="#">tFrac32</a> f32In2 register <a href="#">tFrac32</a> f32In3
INLINE tFrac32	<a href="#">MLIB_Mac_F32F16F16</a>	register <a href="#">tFrac32</a> f32In1 register <a href="#">tFrac16</a> f16In2 register <a href="#">tFrac16</a> f16In3
INLINE tFloat	<a href="#">MLIB_MacFLT</a>	register <a href="#">tFloat</a> fltIn1 register <a href="#">tFloat</a> fltIn2 register <a href="#">tFloat</a> fltIn3
INLINE tFrac16	<a href="#">MLIB_Mnac_F16</a>	register <a href="#">tFrac16</a> f16In1 register <a href="#">tFrac16</a> f16In2 register <a href="#">tFrac16</a> f16In3
INLINE tFrac32	<a href="#">MLIB_Mnac_F32</a>	register <a href="#">tFrac32</a> f32In1 register <a href="#">tFrac32</a> f32In2 register <a href="#">tFrac32</a> f32In3
INLINE tFrac32	<a href="#">MLIB_Mnac_F32F16F16</a>	register <a href="#">tFrac32</a> f32In1 register <a href="#">tFrac16</a> f16In2 register <a href="#">tFrac16</a> f16In3
INLINE tFloat	<a href="#">MLIB_MnacFLT</a>	register <a href="#">tFloat</a> fltIn1 register <a href="#">tFloat</a> fltIn2 register <a href="#">tFloat</a> fltIn3
INLINE tFrac16	<a href="#">MLIB_Msu_F16</a>	register <a href="#">tFrac16</a> f16In1 register <a href="#">tFrac16</a> f16In2 register <a href="#">tFrac16</a> f16In3
INLINE tFrac32	<a href="#">MLIB_Msu_F32</a>	register <a href="#">tFrac32</a> f32In1 register <a href="#">tFrac32</a> f32In2 register <a href="#">tFrac32</a> f32In3
INLINE tFrac32	<a href="#">MLIB_Msu_F32F16F16</a>	register <a href="#">tFrac32</a> f32In1 register <a href="#">tFrac16</a> f16In2 register <a href="#">tFrac16</a> f16In3
INLINE tFloat	<a href="#">MLIB_MsuFLT</a>	register <a href="#">tFloat</a> fltIn1 register <a href="#">tFloat</a> fltIn2 register <a href="#">tFloat</a> fltIn3
INLINE tFrac16	<a href="#">MLIB_MulSat_F16</a>	register <a href="#">tFrac16</a> f16In1 register <a href="#">tFrac16</a> f16In2
INLINE tFrac32	<a href="#">MLIB_MulSat_F32</a>	register <a href="#">tFrac32</a> f32In1 register <a href="#">tFrac32</a> f32In2
INLINE tFrac32	<a href="#">MLIB_MulSat_F32F16F16</a>	register <a href="#">tFrac16</a> f16In1 register <a href="#">tFrac16</a> f16In2
INLINE tFrac16	<a href="#">MLIB_Mul_F16</a>	register <a href="#">tFrac16</a> f16In1 register <a href="#">tFrac16</a> f16In2

Type	Name	Arguments
INLINE tFrac32	<a href="#">MLIB_Mul_F32</a>	register <a href="#">tFrac32</a> f32In1 register <a href="#">tFrac32</a> f32In2
INLINE tFrac32	<a href="#">MLIB_Mul_F32F16F16</a>	register <a href="#">tFrac16</a> f16In1 register <a href="#">tFrac16</a> f16In2
INLINE tFloat	<a href="#">MLIB_Mul_FLT</a>	register <a href="#">tFloat</a> fltIn1 register <a href="#">tFloat</a> fltIn2
INLINE tFrac16	<a href="#">MLIB_NegSat_F16</a>	register <a href="#">tFrac16</a> f16In
INLINE tFrac32	<a href="#">MLIB_NegSat_F32</a>	register <a href="#">tFrac32</a> f32In
INLINE tFrac16	<a href="#">MLIB_Neg_F16</a>	register <a href="#">tFrac16</a> f16In
INLINE tFrac32	<a href="#">MLIB_Neg_F32</a>	register <a href="#">tFrac32</a> f32In
INLINE tFloat	<a href="#">MLIB_Neg_FLT</a>	register <a href="#">tFloat</a> fltIn
INLINE tU16	<a href="#">MLIB_Norm_F16</a>	register <a href="#">tFrac16</a> f16In
INLINE tU16	<a href="#">MLIB_Norm_F32</a>	register <a href="#">tFrac32</a> f32In
INLINE tFrac16	<a href="#">MLIB_RndSat_F16F32</a>	register <a href="#">tFrac32</a> f32In
INLINE tFrac16	<a href="#">MLIB_Round_F16</a>	register <a href="#">tFrac16</a> f16In1 register <a href="#">tU16</a> u16In2
INLINE tFrac32	<a href="#">MLIB_Round_F32</a>	register <a href="#">tFrac32</a> f32In1 register <a href="#">tU16</a> u16In2
INLINE tFrac16	<a href="#">MLIB_ShBiSat_F16</a>	register <a href="#">tFrac16</a> f16In1 register <a href="#">tS16</a> s16In2
INLINE tFrac32	<a href="#">MLIB_ShBiSat_F32</a>	register <a href="#">tFrac32</a> f32In1 register <a href="#">tS16</a> s16In2
INLINE tFrac16	<a href="#">MLIB_ShBi_F16</a>	register <a href="#">tFrac16</a> f16In1 register <a href="#">tS16</a> s16In2
INLINE tFrac32	<a href="#">MLIB_ShBi_F32</a>	register <a href="#">tFrac32</a> f32In1 register <a href="#">tS16</a> s16In2
INLINE tFrac16	<a href="#">MLIB_ShLSat_F16</a>	register <a href="#">tFrac16</a> f16In1 register <a href="#">tU16</a> u16In2
INLINE tFrac32	<a href="#">MLIB_ShLSat_F32</a>	register <a href="#">tFrac32</a> f32In1 register <a href="#">tU16</a> u16In2
INLINE tFrac16	<a href="#">MLIB_ShL_F16</a>	register <a href="#">tFrac16</a> f16In1 register <a href="#">tU16</a> u16In2
INLINE tFrac32	<a href="#">MLIB_ShL_F32</a>	register <a href="#">tFrac32</a> f32In1 register <a href="#">tU16</a> u16In2
INLINE tFrac16	<a href="#">MLIB_ShR_F16</a>	register <a href="#">tFrac16</a> f16In1 register <a href="#">tU16</a> u16In2
INLINE tFrac32	<a href="#">MLIB_ShR_F32</a>	register <a href="#">tFrac32</a> f32In1 register <a href="#">tU16</a> u16In2
INLINE tFrac16	<a href="#">MLIB_SubSat_F16</a>	register <a href="#">tFrac16</a> f16In1 register <a href="#">tFrac16</a> f16In2
INLINE tFrac32	<a href="#">MLIB_SubSat_F32</a>	register <a href="#">tFrac32</a> f32In1 register <a href="#">tFrac32</a> f32In2

Type	Name	Arguments
INLINE tFrac16	<a href="#">MLIB_Sub_F16</a>	register <a href="#">tFrac16</a> f16In1 register <a href="#">tFrac16</a> f16In2
INLINE tFrac32	<a href="#">MLIB_Sub_F32</a>	register <a href="#">tFrac32</a> f32In1 register <a href="#">tFrac32</a> f32In2
INLINE tFloat	<a href="#">MLIB_Sub_FLT</a>	register <a href="#">tFloat</a> fltIn1 register <a href="#">tFloat</a> fltIn2
INLINE void	<a href="#">MLIB_VAdd_F16</a>	SWLIBS_2Syst_F16 *const f16Out const SWLIBS_2Syst_F16 *const f16In1 const SWLIBS_2Syst_F16 *const f16In2
INLINE void	<a href="#">MLIB_VAdd_F32</a>	SWLIBS_2Syst_F32 *const f32Out const SWLIBS_2Syst_F32 *const f32In1 const SWLIBS_2Syst_F32 *const f32In2
INLINE void	<a href="#">MLIB_VAdd_FLT</a>	SWLIBS_2Syst_FLT *const fltOut const SWLIBS_2Syst_FLT *const fltIn1 const SWLIBS_2Syst_FLT *const fltIn2
INLINE tFrac16	<a href="#">MLIB_VMac_F16</a>	register <a href="#">tFrac16</a> f16In1 register <a href="#">tFrac16</a> f16In2 register <a href="#">tFrac16</a> f16In3 register <a href="#">tFrac16</a> f16In4
INLINE tFrac32	<a href="#">MLIB_VMac_F32</a>	register <a href="#">tFrac32</a> f32In1 register <a href="#">tFrac32</a> f32In2 register <a href="#">tFrac32</a> f32In3 register <a href="#">tFrac32</a> f32In4
INLINE tFrac32	<a href="#">MLIB_VMac_F32F16F16</a>	register <a href="#">tFrac16</a> f16In1 register <a href="#">tFrac16</a> f16In2 register <a href="#">tFrac16</a> f16In3 register <a href="#">tFrac16</a> f16In4
INLINE tFloat	<a href="#">MLIB_VMac_FLT</a>	register <a href="#">tFloat</a> fltIn1 register <a href="#">tFloat</a> fltIn2 register <a href="#">tFloat</a> fltIn3 register <a href="#">tFloat</a> fltIn4
INLINE void	<a href="#">MLIB_VScale_F16</a>	SWLIBS_2Syst_F16 *const f16OutVec const SWLIBS_2Syst_F16 *const f16InVec <a href="#">tFrac16</a> f16InScale
INLINE void	<a href="#">MLIB_VScale_F32</a>	SWLIBS_2Syst_F32 *const f32OutVec const SWLIBS_2Syst_F32 *const f32InVec <a href="#">tFrac32</a> f32InScale
INLINE void	<a href="#">MLIB_VScale_FLT</a>	SWLIBS_2Syst_FLT *const fltOutVec const SWLIBS_2Syst_FLT *const fltInVec <a href="#">tFloat</a> fltInScale
INLINE void	<a href="#">MLIB_VSub_F16</a>	SWLIBS_2Syst_F16 *const f16Out const SWLIBS_2Syst_F16 *const f16In1 const SWLIBS_2Syst_F16 *const f16In2

Type	Name	Arguments
INLINE void	<a href="#">MLIB_VSub_F32</a>	SWLIBS_2Syst_F32 *const f32Out const SWLIBS_2Syst_F32 *const f32In1 const SWLIBS_2Syst_F32 *const f32In2
INLINE void	<a href="#">MLIB_VSub_FLT</a>	SWLIBS_2Syst_FLT *const fltOut const SWLIBS_2Syst_FLT *const fltIn1 const SWLIBS_2Syst_FLT *const fltIn2
const SWLIBS_VERSION_T *	<a href="#">SWLIBS_GetVersion</a>	void

## 2.2 Function AMCLIB\_BemfObsrvDQ

This function calculates the algorithm of the back electromotive force observer in the rotating reference frame and returns a phase error between the real rotating reference frame and the estimated one.

### Description

The Back Electromotive Force (BEMF) observer detects the voltages induced by the permanent magnets of a Permanent Magnet Synchronous Motor (PMSM) in a quasi-synchronous reference frame. The observed BEMF allows estimation of the motor speed and position in a sensorless motor control application with Field-Oriented Control (FOC). The BEMF observer is suitable for medium to high motor speeds.

The input voltages and currents are supplied in a stationary reference frame  $\alpha/\beta$ . The BEMF observer transforms these quantities into a quasi-synchronous reference frame  $\gamma/\delta$  that follows the real synchronous rotor flux frame  $d/q$  with an error  $\theta_{err}$ , see [Figure 32](#).

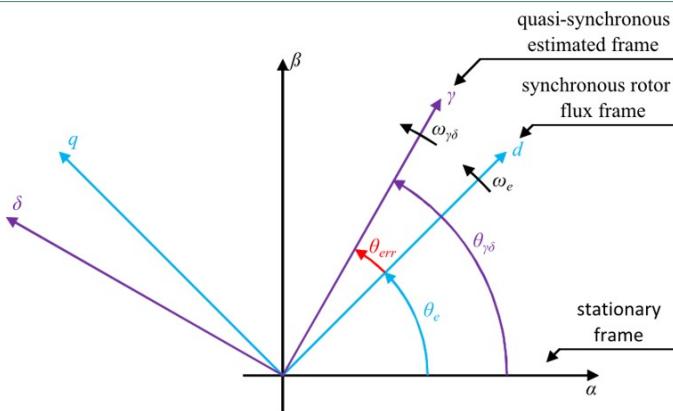


Figure 32. Rotor reference frames

The BEMF observer fits the input voltages and currents to a mathematical model of the motor. The model mirrors the behavior of the PMSM; see the following equation:

$$\begin{bmatrix} u_\gamma \\ u_\delta \end{bmatrix} = \begin{bmatrix} R_s + s \cdot L_d & -\omega_{\gamma\delta} \cdot L_q \\ \omega_{\gamma\delta} \cdot L_q & R_s + s \cdot L_d \end{bmatrix} \begin{bmatrix} i_\gamma \\ i_\delta \end{bmatrix} + E_{sal} \begin{bmatrix} -\sin(\theta_{err}) \\ \cos(\theta_{err}) \end{bmatrix}$$

Equation AMCLIB\_BemfObsrvDQ\_Eq1

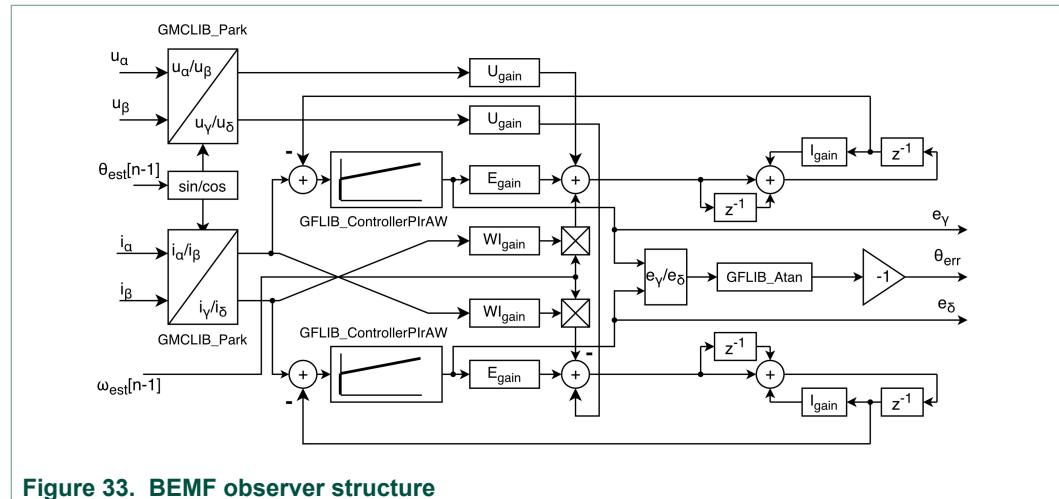
where

- $R_s$  is the resistance of one stator phase [ $\Omega$ ],
- $L_d$  and  $L_q$  are the d-axis and q-axis inductances [ $H$ ],
- $\omega_{y\delta}$  is the estimated electrical angular velocity of the rotor [rad/s],
- $u_y$  and  $u_\delta$  are the estimated stator voltages [V],
- $i_y$  and  $i_\delta$  are the estimated stator currents [A],
- $E_{sal}$  is the saliency-based BEMF magnitude [V],
- $\theta_{err}$  is the phase error between the estimated quasi-synchronous frame  $y/\delta$  and the synchronous rotor flux frame d/q [rad],
- $s$  is the Laplace-Carson differential operator.

Note that in this motor model, the voltage in the  $\delta$  coordinate is calculated from the  $L_d$  inductance instead of  $L_q$ . Because of this, the response to the measurement errors of the  $R_s$  and  $L_d$  parameters is the same in both axes. The BEMF observer is formed in both of these axes and the resulting  $\theta_{err}$  is extracted from the division  $E_y/E_\delta$ . Assuming sufficient motor speed, the result of the division is insensitive to the  $E_{sal}$ . This allows correct setup of the controllers.

As seen from [AMCLIB\\_BemfObsrvDQ\\_Eq1](#) only the BEMF terms depend on the phase error  $\theta_{err}$  between the quasi-synchronous reference frame  $y/\delta$  and the synchronous rotor flux frame d/q. The saliency-based BEMF term is not modeled in the stator current observer however it is estimated as a disturbance, produced by the observer PI controller. The observer is a closed loop current observer and acts as a BEMF state filter. The estimated BEMF values are used for calculating the phase error  $\theta_{err}$ , which is provided as an output of the BEMF observer.

The structure of the BEMF observer is depicted in [Figure 33](#).



**Figure 33. BEMF observer structure**

The BEMF observer loop consists of a model of R-L circuit which represents the motor winding and a proportional-integral controller with an output referring to the BEMF signals. Refer to the [GFLIB\\_ControllerPlrAW](#) function documentation for details on the implementation of the controller and its parameters. Discrete-time integrators are approximated using the trapezoidal rule. The motor model is characterized by the following difference equations:

$$\begin{aligned}
 i_{dSC}(n) &= (U_{gain} \cdot u_{dSC}(n) + WI_{gain} \cdot \omega_{SC}(n) \cdot i_{qSC}(n) + E_{gain} \cdot e_{dSC}(n)) \cdot 2^{s16Shift} + I_{gain} \cdot i_{dSC}(n-1) + x_d(n-1) \\
 x_d(n-1) &= (U_{gain} \cdot u_{dSC}(n-1) + WI_{gain} \cdot \omega_{SC}(n-1) \cdot i_{qSC}(n-1) + E_{gain} \cdot e_{dSC}(n-1)) \cdot 2^{s16Shift} \\
 i_{qSC}(n) &= (U_{gain} \cdot u_{qSC}(n) - WI_{gain} \cdot \omega_{SC}(n) \cdot i_{dSC}(n) + E_{gain} \cdot e_{qSC}(n)) \cdot 2^{s16Shift} + I_{gain} \cdot i_{qSC}(n-1) + x_q(n-1) \\
 x_q(n-1) &= (U_{gain} \cdot u_{qSC}(n-1) - WI_{gain} \cdot \omega_{SC}(n-1) \cdot i_{dSC}(n-1) + E_{gain} \cdot e_{qSC}(n-1)) \cdot 2^{s16Shift}
 \end{aligned}$$

Equation AMCLIB\_BemfObsrvDQ\_Eq2

where the subscript SC indicates values scaled to the fractional range [-1, 1].

Before using the BEMF observer with a particular motor, the user needs to provide a set of coefficients through the pCtrl input pointer. The BEMF observer coefficient values can be calculated from motor parameters. A method for measuring the motor parameters is described in PMSM Electrical Parameters Measurement (document [AN4680](#)).

Refer to the following resources to find out how the NXP motor control tuning and debugging tools for NXP microcontrollers can help you deploy the AMCLIB BEMF observer in your application:

- [AN4642](#) - Motor Control Application Tuning (MCAT) Tool for 3-Phase PMSM
- [FREEMASTER](#) - FreeMASTER Run-Time Debugging Tool

**Note:** The input/output pointers must contain valid addresses, otherwise a fault may occur (MemManage, BusFault, UsageFault, HardFault).

## Re-entrancy

The function is re-entrant for a different pCtrl.

### 2.2.1 Function AMCLIB\_BemfObsrvDQ\_F32

#### Declaration

```
tFrac32 AMCLIB_BemfObsrvDQ_F32 (const SWLIBS_2Syst_F32 *const pIAB, const SWLIBS_2Syst_F32 *const pUAB, tFrac32 f32Velocity, tFrac32 f32Phase, AMCLIB_BEMF_OBSRV_DQ_T_F32 *const pCtrl);
```

#### Arguments

Table 3. AMCLIB\_BemfObsrvDQ\_F32 arguments

Type	Name	Direction	Description
const SWLIBS_2Syst_F32 *const	pIAB	input	Pointer to the structure with Alpha/Beta current components.
const SWLIBS_2Syst_F32 *const	pUAB	input	Pointer to the structure with Alpha/Beta voltage components.
tFrac32	f32Velocity	input	Estimated electrical angular velocity.
tFrac32	f32Phase	input	Estimated rotor flux angle.
AMCLIB_BEMF_OBSRV_DQ_T_F32 *const	pCtrl	input, output	Pointer to the structure with BEMF observer parameters and state.

#### Return

Phase error between the real rotating reference frame and the estimated one.

**Implementation details**

Prior to calculating the BEMF observer coefficients, it is necessary to set the scaling constants. All inputs and outputs of the algorithm are limited to the fractional range [-1, 1]. The incorrect setting of the scaling constants may lead to an undesirable overflow or saturation during the computation. There are two different scaling systems involved, one for the FOC part of the motor control algorithm, and another for the BEMF observer. Scaling constants must be positive values equal to or greater than the expected maxima of the corresponding physical quantities. The following scaling constants are applied to the BEMF observer coefficients:

**Table 4. Scaling constants**

Scaling constant	Symbol	Calculation
Maximum stator phase voltage [V]	$U_{MAX}$	$U_{MAX} = U_{DC\_Bus\_Max}$
Maximum phase current [A]	$I_{MAX}$	Maximum current of the inverter or nominal current of the motor (whichever is lower).
Maximum speed [rad/s]	$\Omega_{MAX}$	Maximum application required speed, at least the motor electrical rated speed.
Maximum BEMF voltage [V]	$E_{MAX}$	In normal operation is equal to $U_{MAX}$ , in case of, e.g., field weakening, might be much higher.

Parameters of the PIrAW controllers inside the BEMF observer can be calculated using the following equations:

$$\begin{aligned}
 pParamD.f32CC1sc &= FRAC32\left(\left(K_p + \frac{K_I T_S}{2}\right) \cdot \frac{I_{MAX}}{E_{MAX}} \cdot 2^{NShift}\right) \\
 pParamD.f32CC2sc &= FRAC32\left(\left(-K_p + \frac{K_I T_S}{2}\right) \cdot \frac{I_{MAX}}{E_{MAX}} \cdot 2^{NShift}\right) \\
 pParamDu16NShift &= NShift \\
 pParamQf32CC1sc &= FRAC32\left(\left(K_p + \frac{K_I T_S}{2}\right) \cdot \frac{I_{MAX}}{E_{MAX}} \cdot 2^{NShift}\right) \\
 pParamQf32CC2sc &= FRAC32\left(\left(-K_p + \frac{K_I T_S}{2}\right) \cdot \frac{I_{MAX}}{E_{MAX}} \cdot 2^{NShift}\right) \\
 pParamQu16NShift &= NShift
 \end{aligned}$$

Equation AMCLIB\_BemfObsrvDQ\_F32\_Eq1

where  $T_S$  is the sampling period,  $K_p$  is the proportional gain, and  $K_I$  is the integral gain. The upper and lower limits of the PIrAW controller should be set based on the expected dynamics of the system.  $NShift$  is the smallest nonnegative integer value that ensures that the controller coefficients fit in the fractional range [-1, 1). The gains can be calculated as follows:

$$\begin{aligned}
 K_p &= 2 \cdot \xi \cdot \omega_0 \cdot L_d \cdot R_S \\
 K_I &= \omega_0^2 \cdot L_d
 \end{aligned}$$

Equation AMCLIB\_BemfObsrvDQ\_F32\_Eq2

where  $\xi$  is the current loop attenuation, and  $\omega_0$  is the current loop natural frequency [rad/s]. Coefficients  $\xi$  and  $\omega_0$  should correspond to the values chosen for the FOC current loop.

The winding model (R-L circuit) and cross-coupling constants can be set according to the following equations:

$$\begin{aligned}
 f32IGain &= FRAC32\left(\frac{2L_d T_s R_s}{2L_d + T_s R_s}\right) \\
 f32UGain &= FRAC32\left(\frac{T_s}{2L_d + T_s R_s} \cdot \frac{U_{MAX}}{I_{MAX}} \cdot 2^{NShiftRL}\right) \\
 f32WIGain &= FRAC32\left(\frac{T_s L_q}{2L_d + T_s R_s} \cdot \Omega_{MAX} \cdot 2^{NShiftRL}\right) \\
 f32EGain &= FRAC32\left(\frac{T_s}{2L_d + T_s R_s} \cdot \frac{E_{MAX}}{I_{MAX}} \cdot 2^{NShiftRL}\right) \\
 s16Shift &= NShiftRL
 \end{aligned}$$

Equation AMCLIB\_BemfObsrvDQ\_F32\_Eq3

*NShiftRL* is set to ensure that the gains fit in the fractional range [-1, 1).

The following m-script can be passed to the Matlab® command window to calculate the BEMF observer coefficients from the motor parameters:

```

% Motor parameters
% (to be set according to measurements)
Ld = 3.0e-4; % inductance in d-axis [H]
Lq = 3.0e-4; % inductance in q-axis [H]
Rs = 0.33; % resistance of one stator phase [Ω]

% Scaling constants
% (to be set according to known maxima)
Imax = 20; % maximum stator phase current [A]
Umax = 14.4; % maximum stator phase voltage [V]
Wmax = 2618; % maximum angular velocity [rad/s]
Emax = 14.4; % maximum BEMF [V]

% Control system parameters
% (to be set according to the chosen control system dynamics)
Ts = 1e-4; % sampling period [s]
i_Ksi = 1; % current loop attenuation
i_fo = 350; % current loop natural frequency [Hz]
i_wo = 2*pi()*i_fo; % current loop natural angular frequency [rad/s]
Kp = 2*i_Ksi*i_wo*Ld-Rs;
Ki = i_wo^2*Ld;

disp('--- AMCLIB_BemfObsrvDQ_F32 coefficients ---')
% PIrAW controller parameters
maxCoeff = max(abs([(Kp + Ki*Ts/2)*Imax/Umax, ...
    (-Kp + Ki*Ts/2)*Imax/Umax]));
NShift = max(0, ceil(log2(maxCoeff)));
if (NShift > 14)
    error('Inputted parameters cannot be used - u16NShift exceeds 14');
end
pCtrl_pParamD_f32CC1sc = (Kp + Ki*Ts/2)*Imax/Umax*2^-NShift;
pCtrl_pParamD_f32CC1sc = round(pCtrl_pParamD_f32CC1sc * 2^31);
pCtrl_pParamD_f32CC1sc(pCtrl_pParamD_f32CC1sc < -(2^31)) = -(2^31);
pCtrl_pParamD_f32CC1sc(pCtrl_pParamD_f32CC1sc > (2^31)-1) = (2^31)-1;
pCtrl_pParamD_f32CC2sc = (-Kp + Ki*Ts/2)*Imax/Umax*2^-NShift;
pCtrl_pParamD_f32CC2sc = round(pCtrl_pParamD_f32CC2sc * 2^31);
pCtrl_pParamD_f32CC2sc(pCtrl_pParamD_f32CC2sc < -(2^31)) = -(2^31);
pCtrl_pParamD_f32CC2sc(pCtrl_pParamD_f32CC2sc > (2^31)-1) = (2^31)-1;
pCtrl_pParamD_u16NShift = NShift;
pCtrl_pParamQ_f32CC1sc = pCtrl_pParamD_f32CC1sc;
pCtrl_pParamQ_f32CC2sc = pCtrl_pParamD_f32CC2sc;
pCtrl_pParamQ_u16NShift = NShift;
disp(['Ctrl.pParamD.f32CC1sc = ' num2str(pCtrl_pParamD_f32CC1sc) ';'])

```

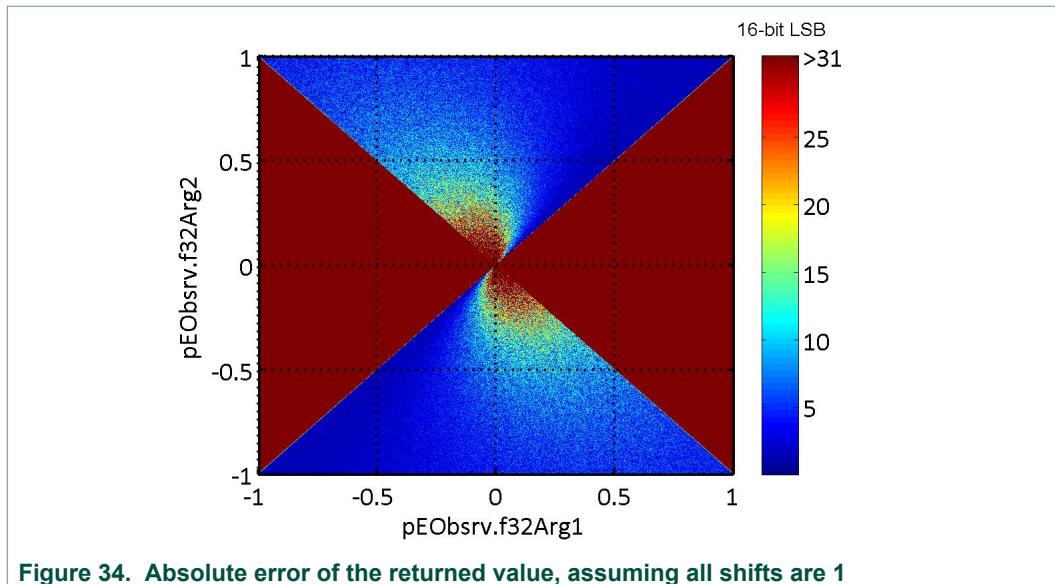
```

disp(['Ctrl.pParamD.f32CC2sc = ' num2str(pCtrl_pParamD_f32CC2sc) ';' ])
disp(['Ctrl.pParamD.u16NShift = ' num2str(NShift) ';' ])
disp(['Ctrl.pParamQ.f32CC1sc = ' num2str(pCtrl_pParamQ_f32CC1sc) ';' ])
disp(['Ctrl.pParamQ.f32CC2sc = ' num2str(pCtrl_pParamQ_f32CC2sc) ';' ])
disp(['Ctrl.pParamQ.u16NShift = ' num2str(NShift) ';' ])
disp(' Ctrl.pParamD.f32UpperLimit, Ctrl.pParamD.f32LowerLimit, ')
disp(' Ctrl.pParamQ.f32UpperLimit, and Ctrl.pParamQ.f32LowerLimit')
disp(' shall be set according to the expected dynamics')

% RL circuit parameters
maxCoeffRL = max(abs([Ts/(2*Ld+Ts*Rs)*Umax/Imax, ...
                      Ts*Lq/(2*Ld+Ts*Rs)*Wmax, ...
                      Ts/(2*Ld+Ts*Rs)*Emax/Imax]));
NShiftRL = ceil(log2(maxCoeffRL));
if (NShiftRL < -14)
    NShiftRL = -14;
end
if (NShiftRL > 14)
    error('Inputted parameters cannot be used - s16Shift exceeds 14');
end
pCtrl_f32IGain = (2*Ld-Ts*Rs)/(2*Ld+Ts*Rs);
pCtrl_f32IGain = round(pCtrl_f32IGain * 2^31);
pCtrl_f32IGain(pCtrl_f32IGain < -(2^31)) = -(2^31);
pCtrl_f32IGain(pCtrl_f32IGain > (2^31)-1) = (2^31)-1;
pCtrl_f32UGain = Ts/(2*Ld+Ts*Rs)*Umax/Imax*2^-NShiftRL;
pCtrl_f32UGain = round(pCtrl_f32UGain * 2^31);
pCtrl_f32UGain(pCtrl_f32UGain < -(2^31)) = -(2^31);
pCtrl_f32UGain(pCtrl_f32UGain > (2^31)-1) = (2^31)-1;
pCtrl_f32WIGain = Ts*Lq/(2*Ld+Ts*Rs)*Wmax*2^-NShiftRL;
pCtrl_f32WIGain = round(pCtrl_f32WIGain * 2^31);
pCtrl_f32WIGain(pCtrl_f32WIGain < -(2^31)) = -(2^31);
pCtrl_f32WIGain(pCtrl_f32WIGain > (2^31)-1) = (2^31)-1;
pCtrl_f32EGain = Ts/(2*Ld+Ts*Rs)*Emax/Imax*2^-NShiftRL;
pCtrl_f32EGain = round(pCtrl_f32EGain * 2^31);
pCtrl_f32EGain(pCtrl_f32EGain < -(2^31)) = -(2^31);
pCtrl_f32EGain(pCtrl_f32EGain > (2^31)-1) = (2^31)-1;
disp(['Ctrl.f32IGain = ' num2str(pCtrl_f32IGain) ';' ])
disp(['Ctrl.f32UGain = ' num2str(pCtrl_f32UGain) ';' ])
disp(['Ctrl.f32WIGain = ' num2str(pCtrl_f32WIGain) ';' ])
disp(['Ctrl.f32EGain = ' num2str(pCtrl_f32EGain) ';' ])
disp(['Ctrl.s16Shift = ' num2str(NShiftRL) ';' ])

```

The accuracy of results is guaranteed for the outputs pEObsrv.f32Arg1 and pEObsrv.f32Arg2 only in cases when pParamD.u16NShift, pParamQ.u16NShift, and s16Shift are not greater than 1. There is no limit of computational error specified for the returned value. The actual error depends on the values of pEObsrv.f32Arg1 and pEObsrv.f32Arg2. The following figure shows the expected values of absolute error [16-bit LSB] contained in the returned value in the cases when all shifts are equal to 1.

**Figure 34.** Absolute error of the returned value, assuming all shifts are 1

**Note:** The BEMF observer coefficients *f32IGain*, *f32UGain*, *f32WIGain*, and *f32EGain* must not contain the largest negative value, otherwise the accuracy of the results is not guaranteed.

Keep the values of *pParamD.u16NShift*, *pParamQ.u16NShift*, and *s16Shift* within the allowed limits to prevent an overflow of intermediate results.

The function performs the fastest when *s16Shift* is equal to zero.

Due to effectivity reasons, this function is implemented using inline assembly and is therefore not ANSI-C compliant.

### Code Example

```
#include "amclib.h"

SWLIBS_2Syst_F32          mcIab, mcUab;
AMCLIB_BEMF_OBSRV_DQ_T_F32 BemfObsrv;
tFrac32                     f32Velocity;
tFrac32                     f32Phase;

void main (void)
{
    // Clear BEMF observer state variables:

    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB_BemfObsrvDOInit_F32(&BemfObsrv);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB_BemfObsrvDQInit(&BemfObsrv, F32);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
}
```

```
// only if 32-bit fractional implementation is selected as default.  
AMCLIB_BemfObsrvDQInit(&BemfObsrv);  
  
// Set BEMF observer parameters:  
BemfObsrv.pParamD.f32CC1sc      = (tFrac32)1583784859;  
BemfObsrv.pParamD.f32CC2sc      = (tFrac32)-1367421132;  
BemfObsrv.pParamD.f32UpperLimit = (tFrac32)2147483647;  
BemfObsrv.pParamD.f32LowerLimit = (tFrac32)-2147483648;  
BemfObsrv.pParamD.u16NShift    = (tU16)1;  
BemfObsrv.pParamQ.f32CC1sc      = (tFrac32)1583784859;  
BemfObsrv.pParamQ.f32CC2sc      = (tFrac32)-1367421132;  
BemfObsrv.pParamQ.f32UpperLimit = (tFrac32)2147483647;  
BemfObsrv.pParamQ.f32LowerLimit = (tFrac32)-2147483648;  
BemfObsrv.pParamQ.u16NShift    = (tU16)1;  
BemfObsrv.f32IGain   = (tFrac32)1923575400;  
BemfObsrv.f32UGain   = (tFrac32)1954108343;  
BemfObsrv.f32WIGain  = (tFrac32)2131606518;  
BemfObsrv.f32EGain   = (tFrac32)1954108343;  
BemfObsrv.s16Shift   = (ts16)-3;  
  
// Obtain the initial estimate of the rotor position and velocity,  
// measure the alpha/beta voltages and currents  
// (...)  
  
// Initialize the internal states of the observer to achieve  
// seamless transition from an uncontrolled state of the motor  
// to the full feedback control:  
  
// Alternative 1: API call with postfix  
// (only one alternative shall be used).  
AMCLIB_BemfObsrvDQSetState_F32(&mciLab, &mcUab, f32Velocity,  
f32Phase, &BemfObsrv);  
  
// Alternative 2: API call with implementation parameter  
// (only one alternative shall be used).  
AMCLIB_BemfObsrvDQSetState(&mciLab, &mcUab, f32Velocity,  
f32Phase, &BemfObsrv, F32);  
  
// Alternative 3: API call with global configuration of implementation  
// (only one alternative shall be used). This alternative is available  
// only if 32-bit fractional implementation is selected as default.  
AMCLIB_BemfObsrvDQSetState(&mciLab, &mcUab, f32Velocity,  
f32Phase, &BemfObsrv);  
  
while(1);  
}  
  
// Periodical function or interrupt  
void ISR(void)  
{  
    tFrac32 f32PhaseErr;  
  
    // Read the A/D, calculate alpha-beta values, etc.  
    // (...)  
  
    // Calculate one iteration of the observer:  
  
    // Alternative 1: API call with postfix  
    // (only one alternative shall be used).  
    f32PhaseErr = AMCLIB_BemfObsrvDQ_F32(&mciLab, &mcUab, f32Velocity,
```

```

f32Phase, &BemfObsrv);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
f32PhaseErr = AMCLIB_BemfObsrvDQ(&mcIab, &mcUab, f32Velocity,
    f32Phase, &BemfObsrv, F32);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 32-bit fractional implementation is selected as default.
f32PhaseErr = AMCLIB_BemfObsrvDQ(&mcIab, &mcUab, f32Velocity,
    f32Phase, &BemfObsrv);

// Pass f32PhaseErr to the #AMCLIB_TrackObsrv_F32
// (...)

}

```

## 2.2.2 Function AMCLIB\_BemfObsrvDQ\_F16

### Declaration

```
tFrac16 AMCLIB_BemfObsrvDQ_F16(const SWLIBS_2Syst_F16 *const
    pIAB, const SWLIBS_2Syst_F16 *const pUAB, tFrac16 f16Velocity,
    tFrac16 f16Phase, AMCLIB_BEMF_OBSRV_DQ_T_F16 *const pCtrl);
```

### Arguments

**Table 5. AMCLIB\_BemfObsrvDQ\_F16 arguments**

Type	Name	Direction	Description
const SWLIBS_2Syst_F16 *const	pIAB	input	Pointer to the structure with Alpha/Beta current components.
const SWLIBS_2Syst_F16 *const	pUAB	input	Pointer to the structure with Alpha/Beta voltage components.
tFrac16	f16Velocity	input	Estimated electrical angular velocity.
tFrac16	f16Phase	input	Estimated rotor flux angle.
AMCLIB_BEMF_OBSRV_DQ_T_F16 *const	pCtrl	input, output	Pointer to the structure with BEMF observer parameters and state.

### Return

Phase error between the real rotating reference frame and the estimated one.

### Implementation details

Prior to calculating the BEMF observer coefficients, it is necessary to set the scaling constants. All inputs and outputs of the algorithm are limited to the fractional range [-1, 1]. The incorrect setting of the scaling constants may lead to an undesirable overflow or saturation during the computation. There are two different scaling systems involved, one for the FOC part of the motor control algorithm, and another for the BEMF observer. Scaling constants must be positive values equal to or greater than the expected maxima of the corresponding physical quantities. The following scaling constants are applied to the BEMF observer coefficients:

**Table 6. Scaling constants**

Scaling constant	Symbol	Calculation
------------------	--------	-------------

Maximum stator phase voltage [V]	$U_{MAX}$	$U_{MAX}=U_{DC\_Bus\_Max}$
Maximum phase current [A]	$I_{MAX}$	Maximum current of the inverter or nominal current of the motor (whichever is lower).
Maximum speed [rad/s]	$\Omega_{MAX}$	Maximum application required speed, at least the motor electrical rated speed.
Maximum BEMF voltage [V]	$E_{MAX}$	In normal operation is equal to $U_{MAX}$ , in case of, e.g., field weakening, might be much higher.

Parameters of the PIrAW controllers inside the BEMF observer can be calculated using the following equations:

$$\begin{aligned}
 pParamD.f16CC1sc &= FRAC16\left(\left(K_P + \frac{K_I T_S}{2}\right) \cdot \frac{I_{MAX}}{E_{MAX}} \cdot 2^{NShift}\right) \\
 pParamD.f16CC2sc &= FRAC16\left(\left(-K_P + \frac{K_I T_S}{2}\right) \cdot \frac{I_{MAX}}{E_{MAX}} \cdot 2^{NShift}\right) \\
 pParamDu16NShift &= NShift \\
 pParamQ.f16CC1sc &= FRAC16\left(\left(K_P + \frac{K_I T_S}{2}\right) \cdot \frac{I_{MAX}}{E_{MAX}} \cdot 2^{NShift}\right) \\
 pParamQ.f16CC2sc &= FRAC16\left(\left(-K_P + \frac{K_I T_S}{2}\right) \cdot \frac{I_{MAX}}{E_{MAX}} \cdot 2^{NShift}\right) \\
 pParamQu16NShift &= NShift
 \end{aligned}$$

Equation AMCLIB\_BemfObsrvDQ\_F16\_Eq1

where  $T_S$  is the sampling period,  $K_P$  is the proportional gain, and  $K_I$  is the integral gain. The upper and lower limits of the PIrAW controller should be set based on the expected dynamics of the system.  $NShift$  is the smallest nonnegative integer value that ensures that the controller coefficients fit in the fractional range [-1, 1). The gains can be calculated as follows:

$$\begin{aligned}
 K_P &= 2 \cdot \xi \cdot \omega_0 \cdot L_d \cdot R_S \\
 K_I &= \omega_0^2 \cdot L_d
 \end{aligned}$$

Equation AMCLIB\_BemfObsrvDQ\_F16\_Eq2

where  $\xi$  is the current loop attenuation, and  $\omega_0$  is the current loop natural frequency [rad/s]. Coefficients  $\xi$  and  $\omega_0$  should correspond to the values chosen for the FOC current loop.

The winding model (R-L circuit) and cross-coupling constants can be set according to the following equations:

$$\begin{aligned}
 f16IGain &= FRAC16\left(\frac{2L_d T_S R_S}{2L_d + T_S R_S}\right) \\
 f16UGain &= FRAC16\left(\frac{T_S}{2L_d + T_S R_S} \cdot \frac{U_{MAX}}{I_{MAX}} \cdot 2^{NShiftRL}\right) \\
 f16WIGain &= FRAC16\left(\frac{T_S L_q}{2L_d + T_S R_S} \cdot \Omega_{MAX} \cdot 2^{NShiftRL}\right) \\
 f16EGain &= FRAC16\left(\frac{T_S}{2L_d + T_S R_S} \cdot \frac{E_{MAX}}{I_{MAX}} \cdot 2^{NShiftRL}\right) \\
 s16Shift &= NShiftRL
 \end{aligned}$$

Equation AMCLIB\_BemfObsrvDQ\_F16\_Eq3

$NShiftRL$  is set to ensure that the gains fit in the fractional range [-1, 1).

The following m-script can be passed to the Matlab<sup>®</sup> command window to calculate the BEMF observer coefficients from the motor parameters:

```
% Motor parameters
% (to be set according to measurements)
Ld = 3.0e-4; % inductance in d-axis [H]
Lq = 3.0e-4; % inductance in q-axis [H]
Rs = 0.33; % resistance of one stator phase [ $\Omega$ ]

% Scaling constants
% (to be set according to known maxima)
Imax = 20; % maximum stator phase current [A]
Umax = 14.4; % maximum stator phase voltage [V]
Wmax = 2618; % maximum angular velocity [rad/s]
Emax = 14.4; % maximum BEMF [V]

% Control system parameters
% (to be set according to the chosen control system dynamics)
Ts = 1e-4; % sampling period [s]
i_Ksi = 1; % current loop attenuation
i_fo = 350; % current loop natural frequency [Hz]
i_wo = 2*pi()*i_fo; % current loop natural angular frequency [rad/s]
Kp = 2*i_Ksi*i_wo*Ld-Rs;
Ki = i_wo^2*Ld;

disp('--- AMCLIB_BemfObsrvDQ_F16 coefficients ---')
% PIrAW controller parameters
maxCoeff = max(abs([(Kp + Ki*Ts/2)*Imax/Umax, ...
                    (-Kp + Ki*Ts/2)*Imax/Umax]));
NShift = max(0, ceil(log2(maxCoeff)));
if (NShift > 14)
    error('Inputted parameters cannot be used - u16NShift exceeds 14');
end
pCtrl_pParamD_f16CC1sc = (Kp + Ki*Ts/2)*Imax/Umax*2^-NShift;
pCtrl_pParamD_f16CC1sc = round(pCtrl_pParamD_f16CC1sc * 2^15);
pCtrl_pParamD_f16CC1sc(pCtrl_pParamD_f16CC1sc < -(2^15)) = -(2^15);
pCtrl_pParamD_f16CC1sc(pCtrl_pParamD_f16CC1sc > (2^15)-1) = (2^15)-1;
pCtrl_pParamD_f16CC2sc = (-Kp + Ki*Ts/2)*Imax/Umax*2^-NShift;
pCtrl_pParamD_f16CC2sc = round(pCtrl_pParamD_f16CC2sc * 2^15);
pCtrl_pParamD_f16CC2sc(pCtrl_pParamD_f16CC2sc < -(2^15)) = -(2^15);
pCtrl_pParamD_f16CC2sc(pCtrl_pParamD_f16CC2sc > (2^15)-1) = (2^15)-1;
pCtrl_pParamD_u16NShift = NShift;
pCtrl_pParamQ_f16CC1sc = pCtrl_pParamD_f16CC1sc;
pCtrl_pParamQ_f16CC2sc = pCtrl_pParamD_f16CC2sc;
pCtrl_pParamQ_u16NShift = NShift;
disp(['Ctrl.pParamD.f16CC1sc = ' num2str(pCtrl_pParamD_f16CC1sc) ';'])
disp(['Ctrl.pParamD.f16CC2sc = ' num2str(pCtrl_pParamD_f16CC2sc) ';'])
disp(['Ctrl.pParamD.u16NShift = ' num2str(NShift) ';'])
disp(['Ctrl.pParamQ.f16CC1sc = ' num2str(pCtrl_pParamQ_f16CC1sc) ';'])
disp(['Ctrl.pParamQ.f16CC2sc = ' num2str(pCtrl_pParamQ_f16CC2sc) ';'])
disp(['Ctrl.pParamQ.u16NShift = ' num2str(NShift) ';'])
disp(' Ctrl.pParamD.f16UpperLimit, Ctrl.pParamD.f16LowerLimit, ')
disp(' Ctrl.pParamQ.f16UpperLimit, and Ctrl.pParamQ.f16LowerLimit')
disp(' shall be set according to the expected dynamics')

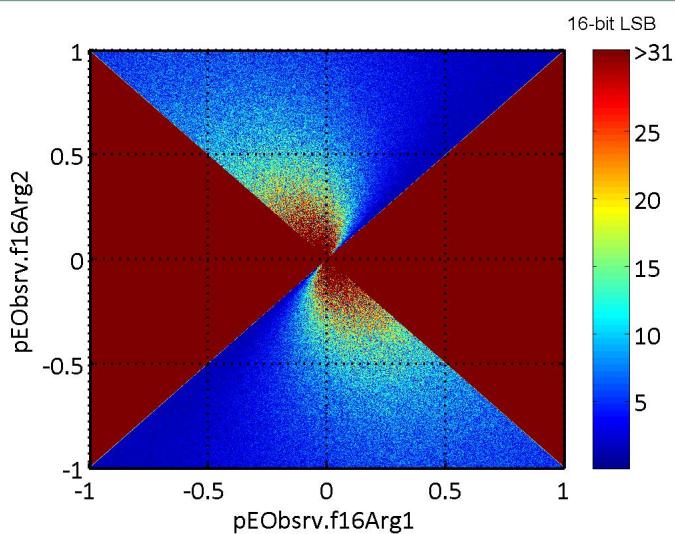
% RL circuit parameters
maxCoeffRL = max(abs([Ts/(2*Ld+Ts*Rs)*Umax/Imax, ...
                      Ts*Lq/(2*Ld+Ts*Rs)*Wmax, ...
                      Ts/(2*Ld+Ts*Rs)*Emax/Imax]));
```

```

NShiftRL = ceil(log2(maxCoeffRL)) ;
if (NShiftRL < -14)
    NShiftRL = -14;
end
if (NShiftRL > 14)
    error('Inputted parameters cannot be used - s16Shift exceeds 14');
end
pCtrl_f16IGain = (2*Ld-Ts*Rs)/(2*Ld+Ts*Rs);
pCtrl_f16IGain = round(pCtrl_f16IGain * 2^15);
pCtrl_f16IGain(pCtrl_f16IGain < -(2^15)) = -(2^15);
pCtrl_f16IGain(pCtrl_f16IGain > (2^15)-1) = (2^15)-1;
pCtrl_f16UGain = Ts/(2*Ld+Ts*Rs)*Umax/Imax*2^-NShiftRL;
pCtrl_f16UGain = round(pCtrl_f16UGain * 2^15);
pCtrl_f16UGain(pCtrl_f16UGain < -(2^15)) = -(2^15);
pCtrl_f16UGain(pCtrl_f16UGain > (2^15)-1) = (2^15)-1;
pCtrl_f16WIGain = Ts*Lq/(2*Ld+Ts*Rs)*Wmax*2^-NShiftRL;
pCtrl_f16WIGain = round(pCtrl_f16WIGain * 2^15);
pCtrl_f16WIGain(pCtrl_f16WIGain < -(2^15)) = -(2^15);
pCtrl_f16WIGain(pCtrl_f16WIGain > (2^15)-1) = (2^15)-1;
pCtrl_f16EGain = Ts/(2*Ld+Ts*Rs)*Emax/Imax*2^-NShiftRL;
pCtrl_f16EGain = round(pCtrl_f16EGain * 2^15);
pCtrl_f16EGain(pCtrl_f16EGain < -(2^15)) = -(2^15);
pCtrl_f16EGain(pCtrl_f16EGain > (2^15)-1) = (2^15)-1;
disp(['Ctrl.f16IGain = ' num2str(pCtrl_f16IGain) ';'])
disp(['Ctrl.f16UGain = ' num2str(pCtrl_f16UGain) ';'])
disp(['Ctrl.f16WIGain = ' num2str(pCtrl_f16WIGain) ';'])
disp(['Ctrl.f16EGain = ' num2str(pCtrl_f16EGain) ';'])
disp(['Ctrl.s16Shift = ' num2str(NShiftRL) ';'])

```

The accuracy of results is guaranteed for the outputs pEObsrv.f16Arg1 and pEObsrv.f16Arg2 only in cases when pParamD.u16NShift, pParamQ.u16NShift, and s16Shift are not greater than 1. There is no limit of computational error specified for the returned value. The actual error depends on the values of pEObsrv.f16Arg1 and pEObsrv.f16Arg2. The following figure shows the expected values of absolute error [16-bit LSB] contained in the returned value in the cases when all shifts are equal to 1.



**Figure 35. Absolute error of the returned value, assuming all shifts are 1**

**Note:** The BEMF observer coefficients `f16IGain`, `f16UGain`, `f16WIGain`, and `f16EGain` must not contain the largest negative value, otherwise the accuracy of the results is not guaranteed.

Keep the values of `pParamD.u16NShift`, `pParamQ.u16NShift`, and `s16Shift` within the allowed limits to prevent an overflow of intermediate results.

The function performs the fastest when `s16Shift` is equal to zero.

Due to effectivity reasons, this function is implemented using inline assembly and is therefore not ANSI-C compliant.

### Code Example

```
#include "amclib.h"

_SWLIBS_2Syst_F16          mcLab, mcUab;
_AMCLIB_BEMF_OBSRV_DO_T_F16 BemfObsrv;
tFrac16                     f16Velocity;
tFrac16                     f16Phase;

void main (void)
{
    // Clear BEMF observer state variables:

    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    _AMCLIB_BemfObsrvDQInit_F16(&BemfObsrv);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB_BemfObsrvDQInit(&BemfObsrv, F16);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 16-bit fractional implementation is selected as default.
    AMCLIB_BemfObsrvDQInit(&BemfObsrv);

    // Set BEMF observer parameters:
    BemfObsrv.pParamD.f16CC1sc      = (tFrac16)24167;
    BemfObsrv.pParamD.f16CC2sc      = (tFrac16)-20865;
    BemfObsrv.pParamD.f16UpperLimit = (tFrac16)32767;
    BemfObsrv.pParamD.f16LowerLimit = (tFrac16)-32768;
    BemfObsrv.pParamD.u16NShift    = (tU16)1;
    BemfObsrv.pParamQ.f16CC1sc      = (tFrac16)24167;
    BemfObsrv.pParamQ.f16CC2sc      = (tFrac16)-20865;
    BemfObsrv.pParamQ.f16UpperLimit = (tFrac16)32767;
    BemfObsrv.pParamQ.f16LowerLimit = (tFrac16)-32768;
    BemfObsrv.pParamQ.u16NShift    = (tU16)1;
    BemfObsrv.f16IGain     = (tFrac16)29351;
    BemfObsrv.f16UGain     = (tFrac16)29817;
    BemfObsrv.f16WIGain    = (tFrac16)32526;
    BemfObsrv.f16EGain     = (tFrac16)29817;
    BemfObsrv.s16Shift     = (tS16)-3;

    // Obtain the initial estimate of the rotor position and velocity,
    // measure the alpha/beta voltages and currents
```

```
// (...)

// Initialize the internal states of the observer to achieve
// seamless transition from an uncontrolled state of the motor
// to the full feedback control:

// Alternative 1: API call with postfix
// (only one alternative shall be used).
AMCLIB_BemfObsrvDQSetState_F16(&mcIab, &mcUab, f16Velocity,
    f16Phase, &BemfObsrv);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
AMCLIB_BemfObsrvDQSetState(&mcIab, &mcUab, f16Velocity,
    f16Phase, &BemfObsrv, F16);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 16-bit fractional implementation is selected as default.
AMCLIB_BemfObsrvDQSetState(&mcIab, &mcUab, f16Velocity,
    f16Phase, &BemfObsrv);

while(1);
}

// Periodical function or interrupt
void ISR(void)
{
    tFrac16 f16PhaseErr;

    // Read the A/D, calculate alpha-beta values, etc.
    // (...)

    // Calculate one iteration of the observer:

    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    f16PhaseErr = AMCLIB_BemfObsrvDQ_F16(&mcIab, &mcUab, f16Velocity,
        f16Phase, &BemfObsrv);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    f16PhaseErr = AMCLIB_BemfObsrvDQ(&mcIab, &mcUab, f16Velocity,
        f16Phase, &BemfObsrv, F16);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 16-bit fractional implementation is selected as default.
    f16PhaseErr = AMCLIB_BemfObsrvDQ(&mcIab, &mcUab, f16Velocity,
        f16Phase, &BemfObsrv);

    // Pass f16PhaseErr to the #AMCLIB_TrackObsrv_F16
    // (...)

}
```

### 2.2.3 Function AMCLIB\_BemfObsrvDQ\_FLT

#### Declaration

```
tFloat AMCLIB_BemfObsrvDQ_FLT(const SWLIBS_2Syst_FLT *const pIAB,
const SWLIBS_2Syst_FLT *const pUAB, tFloat fltVelocity, tFloat
fltPhase, AMCLIB_BEMF_OBSRV_DQ_T_FLT *const pCtrl);
```

#### Arguments

Table 7. AMCLIB\_BemfObsrvDQ\_FLT arguments

Type	Name	Direction	Description
const SWLIBS_2Syst_FLT *const	pIAB	input	Pointer to the structure with Alpha/Beta current components [A].
const SWLIBS_2Syst_FLT *const	pUAB	input	Pointer to the structure with Alpha/Beta voltage components [V].
tFloat	fltVelocity	input	Estimated electrical angular velocity [rad/s].
tFloat	fltPhase	input	Estimated rotor flux angle [rad], must be in range [-π, π].
AMCLIB_BEMF_OBSRV_DQ_T_FLT *const	pCtrl	input, output	Pointer to the structure with BEMF observer parameters and state.

#### Return

Phase error between the real rotating reference frame and the estimated one [rad].

#### Implementation details

Parameters of the PIrAW controllers inside the BEMF observer can be calculated using the following equations:

$$\begin{aligned} pParamD.fltCC1sc &= K_p + \frac{K_i T_s}{2} \\ pParamD.fltCC2sc &= -K_p + \frac{K_i T_s}{2} \\ pParamQ.fltCC1sc &= K_p + \frac{K_i T_s}{2} \\ pParamQ.fltCC2sc &= -K_p + \frac{K_i T_s}{2} \end{aligned}$$

Equation AMCLIB\_BemfObsrvDQ\_FLT\_Eq1

where  $T_s$  is the sampling period,  $K_p$  is the proportional gain, and  $K_i$  is the integral gain. The upper and lower limits of the PIrAW controller should be set based on the expected dynamics of the system. The gains can be calculated as follows:

$$\begin{aligned} K_p &= 2 \cdot \xi \cdot \omega_0 \cdot L_d - R_s \\ K_i &= \omega_0^2 \cdot L_d \end{aligned}$$

Equation AMCLIB\_BemfObsrvDQ\_FLT\_Eq2

where  $\xi$  is the current loop attenuation, and  $\omega_0$  is the current loop natural frequency [rad/s]. Coefficients  $\xi$  and  $\omega_0$  should correspond to the values chosen for the FOC current loop.

The winding model (R-L circuit) and cross-coupling constants can be set according to the following equations:

$$\begin{aligned} \text{fltIGain} &= \frac{2L_d T_s R_s}{2L_d + T_s R_s} \\ \text{fltUGain} &= \frac{T_s}{2L_d + T_s R_s} \\ \text{fltWIGain} &= \frac{T_s L_q}{2L_d + T_s R_s} \\ \text{fltEGain} &= \frac{T_s}{2L_d + T_s R_s} \end{aligned}$$

Equation AMCLIB\_BemfObsrvDQ\_FLT\_Eq3

The following m-script can be passed to the Matlab® command window to calculate the BEMF observer coefficients from the motor parameters:

```
% Motor parameters
% (to be set according to measurements)
Ld = 3.0e-4; % inductance in d-axis [H]
Lq = 3.0e-4; % inductance in q-axis [H]
Rs = 0.33; % resistance of one stator phase [Ω]

% Control system parameters
% (to be set according to the chosen control system dynamics)
Ts = 1e-4; % sampling period [s]
i_Ksi = 1; % current loop attenuation
i_fo = 350; % current loop natural frequency [Hz]
i_wo = 2*pi()*i_fo; % current loop natural angular frequency [rad/s]
Kp = 2*i_Ksi*i_wo*Ld-Rs;
Ki = i_wo^2*Ld;

disp('--- AMCLIB_BemfObsrvDQ_FLT coefficients ---')
% PIrAW controller coefficients
pCtrl_pParamD_fltCC1sc = Kp + Ki*Ts/2;
pCtrl_pParamD_fltCC2sc = -Kp + Ki*Ts/2;
pCtrl_pParamQ_fltCC1sc = pCtrl_pParamD_fltCC1sc;
pCtrl_pParamQ_fltCC2sc = pCtrl_pParamD_fltCC2sc;
disp(['Ctrl.pParamD.fltCC1sc = ' ...
    num2str(pCtrl_pParamD_fltCC1sc,'%1.16e\n') ';' ])
disp(['Ctrl.pParamD.fltCC2sc = ' ...
    num2str(pCtrl_pParamD_fltCC2sc,'%1.16e\n') ';' ])
disp(['Ctrl.pParamQ.fltCC1sc = ' ...
    num2str(pCtrl_pParamQ_fltCC1sc,'%1.16e\n') ';' ])
disp(['Ctrl.pParamQ.fltCC2sc = ' ...
    num2str(pCtrl_pParamQ_fltCC2sc,'%1.16e\n') ';' ])
disp(' Ctrl.pParamD.fltUpperLimit, Ctrl.pParamD.fltLowerLimit, ')
disp(' Ctrl.pParamQ.fltUpperLimit, and Ctrl.pParamQ.fltLowerLimit')
disp(' shall be set according to the expected dynamics')

% R-L circuit coefficients
pCtrl_fltIGain = (2*Ld-Ts*Rs) / (2*Ld+Ts*Rs);
pCtrl_fltUGain = Ts/(2*Ld+Ts*Rs);
pCtrl_fltWIGain = Ts*Lq/(2*Ld+Ts*Rs);
pCtrl_fltEGain = Ts/(2*Ld+Ts*Rs);
disp(['Ctrl.fltIGain = ' num2str(pCtrl_fltIGain,'%1.16e\n') ';' ])
disp(['Ctrl.fltUGain = ' num2str(pCtrl_fltUGain,'%1.16e\n') ';' ])
disp(['Ctrl.fltWIGain = ' num2str(pCtrl_fltWIGain,'%1.16e\n') ';' ])
disp(['Ctrl.fltEGain = ' num2str(pCtrl_fltEGain,'%1.16e\n') ';' ])
```

The following histogram shows the empirical probability of the relative error magnitude contained in the returned value in a typical motor control application. The x-axis scale corresponds to the floating-point mantissa binary digits; 0 = LSB, 23 = MSB.

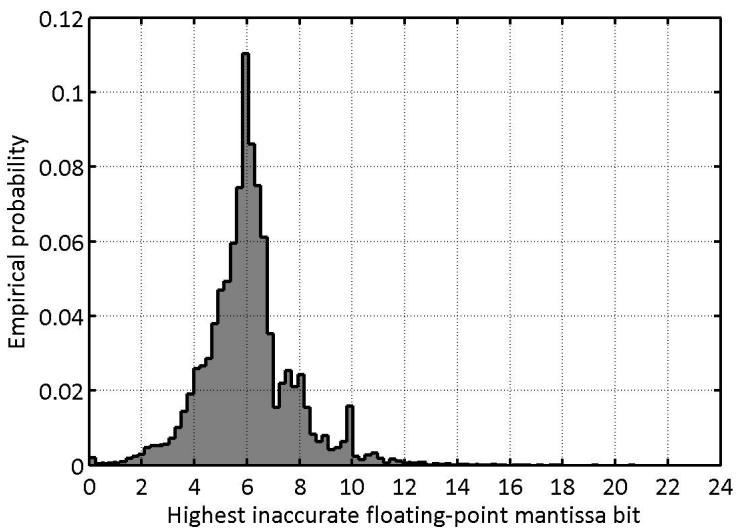


Figure 36. Histogram of the returned value error

**Note:** The function may raise floating-point exceptions (invalid operation, overflow, underflow, inexact, input denormal). The floating-point unit must be enabled in CPACR register to prevent NOCP UsageFault exception.

### Code Example

```
#include "amclib.h"

#define Ts    (1e-4F)
#define Ksi   (1.0F)
#define w0    (2.0F*3.14F*350.0F)
#define Ld    (3e-4F)
#define Lq    (3e-4F)
#define Rs    (0.33F)
#define Kp    (2.0F*Ksi*w0*Ld-Rs)
#define Ki    (w0*w0*Ld)

SWLIBS_2Syst_FLT          mcIab, mcUab;
AMCLIB_BEMF_OBSRV_DO_T_FLT BemfObsrv;
tFloat                      fltVelocity;
tFloat                      fltPhase;

void main (void)
{
    // Clear BEMF observer state variables:

    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB_BemfObsrvDOInit_FLT(&BemfObsrv);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
}
```

```
AMCLIB_BemfObsrvDQInit(&BemfObsrv, FLT);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if floating point implementation is selected as default.
AMCLIB_BemfObsrvDQInit(&BemfObsrv);

// Set BEMF observer parameters:
BemfObsrv.pParamD.fltCC1sc      = Kp+Ki*Ts/2.0F;
BemfObsrv.pParamD.fltCC2sc      = -Kp+Ki*Ts/2.0F;
BemfObsrv.pParamD.fltUpperLimit = 1000.0F;
BemfObsrv.pParamD.fltLowerLimit = -1000.0F;
BemfObsrv.pParamQ.fltCC1sc      = Kp+Ki*Ts/2.0F;
BemfObsrv.pParamQ.fltCC2sc      = -Kp+Ki*Ts/2.0F;
BemfObsrv.pParamQ.fltUpperLimit = 1000.0F;
BemfObsrv.pParamQ.fltLowerLimit = -1000.0F;
BemfObsrv.fltIGain   = (2.0F*Ld-Ts*Rs)/(2.0F*Ld+Ts*Rs);
BemfObsrv.fltUGain   = Ts/(2.0F*Ld+Ts*Rs);
BemfObsrv.fltWIGain  = Ts*Lq/(2.0F*Ld+Ts*Rs);
BemfObsrv.fltEGain   = Ts/(2.0F*Ld+Ts*Rs);

// Obtain the initial estimate of the rotor position and velocity,
// measure the alpha/beta voltages and currents
// (...)

// Initialize the internal states of the observer to achieve
// seamless transition from an uncontrolled state of the motor
// to the full feedback control:

// Alternative 1: API call with postfix
// (only one alternative shall be used).
AMCLIB_BemfObsrvDQSetState_FLT(&mciLab, &mcUab, fltVelocity,
                               fltPhase, &BemfObsrv);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
AMCLIB_BemfObsrvDQSetState(&mciLab, &mcUab, fltVelocity,
                           fltPhase, &BemfObsrv, FLT);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if floating point implementation is selected as default.
AMCLIB_BemfObsrvDQSetState(&mciLab, &mcUab, fltVelocity,
                           fltPhase, &BemfObsrv);

while(1);

// Periodical function or interrupt
void ISR(void)
{
    tFloat fltPhaseErr;

    // Read the A/D, calculate alpha-beta values, etc.
    // (...)

    // Calculate one iteration of the observer:

    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
```

```

    fltPhaseErr = AMCLIB_BemfObsrvDQ_FLT(&mcIab, &mcUab, fltVelocity,
                                            fltPhase, &BemfObsrv);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    fltPhaseErr = AMCLIB_BemfObsrvDQ(&mcIab, &mcUab, fltVelocity,
                                      fltPhase, &BemfObsrv,FLT);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if floating point implementation is selected as default.
    fltPhaseErr = AMCLIB_BemfObsrvDQ(&mcIab, &mcUab, fltVelocity,
                                      fltPhase, &BemfObsrv);

    // Pass fltPhaseErr to the #AMCLIB_TrackObsrv_FLT
    // (...)

}

```

## 2.2.4 Function AMCLIB\_BemfObsrvDQInit

### Description

This function initializes all state variables of the BEMF observer to zero.

**Note:** The input/output pointers must contain valid addresses, otherwise a fault may occur (MemManage, BusFault, UsageFault, HardFault).

### Re-entrancy

The function is re-entrant for a different pCtrl.

#### 2.2.4.1 Function AMCLIB\_BemfObsrvDQInit\_F32

##### Declaration

```
void AMCLIB_BemfObsrvDQInit_F32(AMCLIB\_BEMF\_OBSRV\_DQ\_T\_F32 *const pCtrl);
```

##### Arguments

Table 8. AMCLIB\_BemfObsrvDQInit\_F32 arguments

Type	Name	Direction	Description
<a href="#">AMCLIB_BEMF_OBSRV_DQ_T_F32</a> *const	pCtrl	input, output	Pointer to the structure with BEMF observer parameters and state.

### Code Example

```
#include "amclib.h"

SWLIBS\_2Syst\_F32 mcIab, mcUab;
AMCLIB\_BEMF\_OBSRV\_DQ\_T\_F32 BemfObsrv;
tFrac32 f32Velocity;
tFrac32 f32Phase;
```

```
void main (void)
{
    // Clear BEMF observer state variables:

    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB_BemfObsrvDQInit_F32(&BemfObsrv);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB_BemfObsrvDQInit(&BemfObsrv, F32);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 32-bit fractional implementation is selected as default.
    AMCLIB_BemfObsrvDQInit(&BemfObsrv);

    // Set BEMF observer parameters:
    BemfObsrv.pParamD.f32CC1sc      = (tFrac32)1583784859;
    BemfObsrv.pParamD.f32CC2sc      = (tFrac32)-1367421132;
    BemfObsrv.pParamD.f32UpperLimit = (tFrac32)2147483647;
    BemfObsrv.pParamD.f32LowerLimit = (tFrac32)-2147483648;
    BemfObsrv.pParamD.u16NShift    = (tU16)1;
    BemfObsrv.pParamQ.f32CC1sc      = (tFrac32)1583784859;
    BemfObsrv.pParamQ.f32CC2sc      = (tFrac32)-1367421132;
    BemfObsrv.pParamQ.f32UpperLimit = (tFrac32)2147483647;
    BemfObsrv.pParamQ.f32LowerLimit = (tFrac32)-2147483648;
    BemfObsrv.pParamQ.u16NShift    = (tU16)1;
    BemfObsrv.f32IGain   = (tFrac32)1923575400;
    BemfObsrv.f32UGain   = (tFrac32)1954108343;
    BemfObsrv.f32WIGain  = (tFrac32)2131606518;
    BemfObsrv.f32EGain   = (tFrac32)1954108343;
    BemfObsrv.s16Shift   = (ts16)-3;

    // Obtain the initial estimate of the rotor position and velocity,
    // measure the alpha/beta voltages and currents
    // (...)

    // Initialize the internal states of the observer to achieve
    // seamless transition from an uncontrolled state of the motor
    // to the full feedback control:

    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB_BemfObsrvDQSetState_F32(&mcIab, &mcUab, f32Velocity,
                                    f32Phase, &BemfObsrv);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB_BemfObsrvDQSetState(&mcIab, &mcUab, f32Velocity,
                                f32Phase, &BemfObsrv, F32);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 32-bit fractional implementation is selected as default.
    AMCLIB_BemfObsrvDQSetState(&mcIab, &mcUab, f32Velocity,
                                f32Phase, &BemfObsrv);

    while(1);
}
```

```

// Periodical function or interrupt
void ISR(void)
{
    tFrac32 f32PhaseErr;

    // Read the A/D, calculate alpha-beta values, etc.
    // (...)

    // Calculate one iteration of the observer:

    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    f32PhaseErr = AMCLIB_BemfObsrvDO_F32(&mcLab, &mcUab, f32Velocity,
                                             f32Phase, &BemfObsrv);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    f32PhaseErr = AMCLIB_BemfObsrvDQ(&mcLab, &mcUab, f32Velocity,
                                       f32Phase, &BemfObsrv, F32);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 32-bit fractional implementation is selected as default.
    f32PhaseErr = AMCLIB_BemfObsrvDQ(&mcLab, &mcUab, f32Velocity,
                                       f32Phase, &BemfObsrv);

    // Pass f32PhaseErr to the #AMCLIB_TrackObsrv_F32
    // (...)

}

```

#### 2.2.4.2 Function AMCLIB\_BemfObsrvDQInit\_F16

##### Declaration

```
void AMCLIB_BemfObsrvDQInit_F16(AMCLIB_BEMF_OBSRV_DO_T_F16 *const pCtrl);
```

##### Arguments

**Table 9. AMCLIB\_BemfObsrvDQInit\_F16 arguments**

Type	Name	Direction	Description
<u>AMCLIB_BEMF_OBSRV_DO_T_F16</u> *const	pCtrl	input, output	Pointer to the structure with BEMF observer parameters and state.

##### Code Example

```

#include "amclib.h"

SWLIBS_2Syst_F16
AMCLIB_BEMF_OBSRV_DO_T_F16
tFrac16
tFrac16

void main (void)
{

```

```
// Clear BEMF observer state variables:  
  
// Alternative 1: API call with postfix  
// (only one alternative shall be used).  
AMCLIB_BemfObsrvDQInit_F16(&BemfObsrv);  
  
// Alternative 2: API call with implementation parameter  
// (only one alternative shall be used).  
AMCLIB_BemfObsrvDQInit(&BemfObsrv, F16);  
  
// Alternative 3: API call with global configuration of implementation  
// (only one alternative shall be used). This alternative is available  
// only if 16-bit fractional implementation is selected as default.  
AMCLIB_BemfObsrvDQInit(&BemfObsrv);  
  
// Set BEMF observer parameters:  
BemfObsrv.pParamD.f16CC1sc = (tFrac16)24167;  
BemfObsrv.pParamD.f16CC2sc = (tFrac16)-20865;  
BemfObsrv.pParamD.f16UpperLimit = (tFrac16)32767;  
BemfObsrv.pParamD.f16LowerLimit = (tFrac16)-32768;  
BemfObsrv.pParamD.u16NShift = (tU16)1;  
BemfObsrv.pParamQ.f16CC1sc = (tFrac16)24167;  
BemfObsrv.pParamQ.f16CC2sc = (tFrac16)-20865;  
BemfObsrv.pParamQ.f16UpperLimit = (tFrac16)32767;  
BemfObsrv.pParamQ.f16LowerLimit = (tFrac16)-32768;  
BemfObsrv.pParamQ.u16NShift = (tU16)1;  
BemfObsrv.f16IGain = (tFrac16)29351;  
BemfObsrv.f16UGain = (tFrac16)29817;  
BemfObsrv.f16WIGain = (tFrac16)32526;  
BemfObsrv.f16EGain = (tFrac16)29817;  
BemfObsrv.s16Shift = (tS16)-3;  
  
// Obtain the initial estimate of the rotor position and velocity,  
// measure the alpha/beta voltages and currents  
// (...)  
  
// Initialize the internal states of the observer to achieve  
// seamless transition from an uncontrolled state of the motor  
// to the full feedback control:  
  
// Alternative 1: API call with postfix  
// (only one alternative shall be used).  
AMCLIB_BemfObsrvDQSetState_F16(&mcIab, &mcUab, f16Velocity,  
f16Phase, &BemfObsrv);  
  
// Alternative 2: API call with implementation parameter  
// (only one alternative shall be used).  
AMCLIB_BemfObsrvDQSetState(&mcIab, &mcUab, f16Velocity,  
f16Phase, &BemfObsrv, F16);  
  
// Alternative 3: API call with global configuration of implementation  
// (only one alternative shall be used). This alternative is available  
// only if 16-bit fractional implementation is selected as default.  
AMCLIB_BemfObsrvDQSetState(&mcIab, &mcUab, f16Velocity,  
f16Phase, &BemfObsrv);  
  
while(1);  
}  
  
// Periodical function or interrupt
```

```

void ISR(void)
{
    tFrac16 f16PhaseErr;

    // Read the A/D, calculate alpha-beta values, etc.
    // (...)

    // Calculate one iteration of the observer:

    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    f16PhaseErr = AMCLIB_BemfObsrvDO_F16(&mcLab, &mcUab, f16Velocity,
                                             f16Phase, &BemfObsrv);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    f16PhaseErr = AMCLIB_BemfObsrvDQ(&mcLab, &mcUab, f16Velocity,
                                       f16Phase, &BemfObsrv, F16);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 16-bit fractional implementation is selected as default.
    f16PhaseErr = AMCLIB_BemfObsrvDQ(&mcLab, &mcUab, f16Velocity,
                                       f16Phase, &BemfObsrv);

    // Pass f16PhaseErr to the #AMCLIB_TrackObsrv_F16
    // (...)

}

```

#### 2.2.4.3 Function AMCLIB\_BemfObsrvDQInit\_FLT

##### Declaration

```
void AMCLIB_BemfObsrvDQInit_FLT(AMCLIB_BEMF_OBSRV_DQ_T_FLT *const pCtrl);
```

##### Arguments

**Table 10. AMCLIB\_BemfObsrvDQInit\_FLT arguments**

Type	Name	Direction	Description
<u>AMCLIB_BEMF_OBSRV_DQ_T_FLT</u> *const	pCtrl	input, output	Pointer to the structure with BEMF observer parameters and state.

**Note:** The function may raise floating-point exceptions (invalid operation, overflow, underflow, inexact, input denormal). The floating-point unit must be enabled in CPACR register to prevent NOCP UsageFault exception.

##### Code Example

```

#include "amclib.h"

#define Ts  (1e-4F)
#define KsI (1.0F)
#define w0  (2.0F*3.14F*350.0F)
#define Ld  (3e-4F)

```

```
#define Lq  (3e-4F)
#define Rs  (0.33F)
#define Kp   (2.0F*Ksi*w0*Ld-Rs)
#define Ki   (w0*w0*Ld)

_SWLIBS_2Syst_FLT          mcIab, mcUab;
_AMCLIB_BEMF_OBSRV_DQ_T_FLT BemfObsrv;
tFloat                      fltVelocity;
tFloat                      fltPhase;

void main (void)
{
    // Clear BEMF observer state variables:

    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB_BemfObsrvDQInit_FLT(&BemfObsrv);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB_BemfObsrvDQInit(&BemfObsrv, FLT);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if floating point implementation is selected as default.
    AMCLIB_BemfObsrvDQInit(&BemfObsrv);

    // Set BEMF observer parameters:
    BemfObsrv.pParamD.fltCC1sc      = Kp+Ki*Ts/2.0F;
    BemfObsrv.pParamD.fltCC2sc      = -Kp+Ki*Ts/2.0F;
    BemfObsrv.pParamD.fltUpperLimit = 1000.0F;
    BemfObsrv.pParamD.fltLowerLimit = -1000.0F;
    BemfObsrv.pParamQ.fltCC1sc      = Kp+Ki*Ts/2.0F;
    BemfObsrv.pParamQ.fltCC2sc      = -Kp+Ki*Ts/2.0F;
    BemfObsrv.pParamQ.fltUpperLimit = 1000.0F;
    BemfObsrv.pParamQ.fltLowerLimit = -1000.0F;
    BemfObsrv.fltIGain = (2.0F*Ld-Ts*Rs) / (2.0F*Ld+Ts*Rs);
    BemfObsrv.fltUGain = Ts / (2.0F*Ld+Ts*Rs);
    BemfObsrv.fltWIGain = Ts*Lq / (2.0F*Ld+Ts*Rs);
    BemfObsrv.fltEGain = Ts / (2.0F*Ld+Ts*Rs);

    // Obtain the initial estimate of the rotor position and velocity,
    // measure the alpha/beta voltages and currents
    // (...)

    // Initialize the internal states of the observer to achieve
    // seamless transition from an uncontrolled state of the motor
    // to the full feedback control:

    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB_BemfObsrvDQSetState_FLT(&mcIab, &mcUab, fltVelocity,
                                    fltPhase, &BemfObsrv);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB_BemfObsrvDQSetState(&mcIab, &mcUab, fltVelocity,
                                fltPhase, &BemfObsrv, FLT);

    // Alternative 3: API call with global configuration of implementation
```

```
// (only one alternative shall be used). This alternative is available
// only if floating point implementation is selected as default.
AMCLIB_BemfObsrvDQSetState(&mciab, &mcuab, fltVelocity,
    fltPhase, &BemfObsrv);

while(1);

// Periodical function or interrupt
void ISR(void)
{
    tFloat fltPhaseErr;

    // Read the A/D, calculate alpha-beta values, etc.
    // (...)

    // Calculate one iteration of the observer:

    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    fltPhaseErr = AMCLIB_BemfObsrvDO_FLT(&mciab, &mcuab, fltVelocity,
        fltPhase, &BemfObsrv);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    fltPhaseErr = AMCLIB_BemfObsrvDQ(&mciab, &mcuab, fltVelocity,
        fltPhase, &BemfObsrv,FLT);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if floating point implementation is selected as default.
    fltPhaseErr = AMCLIB_BemfObsrvDQ(&mciab, &mcuab, fltVelocity,
        fltPhase, &BemfObsrv);

    // Pass fltPhaseErr to the #AMCLIB_TrackObsrv_FLT
    // (...)

}
```

## 2.2.5 Function AMCLIB\_BemfObsrvDQSetState

### Description

This function sets the BEMF observer state variables to achieve zero difference between the measured and the predicted current and a zero voltage sum on the input of the current observer. The input voltages and currents are supplied in a stationary reference frame  $\alpha/\beta$  and should be derived from the actual measured 3-phase motor voltages and currents (refer to the function [GMCLIB\\_Clark](#) for a transformation between the 3-phase coordinate system and the 2-phase  $\alpha/\beta$  reference frame).

Setting the input flux angle and the angular velocity in accordance with the actual estimated position and velocity of the rotor enables seamless transition from an uncontrolled rotation of the rotor into a controlled state. A robust set of initial estimates can be obtained from function [AMCLIB\\_Windmilling](#).

**Note:** The observer parameters must be already set in the state structure pointed to by *pCtrl* before this function can be called.

**Note:** The input/output pointers must contain valid addresses, otherwise a fault may occur (MemManage, BusFault, UsageFault, HardFault).

### Re-entrancy

The function is re-entrant for a different pCtrl.

#### 2.2.5.1 Function AMCLIB\_BemfObsrvDQSetState\_F32

##### Declaration

```
void AMCLIB_BemfObsrvDQSetState_F32(const SWLIBS\_2Syst\_F32 *const pIAB, const SWLIBS\_2Syst\_F32 *const pUAB, tFrac32 f32Velocity, tFrac32 f32Phase, AMCLIB\_BEMF\_OBSRV\_DQ\_T\_F32 *const pCtrl);
```

##### Arguments

Table 11. AMCLIB\_BemfObsrvDQSetState\_F32 arguments

Type	Name	Direction	Description
const <a href="#">SWLIBS_2Syst_F32</a> *const	pIAB	input	Pointer to the structure with Alpha/Beta current components.
const <a href="#">SWLIBS_2Syst_F32</a> *const	pUAB	input	Pointer to the structure with Alpha/Beta voltage components.
<a href="#">tFrac32</a>	f32Velocity	input	Estimated electrical angular velocity.
<a href="#">tFrac32</a>	f32Phase	input	Estimated rotor flux angle.
<a href="#">AMCLIB_BEMF_OBSRV_DQ_T_F32</a> *const	pCtrl	input, output	Pointer to the structure with BEMF observer parameters and state.

##### Implementation details

The scaling of the input values follows the same rules as the corresponding inputs of [AMCLIB\\_BemfObsrvDQ\\_F32](#).

##### Code Example

```
#include "amclib.h"

SWLIBS\_2Syst\_F32 mcIab, mcUab;
AMCLIB\_BEMF\_OBSRV\_DQ\_T\_F32 BemfObsrv;
tFrac32 f32Velocity;
tFrac32 f32Phase;

void main (void)
{
    // Clear BEMF observer state variables:

    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB\_BemfObsrvDQInit\_F32(&BemfObsrv);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB_BemfObsrvDQInit(&BemfObsrv, F32);

    // Alternative 3: API call with global configuration of implementation
```

```
// (only one alternative shall be used). This alternative is available
// only if 32-bit fractional implementation is selected as default.
AMCLIB_BemfObsrvDQInit(&BemfObsrv);

// Set BEMF observer parameters:
BemfObsrv.pParamD.f32CC1sc = (tFrac32)1583784859;
BemfObsrv.pParamD.f32CC2sc = (tFrac32)-1367421132;
BemfObsrv.pParamD.f32UpperLimit = (tFrac32)2147483647;
BemfObsrv.pParamD.f32LowerLimit = (tFrac32)-2147483648;
BemfObsrv.pParamD.u16NShift = (tU16)1;
BemfObsrv.pParamQ.f32CC1sc = (tFrac32)1583784859;
BemfObsrv.pParamQ.f32CC2sc = (tFrac32)-1367421132;
BemfObsrv.pParamQ.f32UpperLimit = (tFrac32)2147483647;
BemfObsrv.pParamQ.f32LowerLimit = (tFrac32)-2147483648;
BemfObsrv.pParamQ.u16NShift = (tU16)1;
BemfObsrv.f32IGain = (tFrac32)1923575400;
BemfObsrv.f32UGain = (tFrac32)1954108343;
BemfObsrv.f32WIGain = (tFrac32)2131606518;
BemfObsrv.f32EGain = (tFrac32)1954108343;
BemfObsrv.s16Shift = (tS16)-3;

// Obtain the initial estimate of the rotor position and velocity,
// measure the alpha/beta voltages and currents
// (...)

// Initialize the internal states of the observer to achieve
// seamless transition from an uncontrolled state of the motor
// to the full feedback control:

// Alternative 1: API call with postfix
// (only one alternative shall be used).
AMCLIB_BemfObsrvDQSetState_F32(&mciLab, &mcUab, f32Velocity,
    f32Phase, &BemfObsrv);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
AMCLIB_BemfObsrvDQSetState(&mciLab, &mcUab, f32Velocity,
    f32Phase, &BemfObsrv, F32);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 32-bit fractional implementation is selected as default.
AMCLIB_BemfObsrvDQSetState(&mciLab, &mcUab, f32Velocity,
    f32Phase, &BemfObsrv);

while(1);
}

// Periodical function or interrupt
void ISR(void)
{
    tFrac32 f32PhaseErr;

    // Read the A/D, calculate alpha-beta values, etc.
    // (...)

    // Calculate one iteration of the observer:

    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
```

```

f32PhaseErr = AMCLIB\_BemfObsrvDO\_F32(&mcIab, &mcUab, f32Velocity,
    f32Phase, &BemfObsrv);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
f32PhaseErr = AMCLIB\_BemfObsrvDQ(&mcIab, &mcUab, f32Velocity,
    f32Phase, &BemfObsrv, F32);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 32-bit fractional implementation is selected as default.
f32PhaseErr = AMCLIB\_BemfObsrvDQ(&mcIab, &mcUab, f32Velocity,
    f32Phase, &BemfObsrv);

// Pass f32PhaseErr to the #AMCLIB_TrackObsrv_F32
// (...)
}

```

### 2.2.5.2 Function [AMCLIB\\_BemfObsrvDQSetState\\_F16](#)

#### Declaration

```
void AMCLIB_BemfObsrvDQSetState_F16(const SWLIBS\_2Syst\_F16 *const
    pIAB, const SWLIBS\_2Syst\_F16 *const pUAB, tFrac16 f16Velocity,
    tFrac16 f16Phase, AMCLIB\_BEMF\_OBSRV\_DQ\_T\_F16 *const pCtrl);
```

#### Arguments

**Table 12. [AMCLIB\\_BemfObsrvDQSetState\\_F16](#) arguments**

Type	Name	Direction	Description
const <a href="#">SWLIBS_2Syst_F16</a> *const	pIAB	input	Pointer to the structure with Alpha/Beta current components.
const <a href="#">SWLIBS_2Syst_F16</a> *const	pUAB	input	Pointer to the structure with Alpha/Beta voltage components.
<a href="#">tFrac16</a>	f16Velocity	input	Estimated electrical angular velocity.
<a href="#">tFrac16</a>	f16Phase	input	Estimated rotor flux angle.
<a href="#">AMCLIB_BEMF_OBSRV_DQ_T_F16</a> *const	pCtrl	input, output	Pointer to the structure with BEMF observer parameters and state.

#### Implementation details

The scaling of the input values follows the same rules as the corresponding inputs of [AMCLIB\\_BemfObsrvDQ\\_F16](#).

#### Code Example

```

#include "amclib.h"

SWLIBS\_2Syst\_F16 mcIab, mcUab;
AMCLIB\_BEMF\_OBSRV\_DQ\_T\_F16 BemfObsrv;
tFrac16 f16Velocity;
tFrac16 f16Phase;

void main (void)

```

```
{  
    // Clear BEMF observer state variables:  
  
    // Alternative 1: API call with postfix  
    // (only one alternative shall be used).  
    AMCLIB_BemfObsrvDQInit_F16(&BemfObsrv);  
  
    // Alternative 2: API call with implementation parameter  
    // (only one alternative shall be used).  
    AMCLIB_BemfObsrvDQInit(&BemfObsrv, F16);  
  
    // Alternative 3: API call with global configuration of implementation  
    // (only one alternative shall be used). This alternative is available  
    // only if 16-bit fractional implementation is selected as default.  
    AMCLIB_BemfObsrvDQInit(&BemfObsrv);  
  
    // Set BEMF observer parameters:  
    BemfObsrv.pParamD.f16CC1sc      = (tFrac16)24167;  
    BemfObsrv.pParamD.f16CC2sc      = (tFrac16)-20865;  
    BemfObsrv.pParamD.f16UpperLimit = (tFrac16)32767;  
    BemfObsrv.pParamD.f16LowerLimit = (tFrac16)-32768;  
    BemfObsrv.pParamD.u16NShift     = (tU16)1;  
    BemfObsrv.pParamQ.f16CC1sc      = (tFrac16)24167;  
    BemfObsrv.pParamQ.f16CC2sc      = (tFrac16)-20865;  
    BemfObsrv.pParamQ.f16UpperLimit = (tFrac16)32767;  
    BemfObsrv.pParamQ.f16LowerLimit = (tFrac16)-32768;  
    BemfObsrv.pParamQ.u16NShift     = (tU16)1;  
    BemfObsrv.f16IGain      = (tFrac16)29351;  
    BemfObsrv.f16UGain       = (tFrac16)29817;  
    BemfObsrv.f16WIGain      = (tFrac16)32526;  
    BemfObsrv.f16EGain       = (tFrac16)29817;  
    BemfObsrv.s16Shift       = (tS16)-3;  
  
    // Obtain the initial estimate of the rotor position and velocity,  
    // measure the alpha/beta voltages and currents  
    // (...)  
  
    // Initialize the internal states of the observer to achieve  
    // seamless transition from an uncontrolled state of the motor  
    // to the full feedback control:  
  
    // Alternative 1: API call with postfix  
    // (only one alternative shall be used).  
    AMCLIB_BemfObsrvDQSetState_F16(&mciLab, &mcUab, f16Velocity,  
                                    f16Phase, &BemfObsrv);  
  
    // Alternative 2: API call with implementation parameter  
    // (only one alternative shall be used).  
    AMCLIB_BemfObsrvDQSetState(&mciLab, &mcUab, f16Velocity,  
                               f16Phase, &BemfObsrv, F16);  
  
    // Alternative 3: API call with global configuration of implementation  
    // (only one alternative shall be used). This alternative is available  
    // only if 16-bit fractional implementation is selected as default.  
    AMCLIB_BemfObsrvDQSetState(&mciLab, &mcUab, f16Velocity,  
                               f16Phase, &BemfObsrv);  
  
    while(1);  
}
```

```

// Periodical function or interrupt
void ISR(void)
{
    tFrac16 f16PhaseErr;

    // Read the A/D, calculate alpha-beta values, etc.
    // (...)

    // Calculate one iteration of the observer:

    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    f16PhaseErr = AMCLIB_BemfObsrvDQ_F16(&mcIab, &mcUab, f16Velocity,
                                           f16Phase, &BemfObsrv);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    f16PhaseErr = AMCLIB_BemfObsrvDQ(&mcIab, &mcUab, f16Velocity,
                                       f16Phase, &BemfObsrv, F16);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 16-bit fractional implementation is selected as default.
    f16PhaseErr = AMCLIB_BemfObsrvDQ(&mcIab, &mcUab, f16Velocity,
                                       f16Phase, &BemfObsrv);

    // Pass f16PhaseErr to the #AMCLIB_TrackObsrv_F16
    // (...)

}

```

### 2.2.5.3 Function AMCLIB\_BemfObsrvDQSetState\_FLT

#### Declaration

```
void AMCLIB_BemfObsrvDQSetState_FLT(const SWLIBS_2Syst_FLT *const
                                      pIAB, const SWLIBS_2Syst_FLT *const pUAB, tFloat fltVelocity,
                                      tFloat fltPhase, AMCLIB_BEMF_OBSRV_DQ_T_FLT *const pCtrl);
```

#### Arguments

**Table 13. AMCLIB\_BemfObsrvDQSetState\_FLT arguments**

Type	Name	Direction	Description
const SWLIBS_2Syst_FLT *const	pIAB	input	Pointer to the structure with Alpha/Beta current components [A].
const SWLIBS_2Syst_FLT *const	pUAB	input	Pointer to the structure with Alpha/Beta voltage components [V].
tFloat	fltVelocity	input	Estimated electrical angular velocity [rad/s].
tFloat	fltPhase	input	Estimated rotor flux angle [rad], must be in range [-π, π].
AMCLIB_BEMF_OBSRV_DQ_T_FLT *const	pCtrl	input, output	Pointer to the structure with BEMF observer parameters and state.

**Note:** The function may raise floating-point exceptions (invalid operation, overflow, underflow, inexact, input denormal). The floating-point unit must be enabled in CPACR register to prevent NOCP UsageFault exception.

## Code Example

```
#include "amclib.h"

#define Ts    (1e-4F)
#define KsI   (1.0F)
#define w0    (2.0F*3.14F*350.0F)
#define Ld    (3e-4F)
#define Lq    (3e-4F)
#define Rs    (0.33F)
#define Kp    (2.0F*KsI*w0*Ld-Rs)
#define Ki    (w0*w0*Ld)

_SWLIBS_2Syst_FLT          mcIab, mcUab;
AMCLIB_BEMF_OBSRV_DQ_T_FLT BemfObsrv;
tFloat                      fltVelocity;
tFloat                      fltPhase;

void main (void)
{
    // Clear BEMF observer state variables:

    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB_BemfObsrvDQInit_FLT(&BemfObsrv);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB_BemfObsrvDQInit(&BemfObsrv, FLT);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if floating point implementation is selected as default.
    AMCLIB_BemfObsrvDQInit(&BemfObsrv);

    // Set BEMF observer parameters:
    BemfObsrv.pParamD.fltCC1sc      = Kp+Ki*Ts/2.0F;
    BemfObsrv.pParamD.fltCC2sc      = -Kp+Ki*Ts/2.0F;
    BemfObsrv.pParamD.fltUpperLimit = 1000.0F;
    BemfObsrv.pParamD.fltLowerLimit = -1000.0F;
    BemfObsrv.pParamQ.fltCC1sc      = Kp+Ki*Ts/2.0F;
    BemfObsrv.pParamQ.fltCC2sc      = -Kp+Ki*Ts/2.0F;
    BemfObsrv.pParamQ.fltUpperLimit = 1000.0F;
    BemfObsrv.pParamQ.fltLowerLimit = -1000.0F;
    BemfObsrv.fltIGain   = (2.0F*Ld-Ts*Rs) / (2.0F*Ld+Ts*Rs);
    BemfObsrv.fltUGain   = Ts / (2.0F*Ld+Ts*Rs);
    BemfObsrv.fltWIGain  = Ts*Lq / (2.0F*Ld+Ts*Rs);
    BemfObsrv.fltEGain   = Ts / (2.0F*Ld+Ts*Rs);

    // Obtain the initial estimate of the rotor position and velocity,
    // measure the alpha/beta voltages and currents
    // (...)

    // Initialize the internal states of the observer to achieve
    // seamless transition from an uncontrolled state of the motor
    // to the full feedback control:

    // Alternative 1: API call with postfix
```

```
// (only one alternative shall be used).
AMCLIB_BemfObsrvDQSetState_FLT(&mcIab, &mcUab, fltVelocity,
    fltPhase, &BemfObsrv);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
AMCLIB_BemfObsrvDQSetState(&mcIab, &mcUab, fltVelocity,
    fltPhase, &BemfObsrv,FLT);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if floating point implementation is selected as default.
AMCLIB_BemfObsrvDQSetState(&mcIab, &mcUab, fltVelocity,
    fltPhase, &BemfObsrv);

while(1);

// Periodical function or interrupt
void ISR(void)
{
    tFloat fltPhaseErr;

    // Read the A/D, calculate alpha-beta values, etc.
    // (...)

    // Calculate one iteration of the observer:

    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    fltPhaseErr = AMCLIB_BemfObsrvDQ_FLT(&mcIab, &mcUab, fltVelocity,
        fltPhase, &BemfObsrv);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    fltPhaseErr = AMCLIB_BemfObsrvDQ(&mcIab, &mcUab, fltVelocity,
        fltPhase, &BemfObsrv,FLT);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if floating point implementation is selected as default.
    fltPhaseErr = AMCLIB_BemfObsrvDQ(&mcIab, &mcUab, fltVelocity,
        fltPhase, &BemfObsrv);

    // Pass fltPhaseErr to the #AMCLIB_TrackObsrv_FLT
    // (...)

}
```

## 2.3 Function AMCLIB\_CurrentLoop

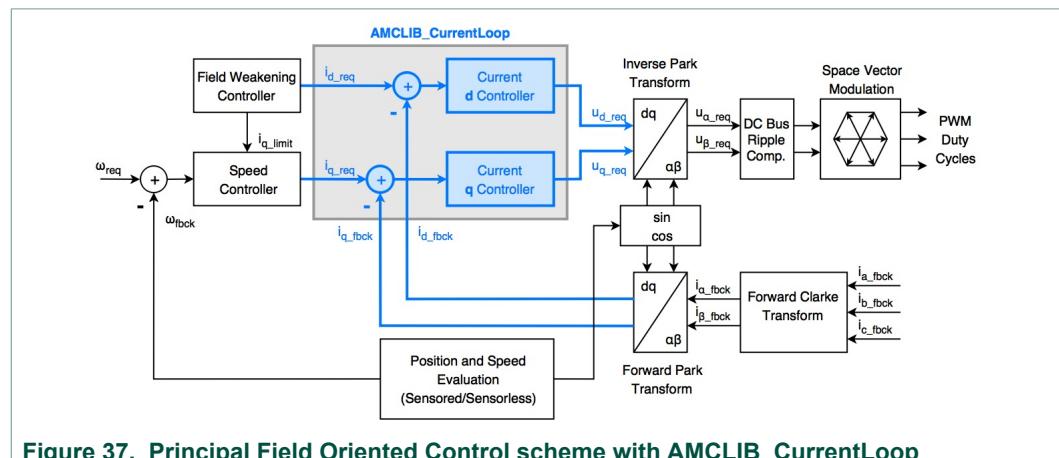
This library function implements the current control loop which is an integral part of FOC and represents the most inner loop in the cascade control structure.

### Description

Current control is essential in a variable-frequency drive control methods known as field-oriented control (FOC). The transformation of the stator phase currents into the orthogonal rotational two-phase system results into two independent current

components. The direct (d-axis) current component is known as flux-producing component and the orthogonal (q-axis) component to the magnetic flux is known as torque-producing component. The current control loop is used to control the torque and magnetic flux of the 3-phase motor with high accuracy, dynamics and bandwidth. To achieve this, PI controllers are typically used to control measured current components to their reference values. These reference values are usually given by an outer loop speed and field-weakening controllers.

AMCLIB\_CurrentLoop implements two current PI controllers to control both components of the current vector separately, as highlighted in [Figure 37](#).



**Figure 37. Principal Field Oriented Control scheme with AMCLIB\_CurrentLoop**

Before using the FOC with a particular motor, the user needs to provide a set of coefficients through the pCtrl input pointer. The controller coefficient values can be calculated from motor parameters. A method for measuring the motor parameters is described in PMSM Electrical Parameters Measurement (document [AN4680](#)).

Refer to the following resources to find out how the NXP motor control tuning and debugging tools for NXP microcontrollers can help you deploy the FOC in your application:

- [AN4642](#) - Motor Control Application Tuning (MCAT) Tool for 3-Phase PMSM
- [FREEMASTER](#) - FreeMASTER Run-Time Debugging Tool

**Note:** The input/output pointers must contain valid addresses, otherwise a fault may occur (MemManage, BusFault, UsageFault, HardFault).

## Re-entrancy

The function is re-entrant for a different pCtrl.

### 2.3.1 Function AMCLIB\_CurrentLoop\_F32

#### Declaration

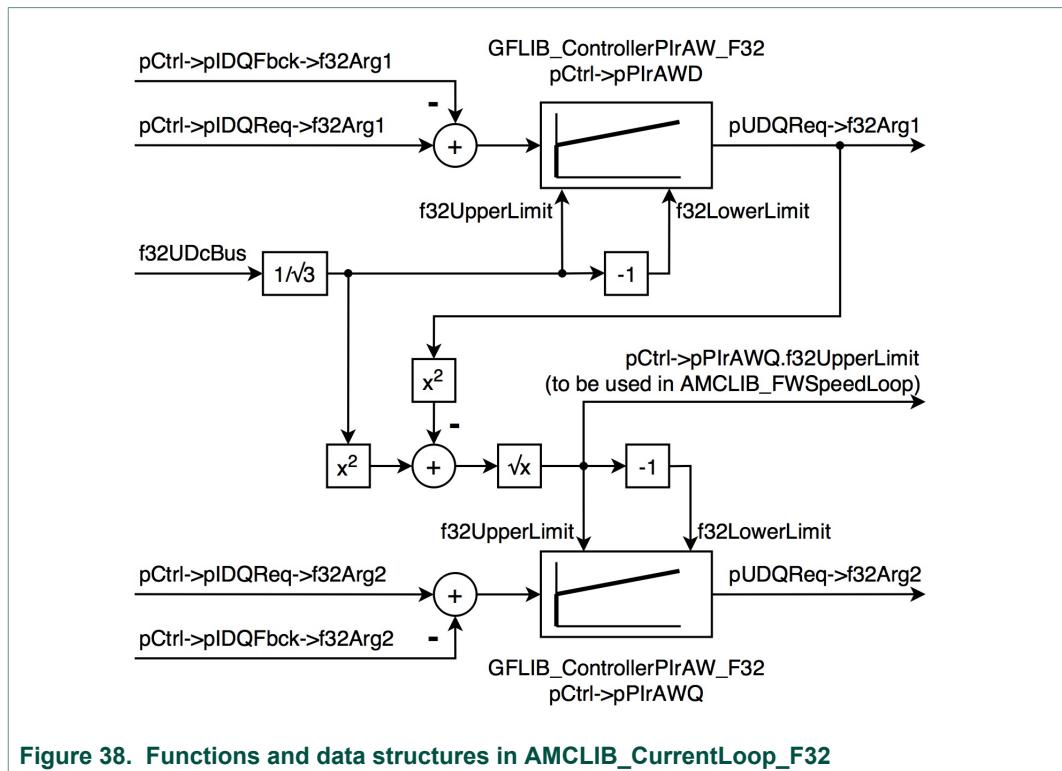
```
void AMCLIB_CurrentLoop_F32 (tFrac32 f32UDcBus, SWLIBS_2Syst_F32
*const pUDQReq, AMCLIB_CURRENT_LOOP_T_F32 *pCtrl);
```

**Arguments****Table 14. AMCLIB\_CurrentLoop\_F32 arguments**

Type	Name	Direction	Description
tFrac32	f32UDcBus	input	DC bus voltage.
SWLIBS_2Syst_F32 *const	pUDQReq	output	Pointer to the structure with the required stator voltages in the two-phase rotational orthogonal system (d-q).
AMCLIB_CURRENT_LOOP_T_F32 *	pCtrl	input, output	Pointer to the structure with AMCLIB_CurrentLoop state.

**Implementation details**

The following block diagram shows the internal functions and data structures of AMCLIB\_CurrentLoop\_F32.

**Figure 38. Functions and data structures in AMCLIB\_CurrentLoop\_F32**

Prior to calculating the controller coefficients, it is necessary to set the scaling constants. All inputs and outputs of the algorithm are limited to the fractional range [-1, 1]. Incorrect setting of the scaling constants may lead to undesirable overflow or saturation during the computation.

**Table 15. Scaling constants**

Scaling constant	Symbol	Calculation
Maximum stator phase voltage [V]	$U_{MAX}$	$U_{MAX} = U_{DC\_Bus\_Max}$
Maximum phase current [A]	$I_{MAX}$	Maximum current of the inverter or nominal current of the motor (whichever is lower).

Parameters of the PIrAW controllers (using bilinear transform) can be calculated using the following equations:

$$\begin{aligned}
 pPIrAWD.f32CC1sc &= \text{FRAC32}\left(\left(K_{pD} + \frac{K_{iD}T_S}{2}\right) \cdot \frac{I_{MAX}}{U_{MAX}} \cdot 2^{NShiftD}\right) \\
 pPIrAWD.f32CC2sc &= \text{FRAC32}\left(\left(-K_{pD} + \frac{K_{iD}T_S}{2}\right) \cdot \frac{I_{MAX}}{U_{MAX}} \cdot 2^{NShiftD}\right) \\
 pPIrAWDu16NShift &= NShiftD \\
 pPIrAWQ.f32CC1sc &= \text{FRAC32}\left(\left(K_{pQ} + \frac{K_{iQ}T_S}{2}\right) \cdot \frac{I_{MAX}}{U_{MAX}} \cdot 2^{NShiftQ}\right) \\
 pPIrAWQ.f32CC2sc &= \text{FRAC32}\left(\left(-K_{pQ} + \frac{K_{iQ}T_S}{2}\right) \cdot \frac{I_{MAX}}{U_{MAX}} \cdot 2^{NShiftQ}\right) \\
 pPIrAWQu16NShift &= NShiftQ
 \end{aligned}$$

Equation AMCLIB\_CurrentLoop\_F32\_Eq1

where  $T_S$  is the sampling period,  $K_{pD}$  is the proportional gain in the d-axis,  $K_{iD}$  is the integral gain in the d-axis,  $K_{pQ}$  is the proportional gain in the q-axis, and  $K_{iQ}$  is the integral gain in the q-axis.  $NShiftD$  and  $NShiftQ$  are the smallest nonnegative integer values which ensure that the controller coefficients in the d and q axes fit in the fractional range [-1, 1). The gains can be calculated as follows:

$$\begin{aligned}
 K_{pD} &= 2 \cdot \xi \cdot \omega_0 \cdot L_D - R_S \\
 K_{iD} &= \omega_0^2 \cdot L_D \\
 K_{pQ} &= 2 \cdot \xi \cdot \omega_0 \cdot L_Q - R_S \\
 K_{iQ} &= \omega_0^2 \cdot L_Q
 \end{aligned}$$

Equation AMCLIB\_CurrentLoop\_F32\_Eq2

where  $\xi$  is the current loop attenuation, and  $\omega_0$  is the current loop natural frequency [rad/s],  $R_S$  is the phase resistance [ohm],  $L_D$  and  $L_Q$  are phase inductances in the d and q axes, respectively.

The specified accuracy of results is guaranteed only in cases when  $pCtrl->pPIrAWQ.f32UpperLimit > \text{FRAC32}(0.0000305176)$ .

**Note:** Due to effectivity reasons, this function is implemented using inline assembly and is therefore not ANSI-C compliant.

### Code Example

```

#include "amclib.h"

AMCLIB_CURRENT_LOOP_T_F32 CurrentLoop;
SWLIBS_2Syst_F32 IDQReq;           // required dq currents
SWLIBS_2Syst_F32 IDQFbck;         // feedback dq currents
SWLIBS_2Syst_F32 UDQReq;          // required dq voltages
tFrac32 f32VelocityReq;           // required velocity
tFrac32 f32VelocityFbck;          // actual velocity
tFrac32 f32UDcBus;               // DC bus voltage

void main (void)
{
    tFrac32 f32ControllerPIrAWDOut;
    tFrac32 f32ControllerPIrAWQOut;

    // Initialize the parameters and pointers in CurrentLoop
    CurrentLoop.pPIrAWD.f32CC1sc     = (tFrac32)FRAC32(0.1345);
    CurrentLoop.pPIrAWD.f32CC2sc     = (tFrac32)FRAC32(0.3498);
    CurrentLoop.pPIrAWD.u16Nshift    = 1u;
}

```

```
CurrentLoop.pPIrAWQ.f32CC1sc      = (tFrac32) FRAC32(0.6432);  
CurrentLoop.pPIrAWQ.f32CC2sc      = (tFrac32) FRAC32(0.2735);  
CurrentLoop.pPIrAWQ.ul6NShift     = 1u;  
CurrentLoop.pIDQReq = &IDQReq;  
CurrentLoop.pIDQFbck = &IDQFbck;  
  
// Clear AMCLIB_CurrentLoop state variables  
// Alternative 1: API call with postfix  
// (only one alternative shall be used).  
AMCLIB\_CurrentLoopInit\_F32(&CurrentLoop);  
  
// Alternative 2: API call with implementation parameter  
// (only one alternative shall be used).  
AMCLIB\_CurrentLoopInit(&CurrentLoop, F32);  
  
// Alternative 3: API call with global configuration of implementation  
// (only one alternative shall be used). This alternative is available  
// only if 32-bit fractional implementation is selected as default.  
AMCLIB\_CurrentLoopInit(&CurrentLoop);  
  
// Initialize the AMCLIB_CurrentLoop state variables to predefined values  
// Warning: Parameters in CurrentLoop must be already initialized.  
f32ControllerPIrAWDOut = (tFrac32)123L;  
f32ControllerPIrAWQOut = (tFrac32)123L;  
// Alternative 1: API call with postfix  
// (only one alternative shall be used).  
AMCLIB\_CurrentLoopSetState\_F32(f32ControllerPIrAWDOut,  
    f32ControllerPIrAWQOut, &CurrentLoop);  
  
// Alternative 2: API call with implementation parameter  
// (only one alternative shall be used).  
AMCLIB\_CurrentLoopSetState(f32ControllerPIrAWDOut,  
    f32ControllerPIrAWQOut, &CurrentLoop, F32);  
  
// Alternative 3: API call with global configuration of implementation  
// (only one alternative shall be used). This alternative is available  
// only if 32-bit fractional implementation is selected as default.  
AMCLIB\_CurrentLoopSetState(f32ControllerPIrAWDOut,  
    f32ControllerPIrAWQOut, &CurrentLoop);  
  
f32VelocityReq = (tFrac32)100L;  
  
while(1);  
}  
  
// Periodical function or interrupt - current control loop  
void FastLoop(void)  
{  
    // Calculate new values of IDQFbck, f32VelocityFbck, measure f32UDcBus  
    // (...)  
  
    // Alternative 1: API call with postfix  
    // (only one alternative shall be used).  
AMCLIB\_CurrentLoop\_F32(f32UDcBus, &UDQReq, &CurrentLoop);  
  
    // Alternative 2: API call with implementation parameter  
    // (only one alternative shall be used).  
AMCLIB\_CurrentLoop(f32UDcBus, &UDQReq, &CurrentLoop, F32);  
  
    // Alternative 3: API call with global configuration of implementation
```

```

// (only one alternative shall be used). This alternative is available
// only if 32-bit fractional implementation is selected as default.
AMCLIB_CurrentLoop(f32UDcBus, &UDQReq, &CurrentLoop);

// Calculate new PWM values from UDQReq
// (...)

}

// Periodical function or interrupt - speed control loop
void SlowLoop(void)
{
    // Calculate new values of IDQReq
    // (...)

}

```

### 2.3.2 Function AMCLIB\_CurrentLoop\_F16

#### Declaration

```
void AMCLIB_CurrentLoop_F16(tFrac16 f16UDcBus, SWLIBS\_2Syst\_F16  
*const pUDQReq, AMCLIB\_CURRENT\_LOOP\_T\_F16 *pCtrl);
```

#### Arguments

Table 16. AMCLIB\_CurrentLoop\_F16 arguments

Type	Name	Direction	Description
<a href="#">tFrac16</a>	f16UDcBus	<a href="#">input</a>	DC bus voltage.
<a href="#">SWLIBS_2Syst_F16</a> *const	pUDQReq	<a href="#">output</a>	Pointer to the structure with the required stator voltages in the two-phase rotational orthogonal system (d-q).
<a href="#">AMCLIB_CURRENT_LOOP_T_F16</a> *	pCtrl	<a href="#">input, output</a>	Pointer to the structure with AMCLIB_CurrentLoop state.

#### Implementation details

The following block diagram shows the internal functions and data structures of AMCLIB\_CurrentLoop\_F16.

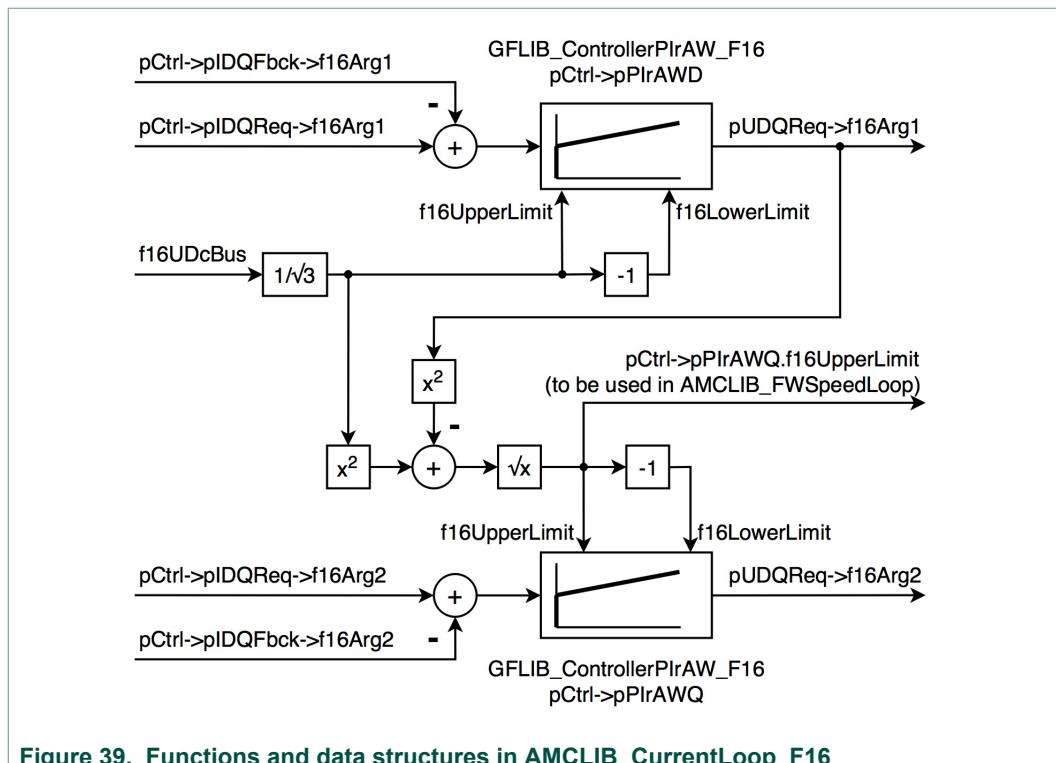


Figure 39. Functions and data structures in AMCLIB\_CurrentLoop\_F16

Prior to calculating the controller coefficients, it is necessary to set the scaling constants. All inputs and outputs of the algorithm are limited to the fractional range [-1, 1). Incorrect setting of the scaling constants may lead to undesirable overflow or saturation during the computation.

Table 17. Scaling constants

Scaling constant	Symbol	Calculation
Maximum stator phase voltage [V]	$U_{MAX}$	$U_{MAX} = U_{DC\_Bus\_Max}$
Maximum phase current [A]	$I_{MAX}$	Maximum current of the inverter or nominal current of the motor (whichever is lower).

Parameters of the PIRAW controllers (using bilinear transform) can be calculated using the following equations:

$$\begin{aligned}
 pPIrAWD.f16CC1sc &= FRAC16\left(\left(K_{pD} + \frac{K_{iD}T_S}{2}\right) \cdot \frac{I_{MAX}}{U_{MAX}} \cdot 2^{NShiftD}\right) \\
 pPIrAWD.f16CC2sc &= FRAC16\left(\left(-K_{pD} + \frac{K_{iD}T_S}{2}\right) \cdot \frac{I_{MAX}}{U_{MAX}} \cdot 2^{NShiftD}\right) \\
 pPIrAWD.u16NShift &= NShiftD \\
 pPIrAWQ.f16CC1sc &= FRAC16\left(\left(K_{pQ} + \frac{K_{iQ}T_S}{2}\right) \cdot \frac{I_{MAX}}{U_{MAX}} \cdot 2^{NShiftQ}\right) \\
 pPIrAWQ.f16CC2sc &= FRAC16\left(\left(-K_{pQ} + \frac{K_{iQ}T_S}{2}\right) \cdot \frac{I_{MAX}}{U_{MAX}} \cdot 2^{NShiftQ}\right) \\
 pPIrAWD.u16NShift &= NShiftQ
 \end{aligned}$$

Equation AMCLIB\_CurrentLoop\_F16\_Eq1

where  $T_S$  is the sampling period,  $K_{pD}$  is the proportional gain in the d-axis,  $K_{iD}$  is the integral gain in the d-axis,  $K_{pQ}$  is the proportional gain in the q-axis, and  $K_{iQ}$  is the integral gain in the q-axis.  $NShiftD$  and  $NShiftQ$  are the smallest nonnegative integer

values which ensure that the controller coefficients in the d and q axes fit in the fractional range [-1, 1). The gains can be calculated as follows:

$$K_{pD} = 2 \cdot \xi \cdot \omega_0 \cdot L_D - R_S$$

$$K_{iD} = \omega_0^2 \cdot L_D$$

$$K_{pQ} = 2 \cdot \xi \cdot \omega_0 \cdot L_Q - R_S$$

$$K_{iQ} = \omega_0^2 \cdot L_Q$$

Equation AMCLIB\_CurrentLoop\_F16\_Eq2

where  $\xi$  is the current loop attenuation, and  $\omega_0$  is the current loop natural frequency [rad/s],  $R_S$  is the phase resistance [ohm],  $L_D$  and  $L_Q$  are phase inductances in the d and q axes, respectively.

**Note:** Due to effectivity reasons, this function is implemented using inline assembly and is therefore not ANSI-C compliant.

### Code Example

```
#include "amclib.h"

AMCLIB_CURRENT_LOOP_T_F16 CurrentLoop;
SWLIBS_2Syst_F16 IDQReq;           // required dq currents
SWLIBS_2Syst_F16 IDQFbck;         // feedback dq currents
SWLIBS_2Syst_F16 UDQReq;          // required dq voltages
tFrac16 f16VelocityReq;           // required velocity
tFrac16 f16VelocityFbck;          // actual velocity
tFrac16 f16UDcBus;               // DC bus voltage

void main (void)
{
    tFrac16 f16ControllerPIrAWDOut;
    tFrac16 f16ControllerPIrAWQOut;

    // Initialize the parameters and pointers in CurrentLoop
    CurrentLoop.pPIrAWD.f16CC1sc      = (tFrac16)FRAC16(0.1345);
    CurrentLoop.pPIrAWD.f16CC2sc      = (tFrac16)FRAC16(0.3498);
    CurrentLoop.pPIrAWD.ul6NShift     = 1u;
    CurrentLoop.pPIrAWQ.f16CC1sc      = (tFrac16)FRAC16(0.6432);
    CurrentLoop.pPIrAWQ.f16CC2sc      = (tFrac16)FRAC16(0.2735);
    CurrentLoop.pPIrAWQ.ul6NShift     = 1u;
    CurrentLoop.pIDQReq = &IDQReq;
    CurrentLoop.pIDQFbck = &IDQFbck;

    // Clear AMCLIB_CurrentLoop state variables
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB_CurrentLoopInit_F16(&CurrentLoop);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB_CurrentLoopInit(&CurrentLoop, F16);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 16-bit fractional implementation is selected as default.
    AMCLIB_CurrentLoopInit(&CurrentLoop);
```

```
// Initialize the AMCLIB_CurrentLoop state variables to predefined values
// Warning: Parameters in CurrentLoop must be already initialized.
f16ControllerPIrAWDOut = (tFrac16)123;
f16ControllerPIrAWQOut = (tFrac16)123;
// Alternative 1: API call with postfix
// (only one alternative shall be used).
AMCLIB_CurrentLoopSetState_F16(f16ControllerPIrAWDOut,
    f16ControllerPIrAWQOut, &CurrentLoop);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
AMCLIB_CurrentLoopSetState(f16ControllerPIrAWDOut,
    f16ControllerPIrAWQOut, &CurrentLoop, F16);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 16-bit fractional implementation is selected as default.
AMCLIB_CurrentLoopSetState(f16ControllerPIrAWDOut,
    f16ControllerPIrAWQOut, &CurrentLoop);

f16VelocityReq = (tFrac16)100;

while(1);
}

// Periodical function or interrupt - current control loop
void FastLoop(void)
{
    // Calculate new values of IDQFbck, f16VelocityFbck, measure f16UDcBus
    // (...)

    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB_CurrentLoop_F16(f16UDcBus, &UDQReq, &CurrentLoop);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB_CurrentLoop(f16UDcBus, &UDQReq, &CurrentLoop, F16);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 16-bit fractional implementation is selected as default.
    AMCLIB_CurrentLoop(f16UDcBus, &UDQReq, &CurrentLoop);

    // Calculate new PWM values from UDQReq
    // (...)

}

// Periodical function or interrupt - speed control loop
void SlowLoop(void)
{
    // Calculate new values of IDQReq
    // (...)

}
```

### 2.3.3 Function AMCLIB\_CurrentLoop\_FLT

#### Declaration

```
void AMCLIB_CurrentLoop_FLT(tFloat fltUDcBus, SWLIBS_2Syst_FLT  
*const pUDQReq, AMCLIB_CURRENT_LOOP_T_FLT *pCtrl);
```

#### Arguments

Table 18. AMCLIB\_CurrentLoop\_FLT arguments

Type	Name	Direction	Description
<b>tFloat</b>	fltUDcBus	input	DC bus voltage.
<b>SWLIBS_2Syst_FLT</b> *const	pUDQReq	output	Pointer to the structure with the required stator voltages in the two-phase rotational orthogonal system (d-q).
<b>AMCLIB_CURRENT_LOOP_T_FLT</b> *	pCtrl	input, output	Pointer to the structure with AMCLIB_CurrentLoop state.

#### Implementation details

The following block diagram shows the internal functions and data structures of AMCLIB\_CurrentLoop\_FLT.

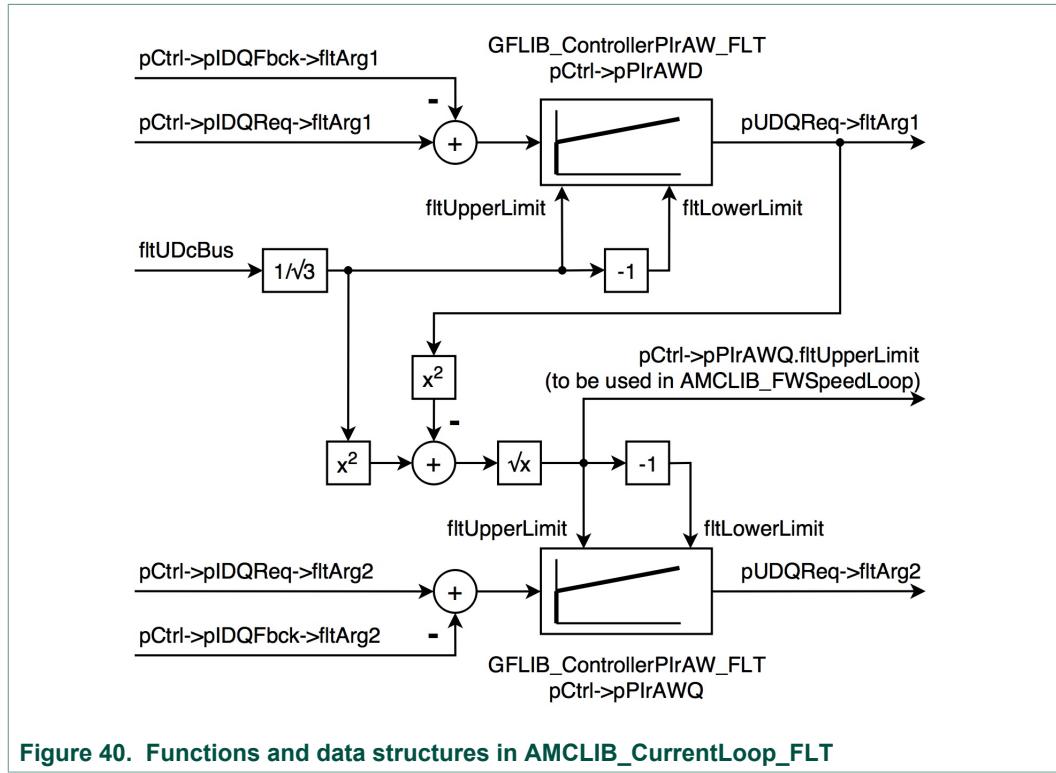


Figure 40. Functions and data structures in AMCLIB\_CurrentLoop\_FLT

Parameters of the PIrAW controllers (using bilinear transform) can be calculated using the following equations:

$$\begin{aligned} pPIrAWD.fltCC1sc &= K_{pD} + \frac{K_{iD}T_S}{2} \\ pPIrAWD.fltCC2sc &= -K_{pD} + \frac{K_{iD}T_S}{2} \\ pPIrAWQ.fltCC1sc &= K_{pQ} + \frac{K_{iQ}T_S}{2} \\ pPIrAWQ.fltCC2sc &= -K_{pQ} + \frac{K_{iQ}T_S}{2} \end{aligned}$$

Equation AMCLIB\_CurrentLoop\_FLT\_Eq1

where  $T_S$  is the sampling period,  $K_{pD}$  is the proportional gain in the d-axis,  $K_{iD}$  is the integral gain in the d-axis,  $K_{pQ}$  is the proportional gain in the q-axis, and  $K_{iQ}$  is the integral gain in the q-axis. The gains can be calculated as follows:

$$\begin{aligned} K_{pD} &= 2 \cdot \xi \cdot \omega_0 \cdot L_D - R_S \\ K_{iD} &= \omega_0^2 \cdot L_D \\ K_{pQ} &= 2 \cdot \xi \cdot \omega_0 \cdot L_Q - R_S \\ K_{iQ} &= \omega_0^2 \cdot L_Q \end{aligned}$$

Equation AMCLIB\_CurrentLoop\_FLT\_Eq2

where  $\xi$  is the current loop attenuation, and  $\omega_0$  is the current loop natural frequency [rad/s],  $R_S$  is the phase resistance [ohm],  $L_D$  and  $L_Q$  are phase inductances in the d and q axes, respectively.

**Note:** The function may raise floating-point exceptions (invalid operation, overflow, underflow, inexact, input denormal). The floating-point unit must be enabled in CPACR register to prevent NOCP UsageFault exception.

### Code Example

```
#include "amclib.h"

AMCLIB_CURRENT_LOOP_T_FLT CurrentLoop;
SWLIBS_2Syst_FLT IDQReq;           // required dq currents
SWLIBS_2Syst_FLT IDQFbck;         // feedback dq currents
SWLIBS_2Syst_FLT UDQReq;          // required dq voltages
tFloat fltVelocityReq;            // required velocity
tFloat fltVelocityFbck;           // actual velocity
tFloat fltUDCBus;                // DC bus voltage

void main (void)
{
    tFloat fltControllerPIrAWDOut;
    tFloat fltControllerPIrAWQOut;

    // Initialize the parameters and pointers in CurrentLoop
    CurrentLoop.pPIrAWD.fltCC1sc = (tFloat)0.1345;
    CurrentLoop.pPIrAWD.fltCC2sc = (tFloat)0.3498;
    CurrentLoop.pPIrAWQ.fltCC1sc = (tFloat)0.6432;
    CurrentLoop.pPIrAWQ.fltCC2sc = (tFloat)0.2735;
    CurrentLoop.pIDQReq = &IDQReq;
    CurrentLoop.pIDQFbck = &IDQFbck;

    // Clear AMCLIB_CurrentLoop state variables
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
}
```

```
AMCLIB_CurrentLoopInit_FLT(&CurrentLoop);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
AMCLIB_CurrentLoopInit(&CurrentLoop, FLT);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if floating point implementation is selected as default.
AMCLIB_CurrentLoopInit(&CurrentLoop);

// Initialize the AMCLIB_CurrentLoop state variables to predefined values
// Warning: Parameters in CurrentLoop must be already initialized.
fltControllerPIrAWDOut = 123.0F;
fltControllerPIrAWQOut = 123.0F;
// Alternative 1: API call with postfix
// (only one alternative shall be used).
AMCLIB_CurrentLoopSetState_FLT(fltControllerPIrAWDOut,
    fltControllerPIrAWQOut, &CurrentLoop);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
AMCLIB_CurrentLoopSetState(fltControllerPIrAWDOut,
    fltControllerPIrAWQOut, &CurrentLoop, FLT);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if floating point implementation is selected as default.
AMCLIB_CurrentLoopSetState(fltControllerPIrAWDOut,
    fltControllerPIrAWQOut, &CurrentLoop);

fltVelocityReq = 123.0F;
while(1);
}

// Periodical function or interrupt - current control loop
void FastLoop(void)
{
    // Calculate new values of IDQFbck, fltVelocityFbck, measure fltUDcBus
    // (...)

    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB_CurrentLoop_FLT(fltUDcBus, &UDQReq, &CurrentLoop);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB_CurrentLoop(fltUDcBus, &UDQReq, &CurrentLoop, FLT);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if floating point implementation is selected as default.
    AMCLIB_CurrentLoop(fltUDcBus, &UDQReq, &CurrentLoop);

    // Calculate new PWM values from UDQReq
    // (...)

}

// Periodical function or interrupt - speed control loop
void SlowLoop(void)
```

```
{
    // Calculate new values of IDQReq
    // (...)

}
```

### 2.3.4 Function AMCLIB\_CurrentLoopInit

#### Description

This function clears the AMCLIB\_CurrentLoop state variables.

**Note:** The input/output pointers must contain valid addresses, otherwise a fault may occur (MemManage, BusFault, UsageFault, HardFault).

#### Re-entrancy

The function is re-entrant for a different pCtrl.

#### 2.3.4.1 Function AMCLIB\_CurrentLoopInit\_F32

##### Declaration

```
void AMCLIB_CurrentLoopInit_F32 (AMCLIB_CURRENT_LOOP_T_F32 *const pCtrl);
```

##### Arguments

Table 19. AMCLIB\_CurrentLoopInit\_F32 arguments

Type	Name	Direction	Description
AMCLIB_CURRENT_LOOP_T_F32 *const	pCtrl	input, output	Pointer to the structure with AMCLIB_CurrentLoop state.

#### Code Example

```
#include "amclib.h"

AMCLIB_CURRENT_LOOP_T_F32 CurrentLoop;
SWLIBS_2Syst_F32 IDQReq;           // required dq currents
SWLIBS_2Syst_F32 IDQFbck;         // feedback dq currents
SWLIBS_2Syst_F32 UDQReq;          // required dq voltages
tFrac32 f32VelocityReq;           // required velocity
tFrac32 f32VelocityFbck;          // actual velocity
tFrac32 f32UDcBus;               // DC bus voltage

void main (void)
{
    tFrac32 f32ControllerPIrAWDOut;
    tFrac32 f32ControllerPIrAWQOut;

    // Initialize the parameters and pointers in CurrentLoop
    CurrentLoop.pPIrAWD.f32CC1sc      = (tFrac32)FRAC32(0.1345);
    CurrentLoop.pPIrAWD.f32CC2sc      = (tFrac32)FRAC32(0.3498);
    CurrentLoop.pPIrAWD.u16NShift     = 1u;
    CurrentLoop.pPIrAWQ.f32CC1sc      = (tFrac32)FRAC32(0.6432);
    CurrentLoop.pPIrAWQ.f32CC2sc      = (tFrac32)FRAC32(0.2735);
```

```
CurrentLoop.pPIrAWQ.ul6NShift      = 1u;
CurrentLoop.pIDQReq = &IDQReq;
CurrentLoop.pIDQFbck = &IDQFbck;

// Clear AMCLIB_CurrentLoop state variables
// Alternative 1: API call with postfix
// (only one alternative shall be used).
AMCLIB_CurrentLoopInit_F32(&CurrentLoop);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
AMCLIB_CurrentLoopInit(&CurrentLoop, F32);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 32-bit fractional implementation is selected as default.
AMCLIB_CurrentLoopInit(&CurrentLoop);

// Initialize the AMCLIB_CurrentLoop state variables to predefined values
// Warning: Parameters in CurrentLoop must be already initialized.
f32ControllerPIrAWDOut = (tFrac32)123L;
f32ControllerPIrAWQOut = (tFrac32)123L;
// Alternative 1: API call with postfix
// (only one alternative shall be used).
AMCLIB\_CurrentLoopSetState\_F32(f32ControllerPIrAWDOut,
    f32ControllerPIrAWQOut, &CurrentLoop);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
AMCLIB_CurrentLoopSetState(f32ControllerPIrAWDOut,
    f32ControllerPIrAWQOut, &CurrentLoop, F32);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 32-bit fractional implementation is selected as default.
AMCLIB_CurrentLoopSetState(f32ControllerPIrAWDOut,
    f32ControllerPIrAWQOut, &CurrentLoop);

f32VelocityReq = (tFrac32)100L;

while(1);
}

// Periodical function or interrupt - current control loop
void FastLoop(void)
{
    // Calculate new values of IDQFbck, f32VelocityFbck, measure f32UDcBus
    // (...)

    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB\_CurrentLoop\_F32(f32UDcBus, &UDQReq, &CurrentLoop);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB_CurrentLoop(f32UDcBus, &UDQReq, &CurrentLoop, F32);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 32-bit fractional implementation is selected as default.
```

```

AMCLIB_CurrentLoop(f32UDcBus, &UDQReq, &CurrentLoop);

// Calculate new PWM values from UDQReq
// (...)

}

// Periodical function or interrupt - speed control loop
void SlowLoop(void)
{
    // Calculate new values of IDQReq
    // (...)

}

```

### 2.3.4.2 Function AMCLIB\_CurrentLoopInit\_F16

#### Declaration

```
void AMCLIB_CurrentLoopInit_F16(AMCLIB\_CURRENT\_LOOP\_T\_F16 *const pCtrl);
```

#### Arguments

**Table 20.** AMCLIB\_CurrentLoopInit\_F16 arguments

Type	Name	Direction	Description
<a href="#">AMCLIB_CURRENT_LOOP_T_F16</a> *const	pCtrl	input, output	Pointer to the structure with AMCLIB_CurrentLoop state.

#### Code Example

```

#include "amclib.h"

AMCLIB\_CURRENT\_LOOP\_T\_F16 CurrentLoop;
SWLIBS\_2Syst\_F16 IDQReq;           // required dq currents
SWLIBS\_2Syst\_F16 IDQFbck;        // feedback dq currents
SWLIBS\_2Syst\_F16 UDQReq;         // required dq voltages
tFrac16 f16VelocityReq;       // required velocity
tFrac16 f16VelocityFbck;      // actual velocity
tFrac16 f16UDcBus;           // DC bus voltage

void main (void)
{
    tFrac16 f16ControllerPIrAWDOut;
    tFrac16 f16ControllerPIrAWQOut;

    // Initialize the parameters and pointers in CurrentLoop
    CurrentLoop.pPIrAWD.f16CC1sc     = (tFrac16)FRAC16(0.1345);
    CurrentLoop.pPIrAWD.f16CC2sc     = (tFrac16)FRAC16(0.3498);
    CurrentLoop.pPIrAWD.u16NShift    = 1u;
    CurrentLoop.pPIrAWQ.f16CC1sc     = (tFrac16)FRAC16(0.6432);
    CurrentLoop.pPIrAWQ.f16CC2sc     = (tFrac16)FRAC16(0.2735);
    CurrentLoop.pPIrAWQ.u16NShift    = 1u;
    CurrentLoop.pIDQReq = &IDQReq;
    CurrentLoop.pIDQFbck = &IDQFbck;

    // Clear AMCLIB_CurrentLoop state variables
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
}

```

```
AMCLIB_CurrentLoopInit_F16(&CurrentLoop);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
AMCLIB_CurrentLoopInit(&CurrentLoop, F16);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 16-bit fractional implementation is selected as default.
AMCLIB_CurrentLoopInit(&CurrentLoop);

// Initialize the AMCLIB_CurrentLoop state variables to predefined values
// Warning: Parameters in CurrentLoop must be already initialized.
f16ControllerPIrAWDOut = (tFrac16)123;
f16ControllerPIrAWQOut = (tFrac16)123;
// Alternative 1: API call with postfix
// (only one alternative shall be used).
AMCLIB\_CurrentLoopSetState\_F16(f16ControllerPIrAWDOut,
    f16ControllerPIrAWQOut, &CurrentLoop);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
AMCLIB_CurrentLoopSetState(f16ControllerPIrAWDOut,
    f16ControllerPIrAWQOut, &CurrentLoop, F16);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 16-bit fractional implementation is selected as default.
AMCLIB_CurrentLoopSetState(f16ControllerPIrAWDOut,
    f16ControllerPIrAWQOut, &CurrentLoop);

f16VelocityReq = (tFrac16)100;

while(1);
}

// Periodical function or interrupt - current control loop
void FastLoop(void)
{
    // Calculate new values of IDQFbck, f16VelocityFbck, measure f16UDcBus
    // (...)

    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB\_CurrentLoop\_F16(f16UDcBus, &UDQReq, &CurrentLoop);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB_CurrentLoop(f16UDcBus, &UDQReq, &CurrentLoop, F16);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 16-bit fractional implementation is selected as default.
    AMCLIB_CurrentLoop(f16UDcBus, &UDQReq, &CurrentLoop);

    // Calculate new PWM values from UDQReq
    // (...)

}

// Periodical function or interrupt - speed control loop
```

```
void SlowLoop(void)
{
    // Calculate new values of IDQReq
    // (...)

}
```

### 2.3.4.3 Function AMCLIB\_CurrentLoopInit\_FLT

#### Declaration

```
void AMCLIB_CurrentLoopInit_FLT(AMCLIB\_CURRENT\_LOOP\_T\_FLT *const pCtrl);
```

#### Arguments

**Table 21.** AMCLIB\_CurrentLoopInit\_FLT arguments

Type	Name	Direction	Description
<a href="#">AMCLIB_CURRENT_LOOP_T_FLT</a> *const	pCtrl	input, output	Pointer to the structure with AMCLIB_CurrentLoop state.

#### Code Example

```
#include "amclib.h"

AMCLIB\_CURRENT\_LOOP\_T\_FLT CurrentLoop;
SWLIBS\_2Syst\_FLT IDQReq;           // required dq currents
SWLIBS\_2Syst\_FLT IDQFbck;        // feedback dq currents
SWLIBS\_2Syst\_FLT UDQReq;          // required dq voltages
tFloat fltVelocityReq;         // required velocity
tFloat fltVelocityFbck;        // actual velocity
tFloat fltUDCBus;             // DC bus voltage

void main (void)
{
    tFloat fltControllerPIrAWDOut;
    tFloat fltControllerPIrAWQOut;

    // Initialize the parameters and pointers in CurrentLoop
    CurrentLoop.pPIrAWD.fltCC1sc      = (tFloat)0.1345;
    CurrentLoop.pPIrAWD.fltCC2sc      = (tFloat)0.3498;
    CurrentLoop.pPIrAWQ.fltCC1sc      = (tFloat)0.6432;
    CurrentLoop.pPIrAWQ.fltCC2sc      = (tFloat)0.2735;
    CurrentLoop.pIDQReq = &IDQReq;
    CurrentLoop.pIDQFbck = &IDQFbck;

    // Clear AMCLIB_CurrentLoop state variables
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB_CurrentLoopInit_FLT(&CurrentLoop);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB_CurrentLoopInit(&CurrentLoop, FLT);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if floating point implementation is selected as default.
```

```
AMCLIB_CurrentLoopInit(&CurrentLoop);

// Initialize the AMCLIB_CurrentLoop state variables to predefined values
// Warning: Parameters in CurrentLoop must be already initialized.
fltControllerPIrAWDOut = 123.0F;
fltControllerPIrAWQOut = 123.0F;
// Alternative 1: API call with postfix
// (only one alternative shall be used).
AMCLIB_CurrentLoopSetState_FLT(fltControllerPIrAWDOut,
    fltControllerPIrAWQOut, &CurrentLoop);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
AMCLIB_CurrentLoopSetState(fltControllerPIrAWDOut,
    fltControllerPIrAWQOut, &CurrentLoop, FLT);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if floating point implementation is selected as default.
AMCLIB_CurrentLoopSetState(fltControllerPIrAWDOut,
    fltControllerPIrAWQOut, &CurrentLoop);

    fltVelocityReq = 123.0F;
    while(1);
}

// Periodical function or interrupt - current control loop
void FastLoop(void)
{
    // Calculate new values of IDQFbck, fltVelocityFbck, measure fltUDcBus
    // (...)

    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB_CurrentLoop_FLT(fltUDcBus, &UDQReq, &CurrentLoop);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB_CurrentLoop(fltUDcBus, &UDQReq, &CurrentLoop, FLT);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if floating point implementation is selected as default.
    AMCLIB_CurrentLoop(fltUDcBus, &UDQReq, &CurrentLoop);

    // Calculate new PWM values from UDQReq
    // (...)

}

// Periodical function or interrupt - speed control loop
void SlowLoop(void)
{
    // Calculate new values of IDQReq
    // (...)

}
```

### 2.3.5 Function AMCLIB\_CurrentLoopSetState

**Description**

This function initializes the AMCLIB\_CurrentLoop state variables to achieve the required output values.

**Note:** The input/output pointers must contain valid addresses, otherwise a fault may occur (MemManage, BusFault, UsageFault, HardFault).

**Re-entrancy**

The function is re-entrant for a different pCtrl.

**2.3.5.1 Function AMCLIB\_CurrentLoopSetState\_F32****Declaration**

```
void AMCLIB_CurrentLoopSetState_F32(tFrac32
f32ControllerPIrAWDOut, tFrac32 f32ControllerPIrAWQOut,
AMCLIB\_CURRENT\_LOOP\_T\_F32 *pCtrl);
```

**Arguments****Table 22. AMCLIB\_CurrentLoopSetState\_F32 arguments**

Type	Name	Direction	Description
<a href="#">tFrac32</a>	f32ControllerPIrAWDOut <input/>		Required output of the d-axis ControllerPIrAW.
<a href="#">tFrac32</a>	f32ControllerPIrAWQOut <input/>		Required output of the q-axis ControllerPIrAW.
<a href="#">AMCLIB_CURRENT_LOOP_T_F32</a> *	pCtrl	input, output	Pointer to the structure with AMCLIB_CurrentLoop state.

**Caution:** Set the parameters in the structure pointed to by pCtrl before calling this function.

**Code Example**

```
#include "amclib.h"

AMCLIB_CURRENT_LOOP_T_F32 CurrentLoop;
SWLIBS\_2Syst\_F32 IDQReq;           // required dq currents
SWLIBS\_2Syst\_F32 IDQFbck;         // feedback dq currents
SWLIBS\_2Syst\_F32 UDQReq;          // required dq voltages
tFrac32 f32VelocityReq;        // required velocity
tFrac32 f32VelocityFbck;       // actual velocity
tFrac32 f32UDcBus;            // DC bus voltage

void main (void)
{
    tFrac32 f32ControllerPIrAWDOut;
    tFrac32 f32ControllerPIrAWQOut;

    // Initialize the parameters and pointers in CurrentLoop
    CurrentLoop.pPIrAWD.f32CC1sc      = (tFrac32) FRAC32(0.1345);
    CurrentLoop.pPIrAWD.f32CC2sc      = (tFrac32) FRAC32(0.3498);
    CurrentLoop.pPIrAWD.u16NShift     = 1u;
    CurrentLoop.pPIrAWQ.f32CC1sc      = (tFrac32) FRAC32(0.6432);
    CurrentLoop.pPIrAWQ.f32CC2sc      = (tFrac32) FRAC32(0.2735);
    CurrentLoop.pPIrAWQ.u16NShift     = 1u;
```

```
CurrentLoop.pIDQReq = &IDQReq;
CurrentLoop.pIDQFbck = &IDQFbck;

// Clear AMCLIB_CurrentLoop state variables
// Alternative 1: API call with postfix
// (only one alternative shall be used).
AMCLIB_CurrentLoopInit_F32(&CurrentLoop);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
AMCLIB_CurrentLoopInit(&CurrentLoop, F32);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 32-bit fractional implementation is selected as default.
AMCLIB_CurrentLoopInit(&CurrentLoop);

// Initialize the AMCLIB_CurrentLoop state variables to predefined values
// Warning: Parameters in CurrentLoop must be already initialized.
f32ControllerPIrAWDOut = (tFrac32)123L;
f32ControllerPIrAWQOut = (tFrac32)123L;
// Alternative 1: API call with postfix
// (only one alternative shall be used).
AMCLIB_CurrentLoopSetState_F32(f32ControllerPIrAWDOut,
    f32ControllerPIrAWQOut, &CurrentLoop);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
AMCLIB_CurrentLoopSetState(f32ControllerPIrAWDOut,
    f32ControllerPIrAWQOut, &CurrentLoop, F32);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 32-bit fractional implementation is selected as default.
AMCLIB_CurrentLoopSetState(f32ControllerPIrAWDOut,
    f32ControllerPIrAWQOut, &CurrentLoop);

f32VelocityReq = (tFrac32)100L;

while(1);
}

// Periodical function or interrupt - current control loop
void FastLoop(void)
{
    // Calculate new values of IDQFbck, f32VelocityFbck, measure f32UDcBus
    // (...)

    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB_CurrentLoop_F32(f32UDcBus, &UDQReq, &CurrentLoop);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB_CurrentLoop(f32UDcBus, &UDQReq, &CurrentLoop, F32);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 32-bit fractional implementation is selected as default.
    AMCLIB_CurrentLoop(f32UDcBus, &UDQReq, &CurrentLoop);
```

```

    // Calculate new PWM values from UDQReq
    // (...)

}

// Periodical function or interrupt - speed control loop
void SlowLoop(void)
{
    // Calculate new values of IDQReq
    // (...)

}

```

### 2.3.5.2 Function AMCLIB\_CurrentLoopSetState\_F16

#### Declaration

```

void AMCLIB_CurrentLoopSetState_F16(tFrac16
    f16ControllerPIrAWDOut, tFrac16 f16ControllerPIrAWQOut,
    AMCLIB_CURRENT_LOOP_T_F16 *pCtrl);

```

#### Arguments

**Table 23. AMCLIB\_CurrentLoopSetState\_F16 arguments**

Type	Name	Direction	Description
<b>tFrac16</b>	f16ControllerPIrAWDOut	input	Required output of the d-axis ControllerPIrAW.
<b>tFrac16</b>	f16ControllerPIrAWQOut	input	Required output of the q-axis ControllerPIrAW.
<b>AMCLIB_CURRENT_LOOP_T_F16</b> *	pCtrl	input, output	Pointer to the structure with AMCLIB_CurrentLoop state.

**Caution:** Set the parameters in the structure pointed to by pCtrl before calling this function.

#### Code Example

```

#include "amclib.h"

AMCLIB_CURRENT_LOOP_T_F16 CurrentLoop;
SWLIBS_2Syst_F16 IDQReq;           // required dq currents
SWLIBS_2Syst_F16 IDQFbck;         // feedback dq currents
SWLIBS_2Syst_F16 UDQReq;          // required dq voltages
tFrac16 f16VelocityReq;           // required velocity
tFrac16 f16VelocityFbck;          // actual velocity
tFrac16 f16UDcBus;               // DC bus voltage

void main (void)
{
    tFrac16 f16ControllerPIrAWDOut;
    tFrac16 f16ControllerPIrAWQOut;

    // Initialize the parameters and pointers in CurrentLoop
    CurrentLoop.pPIrAWD.f16CC1sc      = (tFrac16) FRAC16(0.1345);
    CurrentLoop.pPIrAWD.f16CC2sc      = (tFrac16) FRAC16(0.3498);
    CurrentLoop.pPIrAWD.u16NShift     = 1u;
    CurrentLoop.pPIrAWQ.f16CC1sc      = (tFrac16) FRAC16(0.6432);
    CurrentLoop.pPIrAWQ.f16CC2sc      = (tFrac16) FRAC16(0.2735);
    CurrentLoop.pPIrAWQ.u16NShift     = 1u;
}

```

```
CurrentLoop.pIDQReq = &IDQReq;
CurrentLoop.pIDQFbck = &IDQFbck;

// Clear AMCLIB_CurrentLoop state variables
// Alternative 1: API call with postfix
// (only one alternative shall be used).
AMCLIB_CurrentLoopInit_F16(&CurrentLoop);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
AMCLIB_CurrentLoopInit(&CurrentLoop, F16);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 16-bit fractional implementation is selected as default.
AMCLIB_CurrentLoopInit(&CurrentLoop);

// Initialize the AMCLIB_CurrentLoop state variables to predefined values
// Warning: Parameters in CurrentLoop must be already initialized.
f16ControllerPIrAWDOut = (tFrac16)123;
f16ControllerPIrAWQOut = (tFrac16)123;
// Alternative 1: API call with postfix
// (only one alternative shall be used).
AMCLIB_CurrentLoopSetState_F16(f16ControllerPIrAWDOut,
    f16ControllerPIrAWQOut, &CurrentLoop);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
AMCLIB_CurrentLoopSetState(f16ControllerPIrAWDOut,
    f16ControllerPIrAWQOut, &CurrentLoop, F16);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 16-bit fractional implementation is selected as default.
AMCLIB_CurrentLoopSetState(f16ControllerPIrAWDOut,
    f16ControllerPIrAWQOut, &CurrentLoop);

f16VelocityReq = (tFrac16)100;

while(1);
}

// Periodical function or interrupt - current control loop
void FastLoop(void)
{
    // Calculate new values of IDQFbck, f16VelocityFbck, measure f16UDcBus
    // (...)

    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB_CurrentLoop_F16(f16UDcBus, &UDQReq, &CurrentLoop);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB_CurrentLoop(f16UDcBus, &UDQReq, &CurrentLoop, F16);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 16-bit fractional implementation is selected as default.
    AMCLIB_CurrentLoop(f16UDcBus, &UDQReq, &CurrentLoop);
```

```

    // Calculate new PWM values from UDQReq
    // (...)

}

// Periodical function or interrupt - speed control loop
void SlowLoop(void)
{
    // Calculate new values of IDQReq
    // (...)

}

```

### 2.3.5.3 Function AMCLIB\_CurrentLoopSetState\_FLT

#### Declaration

```

void AMCLIB_CurrentLoopSetState_FLT(tFloat
    fltControllerPIrAWDOut, tFloat fltControllerPIrAWQOut,
    AMCLIB_CURRENT_LOOP_T_FLT *pCtrl);

```

#### Arguments

**Table 24. AMCLIB\_CurrentLoopSetState\_FLT arguments**

Type	Name	Direction	Description
<b>tFloat</b>	fltControllerPIrAWDOut	<b>input</b>	Required output of the d-axis ControllerPIrAW.
<b>tFloat</b>	fltControllerPIrAWQOut	<b>input</b>	Required output of the q-axis ControllerPIrAW.
<b>AMCLIB_CURRENT_LOOP_T_FLT</b> *	pCtrl	<b>input, output</b>	Pointer to the structure with AMCLIB_CurrentLoop state.

**Caution:** Set the parameters in the structure pointed to by pCtrl before calling this function.

#### Code Example

```

#include "amclib.h"

AMCLIB_CURRENT_LOOP_T_FLT CurrentLoop;
SWLIBS_2Syst_FLT IDQReq;           // required dq currents
SWLIBS_2Syst_FLT IDQFbck;          // feedback dq currents
SWLIBS_2Syst_FLT UDQReq;           // required dq voltages
tFloat fltVelocityReq;             // required velocity
tFloat fltVelocityFbck;            // actual velocity
tFloat fltUDcBus;                 // DC bus voltage

void main (void)
{
    tFloat fltControllerPIrAWDOut;
    tFloat fltControllerPIrAWQOut;

    // Initialize the parameters and pointers in CurrentLoop
    CurrentLoop.pPIrAWD.fltCC1sc      = (tFloat)0.1345;
    CurrentLoop.pPIrAWD.fltCC2sc      = (tFloat)0.3498;
    CurrentLoop.pPIrAWQ.fltCC1sc      = (tFloat)0.6432;
    CurrentLoop.pPIrAWQ.fltCC2sc      = (tFloat)0.2735;
    CurrentLoop.pIDQReq = &IDQReq;
    CurrentLoop.pIDQFbck = &IDQFbck;
}

```

```
// Clear AMCLIB_CurrentLoop state variables
// Alternative 1: API call with postfix
// (only one alternative shall be used).
AMCLIB_CurrentLoopInit_FLT(&CurrentLoop);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
AMCLIB_CurrentLoopInit(&CurrentLoop, FLT);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if floating point implementation is selected as default.
AMCLIB_CurrentLoopInit(&CurrentLoop);

// Initialize the AMCLIB_CurrentLoop state variables to predefined values
// Warning: Parameters in CurrentLoop must be already initialized.
fltControllerPIrAWDOut = 123.0F;
fltControllerPIrAWQOut = 123.0F;
// Alternative 1: API call with postfix
// (only one alternative shall be used).
AMCLIB_CurrentLoopSetState_FLT(fltControllerPIrAWDOut,
    fltControllerPIrAWQOut, &CurrentLoop);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
AMCLIB_CurrentLoopSetState(fltControllerPIrAWDOut,
    fltControllerPIrAWQOut, &CurrentLoop, FLT);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if floating point implementation is selected as default.
AMCLIB_CurrentLoopSetState(fltControllerPIrAWDOut,
    fltControllerPIrAWQOut, &CurrentLoop);

fltVelocityReq = 123.0F;
while(1);
}

// Periodical function or interrupt - current control loop
void FastLoop(void)
{
    // Calculate new values of IDQFbck, fltVelocityFbck, measure fltUDcBus
    // (...)

    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB_CurrentLoop_FLT(fltUDcBus, &UDQReq, &CurrentLoop);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB_CurrentLoop(fltUDcBus, &UDQReq, &CurrentLoop, FLT);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if floating point implementation is selected as default.
    AMCLIB_CurrentLoop(fltUDcBus, &UDQReq, &CurrentLoop);

    // Calculate new PWM values from UDQReq
    // (...)
```

```

}

// Periodical function or interrupt - speed control loop
void SlowLoop(void)
{
    // Calculate new values of IDQReq
    // (...)

}
  
```

## 2.4 Function AMCLIB\_FW

This library function implements the field-weakening algorithm for permanent magnet synchronous motors.

### Description

Field weakening represents an advanced control approach to run the electric motor beyond a base speed. The back electromotive force (EMF) is proportional to the rotor speed and counteracts the motor supply voltage. If a given speed is to be reached, the terminal voltage must be increased to match the increased stator back-EMF. A sufficient voltage is available from the inverter in the operation up to the base speed. Beyond the base speed, motor voltages  $u_d$  and  $u_q$  are limited and cannot be increased because of the ceiling voltage of a given inverter. As the difference between the induced back-EMF and the supply voltage decreases, the phase current flow is limited, hence the currents  $i_d$  and  $i_q$  cannot be controlled sufficiently. Further increase of speed would eventually result in back-EMF voltage equal to the limited stator voltage, which means a complete loss of current control. The only way to retain the current control even beyond the base speed is to lower the generated back-EMF by weakening the flux that links the stator winding.

Base speed splits the whole speed motor operation into two regions: constant torque and constant power, see [Figure 41](#).

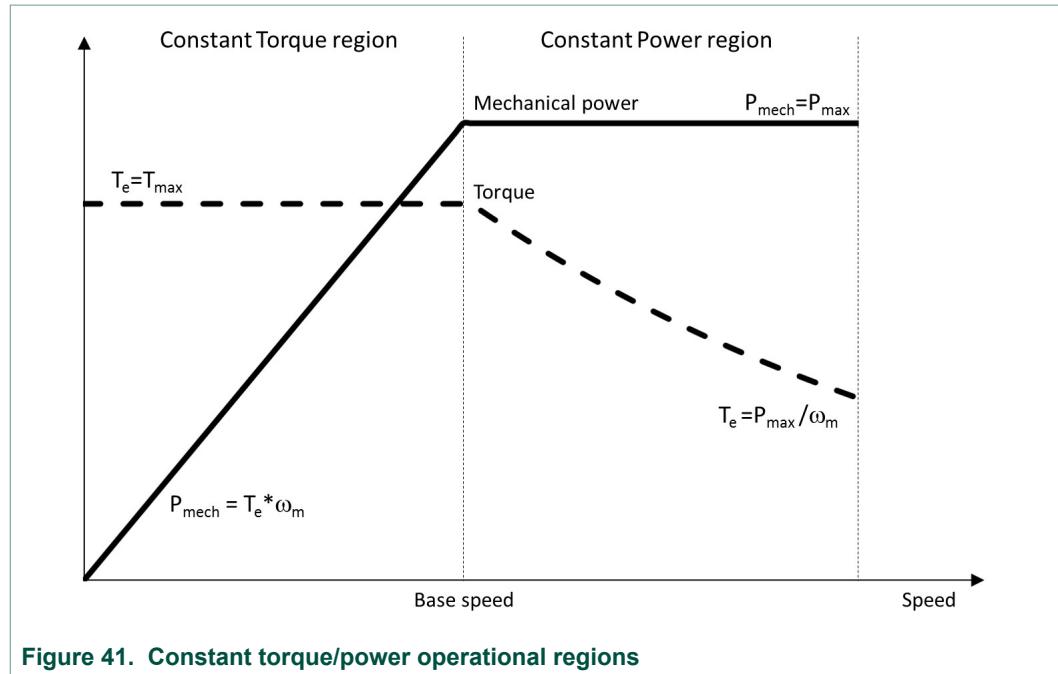
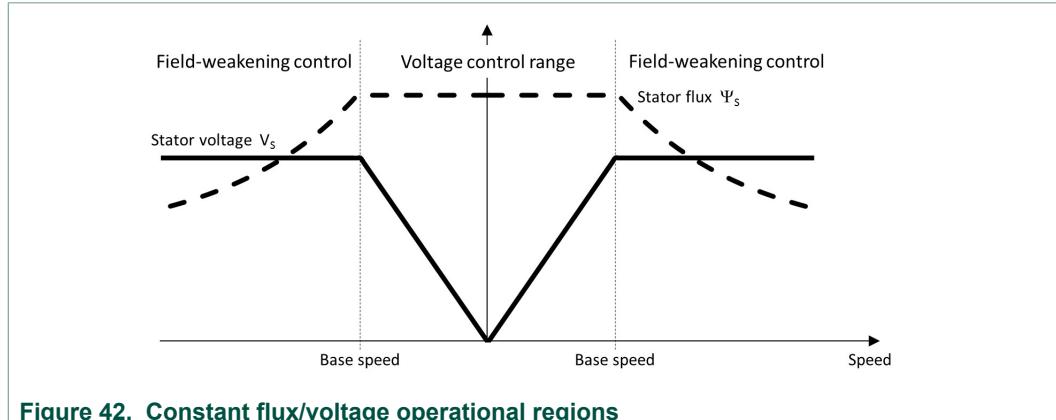


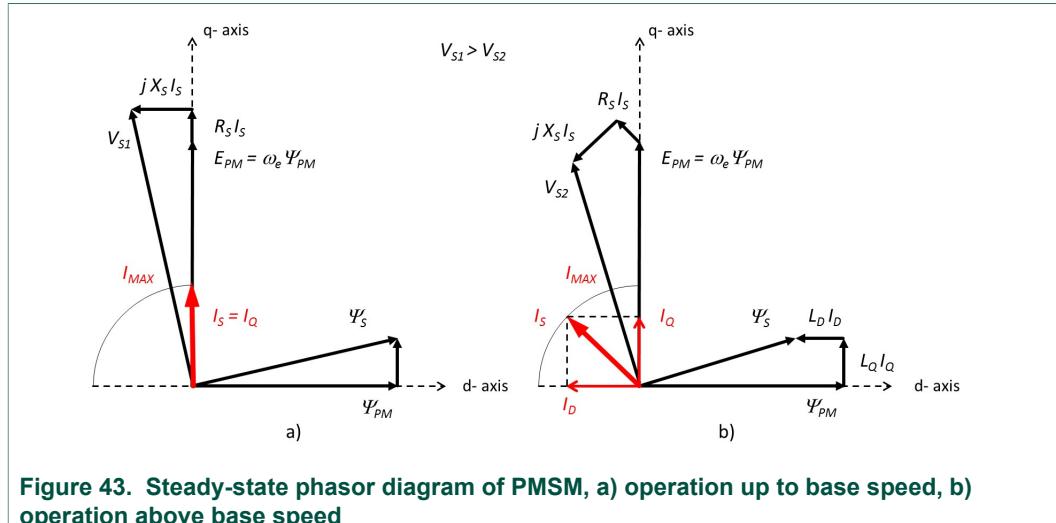
Figure 41. Constant torque/power operational regions

Operation in constant torque region means that maximal torque can be constantly developed while the output power increases with the rotor speed. The phase voltage increases linearly with the speed and the current is controlled to its reference. The operation in constant power region is characterized by a rapid decrease in developed torque while the output power remains constant. The phase voltage is at its limit while the phase current and the stator flux decrease proportionally with the rotor speed, see [Figure 42](#).



**Figure 42. Constant flux/voltage operational regions**

Direct field weakening is possible only in drives with separate excitation. Nevertheless, the same effect can be achieved in drives with permanent magnets using an appropriate stator current control technique as outlined in the following [Figure 43](#).



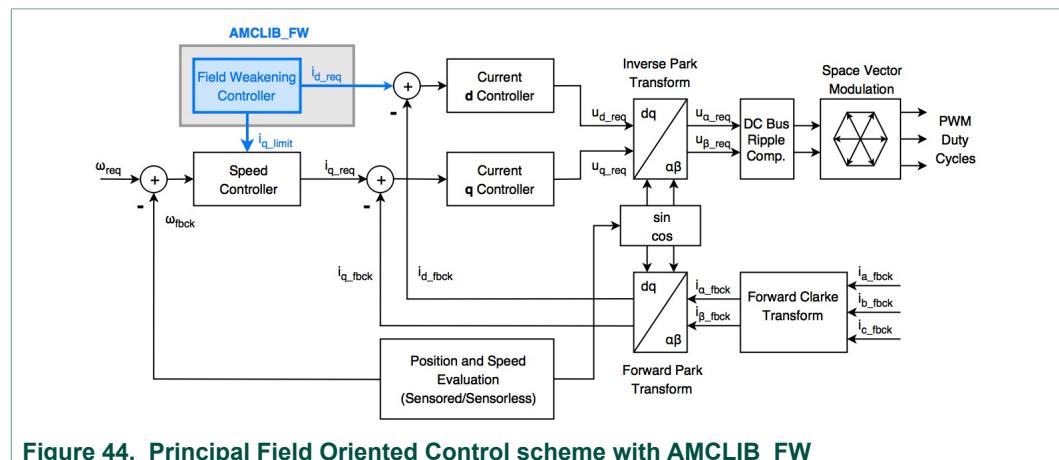
**Figure 43. Steady-state phasor diagram of PMSM, a) operation up to base speed, b) operation above base speed**

The left diagram in [Figure 43](#) depicts the normal operation when the stator current space phasor  $I_S$  is aligned with the q-axis. Consequently, the entire phase current is utilized for torque production, i.e.  $I_Q = I_S$  and  $I_D = 0$ . The rotor speed is  $\omega_e$  and the terminal stator voltage is  $V_{S1}$ .  $R_S$  and  $jX_S$  represent the stator resistance and reactance, respectively,  $j$  is the imaginary unit.

The right diagram in [Figure 43](#) corresponds to the field weakening operation. The displacement of the current space phasor  $I_S$  yields to the production of the nonzero current component  $I_D$  along the negative d-axis of the rotor reference frame. The motor spins at the same speed  $\omega_e$ ; however, the terminal voltage  $V_{S2}$  is now lower than

the previous  $V_{S1}$ . This is achieved thanks to the negative flux component  $L_{DI_D}$  which counteracts the flux of the permanent magnets.

AMCLIB\_FW implements the field weakening controller in the outer control loop highlighted in [Figure 44](#). AMCLIB\_FW is intended to be used in combination with [AMCLIB\\_SpeedLoop](#). The code can be simplified further by utilizing [AMCLIB\\_FWSpeedLoop](#) which combines the speed controller with the field weakening controller in one library function. AMCLIB\_FW does not allow debugging of all internal variables. Use [AMCLIB\\_FWDebug](#) for debugging purposes and replace it with AMCLIB\_FW once the debugging is finished.



**Figure 44. Principal Field Oriented Control scheme with AMCLIB\_FW**

Field weakening technique employed in AMCLIB\_FW extends the speed range of PMSM beyond the base speed value by reducing the linkage magnetic flux. The algorithm implemented in the library brings the following key advantages:

- Fully utilizes the drive capabilities (speed range, load torque).
- Reduces the total linkage flux only when necessary.
- Achieves smooth transition between the normal and field weakening operating speed range, regains full control when recovering from voltage saturation.
- The algorithm is very robust - as a result, the PMSM behaves as a separately excited wound field synchronous motor drive.
- Allows maximum torque optimal control.

This algorithm is protected by US Patent No. US 2011/0050152 A1.

#### Caution:

1. A motor operated at speeds over the base speed region generates higher back-EMF voltage than the supply stator voltage. By applying the field weakening technique, the back-EMF is actively kept under control, i.e. lower than the stator voltage. It is dangerous to break the control loop during the field weakening operation. A loss of control (e.g. due to a fault state or turning the application off while running) may result in damage of the electronics and hardware due to the excessive back-EMF which would no longer be suppressed by the control algorithm.
2. The field is weakened by applying a negative current d-component. Too high negative current can cause magnet damage by its demagnetization. Make sure to set a correct lower limit of the field weakening PI controller to prevent damage to the motor.

**Note:** The input/output pointers must contain valid addresses, otherwise a fault may occur (MemManage, BusFault, UsageFault, HardFault).

## Re-entrancy

The function is re-entrant for a different pCtrl.

### 2.4.1 Function AMCLIB\_FW\_F32

#### Declaration

```
void AMCLIB_FW_F32 (tFrac32 f32IDQReqAmp, tFrac32 f32VelocityFbck,  
SWLIBS_2Syst_F32 *const pIDQReq, AMCLIB_FW_T_F32 *pCtrl);
```

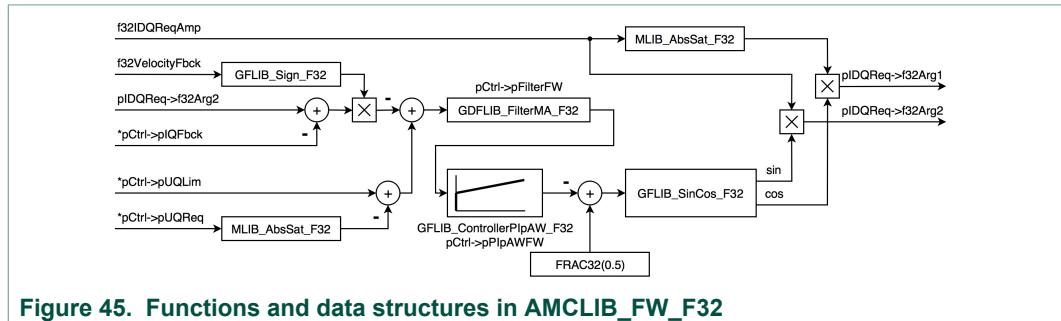
#### Arguments

**Table 25. AMCLIB\_FW\_F32 arguments**

Type	Name	Direction	Description
<u>tFrac32</u>	f32IDQReqAmp	input	Required amplitude of the currents Id and Iq in the two-phase rotational orthogonal system (d-q).
<u>tFrac32</u>	f32VelocityFbck	input	Actual electrical angular velocity from the feedback.
<u>SWLIBS_2Syst_F32</u> *const	pIDQReq	input, output	Pointer to the structure with the required stator currents in the two-phase rotational orthogonal system (d-q).
<u>AMCLIB_FW_T_F32</u> *	pCtrl	input, output	Pointer to the structure with AMCLIB_FW state.

#### Implementation details

The following block diagram shows the internal functions and data structures of AMCLIB\_FW\_F32.



**Figure 45. Functions and data structures in AMCLIB\_FW\_F32**

Prior to calculating the controller coefficients, it is necessary to set the scaling constants. All inputs and outputs of the algorithm are limited to the fractional range [-1, 1]. Incorrect setting of the scaling constants may lead to undesirable overflow or saturation during the computation.

**Table 26. Scaling constants**

Scaling constant	Symbol	Calculation
Maximum stator phase voltage [V]	$U_{MAX}$	$U_{MAX}=U_{DC\_Bus\_Max}$
Maximum phase current [A]	$I_{MAX}$	Maximum current of the inverter or nominal current of the motor (whichever is lower).
Maximum speed [rad/s]	$\Omega_{MAX}$	Maximum application required speed, at least the motor electrical rated speed.

An initial estimate of the field weakening PIpAW controller parameters can be taken from the speed PIpAW controller, see [AMCLIB\\_SpeedLoop\\_F32\\_Eq1](#). The upper limit of the controller output shall be set to zero, i.e. pPIpAWFW.f32UpperLimit = 0. The lower limit shall be set to an adequate value from the interval <FRAC32(-0.5); 0>. It is important to set the lower limit correctly to prevent motor damage by irreversible demagnetization of the permanent magnets. Let  $I_{D\_MAX}$  be the maximum permitted negative d-axis current for field weakening, then

$$pPIpAWFW.f32LowerLimit = \text{FRAC32}\left(\frac{1}{\pi} \cdot \cos^{-1}\left(\frac{I_{D\_MAX}}{I_{MAX}}\right)\right) - 0.5$$

Equation AMCLIB\_FW\_F32\_Eq1

Further refinement of the field weakening PIpAW controller parameters should be done interactively based on the observed performance. This can be easily achieved with the [FREEMASTER](#) real-time debugging tool. The following points should be considered:

- The field weakening PIpAW controller tweaks the angle of the current space vector  $I_S$  to produce the negative flux-producing current component  $I_D$  and sets the limits of the torque-producing current component  $I_Q$ .
- The magnitude of  $I_D$  depends also on the output of the speed PI controller.
- There are two different paths producing the PIpAW input error. The weight of each depends on the motor operation mode.

The performance of the field weakening controller can be affected by measurement noise. The noise immunity can be improved by tweaking the smoothing factor of the moving average filter [GDFLIB\\_FilterMA\\_F32](#) which preprocesses the input to the field weakening controller.

**Note:** Due to effectivity reasons, this function is implemented using inline assembly and is therefore not ANSI-C compliant.

### Code Example

```
#include "amclib.h"

AMCLIB_FW_T_F32 FWState;
SWLIBS_2Syst_F32 IDQReq;           // required dq currents
SWLIBS_2Syst_F32 f32IDQFbck;      // calculated dq currents from the feedback
SWLIBS_2Syst_F32 f32UDQReq;        // required dq voltages
tFrac32 f32VelocityReq;           // required velocity
tFrac32 f32VelocityFbck;          // actual velocity
AMCLIB_CURRENT_LOOP_T_F32 CurrentLoop;

void main (void)
{
    tFrac32 f32FilterMAFWOut;
    tFrac32 f32ControllerPIpAWFWOut;

    // Initialize the parameters and pointers in FWState
    FW.pFilterFW.u16NSamples         = 3u;
    FW.pPIpAWFW.f32PropGain         = (tFrac32)FRAC32(0.2348);
    FW.pPIpAWFW.f32IntegGain        = (tFrac32)FRAC32(0.3457);
    FW.pPIpAWFW.s16PropGainShift    = 1;
    FW.pPIpAWFW.s16IntegGainShift   = 1;
    FW.pPIpAWFW.f32LowerLimit       = (tFrac32)-2012406926L;
    FW.pPIpAWFW.f32UpperLimit       = (tFrac32)2068885746L;
```

```
FW.pIQFbck = &f32IDQFbck.f32Arg2;
FW.pUQReq = &f32UDQReq.f32Arg2;
FW.pUQLim = &CurrentLoop.pPIrAWQ.f32UpperLimit;

// Clear AMCLIB_FW state variables
// Alternative 1: API call with postfix
// (only one alternative shall be used).
AMCLIB_FWInit_F32(&FWState);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
AMCLIB_FWInit(&FWState, F32);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 32-bit fractional implementation is selected as default.
AMCLIB_FWInit(&FWState);

// Initialize the AMCLIB_FW state variables to predefined values
// Warning: Parameters in FWState must be already initialized.
f32FilterMAFWOut = (tFrac32)123L;
f32ControllerPIpAWFWOut = (tFrac32)123L;
// Alternative 1: API call with postfix
// (only one alternative shall be used).
AMCLIB_FWSetState_F32(f32FilterMAFWOut, f32ControllerPIpAWFWOut,
    &FWState);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
AMCLIB_FWSetState(f32FilterMAFWOut, f32ControllerPIpAWFWOut,
    &FWState, F32);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 32-bit fractional implementation is selected as default.
AMCLIB_FWSetState(f32FilterMAFWOut, f32ControllerPIpAWFWOut,
    &FWState);

f32VelocityReq = (tFrac32)100L;
while(1);
}

// Periodical function or interrupt - speed control loop
void SlowLoop(void)
{
    tFrac32 f32IDQReqAmp;

    // Calculate f32IDQReqAmp from f32VelocityReq and f32VelocityFbck
    // using AMCLIB_SpeedLoop
    // (...)

    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB_FW_F32(f32IDQReqAmp, f32VelocityFbck, &IDQReq, &FWState);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB_FW(f32IDQReqAmp, f32VelocityFbck, &IDQReq, &FWState, F32);

    // Alternative 3: API call with global configuration of implementation
```

```

// (only one alternative shall be used). This alternative is available
// only if 32-bit fractional implementation is selected as default.
AMCLIB_FW(f32IDQReqAmp, f32VelocityFbck, &IDQReq, &FWState);
}

// Periodical function or interrupt - current control loop
void FastLoop(void)
{
    // Calculate new values of f32VelocityFbck, f32IDQFbck, f32UDQReq
    // (...)

}

```

## 2.4.2 Function AMCLIB\_FW\_F16

### Declaration

```
void AMCLIB_FW_F16(tFrac16 f16IDQReqAmp, tFrac16 f16VelocityFbck,
SWLIBS\_2Syst\_F16 *const pIDQReq, AMCLIB\_FW\_T\_F16 *pCtrl);
```

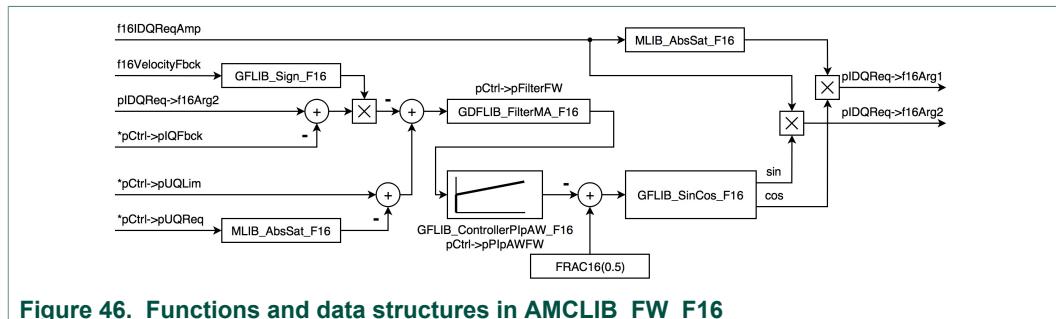
### Arguments

**Table 27. AMCLIB\_FW\_F16 arguments**

Type	Name	Direction	Description
<a href="#">tFrac16</a>	f16IDQReqAmp	input	Required amplitude of the currents Id and Iq in the two-phase rotational orthogonal system (d-q).
<a href="#">tFrac16</a>	f16VelocityFbck	input	Actual electrical angular velocity from the feedback.
<a href="#">SWLIBS_2Syst_F16</a> *const	pIDQReq	input, output	Pointer to the structure with the required stator currents in the two-phase rotational orthogonal system (d-q).
<a href="#">AMCLIB_FW_T_F16</a> *	pCtrl	input, output	Pointer to the structure with AMCLIB_FW state.

### Implementation details

The following block diagram shows the internal functions and data structures of AMCLIB\_FW\_F16.



**Figure 46. Functions and data structures in AMCLIB\_FW\_F16**

Prior to calculating the controller coefficients, it is necessary to set the scaling constants. All inputs and outputs of the algorithm are limited to the fractional range [-1, 1]. Incorrect setting of the scaling constants may lead to undesirable overflow or saturation during the computation.

**Table 28. Scaling constants**

Scaling constant	Symbol	Calculation
Maximum stator phase voltage [V]	$U_{MAX}$	$U_{MAX} = U_{DC\_Bus\_Max}$

Maximum phase current [A]	$I_{MAX}$	Maximum current of the inverter or nominal current of the motor (whichever is lower).
Maximum speed [rad/s]	$\Omega_{MAX}$	Maximum application required speed, at least the motor electrical rated speed.

An initial estimate of the field weakening PIpAW controller parameters can be taken from the speed PIpAW controller, see [AMCLIB\\_SpeedLoop\\_F16\\_Eq1](#). The upper limit of the controller output shall be set to zero, i.e.  $pPIpAWFW.f16UpperLimit = 0$ . The lower limit shall be set to an adequate value from the interval  $<FRAC16(-0.5); 0>$ . It is important to set the lower limit correctly to prevent motor damage by irreversible demagnetization of the permanent magnets. Let  $I_{D\_MAX}$  be the maximum permitted negative d-axis current for field weakening, then

$$pPIpAWFW.f16LowerLimit = FRAC16\left(\frac{1}{\pi} \cdot \cos^{-1}\left(\frac{|I_{D\_MAX}|}{I_{MAX}}\right)\right) - 0.5$$

Equation AMCLIB\_FW\_F16\_Eq1

Further refinement of the field weakening PIpAW controller parameters should be done interactively based on the observed performance. This can be easily achieved with the [FREEMASTER](#) real-time debugging tool. The following points should be considered:

- The field weakening PIpAW controller tweaks the angle of the current space vector  $I_S$  to produce the negative flux-producing current component  $I_D$  and sets the limits of the torque-producing current component  $I_Q$ .
- The magnitude of  $I_D$  depends also on the output of the speed PI controller.
- There are two different paths producing the PIpAW input error. The weight of each depends on the motor operation mode.

The performance of the field weakening controller can be affected by measurement noise. The noise immunity can be improved by tweaking the smoothing factor of the moving average filter [GDFLIB\\_FilterMA\\_F16](#) which preprocesses the input to the field weakening controller.

**Note:** Due to effectivity reasons, this function is implemented using inline assembly and is therefore not ANSI-C compliant.

### Code Example

```
#include "amclib.h"

AMCLIB_FW_T_F16 FWState;
SWLIBS_2Syst_F16 IDQReq;           // required dq currents
SWLIBS_2Syst_F16 f16IDQFbck;      // calculated dq currents from the feedback
SWLIBS_2Syst_F16 f16UDQReq;        // required dq voltages
tFrac16 f16VelocityReq;           // required velocity
tFrac16 f16VelocityFbck;          // actual velocity
AMCLIB_CURRENT_LOOP_T_F16 CurrentLoop;

void main (void)
{
    tFrac16 f16FilterMAFWOut;
    tFrac16 f16ControllerPIpAWFWOut;

    // Initialize the parameters and pointers in FWState
    FW.pFilterFW.u16NSamples = 3u;
}
```

```
FW.pPIpAWFW.f16PropGain      = (tFrac16)FRAC16(0.2348);  
FW.pPIpAWFW.f16IntegGain    = (tFrac16)FRAC16(0.3457);  
FW.pPIpAWFW.s16PropGainShift = 1;  
FW.pPIpAWFW.s16IntegGainShift = 1;  
FW.pPIpAWFW.f16LowerLimit   = (tFrac16)-30706;  
FW.pPIpAWFW.f16UpperLimit   = (tFrac16)31568;  
FW.pIQFbck = &f16IDQFbck.f16Arg2;  
FW.pUQReq = &f16UDQReq.f16Arg2;  
FW.pUQLim = &CurrentLoop.pPIrAWQ.f16UpperLimit;  
  
// Clear AMCLIB_FW state variables  
// Alternative 1: API call with postfix  
// (only one alternative shall be used).  
AMCLIB_FWInit_F16(&FWState);  
  
// Alternative 2: API call with implementation parameter  
// (only one alternative shall be used).  
AMCLIB_FWInit(&FWState, F16);  
  
// Alternative 3: API call with global configuration of implementation  
// (only one alternative shall be used). This alternative is available  
// only if 16-bit fractional implementation is selected as default.  
AMCLIB_FWInit(&FWState);  
  
// Initialize the AMCLIB_FW state variables to predefined values  
// Warning: Parameters in FWState must be already initialized.  
f16FilterMAFWOut = (tFrac16)123;  
f16ControllerPIpAWFWOut = (tFrac16)123;  
// Alternative 1: API call with postfix  
// (only one alternative shall be used).  
AMCLIB_FWSetState_F16(f16FilterMAFWOut, f16ControllerPIpAWFWOut,  
    &FWState);  
  
// Alternative 2: API call with implementation parameter  
// (only one alternative shall be used).  
AMCLIB_FWSetState(f16FilterMAFWOut, f16ControllerPIpAWFWOut,  
    &FWState, F16);  
  
// Alternative 3: API call with global configuration of implementation  
// (only one alternative shall be used). This alternative is available  
// only if 16-bit fractional implementation is selected as default.  
AMCLIB_FWSetState(f16FilterMAFWOut, f16ControllerPIpAWFWOut,  
    &FWState);  
  
f16VelocityReq = (tFrac16)100;  
while(1);  
}  
  
// Periodical function or interrupt - speed control loop  
void SlowLoop(void)  
{  
    tFrac16 f16IDQReqAmp;  
  
    // Calculate f16IDQReqAmp from f16VelocityReq and f16VelocityFbck  
    // using AMCLIB_SpeedLoop  
    // (...)  
  
    // Alternative 1: API call with postfix  
    // (only one alternative shall be used).  
    AMCLIB_FW_F16(f16IDQReqAmp, f16VelocityFbck, &IDQReq, &FWState);
```

```

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
AMCLIB_FW(f16IDQReqAmp, f16VelocityFbck, &IDQReq, &FWState, F16);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 16-bit fractional implementation is selected as default.
AMCLIB_FW(f16IDQReqAmp, f16VelocityFbck, &IDQReq, &FWState);
}

// Periodical function or interrupt - current control loop
void FastLoop(void)
{
    // Calculate new values of f16VelocityFbck, f16IDQFbck, f16UDQReq
    // (...)

}

```

### 2.4.3 Function AMCLIB\_FW\_FLT

#### Declaration

```
void AMCLIB_FW_FLT(tFloat fltIDQReqAmp, tFloat fltVelocityFbck,
SWLIBS\_2SystFLT *const pIDQReq, AMCLIB\_FW\_T\_FLT *pCtrl);
```

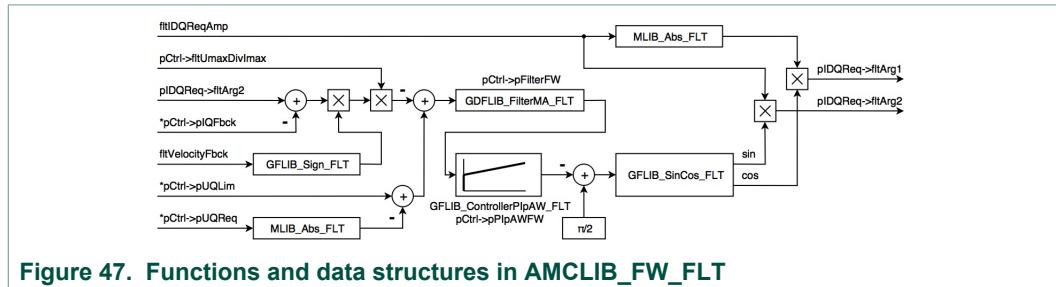
#### Arguments

**Table 29. AMCLIB\_FW\_FLT arguments**

Type	Name	Direction	Description
<a href="#">tFloat</a>	fltIDQReqAmp	input	Required amplitude of the currents Id and Iq in the two-phase rotational orthogonal system (d-q).
<a href="#">tFloat</a>	fltVelocityFbck	input	Actual electrical angular velocity from the feedback.
<a href="#">SWLIBS_2SystFLT</a> *const	pIDQReq	input, output	Pointer to the structure with the required stator currents in the two-phase rotational orthogonal system (d-q).
<a href="#">AMCLIB_FW_T_FLT</a> *	pCtrl	input, output	Pointer to the structure with AMCLIB_FW state.

#### Implementation details

The following block diagram shows the internal functions and data structures of AMCLIB\_FW\_FLT.



**Figure 47. Functions and data structures in AMCLIB\_FW\_FLT**

An initial estimate of the field weakening PIPAW controller parameters can be taken from the speed PIPAW controller, see [AMCLIB\\_SpeedLoop\\_FLT\\_Eq1](#). The upper limit of the controller output shall be set to zero, i.e. pPIPWF.fltUpperLimit = 0. The lower

limit shall be set to an adequate value from the interval  $<-\pi/2; 0>$ . It is important to set the lower limit correctly to prevent motor damage by irreversible demagnetization of the permanent magnets. Let  $I_{D\_MAX}$  be the maximum permitted negative d-axis current for field weakening, then

$$pPIpAWFW.fltLowerLimit = \cos^{-1}\left(\frac{|I_{D\_MAX}|}{I_{MAX}}\right) - \frac{\pi}{2}$$

Equation AMCLIB\_FW\_FLT\_Eq1

Further refinement of the field weakening PIpAW controller parameters should be done interactively based on the observed performance. This can be easily achieved with the [FREEMASTER](#) real-time debugging tool. The following points should be considered:

- The field weakening PIpAW controller tweaks the angle of the current space vector  $I_S$  to produce the negative flux-producing current component  $I_D$  and sets the limits of the torque-producing current component  $I_Q$ .
- The magnitude of  $I_D$  depends also on the output of the speed PI controller.
- There are two different paths producing the PIpAW input error. The weight of each depends on the motor operation mode.

The performance of the field weakening controller can be affected by measurement noise. The noise immunity can be improved by tweaking the smoothing factor of the moving average filter [GDFLIB\\_FilterMA\\_FLT](#) which preprocesses the input to the field weakening controller.

**Note:** *The function may raise floating-point exceptions (invalid operation, overflow, underflow, inexact, input denormal). The floating-point unit must be enabled in CPACR register to prevent NOCP UsageFault exception.*

### Code Example

```
#include "amclib.h"

AMCLIB_FW_T_FLT FWState;
SWLIBS_2Syst_FLT IDQReq;           // required dq currents
SWLIBS_2Syst_FLT fltIDQFbck;      // calculated dq currents from the feedback
SWLIBS_2Syst_FLT fltUDQReq;        // required dq voltages
tFloat fltVelocityReq;             // required velocity
tFloat fltVelocityFbck;            // actual velocity
AMCLIB_CURRENT_LOOP_T_FLT CurrentLoop;

void main (void)
{
    tFloat fltFilterMAFWOut;
    tFloat fltControllerPIpAWFWOut;

    // Initialize the parameters and pointers in FWState
    FW.pFilterFW.fltLambda          = (tFloat)0.6;
    FW.pPIpAWFW.fltPropGain         = (tFloat)0.2348;
    FW.pPIpAWFW.fltIntegGain        = (tFloat)0.3457;
    FW.pPIpAWFW.fltLowerLimit       = (tFloat)-0.937099993228912;
    FW.pPIpAWFW.fltUpperLimit       = (tFloat)0.963400006294251;
    FW.pIQFbck = &fltIDQFbck.fltArg2;
    FW.pUQReq = &fltUDQReq.fltArg2;
    FW.pUQLim = &CurrentLoop.pPIrAWQ.fltUpperLimit;
    FW.fltUmaxDivImax              = (tFloat)1.0;
}
```

```
// Clear AMCLIB_FW state variables
// Alternative 1: API call with postfix
// (only one alternative shall be used).
AMCLIB_FWInit_FLT(&FWState);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
AMCLIB_FWInit(&FWState, FLT);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if single precision floating point implementation is selected
// as default.
AMCLIB_FWInit(&FWState);

// Initialize the AMCLIB_FW state variables to predefined values
// Warning: Parameters in FWState must be already initialized.
fltFilterMAFWOut = 123.0F;
fltControllerPIpAWFWOut = 123.0F;
// Alternative 1: API call with postfix
// (only one alternative shall be used).
AMCLIB_FWSetState_FLT(fltFilterMAFWOut, fltControllerPIpAWFWOut,
    &FWState);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
AMCLIB_FWSetState(fltFilterMAFWOut, fltControllerPIpAWFWOut,
    &FWState, FLT);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if single precision floating point implementation is selected
// as default.
AMCLIB_FWSetState(fltFilterMAFWOut, fltControllerPIpAWFWOut,
    &FWState);

fltVelocityReq = 100.0F;
while(1);
}

// Periodical function or interrupt - speed control loop
void SlowLoop(void)
{
    tFloat fltIDQReqAmp;

    // Calculate fltIDQReqAmp from fltVelocityReq and fltVelocityFbck
    // using AMCLIB_SpeedLoop
    // (...)

    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB_FW_FLT(fltIDQReqAmp, fltVelocityFbck, &IDQReq, &FWState);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB_FW(fltIDQReqAmp, fltVelocityFbck, &IDQReq, &FWState, FLT);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if single precision floating point implementation is selected
```

```

    // as default.
    AMCLIB_FW(fltIDQReqAmp, fltVelocityFbck, &IDQReq, &FWState);
}

// Periodical function or interrupt - current control loop
void FastLoop(void)
{
    // Calculate new values of fltVelocityFbck, fltIDQFbck, fltUDQReq
    // (...)
}

```

#### 2.4.4 Function AMCLIB\_FWInit

##### Description

This function clears the AMCLIB\_FW state variables.

**Note:** The input/output pointers must contain valid addresses, otherwise a fault may occur (MemManage, BusFault, UsageFault, HardFault).

##### Re-entrancy

The function is re-entrant for a different pCtrl.

##### 2.4.4.1 Function AMCLIB\_FWInit\_F32

###### Declaration

```
void AMCLIB_FWInit_F32(AMCLIB\_FW\_T\_F32 *const pCtrl);
```

###### Arguments

Table 30. AMCLIB\_FWInit\_F32 arguments

Type	Name	Direction	Description
<a href="#">AMCLIB_FW_T_F32</a> *const	pCtrl	input, output	Pointer to the structure with AMCLIB_FW state.

**Note:** If pCtrl points to a structure of type [AMCLIB\\_FW\\_DEBUG\\_T\\_F32](#), it must be recasted to [AMCLIB\\_FW\\_T\\_F32](#).

###### Code Example

```

#include "amclib.h"

AMCLIB\_FW\_T\_F32 FWState;
SWLIBS\_2Syst\_F32 IDQReq;           // required dq currents
SWLIBS\_2Syst\_F32 f32IDQFbck;      // calculated dq currents from the feedback
SWLIBS\_2Syst\_F32 f32UDQReq;        // required dq voltages
tFrac32 f32VelocityReq;         // required velocity
tFrac32 f32VelocityFbck;        // actual velocity
AMCLIB\_CURRENT\_LOOP\_T\_F32 CurrentLoop;

void main (void)
{
    tFrac32 f32FilterMAFWOut;
}

```

```
tFrac32 f32ControllerPIpAWFWOut;

// Initialize the parameters and pointers in FWState
FW.pFilterFW.u16NSamples      = 3u;
FW.pPIpAWFW.f32PropGain       = (tFrac32)FRAC32(0.2348);
FW.pPIpAWFW.f32IntegGain      = (tFrac32)FRAC32(0.3457);
FW.pPIpAWFW.s16PropGainShift  = 1;
FW.pPIpAWFW.s16IntegGainShift = 1;
FW.pPIpAWFW.f32LowerLimit    = (tFrac32)-2012406926L;
FW.pPIpAWFW.f32UpperLimit    = (tFrac32)2068885746L;
FW.pIQFbck = &f32IDQFbck.f32Arg2;
FW.pUQReq = &f32UDQReq.f32Arg2;
FW.pUQLim = &CurrentLoop.pPIrAWQ.f32UpperLimit;

// Clear AMCLIB_FW state variables
// Alternative 1: API call with postfix
// (only one alternative shall be used).
AMCLIB_FWInit_F32(&FWState);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
AMCLIB_FWInit(&FWState, F32);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 32-bit fractional implementation is selected as default.
AMCLIB_FWInit(&FWState);

// Initialize the AMCLIB_FW state variables to predefined values
// Warning: Parameters in FWState must be already initialized.
f32FilterMAFWOut = (tFrac32)123L;
f32ControllerPIpAWFWOut = (tFrac32)123L;
// Alternative 1: API call with postfix
// (only one alternative shall be used).
AMCLIB_FWSetState_F32(f32FilterMAFWOut, f32ControllerPIpAWFWOut,
                      &FWState);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
AMCLIB_FWSetState(f32FilterMAFWOut, f32ControllerPIpAWFWOut,
                  &FWState, F32);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 32-bit fractional implementation is selected as default.
AMCLIB_FWSetState(f32FilterMAFWOut, f32ControllerPIpAWFWOut,
                  &FWState);

f32VelocityReq = (tFrac32)100L;
while(1);

}

// Periodical function or interrupt - speed control loop
void SlowLoop(void)
{
    tFrac32 f32IDQReqAmp;

    // Calculate f32IDQReqAmp from f32VelocityReq and f32VelocityFbck
    // using AMCLIB_SpeedLoop
    // (...)
```

```

// Alternative 1: API call with postfix
// (only one alternative shall be used).
AMCLIB_FW_F32(f32IDQReqAmp, f32VelocityFbck, &IDQReq, &FWState);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
AMCLIB_FW(f32IDQReqAmp, f32VelocityFbck, &IDQReq, &FWState, F32);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 32-bit fractional implementation is selected as default.
AMCLIB_FW(f32IDQReqAmp, f32VelocityFbck, &IDQReq, &FWState);
}

// Periodical function or interrupt - current control loop
void FastLoop(void)
{
    // Calculate new values of f32VelocityFbck, f32IDQFbck, f32UDQReq
    // (...)

}

```

#### 2.4.4.2 Function AMCLIB\_FWInit\_F16

##### Declaration

```
void AMCLIB_FWInit_F16(AMCLIB_FW_T_F16 *const pCtrl);
```

##### Arguments

**Table 31. AMCLIB\_FWInit\_F16 arguments**

Type	Name	Direction	Description
AMCLIB_FW_T_F16 *const	pCtrl	input, output	Pointer to the structure with AMCLIB_FW state.

**Note:** If pCtrl points to a structure of type AMCLIB\_FW\_DEBUG\_T\_F16, it must be recasted to AMCLIB\_FW\_T\_F16 \*.

##### Code Example

```

#include "amclib.h"

AMCLIB_FW_T_F16 FWState;
SWLIBS_2Syst_F16 IDQReq;           // required dq currents
SWLIBS_2Syst_F16 f16IDQFbck;      // calculated dq currents from the feedback
SWLIBS_2Syst_F16 f16UDQReq;        // required dq voltages
tFrac16 f16VelocityReq;           // required velocity
tFrac16 f16VelocityFbck;          // actual velocity
AMCLIB_CURRENT_LOOP_T_F16 CurrentLoop;

void main (void)
{
    tFrac16 f16FilterMAFWOut;
    tFrac16 f16ControllerPIpAWFWOut;

    // Initialize the parameters and pointers in FWState
    FW.pFilterFW.ul6NSamples = 3u;
}
```

```
FW.pPIpAWFW.f16PropGain      = (tFrac16)FRAC16(0.2348);  
FW.pPIpAWFW.f16IntegGain    = (tFrac16)FRAC16(0.3457);  
FW.pPIpAWFW.s16PropGainShift = 1;  
FW.pPIpAWFW.s16IntegGainShift = 1;  
FW.pPIpAWFW.f16LowerLimit   = (tFrac16) -30706;  
FW.pPIpAWFW.f16UpperLimit   = (tFrac16) 31568;  
FW.pIQFbck = &f16IDQFbck.f16Arg2;  
FW.pUQReq = &f16UDQReq.f16Arg2;  
FW.pUQLim = &CurrentLoop.pPIrAWQ.f16UpperLimit;  
  
// Clear AMCLIB_FW state variables  
// Alternative 1: API call with postfix  
// (only one alternative shall be used).  
AMCLIB_FWInit_F16(&FWState);  
  
// Alternative 2: API call with implementation parameter  
// (only one alternative shall be used).  
AMCLIB_FWInit(&FWState, F16);  
  
// Alternative 3: API call with global configuration of implementation  
// (only one alternative shall be used). This alternative is available  
// only if 16-bit fractional implementation is selected as default.  
AMCLIB_FWInit(&FWState);  
  
// Initialize the AMCLIB_FW state variables to predefined values  
// Warning: Parameters in FWState must be already initialized.  
f16FilterMAFWOut = (tFrac16)123;  
f16ControllerPIpAWFWOut = (tFrac16)123;  
// Alternative 1: API call with postfix  
// (only one alternative shall be used).  
AMCLIB\_FWSetState\_F16(f16FilterMAFWOut, f16ControllerPIpAWFWOut,  
    &FWState);  
  
// Alternative 2: API call with implementation parameter  
// (only one alternative shall be used).  
AMCLIB_FWSetState(f16FilterMAFWOut, f16ControllerPIpAWFWOut,  
    &FWState, F16);  
  
// Alternative 3: API call with global configuration of implementation  
// (only one alternative shall be used). This alternative is available  
// only if 16-bit fractional implementation is selected as default.  
AMCLIB_FWSetState(f16FilterMAFWOut, f16ControllerPIpAWFWOut,  
    &FWState);  
  
f16VelocityReq = (tFrac16)100;  
while(1);  
}  
  
// Periodical function or interrupt - speed control loop  
void SlowLoop(void)  
{  
    tFrac16 f16IDQReqAmp;  
  
    // Calculate f16IDQReqAmp from f16VelocityReq and f16VelocityFbck  
    // using AMCLIB_SpeedLoop  
    // (...)  
  
    // Alternative 1: API call with postfix  
    // (only one alternative shall be used).  
AMCLIB\_FW\_F16(f16IDQReqAmp, f16VelocityFbck, &IDQReq, &FWState);
```

```

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
AMCLIB_FW(f16IDQReqAmp, f16VelocityFbck, &IDQReq, &FWState, F16);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 16-bit fractional implementation is selected as default.
AMCLIB_FW(f16IDQReqAmp, f16VelocityFbck, &IDQReq, &FWState);
}

// Periodical function or interrupt - current control loop
void FastLoop(void)
{
    // Calculate new values of f16VelocityFbck, f16IDQFbck, f16UDQReq
    // (...)

}

```

#### 2.4.4.3 Function AMCLIB\_FWInit\_FLT

##### Declaration

```
void AMCLIB_FWInit_FLT(AMCLIB\_FW\_T\_FLT *const pCtrl);
```

##### Arguments

**Table 32. AMCLIB\_FWInit\_FLT arguments**

Type	Name	Direction	Description
<a href="#">AMCLIB_FW_T_FLT</a> *const	pCtrl	input, output	Pointer to the structure with AMCLIB_FW state.

**Note:** If pCtrl points to a structure of type [AMCLIB\\_FW\\_DEBUG\\_T\\_FLT](#), it must be recasted to [AMCLIB\\_FW\\_T\\_FLT](#).

##### Code Example

```

#include "amclib.h"

AMCLIB_FW_T_FLT FWState;
SWLIBS_2Syst_FLT IDQReq;          // required dq currents
SWLIBS_2Syst_FLT fltIDQFbck;     // calculated dq currents from the feedback
SWLIBS_2Syst_FLT fltUDQReq;      // required dq voltages
tFloat fltVelocityReq;           // required velocity
tFloat fltVelocityFbck;          // actual velocity
AMCLIB_CURRENT_LOOP_T_FLT CurrentLoop;

void main (void)
{
    tFloat fltFilterMAFWOut;
    tFloat fltControllerPIpAWFWOut;

    // Initialize the parameters and pointers in FWState
    FW.pFilterFW.fltLambda        = (tFloat) 0.6;
    FW.pPIpAWFW.fltPropGain       = (tFloat) 0.2348;
    FW.pPIpAWFW.fltIntegGain      = (tFloat) 0.3457;
    FW.pPIpAWFW.fltLowerLimit     = (tFloat) -0.937099993228912;
    FW.pPIpAWFW.fltUpperLimit     = (tFloat) 0.963400006294251;
}

```

```
FW.pIQFbck = &fltIDQFbck.fltArg2;
FW.pUQReq = &fltUDQReq.fltArg2;
FW.pUQLim = &CurrentLoop.pPIrAWQ.fltUpperLimit;
FW.fltUmaxDivImax = (tFloat)1.0;

// Clear AMCLIB_FW state variables
// Alternative 1: API call with postfix
// (only one alternative shall be used).
AMCLIB_FWInit_FLT(&FWState);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
AMCLIB_FWInit(&FWState, FLT);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if single precision floating point implementation is selected
// as default.
AMCLIB_FWInit(&FWState);

// Initialize the AMCLIB_FW state variables to predefined values
// Warning: Parameters in FWState must be already initialized.
fltFilterMAFWOut = 123.0F;
fltControllerPIpAWFWOut = 123.0F;
// Alternative 1: API call with postfix
// (only one alternative shall be used).
AMCLIB_FWSetState_FLT(fltFilterMAFWOut, fltControllerPIpAWFWOut,
&FWState);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
AMCLIB_FWSetState(fltFilterMAFWOut, fltControllerPIpAWFWOut,
&FWState, FLT);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if single precision floating point implementation is selected
// as default.
AMCLIB_FWSetState(fltFilterMAFWOut, fltControllerPIpAWFWOut,
&FWState);

fltVelocityReq = 100.0F;
while(1);
}

// Periodical function or interrupt - speed control loop
void SlowLoop(void)
{
    tFloat fltIDQReqAmp;

// Calculate fltIDQReqAmp from fltVelocityReq and fltVelocityFbck
// using AMCLIB_SpeedLoop
// (...)

// Alternative 1: API call with postfix
// (only one alternative shall be used).
AMCLIB_FW_FLT(fltIDQReqAmp, fltVelocityFbck, &IDQReq, &FWState);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
```

```

AMCLIB_FW(fltIDQReqAmp, fltVelocityFbck, &IDQReq, &FWState, FLT);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if single precision floating point implementation is selected
// as default.
AMCLIB_FW(fltIDQReqAmp, fltVelocityFbck, &IDQReq, &FWState);
}

// Periodical function or interrupt - current control loop
void FastLoop(void)
{
    // Calculate new values of fltVelocityFbck, fltIDQFbck, fltUDQReq
    // (...)

}

```

## 2.4.5 Function AMCLIB\_FWSetState

### Description

This function initializes the AMCLIB\_FW state variables to achieve the required output values.

**Note:** The input/output pointers must contain valid addresses, otherwise a fault may occur (MemManage, BusFault, UsageFault, HardFault).

### Re-entrancy

The function is re-entrant for a different pCtrl.

#### 2.4.5.1 Function AMCLIB\_FWSetState\_F32

##### Declaration

```
void AMCLIB_FWSetState_F32(tFrac32 f32FilterMAFWOut, tFrac32
f32ControllerPIpAWFWOut, AMCLIB\_FW\_T\_F32 *pCtrl);
```

##### Arguments

Table 33. AMCLIB\_FWSetState\_F32 arguments

Type	Name	Direction	Description
<a href="#">tFrac32</a>	f32FilterMAFWOut	input	Required output of the FilterMA.
<a href="#">tFrac32</a>	f32ControllerPIpAWFWOut	output	Required output of the ControllerPIpAW.
<a href="#">AMCLIB_FW_T_F32</a> *	pCtrl	input, output	Pointer to the structure with AMCLIB_FW state.

**Caution:** Set the parameters in the structure pointed to by pCtrl before calling this function.

**Note:** If pCtrl points to a structure of type [AMCLIB\\_FW\\_DEBUG\\_T\\_F32](#), it must be recasted to [AMCLIB\\_FW\\_T\\_F32](#) \*.

### Code Example

```
#include "amclib.h"
```

```
AMCLIB_FW_T_F32 FWState;
SWLIBS_2Syst_F32 IDQReq;           // required dq currents
SWLIBS_2Syst_F32 f32IDQFbck;     // calculated dq currents from the feedback
SWLIBS_2Syst_F32 f32UDQReq;       // required dq voltages
tFrac32 f32VelocityReq;          // required velocity
tFrac32 f32VelocityFbck;         // actual velocity
AMCLIB_CURRENT_LOOP_T_F32 CurrentLoop;

void main (void)
{
    tFrac32 f32FilterMAFWOut;
    tFrac32 f32ControllerPIpAWFWOut;

    // Initialize the parameters and pointers in FWState
    FW.pFilterFW.u16NSamples      = 3u;
    FW.pPIpAWFW.f32PropGain      = (tFrac32)FRAC32(0.2348);
    FW.pPIpAWFW.f32IntegGain     = (tFrac32)FRAC32(0.3457);
    FW.pPIpAWFW.s16PropGainShift = 1;
    FW.pPIpAWFW.s16IntegGainShift = 1;
    FW.pPIpAWFW.f32LowerLimit   = (tFrac32)-2012406926L;
    FW.pPIpAWFW.f32UpperLimit   = (tFrac32)2068885746L;
    FW.pIQFbck = &f32IDQFbck.f32Arg2;
    FW.pUQReq = &f32UDQReq.f32Arg2;
    FW.puQLim = &CurrentLoop.pPIrAWQ.f32UpperLimit;

    // Clear AMCLIB_FW state variables
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB_FWInit_F32(&FWState);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB_FWInit(&FWState, F32);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 32-bit fractional implementation is selected as default.
    AMCLIB_FWInit(&FWState);

    // Initialize the AMCLIB_FW state variables to predefined values
    // Warning: Parameters in FWState must be already initialized.
    f32FilterMAFWOut = (tFrac32)123L;
    f32ControllerPIpAWFWOut = (tFrac32)123L;
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB_FWSetState_F32(f32FilterMAFWOut, f32ControllerPIpAWFWOut,
                          &FWState);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB_FWSetState(f32FilterMAFWOut, f32ControllerPIpAWFWOut,
                      &FWState, F32);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 32-bit fractional implementation is selected as default.
    AMCLIB_FWSetState(f32FilterMAFWOut, f32ControllerPIpAWFWOut,
                      &FWState);
```

```

f32VelocityReq = (tFrac32)100L;
while(1);
}

// Periodical function or interrupt - speed control loop
void SlowLoop(void)
{
    tFrac32 f32IDQReqAmp;

    // Calculate f32IDQReqAmp from f32VelocityReq and f32VelocityFbck
    // using AMCLIB_SpeedLoop
    // (...)

    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB\_FW\_F32(f32IDQReqAmp, f32VelocityFbck, &IDQReq, &FWState);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB_FW(f32IDQReqAmp, f32VelocityFbck, &IDQReq, &FWState, F32);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 32-bit fractional implementation is selected as default.
    AMCLIB_FW(f32IDQReqAmp, f32VelocityFbck, &IDQReq, &FWState);
}

// Periodical function or interrupt - current control loop
void FastLoop(void)
{
    // Calculate new values of f32VelocityFbck, f32IDQFbck, f32UDQReq
    // (...)

}

```

#### 2.4.5.2 Function AMCLIB\_FWSetState\_F16

##### Declaration

```
void AMCLIB_FWSetState_F16(tFrac16 f16FilterMAFWOut, tFrac16
f16ControllerPipAWFWOut, AMCLIB\_FW\_T\_F16 *pCtrl);
```

##### Arguments

**Table 34. AMCLIB\_FWSetState\_F16 arguments**

Type	Name	Direction	Description
<a href="#">tFrac16</a>	f16FilterMAFWOut	input	Required output of the FilterMA.
<a href="#">tFrac16</a>	f16ControllerPipAWFWOut	input	Required output of the ControllerPipAW.
<a href="#">AMCLIB_FW_T_F16</a> *	pCtrl	input, output	Pointer to the structure with AMCLIB_FW state.

**Caution:** Set the parameters in the structure pointed to by pCtrl before calling this function.

**Note:** If pCtrl points to a structure of type [AMCLIB\\_FW\\_DEBUG\\_T\\_F16](#), it must be recasted to [AMCLIB\\_FW\\_T\\_F16](#) \*.

## Code Example

```
#include "amclib.h"

AMCLIB_FW_T_F16 FWState;
SWLIBS_2Syst_F16 IDQReq;           // required dq currents
SWLIBS_2Syst_F16 f16IDQFbck;      // calculated dq currents from the feedback
SWLIBS_2Syst_F16 f16UDQReq;       // required dq voltages
tFrac16 f16VelocityReq;          // required velocity
tFrac16 f16VelocityFbck;         // actual velocity
AMCLIB_CURRENT_LOOP_T_F16 CurrentLoop;

void main (void)
{
    tFrac16 f16FilterMAFWOut;
    tFrac16 f16ControllerPIpAWFWOut;

    // Initialize the parameters and pointers in FWState
    FW.pFilterFW.ul6NSamples = 3u;
    FW.pPIpAWFW.f16PropGain = (tFrac16)FRAC16(0.2348);
    FW.pPIpAWFW.f16IntegGain = (tFrac16)FRAC16(0.3457);
    FW.pPIpAWFW.s16PropGainShift = 1;
    FW.pPIpAWFW.s16IntegGainShift = 1;
    FW.pPIpAWFW.f16LowerLimit = (tFrac16)-30706;
    FW.pPIpAWFW.f16UpperLimit = (tFrac16)31568;
    FW.pIQFbck = &f16IDQFbck.f16Arg2;
    FW.pUQReq = &f16UDQReq.f16Arg2;
    FW.pUQLim = &CurrentLoop.pPIrAWQ.f16UpperLimit;

    // Clear AMCLIB_FW state variables
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB_FWInit_F16(&FWState);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB_FWInit(&FWState, F16);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 16-bit fractional implementation is selected as default.
    AMCLIB_FWInit(&FWState);

    // Initialize the AMCLIB_FW state variables to predefined values
    // Warning: Parameters in FWState must be already initialized.
    f16FilterMAFWOut = (tFrac16)123;
    f16ControllerPIpAWFWOut = (tFrac16)123;
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB_FWSetState_F16(f16FilterMAFWOut, f16ControllerPIpAWFWOut,
    &FWState);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB_FWSetState(f16FilterMAFWOut, f16ControllerPIpAWFWOut,
    &FWState, F16);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
```

```

// only if 16-bit fractional implementation is selected as default.
AMCLIB_FWSetState(f16FilterMAFWOut, f16ControllerPipAWFWOut,
&FWState);

f16VelocityReq = (tFrac16)100;
while(1);
}

// Periodical function or interrupt - speed control loop
void SlowLoop(void)
{
    tFrac16 f16IDQReqAmp;

    // Calculate f16IDQReqAmp from f16VelocityReq and f16VelocityFbck
    // using AMCLIB_SpeedLoop
    // (...)

    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB_FW_F16(f16IDQReqAmp, f16VelocityFbck, &IDQReq, &FWState);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB_FW(f16IDQReqAmp, f16VelocityFbck, &IDQReq, &FWState, F16);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 16-bit fractional implementation is selected as default.
    AMCLIB_FW(f16IDQReqAmp, f16VelocityFbck, &IDQReq, &FWState);
}

// Periodical function or interrupt - current control loop
void FastLoop(void)
{
    // Calculate new values of f16VelocityFbck, f16IDQFbck, f16UDQReq
    // (...)

}

```

#### 2.4.5.3 Function AMCLIB\_FWSetState\_FLT

##### Declaration

```
void AMCLIB_FWSetState_FLT(tFloat fltFilterMAFWOut, tFloat
fltControllerPipAWFWOut, AMCLIB_FW_T_FLT *pCtrl);
```

##### Arguments

**Table 35. AMCLIB\_FWSetState\_FLT arguments**

Type	Name	Direction	Description
tFloat	fltFilterMAFWOut	input	Required output of the FilterMA.
tFloat	fltControllerPipAWFWOut	input	Required output of the ControllerPipAW.
AMCLIB_FW_T_FLT *	pCtrl	input, output	Pointer to the structure with AMCLIB_FW state.

**Caution:** Set the parameters in the structure pointed to by pCtrl before calling this function.

**Note:** If *pCtrl* points to a structure of type `AMCLIB_FW_DEBUG_T_FLT`, it must be recasted to `AMCLIB_FW_T_FLT` \*.

### Code Example

```
#include "amclib.h"

AMCLIB_FW_T_FLT FWState;
SWLIBS_2Syst_FLT IDQReq;           // required dq currents
SWLIBS_2Syst_FLT fltIDQFbck;      // calculated dq currents from the feedback
SWLIBS_2Syst_FLT fltUDQReq;       // required dq voltages
tFloat fltVelocityReq;            // required velocity
tFloat fltVelocityFbck;           // actual velocity
AMCLIB_CURRENT_LOOP_T_FLT CurrentLoop;

void main (void)
{
    tFloat fltFilterMAFWOut;
    tFloat fltControllerPIpAWFWOut;

    // Initialize the parameters and pointers in FWState
    FW.pFilterFW.fltLambda        = (tFloat)0.6;
    FW.pPIpAWFW.fltPropGain       = (tFloat)0.2348;
    FW.pPIpAWFW.fltIntegGain      = (tFloat)0.3457;
    FW.pPIpAWFW.fltLowerLimit     = (tFloat)-0.937099993228912;
    FW.pPIpAWFW.fltUpperLimit     = (tFloat)0.963400006294251;
    FW.pIQFbck = &fltIDQFbck.fltArg2;
    FW.pUQReq  = &fltUDQReq.fltArg2;
    FW.pUQLim  = &CurrentLoop.pPIrAWQ.fltUpperLimit;
    FW.fltUmaxDivImax            = (tFloat)1.0;

    // Clear AMCLIB_FW state variables
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB_FWInit_FLT(&FWState);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB_FWInit(&FWState, FLT);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if single precision floating point implementation is selected
    // as default.
    AMCLIB_FWInit(&FWState);

    // Initialize the AMCLIB_FW state variables to predefined values
    // Warning: Parameters in FWState must be already initialized.
    fltFilterMAFWOut = 123.0F;
    fltControllerPIpAWFWOut = 123.0F;
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB_FWSetState_FLT(fltFilterMAFWOut, fltControllerPIpAWFWOut,
                          &FWState);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB_FWSetState(fltFilterMAFWOut, fltControllerPIpAWFWOut,
```

```
&FWState, FLT);  
  
// Alternative 3: API call with global configuration of implementation  
// (only one alternative shall be used). This alternative is available  
// only if single precision floating point implementation is selected  
// as default.  
AMCLIB_FWSetState(fltFilterMAFWOut, fltControllerPIpAWFWOut,  
    &FWState);  
  
    fltVelocityReq = 100.0F;  
    while(1);  
}  
  
// Periodical function or interrupt - speed control loop  
void SlowLoop(void)  
{  
    tFloat fltIDQReqAmp;  
  
    // Calculate fltIDQReqAmp from fltVelocityReq and fltVelocityFbck  
    // using AMCLIB_SpeedLoop  
    // (...)  
  
    // Alternative 1: API call with postfix  
    // (only one alternative shall be used).  
    AMCLIB_FW_FLT(fltIDQReqAmp, fltVelocityFbck, &IDQReq, &FWState);  
  
    // Alternative 2: API call with implementation parameter  
    // (only one alternative shall be used).  
    AMCLIB_FW(fltIDQReqAmp, fltVelocityFbck, &IDQReq, &FWState, FLT);  
  
    // Alternative 3: API call with global configuration of implementation  
    // (only one alternative shall be used). This alternative is available  
    // only if single precision floating point implementation is selected  
    // as default.  
    AMCLIB_FW(fltIDQReqAmp, fltVelocityFbck, &IDQReq, &FWState);  
}  
  
// Periodical function or interrupt - current control loop  
void FastLoop(void)  
{  
    // Calculate new values of fltVelocityFbck, fltIDQFbck, fltUDQReq  
    // (...)  
}
```

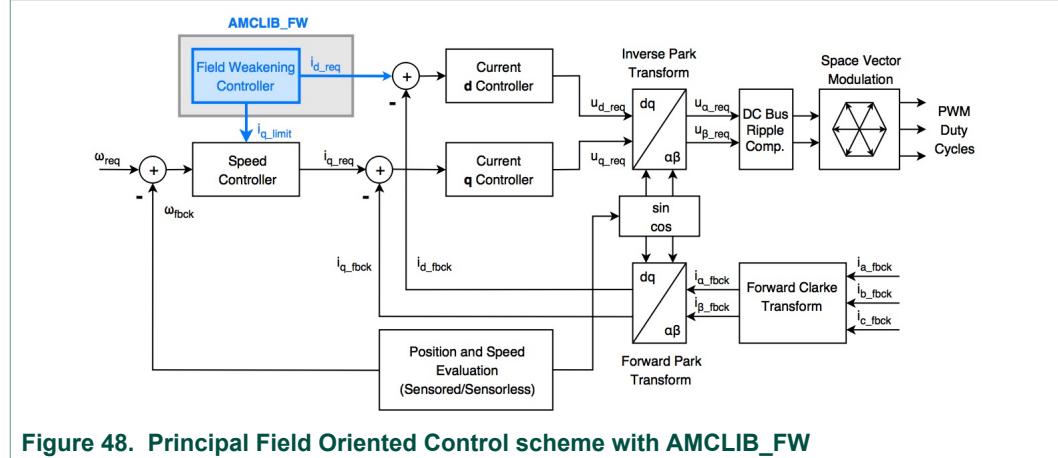
## 2.4.6 Function AMCLIB\_FWDebug

This function implements the PMSM Field Weakening controller. Debugging information is provided.

### Description

AMCLIB\_FWDebug implements the field weakening controller in the outer control loop highlighted in [Figure 48](#). AMCLIB\_FWDebug is intended to be used in combination with [AMCLIB\\_SpeedLoopDebug](#). The code can be simplified further by utilizing [AMCLIB\\_FWSpeedLoopDebug](#) which combines the speed controller with the field weakening controller in one library function. AMCLIB\_FWDebug provides the same functionality as [AMCLIB\\_FW](#). Additionally, this function allows debugging of all internal

variables. The debugging outputs are provided in the structure pointed to by pCtrl. Replace AMCLIB\_FWDebug by [AMCLIB\\_FW](#) once the debugging is finished.



**Figure 48. Principal Field Oriented Control scheme with AMCLIB\_FW**

Field weakening technique employed in AMCLIB\_FWDebug extends the speed range of PMSM beyond the motor base speed by reducing the linkage magnetic flux through negative current  $i_d_{req}$ . See [AMCLIB\\_FW](#) for more details.

#### 2.4.6.1 Function AMCLIB\_FWDebug\_F32

##### Declaration

```
void AMCLIB_FWDebug_F32 (tFrac32 f32IDQReqAmp, tFrac32
f32VelocityFbck, SWLIBS_2Syst_F32 *const pIDQReq,
AMCLIB_FW_DEBUG_T_F32 *pCtrl);
```

##### Arguments

**Table 36. AMCLIB\_FWDebug\_F32 arguments**

Type	Name	Direction	Description
tFrac32	f32IDQReqAmp	input	Required amplitude of the currents $i_d$ and $i_q$ in the two-phase rotational orthogonal system (d-q).
tFrac32	f32VelocityFbck	input	Actual electrical angular velocity from the feedback.
SWLIBS_2Syst_F32 *const	pIDQReq	input, output	Pointer to the structure with the required stator currents in the two-phase rotational orthogonal system (d-q).
AMCLIB_FW_DEBUG_T_F32 *	pCtrl	input, output	Pointer to the structure with AMCLIB_FW state and debugging information.

##### Implementation details

The following block diagram shows the internal functions and data structures of AMCLIB\_FWDebug\_F32.

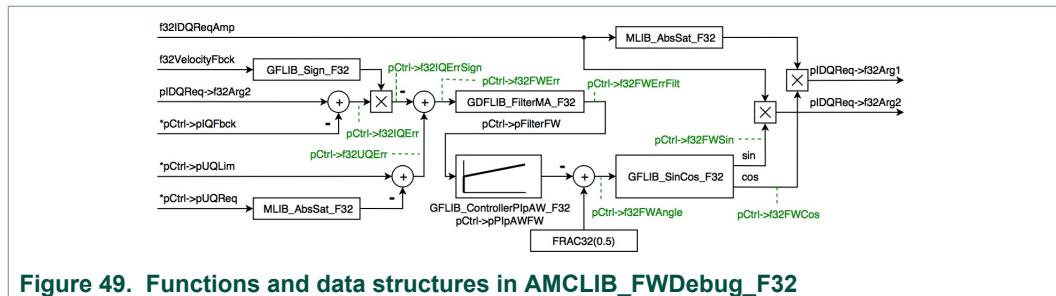


Figure 49. Functions and data structures in AMCLIB\_FWDebug\_F32

Refer to the description of [AMCLIB\\_FW\\_F32](#) function on how to set up the controller parameters.

**Note:** Due to effectivity reasons, this function is implemented using inline assembly and is therefore not ANSI-C compliant.

### Code Example

```
#include "amclib.h"

AMCLIB_FW_DEBUG_T_F32 FWState;
SWLIBS_2Syst_F32 f32IDQFbck; // required dq currents
SWLIBS_2Syst_F32 f32UDQReq; // required dq voltages
tFrac32 f32VelocityReq; // required velocity
tFrac32 f32VelocityFbck; // actual velocity
AMCLIB_CURRENT_LOOP_T_F32 CurrentLoop;

void main (void)
{
    tFrac32 f32FilterMAFWOut;
    tFrac32 f32ControllerPIpAWFWOut;

    // Initialize the parameters and pointers in FWState
    FW.pFilterFW.u16NSamples = 3u;
    FW.pPIpAWFW.f32PropGain = (tFrac32)FRAC32(0.2348);
    FW.pPIpAWFW.f32IntegGain = (tFrac32)FRAC32(0.3457);
    FW.pPIpAWFW.s16PropGainShift = 1;
    FW.pPIpAWFW.s16IntegGainShift = 1;
    FW.pPIpAWFW.f32LowerLimit = (tFrac32)-2012406926L;
    FW.pPIpAWFW.f32UpperLimit = (tFrac32)2068885746L;
    FW.pIQFbck = &f32IDQFbck.f32Arg2;
    FW.pUQReq = &f32UDQReq.f32Arg2;
    FW.pUQLim = &CurrentLoop.pPIrAWQ.f32UpperLimit;

    // Clear AMCLIB_FW state variables
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB_FWInit_F32((AMCLIB_FW_T_F32 *)&FWState);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB_FWInit((AMCLIB_FW_T_F32 *)&FWState, F32);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 32-bit fractional implementation is selected as default.
}
```

```
AMCLIB_FWInit((AMCLIB_FW_T_F32 *)&FWState);

// Initialize the AMCLIB_FW state variables to predefined values
// Warning: Parameters in FWState must be already initialized.
f32FilterMAFWOut = (tFrac32)123L;
f32ControllerPIpAWFWOut = (tFrac32)123L;
// Alternative 1: API call with postfix
// (only one alternative shall be used).
AMCLIB_FWSetState_F32(f32FilterMAFWOut, f32ControllerPIpAWFWOut,
                      (AMCLIB_FW_T_F32 *)&FWState);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
AMCLIB_FWSetState(f32FilterMAFWOut, f32ControllerPIpAWFWOut,
                   (AMCLIB_FW_T_F32 *)&FWState, F32);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 32-bit fractional implementation is selected as default.
AMCLIB_FWSetState(f32FilterMAFWOut, f32ControllerPIpAWFWOut,
                  (AMCLIB_FW_T_F32 *)&FWState);

f32VelocityReq = (tFrac32)100L;
while(1);
}

// Periodical function or interrupt - speed control loop
void SlowLoop(void)
{
    tFrac32 f32IDQReqAmp;

    // Calculate f32IDQReqAmp from f32VelocityReq and f32VelocityFbck
    // using AMCLIB_SpeedLoop
    // (...)

    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB_FWDebug_F32(f32IDQReqAmp, f32VelocityFbck, &IDQReq, &FWState);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB_FWDebug(f32IDQReqAmp, f32VelocityFbck, &IDQReq, &FWState, F32);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 32-bit fractional implementation is selected as default.
    AMCLIB_FWDebug(f32IDQReqAmp, f32VelocityFbck, &IDQReq, &FWState);
}

// Periodical function or interrupt - current control loop
void FastLoop(void)
{
    // Calculate new values of f32VelocityFbck, f32IDQFbck, f32UDQReq
    // (...)

}
```

### 2.4.6.2 Function AMCLIB\_FWDebug\_F16

#### Declaration

```
void AMCLIB_FWDebug_F16(tFrac16 f16IDQReqAmp, tFrac16
f16VelocityFbck, SWLIBS_2Syst_F16 *const pIDQReq,
AMCLIB_FW_DEBUG_T_F16 *pCtrl);
```

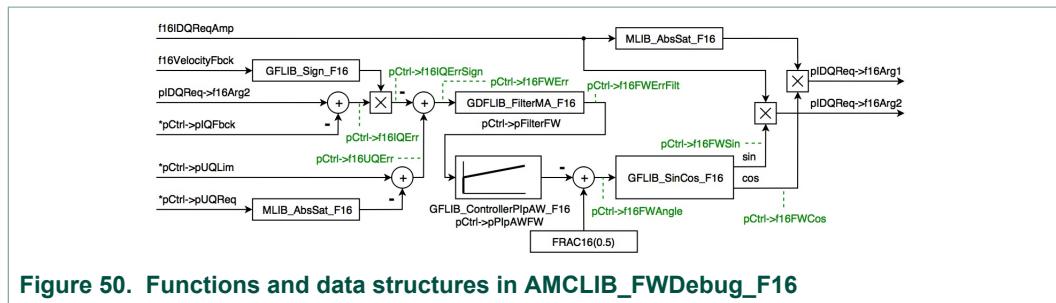
#### Arguments

**Table 37. AMCLIB\_FWDebug\_F16 arguments**

Type	Name	Direction	Description
tFrac16	f16IDQReqAmp	input	Required amplitude of the currents Id and Iq in the two-phase rotational orthogonal system (d-q).
tFrac16	f16VelocityFbck	input	Actual electrical angular velocity from the feedback.
SWLIBS_2Syst_F16 *const	pIDQReq	input, output	Pointer to the structure with the required stator currents in the two-phase rotational orthogonal system (d-q).
AMCLIB_FW_DEBUG_T_F16 *	pCtrl	input, output	Pointer to the structure with AMCLIB_FW state and debugging information.

#### Implementation details

The following block diagram shows the internal functions and data structures of AMCLIB\_FWDebug\_F16.



**Figure 50. Functions and data structures in AMCLIB\_FWDebug\_F16**

Refer to the description of [AMCLIB\\_FW\\_F16](#) function on how to set up the controller parameters.

**Note:** Due to effectivity reasons, this function is implemented using inline assembly and is therefore not ANSI-C compliant.

#### Code Example

```
#include "amclib.h"

AMCLIB_FW_DEBUG_T_F16 FWState;
SWLIBS_2Syst_F16 IDQReq; // required dq currents
SWLIBS_2Syst_F16 f16IDQFbck; // calculated dq currents from the feedback
SWLIBS_2Syst_F16 f16UDQReq; // required dq voltages
tFrac16 f16VelocityReq; // required velocity
tFrac16 f16VelocityFbck; // actual velocity
AMCLIB_CURRENT_LOOP_T_F16 CurrentLoop;

void main (void)
```

```
{  
    tFrac16 f16FilterMAFWOut;  
    tFrac16 f16ControllerPIpAWFWOut;  
  
    // Initialize the parameters and pointers in FWState  
    FW.pFilterFW.u16NSamples = 3u;  
    FW.pPIpAWFW.f16PropGain = (tFrac16)FRAC16(0.2348);  
    FW.pPIpAWFW.f16IntegGain = (tFrac16)FRAC16(0.3457);  
    FW.pPIpAWFW.s16PropGainShift = 1;  
    FW.pPIpAWFW.s16IntegGainShift = 1;  
    FW.pPIpAWFW.f16LowerLimit = (tFrac16)-30706;  
    FW.pPIpAWFW.f16UpperLimit = (tFrac16)31568;  
    FW.pIQFbck = &f16IDQFbck.f16Arg2;  
    FW.pUQReq = &f16UDQReq.f16Arg2;  
    FW.pUQLim = &CurrentLoop.pPIrAWQ.f16UpperLimit;  
  
    // Clear AMCLIB_FW state variables  
    // Alternative 1: API call with postfix  
    // (only one alternative shall be used).  
    AMCLIB_FWInit_F16((AMCLIB_FW_T_F16 *)&FWState);  
  
    // Alternative 2: API call with implementation parameter  
    // (only one alternative shall be used).  
    AMCLIB_FWInit((AMCLIB_FW_T_F16 *)&FWState, F16);  
  
    // Alternative 3: API call with global configuration of implementation  
    // (only one alternative shall be used). This alternative is available  
    // only if 16-bit fractional implementation is selected as default.  
    AMCLIB_FWInit((AMCLIB_FW_T_F16 *)&FWState);  
  
    // Initialize the AMCLIB_FW state variables to predefined values  
    // Warning: Parameters in FWState must be already initialized.  
    f16FilterMAFWOut = (tFrac16)123;  
    f16ControllerPIpAWFWOut = (tFrac16)123;  
    // Alternative 1: API call with postfix  
    // (only one alternative shall be used).  
    AMCLIB_FWSetState_F16(f16FilterMAFWOut, f16ControllerPIpAWFWOut,  
        (AMCLIB_FW_T_F16 *)&FWState);  
  
    // Alternative 2: API call with implementation parameter  
    // (only one alternative shall be used).  
    AMCLIB_FWSetState(f16FilterMAFWOut, f16ControllerPIpAWFWOut,  
        (AMCLIB_FW_T_F16 *)&FWState, F16);  
  
    // Alternative 3: API call with global configuration of implementation  
    // (only one alternative shall be used). This alternative is available  
    // only if 16-bit fractional implementation is selected as default.  
    AMCLIB_FWSetState(f16FilterMAFWOut, f16ControllerPIpAWFWOut,  
        (AMCLIB_FW_T_F16 *)&FWState);  
  
    f16VelocityReq = (tFrac16)100;  
    while(1);  
}  
  
// Periodical function or interrupt - speed control loop  
void SlowLoop(void)  
{  
    tFrac16 f16IDQReqAmp;  
  
    // Calculate f16IDQReqAmp from f16VelocityReq and f16VelocityFbck
```

```

// using AMCLIB_SpeedLoop
// (...)

// Alternative 1: API call with postfix
// (only one alternative shall be used).
AMCLIB_FWDebug_F16(f16IDQReqAmp, f16VelocityFbck, &IDQReq, &FWState);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
AMCLIB_FWDebug(f16IDQReqAmp, f16VelocityFbck, &IDQReq, &FWState, F16);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 16-bit fractional implementation is selected as default.
AMCLIB_FWDebug(f16IDQReqAmp, f16VelocityFbck, &IDQReq, &FWState);
}

// Periodical function or interrupt - current control loop
void FastLoop(void)
{
    // Calculate new values of f16VelocityFbck, f16IDQFbck, f16UDQReq
    // (...)

}

```

#### 2.4.6.3 Function AMCLIB\_FWDebug\_FLT

##### Declaration

```
void AMCLIB_FWDebug_FLT(tFloat fltIDQReqAmp, tFloat
fltVelocityFbck, SWLIBS_2Syst_FLT *const pIDQReq,
AMCLIB_FW_DEBUG_T_FLT *pCtrl);
```

##### Arguments

**Table 38. AMCLIB\_FWDebug\_FLT arguments**

Type	Name	Direction	Description
<b>tFloat</b>	fltIDQReqAmp	<b>input</b>	Required amplitude of the currents Id and Iq in the two-phase rotational orthogonal system (d-q).
<b>tFloat</b>	fltVelocityFbck	<b>input</b>	Actual electrical angular velocity from the feedback.
<b>SWLIBS_2Syst_FLT</b> *const	pIDQReq	<b>input, output</b>	Pointer to the structure with the required stator currents in the two-phase rotational orthogonal system (d-q).
<b>AMCLIB_FW_DEBUG_T_FLT</b> *	pCtrl	<b>input, output</b>	Pointer to the structure with AMCLIB_FW state and debugging information.

##### Implementation details

The following block diagram shows the internal functions and data structures of AMCLIB\_FWDebug\_FLT.

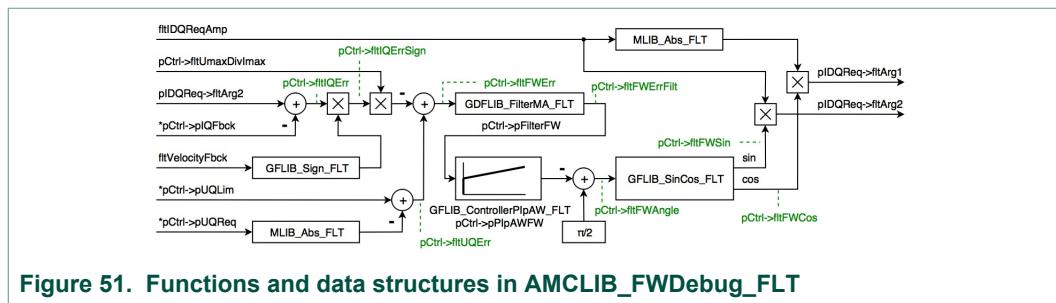


Figure 51. Functions and data structures in AMCLIB\_FWDebug\_FLT

Refer to the description of [AMCLIB\\_FW\\_FLT](#) function on how to set up the controller parameters.

**Note:** The function may raise floating-point exceptions (invalid operation, overflow, underflow, inexact, input denormal). The floating-point unit must be enabled in CPACR register to prevent NOCP UsageFault exception.

### Code Example

```
#include "amclib.h"

AMCLIB_FW_DEBUG_T_FLT FWState;
SWLIBS_2Syst_FLT IDQReq; // required dq currents
SWLIBS_2Syst_FLT fltIDQFbck; // calculated dq currents from the feedback
SWLIBS_2Syst_FLT fltUDQReq; // required dq voltages
tFloat fltVelocityReq; // required velocity
tFloat fltVelocityFbck; // actual velocity
AMCLIB_CURRENT_LOOP_T_FLT CurrentLoop;

void main (void)
{
    tFloat fltFilterMAFWOut;
    tFloat fltControllerPipAWFWOut;

    // Initialize the parameters and pointers in FWState
    FW.pFilterFW.fltLambda = (tFloat)0.6;
    FW.pPipAWFW.fltPropGain = (tFloat)0.2348;
    FW.pPipAWFW.fltIntegGain = (tFloat)0.3457;
    FW.pPipAWFW.fltLowerLimit = (tFloat)-0.937099993228912;
    FW.pPipAWFW.fltUpperLimit = (tFloat)0.963400006294251;
    FW.pIQFbck = &fltIDQFbck.fltArg2;
    FW.pUQReq = &fltUDQReq.fltArg2;
    FW.pQLim = &CurrentLoop.pPIrAWQ.fltUpperLimit;
    FW.fltUmaxDivImax = (tFloat)1.0;

    // Clear AMCLIB_FW state variables
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB_FWInit_FLT((AMCLIB_FW_T_FLT *)&FWState);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB_FWInit((AMCLIB_FW_T_FLT *)&FWState, FLT);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if single precision floating point implementation is selected
}
```

```
// as default.
AMCLIB_FWInit((AMCLIB_FW_T_FLT *)&FWState);

// Initialize the AMCLIB_FW state variables to predefined values
// Warning: Parameters in FWState must be already initialized.
fltFilterMAFWOut = 123.0F;
fltControllerPIpAWFWOut = 123.0F;
// Alternative 1: API call with postfix
// (only one alternative shall be used).
AMCLIB_FWSetState_FLT(fltFilterMAFWOut, fltControllerPIpAWFWOut,
(AMCLIB_FW_T_FLT *)&FWState);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
AMCLIB_FWSetState(fltFilterMAFWOut, fltControllerPIpAWFWOut,
(AMCLIB_FW_T_FLT *)&FWState, FLT);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if single precision floating point implementation is selected
// as default.
AMCLIB_FWSetState(fltFilterMAFWOut, fltControllerPIpAWFWOut,
(AMCLIB_FW_T_FLT *)&FWState);

fltVelocityReq = 100.0F;
while(1);
}

// Periodical function or interrupt - speed control loop
void SlowLoop(void)
{
    tFloat fltIDQReqAmp;

    // Calculate fltIDQReqAmp from fltVelocityReq and fltVelocityFbck
    // using AMCLIB_SpeedLoop
    // (...)

    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB_FWDebug_FLT(fltIDQReqAmp, fltVelocityFbck, &IDQReq, &FWState);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB_FWDebug(fltIDQReqAmp, fltVelocityFbck, &IDQReq, &FWState, FLT);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if single precision floating point implementation is selected
    // as default.
    AMCLIB_FWDebug(fltIDQReqAmp, fltVelocityFbck, &IDQReq, &FWState);
}

// Periodical function or interrupt - current control loop
void FastLoop(void)
{
    // Calculate new values of fltVelocityFbck, fltIDQFbck, fltUDQReq
    // (...)

}
```

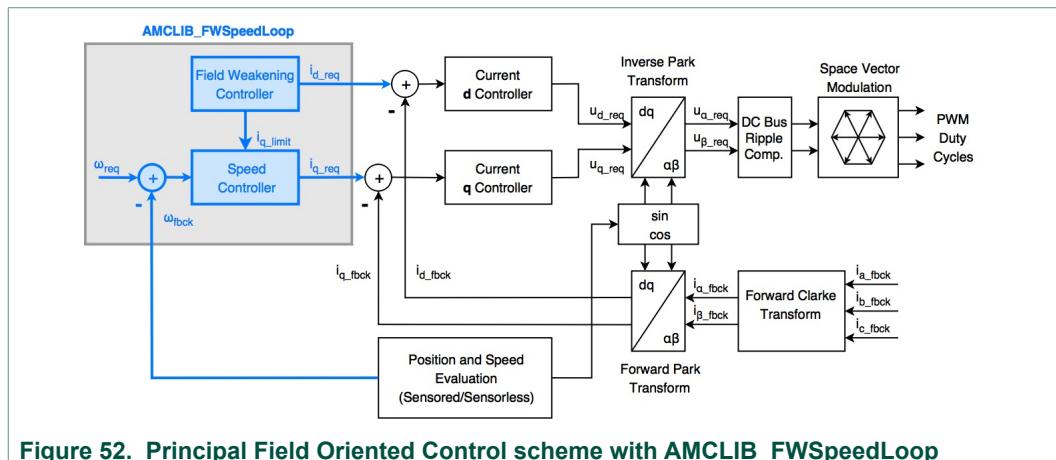
## 2.5 Function AMCLIB\_FWSpeedLoop

This function implements the speed PI controller in the FOC outer control loop and the field-weakening algorithm for permanent magnet synchronous motors.

## Description

This library function implements a portion of Field Oriented Control (FOC) algorithm. FOC (also called vector control) is a widely used control strategy for Permanent Magnet Synchronous Motors (PMSM). FOC is characterized by smooth rotation over the entire speed range of the motor, full torque control at zero speed, and fast acceleration/deceleration. FOC consists of a hierarchical cascade of inner current loop and outer speed loop. PI controllers within these closed loops maintain the required speed and torque based on feedback measurements.

AMCLIB\_FWSpeedLoop implements the speed PI controller and the field weakening controller in the outer control loop highlighted in [Figure 52](#). AMCLIB\_FWSpeedLoop combines the functionalities of [AMCLIB\\_FW](#) and [AMCLIB\\_SpeedLoop](#) in a more integrated form to simplify the application code and improve execution speed. AMCLIB\_FWSpeedLoop does not allow debugging of all internal variables. Use [AMCLIB\\_FWSpeedLoopDebug](#) for debugging purposes and replace it with AMCLIB\_FWSpeedLoop once the debugging is finished.



**Figure 52.** Principal Field Oriented Control scheme with AMCLIB FWSpeedLoop

Field weakening technique employed in AMCLIB\_FWSpeedLoop extends the speed range of PMSM beyond the motor base speed by reducing the linkage magnetic flux through negative current id req. See [AMCLIB\\_FW](#) for more details.

This algorithm is protected by US Patent No. US 2011/0050152 A1.

Before using the FOC with a particular motor, the user needs to provide a set of coefficients through the pCtrl input pointer. The controller coefficient values can be calculated from motor parameters.

Refer to the following resources to find out how the NXP motor control tuning and debugging tools for NXP microcontrollers can help you deploy the FOC in your application:

- [AN4642](#) - Motor Control Application Tuning (MCAT) Tool for 3-Phase PMSM
  - [FREEMASTER](#) - FreeMASTER Run-Time Debugging Tool

***Caution:***

1. A motor operated at speeds over the base speed region generates higher back-EMF voltage than the supply stator voltage. By applying the field weakening technique, the back-EMF is actively kept under control, i.e. lower than the stator voltage. It is dangerous to break the control loop during the field weakening operation. A loss of control (e.g. due to a fault state or turning the application off while running) may result in damage of the electronics and hardware due to the excessive back-EMF which would no longer be suppressed by the control algorithm.
2. The field is weakened by applying a negative current d-component. Too high negative current can cause magnet damage by its demagnetization. Make sure to set a correct lower limit of the field weakening PI controller to prevent damage to the motor.

**Note:** The input/output pointers must contain valid addresses, otherwise a fault may occur (MemManage, BusFault, UsageFault, HardFault).

### Re-entrancy

The function is re-entrant for a different pCtrl.

#### 2.5.1 Function AMCLIB\_FWSpeedLoop\_F32

##### Declaration

```
void AMCLIB_FWSpeedLoop_F32 (tFrac32 f32VelocityReq,
tFrac32 f32VelocityFbck, SWLIBS\_2Syst\_F32 *const pIDQReq,
AMCLIB\_FW\_SPEED\_LOOP\_T\_F32 *pCtrl);
```

##### Arguments

Table 39. AMCLIB\_FWSpeedLoop\_F32 arguments

Type	Name	Direction	Description
<a href="#">tFrac32</a>	f32VelocityReq	input	Required electrical angular velocity (setpoint).
<a href="#">tFrac32</a>	f32VelocityFbck	input	Actual electrical angular velocity from the feedback.
<a href="#">SWLIBS_2Syst_F32</a> *const	pIDQReq	input, output	Pointer to the structure with the required stator currents in the two-phase rotational orthogonal system (d-q).
<a href="#">AMCLIB_FW_SPEED_LOOP_T_F32</a> *	pCtrl	input, output	Pointer to the structure with AMCLIB_FWSpeedLoop state.

##### Implementation details

The following block diagram shows the internal functions and data structures of AMCLIB\_FWSpeedLoop\_F32.

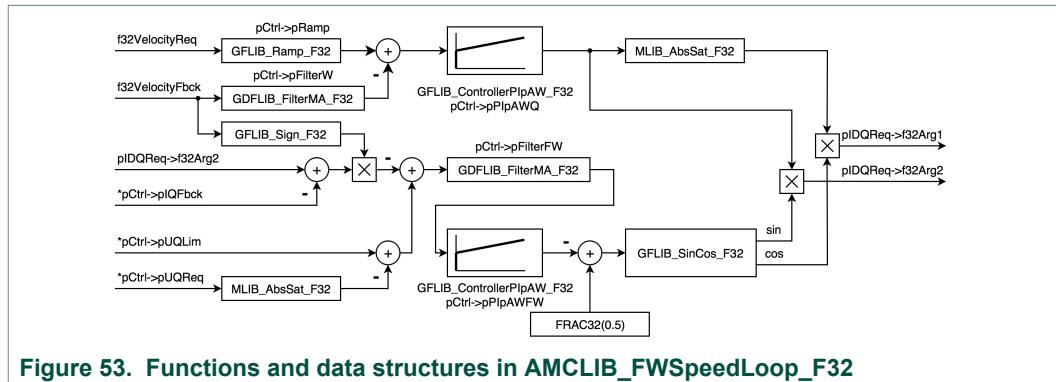


Figure 53. Functions and data structures in AMCLIB\_FWSpeedLoop\_F32

Prior to calculating the controller coefficients, it is necessary to set the scaling constants. All inputs and outputs of the algorithm are limited to the fractional range [-1, 1). Incorrect setting of the scaling constants may lead to undesirable overflow or saturation during the computation.

**Table 40. Scaling constants**

Scaling constant	Symbol	Calculation
Maximum stator phase voltage [V]	$U_{MAX}$	$U_{MAX} = U_{DC\_Bus\_Max}$
Maximum phase current [A]	$I_{MAX}$	Maximum current of the inverter or nominal current of the motor (whichever is lower).
Maximum speed [rad/s]	$\Omega_{MAX}$	Maximum application required speed, at least the motor electrical rated speed.

Parameters of the speed PIpAW controller (using bilinear transform) can be calculated using the following equations:

$$\begin{aligned} pPIpAWQ.f32PropGain &= FRAC32\left(\left(2 \cdot \xi \cdot \omega_0 \cdot \frac{J}{K_T}\right) \cdot \frac{\Omega_{MAX}}{I_{MAX}} \cdot 2^{NShiftP}\right) \\ pPIpAWQ.f32IntegGain &= FRAC32\left(\left(\omega_0^2 \cdot \frac{J}{K_T} \cdot \frac{T_s}{2}\right) \cdot \frac{\Omega_{MAX}}{I_{MAX}} \cdot 2^{NShiftI}\right) \\ pPIpAWQs16PropGainShift &= NShiftP \\ pPIpAWQs16IntegGainShift &= NShiftI \\ pPIpAWQ.f32UpperLimit &= FRAC32(1) \\ pPIpAWQ.f32LowerLimit &= FRAC32(-1) \end{aligned}$$

Equation AMCLIB\_FWSpeedLoop\_F32\_Eq1

where  $\xi$  is the speed loop attenuation,  $\omega_0$  is the speed loop natural frequency [rad/s],  $J$  is the moment of inertia,  $K_T$  is the motor torque constant, and  $T_s$  is the sampling period.  $NShiftP$  and  $NShiftI$  are integer values which ensure that the controller coefficients fit in the fractional range [-1, 1].

The smoothing factor of the [GDFLIB\\_FilterMA\\_F32](#) and the slope of the [GFLIB\\_Ramp\\_F32](#) on the input of the speed controller should reflect the achievable dynamics of the drive. The [AN4642](#) (Motor Control Application Tuning (MCAT) Tool for 3-Phase PMSM) can help with tuning of these parameters.

An initial estimate of the field weakening PIpAW controller parameters can be taken from the speed PIpAW controller as defined in the above [AMCLIB\\_FWSpeedLoop\\_F32\\_Eq1](#). The upper limit of the controller output shall be set to zero, i.e.  $pPIpAWFW.f32UpperLimit = 0$ . The lower limit shall be set to an adequate value from the interval  $<FRAC32(-0.5); 0>$ . It is important to set the lower limit correctly to prevent motor damage by irreversible demagnetization of the permanent magnets. Let  $I_{D\_MAX}$  be the maximum permitted negative d-axis current for field weakening, then

$$pPIpAWFW.f32LowerLimit = FRAC32\left(\frac{1}{\pi} \cdot \cos^{-1}\left(\frac{|I_{D\_MAX}|}{I_{MAX}}\right) - 0.5\right)$$

Equation AMCLIB\_FWSpeedLoop\_F32\_Eq2

Further refinement of the field weakening PIpAW controller parameters should be done interactively based on the observed performance. This can be easily achieved with the [FREEMASTER](#) real-time debugging tool. The following points should be considered:

- The field weakening PIpAW controller tweaks the angle of the current space vector  $I_S$  to produce the negative flux-producing current component  $I_D$  and sets the limits of the torque-producing current component  $I_Q$ .
- The magnitude of  $I_D$  depends also on the output of the speed PI controller.
- There are two different paths producing the PIpAW input error. The weight of each depends on the motor operation mode.

The performance of the field weakening controller can be affected by measurement noise. The noise immunity can be improved by tweaking the smoothing factor of the moving average filter [GDFLIB\\_FilterMA\\_F32](#) which preprocesses the input to the field weakening controller.

**Note:** Due to effectivity reasons, this function is implemented using inline assembly and is therefore not ANSI-C compliant.

### Code Example

```
#include "amclib.h"

AMCLIB_FW_SPEED_LOOP_T_F32 FWSpeedLoop;
SWLIBS_2Syst_F32 IDQReq;           // required dq currents
SWLIBS_2Syst_F32 f32IDQFbck;     // calculated dq currents from the feedback
SWLIBS_2Syst_F32 f32UDQReq;       // required dq voltages
tFrac32 f32VelocityReq;          // required velocity
tFrac32 f32VelocityFbck;         // actual velocity
AMCLIB_CURRENT_LOOP_T_F32 CurrentLoop;

void main (void)
{
    tFrac32 f32FilterMAWOut;
    tFrac32 f32FilterMAFWOut;
    tFrac32 f32ControllerPIpAWQOut;
    tFrac32 f32ControllerPIpAWFWOut;
    tFrac32 f32RampOut;

    // Initialize the parameters and pointers in FWSpeedLoop
    FWSpeedLoop.pFilterW.u16NSamples      = 3u;
    FWSpeedLoop.pFilterFW.u16NSamples      = 3u;
    FWSpeedLoop.pPIpAWQ.f32PropGain       = (tFrac32)FRAC32(0.1234);
    FWSpeedLoop.pPIpAWQ.f32IntegGain      = (tFrac32)FRAC32(0.1675);
    FWSpeedLoop.pPIpAWQ.s16PropGainShift  = 1;
    FWSpeedLoop.pPIpAWQ.s16IntegGainShift = 1;
    FWSpeedLoop.pPIpAWQ.f32LowerLimit    = (tFrac32)-2006823469L;
    FWSpeedLoop.pPIpAWQ.f32UpperLimit    = (tFrac32)2101527497L;
    FWSpeedLoop.pPIpAWFW.f32PropGain      = (tFrac32)FRAC32(0.2348);
    FWSpeedLoop.pPIpAWFW.f32IntegGain     = (tFrac32)FRAC32(0.3457);
    FWSpeedLoop.pPIpAWFW.s16PropGainShift = 1;
    FWSpeedLoop.pPIpAWFW.s16IntegGainShift = 1;
    FWSpeedLoop.pPIpAWFW.f32LowerLimit    = (tFrac32)-2012406926L;
    FWSpeedLoop.pPIpAWFW.f32UpperLimit    = (tFrac32)2068885746L;
    FWSpeedLoop.pRamp.f32RampUp          = (tFrac32)FRAC32(0.4768);
    FWSpeedLoop.pRamp.f32RampDown        = (tFrac32)FRAC32(0.3754);

    FWSpeedLoop.pIQFbck = &f32IDQFbck.f32Arg2;
    FWSpeedLoop.pUQReq  = &f32UDQReq.f32Arg2;
    FWSpeedLoop.pUQLim  = &CurrentLoop.pPIrAWQ.f32UpperLimit;

    // Clear AMCLIB_FWSpeedLoop state variables
    // Alternative 1: API call with postfix
```

```
// (only one alternative shall be used).
AMCLIB_FWSpeedLoopInit_F32(&FWSpeedLoop);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
AMCLIB_FWSpeedLoopInit(&FWSpeedLoop, F32);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 32-bit fractional implementation is selected as default.
AMCLIB_FWSpeedLoopInit(&FWSpeedLoop);

// Initialize the AMCLIB_FWSpeedLoop state variables to predefined values
// Warning: Parameters in FWSpeedLoop must be already initialized.
f32FilterMAWOut = (tFrac32)123L;
f32FilterMAFWOut = (tFrac32)123L;
f32ControllerPIpAWQOut = (tFrac32)123L;
f32ControllerPIpAWFWOut = (tFrac32)123L;
f32RampOut = (tFrac32)123L;
// Alternative 1: API call with postfix
// (only one alternative shall be used).
AMCLIB_FWSpeedLoopSetState_F32(f32FilterMAWOut, f32FilterMAFWOut,
    f32ControllerPIpAWQOut, f32ControllerPIpAWFWOut,
    f32RampOut, &FWSpeedLoop);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
AMCLIB_FWSpeedLoopSetState(f32FilterMAWOut, f32FilterMAFWOut,
    f32ControllerPIpAWQOut, f32ControllerPIpAWFWOut,
    f32RampOut, &FWSpeedLoop);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 32-bit fractional implementation is selected as default.
AMCLIB_FWSpeedLoopSetState(f32FilterMAWOut, f32FilterMAFWOut,
    f32ControllerPIpAWQOut, f32ControllerPIpAWFWOut,
    f32RampOut, &FWSpeedLoop);

f32VelocityReq = (tFrac32)100L;
while(1);
}

// Periodical function or interrupt - speed control loop
void SlowLoop(void)
{
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB_FWSpeedLoop_F32(f32VelocityReq, f32VelocityFbck, &IDQReq,
        &FWSpeedLoop);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB_FWSpeedLoop(f32VelocityReq, f32VelocityFbck,
        &IDQReq, &FWSpeedLoop, F32);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 32-bit fractional implementation is selected as default.
    AMCLIB_FWSpeedLoop(f32VelocityReq, f32VelocityFbck,
        &IDQReq, &FWSpeedLoop);
```

```

}

// Periodical function or interrupt - current control loop
void FastLoop(void)
{
    // Calculate new values of f32VelocityFbck, f32IDQFbck, f32UDQReq
    // (...)

}

```

## 2.5.2 Function AMCLIB\_FWSpeedLoop\_F16

### Declaration

```

void AMCLIB_FWSpeedLoop_F16(tFrac16 f16VelocityReq,
tFrac16 f16VelocityFbck, SWLIBS_2Syst_F16 *const pIDQReq,
AMCLIB_FW_SPEED_LOOP_T_F16 *pCtrl);

```

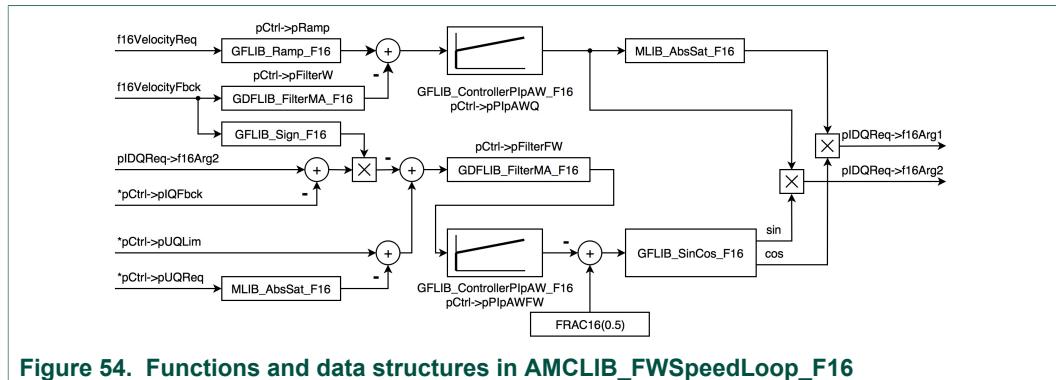
### Arguments

**Table 41. AMCLIB\_FWSpeedLoop\_F16 arguments**

Type	Name	Direction	Description
<b>tFrac16</b>	f16VelocityReq	input	Required electrical angular velocity (setpoint).
<b>tFrac16</b>	f16VelocityFbck	input	Actual electrical angular velocity from the feedback.
<b>SWLIBS_2Syst_F16</b> *const	pIDQReq	input, output	Pointer to the structure with the required stator currents in the two-phase rotational orthogonal system (d-q).
<b>AMCLIB_FW_SPEED_LOOP_T_F16</b> *	pCtrl	input, output	Pointer to the structure with AMCLIB_FWSpeedLoop state.

### Implementation details

The following block diagram shows the internal functions and data structures of AMCLIB\_FWSpeedLoop\_F16.



**Figure 54. Functions and data structures in AMCLIB\_FWSpeedLoop\_F16**

Prior to calculating the controller coefficients, it is necessary to set the scaling constants. All inputs and outputs of the algorithm are limited to the fractional range [-1, 1]. Incorrect setting of the scaling constants may lead to undesirable overflow or saturation during the computation.

**Table 42. Scaling constants**

Scaling constant	Symbol	Calculation
Maximum stator phase voltage [V]	$U_{MAX}$	$U_{MAX} = U_{DC\_Bus\_Max}$

Maximum phase current [A]	$I_{MAX}$	Maximum current of the inverter or nominal current of the motor (whichever is lower).
Maximum speed [rad/s]	$\Omega_{MAX}$	Maximum application required speed, at least the motor electrical rated speed.

Parameters of the speed PIpAW controller (using bilinear transform) can be calculated using the following equations:

$$\begin{aligned} pPIpAWQ.f16PropGain &= FRAC16\left(\left(2 \cdot \xi \cdot \omega_0 \cdot \frac{J}{K_T}\right) \cdot \frac{\Omega_{MAX}}{I_{MAX}} \cdot 2^{NShiftP}\right) \\ pPIpAWQ.f16IntegGain &= FRAC16\left(\left(\omega_0^2 \cdot \frac{J}{K_T} \cdot \frac{T_s}{2}\right) \cdot \frac{\Omega_{MAX}}{I_{MAX}} \cdot 2^{NShiftI}\right) \\ pPIpAWQs16PropGainShift &= NShiftP \\ pPIpAWQs16IntegGainShift &= NShiftI \\ pPIpAWQ.f16UpperLimit &= FRAC16(1) \\ pPIpAWQ.f16LowerLimit &= FRAC16(-1) \end{aligned}$$

Equation AMCLIB\_FWSpeedLoop\_F16\_Eq1

where  $\xi$  is the speed loop attenuation,  $\omega_0$  is the speed loop natural frequency [rad/s],  $J$  is the moment of inertia,  $K_T$  is the motor torque constant, and  $T_s$  is the sampling period.  $NShiftP$  and  $NShiftI$  are integer values which ensure that the controller coefficients fit in the fractional range [-1, 1].

The smoothing factor of the [GDFLIB\\_FilterMA\\_F16](#) and the slope of the [GFLIB\\_Ramp\\_F16](#) on the input of the speed controller should reflect the achievable dynamics of the drive. The [AN4642](#) (Motor Control Application Tuning (MCAT) Tool for 3-Phase PMSM) can help with tuning of these parameters.

An initial estimate of the field weakening PIpAW controller parameters can be taken from the speed PIpAW controller as defined in the above [AMCLIB\\_FWSpeedLoop\\_F16\\_Eq1](#). The upper limit of the controller output shall be set to zero, i.e.  $pPIpAWFW.f16UpperLimit = 0$ . The lower limit shall be set to an adequate value from the interval  $<FRAC16(-0.5); 0>$ . It is important to set the lower limit correctly to prevent motor damage by irreversible demagnetization of the permanent magnets. Let  $I_{D\_MAX}$  be the maximum permitted negative d-axis current for field weakening, then

$$pPIpAWFW.f16LowerLimit = FRAC16\left(\frac{1}{\pi} \cdot \cos^{-1}\left(\frac{|I_{D\_MAX}|}{I_{MAX}}\right) - 0.5\right)$$

Equation AMCLIB\_FWSpeedLoop\_F16\_Eq2

Further refinement of the field weakening PIpAW controller parameters should be done interactively based on the observed performance. This can be easily achieved with the [FREEMASTER](#) real-time debugging tool. The following points should be considered:

- The field weakening PIpAW controller tweaks the angle of the current space vector  $I_S$  to produce the negative flux-producing current component  $I_D$  and sets the limits of the torque-producing current component  $I_Q$ .
- The magnitude of  $I_D$  depends also on the output of the speed PI controller.
- There are two different paths producing the PIpAW input error. The weight of each depends on the motor operation mode.

The performance of the field weakening controller can be affected by measurement noise. The noise immunity can be improved by tweaking the smoothing factor of the

moving average filter [GDFLIB\\_FilterMA\\_F16](#) which preprocesses the input to the field weakening controller.

**Note:** Due to effectivity reasons, this function is implemented using inline assembly and is therefore not ANSI-C compliant.

### Code Example

```
#include "amclib.h"

AMCLIB_FW_SPEED_LOOP_T_F16 FWSpeedLoop;
SWLIBS_2Syst_F16 IDQReq;           // required dq currents
SWLIBS_2Syst_F16 f16IDQFbck;     // calculated dq currents from the feedback
SWLIBS_2Syst_F16 f16UDQReq;       // required dq voltages
tFrac16 f16VelocityReq;          // required velocity
tFrac16 f16VelocityFbck;         // actual velocity
AMCLIB_CURRENT_LOOP_T_F16 CurrentLoop;

void main (void)
{
    tFrac16 f16FilterMAWOut;
    tFrac16 f16FilterMAFWOut;
    tFrac16 f16ControllerPIpAWQOut;
    tFrac16 f16ControllerPIpAWFWOut;
    tFrac32 f32RampOut;

    // Initialize the parameters and pointers in FWSpeedLoop
    FWSpeedLoop.pFilterW.ul6NSamples      = 3u;
    FWSpeedLoop.pFilterFW.ul6NSamples      = 3u;
    FWSpeedLoop.pPIpAWQ.f16PropGain        = (tFrac16)FRAC16(0.1234);
    FWSpeedLoop.pPIpAWQ.f16IntegGain       = (tFrac16)FRAC16(0.1675);
    FWSpeedLoop.pPIpAWQ.s16PropGainShift   = 1;
    FWSpeedLoop.pPIpAWQ.s16IntegGainShift   = 1;
    FWSpeedLoop.pPIpAWQ.f16LowerLimit      = (tFrac16)-30621;
    FWSpeedLoop.pPIpAWQ.f16UpperLimit      = (tFrac16)32066;
    FWSpeedLoop.pPIpAWFW.f16PropGain        = (tFrac16)FRAC16(0.2348);
    FWSpeedLoop.pPIpAWFW.f16IntegGain       = (tFrac16)FRAC16(0.3457);
    FWSpeedLoop.pPIpAWFW.s16PropGainShift   = 1;
    FWSpeedLoop.pPIpAWFW.s16IntegGainShift   = 1;
    FWSpeedLoop.pPIpAWFW.f16LowerLimit      = (tFrac16)-30706;
    FWSpeedLoop.pPIpAWFW.f16UpperLimit      = (tFrac16)31568;
    FWSpeedLoop.pRamp.f32RampUp            = (tFrac32)FRAC32(0.4768);
    FWSpeedLoop.pRamp.f32RampDown          = (tFrac32)FRAC32(0.3754);
    FWSpeedLoop.pIQFbck = &f16IDQFbck.f16Arg2;
    FWSpeedLoop.pUQReq  = &f16UDQReq.f16Arg2;
    FWSpeedLoop.pUQLim  = &CurrentLoop.pPIrAWQ.f16UpperLimit;

    // Clear AMCLIB_FWSpeedLoop state variables
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB_FWSpeedLoopInit_F16(&FWSpeedLoop);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB_FWSpeedLoopInit(&FWSpeedLoop, F16);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
```

```
// only if 16-bit fractional implementation is selected as default.  
AMCLIB_FWSpeedLoopInit(&FWSpeedLoop);  
  
// Initialize the AMCLIB_FWSpeedLoop state variables to predefined values  
// Warning: Parameters in FWSpeedLoop must be already initialized.  
f16FilterMAWOut = (tFrac16)123;  
f16FilterMAFWOut = (tFrac16)123;  
f16ControllerPIpAWQOut = (tFrac16)123;  
f16ControllerPIpAWFWOut = (tFrac16)123;  
f32RampOut = (tFrac32)123;  
// Alternative 1: API call with postfix  
// (only one alternative shall be used).  
AMCLIB_FWSpeedLoopSetState_F16(f16FilterMAWOut, f16FilterMAFWOut,  
    f16ControllerPIpAWQOut, f16ControllerPIpAWFWOut,  
    f32RampOut, &FWSpeedLoop);  
  
// Alternative 2: API call with implementation parameter  
// (only one alternative shall be used).  
AMCLIB_FWSpeedLoopSetState(f16FilterMAWOut, f16FilterMAFWOut,  
    f16ControllerPIpAWQOut, f16ControllerPIpAWFWOut,  
    f32RampOut, &FWSpeedLoop, F16);  
  
// Alternative 3: API call with global configuration of implementation  
// (only one alternative shall be used). This alternative is available  
// only if 16-bit fractional implementation is selected as default.  
AMCLIB_FWSpeedLoopSetState(f16FilterMAWOut, f16FilterMAFWOut,  
    f16ControllerPIpAWQOut, f16ControllerPIpAWFWOut,  
    f32RampOut, &FWSpeedLoop);  
  
f16VelocityReq = (tFrac16)100;  
while(1);  
}  
  
// Periodical function or interrupt - speed control loop  
void SlowLoop(void)  
{  
    // Alternative 1: API call with postfix  
    // (only one alternative shall be used).  
    AMCLIB_FWSpeedLoop_F16(f16VelocityReq, f16VelocityFbck,  
        &IDQReq, &FWSpeedLoop);  
  
    // Alternative 2: API call with implementation parameter  
    // (only one alternative shall be used).  
    AMCLIB_FWSpeedLoop(f16VelocityReq, f16VelocityFbck,  
        &IDQReq, &FWSpeedLoop, F16);  
  
    // Alternative 3: API call with global configuration of implementation  
    // (only one alternative shall be used). This alternative is available  
    // only if 16-bit fractional implementation is selected as default.  
    AMCLIB_FWSpeedLoop(f16VelocityReq, f16VelocityFbck,  
        &IDQReq, &FWSpeedLoop);  
}  
  
// Periodical function or interrupt - current control loop  
void FastLoop(void)  
{  
    // Calculate new values of f16VelocityFbck, f16IDQFbck, f16UDQReq  
    // (...)  
}
```

### 2.5.3 Function AMCLIB\_FWSpeedLoop\_FLT

#### Declaration

```
void AMCLIB_FWSpeedLoop_FLT(tFloat fltVelocityReq,
tFloat fltVelocityFbck, SWLIBS_2Syst_FLT *const pIDQReq,
AMCLIB_FW_SPEED_LOOP_T_FLT *pCtrl);
```

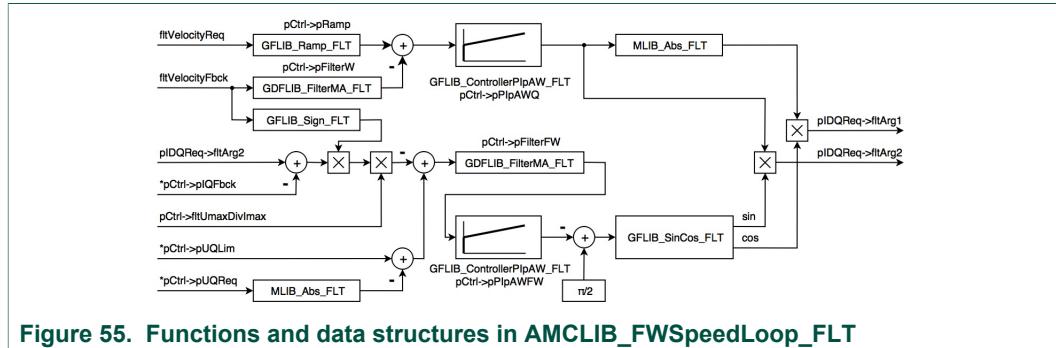
#### Arguments

**Table 43. AMCLIB\_FWSpeedLoop\_FLT arguments**

Type	Name	Direction	Description
<b>tFloat</b>	fltVelocityReq	input	Required electrical angular velocity (setpoint).
<b>tFloat</b>	fltVelocityFbck	input	Actual electrical angular velocity from the feedback.
<b>SWLIBS_2Syst_FLT</b> *const	pIDQReq	input, output	Pointer to the structure with the required stator currents in the two-phase rotational orthogonal system (d-q).
<b>AMCLIB_FW_SPEED_LOOP_T_FLT</b> *	pCtrl	input, output	Pointer to the structure with AMCLIB_FWSpeedLoop state.

#### Implementation details

The following block diagram shows the internal functions and data structures of AMCLIB\_FWSpeedLoop\_FLT.



**Figure 55. Functions and data structures in AMCLIB\_FWSpeedLoop\_FLT**

Parameters of the speed PlpAW controller (using bilinear transform) can be calculated using the following equations:

$$pPipAWQ.fltPropGain = 2 \cdot \xi \cdot \omega_0 \cdot \frac{J}{K_T}$$

$$pPipAWQ.fltIntegGain = \omega_0^2 \cdot \frac{J}{K_T} \cdot \frac{T_S}{2}$$

$$pPipAWQ.fltUpperLimit = I_{MAX}$$

$$pPipAWQ.fltLowerLimit = -I_{MAX}$$

Equation AMCLIB\_FWSpeedLoop\_FLT\_Eq1

where  $\xi$  is the speed loop attenuation,  $\omega_0$  is the speed loop natural frequency [rad/s],  $J$  is the moment of inertia,  $K_T$  is the motor torque constant,  $T_S$  is the sampling period, and  $I_{MAX}$  is the maximum phase current [A].

The smoothing factor of the **GDFLIB\_FilterMA\_FLT** and the slope of the **GFLIB\_Ramp\_FLT** on the input of the speed controller should reflect the achievable dynamics of the drive. The [AN4642](#) (Motor Control Application Tuning (MCAT) Tool for 3-Phase PMSM) can help with tuning of these parameters.

An initial estimate of the field weakening PIpAW controller parameters can be taken from the speed PIpAW controller as defined in the above [AMCLIB\\_FWSpeedLoop\\_FLT\\_Eq1](#). The upper limit of the controller output shall be set to zero, i.e. `pPIpAWFW.fltUpperLimit = 0`. The lower limit shall be set to an adequate value from the interval  $<-\pi/2; 0>$ . It is important to set the lower limit correctly to prevent motor damage by irreversible demagnetization of the permanent magnets. Let  $I_{D\_MAX}$  be the maximum permitted negative d-axis current for field weakening, then

$$pPIpAWFW.fltLowerLimit = \cos^{-1}\left(\frac{|I_{D\_MAX}|}{I_{MAX}}\right) - \frac{\pi}{2}$$

Equation AMCLIB\_FWSpeedLoop\_FLT\_Eq2

Further refinement of the field weakening PIpAW controller parameters should be done interactively based on the observed performance. This can be easily achieved with the [FREEMASTER](#) real-time debugging tool. The following points should be considered:

- The field weakening PIpAW controller tweaks the angle of the current space vector  $I_S$  to produce the negative flux-producing current component  $I_D$  and sets the limits of the torque-producing current component  $I_Q$ .
- The magnitude of  $I_D$  depends also on the output of the speed PI controller.
- There are two different paths producing the PIpAW input error. The weight of each depends on the motor operation mode.

The performance of the field weakening controller can be affected by measurement noise. The noise immunity can be improved by tweaking the smoothing factor of the moving average filter [GDFLIB\\_FilterMA\\_FLT](#) which preprocesses the input to the field weakening controller.

**Note:** *The function may raise floating-point exceptions (invalid operation, overflow, underflow, inexact, input denormal). The floating-point unit must be enabled in CPACR register to prevent NOCP UsageFault exception.*

### Code Example

```
#include "amclib.h"

AMCLIB_FW_SPEED_LOOP_T_FLT FWSpeedLoop;
SWLIBS_2Syst_FLT IDQReq;           // required dq currents
SWLIBS_2Syst_FLT fltIDQFbck;      // calculated dq currents from the feedback
SWLIBS_2Syst_FLT fltUDQReq;        // required dq voltages
tFloat fltVelocityReq;             // required velocity
tFloat fltVelocityFbck;            // actual velocity
AMCLIB_CURRENT_LOOP_T_FLT CurrentLoop;

void main (void)
{
    tFloat fltFilterMAWOut;
    tFloat fltFilterMAFWOut;
    tFloat fltControllerPIpAWQOut;
    tFloat fltControllerPIpAWFWOut;
    tFloat fltRampOut;

    // Initialize the parameters and pointers in FWSpeedLoop
    FWSpeedLoop.pFilterW.fltLambda     = (tFloat) 0.5;
    FWSpeedLoop.pFilterFW.fltLambda     = (tFloat) 0.6;
    FWSpeedLoop.pPIpAWQ.fltPropGain    = (tFloat) 0.1234;
```

```
FWSpeedLoop.pPIpAWQ.fltIntegGain      = (tFloat) 0.1675;
FWSpeedLoop.pPIpAWQ.fltLowerLimit     = (tFloat) -0.934499979019165;
FWSpeedLoop.pPIpAWQ.fltUpperLimit     = (tFloat) 0.978600025177002;
FWSpeedLoop.pPIpAWFW.fltPropGain      = (tFloat) 0.2348;
FWSpeedLoop.pPIpAWFW.fltIntegGain     = (tFloat) 0.3457;
FWSpeedLoop.pPIpAWFW.fltLowerLimit     = (tFloat) -0.937099993228912;
FWSpeedLoop.pPIpAWFW.fltUpperLimit     = (tFloat) 0.963400006294251;
FWSpeedLoop.pRamp.fltRampUp          = (tFloat) 0.4768;
FWSpeedLoop.pRamp.fltRampDown         = (tFloat) 0.3754;
FWSpeedLoop.pIQFbck = &fltIDQFbck.fltArg2;
FWSpeedLoop.pUQReq  = &fltUDQReq.fltArg2;
FWSpeedLoop.pUQLim  = &CurrentLoop.pPIrAWQ.fltUpperLimit;
FWSpeedLoop.fltUmaxDivImax          = (tFloat) 1.0;

// Clear AMCLIB_FWSpeedLoop state variables
// Alternative 1: API call with postfix
// (only one alternative shall be used).
AMCLIB_FWSpeedLoopInit_FLT(&FWSpeedLoop);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
AMCLIB_FWSpeedLoopInit(&FWSpeedLoop, FLT);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if single precision floating point implementation is selected
// as default.
AMCLIB_FWSpeedLoopInit(&FWSpeedLoop);

// Initialize the AMCLIB_FWSpeedLoop state variables to predefined values
// Warning: Parameters in FWSpeedLoop must be already initialized.
fltFilterMAWOut = 123.0F;
fltFilterMAFWOut = 123.0F;
fltControllerPIpAWQOut = 123.0F;
fltControllerPIpAWFWOut = 123.0F;
fltRampOut = 123.0F;
// Alternative 1: API call with postfix
// (only one alternative shall be used).
AMCLIB_FWSpeedLoopSetState_FLT(fltFilterMAWOut, fltFilterMAFWOut,
    fltControllerPIpAWQOut, fltControllerPIpAWFWOut,
    fltRampOut, &FWSpeedLoop);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
AMCLIB_FWSpeedLoopSetState(fltFilterMAWOut, fltFilterMAFWOut,
    fltControllerPIpAWQOut, fltControllerPIpAWFWOut,
    fltRampOut, &FWSpeedLoop, FLT);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if single precision floating point implementation is selected
// as default.
AMCLIB_FWSpeedLoopSetState(fltFilterMAWOut, fltFilterMAFWOut,
    fltControllerPIpAWQOut, fltControllerPIpAWFWOut,
    fltRampOut, &FWSpeedLoop);

fltVelocityReq = 100.0F;
while(1);
}
```

```

// Periodical function or interrupt - speed control loop
void SlowLoop(void)
{
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB_FWSpeedLoop_FLT(fltVelocityReq, fltVelocityFbck,
                           &IDQReq, &FWSpeedLoop);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB_FWSpeedLoop(fltVelocityReq, fltVelocityFbck,
                        &IDQReq, &FWSpeedLoop, FLT);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if single precision floating point implementation is selected
    // as default.
    AMCLIB_FWSpeedLoop(fltVelocityReq, fltVelocityFbck,
                        &IDQReq, &FWSpeedLoop);
}

// Periodical function or interrupt - current control loop
void FastLoop(void)
{
    // Calculate new values of fltVelocityFbck, fltIDQFbck, fltUDQReq
    // (...)

}

```

## 2.5.4 Function AMCLIB\_FWSpeedLoopInit

### Description

This function clears the AMCLIB\_FWSpeedLoop state variables.

**Note:** The input/output pointers must contain valid addresses, otherwise a fault may occur (MemManage, BusFault, UsageFault, HardFault).

### Re-entrancy

The function is re-entrant for a different pCtrl.

#### 2.5.4.1 Function AMCLIB\_FWSpeedLoopInit\_F32

##### Declaration

```
void AMCLIB_FWSpeedLoopInit_F32(AMCLIB\_FW\_SPEED\_LOOP\_T\_F32 *const pCtrl);
```

##### Arguments

Table 44. AMCLIB\_FWSpeedLoopInit\_F32 arguments

Type	Name	Direction	Description
<a href="#">AMCLIB_FW_SPEED_LOOP_T_F32</a> *const	pCtrl	input, output	Pointer to the structure with AMCLIB_FWSpeedLoop state.

**Note:** If pCtrl points to a structure of type [AMCLIB\\_FW\\_SPEED\\_LOOP\\_DEBUG\\_T\\_F32](#), it must be recasted to [AMCLIB\\_FW\\_SPEED\\_LOOP\\_T\\_F32](#).\*

## Code Example

```
#include "amclib.h"

AMCLIB_FW_SPEED_LOOP_T_F32 FWSpeedLoop;
SWLIBS_2Syst_F32 IDQReq;           // required dq currents
SWLIBS_2Syst_F32 f32IDQFbck;     // calculated dq currents from the feedback
SWLIBS_2Syst_F32 f32UDQReq;       // required dq voltages
tFrac32 f32VelocityReq;          // required velocity
tFrac32 f32VelocityFbck;         // actual velocity
AMCLIB_CURRENT_LOOP_T_F32 CurrentLoop;

void main (void)
{
    tFrac32 f32FilterMAWOut;
    tFrac32 f32FilterMAFWOut;
    tFrac32 f32ControllerPIpAWQOut;
    tFrac32 f32ControllerPIpAWFWOut;
    tFrac32 f32RampOut;

    // Initialize the parameters and pointers in FWSpeedLoop
    FWSpeedLoop.pFilterW.u16NSamples      = 3u;
    FWSpeedLoop.pFilterFW.u16NSamples      = 3u;
    FWSpeedLoop.pPIpAWQ.f32PropGain       = (tFrac32)FRAC32(0.1234);
    FWSpeedLoop.pPIpAWQ.f32IntegGain      = (tFrac32)FRAC32(0.1675);
    FWSpeedLoop.pPIpAWQ.s16PropGainShift  = 1;
    FWSpeedLoop.pPIpAWQ.s16IntegGainShift = 1;
    FWSpeedLoop.pPIpAWQ.f32LowerLimit    = (tFrac32)-2006823469L;
    FWSpeedLoop.pPIpAWQ.f32UpperLimit    = (tFrac32)2101527497L;
    FWSpeedLoop.pPIpAWFW.f32PropGain      = (tFrac32)FRAC32(0.2348);
    FWSpeedLoop.pPIpAWFW.f32IntegGain     = (tFrac32)FRAC32(0.3457);
    FWSpeedLoop.pPIpAWFW.s16PropGainShift = 1;
    FWSpeedLoop.pPIpAWFW.s16IntegGainShift = 1;
    FWSpeedLoop.pPIpAWFW.f32LowerLimit   = (tFrac32)-2012406926L;
    FWSpeedLoop.pPIpAWFW.f32UpperLimit   = (tFrac32)2068885746L;
    FWSpeedLoop.pRamp.f32RampUp          = (tFrac32)FRAC32(0.4768);
    FWSpeedLoop.pRamp.f32RampDown        = (tFrac32)FRAC32(0.3754);
    FWSpeedLoop.pIQFbck = &f32IDQFbck.f32Arg2;
    FWSpeedLoop.pUQReq   = &f32UDQReq.f32Arg2;
    FWSpeedLoop.pUQLim   = &CurrentLoop.pPIrAWQ.f32UpperLimit;

    // Clear AMCLIB_FWSpeedLoop state variables
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB_FWSpeedLoopInit_F32(&FWSpeedLoop);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB_FWSpeedLoopInit(&FWSpeedLoop, F32);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 32-bit fractional implementation is selected as default.
    AMCLIB_FWSpeedLoopInit(&FWSpeedLoop);

    // Initialize the AMCLIB_FWSpeedLoop state variables to predefined values
    // Warning: Parameters in FWSpeedLoop must be already initialized.
    f32FilterMAWOut = (tFrac32)123L;
    f32FilterMAFWOut = (tFrac32)123L;
```

```
f32ControllerPIpAWQOut = (tFrac32)123L;
f32ControllerPIpAWFWOut = (tFrac32)123L;
f32RampOut = (tFrac32)123L;
// Alternative 1: API call with postfix
// (only one alternative shall be used).
AMCLIB\_FWSpeedLoopSetState\_F32(f32FilterMAWOut, f32FilterMAFWOut,
    f32ControllerPIpAWQOut, f32ControllerPIpAWFWOut,
    f32RampOut, &FWSpeedLoop);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
AMCLIB\_FWSpeedLoopSetState(f32FilterMAWOut, f32FilterMAFWOut,
    f32ControllerPIpAWQOut, f32ControllerPIpAWFWOut,
    f32RampOut, &FWSpeedLoop, F32);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 32-bit fractional implementation is selected as default.
AMCLIB\_FWSpeedLoopSetState(f32FilterMAWOut, f32FilterMAFWOut,
    f32ControllerPIpAWQOut, f32ControllerPIpAWFWOut,
    f32RampOut, &FWSpeedLoop);

f32VelocityReq = (tFrac32)100L;
while(1);
}

// Periodical function or interrupt - speed control loop
void SlowLoop(void)
{
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB\_FWSpeedLoop\_F32(f32VelocityReq, f32VelocityFbck, &IDQReq,
        &FWSpeedLoop);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB\_FWSpeedLoop(f32VelocityReq, f32VelocityFbck,
        &IDQReq, &FWSpeedLoop, F32);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 32-bit fractional implementation is selected as default.
    AMCLIB\_FWSpeedLoop(f32VelocityReq, f32VelocityFbck,
        &IDQReq, &FWSpeedLoop);
}

// Periodical function or interrupt - current control loop
void FastLoop(void)
{
    // Calculate new values of f32VelocityFbck, f32IDQFbck, f32UDQReq
    // (...)
```

### 2.5.4.2 Function [AMCLIB\\_FWSpeedLoopInit\\_F16](#)

#### Declaration

```
void AMCLIB_FWSpeedLoopInit_F16(AMCLIB\_FW\_SPEED\_LOOP\_T\_F16 *const
    pCtrl);
```

## Arguments

**Table 45. AMCLIB\_FWSpeedLoopInit\_F16 arguments**

Type	Name	Direction	Description
<a href="#">AMCLIB_FW_SPEED_LOOP_T_F16</a> *const	pCtrl	input, output	Pointer to the structure with AMCLIB_FWSpeedLoop state.

**Note:** If pCtrl points to a structure of type [AMCLIB\\_FW\\_SPEED\\_LOOP\\_DEBUG\\_T\\_F16](#), it must be recasted to [AMCLIB\\_FW\\_SPEED\\_LOOP\\_T\\_F16](#).

## Code Example

```
#include "amclib.h"

AMCLIB_FW_SPEED_LOOP_T_F16 FWSpeedLoop;
SWLIBS_2Syst_F16 IDQReq;           // required dq currents
SWLIBS_2Syst_F16 f16IDQFbck;      // calculated dq currents from the feedback
SWLIBS_2Syst_F16 f16UDQReq;       // required dq voltages
tFrac16 f16VelocityReq;          // required velocity
tFrac16 f16VelocityFbck;         // actual velocity
AMCLIB_CURRENT_LOOP_T_F16 CurrentLoop;

void main (void)
{
    tFrac16 f16FilterMAWOut;
    tFrac16 f16FilterMAFWOut;
    tFrac16 f16ControllerPIpAWQOut;
    tFrac16 f16ControllerPIpAWFWOut;
    tFrac32 f32RampOut;

    // Initialize the parameters and pointers in FWSpeedLoop
    FWSpeedLoop.pFilterW.u16NSamples      = 3u;
    FWSpeedLoop.pFilterFW.u16NSamples      = 3u;
    FWSpeedLoop.pPIpAWQ.f16PropGain       = (tFrac16)FRAC16(0.1234);
    FWSpeedLoop.pPIpAWQ.f16IntegGain      = (tFrac16)FRAC16(0.1675);
    FWSpeedLoop.pPIpAWQ.s16PropGainShift   = 1;
    FWSpeedLoop.pPIpAWQ.s16IntegGainShift  = 1;
    FWSpeedLoop.pPIpAWQ.f16LowerLimit     = (tFrac16)-30621;
    FWSpeedLoop.pPIpAWQ.f16UpperLimit     = (tFrac16)32066;
    FWSpeedLoop.pPIpAWFW.f16PropGain       = (tFrac16)FRAC16(0.2348);
    FWSpeedLoop.pPIpAWFW.f16IntegGain      = (tFrac16)FRAC16(0.3457);
    FWSpeedLoop.pPIpAWFW.s16PropGainShift   = 1;
    FWSpeedLoop.pPIpAWFW.s16IntegGainShift  = 1;
    FWSpeedLoop.pPIpAWFW.f16LowerLimit     = (tFrac16)-30706;
    FWSpeedLoop.pPIpAWFW.f16UpperLimit     = (tFrac16)31568;
    FWSpeedLoop.pRamp.f32RampUp           = (tFrac32)FRAC32(0.4768);
    FWSpeedLoop.pRamp.f32RampDown          = (tFrac32)FRAC32(0.3754);
    FWSpeedLoop.pIQFbck = &f16IDQFbck.f16Arg2;
    FWSpeedLoop.pUQReq   = &f16UDQReq.f16Arg2;
    FWSpeedLoop.pUQLim   = &CurrentLoop.pPIrAWQ.f16UpperLimit;

    // Clear AMCLIB FWSpeedLoop state variables
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB_FWSpeedLoopInit_F16(&FWSpeedLoop);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
}
```

```
AMCLIB_FWSpeedLoopInit(&FWSpeedLoop, F16);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 16-bit fractional implementation is selected as default.
AMCLIB_FWSpeedLoopInit(&FWSpeedLoop);

// Initialize the AMCLIB_FWSpeedLoop state variables to predefined values
// Warning: Parameters in FWSpeedLoop must be already initialized.
f16FilterMAWOut = (tFrac16)123;
f16FilterMAFWOut = (tFrac16)123;
f16ControllerPIpAWQOut = (tFrac16)123;
f16ControllerPIpAWFWOut = (tFrac16)123;
f32RampOut = (tFrac32)123;
// Alternative 1: API call with postfix
// (only one alternative shall be used).
AMCLIB_FWSpeedLoopSetState_F16(f16FilterMAWOut, f16FilterMAFWOut,
    f16ControllerPIpAWQOut, f16ControllerPIpAWFWOut,
    f32RampOut, &FWSpeedLoop);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
AMCLIB_FWSpeedLoopSetState(f16FilterMAWOut, f16FilterMAFWOut,
    f16ControllerPIpAWQOut, f16ControllerPIpAWFWOut,
    f32RampOut, &FWSpeedLoop);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 16-bit fractional implementation is selected as default.
AMCLIB_FWSpeedLoopSetState(f16FilterMAWOut, f16FilterMAFWOut,
    f16ControllerPIpAWQOut, f16ControllerPIpAWFWOut,
    f32RampOut, &FWSpeedLoop);

f16VelocityReq = (tFrac16)100;
while(1);
}

// Periodical function or interrupt - speed control loop
void SlowLoop(void)
{
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB_FWSpeedLoop_F16(f16VelocityReq, f16VelocityFbck,
        &IDQReq, &FWSpeedLoop);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB_FWSpeedLoop(f16VelocityReq, f16VelocityFbck,
        &IDQReq, &FWSpeedLoop, F16);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 16-bit fractional implementation is selected as default.
    AMCLIB_FWSpeedLoop(f16VelocityReq, f16VelocityFbck,
        &IDQReq, &FWSpeedLoop);
}

// Periodical function or interrupt - current control loop
void FastLoop(void)
{
```

```
// Calculate new values of f16VelocityFbck, f16IDQFbck, f16UDQReq
// (...)
```

### 2.5.4.3 Function AMCLIB\_FWSpeedLoopInit\_FLT

#### Declaration

```
void AMCLIB_FWSpeedLoopInit_FLT(AMCLIB_FW_SPEED_LOOP_T_FLT *const pCtrl);
```

#### Arguments

**Table 46. AMCLIB\_FWSpeedLoopInit\_FLT arguments**

Type	Name	Direction	Description
AMCLIB_FW_SPEED_LOOP_T_FLT *const	pCtrl	input, output	Pointer to the structure with AMCLIB_FWSpeedLoop state.

**Note:** If pCtrl points to a structure of type AMCLIB\_FW\_SPEED\_LOOP\_DEBUG\_T\_FLT, it must be recasted to AMCLIB\_FW\_SPEED\_LOOP\_T\_FLT \*.

#### Code Example

```
#include "amclib.h"

AMCLIB_FW_SPEED_LOOP_T_FLT FWSpeedLoop;
SWLIBS_2Syst_FLT IDQReq;           // required dq currents
SWLIBS_2Syst_FLT fltIDQFbck;     // calculated dq currents from the feedback
SWLIBS_2Syst_FLT fltUDQReq;       // required dq voltages
tFloat fltVelocityReq;            // required velocity
tFloat fltVelocityFbck;           // actual velocity
AMCLIB_CURRENT_LOOP_T_FLT CurrentLoop;

void main (void)
{
    tFloat fltFilterMAWOut;
    tFloat fltFilterMAFWOut;
    tFloat fltControllerPIpAWQOut;
    tFloat fltControllerPIpAWFWOut;
    tFloat fltRampOut;

    // Initialize the parameters and pointers in FWSpeedLoop
    FWSpeedLoop.pFilterW.fltLambda      = (tFloat)0.5;
    FWSpeedLoop.pFilterFW.fltLambda     = (tFloat)0.6;
    FWSpeedLoop.pPIpAWQ.fltPropGain    = (tFloat)0.1234;
    FWSpeedLoop.pPIpAWQ.fltIntegGain   = (tFloat)0.1675;
    FWSpeedLoop.pPIpAWQ.fltLowerLimit  = (tFloat)-0.934499979019165;
    FWSpeedLoop.pPIpAWQ.fltUpperLimit  = (tFloat)0.978600025177002;
    FWSpeedLoop.pPIpAWFW.fltPropGain   = (tFloat)0.2348;
    FWSpeedLoop.pPIpAWFW.fltIntegGain  = (tFloat)0.3457;
    FWSpeedLoop.pPIpAWFW.fltLowerLimit = (tFloat)-0.937099993228912;
    FWSpeedLoop.pPIpAWFW.fltUpperLimit = (tFloat)0.963400006294251;
    FWSpeedLoop.pRamp.fltRampUp       = (tFloat)0.4768;
    FWSpeedLoop.pRamp.fltRampDown     = (tFloat)0.3754;
    FWSpeedLoop.pIQFbck = &fltIDQFbck.fltArg2;
    FWSpeedLoop.pUQReq  = &fltUDQReq.fltArg2;
    FWSpeedLoop.pUQLim  = &CurrentLoop.pPIrAWQ.fltUpperLimit;
```

```
FWSpeedLoop.fltUmaxDivImax          = (tFloat) 1.0;

// Clear AMCLIB_FWSpeedLoop state variables
// Alternative 1: API call with postfix
// (only one alternative shall be used).
AMCLIB_FWSpeedLoopInit_FLT(&FWSpeedLoop);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
AMCLIB_FWSpeedLoopInit(&FWSpeedLoop, FLT);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if single precision floating point implementation is selected
// as default.
AMCLIB_FWSpeedLoopInit(&FWSpeedLoop);

// Initialize the AMCLIB_FWSpeedLoop state variables to predefined values
// Warning: Parameters in FWSpeedLoop must be already initialized.
fltFilterMAWOut = 123.0F;
fltFilterMAFWOut = 123.0F;
fltControllerPIpAWQOut = 123.0F;
fltControllerPIpAWFWOut = 123.0F;
fltRampOut = 123.0F;
// Alternative 1: API call with postfix
// (only one alternative shall be used).
AMCLIB_FWSpeedLoopSetState_FLT(fltFilterMAWOut, fltFilterMAFWOut,
    fltControllerPIpAWQOut, fltControllerPIpAWFWOut,
    fltRampOut, &FWSpeedLoop);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
AMCLIB_FWSpeedLoopSetState(fltFilterMAWOut, fltFilterMAFWOut,
    fltControllerPIpAWQOut, fltControllerPIpAWFWOut,
    fltRampOut, &FWSpeedLoop);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if single precision floating point implementation is selected
// as default.
AMCLIB_FWSpeedLoopSetState(fltFilterMAWOut, fltFilterMAFWOut,
    fltControllerPIpAWQOut, fltControllerPIpAWFWOut,
    fltRampOut, &FWSpeedLoop);

fltVelocityReq = 100.0F;
while(1);
}

// Periodical function or interrupt - speed control loop
void SlowLoop(void)
{
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB_FWSpeedLoop_FLT(fltVelocityReq, fltVelocityFbck,
        &IDQReq, &FWSpeedLoop);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB_FWSpeedLoop(fltVelocityReq, fltVelocityFbck,
        &IDQReq, &FWSpeedLoop, FLT);
```

```

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if single precision floating point implementation is selected
// as default.
AMCLIB_FWSpeedLoop(fltVelocityReq, fltVelocityFbck,
    &IDQReq, &FWSpeedLoop);
}

// Periodical function or interrupt - current control loop
void FastLoop(void)
{
    // Calculate new values of fltVelocityFbck, fltIDQFbck, fltUDQReq
    // (...)

}

```

## 2.5.5 Function AMCLIB\_FWSpeedLoopSetState

### Description

This function initializes the AMCLIB\_FWSpeedLoop state variables to achieve the required output values.

**Note:** The input/output pointers must contain valid addresses, otherwise a fault may occur (MemManage, BusFault, UsageFault, HardFault).

### Re-entrancy

The function is re-entrant for a different pCtrl.

### 2.5.5.1 Function AMCLIB\_FWSpeedLoopSetState\_F32

#### Declaration

```
void AMCLIB_FWSpeedLoopSetState_F32(tFrac32 f32FilterMAWOut,
tFrac32 f32FilterMAFWOut, tFrac32 f32ControllerPIpAWQOut,
tFrac32 f32ControllerPIpAWFWOut, tFrac32 f32RampOut,
AMCLIB\_FW\_SPEED\_LOOP\_T\_F32 *pCtrl);
```

#### Arguments

Table 47. AMCLIB\_FWSpeedLoopSetState\_F32 arguments

Type	Name	Direction	Description
<a href="#">tFrac32</a>	f32FilterMAWOut	input	Required output of the speed FilterMA.
<a href="#">tFrac32</a>	f32FilterMAFWOut	input	Required output of the field-weakening FilterMA.
<a href="#">tFrac32</a>	f32ControllerPIpAWQOut	input	Required output of the speed ControllerPIpAW.
<a href="#">tFrac32</a>	f32ControllerPIpAWFWOut	input	Required output of the field-weakening ControllerPIpAW.
<a href="#">tFrac32</a>	f32RampOut	input	Required output of the speed ramp.
<a href="#">AMCLIB_FW_SPEED_LOOP_T_F32</a> *	pCtrl	input, output	Pointer to the structure with AMCLIB_FWSpeedLoop state.

**Caution:** Set the parameters in the structure pointed to by pCtrl before calling this function.

**Note:** If *pCtrl* points to a structure of type [AMCLIB\\_FW\\_SPEED\\_LOOP\\_DEBUG\\_T\\_F32](#), it must be recasted to [AMCLIB\\_FW\\_SPEED\\_LOOP\\_T\\_F32](#).

### Code Example

```
#include "amclib.h"

AMCLIB_FW_SPEED_LOOP_T_F32 FWSpeedLoop;
SWLIBS_2Syst_F32 IDQReq;           // required dq currents
SWLIBS_2Syst_F32 f32IDQFbck;      // calculated dq currents from the feedback
SWLIBS_2Syst_F32 f32UDQReq;       // required dq voltages
tFrac32 f32VelocityReq;          // required velocity
tFrac32 f32VelocityFbck;         // actual velocity
AMCLIB_CURRENT_LOOP_T_F32 CurrentLoop;

void main (void)
{
    tFrac32 f32FilterMAWOut;
    tFrac32 f32FilterMAFWOut;
    tFrac32 f32ControllerPIpAWQOut;
    tFrac32 f32ControllerPIpAWFWOut;
    tFrac32 f32RampOut;

    // Initialize the parameters and pointers in FWSpeedLoop
    FWSpeedLoop.pFilterW.u16NSamples      = 3u;
    FWSpeedLoop.pFilterFW.u16NSamples     = 3u;
    FWSpeedLoop.pPIpAWQ.f32PropGain      = (tFrac32)FRAC32(0.1234);
    FWSpeedLoop.pPIpAWQ.f32IntegGain     = (tFrac32)FRAC32(0.1675);
    FWSpeedLoop.pPIpAWQ.s16PropGainShift = 1;
    FWSpeedLoop.pPIpAWQ.s16IntegGainShift = 1;
    FWSpeedLoop.pPIpAWQ.f32LowerLimit   = (tFrac32)-2006823469L;
    FWSpeedLoop.pPIpAWQ.f32UpperLimit   = (tFrac32)2101527497L;
    FWSpeedLoop.pPIpAWFW.f32PropGain     = (tFrac32)FRAC32(0.2348);
    FWSpeedLoop.pPIpAWFW.f32IntegGain    = (tFrac32)FRAC32(0.3457);
    FWSpeedLoop.pPIpAWFW.s16PropGainShift = 1;
    FWSpeedLoop.pPIpAWFW.s16IntegGainShift = 1;
    FWSpeedLoop.pPIpAWFW.f32LowerLimit   = (tFrac32)-2012406926L;
    FWSpeedLoop.pPIpAWFW.f32UpperLimit   = (tFrac32)2068885746L;
    FWSpeedLoop.pRamp.f32RampUp        = (tFrac32)FRAC32(0.4768);
    FWSpeedLoop.pRamp.f32RampDown      = (tFrac32)FRAC32(0.3754);
    FWSpeedLoop.pIQFbck = &f32IDQFbck.f32Arg2;
    FWSpeedLoop.pUQReq   = &f32UDQReq.f32Arg2;
    FWSpeedLoop.pUQLim   = &CurrentLoop.pPIrAWQ.f32UpperLimit;

    // Clear AMCLIB_FWSpeedLoop state variables
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB_FWSpeedLoopInit_F32(&FWSpeedLoop);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB_FWSpeedLoopInit(&FWSpeedLoop, F32);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 32-bit fractional implementation is selected as default.
    AMCLIB_FWSpeedLoopInit(&FWSpeedLoop);
```

```
// Initialize the AMCLIB_FWSpeedLoop state variables to predefined values
// Warning: Parameters in FWSpeedLoop must be already initialized.
f32FilterMAWOut = (tFrac32)123L;
f32FilterMAFWOut = (tFrac32)123L;
f32ControllerPIpAWQOut = (tFrac32)123L;
f32ControllerPIpAWFWOut = (tFrac32)123L;
f32RampOut = (tFrac32)123L;
// Alternative 1: API call with postfix
// (only one alternative shall be used).
AMCLIB_FWSpeedLoopSetState_F32(f32FilterMAWOut, f32FilterMAFWOut,
    f32ControllerPIpAWQOut, f32ControllerPIpAWFWOut,
    f32RampOut, &FWSpeedLoop);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
AMCLIB_FWSpeedLoopSetState(f32FilterMAWOut, f32FilterMAFWOut,
    f32ControllerPIpAWQOut, f32ControllerPIpAWFWOut,
    f32RampOut, &FWSpeedLoop, F32);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 32-bit fractional implementation is selected as default.
AMCLIB_FWSpeedLoopSetState(f32FilterMAWOut, f32FilterMAFWOut,
    f32ControllerPIpAWQOut, f32ControllerPIpAWFWOut,
    f32RampOut, &FWSpeedLoop);

f32VelocityReq = (tFrac32)100L;
while(1);
}

// Periodical function or interrupt - speed control loop
void SlowLoop(void)
{
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB_FWSpeedLoop_F32(f32VelocityReq, f32VelocityFbck, &IDQReq,
        &FWSpeedLoop);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB_FWSpeedLoop(f32VelocityReq, f32VelocityFbck,
        &IDQReq, &FWSpeedLoop, F32);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 32-bit fractional implementation is selected as default.
    AMCLIB_FWSpeedLoop(f32VelocityReq, f32VelocityFbck,
        &IDQReq, &FWSpeedLoop);
}

// Periodical function or interrupt - current control loop
void FastLoop(void)
{
    // Calculate new values of f32VelocityFbck, f32IDQFbck, f32UDQReq
    // ...
}
```

### 2.5.5.2 Function AMCLIB\_FWSpeedLoopSetState\_F16

#### Declaration

```
void AMCLIB_FWSpeedLoopSetState_F16(tFrac16 f16FilterMAWOut,
tFrac16 f16FilterMAFWOut, tFrac16 f16ControllerPIpAWQOut,
tFrac16 f16ControllerPIpAWFWOut, tFrac32 f32RampOut,
AMCLIB_FW_SPEED_LOOP_T_F16 *pCtrl);
```

#### Arguments

**Table 48. AMCLIB\_FWSpeedLoopSetState\_F16 arguments**

Type	Name	Direction	Description
<u>tFrac16</u>	f16FilterMAWOut	input	Required output of the speed FilterMA.
<u>tFrac16</u>	f16FilterMAFWOut	input	Required output of the field-weakening FilterMA.
<u>tFrac16</u>	f16ControllerPIpAWQOut	input	Required output of the speed ControllerPIpAW.
<u>tFrac16</u>	f16ControllerPIpAWFWOut	input	Required output of the field-weakening ControllerPIpAW.
<u>tFrac32</u>	f32RampOut	input	Required output of the speed ramp.
<u>AMCLIB_FW_SPEED_LOOP_T_F16</u> *	pCtrl	input, output	Pointer to the structure with AMCLIB_FWSpeedLoop state.

**Caution:** Set the parameters in the structure pointed to by pCtrl before calling this function.

**Note:** If pCtrl points to a structure of type AMCLIB\_FW\_SPEED\_LOOP\_DEBUG\_T\_F16, it must be recasted to AMCLIB\_FW\_SPEED\_LOOP\_T\_F16 \*.

#### Code Example

```
#include "amclib.h"

AMCLIB_FW_SPEED_LOOP_T_F16 FWspeedLoop;
SWLIBS_2Syst_F16 IDQReq; // required dq currents
SWLIBS_2Syst_F16 f16IDQFbck; // calculated dq currents from the feedback
SWLIBS_2Syst_F16 f16UDQReq; // required dq voltages
tFrac16 f16VelocityReq; // required velocity
tFrac16 f16VelocityFbck; // actual velocity
AMCLIB_CURRENT_LOOP_T_F16 CurrentLoop;

void main (void)
{
    tFrac16 f16FilterMAWOut;
    tFrac16 f16FilterMAFWOut;
    tFrac16 f16ControllerPIpAWQOut;
    tFrac16 f16ControllerPIpAWFWOut;
    tFrac32 f32RampOut;

    // Initialize the parameters and pointers in FWspeedLoop
    FWspeedLoop.pFilterW.u16NSamples = 3u;
    FWspeedLoop.pFilterFW.u16NSamples = 3u;
    FWspeedLoop.pPIpAWQ.f16PropGain = (tFrac16)FRAC16(0.1234);
    FWspeedLoop.pPIpAWQ.f16IntegGain = (tFrac16)FRAC16(0.1675);
    FWspeedLoop.pPIpAWQ.s16PropGainShift = 1;
    FWspeedLoop.pPIpAWQ.s16IntegGainShift = 1;
    FWspeedLoop.pPIpAWQ.f16LowerLimit = (tFrac16) -30621;
    FWspeedLoop.pPIpAWQ.f16UpperLimit = (tFrac16) 32066;
```

```
FWSpeedLoop.pPIpAWFW.f16PropGain      = (tFrac16)FRAC16(0.2348);  
FWSpeedLoop.pPIpAWFW.f16IntegGain     = (tFrac16)FRAC16(0.3457);  
FWSpeedLoop.pPIpAWFW.s16PropGainShift = 1;  
FWSpeedLoop.pPIpAWFW.s16IntegGainShift = 1;  
FWSpeedLoop.pPIpAWFW.f16LowerLimit   = (tFrac16)-30706;  
FWSpeedLoop.pPIpAWFW.f16UpperLimit   = (tFrac16)31568;  
FWSpeedLoop.pRamp.f32RampUp          = (tFrac32)FRAC32(0.4768);  
FWSpeedLoop.pRamp.f32RampDown        = (tFrac32)FRAC32(0.3754);  
FWSpeedLoop.pIQFbck = &f16IDQFbck.f16Arg2;  
FWSpeedLoop.pUQReq  = &f16UDQReq.f16Arg2;  
FWSpeedLoop.pUQLim  = &CurrentLoop.pPIrAWQ.f16UpperLimit;  
  
// Clear AMCLIB_FWSpeedLoop state variables  
// Alternative 1: API call with postfix  
// (only one alternative shall be used).  
AMCLIB_FWSpeedLoopInit_F16(&FWSpeedLoop);  
  
// Alternative 2: API call with implementation parameter  
// (only one alternative shall be used).  
AMCLIB_FWSpeedLoopInit(&FWSpeedLoop, F16);  
  
// Alternative 3: API call with global configuration of implementation  
// (only one alternative shall be used). This alternative is available  
// only if 16-bit fractional implementation is selected as default.  
AMCLIB_FWSpeedLoopInit(&FWSpeedLoop);  
  
// Initialize the AMCLIB_FWSpeedLoop state variables to predefined values  
// Warning: Parameters in FWSpeedLoop must be already initialized.  
f16FilterMAWOut = (tFrac16)123;  
f16FilterMAFWOut = (tFrac16)123;  
f16ControllerPIpAWQOut = (tFrac16)123;  
f16ControllerPIpAWFWOut = (tFrac16)123;  
f32RampOut = (tFrac32)123;  
// Alternative 1: API call with postfix  
// (only one alternative shall be used).  
AMCLIB_FWSpeedLoopSetState_F16(f16FilterMAWOut, f16FilterMAFWOut,  
f16ControllerPIpAWQOut, f16ControllerPIpAWFWOut,  
f32RampOut, &FWSpeedLoop);  
  
// Alternative 2: API call with implementation parameter  
// (only one alternative shall be used).  
AMCLIB_FWSpeedLoopSetState(f16FilterMAWOut, f16FilterMAFWOut,  
f16ControllerPIpAWQOut, f16ControllerPIpAWFWOut,  
f32RampOut, &FWSpeedLoop, F16);  
  
// Alternative 3: API call with global configuration of implementation  
// (only one alternative shall be used). This alternative is available  
// only if 16-bit fractional implementation is selected as default.  
AMCLIB_FWSpeedLoopSetState(f16FilterMAWOut, f16FilterMAFWOut,  
f16ControllerPIpAWQOut, f16ControllerPIpAWFWOut,  
f32RampOut, &FWSpeedLoop);  
  
f16VelocityReq = (tFrac16)100;  
while(1);  
}  
  
// Periodical function or interrupt - speed control loop  
void SlowLoop(void)  
{  
    // Alternative 1: API call with postfix
```

```

// (only one alternative shall be used).
AMCLIB_FWSpeedLoop_F16(f16VelocityReq, f16VelocityFbck,
    &IDQReq, &FWSpeedLoop);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
AMCLIB_FWSpeedLoop(f16VelocityReq, f16VelocityFbck,
    &IDQReq, &FWSpeedLoop, F16);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 16-bit fractional implementation is selected as default.
AMCLIB_FWSpeedLoop(f16VelocityReq, f16VelocityFbck,
    &IDQReq, &FWSpeedLoop);
}

// Periodical function or interrupt - current control loop
void FastLoop(void)
{
    // Calculate new values of f16VelocityFbck, f16IDQFbck, f16UDQReq
    // (...)
}

```

### 2.5.5.3 Function AMCLIB\_FWSpeedLoopSetState\_FLT

#### Declaration

```
void AMCLIB_FWSpeedLoopSetState_FLT(tFloat fltFilterMAWOut,
tFloat fltFilterMAFWOut, tFloat fltControllerPIpAWQOut,
tFloat fltControllerPIpAWFWOut, tFloat fltRampOut,
AMCLIB\_FW\_SPEED\_LOOP\_T\_FLT *pCtrl);
```

#### Arguments

**Table 49. AMCLIB\_FWSpeedLoopSetState\_FLT arguments**

Type	Name	Direction	Description
<a href="#">tFloat</a>	fltFilterMAWOut	input	Required output of the speed FilterMA.
<a href="#">tFloat</a>	fltFilterMAFWOut	input	Required output of the field-weakening FilterMA.
<a href="#">tFloat</a>	fltControllerPIpAWQOut	input	Required output of the speed ControllerPIpAW.
<a href="#">tFloat</a>	fltControllerPIpAWFWOut	input	Required output of the field-weakening ControllerPIpAW.
<a href="#">tFloat</a>	fltRampOut	input	Required output of the speed ramp.
<a href="#">AMCLIB_FW_SPEED_LOOP_T_FLT</a> *	pCtrl	input, output	Pointer to the structure with AMCLIB_FWSpeedLoop state.

**Caution:** Set the parameters in the structure pointed to by pCtrl before calling this function.

**Note:** If pCtrl points to a structure of type [AMCLIB\\_FW\\_SPEED\\_LOOP\\_DEBUG\\_T\\_FLT](#), it must be recasted to [AMCLIB\\_FW\\_SPEED\\_LOOP\\_T\\_FLT](#).

#### Code Example

```
#include "amclib.h"

AMCLIB\_FW\_SPEED\_LOOP\_T\_FLT FWSpeedLoop;
```

```
SWLIBS_2Syst_FLT IDQReq;           // required dq currents
SWLIBS_2Syst_FLT fltIDQFbck;      // calculated dq currents from the feedback
SWLIBS_2Syst_FLT fltUDQReq;        // required dq voltages
tFloat fltVelocityReq;            // required velocity
tFloat fltVelocityFbck;           // actual velocity
AMCLIB_CURRENT_LOOP_T_FLT CurrentLoop;

void main (void)
{
    tFloat fltFilterMAWOut;
    tFloat fltFilterMAFWOut;
    tFloat fltControllerPIpAWQOut;
    tFloat fltControllerPIpAWFWOut;
    tFloat fltRampOut;

    // Initialize the parameters and pointers in FWSpeedLoop
    FWSpeedLoop.pFilterW.fltLambda = (tFloat) 0.5;
    FWSpeedLoop.pFilterFW.fltLambda = (tFloat) 0.6;
    FWSpeedLoop.pPIpAWQ.fltPropGain = (tFloat) 0.1234;
    FWSpeedLoop.pPIpAWQ.fltIntegGain = (tFloat) 0.1675;
    FWSpeedLoop.pPIpAWQ.fltLowerLimit = (tFloat) -0.934499979019165;
    FWSpeedLoop.pPIpAWQ.fltUpperLimit = (tFloat) 0.978600025177002;
    FWSpeedLoop.pPIpAWFW.fltPropGain = (tFloat) 0.2348;
    FWSpeedLoop.pPIpAWFW.fltIntegGain = (tFloat) 0.3457;
    FWSpeedLoop.pPIpAWFW.fltLowerLimit = (tFloat) -0.937099993228912;
    FWSpeedLoop.pPIpAWFW.fltUpperLimit = (tFloat) 0.963400006294251;
    FWSpeedLoop.pRamp.fltRampUp = (tFloat) 0.4768;
    FWSpeedLoop.pRamp.fltRampDown = (tFloat) 0.3754;
    FWSpeedLoop.piQFbck = &fltIDQFbck.fltArg2;
    FWSpeedLoop.pUQReq = &fltUDQReq.fltArg2;
    FWSpeedLoop.pUQLim = &CurrentLoop.pPIrAWQ.fltUpperLimit;
    FWSpeedLoop.fltUmaxDivImax = (tFloat) 1.0;

    // Clear AMCLIB_FWSpeedLoop state variables
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB_FWSpeedLoopInit_FLT(&FWSpeedLoop);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB_FWSpeedLoopInit(&FWSpeedLoop, FLT);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if single precision floating point implementation is selected
    // as default.
    AMCLIB_FWSpeedLoopInit(&FWSpeedLoop);

    // Initialize the AMCLIB_FWSpeedLoop state variables to predefined values
    // Warning: Parameters in FWSpeedLoop must be already initialized.
    fltFilterMAWOut = 123.0F;
    fltFilterMAFWOut = 123.0F;
    fltControllerPIpAWQOut = 123.0F;
    fltControllerPIpAWFWOut = 123.0F;
    fltRampOut = 123.0F;
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB_FWSpeedLoopSetState_FLT(fltFilterMAWOut, fltFilterMAFWOut,
        fltControllerPIpAWQOut, fltControllerPIpAWFWOut,
        fltRampOut, &FWSpeedLoop);
```

```
// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
AMCLIB_FWSpeedLoopSetState(fltFilterMAWOut, fltFilterMAFWOut,
    fltControllerPIpAWQOut, fltControllerPIpAWFWOut,
    fltRampOut, &FWSpeedLoop, FLT);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if single precision floating point implementation is selected
// as default.
AMCLIB_FWSpeedLoopSetState(fltFilterMAWOut, fltFilterMAFWOut,
    fltControllerPIpAWQOut, fltControllerPIpAWFWOut,
    fltRampOut, &FWSpeedLoop);

fltVelocityReq = 100.0F;
while(1);
}

// Periodical function or interrupt - speed control loop
void SlowLoop(void)
{
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB_FWSpeedLoop_FLT(fltVelocityReq, fltVelocityFbck,
        &IDQReq, &FWSpeedLoop);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB_FWSpeedLoop(fltVelocityReq, fltVelocityFbck,
        &IDQReq, &FWSpeedLoop, FLT);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if single precision floating point implementation is selected
    // as default.
    AMCLIB_FWSpeedLoop(fltVelocityReq, fltVelocityFbck,
        &IDQReq, &FWSpeedLoop);
}

// Periodical function or interrupt - current control loop
void FastLoop(void)
{
    // Calculate new values of fltVelocityFbck, fltIDQFbck, fltUDQReq
    // (...)
```

## 2.5.6 Function AMCLIB\_FWSpeedLoopDebug

This function adjusts the torque of the motor to achieve the required speed. The function employs the PMSM Field Weakening technique to extend the available speed range. Debugging information is provided.

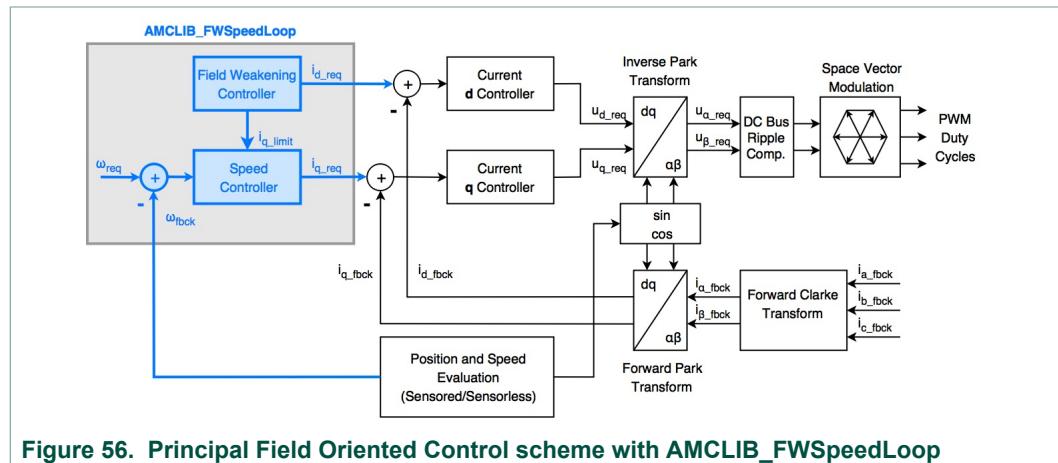
### Description

This library function implements a portion of Field Oriented Control (FOC) algorithm. FOC (also called vector control) is a widely used control strategy for Permanent Magnet Synchronous Motors (PMSM). FOC is characterized by smooth rotation over the entire

speed range of the motor, full torque control at zero speed, and fast acceleration/deceleration. FOC consists of a hierarchical cascade of inner current loop and outer speed loop. PI controllers within these closed loops maintain the required speed and torque based on feedback measurements.

AMCLIB\_FWSpeedLoopDebug implements the speed PI controller and the field weakening controller in the outer control loop highlighted in [Figure 56](#).

AMCLIB\_FWSpeedLoopDebug combines the functionalities of [AMCLIB\\_FWDebug](#) and [AMCLIB\\_SpeedLoopDebug](#) in a more integrated form to simplify the application code and improve execution speed. AMCLIB\_FWSpeedLoopDebug provides the same functionality as [AMCLIB\\_FWSpeedLoop](#). Additionally, this function allows debugging of all internal variables. The debugging outputs are provided in the structure pointed to by pCtrl. Replace AMCLIB\_FWSpeedLoopDebug by [AMCLIB\\_FWSpeedLoop](#) once the debugging is finished.



**Figure 56. Principal Field Oriented Control scheme with AMCLIB\_FWSpeedLoop**

Field weakening technique employed in AMCLIB\_FWSpeedLoopDebug extends the speed range of PMSM beyond the motor base speed by reducing the linkage magnetic flux through negative current  $i_d\_req$ . See [AMCLIB\\_FW](#) for more details.

### 2.5.6.1 Function AMCLIB\_FWSpeedLoopDebug\_F32

#### Declaration

```
void AMCLIB_FWSpeedLoopDebug_F32 (tFrac32 f32VelocityReq,
tFrac32 f32VelocityFbck, SWLIBS\_2Syst\_F32 *const pIDQReq,
AMCLIB\_FW\_SPEED\_LOOP\_DEBUG\_T\_F32 *pCtrl);
```

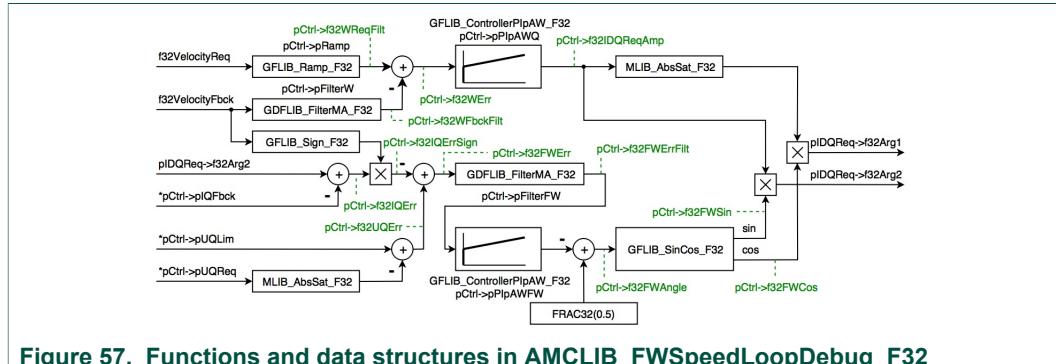
#### Arguments

**Table 50. AMCLIB\_FWSpeedLoopDebug\_F32 arguments**

Type	Name	Direction	Description
<a href="#">tFrac32</a>	f32VelocityReq	input	Required electrical angular velocity (setpoint).
<a href="#">tFrac32</a>	f32VelocityFbck	input	Actual electrical angular velocity from the feedback.
<a href="#">SWLIBS_2Syst_F32</a> *const	pIDQReq	input, output	Pointer to the structure with the required stator currents in the two-phase rotational orthogonal system (d-q).
<a href="#">AMCLIB_FW_SPEED_LOOP_DEBUG_T_F32</a> *	pCtrl	input, output	Pointer to the structure with AMCLIB_FWSpeedLoop state and debugging information.

## Implementation details

The following block diagram shows the internal functions and data structures of `AMCLIB_FWSpeedLoopDebug_F32`.



**Figure 57. Functions and data structures in AMCLIB\_FWSpeedLoopDebug\_F32**

Refer to the description of [AMCLIB\\_FWSpeedLoop\\_F32](#) function on how to set up the controller parameters.

**Note:** Due to effectivity reasons, this function is implemented using inline assembly and is therefore not ANSI-C compliant.

## Code Example

```
#include "amclib.h"

AMCLIB_FW_SPEED_LOOP_DEBUG_T_F32 FWSpeedLoop;
SWLIBS_2Syst_F32 IDQReq; // required dq currents
SWLIBS_2Syst_F32 f32IDQFbck; // calculated dq currents from the feedback
SWLIBS_2Syst_F32 f32UDQReq; // required dq voltages
tFrac32 f32VelocityReq; // required velocity
tFrac32 f32VelocityFbck; // actual velocity
AMCLIB_CURRENT_LOOP_T_F32 CurrentLoop;

void main (void)
{
    tFrac32 f32FilterMAWOut;
    tFrac32 f32FilterMAFWOut;
    tFrac32 f32ControllerPIpAWQOut;
    tFrac32 f32ControllerPIpAWFWOut;
    tFrac32 f32RampOut;

    // Initialize the parameters and pointers in FWSpeedLoop
    FWSpeedLoop.pFilterW.u16NSamples = 3u;
    FWSpeedLoop.pFilterFW.u16NSamples = 3u;
    FWSpeedLoop.pPIpAWQ.f32PropGain = (tFrac32)FRAC32(0.1234);
    FWSpeedLoop.pPIpAWQ.f32IntegGain = (tFrac32)FRAC32(0.1675);
    FWSpeedLoop.pPIpAWQ.s16PropGainShift = 1;
    FWSpeedLoop.pPIpAWQ.s16IntegGainShift = 1;
    FWSpeedLoop.pPIpAWQ.f32LowerLimit = (tFrac32)-2006823469L;
    FWSpeedLoop.pPIpAWQ.f32UpperLimit = (tFrac32)2101527497L;
    FWSpeedLoop.pPIpAWFW.f32PropGain = (tFrac32)FRAC32(0.2348);
    FWSpeedLoop.pPIpAWFW.f32IntegGain = (tFrac32)FRAC32(0.3457);
    FWSpeedLoop.pPIpAWFW.s16PropGainShift = 1;
    FWSpeedLoop.pPIpAWFW.s16IntegGainShift = 1;
```

```
FWSpeedLoop.pPIpAWFW.f32LowerLimit      = (tFrac32) -2012406926L;
FWSpeedLoop.pPIpAWFW.f32UpperLimit      = (tFrac32) 2068885746L;
FWSpeedLoop.pRamp.f32RampUp             = (tFrac32) FRAC32(0.4768);
FWSpeedLoop.pRamp.f32RampDown           = (tFrac32) FRAC32(0.3754);
FWSpeedLoop.pIQFbck = &f32IDQFbck.f32Arg2;
FWSpeedLoop.pUQReq   = &f32UDQReq.f32Arg2;
FWSpeedLoop.pUQLim   = &CurrentLoop.pPIrAWQ.f32UpperLimit;

// Clear AMCLIB_FWSpeedLoop state variables
// Alternative 1: API call with postfix
// (only one alternative shall be used).
AMCLIB_FWSpeedLoopInit_F32((AMCLIB_FW_SPEED_LOOP_T_F32 *)&FWSpeedLoop);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
AMCLIB_FWSpeedLoopInit((AMCLIB_FW_SPEED_LOOP_T_F32 *)&FWSpeedLoop, F32);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 32-bit fractional implementation is selected as default.
AMCLIB_FWSpeedLoopInit((AMCLIB_FW_SPEED_LOOP_T_F32 *)&FWSpeedLoop);

// Initialize the AMCLIB_FWSpeedLoop state variables to predefined values
// Warning: Parameters in FWSpeedLoop must be already initialized.
f32FilterMAWOut = (tFrac32) 123L;
f32FilterMAFWOut = (tFrac32) 123L;
f32ControllerPIpAWQOut = (tFrac32) 123L;
f32ControllerPIpAWFWOut = (tFrac32) 123L;
f32RampOut = (tFrac32) 123L;
// Alternative 1: API call with postfix
// (only one alternative shall be used).
AMCLIB_FWSpeedLoopSetState_F32(f32FilterMAWOut, f32FilterMAFWOut,
    f32ControllerPIpAWQOut, f32ControllerPIpAWFWOut,
    f32RampOut, (AMCLIB_FW_SPEED_LOOP_T_F32 *)&FWSpeedLoop);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
AMCLIB_FWSpeedLoopSetState(f32FilterMAWOut, f32FilterMAFWOut,
    f32ControllerPIpAWQOut, f32ControllerPIpAWFWOut,
    f32RampOut, (AMCLIB_FW_SPEED_LOOP_T_F32 *)&FWSpeedLoop);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 32-bit fractional implementation is selected as default.
AMCLIB_FWSpeedLoopSetState(f32FilterMAWOut, f32FilterMAFWOut,
    f32ControllerPIpAWQOut, f32ControllerPIpAWFWOut,
    f32RampOut, (AMCLIB_FW_SPEED_LOOP_T_F32 *)&FWSpeedLoop);

f32VelocityReq = (tFrac32) 100L;
while(1);

// Periodical function or interrupt - speed control loop
void SlowLoop(void)
{
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB_FWSpeedLoopDebug_F32(f32VelocityReq, f32VelocityFbck, &IDQReq,
        &FWSpeedLoop);
```

```

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
AMCLIB_FWSpeedLoopDebug(f32VelocityReq, f32VelocityFbck,
    &IDQReq, &FWSpeedLoop, F32);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 32-bit fractional implementation is selected as default.
AMCLIB_FWSpeedLoopDebug(f32VelocityReq, f32VelocityFbck,
    &IDQReq, &FWSpeedLoop);
}

// Periodical function or interrupt - current control loop
void FastLoop(void)
{
    // Calculate new values of f32VelocityFbck, f32IDQFbck, f32UDQReq
    // (...)
}

```

### 2.5.6.2 Function AMCLIB\_FWSpeedLoopDebug\_F16

#### Declaration

```
void AMCLIB_FWSpeedLoopDebug_F16(tFrac16 f16VelocityReq,
tFrac16 f16VelocityFbck, SWLIBS\_2Syst\_F16 *const pIDQReq,
AMCLIB\_FW\_SPEED\_LOOP\_DEBUG\_T\_F16 *pCtrl);
```

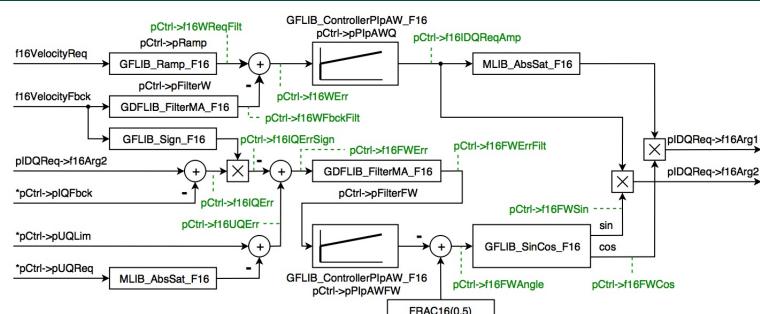
#### Arguments

**Table 51. AMCLIB\_FWSpeedLoopDebug\_F16 arguments**

Type	Name	Direction	Description
<a href="#">tFrac16</a>	f16VelocityReq	input	Required electrical angular velocity (setpoint).
<a href="#">tFrac16</a>	f16VelocityFbck	input	Actual electrical angular velocity from the feedback.
<a href="#">SWLIBS_2Syst_F16</a> *const	pIDQReq	input, output	Pointer to the structure with the required stator currents in the two-phase rotational orthogonal system (d-q).
<a href="#">AMCLIB_FW_SPEED_LOOP_DEBUG_T_F16</a> *	pCtrl	input, output	Pointer to the structure with AMCLIB_FWSpeedLoop state and debugging information.

#### Implementation details

The following block diagram shows the internal functions and data structures of AMCLIB\_FWSpeedLoopDebug\_F16.



**Figure 58. Functions and data structures in AMCLIB\_FWSpeedLoopDebug\_F16**

Refer to the description of [AMCLIB\\_FWSpeedLoop\\_F16](#) function on how to set up the controller parameters.

**Note:** Due to effectivity reasons, this function is implemented using inline assembly and is therefore not ANSI-C compliant.

### Code Example

```
#include "amclib.h"

AMCLIB_FW_SPEED_LOOP_DEBUG_T_F16 FWSpeedLoop;
SWLIBS_2Syst_F16 IDQReq;          // required dq currents
SWLIBS_2Syst_F16 f16IDQFbck;    // calculated dq currents from the feedback
SWLIBS_2Syst_F16 f16UDQReq;     // required dq voltages
tFrac16 f16VelocityReq;        // required velocity
tFrac16 f16VelocityFbck;       // actual velocity
AMCLIB_CURRENT_LOOP_T_F16 CurrentLoop;

void main (void)
{
    tFrac16 f16FilterMAWOut;
    tFrac16 f16FilterMAFWOut;
    tFrac16 f16ControllerPIpAWQOut;
    tFrac16 f16ControllerPIpAWFWOut;
    tFrac32 f32RampOut;

    // Initialize the parameters and pointers in FWSpeedLoop
    FWSpeedLoop.pFilterW.ul6NSamples      = 3u;
    FWSpeedLoop.pFilterFW.ul6NSamples      = 3u;
    FWSpeedLoop.pPIpAWQ.f16PropGain       = (tFrac16)FRAC16(0.1234);
    FWSpeedLoop.pPIpAWQ.f16IntegGain      = (tFrac16)FRAC16(0.1675);
    FWSpeedLoop.pPIpAWQ.s16PropGainShift  = 1;
    FWSpeedLoop.pPIpAWQ.s16IntegGainShift = 1;
    FWSpeedLoop.pPIpAWQ.f16LowerLimit    = (tFrac16)-30621;
    FWSpeedLoop.pPIpAWQ.f16UpperLimit    = (tFrac16)32066;
    FWSpeedLoop.pPIpAWFW.f16PropGain      = (tFrac16)FRAC16(0.2348);
    FWSpeedLoop.pPIpAWFW.f16IntegGain     = (tFrac16)FRAC16(0.3457);
    FWSpeedLoop.pPIpAWFW.s16PropGainShift = 1;
    FWSpeedLoop.pPIpAWFW.s16IntegGainShift = 1;
    FWSpeedLoop.pPIpAWFW.f16LowerLimit    = (tFrac16)-30706;
    FWSpeedLoop.pPIpAWFW.f16UpperLimit    = (tFrac16)31568;
    FWSpeedLoop.pRamp.f32RampUp          = (tFrac32)FRAC32(0.4768);
    FWSpeedLoop.pRamp.f32RampDown         = (tFrac32)FRAC32(0.3754);
    FWSpeedLoop.pIQFbck = &f16IDQFbck.f16Arg2;
    FWSpeedLoop.pUQReq   = &f16UDQReq.f16Arg2;
    FWSpeedLoop.pUQLim  = &CurrentLoop.pPIrAWQ.f16UpperLimit;

    // Clear AMCLIB_FWSpeedLoop state variables
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB_FWSpeedLoopInit_F16((AMCLIB_FW_SPEED_LOOP_T_F16 *)&FWSpeedLoop);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB_FWSpeedLoopInit((AMCLIB_FW_SPEED_LOOP_T_F16 *)&FWSpeedLoop, F16);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
```

```
// only if 16-bit fractional implementation is selected as default.
AMCLIB_FWSpeedLoopInit((AMCLIB_FW_SPEED_LOOP_T_F16 *)&FWSpeedLoop);

// Initialize the AMCLIB_FWSpeedLoop state variables to predefined values
// Warning: Parameters in FWSpeedLoop must be already initialized.
f16FilterMAWOut = (tFrac16)123;
f16FilterMAFWOut = (tFrac16)123;
f16ControllerPIpAWQOut = (tFrac16)123;
f16ControllerPIpAWFWOut = (tFrac16)123;
f32RampOut = (tFrac32)123;
// Alternative 1: API call with postfix
// (only one alternative shall be used).
AMCLIB_FWSpeedLoopSetState_F16(f16FilterMAWOut, f16FilterMAFWOut,
    f16ControllerPIpAWQOut, f16ControllerPIpAWFWOut,
    f32RampOut, (AMCLIB_FW_SPEED_LOOP_T_F16 *)&FWSpeedLoop);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
AMCLIB_FWSpeedLoopSetState(f16FilterMAWOut, f16FilterMAFWOut,
    f16ControllerPIpAWQOut, f16ControllerPIpAWFWOut,
    f32RampOut, (AMCLIB_FW_SPEED_LOOP_T_F16 *)&FWSpeedLoop, F16);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 16-bit fractional implementation is selected as default.
AMCLIB_FWSpeedLoopSetState(f16FilterMAWOut, f16FilterMAFWOut,
    f16ControllerPIpAWQOut, f16ControllerPIpAWFWOut,
    f32RampOut, (AMCLIB_FW_SPEED_LOOP_T_F16 *)&FWSpeedLoop);

f16VelocityReq = (tFrac16)100;
while(1);
}

// Periodical function or interrupt - speed control loop
void SlowLoop(void)
{
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB_FWSpeedLoopDebug_F16(f16VelocityReq, f16VelocityFbck,
        &IDQReq, &FWSpeedLoop);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB_FWSpeedLoopDebug(f16VelocityReq, f16VelocityFbck,
        &IDQReq, &FWSpeedLoop, F16);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 16-bit fractional implementation is selected as default.
    AMCLIB_FWSpeedLoopDebug(f16VelocityReq, f16VelocityFbck,
        &IDQReq, &FWSpeedLoop);
}

// Periodical function or interrupt - current control loop
void FastLoop(void)
{
    // Calculate new values of f16VelocityFbck, f16IDQFbck, f16UDQReq
    // ...
}
```

### 2.5.6.3 Function AMCLIB\_FWSpeedLoopDebug\_FLT

#### Declaration

```
void AMCLIB_FWSpeedLoopDebug_FLT(tFloat fltVelocityReq,
tFloat fltVelocityFbck, SWLIBS_2Syst_FLT *const pIDQReq,
AMCLIB_FW_SPEED_LOOP_DEBUG_T_FLT *pCtrl);
```

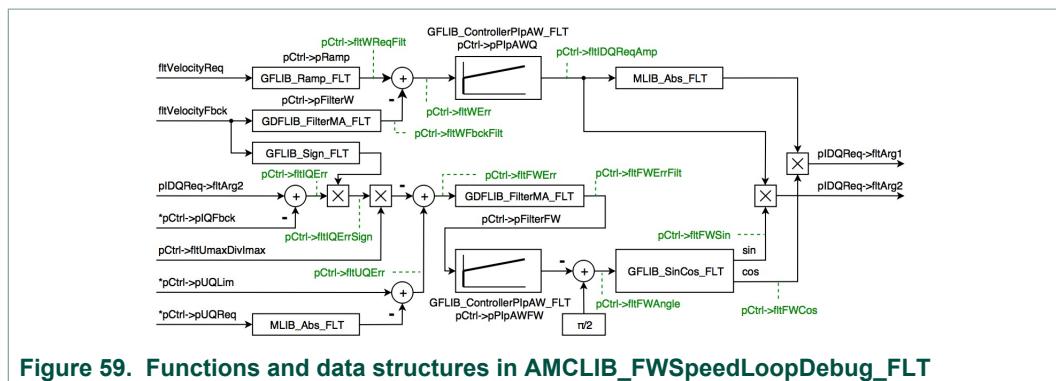
#### Arguments

**Table 52. AMCLIB\_FWSpeedLoopDebug\_FLT arguments**

Type	Name	Direction	Description
<b>tFloat</b>	fltVelocityReq	input	Required electrical angular velocity (setpoint).
<b>tFloat</b>	fltVelocityFbck	input	Actual electrical angular velocity from the feedback.
<b>SWLIBS_2Syst_FLT</b> *const	pIDQReq	input, output	Pointer to the structure with the required stator currents in the two-phase rotational orthogonal system (d-q).
<b>AMCLIB_FW_SPEED_LOOP_DEBUG_T_FLT</b> *	pCtrl	input, output	Pointer to the structure with AMCLIB_FWSpeedLoop state and debugging information.

#### Implementation details

The following block diagram shows the internal functions and data structures of AMCLIB\_FWSpeedLoopDebug\_FLT.



**Figure 59. Functions and data structures in AMCLIB\_FWSpeedLoopDebug\_FLT**

Refer to the description of [AMCLIB\\_FWSpeedLoop\\_FLT](#) function on how to set up the controller parameters.

**Note:** The function may raise floating-point exceptions (invalid operation, overflow, underflow, inexact, input denormal). The floating-point unit must be enabled in CPACR register to prevent NOCP UsageFault exception.

#### Code Example

```
#include "amclib.h"

AMCLIB_FW_SPEED_LOOP_DEBUG_T_FLT FWSpeedLoop;
SWLIBS_2Syst_FLT IDQReq; // required dq currents
SWLIBS_2Syst_FLT fltIDQFbck; // calculated dq currents from the feedback
SWLIBS_2Syst_FLT fltUDQReq; // required dq voltages
tFloat fltVelocityReq; // required velocity
```

```
tFloat fltVelocityFbck;           // actual velocity
AMCLIB_CURRENT_LOOP_T_FLT CurrentLoop;

void main (void)
{
    tFloat fltFilterMAWOut;
    tFloat fltFilterMAFWOut;
    tFloat fltControllerPIpAWQOut;
    tFloat fltControllerPIpAWFWOut;
    tFloat fltRampOut;

    // Initialize the parameters and pointers in FWSpeedLoop
    FWSpeedLoop.pFilterW.fltLambda      = (tFloat) 0.5;
    FWSpeedLoop.pFilterFW.fltLambda     = (tFloat) 0.6;
    FWSpeedLoop.pPIpAWQ.fltPropGain    = (tFloat) 0.1234;
    FWSpeedLoop.pPIpAWQ.fltIntegGain   = (tFloat) 0.1675;
    FWSpeedLoop.pPIpAWQ.fltLowerLimit  = (tFloat) -0.934499979019165;
    FWSpeedLoop.pPIpAWQ.fltUpperLimit  = (tFloat) 0.978600025177002;
    FWSpeedLoop.pPIpAWFW.fltPropGain   = (tFloat) 0.2348;
    FWSpeedLoop.pPIpAWFW.fltIntegGain  = (tFloat) 0.3457;
    FWSpeedLoop.pPIpAWFW.fltLowerLimit = (tFloat) -0.937099993228912;
    FWSpeedLoop.pPIpAWFW.fltUpperLimit = (tFloat) 0.963400006294251;
    FWSpeedLoop.pRamp.fltRampUp       = (tFloat) 0.4768;
    FWSpeedLoop.pRamp.fltRampDown     = (tFloat) 0.3754;
    FWSpeedLoop.piQFbck = &fltIDQFbck.fltArg2;
    FWSpeedLoop.pUQReq  = &fltUDQReq.fltArg2;
    FWSpeedLoop.pUQLim  = &CurrentLoop.pPIrAWQ.fltUpperLimit;
    FWSpeedLoop.fltUmaxDivImax      = (tFloat) 1.0;

    // Clear AMCLIB_FWSpeedLoop state variables
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB_FWSpeedLoopInit_FLT((AMCLIB_FW_SPEED_LOOP_T_FLT *)&FWSpeedLoop);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB_FWSpeedLoopInit((AMCLIB_FW_SPEED_LOOP_T_FLT *)&FWSpeedLoop, FLT);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if single precision floating point implementation is selected
    // as default.
    AMCLIB_FWSpeedLoopInit((AMCLIB_FW_SPEED_LOOP_T_FLT *)&FWSpeedLoop);

    // Initialize the AMCLIB_FWSpeedLoop state variables to predefined values
    // Warning: Parameters in FWSpeedLoop must be already initialized.
    fltFilterMAWOut = 123.0F;
    fltFilterMAFWOut = 123.0F;
    fltControllerPIpAWQOut = 123.0F;
    fltControllerPIpAWFWOut = 123.0F;
    fltRampOut = 123.0F;
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB_FWSpeedLoopSetState_FLT(fltFilterMAWOut, fltFilterMAFWOut,
        fltControllerPIpAWQOut, fltControllerPIpAWFWOut,
        fltRampOut, (AMCLIB_FW_SPEED_LOOP_T_FLT *)&FWSpeedLoop);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB_FWSpeedLoopSetState(fltFilterMAWOut, fltFilterMAFWOut,
```

```
    fltControllerPIpAWQOut, fltControllerPIpAWFWOut,
    fltRampOut, (AMCLIB\_FW\_SPEED\_LOOP\_T\_FLT *)&FWSpeedLoop, FLT);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if single precision floating point implementation is selected
    // as default.
    AMCLIB_FWSpeedLoopSetState(fltFilterMAWOut, fltFilterMAFWOut,
        fltControllerPIpAWQOut, fltControllerPIpAWFWOut,
        fltRampOut, (AMCLIB\_FW\_SPEED\_LOOP\_T\_FLT *)&FWSpeedLoop);

    fltVelocityReq = 100.0F;
    while(1);
}

// Periodical function or interrupt - speed control loop
void SlowLoop(void)
{
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB_FWSpeedLoopDebug_FLT(fltVelocityReq, fltVelocityFbck,
        &IDQReq, &FWSpeedLoop);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB_FWSpeedLoopDebug(fltVelocityReq, fltVelocityFbck,
        &IDQReq, &FWSpeedLoop, FLT);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if single precision floating point implementation is selected
    // as default.
    AMCLIB_FWSpeedLoopDebug(fltVelocityReq, fltVelocityFbck,
        &IDQReq, &FWSpeedLoop);
}

// Periodical function or interrupt - current control loop
void FastLoop(void)
{
    // Calculate new values of fltVelocityFbck, fltIDQFbck, fltUDQReq
    // (...)
```

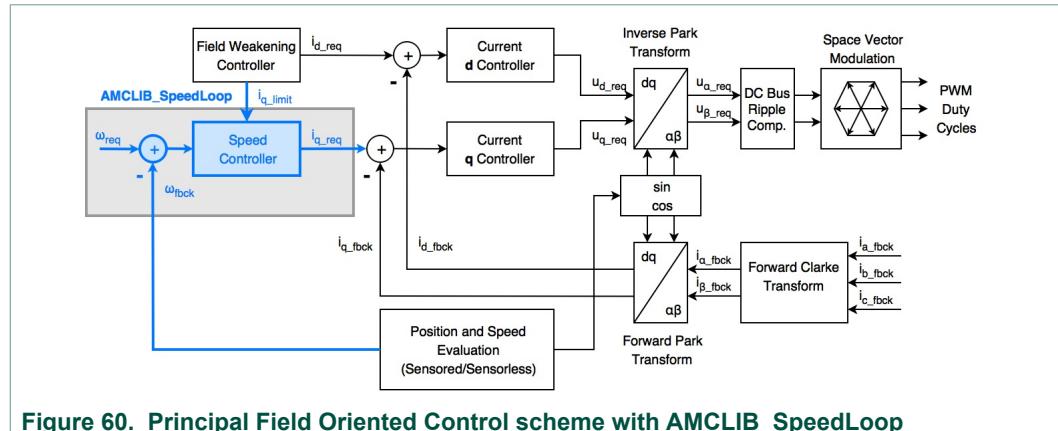
## 2.6 Function AMCLIB\_SpeedLoop

This function implements the PI controller in the speed FOC outer control loop.

### Description

This library function implements the speed control loop which is the outer loop in the cascade control structure of the speed FOC. FOC (also called vector control) is a widely used control strategy for Permanent Magnet Synchronous Motors (PMSM). FOC is characterized by smooth rotation over the entire speed range of the motor, full torque control at zero speed, and fast acceleration/deceleration. A PI controller in speed closed-loop system maintains the required speed by setting the torque producing current in the q-axis.

AMCLIB\_SpeedLoop implements the speed controller in the outer control loop highlighted in [Figure 60](#). Use [AMCLIB\\_FWSpeedLoop](#) instead to take advantage of the field weakening technique when the application requires motor speeds beyond the nominal range. AMCLIB\_SpeedLoop does not allow debugging of all internal variables. Use [AMCLIB\\_SpeedLoopDebug](#) for debugging purposes and replace it with AMCLIB\_SpeedLoop once the debugging is finished.



**Figure 60. Principal Field Oriented Control scheme with AMCLIB\_SpeedLoop**

Before using the FOC with a particular motor, the user needs to provide a set of coefficients through the pCtrl input pointer. The controller coefficient values can be calculated from motor parameters.

Refer to the following resources to find out how the NXP motor control tuning and debugging tools for NXP microcontrollers can help you deploy the FOC in your application:

- [AN4642](#) - Motor Control Application Tuning (MCAT) Tool for 3-Phase PMSM
- [FREEMASTER](#) - FreeMASTER Run-Time Debugging Tool

**Note:** The input/output pointers must contain valid addresses, otherwise a fault may occur (*MemManage*, *BusFault*, *UsageFault*, *HardFault*).

## Re-entrancy

The function is re-entrant for a different pCtrl.

### 2.6.1 Function AMCLIB\_SpeedLoop\_F32

#### Declaration

```
void AMCLIB_SpeedLoop_F32(tFrac32 f32VelocityReq,
tFrac32 f32VelocityFbck, SWLIBS_2Syst_F32 *const pIDQReq,
AMCLIB_SPEED_LOOP_T_F32 *pCtrl);
```

#### Arguments

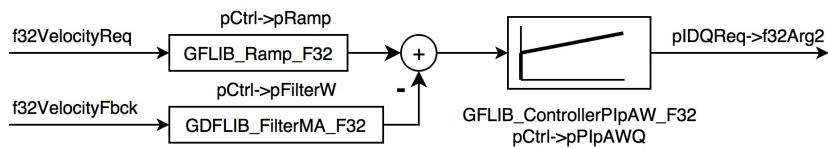
**Table 53. AMCLIB\_SpeedLoop\_F32 arguments**

Type	Name	Direction	Description
tFrac32	f32VelocityReq	input	Required electrical angular velocity (setpoint).
tFrac32	f32VelocityFbck	input	Actual electrical angular velocity from the feedback.

Type	Name	Direction	Description
<a href="#">SWLIBS_2Syst_F32</a> *const	pIDQReq	output	Pointer to the structure with the required stator currents in the two-phase rotational orthogonal system (d-q). Only the q-axis component is written by the function.
<a href="#">AMCLIB_SPEED_LOOP_T_F32</a> *	pCtrl	input, output	Pointer to the structure with AMCLIB_SpeedLoop state.

### Implementation details

The following block diagram shows the internal functions and data structures of AMCLIB\_SpeedLoop\_F32.



**Figure 61. Functions and data structures in AMCLIB\_SpeedLoop\_F32**

Prior to calculating the controller coefficients, it is necessary to set the scaling constants. All inputs and outputs of the algorithm are limited to the fractional range [-1, 1). Incorrect setting of the scaling constants may lead to undesirable overflow or saturation during the computation.

**Table 54. Scaling constants**

Scaling constant	Symbol	Calculation
Maximum phase current [A]	$I_{MAX}$	Maximum current depends on power stage capabilities.
Maximum speed [rad/s]	$\Omega_{MAX}$	Maximum application required speed, at least the motor electrical rated speed.

Parameters of the speed PIPAW controller (using bilinear transform) can be calculated using the following equations:

$$\begin{aligned}
 pPIP AWQ.f32PropGain &= FRAC32\left(\left(2 \cdot \xi \cdot \omega_0 \cdot \frac{J}{K_T}\right) \cdot \frac{\Omega_{MAX}}{I_{MAX}} \cdot 2^{NShiftP}\right) \\
 pPIP AWQ.f32IntegGain &= FRAC32\left(\left(\omega_0^2 \cdot \frac{J}{K_T} \cdot \frac{T_s}{2}\right) \cdot \frac{\Omega_{MAX}}{I_{MAX}} \cdot 2^{NShiftI}\right) \\
 pPIP AWQ.s16PropGainShift &= NShiftP \\
 pPIP AWQ.s16IntegGainShift &= NShiftI \\
 pPIP AWQ.f32UpperLimit &= FRAC32(1) \\
 pPIP AWQ.f32LowerLimit &= FRAC32(-1) \\
 \text{Equation AMCLIB_SpeedLoop_F32_Eq1}
 \end{aligned}$$

where  $\xi$  is the speed loop attenuation,  $\omega_0$  is the speed loop natural frequency [rad/s],  $J$  is the moment of inertia,  $K_T$  is the motor torque constant, and  $T_s$  is the sampling period.  $NShiftP$  and  $NShiftI$  are integer values which ensure that the controller coefficients fit in the fractional range [-1, 1].

The smoothing factor of the [GDFLIB\\_FilterMA\\_F32](#) and the slope of the [GFLIB\\_Ramp\\_F32](#) on the input of the speed controller should reflect the achievable dynamics of the drive. The [AN4642](#) (Motor Control Application Tuning (MCAT) Tool for 3-Phase PMSM) can help with tuning of these parameters.

**Note:** Due to effectivity reasons, this function is implemented using inline assembly and is therefore not ANSI-C compliant.

### Code Example

```
#include "amclib.h"

AMCLIB_SPEED_LOOP_T_F32 SpeedLoop;
SWLIBS_2Syst_F32 IDQReq;           // required dq currents
tFrac32 f32VelocityReq;          // required velocity
tFrac32 f32VelocityFbck;         // actual velocity

void main (void)
{
    tFrac32 f32FilterMAWOut;
    tFrac32 f32ControllerPIpAWQOut;
    tFrac32 f32RampOut;

    // Initialize the parameters in SpeedLoop
    SpeedLoop.pFilterW.u16NSamples      = 3u;
    SpeedLoop.pPIpAWQ.f32PropGain       = (tFrac32)FRAC32(0.1234);
    SpeedLoop.pPIpAWQ.f32IntegGain     = (tFrac32)FRAC32(0.1675);
    SpeedLoop.pPIpAWQ.s16PropGainShift = 1;
    SpeedLoop.pPIpAWQ.s16IntegGainShift = 1;
    SpeedLoop.pPIpAWQ.f32LowerLimit   = (tFrac32)-2006823469L;
    SpeedLoop.pPIpAWQ.f32UpperLimit   = (tFrac32)2101527497L;
    SpeedLoop.pRamp.f32RampUp         = (tFrac32)FRAC32(0.4768);
    SpeedLoop.pRamp.f32RampDown       = (tFrac32)FRAC32(0.3754);

    // Clear AMCLIB_SpeedLoop state variables
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB_SpeedLoopInit_F32(&SpeedLoop);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB_SpeedLoopInit(&SpeedLoop, F32);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 32-bit fractional implementation is selected as default.
    AMCLIB_SpeedLoopInit(&SpeedLoop);

    // Initialize the AMCLIB_SpeedLoop state variables to predefined values
    // Warning: Parameters in SpeedLoop must be already initialized.
    f32FilterMAWOut = (tFrac32)123L;
    f32ControllerPIpAWQOut = (tFrac32)123L;
    f32RampOut = (tFrac32)123L;
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB_SpeedLoopSetState_F32(f32FilterMAWOut, f32ControllerPIpAWQOut,
        f32RampOut, &SpeedLoop);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB_SpeedLoopSetState(f32FilterMAWOut, f32ControllerPIpAWQOut,
        f32RampOut, &SpeedLoop, F32);
```

```

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 32-bit fractional implementation is selected as default.
AMCLIB_SpeedLoopSetState(f32FilterMAWOut, f32ControllerPIpAWQOut,
    f32RampOut, &SpeedLoop);

f32VelocityReq = (tFrac32)100L;
while(1);
}

// Periodical function or interrupt - speed control loop
void SlowLoop(void)
{
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB_SpeedLoop_F32(f32VelocityReq, f32VelocityFbck,
        &IDQReq, &SpeedLoop);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB_SpeedLoop(f32VelocityReq, f32VelocityFbck,
        &IDQReq, &SpeedLoop, F32);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 32-bit fractional implementation is selected as default.
    AMCLIB_SpeedLoop(f32VelocityReq, f32VelocityFbck,
        &IDQReq, &SpeedLoop);
}

```

## 2.6.2 Function AMCLIB\_SpeedLoop\_F16

### Declaration

```
void AMCLIB_SpeedLoop_F16(tFrac16 f16VelocityReq,
tFrac16 f16VelocityFbck, SWLIBS\_2Syst\_F16 *const pIDQReq,
AMCLIB\_SPEED\_LOOP\_T\_F16 *pCtrl);
```

### Arguments

**Table 55. AMCLIB\_SpeedLoop\_F16 arguments**

Type	Name	Direction	Description
<a href="#">tFrac16</a>	f16VelocityReq	input	Required electrical angular velocity (setpoint).
<a href="#">tFrac16</a>	f16VelocityFbck	input	Actual electrical angular velocity from the feedback.
<a href="#">SWLIBS_2Syst_F16</a> *const	pIDQReq	output	Pointer to the structure with the required stator currents in the two-phase rotational orthogonal system (d-q). Only the q-axis component is written by the function.
<a href="#">AMCLIB_SPEED_LOOP_T_F16</a> *	pCtrl	input, output	Pointer to the structure with AMCLIB_SpeedLoop state.

### Implementation details

The following block diagram shows the internal functions and data structures of AMCLIB\_SpeedLoop\_F16.

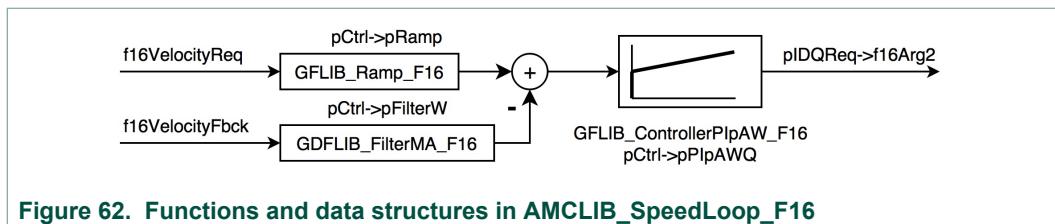


Figure 62. Functions and data structures in AMCLIB\_SpeedLoop\_F16

Prior to calculating the controller coefficients, it is necessary to set the scaling constants. All inputs and outputs of the algorithm are limited to the fractional range [-1, 1). Incorrect setting of the scaling constants may lead to undesirable overflow or saturation during the computation.

Table 56. Scaling constants

Scaling constant	Symbol	Calculation
Maximum phase current [A]	$I_{MAX}$	Maximum current depends on power stage capabilities.
Maximum speed [rad/s]	$\Omega_{MAX}$	Maximum application required speed, at least the motor electrical rated speed.

Parameters of the speed PlpAW controller (using bilinear transform) can be calculated using the following equations:

$$\begin{aligned}
 pPlpAWQ.f16PropGain &= FRAC16\left(\left(2 \cdot \xi \cdot \omega_0 \cdot \frac{J}{K_T}\right) \cdot \frac{\Omega_{MAX}}{T_{MAX}} \cdot 2^{NShiftP}\right) \\
 pPlpAWQ.f16IntegGain &= FRAC16\left(\left(\omega_0^2 \cdot \frac{J}{K_T} \cdot \frac{T_s}{2}\right) \cdot \frac{\Omega_{MAX}}{T_{MAX}} \cdot 2^{NShiftI}\right) \\
 pPlpAWQs16PropGainShift &= NShiftP \\
 pPlpAWQs16IntegGainShift &= NShiftI \\
 pPlpAWQ.f16UpperLimit &= FRAC16(1) \\
 pPlpAWQ.f16LowerLimit &= FRAC16(-1) \\
 \text{Equation AMCLIB_SpeedLoop_F16_Eq1}
 \end{aligned}$$

where  $\xi$  is the speed loop attenuation,  $\omega_0$  is the speed loop natural frequency [rad/s],  $J$  is the moment of inertia,  $K_T$  is the motor torque constant, and  $T_S$  is the sampling period.  $NShiftP$  and  $NShiftI$  are integer values which ensure that the controller coefficients fit in the fractional range [-1, 1].

The smoothing factor of the [GDFLIB\\_FilterMA\\_F16](#) and the slope of the [GFLIB\\_Ramp\\_F16](#) on the input of the speed controller should reflect the achievable dynamics of the drive. The [AN4642](#) (Motor Control Application Tuning (MCAT) Tool for 3-Phase PMSM) can help with tuning of these parameters.

**Note:** Due to effectivity reasons, this function is implemented using inline assembly and is therefore not ANSI-C compliant.

### Code Example

```

#include "amclib.h"

AMCLIB_SPEED_LOOP_T_F16 SpeedLoop;
SWLIBS_2Syst_F16 IDQReq;           // required dq currents
tFrac16 f16VelocityReq;           // required velocity
tFrac16 f16VelocityFbck;          // actual velocity
  
```

```
void main (void)
{
    tFrac16 f16FilterMAWOut;
    tFrac16 f16ControllerPIpAWQOut;
    tFrac32 f32RampOut;

    // Initialize the parameters in SpeedLoop
    SpeedLoop.pFilterW.ul6NSamples      = 3u;
    SpeedLoop.pPIpAWQ.f16PropGain       = (tFrac16)FRAC16(0.1234);
    SpeedLoop.pPIpAWQ.f16IntegGain     = (tFrac16)FRAC16(0.1675);
    SpeedLoop.pPIpAWQ.s16PropGainShift = 1;
    SpeedLoop.pPIpAWQ.s16IntegGainShift = 1;
    SpeedLoop.pPIpAWQ.f16LowerLimit    = (tFrac16)-30621;
    SpeedLoop.pPIpAWQ.f16UpperLimit    = (tFrac16)32066;
    SpeedLoop.pRamp.f32RampUp          = (tFrac32)FRAC32(0.4768);
    SpeedLoop.pRamp.f32RampDown        = (tFrac32)FRAC32(0.3754);

    // Clear AMCLIB_SpeedLoop state variables
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB_SpeedLoopInit_F16(&SpeedLoop);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB_SpeedLoopInit(&SpeedLoop, F16);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 16-bit fractional implementation is selected as default.
    AMCLIB_SpeedLoopInit(&SpeedLoop);

    // Initialize the AMCLIB_SpeedLoop state variables to predefined values
    // Warning: Parameters in SpeedLoop must be already initialized.
    f16FilterMAWOut = (tFrac16)123L;
    f16ControllerPIpAWQOut = (tFrac16)123L;
    f32RampOut = (tFrac32)123L;
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB_SpeedLoopSetState_F16(f16FilterMAWOut, f16ControllerPIpAWQOut,
                                 f32RampOut, &SpeedLoop);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB_SpeedLoopSetState(f16FilterMAWOut, f16ControllerPIpAWQOut,
                            f32RampOut, &SpeedLoop, F16);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 16-bit fractional implementation is selected as default.
    AMCLIB_SpeedLoopSetState(f16FilterMAWOut, f16ControllerPIpAWQOut,
                            f32RampOut, &SpeedLoop);

    f16VelocityReq = (tFrac16)100;
    while(1);
}

// Periodical function or interrupt - speed control loop
void SlowLoop(void)
{
```

```

// Alternative 1: API call with postfix
// (only one alternative shall be used).
AMCLIB_SpeedLoop_F16(f16VelocityReq, f16VelocityFbck,
    &IDQReq, &SpeedLoop);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
AMCLIB_SpeedLoop(f16VelocityReq, f16VelocityFbck,
    &IDQReq, &SpeedLoop, F16);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 16-bit fractional implementation is selected as default.
AMCLIB_SpeedLoop(f16VelocityReq, f16VelocityFbck,
    &IDQReq, &SpeedLoop);
}

```

### 2.6.3 Function AMCLIB\_SpeedLoop\_FLT

#### Declaration

```
void AMCLIB_SpeedLoop_FLT(tFloat fltVelocityReq, tFloat
    fltVelocityFbck, SWLIBS\_2Syst\_FLT *const pIDQReq,
AMCLIB\_SPEED\_LOOP\_T\_FLT *pCtrl);
```

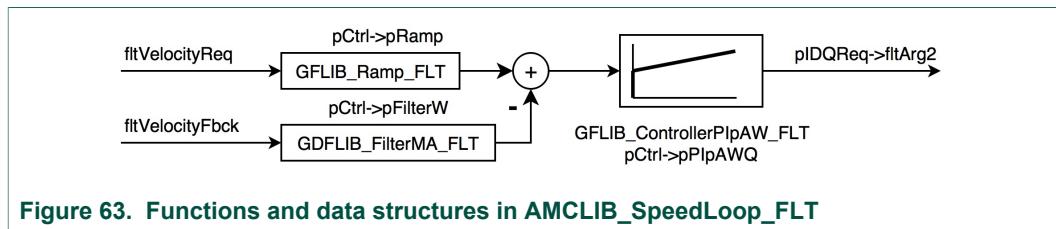
#### Arguments

**Table 57. AMCLIB\_SpeedLoop\_FLT arguments**

Type	Name	Direction	Description
<a href="#">tFloat</a>	fltVelocityReq	input	Required electrical angular velocity (setpoint).
<a href="#">tFloat</a>	fltVelocityFbck	input	Actual electrical angular velocity from the feedback.
<a href="#">SWLIBS_2Syst_FLT</a> *const	pIDQReq	output	Pointer to the structure with the required stator currents in the two-phase rotational orthogonal system (d-q). Only the q-axis component is written by the function.
<a href="#">AMCLIB_SPEED_LOOP_T_FLT</a> *	pCtrl	input, output	Pointer to the structure with AMCLIB_SpeedLoop state.

#### Implementation details

The following block diagram shows the internal functions and data structures of AMCLIB\_SpeedLoop\_FLT.



**Figure 63. Functions and data structures in AMCLIB\_SpeedLoop\_FLT**

Parameters of the speed PlpAW controller (using bilinear transform) can be calculated using the following equations:

$$pPIpAWQ.fltPropGain = 2 \cdot \xi \cdot \omega_0 \cdot \frac{J}{K_T}$$

$$pPIpAWQ.fltIntegGain = \omega_0^2 \cdot \frac{J}{K_T} \cdot \frac{T_s}{2}$$

$$pPIpAWQ.fltUpperLimit = I_{MAX}$$

$$pPIpAWQ.fltLowerLimit = -I_{MAX}$$

Equation AMCLIB\_SpeedLoop\_FLT\_Eq1

where  $\xi$  is the speed loop attenuation,  $\omega_0$  is the speed loop natural frequency [rad/s],  $J$  is the moment of inertia,  $K_T$  is the motor torque constant,  $T_S$  is the sampling period, and  $I_{MAX}$  is the maximum phase current [A].

The smoothing factor of the [GDFLIB\\_FilterMA\\_FLT](#) and the slope of the [GFLIB\\_Ramp\\_FLT](#) on the input of the speed controller should reflect the achievable dynamics of the drive. The [AN4642](#) (Motor Control Application Tuning (MCAT) Tool for 3-Phase PMSM) can help with tuning of these parameters.

**Note:** The function may raise floating-point exceptions (invalid operation, overflow, underflow, inexact, input denormal). The floating-point unit must be enabled in CPACR register to prevent NOCP UsageFault exception.

### Code Example

```
#include "amclib.h"

AMCLIB_SPEED_LOOP_T_FLT SpeedLoop;
SWLIBS_2Syst_FLT IDQReq;           // required dq currents
tFloat fltVelocityReq;             // required velocity
tFloat fltVelocityFbck;            // actual velocity

void main (void)
{
    tFloat fltFilterMAWOut;
    tFloat fltControllerPIpAWQOut;
    tFloat fltRampOut;

    // Initialize the parameters in SpeedLoop
    SpeedLoop.pFilterW.fltLambda      = (tFloat) 0.5;
    SpeedLoop.pPIpAWQ.fltPropGain     = (tFloat) 0.1234;
    SpeedLoop.pPIpAWQ.fltIntegGain    = (tFloat) 0.1675;
    SpeedLoop.pPIpAWQ.fltLowerLimit   = (tFloat) -0.934499979019165;
    SpeedLoop.pPIpAWQ.fltUpperLimit   = (tFloat) 0.978600025177002;
    SpeedLoop.pRamp.fltRampUp         = (tFloat) 0.4768;
    SpeedLoop.pRamp.fltRampDown       = (tFloat) 0.3754;

    // Clear AMCLIB_SpeedLoop state variables
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB_SpeedLoopInit_FLT(&SpeedLoop);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB_SpeedLoopInit(&SpeedLoop, FLT);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if single precision floating point implementation is selected
```

```
// as default.
AMCLIB_SpeedLoopInit(&SpeedLoop);

// Initialize the AMCLIB_SpeedLoop state variables to predefined values
// Warning: Parameters in SpeedLoop must be already initialized.
fltFilterMAWOut = 123.0F;
fltControllerPIpAWQOut = 123.0F;
fltRampOut = 123.0F;
// Alternative 1: API call with postfix
// (only one alternative shall be used).
AMCLIB_SpeedLoopSetState_FLT(fltFilterMAWOut, fltControllerPIpAWQOut,
    fltRampOut, &SpeedLoop);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
AMCLIB_SpeedLoopSetState(fltFilterMAWOut, fltControllerPIpAWQOut,
    fltRampOut, &SpeedLoop, FLT);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if single precision floating point implementation is selected
// as default.
AMCLIB_SpeedLoopSetState(fltFilterMAWOut, fltControllerPIpAWQOut,
    fltRampOut, &SpeedLoop);

fltVelocityReq = 100.0F;
while(1);
}

// Periodical function or interrupt - speed control loop
void SlowLoop(void)
{
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB_SpeedLoop_FLT(fltVelocityReq, fltVelocityFbck,
        &IDQReq, &SpeedLoop);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB_SpeedLoop(fltVelocityReq, fltVelocityFbck,
        &IDQReq, &SpeedLoop, FLT);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if single precision floating point implementation is selected
    // as default.
    AMCLIB_SpeedLoop(fltVelocityReq, fltVelocityFbck,
        &IDQReq, &SpeedLoop);
}
```

## 2.6.4 Function AMCLIB\_SpeedLoopInit

### Description

This function clears the AMCLIB\_SpeedLoop state variables.

**Note:** The input/output pointers must contain valid addresses, otherwise a fault may occur (MemManage, BusFault, UsageFault, HardFault).

**Re-entrancy**

The function is re-entrant for a different pCtrl.

**2.6.4.1 Function AMCLIB\_SpeedLoopInit\_F32****Declaration**

```
void AMCLIB_SpeedLoopInit_F32(AMCLIB\_SPEED\_LOOP\_T\_F32 *const pCtrl);
```

**Arguments****Table 58. AMCLIB\_SpeedLoopInit\_F32 arguments**

Type	Name	Direction	Description
<a href="#">AMCLIB_SPEED_LOOP_T_F32</a> *const	pCtrl	input, output	Pointer to the structure with AMCLIB_SpeedLoop state.

**Note:** If pCtrl points to a structure of type [AMCLIB\\_SPEED\\_LOOP\\_DEBUG\\_T\\_F32](#), it must be recasted to [AMCLIB\\_SPEED\\_LOOP\\_T\\_F32](#).

**Code Example**

```
#include "amclib.h"

AMCLIB\_SPEED\_LOOP\_T\_F32 SpeedLoop;
SWLIBS\_2Syst\_F32 IDQReq;           // required dq currents
tFrac32 f32VelocityReq;        // required velocity
tFrac32 f32VelocityFbck;       // actual velocity

void main (void)
{
    tFrac32 f32FilterMAWOut;
    tFrac32 f32ControllerPIpAWQOut;
    tFrac32 f32RampOut;

    // Initialize the parameters in SpeedLoop
    SpeedLoop.pFilterW.u16NSamples      = 3u;
    SpeedLoop.pPIpAWQ.f32PropGain       = (tFrac32)FRAC32(0.1234);
    SpeedLoop.pPIpAWQ.f32IntegGain     = (tFrac32)FRAC32(0.1675);
    SpeedLoop.pPIpAWQ.s16PropGainShift = 1;
    SpeedLoop.pPIpAWQ.s16IntegGainShift = 1;
    SpeedLoop.pPIpAWQ.f32LowerLimit   = (tFrac32) -2006823469L;
    SpeedLoop.pPIpAWQ.f32UpperLimit   = (tFrac32) 2101527497L;
    SpeedLoop.pRamp.f32RampUp         = (tFrac32)FRAC32(0.4768);
    SpeedLoop.pRamp.f32RampDown       = (tFrac32)FRAC32(0.3754);

    // Clear AMCLIB_SpeedLoop state variables
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB_SpeedLoopInit_F32(&SpeedLoop);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB_SpeedLoopInit(&SpeedLoop, F32);

    // Alternative 3: API call with global configuration of implementation
```

```
// (only one alternative shall be used). This alternative is available
// only if 32-bit fractional implementation is selected as default.
AMCLIB_SpeedLoopInit(&SpeedLoop);

// Initialize the AMCLIB_SpeedLoop state variables to predefined values
// Warning: Parameters in SpeedLoop must be already initialized.
f32FilterMAWOut = (tFrac32)123L;
f32ControllerPIpAWQOut = (tFrac32)123L;
f32RampOut = (tFrac32)123L;
// Alternative 1: API call with postfix
// (only one alternative shall be used).
AMCLIB_SpeedLoopSetState_F32(f32FilterMAWOut, f32ControllerPIpAWQOut,
    f32RampOut, &SpeedLoop);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
AMCLIB_SpeedLoopSetState(f32FilterMAWOut, f32ControllerPIpAWQOut,
    f32RampOut, &SpeedLoop, F32);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 32-bit fractional implementation is selected as default.
AMCLIB_SpeedLoopSetState(f32FilterMAWOut, f32ControllerPIpAWQOut,
    f32RampOut, &SpeedLoop);

f32VelocityReq = (tFrac32)100L;
while(1);
}

// Periodical function or interrupt - speed control loop
void SlowLoop(void)
{
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB_SpeedLoop_F32(f32VelocityReq, f32VelocityFbck,
        &IDQReq, &SpeedLoop);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB_SpeedLoop(f32VelocityReq, f32VelocityFbck,
        &IDQReq, &SpeedLoop, F32);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 32-bit fractional implementation is selected as default.
    AMCLIB_SpeedLoop(f32VelocityReq, f32VelocityFbck,
        &IDQReq, &SpeedLoop);
}
```

### 2.6.4.2 Function AMCLIB\_SpeedLoopInit\_F16

#### Declaration

```
void AMCLIB_SpeedLoopInit_F16(AMCLIB_SPEED_LOOP_T_F16 *const
    pCtrl);
```

## Arguments

**Table 59. AMCLIB\_SpeedLoopInit\_F16 arguments**

Type	Name	Direction	Description
<a href="#">AMCLIB_SPEED_LOOP_T_F16</a> *const	pCtrl	input, output	Pointer to the structure with AMCLIB_SpeedLoop state.

**Note:** If pCtrl points to a structure of type [AMCLIB\\_SPEED\\_LOOP\\_DEBUG\\_T\\_F16](#), it must be recasted to [AMCLIB\\_SPEED\\_LOOP\\_T\\_F16](#).

## Code Example

```
#include "amclib.h"

AMCLIB_SPEED_LOOP_T_F16 SpeedLoop;
SWLIBS_2Syst_F16 IDQReq;           // required dq currents
tFrac16 f16VelocityReq;          // required velocity
tFrac16 f16VelocityFbck;         // actual velocity

void main (void)
{
    tFrac16 f16FilterMAWOut;
    tFrac16 f16ControllerPIpAWQOut;
    tFrac32 f32RampOut;

    // Initialize the parameters in SpeedLoop
    SpeedLoop.pFilterW.u16NSamples      = 3u;
    SpeedLoop.pPIpAWQ.f16PropGain       = (tFrac16)FRAC16(0.1234);
    SpeedLoop.pPIpAWQ.f16IntegGain      = (tFrac16)FRAC16(0.1675);
    SpeedLoop.pPIpAWQ.s16PropGainShift  = 1;
    SpeedLoop.pPIpAWQ.s16IntegGainShift = 1;
    SpeedLoop.pPIpAWQ.f16LowerLimit    = (tFrac16)-30621;
    SpeedLoop.pPIpAWQ.f16UpperLimit    = (tFrac16)32066;
    SpeedLoop.pRamp.f32RampUp          = (tFrac32)FRAC32(0.4768);
    SpeedLoop.pRamp.f32RampDown        = (tFrac32)FRAC32(0.3754);

    // Clear AMCLIB_SpeedLoop state variables
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB_SpeedLoopInit_F16(&SpeedLoop);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB_SpeedLoopInit(&SpeedLoop, F16);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 16-bit fractional implementation is selected as default.
    AMCLIB_SpeedLoopInit(&SpeedLoop);

    // Initialize the AMCLIB_SpeedLoop state variables to predefined values
    // Warning: Parameters in SpeedLoop must be already initialized.
    f16FilterMAWOut = (tFrac16)123L;
    f16ControllerPIpAWQOut = (tFrac16)123L;
    f32RampOut = (tFrac32)123L;
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB\_SpeedLoopSetState\_F16(f16FilterMAWOut, f16ControllerPIpAWQOut,
```

```

f32RampOut, &SpeedLoop);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
AMCLIB_SpeedLoopSetState(f16FilterMAWOut, f16ControllerPIpAWQOut,
    f32RampOut, &SpeedLoop, F16);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 16-bit fractional implementation is selected as default.
AMCLIB_SpeedLoopSetState(f16FilterMAWOut, f16ControllerPIpAWQOut,
    f32RampOut, &SpeedLoop);

f16VelocityReq = (tFrac16)100;
while(1);
}

// Periodical function or interrupt - speed control loop
void SlowLoop(void)
{
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB\_SpeedLoop\_F16(f16VelocityReq, f16VelocityFbck,
        &IDQReq, &SpeedLoop);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB_SpeedLoop(f16VelocityReq, f16VelocityFbck,
        &IDQReq, &SpeedLoop);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 16-bit fractional implementation is selected as default.
    AMCLIB_SpeedLoop(f16VelocityReq, f16VelocityFbck,
        &IDQReq, &SpeedLoop);
}

```

#### 2.6.4.3 Function [AMCLIB\\_SpeedLoopInit\\_FLT](#)

##### Declaration

```
void AMCLIB_SpeedLoopInit_FLT(AMCLIB\_SPEED\_LOOP\_T\_FLT *const pCtrl);
```

##### Arguments

**Table 60. [AMCLIB\\_SpeedLoopInit\\_FLT](#) arguments**

Type	Name	Direction	Description
<a href="#">AMCLIB_SPEED_LOOP_T_FLT</a> *const	pCtrl	input, output	Pointer to the structure with <a href="#">AMCLIB_SpeedLoop</a> state.

**Note:** If pCtrl points to a structure of type [AMCLIB\\_SPEED\\_LOOP\\_DEBUG\\_T\\_FLT](#), it must be recasted to [AMCLIB\\_SPEED\\_LOOP\\_T\\_FLT](#).

##### Code Example

```
#include "amclib.h"
```

```
AMCLIB_SPEED_LOOP_T_FLT SpeedLoop;
SWLIBS_2Syst_FLT IDQReq;           // required dq currents
tFloat fltVelocityReq;            // required velocity
tFloat fltVelocityFbck;          // actual velocity

void main (void)
{
    tFloat fltFilterMAWOut;
    tFloat fltControllerPIpAWQOut;
    tFloat fltRampOut;

    // Initialize the parameters in SpeedLoop
    SpeedLoop.pFilterW.fltLambda      = (tFloat)0.5;
    SpeedLoop.pPIpAWQ.fltPropGain     = (tFloat)0.1234;
    SpeedLoop.pPIpAWQ.fltIntegGain    = (tFloat)0.1675;
    SpeedLoop.pPIpAWQ.fltLowerLimit   = (tFloat)-0.934499979019165;
    SpeedLoop.pPIpAWQ.fltUpperLimit   = (tFloat)0.978600025177002;
    SpeedLoop.pRamp.fltRampUp         = (tFloat)0.4768;
    SpeedLoop.pRamp.fltRampDown       = (tFloat)0.3754;

    // Clear AMCLIB_SpeedLoop state variables
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB_SpeedLoopInit_FLT(&SpeedLoop);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB_SpeedLoopInit(&SpeedLoop, FLT);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if single precision floating point implementation is selected
    // as default.
    AMCLIB_SpeedLoopInit(&SpeedLoop);

    // Initialize the AMCLIB_SpeedLoop state variables to predefined values
    // Warning: Parameters in SpeedLoop must be already initialized.
    fltFilterMAWOut = 123.0F;
    fltControllerPIpAWQOut = 123.0F;
    fltRampOut = 123.0F;
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB_SpeedLoopSetState_FLT(fltFilterMAWOut, fltControllerPIpAWQOut,
        fltRampOut, &SpeedLoop);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB_SpeedLoopSetState(fltFilterMAWOut, fltControllerPIpAWQOut,
        fltRampOut, &SpeedLoop, FLT);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if single precision floating point implementation is selected
    // as default.
    AMCLIB_SpeedLoopSetState(fltFilterMAWOut, fltControllerPIpAWQOut,
        fltRampOut, &SpeedLoop);

    fltVelocityReq = 100.0F;
    while(1);
}
```

```

}

// Periodical function or interrupt - speed control loop
void SlowLoop(void)
{
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB_SpeedLoop_FLT(fltVelocityReq, fltVelocityFbck,
        &IDQReq, &SpeedLoop);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB_SpeedLoop(fltVelocityReq, fltVelocityFbck,
        &IDQReq, &SpeedLoop, FLT);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if single precision floating point implementation is selected
    // as default.
    AMCLIB_SpeedLoop(fltVelocityReq, fltVelocityFbck,
        &IDQReq, &SpeedLoop);
}

```

## 2.6.5 Function AMCLIB\_SpeedLoopSetState

### Description

This function initializes the AMCLIB\_SpeedLoop state variables to achieve the required output values.

**Note:** The input/output pointers must contain valid addresses, otherwise a fault may occur (MemManage, BusFault, UsageFault, HardFault).

### Re-entrancy

The function is re-entrant for a different pCtrl.

#### 2.6.5.1 Function AMCLIB\_SpeedLoopSetState\_F32

##### Declaration

```
void AMCLIB_SpeedLoopSetState_F32(tFrac32 f32FilterMAWOut,
tFrac32 f32ControllerPipAWQOut, tFrac32 f32RampOut,
AMCLIB_SPEED_LOOP_T_F32 *pCtrl);
```

##### Arguments

Table 61. AMCLIB\_SpeedLoopSetState\_F32 arguments

Type	Name	Direction	Description
<u>tFrac32</u>	f32FilterMAWOut	input	Required output of the FilterMA.
<u>tFrac32</u>	f32ControllerPipAWQOut	input	Required output of the ControllerPipAW.
<u>tFrac32</u>	f32RampOut	input	Required output of the speed ramp.
<u>AMCLIB_SPEED_LOOP_T_F32</u> *	pCtrl	input, output	Pointer to the structure with AMCLIB_SpeedLoop state.

**Caution:** Set the parameters in the structure pointed to by pCtrl before calling this function.

**Note:** If pCtrl points to a structure of type [AMCLIB\\_SPEED\\_LOOP\\_DEBUG\\_T\\_F32](#), it must be recasted to [AMCLIB\\_SPEED\\_LOOP\\_T\\_F32](#) \*.

### Code Example

```
#include "amclib.h"

AMCLIB_SPEED_LOOP_T_F32 SpeedLoop;
SWLIBS_2Syst_F32 IDQReq;           // required dq currents
tFrac32 f32VelocityReq;          // required velocity
tFrac32 f32VelocityFbck;         // actual velocity

void main (void)
{
    tFrac32 f32FilterMAWOut;
    tFrac32 f32ControllerPIpAWQOut;
    tFrac32 f32RampOut;

    // Initialize the parameters in SpeedLoop
    SpeedLoop.pFilterW.u16NSamples      = 3u;
    SpeedLoop.pPIpAWQ.f32PropGain       = (tFrac32)FRAC32(0.1234);
    SpeedLoop.pPIpAWQ.f32IntegGain     = (tFrac32)FRAC32(0.1675);
    SpeedLoop.pPIpAWQ.s16PropGainShift = 1;
    SpeedLoop.pPIpAWQ.s16IntegGainShift = 1;
    SpeedLoop.pPIpAWQ.f32LowerLimit   = (tFrac32)-2006823469L;
    SpeedLoop.pPIpAWQ.f32UpperLimit   = (tFrac32)2101527497L;
    SpeedLoop.pRamp.f32RampUp         = (tFrac32)FRAC32(0.4768);
    SpeedLoop.pRamp.f32RampDown       = (tFrac32)FRAC32(0.3754);

    // Clear AMCLIB_SpeedLoop state variables
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB\_SpeedLoopInit\_F32(&SpeedLoop);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB\_SpeedLoopInit\(&SpeedLoop, F32\);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 32-bit fractional implementation is selected as default.
    AMCLIB\_SpeedLoopInit\(&SpeedLoop\);

    // Initialize the AMCLIB_SpeedLoop state variables to predefined values
    // Warning: Parameters in SpeedLoop must be already initialized.
    f32FilterMAWOut = (tFrac32)123L;
    f32ControllerPIpAWQOut = (tFrac32)123L;
    f32RampOut = (tFrac32)123L;
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB\_SpeedLoopSetState\_F32\(f32FilterMAWOut, f32ControllerPIpAWQOut,
        f32RampOut, &SpeedLoop\);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
```

```

AMCLIB_SpeedLoopSetState(f32FilterMAWOut, f32ControllerPIpAWQOut,
    f32RampOut, &SpeedLoop, F32);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 32-bit fractional implementation is selected as default.
AMCLIB_SpeedLoopSetState(f32FilterMAWOut, f32ControllerPIpAWQOut,
    f32RampOut, &SpeedLoop);

f32VelocityReq = (tFrac32)100L;
while(1);
}

// Periodical function or interrupt - speed control loop
void SlowLoop(void)
{
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB\_SpeedLoop\_F32(f32VelocityReq, f32VelocityFbck,
        &IDQReq, &SpeedLoop);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB_SpeedLoop(f32VelocityReq, f32VelocityFbck,
        &IDQReq, &SpeedLoop);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 32-bit fractional implementation is selected as default.
    AMCLIB_SpeedLoop(f32VelocityReq, f32VelocityFbck,
        &IDQReq, &SpeedLoop);
}

```

### 2.6.5.2 Function [AMCLIB\\_SpeedLoopSetState\\_F16](#)

#### Declaration

```
void AMCLIB_SpeedLoopSetState_F16(tFrac16 f16FilterMAWOut,
    tFrac16 f16ControllerPIpAWQOut, tFrac32 f32RampOut,
    AMCLIB\_SPEED\_LOOP\_T\_F16 *pCtrl);
```

#### Arguments

**Table 62. AMCLIB\_SpeedLoopSetState\_F16 arguments**

Type	Name	Direction	Description
<a href="#">tFrac16</a>	f16FilterMAWOut	input	Required output of the FilterMA.
<a href="#">tFrac16</a>	f16ControllerPIpAWQOut	input	Required output of the ControllerPIpAW.
<a href="#">tFrac32</a>	f32RampOut	input	Required output of the speed ramp.
<a href="#">AMCLIB_SPEED_LOOP_T_F16</a> *	pCtrl	input, output	Pointer to the structure with AMCLIB_SpeedLoop state.

**Caution:** Set the parameters in the structure pointed to by pCtrl before calling this function.

**Note:** If pCtrl points to a structure of type [AMCLIB\\_SPEED\\_LOOP\\_DEBUG\\_T\\_F16](#), it must be recasted to [AMCLIB\\_SPEED\\_LOOP\\_T\\_F16](#) \*.

## Code Example

```
#include "amclib.h"

AMCLIB_SPEED_LOOP_T_F16 SpeedLoop;
SWLIBS_2Syst_F16 IDQReq;           // required dq currents
tFrac16 f16VelocityReq;          // required velocity
tFrac16 f16VelocityFbck;         // actual velocity

void main (void)
{
    tFrac16 f16FilterMAWOut;
    tFrac16 f16ControllerPIpAWQOut;
    tFrac32 f32RampOut;

    // Initialize the parameters in SpeedLoop
    SpeedLoop.pFilterW.u16NSamples      = 3u;
    SpeedLoop.pPIpAWQ.f16PropGain       = (tFrac16)FRAC16(0.1234);
    SpeedLoop.pPIpAWQ.f16IntegGain     = (tFrac16)FRAC16(0.1675);
    SpeedLoop.pPIpAWQ.s16PropGainShift = 1;
    SpeedLoop.pPIpAWQ.s16IntegGainShift = 1;
    SpeedLoop.pPIpAWQ.f16LowerLimit   = (tFrac16)-30621;
    SpeedLoop.pPIpAWQ.f16UpperLimit   = (tFrac16)32066;
    SpeedLoop.pRamp.f32RampUp         = (tFrac32)FRAC32(0.4768);
    SpeedLoop.pRamp.f32RampDown        = (tFrac32)FRAC32(0.3754);

    // Clear AMCLIB_SpeedLoop state variables
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB_SpeedLoopInit_F16(&SpeedLoop);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB_SpeedLoopInit(&SpeedLoop, F16);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 16-bit fractional implementation is selected as default.
    AMCLIB_SpeedLoopInit(&SpeedLoop);

    // Initialize the AMCLIB_SpeedLoop state variables to predefined values
    // Warning: Parameters in SpeedLoop must be already initialized.
    f16FilterMAWOut = (tFrac16)123L;
    f16ControllerPIpAWQOut = (tFrac16)123L;
    f32RampOut = (tFrac32)123L;
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB_SpeedLoopSetState_F16(f16FilterMAWOut, f16ControllerPIpAWQOut,
                                  f32RampOut, &SpeedLoop);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB_SpeedLoopSetState(f16FilterMAWOut, f16ControllerPIpAWQOut,
                            f32RampOut, &SpeedLoop, F16);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 16-bit fractional implementation is selected as default.
    AMCLIB_SpeedLoopSetState(f16FilterMAWOut, f16ControllerPIpAWQOut,
```

```

f32RampOut, &SpeedLoop);

f16VelocityReq = (tFrac16)100;
while(1);
}

// Periodical function or interrupt - speed control loop
void SlowLoop(void)
{
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB\_SpeedLoop\_F16(f16VelocityReq, f16VelocityFbck,
        &IDQReq, &SpeedLoop);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB\_SpeedLoop(f16VelocityReq, f16VelocityFbck,
        &IDQReq, &SpeedLoop, F16);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 16-bit fractional implementation is selected as default.
    AMCLIB\_SpeedLoop(f16VelocityReq, f16VelocityFbck,
        &IDQReq, &SpeedLoop);
}

```

### 2.6.5.3 Function [AMCLIB\\_SpeedLoopSetState\\_FLT](#)

#### Declaration

```
void AMCLIB\_SpeedLoopSetState\_FLT(tFloat fltFilterMAWOut,
    tFloat fltControllerPipAWQOut, tFloat fltRampOut,
    AMCLIB\_SPEED\_LOOP\_T\_FLT *pCtrl);
```

#### Arguments

**Table 63. [AMCLIB\\_SpeedLoopSetState\\_FLT](#) arguments**

Type	Name	Direction	Description
<a href="#">tFloat</a>	fltFilterMAWOut	input	Required output of the FilterMA.
<a href="#">tFloat</a>	fltControllerPipAWQOut	input	Required output of the ControllerPipAW.
<a href="#">tFloat</a>	fltRampOut	input	Required output of the speed ramp.
<a href="#">AMCLIB_SPEED_LOOP_T_FLT</a> *	pCtrl	input, output	Pointer to the structure with <a href="#">AMCLIB_SpeedLoop</a> state.

**Caution:** Set the parameters in the structure pointed to by pCtrl before calling this function.

**Note:** If pCtrl points to a structure of type [AMCLIB\\_SPEED\\_LOOP\\_DEBUG\\_T\\_FLT](#), it must be recasted to [AMCLIB\\_SPEED\\_LOOP\\_T\\_FLT](#) \*.

#### Code Example

```
#include "amclib.h"

AMCLIB\_SPEED\_LOOP\_T\_FLT SpeedLoop;
SWLIBS\_2Syst\_FLT IDQReq;           // required dq currents
```

```
tFloat fltVelocityReq;           // required velocity
tFloat fltVelocityFbck;         // actual velocity

void main (void)
{
    tFloat fltFilterMAWOut;
    tFloat fltControllerPIpAWQOut;
    tFloat fltRampOut;

    // Initialize the parameters in SpeedLoop
    SpeedLoop.pFilterW.fltLambda      = (tFloat)0.5;
    SpeedLoop.pPIpAWQ.fltPropGain     = (tFloat)0.1234;
    SpeedLoop.pPIpAWQ.fltIntegGain    = (tFloat)0.1675;
    SpeedLoop.pPIpAWQ.fltLowerLimit   = (tFloat)-0.934499979019165;
    SpeedLoop.pPIpAWQ.fltUpperLimit   = (tFloat)0.978600025177002;
    SpeedLoop.pRamp.fltRampUp        = (tFloat)0.4768;
    SpeedLoop.pRamp.fltRampDown      = (tFloat)0.3754;

    // Clear AMCLIB_SpeedLoop state variables
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB_SpeedLoopInit_FLT(&SpeedLoop);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB_SpeedLoopInit(&SpeedLoop, FLT);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if single precision floating point implementation is selected
    // as default.
    AMCLIB_SpeedLoopInit(&SpeedLoop);

    // Initialize the AMCLIB_SpeedLoop state variables to predefined values
    // Warning: Parameters in SpeedLoop must be already initialized.
    fltFilterMAWOut = 123.0F;
    fltControllerPIpAWQOut = 123.0F;
    fltRampOut = 123.0F;
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB_SpeedLoopSetState_FLT(fltFilterMAWOut, fltControllerPIpAWQOut,
                                  fltRampOut, &SpeedLoop);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB_SpeedLoopSetState(fltFilterMAWOut, fltControllerPIpAWQOut,
                            fltRampOut, &SpeedLoop, FLT);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if single precision floating point implementation is selected
    // as default.
    AMCLIB_SpeedLoopSetState(fltFilterMAWOut, fltControllerPIpAWQOut,
                            fltRampOut, &SpeedLoop);

    fltVelocityReq = 100.0F;
    while(1);
}

// Periodical function or interrupt - speed control loop
```

```

void SlowLoop(void)
{
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB_SpeedLoop_FLT(fltVelocityReq, fltVelocityFbck,
        &IDQReq, &SpeedLoop);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB_SpeedLoop(fltVelocityReq, fltVelocityFbck,
        &IDQReq, &SpeedLoop, FLT);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if single precision floating point implementation is selected
    // as default.
    AMCLIB_SpeedLoop(fltVelocityReq, fltVelocityFbck,
        &IDQReq, &SpeedLoop);
}

```

## 2.6.6 Function **AMCLIB\_SpeedLoopDebug**

This function adjusts the torque of the motor to achieve the required speed. Debugging information is provided.

### Description

This library function implements the speed control loop which is the outer loop in the cascade control structure of the speed FOC. FOC (also called vector control) is a widely used control strategy for Permanent Magnet Synchronous Motors (PMSM). FOC is characterized by smooth rotation over the entire speed range of the motor, full torque control at zero speed, and fast acceleration/deceleration. A PI controller in speed closed-loop system maintains the required speed by setting the torque producing current in the q-axis.

**AMCLIB\_SpeedLoopDebug** implements the speed controller in the outer control loop highlighted in [Figure 64](#). **AMCLIB\_SpeedLoopDebug** provides the same functionality as [AMCLIB\\_SpeedLoop](#). Additionally, this function allows debugging of all internal variables. The debugging outputs are provided in the structure pointed to by *pCtrl*. Replace **AMCLIB\_SpeedLoopDebug** by [AMCLIB\\_SpeedLoop](#) once the debugging is finished.

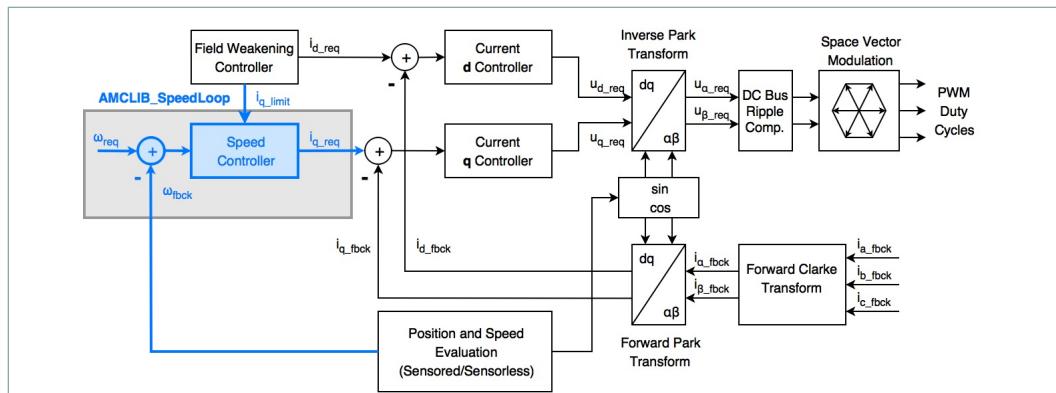


Figure 64. Principal Field Oriented Control scheme with **AMCLIB\_SpeedLoop**

### 2.6.6.1 Function AMCLIB\_SpeedLoopDebug\_F32

#### Declaration

```
void AMCLIB_SpeedLoopDebug_F32(tFrac32 f32VelocityReq,
tFrac32 f32VelocityFbck, SWLIBS_2Syst_F32 *const pIDQReq,
AMCLIB_SPEED_LOOP_DEBUG_T_F32 *pCtrl);
```

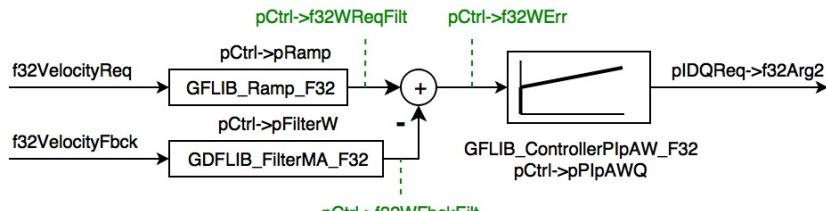
#### Arguments

**Table 64. AMCLIB\_SpeedLoopDebug\_F32 arguments**

Type	Name	Direction	Description
<u>tFrac32</u>	f32VelocityReq	input	Required electrical angular velocity (setpoint).
<u>tFrac32</u>	f32VelocityFbck	input	Actual electrical angular velocity from the feedback.
<u>SWLIBS_2Syst_F32</u> *const	pIDQReq	output	Pointer to the structure with the required stator currents in the two-phase rotational orthogonal system (d-q). Only the q-axis component is written by the function.
<u>AMCLIB_SPEED_LOOP_DEBUG_T_F32</u> *	pCtrl	input, output	Pointer to the structure with AMCLIB_SpeedLoop state and debugging information.

#### Implementation details

The following block diagram shows the internal functions and data structures of AMCLIB\_SpeedLoopDebug\_F32.



**Figure 65. Functions and data structures in AMCLIB\_SpeedLoopDebug\_F32**

Refer to the description of [AMCLIB\\_SpeedLoop\\_F32](#) function on how to set up the controller parameters.

**Note:** Due to effectivity reasons, this function is implemented using inline assembly and is therefore not ANSI-C compliant.

#### Code Example

```
#include "amclib.h"

AMCLIB_SPEED_LOOP_DEBUG_T_F32 SpeedLoop;
SWLIBS_2Syst_F32 IDQReq;           // required dq currents
tFrac32 f32VelocityReq;           // required velocity
tFrac32 f32VelocityFbck;          // actual velocity

void main (void)
{
    tFrac32 f32FilterMAWOut;
```

```
tFrac32 f32ControllerPIpAWQOut;
tFrac32 f32RampOut;

// Initialize the parameters in SpeedLoop
SpeedLoop.pFilterW.ul6NSamples      = 3u;
SpeedLoop.pPIpAWQ.f32PropGain       = (tFrac32)FRAC32(0.1234);
SpeedLoop.pPIpAWQ.f32IntegGain     = (tFrac32)FRAC32(0.1675);
SpeedLoop.pPIpAWQ.s16PropGainShift = 1;
SpeedLoop.pPIpAWQ.s16IntegGainShift = 1;
SpeedLoop.pPIpAWQ.f32LowerLimit    = (tFrac32)-2006823469L;
SpeedLoop.pPIpAWQ.f32UpperLimit    = (tFrac32)2101527497L;
SpeedLoop.pRamp.f32RampUp          = (tFrac32)FRAC32(0.4768);
SpeedLoop.pRamp.f32RampDown        = (tFrac32)FRAC32(0.3754);

// Clear AMCLIB_SpeedLoop state variables
// Alternative 1: API call with postfix
// (only one alternative shall be used).
AMCLIB_SpeedLoopInit_F32((AMCLIB_SPEED_LOOP_T_F32 *)&SpeedLoop);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
AMCLIB_SpeedLoopInit((AMCLIB_SPEED_LOOP_T_F32 *)&SpeedLoop, F32);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 32-bit fractional implementation is selected as default.
AMCLIB_SpeedLoopInit((AMCLIB_SPEED_LOOP_T_F32 *)&SpeedLoop);

// Initialize the AMCLIB_SpeedLoop state variables to predefined values
// Warning: Parameters in SpeedLoop must be already initialized.
f32FilterMAWOut = (tFrac32)123L;
f32ControllerPIpAWQOut = (tFrac32)123L;
f32RampOut = (tFrac32)123L;
// Alternative 1: API call with postfix
// (only one alternative shall be used).
AMCLIB_SpeedLoopSetState_F32(f32FilterMAWOut, f32ControllerPIpAWQOut,
    f32RampOut, (AMCLIB_SPEED_LOOP_T_F32 *)&SpeedLoop);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
AMCLIB_SpeedLoopSetState(f32FilterMAWOut, f32ControllerPIpAWQOut,
    f32RampOut, (AMCLIB_SPEED_LOOP_T_F32 *)&SpeedLoop, F32);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 32-bit fractional implementation is selected as default.
AMCLIB_SpeedLoopSetState(f32FilterMAWOut, f32ControllerPIpAWQOut,
    f32RampOut, (AMCLIB_SPEED_LOOP_T_F32 *)&SpeedLoop);

f32VelocityReq = (tFrac32)100L;
while(1);

// Periodical function or interrupt - speed control loop
void SlowLoop(void)
{
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB_SpeedLoopDebug_F32(f32VelocityReq, f32VelocityFbck,
        &IDQReq, &SpeedLoop);
```

```

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
AMCLIB_SpeedLoopDebug(f32VelocityReq, f32VelocityFbck,
    &IDQReq, &SpeedLoop, F32);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 32-bit fractional implementation is selected as default.
AMCLIB_SpeedLoopDebug(f32VelocityReq, f32VelocityFbck,
    &IDQReq, &SpeedLoop);
}
  
```

### 2.6.6.2 Function AMCLIB\_SpeedLoopDebug\_F16

#### Declaration

```
void AMCLIB_SpeedLoopDebug_F16(tFrac16 f16VelocityReq,
tFrac16 f16VelocityFbck, SWLIBS\_2Syst\_F16 *const pIDQReq,
AMCLIB\_SPEED\_LOOP\_DEBUG\_T\_F16 *pCtrl);
```

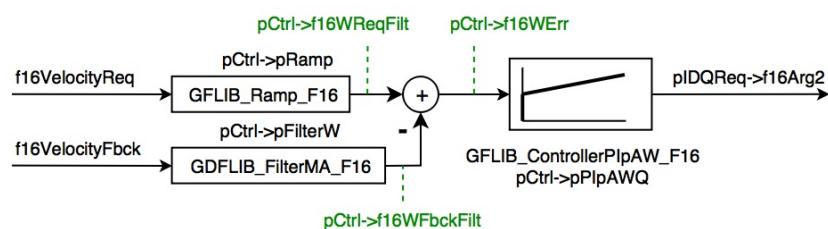
#### Arguments

**Table 65. AMCLIB\_SpeedLoopDebug\_F16 arguments**

Type	Name	Direction	Description
<a href="#">tFrac16</a>	f16VelocityReq	input	Required electrical angular velocity (setpoint).
<a href="#">tFrac16</a>	f16VelocityFbck	input	Actual electrical angular velocity from the feedback.
<a href="#">SWLIBS_2Syst_F16</a> *const	pIDQReq	output	Pointer to the structure with the required stator currents in the two-phase rotational orthogonal system (d-q). Only the q-axis component is written by the function.
<a href="#">AMCLIB_SPEED_LOOP_DEBUG_T_F16</a> *	pCtrl	input, output	Pointer to the structure with AMCLIB_SpeedLoop state and debugging information.

#### Implementation details

The following block diagram shows the internal functions and data structures of AMCLIB\_SpeedLoopDebug\_F16.



**Figure 66. Functions and data structures in AMCLIB\_SpeedLoopDebug\_F16**

Refer to the description of [AMCLIB\\_SpeedLoop\\_F16](#) function on how to set up the controller parameters.

**Note:** Due to effectivity reasons, this function is implemented using inline assembly and is therefore not ANSI-C compliant.

## Code Example

```
#include "amclib.h"

AMCLIB_SPEED_LOOP_DEBUG_T_F16 SpeedLoop;
SWLIBS_2Syst_F16 IDQReq;           // required dq currents
tFrac16 f16VelocityReq;          // required velocity
tFrac16 f16VelocityFbck;         // actual velocity

void main (void)
{
    tFrac16 f16FilterMAWOut;
    tFrac16 f16ControllerPIpAWQOut;
    tFrac32 f32RampOut;

    // Initialize the parameters in SpeedLoop
    SpeedLoop.pFilterW.u16NSamples      = 3u;
    SpeedLoop.pPIpAWQ.f16PropGain       = (tFrac16)FRAC16(0.1234);
    SpeedLoop.pPIpAWQ.f16IntegGain     = (tFrac16)FRAC16(0.1675);
    SpeedLoop.pPIpAWQ.s16PropGainShift = 1;
    SpeedLoop.pPIpAWQ.s16IntegGainShift = 1;
    SpeedLoop.pPIpAWQ.f16LowerLimit   = (tFrac16)-30621;
    SpeedLoop.pPIpAWQ.f16UpperLimit   = (tFrac16)32066;
    SpeedLoop.pRamp.f32RampUp         = (tFrac32)FRAC32(0.4768);
    SpeedLoop.pRamp.f32RampDown        = (tFrac32)FRAC32(0.3754);

    // Clear AMCLIB_SpeedLoop state variables
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB_SpeedLoopInit_F16((AMCLIB_SPEED_LOOP_T_F16 *)&SpeedLoop);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB_SpeedLoopInit((AMCLIB_SPEED_LOOP_T_F16 *)&SpeedLoop, F16);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 16-bit fractional implementation is selected as default.
    AMCLIB_SpeedLoopInit((AMCLIB_SPEED_LOOP_T_F16 *)&SpeedLoop);

    // Initialize the AMCLIB_SpeedLoop state variables to predefined values
    // Warning: Parameters in SpeedLoop must be already initialized.
    f16FilterMAWOut = (tFrac16)123L;
    f16ControllerPIpAWQOut = (tFrac16)123L;
    f32RampOut = (tFrac32)123L;
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB_SpeedLoopSetState_F16(f16FilterMAWOut, f16ControllerPIpAWQOut,
                                  f32RampOut, (AMCLIB_SPEED_LOOP_T_F16 *)&SpeedLoop);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB_SpeedLoopSetState(f16FilterMAWOut, f16ControllerPIpAWQOut,
                            f32RampOut, (AMCLIB_SPEED_LOOP_T_F16 *)&SpeedLoop, F16);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 16-bit fractional implementation is selected as default.
    AMCLIB_SpeedLoopSetState(f16FilterMAWOut, f16ControllerPIpAWQOut,
```

```

f32RampOut, (AMCLIB_SPEED_LOOP_T_F16 *) &SpeedLoop) ;

f16VelocityReq = (tFrac16)100;
while(1);

}

// Periodical function or interrupt - speed control loop
void SlowLoop(void)
{
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB_SpeedLoopDebug_F16(f16VelocityReq, f16VelocityFbck,
        &IDQReq, &SpeedLoop);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB_SpeedLoopDebug(f16VelocityReq, f16VelocityFbck,
        &IDQReq, &SpeedLoop, F16);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 16-bit fractional implementation is selected as default.
    AMCLIB_SpeedLoopDebug(f16VelocityReq, f16VelocityFbck,
        &IDQReq, &SpeedLoop);
}

```

### 2.6.6.3 Function AMCLIB\_SpeedLoopDebug\_FLT

#### Declaration

```
void AMCLIB_SpeedLoopDebug_FLT(tFloat fltVelocityReq,
    tFloat fltVelocityFbck, SWLIBS_2Syst_FLT *const pIDQReq,
    AMCLIB_SPEED_LOOP_DEBUG_T_FLT *pCtrl);
```

#### Arguments

**Table 66. AMCLIB\_SpeedLoopDebug\_FLT arguments**

Type	Name	Direction	Description
tFloat	fltVelocityReq	input	Required electrical angular velocity (setpoint).
tFloat	fltVelocityFbck	input	Actual electrical angular velocity from the feedback.
SWLIBS_2Syst_FLT *const	pIDQReq	output	Pointer to the structure with the required stator currents in the two-phase rotational orthogonal system (d-q). Only the q-axis component is written by the function.
AMCLIB_SPEED_LOOP_DEBUG_T_FLT *	pCtrl	input, output	Pointer to the structure with AMCLIB_SpeedLoop state and debugging information.

#### Implementation details

The following block diagram shows the internal functions and data structures of AMCLIB\_SpeedLoopDebug\_FLT.

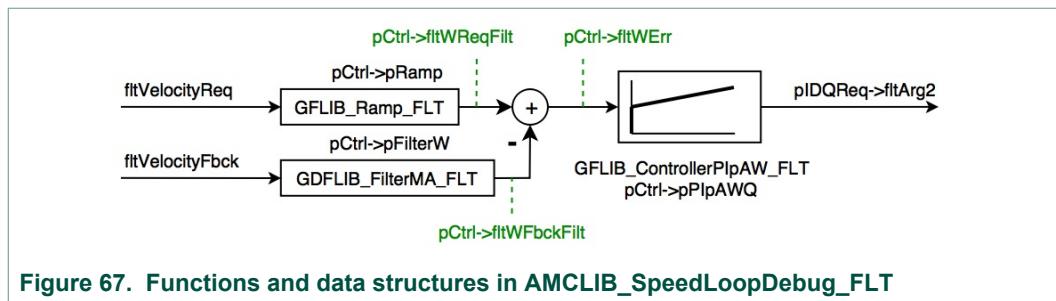


Figure 67. Functions and data structures in AMCLIB\_SpeedLoopDebug\_FLT

Refer to the description of [AMCLIB\\_SpeedLoop\\_FLT](#) function on how to set up the controller parameters.

**Note:** The function may raise floating-point exceptions (invalid operation, overflow, underflow, inexact, input denormal). The floating-point unit must be enabled in CPACR register to prevent NOCP UsageFault exception.

#### Code Example

```
#include "amclib.h"

AMCLIB_SPEED_LOOP_DEBUG_T_FLT SpeedLoop;
SWLIBS_2Syst_FLT IDQReq;           // required dq currents
tFloat fltVelocityReq;             // required velocity
tFloat fltVelocityFbck;            // actual velocity

void main (void)
{
    tFloat fltFilterMAWOut;
    tFloat fltControllerPipAWQOut;
    tFloat fltRampOut;

    // Initialize the parameters in SpeedLoop
    SpeedLoop.pFilterW.fltLambda      = (tFloat)0.5;
    SpeedLoop.pPipAWQ.fltPropGain     = (tFloat)0.1234;
    SpeedLoop.pPipAWQ.fltIntegGain    = (tFloat)0.1675;
    SpeedLoop.pPipAWQ.fltLowerLimit   = (tFloat)-0.934499979019165;
    SpeedLoop.pPipAWQ.fltUpperLimit   = (tFloat)0.978600025177002;
    SpeedLoop.pRamp.fltRampUp         = (tFloat)0.4768;
    SpeedLoop.pRamp.fltRampDown       = (tFloat)0.3754;

    // Clear AMCLIB_SpeedLoop state variables
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB_SpeedLoopInit_FLT((AMCLIB_SPEED_LOOP_T_FLT *)&SpeedLoop);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB_SpeedLoopInit((AMCLIB_SPEED_LOOP_T_FLT *)&SpeedLoop, FLT);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if single precision floating point implementation is selected
    // as default.
    AMCLIB_SpeedLoopInit((AMCLIB_SPEED_LOOP_T_FLT *)&SpeedLoop);

    // Initialize the AMCLIB_SpeedLoop state variables to predefined values
```

```

// Warning: Parameters in SpeedLoop must be already initialized.
fltFilterMAWOut = 123.0F;
fltControllerPIpAWQOut = 123.0F;
fltRampOut = 123.0F;
// Alternative 1: API call with postfix
// (only one alternative shall be used).
AMCLIB_SpeedLoopSetState_FLT(fltFilterMAWOut, fltControllerPIpAWQOut,
    fltRampOut, (AMCLIB_SPEED_LOOP_T_FLT *)&SpeedLoop);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
AMCLIB_SpeedLoopSetState(fltFilterMAWOut, fltControllerPIpAWQOut,
    fltRampOut, (AMCLIB_SPEED_LOOP_T_FLT *)&SpeedLoop, FLT);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if single precision floating point implementation is selected
// as default.
AMCLIB_SpeedLoopSetState(fltFilterMAWOut, fltControllerPIpAWQOut,
    fltRampOut, (AMCLIB_SPEED_LOOP_T_FLT *)&SpeedLoop);

fltVelocityReq = 100.0F;
while(1);
}

// Periodical function or interrupt - speed control loop
void SlowLoop(void)
{
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB_SpeedLoopDebug_FLT(fltVelocityReq, fltVelocityFbck,
        &IDQReq, &SpeedLoop);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB_SpeedLoopDebug(fltVelocityReq, fltVelocityFbck,
        &IDQReq, &SpeedLoop, FLT);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if single precision floating point implementation is selected
    // as default.
    AMCLIB_SpeedLoopDebug(fltVelocityReq, fltVelocityFbck,
        &IDQReq, &SpeedLoop);
}

```

## 2.7 Function AMCLIB\_TrackObsrv

This function calculates the position tracking observer algorithm where the phase-locked loop mechanism is adopted. The input of the function is a phase error from the [AMCLIB\\_BemfObsrvDQ](#) function, representing the state filter providing estimates of the saliency based back-EMF in the estimated  $\gamma$  -  $\delta$  reference frame.

### Description

This function calculates the position tracking observer algorithm, where the phase-locked-loop mechanism is adopted. The input of the function is the phase error from the [AMCLIB\\_BemfObsrvDQ](#) function, representing the state filter providing the estimates

of the saliency based back-EMF in the estimated  $\gamma - \delta$  reference frame. Because of the differences between the actual and estimated parameters used in the observer model, the resulting back-EMF estimates can be divided, to extract the information about the displacement between the estimated and rotor flux reference frames, while reducing the effect of the observer parameter variation. The position displacement  $\Theta_{err}$  is obtained by the following equation:

$$\theta_{err} = \tan^{-1}\left(\frac{e_y}{e_\delta}\right)$$

Equation AMCLIB\_TrackObsrv\_Eq1

The estimated position can then be obtained by driving the position of the estimated reference frame  $\gamma - \delta$ , to achieve zero displacement  $\Theta_{err}=0$ . Therefore a phase locked loop mechanism must be adopted, where the loop compensator ensures the correct tracking of the actual rotor flux position by keeping the error signal  $\Theta_{err}=0$ . The position tracking observer with a standard PI controller used as the loop compensator is depicted in the following picture:

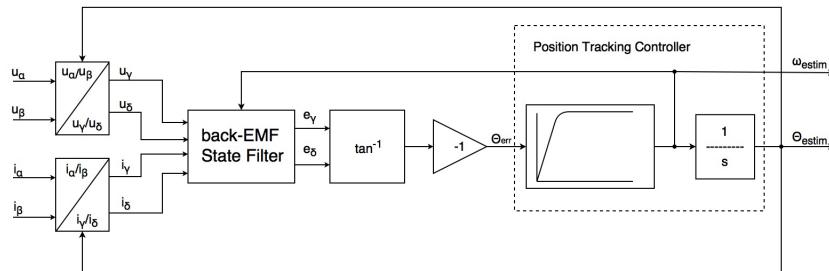


Figure 68. Block diagram of proposed PLL scheme for position estimation

The position tracking structure, described in [Figure 68](#), can be linearized around the operating point  $\Theta_{estim} = \Theta_e$ . The linear approximation of the position tracking observer with standard PI controller is shown in the following picture:

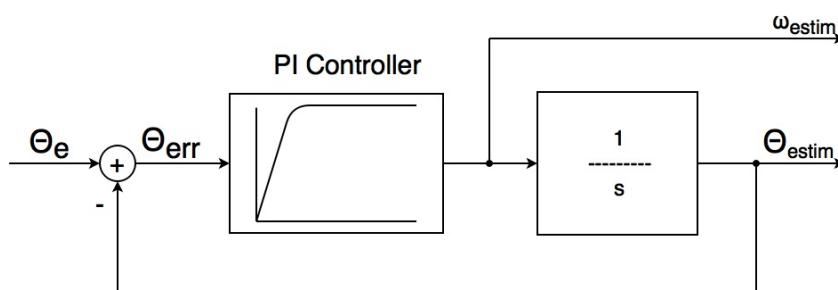


Figure 69. Linear approximation of the position tracking observer with standard PI controller

Linearized position tracking observer, depicted on [Figure 69](#) has the following transfer function:

$$G(s) = \frac{\theta_{estim}(s)}{\theta_e(s)} = \frac{sK_p + K_i}{s^2 + sK_p + K_i}$$

Equation AMCLIB\_TrackObsrv\_Eq2

Considering the  $s^*K_p$  term in the nominator as negligible, the controller gains  $K_p$  and  $K_i$  are calculated by comparing the characteristic polynomial of the resulting transfer function to a standard second order system polynomial:

$$K_p = 4\pi\xi f_0$$

$$K_i = (2\pi f_0)^2$$

Equation AMCLIB\_TrackObsrv\_Eq3

where:

- $\xi$  is the required attenuation
- $f_0$  is the required bandwidth of the position tracking loop. Since the position error signal is calculated by the state filter formed in the rotating reference frame, the dynamics of the position tracking loop includes only frequencies proportional to the rate of change of estimated and rotor flux frames displacement.

As demonstrated in [Figure 68](#), the position tracking controller consists of the standard PI controller and integrator. The output of the ideal standard PI controller is defined by the following equation:

$$\omega_{estim}(t) = \theta_{err}(t) \cdot K_p + K_i \int_0^\infty \theta_{err}(t) dt$$

Equation AMCLIB\_TrackObsrv\_Eq4

and the output of the ideal integrator as follows:

$$\theta_{estim}(t) = \int_0^\infty \omega_{estim}(t) dt$$

Equation AMCLIB\_TrackObsrv\_Eq5

where:

- $K_p$  is the proportional gain
- $K_i$  is integral gain.

Using the Laplace transformation, equations [AMCLIB\\_TrackObsrv\\_Eq4](#) and [AMCLIB\\_TrackObsrv\\_Eq5](#) are transformed in to the continuous time domain:

$$\omega_{estim}(s) = K_p \cdot \theta_{err}(s) + \frac{1}{s} \cdot K_i \cdot \theta_{err}(s)$$

$$\theta_{estim}(s) = \frac{1}{s} \cdot \omega_{estim}(s)$$

Equation AMCLIB\_TrackObsrv\_Eq6

forming two transfer functions:

$$G(s) = \frac{sK_p+K_i}{s}$$

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

Equation AMCLIB\_TrackObsrv\_Eq7

where the  $G(s)$  is the transfer function of the standard PI controller in a continuous time domain, and  $H(s)$  is the transfer function of the integrator in a continuous time domain.

In order to implement the standard PI controller and integrator in the discrete digital control systems, both blocks needs to be transformed into a discrete

time domain. Considering the trapezoid discretization method, the equations [AMCLIB\\_TrackObsrv\\_Eq6](#) are transformed into the following equations:

$$\begin{aligned}\omega_{estim}(k) &= \omega_{estim}(k-1) + \theta_{err}(k) \cdot \left( K_p + \frac{K_f T_s}{2} \right) + \theta_{err}(k-1) \cdot \left( -K_p + \frac{K_f T_s}{2} \right) \\ \theta_{estim}(k) &= \theta_{estim}(k-1) + \omega_{estim}(k) \cdot \frac{T_s}{2} + \omega_{estim}(k-1) \cdot \frac{T_s}{2}\end{aligned}$$

Equation AMCLIB\_TrackObsrv\_Eq8

where:

- $T_s$  is the sampling period [s]
- $\Theta_{err}(k)$  is the input phase error in the current step [rad]
- $\Theta_{err}(k-1)$  is the input phase error in the previous calculation step [rad]
- $\omega_{estim}(k)$  is the estimated angular velocity in the current step [rad/s]
- $\omega_{estim}(k-1)$  is the estimated angular velocity in the previous calculation step [rad/s].

Using the substitution:

$$\begin{aligned}CC_1 &= K_p + \frac{K_f T_s}{2} \\ CC_2 &= -K_p + \frac{K_f T_s}{2} \\ C_1 &= \frac{T_s}{2}\end{math>

Equation AMCLIB_TrackObsrv_Eq9$$

the equation [AMCLIB\\_TrackObsrv\\_Eq8](#) can be rewritten into:

$$\begin{aligned}\omega_{estim}(k) &= \omega_{estim}(k-1) + \theta_{err}(k) \cdot CC_1 + \theta_{err}(k-1) \cdot CC_2 \\ \theta_{estim}(k) &= \theta_{estim}(k-1) + \omega_{estim}(k) \cdot C_1 + \omega_{estim}(k-1) \cdot C_1\end{aligned}$$

Equation AMCLIB\_TrackObsrv\_Eq10

**Note:** The input/output pointers must contain valid addresses, otherwise a fault may occur (MemManage, BusFault, UsageFault, HardFault).

## Re-entrancy

The function is re-entrant for a different pCtrl.

### 2.7.1 Function AMCLIB\_TrackObsrv\_F32

#### Declaration

```
void AMCLIB_TrackObsrv_F32(tFrac32 f32PhaseErr, tFrac32 *pPosEst,  
tFrac32 *pVelocityEst, AMCLIB\_TRACK\_OBSRV\_T\_F32 *pCtrl);
```

#### Arguments

**Table 67. AMCLIB\_TrackObsrv\_F32 arguments**

Type	Name	Direction	Description
<a href="#">tFrac32</a>	f32PhaseErr	input	Input signal representing phase error of system to be estimated.
<a href="#">tFrac32</a> *	pPosEst	output	Estimated output position.
<a href="#">tFrac32</a> *	pVelocityEst	output	Estimated output velocity.

Type	Name	Direction	Description
<a href="#">AMCLIB_TRACK_OBSRV_T_F32</a> *	pCtrl	input, output	Pointer to a tracking observer structure <a href="#">AMCLIB_TRACK_OBSRV_T_F32</a> , which contains algorithm coefficients.

**Implementation details**

In order to implement the discretized equations [AMCLIB\\_TrackObsrv\\_eq10](#) of the position tracking controller on the fixed point arithmetic platforms, the maximal values (scales) of the input and output signals are as follows:

- $\Theta^{MAX}$  - maximal value of the position tracking controller input phase error
- $\Omega^{MAX}$  - maximal value of the position tracking controller output angular velocity must be known. These maximal values are essential for the correct casting of the physical signal values into fixed point values [-1, 1].

Considering the same maximal values for the input phase error  $\Theta_{err}$  and the output phase  $\Theta_{estim}$ , fractional representations of the input and both output signals are obtained using these equations:

$$\theta_f(k) = \frac{\theta_{err}(k)}{\theta^{MAX}}$$

$$\omega_f(k) = \frac{\omega_{err}(k)}{\omega^{MAX}}$$

Equation AMCLIB\_TrackObsrv\_F32\_Eq1

The resulting position tracking controller, discrete time domain equations in a fixed-point fractional representation, is given as follows:

$$\omega_f^{estim}(k) \cdot \omega^{MAX} = \omega_{estim}(k-1) \cdot \omega^{MAX} + \theta_{err}(k) \cdot \theta^{MAX} \cdot CC_1 + \theta_{err}(k-1) \cdot \theta^{MAX} \cdot CC_2$$

$$\theta_f^{estim} \cdot \theta^{MAX} = \theta_{estim}(k-1) \cdot \theta^{MAX} + \omega_{estim}(k) \cdot \omega^{MAX} \cdot C_1 + \omega_{estim}(k-1) \cdot \omega^{MAX} \cdot C_2$$

Equation AMCLIB\_TrackObsrv\_F32\_Eq2

which can be rearranged into the following form:

$$\omega_f^{estim}(k) = \omega_{estim}(k-1) + \theta_{err}(k) \cdot \frac{\theta^{MAX}}{\omega^{MAX}} \cdot CC_1 + \theta_{err}(k-1) \cdot \frac{\theta^{MAX}}{\omega^{MAX}} \cdot CC_2$$

$$\theta_f^{estim} = \theta_{estim}(k-1) + \omega_{estim}(k) \cdot \frac{\omega^{MAX}}{\theta^{MAX}} \cdot C_1 + \omega_{estim}(k-1) \cdot \frac{\omega^{MAX}}{\theta^{MAX}} \cdot C_2$$

Equation AMCLIB\_TrackObsrv\_F32\_Eq3

To further simplify the equation [AMCLIB\\_TrackObsrv\\_F32\\_Eq3](#), let's make the substitution:

$$CC_{1f} = CC_1 \cdot \frac{\theta^{MAX}}{\omega^{MAX}} = \left( K_p + \frac{K_i T_s}{2} \right) \cdot \frac{\theta^{MAX}}{\omega^{MAX}}$$

$$CC_{2f} = CC_2 \cdot \frac{\theta^{MAX}}{\omega^{MAX}} = \left( -K_p + \frac{K_i T_s}{2} \right) \cdot \frac{\theta^{MAX}}{\omega^{MAX}}$$

$$C_{1f} = C_1 \cdot \frac{\omega^{MAX}}{\theta^{MAX}} = \frac{T_s}{2} \cdot \frac{\omega^{MAX}}{\theta^{MAX}}$$

Equation AMCLIB\_TrackObsrv\_F32\_Eq4

where:

- $CC_{1f}$  and  $CC_{2f}$  are the PI controller coefficients adapted according to the input and output scale values
- $C_{1f}$  is the integrator coefficient adapted according to the input and output scale values.

To implement these coefficients as fractional numbers, all three coefficients have to fit in the fractional range [-1,1). However, depending on the  $CC_{1f}$ ,  $CC_{2f}$  and  $C_{1f}$  and on  $\Theta^{MAX}$ ,  $\Omega^{MAX}$  maximum values, calculation of  $CC_{1f}$ ,  $CC_{2f}$  and  $C_{1f}$  may result in values outside the fractional [-1, 1) range. Therefore, a scaling of  $CC_{1f}$ ,  $CC_{2f}$  and  $C_{1f}$  has to be introduced:

$$\begin{aligned} f32CC1_{SC} &= CC_{1f} \cdot 2^{u16NShift} = \left( K_p + \frac{K_f T_s}{2} \right) \cdot \frac{\theta^{MAX}}{\omega^{MAX}} \cdot 2^{u16NShift} \\ f32CC2_{SC} &= CC_{2f} \cdot 2^{u16NShift} = \left( -K_p + \frac{K_f T_s}{2} \right) \cdot \frac{\theta^{MAX}}{\omega^{MAX}} \cdot 2^{u16NShift} \\ f32C1_{SC} &= C_{1f} \cdot 2^{u16NIntegSh} = \frac{T_s}{2} \cdot \frac{\omega^{MAX}}{\theta^{MAX}} \cdot 2^{u16NIntegSh} \end{aligned}$$

Equation AMCLIB\_TrackObsrv\_F32\_Eq5

Both scaling shifts,  $u16NShift$  and  $u16NIntegSh$ , are chosen such that all three  $CC_{1f}$ ,  $CC_{2f}$  and  $C_{1f}$  coefficients reside in the range [-1, 1). To simplify the implementation on the digital controllers, these scaling shifts are chosen to be of a power of 2; thus the final scaling is a simple shift operation on digital controllers. Moreover, the scaling shifts cannot be a negative number, so the scaling operation is always scales the numbers with an absolute value larger than 1 down to fit the range [-1,1). With these discussed requirements, the scaling shifts are calculated as follows:

$$\begin{aligned} u16NShift &= \max \left( \left\lceil \frac{\log(\text{abs}(CC_{1f}))}{\log(2)} \right\rceil, \left\lceil \frac{\log(\text{abs}(CC_{2f}))}{\log(2)} \right\rceil \right) \\ u16NIntegSh &= \left\lceil \frac{\log(\text{abs}(C_{1f}))}{\log(2)} \right\rceil \end{aligned}$$

Equation AMCLIB\_TrackObsrv\_F32\_Eq6

where:

- $u16NShift$  is the scaling shift for the standard PI controller
- $u16NIntegSh$  is the scaling shift for the integrator.

Using [AMCLIB\\_TrackObsrv\\_F32\\_Eq4](#) and [AMCLIB\\_TrackObsrv\\_F32\\_Eq5](#), the [AMCLIB\\_TrackObsrv\\_F32\\_Eq3](#) equations are transformed into a final, scaled, fractional equations of the position tracking controller, represented by the AMCLIB\_TrackObsrv\_F32 function:

$$\begin{aligned} \omega_{estim}(k) &= (\omega_{estim}(k-1) + \theta_{err}(k) \cdot f32CC1_{SC} + \theta_{err}(k-1) \cdot f32CC2_{SC}) \cdot 2^{u16NShift} \\ \theta_{estim}(k) &= (\theta_{estim}(k-1) + \omega_{estim}(k) \cdot f32C1_{SC} + \omega_{estim}(k-1) \cdot f32C1_{SC}) \cdot 2^{u16NIntegSh} \end{aligned}$$

Equation AMCLIB\_TrackObsrv\_F32\_Eq7

where:

- $\omega_{estim}(k)$  is the output estimated angular velocity in the current step
- $\omega_{estim}(k-1)$  is the output estimated angular velocity in the previous calculation step
- $\Theta_{estim}(k)$  is the output estimated position in the current step
- $\Theta_{estim}(k-1)$  is the output estimated position in the previous calculation step
- $f32CC1sc$  is the 1st coefficient of the PI controller
- $f32CC2sc$  is the 2nd coefficient of the PI controller
- $u16NShift$  is the scaling shift of the PI controller
- $f32C1sc$  is the integrator constant
- $u16NIntegSh$  is the scaling shift of the integrator

The output estimated angular velocity signal limitation is implemented in the PI controller. The actual output  $\omega_{estim}(k)$  is bounded so as to not exceed the given UpperLimit and LowerLimit values:

$$\omega_{estim}(k) = \begin{cases} f32UpperLimit & \text{if } \omega_{estim}(k) \geq f32UpperLimit \\ \omega_{estim}(k) & \text{if } f32LowerLimit < \omega_{estim}(k) < f32UpperLimit \\ f32LowerLimit & \text{if } \omega_{estim}(k) \leq f32LowerLimit \end{cases}$$

Equation AMCLIB\_TrackObsrv\_F32\_Eq8

Where the bounds are exceeded, the non-linear saturation characteristics will take effect and influence the dynamic behavior. The output limitation is implemented on the output sum; therefore if the limitation occurs, the controller output is clipped to its bounds.

The accuracy of the results is guaranteed for outputs f32PosEstim and f32VelocityEstim only in cases when  $(pCtrl.pParamPI.u16NShift + pCtrlpParamInteg.u16NShift) < 15$ .

**Note:** If the output of the internal integrator exceeds the fractional range [-1,1] for the  $\Theta_{estim}$  output, an overflow occurs. This behavior allows the continual integration of the angular velocity of a rotor to obtain the actual rotor position, assuming the output range corresponds to one complete revolution.

### Code example

```
#define Wmax (2618.0F)
#define pi (3.1415927F)
#define Ts (1e-4F)
#define f0 (15.0F)
#define xi (0.707F)
#define w0 (2.0F*pi*f0)
#define Ki (w0*w0)
#define Kp (4.0F*pi*xi*f0)

#include "amclib.h"

AMCLIB_TRACK_OBSRV_T_F32 trMyTrObsrv;
tFrac32 f32PhaseErr;
tFrac32 f32PosEstim;
tFrac32 f32VelocityEstim;
tFrac32 f32PosOut;
tFrac32 f32VelocityOut;

void main (void)
{
    // controller parameters
    trMyTrObsrv.pParamPI.f32CC1sc      = FRAC32((Kp+(Ki*Ts)/2)*pi/Wmax);
    trMyTrObsrv.pParamPI.f32CC2sc      = FRAC32((-Kp+(Ki*Ts)/2)*pi/Wmax);
    trMyTrObsrv.pParamPI.f32UpperLimit = FRAC32(1.0);
    trMyTrObsrv.pParamPI.f32LowerLimit = FRAC32(-1.0);
    trMyTrObsrv.pParamPI.u16NShift    = (tu16)0;

    // Setting parameters for integrator
    trMyTrObsrv.pParamInteg.f32C1      = FRAC32((Ts/2)*Wmax/pi);
    trMyTrObsrv.pParamInteg.u16NShift = (tu16)0;

    // Setting of input phase error
    f32PhaseErr = FRAC32(0.25);

    // Initialization of observer internal states to zero:

    // Alternative 1: API call with postfix
}
```

```
// (only one alternative shall be used).
AMCLIB_TrackObsrvInit_F32(&trMyTrObsrv);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
AMCLIB_TrackObsrvInit(&trMyTrObsrv, F32);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 32-bit fractional implementation is selected as default.
AMCLIB_TrackObsrvInit(&trMyTrObsrv);

// Use #AMCLIB_Windmilling_F32 to obtain the estimated rotor position
// and velocity.
// Estimated position: f32PosOut
// Estimated velocity: f32VelocityOut

// Initialization of observer internal states for the required output
// position and velocity:

// Alternative 1: API call with postfix
// (only one alternative shall be used).
AMCLIB_TrackObsrvInitSetState_F32(f32PosOut, f32VelocityOut,
    &trMyTrObsrv);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
AMCLIB_TrackObsrvSetState(f32PosOut, f32VelocityOut,
    &trMyTrObsrv, F32);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 32-bit fractional implementation is selected as default.
AMCLIB_TrackObsrvSetState(f32PosOut, f32VelocityOut, &trMyTrObsrv);

// Calculation of one iteration of the observer:

// Alternative 1: API call with postfix
// (only one alternative shall be used).
AMCLIB_TrackObsrv_F32(f32PhaseErr, &f32PosEstim, &f32VelocityEstim,
    &trMyTrObsrv);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
AMCLIB_TrackObsrv(f32PhaseErr, &f32PosEstim, &f32VelocityEstim, &trMyTrObsrv,
    F32);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 32-bit fractional implementation is selected as default.
AMCLIB_TrackObsrv(f32PhaseErr, &f32PosEstim, &f32VelocityEstim,
    &trMyTrObsrv);

}
```

## 2.7.2 Function AMCLIB\_TrackObsrv\_F16

### Declaration

```
void AMCLIB_TrackObsrv_F16(tFrac16 f16PhaseErr, tFrac16 *pPosEst,
    tFrac16 *pVelocityEst, AMCLIB_TRACK_OBSRV_T_F16 *pCtrl);
```

## Arguments

**Table 68. AMCLIB\_TrackObsrv\_F16 arguments**

Type	Name	Direction	Description
tFrac16	f16PhaseErr	input	Input signal representing phase error of system to be estimated.
tFrac16 *	pPosEst	output	Estimated output position.
tFrac16 *	pVelocityEst	output	Estimated output velocity.
<a href="#">AMCLIB_TRACK_OBSRV_T_F16</a> *	pCtrl	input, output	Pointer to a tracking observer structure <a href="#">AMCLIB_TRACK_OBSRV_T_F16</a> , which contains algorithm coefficients.

## Implementation details

To implement the discretized equations [AMCLIB\\_TrackObsrv\\_eq10](#) of the position tracking controller on the fixed-point arithmetic platforms, the maximal values (scales) of the input and output signals:

- $\Theta^{MAX}$  - the maximal value of the position tracking controller input phase error
- $\Omega^{MAX}$  - the maximal value of the position tracking controller output angular velocity have to be known. These maximal values are essential for the correct casting of the physical signal values into fixed point values [-1, 1].

Considering the same maximal values for the input phase error  $\Theta_{err}$  and the output phase  $\Theta_{estim}$ , the fractional representation input and both output signals are obtained using these equations:

$$\theta_f(k) = \frac{\theta_{err}(k)}{\theta^{MAX}}$$

$$\omega_f(k) = \frac{\omega_{err}(k)}{\omega^{MAX}}$$

Equation AMCLIB\_TrackObsrv\_F16\_Eq1

The resulting position tracking controller, discrete time domain equations in fixed-point fractional representation, are given as follows:

$$\omega_f^{estim}(k) \cdot \omega^{MAX} = \omega_{estim}(k-1) \cdot \omega^{MAX} + \theta_{err}(k) \cdot \theta^{MAX} \cdot CC_1 + \theta_{err}(k-1) \cdot \theta^{MAX} \cdot CC_2$$

$$\theta_f^{estim} \cdot \theta^{MAX} = \theta_{estim}(k-1) \cdot \theta^{MAX} + \omega_{estim}(k) \cdot \omega^{MAX} \cdot C_1 + \omega_{estim}(k-1) \cdot \omega^{MAX} \cdot C_1$$

Equation AMCLIB\_TrackObsrv\_F16\_Eq2

which can be rearranged into this form:

$$\omega_f^{estim}(k) = \omega_{estim}(k-1) + \theta_{err}(k) \cdot \frac{\theta^{MAX}}{\omega^{MAX}} \cdot CC_1 + \theta_{err}(k-1) \cdot \frac{\theta^{MAX}}{\omega^{MAX}} \cdot CC_2$$

$$\theta_f^{estim} = \theta_{estim}(k-1) + \omega_{estim}(k) \cdot \frac{\omega^{MAX}}{\theta^{MAX}} \cdot C_1 + \omega_{estim}(k-1) \cdot \frac{\omega^{MAX}}{\theta^{MAX}} \cdot C_1$$

Equation AMCLIB\_TrackObsrv\_F16\_Eq3

To further simplify the equation [AMCLIB\\_TrackObsrv\\_F16\\_Eq3](#), let's make the substitution:

$$\begin{aligned} CC_{1f} &= CC_1 \cdot \frac{\theta^{MAX}}{\omega^{MAX}} = \left( K_p + \frac{K_i T_s}{2} \right) \cdot \frac{\theta^{MAX}}{\omega^{MAX}} \\ CC_{2f} &= CC_2 \cdot \frac{\theta^{MAX}}{\omega^{MAX}} = \left( -K_p + \frac{K_i T_s}{2} \right) \cdot \frac{\theta^{MAX}}{\omega^{MAX}} \\ C_{1f} &= C_1 \cdot \frac{\omega^{MAX}}{\theta^{MAX}} = \frac{T_s}{2} \cdot \frac{\omega^{MAX}}{\theta^{MAX}} \end{aligned}$$

Equation AMCLIB\_TrackObsrv\_F16\_Eq4

where:

- $CC_{1f}$  and  $CC_{2f}$  are the PI controller coefficients adapted according to the input and output scale values
- $C_{1f}$  is the integrator coefficient adapted according to the input and output scale values.

To implement these coefficients as fractional numbers, all three coefficients must fit in the fractional range [-1,1). However, depending on  $CC_{1f}$ ,  $CC_{2f}$  and  $C_{1f}$  as well as on  $\Theta^{MAX}$  and  $\Omega^{MAX}$  maximum values, calculations of  $CC_{1f}$ ,  $CC_{2f}$  and  $C_{1f}$  may result in values outside the fractional [-1, 1) range. Therefore, a scaling of  $CC_{1f}$ ,  $CC_{2f}$  and  $C_{1f}$  must be introduced:

$$\begin{aligned} f16CC1_{SC} &= CC_{1f} \cdot 2^{u16NShift} = \left( K_p + \frac{K_i T_s}{2} \right) \cdot \frac{\theta^{MAX}}{\omega^{MAX}} \cdot 2^{u16NShift} \\ f16CC2_{SC} &= CC_{2f} \cdot 2^{u16NShift} = \left( -K_p + \frac{K_i T_s}{2} \right) \cdot \frac{\theta^{MAX}}{\omega^{MAX}} \cdot 2^{u16NShift} \\ f16C1_{SC} &= C_{1f} \cdot 2^{u16NIntegSh} = \frac{T_s}{2} \cdot \frac{\omega^{MAX}}{\theta^{MAX}} \cdot 2^{u16NIntegSh} \end{aligned}$$

Equation AMCLIB\_TrackObsrv\_F16\_Eq5

Both scaling shifts,  $u16NShift$  and  $u16NIntegSh$ , are chosen such that all three  $CC_{1f}$ ,  $CC_{2f}$  and  $C_{1f}$  coefficients fit in the range [-1, 1). To simplify the implementation on the digital controllers, these scaling shifts are chosen to be power of the 2; thus the final scaling is a simple shift operation on the digital controllers. Moreover, the scaling shifts cannot be a negative number, so the scaling operation always scales the numbers with an absolute value larger than 1 down to fit the range [-1,1). With these discussed requirements, the scaling shifts are calculated as follows:

$$\begin{aligned} u16NShift &= \max \left( \left\lceil \frac{\log(\text{abs}(CC_{1f}))}{\log(2)} \right\rceil, \left\lceil \frac{\log(\text{abs}(CC_{2f}))}{\log(2)} \right\rceil \right) \\ u16NIntegSh &= \left\lceil \frac{\log(\text{abs}(C_{1f}))}{\log(2)} \right\rceil \end{aligned}$$

Equation AMCLIB\_TrackObsrv\_F16\_Eq6

where:

- $u16NShift$  is the scaling shift for the standard PI controller
- $u16NIntegSh$  is the scaling shift for the integrator.

Using [AMCLIB\\_TrackObsrv\\_F16\\_Eq4](#) and [AMCLIB\\_TrackObsrv\\_F16\\_Eq5](#), the [AMCLIB\\_TrackObsrv\\_F16\\_Eq3](#) equations are transformed into a final, scaled, fractional equations of the position tracking controller, represented by the AMCLIB\_TrackObsrv\_F16 function:

$$\begin{aligned} \omega_{estim}(k) &= (\omega_{estim}(k-1) + \theta_{err}(k) \cdot f16CC1_{SC} + \theta_{err}(k-1) \cdot f16CC2_{SC}) \cdot 2^{u16NShift} \\ \theta_{estim}(k) &= (\theta_{estim}(k-1) + \omega_{estim}(k) \cdot f16C1_{SC} + \omega_{estim}(k-1) \cdot f16C1_{SC}) \cdot 2^{u16NIntegSh} \end{aligned}$$

Equation AMCLIB\_TrackObsrv\_F16\_Eq7

where:

- $\omega_{estim}(k)$  is the output estimated angular velocity in the current step
- $\omega_{estim}(k-1)$  is the output estimated angular velocity in the previous calculation step
- $\Theta_{estim}(k)$  is the output estimated position in the current step
- $\Theta_{estim}(k-1)$  is the output estimated position in the previous calculation step
- f16CC1sc is the 1st coefficient of the PI controller
- f16CC2sc is the 2nd coefficient of the PI controller
- u16NShift is the scaling shift of the PI controller
- f16C1sc is the integrator constant
- u16NIntegSh is the scaling shift of the integrator

The output estimated angular velocity signal limitation is implemented in the PI controller. The actual output  $\omega_{estim}(k)$  is bounded so as to not exceed the given UpperLimit and LowerLimit limit values:

$$\omega_{estim}(k) = \begin{cases} f16UpperLimit & \text{if } \omega_{estim}(k) \geq f16UpperLimit \\ \omega_{estim}(k) & \text{if } f16LowerLimit < \omega_{estim}(k) < f16UpperLimit \\ f16LowerLimit & \text{if } \omega_{estim}(k) \leq f16LowerLimit \end{cases}$$

Equation AMCLIB\_TrackObsrv\_F16\_Eq8

Where the bounds are exceeded, the non-linear saturation characteristics will take effect and influence the dynamic behavior. The output limitation is implemented on the output sum; therefore if the limitation occurs, the controller output is clipped to its bounds.

The accuracy of f16VelocityEstim is guaranteed only for cases when pCtrl.pParamPI.u16NShift <= 15. The accuracy of f16PosEstim is guaranteed only for cases when (pCtrl.pParamPI.u16NShift <= 13 and pCtrl.pParaminteg.u16NShift = 0) or (pCtrl.pParamPI.u16NShift = 0 and pCtrl.pParaminteg.u16NShift <= 1). In other cases the worst case error might rise above the guaranteed limits.

**Note:** If the output of the internal integrator exceeds the fractional range [-1,1] for the  $\Theta_{estim}$  output, an overflow occurs. This behavior allows the continual integration of an angular velocity of a rotor to obtain the actual rotor position, assuming the output range corresponds to one complete revolution.

### Code example

```
#define Wmax (2618.0F)
#define pi (3.1415927F)
#define Ts (1e-4F)
#define f0 (15.0F)
#define xi (0.707F)
#define w0 (2.0F*pi*f0)
#define Ki (w0*w0)
#define Kp (4.0F*pi*xi*f0)

#include "amclib.h"

AMCLIB_TRACK_OBSRV_T_F16 trMyTrObsrv;
tFrac16 f16PhaseErr;
tFrac16 f16PosEstim;
tFrac16 f16VelocityEstim;
tFrac16 f16PosOut;
tFrac16 f16VelocityOut;
```

```
void main (void)
{
    // controller parameters
    trMyTrOsrsv.pParamPI.f16CC1sc      = FRAC16((Kp+(Ki*Ts)/2)*pi/Wmax);
    trMyTrOsrsv.pParamPI.f16CC2sc      = FRAC16((-Kp+(Ki*Ts)/2)*pi/Wmax);
    trMyTrOsrsv.pParamPI.f16UpperLimit = FRAC16(1.0);
    trMyTrOsrsv.pParamPI.f16LowerLimit = FRAC16(-1.0);
    trMyTrOsrsv.pParamPI.ul6NShift    = (tU16)0;

    // Setting parameters for integrator
    trMyTrOsrsv.pParamInteg.f16C1      = FRAC16((Ts/2)*Wmax/pi);
    trMyTrOsrsv.pParamInteg.ul6NShift = (tU16)0;

    // Setting of input phase error
    f16PhaseErr = FRAC16(0.25);

    // Initialization of observer internal states to zero:

    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB_TrackOsrsvInit_F16(&trMyTrOsrsv);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB_TrackOsrsvInit(&trMyTrOsrsv, F16);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 16-bit fractional implementation is selected as default.
    AMCLIB_TrackOsrsvInit(&trMyTrOsrsv);

    // Use #AMCLIB_Windmilling_F16 to obtain the estimated rotor position
    // and velocity.
    // Estimated position: f16PosOut
    // Estimated velocity: f16VelocityOut

    // Initialization of observer internal states for the required output
    // position and velocity:

    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB_TrackOsrsvInitSetState_F16(f16PosOut, f16VelocityOut,
        &trMyTrOsrsv);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB_TrackOsrsvSetState(f16PosOut, f16VelocityOut,
        &trMyTrOsrsv, F16);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 16-bit fractional implementation is selected as default.
    AMCLIB_TrackOsrsvSetState(f16PosOut, f16VelocityOut, &trMyTrOsrsv);

    // Calculation of one iteration of the observer:

    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB_TrackOsrsv_F16(f16PhaseErr, &f16PosEstim, &f16VelocityEstim,
        &trMyTrOsrsv);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB_TrackOsrsv(f16PhaseErr, &f16PosEstim, &f16VelocityEstim, &trMyTrOsrsv,
        F16);

    // Alternative 3: API call with global configuration of implementation
```

```
// (only one alternative shall be used). This alternative is available
// only if 16-bit fractional implementation is selected as default.
AMCLIB_TrackObsrv(f16PhaseErr, &f16PosEstim, &f16VelocityEstim,
&trMyTrObsrv);
}
```

### 2.7.3 Function AMCLIB\_TrackObsrv\_FLT

#### Declaration

```
void AMCLIB_TrackObsrv_FLT(tFloat fltPhaseErr, tFloat *pPosEst,
tFloat *pVelocityEst, AMCLIB\_TRACK\_OBSRV\_T\_FLT *pCtrl);
```

#### Arguments

**Table 69. AMCLIB\_TrackObsrv\_FLT arguments**

Type	Name	Direction	Description
<a href="#">tFloat</a>	fltPhaseErr	input	Input signal representing phase error of system to be estimated.
<a href="#">tFloat</a> *	pPosEst	output	Estimated output position.
<a href="#">tFloat</a> *	pVelocityEst	output	Estimated output velocity.
<a href="#">AMCLIB_TRACK_OBSRV_T_FLT</a> *	pCtrl	input, output	Pointer to a tracking observer structure <a href="#">AMCLIB_TRACK_OBSRV_T_FLT</a> , which contains algorithm coefficients.

#### Implementation details

The output estimated angular velocity signal limitation is implemented in the PI controller. The actual output  $\omega_{estim}(k)$  is bounded so as to not exceed the given UpperLimit and LowerLimit limit values:

$$\omega_{estim}(k) = \begin{cases} fltUpperLimit & \text{if } \omega_{estim}(k) \geq fltUpperLimit \\ \omega_{estim}(k) & \text{if } fltLowerLimit < \omega_{estim}(k) < fltUpperLimit \\ fltLowerLimit & \text{if } \omega_{estim}(k) \leq fltLowerLimit \end{cases}$$

Equation AMCLIB\_TrackObsrv\_FLT\_Eq1

Where the bounds are exceeded, the non-linear saturation characteristics will take effect and influence the dynamic behavior. The output limitation is implemented on the output sum; therefore, if the limitation occurs, the controller output is clipped to its bounds.

**Note:** If the output of the internal integrator exceeds the range  $[-\pi, \pi]$  for the  $\Theta_{estim}$  output, an output wraparound occurs. This behavior allows the continual integration of the angular velocity of a rotor to obtain the actual rotor position, assuming the output range corresponds to one complete revolution.

**Note:** The function may raise floating-point exceptions (invalid operation, overflow, underflow, inexact, input denormal). The floating-point unit must be enabled in CPACR register to prevent NOCP UsageFault exception.

#### Code example:

```
#define pi (3.1415927F)
#define Ts (1e-4F)
#define f0 (15.0F)
#define xi (0.707F)
```

```
#define w0 (2.0F*pi*f0)
#define Ki (w0*w0)
#define Kp (4.0F*pi*xi*f0)

#include "amclib.h"

AMCLIB_TRACK_OBSRV_T_FLT trMyTrObsrv;
tFloat fltPhaseErr;
tFloat fltPosEstim;
tFloat fltVelocityEstim;
tFloat fltPosOut;
tFloat fltVelocityOut;

void main (void)
{
    // controller parameters
    trMyTrObsrv.pParamPI.fltCC1sc      = (tFloat) (Kp+(Ki*Ts)/2);
    trMyTrObsrv.pParamPI.fltCC2sc      = (tFloat) (-Kp+(Ki*Ts)/2);
    trMyTrObsrv.pParamPI.fltUpperLimit = (tFloat) (1.0);
    trMyTrObsrv.pParamPI.fltLowerLimit = (tFloat) (-1.0);

    // Setting parameters for integrator
    trMyTrObsrv.pParamInteg.fltC1     = (tFloat) (Ts/2);

    // Setting of input phase error
    fltPhaseErr = (tFloat) (0.25);

    // Initialization of observer internal states to zero:

    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB_TrackObsrvInit_FLT(&trMyTrObsrv);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB_TrackObsrvInit(&trMyTrObsrv, FLT);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if single precision floating-point implementation is selected
    // as default.
    AMCLIB_TrackObsrvInit(&trMyTrObsrv);

    // Use #AMCLIB_Windmilling_FLT to obtain the estimated rotor position
    // and velocity.
    // Estimated position: fltPosOut
    // Estimated velocity: fltVelocityOut

    // Initialization of observer internal states for the required output
    // position and velocity:

    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB_TrackObsrvInitSetState_FLT(fltPosOut, fltVelocityOut,
        &trMyTrObsrv);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB_TrackObsrvSetState(fltPosOut, fltVelocityOut,
        &trMyTrObsrv, FLT);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if single precision floating-point implementation is selected
    // as default.
    AMCLIB_TrackObsrvSetState(fltPosOut, fltVelocityOut, &trMyTrObsrv);
```

```

// Calculation of one iteration of the observer:

// Alternative 1: API call with postfix
// (only one alternative shall be used).
AMCLIB_TrackObsrv_FLT(fltPhaseErr, &fltPosEstim, &fltVelocityEstim,
&trMyTrObsrv);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
AMCLIB_TrackObsrv(fltPhaseErr, &fltPosEstim, &fltVelocityEstim, &trMyTrObsrv,
FLT);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if single precision floating-point implementation is selected
// as default.
AMCLIB_TrackObsrv(fltPhaseErr, &fltPosEstim, &fltVelocityEstim,
&trMyTrObsrv);
}

```

## 2.7.4 Function AMCLIB\_TrackObsrvInit

### Description

This function initializes all internal states of the tracking observer to zero.

**Note:** The input/output pointers must contain valid addresses otherwise an exception may occur (Data TLB Error, Data Storage, Alignment, Machine Check).

**Note:** The input/output pointers must contain valid addresses, otherwise a fault may occur (MemManage, BusFault, UsageFault, HardFault).

### Re-entrancy:

The function is re-entrant for a different pCtrl.

### 2.7.4.1 Function AMCLIB\_TrackObsrvInit\_F32

#### Declaration

```
void AMCLIB_TrackObsrvInit_F32(AMCLIB_TRACK_OBSRV_T_F32 *pCtrl);
```

#### Arguments

Table 70. AMCLIB\_TrackObsrvInit\_F32 arguments

Type	Name	Direction	Description
AMCLIB_TRACK_OBSRV_T_F32 *	pCtrl	input, output	Pointer to a tracking observer structure AMCLIB_TRACK_OBSRV_T_F32, which contains algorithm coefficients.

#### Code example

```

#define Wmax (2618.0F)
#define pi (3.1415927F)
#define Ts (1e-4F)
#define f0 (15.0F)
#define xi (0.707F)
#define w0 (2.0F*pi*f0)
#define Ki (w0*w0)

```

```
#define Kp (4.0F*pi*xi*f0)

#include "amclib.h"

AMCLIB_TRACK_OBSRV_T_F32 trMyTrObsrv;
tFrac32 f32PhaseErr;
tFrac32 f32PosEstim;
tFrac32 f32VelocityEstim;
tFrac32 f32PosOut;
tFrac32 f32VelocityOut;

void main (void)
{
    // controller parameters
    trMyTrObsrv.pParamPI.f32CC1sc      = FRAC32((Kp+(Ki*Ts)/2)*pi/Wmax);
    trMyTrObsrv.pParamPI.f32CC2sc      = FRAC32((-Kp+(Ki*Ts)/2)*pi/Wmax);
    trMyTrObsrv.pParamPI.f32UpperLimit = FRAC32(1.0);
    trMyTrObsrv.pParamPI.f32LowerLimit = FRAC32(-1.0);
    trMyTrObsrv.pParamPI.u16NShift     = (tu16)0;

    // Setting parameters for integrator
    trMyTrObsrv.pParamInteg.f32C1      = FRAC32((Ts/2)*Wmax/pi);
    trMyTrObsrv.pParamInteg.u16NShift   = (tu16)0;

    // Setting of input phase error
    f32PhaseErr = FRAC32(0.25);

    // Initialization of observer internal states to zero:

    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB_TrackObsrvInit_F32(&trMyTrObsrv);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB_TrackObsrvInit(&trMyTrObsrv, F32);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 32-bit fractional implementation is selected as default.
    AMCLIB_TrackObsrvInit(&trMyTrObsrv);

    // Use #AMCLIB_Windmilling_F32 to obtain the estimated rotor position
    // and velocity.
    // Estimated position: f32PosOut
    // Estimated velocity: f32VelocityOut

    // Initialization of observer internal states for the required output
    // position and velocity:

    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB_TrackObsrvInitSetState_F32(f32PosOut, f32VelocityOut,
                                         &trMyTrObsrv);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB_TrackObsrvSetState(f32PosOut, f32VelocityOut,
                               &trMyTrObsrv, F32);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 32-bit fractional implementation is selected as default.
    AMCLIB_TrackObsrvSetState(f32PosOut, f32VelocityOut, &trMyTrObsrv);

    // Calculation of one iteration of the observer:
```

```

// Alternative 1: API call with postfix
// (only one alternative shall be used).
AMCLIB_TrackObsrv_F32(f32PhaseErr, &f32PosEstim, &f32VelocityEstim,
&trMyTrObsrv);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
AMCLIB_TrackObsrv(f32PhaseErr, &f32PosEstim, &f32VelocityEstim, &trMyTrObsrv,
F32);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 32-bit fractional implementation is selected as default.
AMCLIB_TrackObsrv(f32PhaseErr, &f32PosEstim, &f32VelocityEstim,
&trMyTrObsrv);
}

```

#### 2.7.4.2 Function AMCLIB\_TrackObsrvInit\_F16

##### Declaration

```
void AMCLIB_TrackObsrvInit_F16(AMCLIB_TRACK_OBSRV_T_F16 *pCtrl);
```

##### Arguments

**Table 71. AMCLIB\_TrackObsrvInit\_F16 arguments**

Type	Name	Direction	Description
AMCLIB_TRACK_OBSRV_T_F16 *	pCtrl	input, output	Pointer to a tracking observer structure <a href="#">AMCLIB_TRACK_OBSRV_T_F16</a> , which contains algorithm coefficients.

##### Code example

```

#define Wmax (2618.0F)
#define pi (3.1415927F)
#define Ts (1e-4F)
#define f0 (15.0F)
#define xi (0.707F)
#define w0 (2.0F*pi*f0)
#define Ki (w0*w0)
#define Kp (4.0F*pi*xi*f0)

#include "amclib.h"

AMCLIB_TRACK_OBSRV_T_F16 trMyTrObsrv;
tFrac16 f16PhaseErr;
tFrac16 f16PosEstim;
tFrac16 f16VelocityEstim;
tFrac16 f16PosOut;
tFrac16 f16VelocityOut;

void main (void)
{
    // controller parameters
    trMyTrObsrv.pParamPI.f16CC1sc      = FRAC16((Kp+(Ki*Ts)/2)*pi/Wmax);
    trMyTrObsrv.pParamPI.f16CC2sc      = FRAC16((-Kp+(Ki*Ts)/2)*pi/Wmax);
    trMyTrObsrv.pParamPI.f16UpperLimit = FRAC16(1.0);
    trMyTrObsrv.pParamPI.f16LowerLimit = FRAC16(-1.0);
    trMyTrObsrv.pParamPI.u16NShift     = (tU16)0;

    // Setting parameters for integrator
    trMyTrObsrv.pParamInteg.f16C1      = FRAC16((Ts/2)*Wmax/pi);
}

```

```
trMyTrObsrv.pParamInteg.u16NShift = (tU16) 0;

// Setting of input phase error
f16PhaseErr = FRAC16(0.25);

// Initialization of observer internal states to zero:

// Alternative 1: API call with postfix
// (only one alternative shall be used).
AMCLIB_TrackObsrvInit_F16(&trMyTrObsrv);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
AMCLIB_TrackObsrvInit(&trMyTrObsrv, F16);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 16-bit fractional implementation is selected as default.
AMCLIB_TrackObsrvInit(&trMyTrObsrv);

// Use #AMCLIB_Windmilling_F16 to obtain the estimated rotor position
// and velocity.
// Estimated position: f16PosOut
// Estimated velocity: f16VelocityOut

// Initialization of observer internal states for the required output
// position and velocity:

// Alternative 1: API call with postfix
// (only one alternative shall be used).
AMCLIB_TrackObsrvInitSetState_F16(f16PosOut, f16VelocityOut,
&trMyTrObsrv);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
AMCLIB_TrackObsrvSetState(f16PosOut, f16VelocityOut,
&trMyTrObsrv, F16);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 16-bit fractional implementation is selected as default.
AMCLIB_TrackObsrvSetState(f16PosOut, f16VelocityOut, &trMyTrObsrv);

// Calculation of one iteration of the observer:

// Alternative 1: API call with postfix
// (only one alternative shall be used).
AMCLIB\_TrackObsrv\_F16(f16PhaseErr, &f16PosEstim, &f16VelocityEstim,
&trMyTrObsrv);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
AMCLIB_TrackObsrv(f16PhaseErr, &f16PosEstim, &f16VelocityEstim, &trMyTrObsrv,
F16);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 16-bit fractional implementation is selected as default.
AMCLIB_TrackObsrv(f16PhaseErr, &f16PosEstim, &f16VelocityEstim,
&trMyTrObsrv);
}
```

### 2.7.4.3 Function AMCLIB\_TrackObsrvInit\_FLT

#### Declaration

```
void AMCLIB_TrackObsrvInit_FLT(AMCLIB_TRACK_OBSRV_T_FLT *pCtrl);
```

#### Arguments

**Table 72. AMCLIB\_TrackObsrvInit\_FLT arguments**

Type	Name	Direction	Description
AMCLIB_TRACK_OBSRV_T_FLT *	pCtrl	input, output	Pointer to a tracking observer structure <a href="#">AMCLIB_TRACK_OBSRV_T_FLT</a> , which contains algorithm coefficients.

**Note:** The function may raise floating-point exceptions (invalid operation, overflow, underflow, inexact, input denormal). The floating-point unit must be enabled in CPACR register to prevent NOCP UsageFault exception.

#### Code example:

```
#define pi (3.1415927F)
#define Ts (1e-4F)
#define f0 (15.0F)
#define xi (0.707F)
#define w0 (2.0F*pi*f0)
#define Ki (w0*w0)
#define Kp (4.0F*pi*xi*f0)

#include "amclib.h"

AMCLIB_TRACK_OBSRV_T_FLT trMyTrObsrv;
tFloat fltPhaseErr;
tFloat fltPosEstim;
tFloat fltVelocityEstim;
tFloat fltPosOut;
tFloat fltVelocityOut;

void main (void)
{
    // controller parameters
    trMyTrObsrv.pParamPI.fltCC1sc      = (tFloat) (Kp+(Ki*Ts)/2);
    trMyTrObsrv.pParamPI.fltCC2sc      = (tFloat) (-Kp+(Ki*Ts)/2);
    trMyTrObsrv.pParamPI.fltUpperLimit = (tFloat) (1.0);
    trMyTrObsrv.pParamPI.fltLowerLimit = (tFloat) (-1.0);

    // Setting parameters for integrator
    trMyTrObsrv.pParamInteg.fltC1     = (tFloat) (Ts/2);

    // Setting of input phase error
    fltPhaseErr = (tFloat) (0.25);

    // Initialization of observer internal states to zero:

    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB_TrackObsrvInit_FLT(&trMyTrObsrv);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB_TrackObsrvInit(&trMyTrObsrv, FLT);

    // Alternative 3: API call with global configuration of implementation
```

```
// (only one alternative shall be used). This alternative is available
// only if single precision floating-point implementation is selected
// as default.
AMCLIB_TrackObsrvInit(&trMyTrObsrv);

// Use #AMCLIB_Windmilling_FLT to obtain the estimated rotor position
// and velocity.
// Estimated position: fltPosOut
// Estimated velocity: fltVelocityOut

// Initialization of observer internal states for the required output
// position and velocity:

// Alternative 1: API call with postfix
// (only one alternative shall be used).
AMCLIB_TrackObsrvInitSetState_FLT(fltPosOut, fltVelocityOut,
    &trMyTrObsrv);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
AMCLIB_TrackObsrvSetState(fltPosOut, fltVelocityOut,
    &trMyTrObsrv, FLT);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if single precision floating-point implementation is selected
// as default.
AMCLIB_TrackObsrvSetState(fltPosOut, fltVelocityOut, &trMyTrObsrv);

// Calculation of one iteration of the observer:

// Alternative 1: API call with postfix
// (only one alternative shall be used).
AMCLIB_TrackObsrv_FLT(fltPhaseErr, &fltPosEstim, &fltVelocityEstim,
    &trMyTrObsrv);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
AMCLIB_TrackObsrv(fltPhaseErr, &fltPosEstim, &fltVelocityEstim, &trMyTrObsrv,
    FLT);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if single precision floating-point implementation is selected
// as default.
AMCLIB_TrackObsrv(fltPhaseErr, &fltPosEstim, &fltVelocityEstim,
    &trMyTrObsrv);
}
```

## 2.7.5 Function AMCLIB\_TrackObsrvSetState

### Description

This function sets the tracking observer internal states to attain the required angular velocity and position outputs. Setting the required values to the actual estimated position and velocity of the rotor enables seamless transition from an uncontrolled rotation of the rotor into a controlled state. A robust set of initial estimates can be obtained from function [AMCLIB\\_Windmilling](#).

**Note:** The observer parameters must be already set in the state structure pointed to by *pCtrl* before this function can be called.

**Note:** The input/output pointers must contain valid addresses otherwise an exception may occur (Data TLB Error, Data Storage, Alignment, Machine Check).

**Note:** The input/output pointers must contain valid addresses, otherwise a fault may occur (MemManage, BusFault, UsageFault, HardFault).

#### Re-entrancy:

The function is re-entrant for a different pCtrl.

##### 2.7.5.1 Function AMCLIB\_TrackObsrvSetState\_F32

###### Declaration

```
void AMCLIB_TrackObsrvSetState_F32(tFrac32 f32PosOut, tFrac32 f32VelocityOut, AMCLIB\_TRACK\_OBSRV\_T\_F32 *pCtrl);
```

###### Arguments

Table 73. AMCLIB\_TrackObsrvSetState\_F32 arguments

Type	Name	Direction	Description
<a href="#">tFrac32</a>	f32PosOut	input	Required output of the GFLIB_ControllerPIrAW, i.e., what rotor position shall be outputted in the next iteration.
<a href="#">tFrac32</a>	f32VelocityOut	input	Required output of the GFLIB_IntegratorTR, i.e., what velocity shall be outputted in the next iteration.
<a href="#">AMCLIB_TRACK_OBSRV_T_F32</a> *	pCtrl	input, output	Pointer to a tracking observer structure <a href="#">AMCLIB_TRACK_OBSRV_T_F32</a> , which contains algorithm coefficients.

###### Implementation details

The scaling of the inputs f32PosOut and f32VelocityOut follows the same rules as the corresponding outputs of [AMCLIB\\_TrackObsrv\\_F32](#).

###### Code example

```
#define Wmax (2618.0F)
#define pi (3.1415927F)
#define Ts (1e-4F)
#define f0 (15.0F)
#define xi (0.707F)
#define w0 (2.0F*pi*f0)
#define Ki (w0*w0)
#define Kp (4.0F*pi*xi*f0)

#include "amclib.h"

AMCLIB\_TRACK\_OBSRV\_T\_F32 trMyTrObsrv;
tFrac32 f32PhaseErr;
tFrac32 f32PosEstim;
tFrac32 f32VelocityEstim;
tFrac32 f32PosOut;
tFrac32 f32VelocityOut;

void main (void)
{
    // controller parameters
    trMyTrObsrv.pParamPI.f32CC1sc      = FRAC32((Kp+(Ki*Ts)/2)*pi/Wmax);
    trMyTrObsrv.pParamPI.f32CC2sc      = FRAC32((-Kp+(Ki*Ts)/2)*pi/Wmax);
    trMyTrObsrv.pParamPI.f32UpperLimit = FRAC32(1.0);
```

```
trMyTrObsrv.pParamPI.f32LowerLimit = FRAC32(-1.0);
trMyTrObsrv.pParamPI.u16NShift = (TU16)0;

// Setting parameters for integrator
trMyTrObsrv.pParamInteg.f32C1 = FRAC32((Ts/2)*Wmax/pi);
trMyTrObsrv.pParamInteg.u16NShift = (TU16)0;

// Setting of input phase error
f32PhaseErr = FRAC32(0.25);

// Initialization of observer internal states to zero:

// Alternative 1: API call with postfix
// (only one alternative shall be used).
AMCLIB_TrackObsrvInit_F32(&trMyTrObsrv);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
AMCLIB_TrackObsrvInit(&trMyTrObsrv, F32);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 32-bit fractional implementation is selected as default.
AMCLIB_TrackObsrvInit(&trMyTrObsrv);

// Use #AMCLIB_Windmilling_F32 to obtain the estimated rotor position
// and velocity.
// Estimated position: f32PosOut
// Estimated velocity: f32VelocityOut

// Initialization of observer internal states for the required output
// position and velocity:

// Alternative 1: API call with postfix
// (only one alternative shall be used).
AMCLIB_TrackObsrvInitSetState_F32(f32PosOut, f32VelocityOut,
&trMyTrObsrv);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
AMCLIB_TrackObsrvSetState(f32PosOut, f32VelocityOut,
&trMyTrObsrv, F32);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 32-bit fractional implementation is selected as default.
AMCLIB_TrackObsrvSetState(f32PosOut, f32VelocityOut, &trMyTrObsrv);

// Calculation of one iteration of the observer:

// Alternative 1: API call with postfix
// (only one alternative shall be used).
AMCLIB_TrackObsrv_F32(f32PhaseErr, &f32PosEstim, &f32VelocityEstim,
&trMyTrObsrv);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
AMCLIB_TrackObsrv(f32PhaseErr, &f32PosEstim, &f32VelocityEstim, &trMyTrObsrv,
F32);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 32-bit fractional implementation is selected as default.
AMCLIB_TrackObsrv(f32PhaseErr, &f32PosEstim, &f32VelocityEstim,
&trMyTrObsrv);
}
```

### 2.7.5.2 Function AMCLIB\_TrackObsrvSetState\_F16

#### Declaration

```
void AMCLIB_TrackObsrvSetState_F16(tFrac16 f16PosOut, tFrac16  
f16VelocityOut, AMCLIB\_TRACK\_OBSRV\_T\_F16 *pCtrl);
```

#### Arguments

Table 74. AMCLIB\_TrackObsrvSetState\_F16 arguments

Type	Name	Direction	Description
<a href="#">tFrac16</a>	f16PosOut	<a href="#">input</a>	Required output of the GFLIB_ControllerPIrAW, i.e., what rotor position shall be outputted in the next iteration.
<a href="#">tFrac16</a>	f16VelocityOut	<a href="#">input</a>	Required output of the GFLIB_IntegratorTR, i.e., what velocity shall be outputted in the next iteration.
<a href="#">AMCLIB_TRACK_OBSRV_T_F16</a> *	pCtrl	<a href="#">input, output</a>	Pointer to a tracking observer structure <a href="#">AMCLIB_TRACK_OBSRV_T_F16</a> , which contains algorithm coefficients.

#### Implementation details

The scaling of the inputs f16PosOut and f16VelocityOut follows the same rules as the corresponding outputs of [AMCLIB\\_TrackObsrv\\_F16](#).

#### Code example

```
#define Wmax (2618.0F)
#define pi (3.1415927F)
#define Ts (1e-4F)
#define f0 (15.0F)
#define xi (0.707F)
#define w0 (2.0F*pi*f0)
#define Ki (w0*w0)
#define Kp (4.0F*pi*xi*f0)

#include "amclib.h"

AMCLIB\_TRACK\_OBSRV\_T\_F16 trMyTrObsrv;
tFrac16 f16PhaseErr;
tFrac16 f16PosEstim;
tFrac16 f16VelocityEstim;
tFrac16 f16PosOut;
tFrac16 f16VelocityOut;

void main (void)
{
    // controller parameters
    trMyTrObsrv.pParamPI.f16CC1sc      = FRAC16((Kp+(Ki*Ts)/2)*pi/Wmax);
    trMyTrObsrv.pParamPI.f16CC2sc      = FRAC16((-Kp+(Ki*Ts)/2)*pi/Wmax);
    trMyTrObsrv.pParamPI.f16UpperLimit = FRAC16(1.0);
    trMyTrObsrv.pParamPI.f16LowerLimit = FRAC16(-1.0);
    trMyTrObsrv.pParamPI.ul6NShift    = (tU16)0;

    // Setting parameters for integrator
    trMyTrObsrv.pParamInteg.f16C1      = FRAC16((Ts/2)*Wmax/pi);
    trMyTrObsrv.pParamInteg.ul6NShift = (tU16)0;

    // Setting of input phase error
    f16PhaseErr = FRAC16(0.25);

    // Initialization of observer internal states to zero:
```

```
// Alternative 1: API call with postfix
// (only one alternative shall be used).
AMCLIB_TrackObsrvInit_F16(&trMyTrObsrv);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
AMCLIB_TrackObsrvInit(&trMyTrObsrv, F16);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 16-bit fractional implementation is selected as default.
AMCLIB_TrackObsrvInit(&trMyTrObsrv);

// Use #AMCLIB_Windmilling_F16 to obtain the estimated rotor position
// and velocity.
// Estimated position: f16PosOut
// Estimated velocity: f16VelocityOut

// Initialization of observer internal states for the required output
// position and velocity:

// Alternative 1: API call with postfix
// (only one alternative shall be used).
AMCLIB_TrackObsrvInitSetState_F16(f16PosOut, f16VelocityOut,
    &trMyTrObsrv);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
AMCLIB_TrackObsrvSetState(f16PosOut, f16VelocityOut,
    &trMyTrObsrv, F16);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 16-bit fractional implementation is selected as default.
AMCLIB_TrackObsrvSetState(f16PosOut, f16VelocityOut, &trMyTrObsrv);

// Calculation of one iteration of the observer:

// Alternative 1: API call with postfix
// (only one alternative shall be used).
AMCLIB_TrackObsrv_F16(f16PhaseErr, &f16PosEstim, &f16VelocityEstim,
    &trMyTrObsrv);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
AMCLIB_TrackObsrv(f16PhaseErr, &f16PosEstim, &f16VelocityEstim, &trMyTrObsrv,
    F16);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 16-bit fractional implementation is selected as default.
AMCLIB_TrackObsrv(f16PhaseErr, &f16PosEstim, &f16VelocityEstim,
    &trMyTrObsrv);
}
```

### 2.7.5.3 Function AMCLIB\_TrackObsrvSetState\_FLT

#### Declaration

```
void AMCLIB_TrackObsrvSetState_FLT(tFloat fltPosOut, tFloat
    fltVelocityOut, AMCLIB_TRACK_OBSRV_T_FLT *pCtrl);
```

## Arguments

**Table 75. AMCLIB\_TrackObsrvSetState\_FLT arguments**

Type	Name	Direction	Description
tFloat	fltPosOut	input	Required output of the GFLIB_ControllerPIrAW, i.e., what rotor position shall be outputted in the next iteration.
tFloat	fltVelocityOut	input	Required output of the GFLIB_IntegratorTR, i.e., what velocity shall be outputted in the next iteration.
AMCLIB_TRACK_OBSRV_T_FLT *	pCtrl	input, output	Pointer to a tracking observer structure <a href="#">AMCLIB_TRACK_OBSRV_T_FLT</a> , which contains algorithm coefficients.

**Note:** The function may raise floating-point exceptions (invalid operation, overflow, underflow, inexact, input denormal). The floating-point unit must be enabled in CPACR register to prevent NOCP UsageFault exception.

## Code example:

```
#define pi (3.1415927F)
#define Ts (1e-4F)
#define f0 (15.0F)
#define xi (0.707F)
#define w0 (2.0F*pi*f0)
#define Ki (w0*w0)
#define Kp (4.0F*pi*xi*f0)

#include "amclib.h"

AMCLIB_TRACK_OBSRV_T_FLT trMyTrObsrv;
tFloat fltPhaseErr;
tFloat fltPosEstim;
tFloat fltVelocityEstim;
tFloat fltPosOut;
tFloat fltVelocityOut;

void main (void)
{
    // controller parameters
    trMyTrObsrv.pParamPI.fltCC1sc      = (tFloat) (Kp+(Ki*Ts)/2);
    trMyTrObsrv.pParamPI.fltCC2sc      = (tFloat) (-Kp+(Ki*Ts)/2);
    trMyTrObsrv.pParamPI.fltUpperLimit = (tFloat) (1.0);
    trMyTrObsrv.pParamPI.fltLowerLimit = (tFloat) (-1.0);

    // Setting parameters for integrator
    trMyTrObsrv.pParamInteg.fltC1      = (tFloat) (Ts/2);

    // Setting of input phase error
    fltPhaseErr = (tFloat) (0.25);

    // Initialization of observer internal states to zero:

    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB_TrackObsrvInit_FLT(&trMyTrObsrv);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB_TrackObsrvInit(&trMyTrObsrv, FLT);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if single precision floating-point implementation is selected
```

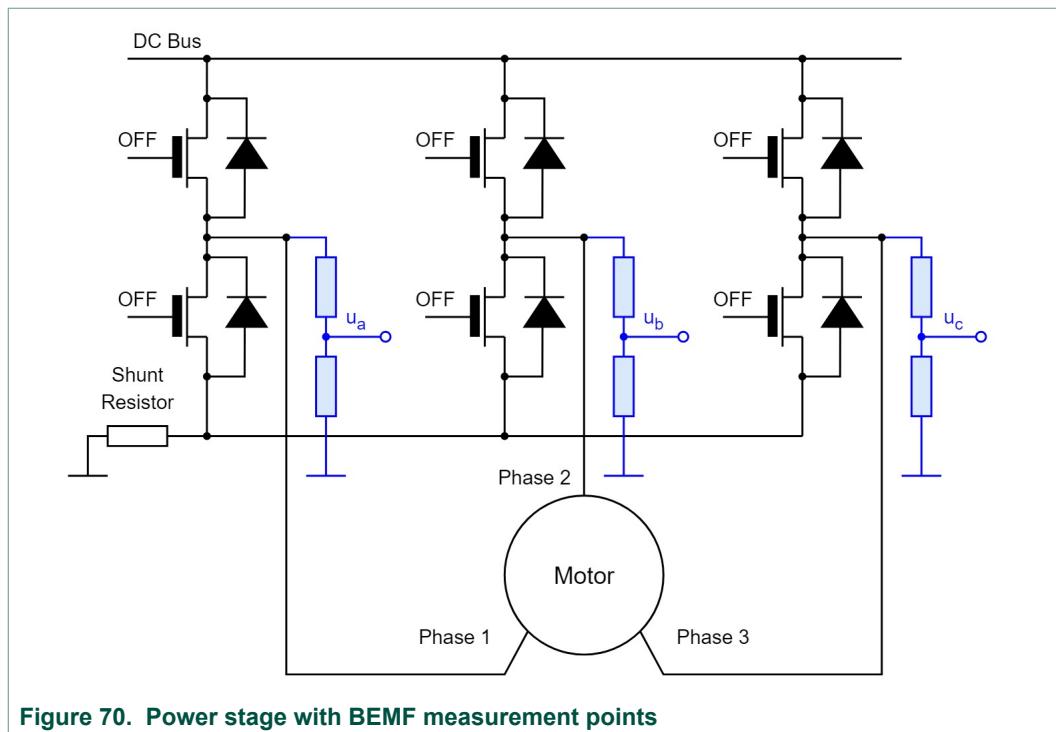
```
// as default.  
AMCLIB_TrackObsrvInit(&trMyTrObsrv);  
  
// Use #AMCLIB_Windmilling_FLT to obtain the estimated rotor position  
// and velocity.  
// Estimated position: fltPosOut  
// Estimated velocity: fltVelocityOut  
  
// Initialization of observer internal states for the required output  
// position and velocity:  
  
// Alternative 1: API call with postfix  
// (only one alternative shall be used).  
AMCLIB_TrackObsrvInitSetState_FLT(fltPosOut, fltVelocityOut,  
    &trMyTrObsrv);  
  
// Alternative 2: API call with implementation parameter  
// (only one alternative shall be used).  
AMCLIB_TrackObsrvSetState(fltPosOut, fltVelocityOut,  
    &trMyTrObsrv, FLT);  
  
// Alternative 3: API call with global configuration of implementation  
// (only one alternative shall be used). This alternative is available  
// only if single precision floating-point implementation is selected  
// as default.  
AMCLIB_TrackObsrvSetState(fltPosOut, fltVelocityOut, &trMyTrObsrv);  
  
// Calculation of one iteration of the observer:  
  
// Alternative 1: API call with postfix  
// (only one alternative shall be used).  
AMCLIB_TrackObsrv_FLT(fltPhaseErr, &fltPosEstim, &fltVelocityEstim,  
    &trMyTrObsrv);  
  
// Alternative 2: API call with implementation parameter  
// (only one alternative shall be used).  
AMCLIB_TrackObsrv(fltPhaseErr, &fltPosEstim, &fltVelocityEstim, &trMyTrObsrv,  
    FLT);  
  
// Alternative 3: API call with global configuration of implementation  
// (only one alternative shall be used). This alternative is available  
// only if single precision floating-point implementation is selected  
// as default.  
AMCLIB_TrackObsrv(fltPhaseErr, &fltPosEstim, &fltVelocityEstim,  
    &trMyTrObsrv);  
}
```

## 2.8 Function AMCLIB\_Windmilling

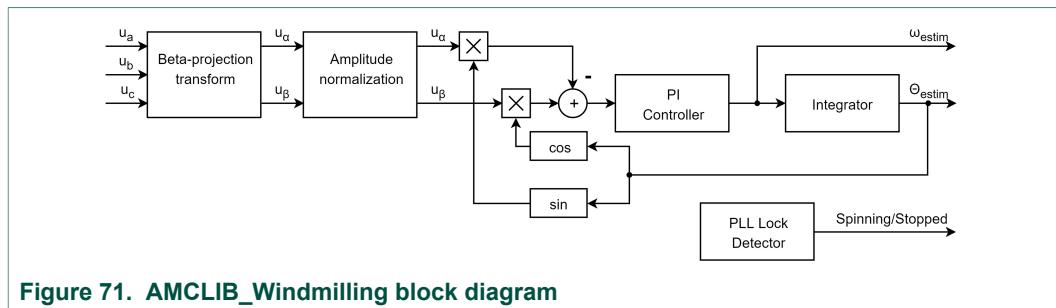
This function detects spontaneous rotation of an unpowered 3-phase permanent magnet synchronous motor by measuring the generated back electromotive force.

### Description

This function detects spontaneous rotation (windmilling) of an unpowered 3-phase permanent magnet synchronous motor (PMSM) by measuring the generated back electromotive force (BEMF). Assuming that the motor is connected to a 3-phase bridge power stage, it is necessary to add three voltage measuring points ( $u_a$ ,  $u_b$ ,  $u_c$ ) as indicated in the following figure.



The 3-phase voltages can be measured by single-ended A/D converters. The patent-pending algorithm employed in AMCLIB\_Windmilling can accept measurements distorted by variable DC offset, limitation, noise, and certain higher harmonics. The function will decide whether the motor is spinning or not and calculate the estimated position and angular velocity of the rotor. The following figure shows the internal structure of AMCLIB\_Windmilling.



The Beta-projection block is also available as a stand-alone function (see [GMCLIB\\_BetaProjection](#)). AMCLIB\_Windmilling internally calls the [AMCLIB\\_TrackObsrv](#) function to realize the phase-locked loop (PLL) feedback structure with a PI controller and an integrator. Refer to the tracking observer chapter in this manual to learn how to set the parameters of the PI controller and the integrator. AMCLIB\_Windmilling provides an automatic detector of the PLL lock, which is used as an indicator whether the rotor is spinning or not. The PLL takes a while to achieve a lock, so the function will return **UNDECIDED** for the first several iterations and then either **SPINNING** or **STOPPED**, depending on the actual state. It is necessary to call the function `AMCLIB_WindmillingInit` before the first use of AMCLIB\_Windmilling to initialize the state variables.

**Note:** The input/output pointers must contain valid addresses, otherwise a fault may occur (`MemManage`, `BusFault`, `UsageFault`, `HardFault`).

## Re-entrancy

The function is re-entrant.

### 2.8.1 Function AMCLIB\_Windmilling\_F32

#### Declaration

```
AMCLIB_WINDMILLING_RET_T AMCLIB_Windmilling_F32(const
SWLIBS_3Syst_F32 *pUabcIn, tFrac32 *pPosEst, tFrac32
*pVelocityEst, AMCLIB_WINDMILLING_T_F32 *const pCtrl);
```

#### Arguments

Table 76. AMCLIB\_Windmilling\_F32 arguments

Type	Name	Direction	Description
const SWLIBS_3Syst_F32 *	pUabcIn	input	Pointer to the structure with the measured 3-phase A/B/C voltages.
tFrac32 *	pPosEst	output	Estimated rotor flux position in the interval <-1; 1) which corresponds to the angle range <-π; π) radians.
tFrac32 *	pVelocityEst	output	Estimated electrical angular velocity of the rotor. The velocity estimates are noisy, especially for low speeds; therefore, it is recommended to use a <a href="#">GDFLIB_FilterMA_F32</a> to further filter the results. The filter should be engaged only after the AMCLIB_Windmilling has returned <b>SPINNING</b> ; use <a href="#">GDFLIB_FilterMASetState_F32</a> to initialize the filter state with the first estimated velocity value.
AMCLIB_WINDMILLING_T_F32 *const	pCtrl	input, output	Pointer to the structure with AMCLIB_Windmilling parameters and state variables.

#### Return

The function returns **UNDECIDED** if it is not yet clear whether the rotor is spinning or not. The AMCLIB\_Windmilling must then be called again in the next sampling period. The function returns **SPINNING** if the rotor is spinning. If the motor spin is not detected within 2000 iterations, the function will return **STOPPED**.

#### Implementation details

This function internally calls the Tracking Observer function which calculates the PI controller and the integrator subblocks. Refer to the documentation of function [AMCLIB\\_TrackOsrsv\\_F32](#) (and its parent chapter describing the AMCLIB\_TrackOsrsv functionality) for a description of the necessary scaling and setup of the parameters of the PI controller and integrator blocks used by the AMCLIB\_Windmilling\_F32. It is necessary to call the [AMCLIB\\_WindmillingInit\\_F32](#) before the first use of AMCLIB\_Windmilling\_F32 to initialize the state variables.

**Note:** Due to effectivity reasons, this function is implemented using inline assembly and is therefore not ANSI-C compliant.

#### Code Example

```
#define ADC_MAX_OFFSET_ERROR 4
#define Wmax (2618.0F)
```

```
#define pi (3.1415927F)
#define Ts (1e-4F)
#define f0 (30.0F)
#define xi (0.707F)
#define w0 (2.0F*pi*f0)
#define Ki (w0*w0)
#define Kp (4.0F*pi*xi*f0)

#include "amclib.h"

SWLIBS_3Syst_F32
AMCLIB_WINDMILLING_T_F32
tFrac32
tFrac32
AMCLIB_WINDMILLING_RET_T
f32Uabc;
windStruct;
f32Position;
f32Velocity;
RetVal = UNDECIDED;

void main (void)
{
    // Controller parameters
    windStruct.pParamATO.pParamPI.f32CC1sc = FRAC32((Kp+(Ki*Ts)/2)*pi/Wmax);
    windStruct.pParamATO.pParamPI.f32CC2sc = FRAC32((-Kp+(Ki*Ts)/2)*pi/Wmax);
    windStruct.pParamATO.pParamPI.f32UpperLimit = FRAC32(1.0);
    windStruct.pParamATO.pParamPI.f32LowerLimit = FRAC32(-1.0);
    windStruct.pParamATO.pParamPI.u16NShift = (tU16)0;
    // Integrator parameters
    windStruct.pParamATO.pParamInteg.f32C1 = FRAC32((Ts/2)*Wmax/pi);
    windStruct.pParamATO.pParamInteg.u16NShift = (tU16)0;

    // Initialize AMCLIB_Windmilling
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB_WindmillingInit_F32(ADC_MAX_OFFSET_ERROR, &windStruct);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB_WindmillingInit(ADC_MAX_OFFSET_ERROR, &windStruct, F32);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 32-bit fractional implementation is selected as default.
    AMCLIB_WindmillingInit(ADC_MAX_OFFSET_ERROR, &windStruct);

    while(1);
}

// Periodical function or interrupt
void ISR(void)
{
    // Measure the 3-phase voltages f32Uabc
    // (...)

    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    retVal = AMCLIB_Windmilling_F32(&f32Uabc, &f32Position,
                                    &f32Velocity, &windStruct);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    retVal = AMCLIB_Windmilling(&f32Uabc, &f32Position,
                                &f32Velocity, &windStruct, F32);
```

```

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 32-bit fractional implementation is selected as default.
retVal = AMCLIB_Windmilling(&f32Uabc, &f32Position,
                            &f32Velocity, &windStruct);
}

```

## 2.8.2 Function AMCLIB\_Windmilling\_F16

### Declaration

```
AMCLIB\_WINDMILLING\_RET\_T AMCLIB_Windmilling_F16(const
SWLIBS\_3Syst\_F16 *pUabcIn, tFrac16 *pPosEst, tFrac16
*pVelocityEst, AMCLIB\_WINDMILLING\_T\_F16 *const pCtrl);
```

### Arguments

Table 77. AMCLIB\_Windmilling\_F16 arguments

Type	Name	Direction	Description
const <a href="#">SWLIBS_3Syst_F16</a> *	pUabcIn	input	Pointer to the structure with the measured 3-phase A/B/C voltages.
<a href="#">tFrac16</a> *	pPosEst	output	Estimated rotor flux position in the interval <-1; 1) which corresponds to the angle range <-π; π) radians.
<a href="#">tFrac16</a> *	pVelocityEst	output	Estimated electrical angular velocity of the rotor. The velocity estimates are noisy, especially for low speeds; therefore, it is recommended to use a <a href="#">GDFLIB_FilterMA_F16</a> to further filter the results. The filter should be engaged only after the AMCLIB_Windmilling has returned <b>SPINNING</b> ; use <a href="#">GDFLIB_FilterMASetState_F16</a> to initialize the filter state with the first estimated velocity value.
<a href="#">AMCLIB_WINDMILLING_T_F16</a> *const	pCtrl	input, output	Pointer to the structure with AMCLIB_Windmilling parameters and state variables.

### Return

The function returns **UNDECIDED** if it is not yet clear whether the rotor is spinning or not. The AMCLIB\_Windmilling must then be called again in the next sampling period. The function returns **SPINNING** if the rotor is spinning. If the motor spin is not detected within 2000 iterations, the function will return **STOPPED**.

### Implementation details

This function internally calls the Tracking Observer function which calculates the PI controller and the integrator subblocks. Refer to the documentation of function [AMCLIB\\_TrackObsrv\\_F16](#) (and its parent chapter describing the AMCLIB\_TrackObsrv functionality) for a description of the necessary scaling and setup of the parameters of the PI controller and integrator blocks used by the AMCLIB\_Windmilling\_F16. It is necessary to call the [AMCLIB\\_WindmillingInit\\_F16](#) before the first use of AMCLIB\_Windmilling\_F16 to initialize the state variables.

**Note:** Due to effectivity reasons, this function is implemented using inline assembly and is therefore not ANSI-C compliant.

## Code Example

```
#define ADC_MAX_OFFSET_ERROR 4
#define Wmax (2618.0F)
#define pi (3.1415927F)
#define Ts (1e-4F)
#define f0 (30.0F)
#define xi (0.707F)
#define w0 (2.0F*pi*f0)
#define Ki (w0*w0)
#define Kp (4.0F*pi*xi*f0)

#include "amclib.h"

SWLIBS_3Syst_F16
AMCLIB_WINDMILLING_T_F16
tFrac16
tFrac16
AMCLIB_WINDMILLING_RET_T     retVal = UNDECIDED;

void main (void)
{
    // Controller parameters
    windStruct.pParamATO.pParamPI.f16CC1sc = FRAC16((Kp+(Ki*Ts)/2)*pi/Wmax);
    windStruct.pParamATO.pParamPI.f16CC2sc = FRAC16((-Kp+(Ki*Ts)/2)*pi/Wmax);
    windStruct.pParamATO.pParamPI.f16UpperLimit = FRAC16(1.0);
    windStruct.pParamATO.pParamPI.f16LowerLimit = FRAC16(-1.0);
    windStruct.pParamATO.pParamPI.u16NShift = (tU16)0;
    // Integrator parameters
    windStruct.pParamATO.pParamInteg.f16C1 = FRAC16((Ts/2)*Wmax/pi);
    windStruct.pParamATO.pParamInteg.u16NShift = (tU16)0;

    // Initialize AMCLIB_Windmilling
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB_WindmillingInit_F16(ADC_MAX_OFFSET_ERROR, &windStruct);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB_WindmillingInit(ADC_MAX_OFFSET_ERROR, &windStruct, F16);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 16-bit fractional implementation is selected as default.
    AMCLIB_WindmillingInit(ADC_MAX_OFFSET_ERROR, &windStruct);

    while(1);
}

// Periodical function or interrupt
void ISR(void)
{
    // Measure the 3-phase voltages f16Uabc
    // (...)

    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    retVal = AMCLIB_Windmilling_F16(&f16Uabc, &f16Position,
```

```

        &f16Velocity, &windStruct);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
retVal = AMCLIB_Windmilling(&f16Uabc, &f16Position,
                            &f16Velocity, &windStruct, F16);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 16-bit fractional implementation is selected as default.
retVal = AMCLIB_Windmilling(&f16Uabc, &f16Position,
                            &f16Velocity, &windStruct);
}

```

### 2.8.3 Function AMCLIB\_Windmilling\_FLT

#### Declaration

```
AMCLIB_WINDMILLING_RET_T AMCLIB_Windmilling_FLT(const
SWLIBS_3Syst_FLT *pUabcIn, tFloat *pPosEst, tFloat *pVelocityEst,
AMCLIB_WINDMILLING_T_FLT *const pCtrl);
```

#### Arguments

Table 78. AMCLIB\_Windmilling\_FLT arguments

Type	Name	Direction	Description
const SWLIBS_3Syst_FLT *	pUabcIn	input	Pointer to the structure with the measured 3-phase A/B/C voltages.
tFloat *	pPosEst	output	Estimated rotor flux position in the interval $<-\pi; \pi$ radians.
tFloat *	pVelocityEst	output	Estimated electrical angular velocity of the rotor. The velocity estimates are noisy, especially for low speeds; therefore, it is recommended to use a <a href="#">GDFLIB_FilterMA_FLT</a> to further filter the results. The filter should be engaged only after the AMCLIB_Windmilling has returned <b>SPINNING</b> ; use <a href="#">GDFLIB_FilterMASetState_FLT</a> to initialize the filter state with the first estimated velocity value.
AMCLIB_WINDMILLING_T_FLT *const	pCtrl	input, output	Pointer to the structure with AMCLIB_Windmilling parameters and state variables.

#### Return

The function returns **UNDECIDED** if it is not yet clear whether the rotor is spinning or not. The AMCLIB\_Windmilling must then be called again in the next sampling period. The function returns **SPINNING** if the rotor is spinning. If the motor spin is not detected within 2000 iterations, the function will return **STOPPED**.

#### Implementation details

This function internally calls the Tracking Observer function which calculates the PI controller and the integrator subblocks. Refer to the documentation of function [AMCLIB\\_TrackObsrv\\_FLT](#) (and its parent chapter describing the AMCLIB\_TrackObsrv functionality) for a description of the setup of the parameters of the PI controller and integrator blocks used by the AMCLIB\_Windmilling\_FLT. It is necessary to call the [AMCLIB\\_WindmillingInit\\_FLT](#) before the first use of AMCLIB\_Windmilling\_FLT to initialize the state variables.

**Note:** The function may raise floating-point exceptions (invalid operation, overflow, underflow, inexact, input denormal). The floating-point unit must be enabled in CPACR register to prevent NOCP UsageFault exception.

### Code Example

```
#define ADC_MAX_OFFSET_ERROR 4e-3
#define pi (3.1415927F)
#define Ts (1e-4F)
#define f0 (30.0F)
#define xi (0.707F)
#define w0 (2.0F*pi*f0)
#define Ki (w0*w0)
#define Kp (4.0F*pi*xi*f0)

#include "amclib.h"

SWLIBS_3SystFLT          fltUabc;
AMCLIB_WINDMILLING_T_FLT windStruct;
tFloat                     fltPosition;
tFloat                     fltVelocity;
AMCLIB_WINDMILLING_RET_T  retVal = UNDECIDED;

void main (void)
{
    // Controller parameters
    windStruct.pParamATO.pParamPI.fltCC1sc = (tFloat) (Kp+(Ki*Ts)/2);
    windStruct.pParamATO.pParamPI.fltCC2sc = (tFloat) (-Kp+(Ki*Ts)/2);
    windStruct.pParamATO.pParamPI.fltUpperLimit = (tFloat) (20000.0);
    windStruct.pParamATO.pParamPI.fltLowerLimit = (tFloat) (-20000.0);
    // Integrator parameters
    windStruct.pParamATO.pParamInteg.fltC1 = (tFloat) (Ts/2);

    // Initialize AMCLIB Windmilling
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB_WindmillingInit_FLT(ADC_MAX_OFFSET_ERROR, &windStruct);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB_WindmillingInit(ADC_MAX_OFFSET_ERROR, &windStruct, FLT);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if single precision floating-point implementation is selected
    // as default.
    AMCLIB_WindmillingInit(ADC_MAX_OFFSET_ERROR, &windStruct);

    while(1);
}

// Periodical function or interrupt
void ISR(void)
{
    // Measure the 3-phase voltages fltUabc
    // (...)

    // Alternative 1: API call with postfix
    AMCLIB_WindmillingUpdate_FLT(fltUabc, &windStruct);
}
```

```

// (only one alternative shall be used).
 retVal = AMCLIB_Windmilling_FLT(&fltUabc, &fltPosition,
                                    &fltVelocity, &windStruct);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
 retVal = AMCLIB_Windmilling(&fltUabc, &fltPosition,
                            &fltVelocity, &windStruct, FLT);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if single precision floating-point implementation is selected
// as default.
 retVal = AMCLIB_Windmilling(&fltUabc, &fltPosition,
                            &fltVelocity, &windStruct);
}

```

## 2.8.4 Function AMCLIB\_WindmillingInit

### Description

This function initializes the internal state variables used by AMCLIB\_Windmilling. AMCLIB\_WindmillingInit must be called before the first use of AMCLIB\_Windmilling.

**Note:** The input/output pointer must contain valid addresses otherwise an exception may occur (Data TLB Error, Data Storage, Alignment, Machine Check).

**Note:** The input/output pointer must contain valid addresses, otherwise a fault may occur (MemManage, BusFault, UsageFault, HardFault).

### Re-entrancy:

The function is re-entrant.

### 2.8.4.1 Function AMCLIB\_WindmillingInit\_F32

#### Declaration

```
void AMCLIB_WindmillingInit_F32(tFrac32 f32ADCMaxError,
AMCLIB\_WINDMILLING\_T\_F32 *const pCtrl);
```

#### Arguments

Table 79. AMCLIB\_WindmillingInit\_F32 arguments

Type	Name	Direction	Description
<a href="#">tFrac32</a>	f32ADCMaxError	input	Maximum differential offset error voltage between any two phases of the 3-phase voltage-sensing A/D converters. This value must be scaled the same way as the measured 3-phase voltages. If the provided value is lower than the actual offset, the AMCLIB_Windmilling may return false positive detection even if the rotor is not spinning.
<a href="#">AMCLIB_WINDMILLING_T_F32</a> *const	pCtrl	input, output	Pointer to the structure with parameters and state variables.

## Implementation details

Function AMCLIB\_WindmillingInit\_F32 must be called before the first use of AMCLIB\_Windmilling\_F32 to initialize the state variables.

## Code Example

```
#define ADC_MAX_OFFSET_ERROR 4
#define Wmax (2618.0F)
#define pi (3.1415927F)
#define Ts (1e-4F)
#define f0 (30.0F)
#define xi (0.707F)
#define w0 (2.0F*pi*f0)
#define Ki (w0*w0)
#define Kp (4.0F*pi*xi*f0)

#include "amclib.h"

SWLIBS_3Syst_F32          f32Uabc;
AMCLIB_WINDMILLING_T_F32   windStruct;
tFrac32                     f32Position;
tFrac32                     f32Velocity;
AMCLIB_WINDMILLING_RET_T    retVal = UNDECIDED;

void main (void)
{
    // Controller parameters
    windStruct.pParamATO.pParamPI.f32CC1sc = FRAC32((Kp+(Ki*Ts)/2)*pi/Wmax);
    windStruct.pParamATO.pParamPI.f32CC2sc = FRAC32((-Kp+(Ki*Ts)/2)*pi/Wmax);
    windStruct.pParamATO.pParamPI.f32UpperLimit = FRAC32(1.0);
    windStruct.pParamATO.pParamPI.f32LowerLimit = FRAC32(-1.0);
    windStruct.pParamATO.pParamPI.u16NShift = (tu16)0;
    // Integrator parameters
    windStruct.pParamATO.pParamInteg.f32C1 = FRAC32((Ts/2)*Wmax/pi);
    windStruct.pParamATO.pParamInteg.u16NShift = (tu16)0;

    // Initialize AMCLIB_Windmilling
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB_WindmillingInit_F32(ADC_MAX_OFFSET_ERROR, &windStruct);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB_WindmillingInit(ADC_MAX_OFFSET_ERROR, &windStruct, F32);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 32-bit fractional implementation is selected as default.
    AMCLIB_WindmillingInit(ADC_MAX_OFFSET_ERROR, &windStruct);

    while(1);
}

// Periodical function or interrupt
void ISR(void)
{
    // Measure the 3-phase voltages f32Uabc
```

```

// (...)

// Alternative 1: API call with postfix
// (only one alternative shall be used).
retVal = AMCLIB\_Windmilling\_F32(&f32Uabc, &f32Position,
                                &f32Velocity, &windStruct);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
retVal = AMCLIB_Windmilling(&f32Uabc, &f32Position,
                            &f32Velocity, &windStruct, F32);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 32-bit fractional implementation is selected as default.
retVal = AMCLIB_Windmilling(&f32Uabc, &f32Position,
                            &f32Velocity, &windStruct);

}

```

#### 2.8.4.2 Function [AMCLIB\\_WindmillingInit\\_F16](#)

##### Declaration

```
void AMCLIB_WindmillingInit_F16(tFrac16 f16ADCMaxError,
AMCLIB\_WINDMILLING\_T\_F16 *const pCtrl);
```

##### Arguments

**Table 80. AMCLIB\_WindmillingInit\_F16 arguments**

Type	Name	Direction	Description
<a href="#">tFrac16</a>	f16ADCMaxError	input	Maximum differential offset error voltage between any two phases of the 3-phase voltage-sensing A/D converters. This value must be scaled the same way as the measured 3-phase voltages. If the provided value is lower than the actual offset, the AMCLIB_Windmilling may return false positive detection even if the rotor is not spinning.
<a href="#">AMCLIB_WINDMILLING_T_F16</a> *const	pCtrl	input, output	Pointer to the structure with parameters and state variables.

##### Implementation details

Function AMCLIB\_WindmillingInit\_F16 must be called before the first use of AMCLIB\_Windmilling\_F16 to initialize the state variables.

##### Code Example

```
#define ADC_MAX_OFFSET_ERROR 4
#define Wmax (2618.0F)
#define pi (3.1415927F)
#define Ts (1e-4F)
#define f0 (30.0F)
#define xi (0.707F)
#define w0 (2.0F*pi*f0)
#define Ki (w0*w0)
#define Kp (4.0F*pi*xi*f0)
```

```
#include "amclib.h"

_SWLIBS_3Syst_F16
AMCLIB_WINDMILLING_T_F16
tFrac16
tFrac16
AMCLIB_WINDMILLING_RET_T           f16Uabc;
windStruct;
f16Position;
f16Velocity;
retval = UNDECIDED;

void main (void)
{
    // Controller parameters
    windStruct.pParamATO.pParamPI.f16CC1sc = FRAC16((Kp+(Ki*Ts)/2)*pi/Wmax);
    windStruct.pParamATO.pParamPI.f16CC2sc = FRAC16((-Kp+(Ki*Ts)/2)*pi/Wmax);
    windStruct.pParamATO.pParamPI.f16UpperLimit = FRAC16(1.0);
    windStruct.pParamATO.pParamPI.f16LowerLimit = FRAC16(-1.0);
    windStruct.pParamATO.pParamPI.ul6NShift = (tU16)0;
    // Integrator parameters
    windStruct.pParamATO.pParamInteg.f16C1 = FRAC16((Ts/2)*Wmax/pi);
    windStruct.pParamATO.pParamInteg.ul6NShift = (tU16)0;

    // Initialize AMCLIB_Windmilling
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    AMCLIB_WindmillingInit_F16(ADC_MAX_OFFSET_ERROR, &windStruct);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    AMCLIB_WindmillingInit(ADC_MAX_OFFSET_ERROR, &windStruct, F16);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 16-bit fractional implementation is selected as default.
    AMCLIB_WindmillingInit(ADC_MAX_OFFSET_ERROR, &windStruct);

    while(1);
}

// Periodical function or interrupt
void ISR(void)
{
    // Measure the 3-phase voltages f16Uabc
    // (...)

    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    retval = AMCLIB_Windmilling_F16(&f16Uabc, &f16Position,
                                    &f16Velocity, &windStruct);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    retval = AMCLIB_Windmilling(&f16Uabc, &f16Position,
                               &f16Velocity, &windStruct, F16);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 16-bit fractional implementation is selected as default.
    retval = AMCLIB_Windmilling(&f16Uabc, &f16Position,
                               &f16Velocity, &windStruct);
```

```
}
```

### 2.8.4.3 Function AMCLIB\_WindmillingInit\_FLT

#### Declaration

```
void AMCLIB_WindmillingInit_FLT(tFloat fltADCMaxError,  
AMCLIB\_WINDMILLING\_T\_FLT *const pCtrl);
```

#### Arguments

**Table 81. AMCLIB\_WindmillingInit\_FLT arguments**

Type	Name	Direction	Description
<a href="#">tFloat</a>	fltADCMaxError	<a href="#">input</a>	Maximum differential offset error voltage between any two phases of the 3-phase voltage-sensing A/D converters. If the provided value is lower than the actual offset, the AMCLIB_Windmilling may return false positive detection even if the rotor is not spinning. This value must not be zero.
<a href="#">AMCLIB_WINDMILLING_T_FLT</a> *const	pCtrl	<a href="#">input, output</a>	Pointer to the structure with parameters and state variables.

#### Implementation details

Function AMCLIB\_WindmillingInit\_FLT must be called before the first use of AMCLIB\_Windmilling\_FLT to initialize the state variables.

**Note:** The function may raise floating-point exceptions (invalid operation, overflow, underflow, inexact, input denormal). The floating-point unit must be enabled in CPACR register to prevent NOCP UsageFault exception.

#### Code Example

```
#define ADC_MAX_OFFSET_ERROR 4e-3
#define pi (3.1415927F)
#define Ts (1e-4F)
#define f0 (30.0F)
#define xi (0.707F)
#define w0 (2.0F*pi*f0)
#define Ki (w0*w0)
#define Kp (4.0F*pi*xi*f0)

#include "amclib.h"

SWLIBS\_3Syst\_FLT fltUabc;
AMCLIB\_WINDMILLING\_T\_FLT windStruct;
tFloat fltPosition;
tFloat fltVelocity;
AMCLIB\_WINDMILLING\_RET\_T retVal = UNDECIDED;

void main (void)
{
    // Controller parameters
    windStruct.pParamATO.pParamPI.fltCC1sc = (tFloat) (Kp+(Ki*Ts)/2);
    windStruct.pParamATO.pParamPI.fltCC2sc = (tFloat) (-Kp+(Ki*Ts)/2);
    windStruct.pParamATO.pParamPI.fltUpperLimit = (tFloat) (1.0);
```

```

windStruct.pParamATO.pParamPI.fltLowerLimit = (tFloat) (-1.0);
// Integrator parameters
windStruct.pParamATO.pParamInteg.fltC1 = (tFloat) (Ts/2);

// Initialize AMCLIB_Windmilling
// Alternative 1: API call with postfix
// (only one alternative shall be used).
AMCLIB_WindmillingInit_FLT(ADC_MAX_OFFSET_ERROR, &windStruct);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
AMCLIB_WindmillingInit(ADC_MAX_OFFSET_ERROR, &windStruct, FLT);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if single precision floating-point implementation is selected
// as default.
AMCLIB_WindmillingInit(ADC_MAX_OFFSET_ERROR, &windStruct);

while(1);

}

// Periodical function or interrupt
void ISR(void)
{
    // Measure the 3-phase voltages fltUabc
    // (...)

    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    retVal = AMCLIB\_Windmilling\_FLT(&fltUabc, &fltPosition,
                                    &fltVelocity, &windStruct);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    retVal = AMCLIB_Windmilling(&fltUabc, &fltPosition,
                                &fltVelocity, &windStruct, FLT);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if single precision floating-point implementation is selected
    // as default.
    retVal = AMCLIB_Windmilling(&fltUabc, &fltPosition,
                                &fltVelocity, &windStruct);
}

```

## 2.9 Function GDFLIB\_FilterFIR

The function performs a single iteration of an finite impulse response (FIR) filter.

### Description

The function calculates the response of a direct-form FIR filter for one input sample. The FIR filter is defined by the difference equation

$$y(n) = \sum_{k=0}^N h_k x(n-k)$$

Equation GDFLIB\_FilterFIR\_Eq1

where  $x$  is the input signal,  $y$  is the output signal,  $h$  are the filter coefficients, and  $N$  is the filter order. An  $N$ -th order FIR filter requires  $N + 1$  coefficients.

Function [GDFLIB\\_FilterFIRInit](#) should be called before the first use of the filter to clear the state buffer.

**Note:** The input/output pointers must contain valid addresses, otherwise a fault may occur (MemManage, BusFault, UsageFault, HardFault).

## Re-entrancy

The function is re-entrant for a different pState.

### 2.9.1 Function GDFLIB\_FilterFIR\_F32

#### Declaration

```
tFrac32 GDFLIB_FilterFIR_F32(tFrac32 f32In,
const GDFLIB_FILTERFIR_PARAM_T_F32 *const pParam,
GDFLIB_FILTERFIR_STATE_T_F32 *const pState);
```

#### Arguments

Table 82. GDFLIB\_FilterFIR\_F32 arguments

Type	Name	Direction	Description
tFrac32	f32In	input	Input sample.
const GDFLIB_FILTERFIR_PARAM_T_F32 *const	pParam	input	Pointer to the parameter structure.
GDFLIB_FILTERFIR_STATE_T_F32 *const	pState	input, output	Pointer to the filter state structure.

#### Return

Filter response for the input sample.

#### Implementation details

The implementation uses a 64-bit accumulator in 32.31 format. Multiplication results are calculated in a precision limited to 31 fractional bits. The accumulator is allowed to wrap-around during calculation, the final accumulator state is saturated to the 1.31 format.

#### Code Example

```
#include "gdflib.h"

#define FIR_ORDER 15
#define FIR_NUMTAPS ((FIR_ORDER) + 1)

GDFLIB_FILTERFIR_PARAM_T_F32 Param;
GDFLIB_FILTERFIR_STATE_T_F32 State0, State1, State2;

tFrac32 f32InBuf[FIR_NUMTAPS];
tFrac32 f32CoefBuf[FIR_NUMTAPS];
```

```
#define OUT_LEN 16

void main(void)
{
    int ii;
    tFrac32 f32OutBuf0[OUT_LEN];
    tFrac32 f32OutBuf1[OUT_LEN];
    tFrac32 f32OutBuf2[OUT_LEN];

    // Define a simple low-pass filter
    // The filter coefficients were calculated by the following
    // Matlab function (coefficients are contained in Hd.Numerator):
    //
    // function Hd = fir_example
    // FIR_EXAMPLE Returns a discrete-time filter object.
    // N      = 15;
    // F6dB  = 0.5;
    //
    // h = fdesign.lowpass('n,fc', N, F6dB);
    //
    // Hd = design(h, 'window');
    // return;
    ii = 0;
    f32CoefBuf[ii++] = 0xFFB10C14;
    f32CoefBuf[ii++] = 0xFF779D25;
    f32CoefBuf[ii++] = 0x01387DD7;
    f32CoefBuf[ii++] = 0x028E6845;
    f32CoefBuf[ii++] = 0xFB245142;
    f32CoefBuf[ii++] = 0xF7183CC7;
    f32CoefBuf[ii++] = 0x11950A3C;
    f32CoefBuf[ii++] = 0x393ED867;
    f32CoefBuf[ii++] = 0x393ED867;
    f32CoefBuf[ii++] = 0x11950A3C;
    f32CoefBuf[ii++] = 0xF7183CC7;
    f32CoefBuf[ii++] = 0xFB245142;
    f32CoefBuf[ii++] = 0x028E6845;
    f32CoefBuf[ii++] = 0x01387DD7;
    f32CoefBuf[ii++] = 0xFF779D25;
    f32CoefBuf[ii++] = 0xFFB10C14;

    Param.u32Order = FIR_ORDER;
    Param.pCoefBuf = &f32CoefBuf[0];

    // Initialize FIR filter
    GDFLIB_FilterFIRInit_F32(&Param, &State0, &f32InBuf[0]);

    // Initialize FIR filter
    GDFLIB_FilterFIRInit(&Param, &State1, &f32InBuf[0], F32);

    // ##### Available only if 32-bit fractional implementation selected
    // as default
    // #####
    // Initialize FIR filter
    GDFLIB_FilterFIRInit(&Param, &State2, &f32InBuf[0]);

    // Compute step response of the filter
    for(ii=0; ii < OUT_LEN; ii++)
    {
```

```

// f32OutBuf0 contains step response of the filter
f32OutBuf0[ii] = GDFLIB_FilterFIR_F32(0x7FFFFFFF, &Param, &State0);

// f32OutBuf1 contains step response of the filter
f32OutBuf1[ii] = GDFLIB_FilterFIR(0x7FFFFFFF, &Param, &State1, F32);

// ######
// Available only if 32-bit fractional implementation selected
// as default
// #####
// f32OutBuf2 contains step response of the filter
f32OutBuf2[ii] = GDFLIB_FilterFIR(0x7FFFFFFF, &Param, &State2);
}
// After the loop the f32OutBuf0, f32OutBuf1, f32OutBuf2 shall contains
// the following values:
// {0xFFB1009E, 0xFF2801B0, 0x005FFF40, 0x02EDFA24, 0xFE1203DC,
// 0xF52A15AC, 0x06BEF282, 0x3FFC8006, 0x793A0D8A, 0x7FFFFFFF,
// 0x7FFFFFFF, 0x7D0B05E8, 0x7F9900CC, 0x7FFFFFFF, 0x7FFFFFFF,
// 0x7FF9000C}
}

```

## 2.9.2 Function GDFLIB\_FilterFIR\_F16

### Declaration

```
tFrac16 GDFLIB_FilterFIR_F16(tFrac16 f16In,
const GDFLIB_FILTERFIR_PARAM_T_F16 *const pParam,
GDFLIB_FILTERFIR_STATE_T_F16 *const pState);
```

### Arguments

Table 83. GDFLIB\_FilterFIR\_F16 arguments

Type	Name	Direction	Description
tFrac16	f16In	input	Input sample.
const GDFLIB_FILTERFIR_PARAM_T_F16 *const	pParam	input	Pointer to the parameter structure.
GDFLIB_FILTERFIR_STATE_T_F16 *const	pState	input, output	Pointer to the filter state structure.

### Return

Filter response for the input sample.

### Implementation details

The implementation uses a 64-bit accumulator in 32.31 format. Multiplication results are calculated in full precision. The accumulator is allowed to wrap-around during calculation, the final accumulator state is saturated to the 1.15 format.

### Code Example

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

```
#define FIR_ORDER 15
#define FIR_NUMTAPS ((FIR_ORDER) + 1)

_GDFLIB_FILTERFIR_PARAM_T_F16 Param;
_GDFLIB_FILTERFIR_STATE_T_F16 State0, State1, State2;

_tFrac16 f16InBuf[FIR_NUMTAPS + 1] __attribute__((aligned(4)));
_tFrac16 f16CoefBuf[FIR_NUMTAPS + 1] __attribute__((aligned(4)));

#define OUT_LEN 16

void main(void)
{
    int ii;
    _tFrac16 f16OutBuf0[OUT_LEN];
    _tFrac16 f16OutBuf1[OUT_LEN];
    _tFrac16 f16OutBuf2[OUT_LEN];

    // Define a simple low-pass filter
    // The filter coefficients were calculated by the following
    // Matlab function (coefficients are contained in Hd.Numerator):
    //
    // function Hd = fir_example
    // FIR_EXAMPLE Returns a discrete-time filter object.
    // N      = 15;
    // F6dB  = 0.5;
    //
    // h = fdesign.lowpass('n,fc', N, F6dB);
    //
    // Hd = design(h, 'window');
    // return;
    ii = 0;
    f16CoefBuf[ii++] = 0xFFB1;
    f16CoefBuf[ii++] = 0xFF77;
    f16CoefBuf[ii++] = 0x0138;
    f16CoefBuf[ii++] = 0x028E;
    f16CoefBuf[ii++] = 0xFB24;
    f16CoefBuf[ii++] = 0xF718;
    f16CoefBuf[ii++] = 0x1195;
    f16CoefBuf[ii++] = 0x393E;
    f16CoefBuf[ii++] = 0x393E;
    f16CoefBuf[ii++] = 0x1195;
    f16CoefBuf[ii++] = 0xF718;
    f16CoefBuf[ii++] = 0xFB24;
    f16CoefBuf[ii++] = 0x028E;
    f16CoefBuf[ii++] = 0x0138;
    f16CoefBuf[ii++] = 0xFF77;
    f16CoefBuf[ii++] = 0xFFB1;

    Param.u16Order = FIR_ORDER;
    Param.pCoefBuf = &f16CoefBuf[0];

    // Initialize FIR filter
    _GDFLIB_FilterFIRInit_F16(&Param, &State0, &f16InBuf[0]);

    // Initialize FIR filter
    _GDFLIB_FilterFIRInit(&Param, &State1, &f16InBuf[0], F16);

    ##### Available only if 16-bit fractional implementation selected
    // Available only if 16-bit fractional implementation selected
```

```

// as default
// #####
// Initialize FIR filter
GDFLIB_FilterFIRInit(&Param, &State2, &f16InBuf[0]);

// Compute step response of the filter
for(ii=0; ii < OUT_LEN; ii++)
{
    // f16OutBuf0 contains step response of the filter
    f16OutBuf0[ii] = GDFLIB_FilterFIR_F16(0x7FFF, &Param, &State0);

    // f16OutBuf1 contains step response of the filter
    f16OutBuf1[ii] = GDFLIB_FilterFIR(0x7FFF, &Param, &State1, F16);

    // #####
    // Available only if 16-bit fractional implementation selected
    // as default
    // #####
    // f16OutBuf2 contains step response of the filter
    f16OutBuf2[ii] = GDFLIB_FilterFIR(0x7FFF, &Param, &State2);
}
// After the loop the f16OutBuf0, f16OutBuf1, f16OutBuf2 shall
// contains the following values:
// {0xFFB1, 0xFF28, 0x005F, 0x02ED, 0xFE12, 0xF52A, 0x06BE, 0x3FFC,
// 0x793A, 0x7FFF, 0x7FFF, 0x7D0B, 0x7F99, 0x7FFF, 0x7FFF, 0x7FF9}
}

```

### 2.9.3 Function GDFLIB\_FilterFIR\_FLT

#### Declaration

```
tFloat GDFLIB_FilterFIR_FLT(tFloat fltIn, const
GDFLIB_FILTERFIR_PARAM_T_FLT *const pParam,
GDFLIB_FILTERFIR_STATE_T_FLT *const pState);
```

#### Arguments

**Table 84. GDFLIB\_FilterFIR\_FLT arguments**

Type	Name	Direction	Description
tFloat	fltIn	input	Input sample.
const GDFLIB_FILTERFIR_PARAM_T_FLT *const	pParam	input	Pointer to a parameter structure.
GDFLIB_FILTERFIR_STATE_T_FLT *const	pState	input, output	Pointer to a filter state structure.

#### Return

Filter response for the input sample.

**Note:** The function may raise floating-point exceptions (invalid operation, overflow, underflow, inexact, input denormal). The floating-point unit must be enabled in CPACR register to prevent NOCP UsageFault exception.

## Code Example

```
#include "gdflib.h"

#define FIR_ORDER 15
#define FIR_NUMTAPS ((FIR_ORDER) + 1)

GDFLIB_FILTERFIR_PARAM_T_FLT Param;
GDFLIB_FILTERFIR_STATE_T_FLT State0, State1, State2;

tFloat fltInBuf[FIR_NUMTAPS];
tFloat fltCoefBuf[FIR_NUMTAPS];

#define OUT_LEN 16

void main(void)
{
    int ii;
    tFloat fltOutBuf0[OUT_LEN];
    tFloat fltOutBuf1[OUT_LEN];
    tFloat fltOutBuf2[OUT_LEN];

    // Define a simple low-pass filter
    // The filter coefficients were calculated by the following
    // Matlab function (coefficients are contained in Hd.Numerator):
    //
    // function Hd = fir_example
    // FIR_EXAMPLE Returns a discrete-time filter object.
    // N = 15;
    // F6dB = 0.5;
    //
    // h = fdesign.lowpass('n,fc', N, F6dB);
    //
    // Hd = design(h, 'window');
    // return;
    ii = 0;
    fltCoefBuf[ii++] = -0.997590551617365;
    fltCoefBuf[ii++] = -0.995837825348525;
    fltCoefBuf[ii++] = 0.0095364856578114;
    fltCoefBuf[ii++] = 0.0199709259997918;
    fltCoefBuf[ii++] = -0.962045819946586;
    fltCoefBuf[ii++] = -0.930427167532233;
    fltCoefBuf[ii++] = 0.137360839237161;
    fltCoefBuf[ii++] = 0.447230387221663;
    fltCoefBuf[ii++] = 0.447230387221663;
    fltCoefBuf[ii++] = 0.137360839237161;
    fltCoefBuf[ii++] = -0.30427167532233;
    fltCoefBuf[ii++] = -0.962045819946586;
    fltCoefBuf[ii++] = 0.199709259997918;
    fltCoefBuf[ii++] = 0.0095364856578114;
    fltCoefBuf[ii++] = -0.995837825348525;
    fltCoefBuf[ii++] = -0.997590551617365;

    Param.u32Order = FIR_ORDER;
    Param.pCoefBuf = &fltCoefBuf[0];

    // Initialize FIR filter
    GDFLIB_FilterFIRInit_FLT(&Param, &State0, &fltInBuf[0]);
}
```

```
// Initialize FIR filter
GDFLIB_FilterFIRInit(&Param, &State1, &fltInBuf[0], FLT);

// ##### Available only if single precision floating point
// implementation selected as default
// #####
// Initialize FIR filter
GDFLIB_FilterFIRInit(&Param, &State2, &fltInBuf[0]);

// Compute step response of the filter
for(ii=0; ii < OUT_LEN; ii++)
{
    // fltOutBuf0 contains step response of the filter
    fltOutBuf0[ii] = GDFLIB_FilterFIR_FLT((tFloat)(1), &Param, &State0);

    // fltOutBuf1 contains step response of the filter
    fltOutBuf1[ii] = GDFLIB_FilterFIR((tFloat)(1), &Param, &State1, FLT);

    // #####
    // Available only if single precision floating point
    // implementation selected as default
    // #####
    // fltOutBuf2 contains step response of the filter
    fltOutBuf2[ii] = GDFLIB_FilterFIR((tFloat)(1), &Param, &State2);
}
// After the loop the fltOutBuf0, fltOutBuf1, fltOutBuf2 shall contains
// the following values:
// {-0.99759054, -1.9934283, -1.9838918, -1.963921, -2.9259667,
// -3.8563938, -3.719033, -3.2718027, -2.8245723, -2.6872115,
// -2.9914832, -3.9535291, -3.7538199, -3.7442834, -4.7401214,
// -5.7377119}
```

#### 2.9.4 Function GDFLIB\_FilterFIRInit

This function initializes the FIR filter buffers.

##### Description

The function performs the initialization procedure for the GDFLIB\_FilterFIR function. In particular, the function performs the following operations:

1. Resets the input buffer index to zero.
2. Initializes the input buffer pointer to the pointer provided as an argument.
3. Resets the input buffer.

After initialization, made by the function, the parameters and state structures should be provided as arguments to calls of the GDFLIB\_FilterFIR function.

**Note:** The input/output pointers must contain valid addresses, otherwise a fault may occur (MemManage, BusFault, UsageFault, HardFault).

*The input buffer pointer (State->pInBuf) must point to a Read/Write memory region, which must be at least the number of the filter taps long. The number of taps in a filter is*

*equal to the filter order + 1. There is no restriction as to the location of the parameters structure as long as it is readable.*

**Caution:** *No check is performed for R/W capability and the length of the input buffer (pState->pInBuf). In case of passing incorrect pointer to the function, an unexpected behavior of the function might be expected including the incorrect memory access exception.*

### Re-entrancy

The function is re-entrant only if the calling code is provided with a distinct instance of the structure pointed to by pState.

#### 2.9.4.1 Function GDFLIB\_FilterFIRInit\_F32

##### Declaration

```
void GDFLIB_FilterFIRInit_F32(const GDFLIB_FILTERFIR_PARAM_T_F32
*const pParam, GDFLIB_FILTERFIR_STATE_T_F32 *const pState,
tFrac32 *pInBuf);
```

##### Arguments

Table 85. GDFLIB\_FilterFIRInit\_F32 arguments

Type	Name	Direction	Description
const GDFLIB_FILTERFIR_PARAM_T_F32 *const	pParam	input	Pointer to the parameters structure.
GDFLIB_FILTERFIR_STATE_T_F32 *const	pState	input, output	Pointer to the state structure.
tFrac32 *	pInBuf	input, output	Pointer to a buffer for storing filter input signal values, must point to a R/W memory region and must be a filter order + 1 long.

##### Implementation details

The function performs the initialization procedure for the [GDFLIB\\_FilterFIR\\_F32](#) function. In particular, the function performs the following operations:

1. Resets the input buffer index to zero.
2. Initializes the input buffer pointer to the pointer provided as an argument.
3. Resets the input buffer.

After initialization, made by the function, the parameters and state structures should be provided as arguments to calls of the [GDFLIB\\_FilterFIR\\_F32](#) function.

##### Code Example

```
#include "gdflib.h"

#define FIR_ORDER 15
#define FIR_NUMTAPS ((FIR_ORDER) + 1)

GDFLIB_FILTERFIR_PARAM_T_F32 Param;
GDFLIB_FILTERFIR_STATE_T_F32 State0, State1, State2;
```

```
tFrac32 f32InBuf[FIR_NUMTAPS];
tFrac32 f32CoefBuf[FIR_NUMTAPS];

#define OUT_LEN 16

void main(void)
{
    int ii;
    tFrac32 f32OutBuf0[OUT_LEN];
    tFrac32 f32OutBuf1[OUT_LEN];
    tFrac32 f32OutBuf2[OUT_LEN];

    // Define a simple low-pass filter
    // The filter coefficients were calculated by the following
    // Matlab function (coefficients are contained in Hd.Numerator):
    //
    // function Hd = fir_example
    // FIR_EXAMPLE Returns a discrete-time filter object.
    // N      = 15;
    // F6dB  = 0.5;
    //
    // h = fdesign.lowpass('n,fc', N, F6dB);
    //
    // Hd = design(h, 'window');
    // return;
    ii = 0;
    f32CoefBuf[ii++] = 0xFFB10C14;
    f32CoefBuf[ii++] = 0xFF779D25;
    f32CoefBuf[ii++] = 0x01387DD7;
    f32CoefBuf[ii++] = 0x028E6845;
    f32CoefBuf[ii++] = 0xFB245142;
    f32CoefBuf[ii++] = 0xF7183CC7;
    f32CoefBuf[ii++] = 0x11950A3C;
    f32CoefBuf[ii++] = 0x393ED867;
    f32CoefBuf[ii++] = 0x393ED867;
    f32CoefBuf[ii++] = 0x11950A3C;
    f32CoefBuf[ii++] = 0xF7183CC7;
    f32CoefBuf[ii++] = 0xFB245142;
    f32CoefBuf[ii++] = 0x028E6845;
    f32CoefBuf[ii++] = 0x01387DD7;
    f32CoefBuf[ii++] = 0xFF779D25;
    f32CoefBuf[ii++] = 0xFFB10C14;

    Param.u32Order = FIR_ORDER;
    Param.pCoefBuf = &f32CoefBuf[0];

    // Initialize FIR filter
    GDFLIB_FilterFIRInit_F32(&Param, &State0, &f32InBuf[0]);

    // Initialize FIR filter
    GDFLIB_FilterFIRInit(&Param, &State1, &f32InBuf[0], F32);

    // ##### Available only if 32-bit fractional implementation selected
    // as default
    // #####
    // Initialize FIR filter
    GDFLIB_FilterFIRInit(&Param, &State2, &f32InBuf[0]);
```

```

// Compute step response of the filter
for(ii=0; ii < OUT_LEN; ii++)
{
    // f32OutBuf0 contains step response of the filter
    f32OutBuf0[ii] = GDFLIB_FilterFIR_F32(0x7FFFFFFF, &Param, &State0);

    // f32OutBuf1 contains step response of the filter
    f32OutBuf1[ii] = GDFLIB_FilterFIR(0x7FFFFFFF, &Param, &State1, F32);

    // ##### Available only if 32-bit fractional implementation selected
    // as default
    // #####
    // #####
    // f32OutBuf2 contains step response of the filter
    f32OutBuf2[ii] = GDFLIB_FilterFIR(0x7FFFFFFF, &Param, &State2);
}
// After the loop the f32OutBuf0, f32OutBuf1, f32OutBuf2 shall contains
// the following values:
// {0xFFB1009E, 0xFF2801B0, 0x005FFF40, 0x02EDFA24, 0xFE1203DC,
// 0xF52A15AC, 0x06BEF282, 0x3FFC8006, 0x793A0D8A, 0x7FFFFFFF,
// 0x7FFFFFFF, 0x7D0B05E8, 0x7F9900CC, 0x7FFFFFFF, 0x7FFFFFFF,
// 0x7FF9000C}
}

```

#### 2.9.4.2 Function GDFLIB\_FilterFIRInit\_F16

##### Declaration

```
void GDFLIB_FilterFIRInit_F16(const GDFLIB_FILTERFIR_PARAM_T_F16
    *const pParam, GDFLIB_FILTERFIR_STATE_T_F16 *const pState,
    tFrac16 *pInBuf);
```

##### Arguments

**Table 86.** GDFLIB\_FilterFIRInit\_F16 arguments

Type	Name	Direction	Description
const GDFLIB_FILTERFIR_PARAM_T_F16 *const	pParam	input	Pointer to the parameters structure.
GDFLIB_FILTERFIR_STATE_T_F16 *const	pState	input, output	Pointer to the state structure.
tFrac16 *	pInBuf	input, output	Pointer to a buffer for storing filter input signal values, must point to a R/W memory region and must be a filter order + 1 long.

##### Code Example

```

#include "gdflib.h"

#define FIR_ORDER 15
#define FIR_NUMTAPS ((FIR_ORDER) + 1)

GDFLIB_FILTERFIR_PARAM_T_F16 Param;
GDFLIB_FILTERFIR_STATE_T_F16 State0, State1, State2;

```

```
tFrac16 f16InBuf[FIR_NUMTAPS];
tFrac16 f16CoefBuf[FIR_NUMTAPS];

#define OUT_LEN 16

void main(void)
{
    int ii;
    tFrac16 f16OutBuf0[OUT_LEN];
    tFrac16 f16OutBuf1[OUT_LEN];
    tFrac16 f16OutBuf2[OUT_LEN];

    // Define a simple low-pass filter
    // The filter coefficients were calculated by the following
    // Matlab function (coefficients are contained in Hd.Numerator):
    //
    // function Hd = fir_example
    // FIR_EXAMPLE Returns a discrete-time filter object.
    // N      = 15;
    // F6dB  = 0.5;
    //
    // h = fdesign.lowpass('n,fc', N, F6dB);
    //
    // Hd = design(h, 'window');
    // return;
    ii = 0;
    f16CoefBuf[ii++] = 0xFFB1;
    f16CoefBuf[ii++] = 0xFF77;
    f16CoefBuf[ii++] = 0x0138;
    f16CoefBuf[ii++] = 0x028E;
    f16CoefBuf[ii++] = 0xFB24;
    f16CoefBuf[ii++] = 0xF718;
    f16CoefBuf[ii++] = 0x1195;
    f16CoefBuf[ii++] = 0x393E;
    f16CoefBuf[ii++] = 0x393E;
    f16CoefBuf[ii++] = 0x1195;
    f16CoefBuf[ii++] = 0xF718;
    f16CoefBuf[ii++] = 0xFB24;
    f16CoefBuf[ii++] = 0x028E;
    f16CoefBuf[ii++] = 0x0138;
    f16CoefBuf[ii++] = 0xFF77;
    f16CoefBuf[ii++] = 0xFFB1;

    Param.ul6Order = FIR_ORDER;
    Param.pCoefBuf = &f16CoefBuf[0];

    // Initialize FIR filter
    GDFLIB_FilterFIRInit_F16(&Param, &State0, &f16InBuf[0]);

    // Initialize FIR filter
    GDFLIB_FilterFIRInit(&Param, &State1, &f16InBuf[0], F16);

    // ##### Available only if 16-bit fractional implementation selected
    // as default
    // #####
    // Initialize FIR filter
    GDFLIB_FilterFIRInit(&Param, &State2, &f16InBuf[0]);
```

```

// Compute step response of the filter
for(ii=0; ii < OUT_LEN; ii++)
{
    // f16OutBuf0 contains step response of the filter
    f16OutBuf0[ii] = GDFLIB_FilterFIR_F16(0x7FFF, &Param, &State0);

    // f16OutBuf1 contains step response of the filter
    f16OutBuf1[ii] = GDFLIB_FilterFIR(0x7FFF, &Param, &State1, F16);

    // ##### Available only if 16-bit fractional implementation selected
    // as default
    // #####
    // f16OutBuf2 contains step response of the filter
    f16OutBuf2[ii] = GDFLIB_FilterFIR(0x7FFF, &Param, &State2);
}
// After the loop the f16OutBuf0, f16OutBuf1, f16OutBuf2 shall
// contains the following values:
// {0xFFB1, 0xFF28, 0x005F, 0x02ED, 0xFE12, 0xF52A, 0x06BE, 0x3FFC,
// 0x793A, 0x7FFF, 0x7FFF, 0x7D0B, 0x7F99, 0x7FFF, 0x7FFF, 0x7FF9}
}

```

#### 2.9.4.3 Function GDFLIB\_FilterFIRInit\_FLT

##### Declaration

```
void GDFLIB_FilterFIRInit_FLT(const GDFLIB_FILTERFIR_PARAM_T_FLT
*const pParam, GDFLIB_FILTERFIR_STATE_T_FLT *const pState, tFloat
*pInBuf);
```

##### Arguments

**Table 87. GDFLIB\_FilterFIRInit\_FLT arguments**

Type	Name	Direction	Description
const GDFLIB_FILTERFIR_PARAM_T_FLT *const	pParam	input	Pointer to a parameters structure.
GDFLIB_FILTERFIR_STATE_T_FLT *const	pState	input, output	Pointer to a state structure.
tFloat *	pInBuf	input, output	Pointer to a buffer for storing filter input signal values, must point to a R/W memory region and must be the filter order + 1 long.

**Note:** The floating-point unit must be enabled in CPACR register to prevent NOCP UsageFault exception.

##### Code Example

```

#include "gdflib.h"

#define FIR_ORDER 15
#define FIR_NUMTAPS ((FIR_ORDER) + 1)

GDFLIB_FILTERFIR_PARAM_T_FLT Param;
GDFLIB_FILTERFIR_STATE_T_FLT State0, State1, State2;

```

```
tFloat fltInBuf[FIR_ORDER];
tFloat fltCoefBuf[FIR_ORDER];

#define OUT_LEN 16

void main(void)
{
    int ii;
    tFloat fltOutBuf0[OUT_LEN];
    tFloat fltOutBuf1[OUT_LEN];
    tFloat fltOutBuf2[OUT_LEN];

    // Define a simple low-pass filter
    // The filter coefficients were calculated by the following
    // Matlab function (coefficients are contained in Hd.Numerator):
    //
    // function Hd = fir_example
    // FIR_EXAMPLE Returns a discrete-time filter object.
    // N      = 15;
    // F6dB = 0.5;
    //
    // h = fdesign.lowpass('n,fc', N, F6dB);
    //
    // Hd = design(h, 'window');
    // return;
    ii = 0;
    fltCoefBuf[ii++] = -0.997590551617365;
    fltCoefBuf[ii++] = -0.995837825348525;
    fltCoefBuf[ii++] = 0.0095364856578114;
    fltCoefBuf[ii++] = 0.0199709259997918;
    fltCoefBuf[ii++] = -0.962045819946586;
    fltCoefBuf[ii++] = -0.930427167532233;
    fltCoefBuf[ii++] = 0.137360839237161;
    fltCoefBuf[ii++] = 0.447230387221663;
    fltCoefBuf[ii++] = 0.447230387221663;
    fltCoefBuf[ii++] = 0.137360839237161;
    fltCoefBuf[ii++] = -0.30427167532233;
    fltCoefBuf[ii++] = -0.962045819946586;
    fltCoefBuf[ii++] = 0.199709259997918;
    fltCoefBuf[ii++] = 0.0095364856578114;
    fltCoefBuf[ii++] = -0.995837825348525;
    fltCoefBuf[ii++] = -0.997590551617365;

    Param.u32Order = FIR_ORDER;
    Param.pCoefBuf = &fltCoefBuf[0];

    // Initialize FIR filter
    GDFLIB_FilterFIRInit_FLT(&Param, &State0, &fltInBuf[0]);

    // Initialize FIR filter
    GDFLIB_FilterFIRInit(&Param, &State1, &fltInBuf[0], FLT);

    // ######
    // Available only if single precision floating point
    // implementation selected as default
    // #####
    //

    // Initialize FIR filter
    GDFLIB_FilterFIRInit(&Param, &State2, &fltInBuf[0]);
}
```

```

// Compute step response of the filter
for(ii=0; ii < OUT_LEN; ii++)
{
    // fltOutBuf0 contains step response of the filter
    fltOutBuf0[ii] = GDFLIB\_FilterFIR\_FLT((tFloat)(1), &Param, &State0);

    // fltOutBuf1 contains step response of the filter
    fltOutBuf1[ii] = GDFLIB_FilterFIR((tFloat)(1), &Param, &State1, FLT);

    // ##### Available only if single precision floating point
    // implementation selected as default
    // #####
    // fltOutBuf2 contains step response of the filter
    fltOutBuf2[ii] = GDFLIB_FilterFIR((tFloat)(1), &Param, &State2);
}
// After the loop the fltOutBuf0, fltOutBuf1, fltOutBuf2 shall contains
// the following values:
// {-0.99759054, -1.9934283, -1.9838918, -1.963921, -2.9259667,
// -3.8563938, -3.719033, -3.2718027, -2.8245723, -2.6872115,
// -2.9914832, -3.9535291, -3.7538199, -3.7442834, -4.7401214,
// -5.7377119}
}

```

## 2.10 Function GDFLIB\_FilterIIR1

This function implements the first order IIR filter.

### Description

This function calculates the first order infinite impulse (IIR) filter. The IIR filters are also called recursive filters because both the input and the previously calculated output values are used for calculation of the filter equation in each step. This form of feedback enables transfer of the energy from the output to the input, which theoretically leads to an infinitely long impulse response (IIR).

A general form of the IIR filter expressed as a transfer function in the Z-domain is described as follows:

$$H(z) = \frac{Y(z)}{X(z)} = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2} + \dots + b_N z^{-N}}{1 + a_1 z^{-1} + a_2 z^{-2} + \dots + a_N z^{-N}}$$

Equation GDFLIB\_FilterIIR1\_Eq1

where N denotes the filter order. The first order IIR filter in the Z-domain is therefore given from equation [GDFLIB\\_FilterIIR1\\_Eq1](#) as:

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

Equation GDFLIB\_FilterIIR1\_Eq2

In order to implement the first order IIR filter on a microcontroller, the discrete time domain representation of the filter, described by equation [GDFLIB\\_FilterIIR1\\_Eq2](#), must be transformed into a time difference equation as follows:

$$y(k) = b_0x(k) + b_1x(k-1) - a_1x(k-1)$$

Equation GDFLIB\_FilterIIR1\_Eq3

Equation [GDFLIB\\_FilterIIR1\\_Eq3](#) represents a Direct Form I implementation of a first order IIR filter. It is well known that Direct Form I (DF-I) and Direct Form II (DF-II) implementations of an IIR filter are generally sensitive to parameter quantization if a finite precision arithmetic is considered. This, however, can be neglected when the filter transfer function is broken down into low order sections, i.e. first or second order. The main difference between DF-I and DF-II implementations of an IIR filter is in the number of delay buffers and in the number of guard bits required to handle the potential overflow (in fixed-point variants of the function). The DF-II implementation requires less delay buffers than DF-I, hence less data memory is utilized. On the other hand, since the poles come first in the DF-II realization, the signal entering the state delay-line typically requires a larger dynamic range than the output signal  $y(k)$ . Therefore, overflow can occur at the delay-line input of the DF-II implementation, unlike in the DF-I implementation (considering a fixed-point implementation).

Because there are two delay buffers necessary for both DF-I and DF-II implementations of the first order IIR filter, the DF-I implementation was chosen to be used in the [GDFLIB\\_FilterIIR1](#) function.

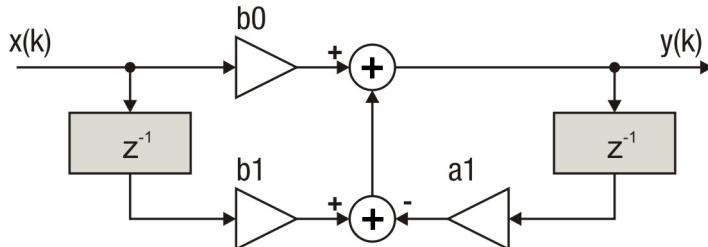


Figure 72. Direct Form 1 first order IIR filter

The coefficients of the filter depicted in [Figure 72](#) can be designed to meet the requirements for the first order Low (LPF) or High Pass (HPF) filters. Filter coefficients can be calculated using various tools, for example, the Matlab® *butter* function (see examples below).

**Note:** The input/output pointers must contain valid addresses, otherwise a fault may occur (*MemManage*, *BusFault*, *UsageFault*, *HardFault*).

## Re-entrancy

The function is re-entrant.

### 2.10.1 Function GDFLIB\_FilterIIR1\_F32

#### Declaration

```
tFrac32 GDFLIB_FilterIIR1_F32(tFrac32 f32In,
GDFLIB_FILTER_IIR1_T_F32 *const pParam);
```

**Arguments****Table 88. GDFLIB\_FilterIIR1\_F32 arguments**

Type	Name	Direction	Description
tFrac32	f32In	input	Value of input signal to be filtered in step (k). The value is a 32-bit number in the 1.31 fractional format.
GDFLIB_FILTER_IIR1_T_F32 *const	pParam	input, output	Pointer to the filter structure with a filter buffer and filter parameters.

**Return**

The function returns a 32-bit value in fractional format 1.31, representing the filtered value of the input signal in step (k).

**Implementation details**

In order to avoid overflow during the calculation of the GDFLIB\_FilterIIR1\_F32 function, filter coefficients must be divided by eight. The coefficients can be calculated using Matlab® as follows:

```

freq_cut = 100;
T_sampling = 100e-6;

[b,a]=butter(1,[freq_cut*T_sampling*2],'low');
sys=tf(b,a,T_sampling);
bode(sys)

f32B0 = b(1);
f32B1 = b(2);
f32A1 = a(2);
disp('Coefficients for GDFLIB_FilterIIR1 function:');
disp(['f32B0 = FRAC32(' num2str(f32B0,'%1.15f') '/8);']);
disp(['f32B1 = FRAC32(' num2str(f32B1,'%1.15f') '/8);']);
disp(['f32A1 = FRAC32(' num2str(f32A1,'%1.15f') '/8);']);

```

**Note:** The filter delay line includes two delay buffers which should be reset after filter initialization. This can be done by assigning to the filter instance a [GDFLIB\\_FILTER\\_IIR1\\_DEFAULT\\_F32](#) macro during instance declaration or by calling the [GDFLIB\\_FilterIIR1Init\\_F32](#) function.

**Caution:** Because of fixed point implementation, and to avoid overflow during the calculation of the GDFLIB\_FilterIIR1\_F32 function, filter coefficients must be divided by eight. Function output is internally multiplied by eight to correct the coefficient scaling.

**Code Example**

```

#include "gdflib.h"

tFrac32 f32In;
tFrac32 f32Out;

GDFLIB_FILTER_IIR1_T_F32 f32trMyIIR1 = GDFLIB_FILTER_IIR1_DEFAULT_F32;

void main(void)
{

```

```

// input value = 0.25
f32In = FRAC32(0.25);

// filter coefficients (LPF 100Hz, Ts=100e-6)
f32trMyIIR1.trFiltCoeff.f32B0 = FRAC32(0.030468747091254/8);
f32trMyIIR1.trFiltCoeff.f32B1 = FRAC32(0.030468747091254/8);
f32trMyIIR1.trFiltCoeff.f32A1 = FRAC32(-0.939062505817492/8);

// output should be 0x00F99998 ~ FRAC32(0.007617)
GDFLIB_FilterIIR1Init_F32(&f32trMyIIR1);
f32Out = GDFLIB_FilterIIR1_F32(f32In, &f32trMyIIR1);

// output should be 0x00F99998 ~ FRAC32(0.007617)
GDFLIB_FilterIIR1Init(&f32trMyIIR1, F32);
f32Out = GDFLIB_FilterIIR1(f32In, &f32trMyIIR1, F32);

// ##### Available only if 32-bit fractional implementation selected
// as default
// #####
// output should be 0x00F99998 ~ FRAC32(0.007617)
GDFLIB_FilterIIR1Init(&f32trMyIIR1);
f32Out = GDFLIB_FilterIIR1(f32In, &f32trMyIIR1);
}

```

## 2.10.2 Function GDFLIB\_FilterIIR1\_F16

### Declaration

```
tFrac16 GDFLIB_FilterIIR1_F16(tFrac16 f16In,
GDFLIB_FILTER_IIR1_T_F16 *const pParam);
```

### Arguments

**Table 89. GDFLIB\_FilterIIR1\_F16 arguments**

Type	Name	Direction	Description
tFrac16	f16In	input	Value of input signal to be filtered in step (k). The value is a 16-bit number in the 1.15 fractional format.
GDFLIB_FILTER_IIR1_T_F16 *const	pParam	input, output	Pointer to the filter structure with a filter buffer and filter parameters.

### Return

The function returns a 16-bit value in fractional format 1.15, representing the filtered value of the input signal in step (k).

### Implementation details

In order to avoid overflow during the calculation of the [GDFLIB\\_FilterIIR1\\_F32](#) function, filter coefficients must be divided by eight. The coefficients can be calculated using Matlab® as follows:

```

freq_cut      = 100;
T_sampling   = 100e-6;
```

```
[b,a]=butter(1,[freq_cut*T_sampling*2],'low');
sys=tf(b,a,T_sampling);
bode(sys)

f16B0 = b(1);
f16B1 = b(2);
f16A1 = a(2);
disp('Coefficients for GDFLIB_FilterIIR1 function:');
disp(['f16B0 = FRAC16(' num2str(f16B0,'%1.15f') '/8);']);
disp(['f16B1 = FRAC16(' num2str(f16B1,'%1.15f') '/8);']);
disp(['f16A1 = FRAC16(' num2str(f16A1,'%1.15f') '/8);']);
```

**Note:** The filter delay line includes two delay buffers which should be reset after filter initialization. This can be done by assigning to the filter instance a GDFLIB\_FILTER\_IIR1\_DEFAULT\_F16 macro during instance declaration or by calling the GDFLIB\_FilterIIR1Init\_F16 function.

**Caution:** Because of fixed point implementation, and to avoid overflow during the calculation of the GDFLIB\_FilterIIR1\_F16 function, filter coefficients must be divided by eight. Function output is internally multiplied by eight to correct the coefficient scaling.

### Code Example

```
#include "gdflib.h"

tFrac16 f16In;
tFrac16 f16Out;

GDFLIB_FILTER_IIR1_T_F16 f16trMyIIR1 = GDFLIB_FILTER_IIR1_DEFAULT_F16;

void main(void)
{
    // input value = 0.25
    f16In = FRAC16(0.25);

    // filter coefficients (LPF 100Hz, Ts=100e-6)
    f16trMyIIR1.trFiltCoeff.f16B0 = FRAC16(0.030468747091254/8);
    f16trMyIIR1.trFiltCoeff.f16B1 = FRAC16(0.030468747091254/8);
    f16trMyIIR1.trFiltCoeff.f16A1 = FRAC16(-0.939062505817492/8);

    // output should be 0x00F9 ~ FRAC16(0.007617)
    GDFLIB_FilterIIR1Init_F16(&f16trMyIIR1);
    f16Out = GDFLIB_FilterIIR1_F16(f16In, &f16trMyIIR1);

    // output should be 0x00F9 ~ FRAC16(0.007617)
    GDFLIB_FilterIIR1Init(&f16trMyIIR1, F16);
    f16Out = GDFLIB_FilterIIR1(f16In, &f16trMyIIR1, F16);

    // #####
    // Available only if 16-bit fractional implementation selected
    // as default
    // #####
    // #####
    // output should be 0x00F9 ~ FRAC16(0.007617)
    GDFLIB_FilterIIR1Init(&f16trMyIIR1);
    f16Out = GDFLIB_FilterIIR1(f16In, &f16trMyIIR1);
}
```

### 2.10.3 Function GDFLIB\_FilterIIR1\_FLT

#### Declaration

```
tFloat GDFLIB_FilterIIR1_FLT(tFloat fltIn,
GDFLIB_FILTER_IIR1_T_FLT *const pParam);
```

#### Arguments

Table 90. GDFLIB\_FilterIIR1\_FLT arguments

Type	Name	Direction	Description
tFloat	fltIn	input	Value of the input signal to be filtered in step (k). Input is a 32-bit number that contains a single precision floating point value.
GDFLIB_FILTER_IIR1_T_FLT *const	pParam	input, output	Pointer to a filter structure with a filter buffer and filter parameters. Arguments of the structure contain single precision floating point values.

#### Return

The function returns a 32-bit value in single precision floating point format, representing the filtered value of the input signal in step (k).

#### Implementation details

The coefficients can be calculated using Matlab® as follows:

```
freq_cut = 100;
T_sampling = 100e-6;

[b,a]=butter(1,[freq_cut*T_sampling*2],'low');
sys=tf(b,a,T_sampling);
bode(sys)

fltB0 = b(1);
fltB1 = b(2);
fltA1 = a(2);
disp('Coefficients for GDFLIB_FilterIIR1 function:');
disp(['fltB0 = (tFloat)(' num2str(fltB0,'%1.15f') ')']);
disp(['fltB1 = (tFloat)(' num2str(fltB1,'%1.15f') ')']);
disp(['fltA1 = (tFloat)(' num2str(fltA1,'%1.15f') ')']);
```

**Note:** The function may raise floating-point exceptions (invalid operation, overflow, underflow, inexact, input denormal). The floating-point unit must be enabled in CPACR register to prevent NOCP UsageFault exception.

**Caution:** The filter delay line includes two delay buffers which should be reset after filter initialization. This can be done by assigning to the filter instance a `GDFLIB_FILTER_IIR1_DEFAULT_FLT` macro during instance declaration or by calling the `GDFLIB_FilterIIR1Init_FLT` function.

#### Code Example

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

```
tFloat fltIn;
tFloat fltOut;

GDFLIB_FILTER_IIR1_T_FLT flttrMyIIR1 = GDFLIB_FILTER_IIR1_DEFAULT_FLT;

void main(void)
{
    // input value = 0.25
    fltIn = (tFloat) 0.25;

    // filter coefficients (LPF 100Hz)
    flttrMyIIR1.trFiltCoeff.fltB0 = (tFloat) (0.030468747091254);
    flttrMyIIR1.trFiltCoeff.fltB1 = (tFloat) (0.030468747091254);
    flttrMyIIR1.trFiltCoeff.fltA1 = (tFloat) (-0.939062505817492);

    // output should be 0.007617
    GDFLIB_FilterIIR1Init_FLT(&flttrMyIIR1);
    fltOut = GDFLIB_FilterIIR1_FLT(fltIn, &flttrMyIIR1);

    // output should be 0.007617
    GDFLIB_FilterIIR1Init(&flttrMyIIR1, FLT);
    fltOut = GDFLIB_FilterIIR1(fltIn, &flttrMyIIR1);

    // ##### Available only if single precision floating point
    // implementation selected as default
    // #####
    // output should be 0.007617
    GDFLIB_FilterIIR1Init(&flttrMyIIR1);
    fltOut = GDFLIB_FilterIIR1(fltIn, &flttrMyIIR1);
}
```

## 2.10.4 Function GDFLIB\_FilterIIR1Init

This function initializes the first order IIR filter buffers.

### Description

This function clears the internal buffers of a first order IIR filter. It shall be called after filter parameter initialization and whenever the filter initialization is required.

**Note:** The input/output pointers must contain valid addresses, otherwise a fault may occur (*MemManage*, *BusFault*, *UsageFault*, *HardFault*).

**Note:** This function shall not be called together with the *GDFLIB\_FilterIIR1* function unless periodic clearing of filter buffers is required.

### Re-entrancy

The function is re-entrant.

### 2.10.4.1 Function GDFLIB\_FilterIIR1Init\_F32

#### Declaration

```
void GDFLIB_FilterIIR1Init_F32(GDFLIB_FILTER_IIR1_T_F32 *const pParam);
```

## Arguments

**Table 91.** GDFLIB\_FilterIIR1Init\_F32 arguments

Type	Name	Direction	Description
GDFLIB_FILTER_IIR1_T_F32 *const	pParam	input, output	Pointer to filter structure with filter buffer and filter parameters.

## Code Example

```
#include "gdflib.h"

GDFLIB_FILTER_IIR1_T_F32 f32trMyIIR1 = GDFLIB_FILTER_IIR1_DEFAULT_F32;

void main(void)
{
    // function returns no value
    GDFLIB_FilterIIR1Init_F32(&f32trMyIIR1);

    // function returns no value
    GDFLIB_FilterIIR1Init(&f32trMyIIR1, F32);

    // ##### Available only if 32-bit fractional implementation selected
    // as default
    // #####
}

// function returns no value
GDFLIB_FilterIIR1Init(&f32trMyIIR1);
}
```

**2.10.4.2 Function GDFLIB\_FilterIIR1Init\_F16**

## Declaration

```
void GDFLIB_FilterIIR1Init_F16(GDFLIB_FILTER_IIR1_T_F16 *const pParam);
```

## Arguments

**Table 92.** GDFLIB\_FilterIIR1Init\_F16 arguments

Type	Name	Direction	Description
GDFLIB_FILTER_IIR1_T_F16 *const	pParam	input, output	Pointer to filter structure with filter buffer and filter parameters.

## Code Example

```
#include "gdflib.h"

GDFLIB_FILTER_IIR1_T_F16 f16trMyIIR1 = GDFLIB_FILTER_IIR1_DEFAULT_F16;

void main(void)
{
    // function returns no value
    GDFLIB_FilterIIR1Init_F16(&f16trMyIIR1);
```

```

// function returns no value
GDFLIB_FilterIIR1Init(&f16trMyIIR1, F16);

// ######
// Available only if 16-bit fractional implementation selected
// as default
// #####
// function returns no value
GDFLIB_FilterIIR1Init(&f16trMyIIR1);
}

```

#### 2.10.4.3 Function GDFLIB\_FilterIIR1Init\_FLT

##### Declaration

```
void GDFLIB_FilterIIR1Init_FLT(GDFLIB\_FILTER\_IIR1\_T\_FLT *const pParam);
```

##### Arguments

**Table 93. GDFLIB\_FilterIIR1Init\_FLT arguments**

Type	Name	Direction	Description
<a href="#">GDFLIB_FILTER_IIR1_T_FLT</a> *const	pParam	input, output	Pointer to a filter structure with filter buffer and filter parameters. Arguments of the structure contain single precision floating point values.

##### Code Example

```

#include "gdflib.h"

GDFLIB\_FILTER\_IIR1\_T\_FLT flttrMyIIR1 = GDFLIB\_FILTER\_IIR1\_DEFAULT\_FLT;

void main(void)
{
    // function returns no value
    GDFLIB_FilterIIR1Init_FLT(&flttrMyIIR1);

    // function returns no value
    GDFLIB_FilterIIR1Init(&flttrMyIIR1, FLT);

    // ######
    // Available only if single precision floating point
    // implementation selected as default
    // #####
}

// function returns no value
GDFLIB_FilterIIR1Init(&flttrMyIIR1);
}

```

## 2.11 Function GDFLIB\_FilterIIR2

This function implements the second order IIR filter.

## Description

This function calculates the second order infinite impulse (IIR) filter. The IIR filters are also called recursive filters because both the input and the previously calculated output values are used for calculation of the filter equation in each step. This form of feedback enables transfer of the energy from the output to the input, which theoretically leads to an infinitely long impulse response (IIR).

A general form of the IIR filter expressed as a transfer function in the Z-domain is described as follows:

$$H(z) = \frac{Y(z)}{X(z)} = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2} + \dots + b_N z^{-N}}{1 + a_1 z^{-1} + a_2 z^{-2} + \dots + a_N z^{-N}}$$

Equation GDFLIB\_FilterIIR2\_Eq1

where N denotes the filter order. The second order IIR filter in the Z-domain is therefore given from eq. [GDFLIB\\_FilterIIR2\\_Eq1](#) as:

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

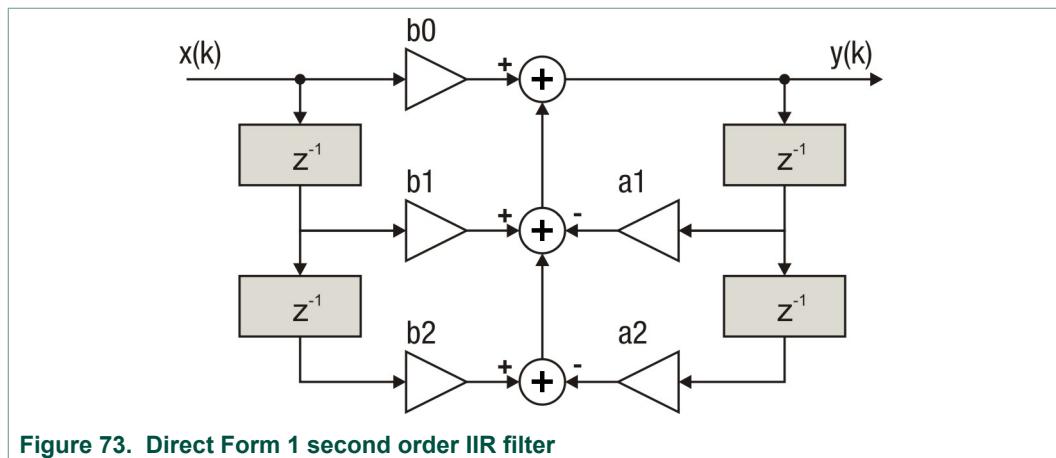
Equation GDFLIB\_FilterIIR2\_Eq2

In order to implement the second order IIR filter on a microcontroller, the discrete time domain representation of the filter, described by eq. [GDFLIB\\_FilterIIR2\\_Eq2](#), must be transformed into a time difference equation as follows:

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

Equation GDFLIB\_FilterIIR2\_Eq3

Equation [GDFLIB\\_FilterIIR2\\_Eq3](#) represents a Direct Form I implementation of a second order IIR filter. It is well known that Direct Form I (DF-I) and Direct Form II (DF-II) implementations of an IIR filter are generally sensitive to parameter quantization if a finite precision arithmetic is considered. This, however, can be neglected when the filter transfer function is broken down into low order sections, i.e. first or second order. The main difference between DF-I and DF-II implementations of an IIR filter is in the number of delay buffers and in the number of guard bits required to handle the potential overflow (in fixed-point variants of the function). The DF-II implementation requires fewer delay buffers than DF-I, hence less data memory is utilized. On the other hand, since the poles come first in the DF-II realization, the signal entering the state delay-line typically requires a larger dynamic range than the output signal y(k). Therefore, overflow can occur at the delay-line input of the DF-II implementation, unlike in the DF-I implementation (considering a fixed-point implementation).



The coefficients of the filter depicted in [Figure 73](#) can be designed to meet the requirements for the second order Band Pass (BPF) or Band Stop (BSF) filters. Filter coefficients can be calculated using various tools, for example, the Matlab® *butter* function (see examples below).

**Note:** The input/output pointers must contain valid addresses, otherwise a fault may occur (*MemManage*, *BusFault*, *UsageFault*, *HardFault*).

## Re-entrancy

The function is re-entrant.

### 2.11.1 Function GDFLIB\_FilterIIR2\_F32

#### Declaration

```
tFrac32 GDFLIB_FilterIIR2_F32(tFrac32 f32In,
GDFLIB_FILTER_IIR2_T_F32 *const pParam);
```

#### Arguments

**Table 94. GDFLIB\_FilterIIR2\_F32 arguments**

Type	Name	Direction	Description
tFrac32	f32In	input	Value of input signal to be filtered in step (k). The value is a 32-bit number in the 1.31 fractional format.
GDFLIB_FILTER_IIR2_T_F32 *const	pParam	input, output	Pointer to the filter structure with a filter buffer and filter parameters.

#### Return

The function returns a 32-bit value in fractional format 1.31, representing the filtered value of the input signal in step (k).

#### Implementation details

In order to avoid overflow during the calculation of the GDFLIB\_FilterIIR2\_F32 function, filter coefficients must be divided by eight. The coefficients can be calculated using Matlab® as follows:

```

freq_bot      = 400;
freq_top      = 625;
T_sampling    = 100e-6;

[b,a] = butter(1,[freq_bot freq_top]*T_sampling *2, 'bandpass');
sys = tf(b,a,T_sampling);
bode(sys,[freq_bot:1:freq_top]*2*pi)

f32B0 = b(1);
f32B1 = b(2);
f32B2 = b(3);
f32A1 = a(2);
f32A2 = a(3);
disp (' Coefficients for GDFLIB_FilterIIR2 function :');
disp ([ 'f32B0 = FRAC32(' num2str( f32B0,'%1.15f' ) '/8);']);
disp ([ 'f32B1 = FRAC32(' num2str( f32B1,'%1.15f' ) '/8);']);
disp ([ 'f32B2 = FRAC32(' num2str( f32B2,'%1.15f' ) '/8);']);
disp ([ 'f32A1 = FRAC32(' num2str( f32A1,'%1.15f' ) '/8);']);
disp ([ 'f32A2 = FRAC32(' num2str( f32A2,'%1.15f' ) '/8);']);

```

**Note:** The filter delay line includes four delay buffers which should be reset after filter initialization. This can be done by assigning to the filter instance a [GDFLIB\\_FILTER\\_IIR2\\_DEFAULT\\_F32](#) macro during instance declaration or by calling the [GDFLIB\\_FilterIIR2Init\\_F32](#) function.

**Caution:** Because of fixed point implementation, and to avoid overflow during the calculation of the [GDFLIB\\_FilterIIR2\\_F32](#) function, filter coefficients must be divided by eight. Function output is internally multiplied by eight to correct the coefficient scaling.

### Code Example

```

#include "gdflib.h"

tFrac32 f32In;
tFrac32 f32Out;

GDFLIB\_FILTER\_IIR2\_T\_F32 f32trMyIIR2 = GDFLIB\_FILTER\_IIR2\_DEFAULT\_F32;

void main(void)
{
    // input value = 0.25
    f32In = FRAC32(0.25);

    // filter coefficients (BPF 400-625Hz, Ts=100e-6)
    f32trMyIIR2.trFiltCoeff.f32B0 = FRAC32(0.066122101544579/8);
    f32trMyIIR2.trFiltCoeff.f32B1 = FRAC32(0/8);
    f32trMyIIR2.trFiltCoeff.f32B2 = FRAC32(-0.066122101544579/8);
    f32trMyIIR2.trFiltCoeff.f32A1 = FRAC32(-1.776189018043779/8);
    f32trMyIIR2.trFiltCoeff.f32A2 = FRAC32(0.867755796910841/8);

    // output should be 0x021DAC18 ~ FRAC32(0.0165305)
    GDFLIB\_FilterIIR2Init\_F32(&f32trMyIIR2);
    f32Out = GDFLIB_FilterIIR2_F32(f32In, &f32trMyIIR2);

    // output should be 0x021DAC18 ~ FRAC32(0.0165305)
    GDFLIB_FilterIIR2Init(&f32trMyIIR2, F32);
    f32Out = GDFLIB_FilterIIR2(f32In, &f32trMyIIR2, F32);
}

```

```

// ######
// Available only if 32-bit fractional implementation selected
// as default
// #####
// output should be 0x021DAC18 ~ FRAC32(0.0165305)
GDFLIB_FilterIIR2Init(&f32trMyIIR2);
f32Out = GDFLIB_FilterIIR2(f32In, &f32trMyIIR2);
}

```

## 2.11.2 Function GDFLIB\_FilterIIR2\_F16

### Declaration

```
tFrac16 GDFLIB_FilterIIR2_F16(tFrac16 f16In,
GDFLIB_FILTER_IIR2_T_F16 *const pParam);
```

### Arguments

Table 95. GDFLIB\_FilterIIR2\_F16 arguments

Type	Name	Direction	Description
<u>tFrac16</u>	f16In	<b>input</b>	Value of input signal to be filtered in step (k). The value is a 16-bit number in the 1.15 fractional format.
<u>GDFLIB_FILTER_IIR2_T_F16</u> *const	pParam	<b>input, output</b>	Pointer to the filter structure with a filter buffer and filter parameters.

### Return

The function returns a 16-bit value in fractional format 1.15, representing the filtered value of the input signal in step (k).

### Implementation details

In order to avoid overflow during the calculation of the GDFLIB\_FilterIIR2\_F16 function, filter coefficients must be divided by eight. The coefficients can be calculated using Matlab® as follows:

```

freq_bot      = 400;
freq_top      = 625;
T_sampling    = 100e-6;

[b,a] = butter(1,[freq_bot freq_top]*T_sampling *2, 'bandpass');
sys = tf(b,a,T_sampling);
bode(sys,[freq_bot:1:freq_top]*2*pi)

f16B0 = b(1);
f16B1 = b(2);
f16B2 = b(3);
f16A1 = a(2);
f16A2 = a(3);
disp (' Coefficients for GDFLIB_FilterIIR2 function :');
disp ([ 'f16B0 = FRAC16(' num2str( f16B0,'%1.15f' ) '/8);']);
disp ([ 'f16B1 = FRAC16(' num2str( f16B1,'%1.15f' ) '/8);']);
disp ([ 'f16B2 = FRAC16(' num2str( f16B2,'%1.15f' ) '/8);']);
disp ([ 'f16A1 = FRAC16(' num2str( f16A1,'%1.15f' ) '/8);']);

```

```
disp ([ 'f16A2 = FRAC16(' num2str( f16A2,'%1.15f' ) '/8);']);
```

**Note:** The filter delay line includes four delay buffers which should be reset after filter initialization. This can be done by assigning to the filter instance a [GDFLIB\\_FILTER\\_IIR2\\_DEFAULT\\_F16](#) macro during instance declaration or by calling the [GDFLIB\\_FilterIIR2Init\\_F16](#) function.

**Caution:** Because of fixed point implementation, and to avoid overflow during the calculation of the [GDFLIB\\_FilterIIR2\\_F16](#) function, filter coefficients must be divided by eight. Function output is internally multiplied by eight to correct the coefficient scaling.

### Code Example

```
#include "gdflib.h"

tFrac16 f16In;
tFrac16 f16Out;

GDFLIB_FILTER_IIR2_T_F16 f16trMyIIR2 = GDFLIB_FILTER_IIR2_DEFAULT_F16;

void main(void)
{
    // input value = 0.25
    f16In = FRAC16(0.25);

    // filter coefficients (BPF 400-625Hz, Ts=100e-6)
    f16trMyIIR2.trFiltCoeff.f16B0 = FRAC16(0.066122101544579/8);
    f16trMyIIR2.trFiltCoeff.f16B1 = FRAC16(0/8);
    f16trMyIIR2.trFiltCoeff.f16B2 = FRAC16(-0.066122101544579/8);
    f16trMyIIR2.trFiltCoeff.f16A1 = FRAC16(-1.776189018043779/8);
    f16trMyIIR2.trFiltCoeff.f16A2 = FRAC16(0.867755796910841/8);

    // output should be 0x021D ~ FRAC16(0.01651)
    GDFLIB_FilterIIR2Init_F16(&f16trMyIIR2);
    f16Out = GDFLIB_FilterIIR2_F16(f16In, &f16trMyIIR2);

    // output should be 0x021D ~ FRAC16(0.01651)
    GDFLIB_FilterIIR2Init(&f16trMyIIR2, F16);
    f16Out = GDFLIB_FilterIIR2(f16In, &f16trMyIIR2, F16);

    // #####
    // Available only if 16-bit fractional implementation selected
    // as default
    // #####
    // #####
    // output should be 0x021D ~ FRAC16(0.01651)
    GDFLIB_FilterIIR2Init(&f16trMyIIR2);
    f16Out = GDFLIB_FilterIIR2(f16In, &f16trMyIIR2);
}
```

### 2.11.3 Function GDFLIB\_FilterIIR2\_FLT

#### Declaration

```
tFloat GDFLIB_FilterIIR2_FLT(tFloat fltIn,
    GDFLIB_FILTER_IIR2_T_FLT *const pParam);
```

**Arguments****Table 96. GDFLIB\_FilterIIR2\_FLT arguments**

Type	Name	Direction	Description
tFloat	fltIn	input	Value of the input signal to be filtered in step (k). Input is a 32-bit number that contains a single precision floating point value.
GDFLIB_FILTER_IIR2_T_FLT *const	pParam	input, output	Pointer to a filter structure with a filter buffer and filter parameters. Arguments of the structure contain single precision floating point values.

**Return**

The function returns a 32-bit value in single precision floating point format, representing the filtered value of the input signal in step (k).

**Implementation details**

The coefficients can be calculated using Matlab® as follows:

```

freq_bot      = 400;
freq_top      = 625;
T_sampling    = 100e-6;

[b,a]= butter(1,[freq_bot freq_top]*T_sampling *2, 'bandpass');
sys =tf(b,a,T_sampling);
bode(sys,[freq_bot:1:freq_top]*2*pi)

fltB0 = b(1);
fltB1 = b(2);
fltB2 = b(3);
fltA1 = a(2);
fltA2 = a(3);
disp (' Coefficients for GDFLIB_FilterIIR2 function :');
disp ([ 'fltB0 = (tFloat) (' num2str( fltB0,'%1.15f' ) ')';']);
disp ([ 'fltB1 = (tFloat) (' num2str( fltB1,'%1.15f' ) ')';']);
disp ([ 'fltB2 = (tFloat) (' num2str( fltB2,'%1.15f' ) ')';']);
disp ([ 'fltA1 = (tFloat) (' num2str( fltA1,'%1.15f' ) ')';']);
disp ([ 'fltA2 = (tFloat) (' num2str( fltA2,'%1.15f' ) ')';']);

```

**Note:** The function may raise floating-point exceptions (invalid operation, overflow, underflow, inexact, input denormal). The floating-point unit must be enabled in CPACR register to prevent NOCP UsageFault exception.

**Caution:** The filter delay line includes four delay buffers which should be reset after filter initialization. This can be done by assigning to the filter instance a [GDFLIB\\_FILTER\\_IIR2\\_DEFAULT\\_FLT](#) macro during instance declaration or by calling the [GDFLIB\\_FilterIIR2Init\\_FLT](#) function.

**Code Example**

```
#include "gdflib.h"

tFloat fltIn;
tFloat fltOut;
```

```
GDFLIB_FILTER_IIR2_T_FLT flttrMyIIR2 = GDFLIB_FILTER_IIR2_DEFAULT_FLT;

void main(void)
{
    // input value = 0.25
    fltIn = (tFloat)(0.25);

    // filter coefficients (BPF 400-625Hz, Ts=100e-6)
    flttrMyIIR2.trFiltCoeff.fltB0 = (tFloat)(0.066122101544579);
    flttrMyIIR2.trFiltCoeff.fltB1 = (tFloat)(0.0);
    flttrMyIIR2.trFiltCoeff.fltB2 = (tFloat)(-0.066122101544579);
    flttrMyIIR2.trFiltCoeff.fltA1 = (tFloat)(-1.776189018043779);
    flttrMyIIR2.trFiltCoeff.fltA2 = (tFloat)(0.867755796910841);

    // output should be 0.01651
    GDFLIB_FilterIIR2Init_FLT(&flttrMyIIR2);
    fltOut = GDFLIB_FilterIIR2_FLT(fltIn, &flttrMyIIR2);

    // output should be 0.01651
    GDFLIB_FilterIIR2Init(&flttrMyIIR2, FLT);
    fltOut = GDFLIB_FilterIIR2(fltIn, &flttrMyIIR2, FLT);

    // ##### Available only if single precision floating point
    // implementation selected as default
    // #####
    // output should be 0.01651
    GDFLIB_FilterIIR2Init(&flttrMyIIR2);
    fltOut = GDFLIB_FilterIIR2(fltIn, &flttrMyIIR2);
}
```

## 2.11.4 Function GDFLIB\_FilterIIR2Init

This function initializes the second order IIR filter buffers.

### Description

This function clears the internal buffers of a second order IIR filter. It shall be called after filter parameter initialization and whenever the filter initialization is required.

**Note:** The input/output pointers must contain valid addresses, otherwise a fault may occur (MemManage, BusFault, UsageFault, HardFault).

**Note:** This function shall not be called together with the GDFLIB\_FilterIIR2 function unless periodic clearing of filter buffers is required.

### Re-entrancy

The function is re-entrant.

### 2.11.4.1 Function GDFLIB\_FilterIIR2Init\_F32

#### Declaration

```
void GDFLIB_FilterIIR2Init_F32(GDFLIB_FILTER_IIR2_T_F32 *const pParam);
```

## Arguments

**Table 97. GDFLIB\_FilterIIR2Init\_F32 arguments**

Type	Name	Direction	Description
GDFLIB_FILTER_IIR2_T_F32 *const	pParam	input, output	Pointer to the filter structure with a filter buffer and filter parameters.

## Code Example

```
#include "gdflib.h"

GDFLIB_FILTER_IIR2_T_F32 f32trMyIIR2 = GDFLIB_FILTER_IIR2_DEFAULT_F32;

void main(void)
{
    // function returns no value
    GDFLIB_FilterIIR2Init_F32(&f32trMyIIR2);

    // function returns no value
    GDFLIB_FilterIIR2Init(&f32trMyIIR2, F32);

    // ##### Available only if 32-bit fractional implementation selected
    // as default
    // #####
    // function returns no value
    GDFLIB_FilterIIR2Init(&f32trMyIIR2);

}
```

**2.11.4.2 Function GDFLIB\_FilterIIR2Init\_F16**

## Declaration

```
void GDFLIB_FilterIIR2Init_F16(GDFLIB_FILTER_IIR2_T_F16 *const pParam);
```

## Arguments

**Table 98. GDFLIB\_FilterIIR2Init\_F16 arguments**

Type	Name	Direction	Description
GDFLIB_FILTER_IIR2_T_F16 *const	pParam	input, output	Pointer to the filter structure with a filter buffer and filter parameters.

## Code Example

```
#include "gdflib.h"

GDFLIB_FILTER_IIR2_T_F16 f16trMyIIR2 = GDFLIB_FILTER_IIR2_DEFAULT_F16;

void main(void)
{
    // function returns no value
    GDFLIB_FilterIIR2Init_F16(&f16trMyIIR2);
```

```

// function returns no value
GDFLIB_FilterIIR2Init(&f16trMyIIR2, F16);

// ##### Available only if 16-bit fractional implementation selected
// as default
// #####
// function returns no value
GDFLIB_FilterIIR2Init(&f16trMyIIR2);
}

```

#### 2.11.4.3 Function GDFLIB\_FilterIIR2Init\_FLT

##### Declaration

```
void GDFLIB_FilterIIR2Init_FLT(GDFLIB\_FILTER\_IIR2\_T\_FLT *const pParam);
```

##### Arguments

**Table 99. GDFLIB\_FilterIIR2Init\_FLT arguments**

Type	Name	Direction	Description
<a href="#">GDFLIB_FILTER_IIR2_T_FLT</a> *const	pParam	input, output	Pointer to a filter structure with filter buffer and filter parameters. Arguments of the structure contain single precision floating point values.

**Note:** The floating-point unit must be enabled in CPACR register to prevent NOCP UsageFault exception.

##### Code Example

```

#include "gdflib.h"

GDFLIB\_FILTER\_IIR2\_T\_FLT flttrMyIIR2 = GDFLIB\_FILTER\_IIR2\_DEFAULT\_FLT;

void main(void)
{
    // function returns no value
    GDFLIB_FilterIIR2Init_FLT(&flttrMyIIR2);

    // function returns no value
    GDFLIB_FilterIIR2Init(&flttrMyIIR2, FLT);

    // ##### Available only if single precision floating point
    // implementation selected as default
    // #####
    // function returns no value
    GDFLIB_FilterIIR2Init(&flttrMyIIR2);

}

```

## 2.12 Function GDFLIB\_FilterMA

This function implements an exponential moving average filter.

### Description

This function calculates one iteration of an exponential moving average filter (also known as the exponentially weighted moving average, EWMA). The filter is characterized by the following difference equation:

$$y(k) = \lambda \cdot x(k) + (1 - \lambda) \cdot y(k-1)$$

Equation GDFLIB\_FilterMA\_Eq1

where  $x(k)$  is the filter input,  $y(k)$  is the filter output in the current step,  $y(k-1)$  is the filter output of the previous step and  $\lambda$  is a smoothing factor,  $0 < \lambda < 1$ . Values of  $\lambda$  close to one lead to less smoothing and give greater weight to recent changes in the input data, while values of  $\lambda$  closer to zero cause greater smoothing and the filter is less responsive to recent changes.

There is no direct equivalence between the smoothing factor  $\lambda$  of the exponential moving average filter and the number of averaged samples  $N$  of a uniform sliding-window moving average filter. Nevertheless, the implementation uses the following approximation to relate the two filtering approaches:

$$\lambda = \frac{1}{N}$$

Equation GDFLIB\_FilterMA\_Eq2

When  $\lambda$  is set according to the above formula, the amplitude signal-to-noise ratio improvement achievable by both types of filters is the same.

**Note:** The input/output pointers must contain valid addresses, otherwise a fault may occur (MemManage, BusFault, UsageFault, HardFault).

### Re-entrancy

The function is re-entrant.

### 2.12.1 Function GDFLIB\_FilterMA\_F32

#### Declaration

```
tFrac32 GDFLIB_FilterMA_F32(tFrac32 f32In, GDFLIB_FILTER_MA_T_F32 *pParam);
```

#### Arguments

Table 100. GDFLIB\_FilterMA\_F32 arguments

Type	Name	Direction	Description
tFrac32	f32In	input	Value of input signal to be filtered in step (k). The value is a 32-bit number in the Q1.31 format.
GDFLIB_FILTER_MA_T_F32 *	pParam	input, output	Pointer to the filter structure with a filter accumulator and a smoothing factor.

## Return

The function returns a 32-bit value in format Q1.31, representing the filtered value of the input signal in step (k).

## Implementation details

The library function expects the smoothing factor to be supplied in the form of the u16NSamples variable stored within the filter structure. This variable represents the binary logarithm of the number of averaged samples N of the corresponding uniform sliding-window moving average filter:

$$N = 2^{u16NSamples} \quad 0 \leq u16NSamples \leq 31$$

Equation GDFLIB\_FilterMA\_F32\_Eq1

**Note:** The recalculated smoothing factor u16NSamples needs to be defined prior to calling this function and must be equal to or greater than 0, and equal to or smaller than 31. Incorrect setting of this parameter will yield meaningless results.

## Code Example

```
#include "gdflib.h"

tFrac32 f32Input;
tFrac32 f32Output;

GDFLIB_FILTER_MA_T_F32 f32trMyMA = GDFLIB_FILTER_MA_DEFAULT_F32;

void main(void)
{
    // input value = 0.25
    f32Input = FRAC32(0.25);

    // filter window = 2^5 = 32 samples
    f32trMyMA.u16NSamples = 5;
    GDFLIB_FilterMAInit_F32(&f32trMyMA);

    // output should be 0x1000000 = FRAC32(0.0078125)
    f32Output = GDFLIB_FilterMA_F32(f32Input,&f32trMyMA);

    // output should be 0x1000000 = FRAC32(0.0078125)
    f32Output = GDFLIB_FilterMA(f32Input,&f32trMyMA,F32);

    // ##### Available only if 32-bit fractional implementation selected
    // as default
    // #####
    // output should be 0x1000000 = FRAC32(0.0078125)
    f32Output = GDFLIB_FilterMA(f32Input,&f32trMyMA);
}
```

## 2.12.2 Function GDFLIB\_FilterMA\_F16

### Declaration

```
tFrac16 GDFLIB_FilterMA_F16(tFrac16 f16In, GDFLIB_FILTER_MA_T_F16
    *pParam);
```

### Arguments

Table 101. GDFLIB\_FilterMA\_F16 arguments

Type	Name	Direction	Description
tFrac16	f16In	input	Value of input signal to be filtered in step (k). The value is a 16-bit number in the Q1.15 format.
GDFLIB_FILTER_MA_T_F16 *	pParam	input, output	Pointer to the filter structure with a filter accumulator and a smoothing factor.

### Return

The function returns a 16-bit value in format Q1.15, representing the filtered value of the input signal in step (k).

### Implementation details

The library function expects the smoothing factor to be supplied in the form of the u16NSamples variable stored within the filter structure. This variable represents the binary logarithm of the number of averaged samples N of the corresponding uniform sliding-window moving average filter:

$$N = 2^{u16NSamples} \quad 0 \leq u16NSamples \leq 15$$

Equation GDFLIB\_FilterMA\_F16\_Eq1

**Note:** The recalculated smoothing factor u16NSamples needs to be defined prior to calling this function and must be equal to or greater than 0, and equal to or smaller than 16. Incorrect setting of this parameter will yield meaningless results. In case the filter window size is greater than 8, the output error may exceed the guaranteed range.

### Code Example

```
#include "gdflib.h"

tFrac16 f16Input;
tFrac16 f16Output;

GDFLIB_FILTER_MA_T_F16 f16trMyMA = GDFLIB_FILTER_MA_DEFAULT_F16;

void main(void)
{
    // input value = 0.25
    f16Input = FRAC16(0.25);

    // filter window = 2^3 = 8 samples
    f16trMyMA.u16NSamples = 3;

    GDFLIB_FilterMAInit_F16(&f16trMyMA);
```

```

// output should be 0x0400 = FRAC16(0.03125)
f16Output = GDFLIB_FilterMA_F16(f16Input,&f16trMyMA);

// output should be 0x0400 = FRAC16(0.03125)
f16Output = GDFLIB_FilterMA(f16Input,&f16trMyMA,F16);

// ##### Available only if 16-bit fractional implementation selected
// as default
// #####
// output should be 0x0400 = FRAC16(0.03125)
f16Output = GDFLIB_FilterMA(f16Input,&f16trMyMA);
}

```

### 2.12.3 Function GDFLIB\_FilterMA\_FLT

#### Declaration

```
tFloat GDFLIB_FilterMA_FLT(tFloat fltIn, GDFLIB_FILTER_MA_T_FLT
*pParam);
```

#### Arguments

**Table 102. GDFLIB\_FilterMA\_FLT arguments**

Type	Name	Direction	Description
tFloat	fltIn	input	Value of the input signal to be filtered in step (k). The value is a single precision floating point data type.
GDFLIB_FILTER_MA_T_FLT *	pParam	input, output	Pointer to the filter structure with a filter accumulator and a smoothing factor.

#### Return

The function returns a single precision floating point value, representing the filtered value of the input signal in step (k).

#### Implementation details

Setting the smoothing factor according to the following formula will yield results equivalent to the fixed-point variants of the GDFLIB\_FilterMA function:

$$\lambda = \frac{1}{N}$$

Equation GDFLIB\_FilterMA\_FLT\_Eq1

**Note:** The function may raise floating-point exceptions (invalid operation, overflow, underflow, inexact, input denormal). The floating-point unit must be enabled in CPACR register to prevent NOCP UsageFault exception.

The smoothing factor  $\lambda$  need to be entered in the input parameter `fltLambda` prior to calling this function. If  $\lambda < 0$  or  $\lambda > 1$ , the output values are meaningless numbers. If any of the inputs is a subnormal value, infinity, or a NaN, the filter output for the current iteration and all future iterations is meaningless until the filter is re-initialized.

### Code Example

EXAMPLE 1: Using a smoothing factor calculation equivalent to the fixed-point variants of the GDFLIB\_FilterMA function:

```
#include "gdflib.h"

tFloat fltInput;
tFloat fltOutput;

GDFLIB_FILTER_MA_T_FLT flttrMyMA = GDFLIB_FILTER_MA_DEFAULT_FLT;

void main(void)
{
    // input value = 0.25
    fltInput = (tFloat)(0.25);

    // filter window = 8 samples
    flttrMyMA.fltLambda = (tFloat)(1.0/8.0);

    GDFLIB_FilterMAInit_FLT(&flttrMyMA);

    // output should be 0.031250000
    fltOutput = GDFLIB_FilterMA_FLT(fltInput,&flttrMyMA);

    // output should be 0.031250000
    fltOutput = GDFLIB_FilterMA(fltInput,&flttrMyMA,FLT);

    // ##### Available only if single precision floating point
    // implementation selected as default
    // #####
    // output should be 0.031250000
    fltOutput = GDFLIB_FilterMA(fltInput,&flttrMyMA);
}
```

EXAMPLE 2: Using a smoothing factor calculation to achieve the same signal-to-noise ratio improvement as if a uniform sliding-window moving average filter was used:

```
#include "gdflib.h"

tFloat fltInput;
tFloat fltOutput;

GDFLIB_FILTER_MA_T_FLT flttrMyMA = GDFLIB_FILTER_MA_DEFAULT_FLT;

void main(void)
{
    // input value = 0.25
    fltInput = (tFloat)(0.25);

    // filter window = 8 samples
    flttrMyMA.fltLambda = (tFloat)(2.0/(8.0 + 1.0));

    GDFLIB_FilterMAInit_FLT(&flttrMyMA);
```

```

// output should be 0.055555556
fltOutput = GDFLIB_FilterMA_FLT(fltInput,&flttrMyMA);

// output should be 0.055555556
fltOutput = GDFLIB_FilterMA(fltInput,&flttrMyMA,FLT);

// ######
// Available only if single precision floating point
// implementation selected as default
// #####

// output should be 0.055555556
fltOutput = GDFLIB_FilterMA(fltInput,&flttrMyMA);
}

```

#### 2.12.4 Function GDFLIB\_FilterMAInit

##### Description

This function clears the internal accumulator of a moving average filter. It shall be called after filter parameter initialization and whenever the filter initialization is required.

**Note:** The input/output pointers must contain valid addresses, otherwise a fault may occur (MemManage, BusFault, UsageFault, HardFault).

**Note:** This function shall not be called together with the GDFLIB\_FilterIIR1 function unless periodic clearing of filter buffers is required.

**Note:** This function shall not be called together with the GDFLIB\_FilterMA function unless periodic clearing of filter buffers is required.

##### Re-entrancy

The function is re-entrant.

#### 2.12.4.1 Function GDFLIB\_FilterMAInit\_F32

##### Declaration

```
void GDFLIB_FilterMAInit_F32(GDFLIB\_FILTER\_MA\_T\_F32 *pParam);
```

##### Arguments

Table 103. GDFLIB\_FilterMAInit\_F32 arguments

Type	Name	Direction	Description
<a href="#">GDFLIB_FILTER_MA_T_F32</a> *	pParam	input, output	Pointer to the filter structure with a filter accumulator and a smoothing factor.

##### Code Example

```

#include "gdflib.h"

GDFLIB\_FILTER\_MA\_T\_F32 f32trMyMA = GDFLIB\_FILTER\_MA\_DEFAULT\_F32;

void main(void)
{

```

```

// filter window = 2^5 = 32 samples
f32trMyMA.u16NSamples = 5;

GDFLIB_FilterMAInit_F32(&f32trMyMA);

GDFLIB_FilterMAInit(&f32trMyMA, F32);

// ##### Available only if 32-bit fractional implementation selected
// as default
// #####
GDFLIB_FilterMAInit(&f32trMyMA);
}

```

#### 2.12.4.2 Function GDFLIB\_FilterMAInit\_F16

##### Declaration

```
void GDFLIB_FilterMAInit_F16(GDFLIB_FILTER_MA_T_F16 *pParam);
```

##### Arguments

**Table 104. GDFLIB\_FilterMAInit\_F16 arguments**

Type	Name	Direction	Description
GDFLIB_FILTER_MA_T_F16 *	pParam	input, output	Pointer to the filter structure with a filter accumulator and a smoothing factor.

##### Code Example

```

#include "gdflib.h"

GDFLIB_FILTER_MA_T_F16 f16trMyMA = GDFLIB_FILTER_MA_DEFAULT_F16;

void main(void)
{
    // filter window = 2^3 = 8 samples
    f16trMyMA.u16NSamples = 3;

    GDFLIB_FilterMAInit_F16(&f16trMyMA);

    GDFLIB_FilterMAInit(&f16trMyMA, F16);

    // ##### Available only if 16-bit fractional implementation selected
    // as default
    // #####
    GDFLIB_FilterMAInit(&f16trMyMA);
}

```

#### 2.12.4.3 Function GDFLIB\_FilterMAInit\_FLT

##### Declaration

```
void GDFLIB_FilterMAInit_FLT(GDFLIB_FILTER_MA_T_FLT *pParam);
```

## Arguments

**Table 105. GDFLIB\_FilterMAInit\_FLT arguments**

Type	Name	Direction	Description
<a href="#">GDFLIB_FILTER_MA_T_FLT *</a>	pParam	input, output	Pointer to a filter structure with a filter accumulator and a smoothing factor.

**Note:** The floating-point unit must be enabled in CPACR register to prevent NOCP UsageFault exception.

## Code Example

```
#include "gdflib.h"

GDFLIB_FILTER_MA_T_FLT flttrMyMA = GDFLIB_FILTER_MA_DEFAULT_FLT;

void main(void)
{
    // filter window = 8 samples (using smoothing factor calculation
    // equivalent to the fixed-point implementation of GDFLIB_FilterMA)
    flttrMyMA.fltLambda = (tFloat)(1.0/8.0);

    GDFLIB_FilterMAInit_FLT(&flttrMyMA);

    GDFLIB_FilterMAInit(&flttrMyMA, FLT);

    // ######
    // Available only if single precision floating point
    // implementation selected as default
    // #####
    GDFLIB_FilterMAInit(&flttrMyMA);
}
```

## 2.12.5 Function GDFLIB\_FilterMASetState

### Description

This function initializes the internal accumulator of a moving average filter to achieve the required output value.

**Note:** The input/output pointers must contain valid addresses, otherwise a fault may occur (MemManage, BusFault, UsageFault, HardFault).

**Note:** This function should not be called periodically together with the GDFLIB\_FilterMA function unless periodic setting of the filter accumulator is required.

### Re-entrancy

The function is re-entrant for a different pParam.

### 2.12.5.1 Function GDFLIB\_FilterMASetState\_F32

#### Declaration

```
void GDFLIB_FilterMASetState_F32(tFrac32 f32FilterMAOut,  
GDFLIB\_FILTER\_MA\_T\_F32 *pParam);
```

#### Arguments

**Table 106. GDFLIB\_FilterMASetState\_F32 arguments**

Type	Name	Direction	Description
<a href="#">tFrac32</a>	f32FilterMAOut	<a href="#">input</a>	Required output of the FilterMA.
<a href="#">GDFLIB_FILTER_MA_T_F32</a> *	pParam	<a href="#">input, output</a>	Pointer to the filter structure with a filter accumulator and a smoothing factor.

#### Code Example

```
#include "gdflib.h"

GDFLIB\_FILTER\_MA\_T\_F32 f32trMyMA = GDFLIB\_FILTER\_MA\_DEFAULT\_F32;

void main(void)
{
    // filter window = 2^5 = 32 samples
    f32trMyMA.u16NSamples = 5;
    tFrac32 f32FilterMAOut;

    // Initialize the GDFLIB_FilterMA state variable to a predefined value
    // Warning: The u16NSamples parameter in f32trMyMA must be already
    // initialized.
    f32FilterMAOut = (tFrac32)123L; // required output value
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    GDFLIB_FilterMASetState_F32(f32FilterMAOut, &f32trMyMA);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    GDFLIB_FilterMASetState(f32FilterMAOut, &f32trMyMA, F32);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 32-bit fractional implementation is selected as default.
    GDFLIB_FilterMASetState(f32FilterMAOut, &f32trMyMA);
}
```

### 2.12.5.2 Function GDFLIB\_FilterMASetState\_F16

#### Declaration

```
void GDFLIB_FilterMASetState_F16(tFrac16 f16FilterMAOut,  
GDFLIB\_FILTER\_MA\_T\_F16 *pParam);
```

## Arguments

**Table 107.** GDFLIB\_FilterMASetState\_F16 arguments

Type	Name	Direction	Description
tFrac16	f16FilterMAOut	input	Required output of the FilterMA.
GDFLIB_FILTER_MA_T_F16 *	pParam	input, output	Pointer to the filter structure with a filter accumulator and a smoothing factor.

## Code Example

```
#include "gdflib.h"

GDFLIB_FILTER_MA_T_F16 f16trMyMA = GDFLIB_FILTER_MA_DEFAULT_F16;

void main(void)
{
    // filter window = 2^5 = 32 samples
    f16trMyMA.u16NSamples = 5;
    tFrac16 f16FilterMAOut;

    // Initialize the GDFLIB_FilterMA state variable to a predefined value
    // Warning: The u16NSamples parameter in f16trMyMA must be already
    // initialized.
    f16FilterMAOut = (tFrac16)123; // required output value
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    GDFLIB_FilterMASetState_F16(f16FilterMAOut, &f16trMyMA);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    GDFLIB_FilterMASetState(f16FilterMAOut, &f16trMyMA, F16);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 16-bit fractional implementation is selected as default.
    GDFLIB_FilterMASetState(f16FilterMAOut, &f16trMyMA);
}
```

**2.12.5.3 Function GDFLIB\_FilterMASetState\_FLT**

## Declaration

```
void GDFLIB_FilterMASetState_FLT(tFloat fltFilterMAOut,
                                 GDFLIB_FILTER_MA_T_FLT *pParam);
```

## Arguments

**Table 108.** GDFLIB\_FilterMASetState\_FLT arguments

Type	Name	Direction	Description
tFloat	fltFilterMAOut	input	Required output of the FilterMA.
GDFLIB_FILTER_MA_T_FLT *	pParam	input, output	Pointer to the filter structure with a filter accumulator and a smoothing factor.

## Code Example

```
#include "gdflib.h"

_GDFLIB_FILTER_MA_T_FILT flttrMyMA = _GDFLIB_FILTER_MA_DEFAULT_FLT;

void main(void)
{
    flttrMyMA.fltLambda = 0.5f;
    tFloat fltFilterMAOut;

    // Initialize the GDFLIB_FilterMA state variable to a predefined value
    // Warning: The fltLambda parameter in flttrMyMA must be already
    // initialized.
    fltFilterMAOut = 123.0f; // required output value
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    GDFLIB_FilterMASetState_FLT(fltFilterMAOut, &flttrMyMA);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    GDFLIB_FilterMASetState(fltFilterMAOut, &flttrMyMA, FLT);

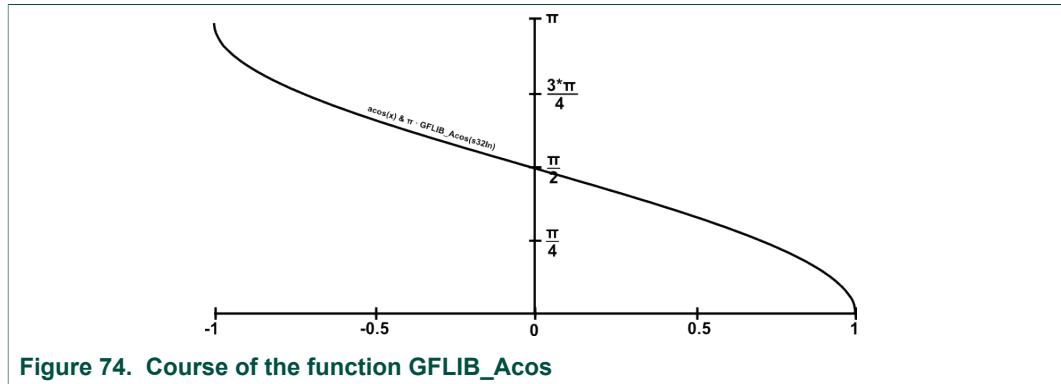
    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if single precision floating point implementation is selected as default.
    GDFLIB_FilterMASetState(fltFilterMAOut, &flttrMyMA);
}
```

## 2.13 Function GFLIB\_Acos

This function implements an approximation of arccosine function.

### Description

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



### Re-entrancy

The function is re-entrant.

### 2.13.1 Function `GFLIB_Acos_F32`

#### Declaration

```
tFrac32 GFLIB_Acos_F32(tFrac32 f32In, const GFLIB_ACOS_T_F32
*const pParam);
```

#### Arguments

Table 109. `GFLIB_Acos_F32` arguments

Type	Name	Direction	Description
<code>tFrac32</code>	<code>f32In</code>	<code>input</code>	Input argument is a 32-bit number that contains a value between [-1,1).
<code>const GFLIB_ACOS_T_F32 *const</code>	<code>pParam</code>	<code>input</code>	Pointer to an array of Taylor coefficients. In case the default approximation coefficients are used, the <code>&amp;pParam</code> must be replaced with <code>GFLIB_ACOS_DEFAULT_F32</code> symbol.

#### Return

The function returns  $\arccos(f32In)/\pi$  as a fixed point 32-bit number, normalized between [0,1).

#### Implementation details

The computational algorithm uses the relation between arccosine and arcsine function. At first the  $\arcsin(x)$  functional value is computed and then the result is corrected by following formula:

$$\cos^{-1}(x) = \frac{\pi}{2} - \sin^{-1}(x)$$

Equation GFLIB\_Acos\_F32\_Eq1

The computation of  $\arcsin(x)$  uses the symmetry of the function around the point (0, 0), which allows to for computing the function values just in the interval [0, 1) and to compute the function values in the interval [-1, 0) by the simple formula:

$$\sin^{-1}(-x) = -\sin^{-1}(x) \quad x \in [0, 1]$$

Equation GFLIB\_Acos\_F32\_Eq2

Additionally, because the  $\arcsin(x)$  function is difficult for polynomial approximation for  $x$  approaching 1 (or -1 by symmetry), due to its derivatives approaching infinity, a special transformation is used to transform the range of  $x$  from [0.5, 1] to (0, 0.5):

$$\sin^{-1}(\sqrt{1-x}) = \frac{\pi}{2} - \sin^{-1}(\sqrt{x})$$

Equation GFLIB\_Acos\_F32\_Eq3

In this way, the computation of the  $\arcsin(x)$  function in the range [0.5, 1) can be replaced by the computation in the range (0, 0.5], in which approximation is easier.

Moreover for interval [0.997, 1), different approximation coefficients are used to eliminate the imprecision of the polynom in this range.

For the interval (0, 0.5], the algorithm uses polynomial approximation as follows:

$$\frac{\sin^{-1}(fIn)}{\pi} = (a_0 + a_1 \cdot |fIn| + a_2 \cdot |fIn|^2 + a_3 \cdot |fIn|^3 + a_4 \cdot |fIn|^4) \cdot \text{sign}(fIn)$$

Equation GFLIB\_Acos\_F32\_Eq4

The division of the [0,1] interval into three sub-intervals, with polynomial coefficients calculated for each sub-interval, is noted in [Table 110](#).

**Table 110. Integer polynomial coefficients for each interval**

Interval	a <sub>0</sub>	a <sub>1</sub>	a <sub>2</sub>	a <sub>3</sub>	a <sub>4</sub>
<0 , 1/2)	12751	682829947	9729967	66340080	91918582
<1/2, 0.997)	1073630175	-964576326	-15136243	-36708911	-52453538
<0.997, 1)	1073739175	-966167437	-15136243	-36708911	-52453538

Until this point the implementation of the GFLIB\_Acos\_F32 is the same as in the function [GFLIB\\_Asin\\_F32](#). As last step the output of the GFLIB\_Acos\_F32 is corrected as follows:

$$fOut = \frac{\cos^{-1}(fIn)}{\pi} = \frac{1}{2} - \frac{\sin^{-1}(fIn)}{\pi}$$

Equation GFLIB\_Acos\_F32\_Eq5

The polynomial coefficients were obtained using the Matlab fitting function, where a polynomial of 5th order was used for the fitting of each respective sub-interval. The functions *arcsine* and *arccosine* are similar, therefore the GFLIB\_Acos\_F32 function uses the same polynomial coefficients as the [GFLIB\\_Asin\\_F32](#) function.

**Note:** The input pointer must contain a valid address otherwise a fault may occur (MemManage, BusFault, UsageFault, HardFault).

### Code Example

```
#include "gflib.h"

tFrac32 f32Input;
tFrac32 f32Angle;

void main(void)
{
    // input f32Input = 0
    f32Input = FRAC32(0);

    // output should be 0x400031CE = 0.5 => pi/2
    f32Angle = GFLIB_Acos_F32(f32Input, GFLIB_ACOS_DEFAULT_F32);

    // output should be 0x400031CE = 0.5 => pi/2
    f32Angle = GFLIB_Acos(f32Input, GFLIB_ACOS_DEFAULT_F32, F32);

    // ##### Available only if 32-bit fractional implementation selected #####
    // as default
    // #####
    // output should be 0x400031CE = 0.5 => pi/2
    f32Angle = GFLIB_Acos(f32Input);
}
```

## 2.13.2 Function `GFLIB_Acos_F16`

### Declaration

```
tFrac16 GFLIB_Acos_F16(tFrac16 f16In, const GFLIB_ACOS_T_F16
*const pParam);
```

### Arguments

Table 111. `GFLIB_Acos_F16` arguments

Type	Name	Direction	Description
<code>tFrac16</code>	<code>f16In</code>	<code>input</code>	Input argument is a 16-bit number that contains a value between [-1,1).
<code>const GFLIB_ACOS_T_F16 *const</code>	<code>pParam</code>	<code>input</code>	Pointer to an array of Taylor coefficients. In case the default approximation coefficients are used, the <code>&amp;pParam</code> must be replaced with <code>GFLIB_ACOS_DEFAULT_F16</code> symbol.

### Return

The function returns  $\arccos(f16In)/\pi$  as a fixed point 16-bit number, normalized between [0,1).

### Implementation details

The computational algorithm uses the relation between arccosine and arcsine function. At first the  $\arcsin(x)$  functional value is computed and then the result is corrected by following formula:

$$\cos^{-1}(x) = \frac{\pi}{2} - \sin^{-1}(x)$$

Equation GFLIB\_Acos\_F16\_Eq1

The computation of  $\arcsin(x)$  uses the symmetry of the function around the point (0, 0), which allows to for computing the function values just in the interval [0, 1) and to compute the function values in the interval [-1, 0) by the simple formula:

$$\sin^{-1}(-x) = -\sin^{-1}(x) \quad x \in [0, 1]$$

Equation GFLIB\_Acos\_F16\_Eq2

Additionally, because the  $\arcsin(x)$  function is difficult for polynomial approximation for  $x$  approaching 1 (or -1 by symmetry), due to its derivatives approaching infinity, a special transformation is used to transform the range of  $x$  from [0.5, 1] to (0, 0.5]:

$$\sin^{-1}(\sqrt{1-x}) = \frac{\pi}{2} - \sin^{-1}(\sqrt{x})$$

Equation GFLIB\_Acos\_F16\_Eq3

In this way, the computation of the  $\arcsin(x)$  function in the range [0.5, 1) can be replaced by the computation in the range (0, 0.5], in which approximation is easier.

For the interval (0, 0.5], the algorithm uses polynomial approximation as follows:

$$\frac{\sin^{-1}(fIn)}{\pi} = (a_0 + a_1 \cdot |fIn| + a_2 \cdot |fIn|^2 + a_3 \cdot |fIn|^3 + a_4 \cdot |fIn|^4) \cdot sign(fIn)$$

Equation GFLIB\_Acos\_F16\_Eq4

The division of the [0,1) interval into two sub-intervals, with polynomial coefficients calculated for each sub-interval, is noted in [Table 112](#).

**Table 112. Integer polynomial coefficients for each interval**

Interval	a <sub>0</sub>	a <sub>1</sub>	a <sub>2</sub>	a <sub>3</sub>	a <sub>4</sub>
<0 , 1/2)	1	10419	148	1012	1403
<1/2, 1)	16384	-14718	-231	-560	-800

Until this point the implementation of the GFLIB\_Acos\_F16 is the same as in the function [GFLIB\\_Asin\\_F16](#). As last step the output of the GFLIB\_Acos\_F16 is corrected as follows:

$$fOut = \frac{\cos^4(fIn)}{\pi} = \frac{1}{2} - \frac{\sin^4(fIn)}{\pi}$$

Equation GFLIB\_Acos\_F16\_Eq5

The polynomial coefficients were obtained using the Matlab fitting function, where a polynomial of 5th order was used for the fitting of each respective sub-interval. The functions *arcsine* and *arccosine* are similar, therefore the GFLIB\_Acos\_F16 function uses the same polynomial coefficients as the [GFLIB\\_Asin\\_F16](#) function.

**Note:** *The input pointer must contain a valid address otherwise a fault may occur (MemManage, BusFault, UsageFault, HardFault).*

### Code Example

```
#include "gflib.h"

tFrac16 f16Input;
tFrac16 f16Angle;

void main(void)
{
    // input f16Input = 0
    f16Input = FRAC16(0);

    // output should be 0x4000 = 0.5 => pi/2
    f16Angle = GFLIB_Acos_F16(f16Input, GFLIB_ACOS_DEFAULT_F16);

    // output should be 0x4000 = 0.5 => pi/2
    f16Angle = GFLIB_Acos(f16Input, GFLIB_ACOS_DEFAULT_F16, F16);

    // ##### Available only if 16-bit fractional implementation selected #####
    // as default
    // #####
    // output should be 0x4000 = 0.5 => pi/2
    f16Angle = GFLIB_Acos(f16Input);
}
```

### 2.13.3 Function `GFLIB_Acos_FLT`

#### Declaration

```
tFloat GFLIB_Acos_FLT(tFloat fltIn, const GFLIB_ACOS_T_FLT *const pParam);
```

#### Arguments

**Table 113. GFLIB\_Acos\_FLT arguments**

Type	Name	Direction	Description
<code>tFloat</code>	<code>fltIn</code>	<code>input</code>	Input argument is a 32-bit number that contains a single precision floating point value.
<code>const GFLIB_ACOS_T_FLT *const</code>	<code>pParam</code>	<code>input</code>	Pointer to an array of approximation coefficients. In case the default approximation coefficients are used, the <code>&amp;pParam</code> must be replaced with <code>GFLIB_ACOS_DEFAULT_FLT</code> symbol.

#### Return

The function returns  $\arccos(fltIn)$  as a single precision floating point number.

#### Implementation details

The computational algorithm uses the relation between arccosine and arcsine function. At first the  $\arcsin(x)$  functional value is computed and then the result is corrected by following formula:

$$\cos^{-1}(x) = \frac{\pi}{2} - \sin^{-1}(x)$$

Equation GFLIB\_Acos\_FLT\_Eq1

The computation algorithm for  $\arcsin(x)$  uses the symmetry of the function around the point  $(0, 0)$ , which allows the calculation of the function values just in interval  $[0, 1]$ , and to calculate the function values in the interval  $[-1, 0]$  use the simple formula:

$$\sin^{-1}(-x) = -\sin^{-1}(x) \quad x \in [0, 1]$$

Equation GFLIB\_Acos\_FLT\_Eq2

For the interval  $[0, 1]$ , the algorithm uses a polynomial approximation as follows:

$$\sin^{-1}(fIn) = \left[ \frac{\pi}{2} - \sqrt{1-fIn} \cdot (a_0 + a_1 \cdot |fIn| + a_2 \cdot |fIn|^2 + a_3 \cdot |fIn|^3) \right] \cdot sign(fIn)$$

Equation GFLIB\_Acos\_FLT\_Eq3

Polynomial approximation coefficients used for GFLIB\_Acos\_FLT calculation are noted in [Table 114](#).

**Table 114. Approximation polynomial coefficients**

$a_0$	$a_1$	$a_2$	$a_3$	$a_4$
1.5707811e+00	-2.1389899e-01	8.3511069e-02	-3.3877850e-02	7.6983864e-03

In order to get functional value of arccosine function, output of [GFLIB\\_Acos\\_FLT\\_Eq3](#) must be corrected by [GFLIB\\_Acos\\_FLT\\_Eq1](#).

**Note:** The input pointer must contain a valid address otherwise a fault may occur (MemManage, BusFault, UsageFault, HardFault).

The function may raise floating-point exceptions (invalid operation, division by zero, overflow, underflow, inexact, input denormal). The floating-point unit must be enabled in CPACR register to prevent NOCP UsageFault exception.

### Code Example

```
#include "gflib.h"

tFloat fltInput;
tFloat fltAngle;

void main(void)
{
    // input fltInput = 0
    fltInput = 0;

    // output should be 1.570796
    fltAngle = GFLIB_Acos_FLT(fltInput, GFLIB_ACOS_DEFAULT_FLT);

    // output should be 1.570796
    fltAngle = GFLIB_Acos(fltInput, GFLIB_ACOS_DEFAULT_FLT, FLT);

    // ######
    // Available only if single precision floating point
    // implementation selected as default
    // #####
    // output should be 1.570796
    fltAngle = GFLIB_Acos(fltInput);
}
```

## 2.14 Function GFLIB\_Asin

This function implements polynomial approximation of arcsine function.

### Description

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

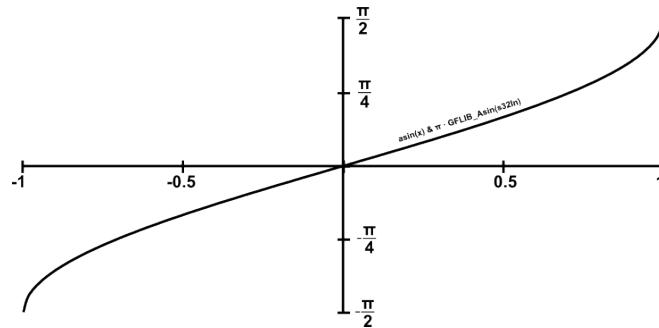


Figure 75. Course of the function GFLIB\_Asin

### Re-entrancy

The function is re-entrant.

#### 2.14.1 Function GFLIB\_Asin\_F32

##### Declaration

```
tFrac32 GFLIB_Asin_F32(tFrac32 f32In, const GFLIB_ASIN_T_F32
*const pParam);
```

##### Arguments

Table 115. GFLIB\_Asin\_F32 arguments

Type	Name	Direction	Description
tFrac32	f32In	input	Input argument is a 32-bit number that contains a value between [-1,1].
const GFLIB_ASIN_T_F32 *const	pParam	input	Pointer to an array of Taylor coefficients. In case the default approximation coefficients are used, the &pParam must be replaced with GFLIB_ASIN_DEFAULT_F32 symbol.

##### Return

The function returns  $\arcsin(f32In)/\pi$  as a fixed point 32-bit number, normalized between [-0.5,0.5].

##### Implementation details

The computational algorithm uses the symmetry of the  $\arcsin(x)$  function around the point  $(0, 0)$ , which allows to for computing the function values just in the interval  $[0, 1]$  and to compute the function values in the interval  $[-1, 0]$  by the simple formula:

$$\sin^{-1}(-x) = -\sin^{-1}(x) \quad x \in [0, 1]$$

Equation GFLIB\_Asin\_F32\_Eq1

Additionally, because the  $\arcsin(x)$  function is difficult for polynomial approximation for  $x$  approaching 1 (or -1 by symmetry), due to its derivatives approaching infinity, a special transformation is used to transform the range of  $x$  from  $[0.5, 1]$  to  $(0, 0.5]$ :

$$\sin^{-1}(\sqrt{1-x}) = \frac{\pi}{2} - \sin^{-1}(\sqrt{x})$$

Equation GFLIB\_Asin\_F32\_Eq2

In this way, the computation of the  $\arcsin(x)$  function in the range  $[0.5, 1]$  can be replaced by the computation in the range  $(0, 0.5]$ , in which approximation is easier.

Moreover for interval  $[0.997, 1)$ , different approximation coefficients are used to eliminate the imprecision of the polynom in this range.

For the interval  $(0, 0.5]$ , the algorithm uses polynomial approximation as follows:

$$fOut = \frac{\sin^{-1}(fIn)}{\pi} = (a_0 + a_1 \cdot |fIn| + a_2 \cdot |fIn|^2 + a_3 \cdot |fIn|^3 + a_4 \cdot |fIn|^4) \cdot sign(fIn)$$

Equation GFLIB\_Asin\_F32\_Eq3

The division of the  $[0,1]$  interval into two sub-intervals, with polynomial coefficients calculated for each sub-interval, is noted in [Table 116](#).

**Table 116. Integer polynomial coefficients for each interval**

Interval	a <sub>0</sub>	a <sub>1</sub>	a <sub>2</sub>	a <sub>3</sub>	a <sub>4</sub>
<0 , 1/2)	12751	682829947	9729967	66340080	91918582
<1/2, 0.997)	1073630175	-964576326	-15136243	-36708911	-52453538
<0.997, 1)	1073739175	-966167437	-15136243	-36708911	-52453538

Polynomial coefficients were obtained using the Matlab fitting function, where a polynomial of the 5th order was used for the fitting of each respective sub-interval. The Matlab was used as follows:

```

clear all
clc

number_of_range = 2;
i = 1;
range = 0;

Range = 1 / number_of_range;
x(i,:) = (((i-1)*Range):(1/(2^15)):((i)*Range))';
y(i,:) = asin(x(i,:))/pi;
p(i,:) = polyfit((x(i,:)),(y(i,:)),4);

i=i+1;

Range = 1 / number_of_range;
x(i,:) = (((i-1)*Range):(1/(2^15)):((i)*Range))';
y(i,:) = asin(x(i,:))/pi;
x1(i,:) = ((x(i,:) - ((i-1)*Range)));
x1(i,:) = 0.5 - x1(i,:);
x2(i,:) = sqrt(x1(i,:));
p(i,:) = polyfit((x2(i,:)),(y(i,:)),4);
i=i+1;

f(2,:) = polyval(p(2,:),x2(2,:));
f(1,:) = polyval(p(1,:),x(1,:));
error_1 = abs(f(2,:) - y(2,:));
max(error_1 * (2^15))
error_2 = abs(f(1,:) - y(1,:));

```

```
max(error_2 * (2^15))
plot(x(2,:),y(2,:),'-',x(2,:),f(2,:),'-',x(1,:),y(1,:),'-',x(1,:),f(1,:),'-');
coef = round(p * (2^31))
```

**Note:** The input pointer must contain a valid address otherwise a fault may occur (MemManage, BusFault, UsageFault, HardFault).

### Code Example

```
#include "gflib.h"

tFrac32 f32Input;
tFrac32 f32Angle;

void main(void)
{
    // input f32Input = (1-(2^-31))
    f32Input = (tFrac32)(0x7FFFFFFF);

    // output should be 0x3FFF5A7 = 0.4999987665
    f32Angle = GFLIB_Asin_F32(f32Input, GFLIB_ASIN_DEFAULT_F32);

    // output should be 0x3FFF5A7 = 0.4999987665
    f32Angle = GFLIB_Asin(f32Input, GFLIB_ASIN_DEFAULT_F32, F32);

    // ##### Available only if 32-bit fractional implementation selected
    // as default
    // #####
    // output should be 0x3FFF5A7 = 0.4999987665
    f32Angle = GFLIB_Asin(f32Input);
}
```

## 2.14.2 Function GFLIB\_Asin\_F16

### Declaration

```
tFrac16 GFLIB_Asin_F16(tFrac16 f16In, const GFLIB_ASIN_T_F16
*const pParam);
```

### Arguments

Table 117. GFLIB\_Asin\_F16 arguments

Type	Name	Direction	Description
tFrac16	f16In	input	Input argument is a 16-bit number that contains a value between [-1,1].
const GFLIB_ASIN_T_F16 *const	pParam	input	Pointer to an array of Taylor coefficients. In case the default approximation coefficients are used, the &pParam must be replaced with GFLIB_ASIN_DEFAULT_F16 symbol.

### Return

The function returns  $\arcsin(f16In)/\pi$  as a fixed point 16-bit number, normalized between [-0.5,0.5].

**Implementation details**

The computational algorithm uses the symmetry of the  $\arcsin(x)$  function around the point  $(0, 0)$ , which allows to for computing the function values just in the interval  $[0, 1)$  and to compute the function values in the interval  $[-1, 0)$  by the simple formula:

$$\sin^{-1}(-x) = -\sin^{-1}(x) \quad x \in [0, 1]$$

Equation GFLIB\_Asin\_F16\_Eq1

Additionally, because the  $\arcsin(x)$  function is difficult for polynomial approximation for  $x$  approaching 1 (or -1 by symmetry), due to its derivatives approaching infinity, a special transformation is used to transform the range of  $x$  from  $[0.5, 1)$  to  $(0, 0.5]$ :

$$\sin^{-1}(\sqrt{1-x}) = \frac{\pi}{2} - \sin^{-1}(\sqrt{x})$$

Equation GFLIB\_Asin\_F16\_Eq2

In this way, the computation of the  $\arcsin(x)$  function in the range  $[0.5, 1)$  can be replaced by the computation in the range  $(0, 0.5]$ , in which approximation is easier.

For the interval  $(0, 0.5]$ , the algorithm uses polynomial approximation as follows:

$$fOut = \frac{\sin^{-1}(fIn)}{\pi} = (a_0 + a_1 \cdot |fIn| + a_2 \cdot |fIn|^2 + a_3 \cdot |fIn|^3 + a_4 \cdot |fIn|^4) \cdot sign(fIn)$$

Equation GFLIB\_Asin\_F16\_Eq3

The division of the  $[0, 1)$  interval into two sub-intervals, with polynomial coefficients calculated for each sub-interval, is noted in [Table 118](#).

**Table 118. Integer polynomial coefficients for each interval**

Interval	$a_0$	$a_1$	$a_2$	$a_3$	$a_4$
$<0, 1/2)$	1	10419	148	1012	1403
$<1/2, 1)$	16384	-14718	-231	-560	-800

Polynomial coefficients were obtained using the Matlab fitting function, where a polynomial of the 5th order was used for the fitting of each respective sub-interval. The Matlab was used as follows:

```
clear all
clc

number_of_range = 2;
i = 1;
range = 0;

Range = 1 / number_of_range;
x(i,:) = (((i-1)*Range):(1/(2^15)):((i)*Range))';
y(i,:) = asin(x(i,:))/pi;
p(i,:) = polyfit((x(i,:)),(y(i,:)),4);

i=i+1;

Range = 1 / number_of_range;
x(i,:) = (((i-1)*Range):(1/(2^15)):((i)*Range))';
y(i,:) = asin(x(i,:))/pi;
```

```

x1(i,:) = ((x(i,:) - ((i-1)*Range)));
x1(i,:) = 0.5 - x1(i,:);
x2(i,:) = sqrt(x1(i,:));
p(i,:) = polyfit((x2(i,:)),(y(i,:)),4);
i=i+1;

f(2,:) = polyval(p(2,:),x2(2,:));
f(1,:) = polyval(p(1,:),x(1,:));
error_1 = abs(f(2,:) - y(2,:));
max(error_1 * (2^15))
error_2 = abs(f(1,:) - y(1,:));
max(error_2 * (2^15))
plot(x(2,:),y(2,:),'-',x(2,:),f(2,:),'-',x(1,:),y(1,:),'-',x(1,:),f(1,:),'-');
coef = round(p * (2^31))

```

**Note:** The input pointer must contain a valid address otherwise a fault may occur (MemManage, BusFault, UsageFault, HardFault).

### Code Example

```

#include "gflib.h"

tFrac16 f16Input;
tFrac16 f16Angle;

void main(void)
{
    // input f16Input = (1-(2^-15))
    f16Input = (tFrac16)(0x7FFF);

    // output should be 0x3FAE = 0.4974975
    f16Angle = GFLIB_Asin_F16(f16Input, GFLIB_ASIN_DEFAULT_F16);

    // output should be 0x3FAE = 0.4974975
    f16Angle = GFLIB_Asin(f16Input, GFLIB_ASIN_DEFAULT_F16, F16);

    // ##### Available only if 16-bit fractional implementation selected
    // as default
    // #####
    // output should be 0x3FAE = 0.4974975
    f16Angle = GFLIB_Asin(f16Input);
}

```

### 2.14.3 Function GFLIB\_Asin\_FLT

#### Declaration

```

tFloat GFLIB_Asin_FLT(tFloat fltIn, const GFLIB_ASIN_T_FLT *const
pParam);

```

**Arguments****Table 119.** `GFLIB_Asin_FLT` arguments

Type	Name	Direction	Description
<code>tFloat</code>	<code>fIn</code>	<code>input</code>	Input argument is a 32-bit number that contains a single precision floating point value.
<code>const GFLIB_ASIN_T FLT *const</code>	<code>pParam</code>	<code>input</code>	Pointer to an array of approximation coefficients. In case the default approximation coefficients are used, the <code>&amp;pParam</code> must be replaced with <code>GFLIB_ASIN_DEFAULT_FLT</code> symbol.

**Return**

The function returns  $\arcsin(fIn)$  as a single precision floating point number.

**Implementation details**

The computation algorithm for  $\arcsin(x)$  uses the symmetry of the  $\arcsin(x)$  function around the point  $(0, 0)$ , which allows the calculation of the function values just in interval  $[0, 1]$ , and to calculate of the function values in the interval  $[-1, 0)$ , use the simple formula:

$$\sin^{-1}(-x) = -\sin^{-1}(x) \quad x \in [0, 1]$$

Equation GFLIB\_Asin\_FLT\_Eq1

For the interval  $[0, 0.41]$ , the algorithm uses a Minimax approximation as follows:

$$\sin^{-1}(fIn) = (a_0 \cdot |fIn| + a_1 \cdot |fIn|^3 + a_2 \cdot |fIn|^5) \cdot sign(fIn)$$

Equation GFLIB\_Asin\_FLT\_Eq2

For the interval  $[0.41, 1)$ , the algorithm uses a polynomial approximation as follows.

$$\sin^{-1}(fIn) = \left[ \frac{\pi}{2} - \sqrt{1-fIn} \cdot (a_0 + a_1 \cdot |fIn| + a_2 \cdot |fIn|^2 + a_3 \cdot |fIn|^3) \right] \cdot sign(fIn)$$

Equation GFLIB\_Asin\_FLT\_Eq3

Minimax approximation coefficients used for `GFLIB_Asin_FLT` calculation in interval  $[0, 0.41]$ , are noted in [Table 120](#).

**Table 120.** Approximation polynomial coefficients

$a_0$	$a_1$	$a_2$
1.0000083e+00	1.6580229e-01	8.8028289e-02

In [Table 121](#) there are the approximation coefficients used for `GFLIB_Asin_FLT` calculation in interval  $[0.41, 1)$ .

**Table 121.** Approximation polynomial coefficients

$a_0$	$a_1$	$a_2$	$a_3$
1.5693614e+00	-2.0464350e-01	6.1917335e-02	-1.2426286e-02

**Note:** The function may raise floating-point exceptions (invalid operation, division by zero, overflow, underflow, inexact, input denormal). The floating-point unit must be enabled in CPACR register to prevent NOCP UsageFault exception.

The input pointer must contain a valid address otherwise a fault may occur (MemManage, BusFault, UsageFault, HardFault).

### Code Example

```
#include "gflib.h"

tFloat fltInput;
tFloat fltAngle;

void main(void)
{
    // input fltInput = 1
    fltInput = 1;

    // output should be 1.570796
    fltAngle = GFLIB_Asin_FLT(fltInput, GFLIB_ASIN_DEFAULT_FLT);

    // output should be 1.570796
    fltAngle = GFLIB_Asin(fltInput, GFLIB_ASIN_DEFAULT_FLT, FLT);

    // ##### Available only if single precision floating point
    // implementation selected as default
    // #####
    // output should be 1.570796
    fltAngle = GFLIB_Asin(fltInput);
}
```

## 2.15 Function GFLIB\_Atan

This function implements minimax polynomial approximation of arctangent function.

### Description

The GFLIB\_Atan function provides a computational method for calculation of a standard trigonometric *arctangent* function  $\arctan(x)$ . 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 the angle divided by the length of the side adjacent to the angle. The graph of  $\arctan(x)$  is shown in [Figure 76](#).

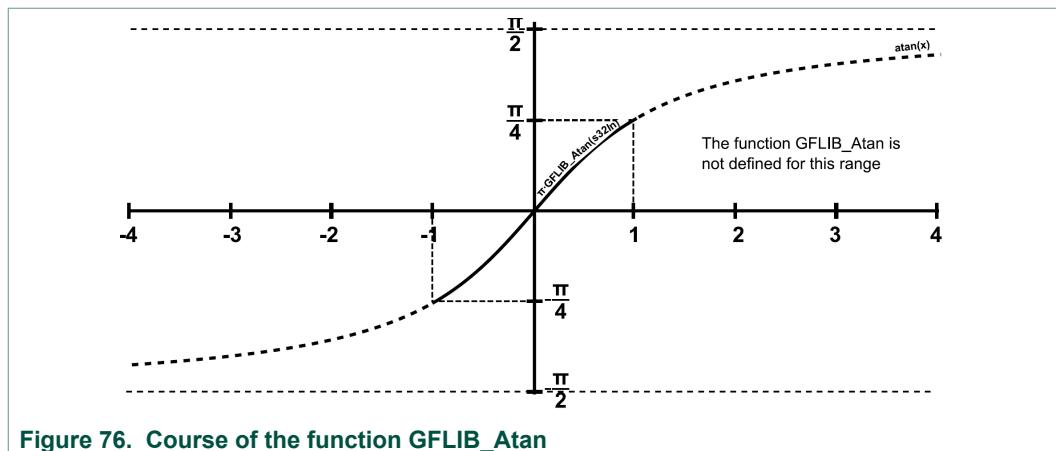


Figure 76. Course of the function GFLIB\_Atan

## Re-entrancy

The function is re-entrant.

### 2.15.1 Function GFLIB\_Atan\_F32

#### Declaration

```
tFrac32 GFLIB_Atan_F32(tFrac32 f32In, const GFLIB_ATAN_T_F32
*const pParam);
```

#### Arguments

Table 122. GFLIB\_Atan\_F32 arguments

Type	Name	Direction	Description
<code>tFrac32</code>	<code>f32In</code>	<code>input</code>	Input argument is a 32-bit number between [-1,1].
<code>const GFLIB_ATAN_T_F32 *const</code>	<code>pParam</code>	<code>input</code>	Pointer to an array of minimax approximation coefficients. In case the default approximation coefficients are used, the <code>&amp;pParam</code> must be replaced with <code>GFLIB_ATAN_DEFAULT_F32</code> symbol.

#### Return

The function returns the  $\text{atan}(f32In)/\pi$  as a fixed point 32-bit number, normalized between [-0.25, 0.25].

#### Implementation details

The computational algorithm uses the symmetry of the arctangent function around the point  $(0, 0)$ , which allows to for computing the function values just in the interval  $[0, 1)$  and to compute the function values in the interval  $[-1, 0)$  by the simple formula:

$$\tan^{-1}(-x) = -\tan^{-1}(x)$$

Equation GFLIB\_Atan\_F32\_Eq1

The GFLIB\_Atan\_F32 function approximates the arctangent function using a piece-wise minimax polynomial approximation. The input range  $[0, 1)$  is divided into eight equally spaced sub intervals, each with a distinct set of minimax coefficients. Negative inputs are calculated according to the antisymmetry of the function.

The GFLIB\_Atan\_F32 function uses fixed point fractional arithmetic, so to cast the fractional value of the output angle [-0.25, 0.25) into the correct range [- $\pi/4$ ,  $\pi/4$ ), the fixed point output angle can be multiplied by  $\pi$  for an angle in radians. Then, the fixed point fractional implementation of the minimax approximation polynomial, used for calculation of each sub sector, is defined as follows:

$$fOut = \frac{\tan^{-1}(fIn)}{\pi} = (a_0 + a_1 \cdot |fIn| + a_2 \cdot |fIn|^2) \cdot \text{sign}(fIn)$$

Equation GFLIB\_Atan\_F32\_Eq2

The division of the [0, 1) interval into eight sub-intervals, with polynomial coefficients calculated for each sub-interval, is noted in [Table 123](#).

**Table 123. Integer minimax polynomial coefficients for each interval**

Interval	a <sub>0</sub>	a <sub>1</sub>	a <sub>2</sub>
<0, 1/8)	42667172	42515925	-164794
<1/8, 2/8)	126697014	41238272	-465182
<2/8, 3/8)	207041074	38899574	-690034
<3/8, 4/8)	281909001	35848645	-820713
<4/8, 5/8)	350251355	32453241	-865105
<5/8, 6/8)	411702516	29016149	-845462
<6/8, 7/8)	466407809	25743137	-786689
<7/8, 1)	514828039	22748418	-708969

**Note:** The input pointer must contain a valid address otherwise a fault may occur (MemManage, BusFault, UsageFault, HardFault).

### Code Example

```
#include "gflib.h"

tFrac32 f32Input;
tFrac32 f32Angle;

void main(void)
{
    // input ratio = 0x7FFFFFFF
    f32Input = (tFrac32)(0x7FFFFFFF);

    // output angle should be 0x1FFF29F = 0.249999 => pi/4
    f32Angle = GFLIB_Atan_F32(f32Input, GFLIB_ATAN_DEFAULT_F32);

    // output angle should be 0x1FFF29F = 0.249999 => pi/4
    f32Angle = GFLIB_Atan(f32Input, GFLIB_ATAN_DEFAULT_F32, F32);

    // ##### Available only if 32-bit fractional implementation selected
    // as default
    // #####
    // output angle should be 0x1FFF29F = 0.249999 => pi/4
    f32Angle = GFLIB_Atan(f32Input);
}
```

## 2.15.2 Function `GFLIB_Atan_F16`

### Declaration

```
tFrac16 GFLIB_Atan_F16(tFrac16 f16In, const GFLIB_ATAN_T_F16
*const pParam);
```

### Arguments

**Table 124. GFLIB\_Atan\_F16 arguments**

Type	Name	Direction	Description
<code>tFrac16</code>	<code>f16In</code>	<code>input</code>	Input argument is a 16-bit number between [-1, 1).
<code>const GFLIB_ATAN_T_F16 *const</code>	<code>pParam</code>	<code>input</code>	Pointer to an array of minimax approximation coefficients. In case the default approximation coefficients are used, the <code>&amp;pParam</code> must be replaced with <code>GFLIB_ATAN_DEFAULT_F16</code> symbol.

### Return

The function returns the  $\text{atan}(f32In)/\pi$  as a fixed point 32-bit number, normalized between [-0.25, 0.25].

### Implementation details

The computational algorithm uses the symmetry of the arctangent function around the point (0, 0), which allows to for computing the function values just in the interval [0, 1) and to compute the function values in the interval [-1, 0) by the simple formula:

$$\tan^{-1}(-x) = -\tan^{-1}(x)$$

Equation GFLIB\_Atan\_F16\_Eq1

The GFLIB\_Atan\_F16 function approximates the arctangent function using a piece-wise minimax polynomial approximation. The input range [0, 1) is divided into eight equally spaced sub intervals, each with a distinct set of minimax coefficients. Negative inputs are calculated according to the antisymmetry of the function.

The GFLIB\_Atan\_F16 function uses fixed point fractional arithmetic, so to cast the fractional value of the output angle [-0.25, 0.25] into the correct range [- $\pi/4$ ,  $\pi/4$ ], the fixed point output angle can be multiplied by  $\pi$  for an angle in radians. Then, the fixed point fractional implementation of the minimax approximation polynomial, used for calculation of each sub sector, is defined as follows:

$$fOut = \frac{\tan^{-1}(fIn)}{\pi} = (a_0 + a_1 \cdot |fIn| + a_2 \cdot |fIn|^2) \cdot sign(fIn)$$

Equation GFLIB\_Atan\_F16\_Eq2

The division of the [0, 1) interval into eight sub-intervals, with minimax polynomial coefficients calculated for each sub-interval, is noted in [Table 125](#).

**Table 125. Integer polynomial coefficients for each interval**

Interval	$a_0$	$a_1$	$a_2$
<0, 1/8)	652	649	-3
<1/8, 2/8)	1934	630	-7
<2/8, 3/8)	3160	594	-11

<3/8, 4/8)	4302	547	-13
<4/8, 5/8)	5345	495	-13
<5/8, 6/8)	6283	443	-13
<6/8, 7/8)	7117	393	-12
<7/8, 1)	7856	347	-11

**Note:** The input pointer must contain a valid address otherwise a fault may occur (MemManage, BusFault, UsageFault, HardFault).

### Code Example

```
#include "gflib.h"

tFrac16 f16Input;
tFrac16 f16Angle;

void main(void)
{
    // input ratio = 0x7FFF
    f16Input = (tFrac16)(0x7FFF);

    // output angle should be 0x1FFF = 0.249999 => pi/4
    f16Angle = GFLIB_Atan_F16(f16Input, GFLIB_ATAN_DEFAULT_F16);

    // output angle should be 0x1FFF = 0.249999 => pi/4
    f16Angle = GFLIB_Atan(f16Input, GFLIB_ATAN_DEFAULT_F16, F16);

    // ##### Available only if 16-bit fractional implementation selected
    // as default
    // #####
    // output angle should be 0x1FFF = 0.249999 => pi/4
    f16Angle = GFLIB_Atan(f16Input);
}
```

### 2.15.3 Function GFLIB\_Atan\_FLT

#### Declaration

```
tFloat GFLIB_Atan_FLT(tFloat fltIn, const GFLIB_ATAN_T_FLT *const pParam);
```

#### Arguments

Table 126. GFLIB\_Atan\_FLT arguments

Type	Name	Direction	Description
tFloat	fltIn	input	Input argument is a single precision floating point number between (- 2 <sup>128</sup> , 2 <sup>128</sup> ).
const GFLIB_ATAN_T_FLT *const	pParam	input	Pointer to an array of rational polynomial coefficients. In case the default approximation coefficients are used, the &pParam must be replaced with GFLIB_ATAN_DEFAULT_FLT symbol.

**Return**

The function returns the atan of the input argument as a single precision floating point number that contains the angle in radians between (- $\pi/2$ ,  $\pi/2$ ).

**Implementation details**

The computational algorithm uses the symmetry of the arctangent function around the point (0, 0), which allows to for computing the function values just in the interval [0, 1) and to compute the function values in the interval [-1, 0) by the simple formula:

$$\tan^{-1}(-x) = -\tan^{-1}(x)$$

Equation GFLIB\_Atan\_FLT\_Eq1

The GFLIB\_Atan\_FLT function approximates the arctangent function using a rational polynomial approximation. The base interval of input values [0,  $\tan(\pi/12)$ ] is calculated using the equation

$$fOut = \frac{fIn[a_0 + fIn^2(a_1 + a_2 fIn^2)]}{a_3 + fIn^2[a_4 + fIn^2(a_5 + fIn^2)]} \cdot sign(fIn)$$

Equation GFLIB\_Atan\_FLT\_Eq2

The rational polynomial coefficients are noted in [Table 127](#).

**Table 127. Rational polynomial approximation coefficients**

a <sub>0</sub>	a <sub>1</sub>	a <sub>2</sub>	a <sub>3</sub>	a <sub>4</sub>	a <sub>5</sub>
48.7010700440489	49.5326328772254	9.40604354231624	65.7663270508956	21.5878701670202	48.7010700440499

Input values in the interval ( $\tan(\pi/12)$ , 1] are transformed into the base interval [0,  $\tan(\pi/12)$ ] as follows:

$$fIn = \frac{fIn + \tan(\frac{\pi}{12})}{1 + fIn \tan(\frac{\pi}{12})} \quad if \quad |fIn| > \tan(\frac{\pi}{12})$$

Equation GFLIB\_Atan\_FLT\_Eq3

For input values greater than one, a reciprocal value is used in the rational polynomial approximation and the result is subtracted from  $\pi/2$ .

Results for the negative inputs are calculated according to the antisymmetry of the arctangent function.

**Note:** The input pointer must contain a valid address otherwise a fault may occur (MemManage, BusFault, UsageFault, HardFault).

The function may raise floating-point exceptions (invalid operation, division by zero, overflow, underflow, inexact, input denormal). The floating-point unit must be enabled in CPACR register to prevent NOCP UsageFault exception.

**Code Example**

```
#include "gflib.h"

tFloat fltInput;
tFloat fltAngle;
```

```

void main(void)
{
    // input ratio = 1
    fltInput = 1;

    // output angle should be 0.7853981634 => pi/4
    fltAngle = GFLIB_Atan_FLT(fltInput, GFLIB\_ATAN\_DEFAULT\_FLT);

    // output angle should be 0.7853981634 => pi/4
    fltAngle = GFLIB_Atan(fltInput, GFLIB\_ATAN\_DEFAULT\_FLT, FLT);

    #####
    // Available only if single precision floating point
    // implementation selected as default
    #####
    #####

    // output angle should be 0.7853981634 => pi/4
    fltAngle = GFLIB_Atan(fltInput);

}

```

## 2.16 Function GFLIB\_AtanYX

This function calculates the angle between the positive x-axis and the direction of a vector given by the (x, y) coordinates.

### Description

The function returns the angle between the positive x-axis of a plane and the direction of the vector given by the x and y coordinates provided as parameters. The first parameter is the ordinate (the y coordinate) and the second one is the abscissa (the x coordinate).

### Re-entrancy

The function is re-entrant.

#### 2.16.1 Function GFLIB\_AtanYX\_F32

##### Declaration

```
tFrac32 GFLIB_AtanYX_F32 (tFrac32 f32InY, tFrac32 f32InX);
```

##### Arguments

Table 128. GFLIB\_AtanYX\_F32 arguments

Type	Name	Direction	Description
<a href="#">tFrac32</a>	f32InY	input	The ordinate of the input vector (y coordinate).
<a href="#">tFrac32</a>	f32InX	input	The abscissa of the input vector (x coordinate).

##### Return

The function returns the angle between the positive x-axis of a plane and the direction of the vector given by the x and y coordinates provided as parameters.

### Implementation details

Both the input parameters are assumed to be in the fractional range of [-1, 1). The computed angle is limited by the fractional range of [-1, 1), which corresponds to the real range of [- $\pi$ ,  $\pi$ ). The counter-clockwise direction is assumed to be positive and thus a positive angle will be computed if the provided ordinate (f32InY) is positive. Similarly, a negative angle will be computed for the negative ordinate.

The calculations are performed in a few 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 deg (half-quarter) and the remaining offset within the 45 deg 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 ordinary arctangent function.

The sum of the integral multiple of half-quarters and the angle offset within a single half-quarter form the angle to be computed. The function will return 0 if both input arguments are 0.

In comparison to the [GFLIB\\_Atan\\_F32](#) function, the GFLIB\_AtanYX\_F32 function correctly places the calculated angle within the whole fractional range of [-1, 1), which corresponds to the real angle range of [- $\pi$ ,  $\pi$ ].

**Note:** The function calls the [GFLIB\\_Atan\\_F32](#) function. The computed value is within the range of [-1, 1).

### Code Example

```
#include "gflib.h"

tFrac32 f32InY;
tFrac32 f32InX;
tFrac32 f32Ang;

void main(void)
{
    // Angle 45 deg = PI/4 rad
    f32InY = FRAC32(0.5);
    f32InX = FRAC32(0.5);

    // output should be close to 0x200034EA
    f32Ang = GFLIB_AtanYX_F32(f32InY, f32InX);

    // output should be close to 0x200034EA
    f32Ang = GFLIB_AtanYX(f32InY, f32InX, F32);

    // ##### Available only if 32-bit fractional implementation selected
    // as default
    // #####
    // output should be close to 0x200034EA
    f32Ang = GFLIB_AtanYX(f32InY, f32InX);
```

{}

## 2.16.2 Function [GFLIB\\_AtanYX\\_F16](#)

### Declaration

```
tFrac16 GFLIB_AtanYX_F16(tFrac16 f16InY, tFrac16 f16InX);
```

### Arguments

**Table 129. GFLIB\_AtanYX\_F16 arguments**

Type	Name	Direction	Description
<a href="#">tFrac16</a>	f16InY	input	The ordinate of the input vector (y coordinate).
<a href="#">tFrac16</a>	f16InX	input	The abscissa of the input vector (x coordinate).

### Return

The function returns the angle between the positive x-axis of a plane and the direction of the vector given by the x and y coordinates provided as parameters.

### Implementation details

Both the input parameters are assumed to be in the fractional range of [-1, 1). The computed angle is limited by the fractional range of [-1, 1), which corresponds to the real range of [- $\pi$ ,  $\pi$ ). The counter-clockwise direction is assumed to be positive and thus a positive angle will be computed if the provided ordinate (f16InY) is positive. Similarly, a negative angle will be computed for the negative ordinate.

The calculations are performed in a few 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 deg (half-quarter) and the remaining offset within the 45 deg 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 ordinary arctangent function.

The sum of the integral multiple of half-quarters and the angle offset within a single half-quarter form the angle to be computed. The function will return 0 if both input arguments are 0.

In comparison to the [GFLIB\\_Atan\\_F16](#) function, the GFLIB\_AtanYX\_F16 function correctly places the calculated angle within the whole fractional range of [-1, 1), which corresponds to the real angle range of [- $\pi$ ,  $\pi$ ].

**Note:** The function calls the [GFLIB\\_Atan\\_F16](#) function. The computed value is within the range of [-1, 1).

### Code Example

```
#include "gflib.h"

tFrac16 f16InY;
```

```

tFrac16 f16InX;
tFrac16 f16Ang;

void main(void)
{
    // Angle 45 deg = PI/4 rad
    f16InY = FRAC16(0.5);
    f16InX = FRAC16(0.5);

    // output should be close to 0x2000
    f16Ang = GFLIB_AtanYX_F16(f16InY, f16InX);

    // output should be close to 0x2000
    f16Ang = GFLIB_AtanYX(f16InY, f16InX, F16);

    // ##### Available only if 16-bit fractional implementation selected
    // as default
    // #####
    // output should be close to 0x2000
    f16Ang = GFLIB_AtanYX(f16InY, f16InX);
}

```

### 2.16.3 Function GFLIB\_AtanYX\_FLT

#### Declaration

`tFloat GFLIB_AtanYX_FLT(tFloat fltInY, tFloat fltInX);`

#### Arguments

Table 130. GFLIB\_AtanYX\_FLT arguments

Type	Name	Direction	Description
tFloat	fltInY	input	The ordinate of the input vector (y coordinate).
tFloat	fltInX	input	The abscissa of the input vector (x coordinate).

#### Return

The function returns the angle between the positive x-axis of a plane and the direction of the vector given by the x and y coordinates provided as parameters.

#### Implementation details

Both the input parameters are assumed to be in the single precision floating point range of  $(-2^{128}, 2^{128})$ . The computed angle is in the range of  $[-\pi, \pi]$ . The counter-clockwise direction is assumed to be positive, and thus a positive angle will be computed if the provided ordinate (fltInY) is positive. Similarly, a negative angle will be computed for the negative ordinate. The calculations are performed in a few steps.

In the first step, the angle is positioned within the correct half-quarter of the circumference of a circle, and if necessary the final angle addition is prepared or the sign of the input value is corrected.

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 ordinary arctangent function.

The sum of the angle addition and the angle offset within a single quarter form the angle to be computed. The function will return 0 if both input arguments are 0.

**Note:** The function may raise floating-point exceptions (invalid operation, division by zero, overflow, underflow, inexact, input denormal). The floating-point unit must be enabled in CPACR register to prevent NOCP UsageFault exception.

The function calls the [GFLIB\\_Atan\\_FLT](#) function. The computed value is within the range of  $[-\pi, \pi]$ .

### Code Example

```
#include "gflib.h"

tFloat fltInY;
tFloat fltInX;
tFloat fltAng;

void main(void)
{
    // Angle 45 deg = PI/4 rad
    fltInY = (tFloat)(0.5);
    fltInX = (tFloat)(0.5);

    // Output angle should be 45 deg = PI/4 rad = 0.7853981634
    fltAng = GFLIB_AtanYX_FLT(fltInY, fltInX);

    // Output angle should be 45 deg = PI/4 rad = 0.7853981634
    fltAng = GFLIB_AtanYX(fltInY, fltInX, FLT);

    // ######
    // Available only if single precision floating point
    // implementation selected as default
    // #####
    // Output angle should be 45 deg = PI/4 rad = 0.7853981634
    fltAng = GFLIB_AtanYX(fltInY, fltInX);
}
```

## 2.17 Function GFLIB\_AtanYXShifted

This function calculates the angle of two sine waves shifted in phase to each other.

### Description

The function calculates the angle of two sinusoidal signals, one shifted in phase to the other. The phase shift between sinusoidal signals does not have to be  $\pi/2$  and can be any value.

It is assumed that the arguments of the function are as follows:

$$\begin{aligned} y &= \sin(\theta) \\ x &= \sin(\theta + \Delta\theta) \end{aligned}$$

Equation GFLIB\_AtanYXShifted\_Eq1

where:

- x, y are respectively, the InX and InY arguments
- $\theta$  is the angle to be computed by the function
- $\Delta\theta$  is the phase difference between the x, y signals

At the end of computations, an angle offset  $\theta_{Offset}$  is added to the computed angle  $\theta$ . The angle offset is an additional parameter, which can be used to set the zero of the  $\theta$  axis. If  $\theta_{Offset}$  is zero, then the angle computed by the function will be exactly  $\theta$ .

**Note:** The input/output pointers must contain valid addresses, otherwise a fault may occur (MemManage, BusFault, UsageFault, HardFault).

## Re-entrancy

The function is re-entrant.

### 2.17.1 Function `GFLIB_AtanYXShifted_F32`

#### Declaration

```
tFrac32 GFLIB_AtanYXShifted_F32 (tFrac32 f32InY, tFrac32 f32InX,
const GFLIB_ATANYXSHIFTED_T_F32 *pParam);
```

#### Arguments

Table 131. `GFLIB_AtanYXShifted_F32` arguments

Type	Name	Direction	Description
tFrac32	f32InY	input	The value of the first signal, assumed to be $\sin(\theta)$ .
tFrac32	f32InX	input	The value of the second signal, assumed to be $\sin(\theta + \Delta\theta)$ .
const GFLIB_ATANYXSHIFTED_T_F32 *	pParam	input, output	The parameters for the function.

#### Return

The function returns the angle of two sine waves shifted in phase to each other.

#### Implementation details

The `GFLIB_AtanYXShifted_F32` function does not directly use the angle offset  $\theta_{Offset}$  and the phase difference  $\theta$ . The function's parameters, contained in the function parameters structure `GFLIB_ATANYXSHIFTED_T_F32`, need to be computed by means of the provided Matlab function (see below).

If  $\Delta\theta = \pi/2$  or  $\Delta\theta = -\pi/2$ , then the function is similar to the `GFLIB_AtanYX_F32` function, however, the `GFLIB_AtanYX_F32` function in this case is more effective with regard to execution time and accuracy.

In order to use the function, the following necessary steps need to be completed:

- define  $\Delta\theta$  and  $\theta_{Offset}$ , the  $\Delta\theta$  shall be known from the input sinusoidal signals, the  $\theta_{Offset}$  needs to be set arbitrarily
- compute values for the function parameters structure by means of the provided Matlab function
- convert the computed values into integer format and insert them into the C code (see also the C code example)

The function uses the following algorithm for computing the angle:

$$\begin{aligned} b &= \frac{S}{2\cos(\frac{\Delta\theta}{2})} (y + x) \\ a &= \frac{S}{2\sin(\frac{\Delta\theta}{2})} (x - y) \\ \theta &= \tan^{-1}\left(\frac{y}{x}\right) - \left(\frac{\Delta\theta}{2} - \theta_{offset}\right) \end{aligned}$$

Equation GFLIB\_AtanYXShifted\_F32\_Eq1

where:

- x, y are respectively, the f32InX, and f32InY
- $\theta$  is the angle to be computed by the function, see the previous equation
- $\Delta\theta$  is the phase difference between the x, y signals, see the previous equation
- S is a scaling coefficient, S is almost 1, ( $S < 1$ ), see also the explanation below
- a, b intermediate variables
- $\theta_{Offset}$  is the additional phase shift, the computed angle will be  $\theta + \theta_{Offset}$

The scale coefficient S is used to prevent overflow and to assure symmetry around 0 for the entire fractional range. S shall be less than 1.0, but as large as possible. The algorithm implemented in this function uses the value of  $1 - 2^{-15}$ .

The algorithm can be easily justified by proving the trigonometric identity:

$$\tan(\theta + \Delta\theta) = \frac{(y+x)\cos(\frac{\Delta\theta}{2})}{(x-y)\sin(\frac{\Delta\theta}{2})}$$

Equation GFLIB\_AtanYXShifted\_F32\_Eq2

For the purposes of fractional arithmetic, the algorithm is implemented such that additional values are used as shown in the equation below:

$$\begin{aligned} \frac{S}{2\cos(\frac{\Delta\theta}{2})} &= C_y = K_y \cdot 2^{N_y} \\ \frac{S}{2\sin(\frac{\Delta\theta}{2})} &= C_x = K_x \cdot 2^{N_x} \\ \theta_{adj} &= \frac{\Delta\theta}{2} - \theta_{offset} \end{aligned}$$

Equation GFLIB\_AtanYXShifted\_F32\_Eq3

where:

- $C_y, C_x$  are the algorithm coefficients for y and x signals
- $K_y$  is multiplication coefficient of the y signal, represented by the parameters structure member pParam->f32Ky
- $K_x$  is multiplication coefficient of the x signal, represented by the parameters structure member pParam->f32Kx
- $N_y$  is scaling coefficient of the y signal, represented by the parameters structure member pParam->s32Ny
- $N_x$  is scaling coefficient of the x signal, represented by the parameters structure member pParam->s32Nx
- $\theta_{adj}$  is an adjusting angle, represented by the parameters structure member pParam->f32ThetaAdj

The multiplication and scaling coefficients, and the adjusting angle, shall be defined in a parameters structure provided as the function input parameter.

The function initialization parameters can be calculated as shown in the following Matlab code:

```
function [KY, KX, NY, NX, THETAADJ] = atanyxshiftedpar(dthdeg, thoffsetdeg)
// ATANYXSHIFTEDPAR calculation of parameters for atanyxshifted() function
//
// [KY, KX, NY, NX, THETAADJ] = atanyxshiftedpar(dthdeg, thoffsetdeg)
//
// dthdeg = phase shift (delta theta) between sine waves in degrees
// thoffsetdeg = angle offset (theta offset) in degrees
// NY - scaling coefficient of y signal
// NX - scaling coefficient of x signal
// KY - multiplication coefficient of y signal
// KX - multiplication coefficient of x signal
// THETAADJ - adjusting angle in radians, scaled from [-pi, pi) to [-1, 1)

if (dthdeg < -180) || (dthdeg >= 180)
    error('atanyxshiftedpar: dthdeg out of range');
end
if (thoffsetdeg < -180) || (thoffsetdeg >= 180)
    error('atanyxshiftedpar: thoffsetdeg out of range');
end

dth2 = ((dthdeg/2)/180*pi);
thoffset = (thoffsetdeg/180*pi);
CY = (1 - 2^-15)/(2*cos(dth2));
CX = (1 - 2^-15)/(2*sin(dth2));
if(abs(CY) >= 1) NY = ceil(log2(abs(CY)));
else NY = 0;
end
if(abs(CX) >= 1) NX = ceil(log2(abs(CX)));
else NX = 0;
end
KY = CY/2^NY;
KX = CX/2^NX;
THETAADJ = dthdeg/2 - thoffsetdeg;

if THETAADJ >= 180
    THETAADJ = THETAADJ - 360;
elseif THETAADJ < -180
    THETAADJ = THETAADJ + 360;
end

THETAADJ = THETAADJ/180;

return;
```

While applying the function, some general guidelines should be considered as stated below.

At some values of the phase shift, and particularly at phase shift approaching -180, 0 or 180 degrees, the algorithm may become numerically unstable, causing any error, contributed by input signal imperfections or through finite precision arithmetic, to be magnified significantly. Therefore, some care should be taken to avoid error where possible. The detailed error analysis of the algorithm is beyond the scope of this documentation, however, general guidelines are provided.

There are several sources of error in the function:

- error of the supplied signal values due to the finite resolution of the AD conversion
- error contributed by higher order harmonics appearing in the input signals
- computational error of the multiplication due to the finite length of registers
- error of the phase shift  $\Delta\theta$  representation in the finite precision arithmetic and in the values
- error due to differences in signal amplitudes

It should be noted that the function requires both signals to have the same amplitude. To minimize the output error, the amplitude of both signals should be as close to 1.0 as much as possible.

The function has been tested to be reliable at a phase shift in the range of [-165, -15] and [15, 165] degrees for perfectly sinusoidal input signals. Beyond this range, the function operates correctly, however, the output error can be beyond the guaranteed value. In a real application, an error, contributed by an AD conversion and by higher order harmonics of the input signals, should be also taken into account.

**Note:** The function calls the [GFLIB\\_AtanYX\\_F32](#) function. The function may become numerically unstable for a phase shift approaching -180, 0 or 180 degrees. The function accuracy is guaranteed for a phase shift in the range of [-165, -15] and [15, 165] degrees at perfect input signals.

**Caution:** Due to the cyclic character of the [GFLIB\\_AtanYX\\_F16](#), in case the difference between the adjusting angle  $\theta_{adj}$  and the input vector angle is approaching to  $1 - 2^{-15}$  or -1, the [GFLIB\\_AtanYX\\_F16](#) function operates correctly, however the output error might exceed the guaranteed limits.

### Code Example

```
#include "gplib.h"

tFrac32 f32InY;
tFrac32 f32InX;
tFrac32 f32Ang;
GFLIB_ATANYXSHIFTED_T_F32 Param;

void main(void)
{
    // dtheta = 69.33deg, thetaoffset = 10deg
    // CY = (1 - 2^-15)/(2*cos((69.33/2)/180*pi))= 0.60789036201452440
    // CX = (1 - 2^-15)/(2*sin((69.33/2)/180*pi))= 0.87905201358520957
    // NY = 0 (abs(CY) < 1)
    // NX = 0 (abs(CX) < 1)
    // KY = 0.60789/2^0 = 0.60789036201452440
    // KX = 0.87905/2^0 = 0.87905201358520957
    // THETAADJ = 10/180 = 0.0555555555555

    Param.f32Ky = FRAC32(0.60789036201452440);
    Param.f32Kx = FRAC32(0.87905201358520957);
    Param.s32Ny = 0;
    Param.s32Nx = 0;
    Param.f32ThetaAdj = FRAC32(0.0555555555555);

    // theta = 15 deg
    // Y = sin(theta) = 0.2588190
    // X = sin(theta + dtheta) = 0.9951074
    f32InY = FRAC32(0.2588190);
```

```

f32InX = FRAC32(0.9951074);

// f32Ang output should be close to 0x1C34824A
f32Ang = GFLIB\_AtanYXShifted\_F32(f32InY, f32InX, &Param);

// f32Ang output should be close to 0x1C34824A
f32Ang = GFLIB\_AtanYXShifted(f32InY, f32InX, &Param, F32);

// ##### Available only if 32-bit fractional implementation selected
// as default
// #####
// f32Ang output should be close to 0x1C34824A
f32Ang = GFLIB\_AtanYXShifted(f32InY, f32InX, &Param);
}

```

## 2.17.2 Function [GFLIB\\_AtanYXShifted\\_F16](#)

### Declaration

```
tFrac16 GFLIB_AtanYXShifted_F16(tFrac16 f16InY, tFrac16 f16InX,
const GFLIB\_ATANYXSHIFTED\_T\_F16 *pParam);
```

### Arguments

Table 132. [GFLIB\\_AtanYXShifted\\_F16](#) arguments

Type	Name	Direction	Description
<a href="#">tFrac16</a>	f16InY	input	The value of the first signal, assumed to be $\sin(\theta)$ .
<a href="#">tFrac16</a>	f16InX	input	The value of the second signal, assumed to be $\sin(\theta + \Delta\theta)$ .
const <a href="#">GFLIB_ATANYXSHIFTED_T_F16</a> *	pParam	input, output	The parameters for the function.

### Return

The function returns the angle of two sine waves shifted in phase to each other.

### Implementation details

The [GFLIB\\_AtanYXShifted\\_F16](#) function does not directly use the angle offset  $\theta_{\text{Offset}}$  and the phase difference  $\theta$ . The function's parameters, contained in the function parameters structure [GFLIB\\_ATANYXSHIFTED\\_T\\_F16](#), need to be computed by means of the provided Matlab function (see below).

If  $\Delta\theta = \pi/2$  or  $\Delta\theta = -\pi/2$ , then the function is similar to the [GFLIB\\_AtanYX\\_F16](#) function, however, the [GFLIB\\_AtanYX\\_F16](#) function in this case is more effective with regard to execution time and accuracy.

In order to use the function, the following necessary steps need to be completed:

- define  $\Delta\theta$  and  $\theta_{\text{Offset}}$ , the  $\Delta\theta$  shall be known from the input sinusoidal signals, the  $\theta_{\text{Offset}}$  needs to be set arbitrarily
- compute values for the function parameters structure by means of the provided Matlab function
- convert the computed values into integer format and insert them into the C code (see also the C code example)

The function uses the following algorithm for computing the angle:

$$\begin{aligned} b &= \frac{S}{2\cos(\frac{\Delta\theta}{2})} (y + x) \\ a &= \frac{S}{2\sin(\frac{\Delta\theta}{2})} (x - y) \\ \theta &= \tan^{-1}\left(\frac{y}{x}\right) - \left(\frac{\Delta\theta}{2} - \theta_{offset}\right) \end{aligned}$$

Equation GFLIB\_AtanYXShifted\_F16\_Eq1

where:

- x, y are respectively, the f16InX, and f16InY
- $\theta$  is the angle to be computed by the function, see the previous equation
- $\Delta\theta$  is the phase difference between the x, y signals, see the previous equation
- S is a scaling coefficient, S is almost 1, ( $S < 1$ ), see also the explanation below
- a, b intermediate variables
- $\theta_{Offset}$  is the additional phase shift, the computed angle will be  $\theta + \theta_{Offset}$

The scale coefficient S is used to prevent overflow and to assure symmetry around 0 for the entire fractional range. S shall be less than 1.0, but as large as possible. The algorithm implemented in this function uses the value of  $1 - 2^{-15}$ .

The algorithm can be easily justified by proving the trigonometric identity:

$$\tan(\theta + \Delta\theta) = \frac{(y+x)\cos(\frac{\Delta\theta}{2})}{(x-y)\sin(\frac{\Delta\theta}{2})}$$

Equation GFLIB\_AtanYXShifted\_F16\_Eq2

For the purposes of fractional arithmetic, the algorithm is implemented such that additional values are used as shown in the equation below:

$$\begin{aligned} \frac{S}{2\cos(\frac{\Delta\theta}{2})} &= C_y = K_y \cdot 2^{N_y} \\ \frac{S}{2\sin(\frac{\Delta\theta}{2})} &= C_x = K_x \cdot 2^{N_x} \\ \theta_{adj} &= \frac{\Delta\theta}{2} - \theta_{offset} \end{aligned}$$

Equation GFLIB\_AtanYXShifted\_F16\_Eq3

where:

- $C_y, C_x$  are the algorithm coefficients for y and x signals
- $K_y$  is multiplication coefficient of the y signal, represented by the parameters structure member pParam->f16Ky
- $K_x$  is multiplication coefficient of the x signal, represented by the parameters structure member pParam->f16Kx
- $N_y$  is scaling coefficient of the y signal, represented by the parameters structure member pParam->s16Ny
- $N_x$  is scaling coefficient of the x signal, represented by the parameters structure member pParam->s16Nx
- $\theta_{adj}$  is an adjusting angle, represented by the parameters structure member pParam->f16ThetaAdj

The multiplication and scaling coefficients, and the adjusting angle, shall be defined in a parameters structure provided as the function input parameter.

The function initialization parameters can be calculated as shown in the following Matlab code:

```
function [KY, KX, NY, NX, THETAADJ] = atanyxshiftedpar(dthdeg, thoffsetdeg)
// ATANYXSHIFTEDPAR calculation of parameters for atanyxshifted() function
//
// [KY, KX, NY, NX, THETAADJ] = atanyxshiftedpar(dthdeg, thoffsetdeg)
//
// dthdeg = phase shift (delta theta) between sine waves in degrees
// thoffsetdeg = angle offset (theta offset) in degrees
// NY - scaling coefficient of y signal
// NX - scaling coefficient of x signal
// KY - multiplication coefficient of y signal
// KX - multiplication coefficient of x signal
// THETAADJ - adjusting angle in radians, scaled from [-pi, pi) to [-1, 1)

if (dthdeg < -180) || (dthdeg >= 180)
    error('atanyxshiftedpar: dthdeg out of range');
end
if (thoffsetdeg < -180) || (thoffsetdeg >= 180)
    error('atanyxshiftedpar: thoffsetdeg out of range');
end

dth2 = ((dthdeg/2)/180*pi);
thoffset = (thoffsetdeg/180*pi);
CY = (1 - 2^-15)/(2*cos(dth2));
CX = (1 - 2^-15)/(2*sin(dth2));
if(abs(CY) >= 1) NY = ceil(log2(abs(CY)));
else NY = 0;
end
if(abs(CX) >= 1) NX = ceil(log2(abs(CX)));
else NX = 0;
end
KY = CY/2^NY;
KX = CX/2^NX;
THETAADJ = dthdeg/2 - thoffsetdeg;

if THETAADJ >= 180
    THETAADJ = THETAADJ - 360;
elseif THETAADJ < -180
    THETAADJ = THETAADJ + 360;
end

THETAADJ = THETAADJ/180;

return;
```

While applying the function, some general guidelines should be considered as stated below.

At some values of the phase shift, and particularly at phase shift approaching -180, 0 or 180 degrees, the algorithm may become numerically unstable, causing any error, contributed by input signal imperfections or through finite precision arithmetic, to be magnified significantly. Therefore, some care should be taken to avoid error where possible. The detailed error analysis of the algorithm is beyond the scope of this documentation, however, general guidelines are provided.

There are several sources of error in the function:

- error of the supplied signal values due to the finite resolution of the AD conversion
  - error contributed by higher order harmonics appearing in the input signals
  - computational error of the multiplication due to the finite length of registers
  - error of the phase shift  $\Delta\theta$  representation in the finite precision arithmetic and in the values
  - error due to differences in signal amplitudes

It should be noted that the function requires both signals to have the same amplitude. To minimize the output error, the amplitude of both signals should be as close to 1.0 as much as possible.

The function has been tested to be reliable at a phase shift in the range of [-165, -15] and [15, 165] degrees for perfectly sinusoidal input signals. Beyond this range, the function operates correctly, however, the output error can be beyond the guaranteed value. In a real application, an error, contributed by an AD conversion and by higher order harmonics of the input signals, should be also taken into account.

**Note:** The function calls the [GFLIB\\_AtanYX\\_F16](#) function. The function may become numerically unstable for a phase shift approaching -180, 0 or 180 degrees. The function accuracy is guaranteed for a phase shift in the range of [-175, -5] and [5, 175] degrees at perfect input signals. To eliminate the calculation error the function uses the 32-bit internal accumulators.

**Caution:** Due to the cyclic character of the [GFLIB\\_AtanYX\\_F16](#), in case the difference between the adjusting angle  $\theta_{\text{adj}}$  and the input vector angle is approaching to  $1 - 2^{-15}$  or  $-1$ , the [GFLIB\\_AtanYX\\_F16](#) function operates correctly, however the output error might exceed the guaranteed limits.

## Code Example

```

#include "gplib.h"

tFrac16 f16InY;
tFrac16 f16InX;
tFrac16 f16Ang;
GFLIB_ATANYXSHIFTED_T_F16 Param;

void main(void)
{
    // dtheta = 69.33deg, thetaoffset = 10deg
    // CY = (1 - 2^-15)/(2*cos((69.33/2)/180*pi)) = 0.60789036201452440
    // CX = (1 - 2^-15)/(2*sin((69.33/2)/180*pi)) = 0.87905201358520957
    // NY = 0 (abs(CY) < 1)
    // NX = 0 (abs(CX) < 1)
    // KY = 0.60789/2^0 = 0.60789036201452440
    // KX = 0.87905/2^0 = 0.87905201358520957
    // THETAADJ = 10/180 = 0.0555555555555

    Param.f16Ky = FRAC16(0.60789036201452440);
    Param.f16Kx = FRAC16(0.87905201358520957);
    Param.s16Ny = 0;
    Param.s16Nx = 0;
    Param.f16ThetaAdj = FRAC16(0.0555555555555);

    // theta = 15 deg
    // Y = sin(theta) = 0.2588190
    // X = sin(theta + dtheta) = 0.9951074
}

```

```

f16InY = FRAC16(0.2588190);
f16InX = FRAC16(0.9951074);

// f16Ang output should be close to 0x1C34
f16Ang = GFLIB\_AtanYXShifted\_F16(f16InY, f16InX, &Param);

// f16Ang output should be close to 0x1C34
f16Ang = GFLIB\_AtanYXShifted(f16InY, f16InX, &Param, F16);

// ##### Available only if 16-bit fractional implementation selected
// as default
// #####
// f16Ang output should be close to 0x1C34
f16Ang = GFLIB\_AtanYXShifted(f16InY, f16InX, &Param);
}

```

### 2.17.3 Function [GFLIB\\_AtanYXShifted\\_FLT](#)

#### Declaration

```
tFloat GFLIB\_AtanYXShifted\_FLT(tFloat fltInY, tFloat fltInX,
const GFLIB\_ATANYXSHIFTED\_T\_FLT *pParam);
```

#### Arguments

Table 133. [GFLIB\\_AtanYXShifted\\_FLT](#) arguments

Type	Name	Direction	Description
<a href="#">tFloat</a>	fltInY	input	The value of the first signal, assumed to be $\sin(\theta)$ .
<a href="#">tFloat</a>	fltInX	input	The value of the second signal, assumed to be $\sin(\theta + \Delta\theta)$ .
const <a href="#">GFLIB_ATANYXSHIFTED_T_FLT</a> *	pParam	input, output	The parameters for the function.

#### Return

The function returns the angle of two sine waves shifted in phase to each other.

#### Implementation details

At the end of computations, an angle offset  $\theta_{\text{Offset}}$  is added to the computed angle  $\theta$ . The angle offset is an additional parameter which can be used to set the zero of the  $\theta$  axis. If  $\theta_{\text{Offset}}$  is zero, then the angle computed by the function will be exactly  $\theta$ .

The [GFLIB\\_AtanYXShifted\\_FLT](#) function does not directly use the angle offset  $\theta_{\text{Offset}}$  and the phase difference  $\theta$ . The function's parameters, contained in the function parameters structure [GFLIB\\_ATANYXSHIFTED\\_T\\_FLT](#), need to be computed by means of the provided Matlab function (see below).

If  $\Delta\theta = \pi/2$  or  $\Delta\theta = -\pi/2$ , then the function is similar to the [GFLIB\\_AtanYX\\_FLT](#) function, however, the [GFLIB\\_AtanYX\\_FLT](#) function in this case is more effective with regards to execution time and accuracy.

In order to use the function, the following necessary steps need to be completed:

- define  $\Delta\theta$  and  $\theta_{Offset}$ , the  $\Delta\theta$  shall be known from the input sinusoidal signals, the  $\theta_{Offset}$  needs to be set arbitrarily
- compute values for the function parameters structure by means of the provided Matlab function

The function uses the following algorithm for computing the angle:

$$\begin{aligned} b &= \frac{s}{2\cos(\frac{\Delta\theta}{2})} (y + x) \\ a &= \frac{s}{2\sin(\frac{\Delta\theta}{2})} (x - y) \\ \theta &= \tan^{-1}\left(\frac{y}{x}\right) - \left(\frac{\Delta\theta}{2} - \theta_{offset}\right) \end{aligned}$$

Equation GFLIB\_AtanYXShifted\_FLT\_Eq1

where:

- $x, y$  are respectively, the `fltInX`, and `fltInY`
- $\theta$  is the angle to be computed by the function, see the previous equation
- $\Delta\theta$  is the phase difference between the  $x, y$  signals, see the previous equation
- $a, b$  intermediate variables
- $\theta_{Offset}$  is the additional phase shift in radians, the computed angle will be  $\theta + \theta_{Offset}$

The algorithm can be easily justified by proving the trigonometric identity:

$$\tan(\theta + \Delta\theta) = \frac{(y+x)\cos(\frac{\Delta\theta}{2})}{(x-y)\sin(\frac{\Delta\theta}{2})}$$

Equation GFLIB\_AtanYXShifted\_FLT\_Eq2

For the purpose of the calculation, the multiplication coefficients  $K_y$  and  $K_x$  are representing the fractions in the previous equation and are defined as follows:

$$\begin{aligned} \frac{s}{2\cos(\frac{\Delta\theta}{2})} &= C_y = K_y \\ \frac{s}{2\sin(\frac{\Delta\theta}{2})} &= C_x = K_x \\ \theta_{adj} &= \frac{\Delta\theta}{2} - \theta_{offset} \end{aligned}$$

Equation GFLIB\_AtanYXShifted\_FLT\_Eq3

where:

- $K_y$  is multiplication coefficient of the  $y$  signal, represented by the parameters structure member `pParam->fltKy`
- $K_x$  is multiplication coefficient of the  $x$  signal, represented by the parameters structure member `pParam->fltKx`
- $\theta_{adj}$  is an adjusting angle, represented by the parameters structure member `pParam->fltThetaAdj`

The function initialization parameters can be calculated as shown in the following Matlab code:

```
function [KY, KX, THETAADJ] = atanyxshiftedpar(dthdeg, thoffsetdeg)
// ATANYXSHIFTEDPAR calculation of parameters for atanyxshifted() function
//
// [KY, KX, NY, NX, THETAADJ] = atanyxshiftedpar(dthdeg, thoffsetdeg)
```

```
//  
// dthdeg = phase shift (delta theta) between sine waves in degrees  
// thoffsetdeg = angle offset (theta offset) in degrees  
// KY - multiplication coefficient of y signal  
// KX - multiplication coefficient of x signal  
// THETAADJ - adjusting angle in radians, scaled from [-pi, pi) to [-1, 1)  
  
if (dthdeg < -180) || (dthdeg >= 180)  
    error('atanyxshiftedpar: dthdeg out of range');  
end  
if (thoffsetdeg < -180) || (thoffsetdeg >= 180)  
    error('atanyxshiftedpar: thoffsetdeg out of range');  
end  
  
dth2 = ((dthdeg/2)/180*pi);  
thoffset = (thoffsetdeg/180*pi);  
KY = 1/(2*cos(dth2));  
KX = 1/(2*sin(dth2));  
THETAADJ = dthdeg/2 - thoffsetdeg;  
  
if THETAADJ >= 180  
    THETAADJ = THETAADJ - 360;  
elseif THETAADJ < -180  
    THETAADJ = THETAADJ + 360;  
end  
  
THETAADJ = THETAADJ/180;  
  
return;
```

While applying the function, some general guidelines should be considered as stated below.

At some values of the phase shift, and particularly at a phase shift approaching -180, 0, or 180 degrees, the algorithm may become numerically unstable, causing any error, contributed by input signal imperfections or through finite precision arithmetic, to be magnified significantly.

There are several sources of error in the function:

- error of the supplied signal values due to the finite resolution of the AD conversion
- error contributed by higher order harmonics appearing in the input signals
- computational error of arithmetic operations due to the finite length of registers
- error of the phase shift  $\Delta\theta$  representation in the finite precision arithmetic and in the values
- error due to differences in signal amplitudes

To minimize the output error, both signals should have the same amplitude.

**Note:** The function may raise floating-point exceptions (invalid operation, division by zero, overflow, underflow, inexact, input denormal). The floating-point unit must be enabled in CPACR register to prevent NOCP UsageFault exception.

### Code Example

```
#include "gflib.h"  
  
tFloat fltInY;
```

```

tFloat fltInX;
tFloat fltAng;
GFLIB_ATANYXSHIFTED_T_FLT Param;

void main(void)
{
    // dtheta = 69.33deg, thetaoffset = 10deg
    // KY = 1/(2*cos((69.33/2)/180*pi))= 0.6079089139
    // KX = 1/(2*sin((69.33/2)/180*pi))= 0.8790788409

    // THETAADJ = 10*pi/180 = 0.1745329252

    Param.fltKy = (tFloat)(0.6079089139);
    Param.fltKx = (tFloat)(0.8790788409);
    Param.fltThetaAdj = (tFloat)(0.1745329252);

    // theta = 15 deg
    // Y = sin(theta) = 0.2588190
    // X = sin(theta + dtheta) = 0.9951074
    fltInY = (tFloat)(0.2588190);
    fltInX = (tFloat)(0.9951074);

    // Output angle should be 0.69228911 rad
    fltAng = GFLIB_AtanYXShifted_FLT(fltInY, fltInX, &Param);

    // Output angle should be 0.69228911 rad
    fltAng = GFLIB_AtanYXShifted(fltInY, fltInX, &Param, FLT);

    // ##### Available only if single precision floating point
    // implementation selected as default
    // #####
    // Output angle should be 0.69228911 rad
    fltAng = GFLIB_AtanYXShifted(fltInY, fltInX, &Param);
}

```

## 2.18 Function GFLIB\_ControllerPIDpAW

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

### Description

A PID controller attempts to correct the error between a measured process variable and a desired set-point by calculating and then outputting a corrective action that can adjust the process accordingly. The GFLIB\_ControllerPIDpAW function calculates the discrete-time approximation of the Proportional-Integral-Derivative (PID) algorithm according to the following equation:

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

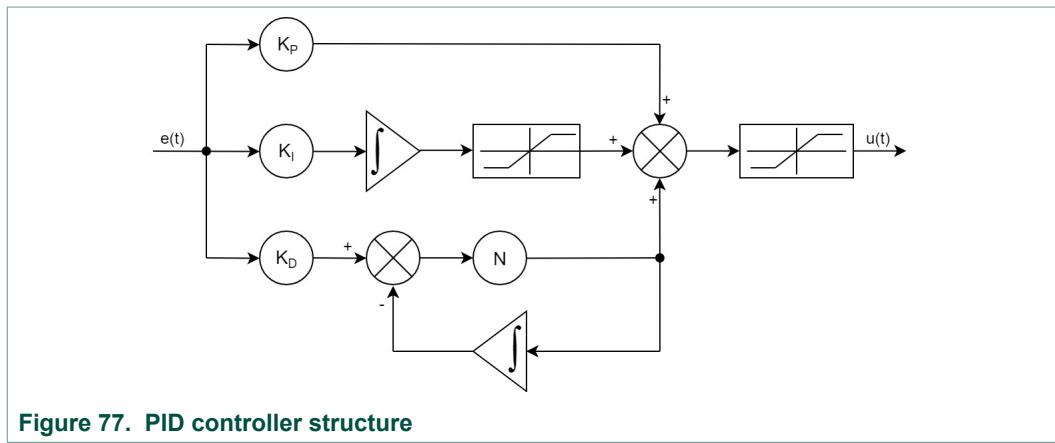
Equation GFLIB\_ControllerPIDpAW\_Eq1

where

- $e(t)$  - input error in the continuous time domain

- $u(t)$  - controller output in the continuous time domain
- $K_P$  - proportional gain
- $K_I$  - integral gain
- $K_D$  - derivative gain

The PID algorithm is implemented in the parallel (non-interacting) form, allowing the user to define the P, I and D parameters independently without interaction. The controller output is limited and the limit values (UpperLimit and LowerLimit) are defined by the user. The PID controller algorithm also returns a limitation flag. This flag ( $u16LimitFlag$ ) is a member of the structure of the PID controller parameters. If the PID controller output reaches the upper or lower limit then  $u16LimitFlag = 1$ , otherwise  $u16LimitFlag = 0$  (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. In the `GFLIB_ControllerPIDpAW` function is implemented filtration of derivative term. Structure of continuous-time equivalent of the implemented PID controller is shown in figure [Figure 77](#).



**Figure 77. PID controller structure**

This system can be described using the Laplace transformation as follows:

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

Equation `GFLIB_ControllerPIDpAW_Eq2`

The proportional part of equation [GFLIB\\_ControllerPIDpAW\\_Eq2](#) is transformed into the discrete-time domain simply as:

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

Equation `GFLIB_ControllerPIDpAW_Eq3`

Transforming the integral part of equation [GFLIB\\_ControllerPIDpAW\\_Eq2](#) into a discrete-time domain using the Bilinear method, also known as trapezoidal approximation, leads to the following equation:

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

Equation `GFLIB_ControllerPIDpAW_Eq4`

where  $T_s$  [sec] is the sampling time.

The derivative part of equation [GFLIB\\_ControllerPIDpAW\\_Eq2](#) is transformed into discrete-time domain using impulse invariance method as:

$$u_D(k) = \frac{K_D}{T_s} \cdot (1 - e^{-T_s N}) \cdot [e(k) - e(k-1)] + e^{-T_s N} \cdot u_D(k-1)$$

Equation GFLIB\_ControllerPIDpAW\_Eq5

**Note:** The input pointer must contain a valid address otherwise a fault may occur (MemManage, BusFault, UsageFault, HardFault).

### Re-entrancy

The function is re-entrant for a different pCtrl.

#### 2.18.1 Function GFLIB\_ControllerPIDpAW\_F32

##### Declaration

```
tFrac32 GFLIB_ControllerPIDpAW_F32(tFrac32 f32InErr,
GFLIB_CONTROLLER_PID_P_AW_T_F32 *const pParam);
```

##### Arguments

**Table 134. GFLIB\_ControllerPIDpAW\_F32 arguments**

Type	Name	Direction	Description
tFrac32	f32InErr	input	Input error signal to the controller is a 32-bit number normalized between [-1, 1).
GFLIB_CONTROLLER_PID_P_AW_T_F32 *const	pParam	input, output	Pointer to the controller parameters structure.

##### Return

The function returns a 32-bit value in format 1.31, representing the signal to be applied to the controlled system so that the input error is forced to zero.

##### Implementation details

In order to implement the discrete equation of the controller on the fixed point arithmetic platform, the maximal values (scales) of input and output signals have to be known a priori. This is essential for correct casting of the physical signal values into fixed point values:

- $E^{MAX}$  - maximal value of the controller input error signal
- $U^{MAX}$  - maximal value of the controller output signal

The fractional representation of both input and output signals, normalized between [-1, 1), is obtained as follows:

$$e_f(k) = \frac{e(k)}{E^{MAX}}$$

$$u_f(k) = \frac{u(k)}{U^{MAX}}$$

Equation GFLIB\_ControllerPIDpAW\_F32\_Eq1

Applying such scaling (normalization) on the proportional term of equation [GFLIB\\_ControllerPIDpAW\\_eq3](#) results in:

$$u_{Pf}(k) = K_{P_{SC}} \cdot e_f(k) \quad \text{where} \quad K_{P_{SC}} = K_P \cdot \frac{E^{MAX}}{U^{MAX}}$$

Equation GFLIB\_ControllerPIDpAW\_F32\_Eq2

where  $K_{P_{SC}}$  is the proportional gain parameter considering input/output scaling.

Analogically, scaling the integral term of equation [GFLIB\\_ControllerPIDpAW\\_eq4](#) results in:

$$u_{If}(k) = K_{I_{SC}} \cdot [e_f(k) + e_f(k-1)] + u_{If}(k-1) \quad \text{where} \quad K_{I_{SC}} = \frac{K_I T_s}{2} \cdot \frac{E^{MAX}}{U^{MAX}}$$

Equation GFLIB\_ControllerPIDpAW\_F32\_Eq3

where  $K_{I_{SC}}$  is the integral gain parameter considering input/output scaling.

And finally, scaling the derivative term, equation [GFLIB\\_ControllerPIDpAW\\_eq5](#) results in:

$$u_{If}(k) = K_{D_{SC}} \cdot [e_f(k) - e_f(k-1)] + e^{T_s N} \cdot u_{Df}(k-1) \quad \text{where} \quad K_{D_{SC}} = \frac{K_D}{T_s} \cdot (1 - e^{T_s N}) \cdot \frac{E^{MAX}}{U^{MAX}}$$

Equation GFLIB\_ControllerPIDpAW\_F32\_Eq4

where  $K_{D_{SC}}$  is the derivative gain parameter considering input/output scaling.

The problem is however, that either of the gain parameters  $K_{P_{SC}}$ ,  $K_{I_{SC}}$  or  $K_{D_{SC}}$  can be out of the  $[-1, 1]$  range, hence can not be directly interpreted as fractional values. To overcome this, it is necessary to scale these gain parameters using the shift values as follows:

$$\begin{aligned} f32PropGain &= K_{P_{SC}} \cdot 2^{s16PropGainShift} && \text{where } s16PropGainShift = \lceil \log_2 K_{P_{SC}} \rceil \\ f32IntegGain &= K_{I_{SC}} \cdot 2^{s16IntegGainShift} && \text{where } s16IntegGainShift = \lceil \log_2 K_{I_{SC}} \rceil \\ f32DerivGain &= K_{D_{SC}} \cdot 2^{s16DerivGainShift} && \text{where } s16DerivGainShift = \lceil \log_2 K_{D_{SC}} \rceil \end{aligned}$$

Equation GFLIB\_ControllerPIDpAW\_F32\_Eq5

where

- f32PropGain - is the scaled value of proportional gain  $[-1, 1]$
- s16PropGainShift - is the scaling shift for proportional gain  $[-31, 31]$
- f32IntegGain - is the scaled value of integral gain  $[-1, 1]$
- s16IntegGainShift - is the scaling shift for integral gain  $[-31, 31]$
- f32DerivGain - is the scaled value of derivative gain  $[-1, 1]$
- s16DerivGainShift - is the scaling shift for derivative gain  $[-31, 31]$

Filtration coefficient for the derivative term given by [GFLIB\\_ControllerPIDpAW\\_F32\\_Eq6](#) does not need scaling since  $T_s$  and  $N$  are positive numbers.

$$f32FiltCoef = e^{T_s N}$$

Equation GFLIB\_ControllerPIDpAW\_F32\_Eq6

The sum of the scaled proportional, integral and derivative terms gives a complete equation of the controller:

$$u_f(k) = K_{P\_SC} \cdot e_f(k) + K_{I\_SC} \cdot [e_f(k) + e_f(k-1)] + u_{If}(k-1) + K_{D\_SC} \cdot [e_f(k) - e_f(k-1)] + K_F \cdot u_{Df}(k-1)$$

Equation GFLIB\_ControllerPIDpAW\_F32\_Eq7

The output signal limitation is implemented in this controller. The actual output  $u(k)$  is bounded not to exceed the given limit values  $f32UpperLimit$ ,  $f32LowerLimit$ . This is due to either the bounded power of the actuator or to the physical constraints of the plant.

$$u_f(k) = \begin{cases} f32UpperLimit & \text{if } u_f(k) \geq f32UpperLimit \\ u_f(k) & \text{if } f32LowerLimit < u_f(k) < f32UpperLimit \\ f32LowerLimit & \text{if } u_f(k) \leq f32LowerLimit \end{cases}$$

Equation GFLIB\_ControllerPIDpAW\_F32\_Eq8

When the bounds are exceeded the non-linear 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.

All controller parameters and states can be reset during declaration using the [GFLIB\\_CONTROLLER\\_PID\\_P\\_AW\\_DEFAULT\\_F32](#) macro.

### Code Example

```
#include "gplib.h"

tFrac32 f32InErr;
tFrac32 f32Output;

GFLIB_CONTROLLER_PID_P_AW_T_F32 trMyPID = GFLIB_CONTROLLER_PID_P_AW_DEFAULT_F32;

void main(void)
{
    tFrac32 f32ControllerPIDpAWOut;

    // Set the input error
    f32InErr = FRAC32(0.25);

    // Initialize the controller parameters
    trMyPID.f32PropGain      = FRAC32(0.1);
    trMyPID.f32IntegGain     = FRAC32(0.2);
    trMyPID.f32DerivGain     = FRAC32(0.3);
    trMyPID.f32FiltCoef      = FRAC32(0.4);
    trMyPID.s16PropGainShift = 1;
    trMyPID.s16IntegGainShift = 1;
    trMyPID.s16DerivGainShift = 1;
    trMyPID.f32UpperLimit    = FRAC32(1.0);
    trMyPID.f32LowerLimit    = FRAC32(-1.0);

    // Clear the state variables
    // Alternative 1: API call with postfix
}
```

```
// (only one alternative shall be used).
GFLIB_ControllerPIDpAWInit_F32(&trMyPID);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
GFLIB_ControllerPIDpAWInit(&trMyPID, F32);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 32-bit fractional implementation is selected as default.
GFLIB_ControllerPIDpAWInit(&trMyPID);

// Initialize the state variables to predefined values
f32ControllerPIDpAWOut = FRAC32(0.5);
// Alternative 1: API call with postfix
// (only one alternative shall be used).
GFLIB_ControllerPIDpAWSetState_F32(f32ControllerPIDpAWOut, &trMyPID);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
GFLIB_ControllerPIDpAWSetState(f32ControllerPIDpAWOut, &trMyPID, F32);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 32-bit fractional implementation is selected as default.
GFLIB_ControllerPIDpAWSetState(f32ControllerPIDpAWOut, &trMyPID);
}

// Periodical function or interrupt - control loop
void ControlLoop(void)
{
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    f32Output = GFLIB_ControllerPIDpAW_F32(f32InErr, &trMyPID);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    f32Output = GFLIB_ControllerPIDpAW(f32InErr, &trMyPID, F32);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 32-bit fractional implementation is selected as default.
    f32Output = GFLIB_ControllerPIDpAW(f32InErr, &trMyPID);
}
```

## 2.18.2 Function **GFLIB\_ControllerPIDpAW\_F16**

### Declaration

```
tFrac16 GFLIB_ControllerPIDpAW_F16(tFrac16 f16InErr,
GFLIB_CONTROLLER_PID_P_AW_T_F16 *const pParam);
```

**Arguments****Table 135. [GFLIB\\_ControllerPIDpAW\\_F16](#) arguments**

Type	Name	Direction	Description
tFrac16	f16InErr	input	Input error signal to the controller is a 16-bit number normalized between [-1, 1).
<a href="#">GFLIB_CONTROLLER_PID_P_AW_T_F16</a> *const	pParam	input, output	Pointer to the controller parameters structure.

**Return**

The function returns a 16-bit value in format 1.15, representing the signal to be applied to the controlled system so that the input error is forced to zero.

**Implementation details**

In order to implement the discrete equation of the controller on the fixed point arithmetic platform, the maximal values (scales) of input and output signals have to be known a priori. This is essential for correct casting of the physical signal values into fixed point values:

- $E^{MAX}$  - maximal value of the controller input error signal
- $U^{MAX}$  - maximal value of the controller output signal

The fractional representation of both input and output signals, normalized between [-1, 1), is obtained as follows:

$$e_f(k) = \frac{e(k)}{E^{MAX}}$$

$$u_f(k) = \frac{u(k)}{U^{MAX}}$$

Equation GFLIB\_ControllerPIDpAW\_F16\_Eq1

Applying such scaling (normalization) on the proportional term of equation [GFLIB\\_ControllerPIDpAW\\_eq3](#) results in:

$$u_{pf}(k) = K_{P_{SC}} \cdot e_f(k) \quad \text{where} \quad K_{P_{SC}} = K_p \cdot \frac{E^{MAX}}{U^{MAX}}$$

Equation GFLIB\_ControllerPIDpAW\_F16\_Eq2

where  $K_{P_{SC}}$  is the proportional gain parameter considering input/output scaling.

Analogically, scaling the integral term of equation [GFLIB\\_ControllerPIDpAW\\_eq4](#) results in:

$$u_{if}(k) = K_{I_{SC}} \cdot [e_f(k) + e_f(k-1)] + u_{if}(k-1) \quad \text{where} \quad K_{I_{SC}} = \frac{K_i T_s}{2} \cdot \frac{E^{MAX}}{U^{MAX}}$$

Equation GFLIB\_ControllerPIDpAW\_F16\_Eq3

where  $K_{I_{SC}}$  is the integral gain parameter considering input/output scaling.

And finally, scaling the derivative term, equation [GFLIB\\_ControllerPIDpAW\\_eq5](#) results in:

$$u_{If}(k) = K_{D\_SC} \cdot [e_f(k) - e_f(k-1)] + e^{T_s N} \cdot u_{Df}(k-1) \quad \text{where } K_{D\_SC} = \frac{K_D}{T_s} \cdot (1 - e^{T_s N}) \cdot \frac{E^{MAX}}{U^{MAX}}$$

Equation [GFLIB\\_ControllerPIDpAW\\_F16\\_Eq4](#)

where  $K_{D\_SC}$  is the derivative gain parameter considering input/output scaling.

The problem is however, that either of the gain parameters  $K_{P\_SC}$ ,  $K_{I\_SC}$  or  $K_{D\_SC}$  can be out of the [-1, 1] range, hence can not be directly interpreted as fractional values. To overcome this, it is necessary to scale these gain parameters using the shift values as follows:

$$\begin{aligned} f16PropGain &= K_{P\_SC} \cdot 2^{s16PropGainShift} \quad \text{where } s16PropGainShift = \lceil \log_2 K_{P\_SC} \rceil \\ f16IntegGain &= K_{I\_SC} \cdot 2^{s16IntegGainShift} \quad \text{where } s16IntegGainShift = \lceil \log_2 K_{I\_SC} \rceil \\ f16DerivGain &= K_{D\_SC} \cdot 2^{s16DerivGainShift} \quad \text{where } s16DerivGainShift = \lceil \log_2 K_{D\_SC} \rceil \end{aligned}$$

Equation [GFLIB\\_ControllerPIDpAW\\_F16\\_Eq5](#)

where

- f16PropGain - is the scaled value of proportional gain [-1, 1]
- s16PropGainShift - is the scaling shift for proportional gain [-31, 31]
- f16IntegGain - is the scaled value of integral gain [-1, 1]
- s16IntegGainShift - is the scaling shift for integral gain [-31, 31]
- f16DerivGain - is the scaled value of derivative gain [-1, 1]
- s16DerivGainShift - is the scaling shift for derivative gain [-31, 31]

Filtration coefficient for the derivative term given by [GFLIB\\_ControllerPIDpAW\\_F16\\_Eq6](#) does not need scaling since  $T_s$  and  $N$  are positive numbers.

$$f16FiltCof = e^{T_s N}$$

Equation [GFLIB\\_ControllerPIDpAW\\_F16\\_Eq6](#)

The sum of the scaled proportional, integral and derivative terms gives a complete equation of the controller:

$$u_f(k) = K_{P\_SC} \cdot e_f(k) + K_{I\_SC} \cdot [e_f(k) + e_f(k-1)] + u_{If}(k-1) + K_{D\_SC} \cdot [e_f(k) - e_f(k-1)] + K_F \cdot u_{Df}(k-1)$$

Equation [GFLIB\\_ControllerPIDpAW\\_F16\\_Eq7](#)

The output signal limitation is implemented in this controller. The actual output  $u(k)$  is bounded not to exceed the given limit values  $f16UpperLimit$ ,  $f16LowerLimit$ . This is due to either the bounded power of the actuator or to the physical constraints of the plant.

$$u_f(k) = \begin{cases} f16UpperLimit & \text{if } u_f(k) \geq f16UpperLimit \\ u_f(k) & \text{if } f16LowerLimit < u_f(k) < f16UpperLimit \\ f16LowerLimit & \text{if } u_f(k) \leq f16LowerLimit \end{cases}$$

Equation [GFLIB\\_ControllerPIDpAW\\_F16\\_Eq8](#)

When the bounds are exceeded the non-linear 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.

All controller parameters and states can be reset during declaration using the [GFLIB\\_CONTROLLER\\_PID\\_P\\_AW\\_DEFAULT\\_F16](#) macro.

### Code Example

```
#include "gflib.h"

tFrac16 f16InErr;
tFrac16 f16Output;

GFLIB_CONTROLLER_PID_P_AW_T_F16 trMyPID = GFLIB\_CONTROLLER\_PID\_P\_AW\_DEFAULT\_F16;

void main(void)
{
    tFrac16 f16ControllerPIDpAWOut;

    // Set the input error
    f16InErr = FRAC16(0.25);

    // Initialize the controller parameters
    trMyPID.f16PropGain      = FRAC16(0.1);
    trMyPID.f16IntegGain     = FRAC16(0.2);
    trMyPID.f16DerivGain     = FRAC16(0.3);
    trMyPID.f16FiltCoef      = FRAC16(0.4);
    trMyPID.s16PropGainShift = 1;
    trMyPID.s16IntegGainShift = 1;
    trMyPID.s16DerivGainShift = 1;
    trMyPID.f16UpperLimit    = FRAC16(1.0);
    trMyPID.f16LowerLimit    = FRAC16(-1.0);

    // Clear the state variables
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    GFLIB\_ControllerPIDpAWInit\_F16(&trMyPID);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    GFLIB_ControllerPIDpAWInit(&trMyPID, F16);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 16-bit fractional implementation is selected as default.
    GFLIB_ControllerPIDpAWInit(&trMyPID);

    // Initialize the state variables to predefined values
    f16ControllerPIDpAWOut = FRAC16(0.5);
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    GFLIB\_ControllerPIDpAWSetState\_F16(f16ControllerPIDpAWOut, &trMyPID);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    GFLIB_ControllerPIDpAWSetState(f16ControllerPIDpAWOut, &trMyPID, F16);
```

```

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 16-bit fractional implementation is selected as default.
GFLIB_ControllerPIDpAWSetState(f16ControllerPIDpAWOut, &trMyPID);
}

// Periodical function or interrupt - control loop
void ControlLoop(void)
{
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    f16Output = GFLIB_ControllerPIDpAW_F16(f16InErr, &trMyPID);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    f16Output = GFLIB_ControllerPIDpAW(f16InErr, &trMyPID, F16);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 16-bit fractional implementation is selected as default.
    f16Output = GFLIB_ControllerPIDpAW(f16InErr, &trMyPID);
}

```

### 2.18.3 Function GFLIB\_ControllerPIDpAW\_FLT

#### Declaration

```
tFloat GFLIB_ControllerPIDpAW_FLT(tFloat fltInErr,
GFLIB_CONTROLLER_PID_P_AW_T_FLT *const pParam);
```

#### Arguments

Table 136. GFLIB\_ControllerPIDpAW\_FLT arguments

Type	Name	Direction	Description
<b>tFloat</b>	fltInErr	input	Input error signal to the controller in single precision floating format.
<b>GFLIB_CONTROLLER_PID_P_AW_T_FLT</b> *const	pParam	input, output	Pointer to the controller parameters structure.

#### Return

The function returns a single precision floating point value, representing the signal to be applied to the controlled system so that the input error is forced to zero.

#### Implementation details

Output of the proportional, integral and derivative terms are computed according to equations [GFLIB\\_ControllerPIDpAW\\_eq3](#), [GFLIB\\_ControllerPIDpAW\\_eq4](#) and [GFLIB\\_ControllerPIDpAW\\_eq5](#) respectively. To ensure that behavior of GFLIB\_ControllerPIDpAW\_FLT will be consistent with continuous-time controller shown in [GFLIB\\_ControllerPIDpAW\\_fig1](#), controller parameters must be calculated by following equations:

$$\begin{aligned} \text{fltPropGain} &= K_p \\ \text{fltIntegGain} &= \frac{K_p T_s}{2} \\ \text{fltDerivGain} &= \frac{K_p}{T_s} \cdot (1 - e^{-T_s N}) \\ \text{fltFiltCoef} &= e^{-T_s N} \end{aligned}$$

Equation GFLIB\_ControllerPIDpAW\_FLT\_Eq1

The sum of the proportional, integral and derivative terms gives a complete equation of the controller:

$$u(k) = K_p \cdot e(k) + \frac{K_p T_s}{2} \cdot [e(k) + e(k-1)] + u_l(k-1) + \frac{K_p}{T_s} \cdot (1 - e^{-T_s N}) \cdot [e(k) - e(k-1)] + e^{-T_s N} \cdot u_d(k-1)$$

Equation GFLIB\_ControllerPIDpAW\_FLT\_Eq2

The output signal limitation is implemented in this controller. The actual output  $u(k)$  is bounded not to exceed the given limit values  $\text{fltUpperLimit}$ ,  $\text{fltLowerLimit}$ . This is due to either the bounded power of the actuator or to the physical constraints of the plant.

$$u(k) = \begin{cases} \text{fltUpperLimit} & \text{if } u(k) \geq \text{fltUpperLimit} \\ u(k) & \text{if } \text{fltLowerLimit} < u(k) < \text{fltUpperLimit} \\ \text{fltLowerLimit} & \text{if } u(k) \leq \text{fltLowerLimit} \end{cases}$$

Equation GFLIB\_ControllerPIDpAW\_FLT\_Eq3

When the bounds are exceeded, the non-linear saturation characteristic will take effect and influence the dynamic behavior. The described limitation is implemented 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.

All controller parameters and states can be reset during declaration using the [GFLIB\\_CONTROLLER\\_PID\\_P\\_AW\\_DEFAULT\\_FLT](#) macro.

### Code Example

```
#include "gplib.h"

tFloat fltInErr;
tFloat fltOutput;

GFLIB_CONTROLLER_PID_P_AW_T_FLT trMyPID = GFLIB_CONTROLLER_PID_P_AW_DEFAULT_FLT;

void main(void)
{
    tFloat fltControllerPIDpAWOut;

    // Set the input error
    fltInErr = 0.25F;

    // Initialize the controller parameters
    trMyPID.fltPropGain      = 0.1F;
    trMyPID.fltIntegGain     = 0.2F;
    trMyPID.fltDerivGain     = 0.3F;
    trMyPID.fltFiltCoef      = 0.4F;
    trMyPID.fltUpperLimit    = 1.0F;
```

```
trMyPID.fltLowerLimit      = -1.0F;

// Clear the state variables
// Alternative 1: API call with postfix
// (only one alternative shall be used).
GFLIB_ControllerPIDpAWInit_FLT(&trMyPID);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
GFLIB_ControllerPIDpAWInit(&trMyPID, FLT);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if single precision floating-point implementation is selected
// as default.
GFLIB_ControllerPIDpAWInit(&trMyPID);

// Initialize the state variables to predefined values
fltControllerPIDpAWOut = 0.5F;
// Alternative 1: API call with postfix
// (only one alternative shall be used).
GFLIB_ControllerPIDpAWSetState_FLT(fltControllerPIDpAWOut, &trMyPID);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
GFLIB_ControllerPIDpAWSetState(fltControllerPIDpAWOut, &trMyPID, FLT);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if single precision floating-point implementation is selected
// as default.
GFLIB_ControllerPIDpAWSetState(fltControllerPIDpAWOut, &trMyPID);
}

// Periodical function or interrupt - control loop
void ControlLoop(void)
{
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    fltOutput = GFLIB_ControllerPIDpAW_FLT(fltInErr, &trMyPID);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    fltOutput = GFLIB_ControllerPIDpAW(fltInErr, &trMyPID, FLT);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if single precision floating-point implementation is selected
    // as default.
    fltOutput = GFLIB_ControllerPIDpAW(fltInErr, &trMyPID);
}
```

## 2.18.4 Function GFLIB\_ControllerPIDpAWInit

### Description

This function clears the GFLIB\_ControllerPIDpAW state variables.

**Note:** The input/output pointer must contain a valid address, otherwise a fault may occur (MemManage, BusFault, UsageFault, HardFault).

### Re-entrancy

The function is re-entrant for a different pCtrl.

#### 2.18.4.1 Function `GFLIB_ControllerPIDpAWInit_F32`

##### Declaration

```
void
GFLIB_ControllerPIDpAWInit_F32 (GFLIB\_CONTROLLER\_PID\_P\_AW\_T\_F32
*const pParam);
```

##### Arguments

**Table 137. GFLIB\_ControllerPIDpAWInit\_F32 arguments**

Type	Name	Direction	Description
<a href="#">GFLIB_CONTROLLER_PID_P_AW_T_F32</a> *const	pParam	input, output	Pointer to the structure with GFLIB_ControllerPIDpAW state.

##### Code Example

```
#include "gflib.h"

tFrac32 f32InErr;
tFrac32 f32Output;

GFLIB\_CONTROLLER\_PID\_P\_AW\_T\_F32 trMyPID = GFLIB\_CONTROLLER\_PID\_P\_AW\_DEFAULT\_F32;

void main(void)
{
    tFrac32 f32ControllerPIDpAWOut;

    // Set the input error
    f32InErr = FRAC32(0.25);

    // Initialize the controller parameters
    trMyPID.f32PropGain      = FRAC32(0.1);
    trMyPID.f32IntegGain     = FRAC32(0.2);
    trMyPID.f32DerivGain     = FRAC32(0.3);
    trMyPID.f32FiltCoef      = FRAC32(0.4);
    trMyPID.s16PropGainShift = 1;
    trMyPID.s16IntegGainShift = 1;
    trMyPID.s16DerivGainShift = 1;
    trMyPID.f32UpperLimit    = FRAC32(1.0);
    trMyPID.f32LowerLimit    = FRAC32(-1.0);

    // Clear the state variables
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    GFLIB_ControllerPIDpAWInit_F32(&trMyPID);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
}
```

```

GFLIB_ControllerPIDpAWInit(&trMyPID, F32);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 32-bit fractional implementation is selected as default.
GFLIB_ControllerPIDpAWInit(&trMyPID);

// Initialize the state variables to predefined values
f32ControllerPIDpAWOut = FRAC32(0.5);
// Alternative 1: API call with postfix
// (only one alternative shall be used).
GFLIB\_ControllerPIDpAWSetState\_F32(f32ControllerPIDpAWOut, &trMyPID);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
GFLIB_ControllerPIDpAWSetState(f32ControllerPIDpAWOut, &trMyPID, F32);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 32-bit fractional implementation is selected as default.
GFLIB_ControllerPIDpAWSetState(f32ControllerPIDpAWOut, &trMyPID);
}

// Periodical function or interrupt - control loop
void ControlLoop(void)
{
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    f32Output = GFLIB\_ControllerPIDpAW\_F32(f32InErr, &trMyPID);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    f32Output = GFLIB_ControllerPIDpAW(f32InErr, &trMyPID, F32);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 32-bit fractional implementation is selected as default.
    f32Output = GFLIB_ControllerPIDpAW(f32InErr, &trMyPID);
}

```

#### 2.18.4.2 Function GFLIB\_ControllerPIDpAWInit\_F16

##### Declaration

```

void
GFLIB_ControllerPIDpAWInit_F16(GFLIB\_CONTROLLER\_PID\_P\_AW\_T\_F16
*const pParam);

```

##### Arguments

**Table 138. GFLIB\_ControllerPIDpAWInit\_F16 arguments**

Type	Name	Direction	Description
<a href="#">GFLIB_CONTROLLER_PID_P_AW_T_F16</a> *const	pParam	input, output	Pointer to the structure with GFLIB_ControllerPIDpAW state.

## Code Example

```
#include "gplib.h"

tFrac16 f16InErr;
tFrac16 f16Output;

GFLIB_CONTROLLER_PID_P_AW_T_F16 trMyPID = GFLIB_CONTROLLER_PID_P_AW_DEFAULT_F16;

void main(void)
{
    tFrac16 f16ControllerPIDpAWOut;

    // Set the input error
    f16InErr = FRAC16(0.25);

    // Initialize the controller parameters
    trMyPID.f16PropGain      = FRAC16(0.1);
    trMyPID.f16IntegGain     = FRAC16(0.2);
    trMyPID.f16DerivGain     = FRAC16(0.3);
    trMyPID.f16FiltCoef      = FRAC16(0.4);
    trMyPID.s16PropGainShift = 1;
    trMyPID.s16IntegGainShift = 1;
    trMyPID.s16DerivGainShift = 1;
    trMyPID.f16UpperLimit   = FRAC16(1.0);
    trMyPID.f16LowerLimit   = FRAC16(-1.0);

    // Clear the state variables
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    GFLIB_ControllerPIDpAWInit_F16(&trMyPID);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    GFLIB_ControllerPIDpAWInit(&trMyPID, F16);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 16-bit fractional implementation is selected as default.
    GFLIB_ControllerPIDpAWInit(&trMyPID);

    // Initialize the state variables to predefined values
    f16ControllerPIDpAWOut = FRAC16(0.5);
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    GFLIB_ControllerPIDpAWSetState_F16(f16ControllerPIDpAWOut, &trMyPID);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    GFLIB_ControllerPIDpAWSetState(f16ControllerPIDpAWOut, &trMyPID, F16);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 16-bit fractional implementation is selected as default.
    GFLIB_ControllerPIDpAWSetState(f16ControllerPIDpAWOut, &trMyPID);
}

// Periodical function or interrupt - control loop
```

```

void ControlLoop (void)
{
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    f16Output = GFLIB\_ControllerPIDpAW\_F16(f16InErr, &trMyPID);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    f16Output = GFLIB_ControllerPIDpAW(f16InErr, &trMyPID, F16);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 16-bit fractional implementation is selected as default.
    f16Output = GFLIB_ControllerPIDpAW(f16InErr, &trMyPID);
}

```

#### 2.18.4.3 Function [GFLIB\\_ControllerPIDpAWInit\\_FLT](#)

##### Declaration

```

void
GFLIB_ControllerPIDpAWInit_FLT (GFLIB\_CONTROLLER\_PID\_P\_AW\_T\_FLT
*const pParam);

```

##### Arguments

**Table 139. [GFLIB\\_ControllerPIDpAWInit\\_FLT](#) arguments**

Type	Name	Direction	Description
<a href="#">GFLIB_CONTROLLER_PID_P_AW_T_FLT</a> *const	pParam	input, output	Pointer to the structure with <a href="#">GFLIB_ControllerPIDpAW</a> state.

##### Code Example

```

#include "gflib.h"

tFloat fltInErr;
tFloat fltOutput;

GFLIB\_CONTROLLER\_PID\_P\_AW\_T\_FLT trMyPID = GFLIB\_CONTROLLER\_PID\_P\_AW\_DEFAULT\_FLT;

void main (void)
{
    tFloat fltControllerPIDpAWOut;

    // Set the input error
    fltInErr = 0.25F;

    // Initialize the controller parameters
    trMyPID.fltPropGain      = 0.1F;
    trMyPID.fltIntegGain     = 0.2F;
    trMyPID.fltDerivGain     = 0.3F;
    trMyPID.fltFiltCoef      = 0.4F;
    trMyPID.fltUpperLimit    = 1.0F;
    trMyPID.fltLowerLimit    = -1.0F;

    // Clear the state variables

```

```
// Alternative 1: API call with postfix
// (only one alternative shall be used).
GFLIB_ControllerPIDpAWInit_FLT(&trMyPID);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
GFLIB_ControllerPIDpAWInit(&trMyPID, FLT);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if single precision floating-point implementation is selected
// as default.
GFLIB_ControllerPIDpAWInit(&trMyPID);

// Initialize the state variables to predefined values
fltControllerPIDpAWOut = 0.5F;
// Alternative 1: API call with postfix
// (only one alternative shall be used).
GFLIB\_ControllerPIDpAWSetState\_FLT(fltControllerPIDpAWOut, &trMyPID);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
GFLIB_ControllerPIDpAWSetState(fltControllerPIDpAWOut, &trMyPID, FLT);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if single precision floating-point implementation is selected
// as default.
GFLIB_ControllerPIDpAWSetState(fltControllerPIDpAWOut, &trMyPID);
}

// Periodical function or interrupt - control loop
void ControlLoop(void)
{
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    fltOutput = GFLIB\_ControllerPIDpAW\_FLT(fltInErr, &trMyPID);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    fltOutput = GFLIB_ControllerPIDpAW(fltInErr, &trMyPID, FLT);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if single precision floating-point implementation is selected
    // as default.
    fltOutput = GFLIB_ControllerPIDpAW(fltInErr, &trMyPID);
}
```

## 2.18.5 Function [GFLIB\\_ControllerPIDpAWSetState](#)

### Description

This function initializes the GFLIB\_ControllerPIDpAW state variables to achieve the required output values.

**Note:** The input/output pointer must contain a valid address, otherwise a fault may occur (MemManage, BusFault, UsageFault, HardFault).

### Re-entrancy

The function is re-entrant for a different pCtrl.

#### 2.18.5.1 Function `GFLIB_ControllerPIDpAWSetState_F32`

##### Declaration

```
void GFLIB_ControllerPIDpAWSetState_F32 (tFrac32
f32ControllerPIDpAWOut, GFLIB_CONTROLLER_PID_P_AW_T_F32 *const
pParam);
```

##### Arguments

Table 140. `GFLIB_ControllerPIDpAWSetState_F32` arguments

Type	Name	Direction	Description
<u>tFrac32</u>	f32ControllerPIDpAWOut <input/>		Required output of the <code>GFLIB_ControllerPIDpAW</code> .
<u>GFLIB_CONTROLLER_PID_P_AW_T_F32</u> *const	pParam	input, output	Pointer to the structure with <code>GFLIB_ControllerPIDpAW</code> state.

##### Code Example

```
#include "gflib.h"

tFrac32 f32InErr;
tFrac32 f32Output;

GFLIB_CONTROLLER_PID_P_AW_T_F32 trMyPID = GFLIB_CONTROLLER_PID_P_AW_DEFAULT_F32;

void main(void)
{
    tFrac32 f32ControllerPIDpAWOut;

    // Set the input error
    f32InErr = FRAC32(0.25);

    // Initialize the controller parameters
    trMyPID.f32PropGain      = FRAC32(0.1);
    trMyPID.f32IntegGain     = FRAC32(0.2);
    trMyPID.f32DerivGain     = FRAC32(0.3);
    trMyPID.f32FiltCoef      = FRAC32(0.4);
    trMyPID.s16PropGainShift = 1;
    trMyPID.s16IntegGainShift = 1;
    trMyPID.s16DerivGainShift = 1;
    trMyPID.f32UpperLimit   = FRAC32(1.0);
    trMyPID.f32LowerLimit   = FRAC32(-1.0);

    // Clear the state variables
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    GFLIB_ControllerPIDpAWInit_F32(&trMyPID);

    // Alternative 2: API call with implementation parameter
```

```

// (only one alternative shall be used).
GFLIB_ControllerPIDpAWInit(&trMyPID, F32);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 32-bit fractional implementation is selected as default.
GFLIB_ControllerPIDpAWInit(&trMyPID);

// Initialize the state variables to predefined values
f32ControllerPIDpAWOut = FRAC32(0.5);
// Alternative 1: API call with postfix
// (only one alternative shall be used).
GFLIB_ControllerPIDpAWSetState_F32(f32ControllerPIDpAWOut, &trMyPID);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
GFLIB_ControllerPIDpAWSetState(f32ControllerPIDpAWOut, &trMyPID, F32);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 32-bit fractional implementation is selected as default.
GFLIB_ControllerPIDpAWSetState(f32ControllerPIDpAWOut, &trMyPID);
}

// Periodical function or interrupt - control loop
void ControlLoop(void)
{
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    f32Output = GFLIB\_ControllerPIDpAW\_F32(f32InErr, &trMyPID);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    f32Output = GFLIB_ControllerPIDpAW(f32InErr, &trMyPID, F32);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 32-bit fractional implementation is selected as default.
    f32Output = GFLIB_ControllerPIDpAW(f32InErr, &trMyPID);
}

```

### 2.18.5.2 Function GFLIB\_ControllerPIDpAWSetState\_F16

#### Declaration

```
void GFLIB_ControllerPIDpAWSetState_F16(tFrac16
f16ControllerPIDpAWOut, GFLIB\_CONTROLLER\_PID\_P\_AW\_T\_F16 *const
pParam);
```

#### Arguments

**Table 141. GFLIB\_ControllerPIDpAWSetState\_F16 arguments**

Type	Name	Direction	Description
<a href="#">tFrac16</a>	f16ControllerPIDpAWOut	<a href="#">input</a>	Required output of the GFLIB_ControllerPIDpAW.
<a href="#">GFLIB_CONTROLLER_PID_P_AW_T_F16</a> *const	pParam	<a href="#">input, output</a>	Pointer to the structure with GFLIB_ControllerPIDpAW state.

## Code Example

```
#include "gplib.h"

tFrac16 f16InErr;
tFrac16 f16Output;

GFLIB_CONTROLLER_PID_P_AW_T_F16 trMyPID = GFLIB_CONTROLLER_PID_P_AW_DEFAULT_F16;

void main(void)
{
    tFrac16 f16ControllerPIDpAWOut;

    // Set the input error
    f16InErr = FRAC16(0.25);

    // Initialize the controller parameters
    trMyPID.f16PropGain      = FRAC16(0.1);
    trMyPID.f16IntegGain     = FRAC16(0.2);
    trMyPID.f16DerivGain     = FRAC16(0.3);
    trMyPID.f16FiltCoef      = FRAC16(0.4);
    trMyPID.s16PropGainShift = 1;
    trMyPID.s16IntegGainShift = 1;
    trMyPID.s16DerivGainShift = 1;
    trMyPID.f16UpperLimit   = FRAC16(1.0);
    trMyPID.f16LowerLimit   = FRAC16(-1.0);

    // Clear the state variables
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    GFLIB_ControllerPIDpAWInit_F16(&trMyPID);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    GFLIB_ControllerPIDpAWInit(&trMyPID, F16);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 16-bit fractional implementation is selected as default.
    GFLIB_ControllerPIDpAWInit(&trMyPID);

    // Initialize the state variables to predefined values
    f16ControllerPIDpAWOut = FRAC16(0.5);
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    GFLIB_ControllerPIDpAWSetState_F16(f16ControllerPIDpAWOut, &trMyPID);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    GFLIB_ControllerPIDpAWSetState(f16ControllerPIDpAWOut, &trMyPID, F16);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 16-bit fractional implementation is selected as default.
    GFLIB_ControllerPIDpAWSetState(f16ControllerPIDpAWOut, &trMyPID);
}

// Periodical function or interrupt - control loop
```

```

void ControlLoop (void)
{
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    f16Output = GFLIB_ControllerPIDpAW_F16(f16InErr, &trMyPID);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    f16Output = GFLIB_ControllerPIDpAW(f16InErr, &trMyPID, F16);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 16-bit fractional implementation is selected as default.
    f16Output = GFLIB_ControllerPIDpAW(f16InErr, &trMyPID);
}

```

### 2.18.5.3 Function GFLIB\_ControllerPIDpAWSetState\_FLT

#### Declaration

```

void GFLIB_ControllerPIDpAWSetState_FLT (tFloat
                                         fltControllerPIDpAWOut, GFLIB_CONTROLLER_PID_P_AW_T_FLT *const
                                         pParam);

```

#### Arguments

**Table 142. GFLIB\_ControllerPIDpAWSetState\_FLT arguments**

Type	Name	Direction	Description
<u>tFloat</u>	fltControllerPIDpAWOut	<b>input</b>	Required output of the GFLIB_ControllerPIDpAW.
<u>GFLIB_CONTROLLER_PID_P_AW_T_FLT</u> *const	pParam	<b>input, output</b>	Pointer to the structure with GFLIB_ControllerPIDpAW state.

#### Code Example

```

#include "gplib.h"

tFloat fltInErr;
tFloat fltOutput;

GFLIB_CONTROLLER_PID_P_AW_T_FLT trMyPID = GFLIB_CONTROLLER_PID_P_AW_DEFAULT_FLT;

void main (void)
{
    tFloat fltControllerPIDpAWOut;

    // Set the input error
    fltInErr = 0.25F;

    // Initialize the controller parameters
    trMyPID.fltPropGain      = 0.1F;
    trMyPID.fltIntegGain     = 0.2F;
    trMyPID.fltDerivGain     = 0.3F;
    trMyPID.fltFiltCoef      = 0.4F;
    trMyPID.fltUpperLimit    = 1.0F;
    trMyPID.fltLowerLimit    = -1.0F;
}

```

```
// Clear the state variables
// Alternative 1: API call with postfix
// (only one alternative shall be used).
GFLIB_ControllerPIDpAWInit_FLT(&trMyPID);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
GFLIB_ControllerPIDpAWInit(&trMyPID, FLT);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if single precision floating-point implementation is selected
// as default.
GFLIB_ControllerPIDpAWInit(&trMyPID);

// Initialize the state variables to predefined values
fltControllerPIDpAWOut = 0.5F;
// Alternative 1: API call with postfix
// (only one alternative shall be used).
GFLIB_ControllerPIDpAWSetState_FLT(fltControllerPIDpAWOut, &trMyPID);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
GFLIB_ControllerPIDpAWSetState(fltControllerPIDpAWOut, &trMyPID, FLT);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if single precision floating-point implementation is selected
// as default.
GFLIB_ControllerPIDpAWSetState(fltControllerPIDpAWOut, &trMyPID);
}

// Periodical function or interrupt - control loop
void ControlLoop(void)
{
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    fltOutput = GFLIB_ControllerPIDpAW_FLT(fltInErr, &trMyPID);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    fltOutput = GFLIB_ControllerPIDpAW(fltInErr, &trMyPID, FLT);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if single precision floating-point implementation is selected
    // as default.
    fltOutput = GFLIB_ControllerPIDpAW(fltInErr, &trMyPID);
}
```

## 2.19 Function GFLIB\_ControllerPip

This function calculates a parallel form of the Proportional-Integral controller, without integral anti-windup.

## Description

A PI controller attempts to correct the error between a measured process variable and a desired set-point by calculating and then outputting a corrective action that can adjust the process accordingly. The GFLIB\_ControllerPlp function calculates the Proportional-Integral (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 without interaction. An anti-windup strategy is not implemented in this function.

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

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

Equation GFLIB\_ControllerPlp\_Eq1

where

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

Equation [GFLIB\\_ControllerPlp\\_Eq1](#) can be described using the Laplace transformation as follows:

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

Equation GFLIB\_ControllerPlp\_Eq2

The proportional part of equation [GFLIB\\_ControllerPlp\\_Eq2](#) is transformed into the discrete time domain simply as:

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

Equation GFLIB\_ControllerPlp\_Eq3

Transforming the integral part of equation [GFLIB\\_ControllerPlp\\_Eq2](#) into a discrete time domain using the Bilinear method, also known as trapezoidal approximation, leads to the following equation:

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

Equation GFLIB\_ControllerPlp\_Eq4

where  $T_s$  [sec] is the sampling time.

**Note:** The input pointer must contain a valid address otherwise a fault may occur (MemManage, BusFault, UsageFault, HardFault).

## Re-entrancy

The function is re-entrant.

### 2.19.1 Function `GFLIB_ControllerPip_F32`

#### Declaration

```
tFrac32 GFLIB_ControllerPip_F32 (tFrac32 f32InErr,
GFLIB_CONTROLLER_PI_P_T_F32 *const pParam);
```

#### Arguments

Table 143. `GFLIB_ControllerPip_F32` arguments

Type	Name	Direction	Description
<code>tFrac32</code>	<code>f32InErr</code>	<code>input</code>	Input error signal to the controller is a 32-bit number normalized between [-1, 1).
<code>GFLIB_CONTROLLER_PI_P_T_F32 *const</code>	<code>pParam</code>	<code>input, output</code>	Pointer to the controller parameters structure.

#### Return

The function returns a 32-bit value in format 1.31, representing the signal to be applied to the controlled system so that the input error is forced to zero.

#### Implementation details

In order to implement the discrete equation of the controller on the fixed point arithmetic platform, the maximal values (scales) of the input and output signals have to be known a priori. This is essential for correct casting of the physical signal values into fixed point values:

- $E^{MAX}$  - maximal value of the controller input error signal
- $U^{MAX}$  - maximal value of the controller output signal

The fractional representation of both input and output signals, normalized between [-1, 1), is obtained as follows:

$$e_f(k) = \frac{e(k)}{E^{MAX}}$$

$$u_f(k) = \frac{u(k)}{U^{MAX}}$$

Equation `GFLIB_ControllerPip_F32_Eq1`

Applying such scaling (normalization) on the proportional term of equation [`GFLIB\_ControllerPip\_eq3`](#) results in:

$$u_{pf}(k) = e_f(k) \cdot K_{P\_SC} \text{ where } K_{P\_SC} = K_p \cdot \frac{E^{MAX}}{U^{MAX}}$$

Equation `GFLIB_ControllerPip_F32_Eq2`

where  $K_{P\_SC}$  is the proportional gain parameter considering input/output scaling.

Analogically, scaling the integral term of equation [`GFLIB\_ControllerPip\_eq4`](#) results in:

$$u_{if}(k) = u_{if}(k-1) + K_{I\_SC} \cdot e_f(k) + K_{I\_SC} \cdot e_f(k-1) \text{ where } K_{I\_SC} = \frac{K_i T_s}{2} \cdot \frac{E^{MAX}}{U^{MAX}}$$

Equation `GFLIB_ControllerPip_F32_Eq3`

where  $K_{I\_SC}$  is the integral gain parameter considering input/output scaling.

The sum of the scaled proportional and integral terms gives a complete equation of the controller:

$$u_f(k) = e_f(k) \cdot K_{P\_SC} + u_{I_f}(k-1) + K_{I\_SC} \cdot e_f(k) + K_{I\_SC} \cdot e_f(k-1)$$

Equation GFLIB\_ControllerPlp\_F32\_Eq4

The problem is however, that either of the gain parameters  $K_{P\_SC}$ ,  $K_{I\_SC}$  can be out of the [-1, 1) range, hence cannot be directly interpreted as fractional values. To overcome this, it is necessary to scale these gain parameters using the shift values as follows:

$$f32PropGain = K_{P\_SC} \cdot 2^{s16PropGainShift}$$

$$f32IntegGain = K_{I\_SC} \cdot 2^{s16IntegGainShift}$$

Equation GFLIB\_ControllerPlp\_F32\_Eq5

where

- f32PropGain - is the scaled value of proportional gain [-1, 1)
- s16PropGainShift - is the scaling shift for proportional gain [-31,31]
- f32IntegGain - is the scaled value of integral gain [-1, 1)
- s16IntegGainShift - is the scaling shift for integral gain [-31,31]

All controller parameters and states can be reset during declaration using the [GFLIB\\_CONTROLLER\\_PI\\_P\\_DEFAULT\\_F32](#) macro.

### Code Example

```
#include "gflib.h"

tFrac32 f32InErr;
tFrac32 f32Output;

GFLIB\_CONTROLLER\_PI\_P\_T\_F32 trMyPI = GFLIB\_CONTROLLER\_PI\_P\_DEFAULT\_F32;

void main(void)
{
    tFrac32 f32ControllerPlpOut;

    // Set the input error
    f32InErr = FRAC32(0.25);

    // Initialize the controller parameters
    trMyPI.f32PropGain      = FRAC32(0.01);
    trMyPI.f32IntegGain     = FRAC32(0.02);
    trMyPI.s16PropGainShift = 1;
    trMyPI.s16IntegGainShift = 1;

    // Clear the state variables
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    GFLIB\_ControllerPlpInit\_F32(&trMyPI);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    GFLIB_ControllerPlpInit(&trMyPI, F32);
```

```

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 32-bit fractional implementation is selected as default.
GFLIB_ControllerPIpInit(&trMyPI);

// Initialize the state variables to predefined values
f32ControllerPIpOut = FRAC32(0.03);
// Alternative 1: API call with postfix
// (only one alternative shall be used).
GFLIB_ControllerPIpSetState_F32(f32ControllerPIpOut, &trMyPI);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
GFLIB_ControllerPIpSetState(f32ControllerPIpOut, &trMyPI, F32);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 32-bit fractional implementation is selected as default.
GFLIB_ControllerPIpSetState(f32ControllerPIpOut, &trMyPI);
}

// Periodical function or interrupt - control loop
void ControlLoop(void)
{
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    f32Output = GFLIB_ControllerPIp_F32(f32InErr, &trMyPI);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    f32Output = GFLIB_ControllerPIp(f32InErr, &trMyPI, F32);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 32-bit fractional implementation is selected as default.
    f32Output = GFLIB_ControllerPIp(f32InErr, &trMyPI);
}

```

## 2.19.2 Function GFLIB\_ControllerPIp\_F16

### Declaration

```
tFrac16 GFLIB_ControllerPIp_F16(tFrac16 f16InErr,
GFLIB_CONTROLLER_PI_P_T_F16 *const pParam);
```

### Arguments

Table 144. GFLIB\_ControllerPIp\_F16 arguments

Type	Name	Direction	Description
<u>tFrac16</u>	f16InErr	input	Input error signal to the controller is a 16-bit number normalized between [-1, 1).
<u>GFLIB_CONTROLLER_PI_P_T_F16</u> *const	pParam	input, output	Pointer to the controller parameters structure.

**Return**

The function returns a 16-bit value in format 1.15, representing the signal to be applied to the controlled system so that the input error is forced to zero.

**Implementation details**

In order to implement the discrete equation of the controller on the fixed point arithmetic platform, the maximal values (scales) of the input and output signals have to be known a priori. This is essential for correct casting of the physical signal values into fixed point values:

- $E^{MAX}$  - maximal value of the controller input error signal
- $U^{MAX}$  - maximal value of the controller output signal

The fractional representation of both input and output signals, normalized between [-1, 1), is obtained as follows:

$$e_f(k) = \frac{e(k)}{E^{MAX}}$$

$$u_f(k) = \frac{u(k)}{U^{MAX}}$$

Equation [GFLIB\\_ControllerPip\\_F16\\_Eq1](#)

Applying such scaling (normalization) on the proportional term of equation [GFLIB\\_ControllerPip\\_eq3](#) results in:

$$u_{pf}(k) = e_f(k) \cdot K_{P\_SC} \quad \text{where } K_{P\_SC} = K_P \cdot \frac{E^{MAX}}{U^{MAX}}$$

Equation [GFLIB\\_ControllerPip\\_F16\\_Eq2](#)

where  $K_{P\_SC}$  is the proportional gain parameter considering input/output scaling.

Analogically, scaling the integral term of equation [GFLIB\\_ControllerPip\\_eq4](#) results in:

$$u_{if}(k) = u_{if}(k-1) + K_{I\_SC} \cdot e_f(k) + K_{I\_SC} \cdot e_f(k-1) \quad \text{where } K_{I\_SC} = \frac{K_I T_s}{2} \cdot \frac{E^{MAX}}{U^{MAX}}$$

Equation [GFLIB\\_ControllerPip\\_F16\\_Eq3](#)

where  $K_{I\_SC}$  is the integral gain parameter considering input/output scaling.

The sum of the scaled proportional and integral terms gives a complete equation of the controller:

$$u_f(k) = e_f(k) \cdot K_{P\_SC} + u_{if}(k-1) + K_{I\_SC} \cdot e_f(k) + K_{I\_SC} \cdot e_f(k-1)$$

Equation [GFLIB\\_ControllerPip\\_F16\\_Eq4](#)

The problem is however, that either of the gain parameters  $K_{P\_SC}$ ,  $K_{I\_SC}$  can be out of the [-1, 1) range, hence cannot be directly interpreted as fractional values. To overcome this, it is necessary to scale these gain parameters using the shift values as follows:

$$f16PropGain = K_{P\_SC} \cdot 2^{s16PropGainShift}$$

$$f16IntegGain = K_{I\_SC} \cdot 2^{s16IntegGainShift}$$

Equation [GFLIB\\_ControllerPip\\_F16\\_Eq5](#)

where

- f16PropGain - is the scaled value of proportional gain [-1, 1)
- s16PropGainShift - is the scaling shift for proportional gain [-15, 15)
- f16IntegGain - is the scaled value of integral gain [-1, 1)
- s16IntegGainShift - is the scaling shift for integral gain [-15, 15)

All controller parameters and states can be reset during declaration using the [GFLIB\\_CONTROLLER\\_PI\\_P\\_DEFAULT\\_F16](#) macro.

### Code Example

```
#include "gflib.h"

tFrac16 f16InErr;
tFrac16 f16Output;

GFLIB_CONTROLLER_PI_P_T_F16 trMyPI = GFLIB\_CONTROLLER\_PI\_P\_DEFAULT\_F16;

void main(void)
{
    tFrac16 f16ControllerPIpOut;

    // Set the input error
    f16InErr = FRAC16\(0.25\);

    // Initialize the controller parameters
    trMyPI.f16PropGain      = FRAC16\\(0.01\\);
    trMyPI.f16IntegGain     = FRAC16\\\(0.02\\\);
    trMyPI.s16PropGainShift = 1;
    trMyPI.s16IntegGainShift = 1;
    trMyPI.f16UpperLimit   = FRAC16\\\\(1.0\\\\);
    trMyPI.f16LowerLimit   = FRAC16\\\\\(-1.0\\\\\);

    // Clear the state variables
    // Alternative 1: API call with postfix
    // \\\\\(only one alternative shall be used\\\\\).
    GFLIB\\\\\\_ControllerPIpInit\\\\\\_F16\\\\\\(&trMyPI\\\\\\);

    // Alternative 2: API call with implementation parameter
    // \\\\\\(only one alternative shall be used\\\\\\).
    GFLIB\\\\\\_ControllerPIpInit\\\\\\(&trMyPI, F16\\\\\\);

    // Alternative 3: API call with global configuration of implementation
    // \\\\\\(only one alternative shall be used\\\\\\). This alternative is available
    // only if 16-bit fractional implementation is selected as default.
    GFLIB\\\\\\_ControllerPIpInit\\\\\\(&trMyPI\\\\\\);

    // Initialize the state variables to predefined values
    f16ControllerPIpOut = FRAC16\\\\\\\(0.03\\\\\\\);
    // Alternative 1: API call with postfix
    // \\\\\\\(only one alternative shall be used\\\\\\\).
    GFLIB\\\\\\\\_ControllerPIpSetState\\\\\\\\_F16\\\\\\\\(f16ControllerPIpOut, &trMyPI\\\\\\\\);

    // Alternative 2: API call with implementation parameter
    // \\\\\\\\(only one alternative shall be used\\\\\\\\).
    GFLIB\\\\\\\\_ControllerPIpSetState\\\\\\\\(f16ControllerPIpOut, &trMyPI, F16\\\\\\\\);
```

```

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 16-bit fractional implementation is selected as default.
GFLIB_ControllerPIpSetState(f16ControllerPIpOut, &trMyPI);

}

// Periodical function or interrupt - control loop
void ControlLoop(void)
{
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    f16Output = GFLIB_ControllerPIp_F16(f16InErr, &trMyPI);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    f16Output = GFLIB_ControllerPIp(f16InErr, &trMyPI, F16);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 16-bit fractional implementation is selected as default.
    f16Output = GFLIB_ControllerPIp(f16InErr, &trMyPI);
}

```

### 2.19.3 Function GFLIB\_ControllerPIp\_FLT

#### Declaration

```
tFloat GFLIB_ControllerPIp_FLT(tFloat fltInErr,
                                GFLIB_CONTROLLER_PI_P_T_FLT *const pParam);
```

#### Arguments

Table 145. GFLIB\_ControllerPIp\_FLT arguments

Type	Name	Direction	Description
tFloat	fltInErr	input	Input error signal to the controller in single precision floating format.
GFLIB_CONTROLLER_PI_P_T_FLT *const	pParam	input, output	Pointer to the controller parameters structure.

#### Return

The function returns a single precision floating point value, representing the signal to be applied to the controlled system so that the input error is forced to zero.

#### Implementation details

The sum of the scaled proportional and integral terms gives a complete equation of the controller:

$$u_f(k) = e_f(k) \cdot K_{P\_SC} + u_{I_f}(k-1) + K_{I\_SC} \cdot e_f(k) + K_{L\_SC} \cdot e_f(k-1)$$

Equation GFLIB\_ControllerPIp\_FLT\_Eq1

All controller parameters and states can be reset during declaration using the [GFLIB\\_CONTROLLER\\_PI\\_P\\_DEFAULT\\_FLT](#) macro.

**Note:** The function may raise floating-point exceptions (invalid operation, overflow, underflow, inexact, input denormal). The floating-point unit must be enabled in CPACR register to prevent NOCP UsageFault exception.

#### 2.19.4 Function `GFLIB_ControllerPIpInit`

##### Description

This function clears the GFLIB\_ControllerPIp state variables.

**Note:** The input/output pointer must contain a valid address, otherwise a fault may occur (MemManage, BusFault, UsageFault, HardFault).

##### Re-entrancy

The function is re-entrant for a different pCtrl.

##### 2.19.4.1 Function `GFLIB_ControllerPIpInit_F32`

###### Declaration

```
void GFLIB_ControllerPIpInit_F32(GFLIB\_CONTROLLER\_PI\_P\_T\_F32  
*const pParam);
```

###### Arguments

Table 146. `GFLIB_ControllerPIpInit_F32` arguments

Type	Name	Direction	Description
<a href="#">GFLIB_CONTROLLER_PI_P_T_F32</a> *const	pParam	input, output	Pointer to the structure with GFLIB_ControllerPIp state.

##### Code Example

```
#include "gflib.h"

tFrac32 f32InErr;
tFrac32 f32Output;

GFLIB\_CONTROLLER\_PI\_P\_T\_F32 trMyPI = GFLIB\_CONTROLLER\_PI\_P\_DEFAULT\_F32;

void main(void)
{
    tFrac32 f32ControllerPIpOut;

    // Set the input error
    f32InErr = FRAC32(0.25);

    // Initialize the controller parameters
    trMyPI.f32PropGain      = FRAC32(0.01);
    trMyPI.f32IntegGain     = FRAC32(0.02);
    trMyPI.s16PropGainShift = 1;
    trMyPI.s16IntegGainShift = 1;

    // Clear the state variables
    // Alternative 1: API call with postfix
```

```
// (only one alternative shall be used).
GFLIB_ControllerPIpInit_F32(&trMyPI);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
GFLIB_ControllerPIpInit(&trMyPI, F32);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 32-bit fractional implementation is selected as default.
GFLIB_ControllerPIpInit(&trMyPI);

// Initialize the state variables to predefined values
f32ControllerPIpOut = FRAC32(0.03);
// Alternative 1: API call with postfix
// (only one alternative shall be used).
GFLIB\_ControllerPIpSetState\_F32(f32ControllerPIpOut, &trMyPI);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
GFLIB_ControllerPIpSetState(f32ControllerPIpOut, &trMyPI, F32);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 32-bit fractional implementation is selected as default.
GFLIB_ControllerPIpSetState(f32ControllerPIpOut, &trMyPI);
}

// Periodical function or interrupt - control loop
void ControlLoop(void)
{
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    f32Output = GFLIB\_ControllerPIp\_F32(f32InErr, &trMyPI);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    f32Output = GFLIB_ControllerPIp(f32InErr, &trMyPI, F32);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 32-bit fractional implementation is selected as default.
    f32Output = GFLIB_ControllerPIp(f32InErr, &trMyPI);
}
```

#### 2.19.4.2 Function [GFLIB\\_ControllerPIpInit\\_F16](#)

##### Declaration

```
void GFLIB_ControllerPIpInit_F16(GFLIB\_CONTROLLER\_PI\_P\_T\_F16
*const pParam);
```

## Arguments

**Table 147.** GFLIB\_ControllerPIpInit\_F16 arguments

Type	Name	Direction	Description
<u>GFLIB_CONTROLLER_PI_P_T_F16</u> *const	pParam	input, output	Pointer to the structure with GFLIB_ControllerPIp state.

## Code Example

```

#include "gflib.h"

tFrac16 f16InErr;
tFrac16 f16Output;

GFLIB_CONTROLLER_PI_P_T_F16 trMyPI = GFLIB_CONTROLLER_PI_P_DEFAULT_F16;

void main(void)
{
    tFrac16 f16ControllerPIpOut;

    // Set the input error
    f16InErr = FRAC16(0.25);

    // Initialize the controller parameters
    trMyPI.f16PropGain      = FRAC16(0.01);
    trMyPI.f16IntegGain     = FRAC16(0.02);
    trMyPI.s16PropGainShift = 1;
    trMyPI.s16IntegGainShift = 1;
    trMyPI.f16UpperLimit   = FRAC16(1.0);
    trMyPI.f16LowerLimit   = FRAC16(-1.0);

    // Clear the state variables
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    GFLIB_ControllerPIpInit_F16(&trMyPI);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    GFLIB_ControllerPIpInit(&trMyPI, F16);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 16-bit fractional implementation is selected as default.
    GFLIB_ControllerPIpInit(&trMyPI);

    // Initialize the state variables to predefined values
    f16ControllerPIpOut = FRAC16(0.03);
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    GFLIB_ControllerPIpSetState_F16(f16ControllerPIpOut, &trMyPI);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    GFLIB_ControllerPIpSetState(f16ControllerPIpOut, &trMyPI, F16);

    // Alternative 3: API call with global configuration of implementation

```

```

// (only one alternative shall be used). This alternative is available
// only if 16-bit fractional implementation is selected as default.
GFLIB_ControllerPIpSetState(f16ControllerPIpOut, &trMyPI);
}

// Periodical function or interrupt - control loop
void ControlLoop(void)
{
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    f16Output = GFLIB\_ControllerPIp\_F16(f16InErr, &trMyPI);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    f16Output = GFLIB_ControllerPIp(f16InErr, &trMyPI, F16);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 16-bit fractional implementation is selected as default.
    f16Output = GFLIB_ControllerPIp(f16InErr, &trMyPI);
}

```

#### 2.19.4.3 Function [GFLIB\\_ControllerPIpInit\\_FLT](#)

##### Declaration

```
void GFLIB_ControllerPIpInit_FLT(GFLIB\_CONTROLLER\_PI\_P\_T\_FLT
*const pParam);
```

##### Arguments

**Table 148. [GFLIB\\_ControllerPIpInit\\_FLT](#) arguments**

Type	Name	Direction	Description
<a href="#">GFLIB_CONTROLLER_PI_P_T_FLT</a> *const	pParam	input, output	Pointer to the structure with <a href="#">GFLIB_ControllerPIp</a> state.

##### Code Example

```

#include "gplib.h"

tFloat fltInErr;
tFloat fltOutput;

GFLIB\_CONTROLLER\_PI\_P\_T\_FLT trMyPI = GFLIB\_CONTROLLER\_PI\_P\_DEFAULT\_FLT;

void main(void)
{
    tFloat fltControllerPIpOut;

    // Set the input error
    fltInErr = 0.25F;

    // Initialize the controller parameters
    trMyPI.fltPropGain = 0.04F;
    trMyPI.fltIntegGain = 0.02F;
    trMyPI.fltUpperLimit = 1.0F;
}

```

```
trMyPI.fltLowerLimit = -1.0F;

// Clear the state variables
// Alternative 1: API call with postfix
// (only one alternative shall be used).
GFLIB_ControllerPIpInit_FLT(&trMyPI);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
GFLIB_ControllerPIpInit(&trMyPI, FLT);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if single precision floating-point implementation is selected
// as default.
GFLIB_ControllerPIpInit(&trMyPI);

// Initialize the state variables to predefined values
fltControllerPIpOut = 0.03F;
// Alternative 1: API call with postfix
// (only one alternative shall be used).
GFLIB_ControllerPIpSetState_FLT(fltControllerPIpOut, &trMyPI);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
GFLIB_ControllerPIpSetState(fltControllerPIpOut, &trMyPI, FLT);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if single precision floating-point implementation is selected
// as default.
GFLIB_ControllerPIpSetState(fltControllerPIpOut, &trMyPI);
}

// Periodical function or interrupt - control loop
void ControlLoop(void)
{
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    fltOutput = GFLIB_ControllerPIp_FLT(fltInErr, &trMyPI);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    fltOutput = GFLIB_ControllerPIp(fltInErr, &trMyPI, FLT);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if single precision floating-point implementation is selected
    // as default.
    fltOutput = GFLIB_ControllerPIp(fltInErr, &trMyPI);
}
```

## 2.19.5 Function GFLIB\_ControllerPIpSetState

**Description**

This function initializes the GFLIB\_ControllerPIp state variables to achieve the required output values.

**Note:** The input/output pointer must contain a valid address, otherwise a fault may occur (MemManage, BusFault, UsageFault, HardFault).

**Re-entrancy**

The function is re-entrant for a different pCtrl.

**2.19.5.1 Function GFLIB\_ControllerPIpSetState\_F32****Declaration**

```
void GFLIB_ControllerPIpSetState_F32(tFrac32 f32ControllerPIpOut,  
GFLIB\_CONTROLLER\_PI\_P\_T\_F32 *const pParam);
```

**Arguments****Table 149. GFLIB\_ControllerPIpSetState\_F32 arguments**

Type	Name	Direction	Description
<a href="#">tFrac32</a>	f32ControllerPIpOut	input	Required output of the GFLIB_ControllerPIp.
<a href="#">GFLIB_CONTROLLER_PI_P_T_F32</a> *const	pParam	input, output	Pointer to the structure with GFLIB_ControllerPIp state.

**Code Example**

```
#include "gflib.h"

tFrac32 f32InErr;
tFrac32 f32Output;

GFLIB\_CONTROLLER\_PI\_P\_T\_F32 trMyPI = GFLIB\_CONTROLLER\_PI\_P\_DEFAULT\_F32;

void main(void)
{
    tFrac32 f32ControllerPIpOut;

    // Set the input error
    f32InErr = FRAC32(0.25);

    // Initialize the controller parameters
    trMyPI.f32PropGain      = FRAC32(0.01);
    trMyPI.f32IntegGain     = FRAC32(0.02);
    trMyPI.s16PropGainShift = 1;
    trMyPI.s16IntegGainShift = 1;

    // Clear the state variables
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    GFLIB\_ControllerPIpInit\_F32(&trMyPI);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
}
```

```

GFLIB_ControllerPIpInit(&trMyPI, F32);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 32-bit fractional implementation is selected as default.
GFLIB_ControllerPIpInit(&trMyPI);

// Initialize the state variables to predefined values
f32ControllerPIpOut = FRAC32\(0.03\);
// Alternative 1: API call with postfix
// (only one alternative shall be used).
GFLIB_ControllerPIpSetState_F32(f32ControllerPIpOut, &trMyPI);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
GFLIB_ControllerPIpSetState(f32ControllerPIpOut, &trMyPI, F32);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 32-bit fractional implementation is selected as default.
GFLIB_ControllerPIpSetState(f32ControllerPIpOut, &trMyPI);
}

// Periodical function or interrupt - control loop
void ControlLoop(void)
{
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    f32Output = GFLIB\_ControllerPIp\_F32\(f32InErr, &trMyPI\);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    f32Output = GFLIB_ControllerPIp(f32InErr, &trMyPI, F32);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 32-bit fractional implementation is selected as default.
    f32Output = GFLIB_ControllerPIp(f32InErr, &trMyPI);
}

```

### 2.19.5.2 Function GFLIB\_ControllerPIpSetState\_F16

#### Declaration

```
void GFLIB_ControllerPIpSetState_F16(tFrac16 f16ControllerPIpOut,
GFLIB\_CONTROLLER\_PI\_P\_T\_F16 *const pParam);
```

#### Arguments

**Table 150. GFLIB\_ControllerPIpSetState\_F16 arguments**

Type	Name	Direction	Description
<a href="#">tFrac16</a>	f16ControllerPIpOut	input	Required output of the GFLIB_ControllerPIp.
<a href="#">GFLIB_CONTROLLER_PI_P_T_F16</a> *const	pParam	input, output	Pointer to the structure with GFLIB_ControllerPIp state.

## Code Example

```
#include "gplib.h"

tFrac16 f16InErr;
tFrac16 f16Output;

GFLIB_CONTROLLER_PI_P_T_F16 trMyPI = GFLIB_CONTROLLER_PI_P_DEFAULT_F16;

void main(void)
{
    tFrac16 f16ControllerPIpOut;

    // Set the input error
    f16InErr = FRAC16(0.25);

    // Initialize the controller parameters
    trMyPI.f16PropGain      = FRAC16(0.01);
    trMyPI.f16IntegGain     = FRAC16(0.02);
    trMyPI.s16PropGainShift = 1;
    trMyPI.s16IntegGainShift = 1;
    trMyPI.f16UpperLimit   = FRAC16(1.0);
    trMyPI.f16LowerLimit   = FRAC16(-1.0);

    // Clear the state variables
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    GFLIB_ControllerPIpInit_F16(&trMyPI);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    GFLIB_ControllerPIpInit(&trMyPI, F16);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 16-bit fractional implementation is selected as default.
    GFLIB_ControllerPIpInit(&trMyPI);

    // Initialize the state variables to predefined values
    f16ControllerPIpOut = FRAC16(0.03);
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    GFLIB_ControllerPIpSetState_F16(f16ControllerPIpOut, &trMyPI);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    GFLIB_ControllerPIpSetState(f16ControllerPIpOut, &trMyPI, F16);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 16-bit fractional implementation is selected as default.
    GFLIB_ControllerPIpSetState(f16ControllerPIpOut, &trMyPI);
}

// Periodical function or interrupt - control loop
void ControlLoop(void)
{
    // Alternative 1: API call with postfix
    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    GFLIB_ControllerPIpSetState(f16ControllerPIpOut, &trMyPI, F16);
}
```

```

// (only one alternative shall be used).
f16Output = GFLIB\_ControllerPIp\_F16(f16InErr, &trMyPI);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
f16Output = GFLIB\_ControllerPIp(f16InErr, &trMyPI, F16);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 16-bit fractional implementation is selected as default.
f16Output = GFLIB\_ControllerPIp(f16InErr, &trMyPI);
}

```

### 2.19.5.3 Function [GFLIB\\_ControllerPIpSetState\\_FLT](#)

#### Declaration

```
void GFLIB\_ControllerPIpSetState\_FLT(tFloat fltControllerPIpOut,
GFLIB\_CONTROLLER\_PI\_P\_T\_FLT *const pParam);
```

#### Arguments

**Table 151. [GFLIB\\_ControllerPIpSetState\\_FLT](#) arguments**

Type	Name	Direction	Description
<a href="#">tFloat</a>	fltControllerPIpOut	input	Required output of the <a href="#">GFLIB_ControllerPIp</a> .
<a href="#">GFLIB_CONTROLLER_PI_P_T_FLT</a> *const	pParam	input, output	Pointer to the structure with <a href="#">GFLIB_ControllerPIp</a> state.

#### Code Example

```

#include "gplib.h"

tFloat fltInErr;
tFloat fltOutput;

GFLIB\_CONTROLLER\_PI\_P\_T\_FLT trMyPI = GFLIB\_CONTROLLER\_PI\_P\_DEFAULT\_FLT;

void main(void)
{
    tFloat fltControllerPIpOut;

    // Set the input error
    fltInErr = 0.25F;

    // Initialize the controller parameters
    trMyPI.fltPropGain = 0.04F;
    trMyPI.fltIntegGain = 0.02F;
    trMyPI.fltUpperLimit = 1.0F;
    trMyPI.fltLowerLimit = -1.0F;

    // Clear the state variables
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    GFLIB\_ControllerPIpInit\_FLT(&trMyPI);

    // Alternative 2: API call with implementation parameter

```

```
// (only one alternative shall be used).
GFLIB_ControllerPIpInit(&trMyPI, FLT);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if single precision floating-point implementation is selected
// as default.
GFLIB_ControllerPIpInit(&trMyPI);

// Initialize the state variables to predefined values
fltControllerPIpOut = 0.03F;
// Alternative 1: API call with postfix
// (only one alternative shall be used).
GFLIB_ControllerPIpSetState_FLT(fltControllerPIpOut, &trMyPI);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
GFLIB_ControllerPIpSetState(fltControllerPIpOut, &trMyPI, FLT);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if single precision floating-point implementation is selected
// as default.
GFLIB_ControllerPIpSetState(fltControllerPIpOut, &trMyPI);
}

// Periodical function or interrupt - control loop
void ControlLoop(void)
{
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    fltOutput = GFLIB\_ControllerPIp\_FLT(fltInErr, &trMyPI);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    fltOutput = GFLIB_ControllerPIp(fltInErr, &trMyPI, FLT);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if single precision floating-point implementation is selected
    // as default.
    fltOutput = GFLIB_ControllerPIp(fltInErr, &trMyPI);
}
```

## 2.20 Function GFLIB\_ControllerPIpAW

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

### Description

A PI controller attempts to correct the error between a measured process variable and a desired set-point by calculating and then outputting a corrective action that can adjust the process accordingly. The GFLIB\_ControllerPIpAW function calculates the Proportional-Integral (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 without interaction. The controller output is limited and the limit values

(UpperLimit and LowerLimit) are defined by the user. The PI controller algorithm also returns a limitation flag. This flag (u16LimitFlag) is a member of the structure of the PI controller parameters. If the PI controller output reaches the upper or lower limit then u16LimitFlag = 1, otherwise u16LimitFlag = 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 PI algorithm in the continuous time domain can be described as:

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

Equation [GFLIB\\_ControllerPIpAW\\_Eq1](#)

where

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

Equation [GFLIB\\_ControllerPIpAW\\_Eq1](#) can be described using the Laplace transformation as follows:

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

Equation [GFLIB\\_ControllerPIpAW\\_Eq2](#)

The proportional part of equation [GFLIB\\_ControllerPIpAW\\_Eq2](#) is transformed into the discrete time domain simply as:

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

Equation [GFLIB\\_ControllerPIpAW\\_Eq3](#)

Transforming the integral part of equation [GFLIB\\_ControllerPIpAW\\_Eq2](#) into a discrete time domain using the Bilinear method, also known as trapezoidal approximation, leads to the following equation:

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

Equation [GFLIB\\_ControllerPIpAW\\_Eq4](#)

where  $T_s$  [sec] is the sampling time.

**Note:** The input pointer must contain a valid address otherwise a fault may occur (MemManage, BusFault, UsageFault, HardFault).

## Re-entrancy

The function is re-entrant.

### 2.20.1 Function `GFLIB_ControllerPipAW_F32`

#### Declaration

```
tFrac32 GFLIB_ControllerPipAW_F32(tFrac32 f32InErr,
GFLIB_CONTROLLER_PIAW_P_T_F32 *const pParam);
```

#### Arguments

Table 152. `GFLIB_ControllerPipAW_F32` arguments

Type	Name	Direction	Description
<code>tFrac32</code>	<code>f32InErr</code>	<code>input</code>	Input error signal to the controller is a 32-bit number normalized between [-1, 1).
<code>GFLIB_CONTROLLER_PIAW_P_T_F32 *const</code>	<code>pParam</code>	<code>input, output</code>	Pointer to the controller parameters structure.

#### Return

The function returns a 32-bit value in format 1.31, representing the signal to be applied to the controlled system so that the input error is forced to zero.

#### Implementation details

In order to implement the discrete equation of the controller on the fixed point arithmetic platform, the maximal values (scales) of input and output signals have to be known a priori. This is essential for correct casting of the physical signal values into fixed point values:

- $E^{MAX}$  - maximal value of the controller input error signal
- $U^{MAX}$  - maximal value of the controller output signal

The fractional representation of both input and output signals, normalized between [-1, 1), is obtained as follows:

$$e_f(k) = \frac{e(k)}{E^{MAX}}$$

$$u_f(k) = \frac{u(k)}{U^{MAX}}$$

Equation `GFLIB_ControllerPipAW_F32_Eq1`

Applying such scaling (normalization) on the proportional term of equation [`GFLIB\_ControllerPipAW\_eq3`](#) results in:

$$u_{pf}(k) = e_f(k) \cdot K_{P\_SC} \text{ where } K_{P\_SC} = K_p \cdot \frac{E^{MAX}}{U^{MAX}}$$

Equation `GFLIB_ControllerPipAW_F32_Eq2`

where  $K_{P\_SC}$  is the proportional gain parameter considering input/output scaling.

Analogically, scaling the integral term of equation [`GFLIB\_ControllerPipAW\_eq4`](#) results in:

$$u_{if}(k) = u_{if}(k-1) + K_{I\_SC} \cdot e_f(k) + K_{I\_SC} \cdot e_f(k-1) \text{ where } K_{I\_SC} = \frac{K_I T_s}{2} \cdot \frac{E^{MAX}}{U^{MAX}}$$

Equation `GFLIB_ControllerPipAW_F32_Eq3`

where  $K_{I\_sc}$  is the integral gain parameter considering input/output scaling.

The sum of the scaled proportional and integral terms gives a complete equation of the controller. The problem is however, that either of the gain parameters  $K_{P\_sc}$ ,  $K_{I\_sc}$  can be out of the [-1, 1) range, hence can not be directly interpreted as fractional values. To overcome this, it is necessary to scale these gain parameters using the shift values as follows:

$$\begin{aligned} f32PropGain &= K_{P\_sc} \cdot 2^{s16PropGainShift} \\ f32IntegGain &= K_{I\_sc} \cdot 2^{s16IntegGainShift} \end{aligned}$$

Equation GFLIB\_ControllerPipAW\_F32\_Eq4

where

- f16PropGain - is the scaled value of proportional gain [-1, 1)
- s16PropGainShift - is the scaling shift for proportional gain [-31, 31)
- f16IntegGain - is the scaled value of integral gain [-1, 1)
- s16IntegGainShift - is the scaling shift for integral gain [-31, 31)

The sum of the scaled proportional and integral terms gives a complete equation of the controller:

$$u_f(k) = e_f(k) \cdot K_{P\_sc} + u_{if}(k-1) + K_{I\_sc} \cdot e_f(k) + K_{I\_sc} \cdot e_f(k-1)$$

Equation GFLIB\_ControllerPipAW\_F32\_Eq5

The output signal limitation is implemented in this controller. The actual output  $u(k)$  is bounded not to exceed the given limit values  $f32UpperLimit$ ,  $f32LowerLimit$ . This is due to either the bounded power of the actuator or to the physical constraints of the plant.

$$u(k) = \begin{cases} f32UpperLimit & \text{if } u_f(k) \geq f32UpperLimit \\ u_f(k) & \text{if } f32LowerLimit < u_f(k) < f32UpperLimit \\ f32LowerLimit & \text{if } u_f(k) \leq f32LowerLimit \end{cases}$$

Equation GFLIB\_ControllerPipAW\_F32\_Eq6

The bounds are described by a limitation element equation

[GFLIB\\_ControllerPipAW\\_F32\\_Eq6](#). When the bounds are exceeded the non-linear 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.

All controller parameters and states can be reset during declaration using the [GFLIB\\_CONTROLLER\\_PIAW\\_P\\_DEFAULT\\_F32](#) macro.

### Code Example

```
#include "gflib.h"

tFrac32 f32InErr;
tFrac32 f32Output;
```

```
GFLIB_CONTROLLER_PIAW_P_T_F32 trMyPI = GFLIB_CONTROLLER_PIAW_P_DEFAULT_F32;

void main(void)
{
    tFrac32 f32ControllerPIpAWOut;

    // Set the input error
    f32InErr = FRAC32(0.25);

    // Initialize the controller parameters
    trMyPI.f32PropGain      = FRAC32(0.01);
    trMyPI.f32IntegGain     = FRAC32(0.02);
    trMyPI.s16PropGainShift = 1;
    trMyPI.s16IntegGainShift = 1;
    trMyPI.f32UpperLimit   = FRAC32(1.0);
    trMyPI.f32LowerLimit   = FRAC32(-1.0);

    // Clear the state variables
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    GFLIB_ControllerPIpAWInit_F32(&trMyPI);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    GFLIB_ControllerPIpAWInit(&trMyPI, F32);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 32-bit fractional implementation is selected as default.
    GFLIB_ControllerPIpAWInit(&trMyPI);

    // Initialize the state variables to predefined values
    f32ControllerPIpAWOut = FRAC32(0.03);
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    GFLIB_ControllerPIpAWSetState_F32(f32ControllerPIpAWOut, &trMyPI);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    GFLIB_ControllerPIpAWSetState(f32ControllerPIpAWOut, &trMyPI, F32);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 32-bit fractional implementation is selected as default.
    GFLIB_ControllerPIpAWSetState(f32ControllerPIpAWOut, &trMyPI);
}

// Periodical function or interrupt - control loop
void ControlLoop(void)
{
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    f32Output = GFLIB_ControllerPIpAW_F32(f32InErr, &trMyPI);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    f32Output = GFLIB_ControllerPIpAW(f32InErr, &trMyPI, F32);

    // Alternative 3: API call with global configuration of implementation
```

```
// (only one alternative shall be used). This alternative is available
// only if 32-bit fractional implementation is selected as default.
f32Output = GFLIB_ControllerPIpAW(f32InErr, &trMyPI);
}
```

## 2.20.2 Function GFLIB\_ControllerPIpAW\_F16

### Declaration

```
tFrac16 GFLIB_ControllerPIpAW_F16(tFrac16 f16InErr,
GFLIB_CONTROLLER_PIAW_P_T_F16 *const pParam);
```

### Arguments

**Table 153. GFLIB\_ControllerPIpAW\_F16 arguments**

Type	Name	Direction	Description
<b>tFrac16</b>	f16InErr	<b>input</b>	Input error signal to the controller is a 16-bit number normalized between [-1, 1).
<b>GFLIB_CONTROLLER_PIAW_P_T_F16</b> *const	pParam	<b>input, output</b>	Pointer to the controller parameters structure.

### Return

The function returns a 16-bit value in format 1.15, representing the signal to be applied to the controlled system so that the input error is forced to zero.

### Implementation details

In order to implement the discrete equation of the controller on the fixed point arithmetic platform, the maximal values (scales) of input and output signals have to be known a priori. This is essential for correct casting of the physical signal values into fixed point values:

- $E^{MAX}$  - maximal value of the controller input error signal
- $U^{MAX}$  - maximal value of the controller output signal

The fractional representation of both input and output signals, normalized between [-1, 1), is obtained as follows:

$$e_f(k) = \frac{e(k)}{E^{MAX}}$$

$$u_f(k) = \frac{u(k)}{U^{MAX}}$$

Equation GFLIB\_ControllerPIpAW\_F16\_Eq1

Applying such scaling (normalization) on the proportional term of equation [GFLIB\\_ControllerPIpAW\\_eq3](#) results in:

$$u_{pf}(k) = e_f(k) \cdot K_{P\_SC} \text{ where } K_{P\_SC} = K_p \cdot \frac{E^{MAX}}{U^{MAX}}$$

Equation GFLIB\_ControllerPIpAW\_F16\_Eq2

where  $K_{P\_SC}$  is the proportional gain parameter considering input/output scaling.

Analogically, scaling the integral term of equation [GFLIB\\_ControllerPipAW\\_eq4](#) results in:

$$u_{if}(k) = u_{if}(k-1) + K_{I,SC} \cdot e_f(k) + K_{I,SC} \cdot e_f(k-1) \quad \text{where } K_{I,SC} = \frac{K_I T_s}{2} \cdot \frac{E^{MAX}}{U^{MAX}}$$

Equation GFLIB\_ControllerPipAW\_F16\_Eq3

where  $K_{I,SC}$  is the integral gain parameter considering input/output scaling.

The sum of the scaled proportional and integral terms gives a complete equation of the controller. The problem is however, that either of the gain parameters  $K_{P,SC}$ ,  $K_{I,SC}$  can be out of the [-1, 1) range, hence can not be directly interpreted as fractional values. To overcome this, it is necessary to scale these gain parameters using the shift values as follows:

$$\begin{aligned} f16PropGain &= K_{P,SC} \cdot 2^{s16PropGainShift} \\ f16IntegGain &= K_{I,SC} \cdot 2^{s16IntegGainShift} \end{aligned}$$

Equation GFLIB\_ControllerPipAW\_F16\_Eq4

where

- f16PropGain - is the scaled value of proportional gain [-1, 1)
- s16PropGainShift - is the scaling shift for proportional gain [-31, 31)
- f16IntegGain - is the scaled value of integral gain [-1, 1)
- s16IntegGainShift - is the scaling shift for integral gain [-31, 31)

The sum of the scaled proportional and integral terms gives a complete equation of the controller:

$$u_f(k) = e_f(k) \cdot K_{P,SC} + u_{if}(k-1) + K_{I,SC} \cdot e_f(k) + K_{I,SC} \cdot e_f(k-1)$$

Equation GFLIB\_ControllerPipAW\_F16\_Eq5

The output signal limitation is implemented in this controller. The actual output  $u(k)$  is bounded not to exceed the given limit values f16UpperLimit, f16LowerLimit. This is due to either the bounded power of the actuator or to the physical constraints of the plant.

$$u(k) = \begin{cases} f16UpperLimit & \text{if } u_f(k) \geq f16UpperLimit \\ u_f(k) & \text{if } f16LowerLimit < u_f(k) < f16UpperLimit \\ f16LowerLimit & \text{if } u_f(k) \leq f16LowerLimit \end{cases}$$

Equation GFLIB\_ControllerPipAW\_F16\_Eq6

The bounds are described by a limitation element equation [GFLIB\\_ControllerPipAW\\_F16\\_Eq6](#). When the bounds are exceeded the non-linear 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.

All controller parameters and states can be reset during declaration using the [GFLIB\\_CONTROLLER\\_PIAW\\_P\\_DEFAULT\\_F16](#) macro.

## Code Example

```
#include "gplib.h"

tFrac16 f16InErr;
tFrac16 f16Output;

GFLIB_CONTROLLER_PIAW_P_T_F16 trMyPI = GFLIB_CONTROLLER_PIAW_P_DEFAULT_F16;

void main(void)
{
    tFrac16 f16ControllerPIpAWOut;

    // Set the input error
    f16InErr = FRAC16(0.25);

    // Initialize the controller parameters
    trMyPI.f16PropGain      = FRAC16(0.01);
    trMyPI.f16IntegGain     = FRAC16(0.02);
    trMyPI.s16PropGainShift = 1;
    trMyPI.s16IntegGainShift = 1;
    trMyPI.f16UpperLimit   = FRAC16(1.0);
    trMyPI.f16LowerLimit   = FRAC16(-1.0);

    // Clear the state variables
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    GFLIB_ControllerPIpAWInit_F16(&trMyPI);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    GFLIB_ControllerPIpAWInit(&trMyPI, F16);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 16-bit fractional implementation is selected as default.
    GFLIB_ControllerPIpAWInit(&trMyPI);

    // Initialize the state variables to predefined values
    f16ControllerPIpAWOut = FRAC16(0.03);
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    GFLIB_ControllerPIpAWSetState_F16(f16ControllerPIpAWOut, &trMyPI);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    GFLIB_ControllerPIpAWSetState(f16ControllerPIpAWOut, &trMyPI, F16);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 16-bit fractional implementation is selected as default.
    GFLIB_ControllerPIpAWSetState(f16ControllerPIpAWOut, &trMyPI);
}

// Periodical function or interrupt - control loop
void ControlLoop(void)
{
    // Alternative 1: API call with postfix
    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    GFLIB_ControllerPIpAWSetState(f16ControllerPIpAWOut, &trMyPI, F16);
}
```

```

// (only one alternative shall be used).
f16Output = GFLIB_ControllerPIpAW_F16(f16InErr, &trMyPI);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
f16Output = GFLIB_ControllerPIpAW(f16InErr, &trMyPI, F16);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 16-bit fractional implementation is selected as default.
f16Output = GFLIB_ControllerPIpAW(f16InErr, &trMyPI);
}

```

### 2.20.3 Function GFLIB\_ControllerPIpAW\_FLT

#### Declaration

```
tFloat GFLIB_ControllerPIpAW_FLT(tFloat fltInErr,
GFLIB_CONTROLLER_PIAW_P_T_FLT *const pParam);
```

#### Arguments

**Table 154. GFLIB\_ControllerPIpAW\_FLT arguments**

Type	Name	Direction	Description
tFloat	fltInErr	input	Input error signal to the controller is a single precision floating point data type.
GFLIB_CONTROLLER_PIAW_P_T_FLT *const	pParam	input, output	Pointer to the controller parameters structure.

#### Return

The function returns a single precision floating point value, representing the signal to be applied to the controlled system so that the input error is forced to zero.

#### Implementation details

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

$$u(k) = \begin{cases} fltUpperLimit & \text{if } u_f(k) \geq fltUpperLimit \\ u_f(k) & \text{if } fltLowerLimit < u_f(k) < fltUpperLimit \\ fltLowerLimit & \text{if } u_f(k) \leq fltLowerLimit \end{cases}$$

Equation GFLIB\_ControllerPIpAW\_FLT\_Eq1

The sum of the scaled proportional and integral terms gives a complete equation of the controller:

$$u(k) = e_f(k) \cdot K_p + u_i(k-1) + e(k) \cdot \frac{K_i T_s}{2} + e(k-1) \cdot \frac{K_i T_s}{2}$$

Equation GFLIB\_ControllerPIpAW\_FLT\_Eq2

The bounds are described by a limitation element equation

[GFLIB\\_ControllerPIpAW\\_FLT\\_Eq2](#). When the bounds are exceeded, the non-linear saturation characteristic will take effect and influence the dynamic behavior. The described limitation is implemented 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.

All controller parameters and states can be reset during declaration using the [GFLIB\\_CONTROLLER\\_PIAW\\_P\\_DEFAULT\\_FLT](#) macro.

**Note:** *The function may raise floating-point exceptions (invalid operation, overflow, underflow, inexact, input denormal). The floating-point unit must be enabled in CPACR register to prevent NOCP UsageFault exception.*

### Code Example

```
#include "gflib.h"

tFloat fltInErr;
tFloat fltOutput;

GFLIB_CONTROLLER_PIAW_P_T_FLT trMyPI = GFLIB_CONTROLLER_PIAW_P_DEFAULT_FLT;

void main(void)
{
    tFloat fltControllerPIpAWOut;

    // Set the input error
    fltInErr = 0.25F;

    // Initialize the controller parameters
    trMyPI.fltPropGain    = 0.04F;
    trMyPI.fltIntegGain   = 0.02F;
    trMyPI.fltUpperLimit = 1.0F;
    trMyPI.fltLowerLimit = -1.0F;

    // Clear the state variables
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    GFLIB\_ControllerPIpAWInit\_FLT(&trMyPI);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    GFLIB\_ControllerPIpAWInit(&trMyPI, FLT);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if single precision floating-point implementation is selected
    // as default.
    GFLIB\_ControllerPIpAWInit(&trMyPI);

    // Initialize the state variables to predefined values
    fltControllerPIpAWOut = 0.03F;
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    GFLIB\_ControllerPIpAWSetState\_FLT(fltControllerPIpAWOut, &trMyPI);
```

```

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
GFLIB_ControllerPIpAWSetState(fltControllerPIpAWOut, &trMyPI, FLT);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if single precision floating-point implementation is selected
// as default.
GFLIB_ControllerPIpAWSetState(fltControllerPIpAWOut, &trMyPI);
}

// Periodical function or interrupt - control loop
void ControlLoop(void)
{
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    fltOutput = GFLIB_ControllerPIpAW_FLT(fltInErr, &trMyPI);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    fltOutput = GFLIB_ControllerPIpAW(fltInErr, &trMyPI, FLT);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if single precision floating-point implementation is selected
    // as default.
    fltOutput = GFLIB_ControllerPIpAW(fltInErr, &trMyPI);
}

```

## 2.20.4 Function GFLIB\_ControllerPIpAWInit

### Description

This function clears the GFLIB\_ControllerPIpAW state variables.

**Note:** The input/output pointer must contain a valid address, otherwise a fault may occur (MemManage, BusFault, UsageFault, HardFault).

### Re-entrancy

The function is re-entrant for a different pCtrl.

#### 2.20.4.1 Function GFLIB\_ControllerPIpAWInit\_F32

##### Declaration

```
void GFLIB_ControllerPIpAWInit_F32(GFLIB\_CONTROLLER\_PIAW\_P\_T\_F32
*const pParam);
```

##### Arguments

Table 155. GFLIB\_ControllerPIpAWInit\_F32 arguments

Type	Name	Direction	Description
<a href="#">GFLIB_CONTROLLER_PIAW_P_T_F32</a> *const	pParam	input, output	Pointer to the structure with GFLIB_ControllerPIpAW state.

## Code Example

```
#include "gplib.h"

tFrac32 f32InErr;
tFrac32 f32Output;

GFLIB_CONTROLLER_PIAW_P_T_F32 trMyPI = GFLIB_CONTROLLER_PIAW_P_DEFAULT_F32;

void main(void)
{
    tFrac32 f32ControllerPIpAWOut;

    // Set the input error
    f32InErr = FRAC32(0.25);

    // Initialize the controller parameters
    trMyPI.f32PropGain      = FRAC32(0.01);
    trMyPI.f32IntegGain     = FRAC32(0.02);
    trMyPI.s16PropGainShift = 1;
    trMyPI.s16IntegGainShift = 1;
    trMyPI.f32UpperLimit   = FRAC32(1.0);
    trMyPI.f32LowerLimit   = FRAC32(-1.0);

    // Clear the state variables
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    GFLIB_ControllerPIpAWInit_F32(&trMyPI);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    GFLIB_ControllerPIpAWInit(&trMyPI, F32);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 32-bit fractional implementation is selected as default.
    GFLIB_ControllerPIpAWInit(&trMyPI);

    // Initialize the state variables to predefined values
    f32ControllerPIpAWOut = FRAC32(0.03);
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    GFLIB_ControllerPIpAWSetState_F32(f32ControllerPIpAWOut, &trMyPI);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    GFLIB_ControllerPIpAWSetState(f32ControllerPIpAWOut, &trMyPI, F32);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 32-bit fractional implementation is selected as default.
    GFLIB_ControllerPIpAWSetState(f32ControllerPIpAWOut, &trMyPI);
}

// Periodical function or interrupt - control loop
void ControlLoop(void)
{
    // Alternative 1: API call with postfix
    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    GFLIB_ControllerPIpAWSetState(f32ControllerPIpAWOut, &trMyPI, F32);
}
```

```

// (only one alternative shall be used).
f32Output = GFLIB\_ControllerPIpAW\_F32(f32InErr, &trMyPI);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
f32Output = GFLIB_ControllerPIpAW(f32InErr, &trMyPI, F32);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 32-bit fractional implementation is selected as default.
f32Output = GFLIB_ControllerPIpAW(f32InErr, &trMyPI);
}

```

#### 2.20.4.2 Function [GFLIB\\_ControllerPIpAWInit\\_F16](#)

##### Declaration

```
void GFLIB_ControllerPIpAWInit_F16(GFLIB\_CONTROLLER\_PIAW\_P\_T\_F16
*const pParam);
```

##### Arguments

**Table 156. [GFLIB\\_ControllerPIpAWInit\\_F16](#) arguments**

Type	Name	Direction	Description
<a href="#">GFLIB_CONTROLLER_PIAW_P_T_F16</a> *const	pParam	input, output	Pointer to the structure with <a href="#">GFLIB_ControllerPIpAW</a> state.

##### Code Example

```

#include "gflib.h"

tFrac16 f16InErr;
tFrac16 f16Output;

GFLIB\_CONTROLLER\_PIAW\_P\_T\_F16 trMyPI = GFLIB\_CONTROLLER\_PIAW\_P\_DEFAULT\_F16;

void main(void)
{
    tFrac16 f16ControllerPIpAWOut;

    // Set the input error
    f16InErr = FRAC16(0.25);

    // Initialize the controller parameters
    trMyPI.f16PropGain      = FRAC16(0.01);
    trMyPI.f16IntegGain     = FRAC16(0.02);
    trMyPI.s16PropGainShift = 1;
    trMyPI.s16IntegGainShift = 1;
    trMyPI.f16UpperLimit   = FRAC16(1.0);
    trMyPI.f16LowerLimit   = FRAC16(-1.0);

    // Clear the state variables
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    GFLIB_ControllerPIpAWInit_F16(&trMyPI);
}

```

```

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
GFLIB_ControllerPIpAWInit(&trMyPI, F16);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 16-bit fractional implementation is selected as default.
GFLIB_ControllerPIpAWInit(&trMyPI);

// Initialize the state variables to predefined values
f16ControllerPIpAWOut = FRAC16(0.03);
// Alternative 1: API call with postfix
// (only one alternative shall be used).
GFLIB\_ControllerPIpAWSetState\_F16(f16ControllerPIpAWOut, &trMyPI);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
GFLIB_ControllerPIpAWSetState(f16ControllerPIpAWOut, &trMyPI, F16);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 16-bit fractional implementation is selected as default.
GFLIB_ControllerPIpAWSetState(f16ControllerPIpAWOut, &trMyPI);
}

// Periodical function or interrupt - control loop
void ControlLoop(void)
{
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    f16Output = GFLIB\_ControllerPIpAW\_F16(f16InErr, &trMyPI);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    f16Output = GFLIB_ControllerPIpAW(f16InErr, &trMyPI, F16);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 16-bit fractional implementation is selected as default.
    f16Output = GFLIB_ControllerPIpAW(f16InErr, &trMyPI);
}

```

#### 2.20.4.3 Function [GFLIB\\_ControllerPIpAWInit\\_FLT](#)

##### Declaration

```
void GFLIB_ControllerPIpAWInit_FLT(GFLIB\_CONTROLLER\_PIAW\_P\_T\_FLT
*const pParam);
```

##### Arguments

**Table 157. GFLIB\_ControllerPIpAWInit\_FLT arguments**

Type	Name	Direction	Description
<a href="#">GFLIB_CONTROLLER_PIAW_P_T_FLT</a> *const	pParam	input, output	Pointer to the structure with GFLIB_ControllerPIpAW state.

## Code Example

```
#include "gplib.h"

tFloat fltInErr;
tFloat fltOutput;

GFLIB_CONTROLLER_PIAW_P_T_FLT trMyPI = GFLIB_CONTROLLER_PIAW_P_DEFAULT_FLT;

void main(void)
{
    tFloat fltControllerPIpAWOut;

    // Set the input error
    fltInErr = 0.25F;

    // Initialize the controller parameters
    trMyPI.fltPropGain    = 0.04F;
    trMyPI.fltIntegGain   = 0.02F;
    trMyPI.fltUpperLimit  = 1.0F;
    trMyPI.fltLowerLimit  = -1.0F;

    // Clear the state variables
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    GFLIB_ControllerPIpAWInit_FLT(&trMyPI);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    GFLIB_ControllerPIpAWInit(&trMyPI, FLT);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if single precision floating-point implementation is selected
    // as default.
    GFLIB_ControllerPIpAWInit(&trMyPI);

    // Initialize the state variables to predefined values
    fltControllerPIpAWOut = 0.03F;
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    GFLIB_ControllerPIpAWSetState_FLT(fltControllerPIpAWOut, &trMyPI);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    GFLIB_ControllerPIpAWSetState(fltControllerPIpAWOut, &trMyPI, FLT);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if single precision floating-point implementation is selected
    // as default.
    GFLIB_ControllerPIpAWSetState(fltControllerPIpAWOut, &trMyPI);
}

// Periodical function or interrupt - control loop
void ControlLoop(void)
{
    // Alternative 1: API call with postfix
```

```

// (only one alternative shall be used).
fltOutput = GFLIB_ControllerPIpAW_FLT(fltInErr, &trMyPI);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
fltOutput = GFLIB_ControllerPIpAW(fltInErr, &trMyPI, FLT);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if single precision floating-point implementation is selected
// as default.
fltOutput = GFLIB_ControllerPIpAW(fltInErr, &trMyPI);
}

```

## 2.20.5 Function GFLIB\_ControllerPIpAWSetState

### Description

This function initializes the GFLIB\_ControllerPIpAW state variables to achieve the required output values.

**Note:** The input/output pointer must contain a valid address, otherwise a fault may occur (MemManage, BusFault, UsageFault, HardFault).

### Re-entrancy

The function is re-entrant for a different pCtrl.

### 2.20.5.1 Function GFLIB\_ControllerPIpAWSetState\_F32

#### Declaration

```
void GFLIB_ControllerPIpAWSetState_F32(tFrac32
f32ControllerPIpAWOut, GFLIB_CONTROLLER_PIAW_P_T_F32 *const
pParam);
```

#### Arguments

Table 158. GFLIB\_ControllerPIpAWSetState\_F32 arguments

Type	Name	Direction	Description
<u>tFrac32</u>	f32ControllerPIpAWOut	<b>input</b>	Required output of the GFLIB_ControllerPIpAW.
<u>GFLIB_CONTROLLER_PIAW_P_T_F32</u> *const	pParam	<b>input, output</b>	Pointer to the structure with GFLIB_ControllerPIpAW state.

### Code Example

```

#include "gflib.h"

tFrac32 f32InErr;
tFrac32 f32Output;

GFLIB_CONTROLLER_PIAW_P_T_F32 trMyPI = GFLIB_CONTROLLER_PIAW_P_DEFAULT_F32;

void main(void)

```

```
{  
    tFrac32 f32ControllerPIpAWOut;  
  
    // Set the input error  
    f32InErr = FRAC32(0.25);  
  
    // Initialize the controller parameters  
    trMyPI.f32PropGain      = FRAC32(0.01);  
    trMyPI.f32IntegGain     = FRAC32(0.02);  
    trMyPI.s16PropGainShift = 1;  
    trMyPI.s16IntegGainShift = 1;  
    trMyPI.f32UpperLimit   = FRAC32(1.0);  
    trMyPI.f32LowerLimit   = FRAC32(-1.0);  
  
    // Clear the state variables  
    // Alternative 1: API call with postfix  
    // (only one alternative shall be used).  
    GFLIB_ControllerPIpAWInit_F32(&trMyPI);  
  
    // Alternative 2: API call with implementation parameter  
    // (only one alternative shall be used).  
    GFLIB_ControllerPIpAWInit(&trMyPI, F32);  
  
    // Alternative 3: API call with global configuration of implementation  
    // (only one alternative shall be used). This alternative is available  
    // only if 32-bit fractional implementation is selected as default.  
    GFLIB_ControllerPIpAWInit(&trMyPI);  
  
    // Initialize the state variables to predefined values  
    f32ControllerPIpAWOut = FRAC32(0.03);  
    // Alternative 1: API call with postfix  
    // (only one alternative shall be used).  
    GFLIB_ControllerPIpAWSetState_F32(f32ControllerPIpAWOut, &trMyPI);  
  
    // Alternative 2: API call with implementation parameter  
    // (only one alternative shall be used).  
    GFLIB_ControllerPIpAWSetState(f32ControllerPIpAWOut, &trMyPI, F32);  
  
    // Alternative 3: API call with global configuration of implementation  
    // (only one alternative shall be used). This alternative is available  
    // only if 32-bit fractional implementation is selected as default.  
    GFLIB_ControllerPIpAWSetState(f32ControllerPIpAWOut, &trMyPI);  
}  
  
// Periodical function or interrupt - control loop  
void ControlLoop(void)  
{  
    // Alternative 1: API call with postfix  
    // (only one alternative shall be used).  
    f32Output = GFLIB_ControllerPIpAW_F32(f32InErr, &trMyPI);  
  
    // Alternative 2: API call with implementation parameter  
    // (only one alternative shall be used).  
    f32Output = GFLIB_ControllerPIpAW(f32InErr, &trMyPI, F32);  
  
    // Alternative 3: API call with global configuration of implementation  
    // (only one alternative shall be used). This alternative is available  
    // only if 32-bit fractional implementation is selected as default.  
    f32Output = GFLIB_ControllerPIpAW(f32InErr, &trMyPI);
```

```
}
```

### 2.20.5.2 Function `GFLIB_ControllerPIpAWSetState_F16`

#### Declaration

```
void GFLIB_ControllerPIpAWSetState_F16(tFrac16  
f16ControllerPIpAWOut, GFLIB_CONTROLLER_PIAW_P_T_F16 *const  
pParam);
```

#### Arguments

**Table 159. `GFLIB_ControllerPIpAWSetState_F16` arguments**

Type	Name	Direction	Description
<u>tFrac16</u>	f16ControllerPIpAWOut	<b>input</b>	Required output of the <code>GFLIB_ControllerPIpAW</code> .
<u>GFLIB_CONTROLLER_PIAW_P_T_F16</u> *const	pParam	<b>input, output</b>	Pointer to the structure with <code>GFLIB_ControllerPIpAW</code> state.

#### Code Example

```
#include "gflib.h"

tFrac16 f16InErr;
tFrac16 f16Output;

GFLIB_CONTROLLER_PIAW_P_T_F16 trMyPI = GFLIB_CONTROLLER_PIAW_P_DEFAULT_F16;

void main(void)
{
    tFrac16 f16ControllerPIpAWOut;

    // Set the input error
    f16InErr = FRAC16(0.25);

    // Initialize the controller parameters
    trMyPI.f16PropGain      = FRAC16(0.01);
    trMyPI.f16IntegGain     = FRAC16(0.02);
    trMyPI.s16PropGainShift = 1;
    trMyPI.s16IntegGainShift = 1;
    trMyPI.f16UpperLimit   = FRAC16(1.0);
    trMyPI.f16LowerLimit   = FRAC16(-1.0);

    // Clear the state variables
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    GFLIB_ControllerPIpAWInit_F16(&trMyPI);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    GFLIB_ControllerPIpAWInit(&trMyPI, F16);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 16-bit fractional implementation is selected as default.
    GFLIB_ControllerPIpAWInit(&trMyPI);
```

```

// Initialize the state variables to predefined values
f16ControllerPIpAWOut = FRAC16(0.03);
// Alternative 1: API call with postfix
// (only one alternative shall be used).
GFLIB_ControllerPIpAWSetState_F16(f16ControllerPIpAWOut, &trMyPI);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
GFLIB_ControllerPIpAWSetState(f16ControllerPIpAWOut, &trMyPI, F16);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 16-bit fractional implementation is selected as default.
GFLIB_ControllerPIpAWSetState(f16ControllerPIpAWOut, &trMyPI);
}

// Periodical function or interrupt - control loop
void ControlLoop(void)
{
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    f16Output = GFLIB\_ControllerPIpAW\_F16(f16InErr, &trMyPI);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    f16Output = GFLIB_ControllerPIpAW(f16InErr, &trMyPI, F16);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 16-bit fractional implementation is selected as default.
    f16Output = GFLIB_ControllerPIpAW(f16InErr, &trMyPI);
}

```

### 2.20.5.3 Function GFLIB\_ControllerPIpAWSetState\_FLT

#### Declaration

```
void GFLIB_ControllerPIpAWSetState_FLT(tFloat
fltControllerPIpAWOut, GFLIB\_CONTROLLER\_PIAW\_P\_T\_FLT *const
pParam);
```

#### Arguments

**Table 160. GFLIB\_ControllerPIpAWSetState\_FLT arguments**

Type	Name	Direction	Description
<a href="#">tFloat</a>	fltControllerPIpAWOut	input	Required output of the GFLIB_ControllerPIpAW.
<a href="#">GFLIB_CONTROLLER_PIAW_P_T_FLT</a> *const	pParam	input, output	Pointer to the structure with GFLIB_ControllerPIpAW state.

#### Code Example

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

```
tFloat fltInErr;
tFloat fltOutput;

GFLIB_CONTROLLER_PIAW_P_T_FLT trMyPI = GFLIB_CONTROLLER_PIAW_P_DEFAULT_FLT;

void main(void)
{
    tFloat fltControllerPIpAWOut;

    // Set the input error
    fltInErr = 0.25F;

    // Initialize the controller parameters
    trMyPI.fltPropGain = 0.04F;
    trMyPI.fltIntegGain = 0.02F;
    trMyPI.fltUpperLimit = 1.0F;
    trMyPI.fltLowerLimit = -1.0F;

    // Clear the state variables
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    GFLIB_ControllerPIpAWInit_FLT(&trMyPI);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    GFLIB_ControllerPIpAWInit(&trMyPI, FLT);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if single precision floating-point implementation is selected
    // as default.
    GFLIB_ControllerPIpAWInit(&trMyPI);

    // Initialize the state variables to predefined values
    fltControllerPIpAWOut = 0.03F;
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    GFLIB_ControllerPIpAWSetState_FLT(fltControllerPIpAWOut, &trMyPI);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    GFLIB_ControllerPIpAWSetState(fltControllerPIpAWOut, &trMyPI, FLT);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if single precision floating-point implementation is selected
    // as default.
    GFLIB_ControllerPIpAWSetState(fltControllerPIpAWOut, &trMyPI);
}

// Periodical function or interrupt - control loop
void ControlLoop(void)
{
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    fltOutput = GFLIB_ControllerPIpAW_FLT(fltInErr, &trMyPI);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
```

```

    fltOutput = GFLIB_ControllerPIpAW(fltInErr, &trMyPI, FLT);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if single precision floating-point implementation is selected
    // as default.
    fltOutput = GFLIB_ControllerPIpAW(fltInErr, &trMyPI);
}

```

## 2.21 Function GFLIB\_ControllerPIr

This function calculates a standard recurrent form of the Proportional-Integral controller, without integral anti-windup.

### Description

The function GFLIB\_ControllerPIr calculates a standard recurrent form of the Proportional-Integral controller, without integral anti-windup.

The continuous time domain representation of the PI controller is defined as:

$$u(t) = e(t) \cdot K_p + K_i \int_0^t e(t) dt$$

Equation GFLIB\_ControllerPIr\_Eq1

The transfer function for this kind of PI controller, in a continuous time domain, is described using the Laplace transformation as follows:

$$H(s) = \frac{U(s)}{E(s)} = \frac{sK_p+K_i}{s}$$

Equation GFLIB\_ControllerPIr\_Eq2

Transforming equation [GFLIB\\_ControllerPIr\\_Eq2](#) into a discrete time domain leads to the following equation:

$$u(k) = u(k-1) + e(k) \cdot CC1 + e(k-1) \cdot CC2$$

Equation GFLIB\_ControllerPIr\_Eq3

where  $K_p$  is proportional gain,  $K_i$  is integral gain,  $T_s$  is the sampling period,  $u(k)$  is the controller output,  $e(k)$  is the controller input error signal,  $CC1$  and  $CC2$  are controller coefficients calculated depending on the discretization method used, as shown in [Table 161](#).

**Table 161. Calculation of coefficients CC1 and CC2 using various discretization methods**

	Trapezoidal	Backward Rect.	Forward Rect.
CC1=	$K_p + K_i T_s / 2$	$K_p + K_i T_s$	$K_p$
CC2=	$-K_p + K_i T_s / 2$	$-K_p$	$-K_p + K_i T_s$

**Note:** The input pointer must contain a valid address otherwise a fault may occur (MemManage, BusFault, UsageFault, HardFault).

### Re-entrancy

The function is re-entrant.

### 2.21.1 Function `GFLIB_ControllerPlr_F32`

#### Declaration

```
tFrac32 GFLIB_ControllerPlr_F32 (tFrac32 f32InErr,
GFLIB_CONTROLLER_PI_R_T_F32 *const pParam);
```

#### Arguments

Table 162. `GFLIB_ControllerPlr_F32` arguments

Type	Name	Direction	Description
<code>tFrac32</code>	<code>f32InErr</code>	<code>input</code>	Input error signal to the controller is a 32-bit number normalized between [-1, 1).
<code>GFLIB_CONTROLLER_PI_R_T_F32 *const</code>	<code>pParam</code>	<code>input, output</code>	Pointer to the controller parameters structure.

#### Return

The function returns a 32-bit value in fractional format 1.31, representing the signal to be applied to the controlled system so that the input error is forced to zero.

#### Implementation details

In order to implement the discrete equation of the controller [GFLIB\\_ControllerPlr\\_eq3](#) on the fixed point arithmetic platform, the maximal values (scales) of the input and output signals

- $E^{MAX}$  - maximal value of the controller input error signal
- $U^{MAX}$  - maximal value of the controller output signal

have to be known a priori. This is essential for correct casting of the physical signal values into fixed point values [-1, 1].

Then the fractional representation [-1, 1) of both input and output signals is obtained as follows:

$$e_f(k) = \frac{e(k)}{E^{MAX}}$$

$$u_f(k) = \frac{u(k)}{U^{MAX}}$$

Equation `GFLIB_ControllerPlr_F32_Eq1`

The resulting controller discrete time domain equation in fixed point fractional representation is therefore given as:

$$u_f(k) \cdot U^{MAX} = u_f(k-1) \cdot U^{MAX} + e_f(k) \cdot E^{MAX} \cdot CC1 + e_f(k-1) \cdot E^{MAX} \cdot CC2$$

Equation `GFLIB_ControllerPlr_F32_Eq2`

which can be rearranged into the following form:

$$u_f(k) = u_f(k-1) + e_f(k) \cdot CC1_f + e_f(k-1) \cdot CC2_f$$

Equation `GFLIB_ControllerPlr_F32_Eq3`

where

$$CC1_f = CC1 \cdot \frac{E^{MAX}}{U^{MAX}}$$

$$CC2_f = CC2 \cdot \frac{E^{MAX}}{U^{MAX}}$$

Equation GFLIB\_ControllerPIr\_F32\_Eq4

are the controller coefficients adapted according to the input and output scale values. In order to implement both coefficients as fractional numbers, both  $CC1_f$  and  $CC2_f$  must reside in the fractional range [-1, 1). However, depending on values  $CC1$ ,  $CC2$ ,  $E^{MAX}$ ,  $U^{MAX}$ , the calculation of  $CC1_f$  and  $CC2_f$  may result in values outside this fractional range. Therefore, a scaling of  $CC1_f$ ,  $CC2_f$  is introduced as follows:

$$f32CC1sc = CC1_f \cdot 2^{u16NShift}$$

$$f32CC2sc = CC2_f \cdot 2^{u16NShift}$$

Equation GFLIB\_ControllerPIr\_F32\_Eq5

The introduced scaling shift  $u16NShift$  is chosen such that both coefficients  $f32CC1sc$ ,  $f32CC2sc$  reside in the range [-1, 1). To simplify the implementation, this scaling shift is chosen to be a power of 2, so the final scaling is a simple shift operation. Moreover, the scaling shift cannot be a negative number, so the operation of scaling is always to scale numbers with an absolute value larger than 1 down to fit in the range [-1, 1).

$$u16Nshift = \max\left(\left\lceil \frac{\log(\text{abs}(CC1_f))}{\log(2)} \right\rceil, \left\lceil \frac{\log(\text{abs}(CC2_f))}{\log(2)} \right\rceil\right)$$

Equation GFLIB\_ControllerPIr\_F32\_Eq6

The final, scaled, fractional equation of a recurrent PI controller on a 32-bit fixed point platform is therefore implemented as follows:

$$u_f(k) \cdot 2^{u16NShift} = u_f(k-1) \cdot 2^{u16NShift} + e_f(k) \cdot f32CC1sc + e_f(k-1) \cdot f32CC2sc$$

Equation GFLIB\_ControllerPIr\_F32\_Eq7

where:

- $u_f(k)$  - fractional representation [-1, 1) of the controller output
- $e_f(k)$  - fractional representation [-1, 1) of the controller input (error)
- $f32CC1sc$  - fractional representation [-1, 1) of the 1st controller coefficient
- $f32CC2sc$  - fractional representation [-1, 1) of the 2nd controller coefficient
- $u16NShift$  - in range [0,31] - is chosen such that both coefficients  $f32CC1sc$  and  $f32CC2sc$  are in the range [-1, 1)

All controller parameters and states can be reset during declaration using the [GFLIB\\_CONTROLLER\\_PI\\_R\\_DEFAULT\\_F32](#) macro.

### Code Example

```
#include "gplib.h"

tFrac32 f32InErr;
tFrac32 f32Output;

GFLIB_CONTROLLER_PI_R_T_F32 trMyPI = GFLIB_CONTROLLER_PI_R_DEFAULT_F32;
```

```
void main(void)
{
    tFrac32 f32ControllerPIrOut;

    // Set the input error
    f32InErr = FRAC32(0.25);

    // Initialize the controller parameters
    trMyPI.f32CC1sc = FRAC32(0.01);
    trMyPI.f32CC2sc = FRAC32(0.02);
    trMyPI.ul6NShift = 1;

    // Clear the state variables
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    GFLIB_ControllerPIrInit_F32(&trMyPI);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    GFLIB_ControllerPIrInit(&trMyPI, F32);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 32-bit fractional implementation is selected as default.
    GFLIB_ControllerPIrInit(&trMyPI);

    // Initialize the state variables to predefined values
    f32ControllerPIrOut = FRAC32(0.03);
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    GFLIB_ControllerPIrSetState_F32(f32ControllerPIrOut, &trMyPI);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    GFLIB_ControllerPIrSetState(f32ControllerPIrOut, &trMyPI, F32);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 32-bit fractional implementation is selected as default.
    GFLIB_ControllerPIrSetState(f32ControllerPIrOut, &trMyPI);
}

// Periodical function or interrupt - control loop
void ControlLoop(void)
{
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    f32Output = GFLIB_ControllerPIr_F32(f32InErr, &trMyPI);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    f32Output = GFLIB_ControllerPIr(f32InErr, &trMyPI, F32);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 32-bit fractional implementation is selected as default.
    f32Output = GFLIB_ControllerPIr(f32InErr, &trMyPI);
}
```

## 2.21.2 Function `GFLIB_ControllerPIr_F16`

### Declaration

```
tFrac16 GFLIB_ControllerPIr_F16(tFrac16 f16InErr,
GFLIB_CONTROLLER_PI_R_T_F16 *const pParam);
```

### Arguments

Table 163. `GFLIB_ControllerPIr_F16` arguments

Type	Name	Direction	Description
<code>tFrac16</code>	<code>f16InErr</code>	<code>input</code>	Input error signal to the controller is a 16-bit number normalized between [-1, 1).
<code>GFLIB_CONTROLLER_PI_R_T_F16 *const</code>	<code>pParam</code>	<code>input, output</code>	Pointer to the controller parameters structure.

### Return

The function returns a 16-bit value in fractional format 1.15, representing the signal to be applied to the controlled system so that the input error is forced to zero.

### Implementation details

In order to implement the discrete equation of the controller [GFLIB\\_ControllerPIr\\_eq3](#) on the fixed point arithmetic platform, the maximal values (scales) of the input and output signals

- $E^{MAX}$  - maximal value of the controller input error signal
- $U^{MAX}$  - maximal value of the controller output signal

have to be known a priori. This is essential for correct casting of the physical signal values into fixed point values [-1, 1].

Then the fractional representation [-1, 1) of both input and output signals is obtained as follows:

$$e_f(k) = \frac{e(k)}{E^{MAX}}$$

$$u_f(k) = \frac{u(k)}{U^{MAX}}$$

Equation `GFLIB_ControllerPIr_F16_Eq1`

The resulting controller discrete time domain equation in fixed point fractional representation is therefore given as:

$$u_f(k) \cdot U^{MAX} = u_f(k-1) \cdot U^{MAX} + e_f(k) \cdot E^{MAX} \cdot CC1 + e_f(k-1) \cdot E^{MAX} \cdot CC2$$

Equation `GFLIB_ControllerPIr_F16_Eq2`

which can be rearranged into the following form:

$$u_f(k) = u_f(k-1) + e_f(k) \cdot CC1_f + e_f(k-1) \cdot CC2_f$$

Equation `GFLIB_ControllerPIr_F16_Eq3`

where

$$CC1_f = CC1 \cdot \frac{E^{MAX}}{U^{MAX}}$$

$$CC2_f = CC2 \cdot \frac{E^{MAX}}{U^{MAX}}$$

Equation GFLIB\_ControllerPIr\_F16\_Eq4

are the controller coefficients adapted according to the input and output scale values. In order to implement both coefficients as fractional numbers, both  $CC1_f$  and  $CC2_f$  must reside in the fractional range [-1, 1). However, depending on values  $CC1$ ,  $CC2$ ,  $E^{MAX}$ ,  $U^{MAX}$ , the calculation of  $CC1_f$  and  $CC2_f$  may result in values outside this fractional range. Therefore, a scaling of  $CC1_f$ ,  $CC2_f$  is introduced as follows:

$$f16CC1sc = CC1_f \cdot 2^{u16NShift}$$

$$f16CC2sc = CC2_f \cdot 2^{u16NShift}$$

Equation GFLIB\_ControllerPIr\_F16\_Eq5

The introduced scaling shift  $u16NShift$  is chosen such that both coefficients  $f16CC1sc$ ,  $f16CC2sc$  reside in the range [-1, 1). To simplify the implementation, this scaling shift is chosen to be a power of 2, so the final scaling is a simple shift operation. Moreover, the scaling shift cannot be a negative number, so the operation of scaling is always to scale numbers with an absolute value larger than 1 down to fit in the range [-1, 1).

$$u16Nshift = \max\left(\left\lceil \frac{\log(\text{abs}(CC1_f))}{\log(2)} \right\rceil, \left\lceil \frac{\log(\text{abs}(CC2_f))}{\log(2)} \right\rceil\right)$$

Equation GFLIB\_ControllerPIr\_F16\_Eq6

The final, scaled, fractional equation of a recurrent PI controller on a 16-bit fixed point platform is therefore implemented as follows:

$$u_f(k) \cdot 2^{u16NShift} = u_f(k-1) \cdot 2^{u16NShift} + e_f(k) \cdot f16CC1sc + e_f(k-1) \cdot f16CC2sc$$

Equation GFLIB\_ControllerPIr\_F16\_Eq7

where:

- $u_f(k)$  - fractional representation [-1, 1) of the controller output
- $e_f(k)$  - fractional representation [-1, 1) of the controller input (error)
- $f16CC1sc$  - fractional representation [-1, 1) of the 1st controller coefficient
- $f16CC2sc$  - fractional representation [-1, 1) of the 2nd controller coefficient
- $u16NShift$  - in range [0,15] - is chosen such that both coefficients  $f16CC1sc$  and  $f16CC2sc$  are in the range [-1, 1)

All controller parameters and states can be reset during declaration using the [GFLIB\\_CONTROLLER\\_PI\\_R\\_DEFAULT\\_F16](#) macro.

### Code Example

```
#include "gflib.h"

tFrac16 f16InErr;
tFrac16 f16Output;

GFLIB_CONTROLLER_PI_R_T_F16 trMyPI = GFLIB_CONTROLLER_PI_R_DEFAULT_F16;
```

```
void main(void)
{
    tFrac16 f16ControllerPIrOut;

    // Set the input error
    f16InErr = FRAC16(0.25);

    // Initialize the controller parameters
    trMyPI.f16CC1sc = FRAC16(0.01);
    trMyPI.f16CC2sc = FRAC16(0.02);
    trMyPI.ul6NShift = 1;

    // Clear the state variables
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    GFLIB_ControllerPIrInit_F16(&trMyPI);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    GFLIB_ControllerPIrInit(&trMyPI, F16);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 16-bit fractional implementation is selected as default.
    GFLIB_ControllerPIrInit(&trMyPI);

    // Initialize the state variables to predefined values
    f16ControllerPIrOut = FRAC16(0.03);
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    GFLIB_ControllerPIrSetState_F16(f16ControllerPIrOut, &trMyPI);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    GFLIB_ControllerPIrSetState(f16ControllerPIrOut, &trMyPI, F16);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 16-bit fractional implementation is selected as default.
    GFLIB_ControllerPIrSetState(f16ControllerPIrOut, &trMyPI);
}

// Periodical function or interrupt - control loop
void ControlLoop(void)
{
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    f16Output = GFLIB_ControllerPIr_F16(f16InErr, &trMyPI);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    f16Output = GFLIB_ControllerPIr(f16InErr, &trMyPI, F16);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 16-bit fractional implementation is selected as default.
    f16Output = GFLIB_ControllerPIr(f16InErr, &trMyPI);
}
```

### 2.21.3 Function `GFLIB_ControllerPIr_FLT`

#### Declaration

```
tFloat GFLIB_ControllerPIr_FLT(tFloat fltInErr,
GFLIB_CONTROLLER_PI_R_T_FLT *const pParam);
```

#### Arguments

Table 164. `GFLIB_ControllerPIr_FLT` arguments

Type	Name	Direction	Description
<code>tFloat</code>	<code>fltInErr</code>	<code>input</code>	Input error signal to the controller as a single precision floating point value.
<code>GFLIB_CONTROLLER_PI_R_T_FLT *const</code>	<code>pParam</code>	<code>input, output</code>	Pointer to the controller parameters structure.

#### Return

The function returns a single precision floating point value, representing the signal to be applied to the controlled system so that the input error is forced to zero.

#### Implementation details

All controller parameters and states can be reset during declaration using the `GFLIB_CONTROLLER_PI_R_DEFAULT_FLT` macro.

**Note:** The function may raise floating-point exceptions (invalid operation, overflow, underflow, inexact, input denormal). The floating-point unit must be enabled in CPACR register to prevent NOCP UsageFault exception.

#### Code Example

```
#include "gflib.h"

tFloat fltInErr;
tFloat fltOutput;

GFLIB_CONTROLLER_PI_R_T_FLT trMyPI = GFLIB_CONTROLLER_PI_R_DEFAULT_FLT;

void main(void)
{
    tFloat fltControllerPIrOut;

    // Set the input error
    fltInErr = 0.25f;

    // Initialize the controller parameters
    trMyPI.fltCC1sc = 0.01f;
    trMyPI.fltCC2sc = 0.02f;

    // Clear the state variables
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    GFLIB_ControllerPIrInit_FLT(&trMyPI);

    // Alternative 2: API call with implementation parameter
```

```
// (only one alternative shall be used).
GFLIB_ControllerPIrInit(&trMyPI, FLT);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if single precision floating-point implementation is selected
// as default.
GFLIB_ControllerPIrInit(&trMyPI);

// Initialize the state variables to predefined values
fltControllerPIrOut = 0.03f;
// Alternative 1: API call with postfix
// (only one alternative shall be used).
GFLIB_ControllerPIrSetState_FLT(fltControllerPIrOut, &trMyPI);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
GFLIB_ControllerPIrSetState(fltControllerPIrOut, &trMyPI, FLT);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if single precision floating-point implementation is selected
// as default.
GFLIB_ControllerPIrSetState(fltControllerPIrOut, &trMyPI);
}

// Periodical function or interrupt - control loop
void ControlLoop(void)
{
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    fltOutput = GFLIB_ControllerPIr_FLT(fltInErr, &trMyPI);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    fltOutput = GFLIB_ControllerPIr(fltInErr, &trMyPI, FLT);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if single precision floating-point implementation is selected
    // as default.
    fltOutput = GFLIB_ControllerPIr(fltInErr, &trMyPI);
}
```

## 2.21.4 Function GFLIB\_ControllerPIrInit

### Description

This function clears the GFLIB\_ControllerPIr state variables.

**Note:** The input/output pointer must contain a valid address, otherwise a fault may occur (MemManage, BusFault, UsageFault, HardFault).

### Re-entrancy

The function is re-entrant for a different pCtrl.

### 2.21.4.1 Function `GFLIB_ControllerPIrInit_F32`

#### Declaration

```
void GFLIB_ControllerPIrInit_F32(GFLIB\_CONTROLLER\_PI\_R\_T\_F32  
*const pParam);
```

#### Arguments

**Table 165. `GFLIB_ControllerPIrInit_F32` arguments**

Type	Name	Direction	Description
<a href="#">GFLIB_CONTROLLER_PI_R_T_F32</a> *const	pParam	input, output	Pointer to the structure with <code>GFLIB_ControllerPIr</code> state.

#### Code Example

```
#include "gflib.h"

tFrac32 f32InErr;
tFrac32 f32Output;

GFLIB\_CONTROLLER\_PI\_R\_T\_F32 trMyPI = GFLIB\_CONTROLLER\_PI\_R\_DEFAULT\_F32;

void main(void)
{
    tFrac32 f32ControllerPIrOut;

    // Set the input error
    f32InErr = FRAC32(0.25);

    // Initialize the controller parameters
    trMyPI.f32CC1sc = FRAC32(0.01);
    trMyPI.f32CC2sc = FRAC32(0.02);
    trMyPI.ul6NShift = 1;

    // Clear the state variables
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    GFLIB_ControllerPIrInit_F32(&trMyPI);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    GFLIB_ControllerPIrInit(&trMyPI, F32);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 32-bit fractional implementation is selected as default.
    GFLIB_ControllerPIrInit(&trMyPI);

    // Initialize the state variables to predefined values
    f32ControllerPIrOut = FRAC32(0.03);
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    GFLIB\_ControllerPIrSetState\_F32(f32ControllerPIrOut, &trMyPI);
```

```

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
GFLIB_ControllerPIrSetState(f32ControllerPIrOut, &trMyPI, F32);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 32-bit fractional implementation is selected as default.
GFLIB_ControllerPIrSetState(f32ControllerPIrOut, &trMyPI);
}

// Periodical function or interrupt - control loop
void ControlLoop(void)
{
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    f32Output = GFLIB\_ControllerPIr\_F32(f32InErr, &trMyPI);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    f32Output = GFLIB_ControllerPIr(f32InErr, &trMyPI, F32);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 32-bit fractional implementation is selected as default.
    f32Output = GFLIB_ControllerPIr(f32InErr, &trMyPI);
}

```

#### 2.21.4.2 Function [GFLIB\\_ControllerPIrInit\\_F16](#)

##### Declaration

```
void GFLIB_ControllerPIrInit_F16(GFLIB\_CONTROLLER\_PI\_R\_T\_F16  
*const pParam);
```

##### Arguments

Table 166. [GFLIB\\_ControllerPIrInit\\_F16](#) arguments

Type	Name	Direction	Description
<a href="#">GFLIB_CONTROLLER_PI_R_T_F16</a> *const	pParam	input, output	Pointer to the structure with GFLIB_ControllerPIr state.

##### Code Example

```

#include "gflib.h"

tFrac16 f16InErr;
tFrac16 f16Output;

GFLIB\_CONTROLLER\_PI\_R\_T\_F16 trMyPI = GFLIB\_CONTROLLER\_PI\_R\_DEFAULT\_F16;

void main(void)
{
    tFrac16 f16ControllerPIrOut;

    // Set the input error
    f16InErr = FRAC16(0.25);
}

```

```
// Initialize the controller parameters
trMyPI.f16CC1sc = FRAC16(0.01);
trMyPI.f16CC2sc = FRAC16(0.02);
trMyPI.ul6NShift = 1;

// Clear the state variables
// Alternative 1: API call with postfix
// (only one alternative shall be used).
GFLIB_ControllerPIrInit_F16(&trMyPI);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
GFLIB_ControllerPIrInit(&trMyPI, F16);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 16-bit fractional implementation is selected as default.
GFLIB_ControllerPIrInit(&trMyPI);

// Initialize the state variables to predefined values
f16ControllerPIrOut = FRAC16(0.03);
// Alternative 1: API call with postfix
// (only one alternative shall be used).
GFLIB_ControllerPIrSetState_F16(f16ControllerPIrOut, &trMyPI);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
GFLIB_ControllerPIrSetState(f16ControllerPIrOut, &trMyPI, F16);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 16-bit fractional implementation is selected as default.
GFLIB_ControllerPIrSetState(f16ControllerPIrOut, &trMyPI);
}

// Periodical function or interrupt - control loop
void ControlLoop(void)
{
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    f16Output = GFLIB_ControllerPIr_F16(f16InErr, &trMyPI);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    f16Output = GFLIB_ControllerPIr(f16InErr, &trMyPI, F16);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 16-bit fractional implementation is selected as default.
    f16Output = GFLIB_ControllerPIr(f16InErr, &trMyPI);
}
```

### 2.21.4.3 Function `GFLIB_ControllerPIrInit_FLT`

#### Declaration

```
void GFLIB_ControllerPIrInit_FLT(GFLIB\_CONTROLLER\_PI\_R\_T\_FLT  
*const pParam);
```

#### Arguments

**Table 167. `GFLIB_ControllerPIrInit_FLT` arguments**

Type	Name	Direction	Description
<a href="#">GFLIB_CONTROLLER_PI_R_T_FLT</a> *const	pParam	input, output	Pointer to the structure with <code>GFLIB_ControllerPIr</code> state.

#### Code Example

```
#include "gflib.h"

tFloat fltInErr;
tFloat fltOutput;

GFLIB\_CONTROLLER\_PI\_R\_T\_FLT trMyPI = GFLIB\_CONTROLLER\_PI\_R\_DEFAULT\_FLT;

void main(void)
{
    tFloat fltControllerPIrOut;

    // Set the input error
    fltInErr = 0.25f;

    // Initialize the controller parameters
    trMyPI.fltCC1sc = 0.01f;
    trMyPI.fltCC2sc = 0.02f;

    // Clear the state variables
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    GFLIB_ControllerPIrInit_FLT(&trMyPI);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    GFLIB_ControllerPIrInit(&trMyPI, FLT);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if single precision floating-point implementation is selected
    // as default.
    GFLIB_ControllerPIrInit(&trMyPI);

    // Initialize the state variables to predefined values
    fltControllerPIrOut = 0.03f;
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    GFLIB\_ControllerPIrSetState\_FLT(fltControllerPIrOut, &trMyPI);
```

```

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
GFLIB_ControllerPIrSetState(fltControllerPIrOut, &trMyPI, FLT);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if single precision floating-point implementation is selected
// as default.
GFLIB_ControllerPIrSetState(fltControllerPIrOut, &trMyPI);
}

// Periodical function or interrupt - control loop
void ControlLoop(void)
{
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    fltOutput = GFLIB\_ControllerPIr\_FLT(fltInErr, &trMyPI);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    fltOutput = GFLIB_ControllerPIr(fltInErr, &trMyPI, FLT);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if single precision floating-point implementation is selected
    // as default.
    fltOutput = GFLIB_ControllerPIr(fltInErr, &trMyPI);
}

```

## 2.21.5 Function GFLIB\_ControllerPIrSetState

### Description

This function initializes the GFLIB\_ControllerPIr state variables to achieve the required output values.

**Note:** The input/output pointer must contain a valid address, otherwise a fault may occur (MemManage, BusFault, UsageFault, HardFault).

### Re-entrancy

The function is re-entrant for a different pCtrl.

### 2.21.5.1 Function GFLIB\_ControllerPIrSetState\_F32

#### Declaration

```
void GFLIB_ControllerPIrSetState_F32(tFrac32 f32ControllerPIrOut,
GFLIB\_CONTROLLER\_PI\_R\_T\_F32 *const pParam);
```

#### Arguments

Table 168. GFLIB\_ControllerPIrSetState\_F32 arguments

Type	Name	Direction	Description
<a href="#">tFrac32</a>	f32ControllerPIrOut	input	Required output of the GFLIB_ControllerPIr.

Type	Name	Direction	Description
<a href="#">GFLIB_CONTROLLER_PI_R_T_F32</a> *const	pParam	input, output	Pointer to the structure with GFLIB_ControllerPIr state.

## Code Example

```

#include "gflib.h"

_tFrac32 f32InErr;
_tFrac32 f32Output;

GFLIB_CONTROLLER_PI_R_T_F32 trMyPI = GFLIB\_CONTROLLER\_PI\_R\_DEFAULT\_F32;

void main(void)
{
    tFrac32 f32ControllerPIrOut;

    // Set the input error
    f32InErr = FRAC32(0.25);

    // Initialize the controller parameters
    trMyPI.f32CC1sc = FRAC32(0.01);
    trMyPI.f32CC2sc = FRAC32(0.02);
    trMyPI.u16NShift = 1;

    // Clear the state variables
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    GFLIB\_ControllerPIrInit\_F32(&trMyPI);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    GFLIB\_ControllerPIrInit\(&trMyPI, F32\);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 32-bit fractional implementation is selected as default.
    GFLIB\_ControllerPIrInit\(&trMyPI\);

    // Initialize the state variables to predefined values
    f32ControllerPIrOut = FRAC32(0.03);
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    GFLIB\_ControllerPIrSetState\_F32\(f32ControllerPIrOut, &trMyPI\);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    GFLIB\_ControllerPIrSetState\(f32ControllerPIrOut, &trMyPI, F32\);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 32-bit fractional implementation is selected as default.
    GFLIB\_ControllerPIrSetState\(f32ControllerPIrOut, &trMyPI\);
}

// Periodical function or interrupt - control loop

```

```

void ControlLoop(void)
{
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    f32Output = GFLIB_ControllerPIr_F32(f32InErr, &trMyPI);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    f32Output = GFLIB_ControllerPIr(f32InErr, &trMyPI, F32);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 32-bit fractional implementation is selected as default.
    f32Output = GFLIB_ControllerPIr(f32InErr, &trMyPI);
}

```

### 2.21.5.2 Function GFLIB\_ControllerPIrSetState\_F16

#### Declaration

```
void GFLIB_ControllerPIrSetState_F16(tFrac16 f16ControllerPIrOut,
GFLIB_CONTROLLER_PI_R_T_F16 *const pParam);
```

#### Arguments

**Table 169. GFLIB\_ControllerPIrSetState\_F16 arguments**

Type	Name	Direction	Description
<u>tFrac16</u>	f16ControllerPIrOut	input	Required output of the GFLIB_ControllerPIr.
<u>GFLIB_CONTROLLER_PI_R_T_F16</u> *const	pParam	input, output	Pointer to the structure with GFLIB_ControllerPIr state.

#### Code Example

```

#include "gflib.h"

tFrac16 f16InErr;
tFrac16 f16Output;

GFLIB_CONTROLLER_PI_R_T_F16 trMyPI = GFLIB_CONTROLLER_PI_R_DEFAULT_F16;

void main(void)
{
    tFrac16 f16ControllerPIrOut;

    // Set the input error
    f16InErr = FRAC16(0.25);

    // Initialize the controller parameters
    trMyPI.f16CC1sc = FRAC16(0.01);
    trMyPI.f16CC2sc = FRAC16(0.02);
    trMyPI.ul6NShift = 1;

    // Clear the state variables
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    GFLIB_ControllerPIrInit_F16(&trMyPI);
}

```

```

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
GFLIB_ControllerPIrInit(&trMyPI, F16);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 16-bit fractional implementation is selected as default.
GFLIB_ControllerPIrInit(&trMyPI);

// Initialize the state variables to predefined values
f16ControllerPIrOut = FRAC16(0.03);
// Alternative 1: API call with postfix
// (only one alternative shall be used).
GFLIB_ControllerPIrSetState_F16(f16ControllerPIrOut, &trMyPI);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
GFLIB_ControllerPIrSetState(f16ControllerPIrOut, &trMyPI, F16);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 16-bit fractional implementation is selected as default.
GFLIB_ControllerPIrSetState(f16ControllerPIrOut, &trMyPI);
}

// Periodical function or interrupt - control loop
void ControlLoop(void)
{
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    f16Output = GFLIB_ControllerPIr_F16(f16InErr, &trMyPI);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    f16Output = GFLIB_ControllerPIr(f16InErr, &trMyPI, F16);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 16-bit fractional implementation is selected as default.
    f16Output = GFLIB_ControllerPIr(f16InErr, &trMyPI);
}

```

### 2.21.5.3 Function **GFLIB\_ControllerPIrSetState\_FLT**

#### Declaration

```
void GFLIB_ControllerPIrSetState_FLT(tFloat fltControllerPIrOut,
GFLIB_CONTROLLER_PI_R_T_FLT *const pParam);
```

#### Arguments

**Table 170. GFLIB\_ControllerPIrSetState\_FLT arguments**

Type	Name	Direction	Description
<b>tFloat</b>	fltControllerPIrOut	input	Required output of the GFLIB_ControllerPIr.

Type	Name	Direction	Description
<a href="#">GFLIB_CONTROLLER_PI_R_T_FLT</a> *const	pParam	input, output	Pointer to the structure with GFLIB_ControllerPIr state.

## Code Example

```

#include "gflib.h"

tFloat fltInErr;
tFloat fltOutput;

GFLIB_CONTROLLER_PI_R_T_FLT trMyPI = GFLIB\_CONTROLLER\_PI\_R\_DEFAULT\_FLT;

void main(void)
{
    tFloat fltControllerPIrOut;

    // Set the input error
    fltInErr = 0.25f;

    // Initialize the controller parameters
    trMyPI.fltCC1sc = 0.01f;
    trMyPI.fltCC2sc = 0.02f;

    // Clear the state variables
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
GFLIB\_ControllerPIrInit\_FLT(&trMyPI);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    GFLIB\_ControllerPIrInit\(&trMyPI, FLT\);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if single precision floating-point implementation is selected
    // as default.
    GFLIB\_ControllerPIrInit\(&trMyPI\);

    // Initialize the state variables to predefined values
    fltControllerPIrOut = 0.03f;
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    GFLIB\_ControllerPIrSetState\_FLT\(fltControllerPIrOut, &trMyPI\);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    GFLIB\_ControllerPIrSetState\(fltControllerPIrOut, &trMyPI, FLT\);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if single precision floating-point implementation is selected
    // as default.
    GFLIB\_ControllerPIrSetState\(fltControllerPIrOut, &trMyPI\);
}

```

```

// Periodical function or interrupt - control loop
void ControlLoop(void)
{
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    fltOutput = GFLIB\_ControllerPIr\_FLT(fltInErr, &trMyPI);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    fltOutput = GFLIB\_ControllerPIr(fltInErr, &trMyPI, FLT);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if single precision floating-point implementation is selected
    // as default.
    fltOutput = GFLIB\_ControllerPIr(fltInErr, &trMyPI);
}

```

## 2.22 Function GFLIB\_ControllerPIrAW

This function calculates a standard recurrent form of the Proportional-Integral controller, with integral anti-windup.

### Description

The function GFLIB\_ControllerPIrAW calculates a standard recurrent form of the Proportional-Integral controller, with integral anti-windup.

The continuous time domain representation of the PI controller is defined as:

$$u(t) = e(t) \cdot K_p + K_i \int_0^t e(t) dt$$

Equation GFLIB\_ControllerPIrAW\_Eq1

The transfer function for this kind of PI controller, in a continuous time domain is described using the Laplace transformation as follows:

$$H(s) = \frac{U(s)}{E(s)} = \frac{sK_p + K_i}{s}$$

Equation GFLIB\_ControllerPIrAW\_Eq2

Transforming equation [GFLIB\\_ControllerPIrAW\\_Eq2](#) into a discrete time domain leads to the following equation:

$$u(k) = u(k-1) + e(k) \cdot CC1 + e(k-1) \cdot CC2$$

Equation GFLIB\_ControllerPIrAW\_Eq3

where  $K_p$  is proportional gain,  $K_i$  is integral gain,  $T_s$  is the sampling period,  $u(k)$  is the controller output,  $e(k)$  is the controller input error signal,  $CC1$  and  $CC2$  are the controller coefficients calculated depending on the discretization method used, as shown in [Table 171](#).

**Table 171. Calculation of coefficients CC1 and CC2 using various discretization methods**

	Trapezoidal	Backward Rect.	Forward Rect.
--	-------------	----------------	---------------

CC1=	$K_p + K_i T_s / 2$	$K_p + K_i T_s$	$K_p$
CC2=	$-K_p + K_i T_s / 2$	$-K_p$	$-K_p + K_i T_s$

**Note:** The input pointer must contain a valid address otherwise a fault may occur (MemManage, BusFault, UsageFault, HardFault).

### Re-entrancy

The function is re-entrant.

#### 2.22.1 Function `GFLIB_ControllerPIrAW_F32`

##### Declaration

```
tFrac32 GFLIB_ControllerPIrAW_F32 (tFrac32 f32InErr,
GFLIB_CONTROLLER_PIAW_R_T_F32 *const pParam);
```

##### Arguments

Table 172. `GFLIB_ControllerPIrAW_F32` arguments

Type	Name	Direction	Description
<code>tFrac32</code>	f32InErr	input	Input error signal to the controller is a 32-bit number normalized between [-1, 1).
<code>GFLIB_CONTROLLER_PIAW_R_T_F32</code> *const	pParam	input, output	Pointer to the controller parameters structure.

##### Return

The function returns a 32-bit value in fractional format 1.31, representing the signal to be applied to the controlled system so that the input error is forced to zero.

##### Implementation details

In order to implement the discrete equation of the controller [GFLIB\\_ControllerPIrAW\\_F32\\_Eq3](#) on the fixed point arithmetic platform, the maximal values (scales) of input and output signals

- $E^{MAX}$  - maximal value of the controller input error signal
- $U^{MAX}$  - maximal value of the controller output signal

have to be known a priori. This is essential for correct casting of the physical signal values into fixed point values [-1, 1).

Then the fractional representation [-1, 1) of both input and output signals is obtained as follows:

$$e_f(k) = \frac{e(k)}{E^{MAX}}$$

$$u_f(k) = \frac{u(k)}{U^{MAX}}$$

Equation GFLIB\_ControllerPIrAW\_F32\_Eq1

The resulting controller discrete time domain equation in fixed point fractional representation is therefore given as:

$$u_f(k) \cdot U^{MAX} = u_f(k-1) \cdot U^{MAX} + e_f(k) \cdot E^{MAX} \cdot CC1 + e_f(k-1) \cdot E^{MAX} \cdot CC2$$

Equation GFLIB\_ControllerPIrAW\_F32\_Eq2

which can be rearranged into the following form:

$$u_f(k) = u_f(k-1) + e_f(k) \cdot CC1_f + e_f(k-1) \cdot CC2_f$$

Equation GFLIB\_ControllerPIrAW\_F32\_Eq3

where

$$CC1_f = CC1 \cdot \frac{E^{MAX}}{U^{MAX}}$$

$$CC2_f = CC2 \cdot \frac{E^{MAX}}{U^{MAX}}$$

Equation GFLIB\_ControllerPIrAW\_F32\_Eq4

are the controller coefficients adapted according to the input and output scale values. In order to implement both coefficients as fractional numbers, both  $CC1_f$  and  $CC2_f$  must reside in the fractional range [-1, 1). However, depending on values  $CC1$ ,  $CC2$ ,  $E^{MAX}$ ,  $U^{MAX}$ , the calculation of  $CC1_f$  and  $CC2_f$  may result in values outside this fractional range. Therefore, a scaling of  $CC1_f$ ,  $CC2_f$  is introduced as follows:

$$f32CC1sc = CC1_f \cdot 2^{u16NShift}$$

$$f32CC2sc = CC2_f \cdot 2^{u16NShift}$$

Equation GFLIB\_ControllerPIrAW\_F32\_Eq5

The introduced scaling shift  $u16NShift$  is chosen such that both coefficients  $f32CC1sc$ ,  $f32CC2sc$  reside in the range [-1, 1). To simplify the implementation, this scaling shift is chosen to be a power of 2, so the final scaling is a simple shift operation. Moreover, the scaling shift cannot be a negative number, so the operation of scaling is always to scale numbers with an absolute value larger than 1 down to fit in the range [-1, 1).

$$u16NShift = \max \left( \left\lceil \frac{\log(\text{abs}(CC1_f))}{\log(2)} \right\rceil, \left\lceil \frac{\log(\text{abs}(CC2_f))}{\log(2)} \right\rceil \right)$$

Equation GFLIB\_ControllerPIrAW\_F32\_Eq6

The final, scaled, fractional equation of the recurrent PI controller on a 32-bit fixed point platform is therefore implemented as follows:

$$u_f(k) \cdot 2^{u16NShift} = u_f(k-1) \cdot 2^{u16NShift} + e_f(k) \cdot f32CC1sc + e_f(k-1) \cdot f32CC2sc$$

Equation GFLIB\_ControllerPIrAW\_F32\_Eq7

where:

- $u_f(k)$  - fractional representation [-1, 1) of the controller output
- $e_f(k)$  - fractional representation [-1, 1) of the controller input (error)
- $f32CC1sc$  - fractional representation [-1, 1) of the 1st controller coefficient
- $f32CC2sc$  - fractional representation [-1, 1) of the 2nd controller coefficient
- $u16NShift$  - in range [0,31] - is chosen such that both coefficients  $f32CC1sc$  and  $f32CC2sc$  are in the range [-1, 1)

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

$$u(k) = \begin{cases} f32UpperLimit & \text{if } u_f(k) \geq f32UpperLimit \\ u_f(k) & \text{if } f32LowerLimit < u_f(k) < f32UpperLimit \\ f32LowerLimit & \text{if } u_f(k) \leq f32LowerLimit \end{cases}$$

Equation GFLIB\_ControllerPIrAW\_F32\_Eq8

The bounds are described by a limitation element equation `eq13_GFLIB_ControllerPIrAW_F32`. When the bounds are exceeded, the non-linear saturation characteristic will take effect and influence the dynamic behavior. The described limitation is implemented on the output sum. Therefore, if the limitation occurs, the controller output is clipped to its bounds and the wind-up occurrence of the accumulator is avoided by saturating the output sum.

All controller parameters and states can be reset during declaration using the [`GFLIB\_CONTROLLER\_PIAW\_R\_DEFAULT\_F32`](#) macro.

**Note:** Due to effectivity reasons this function is implemented using inline assembly and is therefore not ANSI-C compliant.

### Code Example

```
#include "gflib.h"

tFrac32 f32InErr;
tFrac32 f32Output;

GFLIB_CONTROLLER_PIAW_R_T_F32 trMyPI = GFLIB\_CONTROLLER\_PIAW\_R\_DEFAULT\_F32;

void main(void)
{
    tFrac32 f32ControllerPIrAWOut;

    // Set the input error
    f32InErr = FRAC32\(0.25\);

    // Initialize the controller parameters
    trMyPI.f32CC1sc      = FRAC32\(0.01\);
    trMyPI.f32CC2sc      = FRAC32\(0.02\);
    trMyPI.ul6NShift     = 1;
    trMyPI.f32UpperLimit = FRAC32\(1.0\);
    trMyPI.f32LowerLimit = FRAC32\(-1.0\);

    // Clear the state variables
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    GFLIB\_ControllerPIrAWInit\_F32\(&trMyPI\);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    GFLIB\_ControllerPIrAWInit\(&trMyPI, F32\);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available

```

```

// only if 32-bit fractional implementation is selected as default.
GFLIB_ControllerPIrAWInit(&trMyPI);

// Initialize the state variables to predefined values
f32ControllerPIrAWOut = FRAC32(0.03);
// Alternative 1: API call with postfix
// (only one alternative shall be used).
GFLIB_ControllerPIrAWSetState_F32(f32ControllerPIrAWOut, &trMyPI);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
GFLIB_ControllerPIrAWSetState(f32ControllerPIrAWOut, &trMyPI, F32);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 32-bit fractional implementation is selected as default.
GFLIB_ControllerPIrAWSetState(f32ControllerPIrAWOut, &trMyPI);
}

// Periodical function or interrupt - control loop
void ControlLoop(void)
{
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    f32Output = GFLIB_ControllerPIrAW_F32(f32InErr, &trMyPI);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    f32Output = GFLIB_ControllerPIrAW(f32InErr, &trMyPI, F32);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 32-bit fractional implementation is selected as default.
    f32Output = GFLIB_ControllerPIrAW(f32InErr, &trMyPI);
}

```

## 2.22.2 Function GFLIB\_ControllerPIrAW\_F16

### Declaration

```
tFrac16 GFLIB_ControllerPIrAW_F16(tFrac16 f16InErr,
GFLIB_CONTROLLER_PIAW_R_T_F16 *const pParam);
```

### Arguments

Table 173. GFLIB\_ControllerPIrAW\_F16 arguments

Type	Name	Direction	Description
<u>tFrac16</u>	f16InErr	input	Input error signal to the controller is a 16-bit number normalized between [-1, 1).
<u>GFLIB_CONTROLLER_PIAW_R_T_F16</u> *const	pParam	input, output	Pointer to the controller parameters structure.

### Return

The function returns a 16-bit value in fractional format 1.16, representing the signal to be applied to the controlled system so that the input error is forced to zero.

### Implementation details

In order to implement the discrete equation of the controller [GFLIB\\_ControllerPIrAW\\_F16\\_Eq3](#) on the fixed point arithmetic platform, the maximal values (scales) of input and output signals

- $E^{MAX}$  - maximal value of the controller input error signal
- $U^{MAX}$  - maximal value of the controller output signal

have to be known a priori. This is essential for correct casting of the physical signal values into fixed point values [-1, 1].

Then the fractional representation [-1, 1) of both input and output signals is obtained as follows:

$$e_f(k) = \frac{e(k)}{E^{MAX}}$$

$$u_f(k) = \frac{u(k)}{U^{MAX}}$$

Equation GFLIB\_ControllerPIrAW\_F16\_Eq1

The resulting controller discrete time domain equation in fixed point fractional representation is therefore given as:

$$u_f(k) \cdot U^{MAX} = u_f(k-1) \cdot U^{MAX} + e_f(k) \cdot E^{MAX} \cdot CC1 + e_f(k-1) \cdot E^{MAX} \cdot CC2$$

Equation GFLIB\_ControllerPIrAW\_F16\_Eq2

which can be rearranged into the following form:

$$u_f(k) = u_f(k-1) + e_f(k) \cdot CC1_f + e_f(k-1) \cdot CC2_f$$

Equation GFLIB\_ControllerPIrAW\_F16\_Eq3

where

$$CC1_f = CC1 \cdot \frac{E^{MAX}}{U^{MAX}}$$

$$CC2_f = CC2 \cdot \frac{E^{MAX}}{U^{MAX}}$$

Equation GFLIB\_ControllerPIrAW\_F16\_Eq4

are the controller coefficients adapted according to the input and output scale values.

In order to implement both coefficients as fractional numbers, both  $CC1_f$  and  $CC2_f$  must reside in the fractional range [-1, 1). However, depending on values  $CC1$ ,  $CC2$ ,  $E^{MAX}$ ,  $U^{MAX}$ , the calculation of  $CC1_f$  and  $CC2_f$  may result in values outside this fractional range. Therefore, a scaling of  $CC1_f$ ,  $CC2_f$  is introduced as follows:

$$f16CC1sc = CC1_f \cdot 2^{u16NShift}$$

$$f16CC2sc = CC2_f \cdot 2^{u16NShift}$$

Equation GFLIB\_ControllerPIrAW\_F16\_Eq5

The introduced scaling shift  $u16NShift$  is chosen such that both coefficients  $f16CC1sc$ ,  $f16CC2sc$  reside in the range [-1, 1). To simplify the implementation, this scaling shift is chosen to be a power of 2, so the final scaling is a simple shift operation. Moreover, the scaling shift cannot be a negative number, so the operation of scaling is

always to scale numbers with an absolute value larger than 1 down to fit in the range [-1, 1).

$$u_{16Nshift} = \max\left(\left\lceil \frac{\log(\text{abs}(CC1_f))}{\log(2)} \right\rceil, \left\lceil \frac{\log(\text{abs}(CC2_f))}{\log(2)} \right\rceil\right)$$

Equation GFLIB\_ControllerPIrAW\_F16\_Eq6

The final, scaled, fractional equation of the recurrent PI controller on a 16-bit fixed point platform is therefore implemented as follows:

$$u_f(k) \cdot 2^{u_{16NShift}} = u_f(k-1) \cdot 2^{u_{16NShift}} + e_f(k) \cdot f16CC1sc + e_f(k-1) \cdot f16CC2sc$$

Equation GFLIB\_ControllerPIrAW\_F16\_Eq7

where:

- $u_f(k)$  - fractional representation [-1, 1) of the controller output
- $e_f(k)$  - fractional representation [-1, 1) of the controller input (error)
- $f16CC1sc$  - fractional representation [-1, 1) of the 1st controller coefficient
- $f16CC2sc$  - fractional representation [-1, 1) of the 2nd controller coefficient
- $u_{16NShift}$  - in range [0,15] - is chosen such that both coefficients  $f16CC1sc$  and  $f16CC2sc$  are in the range [-1, 1)

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

$$u(k) = \begin{cases} f16UpperLimit & \text{if } u_f(k) \geq f16UpperLimit \\ u_f(k) & \text{if } f16LowerLimit < u_f(k) < f16UpperLimit \\ f16LowerLimit & \text{if } u_f(k) \leq f16LowerLimit \end{cases}$$

Equation GFLIB\_ControllerPIrAW\_F16\_Eq8

The bounds are described by a limitation element equation eq13\_GFLIB\_ControllerPIrAW\_F16. When the bounds are exceeded, the non-linear saturation characteristic will take effect and influence the dynamic behavior. The described limitation is implemented on the output sum. Therefore, if the limitation occurs, the controller output is clipped to its bounds and the wind-up occurrence of the accumulator is avoided by saturating the output sum.

All controller parameters and states can be reset during declaration using the [GFLIB\\_CONTROLLER\\_PIAW\\_R\\_DEFAULT\\_F16](#) macro.

**Note:** Due to effectivity reasons this function is implemented using inline assembly and is therefore not ANSI-C compliant.

### Code Example

```
#include "gplib.h"

tFrac16 f16InErr;
tFrac16 f16Output;

GFLIB_CONTROLLER_PIAW_R_T_F16 trMyPI = GFLIB_CONTROLLER_PIAW_R_DEFAULT_F16;
```

```
void main(void)
{
    tFrac16 f16ControllerPIrAWOut;

    // Set the input error
    f16InErr = FRAC16(0.25);

    // Initialize the controller parameters
    trMyPI.f16CC1sc      = FRAC16(0.01);
    trMyPI.f16CC2sc      = FRAC16(0.02);
    trMyPI.ul6NShift     = 1;
    trMyPI.f16UpperLimit = FRAC16(1.0);
    trMyPI.f16LowerLimit = FRAC16(-1.0);

    // Clear the state variables
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    GFLIB_ControllerPIrAWInit_F16(&trMyPI);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    GFLIB_ControllerPIrAWInit(&trMyPI, F16);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 16-bit fractional implementation is selected as default.
    GFLIB_ControllerPIrAWInit(&trMyPI);

    // Initialize the state variables to predefined values
    f16ControllerPIrAWOut = FRAC16(0.03);
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    GFLIB_ControllerPIrAWSetState_F16(f16ControllerPIrAWOut, &trMyPI);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    GFLIB_ControllerPIrAWSetState(f16ControllerPIrAWOut, &trMyPI, F16);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 16-bit fractional implementation is selected as default.
    GFLIB_ControllerPIrAWSetState(f16ControllerPIrAWOut, &trMyPI);
}

// Periodical function or interrupt - control loop
void ControlLoop(void)
{
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    f16Output = GFLIB_ControllerPIrAW_F16(f16InErr, &trMyPI);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    f16Output = GFLIB_ControllerPIrAW(f16InErr, &trMyPI, F16);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 16-bit fractional implementation is selected as default.
    f16Output = GFLIB_ControllerPIrAW(f16InErr, &trMyPI);
}
```

```
}
```

### 2.22.3 Function GFLIB\_ControllerPIrAW\_FLT

#### Declaration

```
tFloat GFLIB_ControllerPIrAW_FLT(tFloat fltInErr,
GFLIB_CONTROLLER_PIAW_R_T_FLT *const pParam);
```

#### Arguments

Table 174. GFLIB\_ControllerPIrAW\_FLT arguments

Type	Name	Direction	Description
tFloat	fltInErr	input	Input error signal to the controller in single precision floating point data format.
GFLIB_CONTROLLER_PIAW_R_T_FLT *const	pParam	input, output	Pointer to the controller parameters structure.

#### Return

The function returns a single precision floating point value, representing the signal to be applied to the controlled system so that the input error is forced to zero.

#### Implementation details

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

$$u(k) = \begin{cases} fltUpperLimit & if \quad u_f(k) \geq fltUpperLimit \\ u_f(k) & if \quad fltLowerLimit < u_f(k) < fltUpperLimit \\ fltLowerLimit & if \quad u_f(k) \leq fltLowerLimit \end{cases}$$

Equation GFLIB\_ControllerPIrAW\_FLT\_Eq1

The bounds are described by a limitation element equation eq4\_GFLIB\_ControllerPIrAW\_FLT. When the bounds are exceeded, the non-linear saturation characteristic will take effect and influence the dynamic behavior. The described limitation is implemented on the output sum. Therefore, if the limitation occurs, the controller output is clipped to its bounds and the wind-up occurrence of the accumulator is avoided by saturating the output sum.

All controller parameters and states can be reset during declaration using the [GFLIB\\_CONTROLLER\\_PIAW\\_R\\_DEFAULT\\_FLT](#) macro.

**Note:** The function may raise floating-point exceptions (invalid operation, overflow, underflow, inexact, input denormal). The floating-point unit must be enabled in CPACR register to prevent NOCP UsageFault exception.

#### Code Example

```
#include "gplib.h"
```

```
tFloat fltInErr;
tFloat fltOutput;

GFLIB_CONTROLLER_PIAW_R_T_FLT trMyPI = GFLIB_CONTROLLER_PIAW_R_DEFAULT_FLT;

void main(void)
{
    tFloat fltControllerPIrAWOut;

    // Set the input error
    fltInErr = 0.25f;

    // Initialize the controller parameters
    trMyPI.fltCC1sc      = 0.01F;
    trMyPI.fltCC2sc      = 0.02F;
    trMyPI.fltUpperLimit = 1.0F;
    trMyPI.fltLowerLimit = -1.0F;

    // Clear the state variables
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    GFLIB_ControllerPIrAWInit_FLT(&trMyPI);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    GFLIB_ControllerPIrAWInit(&trMyPI, FLT);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if single precision floating-point implementation is selected
    // as default.
    GFLIB_ControllerPIrAWInit(&trMyPI);

    // Initialize the state variables to predefined values
    fltControllerPIrAWOut = 0.03f;
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    GFLIB_ControllerPIrAWSetState_FLT(fltControllerPIrAWOut, &trMyPI);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    GFLIB_ControllerPIrAWSetState(fltControllerPIrAWOut, &trMyPI, FLT);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if single precision floating-point implementation is selected
    // as default.
    GFLIB_ControllerPIrAWSetState(fltControllerPIrAWOut, &trMyPI);
}

// Periodical function or interrupt - control loop
void ControlLoop(void)
{
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    fltOutput = GFLIB_ControllerPIrAW_FLT(fltInErr, &trMyPI);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
```

```

    fltOutput = GFLIB_ControllerPIrAW(fltInErr, &trMyPI, FLT);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if single precision floating-point implementation is selected
    // as default.
    fltOutput = GFLIB_ControllerPIrAW(fltInErr, &trMyPI);
}

```

## 2.22.4 Function GFLIB\_ControllerPIrAWInit

### Description

This function clears the GFLIB\_ControllerPIrAW state variables.

**Note:** The input/output pointer must contain a valid address, otherwise a fault may occur (MemManage, BusFault, UsageFault, HardFault).

### Re-entrancy

The function is re-entrant for a different pCtrl.

### 2.22.4.1 Function GFLIB\_ControllerPIrAWInit\_F32

#### Declaration

```
void GFLIB_ControllerPIrAWInit_F32(GFLIB\_CONTROLLER\_PIAW\_R\_T\_F32
*const pParam);
```

#### Arguments

Table 175. GFLIB\_ControllerPIrAWInit\_F32 arguments

Type	Name	Direction	Description
<a href="#">GFLIB_CONTROLLER_PIAW_R_T_F32</a> *const	pParam	input, output	Pointer to the structure with GFLIB_ControllerPIrAW state.

### Code Example

```

#include "gflib.h"

tFrac32 f32InErr;
tFrac32 f32Output;

GFLIB\_CONTROLLER\_PIAW\_R\_T\_F32 trMyPI = GFLIB\_CONTROLLER\_PIAW\_R\_DEFAULT\_F32;

void main(void)
{
    tFrac32 f32ControllerPIrAWOut;

    // Set the input error
    f32InErr = FRAC32(0.25);

    // Initialize the controller parameters
    trMyPI.f32CC1sc      = FRAC32(0.01);
}

```

```
trMyPI.f32CC2sc      = FRAC32(0.02);
trMyPI.ul6NShift     = 1;
trMyPI.f32UpperLimit = FRAC32(1.0);
trMyPI.f32LowerLimit = FRAC32(-1.0);

// Clear the state variables
// Alternative 1: API call with postfix
// (only one alternative shall be used).
GFLIB_ControllerPIrAWInit_F32(&trMyPI);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
GFLIB_ControllerPIrAWInit(&trMyPI, F32);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 32-bit fractional implementation is selected as default.
GFLIB_ControllerPIrAWInit(&trMyPI);

// Initialize the state variables to predefined values
f32ControllerPIrAWOut = FRAC32(0.03);
// Alternative 1: API call with postfix
// (only one alternative shall be used).
GFLIB_ControllerPIrAWSetState_F32(f32ControllerPIrAWOut, &trMyPI);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
GFLIB_ControllerPIrAWSetState(f32ControllerPIrAWOut, &trMyPI, F32);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 32-bit fractional implementation is selected as default.
GFLIB_ControllerPIrAWSetState(f32ControllerPIrAWOut, &trMyPI);
}

// Periodical function or interrupt - control loop
void ControlLoop(void)
{
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    f32Output = GFLIB\_ControllerPIrAW\_F32(f32InErr, &trMyPI);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    f32Output = GFLIB_ControllerPIrAW(f32InErr, &trMyPI, F32);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 32-bit fractional implementation is selected as default.
    f32Output = GFLIB_ControllerPIrAW(f32InErr, &trMyPI);
}
```

#### 2.22.4.2 Function [GFLIB\\_ControllerPIrAWInit\\_F16](#)

##### Declaration

```
void GFLIB_ControllerPIrAWInit_F16(GFLIB\_CONTROLLER\_PIAW\_R\_T\_F16
*const pParam);
```

## Arguments

**Table 176.** `GFLIB_ControllerPIrAWInit_F16` arguments

Type	Name	Direction	Description
<code>GFLIB_CONTROLLER_PIAW_R_T_F16 *const</code>	<code>pParam</code>	<code>input, output</code>	Pointer to the structure with <code>GFLIB_ControllerPIrAW</code> state.

## Code Example

```

#include "gflib.h"

tFrac16 f16InErr;
tFrac16 f16Output;

GFLIB_CONTROLLER_PIAW_R_T_F16 trMyPI = GFLIB_CONTROLLER_PIAW_R_DEFAULT_F16;

void main(void)
{
    tFrac16 f16ControllerPIrAWOut;

    // Set the input error
    f16InErr = FRAC16(0.25);

    // Initialize the controller parameters
    trMyPI.f16CC1sc      = FRAC16(0.01);
    trMyPI.f16CC2sc      = FRAC16(0.02);
    trMyPI.ul6NShift     = 1;
    trMyPI.f16UpperLimit = FRAC16(1.0);
    trMyPI.f16LowerLimit = FRAC16(-1.0);

    // Clear the state variables
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    GFLIB_ControllerPIrAWInit(&trMyPI);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    GFLIB_ControllerPIrAWInit(&trMyPI, F16);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 16-bit fractional implementation is selected as default.
    GFLIB_ControllerPIrAWInit(&trMyPI);

    // Initialize the state variables to predefined values
    f16ControllerPIrAWOut = FRAC16(0.03);
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    GFLIB_ControllerPIrAWSetState_F16(f16ControllerPIrAWOut, &trMyPI);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    GFLIB_ControllerPIrAWSetState(f16ControllerPIrAWOut, &trMyPI, F16);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available

```

```

// only if 16-bit fractional implementation is selected as default.
GFLIB_ControllerPIrAWSetState(f16ControllerPIrAWOut, &trMyPI);
}

// Periodical function or interrupt - control loop
void ControlLoop(void)
{
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    f16Output = GFLIB\_ControllerPIrAW\_F16(f16InErr, &trMyPI);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    f16Output = GFLIB_ControllerPIrAW(f16InErr, &trMyPI, F16);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 16-bit fractional implementation is selected as default.
    f16Output = GFLIB_ControllerPIrAW(f16InErr, &trMyPI);
}

```

#### 2.22.4.3 Function [GFLIB\\_ControllerPIrAWInit\\_FLT](#)

##### Declaration

```
void GFLIB_ControllerPIrAWInit_FLT(GFLIB\_CONTROLLER\_PIAW\_R\_T\_FLT
*const pParam);
```

##### Arguments

**Table 177. [GFLIB\\_ControllerPIrAWInit\\_FLT](#) arguments**

Type	Name	Direction	Description
<a href="#">GFLIB_CONTROLLER_PIAW_R_T_FLT</a> *const	pParam	input, output	Pointer to the structure with <a href="#">GFLIB_ControllerPIrAW</a> state.

##### Code Example

```

#include "gflib.h"

tFloat fltInErr;
tFloat fltOutput;

GFLIB\_CONTROLLER\_PIAW\_R\_T\_FLT trMyPI = GFLIB\_CONTROLLER\_PIAW\_R\_DEFAULT\_FLT;

void main(void)
{
    tFloat fltControllerPIrAWOut;

    // Set the input error
    fltInErr = 0.25f;

    // Initialize the controller parameters
    trMyPI.fltCC1sc      = 0.01F;
    trMyPI.fltCC2sc      = 0.02F;
    trMyPI.fltUpperLimit = 1.0F;
    trMyPI.fltLowerLimit = -1.0F;
}

```

```
// Clear the state variables
// Alternative 1: API call with postfix
// (only one alternative shall be used).
GFLIB_ControllerPIrAWInit_FLT(&trMyPI);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
GFLIB_ControllerPIrAWInit(&trMyPI, FLT);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if single precision floating-point implementation is selected
// as default.
GFLIB_ControllerPIrAWInit(&trMyPI);

// Initialize the state variables to predefined values
fltControllerPIrAWOut = 0.03f;
// Alternative 1: API call with postfix
// (only one alternative shall be used).
GFLIB_ControllerPIrAWSetState_FLT(fltControllerPIrAWOut, &trMyPI);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
GFLIB_ControllerPIrAWSetState(fltControllerPIrAWOut, &trMyPI, FLT);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if single precision floating-point implementation is selected
// as default.
GFLIB_ControllerPIrAWSetState(fltControllerPIrAWOut, &trMyPI);
}

// Periodical function or interrupt - control loop
void ControlLoop(void)
{
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    fltOutput = GFLIB_ControllerPIrAW_FLT(fltInErr, &trMyPI);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    fltOutput = GFLIB_ControllerPIrAW(fltInErr, &trMyPI, FLT);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if single precision floating-point implementation is selected
    // as default.
    fltOutput = GFLIB_ControllerPIrAW(fltInErr, &trMyPI);
}
```

## 2.22.5 Function GFLIB\_ControllerPIrAWSetState

### Description

This function initializes the GFLIB\_ControllerPIrAW state variables to achieve the required output values.

**Note:** The input/output pointer must contain a valid address, otherwise a fault may occur (MemManage, BusFault, UsageFault, HardFault).

### Re-entrancy

The function is re-entrant for a different pCtrl.

#### 2.22.5.1 Function `GFLIB_ControllerPIrAWSetState_F32`

##### Declaration

```
void GFLIB_ControllerPIrAWSetState_F32(tFrac32
f32ControllerPIrAWOut, GFLIB_CONTROLLER_PIAW_R_T_F32 *const
pParam);
```

##### Arguments

Table 178. `GFLIB_ControllerPIrAWSetState_F32` arguments

Type	Name	Direction	Description
<u>tFrac32</u>	f32ControllerPIrAWOut	<u>input</u>	Required output of the <code>GFLIB_ControllerPIrAW</code> .
<u>GFLIB_CONTROLLER_PIAW_R_T_F32</u> *const	pParam	<u>input, output</u>	Pointer to the structure with <code>GFLIB_ControllerPIrAW</code> state.

##### Code Example

```
#include "gflib.h"

tFrac32 f32InErr;
tFrac32 f32Output;

GFLIB_CONTROLLER_PIAW_R_T_F32 trMyPI = GFLIB_CONTROLLER_PIAW_R_DEFAULT_F32;

void main(void)
{
    tFrac32 f32ControllerPIrAWOut;

    // Set the input error
    f32InErr = FRAC32(0.25);

    // Initialize the controller parameters
    trMyPI.f32CC1sc      = FRAC32(0.01);
    trMyPI.f32CC2sc      = FRAC32(0.02);
    trMyPI.u16NShift     = 1;
    trMyPI.f32UpperLimit = FRAC32(1.0);
    trMyPI.f32LowerLimit = FRAC32(-1.0);

    // Clear the state variables
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    GFLIB_ControllerPIrAWInit_F32(&trMyPI);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    GFLIB_ControllerPIrAWInit(&trMyPI, F32);

    // Alternative 3: API call with global configuration of implementation
```

```

// (only one alternative shall be used). This alternative is available
// only if 32-bit fractional implementation is selected as default.
GFLIB_ControllerPIrAWInit(&trMyPI);

// Initialize the state variables to predefined values
f32ControllerPIrAWOut = FRAC32(0.03);
// Alternative 1: API call with postfix
// (only one alternative shall be used).
GFLIB_ControllerPIrAWS.setState_F32(f32ControllerPIrAWOut, &trMyPI);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
GFLIB_ControllerPIrAWS.setState(f32ControllerPIrAWOut, &trMyPI, F32);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 32-bit fractional implementation is selected as default.
GFLIB_ControllerPIrAWS.setState(f32ControllerPIrAWOut, &trMyPI);
}

// Periodical function or interrupt - control loop
void ControlLoop(void)
{
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    f32Output = GFLIB\_ControllerPIrAW\_F32(f32InErr, &trMyPI);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    f32Output = GFLIB_ControllerPIrAW(f32InErr, &trMyPI, F32);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 32-bit fractional implementation is selected as default.
    f32Output = GFLIB_ControllerPIrAW(f32InErr, &trMyPI);
}

```

### 2.22.5.2 Function GFLIB\_ControllerPIrAWSetState\_F16

#### Declaration

```
void GFLIB_ControllerPIrAWSetState_F16(tFrac16
f16ControllerPIrAWOut, GFLIB\_CONTROLLER\_PIAW\_R\_T\_F16 *const
pParam);
```

#### Arguments

**Table 179. GFLIB\_ControllerPIrAWSetState\_F16 arguments**

Type	Name	Direction	Description
<a href="#">tFrac16</a>	f16ControllerPIrAWOut	input	Required output of the GFLIB_ControllerPIrAW.
<a href="#">GFLIB_CONTROLLER_PIAW_R_T_F16</a> *const	pParam	input, output	Pointer to the structure with GFLIB_ControllerPIrAW state.

## Code Example

```
#include "gplib.h"

tFrac16 f16InErr;
tFrac16 f16Output;

GFLIB_CONTROLLER_PIAW_R_T_F16 trMyPI = GFLIB_CONTROLLER_PIAW_R_DEFAULT_F16;

void main(void)
{
    tFrac16 f16ControllerPIrAWOut;

    // Set the input error
    f16InErr = FRAC16(0.25);

    // Initialize the controller parameters
    trMyPI.f16CC1sc      = FRAC16(0.01);
    trMyPI.f16CC2sc      = FRAC16(0.02);
    trMyPI.u16NShift     = 1;
    trMyPI.f16UpperLimit = FRAC16(1.0);
    trMyPI.f16LowerLimit = FRAC16(-1.0);

    // Clear the state variables
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    GFLIB_ControllerPIrAWInit_F16(&trMyPI);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    GFLIB_ControllerPIrAWInit(&trMyPI, F16);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 16-bit fractional implementation is selected as default.
    GFLIB_ControllerPIrAWInit(&trMyPI);

    // Initialize the state variables to predefined values
    f16ControllerPIrAWOut = FRAC16(0.03);
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    GFLIB_ControllerPIrAWSetState_F16(f16ControllerPIrAWOut, &trMyPI);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    GFLIB_ControllerPIrAWSetState(f16ControllerPIrAWOut, &trMyPI, F16);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 16-bit fractional implementation is selected as default.
    GFLIB_ControllerPIrAWSetState(f16ControllerPIrAWOut, &trMyPI);
}

// Periodical function or interrupt - control loop
void ControlLoop(void)
{
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
```

```

f16Output = GFLIB\_ControllerPIrAW\_F16(f16InErr, &trMyPI);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
f16Output = GFLIB\_ControllerPIrAW(f16InErr, &trMyPI, F16);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 16-bit fractional implementation is selected as default.
f16Output = GFLIB\_ControllerPIrAW(f16InErr, &trMyPI);
}

```

### 2.22.5.3 Function [GFLIB\\_ControllerPIrAWSetState\\_FLT](#)

#### Declaration

```
void GFLIB\_ControllerPIrAWSetState\_FLT(tFloat
    fltControllerPIrAWOut, GFLIB\_CONTROLLER\_PIAW\_R\_T\_FLT *const
    pParam);
```

#### Arguments

**Table 180. [GFLIB\\_ControllerPIrAWSetState\\_FLT](#) arguments**

Type	Name	Direction	Description
<a href="#">tFloat</a>	fltControllerPIrAWOut	input	Required output of the <a href="#">GFLIB_ControllerPIrAW</a> .
<a href="#">GFLIB_CONTROLLER_PIAW_R_T_FLT</a> *const	pParam	input, output	Pointer to the structure with <a href="#">GFLIB_ControllerPIrAW</a> state.

#### Code Example

```

#include "gflib.h"

tFloat fltInErr;
tFloat fltOutput;

GFLIB\_CONTROLLER\_PIAW\_R\_T\_FLT trMyPI = GFLIB\_CONTROLLER\_PIAW\_R\_DEFAULT\_FLT;

void main(void)
{
    tFloat fltControllerPIrAWOut;

    // Set the input error
    fltInErr = 0.25f;

    // Initialize the controller parameters
    trMyPI.fltCC1sc      = 0.01F;
    trMyPI.fltCC2sc      = 0.02F;
    trMyPI.fltUpperLimit = 1.0F;
    trMyPI.fltLowerLimit = -1.0F;

    // Clear the state variables
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    GFLIB\_ControllerPIrAWInit\_FLT(&trMyPI);
}

```

```
// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
GFLIB_ControllerPIrAWInit(&trMyPI, FLT);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if single precision floating-point implementation is selected
// as default.
GFLIB_ControllerPIrAWInit(&trMyPI);

// Initialize the state variables to predefined values
fltControllerPIrAWOut = 0.03f;
// Alternative 1: API call with postfix
// (only one alternative shall be used).
GFLIB_ControllerPIrAWSSetState_FLT(fltControllerPIrAWOut, &trMyPI);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
GFLIB_ControllerPIrAWSSetState(fltControllerPIrAWOut, &trMyPI, FLT);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if single precision floating-point implementation is selected
// as default.
GFLIB_ControllerPIrAWSSetState(fltControllerPIrAWOut, &trMyPI);
}

// Periodical function or interrupt - control loop
void ControlLoop(void)
{
    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    fltOutput = GFLIB\_ControllerPIrAW\_FLT(fltInErr, &trMyPI);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    fltOutput = GFLIB_ControllerPIrAW(fltInErr, &trMyPI, FLT);

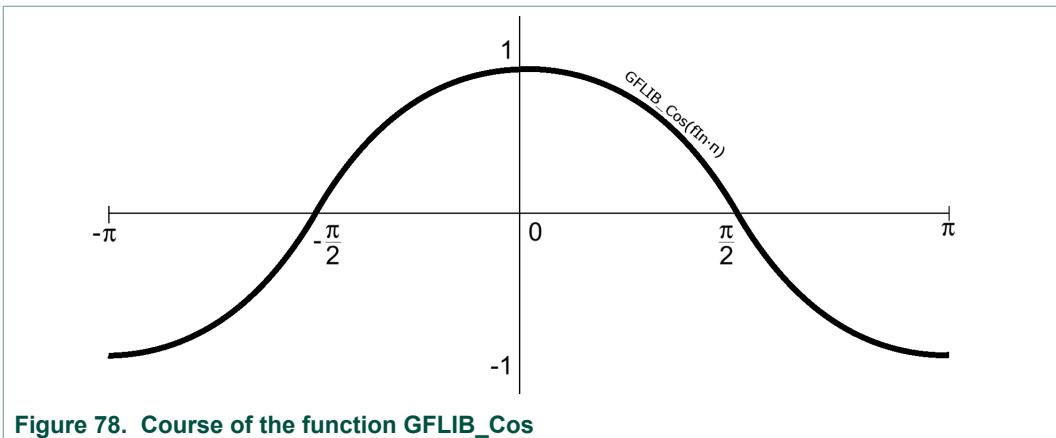
    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if single precision floating-point implementation is selected
    // as default.
    fltOutput = GFLIB_ControllerPIrAW(fltInErr, &trMyPI);
}
```

## 2.23 Function GFLIB\_Cos

This function implements an approximation of cosine function.

### Description

The GFLIB\_Cos function provides a computational method for calculation of the trigonometric cosine function  $\cos(x)$ , using the piece-wise polynomial approximation.



When both sine and cosine of the same argument is needed, use [GFLIB\\_SinCos](#) function instead for better performance.

### Re-entrancy

The function is re-entrant.

#### 2.23.1 Function GFLIB\_Cos\_F32

##### Declaration

```
tFrac32 GFLIB_Cos_F32(tFrac32 f32In, const GFLIB_COS_T_F32 *const pParam);
```

##### Arguments

**Table 181. GFLIB\_Cos\_F32 arguments**

Type	Name	Direction	Description
tFrac32	f32In	input	Input argument is a 32-bit number that contains an angle in radians from interval $[-\pi, \pi]$ normalized between $[-1, 1]$ .
const GFLIB_COS_T_F32 *const	pParam	input	Pointer to an array of Taylor coefficients. In case the default approximation coefficients are used, the &pParam must be replaced with <a href="#">GFLIB_COS_DEFAULT_F32</a> symbol.

##### Return

The function returns the value of the cosine function of the input argument as a fixed point 32-bit number, normalized between  $[-1, 1]$ .

##### Implementation details

The computational algorithm uses the relation between cosine and sine function as shown in following equation:

$$\cos(x) = \sin(\frac{\pi}{2} + x)$$

Equation GFLIB\_Cos\_F32\_Eq1

The input values are scaled from  $[-\pi, \pi]$  radians to  $[-1, 1]$  in order to fit in the available fixed-point fractional range. The function uses a 9th order Taylor

polynomial approximation; the default polynomial coefficients are provided in the [GFLIB\\_COS\\_DEFAULT\\_F32](#) structure.

**Note:** The input pointer must contain a valid address otherwise a fault may occur (MemManage, BusFault, UsageFault, HardFault).

### Code Example

```
#include "gflib.h"

tFrac32 f32Angle;
tFrac32 f32Output;

void main(void)
{
    // input angle = 0.25 => pi/4
    f32Angle = FRAC32(0.25);

    // output should be 0x5A827E94
    f32Output = GFLIB_Cos_F32(f32Angle, GFLIB\_COS\_DEFAULT\_F32);

    // output should be 0x5A827E94
    f32Output = GFLIB_Cos(f32Angle, GFLIB\_COS\_DEFAULT\_F32, F32);

    // ##### Available only if 32-bit fractional implementation selected
    // as default
    // #####
    // output should be 0x5A827E94
    f32Output = GFLIB_Cos(f32Angle);
}
```

## 2.23.2 Function GFLIB\_Cos\_F16

### Declaration

```
tFrac16 GFLIB_Cos_F16(tFrac16 f16In, const GFLIB\_COS\_T\_F16 *const pParam);
```

### Arguments

Table 182. GFLIB\_Cos\_F16 arguments

Type	Name	Direction	Description
<a href="#">tFrac16</a>	f16In	<b>input</b>	Input argument is a 16-bit number that contains an angle in radians from interval [-π, π) normalized between [-1, 1).
const <a href="#">GFLIB_COS_T_F16</a> *const	pParam	<b>input</b>	Pointer to an array of Taylor coefficients. In case the default approximation coefficients are used, the &pParam must be replaced with <a href="#">GFLIB_COS_DEFAULT_F16</a> symbol.

### Return

The function returns the value of the cosine function of the input argument as a fixed point 16-bit number, normalized between [-1, 1).

### Implementation details

The computational algorithm uses the relation between cosine and sine function as shown in following equation:

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

Equation GFLIB\_Cos\_F16\_Eq1

The input values are scaled from  $[-\pi, \pi]$  radians to  $[-1, 1]$  in order to fit in the available fixed-point fractional range. The function uses a 7th order Taylor polynomial approximation; the default polynomial coefficients are provided in the [GFLIB\\_COS\\_DEFAULT\\_F16](#) structure.

**Note:** The input pointer must contain a valid address otherwise a fault may occur (MemManage, BusFault, UsageFault, HardFault).

### Code Example

```
#include "gflib.h"

tFrac16 f16Angle;
tFrac16 f16Output;

void main(void)
{
    // input angle = 0.25 => pi/4
    f16Angle = FRAC16(0.25);

    // output should be 0x5A82
    f16Output = GFLIB_Cos_F16(f16Angle, GFLIB_COS_DEFAULT_F16);

    // output should be 0x5A82
    f16Output = GFLIB_Cos(f16Angle, GFLIB_COS_DEFAULT_F16, F16);

    // ##### Available only if 16-bit fractional implementation selected
    // as default
    // #####
    // output should be 0x5A82
    f16Output = GFLIB_Cos(f16Angle);
}
```

### 2.23.3 Function GFLIB\_Cos\_FLT

#### Declaration

```
tFloat GFLIB_Cos_FLT(tFloat fltIn, const GFLIB_COS_T_FLT *const pParam);
```

## Arguments

**Table 183.** `GFLIB_Cos_FLT` arguments

Type	Name	Direction	Description
<code>tFloat</code>	<code>fltIn</code>	<code>input</code>	Input argument is a single precision floating point number that contains an angle in radians from interval $[-\pi, \pi]$ .
<code>const GFLIB_COS_T</code> <code>FLT *const</code>	<code>pParam</code>	<code>input</code>	Pointer to an array of approximation coefficients. In case the default approximation coefficients are used, the <code>&amp;pParam</code> must be replaced with <code>GFLIB_COS_DEFAULT_FLT</code> symbol.

## Return

The function returns the value of the cosine function of the input argument as a single precision floating point number.

## Implementation details

The computational algorithm uses the relation between cosine and sine function as shown in following equation:

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

Equation GFLIB\_Cos\_FLT\_Eq1

The function uses a 7th order minimax polynomial approximation; the default polynomial coefficients are provided in the `GFLIB_COS_DEFAULT_FLT` structure.

**Note:** The input pointer must contain a valid address otherwise a fault may occur (`MemManage`, `BusFault`, `UsageFault`, `HardFault`).

The function may raise floating-point exceptions (invalid operation, overflow, underflow, inexact, input denormal). The floating-point unit must be enabled in CPACR register to prevent NOCP UsageFault exception.

## Code Example

```
#include "gflib.h"

tFloat fltAngle;
tFloat fltOutput;

void main(void)
{
    // input angle = 0.785398163 = pi/4
    fltAngle = (tFloat)(0.785398163);

    // output should be 0.70710678
    fltOutput = GFLIB_Cos_FLT(fltAngle, GFLIB_COS_DEFAULT_FLT);

    // output should be 0.70710678
    fltOutput = GFLIB_Cos(fltAngle, GFLIB_COS_DEFAULT_FLT, FLT);

    // ##### Available only if single precision floating point
    // implementation selected as default
}
```

```
// ######
// output should be 0.70710678
fltOutput = GFLIB_Cos(fltAngle);
}
```

## 2.24 Function GFLIB\_Hyst

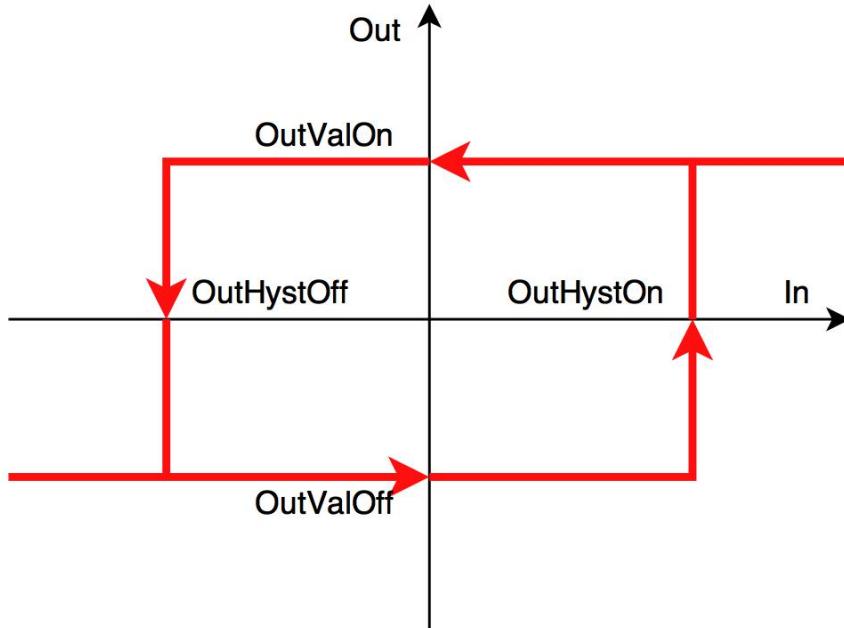
### Description

The GFLIB\_Hyst function provides a computational method for the calculation of a hysteresis (relay) function. The function switches the output between the two predefined values stored in the `OutValOn` and `OutValOff` members of parameter structure. When the value of the input is higher than the upper threshold `HystOn`, then the output value is equal to `OutValOn`. On the other hand, when the input value is lower than the lower threshold `HystOff`, then the output value is equal to `OutValOff`. When the input value is between these two threshold values then the output retains its value (the previous state).

$$\text{OutState}(k) = \begin{cases} \text{OutValOn} & \text{if } \text{In} \geq \text{HystOn} \\ \text{OutValOff} & \text{if } \text{In} \leq \text{HystOff} \\ \text{OutState}(k-1) & \text{if otherwise} \end{cases}$$

Equation GFLIB\_Hyst\_Eq1

A graphical description of GFLIB\_Hyst functionality is shown in [Figure 79](#).



**Figure 79. Hysteresis function**

**Note:** The input pointer must contain a valid address otherwise a fault may occur (MemManage, BusFault, UsageFault, HardFault).

## Re-entrancy

The function is re-entrant.

### 2.24.1 Function `GFLIB_Hyst_F32`

#### Declaration

```
tFrac32 GFLIB_Hyst_F32(tFrac32 f32In, GFLIB_HYST_T_F32 *const pParam);
```

#### Arguments

Table 184. `GFLIB_Hyst_F32` arguments

Type	Name	Direction	Description
<code>tFrac32</code>	<code>f32In</code>	<code>input</code>	Input signal in the form of a 32-bit fixed point number, normalized between [-1, 1).
<code>GFLIB_HYST_T_F32</code> *const	<code>pParam</code>	<code>input, output</code>	Pointer to the structure with parameters and states of the hysteresis function. Arguments of the structure contain fixed point 32-bit values, normalized between [-1, 1).

#### Return

The function returns the value of the hysteresis output, which is equal to either `f32OutValOn` or `f32OutValOff` depending on the value of the input and the state of the function output in the previous calculation step. The output value is interpreted as a fixed point 32-bit number, normalized between [-1, 1).

**Caution:** For correct functionality, the threshold `f32HystOn` value must be greater than the `f32HystOff` value.

**Note:** All parameters and states used by the function can be reset during declaration using the `GFLIB_HYST_DEFAULT_F32` macro.

#### Code Example

```
#include "gplib.h"

tFrac32 f32In;
tFrac32 f32Out;
GFLIB_HYST_T_F32 f32trMyHyst = GFLIB_HYST_DEFAULT_F32;

void main(void)
{
    // Setting parameters for hysteresis
    f32trMyHyst.f32HystOn = FRAC32(0.1289);
    f32trMyHyst.f32HystOff = FRAC32(-0.3634);
    f32trMyHyst.f32OutValOn = FRAC32(0.589);
    f32trMyHyst.f32OutValOff = FRAC32(-0.123);
    f32trMyHyst.f32OutState = FRAC32(-0.3333);

    // input value = -0.41115
    f32In = FRAC32(-0.41115);

    // output should be 0x8FBE76C8 ~ FRAC32(-0.123)
    f32Out = GFLIB_Hyst_F32(f32In, &f32trMyHyst);
```

```

// output should be 0x8FBE76C8 ~ FRAC32 (-0.123)
f32trMyHyst.f32OutState = FRAC32(0);
f32Out = GFLIB_Hyst(f32In, &f32trMyHyst, F32);

// ##### Available only if 32-bit fractional implementation selected
// as default
// #####
// output should be 0x8FBE76C8 ~ FRAC32 (-0.123)
f32trMyHyst.f32OutState = FRAC32(0);
f32Out = GFLIB_Hyst(f32In, &f32trMyHyst);
}

```

## 2.24.2 Function GFLIB\_Hyst\_F16

### Declaration

```
tFrac16 GFLIB_Hyst_F16(tFrac16 f16In, GFLIB\_HYST\_T\_F16 *const pParam);
```

### Arguments

Table 185. GFLIB\_Hyst\_F16 arguments

Type	Name	Direction	Description
<a href="#">tFrac16</a>	f16In	input	Input signal in the form of a 16-bit fixed point number, normalized between [-1, 1).
<a href="#">GFLIB_HYST_T_F16</a> *const	pParam	input, output	Pointer to the structure with parameters and states of the hysteresis function. Arguments of the structure contain fixed point 16-bit values, normalized between [-1, 1).

### Return

The function returns the value of the hysteresis output, which is equal to either f16OutValOn or f16OutValOff depending on the value of the input and the state of the function output in the previous calculation step. The output value is interpreted as a fixed point 16-bit number, normalized between [-1, 1].

**Caution:** For correct functionality, the threshold f16HystOn value must be greater than the f16HystOff value.

**Note:** All parameters and states used by the function can be reset during declaration using the [GFLIB\\_HYST\\_DEFAULT\\_F16](#) macro.

### Code Example

```

#include "gplib.h"

tFrac16 f16In;
tFrac16 f16Out;
GFLIB\_HYST\_T\_F16 f16trMyHyst = GFLIB\_HYST\_DEFAULT\_F16;

void main(void)
{
    // Setting parameters for hysteresis

```

```

f16trMyHyst.f16HystOn = FRAC16(0.1289);
f16trMyHyst.f16HystOff = FRAC16(-0.3634);
f16trMyHyst.f16OutValOn = FRAC16(0.589);
f16trMyHyst.f16OutValOff = FRAC16(-0.123);
f16trMyHyst.f16OutState = FRAC16(-0.3333);

// input value = -0.41115
f16In = FRAC16(-0.41115);

// output should be 0x8FBE ~ FRAC16(-0.123)
f16Out = GFLIB_Hyst_F16(f16In, &f16trMyHyst);

// output should be 0x8FBE ~ FRAC16(-0.123)
f16trMyHyst.f16OutState = FRAC16(0);
f16Out = GFLIB_Hyst(f16In, &f16trMyHyst, F16);

// ##### Available only if 16-bit fractional implementation selected
// as default
// #####
// output should be 0x8FBE ~ FRAC16(-0.123)
f16trMyHyst.f16OutState = FRAC16(0);
f16Out = GFLIB_Hyst(f16In, &f16trMyHyst);
}

```

### 2.24.3 Function GFLIB\_Hyst\_FLT

#### Declaration

```
tFloat GFLIB_Hyst_FLT(tFloat fltIn, GFLIB\_HYST\_T\_FLT *const pParam);
```

#### Arguments

Table 186. GFLIB\_Hyst\_FLT arguments

Type	Name	Direction	Description
<a href="#">tFloat</a>	fltIn	input	Input value, in single precision floating point data format.
<a href="#">GFLIB_HYST_T_FLT</a> *const	pParam	input, output	Pointer to the structure with parameters and states of the hysteresis function. Arguments of the structure contain a single precision floating point values.

#### Return

The function returns the value of the hysteresis output, which is equal to either `fltOutValOn` or `fltOutValOff` depending on the value of the input and the state of the function output in the previous calculation step. The output value is in single precision floating point format.

**Note:** The function may raise floating-point exceptions (invalid operation, input denormal). The floating-point unit must be enabled in CPACR register to prevent NOCP UsageFault exception.

**Caution:** For correct functionality, the threshold `flthystOn` value must be greater than the `flthystOff` value.

**Note:** All parameters and states used by the function can be reset during declaration using the [GFLIB\\_HYST\\_DEFAULT\\_FLT](#) macro.

### Code Example

```
#include "gplib.h"

tFloat fltIn;
tFloat fltOut;
GFLIB_HYST_T_FLT flttrMyHyst = GFLIB_HYST_DEFAULT_FLT;

void main(void)
{
    // Setting parameters for hysteresis
    flttrMyHyst.fltHystOn = (tFloat)(0.1289);
    flttrMyHyst.fltHystOff = (tFloat)(-0.3634);
    flttrMyHyst.fltOutValOn = (tFloat)(0.589);
    flttrMyHyst.fltOutValOff = (tFloat)(-0.123);
    flttrMyHyst.fltOutState = (tFloat)(-0.3333);

    // input value = -0.41115
    fltIn = (tFloat)(-0.41115);

    // output should be -0.123
    fltOut = GFLIB_Hyst_FLT(fltIn, &flttrMyHyst);

    // output should be -0.123
    flttrMyHyst.fltOutState = (tFloat)(0);
    fltOut = GFLIB_Hyst(fltIn, &flttrMyHyst, FLT);

    // ##### Available only if single precision floating point
    // implementation selected as default
    // #####
    // output should be -0.123
    flttrMyHyst.fltOutState = (tFloat)(0);
    fltOut = GFLIB_Hyst(fltIn, &flttrMyHyst);
}
```

## 2.25 Function GFLIB\_IntegratorTR

The function calculates a discrete implementation of the integrator (sum), discretized using a trapezoidal (Bilinear) transformation with overflow on range boundaries.

### Description

The function GFLIB\_IntegratorTR implements a discrete integrator using trapezoidal (Bilinear) transformation.

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

$$u(t) = \int_0^t e(t) dt$$

Equation GFLIB\_IntegratorTR\_Eq1

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

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

Equation GFLIB\_IntegratorTR\_Eq2

Transforming equation [GFLIB\\_IntegratorTR\\_Eq2](#) into a digital time domain using Bilinear transformation, leads to the following transfer function:

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

Equation GFLIB\_IntegratorTR\_Eq3

where  $T_s$  is the sampling period of the system. The discrete implementation of the digital transfer function [GFLIB\\_IntegratorTR\\_Eq3](#) is as follows:

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

Equation GFLIB\_IntegratorTR\_Eq4

**Note:** The input pointer must contain a valid address otherwise a fault may occur (MemManage, BusFault, UsageFault, HardFault).

## Re-entrancy

The function is re-entrant.

### 2.25.1 Function GFLIB\_IntegratorTR\_F32

#### Declaration

```
tFrac32 GFLIB_IntegratorTR_F32 (tFrac32 f32In,
                                GFLIB_INTEGRATOR_TR_T_F32 *const pParam);
```

#### Arguments

Table 187. GFLIB\_IntegratorTR\_F32 arguments

Type	Name	Direction	Description
tFrac32	f32In	input	Input argument to be integrated.
GFLIB_INTEGRATOR_TR_T_F32 *const	pParam	input, output	Pointer to the integrator parameters structure.

#### Return

The function returns a 32-bit value in format Q1.31, which represents the actual integrated value of the input signal with overflow on range boundaries.

#### Implementation details

Considering fractional math implementation, the integrator input and output maximal values (scales) must be known. Then the discrete implementation is given as follows:

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

Equation GFLIB\_IntegratorTR\_F32\_Eq1

where  $E_{MAX}$  is the input scale and  $U_{MAX}$  is the output scale. Then integrator constant  $C1$  is defined as:

$$C1_f = \frac{T_s}{2} \cdot \frac{E_{MAX}}{U_{MAX}}$$

Equation GFLIB\_IntegratorTR\_F32\_Eq2

In order to implement the discrete form integrator as in [GFLIB\\_IntegratorTR\\_F32\\_Eq1](#) on a fixed point platform, the value of  $C1_f$  coefficient must reside in the fractional range [-1,1). Therefore, scaling must be introduced as follows:

$$f32C1 = C1_f \cdot 2^{u16NShift}$$

Equation GFLIB\_IntegratorTR\_F32\_Eq3

The introduced scaling is chosen such that coefficient  $f32C1$  fits into fractional range [-1,1). To simplify the implementation, this scaling is chosen to be a power of 2, so the final scaling is a simple shift operation using the  $u16NShift$  variable. Hence, the shift is calculated as:

$$u16NShift = ceil\left(\frac{\log(abs(C1_f))}{\log(2)}\right)$$

Equation GFLIB\_IntegratorTR\_F32\_Eq4

If the output exceeds the fractional range [-1,1), an overflow occurs. This behavior allows continual integration of an angular velocity of a rotor to obtain the actual rotor position, assuming the output range corresponds to one complete revolution.

**Note:** All parameters and states used by the function can be reset during declaration using the [GFLIB\\_INTEGRATOR\\_TR\\_DEFAULT\\_F32](#) macro.

The specified accuracy of the function is not guaranteed in cases when any of the terms in [GFLIB\\_IntegratorTR\\_F32\\_Eq1](#) overflows.

### Code Example

```
#include "gflib.h"

tFrac32 f32In;
tFrac32 f32Out;

// Definition of one integrator instance
GFLIB_INTEGRATOR_TR_T_F32 trMyIntegrator = GFLIB_INTEGRATOR_TR_DEFAULT_F32;

void main(void)
{
    // Setting parameters for integrator, Ts = 100e-4, E_MAX=U_MAX=1
    trMyIntegrator.f32C1      = FRAC32((100e-4)/2.0);
    trMyIntegrator.u16NShift = (TU16)0;

    // input value = 0.5
    f32In = FRAC32(0.5);

    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
}
```

```

// Initialization of integrator internal state for zero output:
GFLIB_IntegratorTRSetState_F32(0, &trMyIntegrator);
// Calculation of one iteration of the integrator:
f32Out = GFLIB_IntegratorTR_F32(f32In, &trMyIntegrator);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
// Initialization of integrator internal state for zero output:
GFLIB_IntegratorTRSetState(0, &trMyIntegrator, F32);
// Calculation of one iteration of the integrator:
f32Out = GFLIB_IntegratorTR(f32In, &trMyIntegrator, F32);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 32-bit fractional implementation is selected as default.
// Initialization of integrator internal state for zero output:
GFLIB_IntegratorTRSetState(0, &trMyIntegrator);
// Calculation of one iteration of the integrator:
f32Out = GFLIB_IntegratorTR(f32In, &trMyIntegrator);

// Expected output value: 0x0051EB85
}

```

## 2.25.2 Function GFLIB\_IntegratorTR\_F16

### Declaration

```
tFrac16 GFLIB_IntegratorTR_F16(tFrac16 f16In,
GFLIB_INTEGRATOR_TR_T_F16 *const pParam);
```

### Arguments

**Table 188. GFLIB\_IntegratorTR\_F16 arguments**

Type	Name	Direction	Description
tFrac16	f16In	input	Input argument to be integrated.
GFLIB_INTEGRATOR_TR_T_F16 *const	pParam	input, output	Pointer to the integrator parameters structure.

### Return

The function returns a 16-bit value in format Q1.15, which represents the actual integrated value of the input signal with overflow on range boundaries.

### Implementation details

Considering fractional math implementation, the integrator input and output maximal values (scales) must be known. Then the discrete implementation is given as follows:

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

Equation GFLIB\_IntegratorTR\_F16\_Eq1

where  $E_{MAX}$  is the input scale and  $U_{MAX}$  is the output scale. Then integrator constant  $C1$  is defined as:

$$C1_f = \frac{T_s}{2} \cdot \frac{E_{MAX}}{U_{MAX}}$$

Equation GFLIB\_IntegratorTR\_F16\_Eq2

In order to implement the discrete form integrator as in [GFLIB\\_IntegratorTR\\_F16\\_Eq1](#) on a fixed point platform, the value of  $C1_f$  coefficient must reside in the fractional range [-1,1). Therefore, scaling must be introduced as follows:

$$f16C1 = C1_f \cdot 2^{u16NShift}$$

Equation GFLIB\_IntegratorTR\_F16\_Eq3

The introduced scaling is chosen such that coefficient  $f16C1$  fits into fractional range [-1,1). To simplify the implementation, this scaling is chosen to be a power of 2, so the final scaling is a simple shift operation using the `u16NShift` variable. Hence, the shift is calculated as:

$$u16NShift = ceil\left(\frac{\log(abs(C1_f))}{\log(2)}\right)$$

Equation GFLIB\_IntegratorTR\_F16\_Eq4

If the output exceeds the fractional range [-1,1), an overflow occurs. This behavior allows continual integration of an angular velocity of a rotor to obtain the actual rotor position, assuming the output range corresponds to one complete revolution.

**Note:** All parameters and states used by the function can be reset during declaration using the [GFLIB\\_INTEGRATOR\\_TR\\_DEFAULT\\_F16](#) macro.

The specified accuracy of the function is not guaranteed in cases when any of the terms in [GFLIB\\_IntegratorTR\\_F16\\_Eq1](#) overflows.

### Code Example

```
#include "gflib.h"

tFrac16 f16In;
tFrac16 f16Out;

// Definition of one integrator instance
GFLIB_INTEGRATOR_TR_T_F16 trMyIntegrator = GFLIB_INTEGRATOR_TR_DEFAULT_F16;

void main(void)
{
    // Setting parameters for integrator, Ts = 100e-4, E_MAX=U_MAX=1
    trMyIntegrator.f16C1      = FRAC16((100e-4)/2.0);
    trMyIntegrator.u16NShift = (tU16)0;

    // input value = 0.5
    f16In = FRAC16(0.5);

    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    // Initialization of integrator internal state for zero output:
    GFLIB_IntegratorTRSetState_F16(0, &trMyIntegrator);
    // Calculation of one iteration of the integrator:
    f16Out = GFLIB_IntegratorTR_F16(f16In, &trMyIntegrator);
}
```

```

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
// Initialization of integrator internal state for zero output:
GFLIB_IntegratorTRSetState(0, &trMyIntegrator, F16);
// Calculation of one iteration of the integrator:
f16Out = GFLIB_IntegratorTR(f16In, &trMyIntegrator, F16);

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 16-bit fractional implementation is selected as default.
// Initialization of integrator internal state for zero output:
GFLIB_IntegratorTRSetState(0, &trMyIntegrator);
// Calculation of one iteration of the integrator:
f16Out = GFLIB_IntegratorTR(f16In, &trMyIntegrator);

// Expected output value: 0x0051
}

```

### 2.25.3 Function GFLIB\_IntegratorTR\_FLT

#### Declaration

```
tFloat GFLIB_IntegratorTR_FLT(tFloat fltIn,
                               GFLIB_INTEGRATOR_TR_T_FLT *const pParam);
```

#### Arguments

Table 189. GFLIB\_IntegratorTR\_FLT arguments

Type	Name	Direction	Description
tFloat	fltIn	input	Input argument to be integrated.
GFLIB_INTEGRATOR_TR_T_FLT *const	pParam	input, output	Pointer to the integrator parameters structure.

#### Return

The function returns a single precision floating point value, which represents the actual integrated value of the input signal.

**Note:** The function may raise floating-point exceptions (invalid operation, overflow, underflow, inexact, input denormal). The floating-point unit must be enabled in CPACR register to prevent NOCP UsageFault exception.

All parameters and states used by the function can be reset during declaration using the [GFLIB\\_INTEGRATOR\\_TR\\_DEFAULT\\_FLT](#) macro.

#### Code Example

```

#include "gflib.h"

tFloat fltIn;
tFloat fltOut;

// Definition of one integrator instance
GFLIB_INTEGRATOR_TR_T_FLT trMyIntegrator = GFLIB_INTEGRATOR_TR_DEFAULT_FLT;

```

```
void main(void)
{
    // Setting parameters for integrator, Ts = 100e-4,
    trMyIntegrator.fltC1 = (tFloat)((100e-4)/2.0);

    // input value = 0.5
    fltIn = (tFloat)0.5;

    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    // Initialization of integrator internal state for zero output:
    GFLIB_IntegratorTRSetState_FLT(0.0f, &trMyIntegrator);
    // Calculation of one iteration of the integrator:
    fltOut = GFLIB_IntegratorTR_FLT(fltIn, &trMyIntegrator);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    // Initialization of integrator internal state for zero output:
    GFLIB_IntegratorTRSetState(0.0f, &trMyIntegrator, FLT);
    // Calculation of one iteration of the integrator:
    fltOut = GFLIB_IntegratorTR(fltIn, &trMyIntegrator, FLT);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if single precision floating-point implementation is selected
    // as default.
    // Initialization of integrator internal state for zero output:
    GFLIB_IntegratorTRSetState(0.0f, &trMyIntegrator);
    // Calculation of one iteration of the integrator:
    fltOut = GFLIB_IntegratorTR(fltIn, &trMyIntegrator);

    // Expected output value: 2.5e-3
}
```

## 2.25.4 Function GFLIB\_IntegratorTRSetState

### Description

This function initializes the GFLIB\_IntegratorTR state variables to achieve the required output values.

**Note:** The input/output pointer must contain a valid address, otherwise a fault may occur (MemManage, BusFault, UsageFault, HardFault).

### Re-entrancy

The function is re-entrant for a different pCtrl.

### 2.25.4.1 Function GFLIB\_IntegratorTRSetState\_F32

#### Declaration

```
void GFLIB_IntegratorTRSetState_F32(tFrac32 f32IntegratorTROut,
                                     GFLIB_INTEGRATOR_TR_T_F32 *const pParam);
```

## Arguments

**Table 190. GFLIB\_IntegratorTRSetState\_F32 arguments**

Type	Name	Direction	Description
tFrac32	f32IntegratorTROut	input	Required output of the GFLIB_IntegratorTR.
GFLIB_INTEGRATOR_TR_T_F32 *const	pParam	input, output	Pointer to the structure with GFLIB_IntegratorTR state.

## Code Example

```
#include "gflib.h"

tFrac32 f32In;
tFrac32 f32Out;

// Definition of one integrator instance
GFLIB_INTEGRATOR_TR_T_F32 trMyIntegrator = GFLIB_INTEGRATOR_TR_DEFAULT_F32;

void main(void)
{
    // Setting parameters for integrator, Ts = 100e-4, E_MAX=U_MAX=1
    trMyIntegrator.f32C1 = FRAC32((100e-4)/2.0);
    trMyIntegrator.u16NShift = (tU16)0;

    // input value = 0.5
    f32In = FRAC32(0.5);

    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    // Initialization of integrator internal state for zero output:
    GFLIB_IntegratorTRSetState_F32(0, &trMyIntegrator);
    // Calculation of one iteration of the integrator:
    f32Out = GFLIB_IntegratorTR_F32(f32In, &trMyIntegrator);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    // Initialization of integrator internal state for zero output:
    GFLIB_IntegratorTRSetState(0, &trMyIntegrator, F32);
    // Calculation of one iteration of the integrator:
    f32Out = GFLIB_IntegratorTR(f32In, &trMyIntegrator);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 32-bit fractional implementation is selected as default.
    // Initialization of integrator internal state for zero output:
    GFLIB_IntegratorTRSetState(0, &trMyIntegrator);
    // Calculation of one iteration of the integrator:
    f32Out = GFLIB_IntegratorTR(f32In, &trMyIntegrator);

    // Expected output value: 0x0051EB85
}
```

### 2.25.4.2 Function `GFLIB_IntegratorTRSetState_F16`

#### Declaration

```
void GFLIB_IntegratorTRSetState_F16(tFrac16 f16IntegratorTROut,
GFLIB_INTEGRATOR_TR_T_F16 *const pParam);
```

#### Arguments

**Table 191. `GFLIB_IntegratorTRSetState_F16` arguments**

Type	Name	Direction	Description
<u>tFrac16</u>	f16IntegratorTROut	<b>input</b>	Required output of the <code>GFLIB_IntegratorTR</code> .
<u>GFLIB_INTEGRATOR_TR_T_F16</u> *const	pParam	<b>input, output</b>	Pointer to the structure with <code>GFLIB_IntegratorTR</code> state.

#### Code Example

```
#include "gflib.h"

tFrac16 f16In;
tFrac16 f16Out;

// Definition of one integrator instance
GFLIB_INTEGRATOR_TR_T_F16 trMyIntegrator = GFLIB_INTEGRATOR_TR_DEFAULT_F16;

void main(void)
{
    // Setting parameters for integrator, Ts = 100e-4, E_MAX=U_MAX=1
    trMyIntegrator.f16C1      = FRAC16((100e-4)/2.0);
    trMyIntegrator.ul6NShift = (tU16)0;

    // input value = 0.5
    f16In = FRAC16(0.5);

    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    // Initialization of integrator internal state for zero output:
    GFLIB_IntegratorTRSetState_F16(0, &trMyIntegrator);
    // Calculation of one iteration of the integrator:
    f16Out = GFLIB_IntegratorTR_F16(f16In, &trMyIntegrator);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    // Initialization of integrator internal state for zero output:
    GFLIB_IntegratorTRSetState(0, &trMyIntegrator, F16);
    // Calculation of one iteration of the integrator:
    f16Out = GFLIB_IntegratorTR(f16In, &trMyIntegrator, F16);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 16-bit fractional implementation is selected as default.
    // Initialization of integrator internal state for zero output:
    GFLIB_IntegratorTRSetState(0, &trMyIntegrator);
    // Calculation of one iteration of the integrator:
    f16Out = GFLIB_IntegratorTR(f16In, &trMyIntegrator);
```

```
// Expected output value: 0x0051
}
```

### 2.25.4.3 Function `GFLIB_IntegratorTRSetState_FLT`

#### Declaration

```
void GFLIB_IntegratorTRSetState_FLT(tFloat fltIntegratorTROut,  
GFLIB_INTEGRATOR_TR_T_FLT *const pParam);
```

#### Arguments

**Table 192. `GFLIB_IntegratorTRSetState_FLT` arguments**

Type	Name	Direction	Description
<code>tFloat</code>	fltIntegratorTROut	input	Required output of the <code>GFLIB_IntegratorTR</code> .
<code>GFLIB_INTEGRATOR_TR_T_FLT</code> *const	pParam	input, output	Pointer to the structure with <code>GFLIB_IntegratorTR</code> state.

#### Code Example

```
#include "gplib.h"

tFloat fltIn;
tFloat fltOut;

// Definition of one integrator instance
GFLIB_INTEGRATOR_TR_T_FLT trMyIntegrator = GFLIB_INTEGRATOR_TR_DEFAULT_FLT;

void main(void)
{
    // Setting parameters for integrator, Ts = 100e-4,
    trMyIntegrator.fltC1 = (tFloat)((100e-4)/2.0);

    // input value = 0.5
    fltIn = (tFloat)0.5;

    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    // Initialization of integrator internal state for zero output:
    GFLIB_IntegratorTRSetState_FLT(0.0f, &trMyIntegrator);
    // Calculation of one iteration of the integrator:
    fltOut = GFLIB_IntegratorTR_FLT(fltIn, &trMyIntegrator);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    // Initialization of integrator internal state for zero output:
    GFLIB_IntegratorTRSetState(0.0f, &trMyIntegrator, FLT);
    // Calculation of one iteration of the integrator:
    fltOut = GFLIB_IntegratorTR(fltIn, &trMyIntegrator, FLT);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if single precision floating-point implementation is selected
    // as default.
    // Initialization of integrator internal state for zero output:
    GFLIB_IntegratorTRSetState(0.0f, &trMyIntegrator);
```

```

// Calculation of one iteration of the integrator:
fltOut = GFLIB_IntegratorTR(fltIn, &trMyIntegrator);

// Expected output value: 2.5e-3
}

```

## 2.26 Function GFLIB\_Limit

This function tests whether the input value is within the upper and lower limits.

### Description

The GFLIB\_Limit function tests whether the input value is within the upper and lower limits. If so, the input value will be returned. If the input value is above the upper limit, the upper limit will be returned. If the input value is below the lower limit, the lower limit will be returned.

The upper and lower limits can be found in the limits structure, supplied to the function as a pointer pParam.

**Note:** *The input pointer must contain a valid address otherwise a fault may occur (MemManage, BusFault, UsageFault, HardFault).*

### Re-entrancy

The function is re-entrant.

#### 2.26.1 Function GFLIB\_Limit\_F32

##### Declaration

```
tFrac32 GFLIB_Limit_F32(tFrac32 f32In, const GFLIB_LIMIT_T_F32
*const pParam);
```

##### Arguments

Table 193. GFLIB\_Limit\_F32 arguments

Type	Name	Direction	Description
tFrac32	f32In	input	Input value.
const GFLIB_LIMIT_T_F32 *const	pParam	input	Pointer to the limits structure.

##### Return

The input value in case the input value is below the limits, or the upper or lower limit if the input value is above these limits.

**Note:** *The function assumes that the upper limit f32UpperLimit is greater than the lower limit f32LowerLimit. Otherwise, the function returns an undefined value.*

##### Code Example

```
#include "gplib.h"
```

```

tFrac32 f32In;
tFrac32 f32Out;
GFLIB_LIMIT_T_F32 f32trMyLimit = GFLIB_LIMIT_DEFAULT_F32;

void main(void)
{
    // upper/lower limits
    f32trMyLimit.f32UpperLimit = FRAC32(0.5);
    f32trMyLimit.f32LowerLimit = FRAC32(-0.5);

    // input value = 0.75
    f32In = FRAC32(0.75);

    // output should be 0x40000000 ~ FRAC32(0.5)
    f32Out = GFLIB_Limit_F32(f32In, &f32trMyLimit);

    // output should be 0x40000000 ~ FRAC32(0.5)
    f32Out = GFLIB_Limit(f32In, &f32trMyLimit, F32);

    // ##### Available only if 32-bit fractional implementation selected
    // as default
    // #####
    // output should be 0x40000000 ~ FRAC32(0.5)
    f32Out = GFLIB_Limit(f32In, &f32trMyLimit);
}

```

## 2.26.2 Function GFLIB\_Limit\_F16

### Declaration

```
tFrac16 GFLIB_Limit_F16(tFrac16 f16In, const GFLIB_LIMIT_T_F16
*const pParam);
```

### Arguments

Table 194. GFLIB\_Limit\_F16 arguments

Type	Name	Direction	Description
tFrac16	f16In	input	Input value.
const GFLIB_LIMIT_T_F16 *const	pParam	input	Pointer to the limits structure.

### Return

The input value in case the input value is below the limits, or the upper or lower limit if the input value is above these limits.

**Note:** The function assumes that the upper limit `f16UpperLimit` is greater than the lower limit `f16LowerLimit`. Otherwise, the function returns an undefined value.

### Code Example

```
#include "gflib.h"

tFrac16 f16In;
```

```

tFrac16 f16Out;
GFLIB_LIMIT_T_F16 f16trMyLimit = GFLIB_LIMIT_DEFAULT_F16;

void main(void)
{
    // upper/lower limits
    f16trMyLimit.f16UpperLimit = FRAC16(0.5);
    f16trMyLimit.f16LowerLimit = FRAC16(-0.5);

    // input value = 0.75
    f16In = FRAC16(0.75);

    // output should be 0x4000 ~ FRAC16(0.5)
    f16Out = GFLIB_Limit_F16(f16In, &f16trMyLimit);

    // output should be 0x4000 ~ FRAC16(0.5)
    f16Out = GFLIB_Limit(f16In, &f16trMyLimit, F16);

    // ##### Available only if 16-bit fractional implementation selected
    // as default
    // #####
    // output should be 0x4000 ~ FRAC16(0.5)
    f16Out = GFLIB_Limit(f16In, &f16trMyLimit);
}

```

### 2.26.3 Function GFLIB\_Limit\_FLT

#### Declaration

```
tFloat GFLIB_Limit_FLT(tFloat fltIn, const GFLIB_LIMIT_T_FLT
*const pParam);
```

#### Arguments

**Table 195. GFLIB\_Limit\_FLT arguments**

Type	Name	Direction	Description
tFloat	fltIn	input	Input value.
const GFLIB_LIMIT_T_FLT *const	pParam	input	Pointer to the limits structure.

#### Return

The input value in case the input value is below the limits, or the upper or lower limit if the input value is above these limits.

**Note:** The function may raise floating-point exceptions (invalid operation, input denormal). The floating-point unit must be enabled in CPACR register to prevent NOCP UsageFault exception.

*The function assumes that the upper limit fltUpperLimit is greater than the lower limit fltLowerLimit. Otherwise, the function returns an undefined value.*

#### Code Example

```
#include "gflib.h"

_tFloat fltIn;
_tFloat fltOut;
_GFLIB_LIMIT_T_FLT flttrMyLimit = _GFLIB_LIMIT_DEFAULT_FLT;

void main(void)
{
    // upper/lower limits
    flttrMyLimit.fltUpperLimit = 0.5;
    flttrMyLimit.fltLowerLimit = -0.5;

    // input value = 0.75
    fltIn = 0.75;

    // output should be 0.5
    fltOut = GFLIB_Limit_FLT(fltIn, &flttrMyLimit);

    // output should be 0.5
    fltOut = GFLIB_Limit(fltIn, &flttrMyLimit, FLT);

    // ##### Available only if single precision floating point
    // implementation selected as default
    // #####
    // output should be 0.5
    fltOut = GFLIB_Limit(fltIn, &flttrMyLimit);
}
```

## 2.27 Function GFLIB\_Log10\_FLT

This function calculates a base-10 logarithm of an absolute value.

### Declaration

```
_tFloat GFLIB_Log10_FLT(_tFloat fltIn, const _GFLIB_LOG10_T_FLT
*const pParam);
```

### Arguments

**Table 196. GFLIB\_Log10\_FLT arguments**

Type	Name	Direction	Description
<code>tFloat</code>	fltIn	input	Input argument is a 32-bit number that contains a single precision floating point value.
<code>const _GFLIB_LOG10_T_FLT *const</code>	pParam	input	Pointer to an array of approximation coefficients.

### Return

The function returns  $\log_{10}(\text{abs}(\text{fltIn}))$  as a single precision floating point number.

## Description

The function calculates a base-10 logarithm of the absolute value of the input. The default polynomial coefficients are provided in the [GFLIB\\_LOG10\\_DEFAULT\\_FLT](#) structure.

Use [GFLIB\\_VLog10\\_FLT](#) vectorized function instead if there is an array of inputs to be calculated. The vectorized function can process large arrays faster.

**Note:** The function may raise floating-point exceptions (overflow, underflow, inexact, input denormal). The floating-point unit must be enabled in CPACR register to prevent NOCP UsageFault exception.

## Code Example

```
#include "gplib.h"

tFloat fltInput, fltOutput;

void main(void)
{
    // input value = 1.0
    fltInput = 1.0F;

    // output should be 0
    fltOutput = GFLIB_Log10_FLT(fltInput, GFLIB_LOG10_DEFAULT_FLT);

    // output should be 0
    fltOutput = GFLIB_Log10(fltInput, GFLIB_LOG10_DEFAULT_FLT, FLT);

    // ##### Available only if single precision floating point
    // implementation selected as default
    // #####
    // output should be 0
    fltOutput = GFLIB_Log10(fltInput);
}
```

## 2.28 Function GFLIB\_LowerLimit

This function tests whether the input value is above the lower limit.

## Description

The function tests whether the input value is above the lower limit. If so, the input value will be returned. Otherwise, if the input value is below the lower limit, the lower limit will be returned.

The lower limit `LowerLimit` can be found in the limits structure, supplied to the function as a pointer `pParam`.

**Note:** The input pointer must contain a valid address otherwise a fault may occur (MemManage, BusFault, UsageFault, HardFault).

## Re-entrancy

The function is re-entrant.

### 2.28.1 Function `GFLIB_LowerLimit_F32`

#### Declaration

```
tFrac32 GFLIB_LowerLimit_F32(tFrac32 f32In, const
GFLIB_LOWERLIMIT_T_F32 *const pParam);
```

#### Arguments

Table 197. `GFLIB_LowerLimit_F32` arguments

Type	Name	Direction	Description
<code>tFrac32</code>	<code>f32In</code>	<code>input</code>	Input value.
<code>const GFLIB_LOWERLIMIT_T_F32 *const</code>	<code>pParam</code>	<code>input</code>	Pointer to the limits structure.

#### Return

The input value in case the input value is above the limit, or the lower limit if the input value is below the limit.

#### Code Example

```
#include "gflib.h"

tFrac32 f32In;
tFrac32 f32Out;
GFLIB_LOWERLIMIT_T_F32 f32trMyLowerLimit = GFLIB_LOWERLIMIT_DEFAULT_F32;

void main(void)
{
    // lower limit
    f32trMyLowerLimit.f32LowerLimit = FRAC32(0.5);

    // input value = 0.75
    f32In = FRAC32(0.75);

    // output should be 0x60000000 ~ FRAC32(0.75)
    f32Out = GFLIB_LowerLimit_F32(f32In, &f32trMyLowerLimit);

    // output should be 0x60000000 ~ FRAC32(0.75)
    f32Out = GFLIB_LowerLimit(f32In, &f32trMyLowerLimit, F32);

    // ##### Available only if 32-bit fractional implementation selected
    // as default
    // #####
    // output should be 0x60000000 ~ FRAC32(0.75)
    f32Out = GFLIB_LowerLimit(f32In, &f32trMyLowerLimit);
}
```

## 2.28.2 Function `GFLIB_LowerLimit_F16`

### Declaration

```
tFrac16 GFLIB_LowerLimit_F16(tFrac16 f16In, const
GFLIB_LOWERLIMIT_T_F16 *const pParam);
```

### Arguments

Table 198. `GFLIB_LowerLimit_F16` arguments

Type	Name	Direction	Description
<code>tFrac16</code>	<code>f16In</code>	<code>input</code>	Input value.
<code>const GFLIB_LOWERLIMIT_T_F16 *const</code>	<code>pParam</code>	<code>input</code>	Pointer to the limits structure.

### Return

The input value in case the input value is above the limit, or the lower limit if the input value is below the limit.

### Code Example

```
#include "gplib.h"

tFrac16 f16In;
tFrac16 f16Out;
GFLIB_LOWERLIMIT_T_F16 f16trMyLowerLimit = GFLIB_LOWERLIMIT_DEFAULT_F16;

void main(void)
{
    // lower limit
    f16trMyLowerLimit.f16LowerLimit = FRAC16(0.5);

    // input value = 0.75
    f16In = FRAC16(0.75);

    // output should be 0x6000 ~ FRAC16(0.75)
    f16Out = GFLIB_LowerLimit_F16(f16In,&f16trMyLowerLimit);

    // output should be 0x6000 ~ FRAC16(0.75)
    f16Out = GFLIB_LowerLimit(f16In,&f16trMyLowerLimit,F16);

    // #####
    // Available only if 16-bit fractional implementation selected
    // as default
    // #####
    // output should be 0x6000 ~ FRAC16(0.75)
    f16Out = GFLIB_LowerLimit(f16In,&f16trMyLowerLimit);
}
```

### 2.28.3 Function `GFLIB_LowerLimit_FLT`

#### Declaration

```
tFloat GFLIB_LowerLimit_FLT(tFloat fltIn, const
GFLIB_LOWERLIMIT_T_FLT *const pParam);
```

#### Arguments

Table 199. `GFLIB_LowerLimit_FLT` arguments

Type	Name	Direction	Description
<code>tFloat</code>	<code>fltIn</code>	<code>input</code>	Input value.
<code>const GFLIB_LOWERLIMIT_T_FLT</code> <code>*const</code>	<code>pParam</code>	<code>input</code>	Pointer to the limits structure.

#### Return

The input value in case the input value is above the limit, or the lower limit if the input value is below the limit.

**Note:** The function may raise floating-point exceptions (invalid operation, input denormal). The floating-point unit must be enabled in CPACR register to prevent NOCP UsageFault exception.

#### Code Example

```
#include "gplib.h"

tFloat fltIn;
tFloat fltOut;
GFLIB_LOWERLIMIT_T_FLT flttrMyLowerLimit = GFLIB_LOWERLIMIT_DEFAULT_FLT;

void main(void)
{
    // lower limit
    flttrMyLowerLimit.fltLowerLimit = 0.5;

    // input value = 0.75
    fltIn = (tFloat) 0.75;

    // output should be 0.75
    fltOut = GFLIB_LowerLimit_FLT(fltIn,&flttrMyLowerLimit);

    // output should be 0.75
    fltOut = GFLIB_LowerLimit(fltIn,&flttrMyLowerLimit,FLT);

    // #####
    // Available only if single precision floating point
    // implementation selected as default
    // #####
    // #####
    // output should be 0.75
    fltOut = GFLIB_LowerLimit(fltIn,&flttrMyLowerLimit);
}
```

## 2.29 Function `GFLIB_Lut1D`

This function implements the one-dimensional look-up table.

### Description

The GFLIB\_Lut1D function performs one dimensional linear interpolation over a table of data. The data is assumed to represent a one dimensional function sampled at equidistant points. The following interpolation formula is used:

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

Equation GFLIB\_Lut1D\_Eq1

where:

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

**Note:** The input pointer must contain a valid address and the range of the input abscissa values must fall within the range of the interpolating data table otherwise a fault may occur (MemManage, BusFault, UsageFault, HardFault).

### Re-entrancy

The function is re-entrant.

#### 2.29.1 Function `GFLIB_Lut1D_F32`

##### Declaration

```
tFrac32 GFLIB_Lut1D_F32(tFrac32 f32In, const GFLIB_LUT1D_T_F32
*const pParam);
```

##### Arguments

Table 200. GFLIB\_Lut1D\_F32 arguments

Type	Name	Direction	Description
<code>tFrac32</code>	f32In	input	The abscissa for which 1D interpolation is performed.
<code>const GFLIB_LUT1D_T_F32 *const</code>	pParam	input	Pointer to the parameters structure with parameters of the look-up table function.

##### Return

The interpolated value from the look-up table with 16-bit accuracy.

##### Implementation details

The interpolating intervals are defined in the table provided by the `pf32Table` member of the parameters structure. The table contains ordinate values consecutively over the entire interpolating range. The abscissa values are assumed to be defined implicitly by a

single interpolating interval length and a table index, while the interpolating index zero is the table element pointed to by the `pf32Table` parameter. The abscissa value is equal to the multiplication of the interpolating index and the interpolating interval length. For example, let's consider the following interpolating table:

**Table 201. GFLIB\_Lut1D example table**

ordinate (y)	interpolating index	abscissa (x)
-0.5	-1	-1*( 2 <sup>-1</sup> )
0.0	0	0*( 2 <sup>-1</sup> )
0.25	1	1*( 2 <sup>-1</sup> )
0.5	N/A	2*( 2 <sup>-1</sup> )

The [Table 201](#) contains 4 interpolating points (note four rows). The interpolating interval length in this example is equal to 2<sup>-1</sup>. The `pf32Table` parameter points to the second row, defining also the interpolating index 0. The x-coordinates of the interpolating points are calculated in the right column.

It should be noted that the `pf32Table` pointer does not have to point to the start of the memory area of ordinate values. Therefore, the interpolating index can be positive or negative or, even, does not have to have zero in its range.

A special algorithm is used to make the computation efficient, however, under some additional assumptions, as provided below:

- $fIn$  is in range  $\langle a, b \rangle$ , where  $|a| + |b| = 1$
- the values of the interpolated function are in the 32-bit fixed point format
- the length of each interpolating interval is equal to  $2^{-uShamOffset}$ , where `uShamOffset` is an integer in the range of 1, 2, ... 29
- the provided abscissa for interpolation is in the 32-bit fixed point format

The algorithm performs the following steps:

1. Compute the index representing the interval `sIntvl`, in which the linear interpolation will be performed:

$$sIntvl = \left\lfloor \frac{fIn}{2^{uShamOffset}} \right\rfloor$$

Equation GFLIB\_Lut1D\_F32\_Eq1

where the `uShamOffset` is the shift amount provided in the parameters structure as the member `pParam->u32ShamOffset`.

2. Compute the abscissa offset within an interpolating interval  $\Delta x$ :

$$\Delta x = \frac{x}{2^{uShamOffset}} - sIntvl$$

Equation GFLIB\_Lut1D\_F32\_Eq2

where the `uShamOffset` is the shift amount provided in the parameters structure as the member `pParam->u32ShamOffset`.

3. Compute the interpolated value by the linear interpolation between the ordinates read from the table at the start and the end of the computed interval. The computed abscissa offset is used for the linear interpolation.

$$\begin{aligned}
 y_1 &= pTable[sIntvl] \\
 y_2 &= pTable[sIntvl + 1] \\
 y &= y_1 + (y_2 - y_1) \cdot \Delta x
 \end{aligned}$$

Equation GFLIB\_Lut1D\_F32\_Eq3

where  $y$ ,  $y_1$  and  $y_2$  are, respectively, the interpolated value, the ordinate at the start of the interpolating interval, the ordinate at the end of the interpolating interval. The `pTable` is the table containing ordinate values and it is provided as a pointer in the parameters structure `pParam->pf32Table`.

The computations are performed with a 16-bit accuracy. In particular, the 16 least significant bits are ignored in all multiplications.

It should be noted that the input abscissa value can be positive or negative. If it is, positive then the ordinate values are read as in the ordinary data array, that is, at or after the data pointer provided in the parameters structure (`pParam->pf32Table`). However, if it is negative, then the ordinate values are read from the memory with lower address than the `pParam->pfltTable` pointer.

**Note:** The function performs a linear interpolation.

**Caution:** The function does not check whether the input abscissa value is within the range allowed by the interpolating data table `pParam->pf32Table`. If the computed interval index points to data outside the provided data table, then the interpolation will be computed with invalid data. The range of the input abscissa value depends on the position of the pointer in the interpolating data table. Sum of the absolute values of the lower and upper border values is equal to the [FRACT\\_MAX](#). For a better understanding, please, see the extended code example.

### Code Example

```

#include "gflib.h"

tFrac32 f32In;
tFrac32 f32Out;
GFLIB_LUT1D_T_F32 trf32MyLut1D = GFLIB_LUT1D_DEFAULT_F32;
tFrac32 pf32Table1D[9] = {FRAC32(0.8), FRAC32(0.1), FRAC32(-0.2), FRAC32(0.7),
                          FRAC32(0.2), FRAC32(-0.3), FRAC32(-0.8),
                          FRAC32(0.91), FRAC32(0.99)} ;

void main(void)
{
    // #####
    // Pointer is located in the middle of the interpolating data table.
    // #####
    // setting parameters for Lut1D function
    trf32MyLut1D.u32ShamOffset = (tU32)3;
    trf32MyLut1D.pf32Table = &(pf32Table1D[4]);

    // input vector = -0.5
    f32In = FRAC32(-0.5);

    // output should be 0x6666666 ~ FRAC32(0.8)
    f32Out = GFLIB_Lut1D_F32(f32In, &trf32MyLut1D);
}

```

```

// output should be 0x66666666 ~ FRAC32(0.8)
f32Out = GFLIB_Lut1D(f32In, &trf32MyLut1D, F32);

// available only if 32-bit fractional implementation
// selected as default
// output should be 0x66666666 ~ FRAC32(0.8)
f32Out = GFLIB_Lut1D(f32In, &trf32MyLut1D);

// ######
// Pointer is located at the beginning of the interpolating data table.
// #####
// setting parameters for Lut1D function
trf32MyLut1D.u32ShamOffset = (tU32)3;
trf32MyLut1D.pf32Table = &(pf32Table1D[0]);

// input vector = 0
f32In = FRAC32(0);

// output should be 0x66666666 ~ FRAC32(0.8)
f32Out = GFLIB_Lut1D_F32(f32In, &trf32MyLut1D);

// output should be 0x66666666 ~ FRAC32(0.8)
f32Out = GFLIB_Lut1D(f32In, &trf32MyLut1D);

// available only if 32-bit fractional implementation
// selected as default
// output should be 0x66666666 ~ FRAC32(0.8)
f32Out = GFLIB_Lut1D(f32In, &trf32MyLut1D);

// #####
// Pointer is located at the end of the interpolating data table.
// #####
// setting parameters for Lut1D function
trf32MyLut1D.u32ShamOffset = (tU32)3;
trf32MyLut1D.pf32Table = &(pf32Table1D[8]);

// input vector = -1
f32In = FRAC32(-1);

// output should be 0x66666666 ~ FRAC32(0.8)
f32Out = GFLIB_Lut1D_F32(f32In, &trf32MyLut1D);

// output should be 0x66666666 ~ FRAC32(0.8)
f32Out = GFLIB_Lut1D(f32In, &trf32MyLut1D, F32);

// available only if 32-bit fractional implementation
// selected as default
// output should be 0x66666666 ~ FRAC32(0.8)
f32Out = GFLIB_Lut1D(f32In, &trf32MyLut1D);
}

```

## 2.29.2 Function GFLIB\_Lut1D\_F16

### Declaration

```
tFrac16 GFLIB_Lut1D_F16(tFrac16 f16In, const GFLIB_LUT1D_T_F16
*const pParam);
```

**Arguments****Table 202. GFLIB\_Lut1D\_F16 arguments**

Type	Name	Direction	Description
tFrac16	f16In	input	The abscissa for which 1D interpolation is performed.
const GFLIB_LUT1D_T_F16 *const	pParam	input	Pointer to the parameters structure with parameters of the look-up table function.

**Return**

The interpolated value from the look-up table.

**Implementation details**

The interpolating intervals are defined in the table provided by the `pf16Table` member of the parameters structure. The table contains ordinate values consecutively over the entire interpolating range. The abscissa values are assumed to be defined implicitly by a single interpolating interval length and a table index, while the interpolating index zero is the table element pointed to by the `pf16Table` parameter. The abscissa value is equal to the multiplication of the interpolating index and the interpolating interval length. For example, let's consider the following interpolating table:

**Table 203. GFLIB\_Lut1D example table**

ordinate (y)	interpolating index	abscissa (x)
-0.5	-1	-1*( 2 <sup>-1</sup> )
0.0	0	0*( 2 <sup>-1</sup> )
0.25	1	1*( 2 <sup>-1</sup> )
0.5	N/A	2*( 2 <sup>-1</sup> )

The [Table 203](#) contains 4 interpolating points (note four rows). The interpolating interval length in this example is equal to 2<sup>-1</sup>. The `pf16Table` parameter points to the second row, defining also the interpolating index 0. The x-coordinates of the interpolating points are calculated in the right column.

It should be noted that the `pf16Table` pointer does not have to point to the start of the memory area of ordinate values. Therefore, the interpolating index can be positive or negative or, even, does not have to have zero in its range.

A special algorithm is used to make the computation efficient, however, under some additional assumptions, as provided below:

- `fIn` is in range  $\langle a, b \rangle$ , where  $|a| + |b| = 1$
- the values of the interpolated function are in the 16-bit fixed point format
- the length of each interpolating interval is equal to  $2^{-n}$ , where  $n$  is an integer in the range of 1, 2, ... 13
- the provided abscissa for interpolation is in the 16-bit fixed point format

The algorithm performs the following steps:

1. Compute the index representing the interval `sIntvl`, in which the linear interpolation will be performed:

$$sIntvl = \left\lfloor \frac{fIn}{2^{uShamOffset}} \right\rfloor$$

Equation GFLIB\_Lut1D\_F16\_Eq1

where the `uShamOffset` is the shift amount provided in the parameters structure as the member `pParam->u16ShamOffset`.

2. Compute the abscissa offset within an interpolating interval  $\Delta x$ :

$$\Delta x = \frac{x}{2^{uShamOffset}} - sIntvl$$

Equation `GFLIB_Lut1D_F16_Eq2`

where the `uShamOffset` is the shift amount provided in the parameters structure as the member `pParam->u16ShamOffset`.

3. Compute the interpolated value by the linear interpolation between the ordinates read from the table at the start and the end of the computed interval. The computed abscissa offset is used for the linear interpolation.

$$y_1 = pTable[sIntvl]$$

$$y_2 = pTable[sIntvl + 1]$$

$$y = y_1 + (y_2 - y_1) \cdot \Delta x$$

Equation `GFLIB_Lut1D_F16_Eq3`

where  $y$ ,  $y_1$  and  $y_2$  are, respectively, the interpolated value, the ordinate at the start of the interpolating interval, the ordinate at the end of the interpolating interval. The `pTable` is the table containing ordinate values and it is provided as a pointer in the parameters structure `pParam->pf16Table`.

It should be noted that the input abscissa value can be positive or negative. If it is, positive then the ordinate values are read as in the ordinary data array, that is, at or after the data pointer provided in the parameters structure (`pParam->pf16Table`). However, if it is negative, then the ordinate values are read from the memory, which is located behind the `pParam->pf16Table` pointer.

**Note:** The function performs a linear interpolation.

**Caution:** The function does not check whether the input abscissa value is within the range allowed by the interpolating data table `pParam->pf16Table`. If the computed interval index points to data outside the provided data table, then the interpolation will be computed with invalid data. The range of the input abscissa value depends on the position of the pointer in the interpolating data table. Sum of the absolute values of the lower and upper border values is equal to the [SFRAC16\\_MAX](#). For a better understanding, please, see the extended code example.

### Code Example

```
#include "gflib.h"

tFrac16 f16In;
tFrac16 f16Out;
GFLIB_LUT1D_T_F16 trf16MyLut1D = GFLIB_LUT1D_DEFAULT_F16;
tFrac16 pf16Table1D[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)
{
```

```
// ######
// Pointer is located in the middle of the interpolating data table.
// #####
// setting parameters for Lut1D function
trf16MyLut1D.u16ShamOffset = (tU16)3;
trf16MyLut1D.pf16Table = &(pf16Table1D[4]);

// input vector = -0.5
f16In = FRAC16(-0.5);

// output should be 0x6666 ~ FRAC16(0.8)
f16Out = GFLIB_Lut1D_F16(f16In,&trf16MyLut1D);

// output should be 0x6666 ~ FRAC16(0.8)
f16Out = GFLIB_Lut1D(f16In,&trf16MyLut1D,F16);

// available only if 16-bit fractional implementation
// selected as default
// output should be 0x6666 ~ FRAC16(0.8)
f16Out = GFLIB_Lut1D(f16In,&trf16MyLut1D);

// ######
// Pointer is located at the beginning of the interpolating data table.
// #####
// setting parameters for Lut1D function
trf16MyLut1D.u16ShamOffset = (tU16)3;
trf16MyLut1D.pf16Table = &(pf16Table1D[0]);

// input vector = 0
f16In = FRAC16(0);

// output should be 0x6666 ~ FRAC16(0.8)
f16Out = GFLIB_Lut1D_F16(f16In,&trf16MyLut1D);

// output should be 0x6666 ~ FRAC16(0.8)
f16Out = GFLIB_Lut1D(f16In,&trf16MyLut1D,F16);

// available only if 16-bit fractional implementation
// selected as default
// output should be 0x6666 ~ FRAC16(0.8)
f16Out = GFLIB_Lut1D(f16In,&trf16MyLut1D);

// ######
// Pointer is located at the end of the interpolating data table.
// #####
// setting parameters for Lut1D function
trf16MyLut1D.u16ShamOffset = (tU16)3;
trf16MyLut1D.pf16Table = &(pf16Table1D[8]);

// input vector = -1
f16In = FRAC16(-1);

// output should be 0x6666 ~ FRAC16(0.8)
f16Out = GFLIB_Lut1D_F16(f16In,&trf16MyLut1D);

// output should be 0x6666 ~ FRAC16(0.8)
f16Out = GFLIB_Lut1D(f16In,&trf16MyLut1D,F16);

// available only if 16-bit fractional implementation
// selected as default
```

```
// output should be 0x6666 ~ FRAC16(0.8)
f16Out = GFLIB_Lut1D(f16In, &trf16MyLut1D);
}
```

### 2.29.3 Function GFLIB\_Lut1D\_FLT

#### Declaration

```
tFloat GFLIB_Lut1D_FLT(tFloat fltIn, const GFLIB_LUT1D_T_FLT
*const pParam);
```

#### Arguments

**Table 204. GFLIB\_Lut1D\_FLT arguments**

Type	Name	Direction	Description
tFloat	fltIn	input	The abscissa for which 1D interpolation is performed.
const GFLIB_LUT1D_T_FLT *const	pParam	input	Pointer to the parameters structure with parameters of the look-up table function.

#### Return

The interpolated value from the look-up table.

#### Implementation details

The interpolating intervals are defined in the table provided by the `pfltTable` member of the parameters structure. The table contains ordinate values consecutively over the entire interpolating range. The abscissa values are assumed to be defined implicitly by a single interpolating interval length. The abscissa value is equal to the multiplication of the interpolating index and the interpolating interval length. For example, a table contains 4 interpolating points. The interpolating interval length in this example is equal to  $2^{-1}=0.5$ .

It should be noted that the `pfltTable` pointer does not have to point to the start of the memory area of ordinate values. Therefore, the interpolating index can be positive or negative or, even, does not have to have zero in its range.

A special algorithm is used to make the computation efficient, however, under some additional assumptions, as provided below:

- $x$  is in range  $\langle a, b \rangle$ , where  $|a| + |b| = 1$
- the values of the interpolated function are in the single precision floating point format
- the length of each interpolating interval is equal to  $2^{-uShamOffset}$ , where `uShamOffset` is an integer in the range of 1, 2, ... 29
- the abscissa provided for interpolation is in the single precision floating point format

The algorithm performs the following steps:

1. Count the number of equidistant interpolating intervals `fSegments`:

$$fSegments = 2^{uShamOffset}$$

Equation GFLIB\_Lut1D\_FLT\_Eq1

where the `uShamOffset` is the shift amount provided in the parameters structure as the member `pParam->u32ShamOffset`.

2. Compute the interpolating interval, in which the linear interpolation will be performed:

$$sIntvl = \lfloor fSegments \cdot fIn \rfloor$$

Equation GFLIB\_Lut1D\_FLT\_Eq2

where `sIntvl` is the position of the first ordinate value at the beginning of the interpolating interval, the `fSegments` is the equidistant interpolating interval and the `fIn` is the input value provided to the function as an argument.

3. Compute the interpolated value by the linear interpolation between the ordinates read from the table at the start and the end of the computed interval. The computed abscissa offset is used for the linear interpolation.

$$y_1 = pTable[sIntvl]$$

$$y_2 = pTable[sIntvl + 1]$$

$$\Delta x = (fIn \cdot fSegments) - sIntvl$$

$$y = y_1 + (y_2 - y_1) \cdot \Delta x$$

Equation GFLIB\_Lut1D\_FLT\_Eq3

where `y`, `y1` and `y2` are, respectively, the interpolated value, the ordinate at the start of the interpolating interval, the ordinate at the end of the interpolating interval. The `pTable` is the table containing ordinate values and it is provided as a pointer in the parameters structure `pParam->pfltTable`. The `fIn` is the input value provided to the function.

It should be noted that the input value can be positive or negative. If it is positive then the ordinate values are read as in the ordinary data array, that is, at or after the data pointer provided in the parameters structure `pParam->pfltTable`. However, if it is negative, then the ordinate values are read from the memory with lower address than the `pParam->pfltTable` pointer.

**Note:** The function may raise floating-point exceptions (invalid operation, overflow, underflow, inexact, input denormal). The floating-point unit must be enabled in CPACR register to prevent NOCP UsageFault exception.

The function performs a linear interpolation.

**Caution:** The function does not check whether the input abscissa value is within the range allowed by the interpolating data table `pParam->pfltTable`. If the computed interval index points to data outside the provided data table, then the interpolation will be computed with invalid data. The range of the input abscissa value depends on the position of the pointer in the interpolating data table. Sum of the absolute values of the lower and upper border values is equal to the 1. For a better understanding, please, see the extended code example.

### Code Example

```
#include "gplib.h"

tFloat fltIn;
tFloat fltOut;
GFLIB_LUT1D_T_FLT trfltMyLut1D = GFLIB_LUT1D_DEFAULT_FLT;
tFloat pfltTable1D[9] = {0.8, 0.1, -0.2, 0.7, 0.2, -0.3, -0.8, 0.91, 0.99};

void main(void)
```

```
{  
    // #####  
    // Pointer is located near the middle of the interpolating data table.  
    // #####  
    // setting parameters for Lut1D function  
    trfltMyLut1D.pfltTable = &(pfltTable1D[5]);  
    trfltMyLut1D.u32ShamOffset = (tU32)3;  
  
    // input vector = -0.5  
    fltIn = (tFloat)-0.5;  
  
    // output should be 0.1  
    fltOut = GFLIB_Lut1D_FLT(fltIn, &trfltMyLut1D);  
  
    // output should be 0.1  
    fltOut = GFLIB_Lut1D(fltIn, &trfltMyLut1D, FLT);  
  
    // available only if single precision floating point implementation  
    // selected as default  
    // output should be 0.1  
    fltOut = GFLIB_Lut1D(fltIn, &trfltMyLut1D);  
  
    // #####  
    // Pointer is located at the beginning of the interpolating data table.  
    // #####  
    // setting parameters for Lut1D function  
    trfltMyLut1D.pfltTable = &(pfltTable1D[0]);  
    trfltMyLut1D.u32ShamOffset = (tU32)3;  
  
    // input vector = 0  
    fltIn = (tFloat)0;  
  
    // output should be 0.8  
    fltOut = GFLIB_Lut1D_FLT(fltIn, &trfltMyLut1D);  
  
    // output should be 0.8  
    fltOut = GFLIB_Lut1D(fltIn, &trfltMyLut1D, FLT);  
  
    // available only if single precision floating point implementation  
    // selected as default  
    // output should be 0.8  
    fltOut = GFLIB_Lut1D(fltIn, &trfltMyLut1D);  
  
    // #####  
    // Pointer is located at the end of the interpolating data table.  
    // #####  
    // setting parameters for Lut1D function  
    trfltMyLut1D.pfltTable = &(pfltTable1D[8]);  
    trfltMyLut1D.u32ShamOffset = (tU32)3;  
  
    // input vector = -1  
    fltIn = (tFloat)-1;  
  
    // output should be 0.8  
    fltOut = GFLIB_Lut1D_FLT(fltIn, &trfltMyLut1D);  
  
    // output should be 0.8  
    fltOut = GFLIB_Lut1D(fltIn, &trfltMyLut1D, FLT);  
  
    // available only if single precision floating point implementation
```

```
// selected as default
// output should be 0.8
fltOut = GFLIB_Lut1D(fltIn,&trfLutMyLut1D);
}
```

## 2.30 Function GFLIB\_Lut2D

This function implements the two-dimensional look-up table.

### Description

The GFLIB\_Lut2D function performs two dimensional linear interpolation over a 2D table of data.

The following interpolation formulas are used:

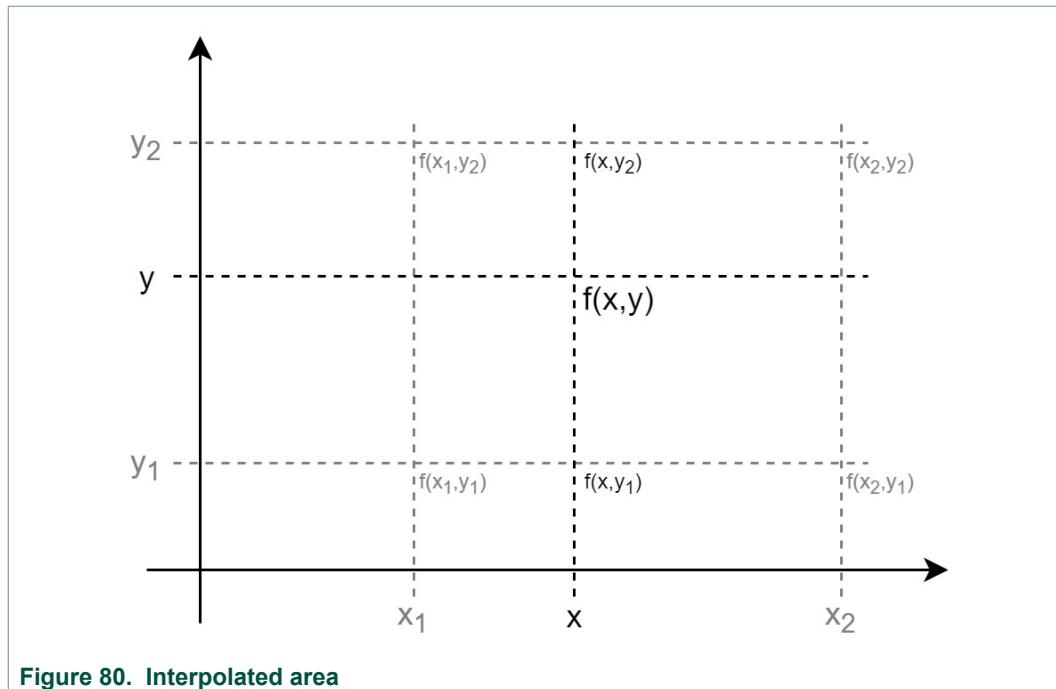
$$\begin{aligned} f(x, y_1) &= f(x_1, y_1) + \frac{f(x_2, y_1) - f(x_1, y_1)}{x_2 - x_1} \cdot (x - x_1) \\ f(x, y_2) &= f(x_1, y_2) + \frac{f(x_2, y_2) - f(x_1, y_2)}{x_2 - x_1} \cdot (x - x_1) \\ f(x, y) &= f(x, y_1) + \frac{f(x, y_2) - f(x, y_1)}{y_2 - y_1} \cdot (y - y_1) \end{aligned}$$

Equation GFLIB\_Lut2D\_Eq1

where:

- the x, y are the input values provided to the function in the In1 and In2 arguments
- $x_1, x_2, y_1$  and  $y_2$  are values on the edge of the interpolated area
- $f(x, y_1)$  and  $f(x, y_2)$  are the intermediate interpolated values at the beginning and the end of the final interpolating interval
- $f(x, y)$  is the interpolated value

The graphical representation of the interpolated area is shown in the following figure.



### Re-entrancy

The function is re-entrant.

#### 2.30.1 Function `GFLIB_Lut2D_F32`

##### Declaration

```
tFrac32 GFLIB_Lut2D_F32(tFrac32 f32In1, tFrac32 f32In2, const
GFLIB_LUT2D_T_F32 *const pParam);
```

##### Arguments

**Table 205. GFLIB\_Lut2D\_F32 arguments**

Type	Name	Direction	Description
<code>tFrac32</code>	f32In1	input	First input variable for which 2D interpolation is performed.
<code>tFrac32</code>	f32In2	input	Second input variable for which 2D interpolation is performed.
<code>const GFLIB_LUT2D_T_F32 *const</code>	pParam	input	Pointer to the parameters structure with specification of the two dimensional look-up table function.

##### Return

The interpolated value from the look-up table with 16-bit accuracy.

##### Implementation details

The interpolating intervals are defined in the table provided by the `pf32Table` member of the parameters structure. The table must be stored in column-major format and the number of elements `N` must comply with followin equation:

$$N = (2^{u16ShamOffset1} + 1) * (2^{u16ShamOffset2} + 1)$$

Equation GFLIB\_Lut2D\_F32\_Eq1

where `u16ShamOffset1` and `u16ShamOffset2` are member of the parameters structure. It should be noticed that the `pf32Table` pointer does not have to point to the start of the memory area with the table values. For example let's consider the following interpolating table with `u16ShamOffset1 = 2` and `u16ShamOffset2 = 1`:

**Table 206. GFLIB\_Lut2D example table**

y\x	0	1	2	3	4
0	0	0.1	0.2	0.3	0.4
1	0.01	0.11	0.21	0.31	0.41
2	0.02	0.12	0.22	0.32	0.42

Let the number 0.31 be the origin of the interpolating table. The coordinates of this value will be 3 in x-axis and 1 in y-axis. According to [GFLIB\\_Lut2D\\_F32\\_Eq1](#) the `pf32Table` shall point to 10. element of the array containing table data. The range of the inputs `f32In1` and `f32In2` depends on the position of the pointer in the interpolating data table. The range can be determined by following equations:

$$\begin{aligned} f32In1_{min} &= \frac{-OriginIdxX}{2^{u16ShamOffset1}} \\ f32In1_{max} &= f32In1_{min} + 1 \\ f32In2_{min} &= \frac{-OriginIdxY}{2^{u16ShamOffset2}} \\ f32In2_{max} &= f32In2_{min} + 1 \end{aligned}$$

Equation GFLIB\_Lut2D\_F32\_Eq2

where `OriginIdxX` and `OriginIdxY` are x and y coordinates of the origin of the interpolating table respectively. In the example above the `f32In1` must be in range (-0.75; 0.25) and `f32In2` in range (-0.5; 0.5).

**Note:** The input pointer must contain a valid address and the range of the input values must fall within the range of the interpolating data table otherwise a fault may occur (`MemManage`, `BusFault`, `UsageFault`, `HardFault`).

**Caution:** The function does not check whether the input values are within a range allowed by the interpolating data table `pParam->pf32Table`. If the computed interval index points to data outside the provided data table then the interpolation will be computed with invalid data.

### Code Example

```
#include "gflib.h"

tFrac32 f32In1;
tFrac32 f32In2;
tFrac32 f32Out;
GFLIB_LUT2D_T_F32 tr32tMyLut2D = GFLIB_LUT2D_DEFAULT_F32;
tFrac32 pf32Table2D[15] = {
    FRAC32(0), FRAC32(0.01), FRAC32(0.02),
    FRAC32(0.1), FRAC32(0.11), FRAC32(0.12),
    FRAC32(0.2), FRAC32(0.21), FRAC32(0.22),
    FRAC32(0.3), FRAC32(0.31), FRAC32(0.32),
}
```

```

FRAC32(0.4), FRAC32(0.41), FRAC32(0.42)};

void main(void)
{
    // setting parameters for Lut2D function
    tr32tMyLut2D.u32ShamOffset1 = (tU32)2;
    tr32tMyLut2D.u32ShamOffset2 = (tU32)1;
    tr32tMyLut2D.pf32Table = &(pf32Table2D[10]);

    // input vector
    f32In1 = FRAC32(-0.5);
    f32In2 = FRAC32(-0.5);

    // output should be 0xFFFFFFFF ~ FRAC32(0.1)
    f32Out = GFLIB_Lut2D_F32(f32In1, f32In2, &tr32tMyLut2D);

    // output should be 0xFFFFFFFF ~ FRAC32(0.1)
    f32Out = GFLIB_Lut2D(f32In1, f32In2, &tr32tMyLut2D, F32);

    // available only if 32-bit fractional implementation
    // selected as default
    // output should be 0xFFFFFFFF ~ FRAC32(0.1)
    f32Out = GFLIB_Lut2D(f32In1, f32In2, &tr32tMyLut2D);
}

```

## 2.30.2 Function GFLIB\_Lut2D\_F16

### Declaration

```
tFrac16 GFLIB_Lut2D_F16(tFrac16 f16In1, tFrac16 f16In2, const
GFLIB\_LUT2D\_T\_F16 *const pParam);
```

### Arguments

Table 207. GFLIB\_Lut2D\_F16 arguments

Type	Name	Direction	Description
<a href="#">tFrac16</a>	f16In1	input	First input variable for which 2D interpolation is performed.
<a href="#">tFrac16</a>	f16In2	input	Second input variable for which 2D interpolation is performed.
const <a href="#">GFLIB_LUT2D_T_F16</a> *const	pParam	input	Pointer to the parameters structure with parameters of the two dimensional look-up table function.

### Return

The interpolated value from the look-up table.

### Implementation details

The interpolating intervals are defined in the table provided by the pf16Table member of the parameters structure. The table must be stored in column-major format and the number of elements N must comply with followin equation:

$$N = (2^{u16ShamOffset1} + 1) * (2^{u16ShamOffset2} + 1)$$

Equation GFLIB\_Lut2D\_F16\_Eq1

where `u16ShamOffset1` and `u16ShamOffset2` are member of the parameters structure. It should be noticed that the `pf16Table` pointer does not have to point to the start of the memory area with the table values. For example let's consider the following interpolating table with `u16ShamOffset1 = 2` and `u16ShamOffset2 = 1`:

**Table 208. GFLIB\_Lut2D example table**

y\x	0	1	2	3	4
0	0	0.1	0.2	0.3	0.4
1	0.01	0.11	0.21	0.31	0.41
2	0.02	0.12	0.22	0.32	0.42

Let the number 0.31 be the origin of the interpolating table. The coordinates of this value will be 3 in x-axis and 1 in y-axis. According to [GFLIB\\_Lut2D\\_F16\\_Eq1](#) the `pf16Table` shall point to 10. element of the array containing table data. The range of the inputs `f16In1` and `f16In2` depends on the position of the pointer in the interpolating data table. The range can be determined by following equations:

$$\begin{aligned} f16In1_{min} &= \frac{-OriginIdxX}{2^{u16ShamOffset1}} \\ f16In1_{max} &= f16In1_{min} + 1 \\ f16In2_{min} &= \frac{-OriginIdxY}{2^{u16ShamOffset2}} \\ f16In2_{max} &= f16In2_{min} + 1 \end{aligned}$$

Equation GFLIB\_Lut2D\_F16\_Eq2

where `OriginIdxX` and `OriginIdxY` are x and y coordinates of the origin of the interpolating table respectively. In the example above the `f16In1` must be in range (-0.75; 0.25) and `f16In2` in range (-0.5; 0.5).

**Note:** The input pointer must contain a valid address and the range of the input values must fall within the range of the interpolating data table otherwise a fault may occur (`MemManage`, `BusFault`, `UsageFault`, `HardFault`).

**Caution:** The function does not check whether the input values are within a range allowed by the interpolating data table `pParam->pf16Table`. If the computed interval index points to data outside the provided data table then the interpolation will be computed with invalid data.

## Code Example

```
#include "gflib.h"

tFrac16 f16In1;
tFrac16 f16In2;
tFrac16 f16Out;
GFLIB_LUT2D_T_F16 tr16tMyLut2D = GFLIB_LUT2D_DEFAULT_F16;
tFrac16 pf16Table2D[15] = {
    FRAC16(0), FRAC16(0.01), FRAC16(0.02),
    FRAC16(0.1), FRAC16(0.11), FRAC16(0.12),
    FRAC16(0.2), FRAC16(0.21), FRAC16(0.22),
    FRAC16(0.3), FRAC16(0.31), FRAC16(0.32),
    FRAC16(0.4), FRAC16(0.41), FRAC16(0.42)};

void main(void)
{
```

```

// setting parameters for Lut2D function
tr16tMyLut2D.ul6ShamOffset1 = (tU16)2;
tr16tMyLut2D.ul6ShamOffset2 = (tU16)1;
tr16tMyLut2D.pf16Table = &(pf16Table2D[10]);

// input vector
f16In1 = FRAC16(-0.5);
f16In2 = FRAC16(-0.5);

// output should be 0xCCCC ~ FRAC16(0.1)
f16Out = GFLIB_Lut2D_F16(f16In1,f16In2,&tr16tMyLut2D);

// output should be 0xCCCC ~ FRAC16(0.1)
f16Out = GFLIB_Lut2D(f16In1,f16In2,&tr16tMyLut2D,F16);

// available only if 16-bit fractional implementation
// selected as default
// output should be 0xCCCC ~ FRAC16(0.1)
f16Out = GFLIB_Lut2D(f16In1,f16In2,&tr16tMyLut2D);
}

```

### 2.30.3 Function GFLIB\_Lut2D\_FLT

#### Declaration

```
tFloat GFLIB_Lut2D_FLT(tFloat fltIn1, tFloat fltIn2, const
GFLIB_LUT2D_T_FLT *const pParam);
```

#### Arguments

Table 209. GFLIB\_Lut2D\_FLT arguments

Type	Name	Direction	Description
<u>tFloat</u>	fltIn1	input	First input variable for which 2D interpolation is performed. Input value is in single precision floating data format.
<u>tFloat</u>	fltIn2	input	Second input variable for which 2D interpolation is performed. Input value is in single precision floating data format.
const <u>GFLIB_LUT2D_T_FLT</u> *const	pParam	input	Pointer to the parameters structure with parameters of the two dimensional look-up table function.

#### Return

The function returns the interpolated value. The output value is in single precision floating point format.

#### Implementation details

The interpolating intervals are defined in the table provided by the pf16Table member of the parameters structure. The table must be stored in column-major format and the number of elements  $N$  must comply with followin equation:

$$N = (2^{ul6ShamOffset1} + 1) * (2^{ul6ShamOffset2} + 1)$$

Equation GFLIB\_Lut2D\_FLT\_Eq1

where `u16ShamOffset1` and `u16ShamOffset2` are member of the parameters structure. It should be noticed that the `pfltTable` pointer does not have to point to the start of the memory area with the table values. For example let's consider the following interpolating table with `u16ShamOffset1 = 2` and `u16ShamOffset2 = 1`:

**Table 210. GFLIB\_Lut2D example table**

y\x	0	1	2	3	4
0	0	0.1	0.2	0.3	0.4
1	0.01	0.11	0.21	0.31	0.41
2	0.02	0.12	0.22	0.32	0.42

Let the number 0.31 be the origin of the interpolating table. The coordinates of this value will be 3 in x-axis and 1 in y-axis. According to [GFLIB\\_Lut2D\\_FLT\\_Eq1](#) the `pfltTable` shall point to 10. element of the array containing table data. The range of the inputs `fltIn1` and `fltIn2` depends on the position of the pointer in the interpolating data table. The range can be determined by following equations:

$$\begin{aligned} fltIn1_{min} &= \frac{-OriginIdxX}{2^{u16ShamOffset1}} \\ fltIn1_{max} &= fltIn1_{min} + 1 \\ fltIn2_{min} &= \frac{-OriginIdxY}{2^{u16ShamOffset2}} \\ fltIn2_{max} &= fltIn2_{min} + 1 \end{aligned}$$

Equation GFLIB\_Lut2D\_FLT\_Eq2

where `OriginIdxX` and `OriginIdxY` are x and y coordinates of the origin of the interpolating table respectively. In the example above the `fltIn1` must be in range (-0.75; 0.25) and `fltIn2` in range (-0.5; 0.5).

**Note:** The function may raise floating-point exceptions (invalid operation, overflow, underflow, inexact, input denormal). The floating-point unit must be enabled in CPACR register to prevent NOCP UsageFault exception.

The input pointer must contain a valid address and the range of the input values must fall within the range of the interpolating data table otherwise a fault may occur (MemManage, BusFault, UsageFault, HardFault).

**Caution:** The function does not check whether the input values are within the range allowed by the interpolating data table `pParam->pfltTable`. If the computed interval index points to data outside the provided data table then the interpolation will be computed with invalid data.

### Code Example

```
#include "gflib.h"

tFloat fltIn1;
tFloat fltIn2;
tFloat fltOut;
GFLIB_LUT2D_T_FLT trfsltMyLut2D = GFLIB_LUT2D_DEFAULT_FLT;
tFloat pfltTable2D[15] = {
    0, 0.01, 0.02,
    0.1, 0.11, 0.12,
    0.2, 0.21, 0.22,
```

```
0.3, 0.31, 0.32,  
0.4, 0.41, 0.42};  
  
void main(void)  
{  
    // setting parameters for Lut2D function  
    trfMyLut2D.u32ShamOffset1 = (tU32)2;  
    trfMyLut2D.u32ShamOffset2 = (tU32)1;  
    trfMyLut2D.pflTable = &(pflTable2D[10]);  
  
    // input vector  
    fltIn1 = (tFloat)-0.5;  
    fltIn2 = (tFloat)-0.5;  
  
    // output should be 0.1  
    fltOut = GFLIB_Lut2D_FLT(fltIn1, fltIn2, &trfMyLut2D);  
  
    // output should be 0.1  
    fltOut = GFLIB_Lut2D(fltIn1, fltIn2, &trfMyLut2D, FLT);  
  
    // available only if single precision floating point implementation  
    // selected as default  
    // output should be 0.1  
    fltOut = GFLIB_Lut2D(fltIn1, fltIn2, &trfMyLut2D);  
}
```

## 2.31 Function GFLIB\_Ramp

The function calculates the up/down ramp with the step increment/decrement defined in the pParam structure.

### Description

The GFLIB\_Ramp function limits the rate of change of the input signal.

If the absolute value of the desired (input) value is greater than the absolute value of the ramp state, the function adds the RampUp coefficient to the actual output value. The absolute value of the output cannot be greater than the absolute value of the desired value.

If the absolute value of the desired value is lower than the absolute value of the actual ramp state, the function subtracts the RampDown coefficient from the actual output value. The absolute value of the output cannot be lower than the absolute value of the desired value.

Functionality of the implemented ramp algorithm can be explained with use of [Figure 81](#)

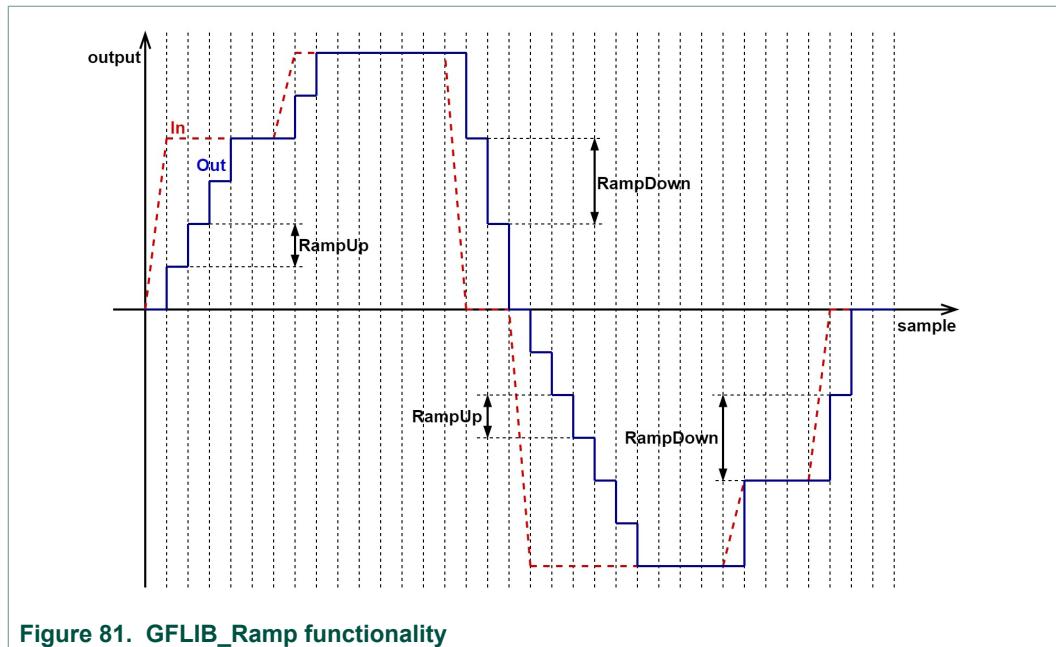


Figure 81. GFLIB\_Ramp functionality

The upper and lower limits can be found in the limits structure, supplied to the function as a pointer `pParam`.

**Note:** The input pointer must contain a valid address otherwise a fault may occur (MemManage, BusFault, UsageFault, HardFault).

### Re-entrancy

The function is re-entrant.

#### 2.31.1 Function GFLIB\_Ramp\_F32

##### Declaration

```
tFrac32 GFLIB_Ramp_F32 (tFrac32 f32In, GFLIB_RAMP_T_F32 *const pParam);
```

##### Arguments

Table 211. GFLIB\_Ramp\_F32 arguments

Type	Name	Direction	Description
<code>tFrac32</code>	<code>f32In</code>	input	Input argument representing the desired output value.
<code>GFLIB_RAMP_T_F32</code> <code>*const</code>	<code>pParam</code>	input, output	Pointer to the ramp parameters structure.

##### Return

The function returns a 32-bit value in format Q1.31, which represents the actual ramp output value. This, in time, is approaching the desired (input) value by step increments defined in the `pParam` structure.

**Note:** All parameters and states used by the function can be reset during declaration using the `GFLIB_RAMP_DEFAULT_F32` macro.

**Code Example**

```
#include "gplib.h"

tFrac32 f32In;
tFrac32 f32Out;
GFLIB_RAMP_T_F32 f32trMyRamp = GFLIB_RAMP_DEFAULT_F32;

void main(void)
{
    // increment/decrement coefficients
    f32trMyRamp.f32RampUp = FRAC32(0.1);
    f32trMyRamp.f32RampDown = FRAC32(0.03333333);

    // input value = 0.5
    f32In = FRAC32(0.5);

    // output should be 0x0CCCCCCC ~ FRAC32(0.1)
    f32Out = GFLIB_Ramp_F32(f32In, &f32trMyRamp);

    // clearing of the internal states
    f32trMyRamp.f32State = (tFrac32)0;
    // output should be 0x0CCCCCCC ~ FRAC32(0.1)
    f32Out = GFLIB_Ramp(f32In, &f32trMyRamp, F32);

    // #####
    // Available only if 32-bit fractional implementation selected
    // as default
    // #####
    // clearing of the internal states
    f32trMyRamp.f32State = (tFrac32)0;
    // output should be 0x0CCCCCCC ~ FRAC32(0.1)
    f32Out = GFLIB_Ramp(f32In, &f32trMyRamp);
}
```

**2.31.2 Function GFLIB\_Ramp\_F16****Declaration**

```
tFrac16 GFLIB_Ramp_F16(tFrac16 f16In, GFLIB_RAMP_T_F16 *const pParam);
```

**Arguments****Table 212. GFLIB\_Ramp\_F16 arguments**

Type	Name	Direction	Description
tFrac16	f16In	input	Input argument representing the desired output value.
GFLIB_RAMP_T_F16 *const	pParam	input, output	Pointer to the ramp parameters structure.

**Return**

The function returns a 16-bit value in format Q1.15, which represents the actual ramp output value. This, in time, is approaching the desired (input) value by step increments defined in the pParam structure.

**Note:** All parameters and states used by the function can be reset during declaration using the [GFLIB\\_RAMP\\_DEFAULT\\_F16](#) macro.

### Code Example

```
#include "gplib.h"

tFrac16 f16In;
tFrac16 f16Out;
GFLIB_RAMP_T_F16 f16trMyRamp = GFLIB_RAMP_DEFAULT_F16;

void main(void)
{
    // increment/decrement coefficients
    f16trMyRamp.f16RampUp = FRAC16(0.1);
    f16trMyRamp.f16RampDown = FRAC16(0.03333333);

    // input value = 0.5
    f16In = FRAC16(0.5);

    // output should be 0x0CCC ~ FRAC16(0.1)
    f16Out = GFLIB_Ramp_F16(f16In, &f16trMyRamp);

    // clearing of the internal states
    f16trMyRamp.f16State = (tFrac16)0;
    // output should be 0x0CCC ~ FRAC16(0.1)
    f16Out = GFLIB_Ramp(f16In, &f16trMyRamp, F16);

    // ######
    // Available only if 16-bit fractional implementation selected
    // as default
    // #####
    // clearing of the internal states
    f16trMyRamp.f16State = (tFrac16)0;
    // output should be 0x0CCC ~ FRAC16(0.1)
    f16Out = GFLIB_Ramp(f16In, &f16trMyRamp);
}
```

### 2.31.3 Function GFLIB\_Ramp\_FLT

#### Declaration

```
tFloat GFLIB_Ramp_FLT(tFloat fltIn, GFLIB_RAMP_T_FLT *const pParam);
```

#### Arguments

Table 213. GFLIB\_Ramp\_FLT arguments

Type	Name	Direction	Description
tFloat	fltIn	input	Input argument representing the desired output value. Input value is in single precision floating data format.
GFLIB_RAMP_T_FLT *const	pParam	input, output	Pointer to the ramp parameters structure. Arguments of the structure contain single precision floating point values.

### Return

The function returns a value in single precision floating point format, which represents the actual ramp output. This value can be also described as a Ramp state value increased/decreased by a slope in order to achieve the desired input value.

**Note:** The function may raise floating-point exceptions (invalid operation, overflow, underflow, inexact, input denormal). The floating-point unit must be enabled in CPACR register to prevent NOCP UsageFault exception.

All parameters and states used by the function can be reset during declaration using the [GFLIB\\_RAMP\\_DEFAULT\\_FLT](#) macro.

### Code Example

```
#include "gflib.h"

tFloat fltIn;
tFloat fltOut;
GFLIB_RAMP_T_FLT flttrMyRamp = GFLIB_RAMP_DEFAULT_FLT;

void main(void)
{
    // increment/decrement coefficients
    flttrMyRamp.fltRampUp = (tFloat)(0.1);
    flttrMyRamp.fltRampDown = (tFloat)(0.03333333);

    // input value = 0.5
    fltIn = (tFloat)(0.5);

    // output should be 0.1
    fltOut = GFLIB_Ramp_FLT(fltIn, &flttrMyRamp);

    // clearing of the internal states
    flttrMyRamp.fltState = (tFloat)0;
    // output should be 0.1
    fltOut = GFLIB_Ramp(fltIn, &flttrMyRamp, FLT);

    // ##### Available only if single precision floating point
    // implementation selected as default
    // #####
    // clearing of the internal states
    flttrMyRamp.fltState = (tFloat)0;
    // output should be 0.1
    fltOut = GFLIB_Ramp(fltIn, &flttrMyRamp);
}
```

## 2.32 Function GFLIB\_Sign

This function returns the signum of input value.

## Description

The GFLIB\_Sign function calculates the sign of the input argument according to the following equation:

$$y_{out} = \begin{cases} 1 & \text{if } x_{in} > 0 \\ 0 & \text{if } x_{in} = 0 \\ -1 & \text{if } x_{in} < 0 \end{cases}$$

Equation GFLIB\_Sign\_Eq1

where:

- $y_{out}$  is the return value
- $x_{in}$  is the input value provided as the In parameter

## Re-entrancy

The function is re-entrant.

### 2.32.1 Function GFLIB\_Sign\_F32

#### Declaration

```
tFrac32 GFLIB_Sign_F32(tFrac32 f32In);
```

#### Arguments

Table 214. GFLIB\_Sign\_F32 arguments

Type	Name	Direction	Description
tFrac32	f32In	input	Input argument.

#### Return

The function returns the sign of the input argument.

#### Implementation details

If the input value is negative, then the return value will be set to "-1" (0x80000000 hex), if the input value is zero, then the function returns "0" (0x0 hex), otherwise if the input value is greater than zero, the return value will be "1" (0x7fffffff hex).

**Note:** The input and the output values are in the 32-bit fixed point fractional data format.

#### Code Example

```
#include "gflib.h"

tFrac32 f32In;
tFrac32 f32Out;

void main(void)
{
    // input value = 0.5
    f32In = FRAC32(0.5);
```

```

// output should be 0x7FFFFFFF ~ FRAC32(1 - (2^-31))
f32Out = GFLIB_Sign_F32(f32In);

// output should be 0x7FFFFFFF ~ FRAC32(1 - (2^-31))
f32Out = GFLIB_Sign(f32In, F32);

// ######
// Available only if 32-bit fractional implementation selected
// as default
// #####
// output should be 0x7FFFFFFF ~ FRAC32(1 - (2^-31))
f32Out = GFLIB_Sign(f32In);
}

```

### 2.32.2 Function GFLIB\_Sign\_F16

#### Declaration

`tFrac16 GFLIB_Sign_F16(tFrac16 f16In);`

#### Arguments

Table 215. GFLIB\_Sign\_F16 arguments

Type	Name	Direction	Description
<code>tFrac16</code>	<code>f16In</code>	<code>input</code>	Input argument.

#### Return

The function returns the sign of the input argument.

#### Implementation details

If the input value is negative, then the return value will be set to "-1" (0x8000 hex), if the input value is zero, then the function returns "0" (0x0 hex), otherwise if the input value is greater than zero, the return value will be "1" (0x7fff hex).

**Note:** The input and the output values are in the 16-bit fixed point fractional data format.

#### Code Example

```

#include "gplib.h"

tFrac16 f16In;
tFrac16 f16Out;

void main(void)
{
    // input value = 0.5
    f16In = FRAC16(0.5);

    // output should be 0x7FFF ~ FRAC16(1 - (2^-15))
    f16Out = GFLIB_Sign_F16(f16In);

    // output should be 0x7FFF ~ FRAC16(1 - (2^-15))
    f16Out = GFLIB_Sign(f16In, F16);
}

```

```

// ######
// Available only if 16-bit fractional implementation selected
// as default
// #####
// output should be 0x7FFF ~ FRAC16(1-(2^-15))
f16Out = GFLIB_Sign(f16In);
}

```

### 2.32.3 Function GFLIB\_Sign\_FLT

#### Declaration

`tFloat GFLIB_Sign_FLT(tFloat fltIn);`

#### Arguments

Table 216. GFLIB\_Sign\_FLT arguments

Type	Name	Direction	Description
<code>tFloat</code>	<code>fltIn</code>	<code>input</code>	Input argument.

#### Return

The function returns the sign of the input argument.

**Note:** The function may raise floating-point exceptions (invalid operation, input denormal). The floating-point unit must be enabled in CPACR register to prevent NOCP UsageFault exception.

The input and the output are in single precision floating point data format.

#### Code Example

```

#include "gflib.h"

tFloat fltIn;
tFloat fltOut;

void main(void)
{
    // input value = 0.5
    fltIn = (tFloat)(0.5);

    // output should be 1
    fltOut = GFLIB_Sign_FLT(fltIn);

    // output should be 1
    fltOut = GFLIB_Sign(fltIn,FLT);

    // #####
    // Available only if single precision floating point
    // implementation selected as default
    // #####
    // output should be 1

```

```

    fltOut = GFLIB_Sign(fltIn);
}

```

## 2.33 Function GFLIB\_Sin

This function implements an approximation of sine function.

### Description

The GFLIB\_Sin function provides a computational method for calculation of the trigonometric sine function  $\sin(x)$ , using the piece-wise polynomial approximation.

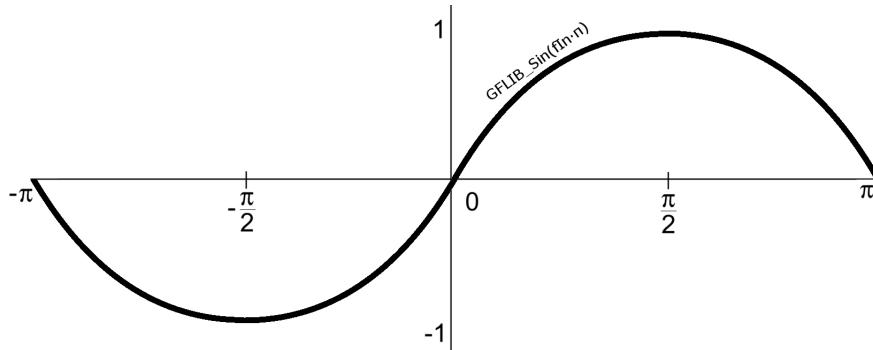


Figure 82. Course of the function GFLIB\_Sin

When both sine and cosine of the same argument is needed, use [GFLIB\\_SinCos](#) function instead for better performance.

### Re-entrancy

The function is re-entrant.

#### 2.33.1 Function GFLIB\_Sin\_F32

##### Declaration

```
tFrac32 GFLIB_Sin_F32(tFrac32 f32In, const GFLIB_SIN_T_F32 *const pParam);
```

##### Arguments

Table 217. GFLIB\_Sin\_F32 arguments

Type	Name	Direction	Description
<a href="#">tFrac32</a>	f32In	input	Input argument is a 32-bit number that contains an angle in radians from interval $[-\pi, \pi]$ normalized between $[-1, 1]$ .
const <a href="#">GFLIB_SIN_T_F32</a> *const	pParam	input	Pointer to an array of Taylor coefficients. In case the default approximation coefficients are used, the &pParam must be replaced with <a href="#">GFLIB_SIN_DEFAULT_F32</a> symbol.

##### Return

The function returns the value of the sine function of the input argument as a fixed point 32-bit number, normalized between  $[-1, 1]$ .

**Implementation details**

The input values are scaled from  $[-\pi, \pi]$  radians to  $[-1, 1]$  in order to fit in the available fixed-point fractional range. The function uses a 9th order Taylor polynomial approximation; the default polynomial coefficients are provided in the [GFLIB\\_SIN\\_DEFAULT\\_F32](#) structure.

**Note:** The input pointer must contain a valid address otherwise a fault may occur (*MemManage*, *BusFault*, *UsageFault*, *HardFault*).

**Code Example**

```
#include "gflib.h"

tFrac32 f32Angle;
tFrac32 f32Output;

void main(void)
{
    // input angle = 0.5 => pi/2
    f32Angle = FRAC32(0.5);

    // output should be 0x7FFFFFFF
    f32Output = GFLIB_Sin_F32(f32Angle, GFLIB_SIN_DEFAULT_F32);

    // output should be 0x7FFFFFFF
    f32Output = GFLIB_Sin(f32Angle, GFLIB_SIN_DEFAULT_F32, F32);

    // ##### Available only if 32-bit fractional implementation selected
    // as default
    // #####
    // output should be 0x7FFFFFFF
    f32Output = GFLIB_Sin(f32Angle);
}
```

**2.33.2 Function GFLIB\_Sin\_F16****Declaration**

```
tFrac16 GFLIB_Sin_F16(tFrac16 f16In, const GFLIB_SIN_T_F16 *const pParam);
```

**Arguments****Table 218. GFLIB\_Sin\_F16 arguments**

Type	Name	Direction	Description
<a href="#">tFrac16</a>	f16In	<a href="#">input</a>	Input argument is a 16-bit number that contains an angle in radians from interval $[-\pi, \pi]$ normalized between $[-1, 1]$ .
const <a href="#">GFLIB_SIN_T_F16</a> *const	pParam	<a href="#">input</a>	Pointer to an array of Taylor coefficients. In case the default approximation coefficients are used, the &pParam must be replaced with <a href="#">GFLIB_SIN_DEFAULT_F16</a> symbol.

## Return

The function returns the value of the sine function of the input argument as a fixed point 16-bit number, normalized between [-1, 1].

## Implementation details

The input values are scaled from  $[-\pi, \pi]$  radians to [-1, 1) in order to fit in the available fixed-point fractional range. The function uses a 7th order Taylor polynomial approximation; the default polynomial coefficients are provided in the [GFLIB\\_SIN\\_DEFAULT\\_F16](#) structure.

**Note:** The input pointer must contain a valid address otherwise a fault may occur (MemManage, BusFault, UsageFault, HardFault).

## Code Example

```
#include "gflib.h"

tFrac16 f16Angle;
tFrac16 f16Output;

void main(void)
{
    // input angle = 0.5 => pi/2
    f16Angle = FRAC16(0.5);

    // output should be 0x7FFF
    f16Output = GFLIB_Sin_F16(f16Angle, GFLIB_SIN_DEFAULT_F16);

    // output should be 0x7FFF
    f16Output = GFLIB_Sin(f16Angle, GFLIB_SIN_DEFAULT_F16, F16);

    // ##### Available only if 16-bit fractional implementation selected
    // as default
    // #####
    // output should be 0x7FFF
    f16Output = GFLIB_Sin(f16Angle);
}
```

### 2.33.3 Function GFLIB\_Sin\_FLT

#### Declaration

```
tFloat GFLIB_Sin_FLT(tFloat fltIn, const GFLIB_SIN_T_FLT *const pParam);
```

#### Arguments

**Table 219. GFLIB\_Sin\_FLT arguments**

Type	Name	Direction	Description
tFloat	fltIn	input	Input argument is a single precision floating point number that contains an angle in radians from interval $[-\pi, \pi]$ .

Type	Name	Direction	Description
const <a href="#">GFLIB_SIN_T</a> <a href="#">FLT</a> *const	pParam	input	Pointer to an array of approximation coefficients. In case the default approximation coefficients are used, the &pParam must be replaced with <a href="#">GFLIB_SIN_DEFAULT_FLT</a> symbol.

**Return**

The function returns the value of the sine function of the input argument as a single precision floating point number.

**Implementation details**

The function uses a 7th order minimax polynomial approximation; the default polynomial coefficients are provided in the [GFLIB\\_SIN\\_DEFAULT\\_FLT](#) structure.

**Note:** The input pointer must contain a valid address otherwise a fault may occur (MemManage, BusFault, UsageFault, HardFault).

*The function may raise floating-point exceptions (invalid operation, overflow, underflow, inexact, input denormal). The floating-point unit must be enabled in CPACR register to prevent NOCP UsageFault exception.*

**Code Example**

```
#include "gflib.h"

tFloat fltAngle;
tFloat fltOutput;

void main(void)
{
    // input angle = 1.5707963 => pi/2
    fltAngle = (tFloat)(1.5707963);

    // output should be 1
    fltOutput = GFLIB_Sin_FLT(fltAngle, GFLIB\_SIN\_DEFAULT\_FLT);

    // output should be 1
    fltOutput = GFLIB_Sin(fltAngle, GFLIB\_SIN\_DEFAULT\_FLT, FLT);

    // ##### Available only if single precision floating point
    // implementation selected as default
    // #####
    // output should be 1
    fltOutput = GFLIB_Sin(fltAngle);
}
```

**2.34 Function GFLIB\_SinCos**

This function implements polynomial approximation of the sine and cosine function.

## Description

The function GFLIB\_SinCos calculates the trigonometric sine and cosine function of the same argument using a polynomial approximation. GFLIB\_SinCos performs faster than the combination of [GFLIB\\_Sin](#) and [GFLIB\\_Cos](#).

## Re-entrancy

The function is re-entrant.

### 2.34.1 Function GFLIB\_SinCos\_F32

#### Declaration

```
void GFLIB_SinCos_F32(tFrac32 f32In, SWLIBS\_2Syst\_F32 *pOut,  
const GFLIB\_SINCOS\_T\_F32 *const pParam);
```

#### Arguments

**Table 220. GFLIB\_SinCos\_F32 arguments**

Type	Name	Direction	Description
<a href="#">tFrac32</a>	f32In	input	Input argument is a 32-bit number that contains an angle in radians from interval [- $\pi$ , $\pi$ ) normalized between [-1, 1).
<a href="#">SWLIBS_2Syst_F32</a> *	pOut	output	Pointer to the structure where the values of the sine and cosine of the input angle are stored. The function returns the sine and cosine of the input argument as a fixed point 32-bit number, normalized between [-1, 1). The <i>sine</i> of input angle is returned in first item of the structure and the <i>cosine</i> of input angle is returned in second item of the structure.
const <a href="#">GFLIB_SINCOS_T_F32</a> *const	pParam	input	Pointer to an array of Taylor coefficients. In case the default approximation coefficients are used, the &pParam must be replaced with <a href="#">GFLIB_SINCOS_DEFAULT_F32</a> symbol.

#### Return

void

#### Implementation details

The input values are scaled from [- $\pi$ ,  $\pi$ ) radians to [-1, 1) in order to fit in the available fixed-point fractional range. The function uses a 9th order Taylor polynomial approximation; the default polynomial coefficients are provided in the [GFLIB\\_SINCOS\\_DEFAULT\\_F32](#) structure.

**Note:** The input pointer must contain a valid address otherwise a fault may occur (*MemManage*, *BusFault*, *UsageFault*, *HardFault*).

#### Code Example

```
#include "gflib.h"

tFrac32 f32Angle;
SWLIBS\_2Syst\_F32 pf32Output;

void main(void)
```

```
{
    // input angle = 0.5 => pi/2
    f32Angle = FRAC32(0.5);

    // output should be:
    // pf32Output.f32Arg1 ~ sin(f32Angle) = 0x7FFF0000
    // pf32Output.f32Arg2 ~ cos(f32Angle) = 0x00000000
    GFLIB\_SinCos\_F32(f32Angle, &pf32Output, GFLIB\_SINCOS\_DEFAULT\_F32);

    // output should be:
    // pf32Output.f32Arg1 ~ sin(f32Angle) = 0x7FFF0000
    // pf32Output.f32Arg2 ~ cos(f32Angle) = 0x00000000
    GFLIB\_SinCos(f32Angle, &pf32Output, GFLIB\_SINCOS\_DEFAULT\_F32, F32);

    // ##### Available only if 32-bit fractional implementation selected
    // as default
    // #####
    // output should be:
    // pf32Output.f32Arg1 ~ sin(f32Angle) = 0x7FFF0000
    // pf32Output.f32Arg2 ~ cos(f32Angle) = 0x00000000
    GFLIB\_SinCos(f32Angle, &pf32Output);
}
```

## 2.34.2 Function [GFLIB\\_SinCos\\_F16](#)

### Declaration

```
void GFLIB\_SinCos\_F16(tFrac16 f16In, SWLIBS\_2Syst\_F16 *pOut,
const GFLIB\_SINCOS\_T\_F16 *const pParam);
```

### Arguments

**Table 221. GFLIB\_SinCos\_F16 arguments**

Type	Name	Direction	Description
<a href="#">tFrac16</a>	f16In	input	Input argument is a 16-bit number that contains an angle in radians from interval [-π, π) normalized between [-1, 1].
<a href="#">SWLIBS_2Syst_F16</a> *	pOut	output	Pointer to the structure where the values of the sine and cosine of the input angle are stored. The function returns the sine and cosine of the input argument as a fixed point 16-bit number, normalized between [-1, 1]. The <i>sine</i> of input angle is returned in first item of the structure and the <i>cosine</i> of input angle is returned in second item of the structure.
const <a href="#">GFLIB_SINCOS_T_F16</a> *const	pParam	input	Pointer to an array of Taylor coefficients. In case the default approximation coefficients are used, the &pParam must be replaced with <a href="#">GFLIB_SINCOS_DEFAULT_F16</a> symbol.

### Return

void

### Implementation details

The input values are scaled from [-π, π) radians to [-1, 1) in order to fit in the available fixed-point fractional range. The function uses a 7th order Taylor

polynomial approximation; the default polynomial coefficients are provided in the [GFLIB\\_SINCOS\\_DEFAULT\\_F16](#) structure.

**Note:** Due to effectivity reasons this function is implemented using inline assembly and is therefore not ANSI-C compliant.

The input pointer must contain a valid address otherwise a fault may occur (MemManage, BusFault, UsageFault, HardFault).

### Code Example

```
#include "gplib.h"

tFrac16 f16Angle;
SWLIBS_2Syst_F16 pf16Output;

void main(void)
{
    // input angle = 0.5 => pi/2
    f16Angle = FRAC16(0.5);

    // output should be:
    // pf16Output.f16Arg1 ~ sin(f16Angle) = 0x7FFF
    // pf16Output.f16Arg2 ~ cos(f16Angle) = 0x0000
    GFLIB_SinCos_F16(f16Angle, &pf16Output, GFLIB_SINCOS_DEFAULT_F16);

    // output should be:
    // pf16Output.f16Arg1 ~ sin(f16Angle) = 0x7FFF
    // pf16Output.f16Arg2 ~ cos(f16Angle) = 0x0000
    GFLIB_SinCos(f16Angle, &pf16Output, GFLIB_SINCOS_DEFAULT_F16, F16);

    // ######
    // Available only if 16-bit fractional implementation selected
    // as default
    // #####
    // output should be:
    // pf16Output.f16Arg1 ~ sin(f16Angle) = 0x7FFF
    // pf16Output.f16Arg2 ~ cos(f16Angle) = 0x0000
    GFLIB_SinCos(f16Angle, &pf16Output);
}
```

### 2.34.3 Function GFLIB\_SinCos\_FLT

#### Declaration

```
void GFLIB_SinCos_FLT(tFloat fltIn, SWLIBS_2Syst_FLT *pOut, const
GFLIB_SINCOS_T_FLT *const pParam);
```

#### Arguments

Table 222. GFLIB\_SinCos\_FLT arguments

Type	Name	Direction	Description
tFloat	fltIn	input	Input argument is a single precision floating point number that contains an angle in radians from interval [- π, π].

Type	Name	Direction	Description
<a href="#">SWLIBS_2Syst_FLT</a> *	pOut	output	Pointer to the structure where the values of the sine and cosine of the input angle are stored. The function returns the sine and cosine of the input argument as a single precision floating point number. The <i>sine</i> of input angle is returned in first item of the structure and the <i>cosine</i> of input angle is returned in second item of the structure.
const <a href="#">GFLIB_SINCOS_T_FLT</a> *const	pParam	input	Pointer to an array of approximation coefficients. In case the default approximation coefficients are used, the &pParam must be replaced with <a href="#">GFLIB_SINCOS_DEFAULT_FLT</a> symbol.

**Return**

void

**Implementation details**

The function uses a 7th order minimax polynomial approximation; the default polynomial coefficients are provided in the [GFLIB\\_SINCOS\\_DEFAULT\\_FLT](#) structure.

**Note:** The input pointer must contain a valid address otherwise a fault may occur (MemManage, BusFault, UsageFault, HardFault).

The function may raise floating-point exceptions (invalid operation, overflow, underflow, inexact, input denormal). The floating-point unit must be enabled in CPACR register to prevent NOCP UsageFault exception.

**Code Example**

```
#include "gplib.h"

tFloat fltAngle;
SWLIBS\_2Syst\_FLT pfltOutput;

void main(void)
{
    // input angle = 1.5707963 => pi/2
    fltAngle = (tFloat)(1.5707963);

    // output should be:
    // pfltOutput.fltArg1 ~ sin(fltAngle) = 1.0f
    // pfltOutput.fltArg2 ~ cos(fltAngle) = 0f
    GFLIB_SinCos_FLT(fltAngle, &pfltOutput, GFLIB\_SINCOS\_DEFAULT\_FLT);

    // output should be:
    // pfltOutput.fltArg1 ~ sin(fltAngle) = 1.0f
    // pfltOutput.fltArg2 ~ cos(fltAngle) = 0f
    GFLIB_SinCos(fltAngle, &pfltOutput, GFLIB\_SINCOS\_DEFAULT\_FLT, FLT);

    // ######
    // Available only if single precision floating point
    // implementation selected as default
    // #####
}

// output should be:
// pfltOutput.fltArg1 ~ sin(fltAngle) = 1.0f
```

```
// pfltOutput.fltArg2 ~ cos(fltAngle) = 0f
GFLIB_SinCos(fltAngle &pfltOutput);
}
```

## 2.35 Function GFLIB\_Sqrt

This function returns the square root of input value.

### Description

The GFLIB\_Sqrt function calculates the square root of the input value.

### Re-entrancy

The function is re-entrant.

#### 2.35.1 Function GFLIB\_Sqrt\_F32

##### Declaration

```
tFrac32 GFLIB_Sqrt_F32(tFrac32 f32In);
```

##### Arguments

Table 223. GFLIB\_Sqrt\_F32 arguments

Type	Name	Direction	Description
<a href="#">tFrac32</a>	f32In	input	The input value.

##### Return

The function returns the square root of the input value. The return value is within the [0, 1) fraction range.

##### Implementation details

The function uses a floating-point square root instruction with appropriate input/output conversions.

**Note:** The function may raise floating-point exceptions (invalid operation, inexact, input denormal). The floating-point unit must be enabled in CPACR register to prevent NOCP UsageFault exception.

The valid input range is [0, 1). Negative inputs will yield undefined results. Due to effectivity reason this function is written as inline assembly and thus is not ANSI-C compliant.

##### Code Example

```
#include "gflib.h"

tFrac32 f32In;
tFrac32 f32Out;

void main(void)
```

```
{
    // input value = 0.5
    f32In = FRAC32(0.5);

    // output should be 0x5A820000 ~ FRAC32(0.70710678)
    f32Out = GFLIB_Sqrt_F32(f32In);

    // output should be 0x5A820000 ~ FRAC32(0.70710678)
    f32Out = GFLIB_Sqrt(f32In, F32);

    // ##### Available only if 32-bit fractional implementation selected
    // as default
    // #####
    // output should be 0x5A820000 ~ FRAC32(0.70710678)
    f32Out = GFLIB_Sqrt(f32In);
}
```

## 2.35.2 Function GFLIB\_Sqrt\_F16

### Declaration

```
tFrac16 GFLIB_Sqrt_F16(tFrac16 f16In);
```

### Arguments

**Table 224. GFLIB\_Sqrt\_F16 arguments**

Type	Name	Direction	Description
tFrac16	f16In	input	The input value.

### Return

The function returns the square root of the input value. The return value is within the [0, 1) fraction range.

### Implementation details

The function uses a floating-point square root instruction with appropriate input/output conversions.

**Note:** The function may raise floating-point exceptions (invalid operation, inexact, input denormal). The floating-point unit must be enabled in CPACR register to prevent NOCP UsageFault exception.

The valid input range is [0, 1). Negative inputs will yield undefined results. Due to effectivity reason this function is written as inline assembly and thus is not ANSI-C compliant.

### Code Example

```
#include "gflib.h"

tFrac16 f16In;
tFrac16 f16Out;
```

```

void main(void)
{
    // input value = 0.5
    f16In = FRAC16(0.5);

    // output should be 0x5A82 ~ FRAC16(0.70710678)
    f16Out = GFLIB\_Sqrt\_F16(f16In);

    // output should be 0x5A82 ~ FRAC16(0.70710678)
    f16Out = GFLIB\_Sqrt(f16In, F16);

    // ##### Available only if 16-bit fractional implementation selected
    // as default
    // #####
    // output should be 0x5A82 ~ FRAC16(0.70710678)
    f16Out = GFLIB\_Sqrt(f16In);
}

```

### 2.35.3 Function [GFLIB\\_Sqrt\\_FLT](#)

#### Declaration

[tFloat](#) [GFLIB\\_Sqrt\\_FLT](#)([tFloat](#) fltIn);

#### Arguments

Table 225. [GFLIB\\_Sqrt\\_FLT](#) arguments

Type	Name	Direction	Description
<a href="#">tFloat</a>	fltIn	input	The input value.

#### Return

The function returns the square root of the input value. The return value is in single precision floating point format.

**Note:** The function may raise floating-point exceptions (invalid operation, inexact, input denormal). The floating-point unit must be enabled in CPACR register to prevent NOCP UsageFault exception.

Valid inputs are nonnegative numbers. Due to effectivity reason this function is written using inline assembly and thus is not ANSI-C compliant.

#### Code Example

```

#include "gflib.h"

tFloat fltIn;
tFloat fltOut;

void main(void)
{
    // input value = 0.5
    fltIn = (tFloat) 0.5;
}

```

```

// output should be 0.70710678
fltOut = GFLIB_Sqrt_FLT(fltIn);

// output should be 0.70710678
fltOut = GFLIB_Sqrt(fltIn,FLT);

// ######
// Available only if single precision floating point
// implementation selected as default
// #####
// #####
// output should be 0.70710678
fltOut = GFLIB_Sqrt(fltIn);
}

```

## 2.36 Function GFLIB\_Tan

This function implements an approximation of tangent function.

### Description

The GFLIB\_Tan function provides a computational method for calculation of the trigonometric tangent function  $\tan(x)$ , using the piece-wise polynomial approximation.

### Re-entrancy

The function is re-entrant.

#### 2.36.1 Function GFLIB\_Tan\_F32

##### Declaration

```
tFrac32 GFLIB_Tan_F32(tFrac32 f32In, const GFLIB_TAN_T_F32 *const pParam);
```

##### Arguments

Table 226. GFLIB\_Tan\_F32 arguments

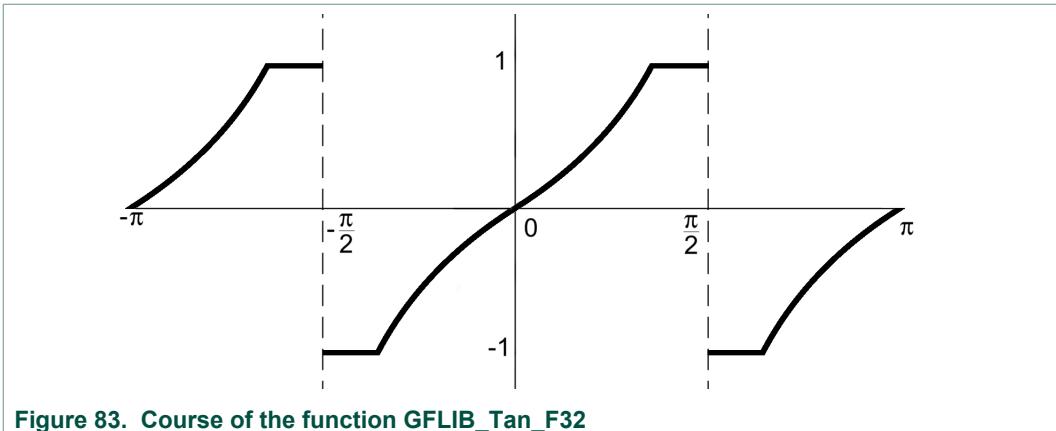
Type	Name	Direction	Description
tFrac32	f32In	input	Input argument is a 32-bit number that contains an angle in radians from interval $[-\pi, \pi]$ normalized between $[-1, 1]$ .
const GFLIB_TAN_T_F32 *const	pParam	input	Pointer to an array of Taylor coefficients. In case the default approximation coefficients are used, the &pParam must be replaced with <a href="#">GFLIB_TAN_DEFAULT_F32</a> symbol.

##### Return

The function returns  $\tan(\pi \cdot f32In)$  as a fixed point 32-bit number, normalized between  $[-1, 1]$ .

##### Implementation details

The input values are scaled from  $[-\pi, \pi]$  radians to  $[-1, 1]$  in order to fit in the available fixed-point fractional range. Output values are saturated. The function uses a piece-wise 4th order polynomial approximation.



**Figure 83. Course of the function GFLIB\_Tan\_F32**

**Note:** The input pointer must contain a valid address otherwise a fault may occur (MemManage, BusFault, UsageFault, HardFault).

#### Code Example

```
#include "gflib.h"

tFrac32 f32Angle;
tFrac32 f32Output;

void main(void)
{
    // input angle = 0.25 => pi/4
    f32Angle = FRAC32(0.25);

    // output should be 0x7FFFFFFF = 1
    f32Output = GFLIB_Tan_F32(f32Angle, GFLIB_TAN_DEFAULT_F32);

    // output should be 0x7FFFFFFF = 1
    f32Output = GFLIB_Tan(f32Angle, GFLIB_TAN_DEFAULT_F32, F32);

    // ######
    // Available only if 32-bit fractional implementation selected
    // as default
    // #####
    // output should be 0x7FFFFFFF = 1
    f32Output = GFLIB_Tan(f32Angle);
}
```

### 2.36.2 Function GFLIB\_Tan\_F16

#### Declaration

```
tFrac16 GFLIB_Tan_F16(tFrac16 f16In, const GFLIB_TAN_T_F16 *const pParam);
```

**Arguments****Table 227. GFLIB\_Tan\_F16 arguments**

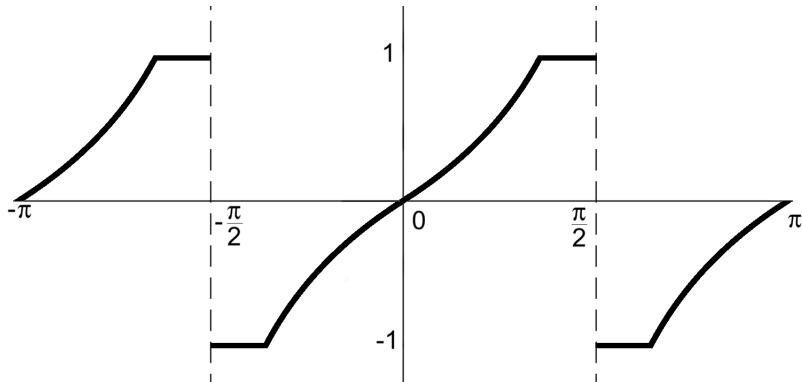
Type	Name	Direction	Description
tFrac16	f16In	input	Input argument is a 16-bit number that contains an angle in radians from interval $[-\pi, \pi]$ normalized between $[-1, 1]$ .
const <a href="#">GFLIB_TAN_T_F16</a> *const	pParam	input	Pointer to an array of Taylor coefficients. In case the default approximation coefficients are used, the &pParam must be replaced with <a href="#">GFLIB_SIN_DEFAULT_F16</a> symbol.

**Return**

The function returns  $\tan(\pi \cdot f16In)$  as a fixed point 16-bit number, normalized between  $[-1, 1]$ .

**Implementation details**

The input values are scaled from  $[-\pi, \pi]$  radians to  $[-1, 1]$  in order to fit in the available fixed-point fractional range. Output values are saturated. The function uses a piece-wise 4th order polynomial approximation; the default polynomial coefficients are provided in the [GFLIB\\_TAN\\_DEFAULT\\_F16](#) structure.

**Figure 84. Course of the function GFLIB\_Tan\_F16**

**Note:** The input pointer must contain a valid address otherwise a fault may occur (MemManage, BusFault, UsageFault, HardFault).

**Code Example**

```
#include "gflib.h"

tFrac16 f16Angle;
tFrac16 f16Output;

void main(void)
{
    // input angle = 0.25 => pi/4
    f16Angle = FRAC16(0.25);

    // output should be 0x7FFF = 1
    f16Output = GFLIB_Tan_F16(f16Angle, GFLIB\_TAN\_DEFAULT\_F16);
```

```

// output should be 0x7FFF = 1
f16Output = GFLIB_Tan(f16Angle, GFLIB\_TAN\_DEFAULT\_F16, F16);

// ##### Available only if 16-bit fractional implementation selected
// as default
// #####
// output should be 0x7FFF = 1
f16Output = GFLIB_Tan(f16Angle);
}

```

### 2.36.3 Function GFLIB\_Tan\_FLT

#### Declaration

```
tFloat GFLIB_Tan_FLT(tFloat fltIn, const GFLIB\_TAN\_T\_FLT *const pParam);
```

#### Arguments

Table 228. GFLIB\_Tan\_FLT arguments

Type	Name	Direction	Description
<a href="#">tFloat</a>	fltIn	input	Input argument is a single precision floating point number that contains an angle in radians from interval $(-\pi, \pi)$ .
const <a href="#">GFLIB_TAN_T_FLT</a> *const	pParam	input	Pointer to an array of approximation coefficients. In case the default approximation coefficients are used, the &pParam must be replaced with <a href="#">GFLIB_TAN_DEFAULT_FLT</a> symbol.

#### Return

The function returns  $\tan(\text{fltIn})$  as a single precision floating point number.

#### Implementation details

The function uses a rational polynomial approximation. The default polynomial coefficients are provided in the [GFLIB\\_TAN\\_DEFAULT\\_FLT](#) structure.

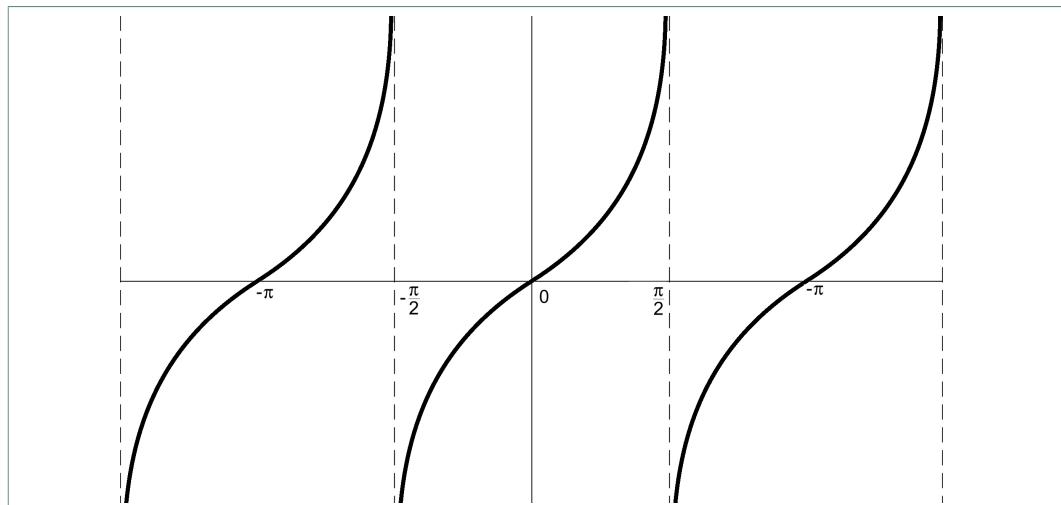


Figure 85. Course of the function GFLIB\_Tan\_FLT

**Note:** The function may raise floating-point exceptions (invalid operation, division by zero, overflow, underflow, inexact, input denormal). The floating-point unit must be enabled in CPACR register to prevent NOCP UsageFault exception.

The input pointer must contain a valid address otherwise a fault may occur (MemManage, BusFault, UsageFault, HardFault).

### Code Example

```
#include "gflib.h"

tFloat fltAngle;
tFloat fltOutput;

void main(void)
{
    // input angle = pi/4
    fltAngle = (tFloat)0.78539816;

    // output should be 1
    fltOutput = GFLIB_Tan_FLT(fltAngle, GFLIB_TAN_DEFAULT_FLT);

    // output should be 1
    fltOutput = GFLIB_Tan(fltAngle, GFLIB_TAN_DEFAULT_FLT, FLT);

    // ##### Available only if single precision floating point
    // implementation selected as default
    // #####
    // output should be 1
    fltOutput = GFLIB_Tan(fltAngle);
}
```

## 2.37 Function GFLIB\_UpperLimit

This function tests whether the input value is below the upper limit.

### Description

The GFLIB\_UpperLimit function tests whether the input value is below the upper limit. If so, the input value will be returned. Otherwise, if the input value is above the upper limit, the upper limit will be returned.

The upper limit UpperLimit can be found in the parameters structure, supplied to the function as a pointer pParam.

**Note:** The input pointer must contain a valid address otherwise a fault may occur (MemManage, BusFault, UsageFault, HardFault).

### Re-entrancy

The function is re-entrant.

### 2.37.1 Function `GFLIB_UpperLimit_F32`

#### Declaration

```
tFrac32 GFLIB_UpperLimit_F32(tFrac32 f32In, const
GFLIB_UPPERLIMIT_T_F32 *const pParam);
```

#### Arguments

Table 229. `GFLIB_UpperLimit_F32` arguments

Type	Name	Direction	Description
<code>tFrac32</code>	<code>f32In</code>	<code>input</code>	Input value.
<code>const GFLIB_UPPERLIMIT_T_F32</code> <code>*const</code>	<code>pParam</code>	<code>input</code>	Pointer to the limits structure.

#### Return

The input value in case the input value is below the limit, or the upper limit if the input value is above the limit.

#### Code Example

```
#include "gplib.h"

tFrac32 f32In;
tFrac32 f32Out;
GFLIB_UPPERLIMIT_T_F32 f32trMyUpperLimit = GFLIB_UPPERLIMIT_DEFAULT_F32;

void main(void)
{
    // upper limit
    f32trMyUpperLimit.f32UpperLimit = FRAC32(0.5);
    // input value = 0.75
    f32In = FRAC32(0.75);

    // output should be 0x40000000 ~ FRAC32(0.5)
    f32Out = GFLIB_UpperLimit_F32(f32In,&f32trMyUpperLimit);

    // output should be 0x40000000 ~ FRAC32(0.5)
    f32Out = GFLIB_UpperLimit(f32In,&f32trMyUpperLimit,F32);

    // ##### Available only if 32-bit fractional implementation selected #####
    // as default
    // #####
}

// output should be 0x40000000 ~ FRAC32(0.5)
f32Out = GFLIB_UpperLimit(f32In,&f32trMyUpperLimit);
```

## 2.37.2 Function `GFLIB_UpperLimit_F16`

### Declaration

```
tFrac16 GFLIB_UpperLimit_F16(tFrac16 f16In, const
GFLIB_UPPERLIMIT_T_F16 *const pParam);
```

### Arguments

Table 230. `GFLIB_UpperLimit_F16` arguments

Type	Name	Direction	Description
<code>tFrac16</code>	<code>f16In</code>	<code>input</code>	Input value.
<code>const GFLIB_UPPERLIMIT_T_F16</code> <code>*const</code>	<code>pParam</code>	<code>input</code>	Pointer to the limits structure.

### Return

The input value in case the input value is below the limit, or the upper limit if the input value is above the limit.

### Code Example

```
#include "gplib.h"

tFrac16 f16In;
tFrac16 f16Out;
GFLIB_UPPERLIMIT_T_F16 f16trMyUpperLimit = GFLIB_UPPERLIMIT_DEFAULT_F16;

void main(void)
{
    // upper limit
    f16trMyUpperLimit.f16UpperLimit = FRAC16(0.5);
    // input value = 0.75
    f16In = FRAC16(0.75);

    // output should be 0x4000 ~ FRAC16(0.5)
    f16Out = GFLIB_UpperLimit_F16(f16In,&f16trMyUpperLimit);

    // output should be 0x4000 ~ FRAC16(0.5)
    f16Out = GFLIB_UpperLimit(f16In,&f16trMyUpperLimit,F16);

    // ##### Available only if 16-bit fractional implementation selected #####
    // as default
    // #####
}

// output should be 0x4000 ~ FRAC16(0.5)
f16Out = GFLIB_UpperLimit(f16In,&f16trMyUpperLimit);
```

### 2.37.3 Function GFLIB\_UpperLimit\_FLT

#### Declaration

```
tFloat GFLIB_UpperLimit_FLT(tFloat fltIn, const
GFLIB_UPPERLIMIT_T_FLT *const pParam);
```

#### Arguments

Table 231. GFLIB\_UpperLimit\_FLT arguments

Type	Name	Direction	Description
tFloat	fltIn	input	Input value.
const GFLIB_UPPERLIMIT_T_FLT *const	pParam	input	Pointer to the limits structure.

#### Return

The input value in case the input value is below the limit, or the upper limit if the input value is above the limit.

**Note:** The function may raise floating-point exceptions (invalid operation, input denormal). The floating-point unit must be enabled in CPACR register to prevent NOCP UsageFault exception.

#### Code Example

```
#include "gplib.h"

tFloat fltIn;
tFloat fltOut;
GFLIB_UPPERLIMIT_T_FLT flttrMyUpperLimit = GFLIB_UPPERLIMIT_DEFAULT_FLT;

void main(void)
{
    // upper limit
    flttrMyUpperLimit.fltUpperLimit = 0.5;
    // input value = 0.75
    fltIn = (tFloat) 0.75;

    // output should be 0.5
    fltOut = GFLIB_UpperLimit_FLT(fltIn,&flttrMyUpperLimit);

    // output should be 0.5
    fltOut = GFLIB_UpperLimit(fltIn,&flttrMyUpperLimit,FLT);

    // ##### Available only if single precision floating point
    // implementation selected as default
    // #####
    // output should be 0.5
    fltOut = GFLIB_UpperLimit(fltIn,&flttrMyUpperLimit);

}
```

## 2.38 Function GFLIB\_VectorLimit

This function limits the magnitude of the input vector.

### Description

The GFLIB\_VectorLimit function limits the magnitude of the input vector, keeping its direction unchanged. Limitation is performed as follows:

$$y_{out} = \begin{cases} \frac{y_{in}}{\sqrt{x_{in}^2 + y_{in}^2}} \cdot L & \text{if } \sqrt{x_{in}^2 + y_{in}^2} > L \\ y_{in} & \text{if } \sqrt{x_{in}^2 + y_{in}^2} \leq L \end{cases}$$

$$x_{out} = \begin{cases} \frac{x_{in}}{\sqrt{x_{in}^2 + y_{in}^2}} \cdot L & \text{if } \sqrt{x_{in}^2 + y_{in}^2} > L \\ x_{in} & \text{if } \sqrt{x_{in}^2 + y_{in}^2} \leq L \end{cases}$$

Equation GFLIB\_VectorLimit\_Eq1

Where:

- $x_{in}$ ,  $y_{in}$  and  $x_{out}$ ,  $y_{out}$  are the co-ordinates of the input and output vector, respectively
- $L$  is the maximum magnitude of the vector

The input vector co-ordinates are defined by the structure pointed to by the `pIn` parameter, and the output vector co-ordinates be found in the structure pointed by the `pOut` parameter. The maximum vector magnitude is defined in the parameters structure pointed to by the `pParam` function parameter.

A graphical interpretation of the function can be seen in the figure below.

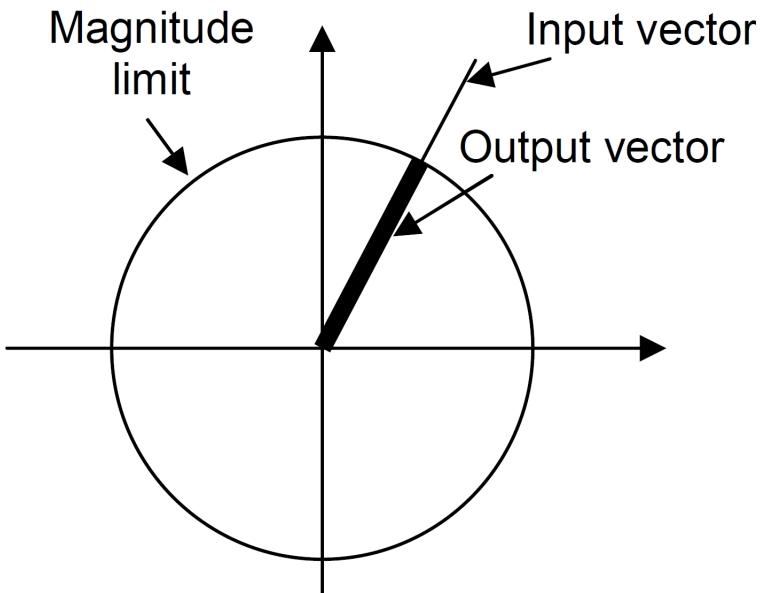


Figure 86. Graphical interpretation of the GFLIB\_VectorLimit function.

If an actual limitation occurs, the function will return [TRUE](#), otherwise the [FALSE](#) will be returned.

**Note:** The input/output pointers must contain valid addresses, otherwise a fault may occur (MemManage, BusFault, UsageFault, HardFault).

### Re-entrancy

The function is re-entrant.

#### 2.38.1 Function `GFLIB_VectorLimit_F32`

##### Declaration

```
tBool GFLIB_VectorLimit_F32(SWLBS_2Syst_F32 *const pOut, const
                           SWLBS_2Syst_F32 *const pIn, const GFLIB_VECTORLIMIT_T_F32 *const
                           pParam);
```

##### Arguments

Table 232. `GFLIB_VectorLimit_F32` arguments

Type	Name	Direction	Description
const <a href="#">SWLBS_2Syst_F32</a> *const	pIn	input	Pointer to the structure of the input vector.
<a href="#">SWLBS_2Syst_F32</a> *const	pOut	output	Pointer to the structure of the limited output vector.
const <a href="#">GFLIB_VECTORLIMIT_T_F32</a> *const	pParam	input	Pointer to the parameters structure.

##### Return

The function will return TRUE if the input vector is being limited or FALSE otherwise.

##### Implementation details

The output vector will be computed as zero if the input vector magnitude is lower than  $2^{-15}$ , regardless of the set maximum magnitude of the input vector. The function returns TRUE in this case.

**Caution:** The 16 least significant bits of `pParam->f32Limit` are ignored. This means that the defined magnitude must be equal to or greater than  $2^{-15}$ , otherwise the result is undefined.

**Note:** The function calls the AMMCLIB square root routine `GFLIB_Sqrt_F32`. The function uses a floating-point square root instruction.

The function may raise floating-point exceptions (invalid operation, inexact, input denormal). The floating-point unit must be enabled in CPACR register to prevent NOCP UsageFault exception.

##### Code Example

```
#include "gflib.h"

SWLBS\_2Syst\_F32 f32pIn;
SWLBS\_2Syst\_F32 f32pOut;
GFLIB\_VECTORLIMIT\_T\_F32 f32trMyVectorLimit = GFLIB\_VECTORLIMIT\_DEFAULT\_F32;
```

```

tBool bLim;

void main(void)
{
    // desired magnitude of the input vector
    f32trMyVectorLimit.f32Limit = FRAC32(0.25);
    // input vector
    f32pIn.f32Arg1 = FRAC32(0.25);
    f32pIn.f32Arg2 = FRAC32(0.25);

    // output should be:
    // bLim = TRUE;
    // f32pOut.f32Arg1 = 0x16A08000 ~ FRAC32(0.17677)
    // f32pOut.f32Arg2 = 0x16A08000 ~ FRAC32(0.17677)
    bLim = GFLIB_VectorLimit_F32(&f32pOut, &f32pIn, &f32trMyVectorLimit);

    // output should be:
    // bLim = TRUE;
    // f32pOut.f32Arg1 = 0x16A08000 ~ FRAC32(0.17677)
    // f32pOut.f32Arg2 = 0x16A08000 ~ FRAC32(0.17677)
    bLim = GFLIB_VectorLimit(&f32pOut, &f32pIn, &f32trMyVectorLimit, F32);

    // ######
    // Available only if 32-bit fractional implementation selected
    // as default
    // #####
    // output should be:
    // bLim = TRUE;
    // f32pOut.f32Arg1 = 0x16A08000 ~ FRAC32(0.17677)
    // f32pOut.f32Arg2 = 0x16A08000 ~ FRAC32(0.17677)
    bLim = GFLIB_VectorLimit(&f32pOut, &f32pIn, &f32trMyVectorLimit);
}

```

## 2.38.2 Function GFLIB\_VectorLimit\_F16

### Declaration

```
tBool GFLIB_VectorLimit_F16(SWLBS_2Syst_F16 *const pOut, const
                            SWLBS_2Syst_F16 *const pIn, const GFLIB_VECTORLIMIT_T_F16 *const
                            pParam);
```

### Arguments

Table 233. GFLIB\_VectorLimit\_F16 arguments

Type	Name	Direction	Description
const SWLBS_2Syst_F16 *const	pIn	input	Pointer to the structure of the input vector.
SWLBS_2Syst_F16 *const	pOut	output	Pointer to the structure of the limited output vector.
const GFLIB_VECTORLIMIT_T_F16 *const	pParam	input	Pointer to the parameters structure.

### Return

The function will return TRUE if the input vector is being limited, or FALSE otherwise.

**Caution:** The maximum vector magnitude in the parameters structure, the `pParam->f16Limit`, must be positive and equal to or greater than "0", otherwise the result is undefined. The function does not check for the valid range of the parameter `pParam->f16Limit`.

**Note:** The function calls the AMMCLIB square root routine [GFLIB\\_Sqrt\\_F16](#). The function uses a floating-point square root instruction.

The function may raise floating-point exceptions (invalid operation, inexact, input denormal). The floating-point unit must be enabled in CPACR register to prevent NOCP UsageFault exception.

### Code Example

```
#include "gflib.h"

SWLIBS_2Syst_F16 f16pIn;
SWLIBS_2Syst_F16 f16pOut;
GFLIB_VECTORLIMIT_T_F16 f16trMyVectorLimit = GFLIB_VECTORLIMIT_DEFAULT_F16;
tBool bLim;

void main(void)
{
    // desired magnitude of the input vector
    f16trMyVectorLimit.f16Limit = FRAC16(0.25);
    // input vector
    f16pIn.f16Arg1 = FRAC16(0.25);
    f16pIn.f16Arg2 = FRAC16(0.25);

    // output should be:
    // bLim = TRUE;
    // f16pOut.f16Arg1 = 0x16A0 ~ FRAC16(0.17677)
    // f16pOut.f16Arg2 = 0x16A0 ~ FRAC16(0.17677)
    bLim = GFLIB_VectorLimit_F16(&f16pOut, &f16pIn, &f16trMyVectorLimit);

    // output should be:
    // bLim = TRUE;
    // f16pOut.f16Arg1 = 0x16A0 ~ FRAC16(0.17677)
    // f16pOut.f16Arg2 = 0x16A0 ~ FRAC16(0.17677)
    bLim = GFLIB_VectorLimit(&f16pOut, &f16pIn, &f16trMyVectorLimit, F16);

    // ######
    // Available only if 16-bit fractional implementation selected
    // as default
    // #####
    // output should be:
    // bLim = TRUE;
    // f16pOut.f16Arg1 = 0x16A0 ~ FRAC16(0.17677)
    // f16pOut.f16Arg2 = 0x16A0 ~ FRAC16(0.17677)
    bLim = GFLIB_VectorLimit(&f16pOut, &f16pIn, &f16trMyVectorLimit);
}
```

### 2.38.3 Function `GFLIB_VectorLimit_FLT`

#### Declaration

```
tBool GFLIB_VectorLimit_FLT(SWLBS_2SystFLT *const pOut, const
                            SWLBS_2SystFLT *const pIn, const GFLIB_VECTORLIMIT_T_FLT *const
                            pParam);
```

#### Arguments

**Table 234. `GFLIB_VectorLimit_FLT` arguments**

Type	Name	Direction	Description
const SWLBS_2SystFLT *const	pIn	input	Pointer to the structure of the input vector.
SWLBS_2SystFLT *const	pOut	output	Pointer to the structure of the limited output vector.
const GFLIB_VECTORLIMIT_T_FLT *const	pParam	input	Pointer to the parameters structure.

#### Return

The function will return TRUE if the input vector is being limited, or FALSE otherwise.

**Note:** The function may raise floating-point exceptions (invalid operation, division by zero, overflow, underflow, inexact, input denormal). The floating-point unit must be enabled in CPACR register to prevent NOCP UsageFault exception.

The function calls the AMMCLIB square root routine [GFLIB\\_Sqrt\\_FLT](#).

**Caution:** The maximum vector magnitude in the parameters structure, the `pParam->fltLimit`, must be positive and equal to or greater than "0", otherwise the result is undefined. The function does not check for the valid range of the parameter `pParam->fltLimit`.

#### Code Example

```
#include "gplib.h"

SWLBS_2SystFLT fltpIn;
SWLBS_2SystFLT fltpOut;
GFLIB_VECTORLIMIT_T_FLT flttrMyVectorLimit = GFLIB_VECTORLIMIT_DEFAULT_FLT;
tBool bLim;

void main(void)
{
    // desired magnitude of the input vector
    flttrMyVectorLimit.fltLimit = (tFloat)0.25;
    // input vector
    fltpIn.fltArg1 = (tFloat)0.25;
    fltpIn.fltArg2 = (tFloat)0.25;

    // output should be:
    // bLim = TRUE;
    // fltpOut.fltArg1 = 0.17677
    // fltpOut.fltArg2 = 0.17677
```

```

bLim = GFLIB_VectorLimit_FLT(&fltpOut,&fpltIn,&fltrMyVectorLimit);

// output should be:
// bLim = TRUE;
// fltpOut.fltArg1 = 0.17677
// fltpOut.fltArg2 = 0.17677
bLim = GFLIB_VectorLimit(&fltpOut,&fltpIn,&fltrMyVectorLimit,FLT);

// ######
// Available only if single precision floating point
// implementation selected as default
// #####
// output should be:
// bLim = TRUE;
// fltpOut.fltArg1 = 0.17677
// fltpOut.fltArg2 = 0.17677
bLim = GFLIB_VectorLimit(&fltpOut,&fltpIn,&fltrMyVectorLimit);
}

```

## 2.39 Function GFLIB\_VLog10\_FLT

This function calculates a base-10 logarithm of absolute values of an array.

### Declaration

```
void GFLIB_VLog10_FLT(tFloat *pInOut, tU32 u32N, const
GFLIB\_VLOG10\_T\_FLT *const pParam);
```

### Arguments

**Table 235. GFLIB\_VLog10\_FLT arguments**

Type	Name	Direction	Description
<a href="#">tFloat</a> *	pInOut	<a href="#">input, output</a>	Pointer to the floating-point input/output data array. Must be aligned to a double-word boundary.
<a href="#">tU32</a>	u32N	<a href="#">input</a>	Array length. Must be divisible by 8 (i.e. 8, 16, 24, ...).
const <a href="#">GFLIB_VLOG10_T_FLT</a> *const	pParam	<a href="#">input</a>	Pointer to an array of approximation coefficients.

### Description

The function calculates a base-10 logarithm of the absolute values of the input array. Results overwrite the inputs (in-place processing). The default approximating polynomial coefficients are provided in the [GFLIB\\_VLOG10\\_DEFAULT\\_FLT](#) structure.

This is a vectorized variant of function [GFLIB\\_Log10\\_FLT](#). The vectorized form provides processing speed benefit for large arrays of input values.

**Note:** The function may raise floating-point exceptions (overflow, underflow, inexact, input denormal). The floating-point unit must be enabled in CPACR register to prevent NOCP UsageFault exception.

## Code Example

```
#include "gplib.h"

tFloat pValues[8] = {1.0F, 1.0F, 1.0F, 1.0F, 1.0F, 1.0F, 1.0F, 1.0F};

void main(void)
{
    // all outputs should be 0
    GFLIB_VLog10_FLT(pValues, 8U, GFLIB_VLOG10_DEFAULT_FLT);

    // all outputs should be 0
    GFLIB_VLog10(pValues, 8U, GFLIB_VLOG10_DEFAULT_FLT, FLT);

    // ##### Available only if single precision floating point
    // implementation selected as default
    // #####
    // all outputs should be 0
    GFLIB_VLog10(pValues, 8U);
}
```

## 2.40 Function GFLIB\_VMin

This function finds the minimum of a vector.

### Description

The function GFLIB\_VMin finds the minimum in a vector of values and returns the index of that value. If there are several equal minimums, the index of the last one is returned.

**Note:** *The input pointer must contain a valid address otherwise a fault may occur (MemManage, BusFault, UsageFault, HardFault).*

### Re-entrancy

The function is re-entrant.

### 2.40.1 Function GFLIB\_VMin\_F32

#### Declaration

```
tU32 GFLIB_VMin_F32(const tFrac32 *pIn, tU32 u32N);
```

#### Arguments

Table 236. GFLIB\_VMin\_F32 arguments

Type	Name	Direction	Description
const tFrac32 *	pIn	input	Pointer to an array of 32-bit fixed-point signed input values.
tU32	u32N	input	Length of the input array.

#### Return

The index of the minimum of the input array.

**Code Example**

```
#include "gplib.h"

void main(void)
{
    tFrac32 pInputArray[5] = { 6, 3, 1, 4, 8 };
    tFrac32 f32MinValue;

    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    f32MinValue = pInputArray[GFLIB_VMin_F32(pInputArray, 5)];

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    f32MinValue = pInputArray[GFLIB_VMin(pInputArray, 5, F32)];

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 32-bit fractional implementation is selected as default.
    f32MinValue = pInputArray[GFLIB_VMin(pInputArray, 5)];
}
```

**2.40.2 Function GFLIB\_VMin\_F16****Declaration**

```
tU16 GFLIB_VMin_F16(const tFrac16 *pIn, tU16 u16N);
```

**Arguments****Table 237. GFLIB\_VMin\_F16 arguments**

Type	Name	Direction	Description
const tFrac16 *	pIn	input	Pointer to an array of 16-bit fixed-point signed input values.
tU16	u16N	input	Length of the input array.

**Return**

The index of the minimum of the input array.

**Note:** The library offers faster inlined version of this function for the input array lengths of 4 to 16, see [GFLIB\\_VMin4\\_F16](#), [GFLIB\\_VMin5\\_F16](#), etc.

**Code Example**

```
#include "gplib.h"

void main(void)
{
    tFrac16 pInputArray[5] = { 6, 3, 1, 4, 8 };
    tFrac16 f16MinValue;

    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    f16MinValue = pInputArray[GFLIB_VMin_F16(pInputArray, 5)];
```

```

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
f16MinValue = pInputArray[GFLIB_VMin(pInputArray, 5, F16)];

// Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 16-bit fractional implementation is selected as default.
f16MinValue = pInputArray[GFLIB_VMin(pInputArray, 5)];
}

```

### 2.40.3 Function GFLIB\_VMin\_FLT

#### Declaration

```
tU32 GFLIB_VMin_FLT(const tFloat *pIn, tU32 u32N);
```

#### Arguments

Table 238. GFLIB\_VMin\_FLT arguments

Type	Name	Direction	Description
const tFloat *	pIn	input	Pointer to an array of single precision floating-point input values.
tU32	u32N	input	Length of the input array.

#### Return

The index of the minimum of the input array.

#### Code Example

```

#include "gflib.h"

void main(void)
{
    tFloat pInputArray[5] = {6.0f, 3.0f, 1.0f, 4.0f, 8.0f};
    tFloat fltMinValue;

    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    fltMinValue = pInputArray[GFLIB_VMin_FLT(pInputArray, 5)];

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    fltMinValue = pInputArray[GFLIB_VMin(pInputArray, 5, FLT)];

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if floating point implementation is selected as default.
    fltMinValue = pInputArray[GFLIB_VMin(pInputArray, 5)];
}

```

#### 2.40.4 Function `GFLIB_VMin4_F16`

##### Declaration

```
INLINE tU16 GFLIB_VMin4_F16(const tFrac16 *pIn);
```

##### Arguments

**Table 239.** `GFLIB_VMin4_F16` arguments

Type	Name	Direction	Description
const <code>tFrac16</code> *	pIn	input	Pointer to an array of 16-bit fixed-point signed input values. The length of the array must be 4 elements.

##### Return

The function returns the index of the smallest element of the input array.

##### Implementation details

The function finds the minimum in a 4-element vector of values and returns the index of that value. If there are several equal minimums, the index of an arbitrary one of them is returned. This function is faster than [GFLIB\\_VMin\\_F16](#).

##### Code Example

```
#include "gflib.h"

void main(void)
{
    tFrac16 pInputArray[4] = {6, 3, 1, 4};
    tFrac16 f16MinValue;

    f16MinValue = pInputArray[GFLIB_VMin4_F16(pInputArray)];
}
```

#### 2.40.5 Function `GFLIB_VMin5_F16`

##### Declaration

```
INLINE tU16 GFLIB_VMin5_F16(const tFrac16 *pIn);
```

##### Arguments

**Table 240.** `GFLIB_VMin5_F16` arguments

Type	Name	Direction	Description
const <code>tFrac16</code> *	pIn	input	Pointer to an array of 16-bit fixed-point signed input values. The length of the array must be 5 elements.

##### Return

The function returns the index of the smallest element of the input array.

**Implementation details**

The function finds the minimum in a 5-element vector of values and returns the index of that value. If there are several equal minimums, the index of an arbitrary one of them is returned. There must be two consecutive readable bytes in the memory following the last element of the input array. This function is faster than [GFLIB\\_VMin\\_F16](#).

**Code Example**

```
#include "gflib.h"

void main(void)
{
    tFrac16 pInputArray[5] = {6, 3, 1, 4, 5};
    tFrac16 f16MinValue;

    f16MinValue = pInputArray[GFLIB_VMin5_F16(pInputArray)];
}
```

**2.40.6 Function GFLIB\_VMin6\_F16****Declaration**

INLINE [tU16](#) GFLIB\_VMin6\_F16(const [tFrac16](#) \*pIn);

**Arguments****Table 241. GFLIB\_VMin6\_F16 arguments**

Type	Name	Direction	Description
const <a href="#">tFrac16</a> *	pIn	input	Pointer to an array of 16-bit fixed-point signed input values. The length of the array must be 6 elements.

**Return**

The function returns the index of the smallest element of the input array.

**Implementation details**

The function finds the minimum in a 6-element vector of values and returns the index of that value. If there are several equal minimums, the index of an arbitrary one of them is returned. This function is faster than [GFLIB\\_VMin\\_F16](#).

**Code Example**

```
#include "gflib.h"

void main(void)
{
    tFrac16 pInputArray[6] = {6, 3, 1, 4, 5, 6};
    tFrac16 f16MinValue;

    f16MinValue = pInputArray[GFLIB_VMin6_F16(pInputArray)];
}
```

### 2.40.7 Function `GFLIB_VMin7_F16`

#### Declaration

```
INLINE tU16 GFLIB_VMin7_F16(const tFrac16 *pIn);
```

#### Arguments

**Table 242. GFLIB\_VMin7\_F16 arguments**

Type	Name	Direction	Description
const <code>tFrac16</code> *	pIn	input	Pointer to an array of 16-bit fixed-point signed input values. The length of the array must be 7 elements.

#### Return

The function returns the index of the smallest element of the input array.

#### Implementation details

The function finds the minimum in a 7-element vector of values and returns the index of that value. If there are several equal minimums, the index of an arbitrary one of them is returned. This function is faster than [GFLIB\\_VMin\\_F16](#).

#### Code Example

```
#include "gflib.h"

void main(void)
{
    tFrac16 pInputArray[7] = {6, 3, 1, 4, 5, 6, 7};
    tFrac16 f16MinValue;

    f16MinValue = pInputArray[GFLIB_VMin7_F16(pInputArray)];
}
```

### 2.40.8 Function `GFLIB_VMin8_F16`

#### Declaration

```
INLINE tU16 GFLIB_VMin8_F16(const tFrac16 *pIn);
```

#### Arguments

**Table 243. GFLIB\_VMin8\_F16 arguments**

Type	Name	Direction	Description
const <code>tFrac16</code> *	pIn	input	Pointer to an array of 16-bit fixed-point signed input values. The length of the array must be 8 elements.

#### Return

The function returns the index of the smallest element of the input array.

**Implementation details**

The function finds the minimum in a 8-element vector of values and returns the index of that value. If there are several equal minimums, the index of an arbitrary one of them is returned. This function is faster than [GFLIB\\_VMin\\_F16](#).

**Code Example**

```
#include "gflib.h"

void main(void)
{
    tFrac16 pInputArray[8] = {6, 3, 1, 4, 5, 6, 7, 8};
    tFrac16 f16MinValue;

    f16MinValue = pInputArray[GFLIB_VMin8_F16(pInputArray)];
}
```

**2.40.9 Function GFLIB\_VMin9\_F16****Declaration**

```
INLINE tU16 GFLIB_VMin9_F16(const tFrac16 *pIn);
```

**Arguments****Table 244. GFLIB\_VMin9\_F16 arguments**

Type	Name	Direction	Description
const tFrac16 *	pIn	input	Pointer to an array of 16-bit fixed-point signed input values. The length of the array must be 9 elements.

**Return**

The function returns the index of the smallest element of the input array.

**Implementation details**

The function finds the minimum in a 9-element vector of values and returns the index of that value. If there are several equal minimums, the index of an arbitrary one of them is returned. There must be two consecutive readable bytes in the memory following the last element of the input array. This function is faster than [GFLIB\\_VMin\\_F16](#).

**Code Example**

```
#include "gflib.h"

void main(void)
{
    tFrac16 pInputArray[9] = {6, 3, 1, 4, 5, 6, 7, 8, 9};
    tFrac16 f16MinValue;

    f16MinValue = pInputArray[GFLIB_VMin9_F16(pInputArray)];
}
```

### 2.40.10 Function `GFLIB_VMin10_F16`

#### Declaration

```
INLINE tU16 GFLIB_VMin10_F16(const tFrac16 *pIn);
```

#### Arguments

**Table 245. GFLIB\_VMin10\_F16 arguments**

Type	Name	Direction	Description
const <code>tFrac16</code> *	pIn	input	Pointer to an array of 16-bit fixed-point signed input values. The length of the array must be 10 elements.

#### Return

The function returns the index of the smallest element of the input array.

#### Implementation details

The function finds the minimum in a 10-element vector of values and returns the index of that value. If there are several equal minimums, the index of an arbitrary one of them is returned. This function is faster than [GFLIB\\_VMin\\_F16](#).

#### Code Example

```
#include "gflib.h"

void main(void)
{
    tFrac16 pInputArray[10] = {6, 3, 1, 4, 5, 6, 7, 8, 9, 10};
    tFrac16 f16MinValue;

    f16MinValue = pInputArray[GFLIB_VMin10_F16(pInputArray)];
}
```

### 2.40.11 Function `GFLIB_VMin11_F16`

#### Declaration

```
INLINE tU16 GFLIB_VMin11_F16(const tFrac16 *pIn);
```

#### Arguments

**Table 246. GFLIB\_VMin11\_F16 arguments**

Type	Name	Direction	Description
const <code>tFrac16</code> *	pIn	input	Pointer to an array of 16-bit fixed-point signed input values. The length of the array must be 11 elements.

#### Return

The function returns the index of the smallest element of the input array.

**Implementation details**

The function finds the minimum in a 11-element vector of values and returns the index of that value. If there are several equal minimums, the index of an arbitrary one of them is returned. There must be two consecutive readable bytes in the memory following the last element of the input array. This function is faster than [GFLIB\\_VMin\\_F16](#).

**Code Example**

```
#include "gflib.h"

void main(void)
{
    tFrac16 pInputArray[11] = { 6, 3, 1, 4, 5, 6, 7, 8, 9, 10, 11};
    tFrac16 f16MinValue;

    f16MinValue = pInputArray[GFLIB_VMin11_F16(pInputArray)];
}
```

**2.40.12 Function GFLIB\_VMin12\_F16****Declaration**

INLINE [tU16](#) GFLIB\_VMin12\_F16(const [tFrac16](#) \*pIn);

**Arguments****Table 247. GFLIB\_VMin12\_F16 arguments**

Type	Name	Direction	Description
const <a href="#">tFrac16</a> *	pIn	input	Pointer to an array of 16-bit fixed-point signed input values. The length of the array must be 12 elements.

**Return**

The function returns the index of the smallest element of the input array.

**Implementation details**

The function finds the minimum in a 12-element vector of values and returns the index of that value. If there are several equal minimums, the index of an arbitrary one of them is returned. This function is faster than [GFLIB\\_VMin\\_F16](#).

**Code Example**

```
#include "gflib.h"

void main(void)
{
    tFrac16 pInputArray[12] = { 6, 3, 1, 4, 5, 6, 7, 8, 9,
                               10, 11, 12};
    tFrac16 f16MinValue;

    f16MinValue = pInputArray[GFLIB_VMin12_F16(pInputArray)];
```

```
}
```

### 2.40.13 Function `GFLIB_VMin13_F16`

#### Declaration

```
INLINE tU16 GFLIB_VMin13_F16(const tFrac16 *pIn);
```

#### Arguments

**Table 248. GFLIB\_VMin13\_F16 arguments**

Type	Name	Direction	Description
const <code>tFrac16</code> *	pIn	input	Pointer to an array of 16-bit fixed-point signed input values. The length of the array must be 13 elements.

#### Return

The function returns the index of the smallest element of the input array.

#### Implementation details

The function finds the minimum in a 13-element vector of values and returns the index of that value. If there are several equal minimums, the index of an arbitrary one of them is returned. There must be two consecutive readable bytes in the memory following the last element of the input array. This function is faster than [GFLIB\\_VMin\\_F16](#).

#### Code Example

```

#include "gplib.h"

void main(void)
{
    tFrac16 pInputArray[13] = {6, 3, 1, 4, 5, 6, 7, 8, 9,
                                  10, 11, 12, 13};
    tFrac16 f16MinValue;

    f16MinValue = pInputArray[GFLIB_VMin13_F16(pInputArray)];
}
```

### 2.40.14 Function `GFLIB_VMin14_F16`

#### Declaration

```
INLINE tU16 GFLIB_VMin14_F16(const tFrac16 *pIn);
```

#### Arguments

**Table 249. GFLIB\_VMin14\_F16 arguments**

Type	Name	Direction	Description
const <code>tFrac16</code> *	pIn	input	Pointer to an array of 16-bit fixed-point signed input values. The length of the array must be 14 elements.

**Return**

The function returns the index of the smallest element of the input array.

**Implementation details**

The function finds the minimum in a 14-element vector of values and returns the index of that value. If there are several equal minimums, the index of an arbitrary one of them is returned. This function is faster than [GFLIB\\_VMin\\_F16](#).

**Code Example**

```
#include "gflib.h"

void main(void)
{
    tFrac16 pInputArray[14] = {6, 3, 1, 4, 5, 6, 7, 8, 9,
                               10, 11, 12, 13, 14};
    tFrac16 f16MinValue;

    f16MinValue = pInputArray[GFLIB_VMin14_F16(pInputArray)];
}
```

**2.40.15 Function GFLIB\_VMin15\_F16****Declaration**

```
INLINE tU16 GFLIB_VMin15_F16(const tFrac16 *pIn);
```

**Arguments****Table 250. GFLIB\_VMin15\_F16 arguments**

Type	Name	Direction	Description
const tFrac16 *	pIn	input	Pointer to an array of 16-bit fixed-point signed input values. The length of the array must be 15 elements.

**Return**

The function returns the index of the smallest element of the input array.

**Implementation details**

The function finds the minimum in a 15-element vector of values and returns the index of that value. If there are several equal minimums, the index of an arbitrary one of them is returned. There must be two consecutive readable bytes in the memory following the last element of the input array. This function is faster than [GFLIB\\_VMin\\_F16](#).

**Code Example**

```
#include "gflib.h"

void main(void)
{
    tFrac16 pInputArray[15] = {6, 3, 1, 4, 5, 6, 7, 8, 9,
```

```

        10, 11, 12, 13, 14, 15};

tFrac16 f16MinValue;

f16MinValue = pInputArray[GFLIB_VMin15_F16(pInputArray)];
}

```

### 2.40.16 Function GFLIB\_VMin16\_F16

#### Declaration

```
INLINE tU16 GFLIB_VMin16_F16(const tFrac16 *pIn);
```

#### Arguments

**Table 251. GFLIB\_VMin16\_F16 arguments**

Type	Name	Direction	Description
const tFrac16 *	pIn	input	Pointer to an array of 16-bit fixed-point signed input values. The length of the array must be 16 elements.

#### Return

The function returns the index of the smallest element of the input array.

#### Implementation details

The function finds the minimum in a 16-element vector of values and returns the index of that value. If there are several equal minimums, the index of an arbitrary one of them is returned. This function is faster than [GFLIB\\_VMin\\_F16](#).

#### Code Example

```

#include "gflib.h"

void main(void)
{
    tFrac16 pInputArray[16] = {6, 3, 1, 4, 5, 6, 7, 8, 9,
                               10, 11, 12, 13, 14, 15, 16};
    tFrac16 f16MinValue;

    f16MinValue = pInputArray[GFLIB_VMin16_F16(pInputArray)];
}

```

### 2.41 Function GMCLIB\_BetaProjection

The function implements the Beta-projection transformation.

#### Description

The Beta-projection transformation is used to transform values from the three-phase (A-B-C) coordinate system to the two-phase ( $\alpha$ - $\beta$ ) orthogonal coordinate system, according to the following equations:

$$\begin{bmatrix} u_\alpha \\ u_\beta \end{bmatrix} = \begin{bmatrix} \frac{2}{3} & -\frac{1}{3} & -\frac{1}{3} \\ 0 & \frac{\sqrt{3}}{3} & -\frac{\sqrt{3}}{3} \end{bmatrix} \cdot \begin{bmatrix} u_A \\ u_B \\ u_C \end{bmatrix}$$

Equation GMCLIB\_BetaProjection\_Eq1

where it is assumed that the axis A (axis of the first phase) and the axis  $\alpha$  are in the same direction. Compared to an ordinary Clarke transformation, the patent-pending Beta-projection provides superior immunity to nonlinear distortions such as DC offset, limitation, and certain higher harmonics. Beta-projection is an essential tool for the reconstruction of the angle of the induced voltage vector in a free-wheeling permanent magnet synchronous motor.

**Note:** The input/output pointers must contain valid addresses, otherwise a fault may occur (MemManage, BusFault, UsageFault, HardFault).

### Re-entrancy

The function is re-entrant.

#### 2.41.1 Function GMCLIB\_BetaProjection\_F32

##### Declaration

```
void GMCLIB_BetaProjection_F32(SWLBS_2Syst_F32 *const pOut,
                                const SWLBS_3Syst_F32 *const pIn);
```

##### Arguments

Table 252. GMCLIB\_BetaProjection\_F32 arguments

Type	Name	Direction	Description
const SWLBS_3Syst_F32 *const	pIn	input	Pointer to the structure containing data of the three-phase stationary system (f32A-f32B-f32C). Arguments of the structure contain fixed point 32-bit values.
SWLBS_2Syst_F32 *const	pOut	output	Pointer to the structure containing data of the two-phase stationary orthogonal system ( $\alpha$ - $\beta$ ). Arguments of the structure contain fixed point 32-bit values.

**Note:** The inputs and the outputs are normalized to fit in the range [-1, 1].

##### Code Example

```
#include "gmclib.h"

SWLBS_3Syst_F32 f32trAbc;
SWLBS_2Syst_F32 f32trAlBe;

void main(void)
{
    // input phase A ~ sin(45) ~ 0.707106781
    // input phase B ~ sin(45 + 120) ~ 0.258819045
    // input phase C ~ sin(45 - 120) ~ -0.965925826
    f32trAbc.f32Arg1 = FRAC32(0.707106781);
    f32trAbc.f32Arg2 = FRAC32(0.258819045);
    f32trAbc.f32Arg3 = FRAC32(-0.965925826);
```

```

// output should be
// f32trAlBe.f32Arg1 = 0x5A82799A ~ FRAC32(0.707106781)
// f32trAlBe.f32Arg2 = 0x5A827999 ~ FRAC32(0.707106781)
GMCLIB_BetaProjection_F32(&f32trAlBe, &f32trAbc);

// output should be
// f32trAlBe.f32Arg1 = 0x5A82799A ~ FRAC32(0.707106781)
// f32trAlBe.f32Arg2 = 0x5A827999 ~ FRAC32(0.707106781)
GMCLIB_BetaProjection(&f32trAlBe, &f32trAbc, F32);

// ##### Available only if 32-bit fractional implementation selected
// as default
// #####
// output should be
// f32trAlBe.f32Arg1 = 0x5A82799A ~ FRAC32(0.707106781)
// f32trAlBe.f32Arg2 = 0x5A827999 ~ FRAC32(0.707106781)
GMCLIB_BetaProjection(&f32trAlBe, &f32trAbc);
}

```

## 2.41.2 Function GMCLIB\_BetaProjection\_F16

### Declaration

```
void GMCLIB_BetaProjection_F16(SWLIBS_2Syst_F16 *const pOut,
                                const SWLIBS_3Syst_F16 *const pIn);
```

### Arguments

**Table 253. GMCLIB\_BetaProjection\_F16 arguments**

Type	Name	Direction	Description
const SWLIBS_3Syst_F16 *const	pIn	input	Pointer to the structure containing data of the three-phase stationary system (f16A-f16B-f16C). Arguments of the structure contain fixed point 16-bit values.
SWLIBS_2Syst_F16 *const	pOut	output	Pointer to the structure containing data of the two-phase stationary orthogonal system ( $\alpha$ - $\beta$ ). Arguments of the structure contain fixed point 16-bit values.

**Note:** The inputs and the outputs are normalized to fit in the range [-1, 1).

### Code Example

```
#include "gmclib.h"

SWLIBS_3Syst_F16 f16trAbc;
SWLIBS_2Syst_F16 f16trAlBe;

void main(void)
{
    // input phase A ~ sin(45) ~ 0.707106781
    // input phase B ~ sin(45 + 120) ~ 0.258819045
    // input phase C ~ sin(45 - 120) ~ -0.965925826
    f16trAbc.f16Arg1 = FRAC16(0.707106781);
    f16trAbc.f16Arg2 = FRAC16(0.258819045);
```

```

f16trAbc.f16Arg3 = FRAC16(-0.965925826);

// output should be
// f16trAlBe.f16Arg1 = 0x5A82 ~ FRAC16(0.707106781)
// f16trAlBe.f16Arg2 = 0x5A83 ~ FRAC16(0.707106781)
GMCLIB_BetaProjection_F16(&f16trAlBe,&f16trAbc);

// output should be
// f16trAlBe.f16Arg1 = 0x5A82 ~ FRAC16(0.707106781)
// f16trAlBe.f16Arg2 = 0x5A83 ~ FRAC16(0.707106781)
GMCLIB_BetaProjection(&f16trAlBe,&f16trAbc,F16);

// ######
// Available only if 16-bit fractional implementation selected
// as default
// #####
// output should be
// f16trAlBe.f16Arg1 = 0x5A82 ~ FRAC16(0.707106781)
// f16trAlBe.f16Arg2 = 0x5A83 ~ FRAC16(0.707106781)
GMCLIB_BetaProjection(&f16trAlBe,&f16trAbc);
}

```

### 2.41.3 Function GMCLIB\_BetaProjection\_FLT

#### Declaration

```
void GMCLIB_BetaProjection_FLT(SWLIBS\_2Syst\_FLT *const pOut,
const SWLIBS\_3Syst\_FLT *const pIn);
```

#### Arguments

**Table 254. GMCLIB\_BetaProjection\_FLT arguments**

Type	Name	Direction	Description
const <a href="#">SWLIBS_3Syst_FLT</a> *const	pIn	input	Pointer to the structure containing data of the three-phase stationary system (fltA-fltB-fltC). Arguments of the structure contain single precision floating point values.
<a href="#">SWLIBS_2Syst_FLT</a> *const	pOut	output	Pointer to the structure containing data of the two-phase stationary orthogonal system ( $\alpha$ - $\beta$ ). Arguments of the structure contain single precision floating point values.

**Note:** The function may raise floating-point exceptions (invalid operation, overflow, underflow, inexact, input denormal). The floating-point unit must be enabled in CPACR register to prevent NOCP UsageFault exception.

#### Code Example

```

#include "gmclib.h"

SWLIBS\_3Syst\_FLT flttrAbc;
SWLIBS\_2Syst\_FLT flttrAlBe;

void main(void)
{
    // input phase A ~ sin(45) ~ 0.707106781
    // input phase B ~ sin(45 + 120) ~ 0.258819045

```

```

// input phase C ~ sin(45 - 120) ~ -0.965925826
flttrAbc.fltArg1 = (tFloat)0.707106781;
flttrAbc.fltArg2 = (tFloat)0.258819045;
flttrAbc.fltArg3 = (tFloat)-0.965925826;

// output should be
// flttrAlBe.fltArg1 = 0.70710677
// flttrAlBe.fltArg2 = 0.70710671
GMCLIB_BetaProjection_FLT(&flttrAlBe, &flttrAbc);

// output should be
// flttrAlBe.fltArg1 = 0.70710677
// flttrAlBe.fltArg2 = 0.70710671
GMCLIB_BetaProjection(&flttrAlBe, &flttrAbc, FLT);

// ##### Available only if single precision floating point
// implementation selected as default
// #####
// output should be
// flttrAlBe.fltArg1 = 0.70710677
// flttrAlBe.fltArg2 = 0.70710671
GMCLIB_BetaProjection(&flttrAlBe, &flttrAbc);
}

```

## 2.42 Function GMCLIB\_BetaProjection3Ph

The function implements the three-phase Beta-projection transformation.

### Description

The three-phase Beta-projection transforms the input three-phase (A-B-C) coordinate system into an equivalent output three-phase coordinate system while eliminating certain types of non-linear distortion (e.g. DC offset, 3rd harmonics, limitation). The patent-pending algorithm calculates the following equations:

$$\begin{bmatrix} u_{Aout} \\ u_{Bout} \\ u_{Cout} \end{bmatrix} = \begin{bmatrix} \frac{2}{3} & -\frac{1}{3} & -\frac{1}{3} \\ -\frac{1}{3} & \frac{2}{3} & -\frac{1}{3} \\ -\frac{1}{3} & -\frac{1}{3} & \frac{2}{3} \end{bmatrix} \cdot \begin{bmatrix} u_A \\ u_B \\ u_C \end{bmatrix}$$

Equation GMCLIB\_BetaProjection3Ph\_Eq1

Refer to [GMCLIB\\_BetaProjection](#) function for a related Beta-projection transform that produces a two-phase output.

**Note:** The input/output pointers must contain valid addresses, otherwise a fault may occur (MemManage, BusFault, UsageFault, HardFault).

### Re-entrancy

The function is re-entrant.

### 2.42.1 Function GMCLIB\_BetaProjection3Ph\_F32

#### Declaration

```
void GMCLIB_BetaProjection3Ph_F32(SWLIBS\_3Syst\_F32 *const pOut,
const SWLIBS\_3Syst\_F32 *const pIn);
```

#### Arguments

**Table 255. GMCLIB\_BetaProjection3Ph\_F32 arguments**

Type	Name	Direction	Description
const <a href="#">SWLIBS_3Syst_F32</a> *const	pIn	input	Pointer to the structure containing data of the input three-phase stationary system (f32A-f32B-f32C). Arguments of the structure contain fixed point 32-bit values.
<a href="#">SWLIBS_3Syst_F32</a> *const	pOut	output	Pointer to the structure containing data of the output three-phase stationary orthogonal system (f32Aout-f32Bout-f32Cout). Arguments of the structure contain fixed point 32-bit values.

*Note:* The inputs and the outputs are normalized to fit in the range [-1, 1).

#### Code Example

```
#include "gmclib.h"

SWLIBS\_3Syst\_F32 f32trAbcIn;
SWLIBS\_3Syst\_F32 f32trAbcOut;

void main(void)
{
    // input phase A ~ sin(45) ~ 0.707106781
    // input phase B ~ sin(45 + 120) ~ 0.258819045
    // input phase C ~ sin(45 - 120) ~ -0.965925826
    f32trAbcIn.f32Arg1 = FRAC32(0.707106781);
    f32trAbcIn.f32Arg2 = FRAC32(0.258819045);
    f32trAbcIn.f32Arg3 = FRAC32(-0.965925826);
    // add DC offset - will be eliminated by the Beta-projection
    f32trAbcIn.f32Arg1 += FRAC32(0.1);
    f32trAbcIn.f32Arg2 += FRAC32(0.1);
    f32trAbcIn.f32Arg3 += FRAC32(0.1);

    // output should be
    // f32trAbcOut.f32Arg1 = 0x5A82799A ~ FRAC32(0.707)
    // f32trAbcOut.f32Arg2 = 0x2120FB83 ~ FRAC32(0.259)
    // f32trAbcOut.f32Arg3 = 0x8B70E520 ~ FRAC32(-0.911)
    GMCLIB_BetaProjection3Ph_F32(&f32trAbcOut, &f32trAbcIn);

    // output should be
    // f32trAbcOut.f32Arg1 = 0x5A82799A ~ FRAC32(0.707)
    // f32trAbcOut.f32Arg2 = 0x2120FB83 ~ FRAC32(0.259)
    // f32trAbcOut.f32Arg3 = 0x8B70E520 ~ FRAC32(-0.911)
    GMCLIB_BetaProjection3Ph(&f32trAbcOut, &f32trAbcIn, F32);

    // #####
    // Available only if 32-bit fractional implementation selected
    // as default
    // #####
}
```

```

// output should be
// f32trAbcOut.f32Arg1 = 0x5A82799A ~ FRAC32(0.707)
// f32trAbcOut.f32Arg2 = 0x2120FB83 ~ FRAC32(0.259)
// f32trAbcOut.f32Arg3 = 0x8B70E520 ~ FRAC32(-0.911)
GMCLIB_BetaProjection3Ph(&f32trAbcOut, &f32trAbcIn);
}

```

## 2.42.2 Function GMCLIB\_BetaProjection3Ph\_F16

### Declaration

```
void GMCLIB_BetaProjection3Ph_F16(SWLIBS\_3Syst\_F16 *const pOut,
const SWLIBS\_3Syst\_F16 *const pIn);
```

### Arguments

**Table 256. GMCLIB\_BetaProjection3Ph\_F16 arguments**

Type	Name	Direction	Description
const <a href="#">SWLIBS_3Syst_F16</a> *const	pIn	input	Pointer to the structure containing data of the input three-phase stationary system (f16A-f16B-f16C). Arguments of the structure contain fixed point 16-bit values.
<a href="#">SWLIBS_3Syst_F16</a> *const	pOut	output	Pointer to the structure containing data of the output three-phase stationary orthogonal system (f16Aout-f16Bout-f16Cout). Arguments of the structure contain fixed point 16-bit values.

**Note:** The inputs and the outputs are normalized to fit in the range [-1, 1).

### Code Example

```
#include "gmclib.h"

SWLIBS\_3Syst\_F16 f16trAbcIn;
SWLIBS\_3Syst\_F16 f16trAbcOut;

void main(void)
{
    // input phase A ~ sin(45) ~ 0.707106781
    // input phase B ~ sin(45 + 120) ~ 0.258819045
    // input phase C ~ sin(45 - 120) ~ -0.965925826
    f16trAbcIn.f16Arg1 = FRAC16(0.707106781);
    f16trAbcIn.f16Arg2 = FRAC16(0.258819045);
    f16trAbcIn.f16Arg3 = FRAC16(-0.965925826);
    // add DC offset - will be eliminated by the Beta-projection
    f16trAbcIn.f16Arg1 += FRAC16(0.1);
    f16trAbcIn.f16Arg2 += FRAC16(0.1);
    f16trAbcIn.f16Arg3 += FRAC16(0.1);

    // output should be
    // f16trAbcOut.f16Arg1 = 0x5A82 ~ FRAC16(0.707)
    // f16trAbcOut.f16Arg2 = 0x2121 ~ FRAC16(0.259)
    // f16trAbcOut.f16Arg3 = 0x8B71 ~ FRAC16(-0.911)
    GMCLIB_BetaProjection3Ph_F16(&f16trAbcOut, &f16trAbcIn);

    // output should be
}
```

```

// f16trAbcOut.f16Arg1 = 0x5A82 ~ FRAC16(0.707)
// f16trAbcOut.f16Arg2 = 0x2121 ~ FRAC16(0.259)
// f16trAbcOut.f16Arg3 = 0x8B71 ~ FRAC16(-0.911)
GMCLIB_BetaProjection3Ph(&f16trAbcOut,&f16trAbcIn,F16);

// ##### Available only if 16-bit fractional implementation selected
// as default
// #####
// output should be
// f16trAbcOut.f16Arg1 = 0x5A82 ~ FRAC16(0.707)
// f16trAbcOut.f16Arg2 = 0x2121 ~ FRAC16(0.259)
// f16trAbcOut.f16Arg3 = 0x8B71 ~ FRAC16(-0.911)
GMCLIB_BetaProjection3Ph(&f16trAbcOut,&f16trAbcIn);
}

```

### 2.42.3 Function GMCLIB\_BetaProjection3Ph\_FLT

#### Declaration

```
void GMCLIB_BetaProjection3Ph_FLT(SWLIBS\_3Syst\_FLT *const pOut,
const SWLIBS\_3Syst\_FLT *const pIn);
```

#### Arguments

**Table 257. GMCLIB\_BetaProjection3Ph\_FLT arguments**

Type	Name	Direction	Description
const <a href="#">SWLIBS_3Syst_FLT</a> *const	pIn	input	Pointer to the structure containing data of the input three-phase stationary system (fltA-fltB-fltC). Arguments of the structure contain single precision floating point values.
<a href="#">SWLIBS_3Syst_FLT</a> *const	pOut	output	Pointer to the structure containing data of the output three-phase stationary orthogonal system (fltAout-fltBout-fltCout). Arguments of the structure contain single precision floating point values.

**Note:** The function may raise floating-point exceptions (invalid operation, overflow, underflow, inexact, input denormal). The floating-point unit must be enabled in CPACR register to prevent NOCP UsageFault exception.

#### Code Example

```

#include "gmclib.h"

SWLIBS\_3Syst\_FLT flttrAbcIn;
SWLIBS\_3Syst\_FLT flttrAbcOut;

void main(void)
{
    // input phase A ~ sin(45) ~ 0.707106781
    // input phase B ~ sin(45 + 120) ~ 0.258819045
    // input phase C ~ sin(45 - 120) ~ -0.965925826
    flttrAbcIn.fltArg1 = (tFloat)0.707106781;
    flttrAbcIn.fltArg2 = (tFloat)0.258819045;
    flttrAbcIn.fltArg3 = (tFloat)-0.965925826;
    // add DC offset - will be eliminated by the Beta-projection
}
```

```

flttrAbcIn.fltArg1 += 0.1f;
flttrAbcIn.fltArg2 += 0.1f;
flttrAbcIn.fltArg3 += 0.1f;

// output should be
// flttrAbcOut.fltArg1 = 0.707106781;
// flttrAbcOut.fltArg2 = 0.258819045;
// flttrAbcOut.fltArg3 = -0.965925826;
GMCLIB_BetaProjection3Ph_FLT(&flttrAbcOut,&flttrAbcIn);

// output should be
// flttrAbcOut.fltArg1 = 0.707106781;
// flttrAbcOut.fltArg2 = 0.258819045;
// flttrAbcOut.fltArg3 = -0.965925826;
GMCLIB_BetaProjection3Ph(&flttrAbcOut,&flttrAbcIn,FLT);

// #####
// Available only if single precision floating point
// implementation selected as default
// #####
// #####
// output should be
// flttrAbcOut.fltArg1 = 0.70710677;
// flttrAbcOut.fltArg2 = 0.25881904;
// flttrAbcOut.fltArg3 = -0.96592587;
GMCLIB_BetaProjection3Ph(&flttrAbcOut,&flttrAbcIn);
}

```

## 2.43 Function GMCLIB\_Clark

The function implements the Clarke transformation.

## Description

The Clarke Transformation is used to transform values from the three-phase (A-B-C) coordinate system to the two-phase ( $\alpha$ - $\beta$ ) orthogonal coordinate system, according to the following equations:

$$i_\alpha = i_A$$

### Equation GMCLIB Clark Eq1

where it is assumed that the axis A (axis of the first phase) and the axis  $\alpha$  are in the same direction.

**Note:** The input/output pointers must contain valid addresses, otherwise a fault may occur (MemManage, BusFault, UsageFault, HardFault).

## Re-entrancy

The function is re-entrant.

### 2.43.1 Function GMCLIB\_Clark\_F32

#### Declaration

```
void GMCLIB_Clark_F32(SWLIBS\_2Syst\_F32 *const pOut, const
SWLIBS\_3Syst\_F32 *const pIn);
```

#### Arguments

**Table 258. GMCLIB\_Clark\_F32 arguments**

Type	Name	Direction	Description
const <a href="#">SWLIBS_3Syst_F32</a> *const	pIn	input	Pointer to the structure containing data of the three-phase stationary system (f32A-f32B-f32C). Arguments of the structure contain fixed point 32-bit values.
<a href="#">SWLIBS_2Syst_F32</a> *const	pOut	output	Pointer to the structure containing data of the two-phase stationary orthogonal system ( $\alpha$ - $\beta$ ). Arguments of the structure contain fixed point 32-bit values.

**Note:** The inputs and the outputs are normalized to fit in the range [-1, 1).

#### Code Example

```
#include "gmclib.h"

SWLIBS\_3Syst\_F32 f32trAbc;
SWLIBS\_2Syst\_F32 f32trAlBe;

void main(void)
{
    // input phase A ~ sin(45) ~ 0.707106781
    // input phase B ~ sin(45 + 120) ~ 0.258819045
    // input phase C ~ sin(45 - 120) ~ -0.965925826
    f32trAbc.f32Arg1 = FRAC32(0.707106781);
    f32trAbc.f32Arg2 = FRAC32(0.258819045);
    f32trAbc.f32Arg3 = FRAC32(-0.965925826);

    // output should be
    // f32trAlBe.f32Arg1 = 0x5A827999 ~ FRAC32(0.707106781)
    // f32trAlBe.f32Arg2 = 0x5A827999 ~ FRAC32(0.707106781)
    GMCLIB_Clark_F32(&f32trAlBe, &f32trAbc);

    // output should be
    // f32trAlBe.f32Arg1 = 0x5A827999 ~ FRAC32(0.707106781)
    // f32trAlBe.f32Arg2 = 0x5A827999 ~ FRAC32(0.707106781)
    GMCLIB_Clark(&f32trAlBe, &f32trAbc, F32);

    // #####
    // Available only if 32-bit fractional implementation selected
    // as default
    // #####
    // output should be
    // f32trAlBe.f32Arg1 = 0x5A827999 ~ FRAC32(0.707106781)
    // f32trAlBe.f32Arg2 = 0x5A827999 ~ FRAC32(0.707106781)
    GMCLIB_Clark(&f32trAlBe, &f32trAbc);
}
```

## 2.43.2 Function GMCLIB\_Clark\_F16

### Declaration

```
void GMCLIB_Clark_F16(SWLIBS\_2Syst\_F16 *const pOut, const
SWLIBS\_3Syst\_F16 *const pIn);
```

### Arguments

**Table 259. GMCLIB\_Clark\_F16 arguments**

Type	Name	Direction	Description
const <a href="#">SWLIBS_3Syst_F16</a> *const	pIn	input	Pointer to the structure containing data of the three-phase stationary system (f16A-f16B-f16C). Arguments of the structure contain fixed point 16-bit values.
<a href="#">SWLIBS_2Syst_F16</a> *const	pOut	output	Pointer to the structure containing data of the two-phase stationary orthogonal system ( $\alpha$ - $\beta$ ). Arguments of the structure contain fixed point 16-bit values.

**Note:** The inputs and the outputs are normalized to fit in the range [-1, 1).

### Code Example

```
#include "gmclib.h"

SWLIBS\_3Syst\_F16 f16trAbc;
SWLIBS\_2Syst\_F16 f16trAlBe;

void main(void)
{
    // input phase A ~ sin(45) ~ 0.707106781
    // input phase B ~ sin(45 + 120) ~ 0.258819045
    // input phase C ~ sin(45 - 120) ~ -0.965925826
    f16trAbc.f16Arg1 = FRAC16(0.707106781);
    f16trAbc.f16Arg2 = FRAC16(0.258819045);
    f16trAbc.f16Arg3 = FRAC16(-0.965925826);

    // output should be
    // f16trAlBe.f16Arg1 = 0x5A82 ~ FRAC16(0.707106781)
    // f16trAlBe.f16Arg2 = 0x5A82 ~ FRAC16(0.707106781)
    GMCLIB_Clark_F16(&f16trAlBe, &f16trAbc);

    // output should be
    // f16trAlBe.f16Arg1 = 0x5A82 ~ FRAC16(0.707106781)
    // f16trAlBe.f16Arg2 = 0x5A82 ~ FRAC16(0.707106781)
    GMCLIB_Clark(&f16trAlBe, &f16trAbc, F16);

    // #####
    // Available only if 16-bit fractional implementation selected
    // as default
    // #####
    // output should be
    // f16trAlBe.f16Arg1 = 0x5A82 ~ FRAC16(0.707106781)
    // f16trAlBe.f16Arg2 = 0x5A82 ~ FRAC16(0.707106781)
    GMCLIB_Clark(&f16trAlBe, &f16trAbc);
}
```

### 2.43.3 Function GMCLIB\_Clark\_FLT

#### Declaration

```
void GMCLIB_Clark_FLT(SWLIBS\_2Syst\_FLT *const pOut, const
SWLIBS\_3Syst\_FLT *const pIn);
```

#### Arguments

**Table 260. GMCLIB\_Clark\_FLT arguments**

Type	Name	Direction	Description
const <a href="#">SWLIBS_3Syst_FLT</a> *const	pIn	input	Pointer to the structure containing data of the three-phase stationary system (fitA-fitB-fitC). Arguments of the structure contain single precision floating point values.
<a href="#">SWLIBS_2Syst_FLT</a> *const	pOut	output	Pointer to the structure containing data of the two-phase stationary orthogonal system ( $\alpha$ - $\beta$ ). Arguments of the structure contain single precision floating point values.

**Note:** The function may raise floating-point exceptions (invalid operation, overflow, underflow, inexact, input denormal). The floating-point unit must be enabled in CPACR register to prevent NOCP UsageFault exception.

#### Code Example

```
#include "gmclib.h"

SWLIBS\_3Syst\_FLT flttrAbc;
SWLIBS\_2Syst\_FLT flttrAlBe;

void main(void)
{
    // input phase A ~ sin(45) ~ 0.707106781
    // input phase B ~ sin(45 + 120) ~ 0.258819045
    // input phase C ~ sin(45 - 120) ~ -0.965925826
    flttrAbc.fltArg1 = (tFloat)0.707106781;
    flttrAbc.fltArg2 = (tFloat)0.258819045;
    flttrAbc.fltArg3 = (tFloat)-0.965925826;

    // output should be
    // flttrAlBe.fltArg1 = 0.707106781
    // flttrAlBe.fltArg2 = 0.707106781
    GMCLIB_Clark_FLT(&flttrAlBe,&flttrAbc);

    // output should be
    // flttrAlBe.fltArg1 = 0.707106781
    // flttrAlBe.fltArg2 = 0.707106781
    GMCLIB_Clark(&flttrAlBe,&flttrAbc,FLT);

    // #####
    // Available only if single precision floating point
    // implementation selected as default
    // #####
}

// output should be
// flttrAlBe.fltArg1 = 0.707106781
// flttrAlBe.fltArg2 = 0.707106781
GMCLIB_Clark(&flttrAlBe,&flttrAbc);
```

```
}
```

## 2.44 Function GMCLIB\_ClarkInv

The function implements the inverse Clarke transformation.

### Description

The GMCLIB\_ClarkInv function calculates the Inverse Clarke transformation, which is used to transform values from the two-phase ( $\alpha$ - $\beta$ ) orthogonal coordinate system to the three-phase (A-B-C) coordinate system, according to these equations:

$$\begin{aligned} i_A &= i_\alpha \\ i_B &= -\frac{1}{2} \cdot i_\alpha + \frac{\sqrt{3}}{2} \cdot i_\beta \\ i_C &= -\frac{1}{2} \cdot i_\alpha - \frac{\sqrt{3}}{2} \cdot i_\beta \end{aligned}$$

Equation GMCLIB\_ClarkInv\_Eq1

**Note:** The input/output pointers must contain valid addresses, otherwise a fault may occur (MemManage, BusFault, UsageFault, HardFault).

### Re-entrancy

The function is re-entrant.

### 2.44.1 Function GMCLIB\_ClarkInv\_F32

#### Declaration

```
void GMCLIB_ClarkInv_F32(SWLIBS\_3Syst\_F32 *const pOut, const
SWLIBS\_2Syst\_F32 *const pIn);
```

#### Arguments

Table 261. GMCLIB\_ClarkInv\_F32 arguments

Type	Name	Direction	Description
const <a href="#">SWLIBS_2Syst_F32</a> *const	pIn	input	Pointer to the structure containing data of the two-phase stationary orthogonal system ( $\alpha$ - $\beta$ ). Arguments of the structure contain fixed point 32-bit values.
<a href="#">SWLIBS_3Syst_F32</a> *const	pOut	output	Pointer to the structure containing data of the three-phase stationary system (f32A-f32B-f32C). Arguments of the structure contain fixed point 32-bit values.

**Note:** The inputs and the outputs are normalized to fit in the range [-1, 1].

#### Code Example

```
#include "gmclib.h"

SWLIBS\_2Syst\_F32 f32trAlBe;
SWLIBS\_3Syst\_F32 f32trAbc;
```

```

void main(void)
{
    // input phase alpha ~ sin(45) ~ 0.707106781
    // input phase beta ~ cos(45) ~ 0.707106781
    f32trAlBe.f32Arg1 = FRAC32(0.707106781);
    f32trAlBe.f32Arg2 = FRAC32(0.707106781);

    // output should be
    // f32trAbc.f32Arg1 = 0x5A827999 ~ FRAC32(0.707106781)
    // f32trAbc.f32Arg2 = 0x2120FB83 ~ FRAC32(0.258819045)
    // f32trAbc.f32Arg3 = 0x845C8AE5 ~ FRAC32(-0.965925826)
    GMCLIB_ClarkInv_F32(&f32trAbc, &f32trAlBe);

    // output should be
    // f32trAbc.f32Arg1 = 0x5A827999 ~ FRAC32(0.707106781)
    // f32trAbc.f32Arg2 = 0x2120FB83 ~ FRAC32(0.258819045)
    // f32trAbc.f32Arg3 = 0x845C8AE5 ~ FRAC32(-0.965925826)
    GMCLIB_ClarkInv(&f32trAbc, &f32trAlBe, F32);

    // #####
    // Available only if 32-bit fractional implementation selected
    // as default
    // #####
    // output should be
    // f32trAbc.f32Arg1 = 0x5A827999 ~ FRAC32(0.707106781)
    // f32trAbc.f32Arg2 = 0x2120FB83 ~ FRAC32(0.258819045)
    // f32trAbc.f32Arg3 = 0x845C8AE5 ~ FRAC32(-0.965925826)
    GMCLIB_ClarkInv(&f32trAbc, &f32trAlBe);
}

```

## 2.44.2 Function GMCLIB\_ClarkInv\_F16

### Declaration

```
void GMCLIB_ClarkInv_F16(SWLIBS\_3Syst\_F16 *const pOut, const
SWLIBS\_2Syst\_F16 *const pIn);
```

### Arguments

**Table 262. GMCLIB\_ClarkInv\_F16 arguments**

Type	Name	Direction	Description
const <a href="#">SWLIBS_2Syst_F16</a> *const	pIn	input	Pointer to the structure containing data of the two-phase stationary orthogonal system ( $\alpha$ - $\beta$ ). Arguments of the structure contain fixed point 16-bit values.
<a href="#">SWLIBS_3Syst_F16</a> *const	pOut	output	Pointer to the structure containing data of the three-phase stationary system (f16A-f16B-f16C). Arguments of the structure contain fixed point 16-bit values.

**Note:** The inputs and the outputs are normalized to fit in the range [-1, 1].

### Code Example

```
#include "gmclib.h"

SWLIBS\_2Syst\_F16 f16trAlBe;
```

```

SWLIBS_3Syst_F16 f16trAbc;

void main(void)
{
    // input phase alpha ~ sin(45) ~ 0.707106781
    // input phase beta ~ cos(45) ~ 0.707106781
    f16trAlBe.f16Arg1 = FRAC16(0.707106781);
    f16trAlBe.f16Arg2 = FRAC16(0.707106781);

    // output should be
    // f16trAbc.f16Arg1 = 0x5A82 ~ FRAC16(0.707106781)
    // f16trAbc.f16Arg2 = 0x2120 ~ FRAC16(0.258819045)
    // f16trAbc.f16Arg3 = 0x845C ~ FRAC16(-0.965925826)
    GMCLIB_ClarkInv_F16(&f16trAbc, &f16trAlBe);

    // output should be
    // f16trAbc.f16Arg1 = 0x5A82 ~ FRAC16(0.707106781)
    // f16trAbc.f16Arg2 = 0x2120 ~ FRAC16(0.258819045)
    // f16trAbc.f16Arg3 = 0x845C ~ FRAC16(-0.965925826)
    GMCLIB_ClarkInv(&f16trAbc, &f16trAlBe, F16);

    // ##### Available only if 16-bit fractional implementation selected
    // as default
    // #####
    // output should be
    // f16trAbc.f16Arg1 = 0x5A82 ~ FRAC16(0.707106781)
    // f16trAbc.f16Arg2 = 0x2120 ~ FRAC16(0.258819045)
    // f16trAbc.f16Arg3 = 0x845C ~ FRAC16(-0.965925826)
    GMCLIB_ClarkInv(&f16trAbc, &f16trAlBe);
}


```

### 2.44.3 Function GMCLIB\_ClarkInv\_FLT

#### Declaration

```

void GMCLIB_ClarkInv_FLT(SWLIBS_3Syst_FLT *const pOut, const
                           SWLIBS_2Syst_FLT *const pIn);

```

#### Arguments

**Table 263. GMCLIB\_ClarkInv\_FLT arguments**

Type	Name	Direction	Description
const SWLIBS_2Syst_FLT *const	pIn	input	Pointer to the structure containing data of the two-phase stationary orthogonal system ( $\alpha$ - $\beta$ ). Arguments of the structure contain single precision floating point values.
SWLIBS_3Syst_FLT *const	pOut	output	Pointer to the structure containing data of the three-phase stationary system (fitA-fitB-fitC). Arguments of the structure contain single precision floating point values.

**Note:** The function may raise floating-point exceptions (invalid operation, overflow, underflow, inexact, input denormal). The floating-point unit must be enabled in CPACR register to prevent NOCP UsageFault exception.

### Code Example

```
#include "gmclib.h"

SWLIBS_2Syst_FLT flttrAlBe;
SWLIBS_3Syst_FLT flttrAbc;

void main(void)
{
    // input phase alpha ~ sin(45) ~ 0.707106781
    // input phase beta ~ cos(45) ~ 0.707106781
    flttrAlBe.fltArg1 = (tFloat)0.707106781;
    flttrAlBe.fltArg2 = (tFloat)0.707106781;

    // output should be
    // flttrAbc.fltArg1 = 0.707106781
    // flttrAbc.fltArg2 = 0.258819045
    // flttrAbc.fltArg3 = -0.965925826
    GMCLIB_ClarkInv_FLT(&flttrAbc,&flttrAlBe);

    // output should be
    // flttrAbc.fltArg1 = 0.707106781
    // flttrAbc.fltArg2 = 0.258819045
    // flttrAbc.fltArg3 = -0.965925826
    GMCLIB_ClarkInv(&flttrAbc,&flttrAlBe,FLT);

    // ######
    // Available only if single precision floating point
    // implementation selected as default
    // #####
    // output should be
    // flttrAbc.fltArg1 ~ phA = 0.707106781
    // flttrAbc.fltArg2 ~ phB = 0.258819045
    // flttrAbc.fltArg3 ~ phC = -0.965925826
    GMCLIB_ClarkInv(&flttrAbc,&flttrAlBe);
}
```

## 2.45 Function GMCLIB\_DecouplingPMSM

This function calculates the cross-coupling voltages to eliminate the dq axis coupling causing on-linearity of the field oriented control.

### Description

The quadrature phase model of a PMSM motor, in a synchronous reference frame, is very popular for field oriented control structures because both controllable quantities, current and voltage, are DC values. This allows employing only simple controllers to force the machine currents into the defined states.

The voltage equations of this model can be obtained by transforming the motor three phase voltage equations into a quadrature phase rotational frame, which is aligned and rotates synchronously with the rotor. Such a transformation, after some mathematical corrections, yields the following set of equations, describing the quadrature phase model of a PMSM motor, in a synchronous reference frame:

$$\begin{bmatrix} u_d \\ u_q \end{bmatrix} = R_S \cdot \begin{bmatrix} i_d \\ i_q \end{bmatrix} + \begin{bmatrix} L_d & 0 \\ 0 & L_q \end{bmatrix} \frac{d}{dt} \begin{bmatrix} i_d \\ i_q \end{bmatrix} + \omega_e \begin{bmatrix} -L_q & 0 \\ 0 & L_d \end{bmatrix} \begin{bmatrix} i_q \\ i_d \end{bmatrix} + \omega_e \psi_{pm} \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

Equation GMCLIB\_DecouplingPMSM\_Eq1

It can be seen that [GMCLIB\\_DecouplingPMSM\\_Eq1](#) represents a non-linear cross dependent system. The linear voltage components cover the model of the phase winding, which is simplified to a resistance in series with inductance (R-L circuit). The cross-coupling components represent the mutual coupling between the two phases of the quadrature phase model, and the back-EMF component (visible only in q-axis voltage) represents the generated back-EMF voltage caused by rotor rotation.

In order to achieve dynamic torque, speed and positional control, the non-linear and back-EMF components from [GMCLIB\\_DecouplingPMSM\\_Eq1](#) must be compensated for. This will result in a fully decoupled flux and torque control of the machine and simplifies the PMSM motor model into two independent R-L circuit models as follows:

$$u_d = R_S i_d + L_d \frac{di_d}{dt}$$

$$u_q = R_S i_q + L_q \frac{di_q}{dt}$$

Equation GMCLIB\_DecouplingPMSM\_Eq2

Such a simplification of the PMSM model also greatly simplifies the design of both the d-q current controllers.

Therefore, it is advantageous to compensate for the cross-coupling terms in [GMCLIB\\_DecouplingPMSM\\_Eq1](#), using the feed-forward voltages  $u_{dq\_comp}$  given from [GMCLIB\\_DecouplingPMSM\\_Eq1](#) as follows:

$$u_{d_{comp}} = -\omega_e L_q i_q$$

$$u_{q_{comp}} = \omega_e L_d i_d$$

Equation GMCLIB\_DecouplingPMSM\_Eq3

The feed-forward voltages  $u_{dq\_comp}$  are added to the voltages generated by the current controllers  $u_{dq}$ , which cover the R-L model. The resulting voltages represent the direct  $u_{dq\_dec}$  and quadrature  $u_{q\_dec}$  components of the decoupled voltage vector that is to be applied on the motor terminals (using a pulse width modulator). The back-EMF voltage component is already considered to be compensated by an external function.

The function [GMCLIB\\_DecouplingPMSM](#) calculates the cross-coupling voltages  $u_{dq\_comp}$  and adds these to the input  $u_{dq}$  voltage vector. Because the back-EMF voltage component is considered compensated, this component is equal to zero. Therefore, calculations performed by [GMCLIB\\_DecouplingPMSM](#) are derived from these two equations:

$$u_{d_{dec}} = u_d + u_{d_{comp}}$$

$$u_{q_{dec}} = u_q + u_{q_{comp}}$$

Equation GMCLIB\_DecouplingPMSM\_Eq4

where  $u_{dq}$  is the voltage vector calculated by the controllers (with the already compensated back-EMF component),  $u_{dq\_comp}$  is the feed-forward compensating voltage vector described in [GMCLIB\\_DecouplingPMSM\\_Eq3](#), and  $u_{dq\_dec}$  is the resulting decoupled voltage vector to be applied on the motor terminals.

**Note:** The input/output pointers must contain valid addresses, otherwise a fault may occur (MemManage, BusFault, UsageFault, HardFault).

### Re-entrancy

The function is re-entrant.

#### 2.45.1 Function GMCLIB\_DcouplingPMSM\_F32

##### Declaration

```
void GMCLIB_DcouplingPMSM_F32 (SWLIBS\_2Syst\_F32 *const pUdqDec,
const SWLIBS\_2Syst\_F32 *const pUdq, const SWLIBS\_2Syst\_F32 *const
pIdq, tFrac32 f32AngularVel, const GMCLIB\_DECOUPLINGPMSM\_T\_F32
*const pParam);
```

##### Arguments

**Table 264. GMCLIB\_DcouplingPMSM\_F32 arguments**

Type	Name	Direction	Description
<a href="#">SWLIBS_2Syst_F32</a> *const	pUdqDec	output	Pointer to the structure containing direct ( $u_{df\_dec}$ ) and quadrature ( $u_{qf\_dec}$ ) components of the decoupled stator voltage vector to be applied on the motor terminals.
const <a href="#">SWLIBS_2Syst_F32</a> *const	pUdq	input	Pointer to the structure containing direct ( $u_{df}$ ) and quadrature ( $u_{qf}$ ) components of the stator voltage vector generated by the current controllers.
const <a href="#">SWLIBS_2Syst_F32</a> *const	pIdq	input	Pointer to the structure containing direct ( $i_{df}$ ) and quadrature ( $i_{qf}$ ) components of the stator current vector measured on the motor terminals.
<a href="#">tFrac32</a>	f32AngularVel	input	Rotor angular velocity in rad/sec, referred to as ( $\omega_{ef}$ ) in the detailed section of the documentation.
const <a href="#">GMCLIB_DECOUPLINGPMSM_T_F32</a> *const	pParam	input	Pointer to the structure containing $k_{df}$ and $k_{qf}$ coefficients (see the detailed section of the documentation) and scale parameters ( $k_{d\_shift}$ ) and ( $k_{q\_shift}$ ).

##### Implementation details

Substituting [GMCLIB\\_DcouplingPMSM\\_eq3](#) into [GMCLIB\\_DcouplingPMSM\\_eq4](#), and normalizing [GMCLIB\\_DcouplingPMSM\\_eq4](#), results in the following set of equations:

$$\begin{aligned} u_{df_{dec}} \cdot U_{max} &= u_{df} \cdot U_{max} - \omega_{ef} \cdot \Omega_{max} \cdot L_q \cdot i_{qf} \cdot I_{max} \\ u_{qf_{dec}} \cdot U_{max} &= u_{qf} \cdot U_{max} + \omega_{ef} \cdot \Omega_{max} \cdot L_d \cdot i_{df} \cdot I_{max} \end{aligned}$$

Equation GMCLIB\_DcouplingPMSM\_F32\_Eq1

where subscript f denotes the fractional representation of the respective quantity, and  $U_{max}$ ,  $I_{max}$ ,  $\Omega_{max}$  are the maximal values (scale values) for the voltage, current and angular velocity respectively.

Real quantities are converted to the fractional range [-1, 1] using the following equations:

$$\begin{aligned} u_{df_{dec}} &= \frac{u_{df_{dec}}}{U_{max}}, & u_{qf_{dec}} &= \frac{u_{qf_{dec}}}{U_{max}}, & u_{df} &= \frac{u_d}{U_{max}}, & u_{qf} &= \frac{u_q}{U_{max}}, \\ i_{df} &= \frac{i_d}{I_{max}}, & i_{qf} &= \frac{i_q}{I_{max}}, & \omega_{ef} &= \frac{\omega_e}{\Omega_{max}} \end{aligned}$$

Equation GMCLIB\_DcouplingPMSM\_F32\_Eq2

Further, rearranging [GMCLIB\\_DcouplingPMSM\\_F32\\_Eq1](#) results in:

$$\begin{aligned} u_{df_{dec}} &= u_{df} - \omega_{ef} \cdot i_{qf} \cdot \frac{\Omega_{max} L_q I_{max}}{U_{max}} = u_{df} - \omega_{ef} \cdot i_{qf} \cdot k_q \\ u_{qf_{dec}} &= u_{qf} + \omega_{ef} \cdot i_{df} \cdot \frac{\Omega_{max} L_d I_{max}}{U_{max}} = u_{qf} + \omega_{ef} \cdot i_{df} \cdot k_d \end{aligned}$$

Equation GMCLIB\_DcouplingPMSM\_F32\_Eq3

where  $k_d$  and  $k_q$  are coefficients calculated as:

$$\begin{aligned} k_d &= L_d \cdot \Omega_{max} \cdot \frac{I_{max}}{U_{max}} \\ k_q &= L_q \cdot \Omega_{max} \cdot \frac{I_{max}}{U_{max}} \end{aligned}$$

Equation GMCLIB\_DcouplingPMSM\_F32\_Eq4

Because function GMCLIB\_DcouplingPMSM\_F32 is implemented using the fractional arithmetic, both the  $k_d$  and  $k_q$  coefficients also have to be scaled to fit into the fractional range [-1, 1). For that purpose, two additional scaling coefficients are defined as:

$$\begin{aligned} k_{d\_shift} &= ceil\left(\frac{\log(k_d)}{\log(2)}\right) \\ k_{q\_shift} &= ceil\left(\frac{\log(k_q)}{\log(2)}\right) \end{aligned}$$

Equation GMCLIB\_DcouplingPMSM\_F32\_Eq5

Using scaling coefficients [GMCLIB\\_DcouplingPMSM\\_F32\\_Eq5](#), the fractional representation of coefficients  $k_d$  and  $k_q$  from [GMCLIB\\_DcouplingPMSM\\_F32\\_Eq4](#) are derived as follows:

$$\begin{aligned} k_{df} &= k_d \cdot 2^{k_{d\_shift}} \\ k_{qf} &= k_q \cdot 2^{k_{q\_shift}} \end{aligned}$$

Equation GMCLIB\_DcouplingPMSM\_F32\_Eq6

Substituting [GMCLIB\\_DcouplingPMSM\\_F32\\_Eq4](#) - [GMCLIB\\_DcouplingPMSM\\_F32\\_Eq6](#) into [GMCLIB\\_DcouplingPMSM\\_F32\\_Eq3](#) results in the final form of the equation set, actually implemented in the GMCLIB\_DcouplingPMSM\_F32 function:

$$\begin{aligned} u_{df_{dec}} &= u_{df} - \omega_{ef} \cdot i_{qf} \cdot k_{qf} \cdot 2^{k_{q\_shift}} \\ u_{qf_{dec}} &= u_{qf} + \omega_{ef} \cdot i_{df} \cdot k_{df} \cdot 2^{k_{d\_shift}} \end{aligned}$$

Equation GMCLIB\_DcouplingPMSM\_F32\_Eq7

Scaling of both equations into the fractional range is done using a multiplication by  $2^{k_{d\_shift}}$ ,  $2^{k_{q\_shift}}$ , respectively. Therefore, it is implemented as a simple left shift with overflow protection.

**Note:** All parameters can be reset during declaration using the [GMCLIB\\_DECOUPLINGPMSM\\_DEFAULT\\_F32](#) macro.

### Code Example

```
#include "gmclib.h"

#define L_D      (50.0e-3)    // Ld inductance = 50mH
#define L_Q      (100.0e-3)   // Lq inductance = 100mH
#define U_MAX    (50.0)       // scale for voltage = 50V
#define I_MAX    (10.0)       // scale for current = 10A
#define W_MAX    (2000.0)     // scale for angular velocity = 2000rad/sec

GMCLIB\_DECOUPLINGPMSM\_T\_F32 f32trDec = GMCLIB\_DECOUPLINGPMSM\_DEFAULT\_F32;
SWLIBS\_2Syst\_F32 f32trUDQ;
SWLIBS\_2Syst\_F32 f32trIDQ;
SWLIBS\_2Syst\_F32 f32trUDecDQ;
tFrac32 f32We;

void main(void)
{
    // input values - scaling coefficients of given decoupling algorithm
    f32trDec.f32Kd = FRAC32(0.625);
    f32trDec.s16KdShift = (ts16) 6;
    f32trDec.f32Kq = FRAC32(0.625);
    f32trDec.s16KqShift = (ts16) 5;
    // d quantity of input voltage vector 5[V]
    f32trUDQ.f32Arg1 = FRAC32(5.0/U_MAX);
    // q quantity of input voltage vector 10[V]
    f32trUDQ.f32Arg2 = FRAC32(10.0/U_MAX);
    // d quantity of measured current vector 6[A]
    f32trIDQ.f32Arg1 = FRAC32(6.0/I_MAX);
    // q quantity of measured current vector 4[A]
    f32trIDQ.f32Arg2 = FRAC32(4.0/I_MAX);
    // rotor angular velocity
    f32We = FRAC32(100.0/W_MAX);

    // output should be
    // f32trUDecDQ.f32Arg1 ~ 0xA6666666 ~ FRAC32(-0.7)*50V ~=-35[V]
    // f32trUDecDQ.f32Arg2 ~ 0x66666666 ~ FRAC32(0.8)*50V ~=40[V]
    GMCLIB_DecouplingPMSM_F32(&f32trUDecDQ,&f32trUDQ,&f32trIDQ,f32We,
                               &f32trDec);

    // output should be
    // f32trUDecDQ.f32Arg1 ~ 0xA6666666 ~ FRAC32(-0.7)*50V ~=-35[V]
    // f32trUDecDQ.f32Arg2 ~ 0x66666666 ~ FRAC32(0.8)*50V ~=40[V]
    GMCLIB_DecouplingPMSM(&f32trUDecDQ,&f32trUDQ,&f32trIDQ,f32We,&f32trDec,
                          F32);

    // #####
    // Available only if 32-bit fractional implementation selected
    // as default
    // #####
    // output should be
    // f32trUDecDQ.f32Arg1 ~ 0xA6666666 ~ FRAC32(-0.7)*50V ~=-35[V]
    // f32trUDecDQ.f32Arg2 ~ 0x66666666 ~ FRAC32(0.8)*50V ~=40[V]
    GMCLIB_DecouplingPMSM(&f32trUDecDQ,&f32trUDQ,&f32trIDQ,f32We,&f32trDec);
```

{}

## 2.45.2 Function GMCLIB\_DecouplingPMSM\_F16

### Declaration

```
void GMCLIB_DecouplingPMSM_F16 (SWLIBS_2Syst_F16 *const pUdqDec,
const SWLIBS_2Syst_F16 *const pUdq, const SWLIBS_2Syst_F16 *const
pIdq, tFrac16 f16AngularVel, const GMCLIB_DECOUPLINGPMSM_T_F16
*const pParam);
```

### Arguments

**Table 265. GMCLIB\_DecouplingPMSM\_F16 arguments**

Type	Name	Direction	Description
<code>SWLIBS_2Syst_F16</code> <code>*const</code>	pUdqDec	<code>output</code>	Pointer to the structure containing direct ( $u_{df\_dec}$ ) and quadrature ( $u_{qf\_dec}$ ) components of the decoupled stator voltage vector to be applied on the motor terminals.
<code>const SWLIBS_2Syst_F16</code> <code>*const</code>	pUdq	<code>input</code>	Pointer to the structure containing direct ( $u_{df}$ ) and quadrature ( $u_{qf}$ ) components of the stator voltage vector generated by the current controllers.
<code>const SWLIBS_2Syst_F16</code> <code>*const</code>	pldq	<code>input</code>	Pointer to the structure containing direct ( $i_{df}$ ) and quadrature ( $i_{qf}$ ) components of the stator current vector measured on the motor terminals.
<code>tFrac16</code>	f16AngularVel	<code>input</code>	Rotor angular velocity in rad/sec, referred to as ( $\omega_{ef}$ ) in the detailed section of the documentation.
<code>const GMCLIB_DECOUPLINGPMSM_T_F16</code> <code>*const</code>	pParam	<code>input</code>	Pointer to the structure containing $k_{df}$ and $k_{qf}$ coefficients (see the detailed section of the documentation) and scale parameters ( $k_{d\_shift}$ ) and ( $k_{q\_shift}$ ).

### Implementation details

Substituting [GMCLIB\\_DecouplingPMSM\\_eq3](#) into [GMCLIB\\_DecouplingPMSM\\_eq4](#), and normalizing [GMCLIB\\_DecouplingPMSM\\_eq4](#), results in the following set of equations:

$$u_{df_{dec}} \cdot U_{max} = u_{df} \cdot U_{max} - \omega_{ef} \cdot \Omega_{max} \cdot L_q \cdot i_{qf} \cdot I_{max}$$

$$u_{qf_{dec}} \cdot U_{max} = u_{qf} \cdot U_{max} + \omega_{ef} \cdot \Omega_{max} \cdot L_d \cdot i_{df} \cdot I_{max}$$

Equation GMCLIB\_DecouplingPMSM\_F16\_Eq1

where subscript f denotes the fractional representation of the respective quantity, and  $U_{max}$ ,  $I_{max}$ ,  $\Omega_{max}$  are the maximal values (scale values) for the voltage, current and angular velocity respectively.

Real quantities are converted to the fractional range [-1, 1) using the following equations:

$$u_{df_{dec}} = \frac{u_{df}}{U_{max}}, \quad u_{qf_{dec}} = \frac{u_{qf}}{U_{max}}, \quad u_{df} = \frac{u_d}{U_{max}}, \quad u_{qf} = \frac{u_q}{U_{max}},$$

$$i_{df} = \frac{i_d}{I_{max}}, \quad i_{qf} = \frac{i_q}{I_{max}}, \quad \omega_{ef} = \frac{\omega_e}{\Omega_{max}}$$

Equation GMCLIB\_DecouplingPMSM\_F16\_Eq2

Further, rearranging [GMCLIB\\_DecouplingPMSM\\_F16\\_Eq1](#) results in:

$$u_{df_{dec}} = u_{df} - \omega_{ef} \cdot i_{qf} \cdot \frac{\Omega_{max} \cdot L_q \cdot I_{max}}{U_{max}} = u_{df} - \omega_{ef} \cdot i_{qf} \cdot k_q$$

$$u_{qf_{dec}} = u_{qf} + \omega_{ef} \cdot i_{df} \cdot \frac{\Omega_{max} \cdot L_d \cdot I_{max}}{U_{max}} = u_{qf} + \omega_{ef} \cdot i_{df} \cdot k_d$$

Equation GMCLIB\_DecouplingPMSM\_F16\_Eq3

where  $k_d$  and  $k_q$  are coefficients calculated as:

$$k_d = L_d \cdot \Omega_{max} \cdot \frac{I_{max}}{U_{max}}$$

$$k_q = L_q \cdot \Omega_{max} \cdot \frac{I_{max}}{U_{max}}$$

Equation GMCLIB\_DecouplingPMSM\_F16\_Eq4

Because function GMCLIB\_DecouplingPMSM\_F16 is implemented using the fractional arithmetic, both the  $k_d$  and  $k_q$  coefficients also have to be scaled to fit into the fractional range [-1, 1). For that purpose, two additional scaling coefficients are defined as:

$$k_{d\_shift} = ceil\left(\frac{\log(k_d)}{\log(2)}\right)$$

$$k_{q\_shift} = ceil\left(\frac{\log(k_q)}{\log(2)}\right)$$

Equation GMCLIB\_DecouplingPMSM\_F16\_Eq5

Using scaling coefficients [GMCLIB\\_DecouplingPMSM\\_F16\\_Eq5](#), the fractional representation of coefficients  $k_d$  and  $k_q$  from [GMCLIB\\_DecouplingPMSM\\_F16\\_Eq4](#) are derived as follows:

$$k_{df} = k_d \cdot 2^{k_{d\_shift}}$$

$$k_{qf} = k_q \cdot 2^{k_{q\_shift}}$$

Equation GMCLIB\_DecouplingPMSM\_F16\_Eq6

Substituting [GMCLIB\\_DecouplingPMSM\\_F16\\_Eq4](#) - [GMCLIB\\_DecouplingPMSM\\_F16\\_Eq6](#) into [GMCLIB\\_DecouplingPMSM\\_F16\\_Eq3](#) results in the final form of the equation set, actually implemented in the GMCLIB\_DecouplingPMSM\_F16 function:

$$u_{df_{dec}} = u_{df} - \omega_{ef} \cdot i_{qf} \cdot k_{qf} \cdot 2^{k_{q\_shift}}$$

$$u_{qf_{dec}} = u_{qf} + \omega_{ef} \cdot i_{df} \cdot k_{df} \cdot 2^{k_{d\_shift}}$$

Equation GMCLIB\_DecouplingPMSM\_F16\_Eq7

Scaling of both equations into the fractional range is done using a multiplication by  $2^{k_{d\_shift}}$ ,  $2^{k_{q\_shift}}$ , respectively. Therefore, it is implemented as a simple left shift with overflow protection.

**Note:** All parameters can be reset during declaration using the [GMCLIB\\_DECOUPLINGPMSM\\_DEFAULT\\_F16](#) macro.

### Code Example

```
#include "gmclib.h"
```

```

#define L_D      (50.0e-3)    // Ld inductance = 50mH
#define L_Q      (100.0e-3)   // Lq inductance = 100mH
#define U_MAX    (50.0)       // scale for voltage = 50V
#define I_MAX    (10.0)       // scale for current = 10A
#define W_MAX    (2000.0)     // scale for angular velocity = 2000rad/sec

GMCLIB_DECOUPLINGPMSM_T_F16 f16trDec = GMCLIB_DECOUPLINGPMSM_DEFAULT_F16;
SWLIBS_2Syst_F16 f16trUDQ;
SWLIBS_2Syst_F16 f16trIDQ;
SWLIBS_2Syst_F16 f16trUDecDQ;
tFrac16 f16We;

void main(void)
{
    // input values - scaling coefficients of given decoupling algorithm
    f16trDec.f16Kd = FRAC16(0.625);
    f16trDec.s16KdShift = (ts16)6;
    f16trDec.f16Kq = FRAC16(0.625);
    f16trDec.s16KqShift = (ts16)5;
    // d quantity of input voltage vector 5[V]
    f16trUDQ.f16Arg1 = FRAC16(5.0/U_MAX);
    // q quantity of input voltage vector 10[V]
    f16trUDQ.f16Arg2 = FRAC16(10.0/U_MAX);
    // d quantity of measured current vector 6[A]
    f16trIDQ.f16Arg1 = FRAC16(6.0/I_MAX);
    // q quantity of measured current vector 4[A]
    f16trIDQ.f16Arg2 = FRAC16(4.0/I_MAX);
    // rotor angular velocity
    f16We = FRAC16(100.0/W_MAX);

    // output should be
    // f16trUDecDQ.f16Arg1 ~ 0xA666 ~ FRAC16(-0.7)*50V == -35[V]
    // f16trUDecDQ.f16Arg2 ~ 0x6666 ~ FRAC16(0.8)*50V == 40[V]
    GMCLIB_DecouplingPMSM_F16(&f16trUDecDQ, &f16trUDQ, &f16trIDQ, f16We,
                               &f16trDec);

    // output should be
    // f16trUDecDQ.f16Arg1 ~ 0xA666 ~ FRAC16(-0.7)*50V == -35[V]
    // f16trUDecDQ.f16Arg2 ~ 0x6666 ~ FRAC16(0.8)*50V == 40[V]
    GMCLIB_DecouplingPMSM(&f16trUDecDQ, &f16trUDQ, &f16trIDQ, f16We, &f16trDec,
                          F16);

    // #####
    // Available only if 16-bit fractional implementation selected
    // as default
    // #####
}

// output should be
// f16trUDecDQ.f16Arg1 ~ 0xA666 ~ FRAC16(-0.7)*50V == -35[V]
// f16trUDecDQ.f16Arg2 ~ 0x6666 ~ FRAC16(0.8)*50V == 40[V]
GMCLIB_DecouplingPMSM(&f16trUDecDQ, &f16trUDQ, &f16trIDQ, f16We, &f16trDec);
}

```

### 2.45.3 Function GMCLIB\_DecouplingPMSM\_FLT

#### Declaration

```
void GMCLIB_DecouplingPMSM_FLT(SWLIBS_2Syst_FLT *const pUdqDec,
                                const SWLIBS_2Syst_FLT *const pUdq, const SWLIBS_2Syst_FLT *const
```

```
pIdq, tFloat fltAngularVel, const GMCLIB\_DECOUPLINGPMSM\_T\_FLT  
*const pParam);
```

### Arguments

Table 266. GMCLIB\_DcouplingPMSM\_FLT arguments

Type	Name	Direction	Description
<a href="#">SWLIBS_2Syst_FLT</a> *const	pUdqDec	<b>output</b>	Pointer to the structure containing direct ( $u_{df\_dec}$ ) and quadrature ( $u_{qf\_dec}$ ) components of the decoupled stator voltage vector to be applied on the motor terminals.
const <a href="#">SWLIBS_2Syst_FLT</a> *const	pUdq	<b>input</b>	Pointer to the structure containing direct ( $u_{df}$ ) and quadrature ( $u_{qf}$ ) components of the stator voltage vector generated by the current controllers.
const <a href="#">SWLIBS_2Syst_FLT</a> *const	pldq	<b>input</b>	Pointer to the structure containing direct ( $i_{df}$ ) and quadrature ( $i_{qf}$ ) components of the stator current vector measured on the motor terminals.
<a href="#">tFloat</a>	fltAngularVel	<b>input</b>	Rotor angular velocity in rad/sec, referred to as ( $\omega_{ef}$ ) in the detailed section of the documentation.
const <a href="#">GMCLIB_DECOUPLINGPMSM_T_FLT</a> *const	pParam	<b>input</b>	Pointer to the structure containing $L_D$ and $L_Q$ coefficients (see the detailed section of the documentation).

**Note:** The function may raise floating-point exceptions (invalid operation, overflow, underflow, inexact, input denormal). The floating-point unit must be enabled in CPACR register to prevent NOCP UsageFault exception.

All parameters can be reset during declaration using the [GMCLIB\\_DECOUPLINGPMSM\\_DEFAULT\\_FLT](#) macro. All inputs and parameters contain single precision floating point data type values.

### Code Example

```
#include "gmclib.h"

GMCLIB\_DECOUPLINGPMSM\_T\_FLT flttrDec = GMCLIB\_DECOUPLINGPMSM\_DEFAULT\_FLT;
SWLIBS\_2Syst\_FLT flttrUDQ;
SWLIBS\_2Syst\_FLT flttrIDQ;
SWLIBS\_2Syst\_FLT flttrUDecDQ;
tFloat fltWe;

void main(void)
{
    // input values
    flttrDec.fltLD = (tFloat) 50e-3; // LD inductance = 50mH
    flttrDec.fltLQ = (tFloat) 100e-3; // LQ inductance = 100mH
    // D quantity of input voltage vector 5[V]
    flttrUDQ.fltArg1 = (tFloat) 5.0;
    // Q quantity of input voltage vector 10[V]
    flttrUDQ.fltArg2 = (tFloat) 10.0;
    // D quantity of measured current vector 6[A]
    flttrIDQ.fltArg1 = (tFloat) 6.0;
    // Q quantity of measured current vector 4[A]
    flttrIDQ.fltArg2 = (tFloat) 4.0;
    fltWe = (tFloat) 100.0; // rotor angular velocity

    // output should be
}
```

```
// flttrUDecDQ.fltArg1 ~= -35[V]
// flttrUDecDQ.fltArg2 ~= 40[V]
GMCLIB_DecouplingPMSM_FLT(&flttrUDecDQ, &flttrUDQ, &flttrIDQ, fltWe,
                           &flttrDec);

// output should be
// flttrUDecDQ.fltArg1 ~= -35[V]
// flttrUDecDQ.fltArg2 ~= 40[V]
GMCLIB_DecouplingPMSM(&flttrUDecDQ, &flttrUDQ, &flttrIDQ, fltWe, &flttrDec,
                      FLT);

// ######
// Available only if single precision floating point
// implementation selected as default
// #####
// #####
// output should be
// flttrUDecDQ.fltArg1 ~= -35[V]
// flttrUDecDQ.fltArg2 ~= 40[V]
GMCLIB_DecouplingPMSM(&flttrUDecDQ, &flttrUDQ, &flttrIDQ, fltWe, &flttrDec);
}
```

## 2.46 Function GMCLIB\_ElimDcBusRip

This function implements the DC Bus voltage ripple elimination.

### Description

The GMCLIB\_ElimDcBusRip function provides a computational method for the recalculation of the direct ( $\alpha$ ) and quadrature ( $\beta$ ) components of the required stator voltage vector, so as to compensate for voltage ripples on the DC bus of the power stage.

Considering a cascaded type structure of the control system in a standard motor control application, the required voltage vector to be applied on motor terminals is generated by a set of controllers (usually P, PI or PID) only with knowledge of the maximal value of the DC bus voltage. The amplitude and phase of the required voltage vector are then used by the pulse width modulator (PWM) for generation of appropriate duty-cycles for the power inverter switches. The amplitude of the generated phase voltage (averaged across one switching period) does not only depend on the actual on/off times of the given phase switches and the maximal value of the DC bus voltage. The actual amplitude of the phase voltage is also directly affected by the actual value of the available DC bus voltage. Therefore, any variations in amplitude of the actual DC bus voltage must be accounted for by modifying the amplitude of the required voltage so that the output phase voltage remains unaffected.

**Note:** The input/output pointers must contain valid addresses, otherwise a fault may occur (MemManage, BusFault, UsageFault, HardFault).

### Re-entrancy

The function is re-entrant.

### 2.46.1 Function GMCLIB\_ElimDcBusRip\_F32

#### Declaration

```
void GMCLIB_ElimDcBusRip_F32(SWLIBS\_2Syst\_F32 *const pOut, const
SWLIBS\_2Syst\_F32 *const pIn, const GMCLIB\_ELIMDCBUSRIP\_T\_F32
*const pParam);
```

#### Arguments

**Table 267. GMCLIB\_ElimDcBusRip\_F32 arguments**

Type	Name	Direction	Description
<a href="#">SWLIBS_2Syst_F32</a> *const	pOut	<b>output</b>	Pointer to the structure with direct ( $\alpha$ ) and quadrature ( $\beta$ ) components of the required stator voltage vector recalculated so as to compensate for voltage ripples on the DC bus.
const <a href="#">SWLIBS_2Syst_F32</a> *const	pIn	<b>input</b>	Pointer to the structure with direct ( $\alpha$ ) and quadrature ( $\beta$ ) components of the required stator voltage vector before compensation of voltage ripples on the DC bus.
const <a href="#">GMCLIB_ELIMDCBUSRIP_T_F32</a> *const	pParam	<b>input</b>	Pointer to the parameters structure.

#### Return

Function returns no value.

#### Implementation details

For better understanding, let's consider the following two simple examples:

Example 1:

- amplitude of the required phase voltage  $U_{reg}=50[V]$
- maximal amplitude of the DC bus voltage  $U_{DC\_BUS\_MAX}=100[V]$
- actual amplitude of the DC bus voltage  $U_{DC\_BUS\_ACTUAL}=100[V]$
- voltage to be applied to the PWM modulator to generate  $U_{reg}=50[V]$  on the inverter phase output:

$$U_{req\_new} = \frac{U_{req} U_{DC\_BUS\_MAX}}{U_{DC\_BUS\_ACTUAL}} = 50V$$

Equation GMCLIB\_ElimDcBusRip\_F32\_Eq1

Example 2:

- amplitude of the required phase voltage  $U_{reg}=50[V]$
- maximal amplitude of the DC bus voltage  $U_{DC\_BUS\_MAX}=100[V]$
- actual amplitude of the DC bus voltage  $U_{DC\_BUS\_ACTUAL}=90[V]$
- voltage to be applied to the PWM modulator to generate  $U_{reg}=50[V]$  on the inverter phase output:

$$U_{req\_new} = \frac{U_{req} U_{DC\_BUS\_MAX}}{U_{DC\_BUS\_ACTUAL}} = 55.5V$$

Equation GMCLIB\_ElimDcBusRip\_F32\_Eq2

The imperfections of the DC bus voltage are compensated for by the modification of amplitudes of the direct-  $\alpha$  and the quadrature-  $\beta$  components of the stator reference voltage vector. The following formulas are used:

$$u_{\alpha}^* = \begin{cases} \frac{f32ModIndex \cdot u_{\alpha}}{f32ArgDCBusMsr/2} & \text{if } abs(f32ModIndex \cdot u_{\alpha}) < \frac{f32ArgDCBusMsr}{2} \\ sign(u_{\alpha}) & \text{otherwise} \end{cases}$$

$$u_{\beta}^* = \begin{cases} \frac{f32ModIndex \cdot u_{\beta}}{f32ArgDCBusMsr/2} & \text{if } abs(f32ModIndex \cdot u_{\beta}) < \frac{f32ArgDCBusMsr}{2} \\ sign(u_{\beta}) & \text{otherwise} \end{cases}$$

Equation GMCLIB\_ElimDcBusRip\_F32\_Eq3

where: f32ModIndex is the inverse modulation index, f32ArgDCBusMsr is the measured DC bus voltage, the  $u_{\alpha}$  and  $u_{\beta}$  are the input voltages, and the  $u_{\alpha}^*$  and  $u_{\beta}^*$  are the output duty-cycle ratios.

The f32ModIndex and f32ArgDCBusMsr are supplied to the function within the parameters structure through its members. The  $u_{\alpha}$ ,  $u_{\beta}$  correspond respectively to the f32Arg1 and f32Arg2 members of the input structure, and the  $u_{\alpha}^*$  and  $u_{\beta}^*$  respectively to the f32Arg1 and f32Arg2 members of the output structure.

The inverse modulation index (see the parameters structure pParam, the f32ModIndex member) must be equal to or greater than zero. Besides this restriction, the f32ModIndex must be set to a valid value resulting from the use of Space Vector Modulation techniques.

The function explicitly avoids the case of division by zero. If f32ArgDCBusMsr equals zero, the outputs will be saturated.

**Note:** Both the inverse modulation index  $pIn->f32ModIndex$  and the measured DC bus voltage  $pIn->f32DcBusMsr$  must be equal to or greater than 0, otherwise the results are undefined.

### Code Example

```
#include "gmclib.h"

#define U_MAX    (36.0) // voltage scale
SWLIBS_2Syst_F32 f32AB;
SWLIBS_2Syst_F32 f32OutAB;
GMCLIB_ELIMDCBUSRIP_T_F32 f32trMyElimDcBusRip =
    GMCLIB_ELIMDCBUSRIP_DEFAULT_F32;

void main(void)
{
    // inverse modulation coefficient for standard space vector modulation
    f32trMyElimDcBusRip.f32ModIndex = FRAC32(0.866025403784439);
    // Input voltage vector 15V @ angle 30deg
    // alpha component of input voltage vector = 12.99[V]
    f32AB.f32Arg1 = FRAC32(12.99/U_MAX);
    // beta component of input voltage vector = 7.5[V]
    f32AB.f32Arg2 = FRAC32(7.5/U_MAX);
    // value of the measured DC bus voltage 17[V]
    f32trMyElimDcBusRip.f32ArgDCBusMsr = FRAC32(17.0/U_MAX);
```

```

// output alpha component of the output vector should be
// f32OutAB.f32Arg1 = (12.99/36)*0.8660/(17.0/36/2) =
//           1.3235 -> FRAC32(1.0) ~ 0x7FFFFFFF
// output beta component of the output vector should be
// f32OutAB.f32Arg2 = (7.5/36)*0.8660/(17.0/36/2) =
//           0.7641 -> FRAC32(0.7641) ~ 0x61CF8000
GMCLIB_ElimDcBusRip_F32(&f32OutAB, &f32AB, &f32trMyElimDcBusRip);

// output alpha component of the output vector should be
// f32OutAB.f32Arg1 = (12.99/36)*0.8660/(17.0/36/2) =
//           1.3235 -> FRAC32(1.0) ~ 0x7FFFFFFF
// output beta component of the output vector should be
// f32OutAB.f32Arg2 = (7.5/36)*0.8660/(17.0/36/2) =
//           0.7641 -> FRAC32(0.7641) ~ 0x61CF8000
GMCLIB_ElimDcBusRip(&f32OutAB, &f32AB, &f32trMyElimDcBusRip, F32);

// ######
// Available only if 32-bit fractional implementation selected
// as default
// #####
// output alpha component of the output vector should be
// f32OutAB.f32Arg1 = (12.99/36)*0.8660/(17.0/36/2) =
//           1.3235 -> FRAC32(1.0) ~ 0x7FFFFFFF
// output beta component of the output vector should be
// f32OutAB.f32Arg2 = (7.5/36)*0.8660/(17.0/36/2) =
//           0.7641 -> FRAC32(0.7641) ~ 0x61CF8000
GMCLIB_ElimDcBusRip(&f32OutAB, &f32AB, &f32trMyElimDcBusRip);
}

```

## 2.46.2 Function GMCLIB\_ElimDcBusRip\_F16

### Declaration

```
void GMCLIB_ElimDcBusRip_F16(SWLIBS\_2Syst\_F16 *const pOut, const
SWLIBS\_2Syst\_F16 *const pIn, const GMCLIB\_ELIMDCBUSRIP\_T\_F16
*const pParam);
```

### Arguments

**Table 268. GMCLIB\_ElimDcBusRip\_F16 arguments**

Type	Name	Direction	Description
<a href="#">SWLIBS_2Syst_F16</a> *const	pOut	<b>output</b>	Pointer to the structure with direct ( $\alpha$ ) and quadrature ( $\beta$ ) components of the required stator voltage vector recalculated so as to compensate for voltage ripples on the DC bus.
const <a href="#">SWLIBS_2Syst_F16</a> *const	pIn	<b>input</b>	Pointer to the structure with direct ( $\alpha$ ) and quadrature ( $\beta$ ) components of the required stator voltage vector before compensation of voltage ripples on the DC bus.
const <a href="#">GMCLIB_ELIMDCBUSRIP_T_F16</a> *const	pParam	<b>input</b>	Pointer to the parameters structure.

### Return

Function returns no value.

**Implementation details**

For better understanding, let's consider the following two simple examples:

Example 1:

- amplitude of the required phase voltage  $U_{reg}=50[V]$
- maximal amplitude of the DC bus voltage  $U_{DC\_BUS\_MAX}=100[V]$
- actual amplitude of the DC bus voltage  $U_{DC\_BUS\_ACTUAL}=100[V]$
- voltage to be applied to the PWM modulator to generate  $U_{reg}=50[V]$  on the inverter phase output:

$$U_{req\_new} = \frac{U_{req}U_{DC\_BUS\_MAX}}{U_{DC\_BUS\_ACTUAL}} = 50V$$

Equation GMCLIB\_ElimDcBusRip\_F16\_Eq1

- amplitude of the required phase voltage  $U_{reg}=50[V]$
- maximal amplitude of the DC bus voltage  $U_{DC\_BUS\_MAX}=100[V]$
- actual amplitude of the DC bus voltage  $U_{DC\_BUS\_ACTUAL}=90[V]$
- voltage to be applied to the PWM modulator to generate  $U_{reg}=50[V]$  on the inverter phase output:

$$U_{req\_new} = \frac{U_{req}U_{DC\_BUS\_MAX}}{U_{DC\_BUS\_ACTUAL}} = 55.5V$$

Equation GMCLIB\_ElimDcBusRip\_F16\_Eq2

$$u_{\alpha}^* = \begin{cases} \frac{f16ModIndexu_{\alpha}}{f16ArgDCBusMsr/2} & \text{if } abs(f16ModIndex \cdot u_{\alpha}) < \frac{f16ArgDCBusMsr}{2} \\ sign(u_{\alpha}) & \text{otherwise} \end{cases}$$

$$u_{\beta}^* = \begin{cases} \frac{f16ModIndexu_{\beta}}{f16ArgDCBusMsr/2} & \text{if } abs(f16ModIndex \cdot u_{\beta}) < \frac{f16ArgDCBusMsr}{2} \\ sign(u_{\beta}) & \text{otherwise} \end{cases}$$

Equation GMCLIB\_ElimDcBusRip\_F16\_Eq3

where: f16ModIndex is the inverse modulation index, f16ArgDcBusMsr is the measured DC bus voltage, the  $u_{\alpha}$  and  $u_{\beta}$  are the input voltages, and the  $u_{\alpha}^*$  and  $u_{\beta}^*$  are the output duty-cycle ratios. The f16ModIndex and f16ArgDcBusMsr are supplied to the function within the parameters structure through its members. The  $u_{\alpha}$ ,  $u_{\beta}$  correspond respectively to the f16Arg1 and f16Arg2 members of the input structure, and the  $u_{\alpha}^*$  and  $u_{\beta}^*$  respectively to the f16Arg1 and f16Arg2 members of the output structure. The inverse modulation index (see the parameters structure pParam, the f16ModIndex member) must be equal to or greater than zero. Besides this restriction, the f16ModIndex must be set to a valid value resulting from the use of Space Vector Modulation techniques. The function explicitly avoids the case of division by zero. If f16ArgDcBusMsr equals zero, the outputs will be saturated.

**Note:** Both the inverse modulation index  $pIn->f16ModIndex$  and the measured DC bus voltage  $pIn->f16DcBusMsr$  must be equal to or greater than 0, otherwise the results are undefined.

**Code Example**

```

#include "gmclib.h"

#define U_MAX    (36.0) // voltage scale
_SWLIBS_2Syst_F16 f16AB;
_SWLIBS_2Syst_F16 f16OutAB;
GMCLIB_ELIMDCBUSRIP_T_F16 f16trMyElimDcBusRip =
    GMCLIB_ELIMDCBUSRIP_DEFAULT_F16;

void main(void)
{
    // inverse modulation coefficient for standard space vector modulation
    f16trMyElimDcBusRip.f16ModIndex = FRAC16(0.866025403784439);
    // Input voltage vector 15V @ angle 30deg
    // alpha component of input voltage vector = 12.99[V]
    f16AB.f16Arg1 = FRAC16(12.99/U_MAX);
    // beta component of input voltage vector = 7.5[V]
    f16AB.f16Arg2 = FRAC16(7.5/U_MAX);
    // value of the measured DC bus voltage 17[V]
    f16trMyElimDcBusRip.f16ArgDcBusMsr = FRAC16(17.0/U_MAX);

    // output alpha component of the output vector should be
    // f16OutAB.f16Arg1 = (12.99/36)*0.8660/(17.0/36/2) =
    //                      1.3235 -> FRAC16(1.0) ~ 0x7FFF
    // output beta component of the output vector should be
    // f16OutAB.f16Arg2 = (7.5/36)*0.8660/(17.0/36/2) =
    //                      0.7641 -> FRAC16(0.7641) ~ 0x61CF
    GMCLIB_ElimDcBusRip_F16(&f16OutAB,&f16AB,&f16trMyElimDcBusRip);

    // output alpha component of the output vector should be
    // f16OutAB.f16Arg1 = (12.99/36)*0.8660/(17.0/36/2) =
    //                      1.3235 -> FRAC16(1.0) ~ 0x7FFF
    // output beta component of the output vector should be
    // f16OutAB.f16Arg2 = (7.5/36)*0.8660/(17.0/36/2) =
    //                      0.7641 -> FRAC16(0.7641) ~ 0x61CF
    GMCLIB_ElimDcBusRip(&f16OutAB,&f16AB,&f16trMyElimDcBusRip,F16);

    // ######
    // Available only if 16-bit fractional implementation selected
    // as default
    // #####
}

// output alpha component of the output vector should be
// f16OutAB.f16Arg1 = (12.99/36)*0.8660/(17.0/36/2) =
//                      1.3235 -> FRAC16(1.0) ~ 0x7FFF
// output beta component of the output vector should be
// f16OutAB.f16Arg2 = (7.5/36)*0.8660/(17.0/36/2) =
//                      0.7641 -> FRAC16(0.7641) ~ 0x61CF
GMCLIB_ElimDcBusRip(&f16OutAB,&f16AB,&f16trMyElimDcBusRip);
}

```

### 2.46.3 Function GMCLIB\_ElimDcBusRip\_FLT

#### Declaration

```

void GMCLIB_ElimDcBusRip_FLT(_SWLIBS_2Syst_FLT *const pOut, const
_SWLIBS_2Syst_FLT *const pIn, const GMCLIB_ELIMDCBUSRIP_T_FLT
*const pParam);

```

**Arguments****Table 269. GMCLIB\_ElimDcBusRip\_FLT arguments**

Type	Name	Direction	Description
<a href="#">SWLIBS_2Syst_FLT</a> *const	pOut	output	Pointer to the structure with direct ( $\alpha$ ) and quadrature ( $\beta$ ) components of the required stator voltage vector re-calculated so as to compensate for voltage ripples on the DC bus. Outputs are normalized to the range [-1; 1].
const <a href="#">SWLIBS_2Syst_FLT</a> *const	pln	input	Pointer to the structure with direct ( $\alpha$ ) and quadrature ( $\beta$ ) components of the required stator voltage vector before compensation of voltage ripples on the DC bus.
const <a href="#">GMCLIB_ELIMDCBUSRIP_T</a> *const	pParam	input	Pointer to the parameters structure.

**Return**

Function returns no value.

**Implementation details**

Outputs are normalized to the range [-1; 1] and can be used as inputs to the function [GMCLIB\\_SvmStd\\_FLT](#). If the required stator voltage exceeds the capabilities of the DC bus, the output is saturated to +/-1.

For better understanding, let's consider the following two simple examples:

**Example 1:**

- amplitude of the required phase voltage  $U_{reg}=50[V]$
- maximal amplitude of the DC bus voltage  $U_{DC\_BUS\_MAX}=100[V]$
- actual amplitude of the DC bus voltage  $U_{DC\_BUS\_ACTUAL}=100[V]$
- voltage to be applied to the PWM modulator to generate  $U_{reg}=50[V]$  on the inverter phase output:

$$U_{req\_new} = \frac{U_{req} U_{DC\_BUS\_MAX}}{U_{DC\_BUS\_ACTUAL}} = 50V$$

Equation GMCLIB\_ElimDcBusRip\_FLT\_Eq1

**Example 2:**

- amplitude of the required phase voltage  $U_{reg}=50[V]$
- maximal amplitude of the DC bus voltage  $U_{DC\_BUS\_MAX}=100[V]$
- actual amplitude of the DC bus voltage  $U_{DC\_BUS\_ACTUAL}=90[V]$
- voltage to be applied to the PWM modulator to generate  $U_{reg}=50[V]$  on the inverter phase output:

$$U_{req\_new} = \frac{U_{req} U_{DC\_BUS\_MAX}}{U_{DC\_BUS\_ACTUAL}} = 55.5V$$

Equation GMCLIB\_ElimDcBusRip\_FLT\_Eq2

The imperfections of the DC bus voltage are compensated for by the modification of amplitudes of the direct-  $\alpha$  and the quadrature-  $\beta$  components of the stator reference voltage vector. The following formulas are used:

$$u_{\alpha}^* = \begin{cases} \frac{\text{fltModIndex} \cdot u_{\alpha}}{\text{fltArgDCBusMsr}/2} & \text{if } \text{abs}(\text{fltModIndex} \cdot u_{\alpha}) < \frac{\text{fltArgDCBusMsr}}{2} \\ \text{sign}(u_{\alpha}) & \text{otherwise} \end{cases}$$

$$u_{\beta}^* = \begin{cases} \frac{\text{fltModIndex} \cdot u_{\beta}}{\text{fltArgDCBusMsr}/2} & \text{if } \text{abs}(\text{fltModIndex} \cdot u_{\beta}) < \frac{\text{fltArgDCBusMsr}}{2} \\ \text{sign}(u_{\beta}) & \text{otherwise} \end{cases}$$

Equation GMCLIB\_ElimDcBusRip\_FLT\_Eq3

where `fltModIndex` is the inverse modulation index, `fltArgDCBusMsr` is the measured DC bus voltage, the  $u_{\alpha}$  and  $u_{\beta}$  are the input voltages, and the  $u_{\alpha}^*$  and  $u_{\beta}^*$  are the output duty-cycle ratios.

The `fltModIndex` and `fltArgDCBusMsr` are supplied to the function within the parameters structure through its members. The  $u_{\alpha}$ ,  $u_{\beta}$  correspond respectively to the `fltArg1` and `fltArg2` members of the input structure `pIn`, and the  $u_{\alpha}^*$  and  $u_{\beta}^*$  respectively to the `fltArg1` and `fltArg2` members of the output structure `pOut`.

The inverse modulation index (see the parameters structure `pParam`, the `fltModIndex` member) must be equal to or greater than zero. Besides this restriction, the `fltModIndex` must be set to a valid value resulting from the use of Space Vector Modulation techniques.

**Note:** The function may raise floating-point exceptions (invalid operation, division by zero, overflow, underflow, inexact, input denormal). The floating-point unit must be enabled in CPACR register to prevent NOCP UsageFault exception.

Both the inverse modulation index `pIn->fltModIndex` and the measured DC bus voltage `pIn->fltArgDCBusMsr` must be equal to or greater than 0, otherwise the results are undefined.

### Code Example

```
#include "gmclib.h"

SWLIBS_2Syst_FLT fltAB;
SWLIBS_2Syst_FLT fltOutAB;
GMCLIB_ElimDCBusRip_T_FLT flttrMyElimDcBusRip =
    GMCLIB_ElimDCBusRip_Default_FLT;

void main(void)
{
    // inverse modulation coefficient for standard space vector modulation
    trMyElimDcBusRip.fltModIndex = 0.866025404;
    // Input voltage vector 15V @ angle 30deg
    // alpha component of input voltage vector = 12.99[V]
    fltAB.fltArg1 = 12.99038106;
    // beta component of input voltage vector = 7.5[V]
    fltAB.fltArg2 = 7.5;
    // value of the measured DC bus voltage 17V
    flttrMyElimDcBusRip.fltArgDCBusMsr = 17;

    // output alpha component of the output vector should be
    // fltOutAB.fltArg1 = 1
    // output beta component of the output vector should be
    // fltOutAB.fltArg2 = 0.764140062
    GMCLIB_ElimDcBusRip_FLT(&fltOutAB, &fltAB, &flttrMyElimDcBusRip);
```

```

// output alpha component of the output vector should be
// fltOutAB.fltArg1 = 1
// output beta component of the output vector should be
// fltOutAB.fltArg2 = 0.764140062
GMCLIB_ElimDcBusRip(&fltOutAB, &fltAB, &flttrMyElimDcBusRip, FLT);

// ######
// Available only if single precision floating point
// implementation selected as default
// #####
// output alpha component of the output vector should be
// fltOutAB.fltArg1 = 1
// output beta component of the output vector should be
// fltOutAB.fltArg2 = 0.764140062
GMCLIB_ElimDcBusRip(&fltOutAB, &fltAB, &flttrMyElimDcBusRip);
}

```

## 2.47 Function GMCLIB\_Park

This function implements the calculation of Park transformation.

### Description

The GMCLIB\_Park function calculates the Park Transformation, which transforms values (flux, voltage, current) from the two-phase ( $\alpha$ - $\beta$ ) stationary orthogonal coordinate system to the two-phase (d-q) rotational orthogonal coordinate system, according to these equations:

$$\begin{aligned} d &= \cos(\theta_e) \cdot \alpha + \sin(\theta_e) \cdot \beta \\ q &= -\sin(\theta_e) \cdot \alpha + \cos(\theta_e) \cdot \beta \end{aligned}$$

Equation GMCLIB\_Park\_Eq1

where  $\theta_e$  represents the electrical position of the rotor flux.

**Note:** The input/output pointers must contain valid addresses, otherwise a fault may occur (MemManage, BusFault, UsageFault, HardFault).

### Re-entrancy

The function is re-entrant.

#### 2.47.1 Function GMCLIB\_Park\_F32

##### Declaration

```
void GMCLIB_Park_F32(SWLIBS\_2Syst\_F32 *pOut, const
SWLIBS\_2Syst\_F32 *const pInAngle, const SWLIBS\_2Syst\_F32 *const
pIn);
```

**Arguments****Table 270. GMCLIB\_Park\_F32 arguments**

Type	Name	Direction	Description
<a href="#">SWLIBS_2Syst_F32</a> *	pOut	<a href="#">input, output</a>	Pointer to the structure containing data of the two-phase rotational orthogonal system (d-q).
const <a href="#">SWLIBS_2Syst_F32</a> *const	pInAngle	<a href="#">input</a>	Pointer to the structure where the values of the sine and cosine of the rotor position are stored.
const <a href="#">SWLIBS_2Syst_F32</a> *const	pIn	<a href="#">input</a>	Pointer to the structure containing data of the two-phase stationary orthogonal system ( $\alpha$ - $\beta$ ).

**Note:** Due to effectivity reasons this function is implemented using inline assembly and is therefore not ANSI-C compliant.

The inputs and the outputs are normalized to fit in the range [-1, 1].

**Code Example**

```
#include "gmclib.h"

SWLIBS\_2Syst\_F32 tr32Angle;
SWLIBS\_2Syst\_F32 tr32AlBe;
SWLIBS\_2Syst\_F32 tr32Dq;

void main(void)
{
    // input angle sin(60) = 0.866025403
    // input angle cos(60) = 0.5
    tr32Angle.f32Arg1 = FRAC32(0.866025403);
    tr32Angle.f32Arg2 = FRAC32(0.5);

    // input alpha = 0.123
    // input beta = 0.654
    tr32AlBe.f32Arg1 = FRAC32(0.123);
    tr32AlBe.f32Arg2 = FRAC32(0.654);

    // output should be
    // tr32Dq.f32Arg1 ~ d = 0x505E6455
    // tr32Dq.f32Arg2 ~ q = 0x1C38ABDC
    GMCLIB_Park_F32(&tr32Dq, &tr32Angle, &tr32AlBe);

    // output should be
    // tr32Dq.f32Arg1 ~ d = 0x505E6455
    // tr32Dq.f32Arg2 ~ q = 0x1C38ABDC
    GMCLIB_Park(&tr32Dq, &tr32Angle, &tr32AlBe, F32);

    // #####
    // Available only if 32-bit fractional implementation selected
    // as default
    // #####
}

// output should be
// tr32Dq.f32Arg1 ~ d = 0x505E6455
// tr32Dq.f32Arg2 ~ q = 0x1C38ABDC
GMCLIB_Park(&tr32Dq, &tr32Angle, &tr32AlBe);
```

## 2.47.2 Function GMCLIB\_Park\_F16

### Declaration

```
void GMCLIB_Park_F16(SWLIBS\_2Syst\_F16 *pOut, const
SWLIBS\_2Syst\_F16 *const pInAngle, const SWLIBS\_2Syst\_F16 *const
pIn);
```

### Arguments

**Table 271. GMCLIB\_Park\_F16 arguments**

Type	Name	Direction	Description
<a href="#">SWLIBS_2Syst_F16</a> *	pOut	input, output	Pointer to the structure containing data of the two-phase rotational orthogonal system (d-q).
const <a href="#">SWLIBS_2Syst_F16</a> *const	pInAngle	input	Pointer to the structure where the values of the sine and cosine of the rotor position are stored.
const <a href="#">SWLIBS_2Syst_F16</a> *const	pIn	input	Pointer to the structure containing data of the two-phase stationary orthogonal system ( $\alpha$ - $\beta$ ).

**Note:** Due to effectivity reasons this function is implemented using inline assembly and is therefore not ANSI-C compliant.

The inputs and the outputs are normalized to fit in the range [-1, 1).

### Code Example

```
#include "gmclib.h"

SWLIBS\_2Syst\_F16 tr16Angle;
SWLIBS\_2Syst\_F16 tr16AlBe;
SWLIBS\_2Syst\_F16 tr16Dq;

void main(void)
{
    // input angle sin(60) = 0.866025403
    // input angle cos(60) = 0.5
    tr16Angle.f16Arg1 = FRAC16(0.866025403);
    tr16Angle.f16Arg2 = FRAC16(0.5);

    // input alpha = 0.123
    // input beta = 0.654
    tr16AlBe.f16Arg1 = FRAC16(0.123);
    tr16AlBe.f16Arg2 = FRAC16(0.654);

    // output should be
    // tr16Dq.f16Arg1 ~ d = 0x505E
    // tr16Dq.f16Arg2 ~ q = 0x1C38
    GMCLIB_Park_F16(&tr16Dq, &tr16Angle, &tr16AlBe);

    // output should be
    // tr16Dq.f16Arg1 ~ d = 0x505E
    // tr16Dq.f16Arg2 ~ q = 0x1C38
    GMCLIB_Park(&tr16Dq, &tr16Angle, &tr16AlBe, F16);

    // ##### Available only if 16-bit fractional implementation selected
}
```

```

// as default
// #####
// output should be
// tr16Dq.fltArg1 ~ d = 0x505E
// tr16Dq.fltArg2 ~ q = 0x1C38
GMCLIB_Park(&tr16Dq, &tr16Angle, &tr16AlBe);
}

```

### 2.47.3 Function GMCLIB\_Park\_FLT

#### Declaration

```
void GMCLIB_Park_FLT(SWLIBS\_2Syst\_FLT *pOut, const
SWLIBS\_2Syst\_FLT *const pInAngle, const SWLIBS\_2Syst\_FLT *const
pIn);
```

#### Arguments

**Table 272. GMCLIB\_Park\_FLT arguments**

Type	Name	Direction	Description
<a href="#">SWLIBS_2Syst_FLT</a> *	pOut	input, output	Pointer to the structure containing data of the two-phase rotational orthogonal system (d-q).
const <a href="#">SWLIBS_2Syst_FLT</a> *const	pInAngle	input	Pointer to the structure where the values of the sine and cosine of the rotor position are stored.
const <a href="#">SWLIBS_2Syst_FLT</a> *const	pIn	input	Pointer to the structure containing data of the two-phase stationary orthogonal system ( $\alpha$ - $\beta$ ).

**Note:** The function may raise floating-point exceptions (invalid operation, overflow, underflow, inexact, input denormal). The floating-point unit must be enabled in CPACR register to prevent NOCP UsageFault exception.

#### Code Example

```

#include "gmclib.h"

SWLIBS\_2Syst\_FLT trfltAngle;
SWLIBS\_2Syst\_FLT trfltAlBe;
SWLIBS\_2Syst\_FLT trfltDq;

void main(void)
{
    // input angle sin(60) = 0.866025403
    // input angle cos(60) = 0.5
    trfltAngle.fltArg1 = 0.866025403;
    trfltAngle.fltArg2 = 0.5;

    // input alpha = 0.123
    // input beta = 0.654
    trfltAlBe.fltArg1 = 0.123;
    trfltAlBe.fltArg2 = 0.654;

    // output should be:
    // trfltDq.fltArg1 ~ d = 0.627880613
    // trfltDq.fltArg2 ~ q = 0.220479472
}

```

```

GMCLIB_Park_FLT(&trf1tDq,&trf1tAngle,&trf1tAlBe);

// output should be:
// trf1tDq.fltArg1 ~ d = 0.627880613
// trf1tDq.fltArg2 ~ q = 0.220479472
GMCLIB_Park(&trf1tDq,&trf1tAngle,&trf1tAlBe,FLT);

// ######
// Available only if single precision floating point
// implementation selected as default
// #####
// output should be:
// trf1tDq.fltArg1 ~ d = 0.627880613
// trf1tDq.fltArg2 ~ q = 0.220479472
GMCLIB_Park(&trf1tDq,&trf1tAngle,&trf1tAlBe);
}

```

## 2.48 Function GMCLIB\_ParkInv

This function implements the inverse Park transformation.

### Description

The GMCLIB\_ParkInv function calculates the Inverse Park Transformation, which transforms quantities (flux, voltage, current) from the two-phase (d-q) rotational orthogonal coordinate system to the two-phase ( $\alpha$ - $\beta$ ) stationary orthogonal coordinate system, according to these equations:

$$\begin{aligned}\alpha &= \cos(\theta_e) \cdot d - \sin(\theta_e) \cdot q \\ \beta &= \sin(\theta_e) \cdot d + \cos(\theta_e) \cdot q\end{aligned}$$

Equation GMCLIB\_ParkInv\_Eq1

where  $\theta_e$  represents the electrical position of the rotor flux.

**Note:** The input/output pointers must contain valid addresses, otherwise a fault may occur (MemManage, BusFault, UsageFault, HardFault).

### Re-entrancy

The function is re-entrant.

#### 2.48.1 Function GMCLIB\_ParkInv\_F32

##### Declaration

```
void GMCLIB_ParkInv_F32(SWLIBS\_2Syst\_F32 *const pOut, const
SWLIBS\_2Syst\_F32 *const pInAngle, const SWLIBS\_2Syst\_F32 *const
pIn);
```

**Arguments****Table 273. GMCLIB\_ParkInv\_F32 arguments**

Type	Name	Direction	Description
<a href="#">SWLIBS_2Syst_F32</a> *const	pOut	input, output	Pointer to the structure containing data of the two-phase stationary orthogonal system ( $\alpha$ - $\beta$ ).
const <a href="#">SWLIBS_2Syst_F32</a> *const	pInAngle	input	Pointer to the structure where the values of the sine and cosine of the rotor position are stored.
const <a href="#">SWLIBS_2Syst_F32</a> *const	pIn	input	Pointer to the structure containing data of the two-phase rotational orthogonal system (d-q).

**Note:** The inputs and the outputs are normalized to fit in the range [-1, 1].

**Code Example**

```
#include "gmclib.h"

SWLIBS\_2Syst\_F32 tr32Angle;
SWLIBS\_2Syst\_F32 tr32Dq;
SWLIBS\_2Syst\_F32 tr32AlBe;

void main(void)
{
    // input angle sin(60) = 0.866025403
    // input angle cos(60) = 0.5
    tr32Angle.f32Arg1 = FRAC32(0.866025403);
    tr32Angle.f32Arg2 = FRAC32(0.5);

    // input d = 0.123
    // input q = 0.654
    tr32Dq.f32Arg1 = FRAC32(0.123);
    tr32Dq.f32Arg2 = FRAC32(0.654);

    // output should be
    // tr32AlBe.f32Arg1 ~ alpha = 0xBF601273
    // tr32AlBe.f32Arg2 ~ beta = 0x377D9EE4
    GMCLIB_ParkInv_F32(&tr32AlBe,&tr32Angle,&tr32Dq);

    // output should be
    // tr32AlBe.f32Arg1 ~ alpha = 0xBF601273
    // tr32AlBe.f32Arg2 ~ beta = 0x377D9EE4
    GMCLIB_ParkInv(&tr32AlBe,&tr32Angle,&tr32Dq,F32);

    // ######
    // Available only if 32-bit fractional implementation selected
    // as default
    // #####
}

// output should be
// tr32AlBe.f32Arg1 ~ alpha = 0xBF601273
// tr32AlBe.f32Arg2 ~ beta = 0x377D9EE4
GMCLIB_ParkInv(&tr32AlBe,&tr32Angle,&tr32Dq);
```

## 2.48.2 Function GMCLIB\_ParkInv\_F16

### Declaration

```
void GMCLIB_ParkInv_F16(SWLIBS\_2Syst\_F16 *const pOut, const
SWLIBS\_2Syst\_F16 *const pInAngle, const SWLIBS\_2Syst\_F16 *const
pIn);
```

### Arguments

**Table 274. GMCLIB\_ParkInv\_F16 arguments**

Type	Name	Direction	Description
<a href="#">SWLIBS_2Syst_F16</a> *const	pOut	input, output	Pointer to the structure containing data of the two-phase stationary orthogonal system ( $\alpha$ - $\beta$ ).
const <a href="#">SWLIBS_2Syst_F16</a> *const	pInAngle	input	Pointer to the structure where the values of the sine and cosine of the rotor position are stored.
const <a href="#">SWLIBS_2Syst_F16</a> *const	pIn	input	Pointer to the structure containing data of the two-phase rotational orthogonal system (d-q).

*Note:* The inputs and the outputs are normalized to fit in the range [-1, 1].

### Code Example

```
#include "gmclib.h"

SWLIBS\_2Syst\_F16 tr16Angle;
SWLIBS\_2Syst\_F16 tr16Dq;
SWLIBS\_2Syst\_F16 tr16AlBe;

void main(void)
{
    // input angle sin(60) = 0.866025403
    // input angle cos(60) = 0.5
    tr16Angle.f16Arg1 = FRAC16(0.866025403);
    tr16Angle.f16Arg2 = FRAC16(0.5);

    // input d = 0.123
    // input q = 0.654
    tr16Dq.f16Arg1 = FRAC16(0.123);
    tr16Dq.f16Arg2 = FRAC16(0.654);

    // output should be
    // tr16AlBe.f16Arg1 ~ alpha = 0xBF61
    // tr16AlBe.f16Arg2 ~ beta = 0x377C
    GMCLIB_ParkInv_F16(&tr16AlBe,&tr16Angle,&tr16Dq);

    // output should be
    // tr16AlBe.f16Arg1 ~ alpha = 0xBF61
    // tr16AlBe.f16Arg2 ~ beta = 0x377C
    GMCLIB_ParkInv(&tr16AlBe,&tr16Angle,&tr16Dq,F16);

    // #####
    // Available only if 16-bit fractional implementation selected
    // as default
    // #####
}

// output should be
```

```

// tr16AlBe.f16Arg1 ~ alpha = 0xBF61
// tr16AlBe.f16Arg2 ~ beta = 0x377C
GMCLIB_ParkInv(&tr16AlBe,&tr16Angle,&tr16Dq);
}

```

### 2.48.3 Function GMCLIB\_ParkInv\_FLT

#### Declaration

```
void GMCLIB_ParkInv_FLT(SWLIBS_2Syst_FLT *const pOut, const
SWLIBS_2Syst_FLT *const pInAngle, const SWLIBS_2Syst_FLT *const
pIn);
```

#### Arguments

**Table 275. GMCLIB\_ParkInv\_FLT arguments**

Type	Name	Direction	Description
<u>SWLIBS_2Syst_FLT</u> *const	pOut	input, output	Pointer to the structure containing data of the two-phase stationary orthogonal system ( $\alpha$ - $\beta$ ).
const <u>SWLIBS_2Syst_FLT</u> *const	pInAngle	input	Pointer to the structure where the values of the sine and cosine of the rotor position are stored.
const <u>SWLIBS_2Syst_FLT</u> *const	pIn	input	Pointer to the structure containing data of the two-phase rotational orthogonal system (d-q).

**Note:** The function may raise floating-point exceptions (invalid operation, overflow, underflow, inexact, input denormal). The floating-point unit must be enabled in CPACR register to prevent NOCP UsageFault exception.

The inputs and the outputs are in single precision floating point data format.

#### Code Example

```

#include "gmclib.h"

SWLIBS_2Syst_FLT trfltAngle;
SWLIBS_2Syst_FLT trfltDq;
SWLIBS_2Syst_FLT trfltAlBe;

void main(void)
{
    // input angle sin(60) = 0.866025403
    // input angle cos(60) = 0.5
    trfltAngle.fltArg1 = 0.866025403;
    trfltAngle.fltArg2 = 0.5;

    // input d = 0.123
    // input q = 0.654
    trfltDq.fltArg1 = 0.123;
    trfltDq.fltArg2 = 0.654;

    // output should be:
    // trfltAlBe.fltArg1 ~ alpha = -0.495119387
    // trfltAlBe.fltArg2 ~ beta = 0.433521139
    GMCLIB_ParkInv_FLT(&trfltAlBe,&trfltAngle,&trfltDq);
}

```

```

// output should be:
// trfltAlBe.fltArg1 ~ alpha = -0.495119387
// trfltAlBe.fltArg2 ~ beta = 0.433521139
GMCLIB_ParkInv(&trfltAlBe,&trfltAngle,&trfltDq,FLT);

// ######
// Available only if single precision floating point
// implementation selected as default
// #####
// output should be:
// trfltAlBe.fltArg1 ~ alpha = -0.495119387
// trfltAlBe.fltArg2 ~ beta = 0.433521139
GMCLIB_ParkInv(&trfltAlBe,&trfltAngle,&trfltDq);
}

```

## 2.49 Function GMCLIB\_PwmIct

This function calculates appropriate duty-cycle ratios which are needed for generating the given stator reference voltage vector using the general sinusoidal modulation technique.

### Description

The GMCLIB\_PwmIct function calculates the appropriate duty-cycle ratios needed to generate the given stator reference voltage vector with the help of the conventional inverse Clarke transform. Refer to the description of [GMLIB\\_SvmStd](#) for explanation of the stator voltage sectors.

The following equations are used to calculate the output duty-cycle ratios in the range  $0 < \text{pwm} < 1$

$$\text{pwm}_A = 0.5 + \frac{u_\alpha}{2}$$

Equation GMCLIB\_PwmIct\_Eq1

$$\text{pwm}_B = 0.5 + \frac{-u_\alpha + \sqrt{3}u_\beta}{4}$$

Equation GMCLIB\_PwmIct\_Eq2

$$\text{pwm}_C = 0.5 + \frac{-u_\alpha - \sqrt{3}u_\beta}{4}$$

Equation GMCLIB\_PwmIct\_Eq3

The following figures show the input and output waveforms.

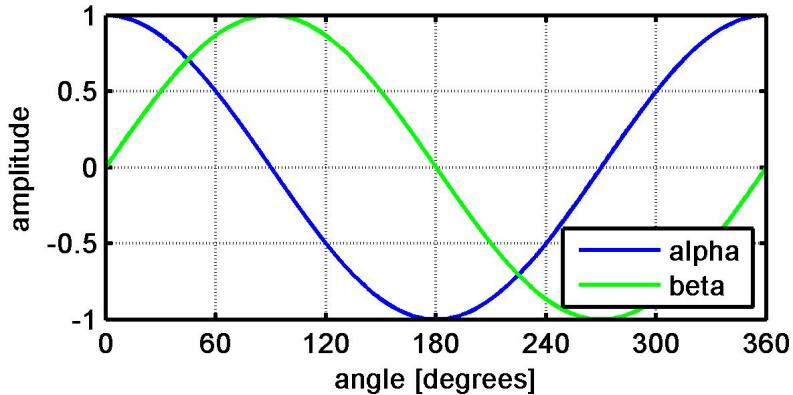


Figure 87. Input waveforms

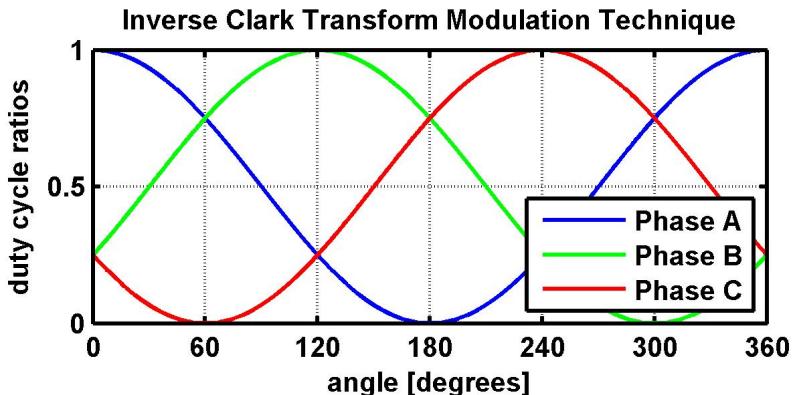


Figure 88. Output waveforms

**Note:** The input/output pointers must contain valid addresses, otherwise a fault may occur (MemManage, BusFault, UsageFault, HardFault).

### Re-entrancy

The function is re-entrant.

#### 2.49.1 Function GMCLIB\_PwmIct\_F32

##### Declaration

```
tU32 GMCLIB_PwmIct_F32(SWLIBS\_3Syst\_F32 *pOut, const
SWLIBS\_2Syst\_F32 *const pIn);
```

##### Arguments

Table 276. GMCLIB\_PwmIct\_F32 arguments

Type	Name	Direction	Description
<a href="#">SWLIBS_3Syst_F32</a> *	pOut	input, output	Pointer to the structure containing calculated duty-cycle ratios of the 3-Phase system.
const <a href="#">SWLIBS_2Syst_F32</a> *const	pIn	input	Pointer to the structure containing direct $U_\alpha$ and quadrature $U_\beta$ components of the stator voltage vector.

**Return**

The function returns a 32-bit integer value representing the sector which contains the stator reference vector  $U_s$ .

**Code Example**

```
#include "gmclib.h"
#define U_MAX 15

SWLIBS_2Syst_F32 inVoltage;
SWLIBS_3Syst_F32 pwmABC;
tU32 sector;

void main(void)
{
    // Input voltage vector 15V @ angle 30deg
    // alpha component of input voltage vector = 12.99[V]
    // beta component of input voltage vector = 7.5[V]
    inVoltage.f32Arg1 = FRAC32(12.99/U_MAX);
    inVoltage.f32Arg2 = FRAC32(7.5/U_MAX);

    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    sector = GMCLIB_PwmIct_F32(&pwmABC,&inVoltage);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    sector = GMCLIB_PwmIct(&pwmABC,&inVoltage,F32);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 32-bit fractional implementation is selected as default.
    sector = GMCLIB_PwmIct(&pwmABC,&inVoltage);

    // Expected output pwm duty-cycles pwmABC:
    // pwmABC.f32Arg1 = 0x776C8B43 ~ FRAC32(0.9330)
    // pwmABC.f32Arg2 = 0x40003546 ~ FRAC32(0.5000)
    // pwmABC.f32Arg3 = 0x08933F77 ~ FRAC32(0.0670)
    // sector = 1
}
```

**2.49.2 Function GMCLIB\_PwmIct\_F16****Declaration**

```
tU16 GMCLIB_PwmIct_F16(SWLIBS_3Syst_F16 *pOut, const
SWLIBS_2Syst_F16 *const pIn);
```

**Arguments****Table 277. GMCLIB\_PwmIct\_F16 arguments**

Type	Name	Direction	Description
SWLIBS_3Syst_F16 *	pOut	input, output	Pointer to the structure containing calculated duty-cycle ratios of the 3-Phase system.

Type	Name	Direction	Description
const <a href="#">SWLIBS_2Syst_F16</a> *const	pIn	input	Pointer to the structure containing direct $U_\alpha$ and quadrature $U_\beta$ components of the stator voltage vector.

**Return**

The function returns a 16-bit integer value representing the sector which contains the stator reference vector  $U_s$ .

**Code Example**

```
#include "gmclib.h"
#define U_MAX 15

SWLIBS\_2Syst\_F16 inVoltage;
SWLIBS\_3Syst\_F16 pwmABC;
tU16 sector;

void main(void)
{
    // Input voltage vector 15V @ angle 30deg
    // alpha component of input voltage vector = 12.99[V]
    // beta component of input voltage vector = 7.5[V]
    inVoltage.f16Arg1 = FRAC16(12.99/U_MAX);
    inVoltage.f16Arg2 = FRAC16(7.5/U_MAX);

    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    sector = GMCLIB_PwmIct_F16(&pwmABC, &inVoltage);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    sector = GMCLIB_PwmIct(&pwmABC, &inVoltage, F16);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if 32-bit fractional implementation is selected as default.
    sector = GMCLIB_PwmIct(&pwmABC, &inVoltage);

    // Expected output pwm duty-cycles pwmABC:
    // pwmABC.f16Arg1 = 0x776C ~ FRAC16(0.9330)
    // pwmABC.f16Arg2 = 0x4000 ~ FRAC16(0.5000)
    // pwmABC.f16Arg3 = 0x0894 ~ FRAC16(0.0670)
    // sector = 1
}
```

**2.49.3 Function GMCLIB\_PwmIct\_FLT****Declaration**

```
tU32 GMCLIB_PwmIct_FLT(SWLIBS\_3Syst\_FLT *pOut, const
SWLIBS\_2Syst\_FLT *const pIn);
```

**Arguments****Table 278. GMCLIB\_PwmIct\_FLT arguments**

Type	Name	Direction	Description
<a href="#">SWLIBS_3Syst_FLT</a> *	pOut	<a href="#">input, output</a>	Pointer to the structure containing calculated duty-cycle ratios of the 3-Phase system.
const <a href="#">SWLIBS_2Syst_FLT</a> *const	pln	<a href="#">input</a>	Pointer to the structure containing direct $U_\alpha$ and quadrature $U_\beta$ components of the stator voltage vector.

**Return**

The function returns a 32-bit integer value representing the sector which contains the stator reference vector  $U_s$ .

**Implementation details**

The function presumes that the input voltages are normalized to fit the range <-1; 1>. In a typical motor control application, this function is preceded by the function [GMCLIB\\_ElimDcBusRip\\_FLT](#) which ensures that the voltages are correctly normalized.

**Note:** *The function may raise floating-point exceptions (invalid operation, overflow, underflow, inexact, input denormal). The floating-point unit must be enabled in CPACR register to prevent NOCP UsageFault exception.*

**Code Example**

```
#include "gmclib.h"
#define U_MAX 15

SWLIBS\_2Syst\_FLT inVoltage;
SWLIBS\_3Syst\_FLT pwmABC;
tU32 sector;

void main(void)
{
    // Input voltage vector 15V @ angle 30deg
    // alpha component of input voltage vector = 12.99[V]
    // beta component of input voltage vector = 7.5[V]
    inVoltage.fltArg1 = (tFloat)(12.99/U_MAX);
    inVoltage.fltArg2 = (tFloat)(7.5/U_MAX);

    // Alternative 1: API call with postfix
    // (only one alternative shall be used).
    sector = GMCLIB_PwmIct_FLT(&pwmABC,&inVoltage);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    sector = GMCLIB_PwmIct(&pwmABC,&inVoltage,FLT);

    // Alternative 3: API call with global configuration of implementation
    // (only one alternative shall be used). This alternative is available
    // only if single precision floating point implementation is selected
    // as default.
    sector = GMCLIB_PwmIct(&pwmABC,&inVoltage);

    // Expected output pwm duty-cycles pwmABC:
    // pwmABC.fltArg1 = 0.93299997
}
```

```
// pwmABC.fltArg2 = 0.50000632
// pwmABC.fltArg3 = 0.066993654
// sector = 1
}
```

## 2.50 Function GMCLIB\_SvmSci

This function calculates appropriate duty-cycle ratios, which are needed for generating the given stator reference voltage vector using the General Sinusoidal Modulation with injection of the third harmonic.

### Description

The GMCLIB\_SvmSci function calculates the appropriate duty-cycle ratios, needed for generating the given stator reference voltage vector with the help of the sinusoidal modulation with Sine Cap Injection algorithm.

Finding the sector in which the reference stator voltage vector  $U_S$  resides is similar to that discussed in [GMCLIB\\_SvmStd](#).

The balanced 3-Phase duty-cycle ratios may be calculated based on Sine Cap Injection algorithm in the following stages:

1. The calculation of the basic duty-cycle ratios using the Inverse Clarke Transformation.

$$\begin{aligned} u_a &= u_\alpha \\ u_b &= \frac{-u_\alpha\sqrt{3}u_\beta}{2} \\ u_c &= \frac{-u_\alpha\sqrt{3}u_\beta}{2} \end{aligned}$$

Equation GMCLIB\_SvmSci\_Eq1

2. An amplitude of the basic duty-cycle ratios  $u_a$ ,  $u_b$  and  $u_c$  calculated by [GMCLIB\\_SvmSci\\_Eq1](#) is in the range [-1, 1]. The basic duty-cycle ratios are then multiplied by the coefficient  $2/\sqrt{3}$ .

$$\begin{aligned} u'_a &= \frac{2}{\sqrt{3}} u_a \\ u'_b &= \frac{2}{\sqrt{3}} u_b \\ u'_c &= \frac{2}{\sqrt{3}} u_c \end{aligned}$$

Equation GMCLIB\_SvmSci\_Eq2

3. If the values of variables  $u'_a$ ,  $u'_b$  and  $u'_c$  exceed the unity, they are stored in an auxiliary variable  $u_0$ . This variable is called the Sine Cap Voltage variable. The procedure to obtain this can be mathematically defined by a following series of formulas:

$$u_0 = \begin{cases} 1-u'_a & \text{if } u'_a > 1 \\ -1-u'_a & \text{if } u'_a < -1 \\ 0 & \text{otherwise} \end{cases}$$

$$u_0 = \begin{cases} 1-u'_b & \text{if } u'_b > 1 \\ -1-u'_b & \text{if } u'_b < -1 \\ 0 & \text{otherwise} \end{cases}$$

$$u_0 = \begin{cases} 1-u'_c & \text{if } u'_c > 1 \\ -1-u'_c & \text{if } u'_c < -1 \\ 0 & \text{otherwise} \end{cases}$$

Equation GMCLIB\_SvmSci\_Eq3

4. Due to the  $120^\circ$  voltage phase shift, distinguishing for balanced three-phase system, only one phase contributes to the building of Sine-Cap Voltage  $u_0$  at each time point.

Finally the duty-cycle ratios are then calculated bu following equations:

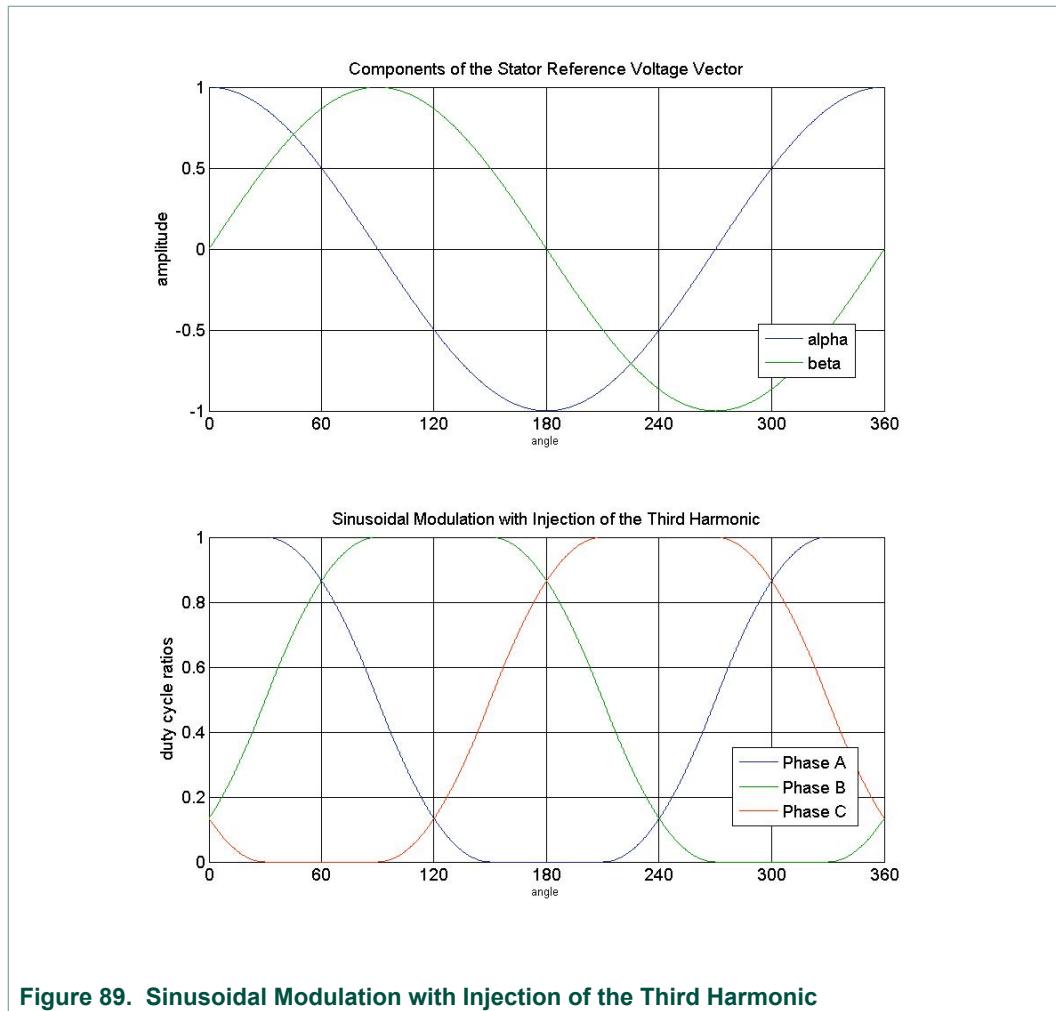
$$pwm_A = \frac{1}{2}(u_0 + u'_a + 1)$$

$$pwm_B = \frac{1}{2}(u_0 + u'_b + 1)$$

$$pwm_C = \frac{1}{2}(u_0 + u'_c + 1)$$

Equation GMCLIB\_SvmSci\_Eq4

[Figure 89](#) shows calculated waveforms of the duty-cycle ratios using the sinusoidal modulation with Sine Cap Injection algorithm.

**Figure 89. Sinusoidal Modulation with Injection of the Third Harmonic**

**Note:** To provide an accurate calculation of the duty-cycle ratios, the stator reference voltage vector given by direct-a and quadrature-b components can not have magnitude exceeding the unity circle.

**Note:** The input/output pointers must contain valid addresses, otherwise a fault may occur (MemManage, BusFault, UsageFault, HardFault).

### Re-entrancy

The function is re-entrant.

#### 2.50.1 Function GMCLIB\_SvmSci\_F32

##### Declaration

```
tU32 GMCLIB_SvmSci_F32(SWLIBS_3Syst_F32 *pOut, const
                           SWLIBS_2Syst_F32 *const pIn);
```

**Arguments****Table 279. GMCLIB\_SvmSci\_F32 arguments**

Type	Name	Direction	Description
<a href="#">SWLBS_3Syst_F32</a> *	pOut	<a href="#">input, output</a>	Pointer to the structure containing calculated duty-cycle ratios of the 3-Phase system.
const <a href="#">SWLBS_2Syst_F32</a> *const	pln	<a href="#">input</a>	Pointer to the structure containing direct $U_\alpha$ and quadrature $U_\beta$ components of the stator voltage vector.

**Return**

The function returns a 32-bit value in integer format, representing the actual space sector which contains the stator reference vector  $U_s$ .

**Code Example**

```
#include "gmclib.h"
#define U_MAX 15

SWLBS\_2Syst\_F32 tr32InVoltage;
SWLBS\_3Syst\_F32 tr32PwmABC;
tU32 u32SvmSector;

void main(void)
{
    // Input voltage vector 15V @ angle 30deg
    // alpha component of input voltage vector = 12.99[V]
    // beta component of input voltage vector = 7.5[V]
    tr32InVoltage.f32Arg1 = FRAC32(12.99/U_MAX);
    tr32InVoltage.f32Arg2 = FRAC32(7.5/U_MAX);

    // output pwm dutycycles stored in structure referenced by tr32PwmABC
    // pwmA dutycycle = 0x7FFF FFFF = FRAC32(1.0000000... )
    // pwmb dutycycle = 0x4000 B88D = FRAC32(0.5000220... )
    // pwmc dutycycle = 0x0000 B88D = FRAC32(0.0000220... )
    // svmSector = 0x1 [sector]
    u32SvmSector = GMCLIB_SvmSci_F32(&tr32PwmABC,&tr32InVoltage);

    // output pwm dutycycles stored in structure referenced by tr32PwmABC
    // pwmA dutycycle = 0x7FFF FFFF = FRAC32(1.0000000... )
    // pwmb dutycycle = 0x4000 B88D = FRAC32(0.5000220... )
    // pwmc dutycycle = 0x0000 B88D = FRAC32(0.0000220... )
    // svmSector = 0x1 [sector]
    u32SvmSector = GMCLIB_SvmSci(&tr32PwmABC,&tr32InVoltage,F32);

    // ######
    // Available only if 32-bit fractional implementation selected
    // as default
    // #####
    // output pwm dutycycles stored in structure referenced by tr32PwmABC
    // pwmA dutycycle = 0x7FFF FFFF = FRAC32(1.0000000... )
    // pwmb dutycycle = 0x4000 B88D = FRAC32(0.5000220... )
    // pwmc dutycycle = 0x0000 B88D = FRAC32(0.0000220... )
    // svmSector = 0x1 [sector]
    u32SvmSector = GMCLIB_SvmSci(&tr32PwmABC,&tr32InVoltage);
}
```

## 2.50.2 Function GMCLIB\_SvmSci\_F16

### Declaration

```
tU16 GMCLIB_SvmSci_F16(SWLIBS_3Syst_F16 *pOut, const
SWLIBS_2Syst_F16 *const pIn);
```

### Arguments

Table 280. GMCLIB\_SvmSci\_F16 arguments

Type	Name	Direction	Description
<code>SWLIBS_3Syst_F16 *</code>	pOut	<code>input, output</code>	Pointer to the structure containing calculated duty-cycle ratios of the 3-Phase system.
<code>const SWLIBS_2Syst_F16 *const</code>	pIn	<code>input</code>	Pointer to the structure containing direct $U_\alpha$ and quadrature $U_\beta$ components of the stator voltage vector.

### Return

The function returns a 16-bit value in integer format, representing the actual space sector which contains the stator reference vector  $U_s$ .

### Code Example

```
#include "gmclib.h"
#define U_MAX 15

SWLIBS_2Syst_F16 tr16InVoltage;
SWLIBS_3Syst_F16 tr16PwmABC;
tU16 u16SvmSector;

void main(void)
{
    // Input voltage vector 15V @ angle 30deg
    // alpha component of input voltage vector = 12.99[V]
    // beta component of input voltage vector = 7.5[V]
    tr16InVoltage.f16Arg1 = FRAC16(12.99/U_MAX);
    tr16InVoltage.f16Arg2 = FRAC16(7.5/U_MAX);

    // output pwm dutycycles stored in structure referenced by tr16PwmABC
    // pwmA dutycycle = 0x7FFF = FRAC32(1.000...)
    // pwmb dutycycle = 0x4000 = FRAC32(0.500...)
    // pwmc dutycycle = 0x0000 = FRAC32(0.000...)
    // svmSector = 0x1 [sector]
    u16SvmSector = GMCLIB_SvmSci_F16(&tr16PwmABC,&tr16InVoltage);

    // output pwm dutycycles stored in structure referenced by tr16PwmABC
    // pwmA dutycycle = 0x7FFF = FRAC32(1.000...)
    // pwmb dutycycle = 0x4000 = FRAC32(0.500...)
    // pwmc dutycycle = 0x0000 = FRAC32(0.000...)
    // svmSector = 0x1 [sector]
    u16SvmSector = GMCLIB_SvmSci(&tr16PwmABC,&tr16InVoltage,F16);

    ##### Available only if 16-bit fractional implementation selected #####
    // as default
    #####
```

```

// output pwm dutycycles stored in structure referenced by tr16PwmABC
// pwmA dutycycle = 0x7FFF = FRAC32(1.000...)
// pwmb dutycycle = 0x4000 = FRAC32(0.500...)
// pwmc dutycycle = 0x0000 = FRAC32(0.000...)
// svmSector      = 0x1 [sector]
u16SvmSector = GMCLIB_SvmSci(&tr16PwmABC,&tr16InVoltage);
}

```

### 2.50.3 Function GMCLIB\_SvmSci\_FLT

#### Declaration

```
tU32 GMCLIB_SvmSci_FLT(SWLIBS\_3Syst\_FLT *pOut, const
SWLIBS\_2Syst\_FLT *const pIn);
```

#### Arguments

**Table 281. GMCLIB\_SvmSci\_FLT arguments**

Type	Name	Direction	Description
<a href="#">SWLIBS_3Syst_FLT</a> *	pOut	input, output	Pointer to the structure containing calculated duty-cycle ratios of the 3-Phase system.
const <a href="#">SWLIBS_2Syst_FLT</a> *const	pIn	input	Pointer to the structure containing direct $U_\alpha$ and quadrature $U_\beta$ components of the stator voltage vector.

#### Return

The function returns a 32-bit value in integer format, representing the actual space sector which contains the stator reference vector  $U_s$ .

#### Implementation details

The function presumes that the input voltages are normalized to fit the range <-1; 1>. In a typical motor control application, this function is preceded by the function [GMCLIB\\_ElimDcBusRip\\_FLT](#) which ensures that the voltages are correctly normalized.

**Note:** The function may raise floating-point exceptions (invalid operation, overflow, underflow, inexact, input denormal). The floating-point unit must be enabled in CPACR register to prevent NOCP UsageFault exception.

#### Code Example

```

#include "gmclib.h"
#define U_MAX 15

SWLIBS\_2Syst\_FLT tInVoltage;
SWLIBS\_3Syst\_FLT tPwmABC;
tU32 u32SvmSector;

void main(void)
{
    // Input voltage vector 15V @ angle 30deg
    // alpha component of input voltage vector = 12.99[V]
    // beta component of input voltage vector = 7.5[V]
    tInVoltage.fltArg1 = (tFloat) (12.99/U_MAX);
    tInVoltage.fltArg2 = (tFloat) (7.5/U_MAX);
}

```

```
// output pwm dutycycles stored in structure referenced by tr32PwmABC
// pwmA dutycycle = (tFloat)(1.0000000)
// pwmb dutycycle = (tFloat)(0.5000220)
// pwmc dutycycle = (tFloat)(0.0000220)
// u32SvmSector = 0x1 [sector]
u32SvmSector = GMCLIB_SvmSci_FLT(&tPwmABC, &tInVoltage);

// output pwm dutycycles stored in structure referenced by tr32PwmABC
// pwmA dutycycle = (tFloat)(1.0000000)
// pwmb dutycycle = (tFloat)(0.5000220)
// pwmc dutycycle = (tFloat)(0.0000220)
// u32SvmSector = 0x1 [sector]
u32SvmSector = GMCLIB_SvmSci(&tPwmABC, &tInVoltage, FLT);

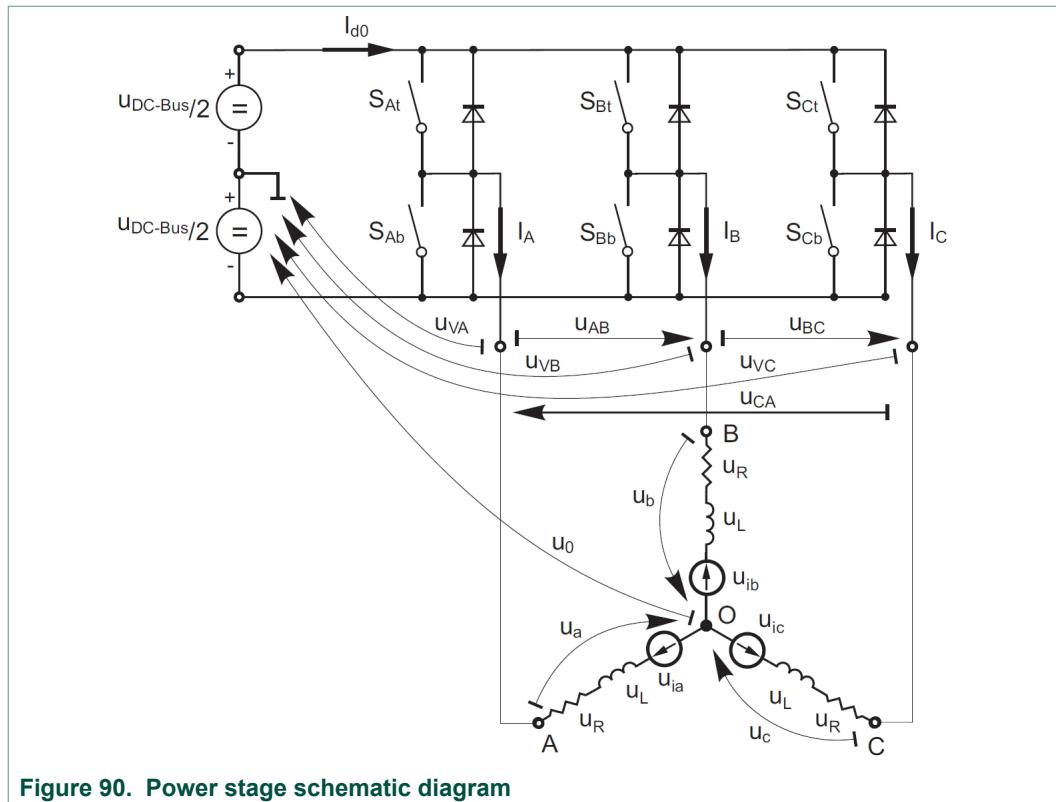
// ######
// Available only if single precision floating point
// implementation selected as default
// #####
// #####
// output pwm dutycycles stored in structure referenced by tr32PwmABC
// pwmA dutycycle = (tFloat)(1.0000000)
// pwmb dutycycle = (tFloat)(0.5000220)
// pwmc dutycycle = (tFloat)(0.0000220)
// u32SvmSector = 0x1 [sector]
u32SvmSector = GMCLIB_SvmSci(&tPwmABC, &tInVoltage);
}
```

## 2.51 Function GMCLIB\_SvmStd

This function calculates the duty-cycle ratios using the Standard Space Vector Modulation technique.

### Description

The GMCLIB\_SvmStd function for calculating duty-cycle ratios is widely-used in the modern electric drive. This function calculates appropriate duty-cycle ratios, which are needed for generating the given stator reference voltage vector using a special Space Vector Modulation technique, termed Standard Space Vector Modulation. The basic principle of the Standard Space Vector Modulation Technique can be explained with the help of the power stage diagram in [Figure 90](#).



Top and bottom switches work in a complementary mode; i.e., if the top switch,  $S_{At}$ , is ON, then the corresponding bottom switch,  $S_{Ab}$ , is OFF, and vice versa. Considering that value 1 is assigned to the ON state of the top switch, and value 0 is assigned to the ON state of the bottom switch, the switching vector,  $[a, b, c]^T$  can be defined. Creating such a vector allows a numerical definition of all possible switching states. In a three-phase power stage configuration (as shown in Figure 90), eight possible switching states (detailed in Figure 91) are feasible.

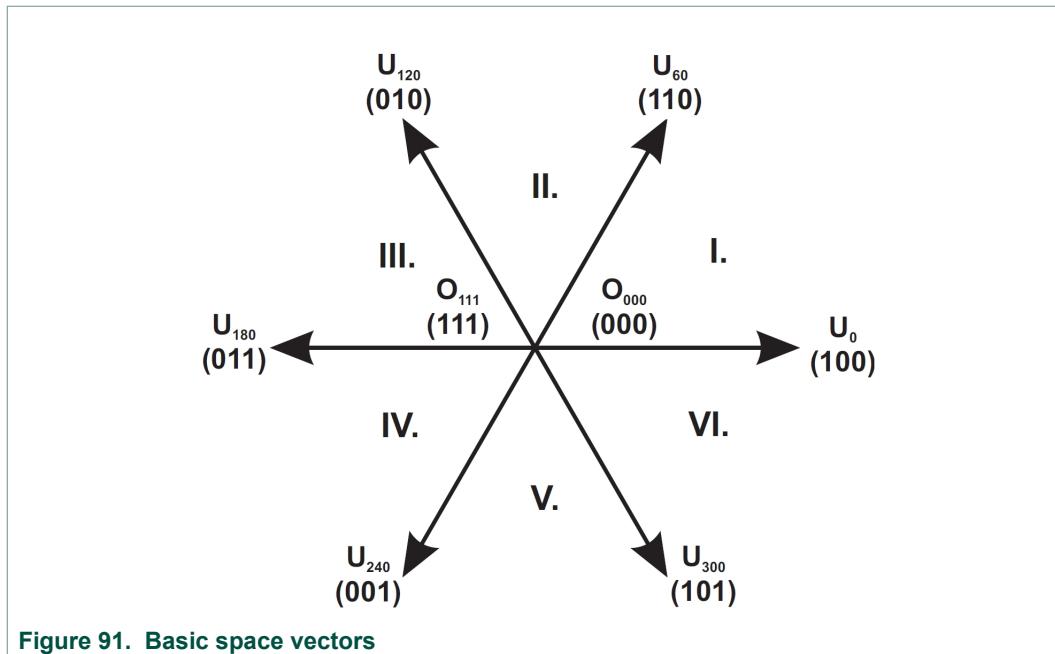


Figure 91. Basic space vectors

These states, together with the resulting instantaneous output line-to-line and phase voltages, are listed in [Table 282](#).

Table 282. Switching patterns

a	b	c	$U_a$	$U_b$	$U_c$	$U_{AB}$	$U_{BC}$	$U_{CA}$	Vector
0	0	0	0	0	0	0	0	0	$O_{000}$
1	0	0	$2/3 \cdot U_{DCBus}$	$-1/3 \cdot U_{DCBus}$	$-1/3 \cdot U_{DCBus}$	$U_{DCBus}$	0	$-U_{DCBus}$	$U_0$
1	1	0	$1/3 \cdot U_{DCBus}$	$1/3 \cdot U_{DCBus}$	$-2/3 \cdot U_{DCBus}$	0	$U_{DCBus}$	$-U_{DCBus}$	$U_{60}$
0	1	0	$-1/3 \cdot U_{DCBus}$	$2/3 \cdot U_{DCBus}$	$-1/3 \cdot U_{DCBus}$	$-U_{DCBus}$	$U_{DCBus}$	0	$U_{120}$
0	1	1	$-2/3 \cdot U_{DCBus}$	$1/3 \cdot U_{DCBus}$	$1/3 \cdot U_{DCBus}$	$-U_{DCBus}$	0	$U_{DCBus}$	$U_{180}$
0	0	1	$-1/3 \cdot U_{DCBus}$	$-1/3 \cdot U_{DCBus}$	$2/3 \cdot U_{DCBus}$	0	$-U_{DCBus}$	$U_{DCBus}$	$U_{240}$
1	0	1	$1/3 \cdot U_{DCBus}$	$-2/3 \cdot U_{DCBus}$	$1/3 \cdot U_{DCBus}$	$U_{DCBus}$	$-U_{DCBus}$	0	$U_{300}$
1	1	1	0	0	0	0	0	0	$O_{111}$

The quantities of the direct- $U_\alpha$  and the quadrature- $U_\beta$  components of the two-phase orthogonal coordinate system, describing the three-phase stator voltages, are expressed by the Clarke Transformation.

$$U_\alpha = \frac{2}{3} \cdot \left( U_a - \frac{U_b}{2} - \frac{U_c}{2} \right)$$

Equation GMCLIB\_SvmStd\_Eq1

$$U_\beta = \frac{2}{3} \cdot \left( 0 + \frac{\sqrt{3}U_b}{2} - \frac{\sqrt{3}U_c}{2} \right)$$

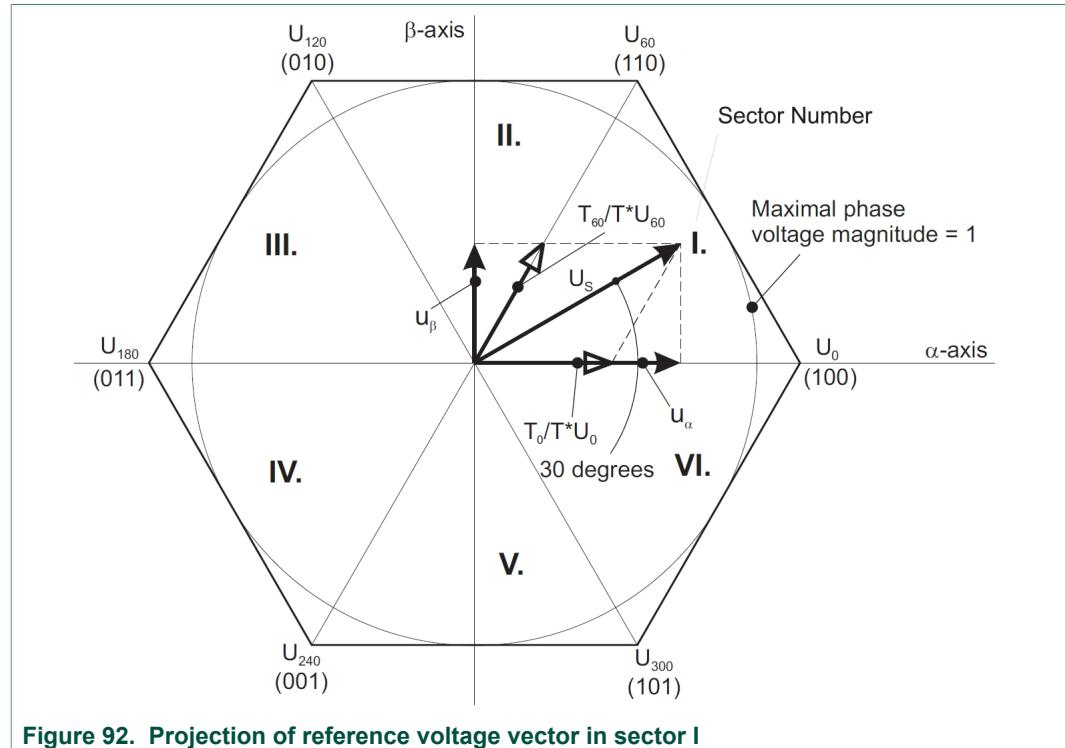
Equation GMCLIB\_SvmStd\_Eq2

The three-phase stator voltages,  $U_a$ ,  $U_b$ , and  $U_c$ , are transformed using the Clarke Transformation into the  $U_\alpha$  and the  $U_\beta$  components of the two-phase orthogonal coordinate system. The transformation results are listed in [Table 283](#).

**Table 283. Switching patterns and space vectors**

a	b	c	$u_\alpha$	$u_\beta$	Vector
0	0	0	0	0	$O_{000}$
1	0	0	$2/3 \cdot U_{DCBus}$	0	$U_0$
1	1	0	$1/3 \cdot U_{DCBus}$	$1/\sqrt{3} \cdot U_{DCBus}$	$U_{60}$
0	1	0	$-1/3 \cdot U_{DCBus}$	$1/\sqrt{3} \cdot U_{DCBus}$	$U_{120}$
0	1	1	$-2/3 \cdot U_{DCBus}$	0	$U_{180}$
0	0	1	$-1/3 \cdot U_{DCBus}$	$-1/\sqrt{3} \cdot U_{DCBus}$	$U_{240}$
1	0	1	$1/3 \cdot U_{DCBus}$	$-1/\sqrt{3} \cdot U_{DCBus}$	$U_{300}$
1	1	1	0	0	$O_{111}$

[Figure 91](#) graphically depicts some feasible basic switching states (vectors). It is clear that there are six non-zero vectors  $U_0$ ,  $U_{60}$ ,  $U_{120}$ ,  $U_{180}$ ,  $U_{240}$ ,  $U_{300}$ , and two zero vectors  $O_{111}$ ,  $O_{000}$ , usable for switching. Therefore, the principle of the Standard Space Vector Modulation resides in applying appropriate switching states for a certain time and thus generating a voltage vector identical to the reference one.

**Figure 92. Projection of reference voltage vector in sector I**

Referring to that principle, an objective of the Standard Space Vector Modulation is an approximation of the reference stator voltage vector  $U_s$  with an appropriate combination of the switching patterns composed of basic space vectors. The graphical explanation of this objective is shown in [Figure 92](#) and [Figure 93](#).

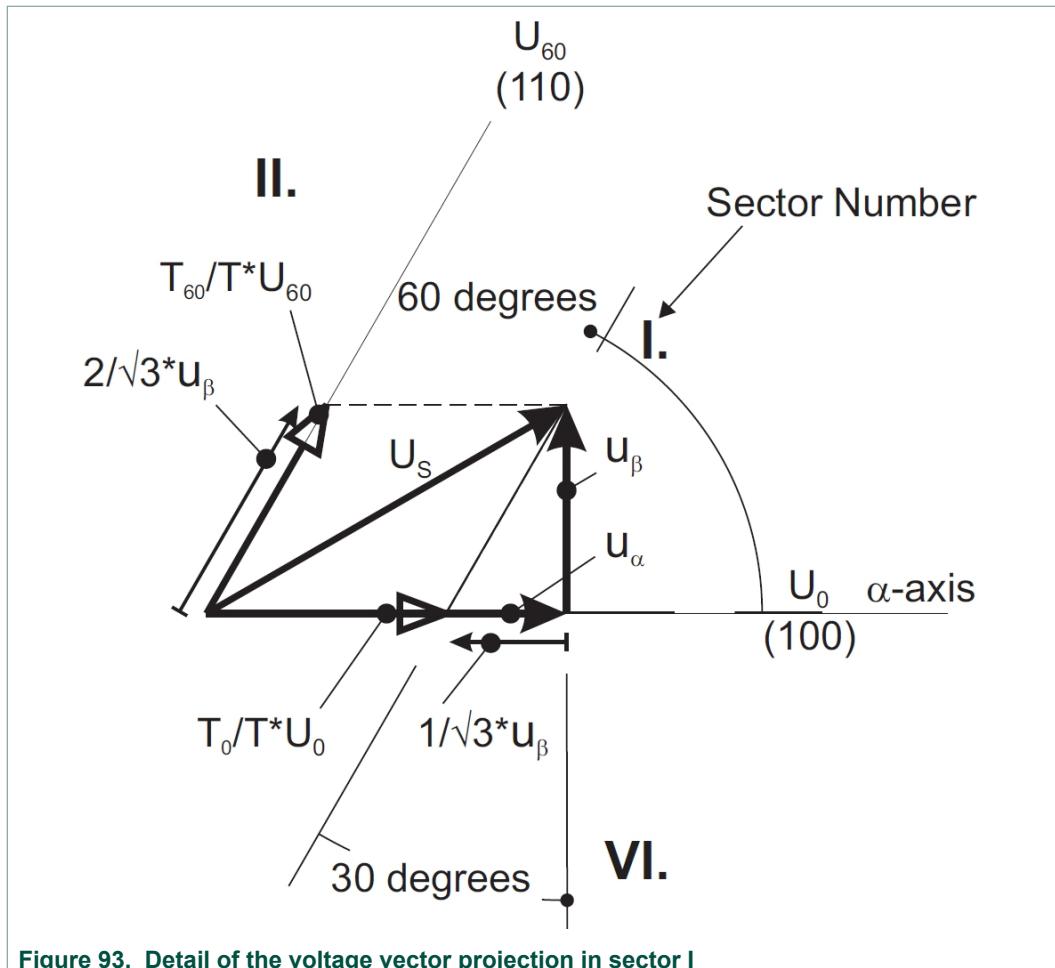


Figure 93. Detail of the voltage vector projection in sector I

The stator reference voltage vector  $U_S$  is phase-advanced by  $30^\circ$  from the axis- $\alpha$  and thus might be generated with an appropriate combination of the adjacent basic switching states  $U_0$  and  $U_{60}$ . These figures also indicate the resultant  $U_\alpha$  and  $U_\beta$  components for space vectors  $U_0$  and  $U_{60}$ .

In this case, the reference stator voltage vector  $U_S$  is located in Sector I and, as previously mentioned, can be generated with the appropriate duty-cycle ratios of the basic switching states  $U_{60}$  and  $U_0$ . The principal equations concerning this vector location are:

$$T = T_{60} + T_0 + T_{\text{null}}$$

Equation GMCLIB\_SvmStd\_Eq3

$$U_S = \frac{T_{60}}{T} \cdot U_{60} + \frac{T_0}{T} \cdot U_0$$

Equation GMCLIB\_SvmStd\_Eq4

where  $T_{60}$  and  $T_0$  are the respective duty-cycle ratios for which the basic space vectors  $U_{60}$  and  $U_0$  should be applied within the time period  $T$ .  $T_{\text{null}}$  is the course of time for which the null vectors  $O_{000}$  and  $O_{111}$  are applied. Those duty-cycle ratios can be calculated using equations:

$$u_\beta = \frac{T_{60}}{T} \cdot |U_{60}| \cdot \sin^{-1} 60^\circ$$

Equation GMCLIB\_SvmStd\_Eq5

$$u_a = \frac{T_0}{T} \cdot |U_0| + \frac{u_\beta}{\tan^{-1} 60^\circ}$$

Equation GMCLIB\_SvmStd\_Eq6

Considering that the normalized magnitudes of the basic space vectors are  $|U_{60}| = |U_0| = 2/\sqrt{3}$  and by substitution of the trigonometric expressions  $\sin(60^\circ)$  and  $\tan(60^\circ)$  by their quantities  $2/\sqrt{3}$  and  $\sqrt{3}$ , respectively, equation [GMCLIB\\_SvmStd\\_Eq5](#) and equation [GMCLIB\\_SvmStd\\_Eq6](#) can be rearranged for the unknown duty-cycle ratios  $T_{60}/T$  and  $T_0/T$ :

$$\frac{T_{60}}{T} = u_\beta$$

Equation GMCLIB\_SvmStd\_Eq7

$$\frac{T_0}{T} = \frac{1}{2} (\sqrt{3} \cdot u_\alpha - u_\beta)$$

Equation GMCLIB\_SvmStd\_Eq8

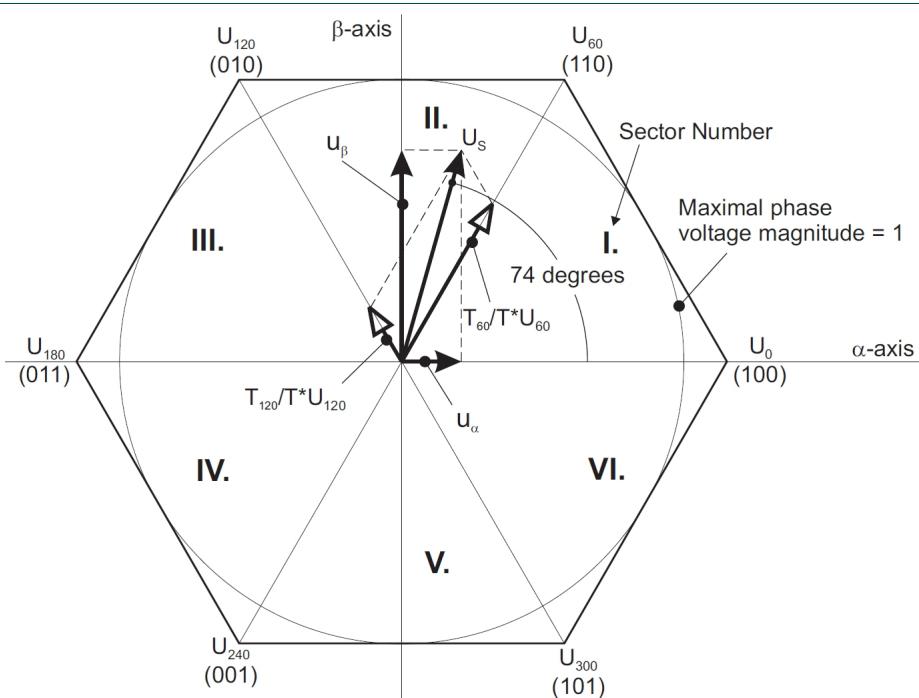


Figure 94. Projection of the reference voltage vector in sector II

Sector II is depicted in [Figure 94](#). In this particular case, the reference stator voltage vector  $U_S$  is generated by the appropriate duty-cycle ratios of the basic switching states  $U_{60}$  and  $U_{120}$ . The basic equations describing this sector are:

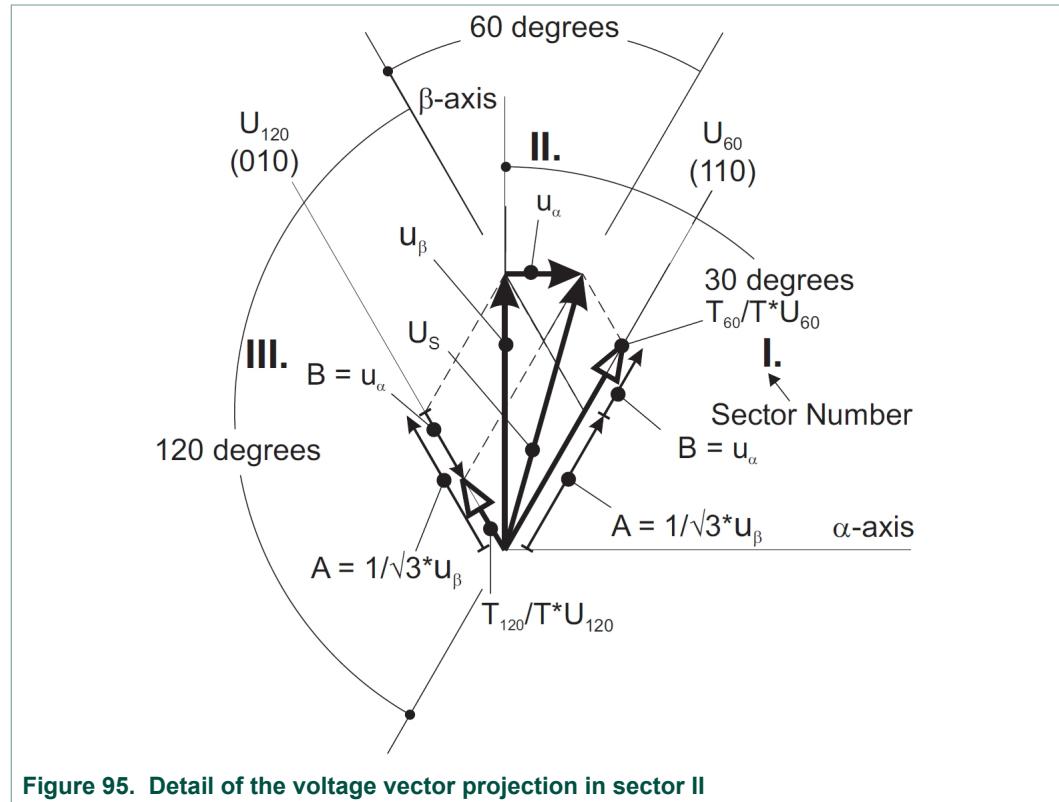
$$T = T_{120} + T_{60} + T_{null}$$

Equation GMCLIB\_SvmStd\_Eq9

$$U_S = \frac{T_{120}}{T} \cdot U_{120} + \frac{T_{60}}{T} \cdot U_{60}$$

Equation GMCLIB\_SvmStd\_Eq10

where  $T_{120}$  and  $T_{60}$  are the respective duty-cycle ratios for which the basic space vectors  $U_{120}$  and  $U_{60}$  should be applied within the time period  $T$ . These resultant duty-cycle ratios are formed from the auxiliary components termed A and B. The graphical representation of the auxiliary components is shown in [Figure 95](#).



**Figure 95. Detail of the voltage vector projection in sector II**

The equations describing those auxiliary time-duration components are:

$$\frac{\sin 130^\circ}{\sin 120^\circ} = \frac{A}{B}$$

Equation GMCLIB\_SvmStd\_Eq11

$$\frac{\sin^{-1}60^\circ}{\sin^{-1}120^\circ} = \frac{B}{u_a}$$

Equation GMCLIB\_SvmStd\_Eq12

Equation [GMCLIB\\_SvmStd\\_Eq11](#) and equation [GMCLIB\\_SvmStd\\_Eq12](#) have been formed using the sine rule. These equations can be rearranged for the calculation of the auxiliary time-duration components A and B. This is done simply by substitution of the trigonometric terms  $\sin(30^\circ)$ ,  $\sin(120^\circ)$  and  $\sin(60^\circ)$  by their numerical representations  $1/2$ ,  $\sqrt{3}/2$  and  $1/\sqrt{3}$ , respectively.

$$A = \frac{1}{\sqrt{3}} \cdot u_\beta$$

Equation GMCLIB\_SvmStd\_Eq13

$$B = u_\alpha$$

Equation GMCLIB\_SvmStd\_Eq14

The resultant duty-cycle ratios,  $T_{120}/T$  and  $T_{60}/T$ , are then expressed in terms of the auxiliary time-duration components defined by equation [GMCLIB\\_SvmStd\\_Eq13](#) and equation [GMCLIB\\_SvmStd\\_Eq14](#), as follows:

$$\frac{T_{120}}{T} \cdot |U_{120}| = A - B$$

Equation GMCLIB\_SvmStd\_Eq15

$$\frac{T_{60}}{T} \cdot |U_{60}| = A + B$$

Equation GMCLIB\_SvmStd\_Eq16

With the help of these equations, and also considering the normalized magnitudes of the basic space vectors to be  $|U_{120}| = |U_{60}| = 2/\sqrt{3}$ , the equations expressed for the unknown duty-cycle ratios of basic space vectors  $T_{120}/T$  and  $T_{60}/T$  can be written:

$$\frac{T_{120}}{T} = \frac{1}{2} \left( u_\beta - \sqrt{3} \cdot u_\alpha \right)$$

Equation GMCLIB\_SvmStd\_Eq17

$$\frac{T_{60}}{T} = \frac{1}{2} \left( u_\beta + \sqrt{3} \cdot u_\alpha \right)$$

Equation GMCLIB\_SvmStd\_Eq18

The duty-cycle ratios in remaining sectors can be derived using the same approach. The resulting equations will be similar to those derived for Sector I and Sector II.

To depict duty-cycle ratios of the basic space vectors for all sectors, we define:

- Three auxiliary variables:

$$X = u_\beta$$

Equation GMCLIB\_SvmStd\_Eq19

$$Y = \frac{1}{2} (u_\beta + \sqrt{3} \cdot u_\alpha)$$

Equation GMCLIB\_SvmStd\_Eq20

$$Z = \frac{1}{2} (u_\beta - \sqrt{3} \cdot u_\alpha)$$

Equation GMCLIB\_SvmStd\_Eq21

Two expressions  $t_1$  and  $t_2$  generally represent duty-cycle ratios of the basic space vectors in the respective sector; e.g., for the first sector,  $t_1$  and  $t_2$  represent duty-cycle ratios of the basic space vectors  $U_{60}$  and  $U_0$ ; for the second sector,  $t_1$  and  $t_2$  represent duty-cycle ratios of the basic space vectors  $U_{120}$  and  $U_{60}$ , etc.

For each sector, the expressions  $t_1$  and  $t_2$ , in terms of auxiliary variables  $X$ ,  $Y$  and  $Z$ , are listed in [Table 284](#).

**Table 284. Determination of  $t_1$  and  $t_2$  expressions**

Sector	$U_0, U_{60}$	$U_{60}, U_{120}$	$U_{120}, U_{180}$	$U_{180}, U_{240}$	$U_{240}, U_{300}$	$U_{300}, U_0$
$t_1$	$X$	$Y$	$-Y$	$Z$	$-Z$	$-X$
$t_2$	$-Z$	$Z$	$X$	$-X$	$-Y$	$Y$

For the determination of auxiliary variables  $X$  equation [GMCLIB\\_SvmStd\\_Eq19](#),  $Y$  equation [GMCLIB\\_SvmStd\\_Eq20](#) and  $Z$  equation [GMCLIB\\_SvmStd\\_Eq21](#), the sector number is required. This information can be obtained by several approaches. One approach discussed here requires the use of a modified Inverse Clark Transformation to transform the direct- $\alpha$  and quadrature- $\beta$  components into a balanced three-phase quantity  $u_{ref1}$ ,  $u_{ref2}$  and  $u_{ref3}$ , used for a straightforward calculation of the sector number, to be shown later.

$$u_{ref1} = u_\beta$$

Equation GMCLIB\_SvmStd\_Eq22

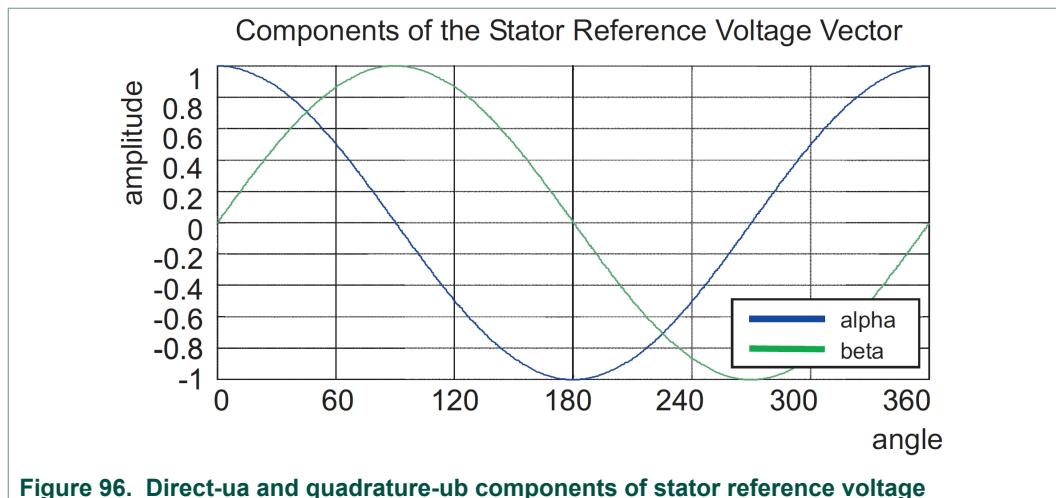
$$u_{ref2} = \frac{1}{2} (-u_\beta + \sqrt{3} \cdot u_\alpha)$$

Equation GMCLIB\_SvmStd\_Eq23

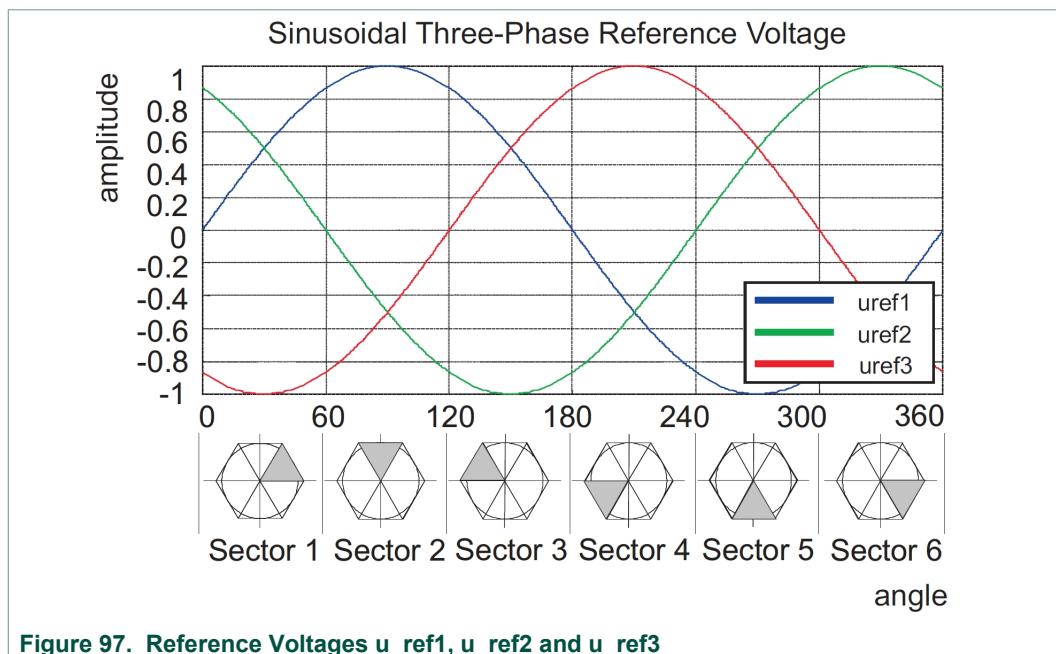
$$u_{ref3} = \frac{1}{2} (-u_\beta - \sqrt{3} \cdot u_\alpha)$$

Equation GMCLIB\_SvmStd\_Eq24

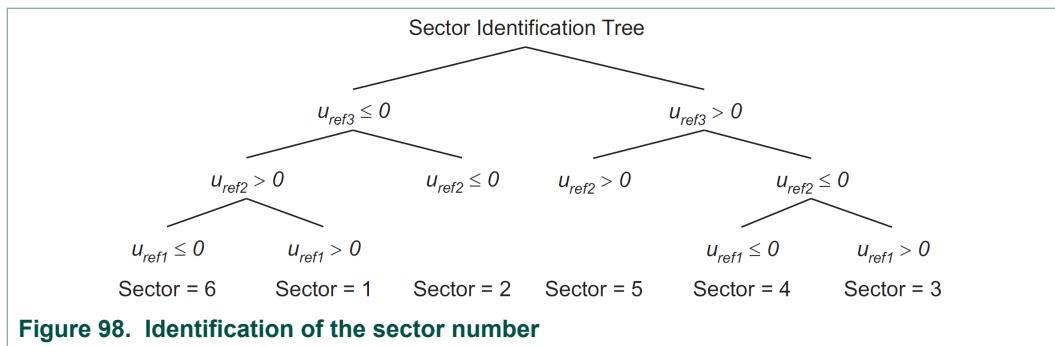
The modified Inverse Clark Transformation projects the quadrature- $u_\beta$  component into  $u_{ref1}$ , as shown in [Figure 96](#) and [Figure 97](#), whereas voltages generated by the conventional Inverse Clark Transformation project the  $u_\alpha$  component into  $u_{ref1}$ .

**Figure 96. Direct- $u_\alpha$  and quadrature- $u_\beta$  components of stator reference voltage**

[Figure 96](#) depicts the  $u_\alpha$  and  $u_\beta$  components of the stator reference voltage vector  $U_S$  that were calculated by the equations  $u_\alpha = \cos(\theta)$  and  $u_\beta = \sin(\theta)$ , respectively.

**Figure 97. Reference Voltages  $u_{\text{ref}1}$ ,  $u_{\text{ref}2}$  and  $u_{\text{ref}3}$** 

The Sector Identification Tree, shown in [Figure 98](#), can be a numerical solution of the approach shown in [Figure 97](#).



It should be pointed out that, in the worst case, three simple comparisons are required to precisely identify the sector of the stator reference voltage vector. For example, if the stator reference voltage vector resides according to the one shown in [Figure 92](#), the stator reference voltage vector is phase-advanced by 30° from the  $\alpha$ -axis, which results in the positive quantities of  $u_{ref1}$  and  $u_{ref2}$  and the negative quantity of  $u_{ref3}$ ; refer to [Figure 97](#). If these quantities are used as the inputs to the Sector Identification Tree, the product of those comparisons will be Sector I. Using the same approach identifies Sector II, if the stator reference voltage vector is located according to the one shown in [Figure 95](#). The variables  $t_1$ ,  $t_2$  and  $t_3$ , representing the switching duty-cycle ratios of the respective three-phase system, are given by the following equations:

$$t_1 = \frac{T-t_1-t_2}{2}$$

Equation GMCLIB\_SvmStd\_Eq25

$$t_2 = t_1 + t_2$$

Equation GMCLIB\_SvmStd\_Eq26

$$t_3 = t_2 + t_3$$

Equation GMCLIB\_SvmStd\_Eq27

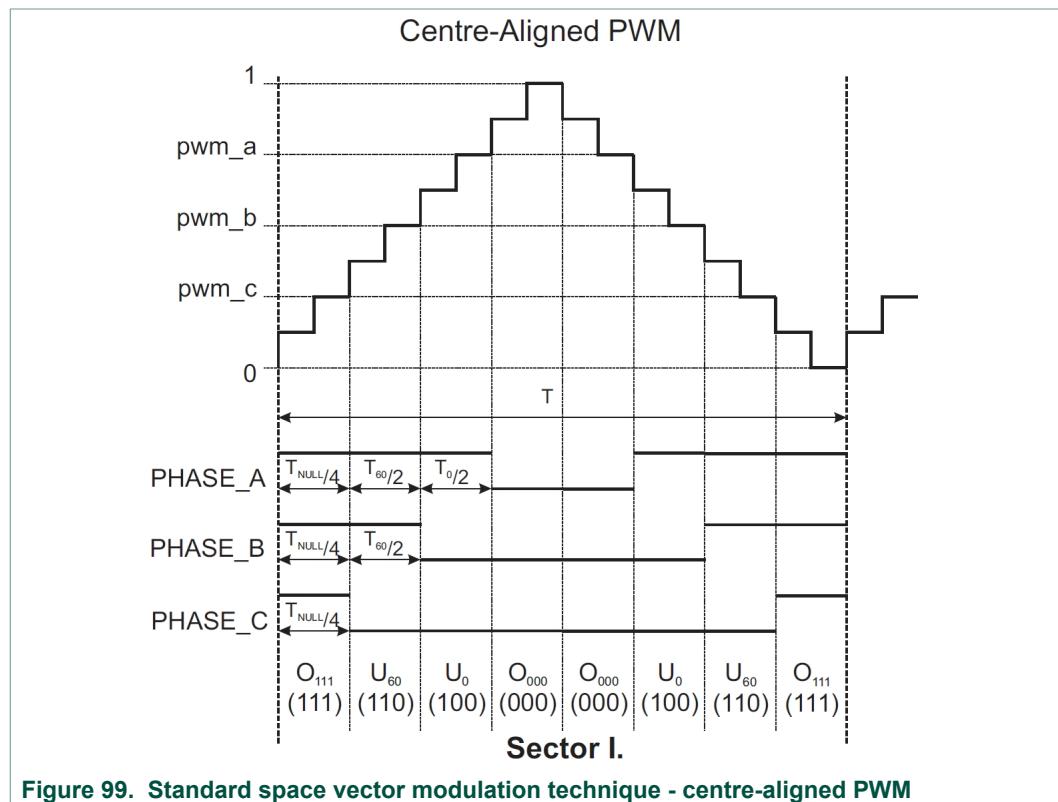
where  $T$  is the switching period,  $t_1$  and  $t_2$  are the duty-cycle ratios (see [Table 284](#)) of the basic space vectors, given for the respective sector. Equation [GMCLIB\\_SvmStd\\_Eq25](#), equation [GMCLIB\\_SvmStd\\_Eq26](#) and equation [GMCLIB\\_SvmStd\\_Eq27](#) are specific solely to the Standard Space Vector Modulation technique; consequently, other Space Vector Modulation techniques discussed later will require deriving different equations.

The next step is to assign the correct duty-cycle ratios,  $t_1$ ,  $t_2$  and  $t_3$ , to the respective motor phases. This is a simple task, accomplished in view of the position of the stator reference voltage vector as shown in [Table 285](#).

**Table 285. Assignment of the duty-cycle ratios to motor phases**

Sector	$U_0, U_{60}$	$U_{60}, U_{120}$	$U_{120}, U_{180}$	$U_{180}, U_{240}$	$U_{240}, U_{300}$	$U_{300}, U_0$
pwm <sub>a</sub>	$t_3$	$t_2$	$t_1$	$t_1$	$t_2$	$t_3$
pwm <sub>b</sub>	$t_2$	$t_3$	$t_3$	$t_2$	$t_1$	$t_1$
pwm <sub>c</sub>	$t_1$	$t_1$	$t_2$	$t_3$	$t_3$	$t_2$

The principle of the Space Vector Modulation technique consists in applying the basic voltage vectors  $U_{XXX}$  and  $O_{XXX}$  for the certain time in such a way that the mean vector, generated by the Pulse Width Modulation approach for the period  $T$ , is equal to the original stator reference voltage vector  $U_S$ . This provides a great variability of the arrangement of the basic vectors during the PWM period  $T$ . Those vectors might be arranged either to lower switching losses or to achieve diverse results, such as centre-aligned PWM, edge-aligned PWM or a minimal number of switching states. A brief discussion of the widely-used centre-aligned PWM follows. Generating the centre-aligned PWM pattern is accomplished practically by comparing the threshold levels,  $pwm_a$ ,  $pwm_b$  and  $pwm_c$  with a free-running up-down counter. The timer counts to a 1 (representing the maximum counter value) and then down to a 0. It is supposed that when a threshold level is larger than the timer value, the respective PWM output is active. Otherwise, it is inactive; see [Figure 99](#)



**Figure 99. Standard space vector modulation technique - centre-aligned PWM**

**Note:** The input/output pointers must contain valid addresses, otherwise a fault may occur (MemManage, BusFault, UsageFault, HardFault).

### Re-entrancy

The function is re-entrant.

#### 2.51.1 Function GMCLIB\_SvmStd\_F32

##### Declaration

```
tU32 GMCLIB_SvmStd_F32(SWLIBS_3Syst_F32 *pOut, const
                           SWLIBS_2Syst_F32 *const pIn);
```

**Arguments****Table 286. GMCLIB\_SvmStd\_F32 arguments**

Type	Name	Direction	Description
<a href="#">SWLBS_3Syst_F32</a> *	pOut	<a href="#">input, output</a>	Pointer to the structure containing calculated duty-cycle ratios of the 3-Phase system.
const <a href="#">SWLBS_2Syst_F32</a> *const	pln	<a href="#">input</a>	Pointer to the structure containing direct $U_\alpha$ and quadrature $U_\beta$ components of the stator voltage vector.

**Return**

The function returns a 32-bit value in format INT, representing the actual space sector which contains the stator reference vector  $U_s$ .

**Code Example**

```
#include "gmclib.h"
#define U_MAX 15

SWLBS\_2Syst\_F32 tr32InVoltage;
SWLBS\_3Syst\_F32 tr32PwmABC;
tU32 u32SvmSector;

void main(void)
{
    // Input voltage vector 15V @ angle 30deg
    // alpha component of input voltage vector = 12.99[V]
    // beta component of input voltage vector = 7.5[V]
    tr32InVoltage.f32Arg1 = FRAC32(12.99/U_MAX);
    tr32InVoltage.f32Arg2 = FRAC32(7.5/U_MAX);

    // output pwm dutycycles stored in structure referenced by tr32PwmABC
    // pwmA dutycycle = 0x7FFF A2C9 = FRAC32(0.9999888...)
    // pwmb dutycycle = 0x4000 5D35 = FRAC32(0.5000111...)
    // pwmc dutycycle = 0x0000 5D35 = FRAC32(0.0000111...)
    // svmSector = 0x1 [sector]
    u32SvmSector = GMCLIB_SvmStd_F32(&tr32PwmABC,&tr32InVoltage);

    // output pwm dutycycles stored in structure referenced by tr32PwmABC
    // pwmA dutycycle = 0x7FFF A2C9 = FRAC32(0.9999888...)
    // pwmb dutycycle = 0x4000 5D35 = FRAC32(0.5000111...)
    // pwmc dutycycle = 0x0000 5D35 = FRAC32(0.0000111...)
    // svmSector = 0x1 [sector]
    u32SvmSector = GMCLIB_SvmStd(&tr32PwmABC,&tr32InVoltage,F32);

    // ######
    // Available only if 32-bit fractional implementation selected
    // as default
    // #####
    // output pwm dutycycles stored in structure referenced by tr32PwmABC
    // pwmA dutycycle = 0x7FFF A2C9 = FRAC32(0.9999888...)
    // pwmb dutycycle = 0x4000 5D35 = FRAC32(0.5000111...)
    // pwmc dutycycle = 0x0000 5D35 = FRAC32(0.0000111...)
    // svmSector = 0x1 [sector]
    u32SvmSector = GMCLIB_SvmStd(&tr32PwmABC,&tr32InVoltage);
}
```

## 2.51.2 Function GMCLIB\_SvmStd\_F16

### Declaration

```
tU16 GMCLIB_SvmStd_F16(SWLIBS_3Syst_F16 *pOut, const
SWLIBS_2Syst_F16 *const pIn);
```

### Arguments

Table 287. GMCLIB\_SvmStd\_F16 arguments

Type	Name	Direction	Description
<code>SWLIBS_3Syst_F16 *</code>	pOut	<code>input, output</code>	Pointer to the structure containing calculated duty-cycle ratios of the 3-Phase system.
<code>const SWLIBS_2Syst_F16 *const</code>	pIn	<code>input</code>	Pointer to the structure containing direct $U_\alpha$ and quadrature $U_\beta$ components of the stator voltage vector.

### Return

The function returns a 16-bit value in format INT, representing the actual space sector which contains the stator reference vector  $U_s$ .

### Code Example

```
#include "gmclib.h"
#define U_MAX 15

SWLIBS_2Syst_F16 tr16InVoltage;
SWLIBS_3Syst_F16 tr16PwmABC;
tU16 u16SvmSector;

void main(void)
{
    // Input voltage vector 15V @ angle 30deg
    // alpha component of input voltage vector = 12.99[V]
    // beta component of input voltage vector = 7.5[V]
    tr16InVoltage.f16Arg1 = FRAC16(12.99/U_MAX);
    tr16InVoltage.f16Arg2 = FRAC16(7.5/U_MAX);

    // output pwm dutycycles stored in structure referenced by tr16PwmABC
    // pwmA dutycycle = 0x7FFF = FRAC16(0.9999...)
    // pwmb dutycycle = 0x4000 = FRAC16(0.5000...)
    // pwmc dutycycle = 0x0000 = FRAC16(0.0000...)
    // svmSector = 0x1 [sector]
    u16SvmSector = GMCLIB_SvmStd_F16(&tr16PwmABC,&tr16InVoltage);

    // output pwm dutycycles stored in structure referenced by tr16PwmABC
    // pwmA dutycycle = 0x7FFF = FRAC16(0.9999...)
    // pwmb dutycycle = 0x4000 = FRAC16(0.5000...)
    // pwmc dutycycle = 0x0000 = FRAC16(0.0000...)
    // svmSector = 0x1 [sector]
    u16SvmSector = GMCLIB_SvmStd(&tr16PwmABC,&tr16InVoltage,F16);

    // #####
    // Available only if 16-bit fractional implementation selected
    // as default
    // #####
}
```

```

// output pwm dutycycles stored in structure referenced by tr16PwmABC
// pwmA dutycycle = 0x7FFF = FRAC16(0.9999...)
// pwmb dutycycle = 0x4000 = FRAC16(0.5000...)
// pwmc dutycycle = 0x0000 = FRAC16(0.0000...)
// svmSector      = 0x1 [sector]
u16SvmSector = GMCLIB_SvmStd(&tr16PwmABC,&tr16InVoltage);
}

```

### 2.51.3 Function GMCLIB\_SvmStd\_FLT

#### Declaration

```
tU32 GMCLIB_SvmStd_FLT(SWLIBS\_3Syst\_FLT *pOut, const
SWLIBS\_2Syst\_FLT *const pIn);
```

#### Arguments

**Table 288. GMCLIB\_SvmStd\_FLT arguments**

Type	Name	Direction	Description
<a href="#">SWLIBS_3Syst_FLT</a> *	pOut	input, output	Pointer to the structure containing calculated duty-cycle ratios of the 3-Phase system.
const <a href="#">SWLIBS_2Syst_FLT</a> *const	pIn	input	Pointer to the structure containing direct $U_\alpha$ and quadrature $U_\beta$ components of the stator voltage vector.

#### Return

The function returns a 32-bit value in format INT, representing the actual space sector which contains the stator reference vector  $U_s$ .

#### Implementation details

The function presumes that the input voltages are normalized to fit the range <-1; 1>. In a typical motor control application, this function is preceded by the function [GMCLIB\\_ElimDcBusRip\\_FLT](#) which ensures that the voltages are correctly normalized.

**Note:** The function may raise floating-point exceptions (invalid operation, overflow, underflow, inexact, input denormal). The floating-point unit must be enabled in CPACR register to prevent NOCP UsageFault exception.

#### Code Example

```

#include "gmclib.h"
#define U_MAX 15

SWLIBS\_2Syst\_FLT tInVoltage;
SWLIBS\_3Syst\_FLT tPwmABC;
tU32 u32SvmSector;

void main(void)
{
    // Input voltage vector 15V @ angle 30deg
    // alpha component of input voltage vector = 12.99[V]
    // beta component of input voltage vector = 7.5[V]
    tInVoltage.fltArg1 = (tFloat) (12.99/U_MAX);
    tInVoltage.fltArg2 = (tFloat) (7.5/U_MAX);
}

```

```

// output pwm dutycycles stored in structure referenced by tr32PwmABC
// pwmA dutycycle = (tFloat) (0.9999888)
// pwmb dutycycle = (tFloat) (0.5000111)
// pwmc dutycycle = (tFloat) (0.0000111)
// u32SvmSector = 0x1 [sector]
u32SvmSector = GMCLIB_SvmStd_FLT(&tPwmABC, &tInVoltage);

// output pwm dutycycles stored in structure referenced by tr32PwmABC
// pwmA dutycycle = (tFloat) (0.9999888)
// pwmb dutycycle = (tFloat) (0.5000111)
// pwmc dutycycle = (tFloat) (0.0000111)
// u32SvmSector = 0x1 [sector]
u32SvmSector = GMCLIB_SvmStd(&tPwmABC, &tInVoltage, FLT);

// ######
// Available only if single precision floating point
// implementation selected as default
// #####
// #####
// output pwm dutycycles stored in structure referenced by tr32PwmABC
// pwmA dutycycle = (tFloat) (0.9999888)
// pwmb dutycycle = (tFloat) (0.5000111)
// pwmc dutycycle = (tFloat) (0.0000111)
// u32SvmSector = 0x1 [sector]
u32SvmSector = GMCLIB_SvmStd(&tPwmABC, &tInVoltage);
}

```

## 2.52 Function GMCLIB\_SvmU0n

This function calculates the duty-cycle ratios using the special Space Vector Modulation technique, termed Space Vector Modulation with O<sub>000</sub> Nulls.

### Description

The GMCLIB\_SvmU0n function for calculating duty-cycle ratios is widely-used in the modern electric drive. This function calculates appropriate duty-cycle ratios, which are needed for generating the given stator reference voltage vector using a special Space Vector Modulation technique, termed Space Vector Modulation with O<sub>000</sub> Nulls.

The derivation approach of the Space Vector Modulation technique with O<sub>000</sub> Nulls is identical, in many aspects, to the approach presented in [GMCLIB\\_SvmStd](#). However, a distinct difference lies in the definition of the variables t<sub>1</sub>, t<sub>2</sub> and t<sub>3</sub> that represents switching duty-cycle ratios of the respective phases:

$$t_1 = 0$$

Equation GMCLIB\_SvmU0n\_Eq1

$$t_2 = t_1 + t_{-1}$$

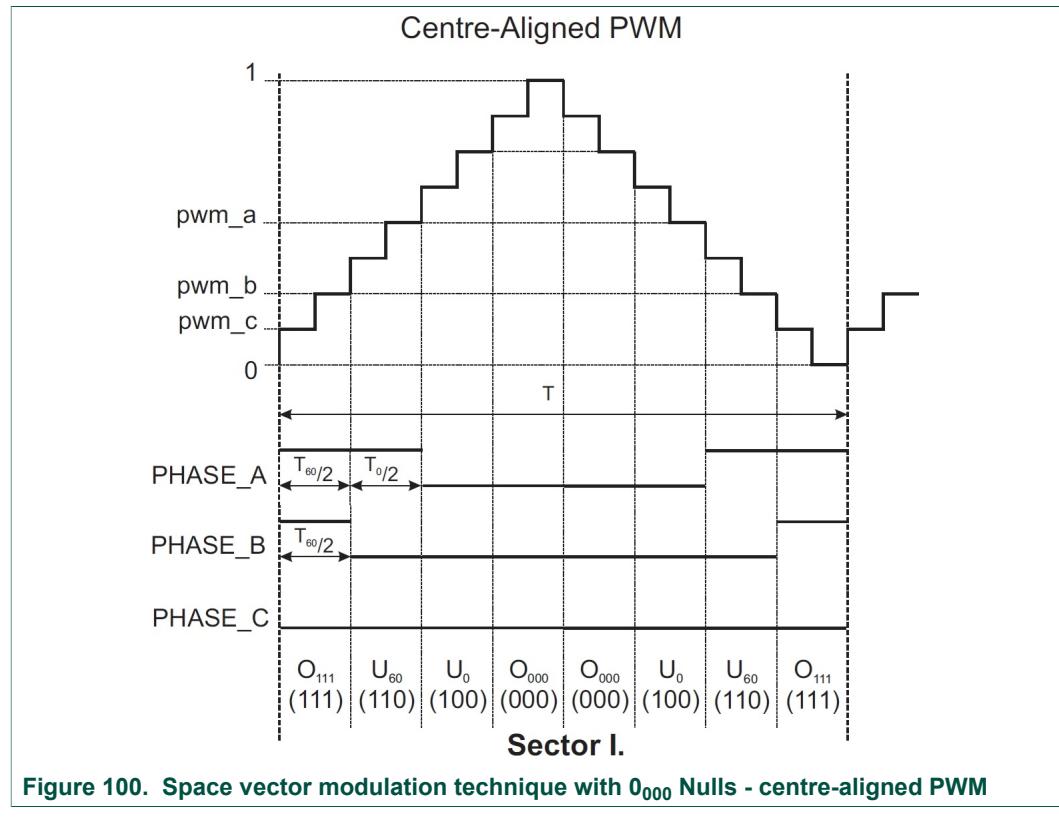
Equation GMCLIB\_SvmU0n\_Eq2

$$t_3 = t_2 + t_{-2}$$

Equation GMCLIB\_SvmU0n\_Eq3

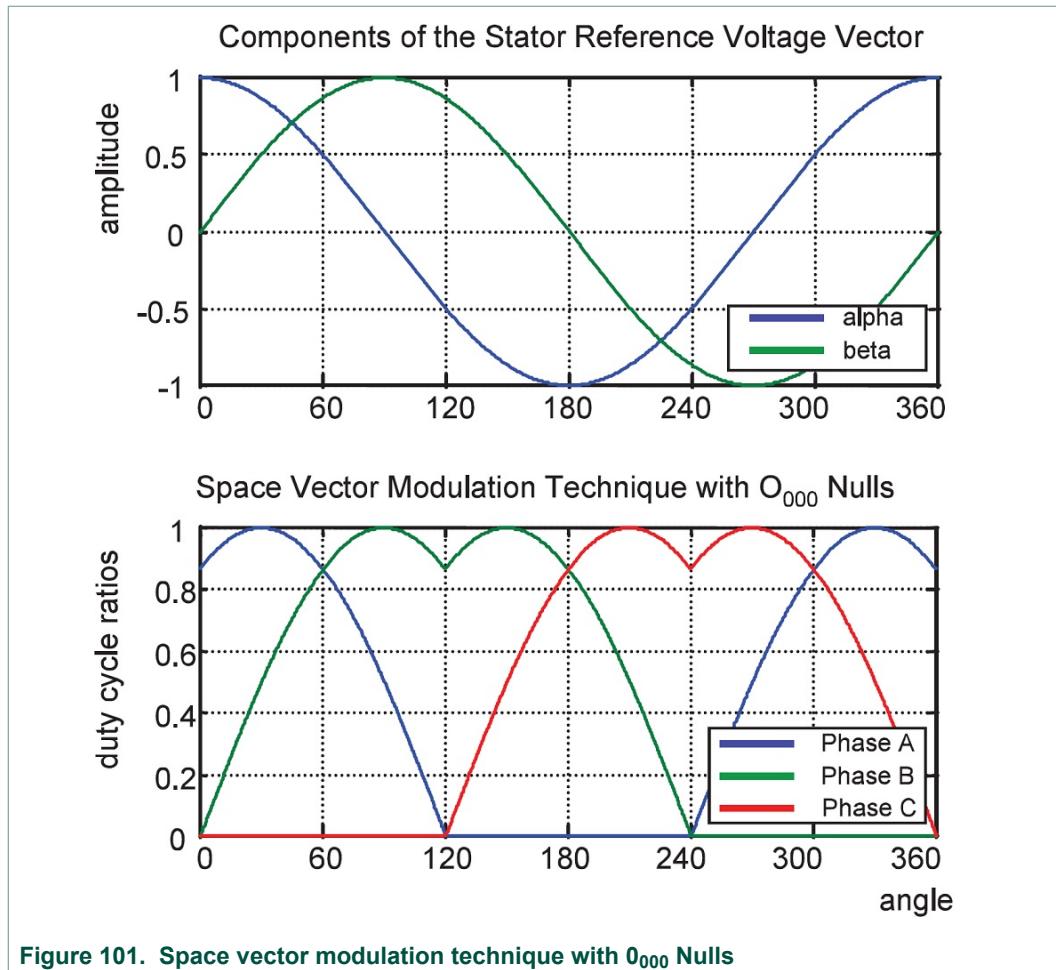
where T is the switching period and  $t_{-1}$  and  $t_{-2}$  are duty-cycle ratios of basic space vectors that are defined for the respective sector in [Table 284](#).

The generally-used center-aligned PWM is discussed briefly in the following sections. Generating the centre-aligned PWM pattern is accomplished practically by comparing the threshold levels,  $pwm_a$ ,  $pwm_b$  and  $pwm_c$  with a free-running up-down counter. The timer counts to a 1 (representing the maximum counter value) and then down to a 0. It is supposed that when a threshold level is larger than the timer value, the respective PWM output is active. Otherwise, it is inactive; see [Figure 100](#)



**Figure 100.** Space vector modulation technique with  $O_{000}$  Nulls - centre-aligned PWM

[Figure 101](#) shows calculated waveforms of the duty cycle ratios using Space Vector Modulation with  $O_{000}$  Nulls.

Figure 101. Space vector modulation technique with O<sub>000</sub> Nulls

**Note:** The input/output pointers must contain valid addresses, otherwise a fault may occur (MemManage, BusFault, UsageFault, HardFault).

### Re-entrancy

The function is re-entrant.

#### 2.52.1 Function GMCLIB\_SvmU0n\_F32

##### Declaration

```
tU32 GMCLIB_SvmU0n_F32(SWLIBS_3Syst_F32 *pOut, const
                           SWLIBS_2Syst_F32 *const pIn);
```

##### Arguments

Table 289. GMCLIB\_SvmU0n\_F32 arguments

Type	Name	Direction	Description
SWLIBS_3Syst_F32 *	pOut	input, output	Pointer to the structure containing calculated duty-cycle ratios of the 3-Phase system.
const SWLIBS_2Syst_F32 *const	pIn	input	Pointer to the structure containing direct U <sub>α</sub> and quadrature U <sub>β</sub> components of the stator voltage vector.

**Return**

The function returns a 32-bit value in format INT, representing the actual space sector which contains the stator reference vector  $U_s$ .

**Code Example**

```
#include "gmclib.h"
#define U_MAX 15

SWLIBS_2Syst_F32 tr32InVoltage;
SWLIBS_3Syst_F32 tr32PwmABC;
tU32 u32SvmSector;

void main(void)
{
    // Input voltage vector 15V @ angle 30deg
    // alpha component of input voltage vector = 12.99[V]
    // beta component of input voltage vector = 7.5[V]
    tr32InVoltage.f32Arg1 = FRAC32(12.99/U_MAX);
    tr32InVoltage.f32Arg2 = FRAC32(7.5/U_MAX);

    // output pwm dutycycles stored in structure referenced by tr32PwmABC
    // pwmA dutycycle = 0x7FFF 4772 = FRAC32(0.9999788...)
    // pwmB dutycycle = 0x4000 0000 = FRAC32(0.5000000...)
    // pwmC dutycycle = 0x0000 0000 = FRAC32(0.0000000...)
    // svmSector = 0x1 [sector]
    u32SvmSector = GMCLIB_SvmU0n_F32(&tr32PwmABC,&tr32InVoltage);

    // output pwm dutycycles stored in structure referenced by tr32PwmABC
    // pwmA dutycycle = 0x7FFF 4772 = FRAC32(0.9999788...)
    // pwmB dutycycle = 0x4000 0000 = FRAC32(0.5000000...)
    // pwmC dutycycle = 0x0000 0000 = FRAC32(0.0000000...)
    // svmSector = 0x1 [sector]
    u32SvmSector = GMCLIB_SvmU0n(&tr32PwmABC,&tr32InVoltage,F32);

    // ######
    // Available only if 32-bit fractional implementation selected
    // as default
    // #####
}

// output pwm dutycycles stored in structure referenced by tr32PwmABC
// pwmA dutycycle = 0x7FFF 4772 = FRAC32(0.9999788...)
// pwmB dutycycle = 0x4000 0000 = FRAC32(0.5000000...)
// pwmC dutycycle = 0x0000 0000 = FRAC32(0.0000000...)
// svmSector = 0x1 [sector]
u32SvmSector = GMCLIB_SvmU0n(&tr32PwmABC,&tr32InVoltage);
```

**2.52.2 Function GMCLIB\_SvmU0n\_F16****Declaration**

```
tU16 GMCLIB_SvmU0n_F16(SWLIBS_3Syst_F16 *pOut, const
SWLIBS_2Syst_F16 *const pIn);
```

**Arguments****Table 290. GMCLIB\_SvmUOn\_F16 arguments**

Type	Name	Direction	Description
<a href="#">SWLIBS_3Syst_F16</a> *	pOut	<a href="#">input, output</a>	Pointer to the structure containing calculated duty-cycle ratios of the 3-Phase system.
const <a href="#">SWLIBS_2Syst_F16</a> *const	pln	<a href="#">input</a>	Pointer to the structure containing direct $U_\alpha$ and quadrature $U_\beta$ components of the stator voltage vector.

**Return**

The function returns a 16-bit value in format INT, representing the actual space sector which contains the stator reference vector  $U_s$ .

**Code Example**

```
#include "gmclib.h"
#define U_MAX 15

SWLIBS\_2Syst\_F16 tr16InVoltage;
SWLIBS\_3Syst\_F16 tr16PwmABC;
tU16 u16SvmSector;

void main(void)
{
    // Input voltage vector 15V @ angle 30deg
    // alpha component of input voltage vector = 12.99[V]
    // beta component of input voltage vector = 7.5[V]
    tr16InVoltage.f16Arg1 = FRAC16(12.99/U_MAX);
    tr16InVoltage.f16Arg2 = FRAC16(7.5/U_MAX);

    // output pwm dutycycles stored in structure referenced by tr16PwmABC
    // pwmA dutycycle = 0x7FFF = FRAC16(0.9999...)
    // pwmb dutycycle = 0x4000 = FRAC16(0.5000...)
    // pwmc dutycycle = 0x0000 = FRAC16(0.0000...)
    // svmSector = 0x1 [sector]
    u16SvmSector = GMCLIB_SvmUOn_F16(&tr16PwmABC,&tr16InVoltage);

    // output pwm dutycycles stored in structure referenced by tr16PwmABC
    // pwmA dutycycle = 0x7FFF = FRAC16(0.9999...)
    // pwmb dutycycle = 0x4000 = FRAC16(0.5000...)
    // pwmc dutycycle = 0x0000 = FRAC16(0.0000...)
    // svmSector = 0x1 [sector]
    u16SvmSector = GMCLIB_SvmUOn(&tr16PwmABC,&tr16InVoltage,F16);

    // ######
    // Available only if 16-bit fractional implementation selected
    // as default
    // #####
    // output pwm dutycycles stored in structure referenced by tr16PwmABC
    // pwmA dutycycle = 0x7FFF = FRAC16(0.9999...)
    // pwmb dutycycle = 0x4000 = FRAC16(0.5000...)
    // pwmc dutycycle = 0x0000 = FRAC16(0.0000...)
    // svmSector = 0x1 [sector]
    u16SvmSector = GMCLIB_SvmUOn(&tr16PwmABC,&tr16InVoltage);
}
```

### 2.52.3 Function GMCLIB\_SvmU0n\_FLT

#### Declaration

```
tU32 GMCLIB_SvmU0n_FLT(SWLIBS\_3Syst\_FLT *pOut, const
SWLIBS\_2Syst\_FLT *const pIn);
```

#### Arguments

Table 291. GMCLIB\_SvmU0n\_FLT arguments

Type	Name	Direction	Description
<a href="#">SWLIBS_3Syst_FLT</a> *	pOut	input, output	Pointer to the structure containing calculated duty-cycle ratios of the 3-Phase system.
const <a href="#">SWLIBS_2Syst_FLT</a> *const	pIn	input	Pointer to the structure containing direct $U_\alpha$ and quadrature $U_\beta$ components of the stator voltage vector.

#### Return

The function returns a 32-bit value in format INT, representing the actual space sector which contains the stator reference vector  $U_s$ .

#### Implementation details

The function presumes that the input voltages are normalized to fit the range <-1; 1>. In a typical motor control application, this function is preceded by the function [GMCLIB\\_ElimDcBusRip\\_FLT](#) which ensures that the voltages are correctly normalized.

**Note:** The function may raise floating-point exceptions (invalid operation, overflow, underflow, inexact, input denormal). The floating-point unit must be enabled in CPACR register to prevent NOCP UsageFault exception.

#### Code Example

```
#include "gmclib.h"
#define U_MAX 15

SWLIBS\_2Syst\_FLT tInVoltage;
SWLIBS\_3Syst\_FLT tPwmABC;
tU32 u32SvmSector;

void main(void)
{
    // Input voltage vector 15V @ angle 30deg
    // alpha component of input voltage vector = 12.99[V]
    // beta component of input voltage vector = 7.5[V]
    tInVoltage.fltArg1 = (tFloat) (12.99/U_MAX);
    tInVoltage.fltArg2 = (tFloat) (7.5/U_MAX);

    // output pwm dutycycles stored in structure referenced by tr32PwmABC
    // pwmA dutycycle = (tFloat) (0.9999780)
    // pwmb dutycycle = (tFloat) (0.5000000)
    // pwmc dutycycle = (tFloat) (0.0000000)
    // u32SvmSector = 0x1 [sector]
    u32SvmSector = GMCLIB_SvmU0n_FLT(&tPwmABC, &tInVoltage);

    // output pwm dutycycles stored in structure referenced by tr32PwmABC
```

```

// pwmA dutycycle = (tFloat) (0.9999780)
// pwmB dutycycle = (tFloat) (0.5000000)
// pwmC dutycycle = (tFloat) (0.0000000)
// u32SvmSector = 0x1 [sector]
u32SvmSector = GMCLIB_SvmU0n(&tPwmABC, &tInVoltage, FLT);

// ##### Available only if single precision floating point
// implementation selected as default
// #####
// output pwm dutycycles stored in structure referenced by tr32PwmABC
// pwmA dutycycle = (tFloat) (0.9999780)
// pwmB dutycycle = (tFloat) (0.5000000)
// pwmC dutycycle = (tFloat) (0.0000000)
// u32SvmSector = 0x1 [sector]
u32SvmSector = GMCLIB_SvmU0n(&tPwmABC, &tInVoltage);
}

```

## 2.53 Function GMCLIB\_SvmU7n

This function calculates the duty-cycle ratios using the special Space Vector Modulation technique, termed Space Vector Modulation with O<sub>111</sub> Ones.

### Description

The GMCLIB\_SvmU7n function for calculating duty-cycle ratios is widely-used in the modern electric drive. This function calculates appropriate duty-cycle ratios, which are needed for generating the given stator reference voltage vector using a special Space Vector Modulation technique, termed Space Vector Modulation with O<sub>111</sub> Ones.

The derivation approach of the Space Vector Modulation technique with O<sub>111</sub> Ones is identical, in many aspects, to the approach presented in [GMCLIB\\_SvmStd](#). However, a distinct difference lies in the definition of the variables t<sub>1</sub>, t<sub>2</sub> and t<sub>3</sub> that represents switching duty-cycle ratios of the respective phases:

$$t_1 = T - t_{-1} - t_{-2}$$

Equation GMCLIB\_SvmU7n\_Eq1

$$t_2 = t_1 + t_{-1}$$

Equation GMCLIB\_SvmU7n\_Eq2

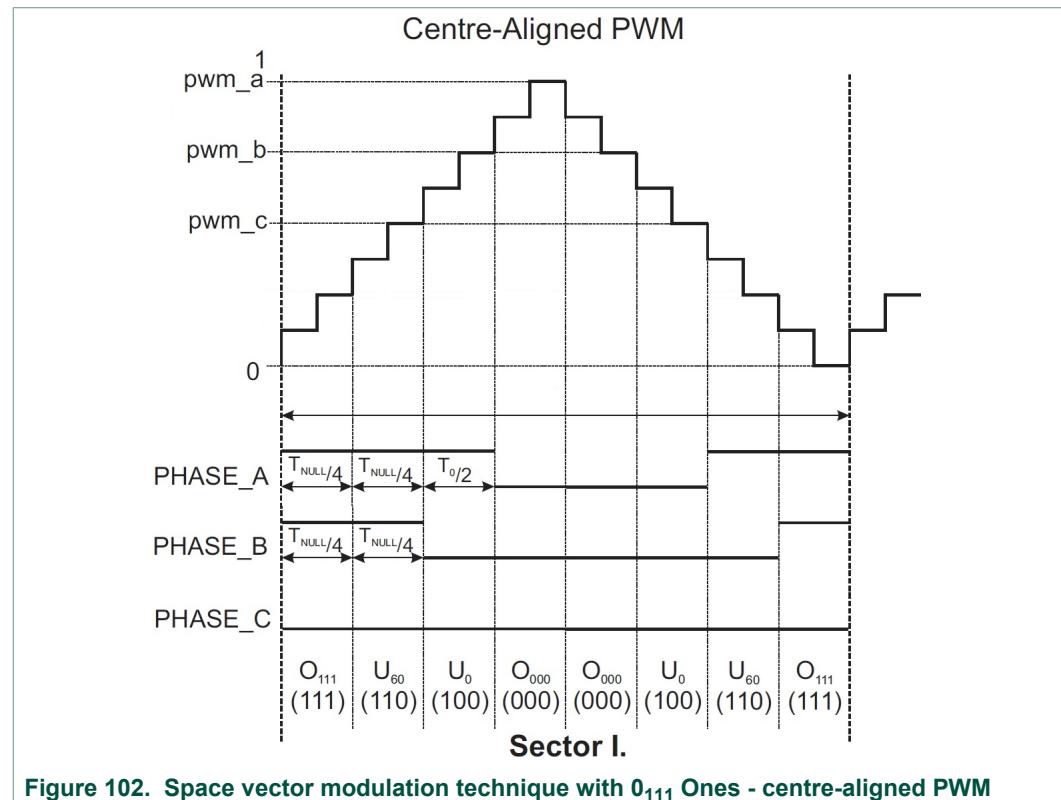
$$t_3 = t_2 + t_{-2}$$

Equation GMCLIB\_SvmU7n\_Eq3

where T is the switching period and t<sub>-1</sub> and t<sub>-2</sub> are duty-cycle ratios of basic space vectors that are defined for the respective sector in [Table 284](#).

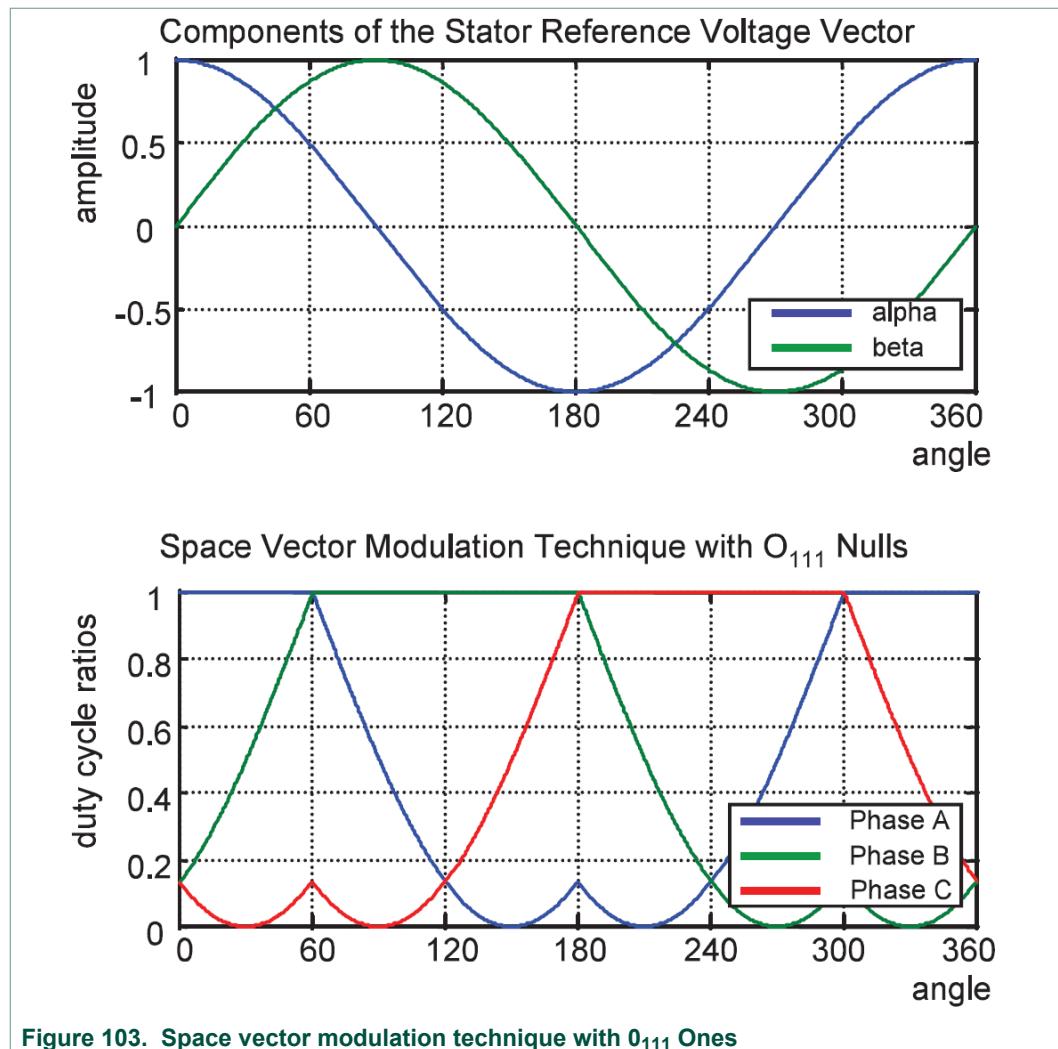
The generally-used center-aligned PWM is discussed briefly in the following sections. Generating the centre-aligned PWM pattern is accomplished practically by comparing the threshold levels, p<sub>m</sub><sub>a</sub>, p<sub>m</sub><sub>b</sub> and p<sub>m</sub><sub>c</sub> with a free-running up-down counter. The

timer counts to a 1 (representing the maximum counter value) and then down to a 0. It is supposed that when a threshold level is larger than the timer value, the respective PWM output is active. Otherwise, it is inactive; see [Figure 102](#)



**Figure 102. Space vector modulation technique with 0<sub>111</sub> Ones - centre-aligned PWM**

[Figure 103](#) shows calculated waveforms of the duty cycle ratios using Space Vector Modulation with O<sub>111</sub> Ones.

Figure 103. Space vector modulation technique with O<sub>111</sub> Ones

**Note:** The input/output pointers must contain valid addresses, otherwise a fault may occur (MemManage, BusFault, UsageFault, HardFault).

## Re-entrancy

The function is re-entrant.

### 2.53.1 Function GMCLIB\_SvmU7n\_F32

#### Declaration

```
tU32 GMCLIB_SvmU7n_F32(SWLBS_3Syst_F32 *pOut, const
SWLBS_2Syst_F32 *const pIn);
```

#### Arguments

Table 292. GMCLIB\_SvmU7n\_F32 arguments

Type	Name	Direction	Description
<a href="#">SWLBS_3Syst_F32</a> *	pOut	input, output	Pointer to the structure containing calculated duty-cycle ratios of the 3-Phase system.

Type	Name	Direction	Description
const <a href="#">SWLIBS_2Syst_F32</a> *const	pln	input	Pointer to the structure containing direct $U_\alpha$ and quadrature $U_\beta$ components of the stator voltage vector.

**Return**

The function returns a 32-bit value in format INT, representing the actual space sector which contains the stator reference vector  $U_s$ .

**Code Example**

```
#include "gmclib.h"
#define U_MAX 15

SWLIBS\_2Syst\_F32 tr32InVoltage;
SWLIBS\_3Syst\_F32 tr32PwmABC;
tU32 u32SvmSector;

void main(void)
{
    // Input voltage vector 15V @ angle 30deg
    // alpha component of input voltage vector = 12.99[V]
    // beta component of input voltage vector = 7.5[V]
    tr32InVoltage.f32Arg1 = FRAC32(12.99/U_MAX);
    tr32InVoltage.f32Arg2 = FRAC32(7.5/U_MAX);

    // output pwm dutycycles stored in structure referenced by tr32PwmABC
    // pwmA dutycycle = 0x7FFF FFFF = FRAC32(1.0000000... )
    // pwmb dutycycle = 0x4000 B88D = FRAC32(0.5000220... )
    // pwmc dutycycle = 0x0000 B88D = FRAC32(0.0000220... )
    // svmSector = 0x1 [sector]
    u32SvmSector = GMCLIB_SvmU7n_F32(&tr32PwmABC,&tr32InVoltage);

    // output pwm dutycycles stored in structure referenced by tr32PwmABC
    // pwmA dutycycle = 0x7FFF FFFF = FRAC32(1.0000000... )
    // pwmb dutycycle = 0x4000 B88D = FRAC32(0.5000220... )
    // pwmc dutycycle = 0x0000 B88D = FRAC32(0.0000220... )
    // svmSector = 0x1 [sector]
    u32SvmSector = GMCLIB_SvmU7n(&tr32PwmABC,&tr32InVoltage,F32);

    // ######
    // Available only if 32-bit fractional implementation selected
    // as default
    // #####
    // output pwm dutycycles stored in structure referenced by tr32PwmABC
    // pwmA dutycycle = 0x7FFF FFFF = FRAC32(1.0000000... )
    // pwmb dutycycle = 0x4000 B88D = FRAC32(0.5000220... )
    // pwmc dutycycle = 0x0000 B88D = FRAC32(0.0000220... )
    // svmSector = 0x1 [sector]
    u32SvmSector = GMCLIB_SvmU7n(&tr32PwmABC,&tr32InVoltage);
}
```

### 2.53.2 Function GMCLIB\_SvmU7n\_F16

#### Declaration

```
tU16 GMCLIB_SvmU7n_F16(SWLIBS_3Syst_F16 *pOut, const
                           SWLIBS_2Syst_F16 *const pIn);
```

#### Arguments

Table 293. GMCLIB\_SvmU7n\_F16 arguments

Type	Name	Direction	Description
<code>SWLIBS_3Syst_F16 *</code>	pOut	<code>input, output</code>	Pointer to the structure containing calculated duty-cycle ratios of the 3-Phase system.
<code>const SWLIBS_2Syst_F16 *const</code>	pIn	<code>input</code>	Pointer to the structure containing direct $U_\alpha$ and quadrature $U_\beta$ components of the stator voltage vector.

#### Return

The function returns a 16-bit value in format INT, representing the actual space sector which contains the stator reference vector  $U_s$ .

#### Code Example

```
#include "gmclib.h"
#define U_MAX 15

SWLIBS_2Syst_F16 tr16InVoltage;
SWLIBS_3Syst_F16 tr16PwmABC;
tU16 u16SvmSector;

void main(void)
{
    // Input voltage vector 15V @ angle 30deg
    // alpha component of input voltage vector = 12.99[V]
    // beta component of input voltage vector = 7.5[V]
    tr16InVoltage.f16Arg1 = FRAC16(12.99/U_MAX);
    tr16InVoltage.f16Arg2 = FRAC16(7.5/U_MAX);

    // output pwm dutycycles stored in structure referenced by tr16PwmABC
    // pwmA dutycycle = 0x7FFF = FRAC32(1.000...)
    // pwmb dutycycle = 0x4000 = FRAC32(0.500...)
    // pwmc dutycycle = 0x0000 = FRAC32(0.000...)
    // svmSector = 0x1 [sector]
    u16SvmSector = GMCLIB_SvmU7n_F16(&tr16PwmABC,&tr16InVoltage);

    // output pwm dutycycles stored in structure referenced by tr16PwmABC
    // pwmA dutycycle = 0x7FFF = FRAC32(1.000...)
    // pwmb dutycycle = 0x4000 = FRAC32(0.500...)
    // pwmc dutycycle = 0x0000 = FRAC32(0.000...)
    // svmSector = 0x1 [sector]
    u16SvmSector = GMCLIB_SvmU7n(&tr16PwmABC,&tr16InVoltage,F16);

    ##### Available only if 16-bit fractional implementation selected #####
    // as default
    #####
```

```

// output pwm dutycycles stored in structure referenced by tr16PwmABC
// pwmA dutycycle = 0x7FFF = FRAC32(1.000...)
// pwmB dutycycle = 0x4000 = FRAC32(0.500...)
// pwmC dutycycle = 0x0000 = FRAC32(0.000...)
// svmSector      = 0x1 [sector]
u16SvmSector = GMCLIB_SvmU7n(&tr16PwmABC, &tr16InVoltage);
}

```

### 2.53.3 Function GMCLIB\_SvmU7n\_FLT

#### Declaration

```
tU32 GMCLIB_SvmU7n_FLT(SWLIBS\_3Syst\_FLT *pOut, const
SWLIBS\_2Syst\_FLT *const pIn);
```

#### Arguments

**Table 294. GMCLIB\_SvmU7n\_FLT arguments**

Type	Name	Direction	Description
<a href="#">SWLIBS_3Syst_FLT</a> *	pOut	input, output	Pointer to the structure containing calculated duty-cycle ratios of the 3-Phase system.
const <a href="#">SWLIBS_2Syst_FLT</a> *const	pIn	input	Pointer to the structure containing direct $U_\alpha$ and quadrature $U_\beta$ components of the stator voltage vector.

#### Return

The function returns a 32-bit value in format INT, representing the actual space sector which contains the stator reference vector  $U_s$ .

#### Implementation details

The function presumes that the input voltages are normalized to fit the range  $<-1; 1>$ . In a typical motor control application, this function is preceded by the function [GMCLIB\\_ElimDcBusRip\\_FLT](#) which ensures that the voltages are correctly normalized.

**Note:** The function may raise floating-point exceptions (invalid operation, overflow, underflow, inexact, input denormal). The floating-point unit must be enabled in CPACR register to prevent NOCP UsageFault exception.

#### Code Example

```

#include "gmclib.h"
#define U_MAX 15

SWLIBS\_2Syst\_FLT tInVoltage;
SWLIBS\_3Syst\_FLT tPwmABC;
tU32 u32SvmSector;

void main(void)
{
    // Input voltage vector 15V @ angle 30deg
    // alpha component of input voltage vector = 12.99[V]
    // beta component of input voltage vector = 7.5[V]
    tInVoltage.fltArg1 = (tFloat) (12.99/U_MAX);
    tInVoltage.fltArg2 = (tFloat) (7.5/U_MAX);
}

```

```
// output pwm dutycycles stored in structure referenced by tr32PwmABC
// pwmA dutycycle = (tFloat)(1.0000000)
// pwmB dutycycle = (tFloat)(0.5000220)
// pwmC dutycycle = (tFloat)(0.0000220)
// u32SvmSector = 0x1 [sector]
u32SvmSector = GMCLIB_SvmU7n_FLT(&tPwmABC, &tInVoltage);

// output pwm dutycycles stored in structure referenced by tr32PwmABC
// pwmA dutycycle = (tFloat)(1.0000000)
// pwmB dutycycle = (tFloat)(0.5000220)
// pwmC dutycycle = (tFloat)(0.0000220)
// u32SvmSector = 0x1 [sector]
u32SvmSector = GMCLIB_SvmU7n(&tPwmABC, &tInVoltage, FLT);

// ######
// Available only if single precision floating point
// implementation selected as default
// #####
// #####
// output pwm dutycycles stored in structure referenced by tr32PwmABC
// pwmA dutycycle = (tFloat)(1.0000000)
// pwmB dutycycle = (tFloat)(0.5000220)
// pwmC dutycycle = (tFloat)(0.0000220)
// u32SvmSector = 0x1 [sector]
u32SvmSector = GMCLIB_SvmU7n(&tPwmABC, &tInVoltage);
}
```

## 2.54 Function GMCLIB\_VRot

This function returns a vector rotated by a given angle.

### Description

This function performs an addition of a given angle to an input vector.

If the angle is defined as  $\alpha$  and the input vector as

$$\vec{u} = (u_1, u_2)$$

Equation GMCLIB\_VRot\_Eq1

then the output of the function  $u'$  is calculated by the following simple equation:

$$\vec{u}' = (u_1 \cdot \cos\alpha - u_2 \cdot \sin\alpha, u_1 \cdot \sin\alpha + u_2 \cdot \cos\alpha)$$

Equation GMCLIB\_VRot\_Eq2

### Re-entrancy

The function is re-entrant.

### 2.54.1 Function GMCLIB\_VRot\_F32

#### Declaration

```
void GMCLIB_VRot_F32(SWLIBS\_2Syst\_F32 *const f32OutVec, const
SWLIBS\_2Syst\_F32 *const f32InVec, tFrac32 f32Angle);
```

#### Arguments

**Table 295. GMCLIB\_VRot\_F32 arguments**

Type	Name	Direction	Description
<a href="#">SWLIBS_2Syst_F32</a> *const	f32OutVec	<b>output</b>	Rotated vector.
const <a href="#">SWLIBS_2Syst_F32</a> *const	f32InVec	<b>input</b>	Input vector to be rotated.
<a href="#">tFrac32</a>	f32Angle	<b>input</b>	Angle the input vector shall be rotated by.

#### Implementation details

The first input as well as output value are considered as vectors defined by two 32-bit fractional coordinates stored in [SWLIBS\\_2Syst\\_F32](#) structure. Input vector is rotated by angle defined by 32-bit fractional number in radians from interval  $[-\pi, \pi]$  normalized between  $[-1, 1]$ .

#### Code Example:

```
#include "gmclib.h"

SWLIBS\_2Syst\_F32 f32InVec;
tFrac32 f32Angle;
SWLIBS\_2Syst\_F32 f32Out;

void main(void)
{
    // input vector = (0.5,0.5)
    f32InVec.f32Arg1 = FRAC32(0.5);
    f32InVec.f32Arg2 = FRAC32(0.5);
    // angle (-pi) = -1
    f32Angle = FRAC32(-1);

    // Both f32Out.f32Arg1 and f32Out.f32Arg2 should be FRAC32(-0.5) = 0xC0000000
    GMCLIB_VRot_F32(&f32Out, &f32InVec, f32Angle);

    // Both f32Out.f32Arg1 and f32Out.f32Arg2 should be FRAC32(-0.5) = 0xC0000000
    GMCLIB_VRot(&f32Out, &f32InVec, f32Angle, F32);

    // ##### Available only if 32-bit fractional implementation selected
    // as default
    // #####
    // Both f32Out.f32Arg1 and f32Out.f32Arg2 should be FRAC32(-0.5) = 0xC0000000
    GMCLIB_VRot(&f32Out, &f32InVec, f32Angle);
}
```

## 2.54.2 Function GMCLIB\_VRot\_F16

### Declaration

```
void GMCLIB_VRot_F16(SWLIBS\_2Syst\_F16 *const f16OutVec, const
SWLIBS\_2Syst\_F16 *const f16InVec, tFrac16 f16Angle);
```

### Arguments

**Table 296. GMCLIB\_VRot\_F16 arguments**

Type	Name	Direction	Description
<a href="#">SWLIBS_2Syst_F16</a> *const	f16OutVec	<b>output</b>	Rotated vector.
const <a href="#">SWLIBS_2Syst_F16</a> *const	f16InVec	<b>input</b>	Input vector to be rotated.
<a href="#">tFrac16</a>	f16Angle	<b>input</b>	Angle the input vector shall be rotated by.

### Implementation details

The first input as well as output value are considered as vectors defined by two 16-bit fractional coordinates stored in [SWLIBS\\_2Syst\\_F16](#) structure. Input vector is rotated by angle defined by 16-bit fractional number in radians from interval [- $\pi$ ,  $\pi$ ) normalized between [-1, 1].

### Code Example:

```
#include "gmclib.h"

SWLIBS\_2Syst\_F16 f16InVec;
tFrac32 f16Angle;
SWLIBS\_2Syst\_F16 f16Out;

void main(void)
{
    // input vector = (0.5,0.5)
    f16InVec.f16Arg1 = FRAC16(0.5);
    f16InVec.f16Arg2 = FRAC16(0.5);
    // angle (-pi) = -1
    f16Angle = FRAC16(-1);

    // Both f16Out.f16Arg1 and f16Out.f16Arg2 should be FRAC16(-0.5) = 0xC000
    GMCLIB_VRot_F16(&f16Out, &f16InVec, f16Angle);

    // Both f16Out.f16Arg1 and f16Out.f16Arg2 should be FRAC16(-0.5) = 0xC000
    GMCLIB_VRot(&f16Out, &f16InVec, f16Angle, F16);

    // ##### Available only if 16-bit fractional implementation selected
    // as default
    // #####
    // Both f16Out.f16Arg1 and f16Out.f16Arg2 should be FRAC16(-0.5) = 0xC000
    GMCLIB_VRot(&f16Out, &f16InVec, f16Angle);
}
```

### 2.54.3 Function GMCLIB\_VRot\_FLT

#### Declaration

```
void GMCLIB_VRot_FLT(SWLIBS\_2Syst\_FLT *const fltOutVec, const
SWLIBS\_2Syst\_FLT *const fltInVec, tFloat fltAngle);
```

#### Arguments

**Table 297. GMCLIB\_VRot\_FLT arguments**

Type	Name	Direction	Description
<a href="#">SWLIBS_2Syst_FLT</a> *const	fltOutVec	<b>output</b>	Rotated vector.
const <a href="#">SWLIBS_2Syst_FLT</a> *const	fltInVec	<b>input</b>	Input vector to be rotated.
<a href="#">tFloat</a>	fltAngle	<b>input</b>	Angle the input vector shall be rotated by.

#### Implementation details

The first input as well as output value are considered as vectors defined by two single precision floating-point coordinates stored in [SWLIBS\\_2Syst\\_FLT](#) structure. Input vector is rotated by angle defined by single precision floating-point number in radians from interval [-  $\pi$ ,  $\pi$ ].

**Note:** The function may raise floating-point exceptions (invalid operation, overflow, underflow, inexact, input denormal). The floating-point unit must be enabled in CPACR register to prevent NOCP UsageFault exception.

#### Code Example

```
#include "gmclib.h"

SWLIBS\_2Syst\_FLT fltInVec;
tFrac32 fltAngle;
SWLIBS\_2Syst\_FLT fltOut;

void main(void)
{
    // input vector = (0.5,0.5)
    fltInVec.fltArg1 = 0.5;
    fltInVec.fltArg2 = 0.5;
    // angle = pi
    fltAngle = FLOAT\_PI;

    // Both fltOut.fltArg1 and fltOut.fltArg2 should be -0.5
    GMCLIB_VRot_FLT(&fltOut, &fltInVec, fltAngle);

    // Both fltOut.fltArg1 and fltOut.fltArg2 should be -0.5
    GMCLIB_VRot(&fltOut, &fltInVec, fltAngle, FLT);

    // ##### Available only if 16-bit fractional implementation selected
    // as default
    // #####
    // Both fltOut.fltArg1 and fltOut.fltArg2 should be -0.5
}
```

```
    GMCLIB_VRot(&fltOut, &fltInVec, fltAngle);
}
```

## 2.55 Function GMCLIB\_VUnit

This function returns an unit vector created by scaling input vector.

### Description

This function performs a vector normalization so the output vector has the same angle as input vector and the magnitude equal to 1.

If the input vector is defined as

$$\vec{u} = (u_1, u_2)$$

Equation GMCLIB\_VUnit\_Eq1

then the output unit vector  $u'$  is calculated by the following simple equation:

$$\vec{u}' = \left( \frac{u_1}{|u|}, \frac{u_2}{|u|} \right)$$

Equation GMCLIB\_VUnit\_Eq2

where  $|u|$  is magnitude of thr input vector.

### Re-entrancy

The function is re-entrant.

#### 2.55.1 Function GMCLIB\_VUnit\_F32

##### Declaration

```
void GMCLIB_VUnit_F32(SWLIBS\_2Syst\_F32 *const f32OutVec, const
SWLIBS\_2Syst\_F32 *const f32InVec);
```

##### Arguments

Table 298. GMCLIB\_VUnit\_F32 arguments

Type	Name	Direction	Description
<a href="#">SWLIBS_2Syst_F32</a> *const	f32OutVec	output	Output unit vector.
const <a href="#">SWLIBS_2Syst_F32</a> *const	f32InVec	input	Input vector to be scaled to unit vector.

##### Implementation details

The input as well as output are considered as vectors defined by two 32-bit fractional coordinates stored in [SWLIBS\\_2Syst\\_F32](#) structure.

##### Code Example:

```
#include "gmclib.h"
```

```

SWLIBS_2Syst_F32 f32In;
SWLIBS_2Syst_F32 f32Out;

void main(void)
{
    // input vector = (0.5,0.5)
    f32InVec.f32Arg1 = FRAC32(0.5);
    f32InVec.f32Arg2 = FRAC32(0.5);

    // Both f32Out.f32Arg1 and f32Out.f32Arg2 should be FRAC32(0.707106781186548)
    GMCLIB_VUnit_F32(&f32Out, &f32In);

    // Both f32Out.f32Arg1 and f32Out.f32Arg2 should be FRAC32(0.707106781186548)
    GMCLIB_VUnit(&f32Out, &f32In, F32);

    // ######
    // Available only if 32-bit fractional implementation selected
    // as default
    // #####
    // Both f32Out.f32Arg1 and f32Out.f32Arg2 should be FRAC32(0.707106781186548)
    GMCLIB_VUnit(&f32Out, &f32In);
}

```

## 2.55.2 Function GMCLIB\_VUnit\_F16

### Declaration

```
void GMCLIB_VUnit_F16(SWLIBS_2Syst_F16 *const f16OutVec, const
SWLIBS_2Syst_F16 *const f16InVec);
```

### Arguments

**Table 299. GMCLIB\_VUnit\_F16 arguments**

Type	Name	Direction	Description
<code>SWLIBS_2Syst_F16</code> *const	f16OutVec	output	Output unit vector.
const <code>SWLIBS_2Syst_F16</code> *const	f16InVec	input	Input vector to be scaled to unit vector.

### Implementation details

The input as well as output are considered as vectors defined by two 16-bit fractional coordinates stored in `SWLIBS_2Syst_F16` structure.

### Code Example:

```

#include "gmclib.h"

SWLIBS_2Syst_F16 f16In;
SWLIBS_2Syst_F16 f16Out;

void main(void)
{
    // input vector = (0.5,0.5)

```

```

f16InVec.f16Arg1 = FRAC16(0.5);
f16InVec.f16Arg2 = FRAC16(0.5);

// Both f16Out.f16Arg1 and f16Out.f16Arg2 should be FRAC16(0.707106781186548)
GMCLIB_VUnit_F16(&f16Out, &f16In);

// Both f16Out.f16Arg1 and f16Out.f16Arg2 should be FRAC16(0.707106781186548)
GMCLIB_VUnit(&f16Out, &f16In, F16);

// ######
// Available only if 16-bit fractional implementation selected
// as default
// #####
// Both f16Out.f16Arg1 and f16Out.f16Arg2 should be FRAC16(0.707106781186548)
GMCLIB_VUnit(&f16Out, &f16In);
}

```

### 2.55.3 Function GMCLIB\_VUnit\_FLT

#### Declaration

```
void GMCLIB_VUnit_FLT(SWLIBS\_2Syst\_FLT *const fltOutVec, const
SWLIBS\_2Syst\_FLT *const fltInVec);
```

#### Arguments

**Table 300. GMCLIB\_VUnit\_FLT arguments**

Type	Name	Direction	Description
<a href="#">SWLIBS_2Syst_FLT</a> *const	fltOutVec	output	Output unit vector.
const <a href="#">SWLIBS_2Syst_FLT</a> *const	fltInVec	input	Input vector to be scaled to unit vector.

#### Implementation details

The input as well as output are considered as vectors defined by two single precision floating-point coordinates stored in [SWLIBS\\_2Syst\\_FLT](#) structure.

**Note:** The function may raise floating-point exceptions (invalid operation, overflow, underflow, inexact, input denormal). The floating-point unit must be enabled in CPACR register to prevent NOCP UsageFault exception.

#### Code Example

```

#include "gmclib.h"

SWLIBS\_2Syst\_FLT fltIn;
SWLIBS\_2Syst\_FLT fltOut;

void main(void)
{
    // input vector = (0.5,0.5)
    fltInVec.fltArg1 = 0.5;
    fltInVec.fltArg2 = 0.5;
}

```

```

// Both fltOut.fltArg1 and fltOut.fltArg2 should be 0.707106781186548
GMCLIB_VUnit_FLT(&fltOut, &fltIn);

// Both fltOut.fltArg1 and fltOut.fltArg2 should be 0.707106781186548
GMCLIB_VUnit(&fltOut, &fltIn, FLT);

// ######
// Available only if single precision floating-point implementation selected
// as default
// #####
// Both fltOut.fltArg1 and fltOut.fltArg2 should be 0.707106781186548
GMCLIB_VUnit(&fltOut, &fltIn);
}

```

## 2.56 Function MLIB\_Abs

This function returns absolute value of input parameter.

### Description

This inline function returns the absolute value of input parameter.

### Re-entrancy

The function is re-entrant.

#### 2.56.1 Function MLIB\_Abs\_F32

##### Declaration

```
INLINE tFrac32 MLIB_Abs_F32(register tFrac32 f32In);
```

##### Arguments

**Table 301. MLIB\_Abs\_F32 arguments**

Type	Name	Direction	Description
register <a href="#">tFrac32</a>	f32In	input	Input value.

##### Return

Absolute value of input parameter.

##### Implementation details

The input value as well as output value is considered as 32-bit fractional data type. The output saturation is not implemented in this function, thus in case the absolute value of input parameter is outside the [-1, 1) interval, the output value will overflow.

The output of the function is defined by the following simple equation:

$$f32Out = \begin{cases} f32In & \text{if } f32In \geq 0 \\ -f32In & \text{if } f32In < 0 \end{cases}$$

Equation MLIB\_Abs\_F32\_Eq1

**Note:** Due to effectivity reason this function is implemented as inline, and thus is not ANSI-C compliant.

### Code Example

```
#include "mlib.h"

tFrac32 f32In;
tFrac32 f32Out;

void main(void)
{
    // input value = -0.25
    f32In = FRAC32(-0.25);

    // output should be FRAC32(0.25)
    f32Out = MLIB_Abs_F32(f32In);

    // output should be FRAC32(0.25)
    f32Out = MLIB_Abs(f32In, F32);

    // ##### Available only if 32-bit fractional implementation selected
    // as default
    // #####
    // output should be FRAC32(0.25)
    f32Out = MLIB_Abs(f32In);
}
```

## 2.56.2 Function MLIB\_Abs\_F16

### Declaration

```
INLINE tFrac16 MLIB_Abs_F16(register tFrac16 f16In);
```

### Arguments

Table 302. MLIB\_Abs\_F16 arguments

Type	Name	Direction	Description
register tFrac16	f16In	input	Input value.

### Return

Absolute value of input parameter.

### Implementation details

The input value as well as output value is considered as 16-bit fractional data type. The output saturation is not implemented in this function, thus in case the absolute value of input parameter is outside the [-1, 1] interval, the output value will overflow.

The output of the function is defined by the following simple equation:

$$f16Out = \begin{cases} f16In & \text{if } f16In \geq 0 \\ -f16In & \text{if } f16In < 0 \end{cases}$$

Equation MLIB\_Abs\_F16\_Eq1

**Note:** Due to effectivity reason this function is implemented as inline, and thus is not ANSI-C compliant.

### Code Example

```
#include "mlib.h"

tFrac16 f16In;
tFrac16 f16Out;

void main(void)
{
    // input value = -0.25
    f16In = FRAC16(-0.25);

    // output should be FRAC16(0.25)
    f16Out = MLIB_Abs_F16(f16In);

    // output should be FRAC16(0.25)
    f16Out = MLIB_Abs(f16In, F16);

    // ##### Available only if 16-bit fractional implementation selected
    // as default
    // #####
    // output should be FRAC16(0.25)
    f16Out = MLIB_Abs(f16In);
}
```

### 2.56.3 Function MLIB\_Abs\_FLT

#### Declaration

```
INLINE tFloat MLIB_Abs_FLT(register tFloat fltIn);
```

#### Arguments

Table 303. MLIB\_Abs\_FLT arguments

Type	Name	Direction	Description
register tFloat	fltIn	input	Input value.

#### Return

Absolute value of input parameter.

#### Implementation details

The input value as well as output value is considered as single precision floating point data type.

The output of the function is defined by the following simple equation:

$$f\text{ltOut} = \begin{cases} f\text{ltIn} & \text{if } f\text{ltIn} \geq 0 \\ -f\text{ltIn} & \text{if } f\text{ltIn} < 0 \end{cases}$$

Equation MLIB\_Abs\_FLT\_Eq1

**Note:** Due to effectivity reason this function is implemented as inline, and thus is not ANSI-C compliant.

### Code Example

```
#include "mlib.h"

tFloat fltIn;
tFloat fltOut;

void main(void)
{
    // input value = -0.25
    fltIn = (tFloat)-0.25;

    // output should be 0.25
    fltOut = MLIB_Abs_FLT(fltIn);

    // output should be 0.25
    fltOut = MLIB_Abs(fltIn, FLT);

    // ##### Available only if single precision floating point
    // implementation selected as default
    // #####
    // output should be 0.25
    fltOut = MLIB_Abs(fltIn);
}
```

## 2.57 Function MLIB\_AbsSat

This function returns absolute value of input parameter and saturate if necessary.

### Description

This inline function returns the absolute value of input parameter and saturates if necessary.

### Re-entrancy

The function is re-entrant.

#### 2.57.1 Function MLIB\_AbsSat\_F32

##### Declaration

```
INLINE tFrac32 MLIB_AbsSat_F32(register tFrac32 f32In);
```

**Arguments****Table 304. MLIB\_AbsSat\_F32 arguments**

Type	Name	Direction	Description
register tFrac32	f32In	input	Input value.

**Return**

Absolute value of input parameter, saturated if necessary.

**Implementation details**

The input values as well as output value is considered as 32-bit fractional data type.

The output of the function is defined by the following simple equation:

$$f32Out = \begin{cases} FRAC32\_MIN & \text{if } |f32In| < FRAC32\_MIN \\ |f32In| & \text{if } FRAC32\_MIN \leq |f32In| \leq FRAC32\_MAX \\ FRAC32\_MAX & \text{if } |f32In| > FRAC32\_MAX \end{cases}$$

Equation MLIB\_AbsSat\_F32\_Eq1

**Note:** Due to effectivity reason this function is implemented as inline assembly, and thus is not ANSI-C compliant.

**Code Example**

```
#include "mlib.h"

tFrac32 f32In;
tFrac32 f32Out;

void main(void)
{
    // input value = -0.25
    f32In = FRAC32(-0.25);

    // output should be FRAC32(0.25)
    f32Out = MLIB_AbsSat_F32(f32In);

    // output should be FRAC32(0.25)
    f32Out = MLIB_AbsSat(f32In, F32);

    // ######
    // Available only if 32-bit fractional implementation selected
    // as default
    // #####
    // output should be FRAC32(0.25)
    f32Out = MLIB_AbsSat(f32In);
}
```

## 2.57.2 Function MLIB\_AbsSat\_F16

### Declaration

```
INLINE tFrac16 MLIB_AbsSat_F16(register tFrac16 f16In);
```

### Arguments

**Table 305. MLIB\_AbsSat\_F16 arguments**

Type	Name	Direction	Description
register <a href="#">tFrac16</a>	f16In	input	Input value.

### Return

Absolute value of input parameter, saturated if necessary.

### Implementation details

The input values as well as output value is considered as 16-bit fractional data type.

The output of the function is defined by the following simple equation:

$$f16Out = \begin{cases} FRAC16\_MIN & \text{if } |f16In| < FRAC16\_MIN \\ |f16In| & \text{if } FRAC16\_MIN \leq |f16In| \leq FRAC16\_MAX \\ FRAC16\_MAX & \text{if } |f16In| > FRAC16\_MAX \end{cases}$$

Equation MLIB\_AbsSat\_F16\_Eq1

**Note:** Due to effectivity reason this function is implemented as inline assembly, and thus is not ANSI-C compliant.

### Code Example

```
#include "mlib.h"

tFrac16 f16In;
tFrac16 f16Out;

void main(void)
{
    // input value = -0.25
    f16In = FRAC16(-0.25);

    // output should be FRAC16(0.25)
    f16Out = MLIB_AbsSat_F16(f16In);

    // output should be FRAC16(0.25)
    f16Out = MLIB_AbsSat(f16In, F16);

    // #####
    // Available only if 16-bit fractional implementation selected
    // as default
    // #####
    // output should be FRAC16(0.25)
    f16Out = MLIB_AbsSat(f16In);
```

```
}
```

## 2.58 Function MLIB\_Add

This function returns sum of two input parameters.

### Description

This inline function returns the sum of two input values.

### Re-entrancy

The function is re-entrant.

#### 2.58.1 Function MLIB\_Add\_F32

##### Declaration

```
INLINE tFrac32 MLIB_Add_F32(register tFrac32 f32In1, register tFrac32 f32In2);
```

##### Arguments

**Table 306. MLIB\_Add\_F32 arguments**

Type	Name	Direction	Description
register <a href="#">tFrac32</a>	f32In1	input	First value to be add.
register <a href="#">tFrac32</a>	f32In2	input	Second value to be add.

##### Return

Sum of two input values.

##### Implementation details

The input values as well as output value is considered as 32-bit fractional data type. The output saturation is not implemented in this function, thus in case the sum of input values is outside the [-1, 1) interval, the output value will overflow.

The output of the function is defined by the following simple equation:

$$f32Out = f32In1 + f32In2$$

Equation MLIB\_Add\_F32\_Eq1

**Note:** Due to effectivity reason this function is implemented as inline and thus is not ANSI-C compliant.

##### Code Example:

```
#include "mlib.h"

tFrac32 f32In1, f32In2;
tFrac32 f32Out;
```

```

void main(void)
{
    // input value 1 = 0.25
    f32In1 = FRAC32(0.25);
    // input value 2 = 0.25
    f32In2 = FRAC32(0.25);

    // output should be FRAC32(0.5) = 0x40000000
    f32Out = MLIB_Add_F32(f32In1, f32In2);

    // output should be FRAC32(0.5) = 0x40000000
    f32Out = MLIB_Add(f32In1, f32In2, F32);

    // ##### Available only if 32-bit fractional implementation selected
    // as default
    // #####
    // output should be FRAC32(0.5) = 0x40000000
    f32Out = MLIB_Add(f32In1, f32In2);
}

```

## 2.58.2 Function MLIB\_Add\_F16

### Declaration

```
INLINE tFrac16 MLIB_Add_F16(register tFrac16 f16In1, register
tFrac16 f16In2);
```

### Arguments

**Table 307. MLIB\_Add\_F16 arguments**

Type	Name	Direction	Description
register <a href="#">tFrac16</a>	f16In1	input	First value to be add.
register <a href="#">tFrac16</a>	f16In2	input	Second value to be add.

### Return

Sum of two input values.

### Implementation details

The input values as well as output value is considered as 16-bit fractional values. The output saturation is not implemented in this function, thus in case the sum of input values is outside the [-1, 1] interval, the output value will overflow.

The output of the function is defined by the following simple equation:

$$f16Out = f16In1 + f16In2$$

Equation MLIB\_Add\_F16\_Eq1

**Note:** Due to effectivity reason this function is implemented as inline and thus is not ANSI-C compliant.

**Code Example**

```
#include "mlib.h"

tFrac16 f16In1, f16In2;
tFrac16 f16Out;

void main(void)
{
    // input value 1 = 0.25
    f16In1 = FRAC16(0.25);
    // input value 2 = 0.25
    f16In2 = FRAC16(0.25);

    // output should be FRAC16(0.5) = 0x4000
    f16Out = MLIB_Add_F16(f16In1, f16In2);

    // output should be FRAC16(0.5) = 0x4000
    f16Out = MLIB_Add(f16In1, f16In2, F16);

    // ##### Available only if 16-bit fractional implementation selected
    // as default
    // #####
    // output should be FRAC16(0.5) = 0x4000
    f16Out = MLIB_Add(f16In1, f16In2);
}
```

**2.58.3 Function MLIB\_Add\_FLT****Declaration**

```
INLINE tFloat MLIB_Add_FLT(register tFloat fltIn1, register
tFloat fltIn2);
```

**Arguments****Table 308. MLIB\_Add\_FLT arguments**

Type	Name	Direction	Description
register tFloat	fltIn1	input	First value to be add.
register tFloat	fltIn2	input	Second value to be add.

**Return**

Sum of two input values.

**Note:** The function may raise floating-point exceptions (invalid operation, overflow, underflow, inexact, input denormal). The floating-point unit must be enabled in CPACR register to prevent NOCP UsageFault exception.

*Due to effectivity reason this function is implemented as inline and thus is not ANSI-C compliant.*

**Code Example**

```
#include "mlib.h"

tFloat fltIn1, fltIn2;
tFloat fltOut;

void main(void)
{
    // input value 1 = 0.25
    fltIn1 = (tFloat)0.25;
    // input value 2 = 0.25
    fltIn2 = (tFloat)0.25;

    // output should be 0.5
    fltOut = MLIB_Add_FLT(fltIn1, fltIn2);

    // output should be 0.5
    fltOut = MLIB_Add(fltIn1, fltIn2, FLT);

    // ##### Available only if single precision floating point
    // implementation selected as default
    // #####
    // output should be 0.5
    fltOut = MLIB_Add(fltIn1, fltIn2);
}
```

**2.59 Function MLIB\_AddSat**

This function returns sum of two input parameters and saturate if necessary.

**Description**

This inline function returns the sum of two input values and saturates the result if necessary.

**Re-entrancy**

The function is re-entrant.

**2.59.1 Function MLIB\_AddSat\_F32****Declaration**

```
INLINE tFrac32 MLIB_AddSat_F32(register tFrac32 f32In1, register
tFrac32 f32In2);
```

**Arguments**

**Table 309. MLIB\_AddSat\_F32 arguments**

Type	Name	Direction	Description
register tFrac32	f32In1	input	First value to be add.

Type	Name	Direction	Description
register tFrac32	f32In2	input	Second value to be add.

**Return**

Sum of two input values, saturated if necessary.

**Implementation details**

The input values as well as output value is considered as 32-bit fractional data type.

The output of the function is defined by the following simple equation:

$$f32Out = \begin{cases} FRAC32\_MIN & \text{if } (f32In1 + f32In2) < FRAC32\_MIN \\ f32In1 + f32In2 & \text{if } FRAC32\_MIN \leq (f32In1 + f32In2) \leq FRAC32\_MAX \\ FRAC32\_MAX & \text{if } (f32In1 + f32In2) > FRAC32\_MAX \end{cases}$$

Equation MLIB\_AddSat\_F32\_Eq1

**Note:** Due to effectivity reason this function is implemented as inline assembly, and thus is not ANSI-C compliant.

**Code Example**

```
#include "mlib.h"

tFrac32 f32In1, f32In2;
tFrac32 f32Out;

void main(void)
{
    // input value 1 = 0.25
    f32In1 = FRAC32(0.25);
    // input value 2 = 0.25
    f32In2 = FRAC32(0.25);

    // output should be FRAC32(0.5) = 0x40000000
    f32Out = MLIB_AddSat_F32(f32In1, f32In2);

    // output should be FRAC32(0.5) = 0x40000000
    f32Out = MLIB_AddSat(f32In1, f32In2, F32);

    // ##### Available only if 32-bit fractional implementation selected
    // as default
    // #####
    // output should be FRAC32(0.5) = 0x40000000
    f32Out = MLIB_AddSat(f32In1, f32In2);
}
```

## 2.59.2 Function MLIB\_AddSat\_F16

### Declaration

```
INLINE tFrac16 MLIB_AddSat_F16(register tFrac16 f16In1, register
tFrac16 f16In2);
```

### Arguments

**Table 310. MLIB\_AddSat\_F16 arguments**

Type	Name	Direction	Description
register <b>tFrac16</b>	f16In1	<b>input</b>	First value to be add.
register <b>tFrac16</b>	f16In2	<b>input</b>	Second value to be add.

### Return

Sum of two input values, saturated if necessary.

### Implementation details

The input values as well as output value is considered as 16-bit fractional data type.

The output of the function is defined by the following simple equation:

$$f16Out = \begin{cases} FRAC16\_MIN & \text{if } (f16In1 + f16In2) < FRAC16\_MIN \\ f16In1 + f16In2 & \text{if } FRAC16\_MIN \leq (f16In1 + f16In2) \leq FRAC16\_MAX \\ FRAC16\_MAX & \text{if } (f16In1 + f16In2) > FRAC16\_MAX \end{cases}$$

Equation MLIB\_AddSat\_F16\_Eq1

**Note:** Due to effectivity reason this function is implemented as inline assembly, and thus is not ANSI-C compliant.

### Code Example

```
#include "mlib.h"

tFrac16 f16In1, f16In2;
tFrac16 f16Out;

void main(void)
{
    // input value 1 = 0.25
    f16In1 = FRAC16(0.25);
    // input value 2 = 0.25
    f16In2 = FRAC16(0.25);

    // output should be FRAC16(0.5) = 0x4000
    f16Out = MLIB_AddSat_F16(f16In1, f16In2);

    // output should be FRAC16(0.5) = 0x4000
    f16Out = MLIB_AddSat(f16In1, f16In2, F16);

    // ##### Available only if 16-bit fractional implementation selected
    // as default
}
```

```
// ######
// output should be FRAC16(0.5) = 0x4000
f16Out = MLIB_AddSat(f16In1, f16In2);
}
```

## 2.60 Function MLIB\_Convert

This function converts the input value to different representation with scale.

### Description

This inline function converts the input value to a different data type. The second input argument represents the scale factor.

### Re-entrancy

The function is re-entrant.

#### 2.60.1 Function MLIB\_Convert\_F32F16

##### Declaration

```
INLINE tFrac32 MLIB_Convert_F32F16(register tFrac16 f16In1,  
register tFrac16 f16In2);
```

##### Arguments

**Table 311. MLIB\_Convert\_F32F16 arguments**

Type	Name	Direction	Description
register <a href="#">tFrac16</a>	f16In1	input	Input value in 16-bit fractional format to be converted.
register <a href="#">tFrac16</a>	f16In2	input	Scale factor in 16-bit fractional format.

##### Return

Converted input value in 32-bit fractional format.

##### Implementation details

The input value is considered as 16-bit fractional data type and output value is considered as 32-bit fractional data type. The second argument is considered as 16-bit fractional data type. The sign of the second value represents the scale mechanism. In case the second value is positive the first input value is multiplied with the second one and converted to the output format. In case the second value is negative, the first input value is divided by absolute value of second input value and converted to the output format. The output saturation is not implemented in this function, thus in case the input value is outside the [-1, 1] interval, the output value will overflow.

The output of the function is defined by the following simple equation:

$$f32Out = \begin{cases} (tFrac32)\frac{f16In1}{|f16In2|} & \text{if } f16In2 < 0 \\ (tFrac32)(f16In1 \cdot f16In2) & \text{if } f16In2 \geq 0 \end{cases}$$

Equation MLIB\_Convert\_F32F16\_Eq1

**Note:** Due to effectivity reason this function is implemented as inline and thus is not ANSI-C compliant.

### Code Example

```
#include "mlib.h"

tFrac16 f16In1, f16In2;
tFrac32 f32Out;

void main(void)
{
    // input value = 0.25 = 0x2000
    f16In1 = FRAC16(0.25);

    // scale value = 0.5 = 0x4000
    f16In2 = FRAC16(0.5);

    // output should be FRAC32(0.125) = 0x10000000
    f32Out = MLIB_Convert_F32F16(f16In1, f16In2);

    // output should be FRAC32(0.125) = 0x10000000
    f32Out = MLIB_Convert(f16In1, f16In2, F32F16);

    // scale value = -0.5 = 0xC000
    f16In2 = FRAC16(-0.5);

    // output should be FRAC32(0.5) = 0x40000000
    f32Out = MLIB_Convert_F32F16(f16In1, f16In2);

    // output should be FRAC32(0.5) = 0x40000000
    f32Out = MLIB_Convert(f16In1, f16In2, F32F16);
}
```

## 2.60.2 Function `MLIB_Convert_F32FLT`

### Declaration

```
INLINE tFrac32 MLIB_Convert_F32FLT(register tFloat fltIn1,
register tFloat fltIn2);
```

### Arguments

**Table 312. `MLIB_Convert_F32FLT` arguments**

Type	Name	Direction	Description
register tFloat	fltIn1	input	Input value in single precision floating point format to be converted.
register tFloat	fltIn2	input	Scale factor in single precision floating point format.

### Return

Converted input value in 32-bit fractional format.

### Implementation details

The input value is considered as single precision floating point data type and output value is considered as 32-bit fractional data type. The second argument is considered as single precision floating point data type. The output saturation is implemented in this function, thus in case the input value is outside the [-1, 1) interval, the output value is limited to the boundary value.

The output of the function is defined by the following simple equation:

$$f32Out = \begin{cases} FRAC32\_MIN & \text{if } (fltIn1 \cdot fltIn2) < FRAC32\_MIN \\ fltIn1 \cdot fltIn2 \cdot ((tFloat)INT32\_MAX + 1) & \text{if } FRAC32\_MIN \leq (fltIn1 \cdot fltIn2) \leq FRAC32\_MAX \\ FRAC32\_MAX & \text{if } (fltIn1 \cdot fltIn2) > FRAC32\_MAX \end{cases}$$

Equation MLIB\_Convert\_F32FLT\_Eq1

**Note:** The function may raise floating-point exceptions (invalid operation, overflow, underflow, inexact, input denormal). The floating-point unit must be enabled in CPACR register to prevent NOCP UsageFault exception.

Due to effectivity reason this function is implemented as inline and thus is not ANSI-C compliant.

### Code Example

```
#include "mlib.h"

tFloat fltIn1, fltIn2;
tFrac32 f32Out;

void main(void)
{
    // input value = 0.25
    fltIn1 = (tFloat)0.25;

    // scale value = 0.5
    fltIn2 = (tFloat)0.5;

    // output should be FRAC32(0.125) = 0x10000000
    f32Out = MLIB_Convert_F32FLT(fltIn1, fltIn2);

    // output should be FRAC32(0.125) = 0x10000000
    f32Out = MLIB_Convert(fltIn1, fltIn2, F32FLT);

    // scale value = 2
    fltIn2 = (tFloat)2;

    // output should be FRAC32(0.5) = 0x40000000
    f32Out = MLIB_Convert_F32FLT(fltIn1, fltIn2);

    // output should be FRAC32(0.5) = 0x40000000
    f32Out = MLIB_Convert(fltIn1, fltIn2, F32FLT);
}
```

### 2.60.3 Function MLIB\_Convert\_F16F32

#### Declaration

```
INLINE tFrac16 MLIB_Convert_F16F32(register tFrac32 f32In1,  
register tFrac32 f32In2);
```

#### Arguments

**Table 313. MLIB\_Convert\_F16F32 arguments**

Type	Name	Direction	Description
register <a href="#">tFrac32</a>	f32In1	<b>input</b>	Input value in 32-bit fractional format to be converted.
register <a href="#">tFrac32</a>	f32In2	<b>input</b>	Scale factor in 32-bit fractional format.

#### Return

Converted input value in 16-bit fractional format.

#### Implementation details

The input value is considered as 32-bit fractional data type and output value is considered as 16-bit fractional data type. The second value is considered as 32-bit fractional data type. The sign of the second value represents the scale mechanism. In case the second value is positive the first input value is multiplied with the second one and converted to the output format. In case the second value is negative, the first input value is divided by absolute value of second input value and converted to the output format. The output saturation is not implemented in this function, thus in case the input value is outside the [-1, 1) interval, the output value will overflow.

The output of the function is defined by the following simple equation:

$$f16Out = \begin{cases} (tFrac16)\frac{f32In1}{|f32In2|} & \text{if } f32In2 < 0 \\ (tFrac16)(f32In1 \cdot f32In2) & \text{if } f32In2 \geq 0 \end{cases}$$

Equation MLIB\_Convert\_F16F32\_Eq1

**Note:** Due to effectivity reason this function is implemented as inline and thus is not ANSI-C compliant.

#### Code Example

```
#include "mlib.h"

tFrac32 f32In1, f32In2;
tFrac16 f16Out;

void main(void)
{
    // input value = 0.25 = 0x20000000
    f32In1 = FRAC32(0.25);

    // scale value = 0.5 = 0x40000000
    f32In2 = FRAC32(0.5);

    // output should be FRAC16(0.125) = 0x1000
}
```

```

f16Out = MLIB_Convert_F16F32(f32In1, f32In2);

// output should be FRAC16(0.125) = 0x1000
f16Out = MLIB_Convert(f32In1, f32In2, F16F32);

// scale value = -0.5 = 0xC0000000
f32In2 = FRAC32(-0.5);

// output should be FRAC16(0.5) = 0x4000
f16Out = MLIB_Convert_F16F32(f32In1, f32In2);

// output should be FRAC16(0.5) = 0x4000
f16Out = MLIB_Convert(f32In1, f32In2, F16F32);
}

```

## 2.60.4 Function [MLIB\\_Convert\\_F16FLT](#)

### Declaration

```
INLINE tFrac16 MLIB_Convert_F16FLT(register tFloat fltIn1,  
register tFloat fltIn2);
```

### Arguments

**Table 314. [MLIB\\_Convert\\_F16FLT](#) arguments**

Type	Name	Direction	Description
register <a href="#">tFloat</a>	fltIn1	input	Input value in single precision floating point format to be converted.
register <a href="#">tFloat</a>	fltIn2	input	Scale factor in single precision floating point format.

### Return

Converted input value in 16-bit fractional format.

### Implementation details

The input value is considered as single precision floating point data type and output value is considered as 16-bit fractional data type. The second value is considered as single precision floating point data type. The output saturation is implemented in this function, thus in case the input value is outside the [-1, 1) interval, the output value is limited to the boundary value.

The output of the function is defined by the following simple equation:

$$f16Out = \begin{cases} FRAC16\_MIN & \text{if } (fltIn1 \cdot fltIn2) < FRAC16\_MIN \\ fltIn1 \cdot fltIn2 \cdot ((tFloat)INT16\_MAX + 1) & \text{if } FRAC16\_MIN \leq (fltIn1 \cdot fltIn2) \leq FRAC16\_MAX \\ FRAC16\_MAX & \text{if } (fltIn1 \cdot fltIn2) > FRAC16\_MAX \end{cases}$$

Equation [MLIB\\_Convert\\_F16FLT\\_Eq1](#)

**Note:** The function may raise floating-point exceptions (invalid operation, overflow, underflow, inexact, input denormal). The floating-point unit must be enabled in CPACR register to prevent NOCP UsageFault exception.

Due to effectivity reason this function is implemented as inline and thus is not ANSI-C compliant.

**Code Example**

```

#include "mlib.h"

tFloat fltIn1, fltIn2;
tFrac16 f16Out;

void main(void)
{
    // input value = 0.25
    fltIn1 = (tFloat)0.25;

    // scale value = 0.5
    fltIn2 = (tFloat)0.5;

    // output should be FRAC16(0.125) = 0x1000
    f16Out = MLIB_Convert_F16FLT(fltIn1, fltIn2);

    // output should be FRAC16(0.125) = 0x1000
    f16Out = MLIB_Convert(fltIn1, fltIn2, F16FLT);

    // scale value = 2
    fltIn2 = (tFloat)2;

    // output should be FRAC16(0.5) = 0x4000
    f16Out = MLIB_Convert_F16FLT(fltIn1, fltIn2);

    // output should be FRAC16(0.5) = 0x4000
    f16Out = MLIB_Convert(fltIn1, fltIn2, F16FLT);
}

```

**2.60.5 Function MLIB\_Convert\_FLTF16****Declaration**

INLINE `tFloat` MLIB\_Convert\_FLTF16(register `tFrac16` f16In1,  
register `tFrac16` f16In2);

**Arguments****Table 315. MLIB\_Convert\_FLTF16 arguments**

Type	Name	Direction	Description
register <code>tFrac16</code>	f16In1	input	Input value in 16-bit fractional format to be converted.
register <code>tFrac16</code>	f16In2	input	Scale factor in 16-bit fractional format.

**Return**

Converted input value in single precision floating point format.

**Implementation details**

The input value is considered as 16-bit fractional data type and output value is considered as single precision floating point data type. The second value is considered as 16-bit fractional data type. The sign of the second value represents the scale mechanism. In case the second value is positive the first input value is multiplied with the

second one and converted to the output format. In case the second value is negative, the first input value is divided by absolute value of second input value and converted to the output format. The output saturation is not implemented in this function.

The output of the function is defined by the following simple equation:

$$f\text{ltOut} = \begin{cases} \frac{f16In1f16In2}{(t\text{Float})INT16_{MAX}+1} & \text{if } f16In2 \geq 0 \\ (t\text{Float})1 & \text{if } f16In2 < 0 \wedge f16In1 \geq -f16In2 \\ (t\text{Float})-1 & \text{if } f16In2 < 0 \wedge f16In1 \leq f16In2 \\ \frac{(t\text{Float})f16In1}{(t\text{Float})f16In2} & \text{otherwise} \end{cases}$$

Equation MLIB\_Convert\_FLTF16\_Eq1

**Note:** The function may raise floating-point exceptions (invalid operation, division by zero, overflow, underflow, inexact, input denormal). The floating-point unit must be enabled in CPACR register to prevent NOCP UsageFault exception.

Due to effectivity reason this function is implemented as inline and thus is not ANSI-C compliant.

### Code Example

```
#include "mlib.h"

tF16 f16In1, f16In2;
tFloat fltOut;

void main(void)
{
    // input value = 0.25 = 0x2000
    f16In1 = FRAC16(0.25);

    // scale value = 0.5 = 0x4000
    f16In2 = FRAC16(0.5);

    // output should be 0.125
    fltOut = MLIB_Convert_FLTF16(f16In1, f16In2);

    // output should be 0.125
    fltOut = MLIB_Convert(f16In1, f16In2, FLTF16);

    // scale value = -0.5 = 0xC000
    f16In2 = FRAC16(-0.5);

    // output should be 0.5
    fltOut = MLIB_Convert_FLTF16(f16In1, f16In2);

    // output should be 0.5
    fltOut = MLIB_Convert(f16In1, f16In2, FLTF16);
}
```

## 2.60.6 Function MLIB\_Convert\_FLTF32

### Declaration

```
INLINE tFloat MLIB_Convert_FLTF32(register tFrac32 f32In1,  
register tFrac32 f32In2);
```

### Arguments

**Table 316. MLIB\_Convert\_FLTF32 arguments**

Type	Name	Direction	Description
register <a href="#">tFrac32</a>	f32In1	input	Input value in 32-bit fractional format to be converted.
register <a href="#">tFrac32</a>	f32In2	input	Scale factor in 32-bit fractional format.

### Return

Converted input value in single precision floating point format.

### Implementation details

The input value is considered as 32-bit fractional data type and output value is considered as single precision floating point data type. The second value is considered as 32-bit fractional data type. The sign of the second value represents the scale mechanism. In case the second value is positive the first input value is multiplied with the second one and converted to the output format. In case the second value is negative, the first input value is divided by absolute value of second input value and converted to the output format. The output saturation is not implemented in this function.

The output of the function is defined by the following simple equation:

$$f_{ltOut} = \begin{cases} \frac{f_{32In1} \cdot f_{32In2}}{(tFloat)INT32\_MAX+1} & \text{if } f_{32In2} \geq 0 \\ (tFloat)1 & \text{if } f_{32In2} < 0 \wedge f_{32In1} \geq -f_{32In2} \\ (tFloat) - 1 & \text{if } f_{32In2} < 0 \wedge f_{32In1} \leq f_{32In2} \\ \frac{(tFloat)f_{32In1}}{(tFloat)f_{32In2}} & \text{otherwise} \end{cases}$$

Equation MLIB\_Convert\_FLTF32\_Eq1

**Note:** The function may raise floating-point exceptions (invalid operation, division by zero, overflow, underflow, inexact, input denormal). The floating-point unit must be enabled in CPACR register to prevent NOCP UsageFault exception.

Due to effectivity reason this function is implemented as inline and thus is not ANSI-C compliant.

### Code Example

```
#include "mlib.h"

tF32 f32In1, f32In2;
tFloat fltOut;

void main(void)
{
    // input value = 0.25 = 0x2000
```

```

f32In1 = FRAC32(0.25);

// scale value = 0.5 = 0x4000
f32In2 = FRAC32(0.5);

// output should be 0.125
fltOut = MLIB_Convert_FLTF32(f32In1, f32In2);

// output should be 0.125
fltOut = MLIB_Convert(f32In1, f32In2, FLTF32);

// scale value = -0.5 = 0xC000
f32In2 = FRAC32(-0.5);

// output should be 0.5
fltOut = MLIB_Convert_FLTF32(f32In1, f32In2);

// output should be 0.5
fltOut = MLIB_Convert(f32In1, f32In2, FLTF32);
}

```

## 2.61 Function [MLIB\\_ConvertPU](#)

This function converts the input value to a different data type.

### Description

This inline function converts the input value to a different data type.

### Re-entrancy

The function is re-entrant.

#### 2.61.1 Function [MLIB\\_ConvertPU\\_F32F16](#)

##### Declaration

```
INLINE tFrac32 MLIB_ConvertPU_F32F16(register tFrac16 f16In);
```

##### Arguments

**Table 317. [MLIB\\_ConvertPU\\_F32F16](#) arguments**

Type	Name	Direction	Description
register <a href="#">tFrac16</a>	f16In	input	Input value in 16-bit fractional format to be converted.

##### Return

Converted input value in 32-bit fractional format.

##### Implementation details

The input value is considered as 16-bit fractional data type and output value is considered as 32-bit fractional data type. The output saturation is not implemented in this function, thus in case the input value is outside the [-1, 1) interval, the output value will overflow.

The output of the function is defined by the following simple equation:

$$f32Out = (tFrac32)f16In$$

Equation MLIB\_ConvertPU\_F32F16\_Eq1

**Note:** Due to effectivity reason this function is implemented as inline, and thus is not ANSI-C compliant.

### Code Example

```
#include "mlib.h"

tFrac16 f16In;
tFrac32 f32Out;

void main(void)
{
    // input value = 0.25 = 0x2000
    f16In = FRAC16(0.25);

    // output should be FRAC32(0.25) = 0x20000000
    f32Out = MLIB_ConvertPU_F32F16(f16In);

    // output should be FRAC32(0.25) = 0x20000000
    f32Out = MLIB_ConvertPU(f16In, F32F16);
}
```

## 2.61.2 Function MLIB\_ConvertPU\_F32FLT

### Declaration

```
INLINE tFrac32 MLIB_ConvertPU_F32FLT(register tFloat fltIn);
```

### Arguments

**Table 318. MLIB\_ConvertPU\_F32FLT arguments**

Type	Name	Direction	Description
register tFloat	fltIn	input	Input value in single precision floating point format to be converted.

### Return

Converted input value in 32-bit fractional format.

### Implementation details

The input value is considered as single precision floating point data type and output value is considered as 32-bit fractional data type. The output saturation is implemented in this function, thus in case the input value is outside the [-1, 1) interval, the output value is limited to the boundary value.

The output of the function is defined by the following simple equation:

$$f32Out = \begin{cases} FRAC32\_MIN & \text{if } fltIn < FRAC32\_MIN \\ \frac{fltIn}{(tFloat)INT32\_MAX+1} & \text{if } FRAC32\_MIN \leq fltIn \leq FRAC32\_MAX \\ FRAC32\_MAX & \text{if } fltIn > FRAC32\_MAX \end{cases}$$

Equation MLIB\_ConvertPU\_F32FLT\_Eq1

**Note:** The function may raise floating-point exceptions (invalid operation, inexact, input denormal). The floating-point unit must be enabled in CPACR register to prevent NOCP UsageFault exception.

Due to effectivity reason this function is implemented as inline, and thus is not ANSI-C compliant.

### Code Example

```
#include "mlib.h"

tFloat fltIn;
tFrac32 f32Out;

void main(void)
{
    // input value = 0.25
    fltIn = (tFloat)0.25;

    // output should be FRAC32(0.25) = 0x20000000
    f32Out = MLIB_ConvertPU_F32FLT(fltIn);

    // output should be FRAC32(0.25) = 0x20000000
    f32Out = MLIB_ConvertPU(fltIn, F32FLT);
}
```

### 2.61.3 Function MLIB\_ConvertPU\_F16F32

#### Declaration

```
INLINE tFrac16 MLIB_ConvertPU_F16F32(register tFrac32 f32In);
```

#### Arguments

Table 319. MLIB\_ConvertPU\_F16F32 arguments

Type	Name	Direction	Description
register tFrac32	f32In	input	Input value in 32-bit fractional format to be converted.

#### Return

Converted input value in 16-bit fractional format.

#### Implementation details

The input value is considered as 32-bit fractional data type and output value is considered as 16-bit fractional data type. The output saturation is not implemented in this function, thus in case the input value is outside the [-1, 1) interval, the output value will overflow without any detection.

The output of the function is defined by the following simple equation:

$$f16Out = (tFrac16)f32In$$

Equation MLIB\_ConvertPU\_F16F32\_Eq1

**Note:** Due to effectivity reason this function is implemented as inline, and thus is not ANSI-C compliant.

### Code Example

```
#include "mlib.h"

tFrac32 f32In;
tFrac16 f16Out;

void main(void)
{
    // input value = 0.25 = 0x2000 0000
    f32In = FRAC32(0.25);

    // output should be FRAC16(0.25) = 0x2000
    f16Out = MLIB_ConvertPU_F16F32(f32In);

    // output should be FRAC16(0.25) = 0x2000
    f16Out = MLIB_ConvertPU(f32In, F16F32);
}
```

## 2.61.4 Function MLIB\_ConvertPU\_F16FLT

### Declaration

```
INLINE tFrac16 MLIB_ConvertPU_F16FLT(register tFloat fltIn);
```

### Arguments

Table 320. MLIB\_ConvertPU\_F16FLT arguments

Type	Name	Direction	Description
register tFloat	fltIn	input	Input value in single precision floating point format to be converted.

### Return

Converted input value in 16-bit fractional format.

### Implementation details

The input value is considered as single precision floating point data type and output value is considered as 16-bit fractional data type. The output saturation is implemented in this function, thus in case the input value is outside the [-1, 1) interval, the output value is limited to the boundary value.

The output of the function is defined by the following simple equation:

$$f16Out = \begin{cases} FRAC16\_MIN & \text{if } f16In < FRAC16\_MIN \\ \frac{f16In}{(tFloat)INT16\_MAX+1} & \text{if } FRAC16\_MIN \leq f16In \leq FRAC16\_MAX \\ FRAC16\_MAX & \text{if } f16In > FRAC16\_MAX \end{cases}$$

Equation MLIB\_ConvertPU\_F16FLT\_Eq1

**Note:** The function may raise floating-point exceptions (invalid operation, inexact, input denormal). The floating-point unit must be enabled in CPACR register to prevent NOCP UsageFault exception.

Due to effectivity reason this function is implemented as inline, and thus is not ANSI-C compliant.

### Code Example

```
#include "mlib.h"

tFloat fltIn;
tFrac16 f16Out;

void main(void)
{
    // input value = 0.25
    fltIn = (tFloat)0.25;

    // output should be FRAC16(0.25) = 0x2000
    f16Out = MLIB_ConvertPU_F16FLT(fltIn);

    // output should be FRAC16(0.25) = 0x2000
    f16Out = MLIB_ConvertPU(fltIn, F16FLT);
}
```

## 2.61.5 Function MLIB\_ConvertPU\_FLTF16

### Declaration

```
INLINE tFloat MLIB_ConvertPU_FLTF16(register tFrac16 f16In);
```

### Arguments

Table 321. MLIB\_ConvertPU\_FLTF16 arguments

Type	Name	Direction	Description
register tFrac16	f16In	input	Input value in 16-bit fractional format to be converted.

### Return

Converted input value in single precision floating point format.

### Implementation details

The input value is considered as 16-bit fractional data type and output value is considered as single precision floating point data type. The output saturation is not implemented in this function.

The output of the function is defined by the following simple equation:

$$f1tOut = \frac{(tFloat)f16In}{(tFloat)INT16\_MAX+1}$$

Equation MLIB\_ConvertPU\_FLTF16\_Eq1

**Note:** The function may raise floating-point exceptions (invalid operation, inexact, input denormal). The floating-point unit must be enabled in CPACR register to prevent NOCP UsageFault exception.

Due to effectivity reason this function is implemented as inline, and thus is not ANSI-C compliant.

### Code Example

```
#include "mlib.h"

tF16 f16In;
tFloat fltOut;

void main(void)
{
    // input value = 0.25 = 0x2000
    f16In = FRAC16(0.25);

    // output should be 0.25
    fltOut = MLIB_ConvertPU_FLTF16(f16In);

    // output should be 0.25
    fltOut = MLIB_ConvertPU(f16In, FLTF16);
}
```

## 2.61.6 Function MLIB\_ConvertPU\_FLTF32

### Declaration

```
INLINE tFloat MLIB_ConvertPU_FLTF32(register tFrac32 f32In);
```

### Arguments

Table 322. MLIB\_ConvertPU\_FLTF32 arguments

Type	Name	Direction	Description
register tFrac32	f32In	input	Input value in 32-bit fractional format to be converted.

### Return

Converted input value in single precision floating point format.

### Implementation details

The input value is considered as 32-bit fractional data type and output value is considered as single precision floating point data type. The output saturation is not implemented in this function.

The output of the function is defined by the following simple equation:

$$f32In = \frac{(tFloat)f32In}{(tFloat)INT32\_MAX+1}$$

Equation MLIB\_ConvertPU\_FLT32\_Eq1

**Note:** The function may raise floating-point exceptions (invalid operation, inexact, input denormal). The floating-point unit must be enabled in CPACR register to prevent NOCP UsageFault exception.

Due to effectivity reason this function is implemented as inline, and thus is not ANSI-C compliant.

### Code Example

```
#include "mlib.h"

tF32 f32In;
tFloat fltOut;

void main(void)
{
    // input value = 0.25 = 0x20000000
    f32In = FRAC32(0.25);

    // output should be 0.25
    fltOut = MLIB_ConvertPU_FLT32(f32In);

    // output should be 0.25
    fltOut = MLIB_ConvertPU(f32In, FLTF32);
}
```

## 2.62 Function MLIB\_Div

This function divides the first parameter by the second one.

### Description

This inline function returns the division of two input values. The first input value is numerator and the second input value is denominator.

### Re-entrancy

The function is re-entrant.

#### 2.62.1 Function MLIB\_Div\_F32

##### Declaration

```
INLINE tFrac32 MLIB_Div_F32(register tFrac32 f32In1, register
tFrac32 f32In2);
```

**Arguments****Table 323. MLIB\_Div\_F32 arguments**

Type	Name	Direction	Description
register tFrac32	f32In1	input	Numerator of division.
register tFrac32	f32In2	input	Denominator of division.

**Return**

Division of two input values.

**Implementation details**

The input values as well as output value is considered as 32-bit fractional data type. The output saturation is not implemented in this function, thus in case the numerator is greater or equal to denominator, the output value is undefined. The function will never cause division by zero exception.

**Note:** Due to effectivity reason this function is implemented as inline, and thus is not ANSI-C compliant.

**Caution:** Due to effectivity reason the division is calculated in 16-bit precision.

**Code Example**

```
#include "mlib.h"

tFrac32 f32In1,f32In2;
tFrac32 f32Out;

void main(void)
{
    // input value 1 = 0.25
    f32In1 = FRAC32(0.25);
    // input value 2 = 0.5
    f32In2 = FRAC32(0.5);

    // output should be FRAC32(0.5) = 0x40000000
    f32Out = MLIB_Div_F32(f32In1,f32In2);

    // output should be FRAC32(0.5) = 0x40000000
    f32Out = MLIB_Div(f32In1,f32In2,F32);

    // ##### Available only if 32-bit fractional implementation selected
    // as default
    // #####
    // output should be FRAC32(0.5) = 0x40000000
    f32Out = MLIB_Div(f32In1,f32In2);
}
```

## 2.62.2 Function MLIB\_Div\_F16

### Declaration

```
INLINE tFrac16 MLIB_Div_F16(register tFrac16 f16In1, register
tFrac16 f16In2);
```

### Arguments

**Table 324. MLIB\_Div\_F16 arguments**

Type	Name	Direction	Description
register <u>tFrac16</u>	f16In1	input	Numerator of division.
register <u>tFrac16</u>	f16In2	input	Denominator of division.

### Return

Division of two input values.

### Implementation details

The input values as well as output value is considered as 16-bit fractional data type. The output saturation is not implemented in this function, thus in case the numerator is greater or equal to denominator, the output value is undefined. The function will never cause division by zero exception.

**Note:** Due to effectivity reason this function is implemented as inline, and thus is not ANSI-C compliant.

### Code Example

```
#include "mlib.h"

tFrac16 f16In1,f16In2;
tFrac16 f16Out;

void main(void)
{
    // input value 1 = 0.25
    f16In1 = FRAC16(0.25);
    // input value 2 = 0.5
    f16In2 = FRAC16(0.5);

    // output should be FRAC16(0.5) = 0x4000
    f16Out = MLIB_Div_F16(f16In1,f16In2);

    // output should be FRAC16(0.5) = 0x4000
    f16Out = MLIB_Div(f16In1,f16In2,F16);

    // ######
    // Available only if 16-bit fractional implementation selected
    // as default
    // #####
    // output should be FRAC16(0.5) = 0x4000
    f16Out = MLIB_Div(f16In1,f16In2);
}
```

### 2.62.3 Function MLIB\_Div\_FLT

#### Declaration

```
INLINE tFloat MLIB_Div_FLT(register tFloat fltIn1, register
tFloat fltIn2);
```

#### Arguments

**Table 325. MLIB\_Div\_FLT arguments**

Type	Name	Direction	Description
register <b>tFloat</b>	fltIn1	<b>input</b>	Numerator of division.
register <b>tFloat</b>	fltIn2	<b>input</b>	Denominator of division.

#### Return

Division of two input values.

#### Implementation details

The input values as well as output value is considered as single precision floating point data type. Testing of input values for division by zero is not implemented in this function.

**Note:** The function may raise floating-point exceptions (invalid operation,division by zero, overflow,underflow,inexact,input denormal). The floating-point unit must be enabled in CPACR register to prevent NOCP UsageFault exception.

*Due to effectivity reason this function is implemented as inline, and thus is not ANSI-C compliant.*

#### Code Example

```
#include "mlib.h"

tFloat fltIn1, fltIn2;
tFloat fltOut;

void main(void)
{
    // input value 1 = 0.25
    fltIn1 = (tFloat)0.25;
    // input value 2 = 0.5
    fltIn2 = (tFloat)0.5;

    // output should be 0.5
    fltOut = MLIB_Div_FLT(fltIn1, fltIn2);

    // output should be 0.5
    fltOut = MLIB_Div(fltIn1, fltIn2, FLT);

    // ##### Available only if single precision floating point
    // implementation selected as default
    // #####
}

// output should be 0.5
```

```

    fltOut = MLIB_Div(fltIn1, fltIn2);
}

```

## 2.63 Function MLIB\_DivSat

This function divides the first parameter by the second one and saturate.

### Description

This inline function returns the saturated division of two input values. The first input value is numerator and the second input value is denominator.

### Re-entrancy

The function is re-entrant.

#### 2.63.1 Function MLIB\_DivSat\_F32

##### Declaration

```
INLINE tFrac32 MLIB_DivSat_F32(register tFrac32 f32In1, register tFrac32 f32In2);
```

##### Arguments

**Table 326. MLIB\_DivSat\_F32 arguments**

Type	Name	Direction	Description
register <a href="#">tFrac32</a>	f32In1	input	Numerator of division.
register <a href="#">tFrac32</a>	f32In2	input	Denominator of division.

##### Return

Division of two input values, saturated if necessary.

##### Implementation details

The input values as well as output value is considered as 32-bit fractional data type.

The output of the function is defined by the following simple equation:

$$f32Out = \begin{cases} FRAC32\_MIN & \text{if } \frac{f32In1}{f32In2} < FRAC32\_MIN \\ \frac{f32In1}{f32In2} & \text{if } FRAC32\_MIN \leq \frac{f32In1}{f32In2} \leq FRAC32\_MAX \\ FRAC32\_MAX & \text{if } \frac{f32In1}{f32In2} > FRAC32\_MAX \end{cases}$$

Equation MLIB\_DivSat\_F32\_Eq1

**Note:** Due to effectivity reason this function is implemented as inline, and thus is not ANSI-C compliant.

##### Code Example

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

```

tFrac32 f32In1, f32In2;
tFrac32 f32Out;

void main(void)
{
    // input value 1 = 0.25
    f32In1 = FRAC32(0.25);
    // input value 2 = 0.5
    f32In2 = FRAC32(0.5);

    // output should be FRAC32(0.5) = 0x40000000
    f32Out = MLIB_DivSat_F32(f32In1, f32In2);

    // output should be FRAC32(0.5) = 0x40000000
    f32Out = MLIB_DivSat(f32In1, f32In2, F32);

    // ##### Available only if 32-bit fractional implementation selected
    // as default
    // #####
    // output should be FRAC32(0.5) = 0x40000000
    f32Out = MLIB_DivSat(f32In1, f32In2);
}

```

## 2.63.2 Function MLIB\_DivSat\_F16

### Declaration

```
INLINE tFrac16 MLIB_DivSat_F16(register tFrac16 f16In1, register tFrac16 f16In2);
```

### Arguments

**Table 327. MLIB\_DivSat\_F16 arguments**

Type	Name	Direction	Description
register tFrac16	f16In1	input	Numerator of division.
register tFrac16	f16In2	input	Denominator of division.

### Return

Division of two input values, saturated if necessary.

### Implementation details

The input values as well as output value is considered as 16-bit fractional data type.

The output of the function is defined by the following simple equation:

$$f16Out = \begin{cases} FRAC16\_MIN & \text{if } \frac{f16In1}{f16In2} < FRAC16\_MIN \\ \frac{f16In1}{f16In2} & \text{if } FRAC16\_MIN \leq \frac{f16In1}{f16In2} \leq FRAC16\_MAX \\ FRAC16\_MAX & \text{if } \frac{f16In1}{f16In2} > FRAC16\_MAX \end{cases}$$

Equation MLIB\_DivSat\_F16\_Eq1

**Note:** Due to effectivity reason this function is implemented as inline, and thus is not ANSI-C compliant.

### Code Example

```
#include "mlib.h"

tFrac16 f16In1, f16In2;
tFrac16 f16Out;

void main(void)
{
    // input value 1 = 0.25
    f16In1 = FRAC16(0.25);
    // input value 2 = 0.5
    f16In2 = FRAC16(0.5);

    // output should be FRAC16(0.5) = 0x4000
    f16Out = MLIB_DivSat_F16(f16In1, f16In2);

    // output should be FRAC16(0.5) = 0x4000
    f16Out = MLIB_DivSat(f16In1, f16In2, F16);

    // ##### Available only if 16-bit fractional implementation selected
    // as default
    // #####
    // output should be FRAC16(0.5) = 0x4000
    f16Out = MLIB_DivSat(f16In1, f16In2);
}
```

## 2.64 Function MLIB\_Mac

This function implements the multiply accumulate function.

### Description

This inline function returns the multiplied second and third input value with adding of first input value.

### Re-entrancy

The function is re-entrant.

### 2.64.1 Function MLIB\_Mac\_F32

#### Declaration

```
INLINE tFrac32 MLIB_Mac_F32(register tFrac32 f32In1, register
tFrac32 f32In2, register tFrac32 f32In3);
```

**Arguments****Table 328. MLIB\_Mac\_F32 arguments**

Type	Name	Direction	Description
register tFrac32	f32In1	input	Input value to be add.
register tFrac32	f32In2	input	First value to be multiplied.
register tFrac32	f32In3	input	Second value to be multiplied.

**Return**

Multiplied second and third input value with adding of first input value.

**Implementation details**

The input values as well as output value is considered as 32-bit fractional values. The output saturation is not implemented in this function, thus in case the output value is outside the [-1, 1) interval, the output value will overflow.

The output of the function is defined by the following simple equation:

$$f32Out = f32In1 + (f32In2 \cdot f32In3)$$

Equation MLIB\_Mac\_F32\_Eq1

**Note:** Due to effectivity reason this function is implemented as inline, and thus is not ANSI-C compliant.

**Code Example**

```
#include "mlib.h"

tFrac32 f32In1;
tFrac32 f32In2;
tFrac32 f32In3;
tFrac32 f32Out;

void main(void)
{
    // input1 value = 0.25
    f32In1 = FRAC32(0.25);

    // input2 value = 0.15
    f32In2 = FRAC32(0.15);

    // input3 value = 0.35
    f32In3 = FRAC32(0.35);

    // output should be FRAC32(0.3025) = 0x26B851EB
    f32Out = MLIB_Mac_F32(f32In1, f32In2, f32In3);

    // output should be FRAC32(0.3025) = 0x26B851EB
    f32Out = MLIB_Mac(f32In1, f32In2, f32In3, F32);

    // ##### Available only if 32-bit fractional implementation selected
    // as default
    // #####
}
```

```
// output should be FRAC32(0.3025) = 0x26B851EB
f32Out = MLIB_Mac(f32In1, f32In2, f32In3);
}
```

## 2.64.2 Function MLIB\_Mac\_F32F16F16

### Declaration

```
INLINE tFrac32 MLIB_Mac_F32F16F16(register tFrac32 f32In1,  
register tFrac16 f16In2, register tFrac16 f16In3);
```

### Arguments

**Table 329. MLIB\_Mac\_F32F16F16 arguments**

Type	Name	Direction	Description
register <a href="#">tFrac32</a>	f32In1	input	Input value to be add.
register <a href="#">tFrac16</a>	f16In2	input	First value to be multiplied.
register <a href="#">tFrac16</a>	f16In3	input	Second value to be multiplied.

### Return

Multiplied second and third input value with adding of first input value.

### Implementation details

The first input value as well as output value is considered as 32-bit fractional values. The second and third input values are considered as 16-bit fractional values. The output saturation is not implemented in this function, thus in case the output value is outside the [-1, 1) interval, the output value will overflow.

The output of the function is defined by the following simple equation:

$$f32Out = f32In1 + (f16In2 \cdot f16In3)$$

Equation MLIB\_Mac\_F32F16F16\_Eq1

This implementation is available if 32-bit fractional implementations are enabled. However it is not possible to use the default implementation based function call, thus the implementation post-fix or additional parameter function call shall be used.

**Note:** Due to effectivity reason this function is implemented as inline, and thus is not ANSI-C compliant.

### Code Example

```
#include "mlib.h"

tFrac32 f32In1;
tFrac16 f16In2;
tFrac16 f16In3;
tFrac32 f32Out;

void main(void)
{
```

```

// input1 value = 0.25
f32In1 = FRAC32(0.25);

// input2 value = 0.15
f16In2 = FRAC16(0.15);

// input3 value = 0.35
f16In3 = FRAC16(0.35);

// output should be FRAC32(0.3025) = 0x26B851EB
f32Out = MLIB_Mac_F32F16F16(f32In1, f16In2, f16In3);

// output should be FRAC32(0.3025) = 0x26B851EB
f32Out = MLIB_Mac(f32In1, f32In2, f32In3, F32F16F16);

}

```

### 2.64.3 Function MLIB\_Mac\_F16

#### Declaration

```
INLINE tFrac16 MLIB_Mac_F16(register tFrac16 f16In1, register
tFrac16 f16In2, register tFrac16 f16In3);
```

#### Arguments

**Table 330. MLIB\_Mac\_F16 arguments**

Type	Name	Direction	Description
register <u>tFrac16</u>	f16In1	input	Input value to be add.
register <u>tFrac16</u>	f16In2	input	First value to be multiplied.
register <u>tFrac16</u>	f16In3	input	Second value to be multiplied.

#### Return

Multiplied second and third input value with adding of first input value.

#### Implementation details

The input values as well as output value is considered as 16-bit fractional values. The output saturation is not implemented in this function, thus in case the output value is outside the [-1, 1) interval, the output value will overflow.

The output of the function is defined by the following simple equation:

$$f16Out = f16In1 + (f16In2 \cdot f16In3)$$

Equation MLIB\_Mac\_F16\_Eq1

**Note:** Due to effectivity reason this function is implemented as inline, and thus is not ANSI-C compliant.

#### Code Example

```
#include "mlib.h"

tFrac16 f16In1;
```

```

tFrac16 f16In2;
tFrac16 f16In3;
tFrac16 f16Out;

void main(void)
{
    // input1 value = 0.25
    f16In1 = FRAC16(0.25);

    // input2 value = 0.15
    f16In2 = FRAC16(0.15);

    // input3 value = 0.35
    f16In3 = FRAC16(0.35);

    // output should be FRAC16(0.3025) = 0x26B8
    f16Out = MLIB_Mac_F16(f16In1, f16In2, f16In3);

    // output should be FRAC16(0.3025) = 0x26B8
    f16Out = MLIB_Mac(f16In1, f16In2, f16In3, F16);

    // ##### Available only if 16-bit fractional implementation selected
    // as default
    // #####
    // output should be FRAC16(0.3025) = 0x26B8
    f16Out = MLIB_Mac(f16In1, f16In2, f16In3);
}

```

#### 2.64.4 Function **MLIB\_Mac\_FLT**

##### Declaration

```
INLINE tFloat MLIB_Mac_FLT(register tFloat fltIn1, register
tFloat fltIn2, register tFloat fltIn3);
```

##### Arguments

**Table 331. MLIB\_Mac\_FLT arguments**

Type	Name	Direction	Description
register tFloat	fltIn1	input	Input value to be add.
register tFloat	fltIn2	input	First value to be multiplied.
register tFloat	fltIn3	input	Second value to be multiplied.

##### Return

Multiplied second and third input value with adding of first input value.

##### Implementation details

The input values as well as output value are considered as single precision floating point data type. Intermediate results are computed in infinite precision.

The output of the function is defined by the following simple equation:

$$fltOut = fltIn1 + (fltIn2 \cdot fltIn3)$$

Equation MLIB\_MacFLT\_Eq1

**Note:** The function may raise floating-point exceptions (invalid operation, overflow, underflow, inexact, input denormal). The floating-point unit must be enabled in CPACR register to prevent NOCP UsageFault exception.

Due to effectivity reason this function is implemented as inline assembly, and thus is not ANSI-C compliant.

### Code Example

```
#include "mlib.h"

tFloat fltIn1;
tFloat fltIn2;
tFloat fltIn3;
tFloat fltOut;

void main(void)
{
    // input1 value = 0.25
    fltIn1 = (tFloat)0.25;

    // input2 value = 0.15
    fltIn2 = (tFloat)0.15;

    // input3 value = 0.35
    fltIn3 = (tFloat)0.35;

    // output should be 0.3025
    fltOut = MLIB_MacFLT(fltIn1, fltIn2, fltIn3);

    // output should be 0.3025
    fltOut = MLIB_Mac(fltIn1, fltIn2, fltIn3, FLT);

    // ######
    // Available only if single precision floating point
    // implementation selected as default
    // #####
    // output should be 0.3025
    fltOut = MLIB_Mac(fltIn1, fltIn2, fltIn3);
}
```

## 2.65 Function MLIB\_MacSat

This function implements the multiply accumulate function saturated if necessary.

### Description

This inline function returns the multiplied second and third input value with adding of first input value. The output value is saturated if necessary.

## Re-entrancy

The function is re-entrant.

### 2.65.1 Function MLIB\_MacSat\_F32

#### Declaration

```
INLINE tFrac32 MLIB_MacSat_F32(register tFrac32 f32In1, register  
tFrac32 f32In2, register tFrac32 f32In3);
```

#### Arguments

**Table 332. MLIB\_MacSat\_F32 arguments**

Type	Name	Direction	Description
register <a href="#">tFrac32</a>	f32In1	<b>input</b>	Input value to be add.
register <a href="#">tFrac32</a>	f32In2	<b>input</b>	First value to be multiplied.
register <a href="#">tFrac32</a>	f32In3	<b>input</b>	Second value to be multiplied.

#### Return

Multiplied second and third input value with adding of first input value. The output value is saturated if necessary.

#### Implementation details

The input values as well as output value is considered as 32-bit fractional values.

The output of the function is defined by the following simple equation:

$$f32Out = \begin{cases} FRAC32\_MIN & \text{if } (f32In1 + (f32In2 \cdot f32In3)) < FRAC32\_MIN \\ f32In1 + (f32In2 \cdot f32In3) & \text{if } FRAC32\_MIN \leq (f32In1 + (f32In2 \cdot f32In3)) \leq FRAC32\_MAX \\ FRAC32\_MAX & \text{if } (f32In1 + (f32In2 \cdot f32In3)) > FRAC32\_MAX \end{cases}$$

Equation MLIB\_MacSat\_F32\_Eq1

**Note:** Due to effectivity reason this function is implemented as inline and thus is not ANSI-C compliant.

#### Code Example

```
#include "mlib.h"

tFrac32 f32In1;
tFrac32 f32In2;
tFrac32 f32In3;
tFrac32 f32Out;

void main(void)
{
    // input1 value = 0.25
    f32In1 = FRAC32(0.25);

    // input2 value = 0.15
    f32In2 = FRAC32(0.15);
```

```

// input3 value = 0.35
f32In3 = FRAC32(0.35);

// output should be FRAC32(0.3025) = 0x26B851EB
f32Out = MLIB_MacSat_F32(f32In1, f32In2, f32In3);

// output should be FRAC32(0.3025) = 0x26B851EB
f32Out = MLIB_MacSat(f32In1, f32In2, f32In3, F32);

// ##### Available only if 32-bit fractional implementation selected
// as default
// #####
// output should be FRAC32(0.3025) = 0x26B851EB
f32Out = MLIB_MacSat(f32In1, f32In2, f32In3);
}

```

## 2.65.2 Function MLIB\_MacSat\_F32F16F16

### Declaration

```
INLINE tFrac32 MLIB_MacSat_F32F16F16(register tFrac32 f32In1,
register tFrac16 f16In2, register tFrac16 f16In3);
```

### Arguments

**Table 333. MLIB\_MacSat\_F32F16F16 arguments**

Type	Name	Direction	Description
register tFrac32	f32In1	input	Input value to be add.
register tFrac16	f16In2	input	First value to be multiplied.
register tFrac16	f16In3	input	Second value to be multiplied.

### Return

Multiplied second and third input value with adding of first input value. The output value is saturated if necessary.

### Implementation details

The first input values as well as output value is considered as 32-bit fractional values, second and third input values are considered as 16-bit fractional values.

The output of the function is defined by the following simple equation:

$$f32Out = \begin{cases} FRAC32\_MIN & \text{if } (f32In1 + (f16In2 \cdot f16In3)) < FRAC32\_MIN \\ f32In1 + (f16In2 \cdot f16In3) & \text{if } FRAC32\_MIN \leq (f32In1 + (f16In2 \cdot f16In3)) \leq FRAC32\_MAX \\ FRAC32\_MAX & \text{if } (f32In1 + (f16In2 \cdot f16In3)) > FRAC32\_MAX \end{cases}$$

Equation MLIB\_MacSat\_F32F16F16\_Eq1

**Note:** Due to effectivity reason this function is implemented as inline and thus is not ANSI-C compliant.

**Code Example**

```

#include "mlib.h"

tFrac32 f32In1;
tFrac16 f16In2;
tFrac16 f16In3;
tFrac32 f32Out;

void main(void)
{
    // input1 value = 0.25
    f32In1 = FRAC32(0.25);

    // input2 value = 0.15
    f16In2 = FRAC16(0.15);

    // input3 value = 0.35
    f16In3 = FRAC16(0.35);

    // output should be FRAC32(0.3025) = 0x26B851EB
    f32Out = MLIB_MacSat_F32F16F16(f32In1, f16In2, f16In3);

    // output should be FRAC32(0.3025) = 0x26B851EB
    f32Out = MLIB_MacSat(f32In1, f16In2, f16In3, F32F16F16);
}

```

**2.65.3 Function MLIB\_MacSat\_F16****Declaration**

INLINE tFrac16 MLIB\_MacSat\_F16(register tFrac16 f16In1, register tFrac16 f16In2, register tFrac16 f16In3);

**Arguments****Table 334. MLIB\_MacSat\_F16 arguments**

Type	Name	Direction	Description
register tFrac16	f16In1	input	Input value to be add.
register tFrac16	f16In2	input	First value to be multiplied.
register tFrac16	f16In3	input	Second value to be multiplied.

**Return**

Multiplied second and third input value with adding of first input value. The output value is saturated if necessary.

**Implementation details**

The input values as well as output value is considered as 16-bit fractional values.

The output of the function is defined by the following simple equation:

$$f16Out = \begin{cases} FRAC16\_MIN & \text{if } (f16In1 + (f16In2 \cdot f16In3)) < FRAC16\_MIN \\ f16In1 + (f16In2 \cdot f16In3) & \text{if } FRAC16\_MIN \leq (f16In1 + (f16In2 \cdot f16In3)) \leq FRAC16\_MAX \\ FRAC16\_MAX & \text{if } (f16In1 + (f16In2 \cdot f16In3)) > FRAC16\_MAX \end{cases}$$

Equation MLIB\_MacSat\_F16\_Eq1

**Note:** Due to effectivity reason this function is implemented as inline and thus is not ANSI-C compliant.

### Code Example

```
#include "mlib.h"

tFrac16 f16In1;
tFrac16 f16In2;
tFrac16 f16In3;
tFrac16 f16Out;

void main(void)
{
    // input1 value = 0.25
    f16In1 = FRAC16(0.25);

    // input2 value = 0.15
    f16In2 = FRAC16(0.15);

    // input3 value = 0.35
    f16In3 = FRAC16(0.35);

    // output should be FRAC16(0.3025) = 0x26B8
    f16Out = MLIB_MacSat_F16(f16In1, f16In2, f16In3);

    // output should be FRAC16(0.3025) = 0x26B8
    f16Out = MLIB_MacSat(f16In1, f16In2, f16In3, F16);

    // ##### Available only if 16-bit fractional implementation selected
    // as default
    // #####
}

// output should be FRAC16(0.3025) = 0x26B8
f16Out = MLIB_MacSat(f16In1, f16In2, f16In3);
```

## 2.66 Function MLIB\_Mnac

This function implements the multiply-subtract function.

### Description

This inline function returns the multiplied second and third input value with subtracted first input value.

### Re-entrancy

The function is re-entrant.

### 2.66.1 Function MLIB\_Mnac\_F32

#### Declaration

```
INLINE tFrac32 MLIB_Mnac_F32(register tFrac32 f32In1, register tFrac32 f32In2, register tFrac32 f32In3);
```

#### Arguments

**Table 335. MLIB\_Mnac\_F32 arguments**

Type	Name	Direction	Description
register <a href="#">tFrac32</a>	f32In1	<a href="#">input</a>	Input value to be subtracted.
register <a href="#">tFrac32</a>	f32In2	<a href="#">input</a>	First value to be multiplied.
register <a href="#">tFrac32</a>	f32In3	<a href="#">input</a>	Second value to be multiplied.

#### Return

Multiplied second and third input value with subtracted first input value.

#### Implementation details

The input values as well as output value is considered as 32-bit fractional values. The output saturation is not implemented in this function, thus in case the output value is outside the [-1, 1) interval, the output value will overflow.

The output of the function is defined by the following simple equation:

$$f32Out = -f32In1 + (f32In2 \cdot f32In3)$$

Equation MLIB\_Mnac\_F32\_Eq1

**Note:** Due to effectivity reason this function is implemented as inline, and thus is not ANSI-C compliant.

#### Code Example

```
#include "mlib.h"

tFrac32 f32In1;
tFrac32 f32In2;
tFrac32 f32In3;
tFrac32 f32Out;

void main(void)
{
    // input1 value = 0.0625
    f32In1 = FRAC32(0.0625);

    // input2 value = 0.5
    f32In2 = FRAC32(0.5);

    // input3 value = 0.25
    f32In3 = FRAC32(0.25);

    // output should be FRAC32(0.0625) = 0x08000000
    f32Out = MLIB_Mnac_F32(f32In1, f32In2, f32In3);
```

```

// output should be FRAC32(0.0625) = 0x08000000
f32Out = MLIB_Mnac(f32In1, f32In2, f32In3, F32);

// ##### Available only if 32-bit fractional implementation selected
// as default
// #####
// output should be FRAC32(0.0625) = 0x08000000
f32Out = MLIB_Mnac(f32In1, f32In2, f32In3);
}

```

## 2.66.2 Function `MLIB_Mnac_F32F16F16`

### Declaration

```
INLINE tFrac32 MLIB_Mnac_F32F16F16(register tFrac32 f32In1,  
register tFrac16 f16In2, register tFrac16 f16In3);
```

### Arguments

**Table 336. `MLIB_Mnac_F32F16F16` arguments**

Type	Name	Direction	Description
register <code>tFrac32</code>	f32In1	input	Input value to be subtracted.
register <code>tFrac16</code>	f16In2	input	First value to be multiplied.
register <code>tFrac16</code>	f16In3	input	Second value to be multiplied.

### Return

Multiplied second and third input value with subtracted first input value.

### Implementation details

The first input value as well as output value is considered as 32-bit fractional values. The second and third input values are considered as 16-bit fractional values. The output saturation is not implemented in this function, thus in case the output value is outside the [-1, 1) interval, the output value will overflow.

The output of the function is defined by the following simple equation:

$$f32Out = -f32In1 + (f16In2 \cdot f16In3)$$

Equation `MLIB_Mnac_F32F16F16_Eq1`

This implementation is available if 32-bit fractional implementations are enabled. However it is not possible to use the default implementation based function call, thus the implementation post-fix or additional parameter function call shall be used.

**Note:** Due to effectivity reason this function is implemented as inline, and thus is not ANSI-C compliant.

### Code Example

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

```

tFrac32 f32In1;
tFrac16 f16In2;
tFrac16 f16In3;
tFrac32 f32Out;

void main(void)
{
    // input1 value = 0.0625
    f32In1 = FRAC32(0.0625);

    // input2 value = 0.5
    f16In2 = FRAC16(0.5);

    // input3 value = 0.25
    f16In3 = FRAC16(0.25);

    // output should be FRAC32(0.0625) = 0x08000000
    f32Out = MLIB_Mnac_F32F16F16(f32In1, f16In2, f16In3);

    // output should be FRAC32(0.0625) = 0x08000000
    f32Out = MLIB_Mnac(f32In1, f32In2, f32In3, F32F16F16);

}

```

### 2.66.3 Function MLIB\_Mnac\_F16

#### Declaration

INLINE tFrac16 MLIB\_Mnac\_F16(register tFrac16 f16In1, register tFrac16 f16In2, register tFrac16 f16In3);

#### Arguments

**Table 337. MLIB\_Mnac\_F16 arguments**

Type	Name	Direction	Description
register tFrac16	f16In1	input	Input value to be subtracted.
register tFrac16	f16In2	input	First value to be multiplied.
register tFrac16	f16In3	input	Second value to be multiplied.

#### Return

Multiplied second and third input value with subtracted first input value.

#### Implementation details

The input values as well as output value is considered as 16-bit fractional values. The output saturation is not implemented in this function, thus in case the output value is outside the [-1, 1) interval, the output value will overflow.

The output of the function is defined by the following simple equation:

$$f16Out = -f16In1 + (f16In2 \cdot f16In3)$$

Equation MLIB\_Mnac\_F16\_Eq1

**Note:** Due to effectivity reason this function is implemented as inline, and thus is not ANSI-C compliant.

### Code Example

```
#include "mlib.h"

tFrac16 f16In1;
tFrac16 f16In2;
tFrac16 f16In3;
tFrac16 f16Out;

void main(void)
{
    // input1 value = 0.0625
    f16In1 = FRAC16(0.0625);

    // input2 value = 0.5
    f16In2 = FRAC16(0.5);

    // input3 value = 0.25
    f16In3 = FRAC16(0.25);

    // output should be FRAC16(0.0625) = 0x0800
    f16Out = MLIB_Mnac_F16(f16In1, f16In2, f16In3);

    // output should be FRAC16(0.0625) = 0x0800
    f16Out = MLIB_Mnac(f16In1, f16In2, f16In3, F16);

    // ##### Available only if 16-bit fractional implementation selected
    // as default
    // #####
    // output should be FRAC16(0.0625) = 0x0800
    f16Out = MLIB_Mnac(f16In1, f16In2, f16In3);
}
```

## 2.66.4 Function MLIB\_Mnac\_FLT

### Declaration

```
INLINE tFloat MLIB_Mnac_FLT(register tFloat fltIn1, register
tFloat fltIn2, register tFloat fltIn3);
```

### Arguments

Table 338. MLIB\_Mnac\_FLT arguments

Type	Name	Direction	Description
register tFloat	fltIn1	input	Input value to be subtracted.
register tFloat	fltIn2	input	First value to be multiplied.
register tFloat	fltIn3	input	Second value to be multiplied.

### Return

Multipled second and third input value with subtracted first input value.

### Implementation details

The input values as well as output value are considered as single precision floating point data type. Intermediate results are computed in infinite precision.

The output of the function is defined by the following simple equation:

$$\text{fltOut} = -\text{fltIn1} + (\text{fltIn2} \cdot \text{fltIn3})$$

Equation MLIB\_Mnac\_FLT\_Eq1

**Note:** The function may raise floating-point exceptions (invalid operation, overflow, underflow, inexact, input denormal). The floating-point unit must be enabled in CPACR register to prevent NOCP UsageFault exception.

Due to effectivity reason this function is implemented as inline assembly, and thus is not ANSI-C compliant.

### Code Example

```
#include "mlib.h"

tFloat fltIn1;
tFloat fltIn2;
tFloat fltIn3;
tFloat fltOut;

void main(void)
{
    // input1 value = 5.775817036628723e-01
    fltIn1 = (tFloat)5.775817036628723e-01f;

    // input2 value = 9.133758544921875e-01
    fltIn2 = (tFloat)9.133758544921875e-01f;

    // input3 value = 6.323592662811279e-01
    fltIn3 = (tFloat)6.323592662811279e-01f;

    // output should be -1.8477294e-08
    fltOut = MLIB_Mnac_FLT(fltIn1, fltIn2, fltIn3);

    // output should be -1.8477294e-08
    fltOut = MLIB_Mnac(fltIn1, fltIn2, fltIn3, FLT);

    // ######
    // Available only if single precision floating point
    // implementation selected as default
    // #####
}

// output should be -1.8477294e-08
fltOut = MLIB_Mnac(fltIn1, fltIn2, fltIn3);
```

## 2.67 Function MLIB\_Msu

This function implements the multiply-subtract-from function.

## Description

This inline function returns the first input value from which the multiplication result of the second and third input values is subtracted.

## Re-entrancy

The function is re-entrant.

### 2.67.1 Function `MLIB_Msu_F32`

#### Declaration

```
INLINE tFrac32 MLIB_Msu_F32(register tFrac32 f32In1, register
tFrac32 f32In2, register tFrac32 f32In3);
```

#### Arguments

**Table 339. `MLIB_Msu_F32` arguments**

Type	Name	Direction	Description
register <code>tFrac32</code>	f32In1	input	Input value from which to subtract.
register <code>tFrac32</code>	f32In2	input	First value to be multiplied.
register <code>tFrac32</code>	f32In3	input	Second value to be multiplied.

#### Return

First input value from which the multiplication result of the second and third input values is subtracted.

#### Implementation details

The input values as well as output value is considered as 32-bit fractional values. The output saturation is not implemented in this function, thus in case the output value is outside the [-1, 1) interval, the output value will overflow.

The output of the function is defined by the following simple equation:

$$f32Out = f32In1 - (f32In2 \cdot f32In3)$$

Equation `MLIB_Msu_F32_Eq1`

**Note:** Due to effectivity reason this function is implemented as inline, and thus is not ANSI-C compliant.

#### Code Example

```
#include "mlib.h"

tFrac32 f32In1;
tFrac32 f32In2;
tFrac32 f32In3;
tFrac32 f32Out;

void main(void)
{
```

```

// input1 value = 0.25
f32In1 = FRAC32(0.25);

// input2 value = 0.5
f32In2 = FRAC32(0.5);

// input3 value = 0.125
f32In3 = FRAC32(0.125);

// output should be FRAC32(0.1875) = 0x18000000
f32Out = MLIB\_Msu\_F32(f32In1, f32In2, f32In3);

// output should be FRAC32(0.1875) = 0x18000000
f32Out = MLIB\_Msu(f32In1, f32In2, f32In3, F32);

// ##### Available only if 32-bit fractional implementation selected
// as default
// #####
// output should be FRAC32(0.1875) = 0x18000000
f32Out = MLIB\_Msu(f32In1, f32In2, f32In3);
}

```

## 2.67.2 Function [MLIB\\_Msu\\_F32F16F16](#)

### Declaration

```
INLINE tFrac32 MLIB\_Msu\_F32F16F16(register tFrac32 f32In1,  
register tFrac16 f16In2, register tFrac16 f16In3);
```

### Arguments

**Table 340. [MLIB\\_Msu\\_F32F16F16](#) arguments**

Type	Name	Direction	Description
register <a href="#">tFrac32</a>	f32In1	input	Input value from which to subtract.
register <a href="#">tFrac16</a>	f16In2	input	First value to be multiplied.
register <a href="#">tFrac16</a>	f16In3	input	Second value to be multiplied.

### Return

First input value from which the multiplication result of the second and third input values is subtracted.

### Implementation details

The first input value as well as output value is considered as 32-bit fractional values. The second and third input values are considered as 16-bit fractional values. The output saturation is not implemented in this function, thus in case the output value is outside the [-1, 1) interval, the output value will overflow.

The output of the function is defined by the following simple equation:

$$f32Out = f32In1 - (f16In2 \cdot f16In3)$$

Equation [MLIB\\_Msu\\_F32F16F16\\_Eq1](#)

This implementation is available if 32-bit fractional implementations are enabled. However it is not possible to use the default implementation based function call, thus the implementation post-fix or additional parameter function call shall be used.

**Note:** Due to effectivity reason this function is implemented as inline, and thus is not ANSI-C compliant.

### Code Example

```
#include "mlib.h"

tFrac32 f32In1;
tFrac16 f16In2;
tFrac16 f16In3;
tFrac32 f32Out;

void main(void)
{
    // input1 value = 0.25
    f32In1 = FRAC32(0.25);

    // input2 value = 0.5
    f16In2 = FRAC16(0.5);

    // input3 value = 0.125
    f16In3 = FRAC16(0.125);

    // output should be FRAC32(0.1875) = 0x18000000
    f32Out = MLIB_Msu_F32F16F16(f32In1, f16In2, f16In3);

    // output should be FRAC32(0.1875) = 0x18000000
    f32Out = MLIB_Msu(f32In1, f32In2, f32In3, F32F16F16);
}
```

### 2.67.3 Function MLIB\_Msu\_F16

#### Declaration

```
INLINE tFrac16 MLIB_Msu_F16(register tFrac16 f16In1, register
tFrac16 f16In2, register tFrac16 f16In3);
```

#### Arguments

Table 341. MLIB\_Msu\_F16 arguments

Type	Name	Direction	Description
register tFrac16	f16In1	input	Input value from which to subtract.
register tFrac16	f16In2	input	First value to be multiplied.
register tFrac16	f16In3	input	Second value to be multiplied.

#### Return

First input value from which the multiplication result of the second and third input values is subtracted.

### Implementation details

The input values as well as output value is considered as 16-bit fractional values. The output saturation is not implemented in this function, thus in case the output value is outside the [-1, 1] interval, the output value will overflow.

The output of the function is defined by the following simple equation:

$$f16Out = f16In1 - (f16In2 \cdot f16In3)$$

Equation MLIB\_Msu\_F16\_Eq1

**Note:** Due to effectivity reason this function is implemented as inline, and thus is not ANSI-C compliant.

### Code Example

```
#include "mlib.h"

tFrac16 f16In1;
tFrac16 f16In2;
tFrac16 f16In3;
tFrac16 f16Out;

void main(void)
{
    // input1 value = 0.25
    f16In1 = FRAC16(0.25);

    // input2 value = 0.5
    f16In2 = FRAC16(0.5);

    // input3 value = 0.125
    f16In3 = FRAC16(0.125);

    // output should be FRAC16(0.1875) = 0x1800
    f16Out = MLIB_Msu_F16(f16In1, f16In2, f16In3);

    // output should be FRAC16(0.1875) = 0x1800
    f16Out = MLIB_Msu(f16In1, f16In2, f16In3, F16);

    // ##### Available only if 16-bit fractional implementation selected
    // as default
    // #####
    // output should be FRAC16(0.1875) = 0x1800
    f16Out = MLIB_Msu(f16In1, f16In2, f16In3);
}
```

## 2.67.4 Function MLIB\_Msu\_FLT

### Declaration

```
INLINE tFloat MLIB_Msu_FLT(register tFloat fltIn1, register
    tFloat fltIn2, register tFloat fltIn3);
```

**Arguments****Table 342. MLIB\_Msu\_FLT arguments**

Type	Name	Direction	Description
register tFloat	fltIn1	input	Input value from which to subtract.
register tFloat	fltIn2	input	First value to be multiplied.
register tFloat	fltIn3	input	Second value to be multiplied.

**Return**

First input value from which the multiplication result of the second and third input values is subtracted.

**Implementation details**

The input values as well as output value are considered as single precision floating point data type. Intermediate results are computed in infinite precision.

The output of the function is defined by the following simple equation:

$$fltOut = fltIn1 - (fltIn2 \cdot fltIn3)$$

Equation MLIB\_Msu\_FLT\_Eq1

**Note:** The function may raise floating-point exceptions (invalid operation, overflow, underflow, inexact, input denormal). The floating-point unit must be enabled in CPACR register to prevent NOCP UsageFault exception.

Due to effectivity reason this function is implemented as inline assembly, and thus is not ANSI-C compliant.

**Code Example**

```
#include "mlib.h"

tFloat fltIn1;
tFloat fltIn2;
tFloat fltIn3;
tFloat fltOut;

void main(void)
{
    // input1 value = 1.150236353278160e-01
    fltIn1 = (tFloat)1.150236353278160e-01f;

    // input2 value = 9.057919383049011e-01
    fltIn2 = (tFloat)9.057919383049011e-01f;

    // input3 value = 1.269868165254593e-01
    fltIn3 = (tFloat)1.269868165254593e-01f;

    // output should be 6.4805139e-10
    fltOut = MLIB_Msu_FLT(fltIn1, fltIn2, fltIn3);

    // output should be 6.4805139e-10
    fltOut = MLIB_Msu(fltIn1, fltIn2, fltIn3, FLT);
```

```

// ######
// Available only if single precision floating point
// implementation selected as default
// #####
// output should be 6.4805139e-10
fltOut = MLIB_Msu(fltIn1, fltIn2, fltIn3);
}

```

## 2.68 Function `MLIB_Mul`

This function multiplies two input parameters.

### Description

This inline function multiplies the two input values.

### Re-entrancy

The function is re-entrant.

### 2.68.1 Function `MLIB_Mul_F32`

#### Declaration

```
INLINE tFrac32 MLIB_Mul_F32(register tFrac32 f32In1, register tFrac32 f32In2);
```

#### Arguments

**Table 343. `MLIB_Mul_F32` arguments**

Type	Name	Direction	Description
register <a href="#">tFrac32</a>	f32In1	input	Operand is a 32-bit number normalized between [-1,1).
register <a href="#">tFrac32</a>	f32In2	input	Operand is a 32-bit number normalized between [-1,1).

#### Return

Fractional multiplication of the input arguments.

#### Implementation details

The input values as well as output value is considered as 32-bit fractional values. The output saturation is not implemented in this function, thus in case the multiplication of input values is outside the [-1, 1] interval, the output value will overflow.

The output of the function is defined by the following simple equation:

$$f32Out = f32In1 \cdot f32In2$$

Equation `MLIB_Mul_F32_Eq1`

**Note:** Overflow is not detected. Due to effectivity reason this function is implemented as inline assembly and thus is not ANSI-C compliant.

**Code Example**

```
#include "mlib.h"

tFrac32 f32In1;
tFrac32 f32In2;
tFrac32 f32Out;

void main(void)
{
    // first input = 0.5
    f32In1 = FRAC32(0.5);

    // second input = 0.25
    f32In2 = FRAC32(0.25);

    // output should be 0x10000000 = FRAC32(0.125)
    f32Out = MLIB_Mul_F32(f32In1,f32In2);

    // output should be 0x10000000 = FRAC32(0.125)
    f32Out = MLIB_Mul(f32In1,f32In2,F32);

    // ######
    // Available only if 32-bit fractional implementation selected
    // as default
    // #####
    // output should be 0x10000000 = FRAC32(0.125)
    f32Out = MLIB_Mul(f32In1,f32In2);
}
```

**2.68.2 Function MLIB\_Mul\_F32F16F16****Declaration**

```
INLINE tFrac32 MLIB_Mul_F32F16F16(register tFrac16 f16In1,
register tFrac16 f16In2);
```

**Arguments****Table 344.** MLIB\_Mul\_F32F16F16 arguments

Type	Name	Direction	Description
register tFrac16	f16In1	input	Operand is a 16-bit number normalized between [-1,1).
register tFrac16	f16In2	input	Operand is a 16-bit number normalized between [-1,1).

**Return**

Fractional multiplication of the input arguments.

**Implementation details**

The input values are considered as 16-bit fractional values and the output value is considered as 32-bit fractional value. The output saturation is not implemented in this function, thus in case the multiplication of input values is outside the [-1, 1) interval, the output value will overflow.

The output of the function is defined by the following simple equation:

$$f32Out = f16In1 \cdot f16In2$$

Equation MLIB\_Mul\_F32F16F16\_Eq1

**Note:** Overflow is not detected. Due to effectivity reason this function is implemented as inline assembly and thus is not ANSI-C compliant.

### Code Example

```
#include "mlib.h"

tFrac16 f16In1;
tFrac16 f16In2;
tFrac32 f32Out;

void main(void)
{
    // first input = 0.5
    f16In1 = FRAC16(0.5);

    // second input = 0.25
    f16In2 = FRAC16(0.25);

    // output should be 0x10000000 = FRAC32(0.125)
    f32Out = MLIB_Mul_F32F16F16(f16In1, f16In2);

    // output should be 0x10000000 = FRAC32(0.125)
    f32Out = MLIB_Mul(f16In1, f16In2, F32F16F16);
}
```

### 2.68.3 Function MLIB\_Mul\_F16

#### Declaration

```
INLINE tFrac16 MLIB_Mul_F16(register tFrac16 f16In1, register
tFrac16 f16In2);
```

#### Arguments

Table 345. MLIB\_Mul\_F16 arguments

Type	Name	Direction	Description
register tFrac16	f16In1	input	Operand is a 16-bit number normalized between [-1,1).
register tFrac16	f16In2	input	Operand is a 16-bit number normalized between [-1,1).

#### Return

Fractional multiplication of the input arguments.

#### Implementation details

The input values as well as output value is considered as 16-bit fractional values. The output saturation is not implemented in this function, thus in case the multiplication of input values is outside the [-1, 1) interval, the output value will overflow.

The output of the function is defined by the following simple equation:

$$f16Out = f16In1 \cdot f16In2$$

Equation MLIB\_Mul\_F16\_Eq1

**Note:** Due to effectivity reason this function is implemented as inline and thus is not ANSI-C compliant.

### Code Example

```
#include "mlib.h"

tFrac16 f16In1;
tFrac16 f16In2;
tFrac16 f16Out;

void main(void)
{
    // first input = 0.5
    f16In1 = FRAC16(0.5);

    // second input = 0.25
    f16In2 = FRAC16(0.25);

    // output should be 0x1000 = FRAC16(0.125)
    f16Out = MLIB_Mul_F16(f16In1, f16In2);

    // output should be 0x1000 = FRAC16(0.125)
    f16Out = MLIB_Mul(f16In1, f16In2, F16);

    // ##### Available only if 16-bit fractional implementation selected
    // as default
    // #####
    // output should be 0x1000 = FRAC16(0.125)
    f16Out = MLIB_Mul(f16In1, f16In2);
}
```

#### 2.68.4 Function MLIB\_Mul\_FLT

##### Declaration

```
INLINE tFloat MLIB_Mul_FLT(register tFloat fltIn1, register
tFloat fltIn2);
```

##### Arguments

Table 346. MLIB\_Mul\_FLT arguments

Type	Name	Direction	Description
register tFloat	fltIn1	input	Operand is a single precision floating point number.
register tFloat	fltIn2	input	Operand is a single precision floating point number.

### Return

Floating point multiplication of the input arguments.

### Implementation details

The input values as well as output value is considered as single precision floating point data type.

The output of the function is defined by the following simple equation:

$$fltOut = fltIn1 \cdot fltIn2$$

Equation MLIB\_Mul\_FLT\_Eq1

**Note:** The function may raise floating-point exceptions (invalid operation, overflow, underflow, inexact, input denormal). The floating-point unit must be enabled in CPACR register to prevent NOCP UsageFault exception.

Due to effectivity reason this function is implemented as inline and thus is not ANSI-C compliant.

### Code Example

```
#include "mlib.h"

tFloat fltIn1;
tFloat fltIn2;
tFloat fltOut;

void main(void)
{
    // first input = 50.5
    fltIn1 = (tFloat)50.5;

    // second input = 25.25
    fltIn2 = (tFloat)25.25;

    // output should be 1275.125
    fltOut = MLIB_Mul_FLT(fltIn1, fltIn2);

    // output should be 1275.125
    fltOut = MLIB_Mul(fltIn1, fltIn2, FLT);

    // ######
    // Available only if single precision floating point
    // implementation selected as default
    // #####
    // output should be 1275.125
    fltOut = MLIB_Mul(fltIn1, fltIn2);
}
```

## 2.69 Function MLIB\_MulSat

This function multiplies two input parameters and saturate if necessary.

## Description

This inline function multiplies the two input values and saturates the result.

## Re-entrancy

The function is re-entrant.

### 2.69.1 Function `MLIB_MulSat_F32`

#### Declaration

```
INLINE tFrac32 MLIB_MulSat_F32(register tFrac32 f32In1, register tFrac32 f32In2);
```

#### Arguments

**Table 347. `MLIB_MulSat_F32` arguments**

Type	Name	Direction	Description
register <a href="#">tFrac32</a>	f32In1	input	Operand is a 32-bit number normalized between [-1,1].
register <a href="#">tFrac32</a>	f32In2	input	Operand is a 32-bit number normalized between [-1,1].

#### Return

Fractional multiplication of the input arguments.

#### Implementation details

The input values as well as output value are considered as 32-bit fractional data type.

The output of the function is defined by the following simple equation:

$$f32Out = \begin{cases} FRAC32\_MIN & \text{if } (f32In1 \cdot f32In2) < FRAC32\_MIN \\ f32In1 \cdot f32In2 & \text{if } FRAC32\_MIN \leq (f32In1 \cdot f32In2) \leq FRAC32\_MAX \\ FRAC32\_MAX & \text{if } (f32In1 \cdot f32In2) > FRAC32\_MAX \end{cases}$$

Equation `MLIB_MulSat_F32_Eq1`

**Note:** Due to effectivity reason this function is implemented as inline assembly and thus is not ANSI-C compliant.

#### Code Example

```
#include "mlib.h"

tFrac32 f32In1;
tFrac32 f32In2;
tFrac32 f32Out;

void main(void)
{
    // first input = 0.8
    f32In1 = FRAC32(0.8);

    // second input = 0.75
}
```

```

f32In2 = FRAC32(0.75);

// output should be 0x4ccccccc = FRAC32(0.6)
f32Out = MLIB\_MulSat\_F32(f32In1,f32In2);

// output should be 0x4ccccccc = FRAC32(0.6)
f32Out = MLIB\_MulSat(f32In1,f32In2,F32);

// ######
// Available only if 32-bit fractional implementation selected
// as default
// #####
// output should be 0x4ccccccc = FRAC32(0.6)
f32Out = MLIB\_MulSat(f32In1,f32In2);
}

```

## 2.69.2 Function [MLIB\\_MulSat\\_F32F16F16](#)

### Declaration

```
INLINE tFrac32 MLIB\_MulSat\_F32F16F16(register tFrac16 f16In1,  
register tFrac16 f16In2);
```

### Arguments

**Table 348. [MLIB\\_MulSat\\_F32F16F16](#) arguments**

Type	Name	Direction	Description
register <a href="#">tFrac16</a>	f16In1	input	Operand is a 16-bit number normalized between [-1,1].
register <a href="#">tFrac16</a>	f16In2	input	Operand is a 16-bit number normalized between [-1,1].

### Return

Fractional multiplication of the input arguments.

### Implementation details

The input values are considered as 16-bit fractional data type and the output value is considered as 32-bit fractional data type.

The output of the function is defined by the following simple equation:

$$f32Out = \begin{cases} FRAC32\_MIN & \text{if } (f16In1 \cdot f16In2) < FRAC32\_MIN \\ f16In1 \cdot f16In2 & \text{if } FRAC32\_MIN \leq (f16In1 \cdot f16In2) \leq FRAC32\_MAX \\ FRAC32\_MAX & \text{if } (f16In1 \cdot f16In2) > FRAC32\_MAX \end{cases}$$

Equation [MLIB\\_MulSat\\_F32F16F16\\_Eq1](#)

**Note:** Due to effectivity reason this function is implemented as inline assembly and thus is not ANSI-C compliant.

### Code Example

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

```

tFrac16 f16In1;
tFrac16 f16In2;
tFrac32 f32Out;

void main(void)
{
    // first input = 0.8
    f16In1 = FRAC16(0.8);

    // second input = 0.75
    f16In2 = FRAC16(0.75);

    // output should be 0x4ccccccc = FRAC32(0.6)
    f32Out = MLIB_MulSat_F32F16F16(f16In1,f16In2);

    // output should be 0x4ccccccc = FRAC32(0.6)
    f32Out = MLIB_MulSat(f32In1,f32In2,F32F16f16);
}

```

### 2.69.3 Function **MLIB\_MulSat\_F16**

#### Declaration

```
INLINE tFrac16 MLIB_MulSat_F16(register tFrac16 f16In1, register tFrac16 f16In2);
```

#### Arguments

**Table 349. MLIB\_MulSat\_F16 arguments**

Type	Name	Direction	Description
register tFrac16	f16In1	input	Operand is a 16-bit number normalized between [-1,1).
register tFrac16	f16In2	input	Operand is a 16-bit number normalized between [-1,1).

#### Return

Fractional multiplication of the input arguments.

#### Implementation details

The input values as well as output value are considered as 16-bit fractional data type.

The output of the function is defined by the following simple equation:

$$f16Out = \begin{cases} FRAC16\_MIN & \text{if } (f16In1 \cdot f16In2) < FRAC16\_MIN \\ f16In1 \cdot f16In2 & \text{if } FRAC16\_MIN \leq (f16In1 \cdot f16In2) \leq FRAC16\_MAX \\ FRAC16\_MAX & \text{if } (f16In1 \cdot f16In2) > FRAC16\_MAX \end{cases}$$

Equation MLIB\_MulSat\_F16\_Eq1

**Note:** Due to effectivity reason this function is implemented as inline and thus is not ANSI-C compliant.

#### Code Example

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

```

tFrac16 f16In1;
tFrac16 f16In2;
tFrac16 f16Out;

void main(void)
{
    // first input = 0.8
    f16In1 = FRAC16(0.8);

    // second input = 0.75
    f16In2 = FRAC16(0.75);

    // output should be 0x4ccc = FRAC16(0.6)
    f16Out = MLIB_MulSat_F16(f16In1,f16In2);

    // output should be 0x4ccc = FRAC16(0.6)
    f16Out = MLIB_MulSat(f16In1,f16In2,F16);

    // ##### Available only if 16-bit fractional implementation selected
    // as default
    // #####
    // output should be 0x4ccc = FRAC32(0.6)
    f16Out = MLIB_MulSat(f16In1,f16In2);
}

```

## 2.70 Function MLIB\_Neg

This function returns negative value of input parameter.

### Description

This inline function returns the negative value of input parameter.

### Re-entrancy

The function is re-entrant.

#### 2.70.1 Function MLIB\_Neg\_F32

##### Declaration

```
INLINE tFrac32 MLIB_Neg_F32(register tFrac32 f32In);
```

##### Arguments

**Table 350. MLIB\_Neg\_F32 arguments**

Type	Name	Direction	Description
register tFrac32	f32In	input	Input value which negative value should be returned.

##### Return

Negative value of input parameter.

**Implementation details**

The input values as well as output value is considered as 32-bit fractional values. The output saturation is not implemented in this function, thus in case the negation of input values is outside the [-1, 1) interval, the output value will overflow.

The output of the function is defined by the following simple equation:

$$f32Out = -f32In$$

Equation MLIB\_Neg\_F32\_Eq1

**Note:** Due to effectivity reason this function is implemented as inline, and thus is not ANSI-C compliant.

**Code Example**

```
#include "mlib.h"

tFrac32 f32In;
tFrac32 f32Out;

void main(void)
{
    // input value = 0.25
    f32In = FRAC32(0.25);

    // output should be FRAC32(-0.25) = 0xA0000000
    f32Out = MLIB_Neg_F32(f32In);

    // output should be FRAC32(-0.25) = 0xA0000000
    f32Out = MLIB_Neg(f32In, F32);

    // ##### Available only if 32-bit fractional implementation selected
    // as default
    // #####
    // output should be FRAC32(-0.25) = 0xA0000000
    f32Out = MLIB_Neg(f32In);
}
```

**2.70.2 Function MLIB\_Neg\_F16****Declaration**

```
INLINE tFrac16 MLIB_Neg_F16(register tFrac16 f16In);
```

**Arguments****Table 351. MLIB\_Neg\_F16 arguments**

Type	Name	Direction	Description
register tFrac16	f16In	input	Input value which negative value should be returned.

## Return

Negative value of input parameter.

## Implementation details

The input values as well as output value is considered as 16-bit fractional values. The output saturation is not implemented in this function, thus in case the negation of input values is outside the [-1, 1) interval, the output value will overflow.

The output of the function is defined by the following simple equation:

$$f16Out = -f16In$$

Equation MLIB\_Neg\_F16\_Eq1

**Note:** Due to effectivity reason this function is implemented as inline, and thus is not ANSI-C compliant.

## Code Example

```
#include "mlib.h"

tFrac16 f16In;
tFrac16 f16Out;

void main(void)
{
    // input value = 0.25
    f16In = FRAC16(0.25);

    // output should be FRAC16(-0.25) = 0xA000
    f16Out = MLIB_Neg_F16(f16In);

    // output should be FRAC16(-0.25) = 0xA000
    f16Out = MLIB_Neg(f16In, F16);

    // ##### Available only if 16-bit fractional implementation selected
    // as default
    // #####
    // output should be FRAC16(-0.25) = 0xA000
    f16Out = MLIB_Neg(f16In);
}
```

### 2.70.3 Function MLIB\_Neg\_FLT

#### Declaration

```
INLINE tFloat MLIB_Neg_FLT(register tFloat fltIn);
```

## Arguments

**Table 352. MLIB\_NegFLT arguments**

Type	Name	Direction	Description
register tFloat	fltIn	input	Input value which negative value should be returned.

## Return

Negative value of input parameter.

## Implementation details

The input values as well as output value is considered as single precision floating point data type.

The output of the function is defined by the following simple equation:

$$fltOut = - fltIn$$

Equation MLIB\_NegFLT\_Eq1

**Note:** Due to effectivity reason this function is implemented as inline, and thus is not ANSI-C compliant.

## Code Example

```
#include "mlib.h"

tFloat fltIn;
tFloat fltOut;

void main(void)
{
    // input value = 0.25
    fltIn = (tFloat)0.25;

    // output should be (-0.25)
    fltOut = MLIB_NegFLT(fltIn);

    // output should be (-0.25)
    fltOut = MLIB_Neg(fltIn, FLT);

    // ##### Available only if single precision floating point
    // implementation selected as default
    // #####
    // output should be (-0.25)
    fltOut = MLIB_Neg(fltIn);
}
```

## 2.71 Function MLIB\_NegSat

This function returns negative value of input parameter and saturate if necessary.

## Description

This inline function returns the negative value of input parameter and saturates the result if necessary.

## Re-entrancy

The function is re-entrant.

### 2.71.1 Function `MLIB_NegSat_F32`

#### Declaration

```
INLINE tFrac32 MLIB_NegSat_F32(register tFrac32 f32In);
```

#### Arguments

**Table 353. `MLIB_NegSat_F32` arguments**

Type	Name	Direction	Description
register <a href="#">tFrac32</a>	f32In	input	Input value which negative value should be returned.

#### Return

Negative value of input parameter.

#### Implementation details

The input values as well as output value is considered as 32-bit fractional data type.

The output of the function is defined by the following simple equation:

$$f32Out = \begin{cases} FRAC32\_MIN & \text{if } (-f32In) < FRAC32\_MIN \\ -f32In & \text{if } FRAC32\_MIN \leq (-f32In) \leq FRAC32\_MAX \\ FRAC32\_MAX & \text{if } (-f32In) > FRAC32\_MAX \end{cases}$$

Equation `MLIB_NegSat_F32_Eq1`

**Note:** Due to effectivity reason this function is implemented as inline assembly, and thus is not ANSI-C compliant.

#### Code Example

```
#include "mlib.h"

tFrac32 f32In;
tFrac32 f32Out;

void main(void)
{
    // input value = 0.25
    f32In = FRAC32(0.25);

    // output should be FRAC32(-0.25) = 0xA0000000
    f32Out = MLIB_NegSat_F32(f32In);

    // output should be FRAC32(-0.25) = 0xA0000000
```

```
f32Out = MLIB_NegSat(f32In, F32);

// ##### Available only if 32-bit fractional implementation selected
// as default
// #####
// output should be FRAC32(-0.25) = 0xA0000000
f32Out = MLIB_NegSat(f32In);
}
```

## 2.71.2 Function `MLIB_NegSat_F16`

### Declaration

```
INLINE tFrac16 MLIB_NegSat_F16(register tFrac16 f16In);
```

### Arguments

**Table 354. `MLIB_NegSat_F16` arguments**

Type	Name	Direction	Description
register <code>tFrac16</code>	f16In	input	Input value which negative value should be returned.

### Return

Negative value of input parameter.

### Implementation details

The input values as well as output value is considered as 16-bit fractional data type.

The output of the function is defined by the following simple equation:

$$f16Out = \begin{cases} FRAC16\_MIN & \text{if } (-f16In) < FRAC16\_MIN \\ -f16In & \text{if } FRAC16\_MIN \leq (-f16In) \leq FRAC16\_MAX \\ FRAC16\_MAX & \text{if } (-f16In) > FRAC16\_MAX \end{cases}$$

Equation `MLIB_NegSat_F16_Eq1`

**Note:** Due to effectivity reason this function is implemented as inline assembly, and thus is not ANSI-C compliant.

### Code Example

```
#include "mlib.h"

tFrac16 f16In;
tFrac16 f16Out;

void main(void)
{
    // input value = 0.25
    f16In = FRAC16(0.25);

    // output should be FRAC16(-0.25) = 0xA000
    f16Out = MLIB_NegSat_F16(f16In);
```

```

// output should be FRAC16(-0.25) = 0xA000
f16Out = MLIB_NegSat(f16In, F16);

// ##### Available only if 16-bit fractional implementation selected
// as default
// #####
// output should be FRAC16(-0.25) = 0xA000
f16Out = MLIB_NegSat(f16In);
}

```

## 2.72 Function `MLIB_Norm`

This function returns the number of left shifts needed to normalize the input parameter.

### Description

The function returns the number of left shifts needed to remove redundant leading sign bits.

### Re-entrancy

The function is re-entrant.

#### 2.72.1 Function `MLIB_Norm_F32`

##### Declaration

```
INLINE tU16 MLIB_Norm_F32(register tFrac32 f32In);
```

##### Arguments

**Table 355. `MLIB_Norm_F32` arguments**

Type	Name	Direction	Description
register <code>tFrac32</code>	f32In	input	The first value to be normalized.

##### Return

The number of left shift needed to normalize the argument.

**Note:** Due to effectivity reason this function is implemented as inline and thus is not ANSI-C compliant.

##### Code Example

```

#include "mlib.h"

tFrac32 f32In;
tU16 u16Out;

void main(void)
{
    // first input = 0.00005
}

```

```

f32In = FRAC32(0.00005);

// output should be 14
u16Out = MLIB\_Norm\_F32(f32In);

// output should be 14
u16Out = MLIB\_Norm(f32In, F32);

// ######
// Available only if 32-bit fractional implementation selected
// as default
// #####
// output should be 14
u16Out = MLIB\_Norm(f32In);
}

```

## 2.72.2 Function [MLIB\\_Norm\\_F16](#)

### Declaration

```
INLINE tU16 MLIB\_Norm\_F16(register tFrac16 f16In);
```

### Arguments

**Table 356. [MLIB\\_Norm\\_F16](#) arguments**

Type	Name	Direction	Description
register <a href="#">tFrac16</a>	f16In	input	The first value to be normalized.

### Return

The number of left shift needed to normalize the argument.

**Note:** Due to effectivity reason this function is implemented as inline and thus is not ANSI-C compliant.

### Code Example

```

#include "mlib.h"

tFrac16 f16In;
tU16 u16Out;

void main(void)
{
    // first input = 0.00005
    f16In = FRAC16(0.00005);

    // output should be 14
    u16Out = MLIB\_Norm\_F16(f16In);

    // output should be 14
    u16Out = MLIB\_Norm(f16In, F16);

    // #####
    // Available only if 16-bit fractional implementation selected
    // as default
}

```

```
// ######
// output should be 14
u16Out = MLIB_Norm(f16In);
}
```

## 2.73 Function MLIB\_RndSat\_F16F32

This function rounds the input value to the nearest saturated value in the output format.

### Declaration

```
INLINE tFrac16 MLIB_RndSat_F16F32(register tFrac32 f32In);
```

### Arguments

**Table 357. MLIB\_RndSat\_F16F32 arguments**

Type	Name	Direction	Description
register <a href="#">tFrac32</a>	f32In	input	Input value.

### Return

Rounded saturated value.

### Description

This inline function rounds the input value to the nearest saturated value in the output format. The input value is considered as 32-bit fractional data type and output value is considered as 16-bit fractional data type.

### Re-entrancy

The function is re-entrant.

### Code Example

```
#include "mlib.h"

tFrac32 f32In;
tFrac16 f16Out;

void main(void)
{
    // input value = 0.25 = 0x2000 0000
    f32In = FRAC32(0.25);

    // output should be FRAC16(0.25) = 0x2000
    f16Out = MLIB_RndSat_F16F32(f32In);
}
```

## 2.74 Function MLIB\_Round

The function rounds the input and saturates.

### Description

The function rounds the first input argument to the nearest value (round half up). The number of trailing zeros in the rounded result is equal to the second input argument. The result is saturated to the fractional range.

### Re-entrancy

The function is re-entrant.

#### 2.74.1 Function MLIB\_Round\_F32

##### Declaration

```
INLINE tFrac32 MLIB_Round_F32(register tFrac32 f32In1, register
tU16 u16In2);
```

##### Arguments

**Table 358. MLIB\_Round\_F32 arguments**

Type	Name	Direction	Description
register <a href="#">tFrac32</a>	f32In1	input	The value to be rounded.
register <a href="#">tU16</a>	u16In2	input	The number of trailing zeros in the rounded result.

##### Return

Rounded 32-bit fractional value.

**Note:** The second input argument must not exceed 30 for positive and 31 for negative f32In1, respectively, otherwise the result is undefined. Due to effectivity reason this function is implemented as inline and thus is not ANSI-C compliant.

##### Code Example

```
#include "mlib.h"

tFrac32 f32In1;
tFrac32 f32Out;
tU16 u16In2;

void main(void)
{
    // Example no. 1
    // first input = 0.25
    f32In1 = FRAC32(0.25);
    // second input = 29
    u16In2 = (tU16)29;

    // output should be 0x20000000 ~ FRAC32(0.25)
    f32Out = MLIB_Round_F32(f32In1,u16In2);
```

```

// output should be 0x20000000 ~ FRAC32(0.25)
f32Out = MLIB_Round(f32In1,u16In2,F32);

// ##### Available only if 32-bit fractional implementation selected
// as default
// #####
// output should be 0x20000000 ~ FRAC32(0.25)
f32Out = MLIB_Round(f32In1,u16In2);

// Example no. 2
// first input = 0.375
f32In1 = FRAC32(0.375);
// second input = 29
u16In2 = (tU16)29;

// output should be 0x40000000 ~ FRAC32(0.5)
f32Out = MLIB_Round_F32(f32In1,u16In2);

// Example no. 3
// first input = -0.375
f32In1 = FRAC32(-0.375);
// second input = 29
u16In2 = (tU16)29;

// output should be 0xE0000000 ~ FRAC32(-0.25)
f32Out = MLIB_Round_F32(f32In1,u16In2);
}

```

## 2.74.2 Function **MLIB\_Round\_F16**

### Declaration

```
INLINE tFrac16 MLIB_Round_F16(register tFrac16 f16In1, register
tU16 u16In2);
```

### Arguments

**Table 359. MLIB\_Round\_F16 arguments**

Type	Name	Direction	Description
register <b>tFrac16</b>	f16In1	input	The value to be rounded.
register <b>tU16</b>	u16In2	input	The number of trailing zeros in the rounded result.

### Return

Rounded 16-bit fractional value.

**Note:** The second input argument must not exceed 14 for positive and 15 for negative f16In1, respectively, otherwise the result is undefined. Due to effectivity reason this function is implemented as inline and thus is not ANSI-C compliant.

### Code Example

```
#include "mlib.h"

_tFrac16 f16In1;
_tFrac16 f16Out;
_tU16 u16In2;

void main(void)
{
    // Example no. 1
    // first input = 0.25
    f16In1 = FRAC16(0.25);
    // second input = 13
    u16In2 = (tU16)13;

    // output should be 0x2000 ~ FRAC16(0.25)
    f16Out = MLIB_Round_F16(f16In1,u16In2);

    // output should be 0x2000 ~ FRAC16(0.25)
    f16Out = MLIB_Round(f16In1,u16In2,F16);

    // ##### Available only if 16-bit fractional implementation selected
    // as default
    // #####
    // output should be 0x2000 ~ FRAC16(0.25)
    f16Out = MLIB_Round(f16In1,u16In2);

    // Example no. 2
    // first input = 0.375
    f16In1 = FRAC16(0.375);
    // second input = 13
    u16In2 = (tU16)13;

    // output should be 0x4000 ~ FRAC16(0.5)
    f16Out = MLIB_Round_F16(f16In1,u16In2);

    // Example no. 3
    // first input = -0.375
    f16In1 = FRAC16(-0.375);
    // second input = 13
    u16In2 = (tU16)13;

    // output should be 0xE000 ~ FRAC16(-0.25)
    f16Out = MLIB_Round_F16(f16In1,u16In2);
}
```

## 2.75 Function MLIB\_ShBi

This function shifts the first argument to left or right by number defined by second argument.

## Description

This function shifts the first parameter by the amount specified in the second parameter. Positive values of the second argument correspond to the left shift, negative values to the right shift. The result of the left shift may overflow.

## Re-entrancy

The function is re-entrant.

### 2.75.1 Function MLIB\_ShBi\_F32

#### Declaration

```
INLINE tFrac32 MLIB_ShBi_F32(register tFrac32 f32In1, register
ts16 s16In2);
```

#### Arguments

**Table 360. MLIB\_ShBi\_F32 arguments**

Type	Name	Direction	Description
register <a href="#">tFrac32</a>	f32In1	<a href="#">input</a>	First value to be shift.
register <a href="#">ts16</a>	s16In2	<a href="#">input</a>	The shift amount value.

#### Return

32-bit fractional value shifted to left or right by the shift amount. The bits beyond the 32-bit boundary are discarded.

**Note:** The shift amount cannot exceed in magnitude the bit-width of the shift value, that means must be within the range -31...31. Otherwise the result of the function is undefined. Due to effectivity reason this function is implemented as inline, and thus is not ANSI-C compliant.

#### Code Example

```
#include "mlib.h"

tFrac32 f32In1;
tFrac32 f32Out;
ts16 s16In2;

void main(void)
{
    // first input = 0.25
    f32In1 = FRAC32(0.25);
    // second input = -1
    s16In2 = (ts16)-1;

    // output should be 0x10000000 ~ FRAC32(0.125)
    f32Out = MLIB_ShBi_F32(f32In1, s16In2);

    // output should be 0x10000000 ~ FRAC32(0.125)
    f32Out = MLIB_ShBi(f32In1, s16In2, F32);
```

```

// ######
// Available only if 32-bit fractional implementation selected
// as default
// #####
// output should be 0x10000000 ~ FRAC32(0.125)
f32Out = MLIB_ShBi(f32In1, s16In2);
}

```

## 2.75.2 Function `MLIB_ShBi_F16`

### Declaration

```
INLINE tFrac16 MLIB_ShBi_F16(register tFrac16 f16In1, register
tS16 s16In2);
```

### Arguments

**Table 361. `MLIB_ShBi_F16` arguments**

Type	Name	Direction	Description
register <code>tFrac16</code>	f16In1	input	First value to be left shift.
register <code>tS16</code>	s16In2	input	The shift amount value.

### Return

16-bit fractional value shifted to left or right by the shift amount. The bits beyond the 16-bit boundary are discarded.

**Note:** The shift amount cannot exceed in magnitude the bit-width of the shift value, that means must be within the range -15...15. Otherwise the result of the function is undefined. Due to effectivity reason this function is implemented as inline, and thus is not ANSI-C compliant.

### Code Example

```

#include "mlib.h"

tFrac16 f16In1;
tFrac16 f16Out;
tS16 s16In2;

void main(void)
{
    // first input = 0.25
    f16In1 = FRAC16(0.25);
    // second input = -1
    s16In2 = (tS16)-1;

    // output should be 0x1000 ~ FRAC16(0.125)
    f16Out = MLIB_ShBi_F16(f16In1, s16In2);

    // output should be 0x1000 ~ FRAC16(0.125)
    f16Out = MLIB_ShBi(f16In1, s16In2, F16);

    // #####
    // Available only if 16-bit fractional implementation selected
}

```

```
// as default
// #####
// output should be 0x1000 ~ FRAC16(0.125)
f16Out = MLIB_ShBi(f16In1, s16In2);
}
```

## 2.76 Function MLIB\_ShBiSat

This function shifts the first argument to left or right by number defined by second argument and saturate if necessary.

### Description

This function shifts the first parameter by the amount specified in the second parameter. Positive values of the second argument correspond to the left shift, negative values to the right shift. The result of the left shift is saturated if necessary.

### Re-entrancy

The function is re-entrant.

#### 2.76.1 Function MLIB\_ShBiSat\_F32

##### Declaration

```
INLINE tFrac32 MLIB_ShBiSat_F32(register tFrac32 f32In1, register
tS16 s16In2);
```

##### Arguments

**Table 362. MLIB\_ShBiSat\_F32 arguments**

Type	Name	Direction	Description
register <a href="#">tFrac32</a>	f32In1	input	First value to be shift.
register <a href="#">tS16</a>	s16In2	input	The shift amount value.

##### Return

32-bit fractional value shifted to left or right by the shift amount. The bits beyond the 32-bit boundary are discarded.

**Note:** The shift amount cannot exceed in magnitude the bit-width of the shift value, that means must be within the range -31...31. Otherwise the result of the function is undefined. Due to effectivity reason this function is implemented as inline, and thus is not ANSI-C compliant.

##### Code Example

```
#include "mlib.h"

tFrac32 f32In1;
tFrac32 f32Out;
tS16 s16In2;
```

```

void main(void)
{
    // first input = 0.25
    f32In1 = FRAC32(0.25);
    // second input = -1
    s16In2 = (tS16) -1;

    // output should be 0x10000000 ~ FRAC32(0.125)
    f32Out = MLIB_ShBiSat_F32(f32In1, s16In2);

    // output should be 0x10000000 ~ FRAC32(0.125)
    f32Out = MLIB_ShBiSat(f32In1, s16In2, F32);

    // ##### Available only if 32-bit fractional implementation selected
    // as default
    // #####
    // output should be 0x10000000 ~ FRAC32(0.125)
    f32Out = MLIB_ShBiSat(f32In1, s16In2);
}

```

## 2.76.2 Function MLIB\_ShBiSat\_F16

### Declaration

```
INLINE tFrac16 MLIB_ShBiSat_F16(register tFrac16 f16In1, register
tS16 s16In2);
```

### Arguments

**Table 363. MLIB\_ShBiSat\_F16 arguments**

Type	Name	Direction	Description
register <b>tFrac16</b>	f16In1	input	First value to be left shift.
register <b>tS16</b>	s16In2	input	The shift amount value.

### Return

16-bit fractional value shifted to left or right by the shift amount. The bits beyond the 16-bit boundary are discarded.

**Note:** The shift amount cannot exceed in magnitude the bit-width of the shift value, that means must be within the range -15...15. Otherwise the result of the function is undefined. Due to effectivity reason this function is implemented as inline, and thus is not ANSI-C compliant.

### Code Example

```

#include "mlib.h"

tFrac16 f16In1;
tFrac16 f16Out;
tS16 s16In2;

void main(void)
{

```

```

// first input = 0.25
f16In1 = FRAC16(0.25);
// second input = -1
s16In2 = (ts16)-1;

// output should be 0x1000 ~ FRAC16(0.125)
f16Out = MLIB_ShBiSat_F16(f16In1, s16In2);

// output should be 0x1000 ~ FRAC16(0.125)
f16Out = MLIB_ShBiSat(f16In1, s16In2, F16);

// ##### Available only if 16-bit fractional implementation selected
// as default
// #####
// output should be 0x1000 ~ FRAC16(0.125)
f16Out = MLIB_ShBiSat(f16In1, s16In2);
}

```

## 2.77 Function MLIB\_ShL

This function shifts the first parameter to left by number defined by second parameter.

### Description

This function shifts the first argument to the left by the amount specified in the second argument. Overflow is not detected.

### Re-entrancy

The function is re-entrant.

#### 2.77.1 Function MLIB\_ShL\_F32

##### Declaration

```
INLINE tFrac32 MLIB_ShL_F32(register tFrac32 f32In1, register
tU16 u16In2);
```

##### Arguments

**Table 364. MLIB\_ShL\_F32 arguments**

Type	Name	Direction	Description
register tFrac32	f32In1	input	First value to be left shift.
register tU16	u16In2	input	The shift amount value.

##### Return

32-bit fractional value shifted to left by the shift amount. The bits beyond the 32-bit boundary are discarded.

**Note:** The shift amount cannot exceed in magnitude the bit-width of the shift value, that means must be within the range 0...31. Otherwise the result of the function is undefined. Due to effectivity reason this function is implemented as inline, and thus is not ANSI-C compliant.

**Code Example**

```
#include "mlib.h"

tFrac32 f32In1;
tFrac32 f32Out;
tU16 u16In2;

void main(void)
{
    // first input = 0.25
    f32In1 = FRAC32(0.25);
    // second input = 1
    u16In2 = (tU16)1;

    // output should be 0x40000000 ~ FRAC32(0.5)
    f32Out = MLIB_ShL_F32(f32In1, u16In2);

    // output should be 0x40000000 ~ FRAC32(0.5)
    f32Out = MLIB_ShL(f32In1, u16In2, F32);

    // ##### Available only if 32-bit fractional implementation selected
    // as default
    // #####
    // output should be 0x40000000 ~ FRAC32(0.5)
    f32Out = MLIB_ShL(f32In1, u16In2);
}
```

**2.77.2 Function MLIB\_ShL\_F16****Declaration**

```
INLINE tFrac16 MLIB_ShL_F16(register tFrac16 f16In1, register
tU16 u16In2);
```

**Arguments****Table 365. MLIB\_ShL\_F16 arguments**

Type	Name	Direction	Description
register tFrac16	f16In1	input	First value to be left shift.
register tU16	u16In2	input	The shift amount value.

**Return**

16-bit fractional value shifted to left by the shift amount. The bits beyond the 16-bit boundary are discarded.

**Note:** The shift amount cannot exceed in magnitude the bit-width of the shift value, that means must be within the range 0...15. Otherwise the result of the function is undefined. Due to effectivity reason this function is implemented as inline, and thus is not ANSI-C compliant.

### Code Example

```
#include "mlib.h"

tFrac16 f16In1;
tFrac16 f16Out;
tU16 u16In2;

void main(void)
{
    // first input = 0.25
    f16In1 = FRAC16(0.25);
    // second input = 1
    u16In2 = (tU16)1;

    // output should be 0x4000 ~ FRAC16(0.5)
    f16Out = MLIB_ShL_F16(f16In1, u16In2);

    // output should be 0x4000 ~ FRAC16(0.5)
    f16Out = MLIB_ShL(f16In1, u16In2, F16);

    // ##### Available only if 16-bit fractional implementation selected
    // as default
    // #####
    // output should be 0x4000 ~ FRAC16(0.5)
    f16Out = MLIB_ShL(f16In1, u16In2);
}
```

## 2.78 Function MLIB\_ShLSat

This function shifts the first parameter to left by number defined by second parameter and saturate if necessary.

### Description

This function shifts the first argument to the left by the amount specified in the second argument. The result is saturated.

### Re-entrancy

The function is re-entrant.

#### 2.78.1 Function MLIB\_ShLSat\_F32

##### Declaration

```
INLINE tFrac32 MLIB_ShLSat_F32(register tFrac32 f32In1, register
tU16 u16In2);
```

**Arguments****Table 366. MLIB\_ShLSat\_F32 arguments**

Type	Name	Direction	Description
register tFrac32	f32In1	input	First value to be left shift.
register tU16	u16In2	input	The shift amount value.

**Return**

32-bit fractional value shifted to left by the shift amount. The bits beyond the 32-bit boundary are discarded.

**Note:** The shift amount cannot exceed in magnitude the bit-width of the shift value, that means must be within the range 0...31. Otherwise the result of the function is undefined. Due to effectivity reason this function is implemented as inline assembly, and thus is not ANSI-C compliant.

**Code Example**

```
#include "mlib.h"

tFrac32 f32In1;
tFrac32 f32Out;
tU16 u16In2;

void main(void)
{
    // first input = 0.25
    f32In1 = FRAC32(0.25);
    // second input = 1
    u16In2 = (tU16)1;

    // output should be 0x40000000 ~ FRAC32(0.5)
    f32Out = MLIB_ShLSat_F32(f32In1, u16In2);

    // output should be 0x40000000 ~ FRAC32(0.5)
    f32Out = MLIB_ShLSat(f32In1, u16In2, F32);

    // ##### Available only if 32-bit fractional implementation selected
    // as default
    // #####
    // output should be 0x40000000 ~ FRAC32(0.5)
    f32Out = MLIB_ShLSat(f32In1, u16In2);
}
```

**2.78.2 Function MLIB\_ShLSat\_F16****Declaration**

```
INLINE tFrac16 MLIB_ShLSat_F16(register tFrac16 f16In1, register
tU16 u16In2);
```

**Arguments****Table 367. MLIB\_ShLSat\_F16 arguments**

Type	Name	Direction	Description
register tFrac16	f16In1	input	First value to be left shift.
register tU16	u16In2	input	The shift amount value.

**Return**

16-bit fractional value shifted to left by the shift amount. The bits beyond the 16-bit boundary are discarded.

**Note:** The shift amount cannot exceed in magnitude the bit-width of the shift value, that means must be within the range 0...15. Otherwise the result of the function is undefined. Due to effectivity reason this function is implemented as inline assembly, and thus is not ANSI-C compliant.

**Code Example**

```
#include "mlib.h"

tFrac16 f16In1;
tFrac16 f16Out;
tU16 u16In2;

void main(void)
{
    // first input = 0.25
    f16In1 = FRAC16(0.25);
    // second input = 1
    u16In2 = (tU16)1;

    // output should be 0x4000 ~ FRAC16(0.5)
    f16Out = MLIB_ShLSat_F16(f16In1, u16In2);

    // output should be 0x4000 ~ FRAC16(0.5)
    f16Out = MLIB_ShLSat(f16In1, u16In2, F16);

    // ##### Available only if 16-bit fractional implementation selected
    // as default
    // #####
    // output should be 0x4000 ~ FRAC16(0.5)
    f16Out = MLIB_ShLSat(f16In1, u16In2);
}
```

**2.79 Function MLIB\_ShR**

This function shifts the first parameter to right by number defined by second parameter.

**Description**

This function shifts the first argument to the right by the amount specified in the second argument.

## Re-entrancy

The function is re-entrant.

### 2.79.1 Function MLIB\_ShR\_F32

#### Declaration

```
INLINE tFrac32 MLIB_ShR_F32(register tFrac32 f32In1, register  
tU16 u16In2);
```

#### Arguments

**Table 368. MLIB\_ShR\_F32 arguments**

Type	Name	Direction	Description
register <a href="#">tFrac32</a>	f32In1	<a href="#">input</a>	First value to be right shift.
register <a href="#">tU16</a>	u16In2	<a href="#">input</a>	The shift amount value.

#### Return

32-bit fractional value shifted right by the shift amount. The bits beyond the 32-bit boundary of the result are discarded.

**Note:** The shift amount cannot exceed in magnitude the bit-width of the shifted value, that means it must be within the range 0...31. Otherwise the result of the function is undefined. Due to effectivity reason this function is implemented as inline, and thus is not ANSI-C compliant.

#### Code Example

```
#include "mlib.h"

tFrac32 f32In1;
tFrac32 f32Out;
tU16 u16In2;

void main(void)
{
    // first input = 0.25
    f32In1 = FRAC32(0.25);
    // second input = 1
    u16In2 = (tU16)1;

    // output should be 0x10000000 ~ FRAC32(0.125)
    f32Out = MLIB_ShR_F32(f32In1, u16In2);

    // output should be 0x10000000 ~ FRAC32(0.125)
    f32Out = MLIB_ShR(f32In1, u16In2, F32);

    // ##### Available only if 32-bit fractional implementation selected
    // as default
    // #####
}

// output should be 0x10000000 ~ FRAC32(0.125)
f32Out = MLIB_ShR(f32In1, u16In2);
```

```
}
```

## 2.79.2 Function MLIB\_ShR\_F16

### Declaration

```
INLINE tFrac16 MLIB_ShR_F16(register tFrac16 f16In1, register tU16 u16In2);
```

### Arguments

**Table 369. MLIB\_ShR\_F16 arguments**

Type	Name	Direction	Description
register <a href="#">tFrac16</a>	f16In1	<a href="#">input</a>	First value to be right shift.
register <a href="#">tU16</a>	u16In2	<a href="#">input</a>	The shift amount value.

### Return

16-bit fractional value shifted right by the shift amount. The bits beyond the 16-bit boundary of the result are discarded.

**Note:** The shift amount cannot exceed in magnitude the bit-width of the shifted value, that means it must be within the range 0...15. Otherwise the result of the function is undefined. Due to effectivity reason this function is implemented as inline, and thus is not ANSI-C compliant.

### Code Example

```
#include "mlib.h"

tFrac16 f16In1;
tFrac16 f16Out;
tU16 u16In2;

void main(void)
{
    // first input = 0.25
    f16In1 = FRAC16(0.25);
    // second input = 1
    u16In2 = (tU16)1;

    // output should be 0x1000 ~ FRAC16(0.125)
    f16Out = MLIB_ShR_F16(f16In1, u16In2);

    // output should be 0x1000 ~ FRAC16(0.125)
    f16Out = MLIB_ShR(f16In1, u16In2, F16);

    // #####
    // Available only if 16-bit fractional implementation selected
    // as default
    // #####
    // output should be 0x1000 ~ FRAC16(0.125)
    f16Out = MLIB_ShR(f16In1, u16In2);
}
```

## 2.80 Function MLIB\_Sub

This function subtracts the second parameter from the first one.

### Description

The second argument is subtracted from the first one.

### Re-entrancy

The function is re-entrant.

#### 2.80.1 Function MLIB\_Sub\_F32

##### Declaration

```
INLINE tFrac32 MLIB_Sub_F32(register tFrac32 f32In1, register tFrac32 f32In2);
```

##### Arguments

**Table 370. MLIB\_Sub\_F32 arguments**

Type	Name	Direction	Description
register <a href="#">tFrac32</a>	f32In1	input	Operand is a 32-bit number normalized between[-1,1).
register <a href="#">tFrac32</a>	f32In2	input	Operand is a 32-bit number normalized between[-1,1).

##### Return

The subtraction of the second argument from the first argument.

##### Implementation details

The input values as well as output value are considered as 32-bit fractional data type. The output saturation is not implemented in this function, thus in case the subtraction of input parameters is outside the [-1, 1) interval, the output value will overflow.

The output of the function is defined by the following simple equation:

$$f32Out = f32In1 - f32In2$$

Equation MLIB\_Sub\_F32\_Eq1

**Note:** Due to effectivity reason this function is implemented as inline and thus is not ANSI-C compliant.

##### Code Example

```
#include "mlib.h"

tFrac32 f32In1;
tFrac32 f32In2;
tFrac32 f32Out;

void main(void)
{
```

```

// first input = 0.5
f32In1 = FRAC32(0.5);

// second input = 0.25
f32In2 = FRAC32(0.25);

// output should be 0x20000000
f32Out = MLIB\_Sub\_F32(f32In1,f32In2);

// output should be 0x20000000
f32Out = MLIB\_Sub(f32In1,f32In2,F32);

// ##### Available only if 32-bit fractional implementation selected
// as default
// #####
// output should be 0x20000000
f32Out = MLIB\_Sub(f32In1,f32In2);
}

```

## 2.80.2 Function [MLIB\\_Sub\\_F16](#)

### Declaration

```
INLINE tFrac16 MLIB\_Sub\_F16(register tFrac16 f16In1, register tFrac16 f16In2);
```

### Arguments

**Table 371. [MLIB\\_Sub\\_F16](#) arguments**

Type	Name	Direction	Description
register <a href="#">tFrac16</a>	f16In1	input	Operand is a 16-bit number normalized between [-1,1].
register <a href="#">tFrac16</a>	f16In2	input	Operand is a 16-bit number normalized between [-1,1].

### Return

The subtraction of the second argument from the first argument.

### Implementation details

The input values as well as output value are considered as 16-bit fractional data type. The output saturation is not implemented in this function, thus in case the subtraction of input parameters is outside the [-1, 1) interval, the output value will overflow.

The output of the function is defined by the following simple equation:

$$f16Out = f16In1 - f16In2$$

Equation [MLIB\\_Sub\\_F16\\_Eq1](#)

**Note:** Due to effectivity reason this function is implemented as inline and thus is not ANSI-C compliant.

### Code Example

```
#include "mlib.h"

tFrac16 f16In1;
tFrac16 f16In2;
tFrac16 f16Out;

void main(void)
{
    // first input = 0.5
    f16In1 = FRAC16(0.5);

    // second input = 0.25
    f16In2 = FRAC16(0.25);

    // output should be 0x2000
    f16Out = MLIB_Sub_F16(f16In1,f16In2);

    // output should be 0x2000
    f16Out = MLIB_Sub(f16In1,f16In2,F16);

    // ##### Available only if 16-bit fractional implementation selected
    // as default
    // #####
    // output should be 0x20000000
    f16Out = MLIB_Sub(f16In1,f16In2);
}
```

### 2.80.3 Function MLIB\_Sub\_FLT

#### Declaration

```
INLINE tFloat MLIB_Sub_FLT(register tFloat fltIn1, register
tFloat fltIn2);
```

#### Arguments

**Table 372. MLIB\_Sub\_FLT arguments**

Type	Name	Direction	Description
register tFloat	fltIn1	input	Operand is a single precision floating point number.
register tFloat	fltIn2	input	Operand is a single precision floating point number.

#### Return

The subtraction of the second argument from the first argument.

#### Implementation details

The input value as well as output value is considered as single precision floating point data type.

The output of the function is defined by the following simple equation:

$$fltOut = fltIn1 - fltIn2$$

Equation MLIB\_Sub\_FLT\_Eq1

**Note:** The function may raise floating-point exceptions (invalid operation, overflow, underflow, inexact, input denormal). The floating-point unit must be enabled in CPACR register to prevent NOCP UsageFault exception.

Due to effectivity reason this function is implemented as inline and thus is not ANSI-C compliant.

### Code Example

```
#include "mlib.h"

tFloat fltIn1;
tFloat fltIn2;
tFloat fltOut;

void main(void)
{
    // first input = 50.5
    fltIn1 = (tFloat)50.5;

    // second input = 25.25
    fltIn2 = (tFloat)25.25;

    // output should be 25.25
    fltOut = MLIB_Sub_FLT(fltIn1, fltIn2);

    // output should be 25.25
    fltOut = MLIB_Sub(fltIn1, fltIn2, FLT);

    // ######
    // Available only if single precision floating point
    // implementation selected as default
    // #####
    // output should be 25.25
    fltOut = MLIB_Sub(fltIn1, fltIn2);
}
```

## 2.81 Function MLIB\_SubSat

This function subtracts the second parameter from the first one and saturate if necessary.

### Description

The second argument is subtracted from the first one. The result is saturated if necessary

### Re-entrancy

The function is re-entrant.

### 2.81.1 Function MLIB\_SubSat\_F32

#### Declaration

```
INLINE tFrac32 MLIB_SubSat_F32(register tFrac32 f32In1, register tFrac32 f32In2);
```

#### Arguments

**Table 373. MLIB\_SubSat\_F32 arguments**

Type	Name	Direction	Description
register <a href="#">tFrac32</a>	f32In1	input	Operand is a 32-bit number normalized between [-1,1).
register <a href="#">tFrac32</a>	f32In2	input	Operand is a 32-bit number normalized between [-1,1).

#### Return

The subtraction of the second argument from the first argument.

#### Implementation details

The input values as well as output value are considered as 32-bit fractional data type.

The output of the function is defined by the following simple equation:

$$f32Out = \begin{cases} FRAC32\_MIN & \text{if } (f32In1 - f32In2) < FRAC32\_MIN \\ f32In1 - f32In2 & \text{if } FRAC32\_MIN \leq (f32In1 - f32In2) \leq FRAC32\_MAX \\ FRAC32\_MAX & \text{if } (f32In1 - f32In2) > FRAC32\_MAX \end{cases}$$

Equation MLIB\_SubSat\_F32\_Eq1

**Note:** Due to effectivity reason this function is implemented as inline assembly and thus is not ANSI-C compliant.

#### Code Example

```
#include "mlib.h"

tFrac32 f32In1;
tFrac32 f32In2;
tFrac32 f32Out;

void main(void)
{
    // first input = 0.5
    f32In1 = FRAC32(0.5);

    // second input = 0.25
    f32In2 = FRAC32(0.25);

    // output should be 0x20000000
    f32Out = MLIB_SubSat_F32(f32In1, f32In2);

    // output should be 0x20000000
    f32Out = MLIB_SubSat(f32In1, f32In2, F32);

    // #####
```

```

// Available only if 32-bit fractional implementation selected
// as default
// #####
// output should be 0x20000000
f32Out = MLIB_SubSat(f32In1,f32In2);
}

```

## 2.81.2 Function `MLIB_SubSat_F16`

### Declaration

```
INLINE tFrac16 MLIB_SubSat_F16(register tFrac16 f16In1, register
tFrac16 f16In2);
```

### Arguments

**Table 374. `MLIB_SubSat_F16` arguments**

Type	Name	Direction	Description
register <code>tFrac16</code>	f16In1	input	Operand is a 16-bit number normalized between [-1,1).
register <code>tFrac16</code>	f16In2	input	Operand is a 16-bit number normalized between [-1,1).

### Return

The subtraction of the second argument from the first argument.

### Implementation details

The input values as well as output value are considered as 16-bit fractional data type.

The output of the function is defined by the following simple equation:

$$f16Out = \begin{cases} FRAC16\_MIN & \text{if } (f16In1 - f16In2) < FRAC16\_MIN \\ f16In1 - f16In2 & \text{if } FRAC16\_MIN \leq (f16In1 - f16In2) \leq FRAC16\_MAX \\ FRAC16\_MAX & \text{if } (f16In1 - f16In2) > FRAC16\_MAX \end{cases}$$

Equation `MLIB_SubSat_F16_Eq1`

**Note:** Due to effectivity reason this function is implemented as inline and thus is not ANSI-C compliant.

### Code Example

```

#include "mlib.h"

tFrac16 f16In1;
tFrac16 f16In2;
tFrac16 f16Out;

void main(void)
{
    // first input = 0.5
    f16In1 = FRAC16(0.5);

    // second input = 0.25
    f16In2 = FRAC16(0.25);
}

```

```

// output should be 0x2000
f16Out = MLIB_SubSat_F16(f16In1,f16In2);

// output should be 0x2000
f16Out = MLIB_SubSat(f16In1,f16In2,F16);

// ##### Available only if 16-bit fractional implementation selected
// as default
// #####
// output should be 0x2000
f16Out = MLIB_SubSat(f16In1,f16In2);
}

```

## 2.82 Function MLIB\_VAdd

This function returns vector sum of two input vectors.

### Description

This inline function returns the vector sum of two input vectors.

If the input vectors are defined as

$$\vec{u} = (u_1, u_2)$$

$$\vec{v} = (v_1, v_2)$$

Equation MLIB\_VAdd\_Eq1

then the output of the function is calculated by the following simple equations:

$$\vec{u} + \vec{v} = (u_1 + v_1, u_2 + v_2)$$

Equation MLIB\_VAdd\_Eq2

### Re-entrancy

The function is re-entrant.

#### 2.82.1 Function MLIB\_VAdd\_F32

##### Declaration

```
INLINE void MLIB_VAdd_F32(SWLIBS\_2Syst\_F32 *const f32Out, const
SWLIBS\_2Syst\_F32 *const f32In1, const SWLIBS\_2Syst\_F32 *const
f32In2);
```

##### Arguments

**Table 375. MLIB\_VAdd\_F32 arguments**

Type	Name	Direction	Description
<a href="#">SWLIBS_2Syst_F32</a> *const	f32Out	output	Sum of two input vector.

Type	Name	Direction	Description
const <a href="#">SWLIBS_2Syst_F32</a> *const	f32In1	input	First vector to be add.
const <a href="#">SWLIBS_2Syst_F32</a> *const	f32In2	input	Second vector to be add.

### Implementation details

The input values as well as output value is considered as vectors defined by two 32-bit fractional coordinates stored in [SWLIBS\\_2Syst\\_F32](#) structure.

**Note:** Due to effectivity reason this function is implemented as inline and thus is not ANSI-C compliant.

### Code Example:

```
#include "mlib.h"

SWLIBS\_2Syst\_F32 f32In1, f32In2;
SWLIBS\_2Syst\_F32 f32Out;

void main(void)
{
    // input vector 1 = (0.25,0.25)
    f32In1.f32Arg1 = FRAC32(0.25);
    f32In1.f32Arg2 = FRAC32(0.25);
    // input vector 2 = (0.25,0.25)
    f32In2.f32Arg1 = FRAC32(0.25);
    f32In2.f32Arg2 = FRAC32(0.25);

    // Both f32Out.f32Arg1 and f32Out.f32Arg2 should be FRAC32(0.5) = 0x40000000
    MLIB\_VAdd\_F32(&f32Out, &f32In1, &f32In2);

    // Both f32Out.f32Arg1 and f32Out.f32Arg2 should be FRAC32(0.5) = 0x40000000
    MLIB\_VAdd\(&f32Out, &f32In1, &f32In2, F32\);

    // ######
    // Available only if 32-bit fractional implementation selected
    // as default
    // #####
    // Both f32Out.f32Arg1 and f32Out.f32Arg2 should be FRAC32\(0.5\) = 0x40000000
    MLIB\\_VAdd\\(&f32Out, &f32In1, &f32In2\\);
}
```

## 2.82.2 Function [MLIB\\_VAdd\\_F16](#)

### Declaration

```
INLINE void MLIB\_VAdd\_F16(SWLIBS\_2Syst\_F16 *const f16Out, const
SWLIBS\_2Syst\_F16 *const f16In1, const SWLIBS\_2Syst\_F16 *const
f16In2);
```

**Arguments****Table 376. MLIB\_VAdd\_F16 arguments**

Type	Name	Direction	Description
<a href="#">SWLIBS_2Syst_F16</a> *const	f16Out	output	Sum of two input vector.
const <a href="#">SWLIBS_2Syst_F16</a> *const	f16In1	input	First vector to be add.
const <a href="#">SWLIBS_2Syst_F16</a> *const	f16In2	input	Second vector to be add.

**Implementation details**

The input values as well as output value is considered as vectors defined by two 16-bit fractional coordinates stored in [SWLIBS\\_2Syst\\_F16](#) structure.

**Note:** Due to effectivity reason this function is implemented as inline and thus is not ANSI-C compliant.

**Code Example:**

```
#include "mlib.h"

SWLIBS\_2Syst\_F16 f16In1, f16In2;
SWLIBS\_2Syst\_F16 f16Out;

void main(void)
{
    // input vector 1 = (0.25,0.25)
    f16In1.f16Arg1 = FRAC16(0.25);
    f16In1.f16Arg2 = FRAC16(0.25);
    // input vector 2 = (0.25,0.25)
    f16In2.f16Arg1 = FRAC16(0.25);
    f16In2.f16Arg2 = FRAC16(0.25);

    // Both f16Out.f16Arg1 and f16Out.f16Arg2 should be FRAC16(0.5) = 0x4000
    MLIB_VAdd_F16(&f16Out, &f16In1, &f16In2);

    // Both f16Out.f16Arg1 and f16Out.f16Arg2 should be FRAC16(0.5) = 0x4000
    MLIB_VAdd(&f16Out, &f16In1, &f16In2, F16);

    // ##### Available only if 16-bit fractional implementation selected
    // as default
    // #####
    // Both f16Out.f16Arg1 and f16Out.f16Arg2 should be FRAC16(0.5) = 0x4000
    MLIB_VAdd(&f16Out, &f16In1, &f16In2);
}
```

### 2.82.3 Function MLIB\_VAdd\_FLT

#### Declaration

```
INLINE void MLIB_VAdd_FLT(SWLIBS\_2Syst\_FLT *const fltOut, const
SWLIBS\_2Syst\_FLT *const fltIn1, const SWLIBS\_2Syst\_FLT *const
fltIn2);
```

#### Arguments

**Table 377. MLIB\_VAdd\_FLT arguments**

Type	Name	Direction	Description
<a href="#">SWLIBS_2Syst_FLT</a> *const	fltOut	output	Sum of two input vector.
const <a href="#">SWLIBS_2Syst_FLT</a> *const	fltIn1	input	First vector to be add.
const <a href="#">SWLIBS_2Syst_FLT</a> *const	fltIn2	input	Second vector to be add.

#### Implementation details

The input values as well as output value is considered as vectors defined by two single precision floating-point coordinates stored in [SWLIBS\\_2Syst\\_FLT](#) structure.

**Note:** The function may raise floating-point exceptions (invalid operation, overflow, underflow, inexact, input denormal). The floating-point unit must be enabled in CPACR register to prevent NOCP UsageFault exception.

*Due to effectivity reason this function is implemented as inline and thus is not ANSI-C compliant.*

#### Code Example

```
#include "mlib.h"

SWLIBS\_2Syst\_FLT fltIn1, fltIn2;
SWLIBS\_2Syst\_FLT fltOut;

void main(void)
{
    // input vector 1 = (0.25,0.25)
    fltIn1.fltArg1 = 0.25;
    fltIn1.fltArg2 = 0.25;
    // input vector 2 = (0.25,0.25)
    fltIn2.fltArg1 = 0.25;
    fltIn2.fltArg2 = 0.25;

    // Both fltOut.fltArg1 and fltOut.fltArg2 should be 0.5
    MLIB_VAdd_FLT(&fltOut, &fltIn1, &fltIn2);

    // Both fltOut.fltArg1 and fltOut.fltArg2 should be 0.5
    MLIB_VAdd(&fltOut, &fltIn1, &fltIn2, FLT);

    // ##### Available only if single precision floating-point
    // implementation selected as default
}
```

```
// ######
// Both fltOut.fltArg1 and fltOut.fltArg2 should be 0.5
MLIB_VAdd(&fltOut, &fltIn1, &fltIn2);
}
```

## 2.83 Function MLIB\_VMac

This function implements the vector multiply accumulate function.

### Description

This inline function returns the dot product of four input values.

### Re-entrancy

The function is re-entrant.

#### 2.83.1 Function MLIB\_VMac\_F32

##### Declaration

```
INLINE tFrac32 MLIB_VMac_F32(register tFrac32 f32In1, register tFrac32 f32In2, register tFrac32 f32In3, register tFrac32 f32In4);
```

##### Arguments

**Table 378. MLIB\_VMac\_F32 arguments**

Type	Name	Direction	Description
register <a href="#">tFrac32</a>	f32In1	input	First input value to first multiplication.
register <a href="#">tFrac32</a>	f32In2	input	Second input value to first multiplication.
register <a href="#">tFrac32</a>	f32In3	input	First input value to second multiplication.
register <a href="#">tFrac32</a>	f32In4	input	Second input value to second multiplication.

##### Return

Vector multiplied input values with addition.

##### Implementation details

The input values as well as output value is considered as 32-bit fractional values. The output saturation is implemented only for multiplication result inside this function. The output saturation is not implemented for addition in this function, thus in case the add of input values is outside the [-1, 1) interval, the output value will overflow.

The output of the function is defined by the following simple equation:

$$f32Out = (f32In1 \cdot f32In2) + (f32In3 \cdot f32In4)$$

Equation MLIB\_VMac\_F32\_Eq1

**Note:** Due to effectivity reason this function is implemented as inline and thus is not ANSI-C compliant.

**Code Example**

```
#include "mlib.h"

tFrac32 f32In1;
tFrac32 f32In2;
tFrac32 f32In3;
tFrac32 f32In4;
tFrac32 f32Out;

void main(void)
{
    // input1 value = 0.25
    f32In1 = FRAC32(0.25);

    // input2 value = 0.15
    f32In2 = FRAC32(0.15);

    // input3 value = 0.35
    f32In3 = FRAC32(0.35);

    // input4 value = 0.45
    f32In4 = FRAC32(0.45);

    // output should be FRAC32(0.195) = 0x18F5C28F
    f32Out = MLIB_VMac_F32(f32In1,f32In2,f32In3,f32In4);

    // output should be FRAC32(0.195) = 0x18F5C28F
    f32Out = MLIB_VMac(f32In1,f32In2,f32In3,f32In4, F32);

    // ##### Available only if 32-bit fractional implementation selected #####
    // as default
    // #####
    // output should be FRAC32(0.195) = 0x18F5C28F
    f32Out = MLIB_VMac(f32In1,f32In2,f32In3,f32In4);
}
```

**2.83.2 Function MLIB\_VMac\_F32F16F16****Declaration**

```
INLINE tFrac32 MLIB_VMac_F32F16F16(register tFrac16 f16In1,
register tFrac16 f16In2, register tFrac16 f16In3, register
tFrac16 f16In4);
```

**Arguments****Table 379. MLIB\_VMac\_F32F16F16 arguments**

Type	Name	Direction	Description
register tFrac16	f16In1	input	First input value to first multiplication.
register tFrac16	f16In2	input	Second input value to first multiplication.
register tFrac16	f16In3	input	First input value to second multiplication.
register tFrac16	f16In4	input	Second input value to second multiplication.

## Return

Vector multiplied input values with addition.

## Implementation details

The input values are considered as 16-bit fractional values and the output value is considered as 32-bit fractional value. The output saturation is implemented only for multiplication result inside this function. The output saturation is not implemented for addition in this function, thus in case the add of input values is outside the [-1, 1) interval, the output value will overflow.

The output of the function is defined by the following simple equation:

$$f32Out = (f16In1 \cdot f16In2) + (f16In3 \cdot f16In4)$$

Equation MLIB\_VMac\_F32F16F16\_Eq1

**Note:** Due to effectivity reason this function is implemented as inline and thus is not ANSI-C compliant.

## Code Example

```
#include "mlib.h"

tFrac16 f16In1;
tFrac16 f16In2;
tFrac16 f16In3;
tFrac16 f16In4;
tFrac32 f32Out;

void main(void)
{
    // input1 value = 0.25
    f16In1 = FRAC16(0.25);

    // input2 value = 0.15
    f16In2 = FRAC16(0.15);

    // input3 value = 0.35
    f16In3 = FRAC16(0.35);

    // input4 value = 0.45
    f16In4 = FRAC16(0.45);

    // output should be FRAC32(0.195) = 0x18F5C28F
    f32Out = MLIB_VMac_F32F16F16(f16In1, f16In2, f16In3, f16In4);

    // output should be FRAC32(0.195) = 0x18F5C28F
    f32Out = MLIB_VMac(f16In1, f16In2, f16In3, f16In4, F32F16F16);
}
```

### 2.83.3 Function MLIB\_VMac\_F16

#### Declaration

```
INLINE tFrac16 MLIB_VMac_F16(register tFrac16 f16In1, register
tFrac16 f16In2, register tFrac16 f16In3, register tFrac16
f16In4);
```

#### Arguments

**Table 380. MLIB\_VMac\_F16 arguments**

Type	Name	Direction	Description
register <b>tFrac16</b>	f16In1	<b>input</b>	First input value to first multiplication.
register <b>tFrac16</b>	f16In2	<b>input</b>	Second input value to first multiplication.
register <b>tFrac16</b>	f16In3	<b>input</b>	First input value to second multiplication.
register <b>tFrac16</b>	f16In4	<b>input</b>	Second input value to second multiplication.

#### Return

Vector multiplied input values with addition.

#### Implementation details

The input values as well as output value is considered as 16-bit fractional values. The output saturation is implemented only for multiplication result inside this function. The output saturation is not implemented for addition in this function, thus in case the add of input values is outside the [-1, 1) interval, the output value will overflow.

The output of the function is defined by the following simple equation:

$$f16Out = (f16In1 \cdot f16In2) + (f16In3 \cdot f16In4)$$

Equation MLIB\_VMac\_F16\_Eq1

**Note:** Due to effectivity reason this function is implemented as inline and thus is not ANSI-C compliant.

#### Code Example

```
#include "mlib.h"

tFrac16 f16In1;
tFrac16 f16In2;
tFrac16 f16In3;
tFrac16 f16In4;
tFrac16 f16Out;

void main(void)
{
    // input1 value = 0.25
    f16In1 = FRAC16(0.25);

    // input2 value = 0.15
    f16In2 = FRAC16(0.15);

    // input3 value = 0.35
```

```

f16In3 = FRAC16(0.35);

// input4 value = 0.45
f16In4 = FRAC16(0.45);

// output should be FRAC16(0.195) = 0x18F5
f16Out = MLIB_VMac_F16(f16In1,f16In2,f16In3,f16In4);

// output should be FRAC16(0.195) = 0x18F5
f16Out = MLIB_VMac(f16In1,f16In2,f16In3,f16In4, F16);

// ######
// Available only if 16-bit fractional implementation selected
// as default
// #####
// output should be FRAC16(0.195) = 0x18F5
f16Out = MLIB_VMac(f16In1,f16In2,f16In3,f16In4);
}

```

## 2.83.4 Function MLIB\_VMac\_FLT

### Declaration

```
INLINE tFloat MLIB_VMac_FLT(register tFloat fltIn1, register
tFloat fltIn2, register tFloat fltIn3, register tFloat fltIn4);
```

### Arguments

**Table 381. MLIB\_VMac\_FLT arguments**

Type	Name	Direction	Description
register <u>tFloat</u>	fltIn1	input	First input value to first multiplication.
register <u>tFloat</u>	fltIn2	input	Second input value to first multiplication.
register <u>tFloat</u>	fltIn3	input	First input value to second multiplication.
register <u>tFloat</u>	fltIn4	input	Second input value to second multiplication.

### Return

Vector multiplied input values with addition.

### Implementation details

The input values as well as output value is considered as single precision floating point values.

The output of the function is defined by the following simple equation:

$$fltOut = (fltIn1 \cdot fltIn2) + (fltIn3 \cdot fltIn4)$$

Equation MLIB\_VMac\_FLT\_Eq1

**Note:** The function may raise floating-point exceptions (invalid operation, overflow, underflow, inexact, input denormal). The floating-point unit must be enabled in CPACR register to prevent NOCP UsageFault exception.

Due to effectivity reason this function is implemented as inline assembly and thus is not ANSI-C compliant.

### Code Example

```
#include "mlib.h"

tFloat fltIn1;
tFloat fltIn2;
tFloat fltIn3;
tFloat fltIn4;
tFloat fltOut;

void main(void)
{
    // input1 value = 0.25
    fltIn1 = (tFloat)0.25;

    // input2 value = 0.15
    fltIn2 = (tFloat)0.15;

    // input3 value = 0.35
    fltIn3 = (tFloat)0.35;

    // input4 value = 0.45
    fltIn4 = (tFloat)0.45;

    // output should be 0.195
    fltOut = MLIB_VMac_FLT(fltIn1, fltIn2, fltIn3, fltIn4);

    // output should be 0.195
    fltOut = MLIB_VMac(fltIn1, fltIn2, fltIn3, fltIn4, FLT);

    // ######
    // Available only if single precision floating point
    // implementation selected as default
    // #####
    // output should be 0.195
    fltOut = MLIB_VMac(fltIn1, fltIn2, fltIn3, fltIn4);
}
```

## 2.84 Function MLIB\_VScale

This function returns input vector scaled by second input value.

### Description

This inline function returns a vector multiplied by a scalar.

If the scale is defined as  $\alpha$  and the input vector as

$$\vec{u} = (u_1, u_2)$$

Equation MLIB\_VScale\_Eq1

then the output of the function is calculated by the following simple equation:

$$\alpha \cdot \vec{u} = (\alpha \cdot u_1, \alpha \cdot u_2)$$

Equation MLIB\_VScale\_Eq2

## Re-entrancy

The function is re-entrant.

### 2.84.1 Function MLIB\_VScale\_F32

#### Declaration

```
INLINE void MLIB_VScale_F32(SWLIBS\_2Syst\_F32 *const f32OutVec,  
const SWLIBS\_2Syst\_F32 *const f32InVec, tFrac32 f32InScale);
```

#### Arguments

**Table 382. MLIB\_VScale\_F32 arguments**

Type	Name	Direction	Description
<a href="#">SWLIBS_2Syst_F32</a> *const	f32OutVec	output	Scaled vector.
const <a href="#">SWLIBS_2Syst_F32</a> *const	f32InVec	input	Input vector to be scaled.
<a href="#">tFrac32</a>	f32InScale	input	Scaling coefficient.

#### Implementation details

The first input as well as output value are considered as vectors defined by two 32-bit fractional coordinates stored in [SWLIBS\\_2Syst\\_F32](#) structure. Input vector is scaled by second scalar input in 32-bit fractional format.

**Note:** Due to effectivity reason this function is implemented as inline and thus is not ANSI-C compliant.

#### Code Example:

```
#include "mlib.h"

SWLIBS\_2Syst\_F32 f32InVec;
tFrac32 f32InScale;
SWLIBS\_2Syst\_F32 f32Out;

void main(void)
{
    // input vector = (0.5,0.5)
    f32InVec.f32Arg1 = FRAC32(0.5);
    f32InVec.f32Arg2 = FRAC32(0.5);
    // input scale = 0.5
    f32InScale = FRAC32(0.5);

    // Both f32Out.f32Arg1 and f32Out.f32Arg2 should be FRAC32(0.25) = 0x20000000
    MLIB_VScale_F32(&f32Out, &f32InVec, f32InScale);

    // Both f32Out.f32Arg1 and f32Out.f32Arg2 should be FRAC32(0.25) = 0x20000000
    MLIB_VScale(&f32Out, &f32InVec, f32InScale, F32);
```

```

// ######
// Available only if 32-bit fractional implementation selected
// as default
// #####
// Both f32Out.f32Arg1 and f32Out.f32Arg2 should be FRAC32(0.25) = 0x20000000
MLIB_VScale(&f32Out, &f32InVec, f32InScale);
}

```

## 2.84.2 Function MLIB\_VScale\_F16

### Declaration

```
INLINE void MLIB_VScale_F16(SWLIBS\_2Syst\_F16 *const f16OutVec,
const SWLIBS\_2Syst\_F16 *const f16InVec, tFrac16 f16InScale);
```

### Arguments

**Table 383. MLIB\_VScale\_F16 arguments**

Type	Name	Direction	Description
<a href="#">SWLIBS_2Syst_F16</a> *const	f16OutVec	<b>output</b>	Scaled vector.
const <a href="#">SWLIBS_2Syst_F16</a> *const	f16InVec	<b>input</b>	Input vector to be scaled.
<a href="#">tFrac16</a>	f16InScale	<b>input</b>	Scaling coefficient.

### Implementation details

The first input as well as output value are considered as vectors defined by two 16-bit fractional coordinates stored in [SWLIBS\\_2Syst\\_F16](#) structure. Input vector is scaled by second scalar input in 16-bit fractional format.

**Note:** Due to effectivity reason this function is implemented as inline and thus is not ANSI-C compliant.

### Code Example:

```

#include "mlib.h"

SWLIBS\_2Syst\_F16 f16InVec;
tFrac16 f16InScale;
SWLIBS\_2Syst\_F16 f16Out;

void main(void)
{
    // input vector 1 = (0.5,0.5)
    f16InVec.f16Arg1 = FRAC16(0.5);
    f16InVec.f16Arg2 = FRAC16(0.5);
    // input scale = 0.5
    f16InScale = FRAC16(0.5);

    // Both f16Out.f16Arg1 and f16Out.f16Arg2 should be FRAC16(0.25) = 0x2000
    MLIB_VScale_F16(&f16Out, &f16InVec, f16InScale);

    // Both f16Out.f16Arg1 and f16Out.f16Arg2 should be FRAC16(0.25) = 0x2000
    MLIB_VScale(&f16Out, &f16InVec, f16InScale, F16);
}

```

```

// ######
// Available only if 16-bit fractional implementation selected
// as default
// #####
// Both f16Out.f16Arg1 and f16Out.f16Arg2 should be FRAC16(0.25) = 0x2000
MLIB_VScale(&f16Out, &f16InVec, f16InScale);
}

```

### 2.84.3 Function `MLIB_VScale_FLT`

#### Declaration

```
INLINE void MLIB_VScale_FLT(SWLIBS\_2Syst\_FLT *const fltOutVec,
const SWLIBS\_2Syst\_FLT *const fltInVec, tFloat fltInScale);
```

#### Arguments

**Table 384. `MLIB_VScale_FLT` arguments**

Type	Name	Direction	Description
<a href="#">SWLIBS_2Syst_FLT</a> *const	fltOutVec	<b>output</b>	Scaled vector.
const <a href="#">SWLIBS_2Syst_FLT</a> *const	fltInVec	<b>input</b>	Input vector to be scaled.
<a href="#">tFloat</a>	fltInScale	<b>input</b>	Scaling coefficient.

#### Implementation details

The first input as well as output value are considered as vectors defined by two single precision floating-point coordinates stored in [SWLIBS\\_2Syst\\_FLT](#) structure. Input vector is scaled by second scalar input in single precision floating-point format.

**Note:** The function may raise floating-point exceptions (invalid operation, overflow, underflow, inexact, input denormal). The floating-point unit must be enabled in CPACR register to prevent NOCP UsageFault exception.

*Due to effectiveness reason this function is implemented as inline and thus is not ANSI-C compliant.*

#### Code Example

```

#include "mlib.h"

SWLIBS\_2Syst\_FLT fltInVec;
tFloat fltInScale;
SWLIBS\_2Syst\_FLT fltOut;

void main(void)
{
    // input vector 1 = (0.5,0.5)
    fltInVec.fltArg1 = 0.5;
    fltInVec.fltArg2 = 0.5;
    // input scale = 0.5
    fltInScale = 0.5;
}

```

```

// Both fltOut.fltArg1 and fltOut.fltArg2 should be 0.25
MLIB_VScale_FLT(&fltOut, &fltInVec, fltInScale);

// Both fltOut.fltArg1 and fltOut.fltArg2 should be 0.25
MLIB_VScale(&fltOut, &fltInVec, fltInScale, FLT);

// ##### Available only if single precision floating-point
// implementation selected as default
// #####
// Both fltOut.fltArg1 and fltOut.fltArg2 should be 0.25
MLIB_VScale(&fltOut, &fltInVec, fltInScale);
}

```

## 2.85 Function MLIB\_VSub

This function returns vector subtract of two input vectors.

### Description

This inline function returns the vector subtract of two input vectors.

If the input vectors are defined as

$$\vec{u} = (u_1, u_2)$$

$$\vec{v} = (v_1, v_2)$$

Equation MLIB\_VAdd\_Eq1

then the output of the function is calculated by the following simple equations:

$$\vec{u} + \vec{v} = (u_1 + v_1, u_2 + v_2)$$

Equation MLIB\_VAdd\_Eq2

### Re-entrancy

The function is re-entrant.

#### 2.85.1 Function MLIB\_VSub\_F32

##### Declaration

```
INLINE void MLIB_VSub_F32(SWLIBS\_2Syst\_F32 *const f32Out, const
SWLIBS\_2Syst\_F32 *const f32In1, const SWLIBS\_2Syst\_F32 *const
f32In2);
```

##### Arguments

**Table 385. MLIB\_VSub\_F32 arguments**

Type	Name	Direction	Description
<a href="#">SWLIBS_2Syst_F32</a> *const	f32Out	output	Subtraction of two input vectors.

Type	Name	Direction	Description
const <a href="#">SWLIBS_2Syst_F32</a> *const	f32In1	input	Vector representing first operand.
const <a href="#">SWLIBS_2Syst_F32</a> *const	f32In2	input	Vector representing second operand.

### Implementation details

The input values as well as output value is considered as vectors defined by two 32-bit fractional coordinates stored in [SWLIBS\\_2Syst\\_F32](#) structure.

**Note:** Due to effectivity reason this function is implemented as inline and thus is not ANSI-C compliant.

### Code Example:

```
#include "mlib.h"

SWLIBS\_2Syst\_F32 f32In1, f32In2;
SWLIBS\_2Syst\_F32 f32Out;

void main(void)
{
    // input vector 1 = (0.5,0.5)
    f32In1.f32Arg1 = FRAC32(0.5);
    f32In1.f32Arg2 = FRAC32(0.5);
    // input vector 2 = (0.25,0.25)
    f32In2.f32Arg1 = FRAC32(0.25);
    f32In2.f32Arg2 = FRAC32(0.25);

    // Both f32Out.f32Arg1 and f32Out.f32Arg2 should be FRAC32(0.25) = 0x20000000
    MLIB_VSub_F32(&f32Out, &f32In1, &f32In2);

    // Both f32Out.f32Arg1 and f32Out.f32Arg2 should be FRAC32(0.25) = 0x20000000
    MLIB_VSub(&f32Out, &f32In1, &f32In2, F32);

    // ######
    // Available only if 32-bit fractional implementation selected
    // as default
    // #####
    // Both f32Out.f32Arg1 and f32Out.f32Arg2 should be FRAC32(0.25) = 0x20000000
    MLIB_VSub(&f32Out, &f32In1, &f32In2);
}
```

## 2.85.2 Function MLIB\_VSub\_F16

### Declaration

```
INLINE void MLIB_VSub_F16(SWLIBS\_2Syst\_F16 *const f16Out, const
SWLIBS\_2Syst\_F16 *const f16In1, const SWLIBS\_2Syst\_F16 *const
f16In2);
```

**Arguments****Table 386. MLIB\_VSub\_F16 arguments**

Type	Name	Direction	Description
<a href="#">SWLIBS_2Syst_F16</a> *const	f16Out	output	Subtraction of two input vectors.
const <a href="#">SWLIBS_2Syst_F16</a> *const	f16In1	input	Vector representing first operand.
const <a href="#">SWLIBS_2Syst_F16</a> *const	f16In2	input	Vector representing second operand.

**Implementation details**

The input values as well as output value is considered as vectors defined by two 16-bit fractional coordinates stored in [SWLIBS\\_2Syst\\_F16](#) structure.

**Note:** Due to effectivity reason this function is implemented as inline and thus is not ANSI-C compliant.

**Code Example:**

```
#include "mlib.h"

SWLIBS\_2Syst\_F16 f16In1, f16In2;
SWLIBS\_2Syst\_F16 f16Out;

void main(void)
{
    // input vector 1 = (0.5,0.5)
    f16In1.f16Arg1 = FRAC16(0.5);
    f16In1.f16Arg2 = FRAC16(0.5);
    // input vector 2 = (0.25,0.25)
    f16In2.f16Arg1 = FRAC16(0.25);
    f16In2.f16Arg2 = FRAC16(0.25);

    // Both f16Out.f16Arg1 and f16Out.f16Arg2 should be FRAC16(0.25) = 0x2000
    MLIB_VSub_F16(&f16Out, &f16In1, &f16In2);

    // Both f16Out.f16Arg1 and f16Out.f16Arg2 should be FRAC16(0.25) = 0x2000
    MLIB_VSub(&f16Out, &f16In1, &f16In2, F16);

    // ##### Available only if 16-bit fractional implementation selected
    // as default
    // #####
    // Both f16Out.f16Arg1 and f16Out.f16Arg2 should be FRAC16(0.25) = 0x2000
    MLIB_VSub(&f16Out, &f16In1, &f16In2);
}
```

### 2.85.3 Function MLIB\_VSub\_FLT

#### Declaration

```
INLINE void MLIB_VSub_FLT(SWLIBS\_2Syst\_FLT *const fltOut, const
SWLIBS\_2Syst\_FLT *const fltIn1, const SWLIBS\_2Syst\_FLT *const
fltIn2);
```

#### Arguments

**Table 387. MLIB\_VSub\_FLT arguments**

Type	Name	Direction	Description
<a href="#">SWLIBS_2Syst_FLT</a> *const	fltOut	output	Subtraction of two input vectors.
const <a href="#">SWLIBS_2Syst_FLT</a> *const	fltIn1	input	Vector representing first operand.
const <a href="#">SWLIBS_2Syst_FLT</a> *const	fltIn2	input	Vector representing second operand.

#### Implementation details

The input values as well as output value is considered as vectors defined by two single precision floating-point coordinates stored in [SWLIBS\\_2Syst\\_FLT](#) structure.

**Note:** The function may raise floating-point exceptions (invalid operation, overflow, underflow, inexact, input denormal). The floating-point unit must be enabled in CPACR register to prevent NOCP UsageFault exception.

*Due to effectivity reason this function is implemented as inline and thus is not ANSI-C compliant.*

#### Code Example

```
#include "mlib.h"

SWLIBS\_2Syst\_FLT fltIn1, fltIn2;
SWLIBS\_2Syst\_FLT fltOut;

void main(void)
{
    // input vector 1 = (0.5,0.5)
    fltIn1.fltArg1 = 0.5;
    fltIn1.fltArg2 = 0.5;
    // input vector 2 = (0.25,0.25)
    fltIn2.fltArg1 = 0.25;
    fltIn2.fltArg2 = 0.25;

    // Both fltOut.fltArg1 and fltOut.fltArg2 should be 0.25
    MLIB_VSub_FLT(&fltOut, &fltIn1, &fltIn2);

    // Both fltOut.fltArg1 and fltOut.fltArg2 should be 0.25
    MLIB_VSub(&fltOut, &fltIn1, &fltIn2, FLT);

    // ##### Available only if single precision floating-point
    // implementation selected as default
}
```

```
// ######
// Both fltOut.fltArg1 and fltOut.fltArg2 should be 0.25
MLIB_VSub(&fltOut, &fltIn1, &fltIn2);
}
```

## 2.86 Function SWLIBS\_GetVersion

This function returns the information about AMMCLIB version.

### Declaration

```
const SWLIBS\_VERSION\_T * SWLIBS_GetVersion();
```

### Return

The function returns the information about the version of Motor Control Library Set.

### Description

The function returns the information about the version of Motor Control Library Set. The information are structured as follows:

- Motor Control Library Set identification code
- Motor Control Library Set version code
- Motor Control Library Set supported implementation code

### Reentrancy

The function is reentrant.

## 3 Typedefs

### Typeface index

**Table 388. Quick typedefs reference**

Type	Name	Description
typedef double	tDouble	double precision float type
typedef float	tFloat	single precision float type
typedef ts16	tFrac16	16-bit signed fractional Q1.15 type
typedef ts32	tFrac32	32-bit Q1.31 type
typedef signed short	ts16	signed 16-bit integer type
typedef signed long	ts32	signed 32-bit integer type
typedef signed long long	ts64	signed 64-bit integer type
typedef signed char	ts8	signed 8-bit integer type
typedef unsigned short	tu16	unsigned 16-bit integer type
typedef unsigned long	tu32	unsigned 32-bit integer type
typedef unsigned long long	tu64	unsigned 64-bit integer type
typedef unsigned char	tu8	unsigned 8-bit integer type

## 4 Enums

This section describes in details the enums available in Automotive Math and Motor Control Library Set for NXP S32K14x devices.

### 4.1 AMCLIB\_WINDMILLING\_RET\_T

Enum type of the AMCLIB\_Windmilling return value.

#### Source File

```
#include <AMCLIB_Windmilling.h>
```

#### Enumerated Type Members

**Table 389. AMCLIB\_WINDMILLING\_RET\_T members description**

Name	Value	Description
UNDECIDED	0	AMCLIB_Windmilling has not yet decided whether the motor is spinning or not. The AMCLIB_Windmilling must be called again in the next sampling period.
SPINNING	1	The motor is spinning.
STOPPED	2	The motor is stopped.

### 4.2 tBool

Enum representing basic boolean type.

#### Source File

```
#include <SWLIBS_Typedefs.h>
```

#### Enumerated Type Members

**Table 390. tBool members description**

Name	Value	Description
FALSE	0	Boolean type FALSE constant
TRUE	1	Boolean type TRUE constant

## 5 Compound data types

This section describes in details the compound data types definitions available in Automotive Math and Motor Control Library Set for NXP S32K14x devices.

### 5.1 AMCLIB\_BEMF\_OBSRV\_DQ\_T\_F16

Observer configuration structure.

**Source File**

```
#include <AMCLIB_BemfObsrvDQ.h>
```

**Compound Type Members****Table 391. AMCLIB\_BEMF\_OBSRV\_DQ\_T\_F16 members description**

Type	Name	Description
<a href="#">SWLIBS_2Syst_F16</a>	pEObsrv	Estimated BEMF - D/Q.
<a href="#">SWLIBS_2Syst_F32</a>	pIObsrv	Estimated current - D/Q.
<a href="#">GFLIB_CONTROLLER_PIAW_R_T_F16</a>	pParamD	Observer parameters for D-axis controller.
<a href="#">GFLIB_CONTROLLER_PIAW_R_T_F16</a>	pParamQ	Observer parameters for Q-axis controller.
<a href="#">SWLIBS_2Syst_F32</a>	pIObsrvln_1L	Inputs of RL circuit at step k-1 - low word
<a href="#">SWLIBS_2Syst_F16</a>	pIObsrvln_1H	Inputs of RL circuit at step k-1 - high word
<a href="#">tFrac16</a>	f16IGain	Scaled RL circuit constant, fractional format normalized to fit into (- $2^{15}$ , $2^{15}-1$ ).
<a href="#">tFrac16</a>	f16UGain	Scaled voltage cross-coupling constant, fractional format normalized to fit into (- $2^{15}$ , $2^{15}-1$ ).
<a href="#">tFrac16</a>	f16WIGain	Scaled angular velocity cross-coupling constant, fractional format normalized to fit into (- $2^{15}$ , $2^{15}-1$ ).
<a href="#">tFrac16</a>	f16EGain	Scaled back-EMF cross-coupling constant, fractional format normalized to fit into (- $2^{15}$ , $2^{15}-1$ ).
<a href="#">tS16</a>	s16Shift	Scaling bitwise shift applied to all cross-coupling constants, integer format in the range [-14, 14]. Function accuracy guaranteed only for range [-1, 1].

**5.2 AMCLIB\_BEMF\_OBSRV\_DQ\_T\_F32**

Observer configuration structure.

**Source File**

```
#include <AMCLIB_BemfObsrvDQ.h>
```

**Compound Type Members****Table 392. AMCLIB\_BEMF\_OBSRV\_DQ\_T\_F32 members description**

Type	Name	Description
<a href="#">SWLIBS_2Syst_F32</a>	pEObsrv	Estimated BEMF - D/Q.
<a href="#">SWLIBS_2Syst_F32</a>	pIObsrv	Estimated current - D/Q.
<a href="#">GFLIB_CONTROLLER_PIAW_R_T_F32</a>	pParamD	Observer parameters for D-axis controller.
<a href="#">GFLIB_CONTROLLER_PIAW_R_T_F32</a>	pParamQ	Observer parameters for Q-axis controller.

Type	Name	Description
<a href="#">SWLIBS_2Syst_F32</a>	pIObsrvIn_1L	Inputs of RL circuit at step k-1 - low word
<a href="#">SWLIBS_2Syst_F16</a>	pIObsrvIn_1H	Inputs of RL circuit at step k-1 - high word
<a href="#">tFrac32</a>	f32IGain	Scaled RL circuit constant, fractional format normalized to fit into (- $2^{31}$ , $2^{31}-1$ ).
<a href="#">tFrac32</a>	f32UGain	Scaled voltage cross-coupling constant, fractional format normalized to fit into (- $2^{31}$ , $2^{31}-1$ ).
<a href="#">tFrac32</a>	f32WIGain	Scaled angular velocity cross-coupling constant, fractional format normalized to fit into (- $2^{31}$ , $2^{31}-1$ ).
<a href="#">tFrac32</a>	f32EGain	Scaled back-EMF cross-coupling constant, fractional format normalized to fit into (- $2^{31}$ , $2^{31}-1$ ).
<a href="#">tS16</a>	s16Shift	Scaling bitwise shift applied to all cross-coupling constants, integer format in the range [-14, 14]. Function accuracy guaranteed only for range [-1, 1].

### 5.3 AMCLIB\_BEMF\_OBSRV\_DQ\_T\_FLT

Observer configuration structure.

#### Source File

```
#include <AMCLIB_BemfObsrvDQ.h>
```

#### Compound Type Members

Table 393. AMCLIB\_BEMF\_OBSRV\_DQ\_T\_FLT members description

Type	Name	Description
<a href="#">SWLIBS_2Syst_FLT</a>	pEObsrv	Estimated BEMF - D/Q.
<a href="#">SWLIBS_2Syst_FLT</a>	pIObsrv	Estimated current - D/Q.
<a href="#">GFLIB_CONTROLLER_PIAW_R_T_FLT</a>	pParamD	Observer parameters for D-axis controller.
<a href="#">GFLIB_CONTROLLER_PIAW_R_T_FLT</a>	pParamQ	Observer parameters for Q-axis controller.
<a href="#">SWLIBS_2Syst_FLT</a>	pIObsrvIn_1	Inputs of RL circuit at step k-1
<a href="#">tFloat</a>	fltIGain	RL circuit constant, single precision floating point format.
<a href="#">tFloat</a>	fltUGain	Voltage cross-coupling constant, single precision floating point format.
<a href="#">tFloat</a>	fltWIGain	Angular velocity cross-coupling constant, single precision floating point format.
<a href="#">tFloat</a>	fltEGain	Back-EMF cross-coupling constant, single precision floating point format.

## 5.4 AMCLIB\_CURRENT\_LOOP\_T\_F16

CurrentLoop configuration structure.

### Source File

```
#include <AMCLIB_CurrentLoop.h>
```

### Compound Type Members

**Table 394. AMCLIB\_CURRENT\_LOOP\_T\_F16 members description**

Type	Name	Description
GFLIB_CONTROLLER_PIAW_R_T_F16	pPIrAWD	D-axis ControllerPIrAW parameters structure.
GFLIB_CONTROLLER_PIAW_R_T_F16	pPIrAWQ	Q-axis ControllerPIrAW parameters structure.
SWLIBS_2Syst_F16 *	pIDQReq	Pointer to the structure with the required current.
SWLIBS_2Syst_F16 *	pIDQFbck	Pointer to the structure with the feedback current.

## 5.5 AMCLIB\_CURRENT\_LOOP\_T\_F32

CurrentLoop configuration structure.

### Source File

```
#include <AMCLIB_CurrentLoop.h>
```

### Compound Type Members

**Table 395. AMCLIB\_CURRENT\_LOOP\_T\_F32 members description**

Type	Name	Description
GFLIB_CONTROLLER_PIAW_R_T_F32	pPIrAWD	D-axis ControllerPIrAW parameters structure.
GFLIB_CONTROLLER_PIAW_R_T_F32	pPIrAWQ	Q-axis ControllerPIrAW parameters structure.
SWLIBS_2Syst_F32 *	pIDQReq	Pointer to the structure with the required current.
SWLIBS_2Syst_F32 *	pIDQFbck	Pointer to the structure with the feedback current.

## 5.6 AMCLIB\_CURRENT\_LOOP\_T\_FLT

CurrentLoop configuration structure.

### Source File

```
#include <AMCLIB_CurrentLoop.h>
```

**Compound Type Members****Table 396. AMCLIB\_CURRENT\_LOOP\_T\_FLT members description**

Type	Name	Description
GFLIB_CONTROLLER_PIAW_R_T_FLT	pPIrAWD	D-axis ControllerPIrAW paremeters structure.
GFLIB_CONTROLLER_PIAW_R_T_FLT	pPIrAWQ	Q-axis ControllerPIrAW paremeters structure.
SWLIBS_2Syst_FLT *	pIDQReq	Pointer to the structure with the required current.
SWLIBS_2Syst_FLT *	pIDQFbck	Pointer to the structure with the feedback current.

**5.7 AMCLIB\_FW\_DEBUG\_T\_F16**

FW configuration structure with debugging information.

**Source File**

```
#include <AMCLIB_FW.h>
```

**Compound Type Members****Table 397. AMCLIB\_FW\_DEBUG\_T\_F16 members description**

Type	Name	Description
GDFLIB_FILTER_MA_T_F16	pFilterFW	Field weakening angle FilterMA paremeters structure.
GFLIB_CONTROLLER_PIAW_P_T_F16	pPIpAWFW	Field weakening angle ControllerPIpAW paremeters structure.
tFrac16 *	pIQFbck	Pointer to the q-axis component of the feedback current in the two-phase rotational orthogonal system (d-q).
tFrac16 *	pUQReq	Pointer to the q-axis component of the required voltage in the two-phase rotational orthogonal system (d-q).
tFrac16 *	pUQLim	Pointer to the q-axis voltage controller upper limit (should point to the <a href="#">AMCLIB_CURRENT_LOOP_T_F16</a> structure element pPIrAWQ.f16UpperLimit). The limit must be a positive value.
tFrac16	f16IQErrSign	FW - Current deviation after sign elimination.
tFrac16	f16IQErr	FW - Current deviation.
tFrac16	f16FWErr	FW - Field weakening feedback control variable.
tFrac16	f16UQErr	FW - Voltage deviation.
tFrac16	f16FWErrFilt	FW - Filtered field weakening feedback control variable.
tFrac16	f16FWAngle	FW - Field weakening angle.

Type	Name	Description
tFrac16	f16FWSin	FW - Q-axis unity current phasor component.
tFrac16	f16FWCos	FW - D-axis unity current phasor component.

## 5.8 AMCLIB\_FW\_DEBUG\_T\_F32

FW configuration structure with debugging information.

### Source File

```
#include <AMCLIB_FW.h>
```

### Compound Type Members

Table 398. AMCLIB\_FW\_DEBUG\_T\_F32 members description

Type	Name	Description
GDFLIB_FILTER_MA_T_F32	pFilterFW	Field weakening angle FilterMA parameters structure.
GFLIB_CONTROLLER_PIAW_P_T_F32	pPIpAWFW	Field weakening angle ControllerPIpAW parameters structure.
tFrac32 *	pIQFbck	Pointer to the q-axis component of the feedback current in the two-phase rotational orthogonal system (d-q).
tFrac32 *	pUQReq	Pointer to the q-axis component of the required voltage in the two-phase rotational orthogonal system (d-q).
tFrac32 *	pUQLim	Pointer to the q-axis voltage controller upper limit. Must be a positive value.
tFrac32	f32IQErrSign	FW - Current deviation after sign elimination.
tFrac32	f32IQErr	FW - Current deviation.
tFrac32	f32FWErr	FW - Field weakening feedback control variable.
tFrac32	f32UQErr	FW - Voltage deviation.
tFrac32	f32FWErrFilt	FW - Filtered field weakening feedback control variable.
tFrac32	f32FWAngle	FW - Field weakening angle.
tFrac32	f32FWSin	FW - Q-axis unity current phasor component.
tFrac32	f32FWCos	FW - D-axis unity current phasor component.

## 5.9 AMCLIB\_FW\_DEBUG\_T\_FLT

FW configuration structure with debugging information.

### Source File

```
#include <AMCLIB_FW.h>
```

**Compound Type Members****Table 399. AMCLIB\_FW\_DEBUG\_T\_FLT members description**

Type	Name	Description
<a href="#">GDFLIB_FILTER_MA_T_FLT</a>	pFilterFW	Field weakening angle FilterMA parameters structure.
<a href="#">GFLIB_CONTROLLER_PIAW_P_T_FLT</a>	pPIpAWFW	Field weakening angle ControllerPlpAW parameters structure.
<a href="#">tFloat *</a>	pIQFbck	Pointer to the q-axis component of the feedback current in the two-phase rotational orthogonal system (d-q).
<a href="#">tFloat *</a>	pUQReq	Pointer to the q-axis component of the required voltage in the two-phase rotational orthogonal system (d-q).
<a href="#">tFloat *</a>	pUQLim	Pointer to the q-axis voltage controller upper limit (should point to the <a href="#">AMCLIB_CURRENT_LOOP_T_FLT</a> structure element pPIrAWQ.fitUpperLimit). The limit must be a positive value.
<a href="#">tFloat</a>	fltUmaxDivImax	Maximal coil voltage divided by the maximal coil current.
<a href="#">tFloat</a>	fltIQErrSign	FW - Current deviation after sign elimination.
<a href="#">tFloat</a>	fltIQErr	FW - Current deviation.
<a href="#">tFloat</a>	fltFWErr	FW - Field weakening feedback control variable.
<a href="#">tFloat</a>	fltUQErr	FW - Voltage deviation.
<a href="#">tFloat</a>	fltFWErrFilt	FW - Filtered field weakening feedback control variable.
<a href="#">tFloat</a>	fltFWAngle	FW - Field weakening angle.
<a href="#">tFloat</a>	fltFWSin	FW - Q-axis unity current phasor component.
<a href="#">tFloat</a>	fltFWCos	FW - D-axis unity current phasor component.

**5.10 AMCLIB\_FW\_SPEED\_LOOP\_DEBUG\_T\_F16**

FWSpeedLoop configuration structure with debugging information.

**Source File**

#include &lt;AMCLIB\_FWSpeedLoop.h&gt;

**Compound Type Members****Table 400. AMCLIB\_FW\_SPEED\_LOOP\_DEBUG\_T\_F16 members description**

Type	Name	Description
<a href="#">GDFLIB_FILTER_MA_T_F16</a>	pFilterW	Velocity FilterMA parameters structure.
<a href="#">GDFLIB_FILTER_MA_T_F16</a>	pFilterFW	Field weakening angle FilterMA parameters structure.

Type	Name	Description
<a href="#">GFLIB_CONTROLLER_PIAW_P_T_F16</a>	pPipAWQ	Q-axis ControllerPipAW parameters structure.
<a href="#">GFLIB_CONTROLLER_PIAW_P_T_F16</a>	pPipAWFW	Field weakening angle ControllerPipAW parameters structure.
<a href="#">GFLIB_RAMP_T_F32</a>	pRamp	Speed ramp parameters structure.
<a href="#">tFrac16 *</a>	pIQFbck	Pointer to the q-axis component of the feedback current in the two-phase rotational orthogonal system (d-q).
<a href="#">tFrac16 *</a>	pUQReq	Pointer to the q-axis component of the required voltage in the two-phase rotational orthogonal system (d-q).
<a href="#">tFrac16 *</a>	pUQLim	Pointer to the q-axis voltage controller upper limit (should point to the <a href="#">AMCLIB_CURRENT_LOOP_T_F16</a> structure element pPirAWQ.f16UpperLimit). The limit must be a positive value.
<a href="#">tFrac16</a>	f16WReqFilt	SpeedLoop - Filtered value of the required angular velocity.
<a href="#">tFrac16</a>	f16WErr	SpeedLoop - Velocity deviation.
<a href="#">tFrac16</a>	f16IDQReqAmp	SpeedLoop - Filtered value of the measured angular velocity.
<a href="#">tFrac16</a>	f16WFbckFilt	SpeedLoop - Required current amplitude.
<a href="#">tFrac16</a>	f16IQErrSign	FW - Current deviation after sign elimination.
<a href="#">tFrac16</a>	f16FWErr	FW - Current deviation.
<a href="#">tFrac16</a>	f16IQErr	FW - Field weakening feedback control variable.
<a href="#">tFrac16</a>	f16UQErr	FW - Voltage deviation.
<a href="#">tFrac16</a>	f16FWErrFilt	FW - Filtered field weakening feedback control variable.
<a href="#">tFrac16</a>	f16FWAngle	FW - Field weakening angle.
<a href="#">tFrac16</a>	f16FWSin	FW - Q-axis unity current phasor component.
<a href="#">tFrac16</a>	f16FWCos	FW - D-axis unity current phasor component.

## 5.11 AMCLIB\_FW\_SPEED\_LOOP\_DEBUG\_T\_F32

FWSpeedLoop configuration structure with debugging information.

### Source File

```
#include <AMCLIB_FWSpeedLoop.h>
```

### Compound Type Members

Table 401. AMCLIB\_FW\_SPEED\_LOOP\_DEBUG\_T\_F32 members description

Type	Name	Description
<a href="#">GDFLIB_FILTER_MA_T_F32</a>	pFilterW	Velocity FilterMA parameters structure.

Type	Name	Description
<a href="#">GDFLIB_FILTER_MA_T_F32</a>	pFilterFW	Field weakening angle FilterMA parameters structure.
<a href="#">GFLIB_CONTROLLER_PIAW_P_T_F32</a>	pPIpAWQ	Q-axis ControllerPIpAW parameters structure.
<a href="#">GFLIB_CONTROLLER_PIAW_P_T_F32</a>	pPIpAWFW	Field weakening angle ControllerPIpAW parameters structure.
<a href="#">GFLIB_RAMP_T_F32</a>	pRamp	Speed ramp parameters structure.
<a href="#">tFrac32</a> *	pIQFbck	Pointer to the q-axis component of the feedback current in the two-phase rotational orthogonal system (d-q).
<a href="#">tFrac32</a> *	pUQReq	Pointer to the q-axis component of the required voltage in the two-phase rotational orthogonal system (d-q).
<a href="#">tFrac32</a> *	pUQLim	Pointer to the q-axis voltage controller upper limit (should point to the <a href="#">AMCLIB_CURRENT_LOOP_T_F32</a> structure element pPIrAWQ.f32UpperLimit). The limit must be a positive value.
<a href="#">tFrac32</a>	f32WReqFilt	SpeedLoop - Filtered value of the required angular velocity.
<a href="#">tFrac32</a>	f32WErr	SpeedLoop - Velocity deviation.
<a href="#">tFrac32</a>	f32WFbckFilt	SpeedLoop - Filtered value of the measured angular velocity.
<a href="#">tFrac32</a>	f32IDQReqAmp	SpeedLoop - Required current amplitude.
<a href="#">tFrac32</a>	f32IQErrSign	FW - Current deviation after sign elimination.
<a href="#">tFrac32</a>	f32FWErr	FW - Current deviation.
<a href="#">tFrac32</a>	f32IQErr	FW - Field weakening feedback control variable.
<a href="#">tFrac32</a>	f32UQErr	FW - Voltage deviation.
<a href="#">tFrac32</a>	f32FWErrFilt	FW - Filtered field weakening feedback control variable.
<a href="#">tFrac32</a>	f32FWAngle	FW - Field weakening angle.
<a href="#">tFrac32</a>	f32FWSin	FW - Q-axis unity current phasor component.
<a href="#">tFrac32</a>	f32FWCos	FW - D-axis unity current phasor component.

## 5.12 AMCLIB\_FW\_SPEED\_LOOP\_DEBUG\_T\_FLT

FWSpeedLoop configuration structure with debugging information.

### Source File

```
#include <AMCLIB_FWSpeedLoop.h>
```

## Compound Type Members

Table 402. AMCLIB\_FW\_SPEED\_LOOP\_DEBUG\_T\_FLT members description

Type	Name	Description
GDFLIB_FILTER_MA_T_FLT	pFilterW	Velocity FilterMA paremeters structure.
GDFLIB_FILTER_MA_T_FLT	pFilterFW	Field weakening angle FilterMA paremeters structure.
GFLIB_CONTROLLER_PIAW_P_T_FLT	pPIpAWQ	Q-axis ControllerPIpAW paremeters structure.
GFLIB_CONTROLLER_PIAW_P_T_FLT	pPIpAWFW	Field weakening angle ControllerPIpAW paremeters structure.
GFLIB_RAMP_T_FLT	pRamp	Speed ramp paremeters structure.
tFloat *	pIQFbck	Pointer to the q-axis component of the feedback current in the two-phase rotational orthogonal system (d-q).
tFloat *	pUQReq	Pointer to the q-axis component of the required voltage in the two-phase rotational orthogonal system (d-q).
tFloat *	pUQLim	Pointer to the q-axis voltage controller upper limit (should point to the <a href="#">AMCLIB_CURRENT_LOOP_T_FLT</a> structure element pPIrAWQ.fitUpperLimit). The limit must be a positive value.
tFloat	fltUmaxDivImax	Maximal coil voltage divided by the maximal coil current.
tFloat	fltWReqFilt	SpeedLoop - Filtered value of the required angular velocity.
tFloat	fltWErr	SpeedLoop - Velocity deviation.
tFloat	fltIDQReqAmp	SpeedLoop - Filtered value of the measured angular velocity.
tFloat	fltWFbckFilt	SpeedLoop - Required current amplitude.
tFloat	fltIQErrSign	FW - Current deviation after sign elimination.
tFloat	fltFWErr	FW - Current deviation.
tFloat	fltIQErr	FW - Field weakening feedback control variable.
tFloat	fltUQErr	FW - Voltage deviation.
tFloat	fltFWErrFilt	FW - Filtered field weakening feedback control variable.
tFloat	fltFWAngle	FW - Field weakening angle.
tFloat	fltFWSin	FW - Q-axis unity current phasor component.
tFloat	fltFWCos	FW - D-axis unity current phasor component.

## 5.13 AMCLIB\_FW\_SPEED\_LOOP\_T\_F16

FWSpeedLoop configuration structure.

**Source File**

```
#include <AMCLIB_FWSpeedLoop.h>
```

**Compound Type Members****Table 403. AMCLIB\_FW\_SPEED\_LOOP\_T\_F16 members description**

Type	Name	Description
<a href="#">GDFLIB_FILTER_MA_T_F16</a>	pFilterW	Velocity FilterMA paremeters structure.
<a href="#">GDFLIB_FILTER_MA_T_F16</a>	pFilterFW	Field weakening angle FilterMA paremeters structure.
<a href="#">GFLIB_CONTROLLER_PIAW_P_T_F16</a>	pPIpAWQ	Q-axis ControllerPIpAW paremeters structure.
<a href="#">GFLIB_CONTROLLER_PIAW_P_T_F16</a>	pPIpAWFW	Field weakening angle ControllerPIpAW paremeters structure.
<a href="#">GFLIB_RAMP_T_F32</a>	pRamp	Speed ramp paremeters structure.
<a href="#">tFrac16 *</a>	pIQFbck	Pointer to the q-axis component of the feedback current in the two-phase rotational orthogonal system (d-q).
<a href="#">tFrac16 *</a>	pUQReq	Pointer to the q-axis component of the required voltage in the two-phase rotational orthogonal system (d-q).
<a href="#">tFrac16 *</a>	pUQLim	Pointer to the q-axis voltage controller upper limit (should point to the <a href="#">AMCLIB_CURRENT_LOOP_T_F16</a> structure element pPIrAWQ.f16UpperLimit). The limit must be a positive value.

**5.14 AMCLIB\_FW\_SPEED\_LOOP\_T\_F32**

FWSpeedLoop configuration structure.

**Source File**

```
#include <AMCLIB_FWSpeedLoop.h>
```

**Compound Type Members****Table 404. AMCLIB\_FW\_SPEED\_LOOP\_T\_F32 members description**

Type	Name	Description
<a href="#">GDFLIB_FILTER_MA_T_F32</a>	pFilterW	Velocity FilterMA paremeters structure.
<a href="#">GDFLIB_FILTER_MA_T_F32</a>	pFilterFW	Field weakening angle FilterMA paremeters structure.
<a href="#">GFLIB_CONTROLLER_PIAW_P_T_F32</a>	pPIpAWQ	Q-axis ControllerPIpAW paremeters structure.
<a href="#">GFLIB_CONTROLLER_PIAW_P_T_F32</a>	pPIpAWFW	Field weakening angle ControllerPIpAW paremeters structure.
<a href="#">GFLIB_RAMP_T_F32</a>	pRamp	Speed ramp paremeters structure.

Type	Name	Description
tFrac32 *	pIQFbck	Pointer to the q-axis component of the feedback current in the two-phase rotational orthogonal system (d-q).
tFrac32 *	pUQReq	Pointer to the q-axis component of the required voltage in the two-phase rotational orthogonal system (d-q).
tFrac32 *	pUQLim	Pointer to the q-axis voltage controller upper limit (should point to the <a href="#">AMCLIB_CURRENT_LOOP_T_F32</a> structure element pPIrAWQ.f32UpperLimit). The limit must be a positive value.

## 5.15 AMCLIB\_FW\_SPEED\_LOOP\_T\_FLT

FWSpeedLoop configuration structure.

### Source File

```
#include <AMCLIB_FWSpeedLoop.h>
```

### Compound Type Members

Table 405. AMCLIB\_FW\_SPEED\_LOOP\_T\_FLT members description

Type	Name	Description
<a href="#">GDFLIB_FILTER_MA_T_FLT</a>	pFilterW	Velocity FilterMA paremeters structure.
<a href="#">GDFLIB_FILTER_MA_T_FLT</a>	pFilterFW	Field weakening angle FilterMA paremeters structure.
<a href="#">GFLIB_CONTROLLER_PIAW_P_T_FLT</a>	pPIpAWQ	Q-axis ControllerPIpAW paremeters structure.
<a href="#">GFLIB_CONTROLLER_PIAW_P_T_FLT</a>	pPIpAWFW	Field weakening angle ControllerPIpAW paremeters structure.
<a href="#">GFLIB_RAMP_T_FLT</a>	pRamp	Speed ramp paremeters structure.
tFloat *	pIQFbck	Pointer to the q-axis component of the feedback current in the two-phase rotational orthogonal system (d-q).
tFloat *	pUQReq	Pointer to the q-axis component of the required voltage in the two-phase rotational orthogonal system (d-q).
tFloat *	pUQLim	Pointer to the q-axis voltage controller upper limit (should point to the <a href="#">AMCLIB_CURRENT_LOOP_T_FLT</a> structure element pPIrAWQ.fltUpperLimit). The limit must be a positive value.
tFloat	fltUmaxDivImax	Maximal coil voltage divided by the maximal coil current.

## 5.16 AMCLIB\_FW\_T\_F16

FW configuration structure.

### Source File

```
#include <AMCLIB_FW.h>
```

### Compound Type Members

**Table 406. AMCLIB\_FW\_T\_F16 members description**

Type	Name	Description
<a href="#">GDFLIB_FILTER_MA_T_F16</a>	pFilterFW	Field weakening angle FilterMA parameters structure.
<a href="#">GFLIB_CONTROLLER_PIAW_P_T_F16</a>	pPipAWFW	Field weakening angle ControllerPipAW parameters structure.
<a href="#">tFrac16 *</a>	pIQFbck	Pointer to the q-axis component of the feedback current in the two-phase rotational orthogonal system (d-q).
<a href="#">tFrac16 *</a>	pUQReq	Pointer to the q-axis component of the required voltage in the two-phase rotational orthogonal system (d-q).
<a href="#">tFrac16 *</a>	pUQLim	Pointer to the q-axis voltage controller upper limit. Must be a positive value.

## 5.17 AMCLIB\_FW\_T\_F32

FW configuration structure.

### Source File

```
#include <AMCLIB_FW.h>
```

### Compound Type Members

**Table 407. AMCLIB\_FW\_T\_F32 members description**

Type	Name	Description
<a href="#">GDFLIB_FILTER_MA_T_F32</a>	pFilterFW	Field weakening angle FilterMA parameters structure.
<a href="#">GFLIB_CONTROLLER_PIAW_P_T_F32</a>	pPipAWFW	Field weakening angle ControllerPipAW parameters structure.
<a href="#">tFrac32 *</a>	pIQFbck	Pointer to the q-axis component of the feedback current in the two-phase rotational orthogonal system (d-q).
<a href="#">tFrac32 *</a>	pUQReq	Pointer to the q-axis component of the required voltage in the two-phase rotational orthogonal system (d-q).
<a href="#">tFrac32 *</a>	pUQLim	Pointer to the q-axis voltage controller upper limit. Must be a positive value.

## 5.18 AMCLIB\_FW\_T\_FLT

FW configuration structure.

### Source File

```
#include <AMCLIB_FW.h>
```

### Compound Type Members

**Table 408. AMCLIB\_FW\_T\_FLT members description**

Type	Name	Description
<a href="#">GDFLIB_FILTER_MA_T_FLT</a>	pFilterFW	Field weakening angle FilterMA paremeters structure.
<a href="#">GFLIB_CONTROLLER_PIAW_P_T_FLT</a>	pPipAWFW	Field weakening angle ControllerPipAW paremeters structure.
<a href="#">tFloat *</a>	pIQFbck	Pointer to the q-axis component of the feedback current in the two-phase rotational orthogonal system (d-q).
<a href="#">tFloat *</a>	pUQReq	Pointer to the q-axis component of the required voltage in the two-phase rotational orthogonal system (d-q).
<a href="#">tFloat *</a>	pUQLim	Pointer to the q-axis voltage controller upper limit (should point to the <a href="#">AMCLIB_CURRENT_LOOP_T_FLT</a> structure element pPirAWQ.fitUpperLimit). The limit must be a positive value.
<a href="#">tFloat</a>	fltUmaxDivImax	Maximal coil voltage divided by the maximal coil current.

## 5.19 AMCLIB\_SPEED\_LOOP\_DEBUG\_T\_F16

SpeedLoop configuration structure with debugging information.

### Source File

```
#include <AMCLIB_SpeedLoop.h>
```

### Compound Type Members

**Table 409. AMCLIB\_SPEED\_LOOP\_DEBUG\_T\_F16 members description**

Type	Name	Description
<a href="#">GDFLIB_FILTER_MA_T_F16</a>	pFilterW	Velocity FilterMA paremeters structure.
<a href="#">GFLIB_CONTROLLER_PIAW_P_T_F16</a>	pPipAWQ	Velocity ControllerPipAW paremeters structure.
<a href="#">GFLIB_RAMP_T_F32</a>	pRamp	Speed ramp paremeters structure.
<a href="#">tFrac16</a>	f16WReqFilt	SpeedLoop - Filtered value of the required angular velocity.
<a href="#">tFrac16</a>	f16WErr	SpeedLoop - Velocity deviation.

Type	Name	Description
tFrac16	f16WFbckFilt	SpeedLoop - Filtered value of the measured angular velocity.

## 5.20 AMCLIB\_SPEED\_LOOP\_DEBUG\_T\_F32

SpeedLoop configuration structure with debugging information.

### Source File

```
#include <AMCLIB_SpeedLoop.h>
```

### Compound Type Members

Table 410. AMCLIB\_SPEED\_LOOP\_DEBUG\_T\_F32 members description

Type	Name	Description
GDFLIB_FILTER_MA_T_F32	pFilterW	Velocity FilterMA paremeters structure.
GFLIB_CONTROLLER_PIAW_P_T_F32	pPlpAWQ	Velocity ControllerPlpAW paremeters structure.
GFLIB_RAMP_T_F32	pRamp	Speed ramp paremeters structure.
tFrac32	f32WReqFilt	SpeedLoop - Filtered value of the required angular velocity.
tFrac32	f32WErr	SpeedLoop - Velocity deviation.
tFrac32	f32WFbckFilt	SpeedLoop - Filtered value of the measured angular velocity.

## 5.21 AMCLIB\_SPEED\_LOOP\_DEBUG\_T\_FLT

SpeedLoop configuration structure with debugging information.

### Source File

```
#include <AMCLIB_SpeedLoop.h>
```

### Compound Type Members

Table 411. AMCLIB\_SPEED\_LOOP\_DEBUG\_T\_FLT members description

Type	Name	Description
GDFLIB_FILTER_MA_T_FLT	pFilterW	Velocity FilterMA paremeters structure.
GFLIB_CONTROLLER_PIAW_P_T_FLT	pPlpAWQ	Velocity ControllerPlpAW paremeters structure.
GFLIB_RAMP_T_FLT	pRamp	Speed ramp paremeters structure.
tFloat	fltWReqFilt	SpeedLoop - Filtered value of the required angular velocity.
tFloat	fltWErr	SpeedLoop - Velocity deviation.
tFloat	fltWFbckFilt	SpeedLoop - Filtered value of the measured angular velocity.

## 5.22 AMCLIB\_SPEED\_LOOP\_T\_F16

SpeedLoop configuration structure.

### Source File

```
#include <AMCLIB_SpeedLoop.h>
```

### Compound Type Members

**Table 412. AMCLIB\_SPEED\_LOOP\_T\_F16 members description**

Type	Name	Description
<a href="#">GDFLIB_FILTER_MA_T_F16</a>	pFilterW	Velocity FilterMA paremeters structure.
<a href="#">GFLIB_CONTROLLER_PIAW_P_T_F16</a>	pPipAWQ	Velocity ControllerPipAW paremeters structure.
<a href="#">GFLIB_RAMP_T_F32</a>	pRamp	Speed ramp paremeters structure.

## 5.23 AMCLIB\_SPEED\_LOOP\_T\_F32

SpeedLoop configuration structure.

### Source File

```
#include <AMCLIB_SpeedLoop.h>
```

### Compound Type Members

**Table 413. AMCLIB\_SPEED\_LOOP\_T\_F32 members description**

Type	Name	Description
<a href="#">GDFLIB_FILTER_MA_T_F32</a>	pFilterW	Velocity FilterMA paremeters structure.
<a href="#">GFLIB_CONTROLLER_PIAW_P_T_F32</a>	pPipAWQ	Velocity ControllerPipAW paremeters structure.
<a href="#">GFLIB_RAMP_T_F32</a>	pRamp	Speed ramp paremeters structure.

## 5.24 AMCLIB\_SPEED\_LOOP\_T\_FLT

SpeedLoop configuration structure.

### Source File

```
#include <AMCLIB_SpeedLoop.h>
```

### Compound Type Members

**Table 414. AMCLIB\_SPEED\_LOOP\_T\_FLT members description**

Type	Name	Description
<a href="#">GDFLIB_FILTER_MA_T_FLT</a>	pFilterW	Velocity FilterMA paremeters structure.
<a href="#">GFLIB_CONTROLLER_PIAW_P_T_FLT</a>	pPipAWQ	Velocity ControllerPipAW paremeters structure.

Type	Name	Description
<a href="#">GFLIB_RAMP_T_FLT</a>	pRamp	Speed ramp parameters structure.

## 5.25 AMCLIB\_TRACK\_OBSRV\_T\_F16

Structure containing the estimated position, estimated velocity and the algorithm parameters.

### Source File

```
#include <AMCLIB_TrackObsrv.h>
```

### Compound Type Members

Table 415. AMCLIB\_TRACK\_OBSRV\_T\_F16 members description

Type	Name	Description
<a href="#">GFLIB_CONTROLLER_PIAW_R_T_F16</a>	pParamPI	Observer PIrAW controller parameters.
<a href="#">GFLIB_INTEGRATOR_TR_T_F16</a>	pParamInteg	Observer integrator parameters.

## 5.26 AMCLIB\_TRACK\_OBSRV\_T\_F32

Structure containing the estimated position, estimated velocity and the algorithm parameters.

### Source File

```
#include <AMCLIB_TrackObsrv.h>
```

### Compound Type Members

Table 416. AMCLIB\_TRACK\_OBSRV\_T\_F32 members description

Type	Name	Description
<a href="#">GFLIB_CONTROLLER_PIAW_R_T_F32</a>	pParamPI	Observer PIrAW controller parameters.
<a href="#">GFLIB_INTEGRATOR_TR_T_F32</a>	pParamInteg	Observer integrator parameters.

## 5.27 AMCLIB\_TRACK\_OBSRV\_T\_FLT

Structure containing the estimated position, estimated velocity and the algorithm parameters.

### Source File

```
#include <AMCLIB_TrackObsrv.h>
```

**Compound Type Members****Table 417. AMCLIB\_TRACK\_OBSRV\_T\_FLT members description**

Type	Name	Description
<a href="#">GFLIB_CONTROLLER_PIAW_R_T_FLT</a>	pParamPI	Observer PIrAW controller parameters.
<a href="#">GFLIB_INTEGRATOR_TR_T_FLT</a>	pParamInteg	Observer integrator parameters.

**5.28 AMCLIB\_WINDMILLING\_T\_F16**

Observer configuration structure.

**Source File**

```
#include <AMCLIB_Windmilling.h>
```

**Compound Type Members****Table 418. AMCLIB\_WINDMILLING\_T\_F16 members description**

Type	Name	Description
<a href="#">tFrac16</a>	f16ADCComp	A/D converter error compensation constant. This parameter is initialized by <a href="#">AMCLIB_WindmillingInit_F16</a> based on user's input.
<a href="#">AMCLIB_TRACK_OBSRV_T_F16</a>	pParamATO	PI controller and integrator parameters and state. These parameters must be configured by the user before the first call of <a href="#">AMCLIB_Windmilling_F16</a> .
<a href="#">GDFLIB_FILTER_MA_T_F16</a>	pDetMA	PLL lock detector internal state. Do not change. This sub-structure and all its parameters are automatically initialized by <a href="#">AMCLIB_WindmillingInit_F16</a> .
<a href="#">tU8</a>	u8DetNoiseCnt	PLL lock detector internal state. Do not change. This parameter is automatically initialized by <a href="#">AMCLIB_WindmillingInit_F16</a> .
<a href="#">tU8</a>	u8DetSpin	PLL lock detector internal state. Do not change. This parameter is automatically initialized by <a href="#">AMCLIB_WindmillingInit_F16</a> .
<a href="#">tU16</a>	u16DetHystCnt	PLL lock detector internal state. Do not change. This parameter is automatically initialized by <a href="#">AMCLIB_WindmillingInit_F16</a> .
<a href="#">tU16</a>	u16DetStopCnt	PLL lock detector internal state. Do not change. This parameter is automatically initialized by <a href="#">AMCLIB_WindmillingInit_F16</a> .

**5.29 AMCLIB\_WINDMILLING\_T\_F32**

Observer configuration structure.

**Source File**

```
#include <AMCLIB_Windmilling.h>
```

**Compound Type Members****Table 419.** AMCLIB\_WINDMILLING\_T\_F32 members description

Type	Name	Description
tFrac32	f32ADCComp	A/D converter error compensation constant. This parameter is initialized by <a href="#">AMCLIB_WindmillingInit_F32</a> based on user's input.
<a href="#">AMCLIB_TRACK_OBSRV_T_F32</a>	pParamATO	PI controller and integrator parameters and state. These parameters must be configured by the user before the first call of <a href="#">AMCLIB_Windmilling_F32</a> .
<a href="#">GDFLIB_FILTER_MA_T_F32</a>	pDetMA	PLL lock detector internal state. Do not change. This sub-structure and all its parameters are automatically initialized by <a href="#">AMCLIB_WindmillingInit_F32</a> .
tU8	u8DetNoiseCnt	PLL lock detector internal state. Do not change. This parameter is automatically initialized by <a href="#">AMCLIB_WindmillingInit_F32</a> .
tU8	u8DetSpin	PLL lock detector internal state. Do not change. This parameter is automatically initialized by <a href="#">AMCLIB_WindmillingInit_F32</a> .
tU16	u16DetHystCnt	PLL lock detector internal state. Do not change. This parameter is automatically initialized by <a href="#">AMCLIB_WindmillingInit_F32</a> .
tU16	u16DetStopCnt	PLL lock detector internal state. Do not change. This parameter is automatically initialized by <a href="#">AMCLIB_WindmillingInit_F32</a> .

**5.30 AMCLIB\_WINDMILLING\_T\_FLT**

Observer configuration structure.

**Source File**

#include &lt;AMCLIB\_Windmilling.h&gt;

**Compound Type Members****Table 420.** AMCLIB\_WINDMILLING\_T\_FLT members description

Type	Name	Description
tFloat	fltADCComp	A/D converter error compensation constant. This parameter is initialized by <a href="#">AMCLIB_WindmillingInit_FLT</a> based on user's input.
<a href="#">AMCLIB_TRACK_OBSRV_T_FLT</a>	pParamATO	PI controller and integrator parameters and state. These parameters must be configured by the user before the first call of <a href="#">AMCLIB_Windmilling_FLT</a> .
<a href="#">GDFLIB_FILTER_MA_T_FLT</a>	pDetMA	PLL lock detector internal state. Do not change. This sub-structure and all its parameters are automatically initialized by <a href="#">AMCLIB_WindmillingInit_FLT</a> .

Type	Name	Description
tU8	u8DetNoiseCnt	PLL lock detector internal state. Do not change. This parameter is automatically initialized by <a href="#">AMCLIB_WindmillingInit_FLT</a> .
tU8	u8DetSpin	PLL lock detector internal state. Do not change. This parameter is automatically initialized by <a href="#">AMCLIB_WindmillingInit_FLT</a> .
tU16	u16DetHystCnt	PLL lock detector internal state. Do not change. This parameter is automatically initialized by <a href="#">AMCLIB_WindmillingInit_FLT</a> .
tU16	u16DetStopCnt	PLL lock detector internal state. Do not change. This parameter is automatically initialized by <a href="#">AMCLIB_WindmillingInit_FLT</a> .

## 5.31 GDFLIB\_FILTER\_IIR1\_COEFF\_T\_F16

Sub-structure containing filter coefficients.

### Source File

```
#include <GDFLIB_FilterIIR1.h>
```

### Compound Type Members

Table 421. GDFLIB\_FILTER\_IIR1\_COEFF\_T\_F16 members description

Type	Name	Description
tFrac16	f16B0	B0 coefficient of an IIR1 filter, fractional format normalized to fit into (- 2 <sup>15</sup> , 2 <sup>15</sup> -1).
tFrac16	f16B1	B1 coefficient of an IIR1 filter, fractional format normalized to fit into (- 2 <sup>15</sup> , 2 <sup>15</sup> -1).
tFrac16	f16A1	A1 coefficient of an IIR1 filter, fractional format normalized to fit into (- 2 <sup>15</sup> , 2 <sup>15</sup> -1).

## 5.32 GDFLIB\_FILTER\_IIR1\_COEFF\_T\_F32

Sub-structure containing filter coefficients.

### Source File

```
#include <GDFLIB_FilterIIR1.h>
```

### Compound Type Members

Table 422. GDFLIB\_FILTER\_IIR1\_COEFF\_T\_F32 members description

Type	Name	Description
tFrac32	f32B0	B0 coefficient of an IIR1 filter, fractional format normalized to fit into (- 2 <sup>31</sup> , 2 <sup>31</sup> -1).
tFrac32	f32B1	B1 coefficient of an IIR1 filter, fractional format normalized to fit into (- 2 <sup>31</sup> , 2 <sup>31</sup> -1).

Type	Name	Description
tFrac32	f32A1	A1 coefficient of an IIR1 filter, fractional format normalized to fit into $(-2^{31}, 2^{31}-1)$ .

### 5.33 GDFLIB\_FILTER\_IIR1\_COEFF\_T\_FLT

Sub-structure containing filter coefficients.

#### Source File

```
#include <GDFLIB_FilterIIR1.h>
```

#### Compound Type Members

**Table 423. GDFLIB\_FILTER\_IIR1\_COEFF\_T\_FLT members description**

Type	Name	Description
tFloat	fltB0	B0 coefficient of an IIR1 filter. The parameter is in full range single precision floating point format.
tFloat	fltB1	B1 coefficient of an IIR1 filter. The parameter is in full range single precision floating point format.
tFloat	fltA1	A1 coefficient of an IIR1 filter. The parameter is in full range single precision floating point format.

### 5.34 GDFLIB\_FILTER\_IIR1\_T\_F16

Structure containing filter buffer and coefficients.

#### Source File

```
#include <GDFLIB_FilterIIR1.h>
```

#### Compound Type Members

**Table 424. GDFLIB\_FILTER\_IIR1\_T\_F16 members description**

Type	Name	Description
GDFLIB_FILTER_IIR1_COEFF_T_F16	trFiltCoeff	Sub-structure containing filter coefficients.
tFrac16	f16FiltBufferX	Input buffer of an IIR1 filter, fractional format normalized to fit into $(-2^{15}, 2^{15}-1)$ .
tFrac32	f32FiltBufferY	Internal accumulator buffer, fractional format normalized to fit into $(-2^{15}, 2^{15}-1)$ .

### 5.35 GDFLIB\_FILTER\_IIR1\_T\_F32

Structure containing filter buffer and coefficients.

**Source File**

```
#include <GDFLIB_FilterIIR1.h>
```

**Compound Type Members****Table 425.** GDFLIB\_FILTER\_IIR1\_T\_F32 members description

Type	Name	Description
GDFLIB_FILTER_IIR1_COEFF_T_F32	trFiltCoeff	Sub-structure containing filter coefficients.
tFrac32	f32FiltBufferX	Input buffer of an IIR1 filter, fractional format normalized to fit into (- 2 <sup>31</sup> , 2 <sup>31</sup> -1).
tFrac32	f32FiltBufferY	Internal accumulator buffer, fractional format normalized to fit into (- 2 <sup>31</sup> , 2 <sup>31</sup> -1).

**5.36 GDFLIB\_FILTER\_IIR1\_T\_FLT**

Structure containing filter buffer and coefficients.

**Source File**

```
#include <GDFLIB_FilterIIR1.h>
```

**Compound Type Members****Table 426.** GDFLIB\_FILTER\_IIR1\_T\_FLT members description

Type	Name	Description
GDFLIB_FILTER_IIR1_COEFF_T_FLT	trFiltCoeff	Sub-structure containing filter coefficients.
tFloat	fltFiltBufferX	Input buffer of an IIR1 filter. The input values are in full range single precision floating point format.
tFloat	fltFiltBufferY	Internal accumulator buffer. The values are in full range single precision floating point format.

**5.37 GDFLIB\_FILTER\_IIR2\_COEFF\_T\_F16**

Sub-structure containing filter coefficients.

**Source File**

```
#include <GDFLIB_FilterIIR2.h>
```

**Compound Type Members****Table 427.** GDFLIB\_FILTER\_IIR2\_COEFF\_T\_F16 members description

Type	Name	Description
tFrac16	f16B0	B0 coefficient of an IIR2 filter, fractional format normalized to fit into (- 2 <sup>15</sup> , 2 <sup>15</sup> -1).

Type	Name	Description
tFrac16	f16B1	B1 coefficient of an IIR2 filter, fractional format normalized to fit into (- $2^{15}$ , $2^{15}-1$ ).
tFrac16	f16B2	B2 coefficient of an IIR2 filter, fractional format normalized to fit into (- $2^{15}$ , $2^{15}-1$ ).
tFrac16	f16A1	A1 coefficient of an IIR2 filter, fractional format normalized to fit into (- $2^{15}$ , $2^{15}-1$ ).
tFrac16	f16A2	A2 coefficient of an IIR2 filter, fractional format normalized to fit into (- $2^{15}$ , $2^{15}-1$ ).

## 5.38 GDFLIB\_FILTER\_IIR2\_COEFF\_T\_F32

Sub-structure containing filter coefficients.

### Source File

```
#include <GDFLIB_FilterIIR2.h>
```

### Compound Type Members

**Table 428. GDFLIB\_FILTER\_IIR2\_COEFF\_T\_F32 members description**

Type	Name	Description
tFrac32	f32B0	B0 coefficient of an IIR2 filter, fractional format normalized to fit into (- $2^{31}$ , $2^{31}-1$ ).
tFrac32	f32B1	B1 coefficient of an IIR2 filter, fractional format normalized to fit into (- $2^{31}$ , $2^{31}-1$ ).
tFrac32	f32B2	B2 coefficient of an IIR2 filter, fractional format normalized to fit into (- $2^{31}$ , $2^{31}-1$ ).
tFrac32	f32A1	A1 coefficient of an IIR2 filter, fractional format normalized to fit into (- $2^{31}$ , $2^{31}-1$ ).
tFrac32	f32A2	A2 coefficient of an IIR2 filter, fractional format normalized to fit into (- $2^{31}$ , $2^{31}-1$ ).

## 5.39 GDFLIB\_FILTER\_IIR2\_COEFF\_T\_FLT

Sub-structure containing filter coefficients.

### Source File

```
#include <GDFLIB_FilterIIR2.h>
```

### Compound Type Members

**Table 429. GDFLIB\_FILTER\_IIR2\_COEFF\_T\_FLT members description**

Type	Name	Description
tFloat	fltB0	B0 coefficient of an IIR2 filter. The parameter is in full range single precision floating point format.

Type	Name	Description
tFloat	fltB1	B1 coefficient of an IIR2 filter. The parameter is in full range single precision floating point format.
tFloat	fltB2	B2 coefficient of an IIR2 filter. The parameter is in full range single precision floating point format.
tFloat	fltA1	A1 coefficient of an IIR2 filter. The parameter is in full range single precision floating point format.
tFloat	fltA2	A2 coefficient of an IIR2 filter. The parameter is in full range single precision floating point format.

## 5.40 GDFLIB\_FILTER\_IIR2\_T\_F16

Structure containing filter buffer and coefficients.

### Source File

```
#include <GDFLIB_FilterIIR2.h>
```

### Compound Type Members

**Table 430. GDFLIB\_FILTER\_IIR2\_T\_F16 members description**

Type	Name	Description
GDFLIB_FILTER_IIR2_COEFF_T_F16	trFiltCoeff	Sub-structure containing filter coefficients.
tFrac16	f16FiltBufferX	Input buffer of an IIR2 filter, fractional format normalized to fit into (- 2 <sup>15</sup> , 2 <sup>15</sup> -1).
tFrac32	f32FiltBufferY	Internal accumulator buffer, fractional format normalized to fit into (- 2 <sup>15</sup> , 2 <sup>15</sup> -1).

## 5.41 GDFLIB\_FILTER\_IIR2\_T\_F32

Structure containing filter buffer and coefficients.

### Source File

```
#include <GDFLIB_FilterIIR2.h>
```

### Compound Type Members

**Table 431. GDFLIB\_FILTER\_IIR2\_T\_F32 members description**

Type	Name	Description
GDFLIB_FILTER_IIR2_COEFF_T_F32	trFiltCoeff	Sub-structure containing filter coefficients.
tFrac32	f32FiltBufferX	Input buffer of an IIR2 filter, fractional format normalized to fit into (- 2 <sup>31</sup> , 2 <sup>31</sup> -1).
tFrac32	f32FiltBufferY	Internal accumulator buffer, fractional format normalized to fit into (- 2 <sup>31</sup> , 2 <sup>31</sup> -1).

## 5.42 GDFLIB\_FILTER\_IIR2\_T\_FLT

Structure containing filter buffer and coefficients.

### Source File

```
#include <GDFLIB_FilterIIR2.h>
```

### Compound Type Members

**Table 432. GDFLIB\_FILTER\_IIR2\_T\_FLT members description**

Type	Name	Description
GDFLIB_FILTER_IIR2_COEFF_T_FLT	trFiltCoeff	Sub-structure containing filter coefficients.
tFloat	fltFiltBufferX	Input buffer of an IIR2 filter. The input values are in full range single precision floating point format.
tFloat	fltFiltBufferY	Internal accumulator buffer. The values are in full range single precision floating point format.

## 5.43 GDFLIB\_FILTER\_MA\_T\_F16

Structure containing filter buffer and coefficients.

### Source File

```
#include <GDFLIB_FilterMA.h>
```

### Compound Type Members

**Table 433. GDFLIB\_FILTER\_MA\_T\_F16 members description**

Type	Name	Description
tFrac32	f32Acc	State variable - filter accumulator.
tU16	u16NSamples	Recalculated smoothing factor [0, 15].

## 5.44 GDFLIB\_FILTER\_MA\_T\_F32

Structure containing filter buffer and coefficients.

### Source File

```
#include <GDFLIB_FilterMA.h>
```

### Compound Type Members

**Table 434. GDFLIB\_FILTER\_MA\_T\_F32 members description**

Type	Name	Description
tFrac32	f32Acc	State variable - filter accumulator.
tU16	u16NSamples	Recalculated smoothing factor [0, 31].

## 5.45 GDFLIB\_FILTER\_MA\_T\_FLT

Structure containing filter buffer and coefficients.

### Source File

```
#include <GDFLIB_FilterMA.h>
```

### Compound Type Members

**Table 435. GDFLIB\_FILTER\_MA\_T\_FLT members description**

Type	Name	Description
<a href="#">tFloat</a>	fltAcc	State variable - filter accumulator.
<a href="#">tFloat</a>	fltLambda	Smoothing factor [0, 1].

## 5.46 GDFLIB\_FILTERFIR\_PARAM\_T\_F16

Structure containing parameters of the filter.

### Source File

```
#include <GDFLIB_FilterFIR.h>
```

### Compound Type Members

**Table 436. GDFLIB\_FILTERFIR\_PARAM\_T\_F16 members description**

Type	Name	Description
<a href="#">tU16</a>	u16Order	FIR filter order, must be in the interval [1, 32767].
const <a href="#">tFrac16</a> *	pCoefBuf	FIR filter coefficients buffer. The array stores (u16Order + 1) filter coefficients. The array must be aligned to a 32-bit boundary and there must be 2 readable bytes after the last element of the array.

## 5.47 GDFLIB\_FILTERFIR\_PARAM\_T\_F32

Structure containing parameters of the filter.

### Source File

```
#include <GDFLIB_FilterFIR.h>
```

### Compound Type Members

**Table 437. GDFLIB\_FILTERFIR\_PARAM\_T\_F32 members description**

Type	Name	Description
<a href="#">tU32</a>	u32Order	FIR filter order, must be in the interval [1, 32767].
const <a href="#">tFrac32</a> *	pCoefBuf	FIR filter coefficients buffer. The array stores (u32Order + 1) filter coefficients.

## 5.48 GDFLIB\_FILTERFIR\_PARAM\_T\_FLT

Structure containing parameters of the filter.

### Source File

```
#include <GDFLIB_FilterFIR.h>
```

### Compound Type Members

**Table 438. GDFLIB\_FILTERFIR\_PARAM\_T\_FLT members description**

Type	Name	Description
tU32	u32Order	FIR filter order, must be in the interval [1, 32767].
const tFloat *	pCoefBuf	FIR filter coefficients buffer. The array stores (u32Order + 1) filter coefficients.

## 5.49 GDFLIB\_FILTERFIR\_STATE\_T\_F16

Structure containing the current state of the filter.

### Source File

```
#include <GDFLIB_FilterFIR.h>
```

### Compound Type Members

**Table 439. GDFLIB\_FILTERFIR\_STATE\_T\_F16 members description**

Type	Name	Description
tU16	u16Idx	Input buffer index.
tFrac16 *	pInBuf	Pointer to the input buffer. The array stores (u16Order + 1) samples. The array must be aligned to a 32-bit boundary and there must be 2 readable bytes after the last element of the array.

## 5.50 GDFLIB\_FILTERFIR\_STATE\_T\_F32

Structure containing the current state of the filter.

### Source File

```
#include <GDFLIB_FilterFIR.h>
```

### Compound Type Members

**Table 440. GDFLIB\_FILTERFIR\_STATE\_T\_F32 members description**

Type	Name	Description
tU32	u32Idx	Input buffer index.
tFrac32 *	pInBuf	Pointer to the input buffer. The array stores (u32Order + 1) samples.

## 5.51 GDFLIB\_FILTERFIR\_STATE\_T\_FLT

Structure containing the current state of the filter.

### Source File

```
#include <GDFLIB_FilterFIR.h>
```

### Compound Type Members

**Table 441. GDFLIB\_FILTERFIR\_STATE\_T\_FLT members description**

Type	Name	Description
tU32	u32Idx	Input buffer index.
tFloat *	pInBuf	Pointer to the input buffer. The array stores (u32Order + 1) samples.

## 5.52 GFLIB\_ACOS\_T\_F16

Default approximation coefficients datatype for arccosine approximation.

### Source File

```
#include <GFLIB_Acos.h>
```

### Description

Output of  $\arccos(f16In)$  for interval  $[0, 1]$  of the input ratio is divided into two sub-sectors. Polynomial approximation is done using a 5th order polynomial, for each sub-sector respectively. Five coefficients for a single sub-interval are stored in [GFLIB\\_ACOS\\_TAYLOR\\_COEF\\_T\\_F16](#) structure.

### Compound Type Members

**Table 442. GFLIB\_ACOS\_T\_F16 members description**

Type	Name	Description
<a href="#">GFLIB_ACOS_TAYLOR_COEF_T_F16</a>	GFLIB_ACOS_SECTOR	Array of two elements for storing two sub-arrays (each sub-array contains five 16-bit coefficients) for all sub-intervals.

## 5.53 GFLIB\_ACOS\_T\_F32

Default approximation coefficients datatype for arccosine approximation.

### Source File

```
#include <GFLIB_Acos.h>
```

### Description

Output of  $\arccos(f32In)$  for interval  $[0, 1]$  of the input ratio is divided into three sub-sectors. Polynomial approximation is done using a 5th order polynomial, for each

sub-sector respectively. Five coefficients for a single sub-interval are stored in [GFLIB\\_ACOS\\_TAYLOR\\_COEF\\_T\\_F32](#) structure.

### Compound Type Members

**Table 443. GFLIB\_ACOS\_T\_F32 members description**

Type	Name	Description
<a href="#">GFLIB_ACOS_TAYLOR_COEF_T_F32</a>	GFLIB_ACOS_SECTOR	Array of two elements for storing three sub-arrays (each sub-array contains five 32-bit coefficients) for all sub-intervals.

## 5.54 GFLIB\_ACOS\_T\_FLT

Default approximation coefficients datatype for arccosine approximation.

### Source File

```
#include <GFLIB_Acos.h>
```

### Description

The polynomial approximation is done using the square root function of the input parameter. The essential assumption of this approach is a fast H/W based calculation of the square root operation. The approximation polynomial coefficients assume an input argument in the  $[-\pi, \pi]$  interval.

### Compound Type Members

**Table 444. GFLIB\_ACOS\_T\_FLT members description**

Type	Name	Description
const <a href="#">tFloat</a>	fltA	Array of approximation coefficients.

## 5.55 GFLIB\_ACOS\_TAYLOR\_COEF\_T\_F16

Array of approximation coefficients for piece-wise polynomial.

### Source File

```
#include <GFLIB_Acos.h>
```

### Description

Output of  $\arccos(f16In)$  for interval  $[0, 1]$  of the input ratio is divided into two sub-sectors. Polynomial approximation is done using a 5th order polynomial, for each sub-sector respectively. Five coefficients for a single sub-interval are stored in this (GFLIB\_ACOS\_TAYLOR\_COEF\_T\_F16) structure.

### Compound Type Members

**Table 445.** [GFLIB\\_ACOS\\_TAYLOR\\_COEF\\_T\\_F16](#) members description

Type	Name	Description
const <a href="#">tFrac16</a>	f16A	Array of five 16-bit elements for storing coefficients of the piece-wise polynomial.

## 5.56 GFLIB\_ACOS\_TAYLOR\_COEF\_T\_F32

Array of approximation coefficients for piece-wise polynomial.

### Source File

```
#include <GFLIB_Acos.h>
```

### Description

Output of arccos(f32In) for interval [0, 1) of the input ratio is divided into three sub-sectors. Polynomial approximation is done using a 5th order polynomial, for each sub-sector respectively. Five coefficients for a single sub-interval are stored in this [GFLIB\\_ACOS\\_TAYLOR\\_COEF\\_T\\_F32](#) structure.

### Compound Type Members

**Table 446.** [GFLIB\\_ACOS\\_TAYLOR\\_COEF\\_T\\_F32](#) members description

Type	Name	Description
const <a href="#">tFrac32</a>	f32A	Array of five 32-bit elements for storing coefficients of the piece-wise polynomial.

## 5.57 GFLIB\_ASIN\_T\_F16

Default approximation coefficients datatype for arcsine approximation.

### Source File

```
#include <GFLIB_Asin.h>
```

### Description

Output of arcsin(f16In) for interval [0, 1) of the input ratio is divided into two sub-sectors. Polynomial approximation is done using a 5th order polynomial, for each sub-sector respectively. Five coefficients for a single sub-interval are stored in [GFLIB\\_ASIN\\_TAYLOR\\_COEF\\_T\\_F16](#) structure.

### Compound Type Members

**Table 447.** [GFLIB\\_ASIN\\_T\\_F16](#) members description

Type	Name	Description
<a href="#">GFLIB_ASIN_TAYLOR_COEF_T_F16</a>	GFLIB_ASIN_SECTOR	Default approximation coefficients datatype for arcsine approximation.

## 5.58 GFLIB\_ASIN\_T\_F32

Default approximation coefficients datatype for arcsine approximation.

### Source File

```
#include <GFLIB_Asin.h>
```

### Description

Output of  $\arcsin(f32In)$  for interval  $[0, 1]$  of the input ratio is divided into three sub-sectors. Polynomial approximation is done using a 5th order polynomial, for each sub-sector respectively. Five coefficients for a single sub-interval are stored in [GFLIB\\_ASIN\\_TAYLOR\\_COEF\\_T\\_F32](#) structure.

### Compound Type Members

**Table 448. GFLIB\_ASIN\_T\_F32 members description**

Type	Name	Description
<a href="#">GFLIB_ASIN_TAYLOR_COEF_T_F32</a>	GFLIB_ASIN_SECTOR	Default approximation coefficients datatype for arcsine approximation.

## 5.59 GFLIB\_ASIN\_T\_FLT

Default approximation coefficients datatype for arcsine approximation.

### Source File

```
#include <GFLIB_Asin.h>
```

### Description

The polynomial approximation is done using the square root function of the input parameter. The essential assumption of this approach is a fast H/W based calculation of the square root operation. The approximation polynomial coefficients assume an input argument in the  $[-\pi, \pi]$  interval.

### Compound Type Members

**Table 449. GFLIB\_ASIN\_T\_FLT members description**

Type	Name	Description
const <a href="#">tFloat</a>	fltA	Default approximation coefficients datatype for arcsine approximation.

## 5.60 GFLIB\_ASIN\_TAYLOR\_COEF\_T\_F16

Array of approximation coefficients for piece-wise polynomial.

### Source File

```
#include <GFLIB_Asin.h>
```

### Description

Output of  $\arcsin(f16In)$  for interval  $[0, 1]$  of the input ratio is divided into two sub-sectors. Polynomial approximation is done using a 5th order polynomial, for each sub-sector respectively. Five coefficients for a single sub-interval are stored in this (GFLIB\_ASIN\_TAYLOR\_COEF\_T\_F16) structure.

### Compound Type Members

**Table 450. GFLIB\_ASIN\_TAYLOR\_COEF\_T\_F16 members description**

Type	Name	Description
const <a href="#">tFrac16</a>	f16A	Array of approximation coefficients for piece-wise polynomial.

## 5.61 GFLIB\_ASIN\_TAYLOR\_COEF\_T\_F32

Array of approximation coefficients for piece-wise polynomial.

### Source File

```
#include <GFLIB_Asin.h>
```

### Description

Output of  $\arcsin(f32In)$  for interval  $[0, 1]$  of the input ratio is divided into three sub-sectors. Polynomial approximation is done using a 5th order polynomial, for each sub-sector respectively. Five coefficients for a single sub-interval are stored in this (GFLIB\_ASIN\_TAYLOR\_COEF\_T\_F32) structure.

### Compound Type Members

**Table 451. GFLIB\_ASIN\_TAYLOR\_COEF\_T\_F32 members description**

Type	Name	Description
const <a href="#">tFrac32</a>	f32A	Array of five 32-bit elements for storing coefficients of the piece-wise polynomial.

## 5.62 GFLIB\_ATAN\_T\_F16

Structure containing eight sub-structures with polynomial coefficients to cover all sub-intervals.

### Source File

```
#include <GFLIB_Atan.h>
```

### Description

Output of  $\arctan(f16In)$  for interval  $[0, 1]$  of the input ratio is divided into eight sub-sectors. Polynomial approximation is done using a 3rd order polynomial, for each sub-sector respectively. Eight arrays, each including three polynomial coefficients for each sub-interval, are stored in this (GFLIB\_ATAN\_T\_F16) structure.

**Compound Type Members****Table 452.** *GFLIB\_ATAN\_T\_F16* members description

Type	Name	Description
const <a href="#">GFLIB_ATAN_TAYLOR_COEF_T_F16</a>	GFLIB_ATAN_SECTOR	Structure containing eight sub-structures with polynomial coefficients to cover all sub-intervals.

**5.63 GFLIB\_ATAN\_T\_F32**

Structure containing eight sub-structures with polynomial coefficients to cover all sub-intervals.

**Source File**

```
#include <GFLIB_Atan.h>
```

**Description**

Output of arctan(f32In) for interval [0, 1) of the input ratio is divided into eight sub-sectors. Polynomial approximation is done using a 3rd order polynomial, for each sub-sector respectively. Eight arrays, each including three polynomial coefficients for each sub-interval, are stored in this (GFLIB\_ATAN\_T\_F32) structure.

**Compound Type Members****Table 453.** *GFLIB\_ATAN\_T\_F32* members description

Type	Name	Description
const <a href="#">GFLIB_ATAN_TAYLOR_COEF_T_F32</a>	GFLIB_ATAN_SECTOR	Structure containing eight sub-structures with polynomial coefficients to cover all sub-intervals.

**5.64 GFLIB\_ATAN\_T\_FLT**

Structure containing the approximation coefficients.

**Source File**

```
#include <GFLIB_Atan.h>
```

**Description**

The approximation uses the rational polynomial approximation which is based on the division of two rational polynomials. The essential assumption of this approach is the fast calculation of a two floating point values division. The approximation polynomial coefficients assume the input argument in the interval (- 2<sup>128</sup>, 2<sup>128</sup>).

**Compound Type Members****Table 454.** *GFLIB\_ATAN\_T\_FLT* members description

Type	Name	Description
const <a href="#">tFloat</a>	fltA	Structure containing the approximation coefficients.

## 5.65 GFLIB\_ATAN\_TAYLOR\_COEF\_T\_F16

Array of polynomial approximation coefficients for one sub-interval.

### Source File

```
#include <GFLIB_Atan.h>
```

### Description

Output of arctan(f16In) for interval [0, 1) of the input ratio is divided into eight sub-sectors. Polynomial approximation is done using a 3rd order polynomial, for each sub-sector respectively. Three coefficients for a single sub-interval are stored in this (GFLIB\_ATAN\_TAYLOR\_COEF\_T\_F16) structure.

### Compound Type Members

**Table 455. GFLIB\_ATAN\_TAYLOR\_COEF\_T\_F16 members description**

Type	Name	Description
const <a href="#">tFrac16</a>	f16A	Array of polynomial approximation coefficients for one sub-interval.

## 5.66 GFLIB\_ATAN\_TAYLOR\_COEF\_T\_F32

Array of minimax polynomial approximation coefficients for one sub-interval.

### Source File

```
#include <GFLIB_Atan.h>
```

### Description

Output of arctan(f32In) for interval [0, 1) of the input ratio is divided into eight sub-sectors. Minimax polynomial approximation is done using a 3rd order polynomial, for each sub-sector respectively. Three coefficients for a single sub-interval are stored in this (GFLIB\_ATAN\_TAYLOR\_COEF\_T\_F32) structure.

### Compound Type Members

**Table 456. GFLIB\_ATAN\_TAYLOR\_COEF\_T\_F32 members description**

Type	Name	Description
const <a href="#">tFrac32</a>	f32A	Array of minimax polynomial approximation coefficients for one sub-interval.

## 5.67 GFLIB\_ATANYXSHIFTED\_T\_F16

Structure containing the parameter for the AtanYXShifted function.

### Source File

```
#include <GFLIB_AtanYXShifted.h>
```

**Compound Type Members****Table 457.** `GFLIB_ATANYXSHIFTED_T_F16` members description

Type	Name	Description
<a href="#">tFrac16</a>	f16Ky	Multiplication coefficient for the y-signal.
<a href="#">tFrac16</a>	f16Kx	Multiplication coefficient for the x-signal.
<a href="#">tS16</a>	s16Ny	Scaling coefficient for the y-signal.
<a href="#">tS16</a>	s16Nx	Scaling coefficient for the x-signal.
<a href="#">tFrac16</a>	f16ThetaAdj	Adjusting angle.

**5.68 GFLIB\_ATANYXSHIFTED\_T\_F32**

Structure containing the parameter for the AtanYXShifted function.

**Source File**

```
#include <GFLIB_AtanYXShifted.h>
```

**Compound Type Members****Table 458.** `GFLIB_ATANYXSHIFTED_T_F32` members description

Type	Name	Description
<a href="#">tFrac32</a>	f32Ky	Multiplication coefficient for the y-signal.
<a href="#">tFrac32</a>	f32Kx	Multiplication coefficient for the x-signal.
<a href="#">tS32</a>	s32Ny	Scaling coefficient for the y-signal.
<a href="#">tS32</a>	s32Nx	Scaling coefficient for the x-signal.
<a href="#">tFrac32</a>	f32ThetaAdj	Adjusting angle.

**5.69 GFLIB\_ATANYXSHIFTED\_T\_FLT**

Structure containing the parameter for the AtanYXShifted function.

**Source File**

```
#include <GFLIB_AtanYXShifted.h>
```

**Compound Type Members****Table 459.** `GFLIB_ATANYXSHIFTED_T_FLT` members description

Type	Name	Description
<a href="#">tFloat</a>	fltKy	Multiplication coefficient for the y-signal.
<a href="#">tFloat</a>	fltKx	Multiplication coefficient for the x-signal.
<a href="#">tFloat</a>	fltThetaAdj	Adjusting angle.

**5.70 GFLIB\_CONTROLLER\_PI\_P\_T\_F16**

Structure containing parameters and states of the parallel form PI controller.

**Source File**

```
#include <GFLIB_ControllerPIp.h>
```

**Compound Type Members****Table 460.** GFLIB\_CONTROLLER\_PI\_P\_T\_F16 members description

Type	Name	Description
tFrac16	f16PropGain	Proportional Gain, fractional format normalized to fit into (- $2^{15}$ , $2^{15}-1$ ).
tFrac16	f16IntegGain	Integral Gain, fractional format normalized to fit into (- $2^{15}$ , $2^{15}-1$ ).
tS16	s16PropGainShift	Proportional Gain Shift, integer format [-15, 15].
tS16	s16IntegGainShift	Integral Gain Shift, integer format [-15, 15].
tFrac32	f32IntegPartK_1	State variable integral part at step k-1.
tFrac16	f16InK_1	State variable input error at step k-1.

**5.71 GFLIB\_CONTROLLER\_PI\_P\_T\_F32**

Structure containing parameters and states of the parallel form PI controller.

**Source File**

```
#include <GFLIB_ControllerPIp.h>
```

**Compound Type Members****Table 461.** GFLIB\_CONTROLLER\_PI\_P\_T\_F32 members description

Type	Name	Description
tFrac32	f32PropGain	Proportional Gain, fractional format normalized to fit into (- $2^{31}$ , $2^{31}-1$ ).
tFrac32	f32IntegGain	Integral Gain, fractional format normalized to fit into (- $2^{31}$ , $2^{31}-1$ ).
tS16	s16PropGainShift	Proportional Gain Shift, integer format [-31, 31].
tS16	s16IntegGainShift	Integral Gain Shift, integer format [-31, 31].
tFrac32	f32IntegPartK_1	State variable integral part at step k-1.
tFrac32	f32InK_1	State variable input error at step k-1.

**5.72 GFLIB\_CONTROLLER\_PI\_P\_T\_FLT**

Structure containing parameters and states of the parallel form PI controller.

**Source File**

```
#include <GFLIB_ControllerPIp.h>
```

**Compound Type Members****Table 462.** **GFLIB\_CONTROLLER\_PI\_P\_T\_FLT** members description

Type	Name	Description
tFloat	fltPropGain	Proportional Gain, single precision floating point format.
tFloat	fltIntegGain	Integral Gain, single precision floating point format.
tFloat	fltIntegPartK_1	State variable integral part at step k-1, single precision floating point format.
tFloat	fltInK_1	State variable input error at step k-1, single precision floating point format.

**5.73 GFLIB\_CONTROLLER\_PI\_R\_T\_F16**

Structure containing parameters and states of the recurrent form PI controller.

**Source File**

```
#include <GFLIB_ControllerPIr.h>
```

**Compound Type Members****Table 463.** **GFLIB\_CONTROLLER\_PI\_R\_T\_F16** members description

Type	Name	Description
tFrac16	f16CC1sc	CC1 coefficient, fractional format normalized to fit into (- $2^{15}$ , $2^{15}-1$ ).
tFrac16	f16CC2sc	CC2 coefficient, fractional format normalized to fit into (- $2^{15}$ , $2^{15}-1$ ).
tFrac32	f32Acc	State variable - internal controller accumulator.
tFrac16	f16InErrK1	State variable - controller input from the previous calculation step.
tU16	u16NShift	Scaling bitwise shift applied to the controller coefficients, integer format [0, 15].

**5.74 GFLIB\_CONTROLLER\_PI\_R\_T\_F32**

Structure containing parameters and states of the recurrent form PI controller.

**Source File**

```
#include <GFLIB_ControllerPIr.h>
```

**Compound Type Members****Table 464.** **GFLIB\_CONTROLLER\_PI\_R\_T\_F32** members description

Type	Name	Description
tFrac32	f32CC1sc	CC1 coefficient, fractional format normalized to fit into (- $2^{31}$ , $2^{31}-1$ ).

Type	Name	Description
tFrac32	f32CC2sc	CC2 coefficient, fractional format normalized to fit into (- $2^{31}$ , $2^{31}-1$ ).
tFrac32	f32Acc	State variable - internal controller accumulator.
tFrac32	f32InErrK1	State variable - controller input from the previous calculation step.
tU16	u16NShift	Scaling bitwise shift applied to the controller coefficients, integer format [0, 31].

## 5.75 GFLIB\_CONTROLLER\_PI\_R\_T\_FLT

Structure containing parameters and states of the recurrent form PI controller.

### Source File

```
#include <GFLIB_ControllerPIr.h>
```

### Compound Type Members

**Table 465. GFLIB\_CONTROLLER\_PI\_R\_T\_FLT members description**

Type	Name	Description
tFloat	fltCC1sc	CC1 coefficient, single precision floating point format.
tFloat	fltCC2sc	CC2 coefficient, single precision floating point format.
tFloat	fltAcc	State variable - internal controller accumulator, single precision floating point format.
tFloat	fltInErrK1	State variable - controller input from the previous calculation step, single precision floating point format.

## 5.76 GFLIB\_CONTROLLER\_PIAW\_P\_T\_F16

Structure containing parameters and states of the parallel form PI controller with anti-windup.

### Source File

```
#include <GFLIB_ControllerPIpAW.h>
```

### Compound Type Members

**Table 466. GFLIB\_CONTROLLER\_PIAW\_P\_T\_F16 members description**

Type	Name	Description
tFrac16	f16PropGain	Proportional Gain, fractional format normalized to fit into (- $2^{15}$ , $2^{15}-1$ ).
tFrac16	f16IntegGain	Integral Gain, fractional format normalized to fit into (- $2^{15}$ , $2^{15}-1$ ).

Type	Name	Description
tS16	s16PropGainShift	Proportional Gain Shift, integer format [-15, 15].
tS16	s16IntegGainShift	Integral Gain Shift, integer format [-15, 15].
tFrac16	f16LowerLimit	Lower Limit of the controller, fractional format normalized to fit into (- $2^{15}$ , $2^{15}-1$ ).
tFrac16	f16UpperLimit	Upper Limit of the controller, fractional format normalized to fit into (- $2^{15}$ , $2^{15}-1$ ).
tFrac32	f32IntegPartK_1	State variable integral part at step k-1.
tFrac16	f16InK_1	State variable input error at step k-1.
tU16	u16LimitFlag	Limitation flag, if set to 1, the controller output has reached either the UpperLimit or LowerLimit.

## 5.77 GFLIB\_CONTROLLER\_PIAW\_P\_T\_F32

Structure containing parameters and states of the parallel form PI controller with anti-windup.

### Source File

```
#include <GFLIB_ControllerPIpAW.h>
```

### Compound Type Members

Table 467. GFLIB\_CONTROLLER\_PIAW\_P\_T\_F32 members description

Type	Name	Description
tFrac32	f32PropGain	Proportional Gain, fractional format normalized to fit into (- $2^{31}$ , $2^{31}-1$ ).
tFrac32	f32IntegGain	Integral Gain, fractional format normalized to fit into (- $2^{31}$ , $2^{31}-1$ ).
tS16	s16PropGainShift	Proportional Gain Shift, integer format [-31, 31].
tS16	s16IntegGainShift	Integral Gain Shift, integer format [-31, 31].
tFrac32	f32LowerLimit	Lower Limit of the controller, fractional format normalized to fit into (- $2^{31}$ , $2^{31}-1$ ).
tFrac32	f32UpperLimit	Upper Limit of the controller, fractional format normalized to fit into (- $2^{31}$ , $2^{31}-1$ ).
tFrac32	f32IntegPartK_1	State variable integral part at step k-1.
tFrac32	f32InK_1	State variable input error at step k-1.
tU16	u16LimitFlag	Limitation flag, if set to 1, the controller output has reached either the UpperLimit or LowerLimit.

## 5.78 GFLIB\_CONTROLLER\_PIAW\_P\_T\_FLT

Structure containing parameters and states of the parallel form PI controller with anti-windup.

**Source File**

```
#include <GFLIB_ControllerPIpAW.h>
```

**Compound Type Members****Table 468. GFLIB\_CONTROLLER\_PIAW\_P\_T\_FLT members description**

Type	Name	Description
tFloat	fltPropGain	Proportional Gain, single precision floating point format.
tFloat	fltIntegGain	Integral Gain, single precision floating point format.
tFloat	fltLowerLimit	Lower Limit of the controller, single precision floating point format.
tFloat	fltUpperLimit	Upper Limit of the controller, single precision floating point format.
tFloat	fltIntegPartK_1	State variable integral part at step k-1, single precision floating point format.
tFloat	fltInK_1	State variable input error at step k-1, single precision floating point format.
tU16	u16LimitFlag	Limitation flag, if set to 1, the controller output has reached either the UpperLimit or LowerLimit.

**5.79 GFLIB\_CONTROLLER\_PIAW\_R\_T\_F16**

Structure containing parameters and states of the recurrent form PI controller with anti-windup.

**Source File**

```
#include <GFLIB_ControllerPIrAW.h>
```

**Compound Type Members****Table 469. GFLIB\_CONTROLLER\_PIAW\_R\_T\_F16 members description**

Type	Name	Description
tFrac16	f16CC1sc	CC1 coefficient, fractional format normalized to fit into (- 2 <sup>15</sup> , 2 <sup>15</sup> -1).
tFrac16	f16CC2sc	CC2 coefficient, fractional format normalized to fit into (- 2 <sup>15</sup> , 2 <sup>15</sup> -1).
tFrac32	f32Acc	State variable - internal controller accumulator.
tFrac16	f16InErrK1	State variable - controller input from the previous calculation step.
tFrac16	f16UpperLimit	Upper Limit of the controller, fractional format normalized to fit into (- 2 <sup>15</sup> , 2 <sup>15</sup> -1).
tFrac16	f16LowerLimit	Lower Limit of the controller, fractional format normalized to fit into (- 2 <sup>15</sup> , 2 <sup>15</sup> -1).

Type	Name	Description
<a href="#">tU16</a>	u16NShift	Scaling bitwise shift applied to the controller coefficients, integer format [0, 15].

## 5.80 [GFLIB\\_CONTROLLER\\_PIAW\\_R\\_T\\_F32](#)

Structure containing parameters and states of the recurrent form PI controller with anti-windup.

### Source File

```
#include <GFLIB_ControllerPIrAW.h>
```

### Compound Type Members

**Table 470. GFLIB\_CONTROLLER\_PIAW\_R\_T\_F32 members description**

Type	Name	Description
<a href="#">tFrac32</a>	f32CC1sc	CC1 coefficient, fractional format normalized to fit into (- $2^{31}$ , $2^{31}-1$ ).
<a href="#">tFrac32</a>	f32CC2sc	CC2 coefficient, fractional format normalized to fit into (- $2^{31}$ , $2^{31}-1$ ).
<a href="#">tFrac32</a>	f32Acc	State variable - internal controller accumulator.
<a href="#">tFrac32</a>	f32InErrK1	State variable - controller input from the previous calculation step.
<a href="#">tFrac32</a>	f32UpperLimit	Upper Limit of the controller, fractional format normalized to fit into (- $2^{31}$ , $2^{31}-1$ ).
<a href="#">tFrac32</a>	f32LowerLimit	Lower Limit of the controller, fractional format normalized to fit into (- $2^{31}$ , $2^{31}-1$ ).
<a href="#">tU16</a>	u16NShift	Scaling bitwise shift applied to the controller coefficients, integer format [0, 31].

## 5.81 [GFLIB\\_CONTROLLER\\_PIAW\\_R\\_T\\_FLT](#)

Structure containing parameters and states of the recurrent form PI controller with anti-windup.

### Source File

```
#include <GFLIB_ControllerPIrAW.h>
```

### Compound Type Members

**Table 471. GFLIB\_CONTROLLER\_PIAW\_R\_T\_FLT members description**

Type	Name	Description
<a href="#">tFloat</a>	fltCC1sc	CC1 coefficient, single precision floating point format.
<a href="#">tFloat</a>	fltCC2sc	CC2 coefficient, single precision floating point format.

Type	Name	Description
tFloat	fltAcc	State variable - internal controller accumulator, single precision floating point format.
tFloat	fltInErrK1	State variable - controller input from the previous calculation step, single precision floating point format.
tFloat	fltUpperLimit	Upper Limit of the controller, single precision floating point format.
tFloat	fltLowerLimit	Lower Limit of the controller, single precision floating point format.

## 5.82 GFLIB\_CONTROLLER\_PID\_P\_AW\_T\_F16

Structure containing parameters and states of the parallel form PI controller.

### Source File

```
#include <GFLIB_ControllerPIDpAW.h>
```

### Compound Type Members

Table 472. GFLIB\_CONTROLLER\_PID\_P\_AW\_T\_F16 members description

Type	Name	Description
tFrac16	f16PropGain	Proportional Gain, fractional format normalized to fit into (0, $2^{15}$ -1).
tFrac16	f16IntegGain	Integral Gain, fractional format normalized to fit into (0, $2^{15}$ -1).
tFrac16	f16DerivGain	Derivative Gain, fractional format normalized to fit into (0, $2^{15}$ -1).
tFrac16	f16FiltCoef	Derivative term filter coefficient, fractional format in range (0, $2^{15}$ -1).
tS16	s16PropGainShift	Proportional Gain Shift, integer format [-15, 15].
tS16	s16IntegGainShift	Integral Gain Shift, integer format [-15, 15].
tS16	s16DerivGainShift	Derivative Gain Shift, integer format [-15, 15].
tFrac16	f16LowerLimit	Lower Limit of the controller, fractional format normalized to fit into (- $2^{15}$ , $2^{15}$ -1).
tFrac16	f16UpperLimit	Upper Limit of the controller, fractional format normalized to fit into (- $2^{15}$ , $2^{15}$ -1).
tFrac32	f32IntegPartK_1	State variable integral part at step k-1.
tFrac16	f16DerivPartK_1	State variable integral part at step k-1.
tFrac16	f16InK_1	State variable input error at step k-1.
tU16	u16LimitFlag	Limitation flag, if set to 1, the controller output has reached either the UpperLimit or LowerLimit.

## 5.83 GFLIB\_CONTROLLER\_PID\_P\_AW\_T\_F32

Structure containing parameters and states of the PID controller.

### Source File

```
#include <GFLIB_ControllerPIDpAW.h>
```

### Compound Type Members

**Table 473. GFLIB\_CONTROLLER\_PID\_P\_AW\_T\_F32 members description**

Type	Name	Description
tFrac32	f32PropGain	Proportional Gain, fractional format normalized to fit into (0, $2^{31}$ -1).
tFrac32	f32IntegGain	Integral Gain, fractional format normalized to fit into (0, $2^{31}$ -1).
tFrac32	f32DerivGain	Derivative Gain, fractional format normalized to fit into (0, $2^{31}$ -1).
tFrac32	f32FiltCoef	Derivative term filter coefficient, fractional format in range (0, $2^{31}$ -1).
tS16	s16PropGainShift	Proportional Gain Shift, integer format [-31, 31].
tS16	s16IntegGainShift	Integral Gain Shift, integer format [-31, 31].
tS16	s16DerivGainShift	Derivative Gain Shift, integer format [-31, 31].
tFrac32	f32LowerLimit	Lower Limit of the controller, fractional format normalized to fit into (- $2^{31}$ , $2^{31}$ -1).
tFrac32	f32UpperLimit	Upper Limit of the controller, fractional format normalized to fit into (- $2^{31}$ , $2^{31}$ -1).
tFrac32	f32IntegPartK_1	State variable integral part at step k-1.
tFrac32	f32DerivPartK_1	State variable derivative part filter at step k-1.
tFrac32	f32InK_1	State variable input error at step k-1.
tU16	u16LimitFlag	Limitation flag, if set to 1, the controller output has reached either the UpperLimit or LowerLimit.

## 5.84 GFLIB\_CONTROLLER\_PID\_P\_AW\_T\_FLT

Structure containing parameters and states of the parallel form PI controller.

### Source File

```
#include <GFLIB_ControllerPIDpAW.h>
```

**Compound Type Members****Table 474. GFLIB\_CONTROLLER\_PID\_P\_AW\_T\_FLT members description**

Type	Name	Description
tFloat	fltPropGain	Proportional Gain, positive number in single precision floating point format.
tFloat	fltIntegGain	Integral Gain, positive number in single precision floating point format.
tFloat	fltDerivGain	Derivative Gain, positive number in single precision floating point format.
tFloat	fltFiltCoef	Derivative term filter coefficient, single precision floating point format.
tFloat	fltLowerLimit	Lower Limit of the controller, single precision floating point format.
tFloat	fltUpperLimit	Upper Limit of the controller, single precision floating point format.
tFloat	fltIntegPartK_1	State variable integral part at step k-1, single precision floating point format.
tFloat	fltDerivPartK_1	State variable integral part at step k-1, single precision floating point format.
tFloat	fltInK_1	State variable input error at step k-1, single precision floating point format.
tU16	u16LimitFlag	Limitation flag, if set to 1, the controller output has reached either the UpperLimit or LowerLimit.

**5.85 GFLIB\_COS\_T\_F16**

Array of four 16-bit elements for storing coefficients of the Taylor polynomial.

**Source File**

```
#include <GFLIB_Cos.h>
```

**Compound Type Members****Table 475. GFLIB\_COS\_T\_F16 members description**

Type	Name	Description
tFrac16	f16A	Array of four 16-bit elements for storing coefficients of the Taylor polynomial.

**5.86 GFLIB\_COS\_T\_F32**

Array of five 32-bit elements for storing coefficients of the Taylor polynomial.

**Source File**

```
#include <GFLIB_Cos.h>
```

**Compound Type Members****Table 476.** **GFLIB\_COS\_T\_F32** members description

Type	Name	Description
<a href="#">tFrac32</a>	f32A	Array of five 32-bit elements for storing coefficients of the Taylor polynomial.

**5.87 GFLIB\_COS\_T\_FLT**

Array of three single precision floating point elements for storing coefficients of the floating point optimized minimax approximation polynomial.

**Source File**

```
#include <GFLIB_Cos.h>
```

**Compound Type Members****Table 477.** **GFLIB\_COS\_T\_FLT** members description

Type	Name	Description
<a href="#">tFloat</a>	fltA	Array of three single precision floating point elements for storing coefficients of the floating point optimized minimax approximation polynomial.

**5.88 GFLIB\_HYST\_T\_F16**

Structure containing parameters and states for the hysteresis function.

**Source File**

```
#include <GFLIB_Hyst.h>
```

**Compound Type Members****Table 478.** **GFLIB\_HYST\_T\_F16** members description

Type	Name	Description
<a href="#">tFrac16</a>	f16HystOn	Value determining the upper threshold.
<a href="#">tFrac16</a>	f16HystOff	Value determining the lower threshold.
<a href="#">tFrac16</a>	f16OutValOn	Value of the output when input is higher than the upper threshold.
<a href="#">tFrac16</a>	f16OutValOff	Value of the output when input is lower than the lower threshold.
<a href="#">tFrac16</a>	f16OutState	Actual state of the output.

**5.89 GFLIB\_HYST\_T\_F32**

Structure containing parameters and states for the hysteresis function.

**Source File**

```
#include <GFLIB_Hyst.h>
```

**Compound Type Members****Table 479.** GFLIB\_HYST\_T\_F32 members description

Type	Name	Description
tFrac32	f32HystOn	Value determining the upper threshold.
tFrac32	f32HystOff	Value determining the lower threshold.
tFrac32	f32OutValOn	Value of the output when input is higher than the upper threshold.
tFrac32	f32OutValOff	Value of the output when input is lower than the lower threshold.
tFrac32	f32OutState	Actual state of the output.

**5.90 GFLIB\_HYST\_T\_FLT**

Structure containing parameters and states for the hysteresis function.

**Source File**

```
#include <GFLIB_Hyst.h>
```

**Compound Type Members****Table 480.** GFLIB\_HYST\_T\_FLT members description

Type	Name	Description
tFloat	fltHystOn	Value determining the upper threshold.
tFloat	fltHystOff	Value determining the lower threshold.
tFloat	fltOutValOn	Value of the output when input is higher than the upper threshold.
tFloat	fltOutValOff	Value of the output when input is lower than the lower threshold.
tFloat	fltOutState	Actual state of the output.

**5.91 GFLIB\_INTEGRATOR\_TR\_T\_F16**

Structure containing integrator parameters and coefficients.

**Source File**

```
#include <GFLIB_IntegratorTR.h>
```

**Compound Type Members****Table 481.** GFLIB\_INTEGRATOR\_TR\_T\_F16 members description

Type	Name	Description
tFrac32	f32State	State variable - integrator state value.

Type	Name	Description
tFrac16	f16InK1	State variable - input value in step k-1.
tFrac16	f16C1	Integrator coefficient = ( E <sub>MAX</sub> / T <sub>s</sub> ) ( U <sub>MAX</sub> *2)*( 2 <sup>-u16NShift</sup> ).
tU16	u16NShift	Scaling bitwise shift applied to the integrator coefficient f16C1, integer format [0, 15].

## 5.92 GFLIB\_INTEGRATOR\_TR\_T\_F32

Structure containing integrator parameters and coefficients.

### Source File

```
#include <GFLIB_IntegratorTR.h>
```

### Compound Type Members

Table 482. GFLIB\_INTEGRATOR\_TR\_T\_F32 members description

Type	Name	Description
tFrac32	f32State	State variable - integrator state value.
tFrac32	f32InK1	State variable - input value in step k-1.
tFrac32	f32C1	Integrator coefficient = ( E <sub>MAX</sub> / T <sub>s</sub> ) ( U <sub>MAX</sub> *2)*( 2 <sup>-u16NShift</sup> ).
tU16	u16NShift	Scaling bitwise shift applied to the integrator coefficient f32C1, integer format [0, 15].

## 5.93 GFLIB\_INTEGRATOR\_TR\_T\_FLT

Structure containing integrator parameters and coefficients.

### Source File

```
#include <GFLIB_IntegratorTR.h>
```

### Compound Type Members

Table 483. GFLIB\_INTEGRATOR\_TR\_T\_FLT members description

Type	Name	Description
tFloat	fltState	State variable - integrator state value, single precision floating point format.
tFloat	fltInK1	State variable - input value in step k-1, single precision floating point format.
tFloat	fltC1	Integrator coefficient, single precision floating point format.

## 5.94 GFLIB\_LIMIT\_T\_F16

Structure containing the limits.

**Source File**

```
#include <GFLIB_Limit.h>
```

**Compound Type Members****Table 484.** GFLIB\_LIMIT\_T\_F16 members description

Type	Name	Description
tFrac16	f16LowerLimit	Value determining the lower limit threshold.
tFrac16	f16UpperLimit	Value determining the upper limit threshold.

**5.95 GFLIB\_LIMIT\_T\_F32**

Structure containing the limits.

**Source File**

```
#include <GFLIB_Limit.h>
```

**Compound Type Members****Table 485.** GFLIB\_LIMIT\_T\_F32 members description

Type	Name	Description
tFrac32	f32LowerLimit	Value determining the lower limit threshold.
tFrac32	f32UpperLimit	Value determining the upper limit threshold.

**5.96 GFLIB\_LIMIT\_T\_FLT**

Structure containing the limits.

**Source File**

```
#include <GFLIB_Limit.h>
```

**Compound Type Members****Table 486.** GFLIB\_LIMIT\_T\_FLT members description

Type	Name	Description
tFloat	fltLowerLimit	Value determining the lower limit threshold.
tFloat	fltUpperLimit	Value determining the upper limit threshold.

**5.97 GFLIB\_LOG10\_T\_FLT**

Array of single precision floating point elements for storing the coefficients of the floating point log10 approximation polynomial.

**Source File**

```
#include <GFLIB_Log10.h>
```

### Compound Type Members

Table 487. GFLIB\_LOG10\_T\_FLT members description

Type	Name	Description
tFloat	fltA	Array of single precision floating point elements for storing the coefficients of the floating point log10 approximation polynomial.

## 5.98 GFLIB\_LOWERLIMIT\_T\_F16

Structure containing the lower limit.

### Source File

```
#include <GFLIB_LowerLimit.h>
```

### Compound Type Members

Table 488. GFLIB\_LOWERLIMIT\_T\_F16 members description

Type	Name	Description
tFrac16	f16LowerLimit	Value determining the lower limit threshold.

## 5.99 GFLIB\_LOWERLIMIT\_T\_F32

Structure containing the lower limit.

### Source File

```
#include <GFLIB_LowerLimit.h>
```

### Compound Type Members

Table 489. GFLIB\_LOWERLIMIT\_T\_F32 members description

Type	Name	Description
tFrac32	f32LowerLimit	Value determining the lower limit threshold.

## 5.100 GFLIB\_LOWERLIMIT\_T\_FLT

Structure containing the lower limit.

### Source File

```
#include <GFLIB_LowerLimit.h>
```

### Compound Type Members

Table 490. GFLIB\_LOWERLIMIT\_T\_FLT members description

Type	Name	Description
tFloat	fltLowerLimit	Value determining the lower limit threshold.

## 5.101 GFLIB\_LUT1D\_T\_F16

Structure containing 1D look-up table parameters.

### Source File

```
#include <GFLIB_Lut1D.h>
```

### Compound Type Members

**Table 491. GFLIB\_LUT1D\_T\_F16 members description**

Type	Name	Description
tU16	u16ShamOffset	Shift amount for extracting the fractional offset within an interpolated interval.
const tFrac16 *	pf16Table	Table holding ordinate values of interpolating intervals.

## 5.102 GFLIB\_LUT1D\_T\_F32

Structure containing 1D look-up table parameters.

### Source File

```
#include <GFLIB_Lut1D.h>
```

### Compound Type Members

**Table 492. GFLIB\_LUT1D\_T\_F32 members description**

Type	Name	Description
tU32	u32ShamOffset	Shift amount for extracting the fractional offset within an interpolated interval.
const tFrac32 *	pf32Table	Table holding ordinate values of interpolating intervals.

## 5.103 GFLIB\_LUT1D\_T\_FLT

Structure containing 1D look-up table parameters.

### Source File

```
#include <GFLIB_Lut1D.h>
```

### Compound Type Members

**Table 493. GFLIB\_LUT1D\_T\_FLT members description**

Type	Name	Description
tU32	u32ShamOffset	Shift amount for extracting the fractional offset within an interpolated interval.
const tFloat *	pfltTable	Table holding ordinate values of interpolating intervals. The ordinate values are in full range single precision floating point format.

## 5.104 GFLIB\_LUT2D\_T\_F16

Structure containing 2D look-up table parameters.

### Source File

```
#include <GFLIB_Lut2D.h>
```

### Compound Type Members

**Table 494. GFLIB\_LUT2D\_T\_F16 members description**

Type	Name	Description
tU16	u16ShamOffset1	Shift amount for extracting the fractional offset within an interpolated interval.
tU16	u16ShamOffset2	Shift amount for extracting the fractional offset within an interpolated interval.
const tFrac16 *	pf16Table	Table holding values of interpolating intervals stored in full column-major format.

## 5.105 GFLIB\_LUT2D\_T\_F32

Structure containing 2D look-up table parameters.

### Source File

```
#include <GFLIB_Lut2D.h>
```

### Compound Type Members

**Table 495. GFLIB\_LUT2D\_T\_F32 members description**

Type	Name	Description
tU32	u32ShamOffset1	Shift amount for extracting the fractional offset within an interpolated interval.
tU32	u32ShamOffset2	Shift amount for extracting the fractional offset within an interpolated interval.
const tFrac32 *	pf32Table	Table holding values of interpolating intervals stored in full column-major format.

## 5.106 GFLIB\_LUT2D\_T\_FLT

Structure containing 2D look-up table parameters.

### Source File

```
#include <GFLIB_Lut2D.h>
```

**Compound Type Members****Table 496.** `GFLIB_LUT2D_T_FLT` members description

Type	Name	Description
<code>tU32</code>	<code>u32ShamOffset1</code>	Shift amount for extracting the fractional offset within an interpolated interval.
<code>tU32</code>	<code>u32ShamOffset2</code>	Shift amount for extracting the fractional offset within an interpolated interval.
const <code>tFloat</code> *	<code>pfltTable</code>	Table holding ordinate of interpolating intervals stored in full column-major format. The ordinate values are in full range single precision floating point format.

**5.107 GFLIB\_RAMP\_T\_F16**

Structure containing increment/decrement coefficients and state value for the ramp function implemented in `GFLIB_Ramp`.

**Source File**

```
#include <GFLIB_Ramp.h>
```

**Compound Type Members****Table 497.** `GFLIB_RAMP_T_F16` members description

Type	Name	Description
<code>tFrac16</code>	<code>f16State</code>	Ramp state value.
<code>tFrac16</code>	<code>f16RampUp</code>	Ramp up increment coefficient.
<code>tFrac16</code>	<code>f16RampDown</code>	Ramp decrement coefficient.

**5.108 GFLIB\_RAMP\_T\_F32**

Structure containing increment/decrement coefficients and state value for the ramp function implemented in `GFLIB_Ramp`.

**Source File**

```
#include <GFLIB_Ramp.h>
```

**Compound Type Members****Table 498.** `GFLIB_RAMP_T_F32` members description

Type	Name	Description
<code>tFrac32</code>	<code>f32State</code>	Ramp state value.
<code>tFrac32</code>	<code>f32RampUp</code>	Ramp up increment coefficient.
<code>tFrac32</code>	<code>f32RampDown</code>	Ramp decrement coefficient.

## 5.109 GFLIB\_RAMP\_T\_FLT

Structure containing increment/decrement coefficients and state value for the ramp function implemented in GFLIB\_Ramp.

### Source File

```
#include <GFLIB_Ramp.h>
```

### Compound Type Members

**Table 499. GFLIB\_RAMP\_T\_FLT members description**

Type	Name	Description
tFloat	fltState	Ramp state value.
tFloat	fltRampUp	Ramp up increment coefficient.
tFloat	fltRampDown	Ramp decrement coefficient.

## 5.110 GFLIB\_SIN\_T\_F16

Array of four 16-bit elements for storing coefficients of the Taylor polynomial.

### Source File

```
#include <GFLIB_Sin.h>
```

### Compound Type Members

**Table 500. GFLIB\_SIN\_T\_F16 members description**

Type	Name	Description
tFrac16	f16A	Array of four 16-bit elements for storing coefficients of the Taylor polynomial.

## 5.111 GFLIB\_SIN\_T\_F32

Array of five 32-bit elements for storing coefficients of the Taylor polynomial.

### Source File

```
#include <GFLIB_Sin.h>
```

### Compound Type Members

**Table 501. GFLIB\_SIN\_T\_F32 members description**

Type	Name	Description
tFrac32	f32A	Array of five 32-bit elements for storing coefficients of the Taylor polynomial.

## 5.112 GFLIB\_SIN\_T\_FLT

Array of three single precision floating point elements for storing coefficients of the floating point optimized minimax approximation polynomial.

**Source File**

```
#include <GFLIB_Sin.h>
```

**Compound Type Members****Table 502.** GFLIB\_SIN\_T\_FLT members description

Type	Name	Description
tFloat	fltA	Array of three single precision floating point elements for storing coefficients of the floating point optimized minimax approximation polynomial.

**5.113 GFLIB\_SINCOS\_T\_F16**

Array of four 16-bit elements for storing coefficients of the Taylor polynomial.

**Source File**

```
#include <GFLIB_SinCos.h>
```

**Compound Type Members****Table 503.** GFLIB\_SINCOS\_T\_F16 members description

Type	Name	Description
tFrac16	f16A	Array of four 16-bit elements for storing coefficients of the Taylor polynomial.

**5.114 GFLIB\_SINCOS\_T\_F32**

Array of five 32-bit elements for storing coefficients of the Taylor polynomial.

**Source File**

```
#include <GFLIB_SinCos.h>
```

**Compound Type Members****Table 504.** GFLIB\_SINCOS\_T\_F32 members description

Type	Name	Description
tFrac32	f32A	Array of five 32-bit elements for storing coefficients of the Taylor polynomial.

**5.115 GFLIB\_SINCOS\_T\_FLT**

Array of three single precision floating point elements for storing coefficients of the floating point optimized minimax approximation polynomial.

**Source File**

```
#include <GFLIB_SinCos.h>
```

**Compound Type Members****Table 505. GFLIB\_SINCOS\_T\_FLT members description**

Type	Name	Description
<a href="#">tFloat</a>	fltA	Array of three single precision floating point elements for storing coefficients of the floating point optimized minimax approximation polynomial.

**5.116 GFLIB\_TAN\_T\_F16**

Output of  $\tan(\pi * f16In)$  for interval  $[0, \pi/4]$  of the input angles is divided into eight sub-sectors. Polynomial approximation is done using a 4th order polynomial, for each sub-sector respectively. Eight arrays, each including four polynomial coefficients for each sub-interval, are stored in this (GFLIB\_TAN\_T\_F16) structure.

**Source File**

```
#include <GFLIB_Tan.h>
```

**Compound Type Members****Table 506. GFLIB\_TAN\_T\_F16 members description**

Type	Name	Description
<a href="#">GFLIB_TAN_TAYLOR_COEF_T_F16</a>	GFLIB_TAN_SECTOR	Output of $\tan(\pi * f16In)$ for interval $[0, \pi/4]$ of the input angles is divided into eight sub-sectors. Polynomial approximation is done using a 4th order polynomial, for each sub-sector respectively. Eight arrays, each including four polynomial coefficients for each sub-interval, are stored in this (GFLIB_TAN_T_F16) structure.

**5.117 GFLIB\_TAN\_T\_F32**

Output of  $\tan(\pi * f32In)$  for interval  $[0, \pi/4]$  of the input angles is divided into eight sub-sectors. Polynomial approximation is done using a 4th order polynomial, for each sub-sector respectively. Eight arrays, each including four polynomial coefficients for each sub-interval, are stored in this (GFLIB\_TAN\_T\_F32) structure.

**Source File**

```
#include <GFLIB_Tan.h>
```

### Compound Type Members

**Table 507.** `GFLIB_TAN_T_F32` members description

Type	Name	Description
<code>GFLIB_TAN_TAYLOR_COEF_T_F32</code>	<code>GFLIB_TAN_SECTOR</code>	Output of $\tan(\pi * f32In)$ for interval $[0, \pi/4]$ of the input angles is divided into eight sub-sectors. Polynomial approximation is done using a 4th order polynomial, for each sub-sector respectively. Eight arrays, each including four polynomial coefficients for each sub-interval, are stored in this ( <code>GFLIB_TAN_T_F32</code> ) structure.

## 5.118 `GFLIB_TAN_T_FLT`

Polynomial coefficient for fractional approximation in single precision floating point format.

### Source File

```
#include <GFLIB_Tan.h>
```

### Compound Type Members

**Table 508.** `GFLIB_TAN_T_FLT` members description

Type	Name	Description
<code>tFloat</code>	<code>f1tA</code>	Polynomial coefficient for fractional approximation in single precision floating point format.

## 5.119 `GFLIB_TAN_TAYLOR_COEF_T_F16`

Structure containing four polynomial coefficients for one sub-interval.

### Source File

```
#include <GFLIB_Tan.h>
```

### Description

Output of  $\tan(\pi * f16In)$  for interval  $[0, \pi/4]$  of the input angles is divided into eight sub-sectors. Polynomial approximation is done using a 4th order polynomial, for each sub-sector respectively. Four coefficients for a single sub-interval are stored in this (`GFLIB_TAN_TAYLOR_COEF_T_F16`) structure.

### Compound Type Members

**Table 509.** `GFLIB_TAN_TAYLOR_COEF_T_F16` members description

Type	Name	Description
const <code>tFrac16</code>	<code>f16A</code>	Structure containing four polynomial coefficients for one sub-interval.

## 5.120 GFLIB\_TAN\_TAYLOR\_COEF\_T\_F32

Structure containing four polynomial coefficients for one sub-interval.

### Source File

```
#include <GFLIB_Tan.h>
```

### Description

Output of  $\tan(\pi * f32In)$  for interval  $[0, \pi/4]$  of the input angles is divided into eight sub-sectors. Polynomial approximation is done using a 4th order polynomial, for each sub-sector respectively. Four coefficients for a single sub-interval are stored in this (GFLIB\_TAN\_TAYLOR\_COEF\_T\_F32) structure.

### Compound Type Members

**Table 510. GFLIB\_TAN\_TAYLOR\_COEF\_T\_F32 members description**

Type	Name	Description
const <a href="#">tFrac32</a>	f32A	Structure containing four polynomial coefficients for one sub-interval.

## 5.121 GFLIB\_UPPERLIMIT\_T\_F16

Structure containing the upper limit.

### Source File

```
#include <GFLIB_UpperLimit.h>
```

### Compound Type Members

**Table 511. GFLIB\_UPPERLIMIT\_T\_F16 members description**

Type	Name	Description
<a href="#">tFrac16</a>	f16UpperLimit	Value determining the upper limit threshold.

## 5.122 GFLIB\_UPPERLIMIT\_T\_F32

Structure containing the upper limit.

### Source File

```
#include <GFLIB_UpperLimit.h>
```

### Compound Type Members

**Table 512. GFLIB\_UPPERLIMIT\_T\_F32 members description**

Type	Name	Description
<a href="#">tFrac32</a>	f32UpperLimit	Value determining the upper limit threshold.

## 5.123 GFLIB\_UPPERLIMIT\_T\_FLT

Structure containing the upper limit.

### Source File

```
#include <GFLIB_UpperLimit.h>
```

### Compound Type Members

**Table 513. GFLIB\_UPPERLIMIT\_T\_FLT members description**

Type	Name	Description
tFloat	fltUpperLimit	Value determining the upper limit threshold.

## 5.124 GFLIB\_VECTORLIMIT\_T\_F16

Structure containing the limit.

### Source File

```
#include <GFLIB_VectorLimit.h>
```

### Compound Type Members

**Table 514. GFLIB\_VECTORLIMIT\_T\_F16 members description**

Type	Name	Description
tFrac16	f16Limit	The maximum magnitude of the input vector. The defined magnitude must be positive and equal to or greater than F16_MAX value.

## 5.125 GFLIB\_VECTORLIMIT\_T\_F32

Structure containing the limit.

### Source File

```
#include <GFLIB_VectorLimit.h>
```

### Compound Type Members

**Table 515. GFLIB\_VECTORLIMIT\_T\_F32 members description**

Type	Name	Description
tFrac32	f32Limit	The maximum magnitude of the input vector. The defined magnitude must be positive and equal to or greater than F32_MAX value.

## 5.126 GFLIB\_VECTORLIMIT\_T\_FLT

Structure containing the limit.

**Source File**

```
#include <GFLIB_VectorLimit.h>
```

**Compound Type Members****Table 516.** GFLIB\_VECTORLIMIT\_T\_FLT members description

Type	Name	Description
tFloat	fltLimit	The maximum magnitude of the input vector. The defined magnitude must be positive and equal to or greater than FLT_MAX value.

**5.127 GFLIB\_VLOG10\_T\_FLT**

Array of single precision floating point elements for storing the coefficients of the floating point log10 approximation polynomial.

**Source File**

```
#include <GFLIB_VLog10.h>
```

**Compound Type Members****Table 517.** GFLIB\_VLOG10\_T\_FLT members description

Type	Name	Description
tFloat	fltA	Array of single precision floating point elements for storing the coefficients of the floating point log10 approximation polynomial.

**5.128 GMCLIB\_DECOUPLINGPMSM\_T\_F16**

Structure containing coefficients for calculation of the decoupling.

**Source File**

```
#include <GMCLIB_DecouplingPMSM.h>
```

**Compound Type Members****Table 518.** GMCLIB\_DECOUPLINGPMSM\_T\_F16 members description

Type	Name	Description
tFrac16	f16Kd	Coefficient k_d.
tS16	s16KdShift	Scaling coefficient k_d_shift.
tFrac16	f16Kq	Coefficient k_q.
tS16	s16KqShift	Scaling coefficient k_q_shift.

**5.129 GMCLIB\_DECOUPLINGPMSM\_T\_F32**

Structure containing coefficients for calculation of the decoupling.

**Source File**

```
#include <GMCLIB_DcouplingPMSM.h>
```

**Compound Type Members****Table 519.** GMCLIB\_DCOUPLINGPMSM\_T\_F32 members description

Type	Name	Description
tFrac32	f32Kd	Coefficient $k_{df}$ .
tS16	s16KdShift	Scaling coefficient $k_{d\_shift}$ .
tFrac32	f32Kq	Coefficient $k_{qf}$ .
tS16	s16KqShift	Scaling coefficient $k_{q\_shift}$ .

**5.130 GMCLIB\_DCOUPLINGPMSM\_T\_FLT**

Structure containing coefficients for calculation of the decoupling.

**Source File**

```
#include <GMCLIB_DcouplingPMSM.h>
```

**Compound Type Members****Table 520.** GMCLIB\_DCOUPLINGPMSM\_T\_FLT members description

Type	Name	Description
tFloat	fLtLD	$L_D$ inductance [H].
tFloat	fLtLQ	$L_Q$ inductance [H].

**5.131 GMCLIB\_ELIMDCBUSRIP\_T\_F16**

Structure containing the PWM modulation index and the measured value of the DC bus voltage.

**Source File**

```
#include <GMCLIB_ElimDcBusRip.h>
```

**Compound Type Members****Table 521.** GMCLIB\_ELIMDCBUSRIP\_T\_F16 members description

Type	Name	Description
tFrac16	f16ArgDcBusMsr	Measured DC bus voltage.
tFrac16	f16ModIndex	Inverse Modulation Index.

**5.132 GMCLIB\_ELIMDCBUSRIP\_T\_F32**

Structure containing the PWM modulation index and the measured value of the DC bus voltage.

**Source File**

```
#include <GMCLIB_ElimDcBusRip.h>
```

**Compound Type Members****Table 522.** GMCLIB\_ELIMDCBUSRIP\_T\_F32 members description

Type	Name	Description
tFrac32	f32ArgDcBusMsr	Measured DC bus voltage.
tFrac32	f32ModIndex	Inverse Modulation Index.

**5.133 GMCLIB\_ELIMDCBUSRIP\_T\_FLT**

Structure containing the PWM modulation index and the measured value of the DC bus voltage.

**Source File**

```
#include <GMCLIB_ElimDcBusRip.h>
```

**Compound Type Members****Table 523.** GMCLIB\_ELIMDCBUSRIP\_T\_FLT members description

Type	Name	Description
tFloat	fltArgDcBusMsr	Measured DC bus voltage.
tFloat	fltModIndex	Inverse Modulation Index.

**5.134 SWLIBS\_2Syst\_F16**

Array of two standard 16-bit fractional arguments.

**Source File**

```
#include <SWLIBS_Typedefs.h>
```

**Compound Type Members****Table 524.** SWLIBS\_2Syst\_F16 members description

Type	Name	Description
tFrac16	f16Arg1	First argument
tFrac16	f16Arg2	Second argument

**5.135 SWLIBS\_2Syst\_F32**

Array of two standard 32-bit fractional arguments.

**Source File**

```
#include <SWLIBS_Typedefs.h>
```

**Compound Type Members****Table 525.** SWLIBS\_2Syst\_F32 members description

Type	Name	Description
tFrac32	f32Arg1	First argument
tFrac32	f32Arg2	Second argument

**5.136 SWLIBS\_2Syst\_FLT**

Array of two standard single precision floating point arguments.

**Source File**

```
#include <SWLIBS_Typedefs.h>
```

**Compound Type Members****Table 526.** SWLIBS\_2Syst\_FLT members description

Type	Name	Description
tFloat	fltArg1	First argument
tFloat	fltArg2	Second argument

**5.137 SWLIBS\_3Syst\_F16**

Array of three standard 16-bit fractional arguments.

**Source File**

```
#include <SWLIBS_Typedefs.h>
```

**Compound Type Members****Table 527.** SWLIBS\_3Syst\_F16 members description

Type	Name	Description
tFrac16	f16Arg1	First argument
tFrac16	f16Arg2	Second argument
tFrac16	f16Arg3	Third argument

**5.138 SWLIBS\_3Syst\_F32**

Array of three standard 32-bit fractional arguments.

**Source File**

```
#include <SWLIBS_Typedefs.h>
```

**Compound Type Members****Table 528.** SWLIBS\_3Syst\_F32 members description

Type	Name	Description
tFrac32	f32Arg1	First argument
tFrac32	f32Arg2	Second argument
tFrac32	f32Arg3	Third argument

**5.139 SWLIBS\_3Syst\_FLT**

Array of three standard single precision floating point arguments.

**Source File**

```
#include <SWLIBS_Typedefs.h>
```

**Compound Type Members****Table 529.** SWLIBS\_3Syst\_FLT members description

Type	Name	Description
tFloat	fltArg1	First argument
tFloat	fltArg2	Second argument
tFloat	fltArg3	Third argument

**5.140 SWLIBS\_VERSION\_T**

Motor Control Library Set identification structure.

**Source File**

```
#include <SWLIBS_Version.h>
```

**Compound Type Members****Table 530.** SWLIBS\_VERSION\_T members description

Type	Name	Description
unsigned char	mclid	MCLIB identification code
unsigned char	mcVersion	MCLIB version code
unsigned char	mclimpl	MCLIB supported implementation code

**6 Macros**

This section describes in details the macro definitions available in Automotive Math and Motor Control Library Set for NXP S32K14x devices.

**6.1 AMCLIB\_BEMF\_OBSRV\_DQ\_DEFAULT\_F32**

Default value for AMCLIB\_BEMF\_OBSRV\_DQ\_T\_F32.

### Source File

```
#include <AMCLIB_BemfObsrvDQ.h>
```

### Macro Definition

```
#define AMCLIB_BEMF_OBSRV_DQ_DEFAULT_F32 {(_tFrac32)0, (_tFrac32)0,  
\ (_tFrac32)0, (_tFrac32)0, \ (_tFrac32)0, (_tFrac32)0, (_tFrac32)0,  
(_tFrac32)0, _INT32_MIN, _INT32_MAX, (_tU16)0, \ (_tFrac32)0, (_tFrac32)0,  
(_tFrac32)0, (_tFrac32)0, _INT32_MIN, _INT32_MAX, (_tU16)0, \ (_tFrac32)0,  
(_tFrac32)0, \ (_tFrac16)0, (_tFrac16)0, \ (_tFrac32)0, (_tFrac32)0,  
(_tFrac32)0, (_tFrac32)0, (_ts16)0}
```

## 6.2 AMCLIB\_BEMF\_OBSRV\_DQ\_DEFAULT\_F16

Default value for AMCLIB\_BEMF\_OBSRV\_DQ\_T\_F16.

### Source File

```
#include <AMCLIB_BemfObsrvDQ.h>
```

### Macro Definition

```
#define AMCLIB_BEMF_OBSRV_DQ_DEFAULT_F16 {(_tFrac16)0, (_tFrac16)0,  
\ (_tFrac32)0, (_tFrac32)0, \ (_tFrac16)0, (_tFrac16)0, (_tFrac32)0,  
(_tFrac16)0, _INT16_MIN, _INT16_MAX, (_tU16)0, \ (_tFrac16)0, (_tFrac16)0,  
(_tFrac32)0, (_tFrac16)0, _INT16_MIN, _INT16_MAX, (_tU16)0, \ (_tFrac32)0,  
(_tFrac32)0, \ (_tFrac16)0, (_tFrac16)0, \ (_tFrac16)0, (_tFrac16)0,  
(_tFrac16)0, (_tFrac16)0, (_ts16)0}
```

## 6.3 AMCLIB\_BEMF\_OBSRV\_DQ\_DEFAULT\_FLT

Default value for AMCLIB\_BEMF\_OBSRV\_DQ\_T\_FLT.

### Source File

```
#include <AMCLIB_BemfObsrvDQ.h>
```

### Macro Definition

```
#define AMCLIB_BEMF_OBSRV_DQ_DEFAULT_FLT {(_tFloat)0, (_tFloat)0,  
\ (_tFloat)0, (_tFloat)0, \ (_tFloat)0, (_tFloat)0, (_tFloat)0,  
(_tFloat)0, _FLOAT_MIN, _FLOAT_MAX, \ (_tFloat)0, (_tFloat)0, (_tFloat)0,  
(_tFloat)0, _FLOAT_MIN, _FLOAT_MAX, \ (_tFloat)0, (_tFloat)0, \ (_tFloat)0,  
(_tFloat)0, (_tFloat)0, (_tFloat)0, (_tFloat)0}
```

## 6.4 AMCLIB\_CURRENT\_LOOP\_DEFAULT\_F32

Default value for AMCLIB\_CURRENT\_LOOP\_T\_F32.

### Source File

```
#include <AMCLIB_CurrentLoop.h>
```

### Macro Definition

```
#define AMCLIB_CURRENT_LOOP_DEFAULT_F32 { \ (tFrac32)0,  
(tFrac32)0, (ts32)0, (ts32)0, INT32_MIN, INT32_MAX, (tFrac32)0,  
(tFrac32)0, (tu16)0, \ (tFrac32)0, (tFrac32)0, (ts32)0,  
(ts32)0, INT32_MIN, INT32_MAX, (tFrac32)0, (tFrac32)0, (tu16)0, \  
(tFrac32)0, (tFrac32)0, \ (tFrac32)0, (tFrac32)0, }
```

## 6.5 AMCLIB\_CURRENT\_LOOP\_DEFAULT\_F16

Default value for AMCLIB\_CURRENT\_LOOP\_T\_F16.

### Source File

```
#include <AMCLIB_CurrentLoop.h>
```

### Macro Definition

```
#define AMCLIB_CURRENT_LOOP_DEFAULT_F16 { \ (tFrac16)0,  
(tFrac16)0, (ts16)0, (ts16)0, INT16_MIN, INT16_MAX, (tFrac32)0,  
(tFrac16)0, (tu16)0, \ (tFrac16)0, (tFrac16)0, (ts16)0,  
(ts16)0, INT16_MIN, INT16_MAX, (tFrac32)0, (tFrac16)0, (tu16)0, \  
(tFrac16)0, (tFrac16)0, \ (tFrac16)0, (tFrac16)0, }
```

## 6.6 AMCLIB\_CURRENT\_LOOP\_DEFAULT\_FLT

Default value for AMCLIB\_CURRENT\_LOOP\_T\_FLT.

### Source File

```
#include <AMCLIB_CurrentLoop.h>
```

### Macro Definition

```
#define AMCLIB_CURRENT_LOOP_DEFAULT_FLT { \ (tFloat)0,  
(tFloat)0, FLOAT_MIN, FLOAT_MAX, (tFloat)0, (tFloat)0, (tu16)0, \  
(tFloat)0, (tFloat)0, FLOAT_MIN, FLOAT_MAX, (tFloat)0, (tFloat)0,  
(tu16)0, \ (tFloat)0, (tFloat)0, \ (tFloat)0, (tFloat)0 }
```

## 6.7 AMCLIB\_FW\_DEFAULT\_F32

Default value for AMCLIB\_FW\_T\_F32.

### Source File

```
#include <AMCLIB_FW.h>
```

### Macro Definition

```
#define AMCLIB_FW_DEFAULT_F32 {0,0, \ (tFrac32)0, (tFrac32)0,  
(ts32)0, (ts32)0, INT32_MIN, INT32_MAX, (tFrac32)0, (tFrac32)0,  
(tu16)0, \ (tFrac32 *)0, \ (tFrac32 *)0, \ (tFrac32 *)0}
```

## 6.8 AMCLIB\_FW\_DEFAULT\_F16

Default value for AMCLIB\_FW\_T\_F16.

### Source File

```
#include <AMCLIB_FW.h>
```

### Macro Definition

```
#define AMCLIB_FW_DEFAULT_F16 {0,0, \ (tFrac16)0,(tFrac16)0,
(tS16)0,(tS16)0,INT16_MIN,INT16_MAX,(tFrac32)0,(tFrac16)0,
(tU16)0, \ (tFrac16 *)0, \ (tFrac16 *)0, \ (tFrac16 *)0}
```

## 6.9 AMCLIB\_FW\_DEFAULT\_FLT

Default value for AMCLIB\_FW\_T\_FLT.

### Source File

```
#include <AMCLIB_FW.h>
```

### Macro Definition

```
#define AMCLIB_FW_DEFAULT_FLT {0,0 \ (tFloat)0,
(tFloat)0,FLOAT_MIN,FLOAT_MAX,(tFloat)0,(tFloat)0,(tU16)0, \
(tFloat *)0, \ (tFloat *)0, \ (tFloat *)0}
```

## 6.10 AMCLIB\_FW\_SPEED\_LOOP\_DEFAULT\_F32

Default value for AMCLIB\_FW\_SPEED\_LOOP\_T\_F32.

### Source File

```
#include <AMCLIB_FWSpeedLoop.h>
```

### Macro Definition

```
#define AMCLIB_FW_SPEED_LOOP_DEFAULT_F32 {0,0, \ 0,0, \ 
(tFrac32)0,(tFrac32)0,(ts32)0,(ts32)0,INT32_MIN,INT32_MAX,
(tFrac32)0,(tFrac32)0,(tU16)0, \ (tFrac32)0,(tFrac32)0,(ts32)0,
(ts32)0,INT32_MIN,INT32_MAX,(tFrac32)0,(tFrac32)0,(tU16)0, \
(tFrac32)0,(tFrac32)0,(tFrac32)0, \ (tFrac32 *)0, \ (tFrac32 *)0,
\ (tFrac32 *)0}
```

## 6.11 AMCLIB\_FW\_SPEED\_LOOP\_DEFAULT\_F16

Default value for AMCLIB\_FW\_SPEED\_LOOP\_T\_F16.

### Source File

```
#include <AMCLIB_FWSpeedLoop.h>
```

**Macro Definition**

```
#define AMCLIB_FW_SPEED_LOOP_DEFAULT_F16 {0,0, \ 0,0, \
(tFrac16)0,(tFrac16)0,(ts16)0,(ts16)0,INT16_MIN,INT16_MAX, \
(tFrac32)0,(tFrac16)0,(tu16)0, \ (tFrac16)0,(tFrac16)0,(ts16)0, \
(ts16)0,INT16_MIN,INT16_MAX,(tFrac32)0,(tFrac16)0,(tu16)0, \
(tFrac32)0,(tFrac32)0, \ (tFrac16 *)0, \ (tFrac16 *)0, \
\ (tFrac16 *)0}
```

**6.12 AMCLIB\_FW\_SPEED\_LOOP\_DEFAULT\_FLT**

Default value for AMCLIB\_FW\_SPEED\_LOOP\_T\_FLT.

**Source File**

```
#include <AMCLIB_FWSpeedLoop.h>
```

**Macro Definition**

```
#define AMCLIB_FW_SPEED_LOOP_DEFAULT_FLT {0,0 \ 0,0 \ (tFloat)0, \
(tFloat)0,FLOAT_MIN,FLOAT_MAX,(tFloat)0,(tFloat)0,(tu16)0, \
(tFloat)0,(tFloat)0,FLOAT_MIN,FLOAT_MAX,(tFloat)0,(tFloat)0, \
(tu16)0, \ (tFloat)0,(tFloat)0,(tFloat)0, \ (tFloat *)0, \
(tFloat *)0, \ (tFloat *)0}
```

**6.13 AMCLIB\_SPEED\_LOOP\_DEFAULT\_F32**

Default value for AMCLIB\_SPEED\_LOOP\_T\_F32.

**Source File**

```
#include <AMCLIB_SpeedLoop.h>
```

**Macro Definition**

```
#define AMCLIB_SPEED_LOOP_DEFAULT_F32 {0,0, \ (tFrac32)0, \
(tFrac32)0,(ts32)0,(ts32)0,INT32_MIN,INT32_MAX,(tFrac32)0, \
(tFrac32)0,(tu16)0, \ (tFrac32)0,(tFrac32)0,(tFrac32)0}
```

**6.14 AMCLIB\_SPEED\_LOOP\_DEFAULT\_F16**

Default value for AMCLIB\_SPEED\_LOOP\_T\_F16.

**Source File**

```
#include <AMCLIB_SpeedLoop.h>
```

**Macro Definition**

```
#define AMCLIB_SPEED_LOOP_DEFAULT_F16 {0,0, \ (tFrac16)0, \
(tFrac16)0,(ts16)0,(ts16)0,INT16_MIN,INT16_MAX,(tFrac32)0, \
(tFrac16)0,(tu16)0, \ (tFrac32)0,(tFrac32)0,(tFrac32)0}
```

## 6.15 AMCLIB\_SPEED\_LOOP\_DEFAULT\_FLT

Default value for AMCLIB\_SPEED\_LOOP\_T\_FLT.

### Source File

```
#include <AMCLIB_SpeedLoop.h>
```

### Macro Definition

```
#define AMCLIB_SPEED_LOOP_DEFAULT_FLT {0,0 \ (tFloat)0,  
(tFloat)0,FLOAT_MIN,FLOAT_MAX,(tFloat)0,(tFloat)0,(tU16)0,\  
(tFloat)0,(tFloat)0,(tFloat)0}
```

## 6.16 AMCLIB\_TRACK\_OBSRV\_T

Definition of AMCLIB\_TRACK\_OBSRV\_T as alias for AMCLIB\_TRACK\_OBSRV\_T\_F16 datatype in case the 16-bit fractional implementation is selected.

### Source File

```
#include <AMCLIB_TrackObsrv.h>
```

### Macro Definition

```
#define AMCLIB_TRACK_OBSRV_T AMCLIB_TRACK_OBSRV_T_F16
```

## 6.17 AMCLIB\_TRACK\_OBSRV\_DEFAULT

Definition of AMCLIB\_TRACK\_OBSRV\_DEFAULT as alias for AMCLIB\_TRACK\_OBSRV\_DEFAULT\_F16 default value in case the 16-bit fractional implementation is selected.

### Source File

```
#include <AMCLIB_TrackObsrv.h>
```

### Macro Definition

```
#define AMCLIB_TRACK_OBSRV_DEFAULT AMCLIB_TRACK_OBSRV_DEFAULT_F16
```

## 6.18 AMCLIB\_TRACK\_OBSRV\_DEFAULT\_F32

Default value for AMCLIB\_TRACK\_OBSRV\_T\_F32.

### Source File

```
#include <AMCLIB_TrackObsrv.h>
```

### Macro Definition

```
#define AMCLIB_TRACK_OBSRV_DEFAULT_F32 { (tFrac32)0, (tFrac32)0,  
(tFrac32)0, (tFrac32)0, INT32_MIN, INT32_MAX, (tU16)0, \ (tFrac32)0,  
(tFrac32)0, (tFrac32)0, (tU16)0 }
```

## 6.19 AMCLIB\_TRACK\_OBSRV\_DEFAULT\_F16

Default value for AMCLIB\_TRACK\_OBSRV\_T\_F16.

### Source File

```
#include <AMCLIB_TrackObsrv.h>
```

### Macro Definition

```
#define AMCLIB_TRACK_OBSRV_DEFAULT_F16 { (tFrac16)0, (tFrac16)0,  
(tFrac32)0, (tFrac16)0, INT16_MIN, INT16_MAX, (tU16)0, \ (tFrac32)0,  
(tFrac16)0, (tFrac16)0, (tU16)0 }
```

## 6.20 AMCLIB\_TRACK\_OBSRV\_DEFAULT\_FLT

Default value for AMCLIB\_TRACK\_OBSRV\_T\_FLT.

### Source File

```
#include <AMCLIB_TrackObsrv.h>
```

### Macro Definition

```
#define AMCLIB_TRACK_OBSRV_DEFAULT_FLT { (tFloat)0, (tFloat)0,  
(tFloat)0, (tFloat)0, FLOAT_MIN, FLOAT_MAX, \ (tFloat)0, (tFloat)0,  
(tFloat)0 }
```

## 6.21 GDFLIB\_FILTERFIR\_PARAM\_T

Definition of alias for GDFLIB\_FILTERFIR\_PARAM\_T\_F16 datatype in case the 16-bit fractional implementation is selected.

### Source File

```
#include <GDFLIB_FilterFIR.h>
```

### Macro Definition

```
#define GDFLIB_FILTERFIR_PARAM_T GDFLIB_FILTERFIR_PARAM_T_F16
```

## 6.22 GDFLIB\_FILTERFIR\_STATE\_T

Definition of alias for GDFLIB\_FILTERFIR\_STATE\_T\_F16 datatype in case the 16-bit fractional implementation is selected.

**Source File**

```
#include <GDFLIB_FilterFIR.h>
```

**Macro Definition**

```
#define GDFLIB_FILTERFIR_STATE_T GDFLIB_FILTERFIR_STATE_T_F16
```

## 6.23 GDFLIB\_FILTER\_IIR1\_T

Definition of GDFLIB\_FILTER\_IIR1\_T as alias for GDFLIB\_FILTER\_IIR1\_T\_F16 datatype in case the 16-bit fractional implementation is selected.

**Source File**

```
#include <GDFLIB_FilterIIR1.h>
```

**Macro Definition**

```
#define GDFLIB_FILTER_IIR1_T GDFLIB_FILTER_IIR1_T_F16
```

## 6.24 GDFLIB\_FILTER\_IIR1\_DEFAULT

Definition of GDFLIB\_FILTER\_IIR1\_DEFAULT as alias for GDFLIB\_FILTER\_IIR1\_DEFAULT\_F16 default value in case the 16-bit fractional implementation is selected.

**Source File**

```
#include <GDFLIB_FilterIIR1.h>
```

**Macro Definition**

```
#define GDFLIB_FILTER_IIR1_DEFAULT GDFLIB\_FILTER\_IIR1\_DEFAULT\_F16
```

## 6.25 GDFLIB\_FILTER\_IIR1\_DEFAULT\_F32

Default value for GDFLIB\_FILTER\_IIR1\_T\_F32.

**Source File**

```
#include <GDFLIB_FilterIIR1.h>
```

**Macro Definition**

```
#define GDFLIB_FILTER_IIR1_DEFAULT_F32 {{\(tFrac32\) 0, \(tFrac32\) 0,  
\(tFrac32\) 0}, {\(tFrac32\) 0}, {\(tFrac32\) 0}}
```

## 6.26 GDFLIB\_FILTER\_IIR1\_DEFAULT\_F16

Default value for GDFLIB\_FILTER\_IIR1\_T\_F16.

**Source File**

```
#include <GDFLIB_FilterIIR1.h>
```

**Macro Definition**

```
#define GDFLIB_FILTER_IIR1_DEFAULT_F16 {{(tFrac16)0,(tFrac16)0,(tFrac16)0},{(tFrac16)0},{(tFrac16)0}}
```

## 6.27 GDFLIB\_FILTER\_IIR1\_DEFAULT\_FLT

Default value for GDFLIB\_FILTER\_IIR1\_T\_FLT.

**Source File**

```
#include <GDFLIB_FilterIIR1.h>
```

**Macro Definition**

```
#define GDFLIB_FILTER_IIR1_DEFAULT_FLT {{(tFloat)0,(tFloat)0,(tFloat)0},{(tFloat)0},{(tFloat)0}}
```

## 6.28 GDFLIB\_FILTER\_IIR2\_T

Definition of GDFLIB\_FILTER\_IIR2\_T as alias for GDFLIB\_FILTER\_IIR2\_T\_F16 datatype in case the 16-bit fractional implementation is selected.

**Source File**

```
#include <GDFLIB_FilterIIR2.h>
```

**Macro Definition**

```
#define GDFLIB_FILTER_IIR2_T GDFLIB_FILTER_IIR2_T_F16
```

## 6.29 GDFLIB\_FILTER\_IIR2\_DEFAULT

Definition of GDFLIB\_FILTER\_IIR2\_DEFAULT GDFLIB\_FILTER\_IIR2\_DEFAULT\_F16 default value in case the 16-bit fractional implementation is selected.

**Source File**

```
#include <GDFLIB_FilterIIR2.h>
```

**Macro Definition**

```
#define GDFLIB_FILTER_IIR2_DEFAULT GDFLIB_FILTER_IIR2_DEFAULT_F16
```

## 6.30 GDFLIB\_FILTER\_IIR2\_DEFAULT\_F32

Default value for GDFLIB\_FILTER\_IIR2\_T\_F32.

### Source File

```
#include <GDFLIB_FilterIIR2.h>
```

### Macro Definition

```
#define GDFLIB_FILTER_IIR2_DEFAULT_F32 {{(tFrac32)0,(tFrac32)0,(tFrac32)0,(tFrac32)0,(tFrac32)0,(tFrac32)0},{(tFrac32)0,(tFrac32)0,(tFrac32)0,(tFrac32)0,(tFrac32)0,(tFrac32)0}}
```

## 6.31 GDFLIB\_FILTER\_IIR2\_DEFAULT\_F16

Default value for GDFLIB\_FILTER\_IIR2\_T\_F16.

### Source File

```
#include <GDFLIB_FilterIIR2.h>
```

### Macro Definition

```
#define GDFLIB_FILTER_IIR2_DEFAULT_F16 {{(tFrac16)0,(tFrac16)0,(tFrac16)0,(tFrac16)0,(tFrac16)0,(tFrac16)0},{(tFrac16)0,(tFrac16)0,(tFrac16)0,(tFrac16)0,(tFrac16)0,(tFrac16)0}}
```

## 6.32 GDFLIB\_FILTER\_IIR2\_DEFAULT\_FLT

Default value for GDFLIB\_FILTER\_IIR2\_T\_FLT.

### Source File

```
#include <GDFLIB_FilterIIR2.h>
```

### Macro Definition

```
#define GDFLIB_FILTER_IIR2_DEFAULT_FLT {{(tFloat)0,(tFloat)0,(tFloat)0,(tFloat)0,(tFloat)0,(tFloat)0},{(tFloat)0,(tFloat)0,(tFloat)0,(tFloat)0,(tFloat)0,(tFloat)0}}
```

## 6.33 GDFLIB\_FILTER\_MA\_T

Definition of GDFLIB\_FILTER\_MA\_T as alias for GDFLIB\_FILTER\_MA\_T\_F16 datatype in case the 16-bit fractional implementation is selected.

### Source File

```
#include <GDFLIB_FilterMA.h>
```

### Macro Definition

```
#define GDFLIB_FILTER_MA_T GDFLIB_FILTER_MA_T_F16
```

### 6.34 GDFLIB\_FILTER\_MA\_DEFAULT

Definition of GDFLIB\_FILTER\_MA\_DEFAULT as alias for GDFLIB\_FILTER\_MA\_DEFAULT\_F16 default value in case the 16-bit fractional implementation is selected.

#### Source File

```
#include <GDFLIB_FilterMA.h>
```

#### Macro Definition

```
#define GDFLIB_FILTER_MA_DEFAULT GDFLIB_FILTER_MA_DEFAULT_F16
```

### 6.35 GDFLIB\_FILTER\_MA\_DEFAULT\_F32

Default value for GDFLIB\_FILTER\_MA\_T\_F32.

#### Source File

```
#include <GDFLIB_FilterMA.h>
```

#### Macro Definition

```
#define GDFLIB_FILTER_MA_DEFAULT_F32 {0,0}
```

### 6.36 GDFLIB\_FILTER\_MA\_DEFAULT\_F16

Default value for GDFLIB\_FILTER\_MA\_T\_F16.

#### Source File

```
#include <GDFLIB_FilterMA.h>
```

#### Macro Definition

```
#define GDFLIB_FILTER_MA_DEFAULT_F16 {0,0}
```

### 6.37 GDFLIB\_FILTER\_MA\_DEFAULT\_FLT

Default value for GDFLIB\_FILTER\_MA\_T\_FLT.

#### Source File

```
#include <GDFLIB_FilterMA.h>
```

#### Macro Definition

```
#define GDFLIB_FILTER_MA_DEFAULT_FLT {0,0}
```

### 6.38 **GFLIB\_ACOS\_T**

Definition of GFLIB\_ACOS\_T as alias for GFLIB\_ACOS\_T\_F16 datatype in case the 16-bit fractional implementation is selected.

#### Source File

```
#include <GFLIB_Acos.h>
```

#### Macro Definition

```
#define GFLIB_ACOS_T GFLIB_ACOS_T_F16
```

### 6.39 **GFLIB\_ACOS\_DEFAULT**

Definition of GFLIB\_ACOS\_DEFAULT as alias for GFLIB\_ACOS\_DEFAULT\_F16 default value in case the 16-bit fractional implementation is selected.

#### Source File

```
#include <GFLIB_Acos.h>
```

#### Macro Definition

```
#define GFLIB_ACOS_DEFAULT GFLIB_ACOS_DEFAULT_F16
```

### 6.40 **GFLIB\_ACOS\_DEFAULT\_F32**

Default approximation coefficients for GFLIB\_Acos\_F32 function.

#### Source File

```
#include <GFLIB_Acos.h>
```

#### Macro Definition

```
#define GFLIB_ACOS_DEFAULT_F32 &f32gflibAcosCoef
```

### 6.41 **GFLIB\_ACOS\_DEFAULT\_F16**

Default approximation coefficients for GFLIB\_Acos\_F16 function.

#### Source File

```
#include <GFLIB_Acos.h>
```

#### Macro Definition

```
#define GFLIB_ACOS_DEFAULT_F16 &f16gflibAcosCoef
```

### 6.42 **GFLIB\_ACOS\_DEFAULT\_FLT**

Default approximation coefficients for GFLIB\_Acos\_FLT function.

**Source File**

```
#include <GFLIB_Acos.h>
```

**Macro Definition**

```
#define GFLIB_ACOS_DEFAULT_FLT &fltgflibAcosCoef
```

## 6.43 GFLIB\_ASIN\_FLT\_MIN

Floating-point min. input value for computation.

**Source File**

```
#include <GFLIB_Asin.c>
```

**Macro Definition**

```
#define GFLIB_ASIN_FLT_MIN ((tFloat)0.005)
```

## 6.44 GFLIB\_ASIN\_FLT\_INT1

Floating-point min. input value for computation in interval 1.

**Source File**

```
#include <GFLIB_Asin.c>
```

**Macro Definition**

```
#define GFLIB_ASIN_FLT_INT1 ((tFloat)0.41)
```

## 6.45 GFLIB\_ASIN\_T

Definition of GFLIB\_ASIN\_T as alias for GFLIB\_ASIN\_T\_F16 datatype in case the 16-bit fractional implementation is selected.

**Source File**

```
#include <GFLIB_Asin.h>
```

**Macro Definition**

```
#define GFLIB_ASIN_T GFLIB_ASIN_T_F16
```

## 6.46 GFLIB\_ASIN\_DEFAULT

Definition of GFLIB\_ASIN\_DEFAULT as alias for GFLIB\_ASIN\_DEFAULT\_F16 default value in case the 16-bit fractional implementation is selected.

**Source File**

```
#include <GFLIB_Asin.h>
```

**Macro Definition**

```
#define GFLIB_ASIN_DEFAULT GFLIB_ASIN_DEFAULT_F16
```

**6.47 GFLIB\_ASIN\_DEFAULT\_F32**

Default approximation coefficients for GFLIB\_Asin\_F32 function.

**Source File**

```
#include <GFLIB_Asin.h>
```

**Macro Definition**

```
#define GFLIB_ASIN_DEFAULT_F32 &f32gflibAsinCoef
```

**6.48 GFLIB\_ASIN\_DEFAULT\_F16**

Default approximation coefficients for GFLIB\_Asin\_F16 function.

**Source File**

```
#include <GFLIB_Asin.h>
```

**Macro Definition**

```
#define GFLIB_ASIN_DEFAULT_F16 &f16gflibAsinCoef
```

**6.49 GFLIB\_ASIN\_DEFAULT\_FLT**

Default approximation coefficients for GFLIB\_Asin\_FLT function.

**Source File**

```
#include <GFLIB_Asin.h>
```

**Macro Definition**

```
#define GFLIB_ASIN_DEFAULT_FLT &fltgflibAsinCoef
```

**6.50 GFLIB\_ATAN\_T**

Definition of GFLIB\_ATAN\_T as alias for GFLIB\_ATAN\_T\_F16 datatype in case the 16-bit fractional implementation is selected.

**Source File**

```
#include <GFLIB_Atan.h>
```

**Macro Definition**

```
#define GFLIB_ATAN_T GFLIB_ATAN_T_F16
```

## 6.51 GFLIB\_ATAN\_DEFAULT

Definition of GFLIB\_ATAN\_DEFAULT as alias for GFLIB\_ATAN\_DEFAULT\_F16 default value in case the 16-bit fractional implementation is selected.

### Source File

```
#include <GFLIB_Atan.h>
```

### Macro Definition

```
#define GFLIB_ATAN_DEFAULT GFLIB_ATAN_DEFAULT_F16
```

## 6.52 GFLIB\_ATAN\_DEFAULT\_F32

Default approximation coefficients for GFLIB\_Atan\_F32 function.

### Source File

```
#include <GFLIB_Atan.h>
```

### Macro Definition

```
#define GFLIB_ATAN_DEFAULT_F32 &f32gflibAtanCoef
```

## 6.53 GFLIB\_ATAN\_DEFAULT\_F16

Default approximation coefficients for GFLIB\_Atan\_F16 function.

### Source File

```
#include <GFLIB_Atan.h>
```

### Macro Definition

```
#define GFLIB_ATAN_DEFAULT_F16 &f16gflibAtanCoef
```

## 6.54 GFLIB\_ATAN\_DEFAULT\_FLT

Default approximation coefficients for GFLIB\_Atan\_FLT function.

### Source File

```
#include <GFLIB_Atan.h>
```

### Macro Definition

```
#define GFLIB_ATAN_DEFAULT_FLT &fltgflibAtanCoef
```

## 6.55 GFLIB\_ATANYXSHIFTED\_T

Definition of GFLIB\_ATANYXSHIFTED\_T as alias for GFLIB\_ATANYXSHIFTED\_T\_F16 datatype in case the 16-bit fractional implementation is selected.

**Source File**

```
#include <GFLIB_AtanYXShifted.h>
```

**Macro Definition**

```
#define GFLIB_ATANYXSHIFTED_T GFLIB_ATANYXSHIFTED_T_F16
```

## 6.56 GFLIB\_CONTROLLER\_PID\_P\_AW\_T

Definition of GFLIB\_CONTROLLER\_PID\_P\_AW\_T as alias for GFLIB\_CONTROLLER\_PID\_P\_AW\_T\_F16 datatype in case the 16-bit fractional implementation is selected.

**Source File**

```
#include <GFLIB_ControllerPIDpAW.h>
```

**Macro Definition**

```
#define GFLIB_CONTROLLER_PID_P_AW_T  
GFLIB_CONTROLLER_PID_P_AW_T_F16
```

## 6.57 GFLIB\_CONTROLLER\_PID\_P\_AW\_DEFAULT

Definition of GFLIB\_CONTROLLER\_PID\_P\_AW\_DEFAULT as alias for GFLIB\_CONTROLLER\_PID\_P\_AW\_DEFAULT\_F16 default value in case the 16-bit fractional implementation is selected.

**Source File**

```
#include <GFLIB_ControllerPIDpAW.h>
```

**Macro Definition**

```
#define GFLIB_CONTROLLER_PID_P_AW_DEFAULT  
GFLIB_CONTROLLER_PID_P_AW_DEFAULT_F16
```

## 6.58 GFLIB\_CONTROLLER\_PID\_P\_AW\_DEFAULT\_F32

Default value for GFLIB\_CONTROLLER\_PID\_P\_AW\_T\_F32.

**Source File**

```
#include <GFLIB_ControllerPIDpAW.h>
```

**Macro Definition**

```
#define GFLIB_CONTROLLER_PID_P_AW_DEFAULT_F32 { (tFrac32) 0,  
(tFrac32) 0, (tFrac32) 0, (tFrac32) 0, (ts16) 0, (ts16) 0, (ts16) 0,  
(tFrac32) 0, (tFrac32) 0, (tFrac32) 0, (tFrac32) 0, (tU16) 0 }
```

## 6.59 GFLIB\_CONTROLLER\_PID\_P\_AW\_DEFAULT\_F16

Default value for GFLIB\_CONTROLLER\_PID\_P\_AW\_T\_F16.

### Source File

```
#include <GFLIB_ControllerPIDpAW.h>
```

### Macro Definition

```
#define GFLIB_CONTROLLER_PID_P_AW_DEFAULT_F16 { (tFrac16) 0,  
                                              (tFrac16) 0, (tFrac16) 0, (tFrac16) 0, (ts16) 0, (ts16) 0,  
                                              (tFrac16) 0, (tFrac16) 0, (tFrac32) 0, (tFrac16) 0, (tu16) 0 }
```

## 6.60 GFLIB\_CONTROLLER\_PID\_P\_AW\_DEFAULT\_FLT

Default value for GFLIB\_CONTROLLER\_PID\_P\_AW\_T\_FLT.

### Source File

```
#include <GFLIB_ControllerPIDpAW.h>
```

### Macro Definition

```
#define GFLIB_CONTROLLER_PID_P_AW_DEFAULT_FLT { (tFloat) 0,  
                                              (tFloat) 0, (tFloat) 0, (tFloat) 0, (tFloat) 0, (tFloat) 0,  
                                              (tFloat) 0, (tFloat) 0, (tu16) 0 }
```

## 6.61 GFLIB\_CONTROLLER\_PI\_P\_T

Definition of GFLIB\_CONTROLLER\_PI\_P\_T as alias for  
GFLIB\_CONTROLLER\_PI\_P\_T\_F16 datatype in case the 16-bit fractional  
implementation is selected.

### Source File

```
#include <GFLIB_ControllerPIp.h>
```

### Macro Definition

```
#define GFLIB_CONTROLLER_PI_P_T GFLIB_CONTROLLER_PI_P_T_F16
```

## 6.62 GFLIB\_CONTROLLER\_PI\_P\_DEFAULT

Definition of GFLIB\_CONTROLLER\_PI\_P\_DEFAULT as alias for  
GFLIB\_CONTROLLER\_PI\_P\_DEFAULT\_F16 default value in case the 16-bit fractional  
implementation is selected.

### Source File

```
#include <GFLIB_ControllerPIp.h>
```

**Macro Definition**

```
#define GFLIB_CONTROLLER_PI_P_DEFAULT  
GFLIB_CONTROLLER_PI_P_DEFAULT_F16
```

**6.63 GFLIB\_CONTROLLER\_PI\_P\_DEFAULT\_F32**

Default value for GFLIB\_CONTROLLER\_PI\_P\_T\_F32.

**Source File**

```
#include <GFLIB_ControllerPIp.h>
```

**Macro Definition**

```
#define GFLIB_CONTROLLER_PI_P_DEFAULT_F32 { (tFrac32) 0, (tFrac32) 0,  
(tS16) 0, (tS16) 0, (tFrac32) 0, (tFrac32) 0 }
```

**6.64 GFLIB\_CONTROLLER\_PI\_P\_DEFAULT\_F16**

Default value for GFLIB\_CONTROLLER\_PI\_P\_T\_F16.

**Source File**

```
#include <GFLIB_ControllerPIp.h>
```

**Macro Definition**

```
#define GFLIB_CONTROLLER_PI_P_DEFAULT_F16 { (tFrac16) 0, (tFrac16) 0,  
(tS16) 0, (tS16) 0, (tFrac32) 0, (tFrac16) 0 }
```

**6.65 GFLIB\_CONTROLLER\_PI\_P\_DEFAULT\_FLT**

Default value for GFLIB\_CONTROLLER\_PI\_P\_T\_FLT.

**Source File**

```
#include <GFLIB_ControllerPIp.h>
```

**Macro Definition**

```
#define GFLIB_CONTROLLER_PI_P_DEFAULT_FLT { (tFloat) 0, (tFloat) 0,  
(tFloat) 0, (tFloat) 0 }
```

**6.66 GFLIB\_CONTROLLER\_PIAW\_P\_T**

Definition of GFLIB\_CONTROLLER\_PIAW\_P\_T as alias for  
GFLIB\_CONTROLLER\_PIAW\_P\_T\_F16 datatype in case the 16-bit fractional  
implementation is selected.

**Source File**

```
#include <GFLIB_ControllerPIpAW.h>
```

### Macro Definition

```
#define GFLIB_CONTROLLER_PIAW_P_T GFLIB_CONTROLLER_PIAW_P_T_F16
```

## 6.67 GFLIB\_CONTROLLER\_PIAW\_P\_DEFAULT

Definition of GFLIB\_CONTROLLER\_PIAW\_P\_DEFAULT as alias for GFLIB\_CONTROLLER\_PIAW\_P\_DEFAULT\_F16 default value in case the 16-bit fractional implementation is selected.

### Source File

```
#include <GFLIB_ControllerPIpAW.h>
```

### Macro Definition

```
#define GFLIB_CONTROLLER_PIAW_P_DEFAULT  
GFLIB_CONTROLLER_PIAW_P_DEFAULT_F16
```

## 6.68 GFLIB\_CONTROLLER\_PIAW\_P\_DEFAULT\_F32

Default value for GFLIB\_CONTROLLER\_PIAW\_P\_T\_F32.

### Source File

```
#include <GFLIB_ControllerPIpAW.h>
```

### Macro Definition

```
#define GFLIB_CONTROLLER_PIAW_P_DEFAULT_F32 { (tFrac32)0,  
(tFrac32)0, (ts32)0, (ts32)0, INT32_MIN, INT32_MAX, (tFrac32)0,  
(tFrac32)0, (tu16)0 }
```

## 6.69 GFLIB\_CONTROLLER\_PIAW\_P\_DEFAULT\_F16

Default value for GFLIB\_CONTROLLER\_PIAW\_P\_T\_F16.

### Source File

```
#include <GFLIB_ControllerPIpAW.h>
```

### Macro Definition

```
#define GFLIB_CONTROLLER_PIAW_P_DEFAULT_F16 { (tFrac16)0,  
(tFrac16)0, (ts16)0, (ts16)0, INT16_MIN, INT16_MAX, (tFrac32)0,  
(tFrac16)0, (tu16)0 }
```

## 6.70 GFLIB\_CONTROLLER\_PIAW\_P\_DEFAULT\_FLT

Default value for GFLIB\_CONTROLLER\_PIAW\_P\_T\_FLT.

### Source File

```
#include <GFLIB_ControllerPIpAW.h>
```

### Macro Definition

```
#define GFLIB_CONTROLLER_PIAW_P_DEFAULT_FLT { (tFloat)0,  
(tFloat)0, FLOAT_MIN, FLOAT_MAX, (tFloat)0, (tFloat)0, (tU16)0 }
```

## 6.71 GFLIB\_CONTROLLER\_PI\_R\_T

Definition of GFLIB\_CONTROLLER\_PI\_R\_T as alias for GFLIB\_CONTROLLER\_PI\_R\_T\_F16 datatype in case the 16-bit fractional implementation is selected.

### Source File

```
#include <GFLIB_ControllerPIr.h>
```

### Macro Definition

```
#define GFLIB_CONTROLLER_PI_R_T GFLIB_CONTROLLER_PI_R_T_F16
```

## 6.72 GFLIB\_CONTROLLER\_PI\_R\_DEFAULT

Definition of GFLIB\_CONTROLLER\_PI\_R\_DEFAULT as alias for GFLIB\_CONTROLLER\_PI\_R\_DEFAULT\_F16 default value in case the 16-bit fractional implementation is selected.

### Source File

```
#include <GFLIB_ControllerPIr.h>
```

### Macro Definition

```
#define GFLIB_CONTROLLER_PI_R_DEFAULT  
GFLIB_CONTROLLER_PI_R_DEFAULT_F16
```

## 6.73 GFLIB\_CONTROLLER\_PI\_R\_DEFAULT\_F32

Default value for GFLIB\_CONTROLLER\_PI\_R\_T\_F32.

### Source File

```
#include <GFLIB_ControllerPIr.h>
```

### Macro Definition

```
#define GFLIB_CONTROLLER_PI_R_DEFAULT_F32 { (tFrac32)0, (tFrac32)0,  
(tFrac32)0, (tFrac32)0, (tU16)0 }
```

## 6.74 GFLIB\_CONTROLLER\_PI\_R\_DEFAULT\_F16

Default value for GFLIB\_CONTROLLER\_PI\_R\_T\_F16.

### Source File

```
#include <GFLIB_ControllerPIr.h>
```

### Macro Definition

```
#define GFLIB_CONTROLLER_PI_R_DEFAULT_F16 {(_tFrac16)0, (_tFrac16)0,  
(_tFrac32)0, (_tFrac16)0, (_tU16)0}
```

## 6.75 GFLIB\_CONTROLLER\_PI\_R\_DEFAULT\_FLT

Default value for GFLIB\_CONTROLLER\_PI\_R\_T\_FLT.

### Source File

```
#include <GFLIB_ControllerPIr.h>
```

### Macro Definition

```
#define GFLIB_CONTROLLER_PI_R_DEFAULT_FLT {(_tFloat)0, (_tFloat)0,  
(_tFloat)0, (_tFloat)0}
```

## 6.76 GFLIB\_CONTROLLER\_PIAW\_R\_T

Definition of GFLIB\_CONTROLLER\_PIAW\_R\_T as alias for GFLIB\_CONTROLLER\_PIAW\_R\_T\_F16 datatype in case the 16-bit fractional implementation is selected.

### Source File

```
#include <GFLIB_ControllerPIrAW.h>
```

### Macro Definition

```
#define GFLIB_CONTROLLER_PIAW_R_T GFLIB_CONTROLLER_PIAW_R_T_F16
```

## 6.77 GFLIB\_CONTROLLER\_PIAW\_R\_DEFAULT

Definition of GFLIB\_CONTROLLER\_PIAW\_R\_DEFAULT as alias for GFLIB\_CONTROLLER\_PIAW\_R\_DEFAULT\_F16 default value in case the 16-bit fractional implementation is selected.

### Source File

```
#include <GFLIB_ControllerPIrAW.h>
```

**Macro Definition**

```
#define GFLIB_CONTROLLER_PIAW_R_DEFAULT  
GFLIB_CONTROLLER_PIAW_R_DEFAULT_F16
```

**6.78 GFLIB\_CONTROLLER\_PIAW\_R\_DEFAULT\_F32**

Default value for GFLIB\_CONTROLLER\_PIAW\_R\_T\_F32.

**Source File**

```
#include <GFLIB_ControllerPIrAW.h>
```

**Macro Definition**

```
#define GFLIB_CONTROLLER_PIAW_R_DEFAULT_F32 { (tFrac32) 0,  
(tFrac32) 0, (tFrac32) 0, (tFrac32) 0, INT32_MIN, INT32_MAX, (tU16) 0 }
```

**6.79 GFLIB\_CONTROLLER\_PIAW\_R\_DEFAULT\_F16**

Default value for GFLIB\_CONTROLLER\_PIAW\_R\_T\_F16.

**Source File**

```
#include <GFLIB_ControllerPIrAW.h>
```

**Macro Definition**

```
#define GFLIB_CONTROLLER_PIAW_R_DEFAULT_F16 { (tFrac16) 0,  
(tFrac16) 0, (tFrac32) 0, (tFrac16) 0, INT16_MIN, INT16_MAX, (tU16) 0 }
```

**6.80 GFLIB\_CONTROLLER\_PIAW\_R\_DEFAULT\_FLT**

Default value for GFLIB\_CONTROLLER\_PIAW\_R\_T\_FLT.

**Source File**

```
#include <GFLIB_ControllerPIrAW.h>
```

**Macro Definition**

```
#define GFLIB_CONTROLLER_PIAW_R_DEFAULT_FLT { (tFloat) 0, (tFloat) 0,  
(tFloat) 0, (tFloat) 0, FLOAT_MIN, FLOAT_MAX }
```

**6.81 GFLIB\_COS\_T**

Definition of GFLIB\_COS\_T as alias for GFLIB\_COS\_T\_F16 datatype in case the 16-bit fractional implementation is selected.

**Source File**

```
#include <GFLIB_Cos.h>
```

**Macro Definition**

```
#define GFLIB_COS_T GFLIB_COS_T_F16
```

**6.82 GFLIB\_COS\_DEFAULT**

Definition of GFLIB\_COS\_DEFAULT as alias for GFLIB\_COS\_DEFAULT\_F16 default value in case the 16-bit fractional implementation is selected.

**Source File**

```
#include <GFLIB_Cos.h>
```

**Macro Definition**

```
#define GFLIB_COS_DEFAULT GFLIB_COS_DEFAULT_F16
```

**6.83 GFLIB\_COS\_DEFAULT\_F32**

Default approximation coefficients for GFLIB\_Cos\_F32 function.

**Source File**

```
#include <GFLIB_Cos.h>
```

**Macro Definition**

```
#define GFLIB_COS_DEFAULT_F32 &f32gflibCosCoef
```

**6.84 GFLIB\_COS\_DEFAULT\_F16**

Default approximation coefficients for GFLIB\_Cos\_F32 function.

**Source File**

```
#include <GFLIB_Cos.h>
```

**Macro Definition**

```
#define GFLIB_COS_DEFAULT_F16 &f16gflibCosCoef
```

**6.85 GFLIB\_COS\_DEFAULT\_FLT**

Default approximation coefficients for GFLIB\_Cos\_FLT function.

**Source File**

```
#include <GFLIB_Cos.h>
```

**Macro Definition**

```
#define GFLIB_COS_DEFAULT_FLT &fltgflibCosCoef
```

## 6.86 GFLIB\_HYST\_T

Definition of GFLIB\_HYST\_T as alias for GFLIB\_HYST\_T\_F16 datatype in case the 16-bit fractional implementation is selected.

### Source File

```
#include <GFLIB_Hyst.h>
```

### Macro Definition

```
#define GFLIB_HYST_T GFLIB_HYST_T_F16
```

## 6.87 GFLIB\_HYST\_DEFAULT

Definition of GFLIB\_HYST\_DEFAULT as alias for GFLIB\_HYST\_DEFAULT\_F16 default value in case the 16-bit fractional implementation is selected.

### Source File

```
#include <GFLIB_Hyst.h>
```

### Macro Definition

```
#define GFLIB_HYST_DEFAULT GFLIB_HYST_DEFAULT_F16
```

## 6.88 GFLIB\_HYST\_DEFAULT\_F32

Default value for GFLIB\_HYST\_T\_F32.

### Source File

```
#include <GFLIB_Hyst.h>
```

### Macro Definition

```
#define GFLIB_HYST_DEFAULT_F32 { (tFrac32) 0, (tFrac32) 0, (tFrac32) 0,  
(tFrac32) 0, (tFrac32) 0 }
```

## 6.89 GFLIB\_HYST\_DEFAULT\_F16

Default value for GFLIB\_HYST\_T\_F16.

### Source File

```
#include <GFLIB_Hyst.h>
```

### Macro Definition

```
#define GFLIB_HYST_DEFAULT_F16 { (tFrac16) 0, (tFrac16) 0, (tFrac16) 0,  
(tFrac16) 0, (tFrac16) 0 }
```

## 6.90 GFLIB\_HYST\_DEFAULT\_FLT

Default value for GFLIB\_HYST\_T\_FLT.

### Source File

```
#include <GFLIB_Hyst.h>
```

### Macro Definition

```
#define GFLIB_HYST_DEFAULT_FLT { (tFloat) 0, (tFloat) 0, (tFloat) 0,  
                                (tFloat) 0, (tFloat) 0 }
```

## 6.91 GFLIB\_INTEGRATOR\_TR\_T

Definition of GFLIB\_INTEGRATOR\_TR\_T as alias for GFLIB\_INTEGRATOR\_TR\_T\_F16 datatype in case the 16-bit fractional implementation is selected.

### Source File

```
#include <GFLIB_IntegratorTR.h>
```

### Macro Definition

```
#define GFLIB_INTEGRATOR_TR_T GFLIB_INTEGRATOR_TR_T_F16
```

## 6.92 GFLIB\_INTEGRATOR\_TR\_DEFAULT

Definition of GFLIB\_INTEGRATOR\_TR\_DEFAULT as alias for GFLIB\_INTEGRATOR\_TR\_DEFAULT\_F16 default value in case the 16-bit fractional implementation is selected.

### Source File

```
#include <GFLIB_IntegratorTR.h>
```

### Macro Definition

```
#define GFLIB_INTEGRATOR_TR_DEFAULT  
GFLIB_INTEGRATOR_TR_DEFAULT_F16
```

## 6.93 GFLIB\_INTEGRATOR\_TR\_DEFAULT\_F32

Default value for GFLIB\_INTEGRATOR\_TR\_T\_F32.

### Source File

```
#include <GFLIB_IntegratorTR.h>
```

### Macro Definition

```
#define GFLIB_INTEGRATOR_TR_DEFAULT_F32 { (tFrac32) 0, (tFrac32) 0,  
                                         (tFrac32) 0, (tU16) 0 }
```

## 6.94 GFLIB\_INTEGRATOR\_TR\_DEFAULT\_F16

Default value for GFLIB\_INTEGRATOR\_TR\_T\_F16.

### Source File

```
#include <GFLIB_IntegratorTR.h>
```

### Macro Definition

```
#define GFLIB_INTEGRATOR_TR_DEFAULT_F16 { (tFrac32) 0, (tFrac16) 0,  
(tFrac16) 0, (tU16) 0 }
```

## 6.95 GFLIB\_INTEGRATOR\_TR\_DEFAULT\_FLT

Default value for GFLIB\_INTEGRATOR\_TR\_T\_FLT.

### Source File

```
#include <GFLIB_IntegratorTR.h>
```

### Macro Definition

```
#define GFLIB_INTEGRATOR_TR_DEFAULT_FLT { (tFloat) 0, (tFloat) 0,  
(tFloat) 0 }
```

## 6.96 GFLIB\_LIMIT\_T

Definition of GFLIB\_LIMIT\_T as alias for GFLIB\_LIMIT\_T\_F16 datatype in case the 16-bit fractional implementation is selected.

### Source File

```
#include <GFLIB_Limit.h>
```

### Macro Definition

```
#define GFLIB_LIMIT_T GFLIB_LIMIT_T_F16
```

## 6.97 GFLIB\_LIMIT\_DEFAULT

Definition of GFLIB\_LIMIT\_DEFAULT as alias for GFLIB\_LIMIT\_DEFAULT\_F16 default value in case the 16-bit fractional implementation is selected.

### Source File

```
#include <GFLIB_Limit.h>
```

### Macro Definition

```
#define GFLIB_LIMIT_DEFAULT GFLIB_LIMIT_DEFAULT_F16
```

## 6.98 GFLIB\_LIMIT\_DEFAULT\_F32

Default value for GFLIB\_LIMIT\_T\_F32.

### Source File

```
#include <GFLIB_Limit.h>
```

### Macro Definition

```
#define GFLIB_LIMIT_DEFAULT_F32 {INT32_MIN, INT32_MAX}
```

## 6.99 GFLIB\_LIMIT\_DEFAULT\_F16

Default value for GFLIB\_LIMIT\_T\_F16.

### Source File

```
#include <GFLIB_Limit.h>
```

### Macro Definition

```
#define GFLIB_LIMIT_DEFAULT_F16 {INT16_MIN, INT16_MAX}
```

## 6.100 GFLIB\_LIMIT\_DEFAULT\_FLT

Default value for GFLIB\_LIMIT\_T\_FLT.

### Source File

```
#include <GFLIB_Limit.h>
```

### Macro Definition

```
#define GFLIB_LIMIT_DEFAULT_FLT {FLOAT_MIN, FLOAT_MAX}
```

## 6.101 GFLIB\_LOG10\_DEFAULT\_FLT

Default approximation coefficients for GFLIB\_Log10\_FLT function.

### Source File

```
#include <GFLIB_Log10.h>
```

### Macro Definition

```
#define GFLIB_LOG10_DEFAULT_FLT &fltgflibLog10Coef
```

## 6.102 GFLIB\_LOWERLIMIT\_T

Definition of GFLIB\_LOWERLIMIT\_T as alias for GFLIB\_LOWERLIMIT\_T\_F16 datatype in case the 16-bit fractional implementation is selected.

**Source File**

```
#include <GFLIB_LowerLimit.h>
```

**Macro Definition**

```
#define GFLIB_LOWERLIMIT_T GFLIB_LOWERLIMIT_T_F16
```

### 6.103 GFLIB\_LOWERLIMIT\_DEFAULT

Definition of GFLIB\_LOWERLIMIT\_DEFAULT as alias for GFLIB\_LOWERLIMIT\_DEFAULT\_F16 default value in case the 16-bit fractional implementation is selected.

**Source File**

```
#include <GFLIB_LowerLimit.h>
```

**Macro Definition**

```
#define GFLIB_LOWERLIMIT_DEFAULT GFLIB_LOWERLIMIT_DEFAULT_F16
```

### 6.104 GFLIB\_LOWERLIMIT\_DEFAULT\_F32

Default value for GFLIB\_LOWERLIMIT\_T\_F32.

**Source File**

```
#include <GFLIB_LowerLimit.h>
```

**Macro Definition**

```
#define GFLIB_LOWERLIMIT_DEFAULT_F32 {INT32_MIN}
```

### 6.105 GFLIB\_LOWERLIMIT\_DEFAULT\_F16

Default value for GFLIB\_LOWERLIMIT\_T\_F16.

**Source File**

```
#include <GFLIB_LowerLimit.h>
```

**Macro Definition**

```
#define GFLIB_LOWERLIMIT_DEFAULT_F16 {INT16_MIN}
```

### 6.106 GFLIB\_LOWERLIMIT\_DEFAULT\_FLT

Default value for GFLIB\_LOWERLIMIT\_T\_FLT.

**Source File**

```
#include <GFLIB_LowerLimit.h>
```

**Macro Definition**

```
#define GFLIB_LOWERLIMIT_DEFAULTFLT {FLOAT_MIN}
```

**6.107 GFLIB\_LUT1D\_T**

Definition of GFLIB\_LUT1D\_T as alias for GFLIB\_LUT1D\_T\_F16 datatype in case the 16-bit fractional implementation is selected.

**Source File**

```
#include <GFLIB_Lut1D.h>
```

**Macro Definition**

```
#define GFLIB_LUT1D_T GFLIB_LUT1D_T_F16
```

**6.108 GFLIB\_LUT1D\_DEFAULT**

Definition of GFLIB\_LUT1D\_DEFAULT as alias for GFLIB\_LUT1D\_DEFAULT\_F16 default value in case the 16-bit fractional implementation is selected.

**Source File**

```
#include <GFLIB_Lut1D.h>
```

**Macro Definition**

```
#define GFLIB_LUT1D_DEFAULT GFLIB_LUT1D_DEFAULT_F16
```

**6.109 GFLIB\_LUT1D\_DEFAULT\_F32**

Default value for GFLIB\_LUT1D\_T\_F32.

**Source File**

```
#include <GFLIB_Lut1D.h>
```

**Macro Definition**

```
#define GFLIB_LUT1D_DEFAULT_F32 { (tU32) 0, (tFrac32*) 0 }
```

**6.110 GFLIB\_LUT1D\_DEFAULT\_F16**

Default value for GFLIB\_LUT1D\_T\_F16.

**Source File**

```
#include <GFLIB_Lut1D.h>
```

**Macro Definition**

```
#define GFLIB_LUT1D_DEFAULT_F16 { (tU16) 0, (tFrac16*) 0 }
```

### 6.111 **GFLIB\_LUT1D\_DEFAULT\_FLT**

Default value for GFLIB\_LUT1D\_T\_FLT.

#### Source File

```
#include <GFLIB_Lut1D.h>
```

#### Macro Definition

```
#define GFLIB_LUT1D_DEFAULT_FLT { (tU32) 0, (tFloat*) 0 }
```

### 6.112 **GFLIB\_LUT2D\_T**

Definition of GFLIB\_LUT2D\_T as alias for GFLIB\_LUT2D\_T\_F16 datatype in case the 16-bit fractional implementation is selected.

#### Source File

```
#include <GFLIB_Lut2D.h>
```

#### Macro Definition

```
#define GFLIB_LUT2D_T GFLIB_LUT2D_T_F16
```

### 6.113 **GFLIB\_LUT2D\_DEFAULT**

Definition of GFLIB\_LUT2D\_DEFAULT as alias for GFLIB\_LUT2D\_DEFAULT\_F16 default value in case the 16-bit fractional implementation is selected.

#### Source File

```
#include <GFLIB_Lut2D.h>
```

#### Macro Definition

```
#define GFLIB_LUT2D_DEFAULT GFLIB_LUT2D_DEFAULT_F16
```

### 6.114 **GFLIB\_LUT2D\_DEFAULT\_F32**

Default value for GFLIB\_LUT2D\_T\_F32.

#### Source File

```
#include <GFLIB_Lut2D.h>
```

#### Macro Definition

```
#define GFLIB_LUT2D_DEFAULT_F32 { (tU32) 0, (tU32) 0, (tFrac32*) 0 }
```

### 6.115 **GFLIB\_LUT2D\_DEFAULT\_F16**

Default value for GFLIB\_LUT2D\_T\_F16.

**Source File**

```
#include <GFLIB_Lut2D.h>
```

**Macro Definition**

```
#define GFLIB_LUT2D_DEFAULT_F16 { (tU16) 0, (tU16) 0, (tFrac16*) 0 }
```

## 6.116 GFLIB\_LUT2D\_DEFAULT\_FLT

Default value for GFLIB\_LUT2D\_T\_FLT.

**Source File**

```
#include <GFLIB_Lut2D.h>
```

**Macro Definition**

```
#define GFLIB_LUT2D_DEFAULT_FLT { (tU32) 0, (tU32) 0, (tFloat*) 0 }
```

## 6.117 GFLIB\_RAMP\_T

Definition of GFLIB\_RAMP\_T as alias for GFLIB\_RAMP\_T\_F16 datatype in case the 16-bit fractional implementation is selected.

**Source File**

```
#include <GFLIB_Ramp.h>
```

**Macro Definition**

```
#define GFLIB_RAMP_T GFLIB_RAMP_T_F16
```

## 6.118 GFLIB\_RAMP\_DEFAULT

Definition of GFLIB\_RAMP\_DEFAULT as alias for GFLIB\_RAMP\_DEFAULT\_F16 default value in case the 16-bit fractional implementation is selected.

**Source File**

```
#include <GFLIB_Ramp.h>
```

**Macro Definition**

```
#define GFLIB_RAMP_DEFAULT GFLIB_RAMP_DEFAULT_F16
```

## 6.119 GFLIB\_RAMP\_DEFAULT\_F32

Default value for GFLIB\_RAMP\_T\_F32.

**Source File**

```
#include <GFLIB_Ramp.h>
```

**Macro Definition**

```
#define GFLIB_RAMP_DEFAULT_F32 { (tFrac32) 0, (tFrac32) 0, (tFrac32) 0 }
```

**6.120 GFLIB\_RAMP\_DEFAULT\_F16**

Default value for GFLIB\_RAMP\_T\_F16.

**Source File**

```
#include <GFLIB_Ramp.h>
```

**Macro Definition**

```
#define GFLIB_RAMP_DEFAULT_F16 { (tFrac16) 0, (tFrac16) 0, (tFrac16) 0 }
```

**6.121 GFLIB\_RAMP\_DEFAULT\_FLT**

Default value for GFLIB\_RAMP\_T\_FLT.

**Source File**

```
#include <GFLIB_Ramp.h>
```

**Macro Definition**

```
#define GFLIB_RAMP_DEFAULT_FLT { (tFloat) 0, (tFloat) 0, (tFloat) 0 }
```

**6.122 GFLIB\_SIN\_FLT\_MIN**

Floating-point min. input value for computation.

**Source File**

```
#include <GFLIB_Sin.c>
```

**Macro Definition**

```
#define GFLIB_SIN_FLT_MIN ((tFloat) 1.22070312500000e-04)
```

**6.123 GFLIB\_SIN\_T**

Definition of GFLIB\_SIN\_T as alias for GFLIB\_SIN\_T\_F16 datatype in case the 16-bit fractional implementation is selected.

**Source File**

```
#include <GFLIB_Sin.h>
```

**Macro Definition**

```
#define GFLIB_SIN_T GFLIB_SIN_T_F16
```

## 6.124 **GFLIB\_SIN\_DEFAULT**

Definition of GFLIB\_SIN\_DEFAULT as alias for GFLIB\_SIN\_DEFAULT\_F16 default value in case the 16-bit fractional implementation is selected.

### Source File

```
#include <GFLIB_Sin.h>
```

### Macro Definition

```
#define GFLIB_SIN_DEFAULT GFLIB_SIN_DEFAULT_F16
```

## 6.125 **GFLIB\_SIN\_DEFAULT\_F32**

Default approximation coefficients for GFLIB\_Sin\_F32 function.

### Source File

```
#include <GFLIB_Sin.h>
```

### Macro Definition

```
#define GFLIB_SIN_DEFAULT_F32 &f32gflibSinCoef
```

## 6.126 **GFLIB\_SIN\_DEFAULT\_F16**

Default approximation coefficients for GFLIB\_Sin\_F16 function.

### Source File

```
#include <GFLIB_Sin.h>
```

### Macro Definition

```
#define GFLIB_SIN_DEFAULT_F16 &f16gflibSinCoef
```

## 6.127 **GFLIB\_SIN\_DEFAULT\_FLT**

Default approximation coefficients for GFLIB\_Sin\_FLT function.

### Source File

```
#include <GFLIB_Sin.h>
```

### Macro Definition

```
#define GFLIB_SIN_DEFAULT_FLT &fltgflibSinCoef
```

## 6.128 **GFLIB\_SINCOS\_FLT\_MIN**

Floating-point min. input value for computation.

**Source File**

```
#include <GFLIB_SinCos.c>
```

**Macro Definition**

```
#define GFLIB_SINCOS_FLT_MIN ((tFloat)1.22070312500000e-04)
```

**6.129 GFLIB\_SINCOS\_T**

Definition of GFLIB\_SINCOS\_T as alias for GFLIB\_SINCOS\_T\_F16 datatype in case the 16-bit fractional implementation is selected.

**Source File**

```
#include <GFLIB_SinCos.h>
```

**Macro Definition**

```
#define GFLIB_SINCOS_T GFLIB_SINCOS_T_F16
```

**6.130 GFLIB\_SINCOS\_DEFAULT**

Definition of GFLIB\_SINCOS\_DEFAULT as alias for GFLIB\_SINCOS\_DEFAULT\_F16 default value in case the 16-bit fractional implementation is selected.

**Source File**

```
#include <GFLIB_SinCos.h>
```

**Macro Definition**

```
#define GFLIB_SINCOS_DEFAULT GFLIB_SINCOS_DEFAULT_F16
```

**6.131 GFLIB\_SINCOS\_DEFAULT\_F32**

Default approximation coefficients for GFLIB\_SinCos\_F32 function.

**Source File**

```
#include <GFLIB_SinCos.h>
```

**Macro Definition**

```
#define GFLIB_SINCOS_DEFAULT_F32 &f32gflibSinCosCoef
```

**6.132 GFLIB\_SINCOS\_DEFAULT\_F16**

Default approximation coefficients for GFLIB\_SinCos\_F16 function.

**Source File**

```
#include <GFLIB_SinCos.h>
```

**Macro Definition**

```
#define GFLIB_SINCOS_DEFAULT_F16 &f16gflibSinCosCoef
```

**6.133 GFLIB\_SINCOS\_DEFAULT\_FLT**

Default approximation coefficients for GFLIB\_SinCos\_FLT function.

**Source File**

```
#include <GFLIB_SinCos.h>
```

**Macro Definition**

```
#define GFLIB_SINCOS_DEFAULT_FLT &fltgflibSinCosCoef
```

**6.134 GFLIB\_TAN\_FLT\_MIN**

Floating-point min. input value for computation.

**Source File**

```
#include <GFLIB_Tan.c>
```

**Macro Definition**

```
#define GFLIB_TAN_FLT_MIN ((tFloat)2.44140625000000e-04)
```

**6.135 GFLIB\_TAN\_FLT\_3P4**

Floating-point constant 3\*PI/4.

**Source File**

```
#include <GFLIB_Tan.c>
```

**Macro Definition**

```
#define GFLIB_TAN_FLT_3P4 ((tFloat)2.356194490192345e+00)
```

**6.136 GFLIB\_TAN\_FLT\_PI**

Floating-point constant PI.

**Source File**

```
#include <GFLIB_Tan.c>
```

**Macro Definition**

```
#define GFLIB_TAN_FLT_PI ((tFloat)3.141592653589793e+00)
```

### 6.137 GFLIB\_TAN\_FLT\_CORR1

Floating-point correction constant PI\_double - PI\_single.

#### Source File

```
#include <GFLIB_Tan.c>
```

#### Macro Definition

```
#define GFLIB_TAN_FLT_CORR1 ((tFloat)-8.742278012618954e-08)
```

### 6.138 GFLIB\_TAN\_FLT\_PI4

Floating-point constant PI/4.

#### Source File

```
#include <GFLIB_Tan.c>
```

#### Macro Definition

```
#define GFLIB_TAN_FLT_PI4 ((tFloat)7.853981633974483e-01)
```

### 6.139 GFLIB\_TAN\_FLT\_PI2

Floating-point constant PI/2.

#### Source File

```
#include <GFLIB_Tan.c>
```

#### Macro Definition

```
#define GFLIB_TAN_FLT_PI2 ((tFloat)1.570796326794897e+00)
```

### 6.140 GFLIB\_TAN\_FLT\_CORR2

Floating-point correction constant PI\_double/2 - PI\_single/2.

#### Source File

```
#include <GFLIB_Tan.c>
```

#### Macro Definition

```
#define GFLIB_TAN_FLT_CORR2 ((tFloat)-4.371139006309477e-08)
```

### 6.141 GFLIB\_TAN\_T

Definition of GFLIB\_TAN\_T as alias for GFLIB\_TAN\_T\_F16 datatype in case the 16-bit fractional implementation is selected.

**Source File**

```
#include <GFLIB_Tan.h>
```

**Macro Definition**

```
#define GFLIB_TAN_T GFLIB_TAN_T_F16
```

## 6.142 GFLIB\_TAN\_DEFAULT

Definition of GFLIB\_TAN\_DEFAULT as alias for GFLIB\_TAN\_DEFAULT\_F16 default value in case the 16-bit fractional implementation is selected.

**Source File**

```
#include <GFLIB_Tan.h>
```

**Macro Definition**

```
#define GFLIB_TAN_DEFAULT GFLIB\_TAN\_DEFAULT\_F16
```

## 6.143 GFLIB\_TAN\_DEFAULT\_F32

Default approximation coefficients for GFLIB\_Tan\_F32 function.

**Source File**

```
#include <GFLIB_Tan.h>
```

**Macro Definition**

```
#define GFLIB_TAN_DEFAULT_F32 &f32gflibTanCoef
```

## 6.144 GFLIB\_TAN\_DEFAULT\_F16

Default approximation coefficients for GFLIB\_Tan\_F16 function.

**Source File**

```
#include <GFLIB_Tan.h>
```

**Macro Definition**

```
#define GFLIB_TAN_DEFAULT_F16 &f16gflibTanCoef
```

## 6.145 GFLIB\_TAN\_DEFAULT\_FLT

Default approximation coefficients for GFLIB\_Tan\_FLT function.

**Source File**

```
#include <GFLIB_Tan.h>
```

**Macro Definition**

```
#define GFLIB_TAN_DEFAULTFLT &fltgfllibTanCoef
```

**6.146 GFLIB\_UPPERLIMIT\_T**

Definition of GFLIB\_UPPERLIMIT\_T as alias for GFLIB\_UPPERLIMIT\_T\_F16 datatype in case the 16-bit fractional implementation is selected.

**Source File**

```
#include <GFLIB_UpperLimit.h>
```

**Macro Definition**

```
#define GFLIB_UPPERLIMIT_T GFLIB_UPPERLIMIT_T_F16
```

**6.147 GFLIB\_UPPERLIMIT\_DEFAULT**

Definition of GFLIB\_UPPERLIMIT\_DEFAULT as alias for GFLIB\_UPPERLIMIT\_DEFAULT\_F16 default value in case the 16-bit fractional implementation is selected.

**Source File**

```
#include <GFLIB_UpperLimit.h>
```

**Macro Definition**

```
#define GFLIB_UPPERLIMIT_DEFAULT GFLIB_UPPERLIMIT_DEFAULT_F16
```

**6.148 GFLIB\_UPPERLIMIT\_DEFAULT\_F32**

Default value for GFLIB\_UPPERLIMIT\_T\_F32.

**Source File**

```
#include <GFLIB_UpperLimit.h>
```

**Macro Definition**

```
#define GFLIB_UPPERLIMIT_DEFAULT_F32 {INT32_MAX}
```

**6.149 GFLIB\_UPPERLIMIT\_DEFAULT\_F16**

Default value for GFLIB\_UPPERLIMIT\_T\_F16.

**Source File**

```
#include <GFLIB_UpperLimit.h>
```

**Macro Definition**

```
#define GFLIB_UPPERLIMIT_DEFAULT_F16 {INT16_MAX}
```

**6.150 GFLIB\_UPPERLIMIT\_DEFAULT\_FLT**

Default value for GFLIB\_UPPERLIMIT\_T\_FLT.

**Source File**

```
#include <GFLIB_UpperLimit.h>
```

**Macro Definition**

```
#define GFLIB_UPPERLIMIT_DEFAULT_FLT {FLOAT_MAX}
```

**6.151 GFLIB\_VECTORLIMIT\_T**

Definition of GFLIB\_VECTORLIMIT\_T as alias for GFLIB\_VECTORLIMIT\_T\_F16 datatype in case the 16-bit fractional implementation is selected.

**Source File**

```
#include <GFLIB_VectorLimit.h>
```

**Macro Definition**

```
#define GFLIB_VECTORLIMIT_T GFLIB_VECTORLIMIT_T_F16
```

**6.152 GFLIB\_VECTORLIMIT\_DEFAULT**

Definition of GFLIB\_VECTORLIMIT\_DEFAULT as alias for GFLIB\_VECTORLIMIT\_DEFAULT\_F16 default value in case the 16-bit fractional implementation is selected.

**Source File**

```
#include <GFLIB_VectorLimit.h>
```

**Macro Definition**

```
#define GFLIB_VECTORLIMIT_DEFAULT GFLIB_VECTORLIMIT_DEFAULT_F16
```

**6.153 GFLIB\_VECTORLIMIT\_DEFAULT\_F32**

Default value for GFLIB\_VECTORLIMIT\_T\_F32.

**Source File**

```
#include <GFLIB_VectorLimit.h>
```

**Macro Definition**

```
#define GFLIB_VECTORLIMIT_DEFAULT_F32 { (tFrac32) 0 }
```

**6.154 GFLIB\_VECTORLIMIT\_DEFAULT\_F16**

Default value for GFLIBVECTORLIMIT\_T\_F16.

**Source File**

```
#include <GFLIB_VectorLimit.h>
```

**Macro Definition**

```
#define GFLIB_VECTORLIMIT_DEFAULT_F16 { (tFrac16) 0 }
```

**6.155 GFLIB\_VECTORLIMIT\_DEFAULT\_FLT**

Default value for GFLIB\_VECTORLIMIT\_T\_FLT.

**Source File**

```
#include <GFLIB_VectorLimit.h>
```

**Macro Definition**

```
#define GFLIB_VECTORLIMIT_DEFAULT_FLT { (tFloat) 0 }
```

**6.156 GFLIB\_VLOG10\_DEFAULT\_FLT**

Default approximation coefficients for GFLIB\_VLog10\_FLT function.

**Source File**

```
#include <GFLIB_VLog10.h>
```

**Macro Definition**

```
#define GFLIB_VLOG10_DEFAULT_FLT &fltgfllibVLog10Coef
```

**6.157 GMCLIB\_DECOUPLINGPMSM\_T**

Definition of GMCLIB\_DECOUPLINGPMSM\_T as alias for GMCLIB\_DECOUPLINGPMSM\_T\_F16 datatype in case the 16-bit fractional implementation is selected.

**Source File**

```
#include <GMCLIB_DecouplingPMSM.h>
```

**Macro Definition**

```
#define GMCLIB_DECOUPLINGPMSM_T GMCLIB_DECOUPLINGPMSM_T_F16
```

## 6.158 GMCLIB\_DECOUPLINGPMSM\_DEFAULT

Definition of GMCLIB\_DECOUPLINGPMSM\_DEFAULT as alias for GMCLIB\_DECOUPLINGPMSM\_DEFAULT\_F16 default value in case the 16-bit fractional implementation is selected.

### Source File

```
#include <GMCLIB_DcouplingPMSM.h>
```

### Macro Definition

```
#define GMCLIB_DECOUPLINGPMSM_DEFAULT  
GMCLIB_DECOUPLINGPMSM_DEFAULT_F16
```

## 6.159 GMCLIB\_DECOUPLINGPMSM\_DEFAULT\_F32

Default value for GMCLIB\_DECOUPLINGPMSM\_T\_F32.

### Source File

```
#include <GMCLIB_DcouplingPMSM.h>
```

### Macro Definition

```
#define GMCLIB_DECOUPLINGPMSM_DEFAULT_F32 { (tFrac32) 0, (ts16) 0,  
(tFrac32) 0, (ts16) 0 }
```

## 6.160 GMCLIB\_DECOUPLINGPMSM\_DEFAULT\_F16

Default value for GMCLIB\_DECOUPLINGPMSM\_T\_F16.

### Source File

```
#include <GMCLIB_DcouplingPMSM.h>
```

### Macro Definition

```
#define GMCLIB_DECOUPLINGPMSM_DEFAULT_F16 { (tFrac16) 0, (ts16) 0,  
(tFrac16) 0, (ts16) 0 }
```

## 6.161 GMCLIB\_DECOUPLINGPMSM\_DEFAULT\_FLT

Default value for GMCLIB\_DECOUPLINGPMSM\_T\_FLT.

### Source File

```
#include <GMCLIB_DcouplingPMSM.h>
```

### Macro Definition

```
#define GMCLIB_DECOUPLINGPMSM_DEFAULT_FLT { (tFloat) 0, (tFloat) 0 }
```

## 6.162 GMCLIB\_ELIMDCBUSRIP\_T

Definition of GMCLIB\_ELIMDCBUSRIP\_T as alias for GMCLIB\_ELIMDCBUSRIP\_T\_F16 datatype in case the 16-bit fractional implementation is selected.

### Source File

```
#include <GMCLIB_ElimDcBusRip.h>
```

### Macro Definition

```
#define GMCLIB_ELIMDCBUSRIP_T GMCLIB_ELIMDCBUSRIP_T_F16
```

## 6.163 GMCLIB\_ELIMDCBUSRIP\_DEFAULT

Definition of GMCLIB\_ELIMDCBUSRIP\_DEFAULT as alias for GMCLIB\_ELIMDCBUSRIP\_DEFAULT\_F16 default value in case the 16-bit fractional implementation is selected.

### Source File

```
#include <GMCLIB_ElimDcBusRip.h>
```

### Macro Definition

```
#define GMCLIB_ELIMDCBUSRIP_DEFAULT  
GMCLIB_ELIMDCBUSRIP_DEFAULT_F16
```

## 6.164 GMCLIB\_ELIMDCBUSRIP\_DEFAULT\_F32

Default value for GMCLIB\_ELIMDCBUSRIP\_T\_F32.

### Source File

```
#include <GMCLIB_ElimDcBusRip.h>
```

### Macro Definition

```
#define GMCLIB_ELIMDCBUSRIP_DEFAULT_F32 { (tFrac32) 0, (tFrac32) 0 }
```

## 6.165 GMCLIB\_ELIMDCBUSRIP\_DEFAULT\_F16

Default value for GMCLIB\_ELIMDCBUSRIP\_T\_F16.

### Source File

```
#include <GMCLIB_ElimDcBusRip.h>
```

### Macro Definition

```
#define GMCLIB_ELIMDCBUSRIP_DEFAULT_F16 { (tFrac16) 0, (tFrac16) 0 }
```

## 6.166 GMCLIB\_ELIMDCBUSRIP\_DEFAULT\_FLT

Default value for GMCLIB\_ELIMDCBUSRIP\_T\_FLT.

### Source File

```
#include <GMCLIB_ElimDcBusRip.h>
```

### Macro Definition

```
#define GMCLIB_ELIMDCBUSRIP_DEFAULT_FLT { (tFloat) 0, (tFloat) 0 }
```

## 6.167 SWLIBS\_SUPPORT\_F32

Enables/disables support of 32-bit fractional implementation.

### Source File

```
#include <SWLIBS_Config.h>
```

### Macro Definition

```
#define SWLIBS_SUPPORT_F32 SWLIBS_STD_ON
```

## 6.168 SWLIBS\_SUPPORT\_F16

Enables/disables support of 16-bit fractional implementation.

### Source File

```
#include <SWLIBS_Config.h>
```

### Macro Definition

```
#define SWLIBS_SUPPORT_F16 SWLIBS_STD_ON
```

## 6.169 SWLIBS\_SUPPORT\_FLT

Enables/disables support of single precision floating point implementation.

### Source File

```
#include <SWLIBS_Config.h>
```

### Macro Definition

```
#define SWLIBS_SUPPORT_FLT SWLIBS_STD_ON
```

## 6.170 SWLIBS\_SUPPORTED\_IMPLEMENTATION

Array of supported implementations.

**Source File**

```
#include <SWLIBS_Config.h>
```

**Macro Definition**

```
#define SWLIBS_SUPPORTED_IMPLEMENTATION {SWLIBS\_SUPPORT\_F32, \
SWLIBS\_SUPPORT\_F16, \ SWLIBS\_SUPPORT\_FLT, \ 0,0,0,0,0,0}
```

## 6.171 SFRACT\_MIN

Constant representing the maximal negative value of a signed 16-bit fixed point fractional number, floating point representation.

**Source File**

```
#include <SWLIBS_Defines.h>
```

**Macro Definition**

```
#define SFRACT_MIN (-1.0)
```

## 6.172 SFRACT\_MAX

Constant representing the maximal positive value of a signed 16-bit fixed point fractional number, floating point representation.

**Source File**

```
#include <SWLIBS_Defines.h>
```

**Macro Definition**

```
#define SFRACT_MAX (0.999969482421875)
```

## 6.173 FRACT\_MIN

Constant representing the maximal negative value of signed 32-bit fixed point fractional number, floating point representation.

**Source File**

```
#include <SWLIBS_Defines.h>
```

**Macro Definition**

```
#define FRACT_MIN (-1.0)
```

## 6.174 FRACT\_MAX

Constant representing the maximal positive value of a signed 32-bit fixed point fractional number, floating point representation.

**Source File**

```
#include <SWLIBS_Defines.h>
```

**Macro Definition**

```
#define FRACT_MAX (0.999999995343387126922607421875)
```

**6.175 FRAC32\_0\_5**

Value 0.5 in 32-bit fixed point fractional format.

**Source File**

```
#include <SWLIBS_Defines.h>
```

**Macro Definition**

```
#define FRAC32_0_5 ((tFrac32) 0x40000000)
```

**6.176 FRAC16\_0\_5**

Value 0.5 in 16-bit fixed point fractional format.

**Source File**

```
#include <SWLIBS_Defines.h>
```

**Macro Definition**

```
#define FRAC16_0_5 ((tFrac16) 0x4000)
```

**6.177 FRAC32\_0\_25**

Value 0.25 in 32-bit fixed point fractional format.

**Source File**

```
#include <SWLIBS_Defines.h>
```

**Macro Definition**

```
#define FRAC32_0_25 ((tFrac32) 0x20000000)
```

**6.178 FRAC16\_0\_25**

Value 0.25 in 16-bit fixed point fractional format.

**Source File**

```
#include <SWLIBS_Defines.h>
```

### Macro Definition

```
#define FRAC16_0_25 ((tFrac16) 0x2000)
```

## 6.179 UINT16\_MAX

Constant representing the maximal positive value of a unsigned 16-bit fixed point integer number, equal to  $2^{15} = 0x8000$ .

### Source File

```
#include <SWLIBS_Defines.h>
```

### Macro Definition

```
#define UINT16_MAX ((tU16) 0x8000)
```

## 6.180 INT16\_MAX

Constant representing the maximal positive value of a signed 16-bit fixed point integer number, equal to  $2^{15}-1 = 0x7fff$ .

### Source File

```
#include <SWLIBS_Defines.h>
```

### Macro Definition

```
#define INT16_MAX ((tS16) 0x7fff)
```

## 6.181 INT16\_MIN

Constant representing the maximal negative value of a signed 16-bit fixed point integer number, equal to  $-2^{15} = 0x8000$ .

### Source File

```
#include <SWLIBS_Defines.h>
```

### Macro Definition

```
#define INT16_MIN ((tS16) 0x8000)
```

## 6.182 UINT32\_MAX

Constant representing the maximal positive value of a unsigned 32-bit fixed point integer number, equal to  $2^{31} = 0x80000000$ .

### Source File

```
#include <SWLIBS_Defines.h>
```

**Macro Definition**

```
#define UINT32_MAX ((tu32) 0x80000000U)
```

**6.183 INT32\_MAX**

Constant representing the maximal positive value of a signed 32-bit fixed point integer number, equal to  $2^{31}-1 = 0x7fff\ ffff$ .

**Source File**

```
#include <SWLIBS_Defines.h>
```

**Macro Definition**

```
#define INT32_MAX ((ts32) 0x7fffffff)
```

**6.184 INT32\_MIN**

Constant representing the maximal negative value of a signed 32-bit fixed point integer number, equal to  $-2^{31} = 0x8000\ 0000$ .

**Source File**

```
#include <SWLIBS_Defines.h>
```

**Macro Definition**

```
#define INT32_MIN ((ts32) 0x80000000U)
```

**6.185 FLOAT\_MIN**

Constant representing the maximal negative value of the 32-bit float type.

**Source File**

```
#include <SWLIBS_Defines.h>
```

**Macro Definition**

```
#define FLOAT_MIN ((tFloat) (-3.4028234e+38F))
```

**6.186 FLOAT\_MAX**

Constant representing the maximal positive value of the 32-bit float type.

**Source File**

```
#include <SWLIBS_Defines.h>
```

**Macro Definition**

```
#define FLOAT_MAX ((tFloat) (3.4028234e+38F))
```

## 6.187 INT16TOINT32

Type casting - signed 16-bit integer value cast to a signed 32-bit integer.

### Source File

```
#include <SWLIBS_Defines.h>
```

### Macro Definition

```
#define INT16TOINT32 ((tS32) (x))
```

## 6.188 INT32TOINT16

Type casting - signed 32-bit integer value cast to a signed 16-bit integer.

### Source File

```
#include <SWLIBS_Defines.h>
```

### Macro Definition

```
#define INT32TOINT16 ((tS16) (x))
```

## 6.189 INT32TOINT64

Type casting - signed 32-bit integer value cast to a signed 64-bit integer.

### Source File

```
#include <SWLIBS_Defines.h>
```

### Macro Definition

```
#define INT32TOINT64 ((tS64) (x))
```

## 6.190 INT64TOINT32

Type casting - signed 64-bit integer value cast to a signed 32-bit integer.

### Source File

```
#include <SWLIBS_Defines.h>
```

### Macro Definition

```
#define INT64TOINT32 ((tS32) (x))
```

## 6.191 F16TOINT16

Type casting - signed 16-bit fractional value cast to a signed 16-bit integer.

**Source File**

```
#include <SWLIBS_Defines.h>
```

**Macro Definition**

```
#define F16TOINT16 ((ts16) (x))
```

**6.192 F32TOINT16**

Type casting - lower 16 bits of a signed 32-bit fractional value cast to a signed 16-bit integer.

**Source File**

```
#include <SWLIBS_Defines.h>
```

**Macro Definition**

```
#define F32TOINT16 ((ts16) (x))
```

**6.193 F64TOINT16**

Type casting - lower 16 bits of a signed 64-bit fractional value cast to a signed 16-bit integer.

**Source File**

```
#include <SWLIBS_Defines.h>
```

**Macro Definition**

```
#define F64TOINT16 ((ts16) (x))
```

**6.194 F16TOINT32**

Type casting - a signed 16-bit fractional value cast to a signed 32-bit integer, the value placed at the lower 16-bits of the 32-bit result.

**Source File**

```
#include <SWLIBS_Defines.h>
```

**Macro Definition**

```
#define F16TOINT32 ((ts32) (x))
```

**6.195 F32TOINT32**

Type casting - signed 32-bit fractional value cast to a signed 32-bit integer.

**Source File**

```
#include <SWLIBS_Defines.h>
```

**Macro Definition**

```
#define F32TOINT32 ((ts32) (x))
```

**6.196 F64TOINT32**

Type casting - lower 32 bits of a signed 64-bit fractional value cast to a signed 32-bit integer.

**Source File**

```
#include <SWLIBS_Defines.h>
```

**Macro Definition**

```
#define F64TOINT32 ((ts32) (x))
```

**6.197 F16TOINT64**

Type casting - signed 16-bit fractional value cast to a signed 64-bit integer, the value placed at the lower 16-bits of the 64-bit result.

**Source File**

```
#include <SWLIBS_Defines.h>
```

**Macro Definition**

```
#define F16TOINT64 ((ts64) (x))
```

**6.198 F32TOINT64**

Type casting - signed 32-bit fractional value cast to a signed 64-bit integer, the value placed at the lower 32-bits of the 64-bit result.

**Source File**

```
#include <SWLIBS_Defines.h>
```

**Macro Definition**

```
#define F32TOINT64 ((ts64) (x))
```

**6.199 F64TOINT64**

Type casting - signed 64-bit fractional value cast to a signed 64-bit integer.

**Source File**

```
#include <SWLIBS_Defines.h>
```

**Macro Definition**

```
#define F64TOINT64 ((tS64) (x))
```

**6.200 INT16TOF16**

Type casting - signed 16-bit integer value cast to a signed 16-bit fractional.

**Source File**

```
#include <SWLIBS_Defines.h>
```

**Macro Definition**

```
#define INT16TOF16 ((tFrac16) (x))
```

**6.201 INT16TOF32**

Type casting - signed 16-bit integer value cast to a signed 32-bit fractional, the value placed at the lower 16 bits of the 32-bit result.

**Source File**

```
#include <SWLIBS_Defines.h>
```

**Macro Definition**

```
#define INT16TOF32 ((tFrac32) (x))
```

**6.202 INT32TOF16**

Type casting - lower 16-bits of a signed 32-bit integer value cast to a signed 16-bit fractional.

**Source File**

```
#include <SWLIBS_Defines.h>
```

**Macro Definition**

```
#define INT32TOF16 ((tFrac16) (x))
```

**6.203 INT32TOF32**

Type casting - signed 32-bit integer value cast to a signed 32-bit fractional.

**Source File**

```
#include <SWLIBS_Defines.h>
```

**Macro Definition**

```
#define INT32TOF32 ((tFrac32) (x))
```

**6.204 INT64TOF16**

Type casting - lower 16-bits of a signed 64-bit integer value cast to a signed 16-bit fractional.

**Source File**

```
#include <SWLIBS_Defines.h>
```

**Macro Definition**

```
#define INT64TOF16 ((tFrac16) (x))
```

**6.205 INT64TOF32**

Type casting - lower 32-bits of a signed 64-bit integer value cast to a signed 32-bit fractional.

**Source File**

```
#include <SWLIBS_Defines.h>
```

**Macro Definition**

```
#define INT64TOF32 ((tFrac32) (x))
```

**6.206 F16\_1\_DIVBY\_SQRT3**

One over sqrt(3) with a 16-bit result, the result is rounded for a better accuracy.

**Source File**

```
#include <SWLIBS_Defines.h>
```

**Macro Definition**

```
#define F16_1_DIVBY_SQRT3 ((tFrac16) 0x49E7)
```

**6.207 F32\_1\_DIVBY\_SQRT3**

One over sqrt(3) with a 32-bit result, the result is rounded for a better accuracy.

**Source File**

```
#include <SWLIBS_Defines.h>
```

**Macro Definition**

```
#define F32_1_DIVBY_SQRT3 ((tFrac32) 0x49E69D16)
```

## 6.208 F16\_SQRT3\_DIVBY\_2

Sqrt(3) divided by two with a 16-bit result, the result is rounded for a better accuracy.

### Source File

```
#include <SWLIBS_Defines.h>
```

### Macro Definition

```
#define F16_SQRT3_DIVBY_2 ((tFrac16) 0x6EDA)
```

## 6.209 F16\_SQRT3\_DIVBY\_4

Sqrt(3) divided by four with a 16-bit result.

### Source File

```
#include <SWLIBS_Defines.h>
```

### Macro Definition

```
#define F16_SQRT3_DIVBY_4 ((tFrac16) 0x376D)
```

## 6.210 F32\_SQRT3\_DIVBY\_2

Sqrt(3) divided by two with a 32-bit result, the result is rounded for a better accuracy.

### Source File

```
#include <SWLIBS_Defines.h>
```

### Macro Definition

```
#define F32_SQRT3_DIVBY_2 ((tFrac32) 0x6ED9EBA1)
```

## 6.211 F32\_SQRT3\_DIVBY\_4

Sqrt(3) divided by four with a 32-bit result.

### Source File

```
#include <SWLIBS_Defines.h>
```

### Macro Definition

```
#define F32_SQRT3_DIVBY_4 ((tFrac32) 0x376CF5D1)
```

## 6.212 F16\_SQRT2\_DIVBY\_2

Sqrt(2) divided by two with a 16-bit result, the result is rounded for a better accuracy.

**Source File**

```
#include <SWLIBS_Defines.h>
```

**Macro Definition**

```
#define F16_SQRT2_DIVBY_2 ((tFrac16) 0x5A82)
```

**6.213 F32\_SQRT2\_DIVBY\_2**

Sqrt(2) divided by two with a 32-bit result, the result is rounded for a better accuracy.

**Source File**

```
#include <SWLIBS_Defines.h>
```

**Macro Definition**

```
#define F32_SQRT2_DIVBY_2 ((tFrac32) 0x5A82799A)
```

**6.214 F16\_1\_DIVBY\_3**

One third in 16-bit resolution, the result is rounded for a better accuracy.

**Source File**

```
#include <SWLIBS_Defines.h>
```

**Macro Definition**

```
#define F16_1_DIVBY_3 ((tFrac16) 0x2AAB)
```

**6.215 F32\_1\_DIVBY\_3**

One third in 32-bit resolution, the result is rounded for a better accuracy.

**Source File**

```
#include <SWLIBS_Defines.h>
```

**Macro Definition**

```
#define F32_1_DIVBY_3 ((tFrac32) 0x2AAAAAAAB)
```

**6.216 F16\_2\_DIVBY\_3**

Two thirds in 16-bit resolution, the result is rounded for a better accuracy.

**Source File**

```
#include <SWLIBS_Defines.h>
```

### Macro Definition

```
#define F16_2_DIVBY_3 ((tFrac16) 0x5555)
```

## 6.217 F32\_2\_DIVBY\_3

Two thirds in 32-bit resolution, the result is rounded for a better accuracy.

### Source File

```
#include <SWLIBS_Defines.h>
```

### Macro Definition

```
#define F32_2_DIVBY_3 ((tFrac32) 0x55555555)
```

## 6.218 FRAC16

Macro converting a signed fractional [-1,1) number into a 16-bit fixed point number in format Q1.15.

### Source File

```
#include <SWLIBS_Defines.h>
```

### Macro Definition

```
#define FRAC16 ((tFrac16) (((x) < SFRACT_MAX) ? (((x) >= SFRACT_MIN) ? ((x)*32768.0) : INT16_MIN) : INT16_MAX))
```

## 6.219 FRAC32

Macro converting a signed fractional [-1,1) number into a 32-bit fixed point number in format Q1.31.

### Source File

```
#include <SWLIBS_Defines.h>
```

### Macro Definition

```
#define FRAC32 ((tFrac32) (((x) < FRACT_MAX) ? (((x) >= FRACT_MIN) ? ((x)*2147483648.0) : INT32_MIN) : INT32_MAX))
```

## 6.220 FLOAT\_DIVBY\_SQRT3

One over sqrt(3) in single precision floating point format.

### Source File

```
#include <SWLIBS_Defines.h>
```

**Macro Definition**

```
#define FLOAT_DIVBY_SQRT3 ((tFloat) 0.5773502691896258)
```

**6.221 FLOAT\_SQRT3**

Sqrt(3) in single precision floating point format.

**Source File**

```
#include <SWLIBS_Defines.h>
```

**Macro Definition**

```
#define FLOAT_SQRT3 ((tFloat) 1.732050776481628)
```

**6.222 FLOAT\_SQRT3\_DIVBY\_2**

Sqrt(3) divided by two in single precision floating point format.

**Source File**

```
#include <SWLIBS_Defines.h>
```

**Macro Definition**

```
#define FLOAT_SQRT3_DIVBY_2 ((tFloat) 0.866025403784439)
```

**6.223 FLOAT\_SQRT3\_DIVBY\_4**

Sqrt(3) divided by four in single precision floating point format.

**Source File**

```
#include <SWLIBS_Defines.h>
```

**Macro Definition**

```
#define FLOAT_SQRT3_DIVBY_4 ((tFloat) 0.4330127018922190)
```

**6.224 FLOAT\_SQRT3\_DIVBY\_4\_CORRECTION**

Sqrt(3) divided by four correction constant.

**Source File**

```
#include <SWLIBS_Defines.h>
```

**Macro Definition**

```
#define FLOAT_SQRT3_DIVBY_4_CORRECTION ((tFloat) 0)
```

## 6.225 FLOAT\_2\_PI

$2^* \pi$  in single precision floating point format.

### Source File

```
#include <SWLIBS_Defines.h>
```

### Macro Definition

```
#define FLOAT_2_PI ((tFloat) 6.28318530717958)
```

## 6.226 FLOAT\_PI

$\pi$  in single precision floating point format.

### Source File

```
#include <SWLIBS_Defines.h>
```

### Macro Definition

```
#define FLOAT_PI ((tFloat) 3.14159265358979)
```

## 6.227 FLOAT\_PI\_DIVBY\_2

$\pi/2$  in single precision floating point format.

### Source File

```
#include <SWLIBS_Defines.h>
```

### Macro Definition

```
#define FLOAT_PI_DIVBY_2 ((tFloat) 1.57079632679490)
```

## 6.228 FLOAT\_TAN\_PI\_DIVBY\_6

$\tan(\pi/6)$  in single precision floating point format.

### Source File

```
#include <SWLIBS_Defines.h>
```

### Macro Definition

```
#define FLOAT_TAN_PI_DIVBY_6 ((tFloat) 0.577350269189626000)
```

## 6.229 FLOAT\_TAN\_PI\_DIVBY\_12

$\tan(\pi/12)$  in single precision floating point format.

**Source File**

```
#include <SWLIBS_Defines.h>
```

**Macro Definition**

```
#define FLOAT_TAN_PI_DIVBY_12 ((tFloat)0.267949192431123000)
```

**6.230 FLOAT\_PI\_DIVBY\_6**

$\pi/6$  in single precision floating point format.

**Source File**

```
#include <SWLIBS_Defines.h>
```

**Macro Definition**

```
#define FLOAT_PI_DIVBY_6 ((tFloat)0.523598775598299000)
```

**6.231 FLOAT\_PI\_SINGLE\_CORRECTION**

Double to single precision correction constant for  $\pi$ , equal to ( $\pi(\text{Double}) - \pi(\text{Single})$ ).

**Source File**

```
#include <SWLIBS_Defines.h>
```

**Macro Definition**

```
#define FLOAT_PI_SINGLE_CORRECTION  
((tFloat)4.371139006309477e-08)
```

**6.232 FLOAT\_PI\_CORRECTION**

Double to single precision correction constant for  $\pi$ , equal to ( $2 * (\pi(\text{Double}) - \pi(\text{Single}))$ ).

**Source File**

```
#include <SWLIBS_Defines.h>
```

**Macro Definition**

```
#define FLOAT_PI_CORRECTION ((tFloat)8.742278012618954e-08)
```

**6.233 FLOAT\_PI\_DIVBY\_4**

$\pi/4$  in single precision floating point format.

**Source File**

```
#include <SWLIBS_Defines.h>
```

**Macro Definition**

```
#define FLOAT_PI_DIVBY_4 ((tFloat) 0.7853981633974480)
```

**6.234 FLOAT\_4\_DIVBY\_PI**

Number four divided by  $\pi$  in single precision floating point format.

**Source File**

```
#include <SWLIBS_Defines.h>
```

**Macro Definition**

```
#define FLOAT_4_DIVBY_PI ((tFloat) 1.2732395447351600)
```

**6.235 FLOAT\_0\_5**

Value 0.5 in single precision floating point format.

**Source File**

```
#include <SWLIBS_Defines.h>
```

**Macro Definition**

```
#define FLOAT_0_5 ((tFloat) 0.5)
```

**6.236 FLOAT\_MINUS\_0\_5**

Value -0.5 in single precision floating point format.

**Source File**

```
#include <SWLIBS_Defines.h>
```

**Macro Definition**

```
#define FLOAT_MINUS_0_5 ((tFloat) -0.5)
```

**6.237 FLOAT\_PLUS\_1**

Value 1 in single precision floating point format.

**Source File**

```
#include <SWLIBS_Defines.h>
```

**Macro Definition**

```
#define FLOAT_PLUS_1 ((tFloat) 1)
```

## 6.238 FLOAT\_MINUS\_1

Value -1 in single precision floating point format.

### Source File

```
#include <SWLIBS_Defines.h>
```

### Macro Definition

```
#define FLOAT_MINUS_1 ((tFloat) -1)
```

## 6.239 FLOAT\_MIN\_NORM

Constant representing the smallest positive normalized 32-bit floating-point value.

### Source File

```
#include <SWLIBS_Defines.h>
```

### Macro Definition

```
#define FLOAT_MIN_NORM ((tFloat) 1.175494350822288e-38)
```

## 6.240 AMMCLIB\_FLT\_EPS

Machine epsilon for single precision floating-point data type.

### Source File

```
#include <SWLIBS_Defines.h>
```

### Macro Definition

```
#define AMMCLIB_FLT_EPS ((tFloat) 1.175494350822288e-38)
```

## 6.241 SWLIBS\_2Syst

Definition of SWLIBS\_2Syst as alias for SWLIBS\_2Syst\_F16 array in case the 16-bit fractional implementation is selected.

### Source File

```
#include <SWLIBS_Typedefs.h>
```

### Macro Definition

```
#define SWLIBS_2Syst SWLIBS_2Syst_F16
```

## 6.242 SWLIBS\_3Syst

Definition of SWLIBS\_3Syst as alias for SWLIBS\_3Syst\_F16 array in case the 16-bit fractional implementation is selected.

**Source File**

```
#include <SWLIBS_Typedefs.h>
```

**Macro Definition**

```
#define SWLIBS_3Syst SWLIBS_3Syst_F16
```

## 6.243 SWLIBS\_ID

Library identification string.

**Source File**

```
#include <SWLIBS_Version.h>
```

**Macro Definition**

```
#define SWLIBS_ID { (unsigned char)0x90U, (unsigned char)0x71U,  
(unsigned char)0x77U, (unsigned char)0x68U}
```

## Contents

<b>1</b>	<b>Introduction .....</b>	<b>3</b>	2.4.5	Function AMCLIB_FWSetState .....	119
1.1	Architecture Overview .....	3	2.4.5.1	Function AMCLIB_FWSetState_F32 .....	119
1.2	General Information .....	4	2.4.5.2	Function AMCLIB_FWSetState_F16 .....	121
1.3	Multiple Implementation Support .....	4	2.4.5.3	Function AMCLIB_FWSetState_FLT .....	123
1.4	Supported Compilers .....	6	2.4.6	Function AMCLIB_FWDebug .....	125
1.5	MATLAB Integration .....	6	2.4.6.1	Function AMCLIB_FWDebug_F32 .....	126
1.6	Installation .....	7	2.4.6.2	Function AMCLIB_FWDebug_F16 .....	129
1.7	Library File Structure .....	10	2.4.6.3	Function AMCLIB_FWDebug_FLT .....	131
1.8	Integration Assumption .....	12	2.5	Function AMCLIB_FWSpeedLoop .....	134
1.9	Library Integration into a Green Hills Multi Development Environment .....	12	2.5.1	Function AMCLIB_FWSpeedLoop_F32 .....	135
1.10	Library Integration into a IAR Project .....	14	2.5.2	Function AMCLIB_FWSpeedLoop_F16 .....	139
1.11	Library Integration into a S32 Design Studio IDE for Arm based MCUs .....	18	2.5.3	Function AMCLIB_FWSpeedLoop_FLT .....	143
1.12	Library Testing .....	21	2.5.4	Function AMCLIB_FWSpeedLoopInit .....	146
1.13	Functions Accuracy .....	24	2.5.4.1	Function AMCLIB_FWSpeedLoopInit_F32 ....	146
<b>2</b>	<b>Functions .....</b>	<b>26</b>	2.5.4.2	Function AMCLIB_FWSpeedLoopInit_F16 ....	148
2.1	Function index .....	26	2.5.4.3	Function AMCLIB_FWSpeedLoopInit_FLT ....	151
2.2	Function AMCLIB_BemfObsrvDQ .....	46	2.5.5	Function AMCLIB_FWSpeedLoopSetState ...	153
2.2.1	Function AMCLIB_BemfObsrvDQ_F32 .....	48	2.5.5.1	Function AMCLIB_FWSpeedLoopSetState_	
2.2.2	Function AMCLIB_BemfObsrvDQ_F16 .....	54	2.5.5.2	F32 .....	153
2.2.3	Function AMCLIB_BemfObsrvDQ_FLT .....	60	2.5.5.3	Function AMCLIB_FWSpeedLoopSetState_	
2.2.4	Function AMCLIB_BemfObsrvDQInit .....	64	2.5.6	F16 .....	156
2.2.4.1	Function AMCLIB_BemfObsrvDQInit_F32 .....	64	2.5.6.1	Function AMCLIB_FWSpeedLoopSetState_	
2.2.4.2	Function AMCLIB_BemfObsrvDQInit_F16 .....	66	2.5.6.2	FLT .....	158
2.2.4.3	Function AMCLIB_BemfObsrvDQInit_FLT .....	68	2.5.6.3	Function AMCLIB_FWSpeedLoopDebug .....	160
2.2.5	Function AMCLIB_BemfObsrvDQSetState .....	70	2.6	Function AMCLIB_FWSpeedLoopDebug_	
2.2.5.1	Function AMCLIB_BemfObsrvDQSetState_		2.6.1	F32 .....	161
	F32 .....	71	2.6.2	Function AMCLIB_FWSpeedLoopDebug_	
2.2.5.2	Function AMCLIB_BemfObsrvDQSetState_		2.6.3	F16 .....	164
	F16 .....	73	2.6.4	Function AMCLIB_FWSpeedLoopDebug_	
2.2.5.3	Function AMCLIB_BemfObsrvDQSetState_		2.6.4.1	FLT .....	167
	FLT .....	75	2.6.4.2	Function AMCLIB_SpeedLoop .....	169
2.3	Function AMCLIB_CurrentLoop .....	77	2.6.4.3	Function AMCLIB_SpeedLoop_F32 .....	170
2.3.1	Function AMCLIB_CurrentLoop_F32 .....	78	2.6.5	Function AMCLIB_SpeedLoop_F16 .....	173
2.3.2	Function AMCLIB_CurrentLoop_F16 .....	82	2.6.5.1	Function AMCLIB_SpeedLoop_FLT .....	176
2.3.3	Function AMCLIB_CurrentLoop_FLT .....	86	2.6.5.2	Function AMCLIB_SpeedLoopInit .....	178
2.3.4	Function AMCLIB_CurrentLoopInit .....	89	2.6.5.3	Function AMCLIB_SpeedLoopInit_F32 .....	179
2.3.4.1	Function AMCLIB_CurrentLoopInit_F32 .....	89	2.6.6	Function AMCLIB_SpeedLoopInit_F16 .....	180
2.3.4.2	Function AMCLIB_CurrentLoopInit_F16 .....	91	2.6.6.1	Function AMCLIB_SpeedLoopInit_FLT .....	182
2.3.4.3	Function AMCLIB_CurrentLoopInit_FLT .....	93	2.6.6.2	Function AMCLIB_SpeedLoopSetState .....	184
2.3.5	Function AMCLIB_CurrentLoopSetState .....	94	2.6.6.3	Function AMCLIB_SpeedLoopSetState_	
2.3.5.1	Function AMCLIB_CurrentLoopSetState_		2.7	F32 .....	184
	F32 .....	95	2.6.7.1	Function AMCLIB_SpeedLoopSetState_	
2.3.5.2	Function AMCLIB_CurrentLoopSetState_		2.7.1	F16 .....	186
	F16 .....	97	2.6.7.2	Function AMCLIB_SpeedLoopSetState_	
2.3.5.3	Function AMCLIB_CurrentLoopSetState_		2.7.3	FLT .....	188
	FLT .....	99	2.6.7.4	Function AMCLIB_SpeedLoopDebug .....	190
2.4	Function AMCLIB_FW .....	101	2.6.6.1	Function AMCLIB_SpeedLoopDebug_F32 .....	191
2.4.1	Function AMCLIB_FW_F32 .....	104	2.6.6.2	Function AMCLIB_SpeedLoopDebug_F16 .....	193
2.4.2	Function AMCLIB_FW_F16 .....	107	2.6.6.3	Function AMCLIB_SpeedLoopDebug_FLT .....	195
2.4.3	Function AMCLIB_FW_FLT .....	110	2.7	Function AMCLIB_TrackObsrv .....	197
2.4.4	Function AMCLIB_FWInit .....	113	2.7.1	Function AMCLIB_TrackObsrv_F32 .....	200
2.4.4.1	Function AMCLIB_FWInit_F32 .....	113	2.7.2	Function AMCLIB_TrackObsrv_F16 .....	204
2.4.4.2	Function AMCLIB_FWInit_F16 .....	115	2.7.3	Function AMCLIB_TrackObsrv_FLT .....	209
2.4.4.3	Function AMCLIB_FWInit_FLT .....	117	2.7.4	Function AMCLIB_TrackObsrvInit .....	211
			2.7.4.1	Function AMCLIB_TrackObsrvInit_F32 .....	211
			2.7.4.2	Function AMCLIB_TrackObsrvInit_F16 .....	213

2.7.4.3	Function AMCLIB_TrackObsrvInit_FLT .....	215	2.14.3	Function GFLIB_Asin_FLT .....	288
2.7.5	Function AMCLIB_TrackObsrvSetState .....	216	2.15	Function GFLIB_Atan .....	290
2.7.5.1	Function AMCLIB_TrackObsrvSetState_ F32 .....	217	2.15.1	Function GFLIB_Atan_F32 .....	291
2.7.5.2	Function AMCLIB_TrackObsrvSetState_ F16 .....	219	2.15.2	Function GFLIB_Atan_F16 .....	293
2.7.5.3	Function AMCLIB_TrackObsrvSetState_ FLT .....	220	2.15.3	Function GFLIB_Atan_FLT .....	294
2.8	Function AMCLIB_Windmilling .....	222	2.16	Function GFLIB_AtanYX .....	296
2.8.1	Function AMCLIB_Windmilling_F32 .....	224	2.16.1	Function GFLIB_AtanYX_F32 .....	296
2.8.2	Function AMCLIB_Windmilling_F16 .....	226	2.16.2	Function GFLIB_AtanYX_F16 .....	298
2.8.3	Function AMCLIB_Windmilling_FLT .....	228	2.16.3	Function GFLIB_AtanYX_FLT .....	299
2.8.4	Function AMCLIB_WindmillingInit .....	230	2.17	Function GFLIB_AtanYXShifted .....	300
2.8.4.1	Function AMCLIB_WindmillingInit_F32 .....	230	2.17.1	Function GFLIB_AtanYXShifted_F32 .....	301
2.8.4.2	Function AMCLIB_WindmillingInit_F16 .....	232	2.17.2	Function GFLIB_AtanYXShifted_F16 .....	305
2.8.4.3	Function AMCLIB_WindmillingInit_FLT .....	234	2.17.3	Function GFLIB_AtanYXShifted_FLT .....	309
2.9	Function GDFLIB_FilterFIR .....	235	2.18	Function GFLIB_ControllerPIDpAW .....	312
2.9.1	Function GDFLIB_FilterFIR_F32 .....	236	2.18.1	Function GFLIB_ControllerPIDpAW_F32 .....	314
2.9.2	Function GDFLIB_FilterFIR_F16 .....	238	2.18.2	Function GFLIB_ControllerPIDpAW_F16 .....	317
2.9.3	Function GDFLIB_FilterFIR_FLT .....	240	2.18.3	Function GFLIB_ControllerPIDpAW_FLT .....	321
2.9.4	Function GDFLIB_FilterFIRInit .....	242	2.18.4	Function GFLIB_ControllerPIDpAWInit .....	323
2.9.4.1	Function GDFLIB_FilterFIRInit_F32 .....	243	2.18.4.1	Function GFLIB_ControllerPIDpAWInit_F32 ..	324
2.9.4.2	Function GDFLIB_FilterFIRInit_F16 .....	245	2.18.4.2	Function GFLIB_ControllerPIDpAWInit_F16 ..	325
2.9.4.3	Function GDFLIB_FilterFIRInit_FLT .....	247	2.18.4.3	Function GFLIB_ControllerPIDpAWInit_FLT ..	327
2.10	Function GDFLIB_FilterIIR1 .....	249	2.18.5	Function GFLIB_ControllerPIDpAWSetState .....	328
2.10.1	Function GDFLIB_FilterIIR1_F32 .....	250	2.18.5.1	Function GFLIB_ControllerPIDpAWSetState .....	328
2.10.2	Function GDFLIB_FilterIIR1_F16 .....	252	2.18.5.2	Function GFLIB_ControllerPIDpAWSetState_F32 .....	329
2.10.3	Function GDFLIB_FilterIIR1_FLT .....	254	2.18.5.3	Function GFLIB_ControllerPIDpAWSetState_F16 ..	330
2.10.4	Function GDFLIB_FilterIIR1Init .....	255	2.19	Function GFLIB_ControllerPlp .....	333
2.10.4.1	Function GDFLIB_FilterIIR1Init_F32 .....	255	2.19.1	Function GFLIB_ControllerPlp_F32 .....	335
2.10.4.2	Function GDFLIB_FilterIIR1Init_F16 .....	256	2.19.2	Function GFLIB_ControllerPlp_F16 .....	337
2.10.4.3	Function GDFLIB_FilterIIR1Init_FLT .....	257	2.19.3	Function GFLIB_ControllerPlp_FLT .....	340
2.11	Function GDFLIB_FilterIIR2 .....	257	2.19.4	Function GFLIB_ControllerPlpInit .....	341
2.11.1	Function GDFLIB_FilterIIR2_F32 .....	259	2.19.4.1	Function GFLIB_ControllerPlpInit_F32 .....	341
2.11.2	Function GDFLIB_FilterIIR2_F16 .....	261	2.19.4.2	Function GFLIB_ControllerPlpInit_F16 .....	342
2.11.3	Function GDFLIB_FilterIIR2_FLT .....	262	2.19.4.3	Function GFLIB_ControllerPlpInit_FLT .....	344
2.11.4	Function GDFLIB_FilterIIR2Init .....	264	2.19.5	Function GFLIB_ControllerPlpSetState .....	345
2.11.4.1	Function GDFLIB_FilterIIR2Init_F32 .....	264	2.19.5.1	Function GFLIB_ControllerPlpSetState_F32 ..	346
2.11.4.2	Function GDFLIB_FilterIIR2Init_F16 .....	265	2.19.5.2	Function GFLIB_ControllerPlpSetState_F16 ..	347
2.11.4.3	Function GDFLIB_FilterIIR2Init_FLT .....	266	2.19.5.3	Function GFLIB_ControllerPlpSetState_FLT .....	349
2.12	Function GDFLIB_FilterMA .....	267	2.20	Function GFLIB_ControllerPlpAW .....	350
2.12.1	Function GDFLIB_FilterMA_F32 .....	267	2.20.1	Function GFLIB_ControllerPlpAW_F32 .....	352
2.12.2	Function GDFLIB_FilterMA_F16 .....	269	2.20.2	Function GFLIB_ControllerPlpAW_F16 .....	355
2.12.3	Function GDFLIB_FilterMA_FLT .....	270	2.20.3	Function GFLIB_ControllerPlpAW_FLT .....	358
2.12.4	Function GDFLIB_FilterMAInit .....	272	2.20.4	Function GFLIB_ControllerPlpAWInit .....	360
2.12.4.1	Function GDFLIB_FilterMAInit_F32 .....	272	2.20.4.1	Function GFLIB_ControllerPlpAWInit_F32 ..	360
2.12.4.2	Function GDFLIB_FilterMAInit_F16 .....	273	2.20.4.2	Function GFLIB_ControllerPlpAWInit_F16 ..	362
2.12.4.3	Function GDFLIB_FilterMAInit_FLT .....	273	2.20.4.3	Function GFLIB_ControllerPlpAWInit_FLT ..	363
2.12.5	Function GDFLIB_FilterMASetState .....	274	2.20.5	Function GFLIB_ControllerPlpAWSetState ..	365
2.12.5.1	Function GDFLIB_FilterMASetState_F32 .....	275	2.20.5.1	Function GFLIB_ControllerPlpAWSetState_F32 ..	365
2.12.5.2	Function GDFLIB_FilterMASetState_F16 .....	275	2.20.5.2	Function GFLIB_ControllerPlpAWSetState_F16 ..	367
2.12.5.3	Function GDFLIB_FilterMASetState_FLT .....	276	2.20.5.3	Function GFLIB_ControllerPlpAWSetState_FLT .....	368
2.13	Function GFLIB_Acos .....	277	2.21	Function GFLIB_ControllerPlpI .....	370
2.13.1	Function GFLIB_Acos_F32 .....	278	2.21.1	Function GFLIB_ControllerPlpI_F32 .....	371
2.13.2	Function GFLIB_Acos_F16 .....	280			
2.13.3	Function GFLIB_Acos_FLT .....	282			
2.14	Function GFLIB_Asin .....	283			
2.14.1	Function GFLIB_Asin_F32 .....	284			
2.14.2	Function GFLIB_Asin_F16 .....	286			

2.21.2	Function <code>GFLIB_ControllerPIr_F16</code> .....	374	2.31.1	Function <code>GFLIB_Ramp_F32</code> .....	452
2.21.3	Function <code>GFLIB_ControllerPIr_FLT</code> .....	377	2.31.2	Function <code>GFLIB_Ramp_F16</code> .....	453
2.21.4	Function <code>GFLIB_ControllerPIrInit</code> .....	378	2.31.3	Function <code>GFLIB_Ramp_FLT</code> .....	454
2.21.4.1	Function <code>GFLIB_ControllerPIrInit_F32</code> .....	379	2.32	Function <code>GFLIB_Sign</code> .....	455
2.21.4.2	Function <code>GFLIB_ControllerPIrInit_F16</code> .....	380	2.32.1	Function <code>GFLIB_Sign_F32</code> .....	456
2.21.4.3	Function <code>GFLIB_ControllerPIrInit_FLT</code> .....	382	2.32.2	Function <code>GFLIB_Sign_F16</code> .....	457
2.21.5	Function <code>GFLIB_ControllerPIrSetState</code> .....	383	2.32.3	Function <code>GFLIB_Sign_FLT</code> .....	458
2.21.5.1	Function <code>GFLIB_ControllerPIrSetState_F32</code> ..	383	2.33	Function <code>GFLIB_Sin</code> .....	459
2.21.5.2	Function <code>GFLIB_ControllerPIrSetState_F16</code> ..	385	2.33.1	Function <code>GFLIB_Sin_F32</code> .....	459
2.21.5.3	Function <code>GFLIB_ControllerPIrSetState_FLT</code> ..	386	2.33.2	Function <code>GFLIB_Sin_F16</code> .....	460
2.22	Function <code>GFLIB_ControllerPIrAW</code> .....	388	2.33.3	Function <code>GFLIB_Sin_FLT</code> .....	461
2.22.1	Function <code>GFLIB_ControllerPIrAW_F32</code> .....	389	2.34	Function <code>GFLIB_SinCos</code> .....	462
2.22.2	Function <code>GFLIB_ControllerPIrAW_F16</code> .....	392	2.34.1	Function <code>GFLIB_SinCos_F32</code> .....	463
2.22.3	Function <code>GFLIB_ControllerPIrAW_FLT</code> .....	396	2.34.2	Function <code>GFLIB_SinCos_F16</code> .....	464
2.22.4	Function <code>GFLIB_ControllerPIrAWInit</code> .....	398	2.34.3	Function <code>GFLIB_SinCos_FLT</code> .....	465
2.22.4.1	Function <code>GFLIB_ControllerPIrAWInit_F32</code> ..	398	2.35	Function <code>GFLIB_Sqrt</code> .....	467
2.22.4.2	Function <code>GFLIB_ControllerPIrAWInit_F16</code> ..	399	2.35.1	Function <code>GFLIB_Sqrt_F32</code> .....	467
2.22.4.3	Function <code>GFLIB_ControllerPIrAWInit_FLT</code> ..	401	2.35.2	Function <code>GFLIB_Sqrt_F16</code> .....	468
2.22.5	Function <code>GFLIB_ControllerPIrAWSetState</code> ..	402	2.35.3	Function <code>GFLIB_Sqrt_FLT</code> .....	469
2.22.5.1	Function <code>GFLIB_ControllerPIrAWSetState_</code> <code>F32</code> .....	403	2.36	Function <code>GFLIB_Tan</code> .....	470
2.22.5.2	Function <code>GFLIB_ControllerPIrAWSetState_</code> <code>F16</code> .....	404	2.36.1	Function <code>GFLIB_Tan_F32</code> .....	470
2.22.5.3	Function <code>GFLIB_ControllerPIrAWSetState_</code> <code>FLT</code> .....	406	2.36.2	Function <code>GFLIB_Tan_F16</code> .....	471
2.23	Function <code>GFLIB_Cos</code> .....	407	2.36.3	Function <code>GFLIB_Tan_FLT</code> .....	473
2.23.1	Function <code>GFLIB_Cos_F32</code> .....	408	2.37	Function <code>GFLIB_UpperLimit</code> .....	474
2.23.2	Function <code>GFLIB_Cos_F16</code> .....	409	2.37.1	Function <code>GFLIB_UpperLimit_F32</code> .....	475
2.23.3	Function <code>GFLIB_Cos_FLT</code> .....	410	2.37.2	Function <code>GFLIB_UpperLimit_F16</code> .....	476
2.24	Function <code>GFLIB_Hyst</code> .....	412	2.37.3	Function <code>GFLIB_UpperLimit_FLT</code> .....	477
2.24.1	Function <code>GFLIB_Hyst_F32</code> .....	413	2.38	Function <code>GFLIB_VectorLimit</code> .....	478
2.24.2	Function <code>GFLIB_Hyst_F16</code> .....	414	2.38.1	Function <code>GFLIB_VectorLimit_F32</code> .....	479
2.24.3	Function <code>GFLIB_Hyst_FLT</code> .....	415	2.38.2	Function <code>GFLIB_VectorLimit_F16</code> .....	480
2.25	Function <code>GFLIB_IntegratorTR</code> .....	416	2.38.3	Function <code>GFLIB_VectorLimit_FLT</code> .....	482
2.25.1	Function <code>GFLIB_IntegratorTR_F32</code> .....	417	2.39	Function <code>GFLIB_VLog10_FLT</code> .....	483
2.25.2	Function <code>GFLIB_IntegratorTR_F16</code> .....	419	2.40	Function <code>GFLIB_VMin</code> .....	484
2.25.3	Function <code>GFLIB_IntegratorTR_FLT</code> .....	421	2.40.1	Function <code>GFLIB_VMin_F32</code> .....	484
2.25.4	Function <code>GFLIB_IntegratorTRSetState</code> .....	422	2.40.2	Function <code>GFLIB_VMin_F16</code> .....	485
2.25.4.1	Function <code>GFLIB_IntegratorTRSetState_F32</code> ..	422	2.40.3	Function <code>GFLIB_VMin_FLT</code> .....	486
2.25.4.2	Function <code>GFLIB_IntegratorTRSetState_F16</code> ..	424	2.40.4	Function <code>GFLIB_VMin4_F16</code> .....	487
2.25.4.3	Function <code>GFLIB_IntegratorTRSetState_FLT</code> ..	425	2.40.5	Function <code>GFLIB_VMin5_F16</code> .....	487
2.26	Function <code>GFLIB_Limit</code> .....	426	2.40.6	Function <code>GFLIB_VMin6_F16</code> .....	488
2.26.1	Function <code>GFLIB_Limit_F32</code> .....	426	2.40.7	Function <code>GFLIB_VMin7_F16</code> .....	489
2.26.2	Function <code>GFLIB_Limit_F16</code> .....	427	2.40.8	Function <code>GFLIB_VMin8_F16</code> .....	489
2.26.3	Function <code>GFLIB_Limit_FLT</code> .....	428	2.40.9	Function <code>GFLIB_VMin9_F16</code> .....	490
2.27	Function <code>GFLIB_Log10_FLT</code> .....	429	2.40.10	Function <code>GFLIB_VMin10_F16</code> .....	491
2.28	Function <code>GFLIB_LowerLimit</code> .....	430	2.40.11	Function <code>GFLIB_VMin11_F16</code> .....	491
2.28.1	Function <code>GFLIB_LowerLimit_F32</code> .....	431	2.40.12	Function <code>GFLIB_VMin12_F16</code> .....	492
2.28.2	Function <code>GFLIB_LowerLimit_F16</code> .....	432	2.40.13	Function <code>GFLIB_VMin13_F16</code> .....	493
2.28.3	Function <code>GFLIB_LowerLimit_FLT</code> .....	433	2.40.14	Function <code>GFLIB_VMin14_F16</code> .....	493
2.29	Function <code>GFLIB_Lut1D</code> .....	434	2.40.15	Function <code>GFLIB_VMin15_F16</code> .....	494
2.29.1	Function <code>GFLIB_Lut1D_F32</code> .....	434	2.40.16	Function <code>GFLIB_VMin16_F16</code> .....	495
2.29.2	Function <code>GFLIB_Lut1D_F16</code> .....	437	2.41	Function <code>GMCLIB_BetaProjection</code> .....	495
2.29.3	Function <code>GFLIB_Lut1D_FLT</code> .....	441	2.41.1	Function <code>GMCLIB_BetaProjection_F32</code> .....	496
2.30	Function <code>GFLIB_Lut2D</code> .....	444	2.41.2	Function <code>GMCLIB_BetaProjection_F16</code> .....	497
2.30.1	Function <code>GFLIB_Lut2D_F32</code> .....	445	2.41.3	Function <code>GMCLIB_BetaProjection_FLT</code> .....	498
2.30.2	Function <code>GFLIB_Lut2D_F16</code> .....	447	2.42	Function <code>GMCLIB_BetaProjection3Ph</code> .....	499
2.30.3	Function <code>GFLIB_Lut2D_FLT</code> .....	449	2.42.1	Function <code>GMCLIB_BetaProjection3Ph_F32</code> ..	500
2.31	Function <code>GFLIB_Ramp</code> .....	451	2.42.2	Function <code>GMCLIB_BetaProjection3Ph_F16</code> ..	501
			2.42.3	Function <code>GMCLIB_BetaProjection3Ph_FLT</code> ..	502
			2.43	Function <code>GMCLIB_Clark</code> .....	503
			2.43.1	Function <code>GMCLIB_Clark_F32</code> .....	504

2.43.2	Function GMCLIB_Clark_F16 .....	505	2.58.2	Function MLIB_Add_F16 .....	587
2.43.3	Function GMCLIB_Clark_FLT .....	506	2.58.3	Function MLIB_Add_FLT .....	588
2.44	Function GMCLIB_ClarkInv .....	507	2.59	Function MLIB_AddSat .....	589
2.44.1	Function GMCLIB_ClarkInv_F32 .....	507	2.59.1	Function MLIB_AddSat_F32 .....	589
2.44.2	Function GMCLIB_ClarkInv_F16 .....	508	2.59.2	Function MLIB_AddSat_F16 .....	591
2.44.3	Function GMCLIB_ClarkInv_FLT .....	509	2.60	Function MLIB_Convert .....	592
2.45	Function GMCLIB_DecouplingPMSM .....	510	2.60.1	Function MLIB_Convert_F32F16 .....	592
2.45.1	Function GMCLIB_DecouplingPMSM_F32 .....	512	2.60.2	Function MLIB_Convert_F32FLT .....	593
2.45.2	Function GMCLIB_DecouplingPMSM_F16 .....	515	2.60.3	Function MLIB_Convert_F16F32 .....	595
2.45.3	Function GMCLIB_DecouplingPMSM_FLT .....	517	2.60.4	Function MLIB_Convert_F16FLT .....	596
2.46	Function GMCLIB_ElimDcBusRip .....	519	2.60.5	Function MLIB_Convert_FLTF16 .....	597
2.46.1	Function GMCLIB_ElimDcBusRip_F32 .....	520	2.60.6	Function MLIB_Convert_FLTF32 .....	599
2.46.2	Function GMCLIB_ElimDcBusRip_F16 .....	522	2.61	Function MLIB_ConvertPU .....	600
2.46.3	Function GMCLIB_ElimDcBusRip_FLT .....	524	2.61.1	Function MLIB_ConvertPU_F32F16 .....	600
2.47	Function GMCLIB_Park .....	527	2.61.2	Function MLIB_ConvertPU_F32FLT .....	601
2.47.1	Function GMCLIB_Park_F32 .....	527	2.61.3	Function MLIB_ConvertPU_F16F32 .....	602
2.47.2	Function GMCLIB_Park_F16 .....	529	2.61.4	Function MLIB_ConvertPU_F16FLT .....	603
2.47.3	Function GMCLIB_Park_FLT .....	530	2.61.5	Function MLIB_ConvertPU_FLTF16 .....	604
2.48	Function GMCLIB_ParkInv .....	531	2.61.6	Function MLIB_ConvertPU_FLTF32 .....	605
2.48.1	Function GMCLIB_ParkInv_F32 .....	531	2.62	Function MLIB_Div .....	606
2.48.2	Function GMCLIB_ParkInv_F16 .....	533	2.62.1	Function MLIB_Div_F32 .....	606
2.48.3	Function GMCLIB_ParkInv_FLT .....	534	2.62.2	Function MLIB_Div_F16 .....	608
2.49	Function GMCLIB_PwmIct .....	535	2.62.3	Function MLIB_Div_FLT .....	609
2.49.1	Function GMCLIB_PwmIct_F32 .....	536	2.63	Function MLIB_DivSat .....	610
2.49.2	Function GMCLIB_PwmIct_F16 .....	537	2.63.1	Function MLIB_DivSat_F32 .....	610
2.49.3	Function GMCLIB_PwmIct_FLT .....	538	2.63.2	Function MLIB_DivSat_F16 .....	611
2.50	Function GMCLIB_SvmSci .....	540	2.64	Function MLIB_Mac .....	612
2.50.1	Function GMCLIB_SvmSci_F32 .....	542	2.64.1	Function MLIB_Mac_F32 .....	612
2.50.2	Function GMCLIB_SvmSci_F16 .....	544	2.64.2	Function MLIB_Mac_F32F16F16 .....	614
2.50.3	Function GMCLIB_SvmSci_FLT .....	545	2.64.3	Function MLIB_Mac_F16 .....	615
2.51	Function GMCLIB_SvmStd .....	546	2.64.4	Function MLIB_Mac_FLT .....	616
2.51.1	Function GMCLIB_SvmStd_F32 .....	557	2.65	Function MLIB_MacSat .....	617
2.51.2	Function GMCLIB_SvmStd_F16 .....	559	2.65.1	Function MLIB_MacSat_F32 .....	618
2.51.3	Function GMCLIB_SvmStd_FLT .....	560	2.65.2	Function MLIB_MacSat_F32F16F16 .....	619
2.52	Function GMCLIB_SvmU0n .....	561	2.65.3	Function MLIB_MacSat_F16 .....	620
2.52.1	Function GMCLIB_SvmU0n_F32 .....	563	2.66	Function MLIB_Mnac .....	621
2.52.2	Function GMCLIB_SvmU0n_F16 .....	564	2.66.1	Function MLIB_Mnac_F32 .....	622
2.52.3	Function GMCLIB_SvmU0n_FLT .....	566	2.66.2	Function MLIB_Mnac_F32F16F16 .....	623
2.53	Function GMCLIB_SvmU7n .....	567	2.66.3	Function MLIB_Mnac_F16 .....	624
2.53.1	Function GMCLIB_SvmU7n_F32 .....	569	2.66.4	Function MLIB_Mnac_FLT .....	625
2.53.2	Function GMCLIB_SvmU7n_F16 .....	571	2.67	Function MLIB_Msu .....	626
2.53.3	Function GMCLIB_SvmU7n_FLT .....	572	2.67.1	Function MLIB_Msu_F32 .....	627
2.54	Function GMCLIB_VRot .....	573	2.67.2	Function MLIB_Msu_F32F16F16 .....	628
2.54.1	Function GMCLIB_VRot_F32 .....	574	2.67.3	Function MLIB_Msu_F16 .....	629
2.54.2	Function GMCLIB_VRot_F16 .....	575	2.67.4	Function MLIB_Msu_FLT .....	630
2.54.3	Function GMCLIB_VRot_FLT .....	576	2.68	Function MLIB_Mul .....	632
2.55	Function GMCLIB_VUnit .....	577	2.68.1	Function MLIB_Mul_F32 .....	632
2.55.1	Function GMCLIB_VUnit_F32 .....	577	2.68.2	Function MLIB_Mul_F32F16F16 .....	633
2.55.2	Function GMCLIB_VUnit_F16 .....	578	2.68.3	Function MLIB_Mul_F16 .....	634
2.55.3	Function GMCLIB_VUnit_FLT .....	579	2.68.4	Function MLIB_Mul_FLT .....	635
2.56	Function MLIB_Abs .....	580	2.69	Function MLIB_MulSat .....	636
2.56.1	Function MLIB_Abs_F32 .....	580	2.69.1	Function MLIB_MulSat_F32 .....	637
2.56.2	Function MLIB_Abs_F16 .....	581	2.69.2	Function MLIB_MulSat_F32F16F16 .....	638
2.56.3	Function MLIB_Abs_FLT .....	582	2.69.3	Function MLIB_MulSat_F16 .....	639
2.57	Function MLIB_AbsSat .....	583	2.70	Function MLIB_Neg .....	640
2.57.1	Function MLIB_AbsSat_F32 .....	583	2.70.1	Function MLIB_Neg_F32 .....	640
2.57.2	Function MLIB_AbsSat_F16 .....	585	2.70.2	Function MLIB_Neg_F16 .....	641
2.58	Function MLIB_Add .....	586	2.70.3	Function MLIB_Neg_FLT .....	642
2.58.1	Function MLIB_Add_F32 .....	586	2.71	Function MLIB_NegSat .....	643

2.71.1	Function MLIB_NegSat_F32 .....	644	5.6	AMCLIB_CURRENT_LOOP_T_FLT .....	690
2.71.2	Function MLIB_NegSat_F16 .....	645	5.7	AMCLIB_FW_DEBUG_T_F16 .....	691
2.72	Function MLIB_Norm .....	646	5.8	AMCLIB_FW_DEBUG_T_F32 .....	692
2.72.1	Function MLIB_Norm_F32 .....	646	5.9	AMCLIB_FW_DEBUG_T_FLT .....	692
2.72.2	Function MLIB_Norm_F16 .....	647	5.10	AMCLIB_FW_SPEED_LOOP_DEBUG_T_	
2.73	Function MLIB_RndSat_F16F32 .....	648		F16 .....	693
2.74	Function MLIB_Round .....	649	5.11	AMCLIB_FW_SPEED_LOOP_DEBUG_T_	
2.74.1	Function MLIB_Round_F32 .....	649		F32 .....	694
2.74.2	Function MLIB_Round_F16 .....	650	5.12	AMCLIB_FW_SPEED_LOOP_DEBUG_T_	
2.75	Function MLIB_ShBi .....	651		FLT .....	695
2.75.1	Function MLIB_ShBi_F32 .....	652	5.13	AMCLIB_FW_SPEED_LOOP_T_F16 .....	696
2.75.2	Function MLIB_ShBi_F16 .....	653	5.14	AMCLIB_FW_SPEED_LOOP_T_F32 .....	697
2.76	Function MLIB_ShBiSat .....	654	5.15	AMCLIB_FW_SPEED_LOOP_T_FLT .....	698
2.76.1	Function MLIB_ShBiSat_F32 .....	654	5.16	AMCLIB_FW_T_F16 .....	699
2.76.2	Function MLIB_ShBiSat_F16 .....	655	5.17	AMCLIB_FW_T_F32 .....	699
2.77	Function MLIB_ShL .....	656	5.18	AMCLIB_FW_T_FLT .....	700
2.77.1	Function MLIB_ShL_F32 .....	656	5.19	AMCLIB_SPEED_LOOP_DEBUG_T_F16 .....	700
2.77.2	Function MLIB_ShL_F16 .....	657	5.20	AMCLIB_SPEED_LOOP_DEBUG_T_F32 .....	701
2.78	Function MLIB_ShLSat .....	658	5.21	AMCLIB_SPEED_LOOP_DEBUG_T_FLT .....	701
2.78.1	Function MLIB_ShLSat_F32 .....	658	5.22	AMCLIB_SPEED_LOOP_T_F16 .....	702
2.78.2	Function MLIB_ShLSat_F16 .....	659	5.23	AMCLIB_SPEED_LOOP_T_F32 .....	702
2.79	Function MLIB_ShR .....	660	5.24	AMCLIB_SPEED_LOOP_T_FLT .....	702
2.79.1	Function MLIB_ShR_F32 .....	661	5.25	AMCLIB_TRACK_OBSRV_T_F16 .....	703
2.79.2	Function MLIB_ShR_F16 .....	662	5.26	AMCLIB_TRACK_OBSRV_T_F32 .....	703
2.80	Function MLIB_Sub .....	663	5.27	AMCLIB_TRACK_OBSRV_T_FLT .....	703
2.80.1	Function MLIB_Sub_F32 .....	663	5.28	AMCLIB_WINDMILLING_T_F16 .....	704
2.80.2	Function MLIB_Sub_F16 .....	664	5.29	AMCLIB_WINDMILLING_T_F32 .....	704
2.80.3	Function MLIB_Sub_FLT .....	665	5.30	AMCLIB_WINDMILLING_T_FLT .....	705
2.81	Function MLIB_SubSat .....	666	5.31	GDFLIB_FILTER_IIR1_COEFF_T_F16 .....	706
2.81.1	Function MLIB_SubSat_F32 .....	667	5.32	GDFLIB_FILTER_IIR1_COEFF_T_F32 .....	706
2.81.2	Function MLIB_SubSat_F16 .....	668	5.33	GDFLIB_FILTER_IIR1_COEFF_T_FLT .....	707
2.82	Function MLIB_VAdd .....	669	5.34	GDFLIB_FILTER_IIR1_T_F16 .....	707
2.82.1	Function MLIB_VAdd_F32 .....	669	5.35	GDFLIB_FILTER_IIR1_T_F32 .....	707
2.82.2	Function MLIB_VAdd_F16 .....	670	5.36	GDFLIB_FILTER_IIR1_T_FLT .....	708
2.82.3	Function MLIB_VAdd_FLT .....	672	5.37	GDFLIB_FILTER_IIR2_COEFF_T_F16 .....	708
2.83	Function MLIB_VMac .....	673	5.38	GDFLIB_FILTER_IIR2_COEFF_T_F32 .....	709
2.83.1	Function MLIB_VMac_F32 .....	673	5.39	GDFLIB_FILTER_IIR2_COEFF_T_FLT .....	709
2.83.2	Function MLIB_VMac_F32F16F16 .....	674	5.40	GDFLIB_FILTER_IIR2_T_F16 .....	710
2.83.3	Function MLIB_VMac_F16 .....	676	5.41	GDFLIB_FILTER_IIR2_T_F32 .....	710
2.83.4	Function MLIB_VMac_FLT .....	677	5.42	GDFLIB_FILTER_IIR2_T_FLT .....	711
2.84	Function MLIB_VScale .....	678	5.43	GDFLIB_FILTER_MA_T_F16 .....	711
2.84.1	Function MLIB_VScale_F32 .....	679	5.44	GDFLIB_FILTER_MA_T_F32 .....	711
2.84.2	Function MLIB_VScale_F16 .....	680	5.45	GDFLIB_FILTER_MA_T_FLT .....	712
2.84.3	Function MLIB_VScale_FLT .....	681	5.46	GDFLIB_FILTERFIR_PARAM_T_F16 .....	712
2.85	Function MLIB_VSub .....	682	5.47	GDFLIB_FILTERFIR_PARAM_T_F32 .....	712
2.85.1	Function MLIB_VSub_F32 .....	682	5.48	GDFLIB_FILTERFIR_PARAM_T_FLT .....	713
2.85.2	Function MLIB_VSub_F16 .....	683	5.49	GDFLIB_FILTERFIR_STATE_T_F16 .....	713
2.85.3	Function MLIB_VSub_FLT .....	685	5.50	GDFLIB_FILTERFIR_STATE_T_F32 .....	713
2.86	Function SWLIBS_GetVersion .....	686	5.51	GDFLIB_FILTERFIR_STATE_T_FLT .....	714
3	TypeDefs .....	686	5.52	GFLIB_ACOS_T_F16 .....	714
4	Enums .....	687	5.53	GFLIB_ACOS_T_F32 .....	714
4.1	AMCLIB_WINDMILLING_RET_T .....	687	5.54	GFLIB_ACOS_T_FLT .....	715
4.2	tBool .....	687	5.55	GFLIB_ACOS_TAYLOR_COEF_T_F16 .....	715
5	Compound data types .....	687	5.56	GFLIB_ACOS_TAYLOR_COEF_T_F32 .....	716
5.1	AMCLIB_BEMF_OBSRV_DQ_T_F16 .....	687	5.57	GFLIB_ASIN_T_F16 .....	716
5.2	AMCLIB_BEMF_OBSRV_DQ_T_F32 .....	688	5.58	GFLIB_ASIN_T_F32 .....	717
5.3	AMCLIB_BEMF_OBSRV_DQ_T_FLT .....	689	5.59	GFLIB_ASIN_T_FLT .....	717
5.4	AMCLIB_CURRENT_LOOP_T_F16 .....	690	5.60	GFLIB_ASIN_TAYLOR_COEF_T_F16 .....	717
5.5	AMCLIB_CURRENT_LOOP_T_F32 .....	690	5.61	GFLIB_ASIN_TAYLOR_COEF_T_F32 .....	718

5.62	GFLIB_ATAN_T_F16 .....	718	5.121	GFLIB_UPPERLIMIT_T_F16 .....	743
5.63	GFLIB_ATAN_T_F32 .....	719	5.122	GFLIB_UPPERLIMIT_T_F32 .....	743
5.64	GFLIB_ATAN_T_FLT .....	719	5.123	GFLIB_UPPERLIMIT_T_FLT .....	744
5.65	GFLIB_ATAN_TAYLOR_COEF_T_F16 .....	720	5.124	GFLIB_VECTORLIMIT_T_F16 .....	744
5.66	GFLIB_ATAN_TAYLOR_COEF_T_F32 .....	720	5.125	GFLIB_VECTORLIMIT_T_F32 .....	744
5.67	GFLIB_ATANYXSHIFTED_T_F16 .....	720	5.126	GFLIB_VECTORLIMIT_T_FLT .....	744
5.68	GFLIB_ATANYXSHIFTED_T_F32 .....	721	5.127	GFLIB_VLOG10_T_FLT .....	745
5.69	GFLIB_ATANYXSHIFTED_T_FLT .....	721	5.128	GMCLIB_DECOUPLINGPMSM_T_F16 .....	745
5.70	GFLIB_CONTROLLER_PI_P_T_F16 .....	721	5.129	GMCLIB_DECOUPLINGPMSM_T_F32 .....	745
5.71	GFLIB_CONTROLLER_PI_P_T_F32 .....	722	5.130	GMCLIB_DECOUPLINGPMSM_T_FLT .....	746
5.72	GFLIB_CONTROLLER_PI_P_T_FLT .....	722	5.131	GMCLIB_ELIMDCBUSRIP_T_F16 .....	746
5.73	GFLIB_CONTROLLER_PI_R_T_F16 .....	723	5.132	GMCLIB_ELIMDCBUSRIP_T_F32 .....	746
5.74	GFLIB_CONTROLLER_PI_R_T_F32 .....	723	5.133	GMCLIB_ELIMDCBUSRIP_T_FLT .....	747
5.75	GFLIB_CONTROLLER_PI_R_T_FLT .....	724	5.134	SWLIBS_2Syst_F16 .....	747
5.76	GFLIB_CONTROLLER_PIAW_P_T_F16 .....	724	5.135	SWLIBS_2Syst_F32 .....	747
5.77	GFLIB_CONTROLLER_PIAW_P_T_F32 .....	725	5.136	SWLIBS_2Syst_FLT .....	748
5.78	GFLIB_CONTROLLER_PIAW_P_T_FLT .....	725	5.137	SWLIBS_3Syst_F16 .....	748
5.79	GFLIB_CONTROLLER_PIAW_R_T_F16 .....	726	5.138	SWLIBS_3Syst_F32 .....	748
5.80	GFLIB_CONTROLLER_PIAW_R_T_F32 .....	727	5.139	SWLIBS_3Syst_FLT .....	749
5.81	GFLIB_CONTROLLER_PIAW_R_T_FLT .....	727	5.140	SWLIBS_VERSION_T .....	749
5.82	GFLIB_CONTROLLER_PID_P_AW_T_F16 ..	728	<b>6</b>	<b>Macros .....</b>	<b>749</b>
5.83	GFLIB_CONTROLLER_PID_P_AW_T_F32 ..	729	6.1	AMCLIB_BEMF_OBSRV_DQ_DEFAULT_F32 .....	749
5.84	GFLIB_CONTROLLER_PID_P_AW_T_FLT ..	729	6.2	AMCLIB_BEMF_OBSRV_DQ_DEFAULT_F16 .....	750
5.85	GFLIB_COS_T_F16 .....	730	6.3	AMCLIB_BEMF_OBSRV_DQ_DEFAULT_FLT .....	750
5.86	GFLIB_COS_T_F32 .....	730	6.4	AMCLIB_CURRENT_LOOP_DEFAULT_F32 .....	750
5.87	GFLIB_COS_T_FLT .....	731	6.5	AMCLIB_CURRENT_LOOP_DEFAULT_F16 .....	751
5.88	GFLIB_HYST_T_F16 .....	731	6.6	AMCLIB_CURRENT_LOOP_DEFAULT_FLT .....	751
5.89	GFLIB_HYST_T_F32 .....	731	6.7	AMCLIB_FW_DEFAULT_F32 .....	751
5.90	GFLIB_HYST_T_FLT .....	732	6.8	AMCLIB_FW_DEFAULT_F16 .....	752
5.91	GFLIB_INTEGRATOR_TR_T_F16 .....	732	6.9	AMCLIB_FW_DEFAULT_FLT .....	752
5.92	GFLIB_INTEGRATOR_TR_T_F32 .....	733	6.10	AMCLIB_FW_SPEED_LOOP_DEFAULT_F32 .....	752
5.93	GFLIB_INTEGRATOR_TR_T_FLT .....	733	6.11	AMCLIB_FW_SPEED_LOOP_DEFAULT_F16 .....	752
5.94	GFLIB_LIMIT_T_F16 .....	733	6.12	AMCLIB_FW_SPEED_LOOP_DEFAULT_FLT .....	753
5.95	GFLIB_LIMIT_T_F32 .....	734	6.13	AMCLIB_SPEED_LOOP_DEFAULT_F32 .....	753
5.96	GFLIB_LIMIT_T_FLT .....	734	6.14	AMCLIB_SPEED_LOOP_DEFAULT_F16 .....	753
5.97	GFLIB_LOG10_T_FLT .....	734	6.15	AMCLIB_SPEED_LOOP_DEFAULT_FLT .....	754
5.98	GFLIB_LOWERLIMIT_T_F16 .....	735	6.16	AMCLIB_TRACK_OBSRV_T .....	754
5.99	GFLIB_LOWERLIMIT_T_F32 .....	735	6.17	AMCLIB_TRACK_OBSRV_DEFAULT .....	754
5.100	GFLIB_LOWERLIMIT_T_FLT .....	735	6.18	AMCLIB_TRACK_OBSRV_DEFAULT_F32 ..	754
5.101	GFLIB_LUT1D_T_F16 .....	736	6.19	AMCLIB_TRACK_OBSRV_DEFAULT_F16 ..	755
5.102	GFLIB_LUT1D_T_F32 .....	736	6.20	AMCLIB_TRACK_OBSRV_DEFAULT_FLT ..	755
5.103	GFLIB_LUT1D_T_FLT .....	736	6.21	GDFLIB_FILTERFIR_PARAM_T .....	755
5.104	GFLIB_LUT2D_T_F16 .....	737	6.22	GDFLIB_FILTERFIR_STATE_T .....	755
5.105	GFLIB_LUT2D_T_F32 .....	737	6.23	GDFLIB_FILTER_IIR1_T .....	756
5.106	GFLIB_LUT2D_T_FLT .....	737	6.24	GDFLIB_FILTER_IIR1_DEFAULT .....	756
5.107	GFLIB_RAMP_T_F16 .....	738	6.25	GDFLIB_FILTER_IIR1_DEFAULT_F32 ..	756
5.108	GFLIB_RAMP_T_F32 .....	738	6.26	GDFLIB_FILTER_IIR1_DEFAULT_F16 ..	756
5.109	GFLIB_RAMP_T_FLT .....	739	6.27	GDFLIB_FILTER_IIR1_DEFAULT_FLT ..	757
5.110	GFLIB_SIN_T_F16 .....	739	6.28	GDFLIB_FILTER_IIR2_T .....	757
5.111	GFLIB_SIN_T_F32 .....	739	6.29	GDFLIB_FILTER_IIR2_DEFAULT .....	757
5.112	GFLIB_SIN_T_FLT .....	739			
5.113	GFLIB_SINCOS_T_F16 .....	740			
5.114	GFLIB_SINCOS_T_F32 .....	740			
5.115	GFLIB_SINCOS_T_FLT .....	740			
5.116	GFLIB_TAN_T_F16 .....	741			
5.117	GFLIB_TAN_T_F32 .....	741			
5.118	GFLIB_TAN_T_FLT .....	742			
5.119	GFLIB_TAN_TAYLOR_COEF_T_F16 .....	742			
5.120	GFLIB_TAN_TAYLOR_COEF_T_F32 .....	743			

6.30	GDFLIB_FILTER_IIR2_DEFAULT_F32 .....	757	6.75	GFLIB_CONTROLLER_PI_R_DEFAULT_
6.31	GDFLIB_FILTER_IIR2_DEFAULT_F16 .....	758		FLT .....
6.32	GDFLIB_FILTER_IIR2_DEFAULT_FLT .....	758	6.76	GFLIB_CONTROLLER_PIAW_R_T .....
6.33	GDFLIB_FILTER_MA_T .....	758	6.77	GFLIB_CONTROLLER_PIAW_R_
6.34	GDFLIB_FILTER_MA_DEFAULT .....	759		DEFAULT .....
6.35	GDFLIB_FILTER_MA_DEFAULT_F32 .....	759	6.78	GFLIB_CONTROLLER_PIAW_R_
6.36	GDFLIB_FILTER_MA_DEFAULT_F16 .....	759		DEFAULT_F32 .....
6.37	GDFLIB_FILTER_MA_DEFAULT_FLT .....	759	6.79	GFLIB_CONTROLLER_PIAW_R_
6.38	GFLIB_ACOS_T .....	760		DEFAULT_F16 .....
6.39	GFLIB_ACOS_DEFAULT .....	760	6.80	GFLIB_CONTROLLER_PIAW_R_
6.40	GFLIB_ACOS_DEFAULT_F32 .....	760		DEFAULT_FLT .....
6.41	GFLIB_ACOS_DEFAULT_F16 .....	760	6.81	GFLIB_COS_T .....
6.42	GFLIB_ACOS_DEFAULT_FLT .....	760	6.82	GFLIB_COS_DEFAULT .....
6.43	GFLIB_ASIN_FLT_MIN .....	761	6.83	GFLIB_COS_DEFAULT_F32 .....
6.44	GFLIB_ASIN_FLT_INT1 .....	761	6.84	GFLIB_COS_DEFAULT_F16 .....
6.45	GFLIB_ASIN_T .....	761	6.85	GFLIB_COS_DEFAULT_FLT .....
6.46	GFLIB_ASIN_DEFAULT .....	761	6.86	GFLIB_HYST_T .....
6.47	GFLIB_ASIN_DEFAULT_F32 .....	762	6.87	GFLIB_HYST_DEFAULT .....
6.48	GFLIB_ASIN_DEFAULT_F16 .....	762	6.88	GFLIB_HYST_DEFAULT_F32 .....
6.49	GFLIB_ASIN_DEFAULT_FLT .....	762	6.89	GFLIB_HYST_DEFAULT_F16 .....
6.50	GFLIB_ATAN_T .....	762	6.90	GFLIB_HYST_DEFAULT_FLT .....
6.51	GFLIB_ATAN_DEFAULT .....	763	6.91	GFLIB_INTEGRATOR_TR_T .....
6.52	GFLIB_ATAN_DEFAULT_F32 .....	763	6.92	GFLIB_INTEGRATOR_TR_DEFAULT .....
6.53	GFLIB_ATAN_DEFAULT_F16 .....	763	6.93	GFLIB_INTEGRATOR_TR_DEFAULT_F32 ..
6.54	GFLIB_ATAN_DEFAULT_FLT .....	763	6.94	GFLIB_INTEGRATOR_TR_DEFAULT_F16 ..
6.55	GFLIB_ATANYXSHIFTED_T .....	763	6.95	GFLIB_INTEGRATOR_TR_DEFAULT_FLT ..
6.56	GFLIB_CONTROLLER_PID_P_AW_T .....	764	6.96	GFLIB_LIMIT_T .....
6.57	GFLIB_CONTROLLER_PID_P_AW_		6.97	GFLIB_LIMIT_DEFAULT .....
	DEFAULT .....	764	6.98	GFLIB_LIMIT_DEFAULT_F32 .....
6.58	GFLIB_CONTROLLER_PID_P_AW_		6.99	GFLIB_LIMIT_DEFAULT_F16 .....
	DEFAULT_F32 .....	764	6.100	GFLIB_LIMIT_DEFAULT_FLT .....
6.59	GFLIB_CONTROLLER_PID_P_AW_		6.101	GFLIB_LOG10_DEFAULT_FLT .....
	DEFAULT_F16 .....	765	6.102	GFLIB_LOWERLIMIT_T .....
6.60	GFLIB_CONTROLLER_PID_P_AW_		6.103	GFLIB_LOWERLIMIT_DEFAULT .....
	DEFAULT_FLT .....	765	6.104	GFLIB_LOWERLIMIT_DEFAULT_F32 .....
6.61	GFLIB_CONTROLLER_PI_P_T .....	765	6.105	GFLIB_LOWERLIMIT_DEFAULT_F16 .....
6.62	GFLIB_CONTROLLER_PI_P_DEFAULT .....	765	6.106	GFLIB_LOWERLIMIT_DEFAULT_FLT .....
6.63	GFLIB_CONTROLLER_PI_P_DEFAULT_		6.107	GFLIB_LUT1D_T .....
	F32 .....	766	6.108	GFLIB_LUT1D_DEFAULT .....
6.64	GFLIB_CONTROLLER_PI_P_DEFAULT_		6.109	GFLIB_LUT1D_DEFAULT_F32 .....
	F16 .....	766	6.110	GFLIB_LUT1D_DEFAULT_F16 .....
6.65	GFLIB_CONTROLLER_PI_P_DEFAULT_		6.111	GFLIB_LUT1D_DEFAULT_FLT .....
	FLT .....	766	6.112	GFLIB_LUT2D_T .....
6.66	GFLIB_CONTROLLER_PIAW_P_T .....	766	6.113	GFLIB_LUT2D_DEFAULT .....
6.67	GFLIB_CONTROLLER_PIAW_P_		6.114	GFLIB_LUT2D_DEFAULT_F32 .....
	DEFAULT .....	767	6.115	GFLIB_LUT2D_DEFAULT_F16 .....
6.68	GFLIB_CONTROLLER_PIAW_P_		6.116	GFLIB_LUT2D_DEFAULT_FLT .....
	DEFAULT_F32 .....	767	6.117	GFLIB_RAMP_T .....
6.69	GFLIB_CONTROLLER_PIAW_P_		6.118	GFLIB_RAMP_DEFAULT .....
	DEFAULT_F16 .....	767	6.119	GFLIB_RAMP_DEFAULT_F32 .....
6.70	GFLIB_CONTROLLER_PIAW_P_		6.120	GFLIB_RAMP_DEFAULT_F16 .....
	DEFAULT_FLT .....	767	6.121	GFLIB_RAMP_DEFAULT_FLT .....
6.71	GFLIB_CONTROLLER_PI_R_T .....	768	6.122	GFLIB_SIN_FLT_MIN .....
6.72	GFLIB_CONTROLLER_PI_R_DEFAULT .....	768	6.123	GFLIB_SIN_T .....
6.73	GFLIB_CONTROLLER_PI_R_DEFAULT_		6.124	GFLIB_SIN_DEFAULT .....
	F32 .....	768	6.125	GFLIB_SIN_DEFAULT_F32 .....
6.74	GFLIB_CONTROLLER_PI_R_DEFAULT_		6.126	GFLIB_SIN_DEFAULT_F16 .....
	F16 .....	769	6.127	GFLIB_SIN_DEFAULT_FLT .....
			6.128	GFLIB_SINCOS_FLT_MIN .....

6.129	GFLIB_SINCOS_T .....	782	6.184	INT32_MIN .....	795
6.130	GFLIB_SINCOS_DEFAULT .....	782	6.185	FLOAT_MIN .....	795
6.131	GFLIB_SINCOS_DEFAULT_F32 .....	782	6.186	FLOAT_MAX .....	795
6.132	GFLIB_SINCOS_DEFAULT_F16 .....	782	6.187	INT16TOINT32 .....	796
6.133	GFLIB_SINCOS_DEFAULT_FLT .....	783	6.188	INT32TOINT16 .....	796
6.134	GFLIB_TAN_FLT_MIN .....	783	6.189	INT32TOINT64 .....	796
6.135	GFLIB_TAN_FLT_3P4 .....	783	6.190	INT64TOINT32 .....	796
6.136	GFLIB_TAN_FLT_PI .....	783	6.191	F16TOINT16 .....	796
6.137	GFLIB_TAN_FLT_CORR1 .....	784	6.192	F32TOINT16 .....	797
6.138	GFLIB_TAN_FLT_PI4 .....	784	6.193	F64TOINT16 .....	797
6.139	GFLIB_TAN_FLT_PI2 .....	784	6.194	F16TOINT32 .....	797
6.140	GFLIB_TAN_FLT_CORR2 .....	784	6.195	F32TOINT32 .....	797
6.141	GFLIB_TAN_T .....	784	6.196	F64TOINT32 .....	798
6.142	GFLIB_TAN_DEFAULT .....	785	6.197	F16TOINT64 .....	798
6.143	GFLIB_TAN_DEFAULT_F32 .....	785	6.198	F32TOINT64 .....	798
6.144	GFLIB_TAN_DEFAULT_F16 .....	785	6.199	F64TOINT64 .....	798
6.145	GFLIB_TAN_DEFAULT_FLT .....	785	6.200	INT16TOF16 .....	799
6.146	GFLIB_UPPERLIMIT_T .....	786	6.201	INT16TOF32 .....	799
6.147	GFLIB_UPPERLIMIT_DEFAULT .....	786	6.202	INT32TOF16 .....	799
6.148	GFLIB_UPPERLIMIT_DEFAULT_F32 .....	786	6.203	INT32TOF32 .....	799
6.149	GFLIB_UPPERLIMIT_DEFAULT_F16 .....	786	6.204	INT64TOF16 .....	800
6.150	GFLIB_UPPERLIMIT_DEFAULT_FLT .....	787	6.205	INT64TOF32 .....	800
6.151	GFLIB_VECTORLIMIT_T .....	787	6.206	F16_1_DIVBY_SQRT3 .....	800
6.152	GFLIB_VECTORLIMIT_DEFAULT .....	787	6.207	F32_1_DIVBY_SQRT3 .....	800
6.153	GFLIB_VECTORLIMIT_DEFAULT_F32 .....	787	6.208	F16_SQRT3_DIVBY_2 .....	801
6.154	GFLIB_VECTORLIMIT_DEFAULT_F16 .....	788	6.209	F16_SQRT3_DIVBY_4 .....	801
6.155	GFLIB_VECTORLIMIT_DEFAULT_FLT .....	788	6.210	F32_SQRT3_DIVBY_2 .....	801
6.156	GFLIB_VLOG10_DEFAULT_FLT .....	788	6.211	F32_SQRT3_DIVBY_4 .....	801
6.157	GMCLIB_DECOUPLINGPMSM_T .....	788	6.212	F16_SQRT2_DIVBY_2 .....	801
6.158	GMCLIB_DECOUPLINGPMSM_DEFAULT .....	789	6.213	F32_SQRT2_DIVBY_2 .....	802
6.159	GMCLIB_DECOUPLINGPMSM_DEFAULT_F32 .....	789	6.214	F16_1_DIVBY_3 .....	802
6.160	GMCLIB_DECOUPLINGPMSM_DEFAULT_F16 .....	789	6.215	F32_1_DIVBY_3 .....	802
6.161	GMCLIB_DECOUPLINGPMSM_DEFAULT_FLT .....	789	6.216	F16_2_DIVBY_3 .....	802
6.162	GMCLIB_ELIMDCBUSRIP_T .....	790	6.217	F32_2_DIVBY_3 .....	803
6.163	GMCLIB_ELIMDCBUSRIP_DEFAULT .....	790	6.218	FRAC16 .....	803
6.164	GMCLIB_ELIMDCBUSRIP_DEFAULT_F32 .....	790	6.219	FRAC32 .....	803
6.165	GMCLIB_ELIMDCBUSRIP_DEFAULT_F16 .....	790	6.220	FLOAT_DIVBY_SQRT3 .....	803
6.166	GMCLIB_ELIMDCBUSRIP_DEFAULT_FLT .....	791	6.221	FLOAT_SQRT3 .....	804
6.167	SWLIBS_SUPPORT_F32 .....	791	6.222	FLOAT_SQRT3_DIVBY_2 .....	804
6.168	SWLIBS_SUPPORT_F16 .....	791	6.223	FLOAT_SQRT3_DIVBY_4 .....	804
6.169	SWLIBS_SUPPORT_FLT .....	791	6.224	FLOAT_SQRT3_DIVBY_4_CORRECTION .....	804
6.170	SWLIBS_SUPPORTED_IMPLEMENTATION .....	791	6.225	FLOAT_2_PI .....	805
6.171	SFRACT_MIN .....	792	6.226	FLOAT_PI .....	805
6.172	SFRACT_MAX .....	792	6.227	FLOAT_PI_DIVBY_2 .....	805
6.173	FRACT_MIN .....	792	6.228	FLOAT_PI_DIVBY_6 .....	805
6.174	FRACT_MAX .....	792	6.229	FLOAT_PI_DIVBY_12 .....	805
6.175	FRAC32_0_5 .....	793	6.230	FLOAT_PI_DIVBY_6 .....	806
6.176	FRAC16_0_5 .....	793	6.231	FLOAT_PI_SINGLE_CORRECTION .....	806
6.177	FRAC32_0_25 .....	793	6.232	FLOAT_PI_CORRECTION .....	806
6.178	FRAC16_0_25 .....	793	6.233	FLOAT_PI_DIVBY_4 .....	806
6.179	UINT16_MAX .....	794	6.234	FLOAT_4_DIVBY_PI .....	807
6.180	INT16_MAX .....	794	6.235	FLOAT_0_5 .....	807
6.181	INT16_MIN .....	794	6.236	FLOAT_MINUS_0_5 .....	807
6.182	UINT32_MAX .....	794	6.237	FLOAT_PLUS_1 .....	807
6.183	INT32_MAX .....	795	6.238	FLOAT_MINUS_1 .....	808
			6.239	FLOAT_MIN_NORM .....	808
			6.240	AMMCLIB_FLT_EPS .....	808
			6.241	SWLIBS_2Syst .....	808
			6.242	SWLIBS_3Syst .....	808

---

6.243 SWLIBS\_ID ..... 809