

Automotive Math and Motor Control Library Set for NXP S32K11x devices

User's Guide

Rev. 8 — 15 April 2020

[User guide](#)



Revision History

Revision	Date	Description
1.0	30-June-2018	Initial version.
2.0	30-September-2018	The Service Release v1.1.14. Changed implementation of GFLIB_Ramp.
3.0	31-December-2018	The Service Release v1.1.15. See release notes for list of changes.
4.0	31-March-2019	The Service Release v1.1.16. See release notes for list of changes.
5.0	30-June-2019	The Service Release v1.1.17. See release notes for list of changes.
6.0	30-September-2019	The Service Release v1.1.18. See release notes for list of changes.
7.0	31-December-2019	The Service Release v1.1.19. See release notes for list of changes.
8.0	31-March-2020	The Service Release v1.1.20. 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 S32K11x 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 S32K11x devices consists of several sub-libraries, functionally connected as depicted in [Figure 1](#).

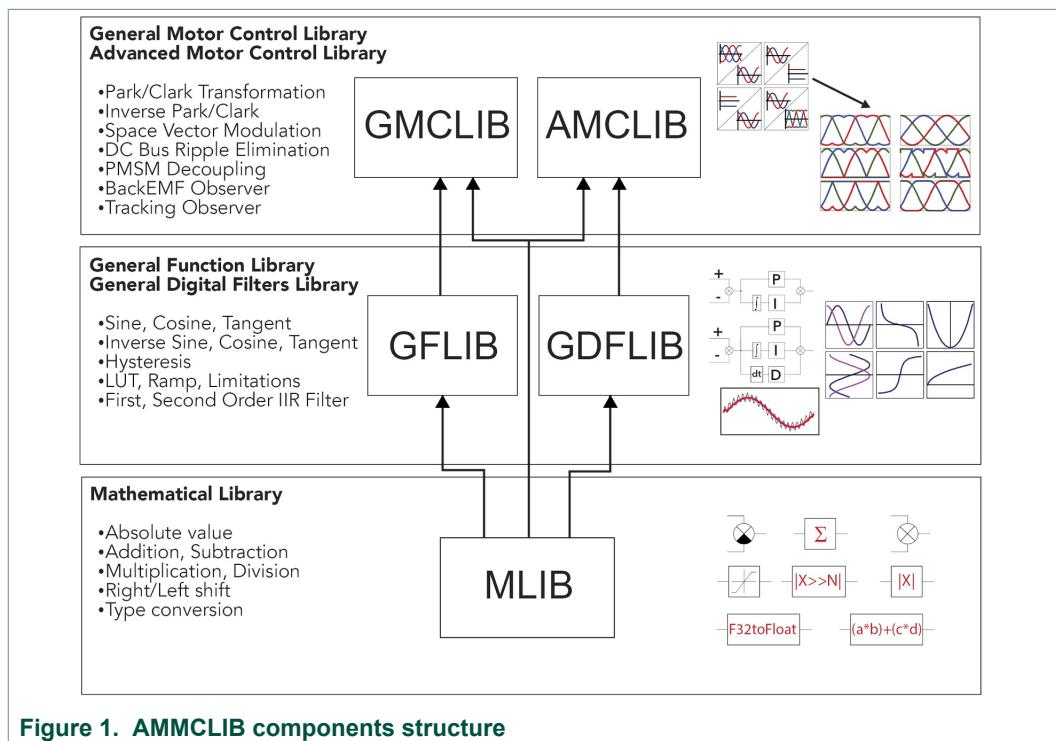


Figure 1. AMMCLIB components structure

The Automotive Math and Motor Control Library Set for NXP S32K11x 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 S32K11x 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 S32K11x devices was developed to support these major implementations:

- Fixed-point 32-bit fractional
- Fixed-point 16-bit fractional

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} .

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 S32K11x 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 S32K11x 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 S32K11x 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 and 16-bit fixed 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 S32K11x 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 S32K11x 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

After proper definition of `SWLIBS_DEFAULT_IMPLEMENTATION` macro the Automotive Math and Motor Control Library Set for NXP S32K11x 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 S32K11x 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 S32K11x 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

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 S32K11x 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

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 S32K11x devices is written in ANSI-C99 ISO/IEC 9899:1999 standard language. The library was built and tested using the following compilers:

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

The library is delivered in a library module "S32K11x_AMMCLIB.a" for the Green Hills compiler. The library module is located in "C:\NXP\AMMCLIB\S32K11x_AMMCLIB_v1.1.20\lib\ghs" folder (considering the default installation path).

The library is delivered in a library module "S32K11x_AMMCLIB.a" for the S32 Design Studio IDE for ARM based MCUs. The library module is located in "C:\NXP\AMMCLIB\S32K11x_AMMCLIB_v1.1.20\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\S32K11x_AMMCLIB_v1.1.20\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*, *gflib.h*, *gdflib.h* and *gmclib.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 S32K11x 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.mdl* 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 S32K11x devices is delivered as a single executable file.

Note: *The Automotive Math and Motor Control Library Set for NXP S32K11x 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 S32K11x 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

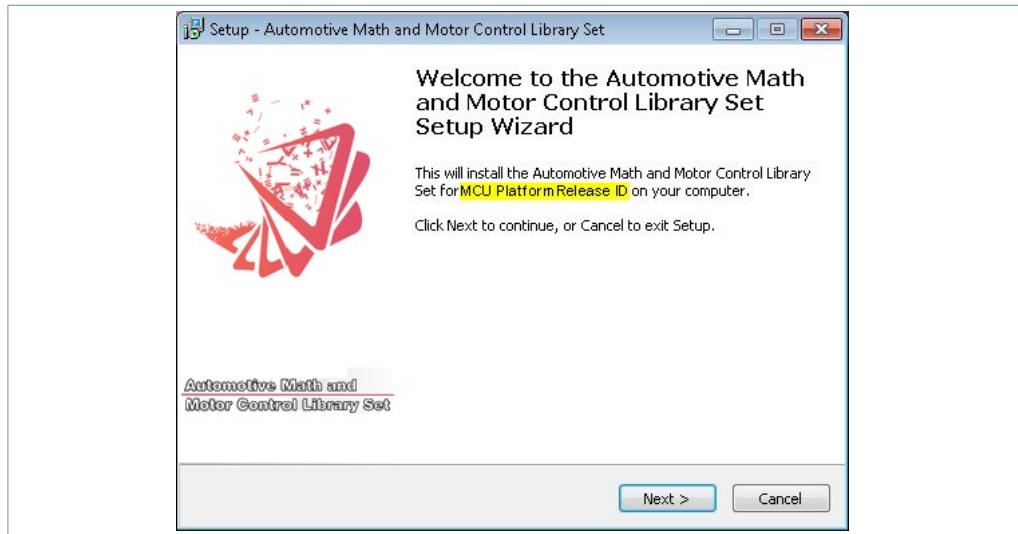


Figure 2. AMMCLib installation - step 1. Highlighted "MCU Platform" and "ReleaseID" identifies the actual release, which is the S32K11x_AMMCLIB_v1.1.20

2. Accept the license agreement

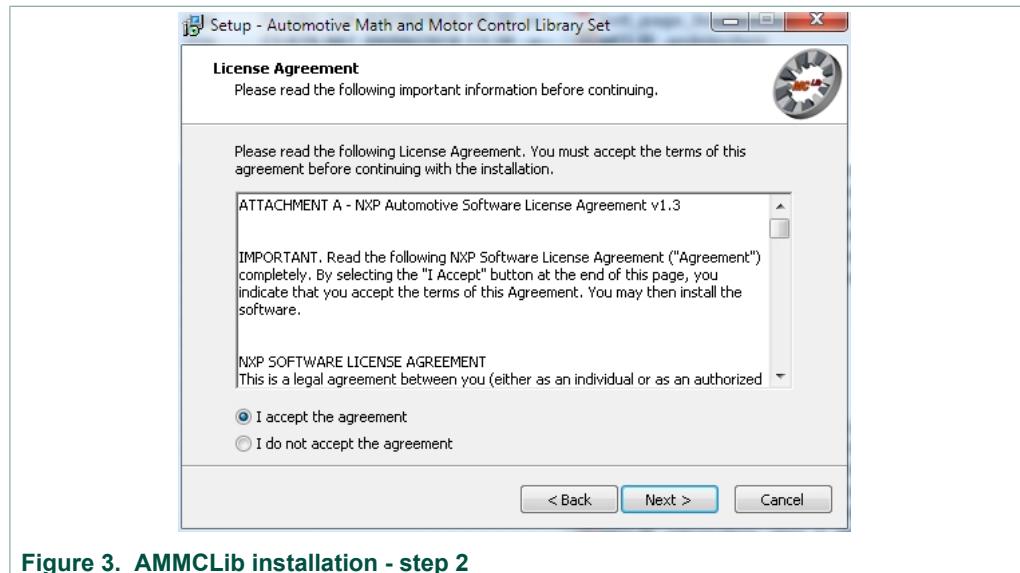


Figure 3. AMMCLib installation - step 2

3. Select the destination directory

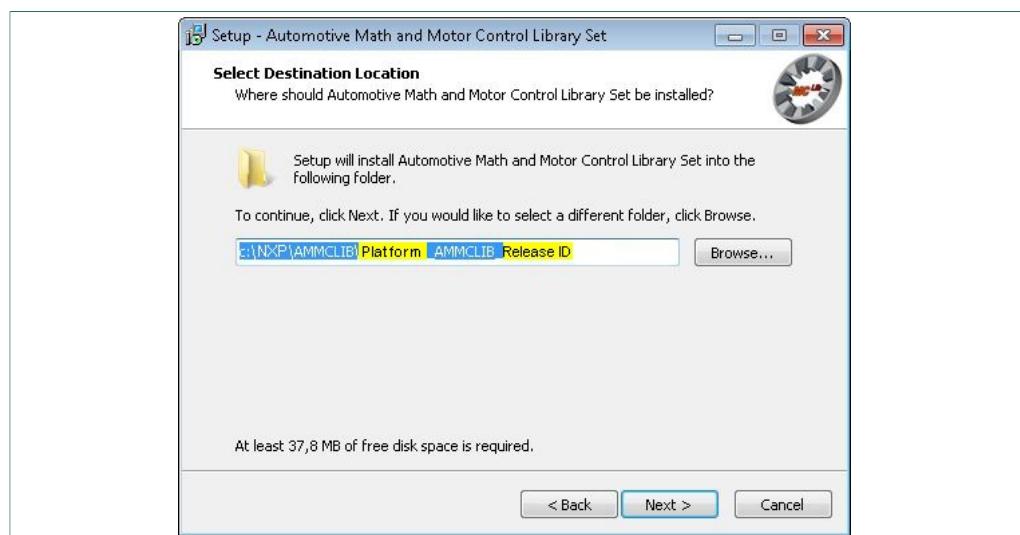


Figure 4. AMMCLib installation - step 3. Highlighted "Platform" and "ReleaseID" identifies the actual release installation path, which is the S32K11x_AMMCLIB_v1.1.20

4. Check the destination directory and confirm the installation

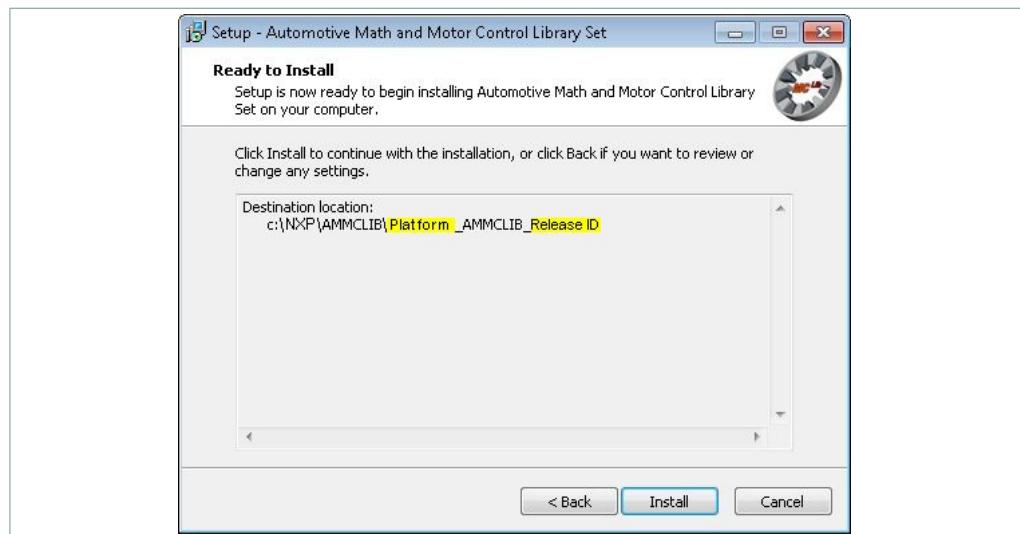


Figure 5. AMMCLib installation - step 4. Highlighted "Platform" and "ReleaseID" identifies the actual release installation path, which is the S32K11x_AMMCLIB_v1.1.20

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

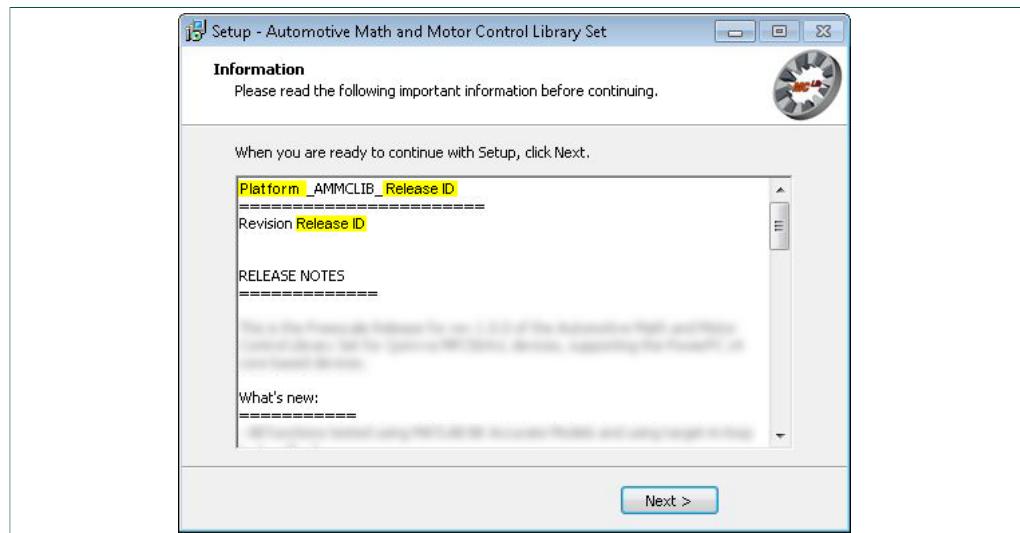


Figure 6. AMMCLib installation - step 5. Highlighted "Platform" and "ReleaseID" identifies the actual release, which is the S32K11x_AMMCLIB_v1.1.20

6. Select Finish to end the installation

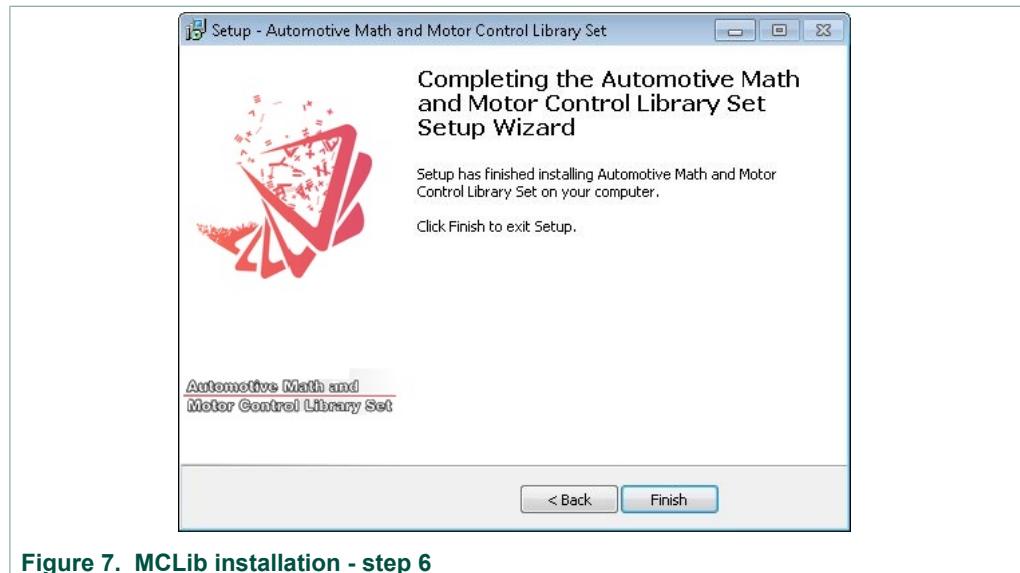


Figure 7. MCLib installation - step 6

1.7 Library File Structure

After a successful installation, the Automotive Math and Motor Control Library Set for NXP S32K11x devices is added by default into the "C:\NXP\AMMCLIB\|S32K11x_AMMCLIB_v1.1.20" subfolder. This folder will contain other nested subfolders and files required by the Automotive Math and Motor Control Library Set for NXP S32K11x devices, as shown in [Figure 8](#).

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

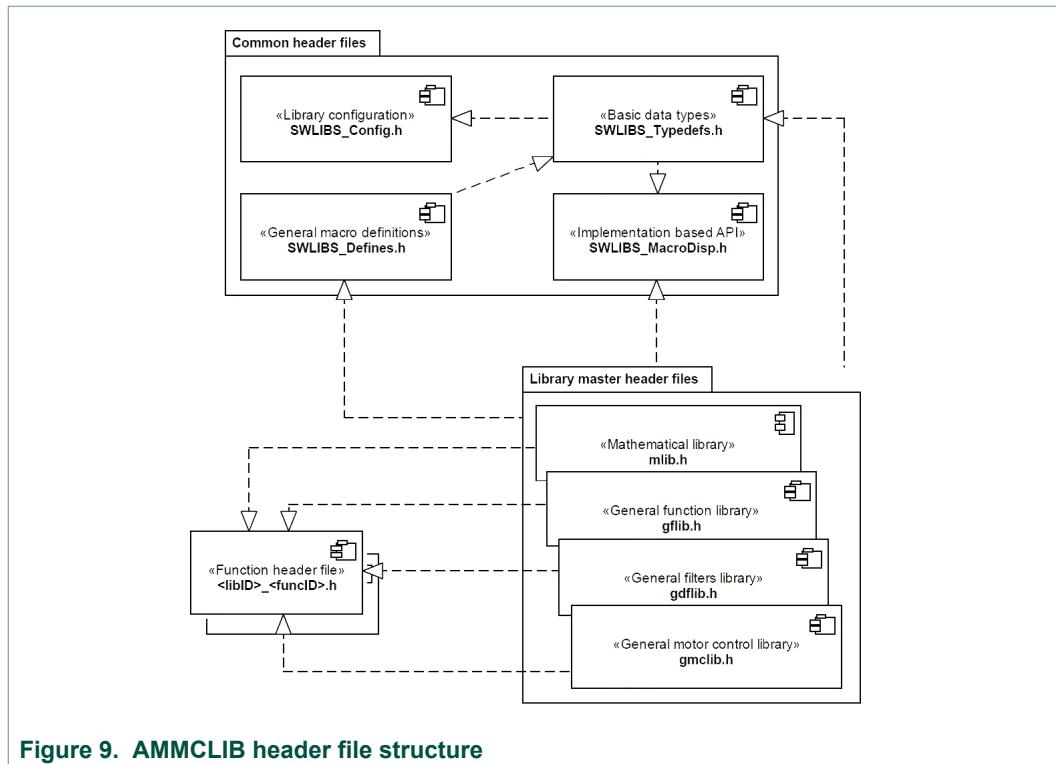
Figure 8. AMMCLIB directory structure

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 S32K11x devices into a new Green Hills compiler based project, the steps described in [Section 1.9](#) section must be performed.

The header files structure of the Automotive Math and Motor Control Library Set for NXP S32K11x 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 S32K11x 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).

1.9 Library Integration into a Green Hills Multi Development Environment

The Automotive Math and Motor Control Library Set for NXP S32K11x 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](#)

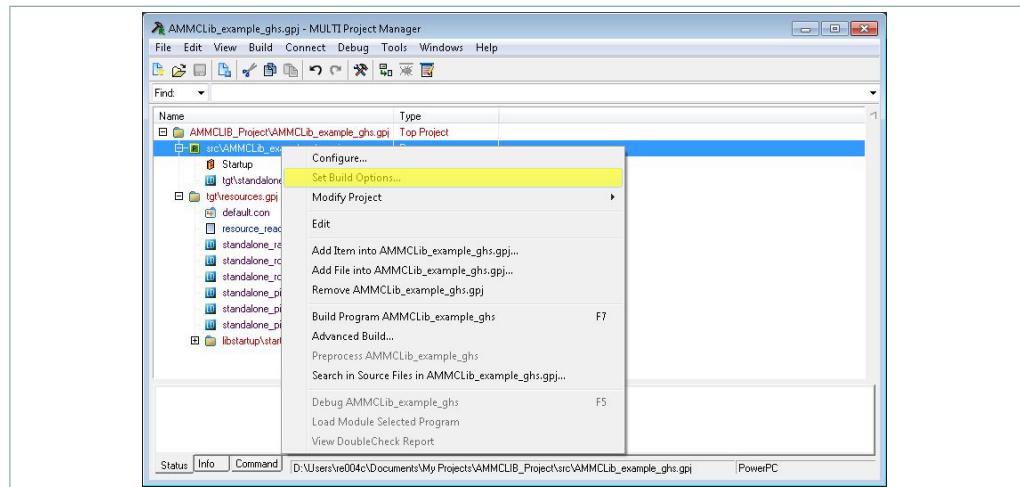


Figure 10. Project build options

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\S32K11x_AMMCLIB_v1.1.20\include".

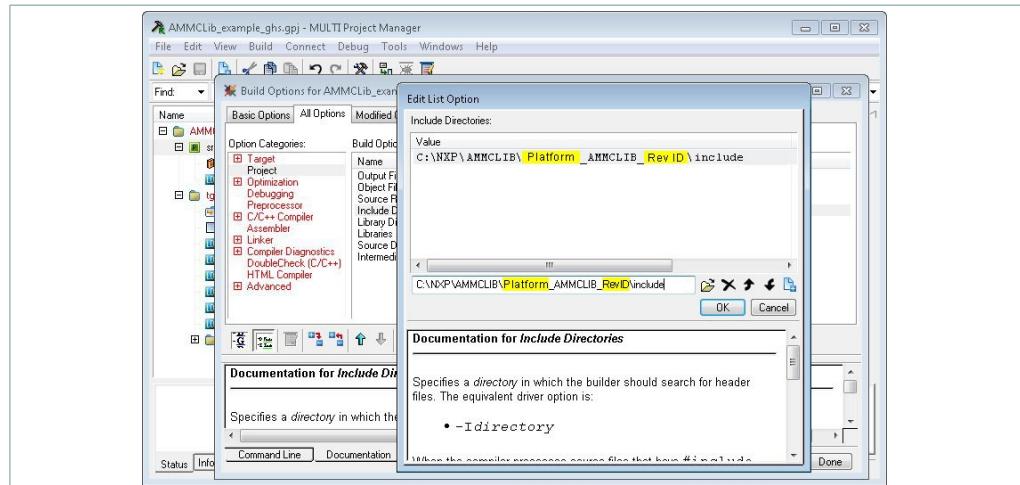


Figure 11. Adding a path to the library header files

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](#).

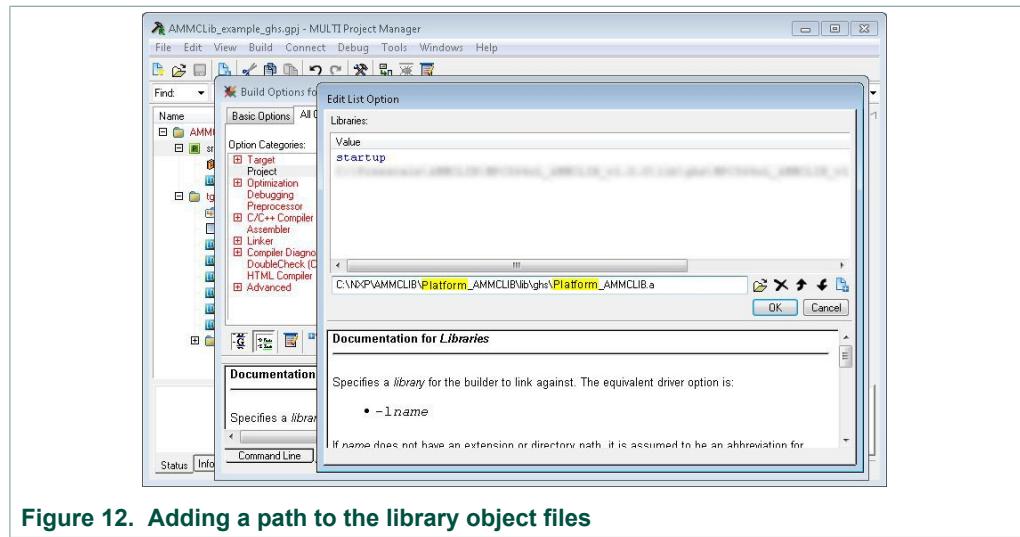


Figure 12. Adding a path to the library object files

Considering default settings, you should add "C:\NXP\AMMCLIB\IS32K11x_AMMCLIB_v1.1.20\lib\ghs\S32K11x_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*, *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 S32K11x 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.

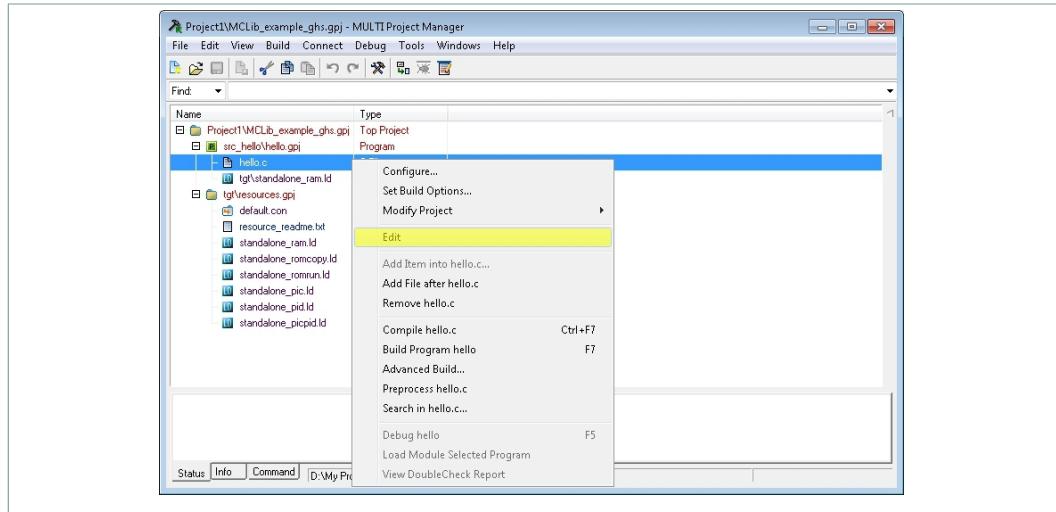


Figure 13. Opening the C editor for editing the application source code

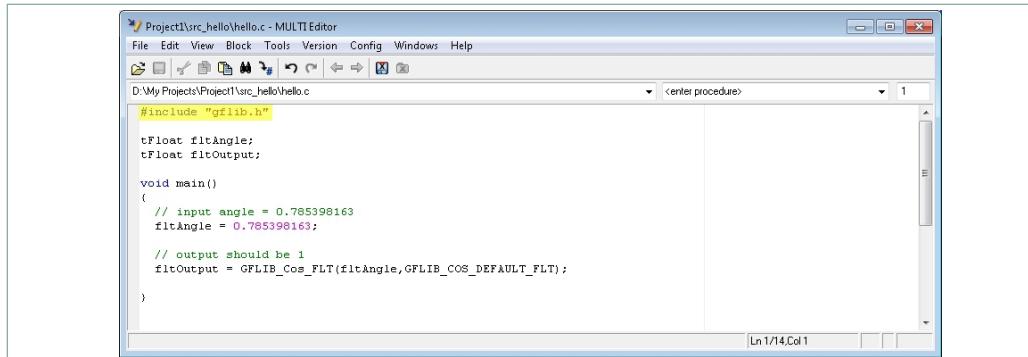


Figure 14. Using the pre-processor directive to include library master header files

At this point, the Automotive Math and Motor Control Library Set for NXP S32K11x 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 S32 Design Studio IDE for Arm based MCUs

In order to use the Automotive Math and Motor Control Library Set for NXP S32K11x 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 S32K11x devices files. The following files shall be added to the user project:

- Library binary files located in the directory "C:\NXP\AMMCLIB\S32K11x_AMMCLIB_v1.1.20\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\S32K11x_AMMCLIB_v1.1.20\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 15](#).

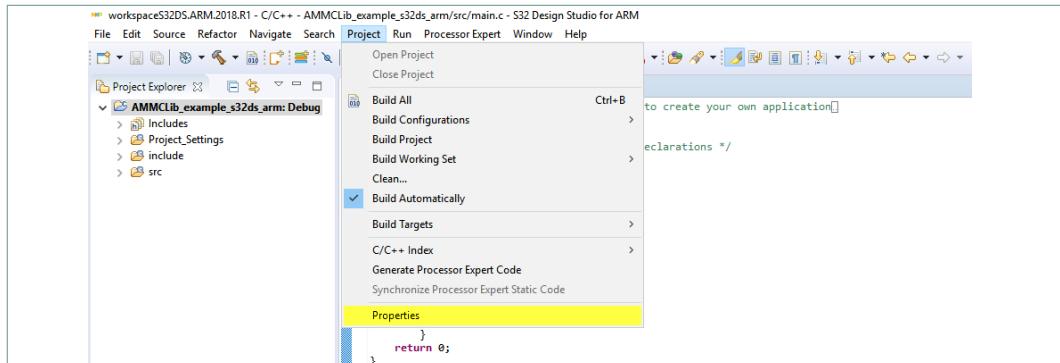


Figure 15. Selection of the project property in the S32 Design Studio IDE for Arm based MCUs

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 16](#).

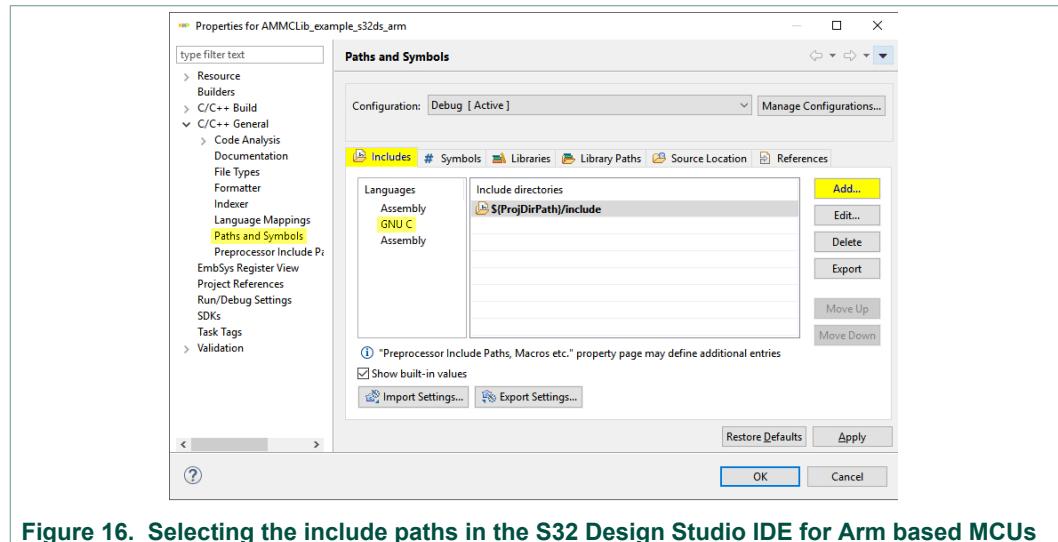


Figure 16. Selecting the include paths in the S32 Design Studio IDE for Arm based MCUs

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\S32K11x_AMMCLIB_v1.1.20\include" shall be added.

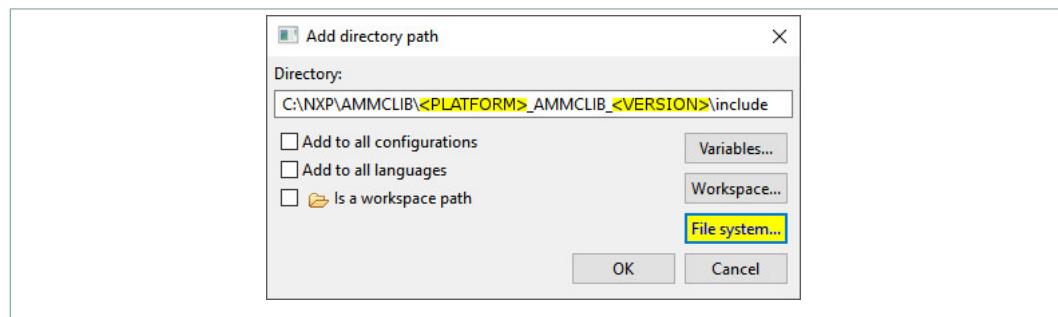


Figure 17. Adding the header files path to the S32 Design Studio IDE for Arm based MCUs project

After adding of the Automotive Math and Motor Control Library Set for NXP S32K11x 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 ":S32K11x_AMMCLIB.a", as described in [Figure 18](#). Be aware, that colon must be added as a prefix of the library name.

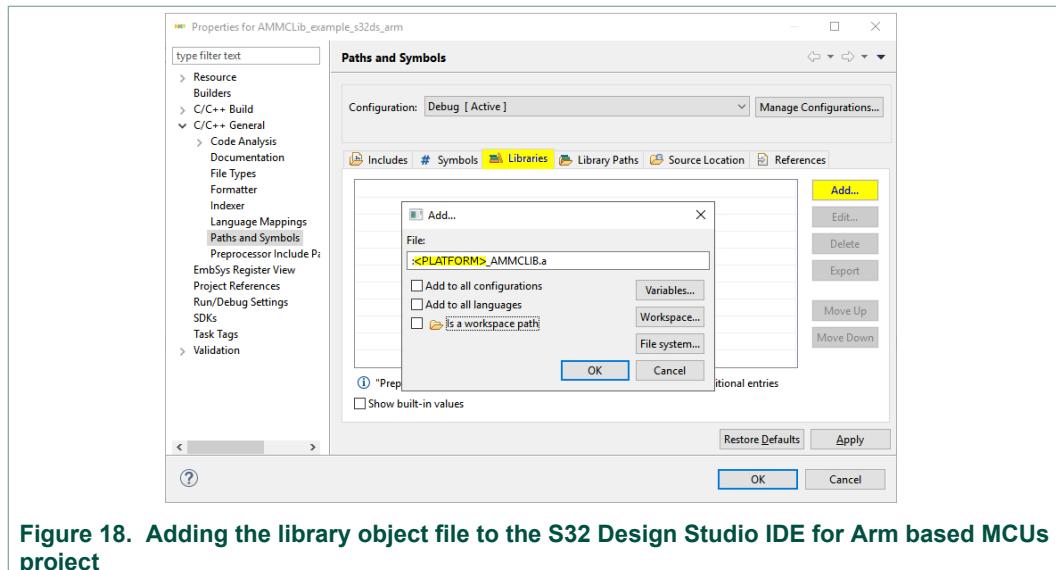


Figure 18. Adding the library object file to the S32 Design Studio IDE for Arm based MCUs project

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\S32K11x_AMMCLIB_v1.1.20\lib\s32ds_arm32" shall be added.

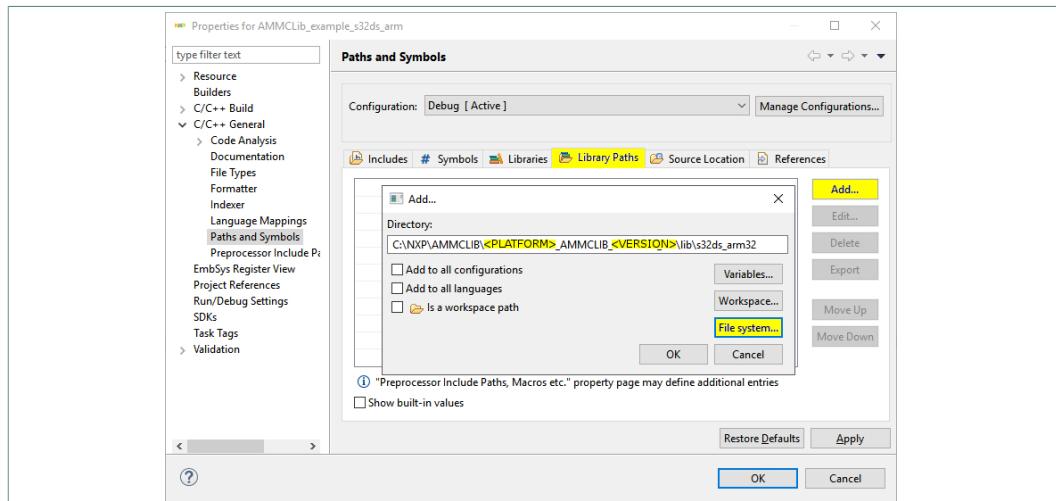


Figure 19. Adding the library object file path to the S32 Design Studio IDE for Arm based MCUs project

Note: In S32 Design Studio IDE for Arm based MCUs it is possible to include Automotive Math and Motor Control Library Set for NXP S32K11x 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 S32K11x 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 S32K11x 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 S32K11x 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 Testing

In order to validate the implementation of the Automotive Math and Motor Control Library Set for NXP S32K11x 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 S32K11x 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 20](#) shows the testing principle:

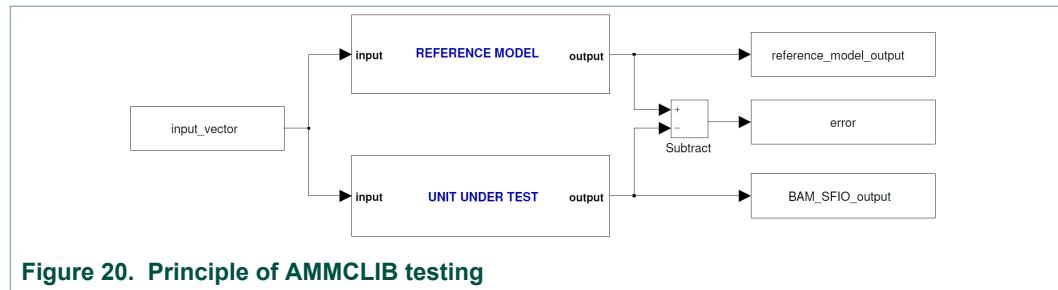
- **Input vector** represents the test vector which enters simultaneously into the Reference Model and into the Unit Under Test (UUT).
- **Reference Model (RM)** implements the real model of the UUT. For simple functions, the models are a part of the Matlab Simulink Toolbox. Advanced functions such as 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 1.12](#) for specific values).

**Figure 20. Principle of AMMCLIB testing**

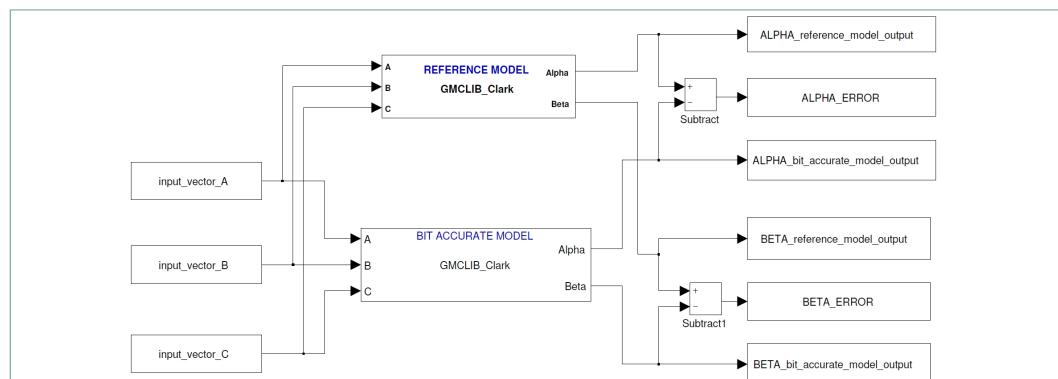
In order to test the UUT under all conditions, three types of test vector sets are used:

- 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 21](#)). 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 22](#). 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 23](#).

**Figure 21. Testing of the GMCLIB_Clark function based on the Matlab Simulink Toolbox**

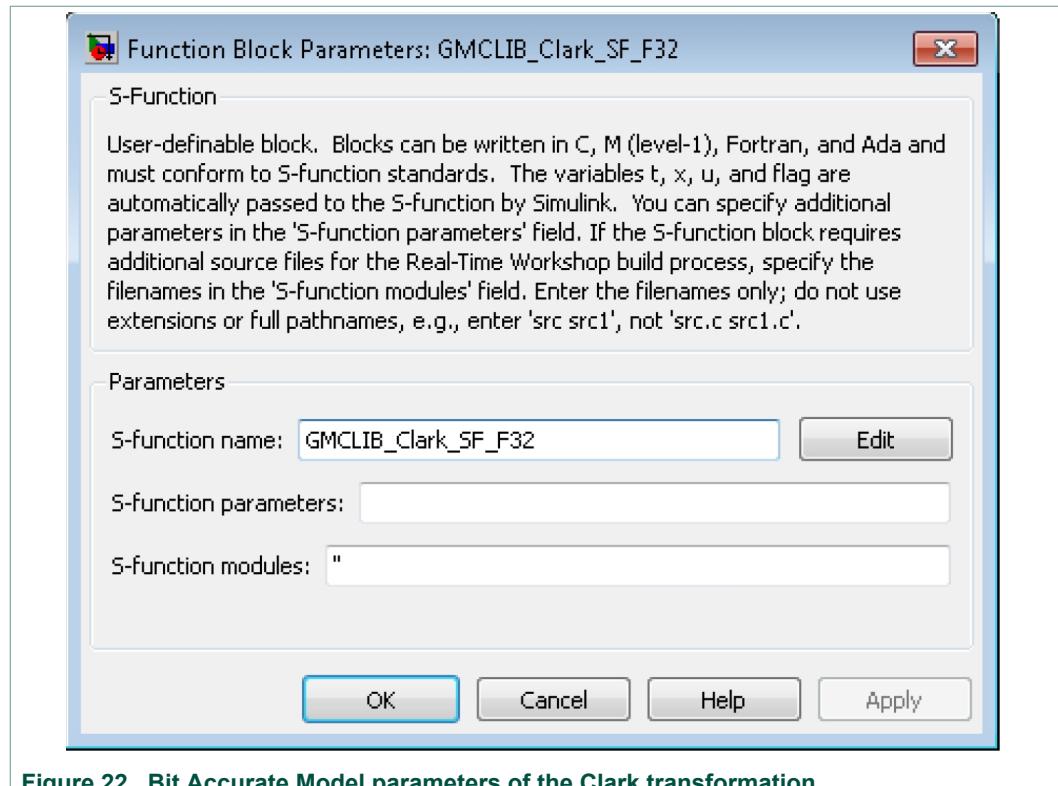


Figure 22. Bit Accurate Model parameters of the Clark transformation

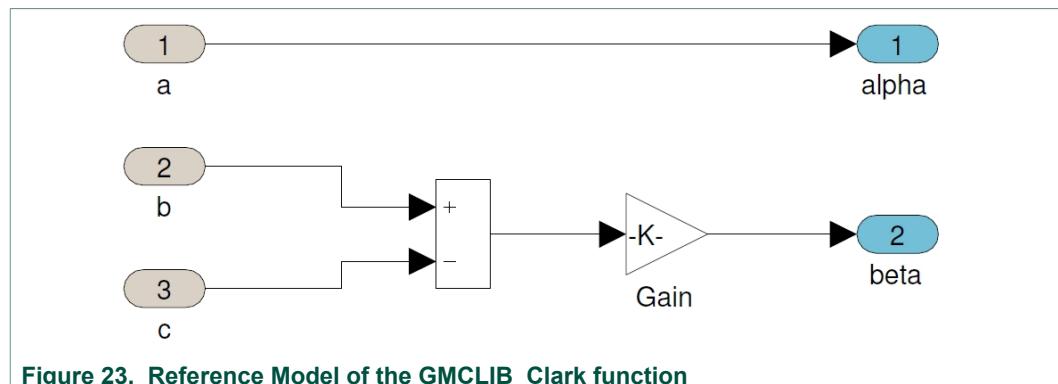


Figure 23. Reference Model of the GMCLIB_Clark function

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

The testing method in [Figure 24](#) 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 S32K11x 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.

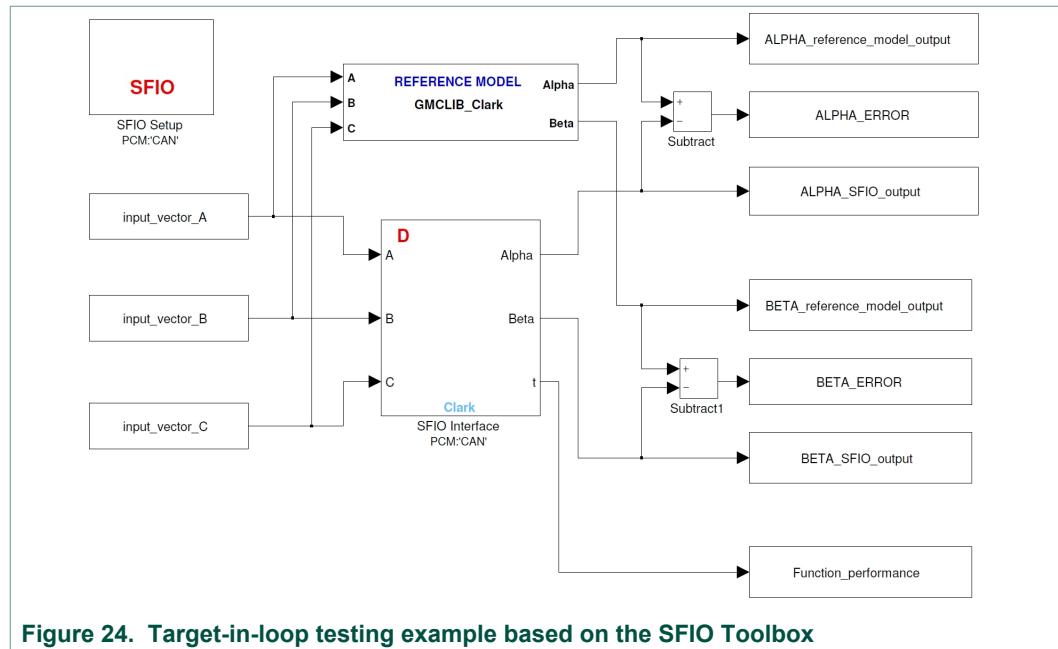


Figure 24. Target-in-loop testing example based on the SFIO Toolbox

1.12 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 1.11](#) 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.

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
GFLIB_ControllerPlrAW	3 LSB16	3 LSB16
GFLIB_Cos	3 LSB16	3 LSB16
GFLIB_Hyst	1 LSB16	1 LSB16

Function name	F32 variant	F16 variant
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_F32F16	1 LSB16	N/A
MLIB_ConvertPU_F16F32	N/A	1 LSB16
MLIB_ConvertPU_F32F16	1 LSB16	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
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

Function name	F32 variant	F16 variant
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 S32K11x devices.

2.1 Function index

Table 2. Quick function reference

Type	Name	Arguments
void	AMCLIB_BemfObsrvDQInit_F16	AMCLIB_BEMF_OBSRV_DQ_T_F16 *const pCtrl
void	AMCLIB_BemfObsrvDQInit_F32	AMCLIB_BEMF_OBSRV_DQ_T_F32 *const pCtrl
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
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
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
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
void	AMCLIB_CurrentLoopInit_F16	AMCLIB_CURRENT_LOOP_T_F16 *const pCtrl

Type	Name	Arguments
void	AMCLIB_CurrentLoopInit_F32	AMCLIB_CURRENT_LOOP_T_F32 *const pCtrl
void	AMCLIB_CurrentLoopSetState_F16	tFrac16 f16ControllerPIrAWDOut tFrac16 f16ControllerPIrAWQOut AMCLIB_CURRENT_LOOP_T_F16 * pCtrl
void	AMCLIB_CurrentLoopSetState_F32	tFrac32 f32ControllerPIrAWDOut tFrac32 f32ControllerPIrAWQOut AMCLIB_CURRENT_LOOP_T_F32 * pCtrl
void	AMCLIB_CurrentLoop_F16	tFrac16 f16UDcBus SWLIBS_2Syst_F16 *const pUDQReq AMCLIB_CURRENT_LOOP_T_F16 * pCtrl
void	AMCLIB_CurrentLoop_F32	tFrac32 f32UDcBus SWLIBS_2Syst_F32 *const pUDQReq AMCLIB_CURRENT_LOOP_T_F32 * pCtrl
void	AMCLIB_FWDebug_F16	tFrac16 f16IDQReqAmp tFrac16 f16VelocityFbck SWLIBS_2Syst_F16 *const pIDQReq AMCLIB_FW_DEBUG_T_F16 * pCtrl
void	AMCLIB_FWDebug_F32	tFrac32 f32IDQReqAmp tFrac32 f32VelocityFbck SWLIBS_2Syst_F32 *const pIDQReq AMCLIB_FW_DEBUG_T_F32 * pCtrl
void	AMCLIB_FWInit_F16	AMCLIB_FW_T_F16 *const pCtrl
void	AMCLIB_FWInit_F32	AMCLIB_FW_T_F32 *const pCtrl
void	AMCLIB_FWSetState_F16	tFrac16 f16FilterMAFWOut tFrac16 f16ControllerPIpAWFWOut AMCLIB_FW_T_F16 * pCtrl
void	AMCLIB_FWSetState_F32	tFrac32 f32FilterMAFWOut tFrac32 f32ControllerPIpAWFWOut AMCLIB_FW_T_F32 * pCtrl
void	AMCLIB_FWSpeedLoopDebug_F16	tFrac16 f16VelocityReq tFrac16 f16VelocityFbck SWLIBS_2Syst_F16 *const pIDQReq AMCLIB_FW_SPEED_LOOP_DEBUG_T_F16 * pCtrl
void	AMCLIB_FWSpeedLoopDebug_F32	tFrac32 f32VelocityReq tFrac32 f32VelocityFbck SWLIBS_2Syst_F32 *const pIDQReq AMCLIB_FW_SPEED_LOOP_DEBUG_T_F32 * pCtrl
void	AMCLIB_FWSpeedLoopInit_F16	AMCLIB_FW_SPEED_LOOP_T_F16 *const pCtrl
void	AMCLIB_FWSpeedLoopInit_F32	AMCLIB_FW_SPEED_LOOP_T_F32 *const pCtrl
void	AMCLIB_FWSpeedLoopSetState_F16	tFrac16 f16FilterMAWOut tFrac16 f16FilterMAFWOut tFrac16 f16ControllerPIpAWQOut tFrac16 f16ControllerPIpAWFWOut tFrac32 f32RampOut AMCLIB_FW_SPEED_LOOP_T_F16 * pCtrl

Type	Name	Arguments
void	AMCLIB_FWSpeedLoopSetState_F32	tFrac32 f32FilterMAWOut tFrac32 f32FilterMAFWOut tFrac32 f32ControllerPipAWQOut tFrac32 f32ControllerPipAWFWOut tFrac32 f32RampOut AMCLIB_FW_SPEED_LOOP_T_F32 * pCtrl
void	AMCLIB_FWSpeedLoop_F16	tFrac16 f16VelocityReq tFrac16 f16VelocityFbck SWLIBS_2Syst_F16 *const pIDQReq AMCLIB_FW_SPEED_LOOP_T_F16 * pCtrl
void	AMCLIB_FWSpeedLoop_F32	tFrac32 f32VelocityReq tFrac32 f32VelocityFbck SWLIBS_2Syst_F32 *const pIDQReq AMCLIB_FW_SPEED_LOOP_T_F32 * pCtrl
void	AMCLIB_FW_F16	tFrac16 f16IDQReqAmp tFrac16 f16VelocityFbck SWLIBS_2Syst_F16 *const pIDQReq AMCLIB_FW_T_F16 * pCtrl
void	AMCLIB_FW_F32	tFrac32 f32IDQReqAmp tFrac32 f32VelocityFbck SWLIBS_2Syst_F32 *const pIDQReq AMCLIB_FW_T_F32 * pCtrl
void	AMCLIB_SpeedLoopDebug_F16	tFrac16 f16VelocityReq tFrac16 f16VelocityFbck SWLIBS_2Syst_F16 *const pIDQReq AMCLIB_SPEED_LOOP_DEBUG_T_F16 * pCtrl
void	AMCLIB_SpeedLoopDebug_F32	tFrac32 f32VelocityReq tFrac32 f32VelocityFbck SWLIBS_2Syst_F32 *const pIDQReq AMCLIB_SPEED_LOOP_DEBUG_T_F32 * pCtrl
void	AMCLIB_SpeedLoopInit_F16	AMCLIB_SPEED_LOOP_T_F16 *const pCtrl
void	AMCLIB_SpeedLoopInit_F32	AMCLIB_SPEED_LOOP_T_F32 *const pCtrl
void	AMCLIB_SpeedLoopSetState_F16	tFrac16 f16FilterMAWOut tFrac16 f16ControllerPipAWQOut tFrac32 f32RampOut AMCLIB_SPEED_LOOP_T_F16 * pCtrl
void	AMCLIB_SpeedLoopSetState_F32	tFrac32 f32FilterMAWOut tFrac32 f32ControllerPipAWQOut tFrac32 f32RampOut AMCLIB_SPEED_LOOP_T_F32 * pCtrl
void	AMCLIB_SpeedLoop_F16	tFrac16 f16VelocityReq tFrac16 f16VelocityFbck SWLIBS_2Syst_F16 *const pIDQReq AMCLIB_SPEED_LOOP_T_F16 * pCtrl

Type	Name	Arguments
void	AMCLIB_SpeedLoop_F32	<code>tFrac32</code> f32VelocityReq <code>tFrac32</code> f32VelocityFbck SWLIBS_2Syst_F32 *const pIDQReq AMCLIB_SPEED_LOOP_T_F32 * pCtrl
void	AMCLIB_TrackObsrvInit_F16	AMCLIB_TRACK_OBSRV_T_F16 * pCtrl
void	AMCLIB_TrackObsrvInit_F32	AMCLIB_TRACK_OBSRV_T_F32 * pCtrl
void	AMCLIB_TrackObsrvSetState_F16	<code>tFrac16</code> f16PosOut <code>tFrac16</code> f16VelocityOut AMCLIB_TRACK_OBSRV_T_F16 * pCtrl
void	AMCLIB_TrackObsrvSetState_F32	<code>tFrac32</code> f32PosOut <code>tFrac32</code> f32VelocityOut AMCLIB_TRACK_OBSRV_T_F32 * pCtrl
void	AMCLIB_TrackObsrv_F16	<code>tFrac16</code> f16PhaseErr <code>tFrac16</code> * pPosEst <code>tFrac16</code> * pVelocityEst AMCLIB_TRACK_OBSRV_T_F16 * pCtrl
void	AMCLIB_TrackObsrv_F32	<code>tFrac32</code> f32PhaseErr <code>tFrac32</code> * pPosEst <code>tFrac32</code> * pVelocityEst AMCLIB_TRACK_OBSRV_T_F32 * pCtrl
void	AMCLIB_WindmillingInit_F16	<code>tFrac16</code> f16ADCMaxError AMCLIB_WINDMILLING_T_F16 *const pCtrl
void	AMCLIB_WindmillingInit_F32	<code>tFrac32</code> f32ADCMaxError AMCLIB_WINDMILLING_T_F32 *const pCtrl
AMCLIB_WINDMILLING_RET_T	AMCLIB_Windmilling_F16	const SWLIBS_3Syst_F16 * pUabcln <code>tFrac16</code> * pPosEst <code>tFrac16</code> * pVelocityEst AMCLIB_WINDMILLING_T_F16 *const pCtrl
AMCLIB_WINDMILLING_RET_T	AMCLIB_Windmilling_F32	const SWLIBS_3Syst_F32 * pUabcln <code>tFrac32</code> * pPosEst <code>tFrac32</code> * pVelocityEst AMCLIB_WINDMILLING_T_F32 *const pCtrl
void	GDFLIB_FilterFIRInit_F16	const GDFLIB_FILTERFIR_PARAM_T_F16 *const pParam GDFLIB_FILTERFIR_STATE_T_F16 *const pState <code>tFrac16</code> * pInBuf
void	GDFLIB_FilterFIRInit_F32	const GDFLIB_FILTERFIR_PARAM_T_F32 *const pParam GDFLIB_FILTERFIR_STATE_T_F32 *const pState <code>tFrac32</code> * pInBuf
<code>tFrac16</code>	GDFLIB_FilterFIR_F16	<code>tFrac16</code> f16In const GDFLIB_FILTERFIR_PARAM_T_F16 *const pParam GDFLIB_FILTERFIR_STATE_T_F16 *const pState

Type	Name	Arguments
tFrac32	GDFLIB_FilterFIR_F32	tFrac32 f32In const GDFLIB_FILTERFIR_PARAM_T_F32 *const pParam GDFLIB_FILTERFIR_STATE_T_F32 *const pState
void	GDFLIB_FilterIIR1Init_F16	GDFLIB_FILTER_IIR1_T_F16 *const pParam
void	GDFLIB_FilterIIR1Init_F32	GDFLIB_FILTER_IIR1_T_F32 *const pParam
tFrac16	GDFLIB_FilterIIR1_F16	tFrac16 f16In GDFLIB_FILTER_IIR1_T_F16 *const pParam
tFrac32	GDFLIB_FilterIIR1_F32	tFrac32 f32In GDFLIB_FILTER_IIR1_T_F32 *const pParam
void	GDFLIB_FilterIIR2Init_F16	GDFLIB_FILTER_IIR2_T_F16 *const pParam
void	GDFLIB_FilterIIR2Init_F32	GDFLIB_FILTER_IIR2_T_F32 *const pParam
tFrac16	GDFLIB_FilterIIR2_F16	tFrac16 f16In GDFLIB_FILTER_IIR2_T_F16 *const pParam
tFrac32	GDFLIB_FilterIIR2_F32	tFrac32 f32In GDFLIB_FILTER_IIR2_T_F32 *const pParam
void	GDFLIB_FilterMAInit_F16	GDFLIB_FILTER_MA_T_F16 * pParam
void	GDFLIB_FilterMAInit_F32	GDFLIB_FILTER_MA_T_F32 * pParam
void	GDFLIB_FilterMASetState_F16	tFrac16 f16FilterMAOut GDFLIB_FILTER_MA_T_F16 * pParam
void	GDFLIB_FilterMASetState_F32	tFrac32 f32FilterMAOut GDFLIB_FILTER_MA_T_F32 * pParam
tFrac16	GDFLIB_FilterMA_F16	tFrac16 f16In GDFLIB_FILTER_MA_T_F16 * pParam
tFrac32	GDFLIB_FilterMA_F32	tFrac32 f32In GDFLIB_FILTER_MA_T_F32 * pParam
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
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
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
tFrac16	GFLIB_AtanYX_F16	tFrac16 f16InY tFrac16 f16InX

Type	Name	Arguments
tFrac32	GFLIB_AtanYX_F32	tFrac32 f32InY tFrac32 f32InX
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
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_ControllerPIDpAWSetState_F16	tFrac16 f16ControllerPIDpAWOut GFLIB_CONTROLLER_PID_P_AW_T_F16 *const pParam
void	GFLIB_ControllerPIDpAWSetState_F32	tFrac32 f32ControllerPIDpAWOut GFLIB_CONTROLLER_PID_P_AW_T_F32 *const pParam
tFrac16	GFLIB_ControllerPIDpAW_F16	tFrac16 f16InErr GFLIB_CONTROLLER_PID_P_AW_T_F16 *const pParam
tFrac32	GFLIB_ControllerPIDpAW_F32	tFrac32 f32InErr GFLIB_CONTROLLER_PID_P_AW_T_F32 *const pParam
void	GFLIB_ControllerPipAWInit_F16	GFLIB_CONTROLLER_PIAW_P_T_F16 *const pParam
void	GFLIB_ControllerPipAWInit_F32	GFLIB_CONTROLLER_PIAW_P_T_F32 *const pParam
void	GFLIB_ControllerPipAWSetState_F16	tFrac16 f16ControllerPipAWOut GFLIB_CONTROLLER_PIAW_P_T_F16 *const pParam
void	GFLIB_ControllerPipAWSetState_F32	tFrac32 f32ControllerPipAWOut GFLIB_CONTROLLER_PIAW_P_T_F32 *const pParam
tFrac16	GFLIB_ControllerPipAW_F16	tFrac16 f16InErr GFLIB_CONTROLLER_PIAW_P_T_F16 *const pParam
tFrac32	GFLIB_ControllerPipAW_F32	tFrac32 f32InErr GFLIB_CONTROLLER_PIAW_P_T_F32 *const pParam
void	GFLIB_ControllerPiplInit_F16	GFLIB_CONTROLLER_PI_P_T_F16 *const pParam
void	GFLIB_ControllerPiplInit_F32	GFLIB_CONTROLLER_PI_P_T_F32 *const pParam
void	GFLIB_ControllerPiplSetState_F16	tFrac16 f16ControllerPiplOut GFLIB_CONTROLLER_PI_P_T_F16 *const pParam
void	GFLIB_ControllerPiplSetState_F32	tFrac32 f32ControllerPiplOut GFLIB_CONTROLLER_PI_P_T_F32 *const pParam

Type	Name	Arguments
tFrac16	GFLIB_ControllerPIp_F16	tFrac16 f16InErr GFLIB_CONTROLLER_PI_P_T_F16 *const pParam
tFrac32	GFLIB_ControllerPIp_F32	tFrac32 f32InErr GFLIB_CONTROLLER_PI_P_T_F32 *const pParam
void	GFLIB_ControllerPIrAWInit_F16	GFLIB_CONTROLLER_PIAW_R_T_F16 *const pParam
void	GFLIB_ControllerPIrAWInit_F32	GFLIB_CONTROLLER_PIAW_R_T_F32 *const pParam
void	GFLIB_ControllerPIrAWSetState_F16	tFrac16 f16ControllerPIrAWOut GFLIB_CONTROLLER_PIAW_R_T_F16 *const pParam
void	GFLIB_ControllerPIrAWSetState_F32	tFrac32 f32ControllerPIrAWOut GFLIB_CONTROLLER_PIAW_R_T_F32 *const pParam
tFrac16	GFLIB_ControllerPIrAW_F16	tFrac16 f16InErr GFLIB_CONTROLLER_PIAW_R_T_F16 *const pParam
tFrac32	GFLIB_ControllerPIrAW_F32	tFrac32 f32InErr GFLIB_CONTROLLER_PIAW_R_T_F32 *const pParam
void	GFLIB_ControllerPIrInit_F16	GFLIB_CONTROLLER_PI_R_T_F16 *const pParam
void	GFLIB_ControllerPIrInit_F32	GFLIB_CONTROLLER_PI_R_T_F32 *const pParam
void	GFLIB_ControllerPIrSetState_F16	tFrac16 f16ControllerPIrOut GFLIB_CONTROLLER_PI_R_T_F16 *const pParam
void	GFLIB_ControllerPIrSetState_F32	tFrac32 f32ControllerPIrOut GFLIB_CONTROLLER_PI_R_T_F32 *const pParam
tFrac16	GFLIB_ControllerPIr_F16	tFrac16 f16InErr GFLIB_CONTROLLER_PI_R_T_F16 *const pParam
tFrac32	GFLIB_ControllerPIr_F32	tFrac32 f32InErr GFLIB_CONTROLLER_PI_R_T_F32 *const pParam
tFrac16	GFLIB_Cos_F16	tFrac16 f16In const GFLIB_COS_T_F16 *const pParam
tFrac32	GFLIB_Cos_F32	tFrac32 f32In const GFLIB_COS_T_F32 *const pParam
tFrac16	GFLIB_Hyst_F16	tFrac16 f16In GFLIB_HYST_T_F16 *const pParam
tFrac32	GFLIB_Hyst_F32	tFrac32 f32In GFLIB_HYST_T_F32 *const pParam
void	GFLIB_IntegratorTRSetState_F16	tFrac16 f16IntegratorTROut GFLIB_INTEGRATOR_TR_T_F16 *const pParam
void	GFLIB_IntegratorTRSetState_F32	tFrac32 f32IntegratorTROut GFLIB_INTEGRATOR_TR_T_F32 *const pParam
tFrac16	GFLIB_IntegratorTR_F16	tFrac16 f16In GFLIB_INTEGRATOR_TR_T_F16 *const pParam

Type	Name	Arguments
tFrac32	GFLIB_IntegratorTR_F32	tFrac32 f32In GFLIB_INTEGRATOR_TR_T_F32 *const pParam
tFrac16	GFLIB_Limit_F16	tFrac16 f16In const GFLIB_LIMIT_T_F16 *const pParam
tFrac32	GFLIB_Limit_F32	tFrac32 f32In const GFLIB_LIMIT_T_F32 *const pParam
tFrac16	GFLIB_LowerLimit_F16	tFrac16 f16In const GFLIB_LOWERLIMIT_T_F16 *const pParam
tFrac32	GFLIB_LowerLimit_F32	tFrac32 f32In const GFLIB_LOWERLIMIT_T_F32 *const pParam
tFrac16	GFLIB_Lut1D_F16	tFrac16 f16In const GFLIB_LUT1D_T_F16 *const pParam
tFrac32	GFLIB_Lut1D_F32	tFrac32 f32In const GFLIB_LUT1D_T_F32 *const pParam
tFrac16	GFLIB_Lut2D_F16	tFrac16 f16In1 tFrac16 f16In2 const GFLIB_LUT2D_T_F16 *const pParam
tFrac32	GFLIB_Lut2D_F32	tFrac32 f32In1 tFrac32 f32In2 const GFLIB_LUT2D_T_F32 *const pParam
tFrac16	GFLIB_Ramp_F16	tFrac16 f16In GFLIB_RAMP_T_F16 *const pParam
tFrac32	GFLIB_Ramp_F32	tFrac32 f32In GFLIB_RAMP_T_F32 *const pParam
tFrac16	GFLIB_Sign_F16	tFrac16 f16In
tFrac32	GFLIB_Sign_F32	tFrac32 f32In
void	GFLIB_SinCos_F16	tFrac16 f16In SWLIBS_2Syst_F16 * pOut const GFLIB_SINCOS_T_F16 *const pParam
void	GFLIB_SinCos_F32	tFrac32 f32In SWLIBS_2Syst_F32 * pOut const GFLIB_SINCOS_T_F32 *const pParam
tFrac16	GFLIB_Sin_F16	tFrac16 f16In const GFLIB_SIN_T_F16 *const pParam
tFrac32	GFLIB_Sin_F32	tFrac32 f32In const GFLIB_SIN_T_F32 *const pParam
tFrac16	GFLIB_Sqrt_F16	tFrac16 f16In
tFrac32	GFLIB_Sqrt_F32	tFrac32 f32In
tFrac16	GFLIB_Tan_F16	tFrac16 f16In const GFLIB_TAN_T_F16 *const pParam
tFrac32	GFLIB_Tan_F32	tFrac32 f32In const GFLIB_TAN_T_F32 *const pParam

Type	Name	Arguments
tFrac16	GFLIB_UpperLimit_F16	const tFrac16 f16In const GFLIB_UPPERLIMIT_T_F16 *const pParam
tFrac32	GFLIB_UpperLimit_F32	const tFrac32 f32In const GFLIB_UPPERLIMIT_T_F32 *const pParam
INLINE tU16	GFLIB_VMin10_F16	const tFrac16 * pln
INLINE tU16	GFLIB_VMin11_F16	const tFrac16 * pln
INLINE tU16	GFLIB_VMin12_F16	const tFrac16 * pln
INLINE tU16	GFLIB_VMin13_F16	const tFrac16 * pln
INLINE tU16	GFLIB_VMin14_F16	const tFrac16 * pln
INLINE tU16	GFLIB_VMin15_F16	const tFrac16 * pln
INLINE tU16	GFLIB_VMin16_F16	const tFrac16 * pln
INLINE tU16	GFLIB_VMin4_F16	const tFrac16 * pln
INLINE tU16	GFLIB_VMin5_F16	const tFrac16 * pln
INLINE tU16	GFLIB_VMin6_F16	const tFrac16 * pln
INLINE tU16	GFLIB_VMin7_F16	const tFrac16 * pln
INLINE tU16	GFLIB_VMin8_F16	const tFrac16 * pln
INLINE tU16	GFLIB_VMin9_F16	const tFrac16 * pln
tU16	GFLIB_VMin_F16	const tFrac16 * pln tU16 u16N
tU32	GFLIB_VMin_F32	const tFrac32 * pln tU32 u32N
tBool	GFLIB_VectorLimit_F16	const SWLIBS_2Syst_F16 *const pln SWLIBS_2Syst_F16 *const pOut const GFLIB_VECTORLIMIT_T_F16 *const pParam
tBool	GFLIB_VectorLimit_F32	const SWLIBS_2Syst_F32 *const pln SWLIBS_2Syst_F32 *const pOut const GFLIB_VECTORLIMIT_T_F32 *const pParam
void	GMCLIB_BetaProjection3Ph_F16	const SWLIBS_3Syst_F16 *const pln SWLIBS_3Syst_F16 *const pOut
void	GMCLIB_BetaProjection3Ph_F32	const SWLIBS_3Syst_F32 *const pln SWLIBS_3Syst_F32 *const pOut
void	GMCLIB_BetaProjection_F16	const SWLIBS_3Syst_F16 *const pln SWLIBS_2Syst_F16 *const pOut
void	GMCLIB_BetaProjection_F32	const SWLIBS_3Syst_F32 *const pln SWLIBS_2Syst_F32 *const pOut
void	GMCLIB_ClarkInv_F16	const SWLIBS_2Syst_F16 *const pln SWLIBS_3Syst_F16 *const pOut
void	GMCLIB_ClarkInv_F32	const SWLIBS_2Syst_F32 *const pln SWLIBS_3Syst_F32 *const pOut
void	GMCLIB_Clark_F16	const SWLIBS_3Syst_F16 *const pln SWLIBS_2Syst_F16 *const pOut

Type	Name	Arguments
void	GMCLIB_Clark_F32	const SWLIBS_3Syst_F32 *const pIn SWLIBS_2Syst_F32 *const pOut
void	GMCLIB_DecouplingPMSM_F16	SWLIBS_2Syst_F16 *const pUdqDec const SWLIBS_2Syst_F16 *const pUdq const SWLIBS_2Syst_F16 *const pldq tFrac16 f16AngularVel const GMCLIB_DECOUPLINGPMSM_T_F16 *const pParam
void	GMCLIB_DecouplingPMSM_F32	SWLIBS_2Syst_F32 *const pUdqDec const SWLIBS_2Syst_F32 *const pUdq const SWLIBS_2Syst_F32 *const pldq tFrac32 f32AngularVel const GMCLIB_DECOUPLINGPMSM_T_F32 *const pParam
void	GMCLIB_ElimDcBusRip_F16	SWLIBS_2Syst_F16 *const pOut const SWLIBS_2Syst_F16 *const pIn const GMCLIB_ELIMDCBUSRIP_T_F16 *const pParam
void	GMCLIB_ElimDcBusRip_F32	SWLIBS_2Syst_F32 *const pOut const SWLIBS_2Syst_F32 *const pIn const GMCLIB_ELIMDCBUSRIP_T_F32 *const pParam
void	GMCLIB_ParkInv_F16	SWLIBS_2Syst_F16 *const pOut const SWLIBS_2Syst_F16 *const pInAngle const SWLIBS_2Syst_F16 *const pIn
void	GMCLIB_ParkInv_F32	SWLIBS_2Syst_F32 *const pOut const SWLIBS_2Syst_F32 *const pInAngle const SWLIBS_2Syst_F32 *const pIn
void	GMCLIB_Park_F16	SWLIBS_2Syst_F16 *pOut const SWLIBS_2Syst_F16 *const pInAngle const SWLIBS_2Syst_F16 *const pIn
void	GMCLIB_Park_F32	SWLIBS_2Syst_F32 * pOut const SWLIBS_2Syst_F32 *const pInAngle const SWLIBS_2Syst_F32 *const pIn
tU16	GMCLIB_PwmIct_F16	SWLIBS_3Syst_F16 * pOut const SWLIBS_2Syst_F16 *const pIn
tU32	GMCLIB_PwmIct_F32	SWLIBS_3Syst_F32 * pOut const SWLIBS_2Syst_F32 *const pIn
tU16	GMCLIB_SvmSci_F16	SWLIBS_3Syst_F16 * pOut const SWLIBS_2Syst_F16 *const pIn
tU32	GMCLIB_SvmSci_F32	SWLIBS_3Syst_F32 * pOut const SWLIBS_2Syst_F32 *const pIn
tU16	GMCLIB_SvmStd_F16	SWLIBS_3Syst_F16 * pOut const SWLIBS_2Syst_F16 *const pIn
tU32	GMCLIB_SvmStd_F32	SWLIBS_3Syst_F32 * pOut const SWLIBS_2Syst_F32 *const pIn

Type	Name	Arguments
tU16	GMCLIB_SvmU0n_F16	SWLIBS_3Syst_F16 * pOut const SWLIBS_2Syst_F16 *const pln
tU32	GMCLIB_SvmU0n_F32	SWLIBS_3Syst_F32 * pOut const SWLIBS_2Syst_F32 *const pln
tU16	GMCLIB_SvmU7n_F16	SWLIBS_3Syst_F16 * pOut const SWLIBS_2Syst_F16 *const pln
tU32	GMCLIB_SvmU7n_F32	SWLIBS_3Syst_F32 * pOut const SWLIBS_2Syst_F32 *const pln
INLINE tFrac16	MLIB_AbsSat_F16	register tFrac16 f16In
INLINE tFrac32	MLIB_AbsSat_F32	register tFrac32 f32In
INLINE tFrac16	MLIB_Abs_F16	register tFrac16 f16In
INLINE tFrac32	MLIB_Abs_F32	register tFrac32 f32In
INLINE tFrac16	MLIB_AddSat_F16	register tFrac16 f16In1 register tFrac16 f16In2
INLINE tFrac32	MLIB_AddSat_F32	register tFrac32 f32In1 register tFrac32 f32In2
INLINE tFrac16	MLIB_Add_F16	register tFrac16 f16In1 register tFrac16 f16In2
INLINE tFrac32	MLIB_Add_F32	register tFrac32 f32In1 register tFrac32 f32In2
INLINE tFrac16	MLIB_ConvertPU_F16F32	register tFrac32 f32In
INLINE tFrac32	MLIB_ConvertPU_F32F16	register tFrac16 f16In
INLINE tFrac16	MLIB_Convert_F16F32	register tFrac32 f32In1 register tFrac32 f32In2
INLINE tFrac32	MLIB_Convert_F32F16	register tFrac16 f16In1 register tFrac16 f16In2
INLINE tFrac16	MLIB_DivSat_F16	register tFrac16 f16In1 register tFrac16 f16In2
INLINE tFrac32	MLIB_DivSat_F32	register tFrac32 f32In1 register tFrac32 f32In2
INLINE tFrac16	MLIB_Div_F16	register tFrac16 f16In1 register tFrac16 f16In2
INLINE tFrac32	MLIB_Div_F32	register tFrac32 f32In1 register tFrac32 f32In2
INLINE tFrac16	MLIB_MacSat_F16	register tFrac16 f16In1 register tFrac16 f16In2 register tFrac16 f16In3
INLINE tFrac32	MLIB_MacSat_F32	register tFrac32 f32In1 register tFrac32 f32In2 register tFrac32 f32In3

Type	Name	Arguments
INLINE tFrac32	MLIB_MacSat_F32F16F16	register tFrac32 f32In1 register tFrac16 f16In2 register tFrac16 f16In3
INLINE tFrac16	MLIB_Mac_F16	register tFrac16 f16In1 register tFrac16 f16In2 register tFrac16 f16In3
INLINE tFrac32	MLIB_Mac_F32	register tFrac32 f32In1 register tFrac32 f32In2 register tFrac32 f32In3
INLINE tFrac32	MLIB_Mac_F32F16F16	register tFrac32 f32In1 register tFrac16 f16In2 register tFrac16 f16In3
INLINE tFrac16	MLIB_Mnac_F16	register tFrac16 f16In1 register tFrac16 f16In2 register tFrac16 f16In3
INLINE tFrac32	MLIB_Mnac_F32	register tFrac32 f32In1 register tFrac32 f32In2 register tFrac32 f32In3
INLINE tFrac32	MLIB_Mnac_F32F16F16	register tFrac32 f32In1 register tFrac16 f16In2 register tFrac16 f16In3
INLINE tFrac16	MLIB_Msu_F16	register tFrac16 f16In1 register tFrac16 f16In2 register tFrac16 f16In3
INLINE tFrac32	MLIB_Msu_F32	register tFrac32 f32In1 register tFrac32 f32In2 register tFrac32 f32In3
INLINE tFrac32	MLIB_Msu_F32F16F16	register tFrac32 f32In1 register tFrac16 f16In2 register tFrac16 f16In3
INLINE tFrac16	MLIB_MulSat_F16	register tFrac16 f16In1 register tFrac16 f16In2
INLINE tFrac32	MLIB_MulSat_F32	register tFrac32 f32In1 register tFrac32 f32In2
INLINE tFrac32	MLIB_MulSat_F32F16F16	register tFrac16 f16In1 register tFrac16 f16In2
INLINE tFrac16	MLIB_Mul_F16	register tFrac16 f16In1 register tFrac16 f16In2
INLINE tFrac32	MLIB_Mul_F32	register tFrac32 f32In1 register tFrac32 f32In2
INLINE tFrac32	MLIB_Mul_F32F16F16	register tFrac16 f16In1 register tFrac16 f16In2
INLINE tFrac16	MLIB_NegSat_F16	register tFrac16 f16In1
INLINE tFrac32	MLIB_NegSat_F32	register tFrac32 f32In1

Type	Name	Arguments
INLINE tFrac16	MLIB_Neg_F16	register tFrac16 f16In
INLINE tFrac32	MLIB_Neg_F32	register tFrac32 f32In
INLINE tU16	MLIB_Norm_F16	register tFrac16 f16In
INLINE tU16	MLIB_Norm_F32	register tFrac32 f32In
INLINE tFrac16	MLIB_RndSat_F16F32	register tFrac32 f32In
INLINE tFrac16	MLIB_Round_F16	register tFrac16 f16In1 register tU16 u16In2
INLINE tFrac32	MLIB_Round_F32	register tFrac32 f32In1 register tU16 u16In2
INLINE tFrac16	MLIB_ShBiSat_F16	register tFrac16 f16In1 register tS16 s16In2
INLINE tFrac32	MLIB_ShBiSat_F32	register tFrac32 f32In1 register tS16 s16In2
INLINE tFrac16	MLIB_ShBi_F16	register tFrac16 f16In1 register tS16 s16In2
INLINE tFrac32	MLIB_ShBi_F32	register tFrac32 f32In1 register tS16 s16In2
INLINE tFrac16	MLIB_ShLSat_F16	register tFrac16 f16In1 register tU16 u16In2
INLINE tFrac32	MLIB_ShLSat_F32	register tFrac32 f32In1 register tU16 u16In2
INLINE tFrac16	MLIB_ShL_F16	register tFrac16 f16In1 register tU16 u16In2
INLINE tFrac32	MLIB_ShL_F32	register tFrac32 f32In1 register tU16 u16In2
INLINE tFrac16	MLIB_ShR_F16	register tFrac16 f16In1 register tU16 u16In2
INLINE tFrac32	MLIB_ShR_F32	register tFrac32 f32In1 register tU16 u16In2
INLINE tFrac16	MLIB_SubSat_F16	register tFrac16 f16In1 register tFrac16 f16In2
INLINE tFrac32	MLIB_SubSat_F32	register tFrac32 f32In1 register tFrac32 f32In2
INLINE tFrac16	MLIB_Sub_F16	register tFrac16 f16In1 register tFrac16 f16In2
INLINE tFrac32	MLIB_Sub_F32	register tFrac32 f32In1 register tFrac32 f32In2
INLINE tFrac16	MLIB_VMac_F16	register tFrac16 f16In1 register tFrac16 f16In2 register tFrac16 f16In3 register tFrac16 f16In4

Type	Name	Arguments
INLINE tFrac32	MLIB_VMac_F32	register tFrac32 f32In1 register tFrac32 f32In2 register tFrac32 f32In3 register tFrac32 f32In4
INLINE tFrac32	MLIB_VMac_F32F16F16	register tFrac16 f16In1 register tFrac16 f16In2 register tFrac16 f16In3 register tFrac16 f16In4
const SWLIBS_VERSION_T *	SWLIBS_GetVersion	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 α/β . The BEMF observer transforms these quantities into a quasi-synchronous reference frame γ/δ that follows the real synchronous rotor flux frame d/q with an error θ_{err} , see [Figure 25](#).

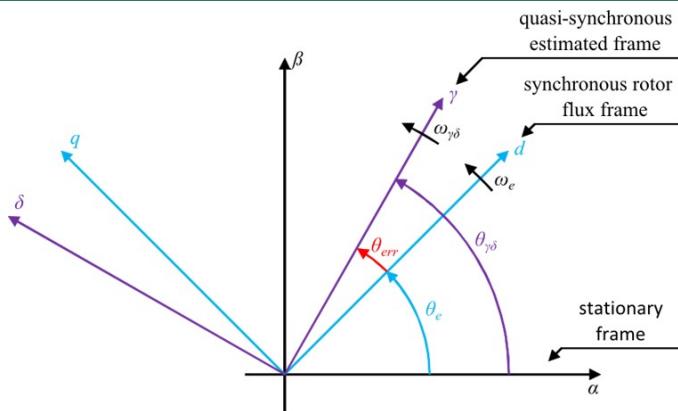


Figure 25. Rotor reference frames

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 [Ω],
- L_d and L_q are the d-axis and q-axis inductances [H],
- $\omega_{\gamma\delta}$ is the estimated electrical angular velocity of the rotor [rad/s],
- u_y and u_δ are the estimated stator voltages [V],
- i_y and i_δ are the estimated stator currents [A],
- E_{sal} is the saliency-based BEMF magnitude [V],
- θ_{err} is the phase error between the estimated quasi-synchronous frame γ/δ 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 δ 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 θ_{err} is extracted from the division E_y/E_δ . 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 θ_{err} between the quasi-synchronous reference frame γ/δ 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 θ_{err} , which is provided as an output of the BEMF observer.

The structure of the BEMF observer is depicted in [Figure 26](#).

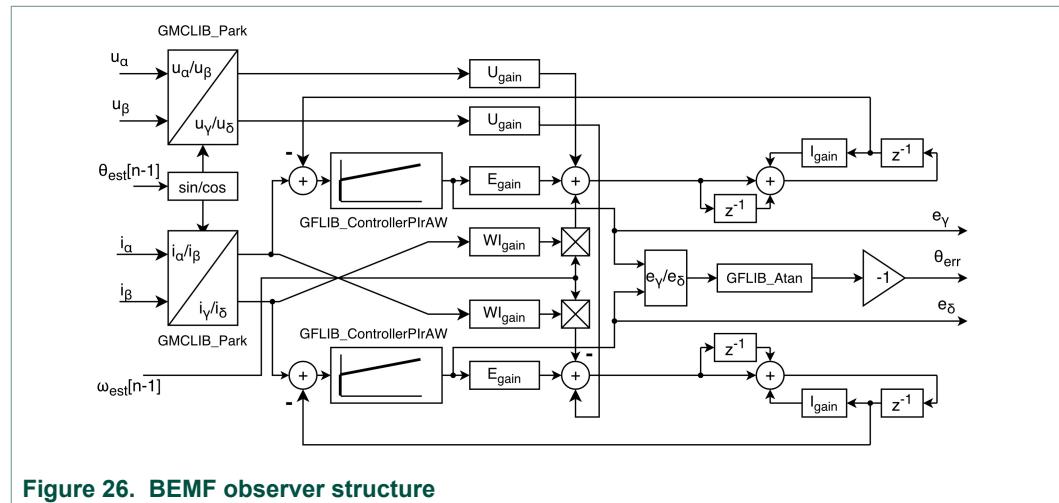


Figure 26. BEMF observer structure

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 (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]	Ω_{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 ξ is the current loop attenuation, and ω_0 is the current loop natural frequency [rad/s]. Coefficients ξ and ω_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) ';'])

```

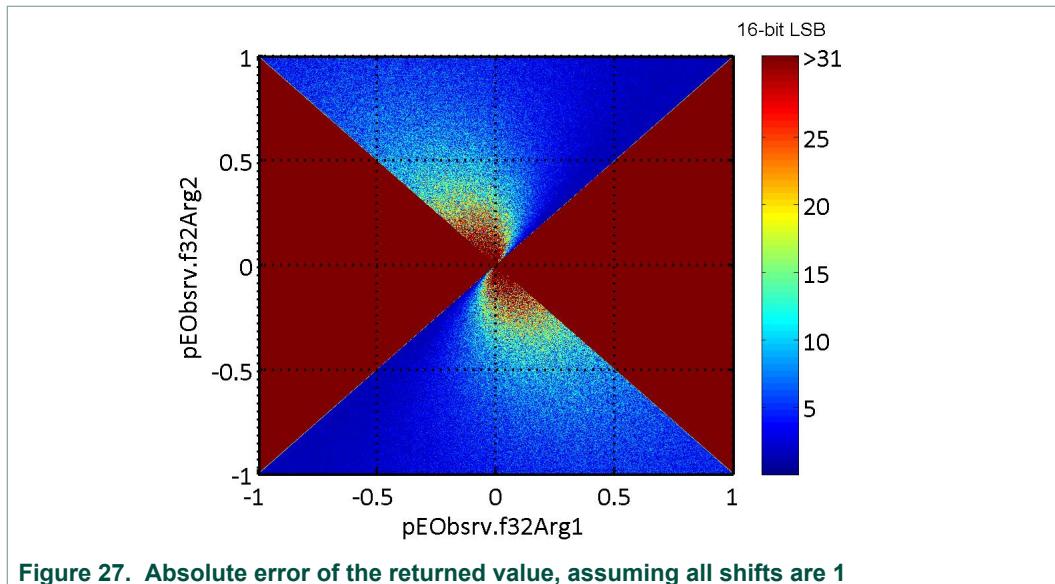
```

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 27. 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.

Code Example

```
#include "amclib.h"

SWLIBS_2Syst_F32          mcLab, 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_BemfObsrvDQ_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
// (...)

}

```

2.2.2 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]	Ω_{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 ξ is the current loop attenuation, and ω_0 is the current loop natural frequency [rad/s]. Coefficients ξ and ω_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[®] 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_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.

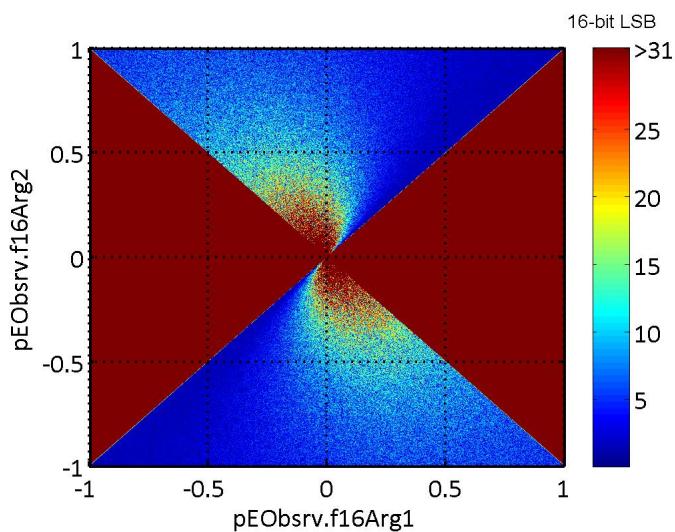


Figure 28. Absolute error of the returned value, assuming all shifts are 1

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.

Code Example

```
#include "amclib.h"

SWLIBS_2Syst_F16          mciLab, 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(&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_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 (HardFault).

Re-entrancy

The function is re-entrant for a different pCtrl.

2.2.3.1 Function AMCLIB_BemfObsrvDQInit_F32

Declaration

```
void AMCLIB_BemfObsrvDQInit_F32 (AMCLIB_BEMF_OBSRV_DQ_T_F32 *const pCtrl);
```

Arguments

Table 7. AMCLIB_BemfObsrvDQInit_F32 arguments

Type	Name	Direction	Description
AMCLIB_BEMF_OBSRV_DQ_T_F32 *const	pCtrl	input, output	Pointer to the structure with BEMF observer parameters and state.

Code Example

```
#include "amclib.h"

SWLIBS_2Syst_F32
AMCLIB_BEMF_OBSRV_DQ_T_F32
tFrac32
tFrac32

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_BemfObsrvDQ_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
// (...)

}
```

2.2.3.2 Function AMCLIB_BemfObsrvDQInit_F16

Declaration

```
void AMCLIB_BemfObsrvDQInit_F16(AMCLIB\_BEMF\_OBSRV\_DQ\_T\_F16 *const pCtrl);
```

Arguments

Table 8. AMCLIB_BemfObsrvDQInit_F16 arguments

Type	Name	Direction	Description
AMCLIB_BEMF_OBSRV_DQ_T_F16 *const	pCtrl	input, output	Pointer to the structure with BEMF observer parameters and state.

Code Example

```
#include "amclib.h"

SWLIBS\_2Syst\_F16 mciLab, 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(&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
```

```
// (...)  
}
```

2.2.4 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 α/β 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 α/β 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 (HardFault).

Re-entrancy

The function is re-entrant for a different `pCtrl`.

2.2.4.1 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 9. AMCLIB_BemfObsrvDQSetState_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.

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(&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(&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
    // (...)

}

```

2.2.4.2 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_DO_T_F16 *const pCtrl);
```

Arguments

Table 10. AMCLIB_BemfObsrvDQSetState_F16 arguments

Type	Name	Direction	Description
const <u>SWLIBS_2Syst_F16</u> *const	pIAB	input	Pointer to the structure with Alpha/Beta current components.
const <u>SWLIBS_2Syst_F16</u> *const	pUAB	input	Pointer to the structure with Alpha/Beta voltage components.
<u>tFrac16</u>	f16Velocity	input	Estimated electrical angular velocity.

Type	Name	Direction	Description
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.

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          mciLab, 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(&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(&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
    // (...)

}
```

2.3 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 29](#).

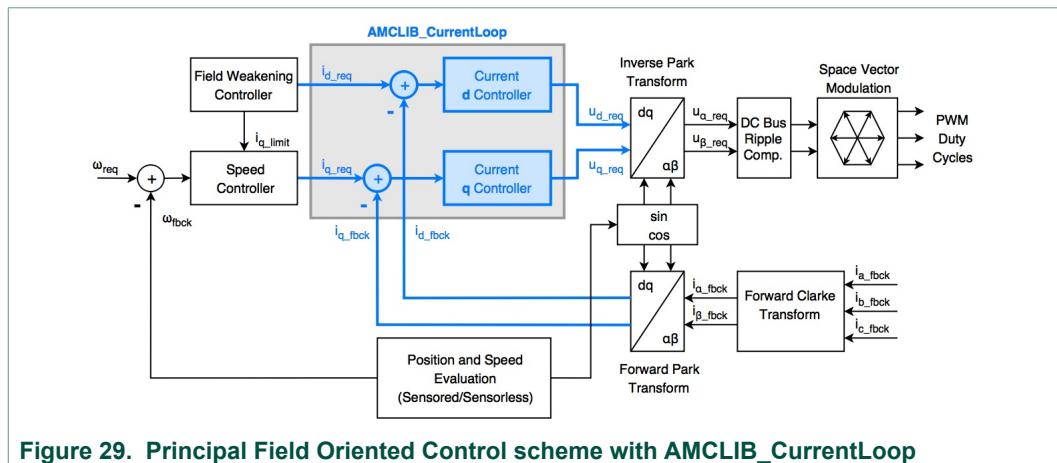


Figure 29. Principal Field Oriented Control scheme with AMCLIB_CurrentLoop

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 (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 11. 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.

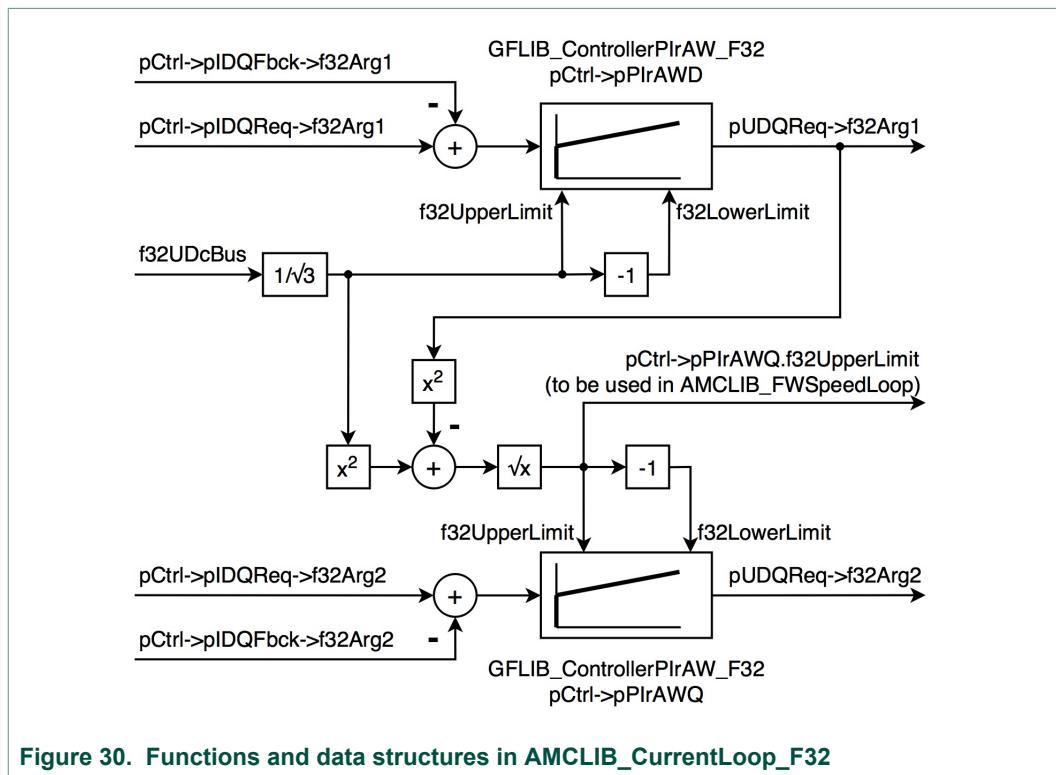


Figure 30. Functions and data structures in AMCLIB_CurrentLoop_F32

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 12. 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:

$$\begin{aligned}
 pPIrAWD.f32CC1sc &= FRAC32\left(\left(K_{pD} + \frac{K_{iD}T_S}{2}\right) \cdot \frac{I_{MAX}}{U_{MAX}} \cdot 2^{NShiftD}\right) \\
 pPIrAWD.f32CC2sc &= FRAC32\left(\left(-K_{pD} + \frac{K_{iD}T_S}{2}\right) \cdot \frac{I_{MAX}}{U_{MAX}} \cdot 2^{NShiftD}\right) \\
 pPIrAWD.u16NShift &= NShiftD \\
 pPIrAWQ.f32CC1sc &= FRAC32\left(\left(K_{pQ} + \frac{K_{iQ}T_S}{2}\right) \cdot \frac{I_{MAX}}{U_{MAX}} \cdot 2^{NShiftQ}\right) \\
 pPIrAWQ.f32CC2sc &= FRAC32\left(\left(-K_{pQ} + \frac{K_{iQ}T_S}{2}\right) \cdot \frac{I_{MAX}}{U_{MAX}} \cdot 2^{NShiftQ}\right) \\
 pPIrAWQ.u16NShift &= NShiftQ
 \end{aligned}$$

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 ξ is the current loop attenuation, and ω_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 $pCtrl->pPIrAWQ.f32UpperLimit > FRAC32(0.0000305176)$.

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.ul6NShift     = 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 13. AMCLIB_CurrentLoop_F16 arguments

Type	Name	Direction	Description
tFrac16	f16UDcBus	input	DC bus voltage.
SWLIBS_2Syst_F16 *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_F16 *	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_F16.

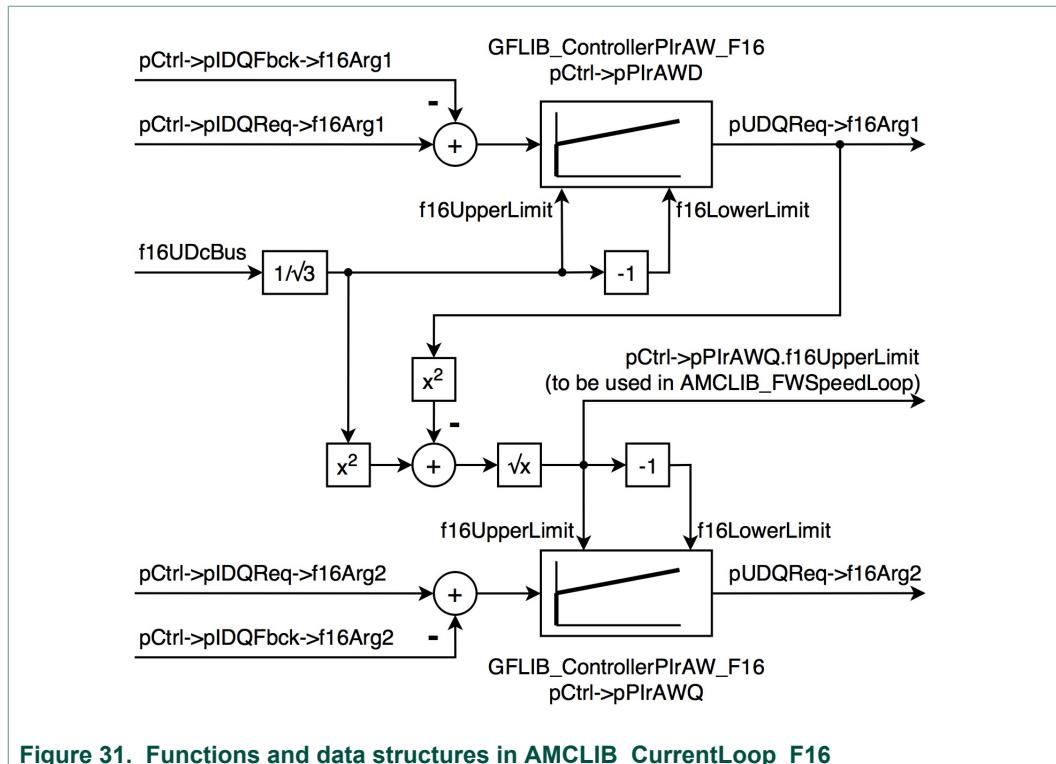


Figure 31. Functions and data structures in AMCLIB_CurrentLoop_F16

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 14. 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 ξ is the current loop attenuation, and ω_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.

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
    // (...)
```

2.3.3 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 (HardFault).

Re-entrancy

The function is re-entrant for a different pCtrl.

2.3.3.1 Function AMCLIB_CurrentLoopInit_F32

Declaration

```
void AMCLIB_CurrentLoopInit_F32 (AMCLIB\_CURRENT\_LOOP\_T\_F32 *const pCtrl);
```

Arguments

Table 15. 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.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
    // (...)

}
```

2.3.3.2 Function AMCLIB_CurrentLoopInit_F16

Declaration

```
void AMCLIB_CurrentLoopInit_F16 (AMCLIB_CURRENT_LOOP_T_F16 *const pCtrl);
```

Arguments

Table 16. AMCLIB_CurrentLoopInit_F16 arguments

Type	Name	Direction	Description
AMCLIB_CURRENT_LOOP_T_F16 *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.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
    // (...)
```

2.3.4 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 (HardFault).

Re-entrancy

The function is re-entrant for a different pCtrl.

2.3.4.1 Function AMCLIB_CurrentLoopSetState_F32**Declaration**

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

Arguments**Table 17. AMCLIB_CurrentLoopSetState_F32 arguments**

Type	Name	Direction	Description
tFrac32	f32ControllerPIrAWDOut <input/>		Required output of the d-axis ControllerPIrAW.
tFrac32	f32ControllerPIrAWQOut <input/>		Required output of the q-axis ControllerPIrAW.
AMCLIB_CURRENT_LOOP_T_F32 *	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
    // (...)

}

```

2.3.4.2 Function AMCLIB_CurrentLoopSetState_F16

Declaration

```

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

```

Arguments

Table 18. AMCLIB_CurrentLoopSetState_F16 arguments

Type	Name	Direction	Description
tFrac16	f16ControllerPIrAWDOut	input	Required output of the d-axis ControllerPIrAW.
tFrac16	f16ControllerPIrAWQOut	input	Required output of the q-axis ControllerPIrAW.
AMCLIB_CURRENT_LOOP_T_F16 *	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     = lu;
    CurrentLoop.pPIrAWQ.f16CC1sc      = (tFrac16) FRAC16(0.6432);
    CurrentLoop.pPIrAWQ.f16CC2sc      = (tFrac16) FRAC16(0.2735);
    CurrentLoop.pPIrAWQ.u16NShift     = lu;
}

```

```
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
    // (...)

}
```

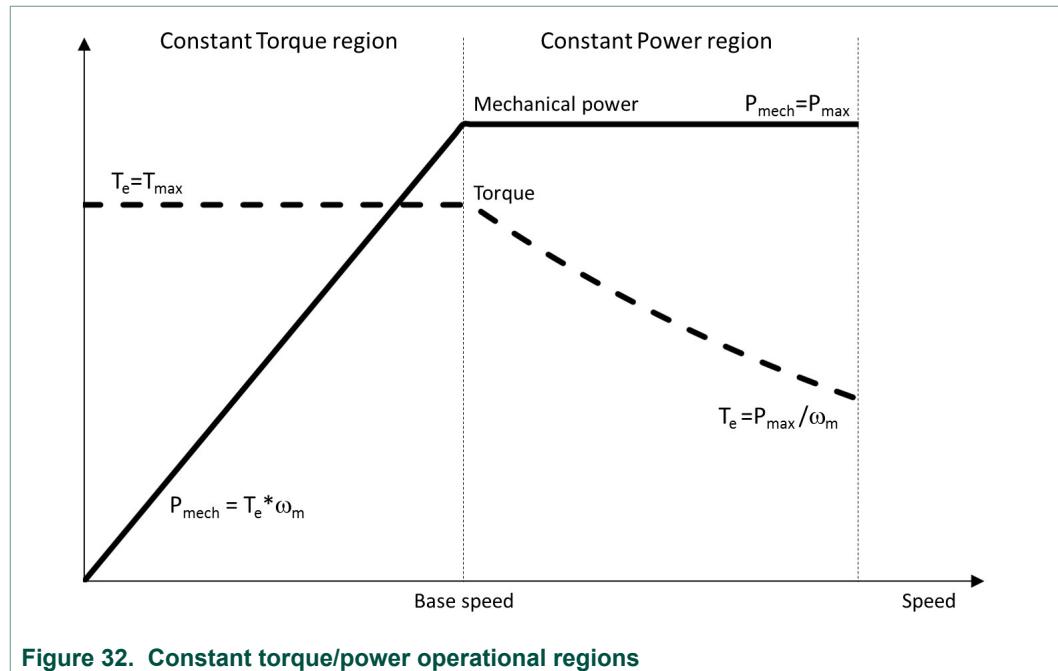
2.4 Function AMCLIB_FW

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

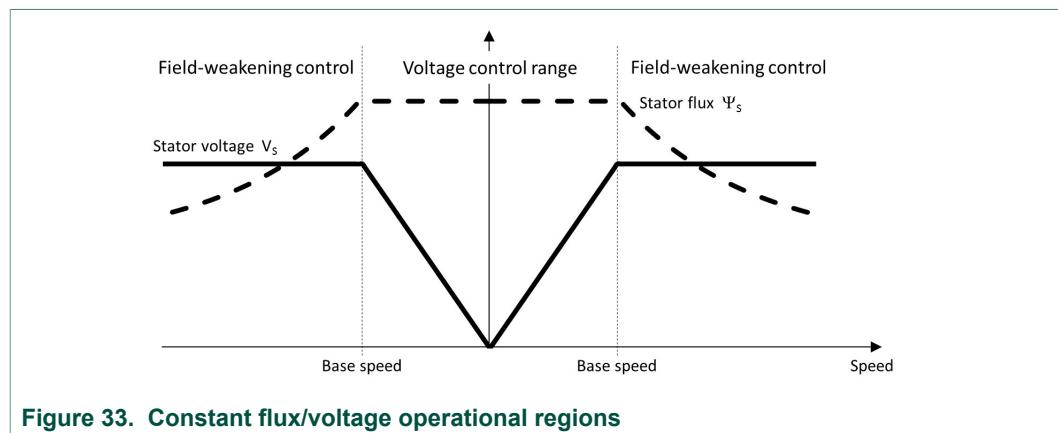
Description

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.

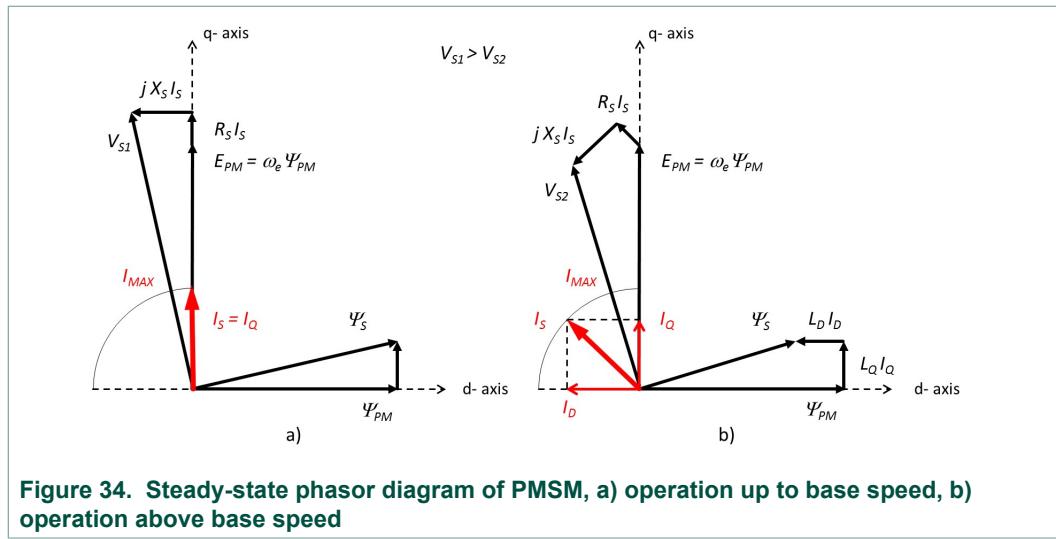
Base speed splits the whole speed motor operation into two regions: constant torque and constant power, see [Figure 32](#).



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 33](#).



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 34](#).



The left diagram in [Figure 34](#) 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 ω_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 34](#) 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 ω_e ; however, the terminal voltage V_{S2} is now lower than the previous V_{S1} . This is achieved thanks to the negative flux component $L_D I_D$ which counteracts the flux of the permanent magnets.

AMCLIB_FW implements the field weakening controller in the outer control loop highlighted in [Figure 35](#). 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.

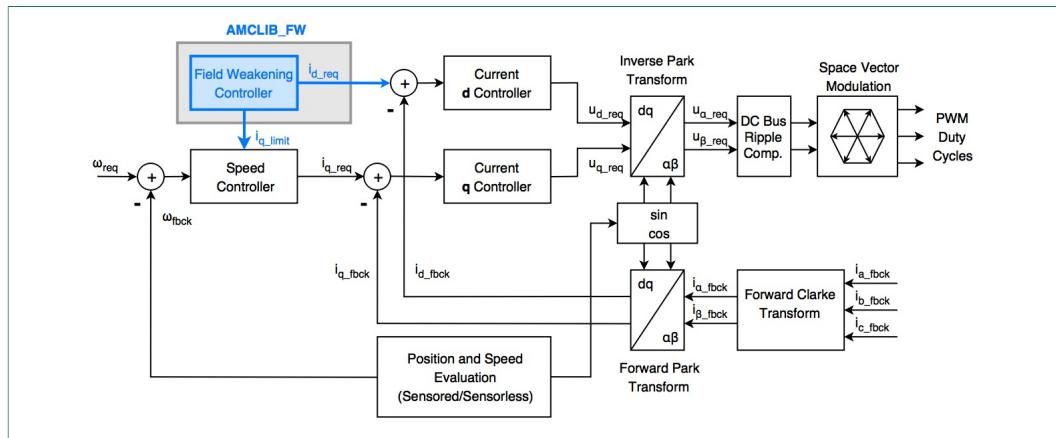


Figure 35. Principal Field Oriented Control scheme with AMCLIB_FW

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 (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 19. AMCLIB_FW_F32 arguments

Type	Name	Direction	Description
tFrac32	f32IDQReqAmp	input	Required amplitude of the currents Id and Iq 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_T_F32 *	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.

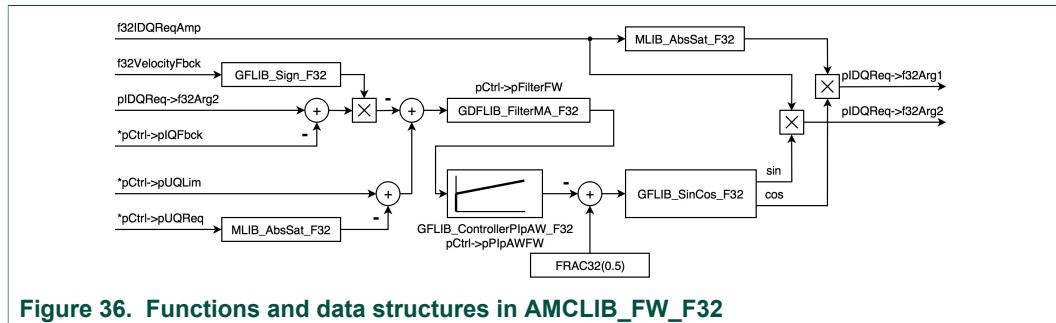


Figure 36. Functions and data structures in AMCLIB_FW_F32

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 20. 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]	Ω_{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

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

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.

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 21. AMCLIB_FW_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.

Type	Name	Direction	Description
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_T_F16 *	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.

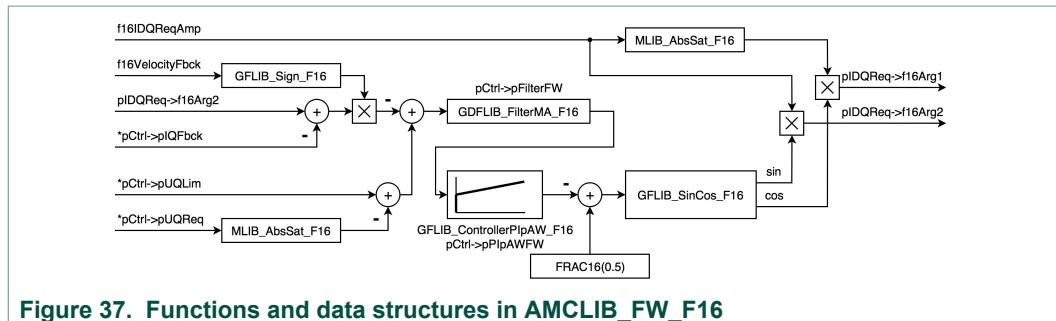


Figure 37. Functions and data structures in AMCLIB_FW_F16

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 22. 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]	Ω_{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 $\langle FRAC16(-0.5); 0 \rangle$. 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.

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_FWInit

Description

This function clears the AMCLIB_FW state variables.

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

Re-entrancy

The function is re-entrant for a different pCtrl.

2.4.3.1 Function AMCLIB_FWInit_F32**Declaration**

```
void AMCLIB_FWInit_F32(AMCLIB_FW_T_F32 *const pCtrl);
```

Arguments**Table 23. AMCLIB_FWInit_F32 arguments**

Type	Name	Direction	Description
AMCLIB_FW_T_F32 *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
    // (...)
```

2.4.3.2 Function AMCLIB_FWInit_F16

Declaration

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

Arguments

Table 24. 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.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
    // (...)
```

2.4.4 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 (HardFault).

Re-entrancy

The function is re-entrant for a different pCtrl.

2.4.4.1 Function AMCLIB_FWSetState_F32**Declaration**

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

Arguments**Table 25. AMCLIB_FWSetState_F32 arguments**

Type	Name	Direction	Description
tFrac32	f32FilterMAFWOut	input	Required output of the FilterMA.
tFrac32	f32ControllerPIpAWFWOut	output	Required output of the ControllerPIpAW.
AMCLIB_FW_T_F32 *	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.ul6NSamples = 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_FWSetState_F16

Declaration

```
void AMCLIB_FWSetState_F16(tFrac16 f16FilterMAFWOut, tFrac16  
f16ControllerPIpAWFWOut, AMCLIB\_FW\_T\_F16 *pCtrl);
```

Arguments

Table 26. AMCLIB_FWSetState_F16 arguments

Type	Name	Direction	Description
tFrac16	f16FilterMAFWOut	input	Required output of the FilterMA.
tFrac16	f16ControllerPIpAWFWOut	output	Required output of the ControllerPIpAW.
AMCLIB_FW_T_F16 *	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.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
    // (...)

}

```

2.4.5 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 38](#). 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.

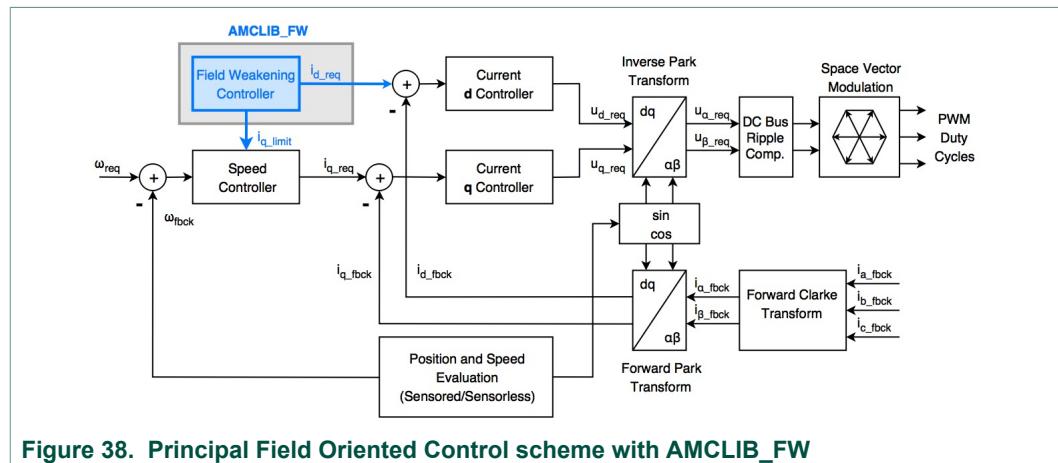


Figure 38. Principal Field Oriented Control scheme with AMCLIB_FW

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 id_req. See [AMCLIB_FW](#) for more details.

2.4.5.1 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 27. AMCLIB_FWDebug_F32 arguments

Type	Name	Direction	Description
tFrac32	f32IDQReqAmp	input	Required amplitude of the currents Id and Iq 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.

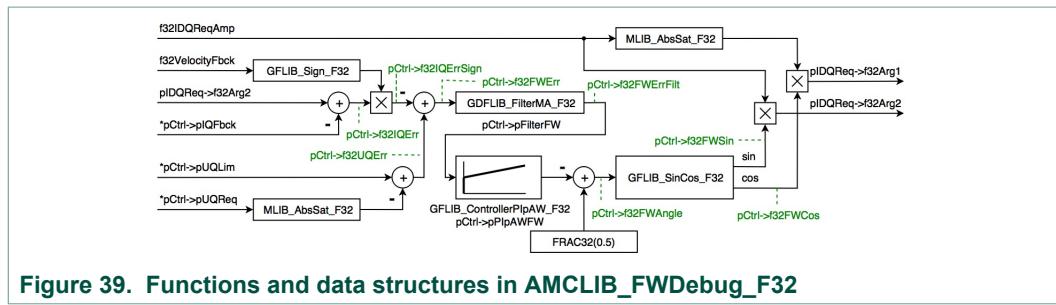


Figure 39. Functions and data structures in AMCLIB_FWDebug_F32

Refer to the description of [AMCLIB_FW_F32](#) function on how to set up the controller parameters.

Code Example

```
#include "amclib.h"

AMCLIB_FW_DEBUG_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.pPIPFW.f32PropGain        = (tFrac32)FRAC32(0.2348);
FW.pPIPFW.f32IntegGain       = (tFrac32)FRAC32(0.3457);
FW.pPIPFW.s16PropGainShift   = 1;
FW.pPIPFW.s16IntegGainShift  = 1;
FW.pPIPFW.f32LowerLimit     = (tFrac32)-2012406926L;
FW.pPIPFW.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
    // (...)

}

```

2.4.5.2 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 28. 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.

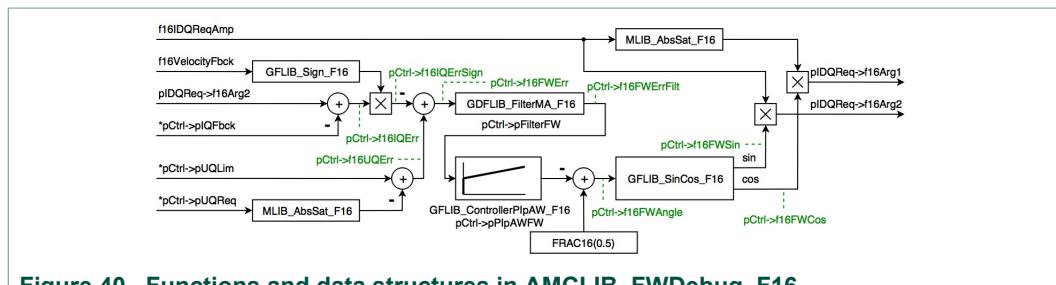


Figure 40. Functions and data structures in AMCLIB_FWDebug_F16

Refer to the description of [AMCLIB_FW_F16](#) function on how to set up the controller parameters.

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
    // (...)

}
```

2.5 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 41](#). 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.

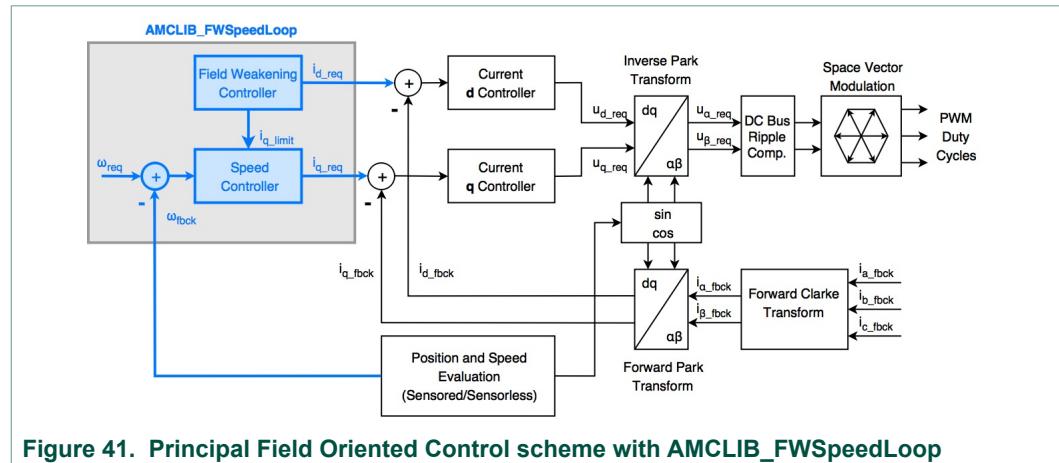


Figure 41. Principal Field Oriented Control scheme with AMCLIB_FWSpeedLoop

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 i_d_{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 (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 29. AMCLIB_FWSpeedLoop_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	input, output	Pointer to the structure with the required stator currents in the two-phase rotational orthogonal system (d-q).
<u>AMCLIB_FW_SPEED_LOOP_T_F32</u> *	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.

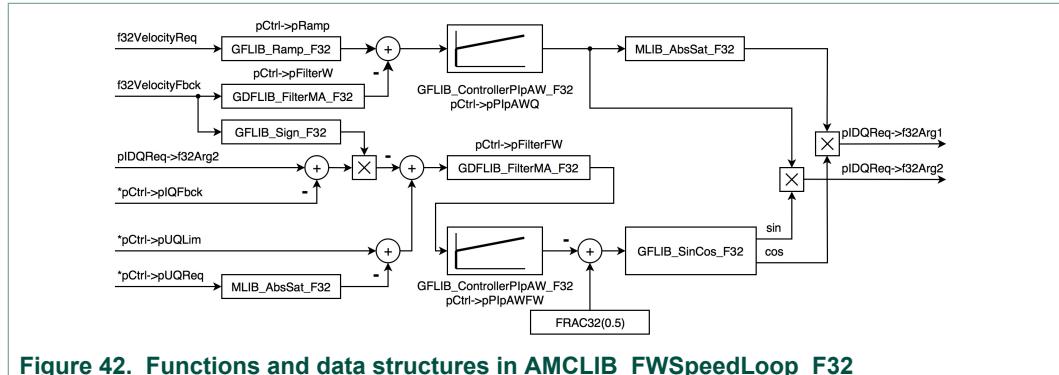


Figure 42. Functions and data structures in AMCLIB_FWSpeedLoop_F32

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 30. 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]	Ω_{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:

$$\begin{aligned}
 pPIpAWQf32PropGain &= FRAC32\left(\left(2 \cdot \xi \cdot \omega_0 \cdot \frac{J}{K_T}\right) \cdot \frac{\Omega_{MAX}}{T_{MAX}} \cdot 2^{-NShiftP}\right) \\
 pPIpAWQf32IntegGain &= FRAC32\left(\left(\omega_0^2 \cdot \frac{J}{K_T} \cdot \frac{T_s}{2}\right) \cdot \frac{\Omega_{MAX}}{T_{MAX}} \cdot 2^{-NShiftI}\right) \\
 pPIpAWQs16PropGainShift &= NShiftP \\
 pPIpAWQs16IntegGainShift &= NShiftI \\
 pPIpAWQf32UpperLimit &= FRAC32(1) \\
 pPIpAWQf32LowerLimit &= FRAC32(-1)
 \end{aligned}$$

Equation AMCLIB_FWSpeedLoop_F32_Eq1

where ξ is the speed loop attenuation, ω_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 $<\text{FRAC32}(-0.5); 0>$. 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

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

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.

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
    // (...)

}

```

2.5.2 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 31. AMCLIB_FWSpeedLoop_F16 arguments

Type	Name	Direction	Description
tFrac16	f16VelocityReq	input	Required electrical angular velocity (setpoint).
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_SPEED_LOOP_T_F16 *	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.

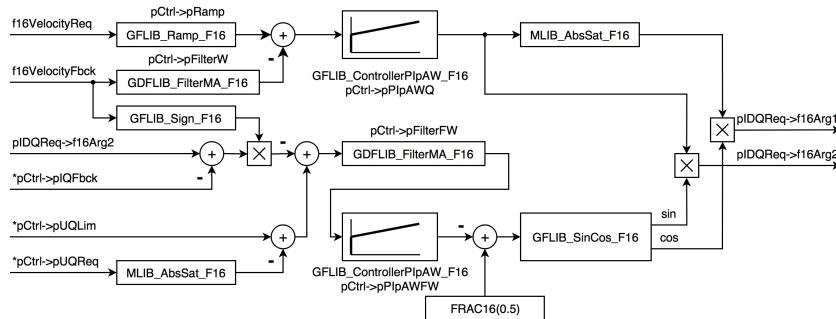


Figure 43. Functions and data structures in AMCLIB_FWSpeedLoop_F16

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 32. 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]	Ω_{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:

$$\begin{aligned}
 pPIpAWQ.f16PropGain &= FRAC16\left(\left(2 \cdot \xi \cdot \omega_0 \cdot \frac{J}{K_T}\right) \cdot \frac{\Omega_{MAX}}{T_{MAX}} \cdot 2^{NShiftP}\right) \\
 pPIpAWQ.f16IntegGain &= FRAC16\left(\left(\omega_0^2 \cdot \frac{J}{K_T} \cdot \frac{T_s}{2}\right) \cdot \frac{\Omega_{MAX}}{T_{MAX}} \cdot 2^{NShiftI}\right) \\
 pPIpAWQs16PropGainShift &= NShiftP \\
 pPIpAWQs16IntegGainShift &= NShiftI \\
 pPIpAWQ.f16UpperLimit &= FRAC16(1) \\
 pPIpAWQ.f16LowerLimit &= FRAC16(-1)
 \end{aligned}$$

Equation AMCLIB_FWSpeedLoop_F16_Eq1

where ξ is the speed loop attenuation, ω_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 $<\text{FRAC16}(-0.5); 0>$. 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

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

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.

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  
    // (...)  
}
```

2.5.3 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 (HardFault).

Re-entrancy

The function is re-entrant for a different pCtrl.

2.5.3.1 Function AMCLIB_FWSpeedLoopInit_F32

Declaration

```
void AMCLIB_FWSpeedLoopInit_F32 (AMCLIB_FW_SPEED_LOOP_T_F32 *const pCtrl);
```

Arguments

Table 33. AMCLIB_FWSpeedLoopInit_F32 arguments

Type	Name	Direction	Description
AMCLIB_FW_SPEED_LOOP_T_F32 *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.pPIpAWQ.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
    // (...)

}

```

2.5.3.2 Function AMCLIB_FWSpeedLoopInit_F16

Declaration

```
void AMCLIB_FWSpeedLoopInit_F16(AMCLIB\_FW\_SPEED\_LOOP\_T\_F16 *const pCtrl);
```

Arguments

Table 34. AMCLIB_FWSpeedLoopInit_F16 arguments

Type	Name	Direction	Description
AMCLIB_FW_SPEED_LOOP_T_F16 *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 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 (HardFault).

Re-entrancy

The function is re-entrant for a different pCtrl.

2.5.4.1 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 35. AMCLIB_FWSpeedLoopSetState_F32 arguments

Type	Name	Direction	Description
<u>tFrac32</u>	f32FilterMAWOut	input	Required output of the speed FilterMA.
<u>tFrac32</u>	f32FilterMAFWOut	input	Required output of the field-weakening FilterMA.
<u>tFrac32</u>	f32ControllerPIpAWQOut	input	Required output of the speed ControllerPIpAW.
<u>tFrac32</u>	f32ControllerPIpAWFWOut	input	Required output of the field-weakening ControllerPIpAW.
<u>tFrac32</u>	f32RampOut	input	Required output of the speed ramp.

Type	Name	Direction	Description
AMCLIB_FW_SPEED_LOOP_T_F32 *	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);

// 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_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 36. AMCLIB_FWSpeedLoopSetState_F16 arguments

Type	Name	Direction	Description
tFrac16	f16FilterMAWOut	input	Required output of the speed FilterMA.
tFrac16	f16FilterMAFWOut	input	Required output of the field-weakening FilterMA.
tFrac16	f16ControllerPIpAWQOut	input	Required output of the speed ControllerPIpAW.
tFrac16	f16ControllerPIpAWFWOut	input	Required output of the field-weakening ControllerPIpAW.
tFrac32	f32RampOut	input	Required output of the speed ramp.
AMCLIB_FW_SPEED_LOOP_T_F16 *	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 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 44](#). 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.

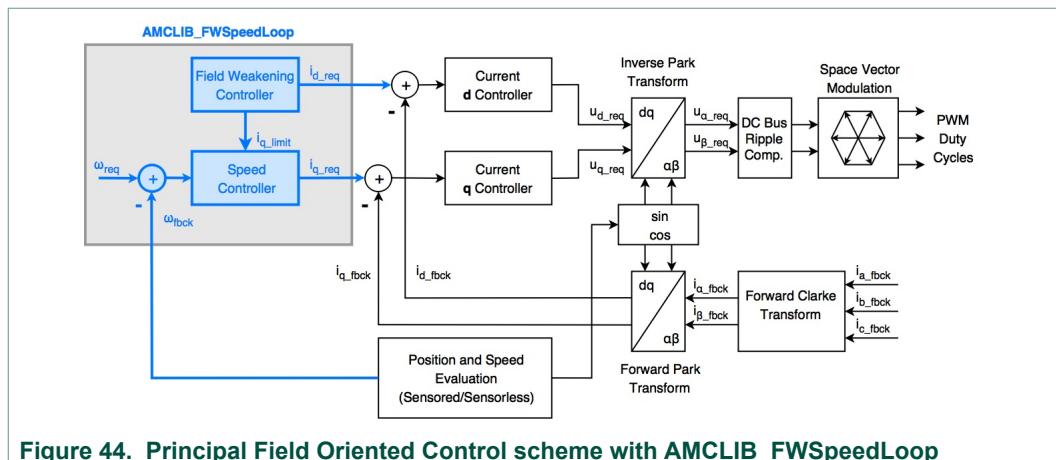


Figure 44. Principal Field Oriented Control scheme with AMCLIB_FWSpeedLoop

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.

2.5.5.1 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 37. AMCLIB_FWSpeedLoopDebug_F32 arguments

Type	Name	Direction	Description
tFrac32	f32VelocityReq	input	Required electrical angular velocity (setpoint).
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_SPEED_LOOP_DEBUG_T_F32 *	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.

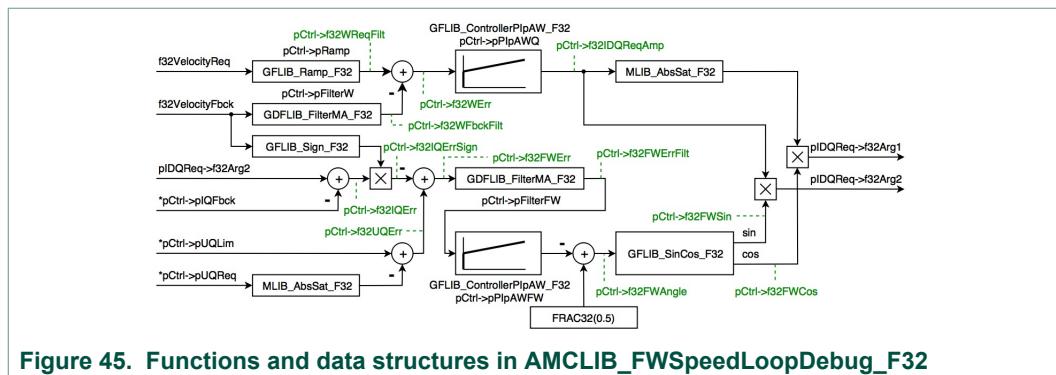


Figure 45. Functions and data structures in AMCLIB_FWSpeedLoopDebug_F32

Refer to the description of [AMCLIB_FWSpeedLoop_F32](#) function on how to set up the controller parameters.

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.pPIpAWQ.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
    // (...)

}
  
```

2.5.5.2 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 38. AMCLIB_FWSpeedLoopDebug_F16 arguments

Type	Name	Direction	Description
tFrac16	f16VelocityReq	input	Required electrical angular velocity (setpoint).
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_SPEED_LOOP_DEBUG_T_F16 *	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.

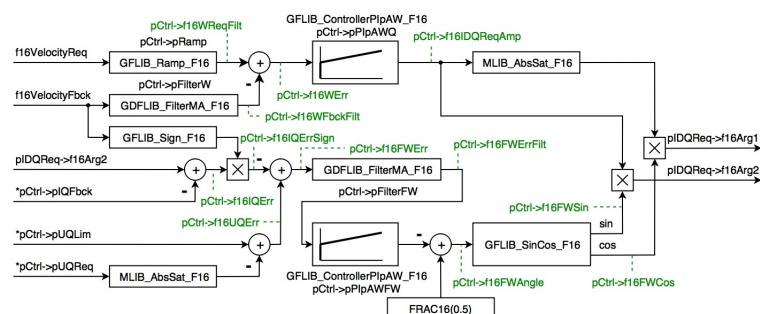


Figure 46. Functions and data structures in AMCLIB_FWSpeedLoopDebug_F16

Refer to the description of [AMCLIB_FWSpeedLoop_F16](#) function on how to set up the controller parameters.

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.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((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
    // (...)

}

```

2.6 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 47](#). 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.

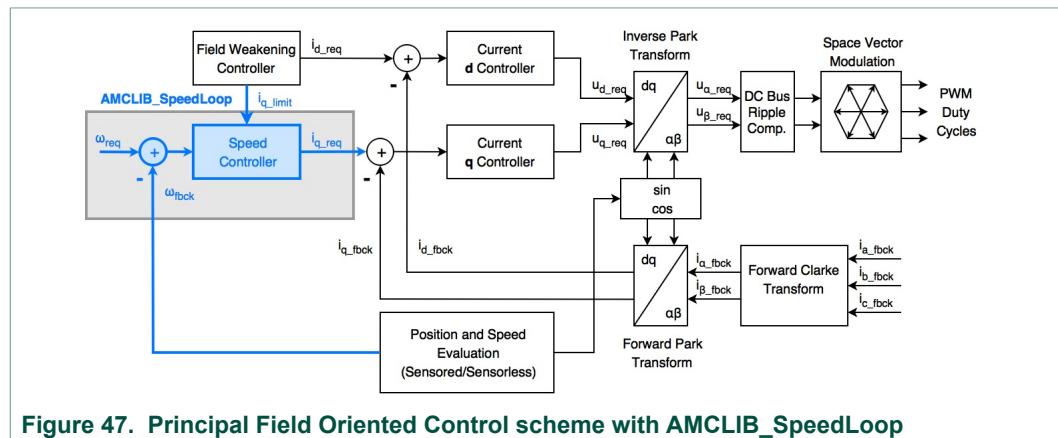


Figure 47. Principal Field Oriented Control scheme with AMCLIB_SpeedLoop

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 (HardFault).

Re-entrancy

The function is re-entrant for a different pCtrl.

2.6.1 Function AMCLIB_SpeedLoop_F32

Declaration

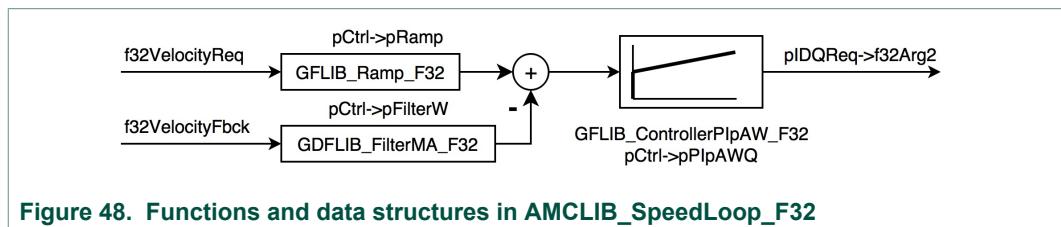
```
void AMCLIB_SpeedLoop_F32(tFrac32 f32VelocityReq,
tFrac32 f32VelocityFbck, SWLIBS\_2Syst\_F32 *const pIDQReq,
AMCLIB\_SPEED\_LOOP\_T\_F32 *pCtrl);
```

Arguments**Table 39. 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.
SWLIBS_2Syst_F32 *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_T_F32 *	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 48. 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 40. Scaling constants

Scaling constant	Symbol	Calculation
Maximum phase current [A]	I_{MAX}	Maximum current depends on power stage capabilities.
Maximum speed [rad/s]	Ω_{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:

$$\begin{aligned}
 pPipAWQ.f32PropGain &= FRAC32\left(\left(2 \cdot \xi \cdot \omega_0 \cdot \frac{J}{K_T}\right) \cdot \frac{\Omega_{MAX}}{T_{MAX}} \cdot 2^{NShiftP}\right) \\
 pPipAWQ.f32IntegGain &= FRAC32\left(\left(\omega_0^2 \cdot \frac{J}{K_T} \cdot \frac{T_s}{2}\right) \cdot \frac{\Omega_{MAX}}{T_{MAX}} \cdot 2^{NShiftI}\right) \\
 pPipAWQs16PropGainShift &= NShiftP \\
 pPipAWQs16IntegGainShift &= NShiftI \\
 pPipAWQ.f32UpperLimit &= FRAC32(1) \\
 pPipAWQ.f32LowerLimit &= FRAC32(-1) \\
 \end{aligned}$$

Equation AMCLIB_SpeedLoop_F32_Eq1

where ξ is the speed loop attenuation, ω_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.

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);
}

```

2.6.2 Function AMCLIB_SpeedLoop_F16

Declaration

```
void AMCLIB_SpeedLoop_F16(tFrac16 f16VelocityReq,
    tFrac16 f16VelocityFbck, SWLIBS\_2Syst\_F16 *const pIDQReq,
    AMCLIB\_SPEED\_LOOP\_T\_F16 *pCtrl);
```

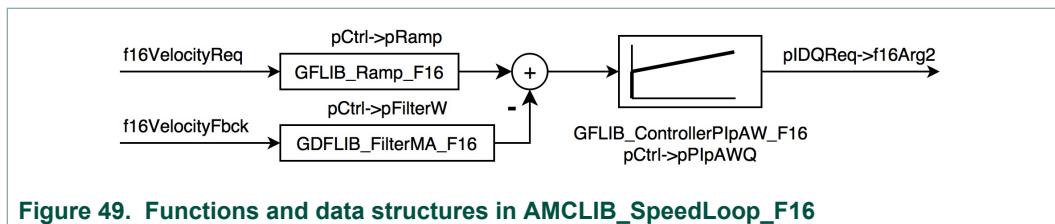
Arguments

Table 41. AMCLIB_SpeedLoop_F16 arguments

Type	Name	Direction	Description
tFrac16	f16VelocityReq	input	Required electrical angular velocity (setpoint).
tFrac16	f16VelocityFbck	input	Actual electrical angular velocity from the feedback.
SWLIBS_2Syst_F16 *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_T_F16 *	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 49. 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 42. Scaling constants

Scaling constant	Symbol	Calculation
Maximum phase current [A]	I_{MAX}	Maximum current depends on power stage capabilities.
Maximum speed [rad/s]	Ω_{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:

$$\begin{aligned}
 pPIpAWQ.f16PropGain &= FRAC16\left(\left(2 \cdot \xi \cdot \omega_0 \cdot \frac{J}{K_T}\right) \cdot \frac{\Omega_{MAX}}{T_{MAX}} \cdot 2^{NShiftP}\right) \\
 pPIpAWQ.f16IntegGain &= FRAC16\left(\left(\omega_0^2 \cdot \frac{J}{K_T} \cdot \frac{T_s}{2}\right) \cdot \frac{\Omega_{MAX}}{T_{MAX}} \cdot 2^{NShiftI}\right) \\
 pPIpAWQs16PropGainShift &= NShiftP \\
 pPIpAWQs16IntegGainShift &= NShiftI \\
 pPIpAWQ.f16UpperLimit &= FRAC16(1) \\
 pPIpAWQ.f16LowerLimit &= FRAC16(-1) \\
 \text{Equation AMCLIB_SpeedLoop_F16_Eq1}
 \end{aligned}$$

where ξ is the speed loop attenuation, ω_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.

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);
}

```

2.6.3 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 (HardFault).

Re-entrancy

The function is re-entrant for a different pCtrl.

2.6.3.1 Function AMCLIB_SpeedLoopInit_F32

Declaration

```
void AMCLIB_SpeedLoopInit_F32(AMCLIB\_SPEED\_LOOP\_T\_F32 *const pCtrl);
```

Arguments

Table 43. AMCLIB_SpeedLoopInit_F32 arguments

Type	Name	Direction	Description
AMCLIB_SPEED_LOOP_T_F32 *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);
}

```

2.6.3.2 Function AMCLIB_SpeedLoopInit_F16

Declaration

```
void AMCLIB_SpeedLoopInit_F16(AMCLIB\_SPEED\_LOOP\_T\_F16 *const pCtrl);
```

Arguments

Table 44. AMCLIB_SpeedLoopInit_F16 arguments

Type	Name	Direction	Description
AMCLIB_SPEED_LOOP_T_F16 *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);

// 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);
}
```

2.6.4 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 (HardFault).

Re-entrancy

The function is re-entrant for a different pCtrl.

2.6.4.1 Function AMCLIB_SpeedLoopSetState_F32

Declaration

```
void AMCLIB_SpeedLoopSetState_F32 (tFrac32 f32FilterMAWOut,
tFrac32 f32ControllerPIpAWQOut, tFrac32 f32RampOut,
AMCLIB\_SPEED\_LOOP\_T\_F32 *pCtrl);
```

Arguments

Table 45. AMCLIB_SpeedLoopSetState_F32 arguments

Type	Name	Direction	Description
tFrac32	f32FilterMAWOut	input	Required output of the FilterMA.
tFrac32	f32ControllerPIpAWQOut	input	Required output of the ControllerPIpAW.
tFrac32	f32RampOut	input	Required output of the speed ramp.
AMCLIB_SPEED_LOOP_T_F32 *	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, 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_SpeedLoopSetState_F16

Declaration

```
void AMCLIB_SpeedLoopSetState_F16(tFrac16 f16FilterMAWOut,
tFrac16 f16ControllerPIpAWQOut, tFrac32 f32RampOut,
AMCLIB\_SPEED\_LOOP\_T\_F16 *pCtrl);
```

Arguments

Table 46. AMCLIB_SpeedLoopSetState_F16 arguments

Type	Name	Direction	Description
tFrac16	f16FilterMAWOut	input	Required output of the FilterMA.
tFrac16	f16ControllerPIpAWQOut	input	Required output of the ControllerPIpAW.
tFrac32	f32RampOut	input	Required output of the speed ramp.
AMCLIB_SPEED_LOOP_T_F16 *	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_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 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 50](#). 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.

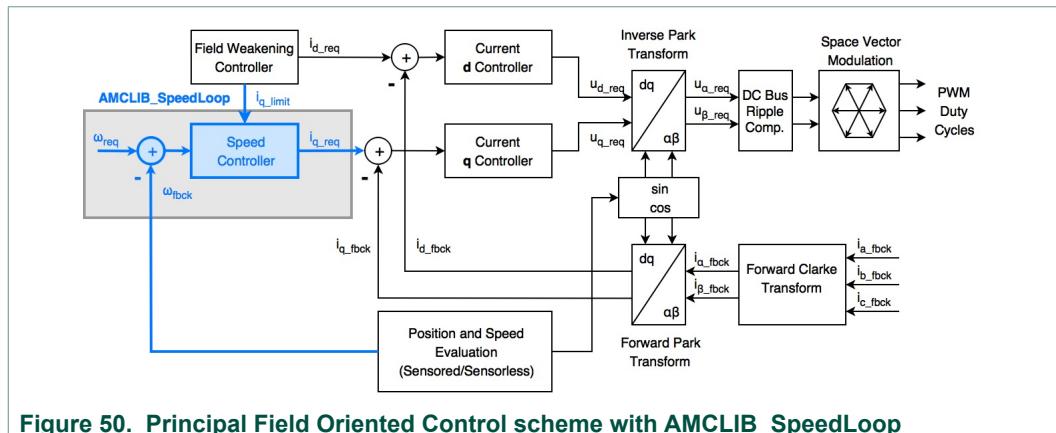


Figure 50. Principal Field Oriented Control scheme with AMCLIB_SpeedLoop

2.6.5.1 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 47. AMCLIB_SpeedLoopDebug_F32 arguments

Type	Name	Direction	Description
tFrac32	f32VelocityReq	input	Required electrical angular velocity (setpoint).
tFrac32	f32VelocityFbck	input	Actual electrical angular velocity from the feedback.
SWLIBS_2Syst_F32 *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.

Type	Name	Direction	Description
AMCLIB_SPEED_LOOP_DEBUG_T_F32 *	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.

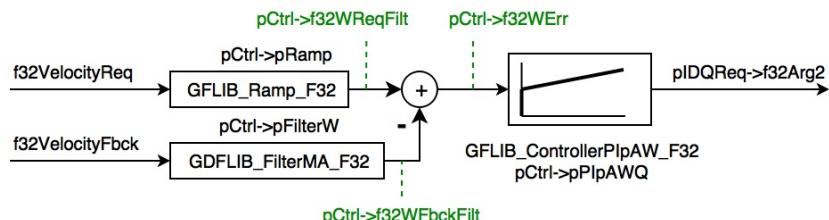


Figure 51. Functions and data structures in AMCLIB_SpeedLoopDebug_F32

Refer to the description of [AMCLIB_SpeedLoop_F32](#) function on how to set up the controller parameters.

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.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((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);
}

```

2.6.5.2 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 48. AMCLIB_SpeedLoopDebug_F16 arguments

Type	Name	Direction	Description
tFrac16	f16VelocityReq	input	Required electrical angular velocity (setpoint).
tFrac16	f16VelocityFbck	input	Actual electrical angular velocity from the feedback.
SWLIBS_2Syst_F16 *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_F16 *	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.

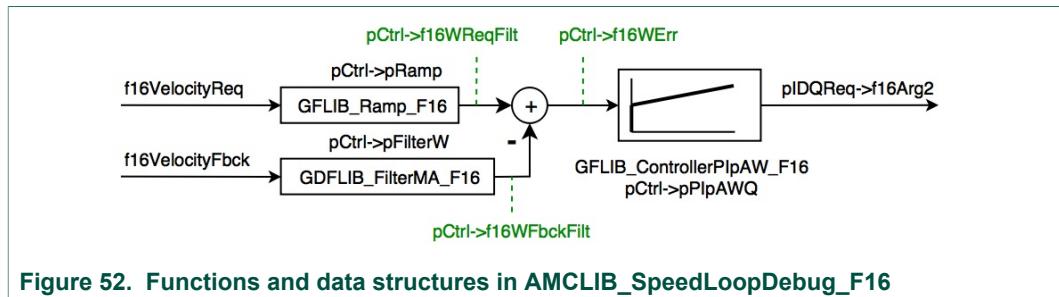


Figure 52. Functions and data structures in AMCLIB_SpeedLoopDebug_F16

Refer to the description of [AMCLIB_SpeedLoop_F16](#) function on how to set up the controller parameters.

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);
}
```

2.7 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 Θ_{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:

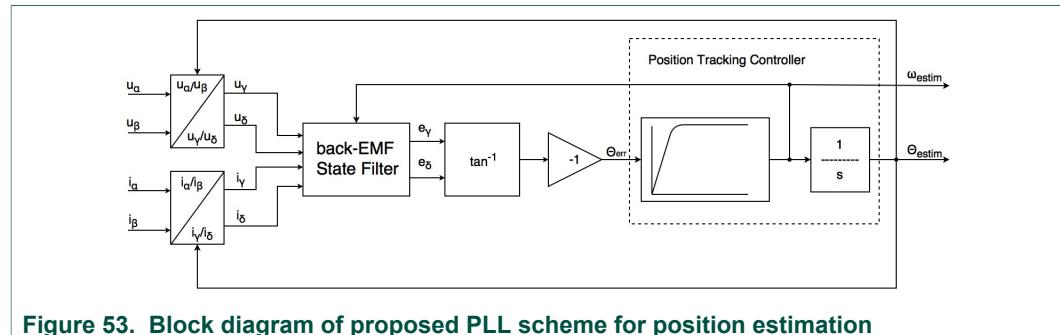


Figure 53. Block diagram of proposed PLL scheme for position estimation

The position tracking structure, described in [Figure 53](#), 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:

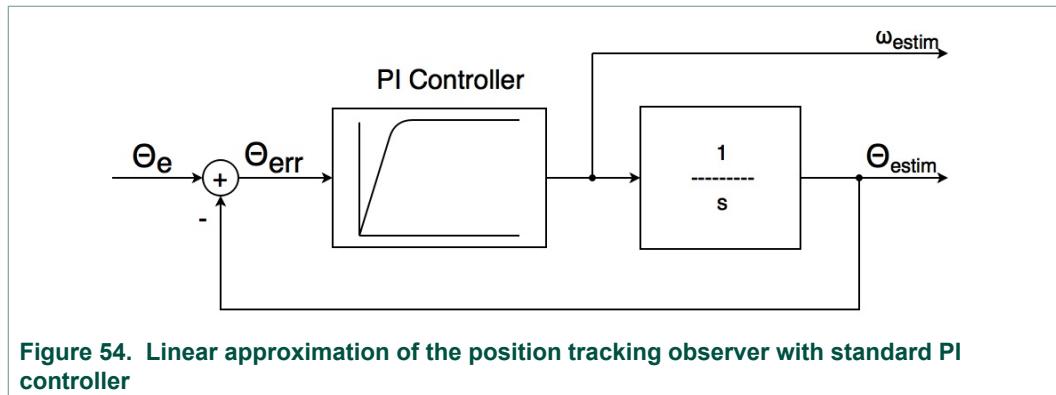


Figure 54. Linear approximation of the position tracking observer with standard PI controller

Linearized position tracking observer, depicted on [Figure 54](#) has the following transfer function:

$$G(s) = \frac{\theta_{\text{estim}}(s)}{\theta_e(s)} = \frac{sK_p + K_i}{s^2 + sK_p + K_i}$$

Equation AMCLIB_TrackObsrv_Eq2

Considering the s^2K_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:

$$\begin{aligned} K_p &= 4\pi\xi f_0 \\ K_i &= (2\pi f_0)^2 \end{aligned}$$

Equation AMCLIB_TrackObsrv_Eq3

where:

- ξ 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 53](#), 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:

$$\omega_{\text{estim}}(t) = \theta_{\text{err}}(t) \cdot K_p + K_i \int_0^\infty \theta_{\text{err}}(t) dt$$

Equation AMCLIB_TrackObsrv_Eq4

and the output of the ideal integrator as follows:

$$\theta_{\text{estim}}(t) = \int_0^\infty \omega_{\text{estim}}(t) dt$$

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:

$$\omega_{estim}(k) = \omega_{estim}(k-1) + \theta_{err}(k) \cdot \left(K_p + \frac{K_i T_s}{2} \right) + \theta_{err}(k-1) \cdot \left(-K_p + \frac{K_i T_s}{2} \right)$$

$$\theta_{estim}(k) = \theta_{estim}(k-1) + \omega_{estim}(k) \cdot \frac{T_s}{2} + \omega_{estim}(k-1) \cdot \frac{T_s}{2}$$

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:

$$CC_1 = K_p + \frac{K_i T_s}{2}$$

$$CC_2 = -K_p + \frac{K_i T_s}{2}$$

$$C_1 = \frac{T_s}{2}$$

Equation AMCLIB_TrackObsrv_Eq9

the equation [AMCLIB_TrackObsrv_Eq8](#) can be rewritten into:

$$\omega_{estim}(k) = \omega_{estim}(k-1) + \theta_{err}(k) \cdot CC_1 + \theta_{err}(k-1) \cdot CC_2$$

$$\theta_{estim}(k) = \theta_{estim}(k-1) + \omega_{estim}(k) \cdot C_1 + \omega_{estim}(k-1) \cdot C_1$$

Equation AMCLIB_TrackObsrv_Eq10

Note: The input/output pointers must contain valid addresses, otherwise a fault may occur (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 49. AMCLIB_TrackObsrv_F32 arguments

Type	Name	Direction	Description
tFrac32	f32PhaseErr	input	Input signal representing phase error of system to be estimated.
tFrac32 *	pPosEst	output	Estimated output position.
tFrac32 *	pVelocityEst	output	Estimated output velocity.
AMCLIB_TRACK_OBSRV_T_F32 *	pCtrl	input, output	Pointer to a tracking observer structure AMCLIB_TRACK_OBSRV_T_F32 , 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:

- Θ^{MAX} - maximal value of the position tracking controller input phase error
- Ω^{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 Θ_{err} and the output phase Θ_{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:

$$\begin{aligned}\omega_{estim}^{estim}(k) &= \omega_{estim}(k-1) + \theta_{err}(k) \cdot \frac{\theta^{MAX}}{\omega^{MAX}} \cdot CC_1 + \theta_{err}(k-1) \cdot \frac{\theta^{MAX}}{\omega^{MAX}} \cdot CC_2 \\ \theta_f^{estim} &= \theta_{estim}(k-1) + \omega_{estim}(k) \cdot \frac{\omega^{MAX}}{\theta^{MAX}} \cdot C_1 + \omega_{estim}(k-1) \cdot \frac{\omega^{MAX}}{\theta^{MAX}} \cdot C_1\end{aligned}$$

Equation AMCLIB_TrackObsrv_F32_Eq3

To further simplify the equation [AMCLIB_TrackObsrv_F32_Eq3](#), let's make the substitution:

$$\begin{aligned}CC_{1f} &= CC_1 \cdot \frac{\theta^{MAX}}{\omega^{MAX}} = \left(K_p + \frac{K_f T_s}{2} \right) \cdot \frac{\theta^{MAX}}{\omega^{MAX}} \\ CC_{2f} &= CC_2 \cdot \frac{\theta^{MAX}}{\omega^{MAX}} = \left(-K_p + \frac{K_f T_s}{2} \right) \cdot \frac{\theta^{MAX}}{\omega^{MAX}} \\ C_{1f} &= C_1 \cdot \frac{\omega^{MAX}}{\theta^{MAX}} = \frac{T_s}{2} \cdot \frac{\omega^{MAX}}{\theta^{MAX}}\end{aligned}$$

Equation AMCLIB_TrackObsrv_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 Θ^{MAX} , Ω^{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 Θ_{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.ul6NShift    = (tU16)0;

    // Setting parameters for integrator
    trMyTrObsrv.pParamInteg.f32C1      = FRAC32((Ts/2)*Wmax/pi);
    trMyTrObsrv.pParamInteg.ul6NShift = (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 50. 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.
AMCLIB_TRACK_OBSRV_T_F16 *	pCtrl	input, output	Pointer to a tracking observer structure AMCLIB_TRACK_OBSRV_T_F16 , 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:

- Θ^{MAX} - the maximal value of the position tracking controller input phase error
- Ω^{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 Θ_{err} and the output phase Θ_{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:

$$\begin{aligned}\omega_f^{estim}(k) \cdot \omega^{MAX} &= \omega_{estim}(k-1) \cdot \omega^{MAX} + \theta_{err}(k) \cdot \theta^{MAX} \cdot CC_1 + \theta_{err}(k-1) \cdot \theta^{MAX} \cdot CC_2 \\ \theta_f^{estim} \cdot \theta^{MAX} &= \theta_{estim}(k-1) \cdot \theta^{MAX} + \omega_{estim}(k) \cdot \omega^{MAX} \cdot C_1 + \omega_{estim}(k-1) \cdot \omega^{MAX} \cdot C_1\end{aligned}$$

Equation AMCLIB_TrackObsrv_F16_Eq2

which can be rearranged into this form:

$$\begin{aligned}\omega_f^{estim}(k) &= \omega_{estim}(k-1) + \theta_{err}(k) \cdot \frac{\theta^{MAX}}{\omega^{MAX}} \cdot CC_1 + \theta_{err}(k-1) \cdot \frac{\theta^{MAX}}{\omega^{MAX}} \cdot CC_2 \\ \theta_f^{estim} &= \theta_{estim}(k-1) + \omega_{estim}(k) \cdot \frac{\omega^{MAX}}{\theta^{MAX}} \cdot C_1 + \omega_{estim}(k-1) \cdot \frac{\omega^{MAX}}{\theta^{MAX}} \cdot C_1\end{aligned}$$

Equation AMCLIB_TrackObsrv_F16_Eq3

To further simplify the equation [AMCLIB_TrackObsrv_F16_Eq3](#), let's make the substitution:

$$\begin{aligned}CC_{1f} &= CC_1 \cdot \frac{\theta^{MAX}}{\omega^{MAX}} = \left(K_p + \frac{K_f T_s}{2} \right) \cdot \frac{\theta^{MAX}}{\omega^{MAX}} \\ CC_{2f} &= CC_2 \cdot \frac{\theta^{MAX}}{\omega^{MAX}} = \left(-K_p + \frac{K_f T_s}{2} \right) \cdot \frac{\theta^{MAX}}{\omega^{MAX}} \\ C_{1f} &= C_1 \cdot \frac{\omega^{MAX}}{\theta^{MAX}} = \frac{T_s}{2} \cdot \frac{\omega^{MAX}}{\theta^{MAX}}\end{aligned}$$

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 Θ^{MAX} and Ω^{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:

$$\begin{aligned}f16CC1_{SC} &= CC_{1f} \cdot 2^{u16NShift} = \left(K_p + \frac{K_f T_s}{2} \right) \cdot \frac{\theta^{MAX}}{\omega^{MAX}} \cdot 2^{u16NShift} \\ f16CC2_{SC} &= CC_{2f} \cdot 2^{u16NShift} = \left(-K_p + \frac{K_f T_s}{2} \right) \cdot \frac{\theta^{MAX}}{\omega^{MAX}} \cdot 2^{u16NShift} \\ f16C1_{SC} &= C_{1f} \cdot 2^{u16NIntegSh} = \frac{T_s}{2} \cdot \frac{\omega^{MAX}}{\theta^{MAX}} \cdot 2^{u16NIntegSh}\end{aligned}$$

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 Θ_{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
    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);
}

```

2.7.3 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 (HardFault).

Re-entrancy:

The function is re-entrant for a different pCtrl.

2.7.3.1 Function AMCLIB_TrackObsrvInit_F32

Declaration

```
void AMCLIB_TrackObsrvInit_F32(AMCLIB_TRACK_OBSRV_T_F32 *pCtrl);
```

Arguments

Table 51. 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.ul6NShift    = (tu16)0;

    // Setting parameters for integrator
    trMyTrObsrv.pParamInteg.f32C1      = FRAC32((Ts/2)*Wmax/pi);
    trMyTrObsrv.pParamInteg.ul6NShift = (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);
}

```

2.7.3.2 Function AMCLIB_TrackObsrvInit_F16

Declaration

```
void AMCLIB_TrackObsrvInit_F16(AMCLIB_TRACK_OBSRV_T_F16 *pCtrl);
```

Arguments

Table 52. AMCLIB_TrackObsrvInit_F16 arguments

Type	Name	Direction	Description
AMCLIB_TRACK_OBSRV_T_F16 *	pCtrl	input, output	Pointer to a tracking observer structure AMCLIB_TRACK_OBSRV_T_F16 , 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.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);
}
```

2.7.4 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 (HardFault).

Re-entrancy:

The function is re-entrant for a different pCtrl.

2.7.4.1 Function AMCLIB_TrackObsrvSetState_F32

Declaration

```
void AMCLIB_TrackObsrvSetState_F32(tFrac32 f32PosOut, tFrac32 f32VelocityOut, AMCLIB\_TRACK\_OBSRV\_T\_F32 *pCtrl);
```

Arguments

Table 53. AMCLIB_TrackObsrvSetState_F32 arguments

Type	Name	Direction	Description
tFrac32	f32PosOut	input	Required output of the GFLIB_ControllerPIrAW, i.e., what rotor position shall be outputted in the next iteration.
tFrac32	f32VelocityOut	input	Required output of the GFLIB_IntegratorTR, i.e., what velocity shall be outputted in the next iteration.
AMCLIB_TRACK_OBSRV_T_F32 *	pCtrl	input, output	Pointer to a tracking observer structure AMCLIB_TRACK_OBSRV_T_F32 , 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);
}

```

2.7.4.2 Function AMCLIB_TrackObsrvSetState_F16

Declaration

```
void AMCLIB_TrackObsrvSetState_F16(tFrac16 f16PosOut, tFrac16
f16VelocityOut, AMCLIB\_TRACK\_OBSRV\_T\_F16 *pCtrl);
```

Arguments

Table 54. AMCLIB_TrackObsrvSetState_F16 arguments

Type	Name	Direction	Description
tFrac16	f16PosOut	input	Required output of the GFLIB_ControllerPIrAW, i.e., what rotor position shall be outputted in the next iteration.
tFrac16	f16VelocityOut	input	Required output of the GFLIB_IntegratorTR, i.e., what velocity shall be outputted in the next iteration.
AMCLIB_TRACK_OBSRV_T_F16 *	pCtrl	input, output	Pointer to a tracking observer structure AMCLIB_TRACK_OBSRV_T_F16 , 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
    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);
}
```

2.8 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.

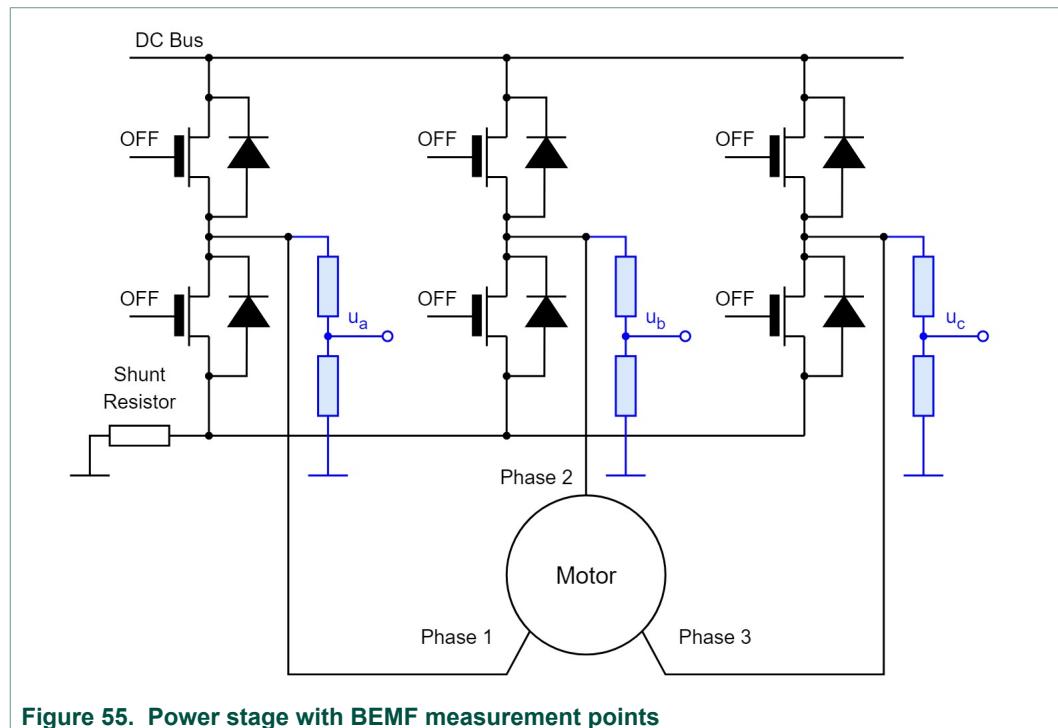


Figure 55. Power stage with BEMF measurement points

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.

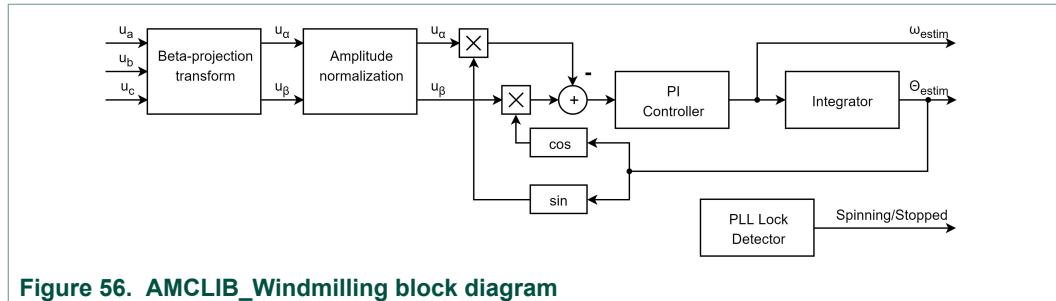


Figure 56. AMCLIB_Windmilling block diagram

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 (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 55. 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 GDFLIB_FilterMA_F32 to further filter the results. The filter should be engaged only after the AMCLIB_Windmilling has returned SPINNING ; use GDFLIB_FilterMASetState_F32 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_TrackObsrv_F32](#) (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_F32. It is necessary to call the [AMCLIB_WindmillingInit_F32](#) 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);
}

```

2.8.2 Function AMCLIB_Windmilling_F16

Declaration

```
AMCLIB_WINDMILLING_RET_T AMCLIB_Windmilling_F16(const
SWLIBS_3Syst_F16 *pUabcln, tFrac16 *pPosEst, tFrac16
*pVelocityEst, AMCLIB_WINDMILLING_T_F16 *const pCtrl);
```

Arguments

Table 56. AMCLIB_Windmilling_F16 arguments

Type	Name	Direction	Description
const SWLIBS_3Syst_F16 *	pUabcln	input	Pointer to the structure with the measured 3-phase A/B/C voltages.
tFrac16 *	pPosEst	output	Estimated rotor flux position in the interval <-1; 1) which corresponds to the angle range <-π; π) radians.
tFrac16 *	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 GDFLIB_FilterMA_F16 to further filter the results. The filter should be engaged only after the AMCLIB_Windmilling has returned SPINNING ; use GDFLIB_FilterMASetState_F16 to initialize the filter state with the first estimated velocity value.

Type	Name	Direction	Description
AMCLIB_WINDMILLING_T_F16 *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.

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.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_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 (HardFault).

Re-entrancy:

The function is re-entrant.

2.8.3.1 Function AMCLIB_WindmillingInit_F32

Declaration

```
void AMCLIB_WindmillingInit_F32(tFrac32 f32ADCMaxError,  
AMCLIB\_WINDMILLING\_T\_F32 *const pCtrl);
```

Arguments

Table 57. AMCLIB_WindmillingInit_F32 arguments

Type	Name	Direction	Description
tFrac32	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.
AMCLIB_WINDMILLING_T_F32 *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);
}

```

2.8.3.2 Function AMCLIB_WindmillingInit_F16

Declaration

```
void AMCLIB_WindmillingInit_F16(tFrac16 f16ADCMaxError,
AMCLIB\_WINDMILLING\_T\_F16 *const pCtrl);
```

Arguments

Table 58. AMCLIB_WindmillingInit_F16 arguments

Type	Name	Direction	Description
tFrac16	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.

Type	Name	Direction	Description
AMCLIB_WINDMILLING_T_F16 *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 f16Uabc;
AMCLIB\_WINDMILLING\_T\_F16 windStruct;
tFrac16 f16Position;
tFrac16 f16Velocity;
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);
}
```

2.9 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 (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 59. 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 60. 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
// 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_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 (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.

2.9.3.1 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 61. 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}
}
```

2.9.3.2 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 62. 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.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  
// 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, 0x7FF9}
```

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

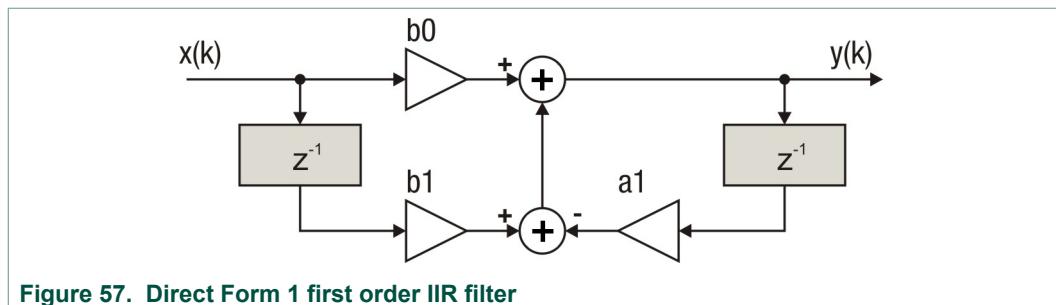
In order to implement the first order IIR filter on a microcontroller, the discrete time domain representation of the filter, described by equation [GDFLIB_FilterIIR1_Eq2](#), must be transformed into a time difference equation as follows:

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

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.



The coefficients of the filter depicted in [Figure 57](#) 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 (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 63. 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 64. 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);

    // ##### 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_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 (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.

2.10.3.1 Function GDFLIB_FilterIIR1Init_F32

Declaration

```
void GDFLIB_FilterIIR1Init_F32(GDFLIB_FILTER_IIR1_T_F32 *const pParam);
```

Arguments

Table 65. 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);
}
```

2.10.3.2 Function GDFLIB_FilterIIR1Init_F16

Declaration

```
void GDFLIB_FilterIIR1Init_F16(GDFLIB_FILTER_IIR1_T_F16 *const pParam);
```

Arguments

Table 66. 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);
}

```

2.11 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).

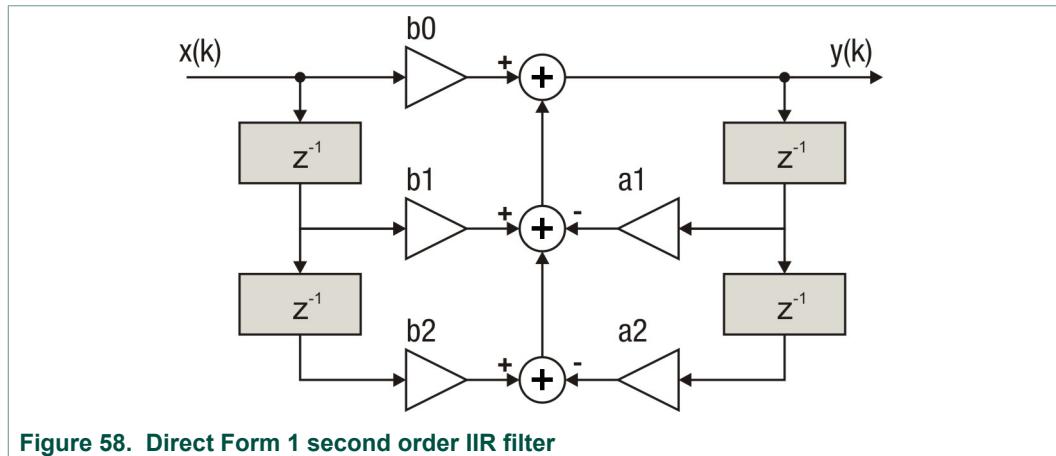


Figure 58. Direct Form 1 second order IIR filter

The coefficients of the filter depicted in [Figure 58](#) 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 (*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 67. 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 68. GDFLIB_FilterIIR2_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_IIR2_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_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_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 (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.

2.11.3.1 Function GDFLIB_FilterIIR2Init_F32

Declaration

```
void GDFLIB_FilterIIR2Init_F32(GDFLIB_FILTER_IIR2_T_F32 *const pParam);
```

Arguments

Table 69. 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);
}
```

2.11.3.2 Function GDFLIB_FilterIIR2Init_F16

Declaration

```
void GDFLIB_FilterIIR2Init_F16(GDFLIB_FILTER_IIR2_T_F16 *const pParam);
```

Arguments

Table 70. 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);
}
```

2.12 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 λ is a smoothing factor, $0 < \lambda < 1$. Values of λ close to one lead to less smoothing and give greater weight to recent changes in the input data,

while values of λ closer to zero cause greater smoothing and the filter is less responsive to recent changes.

There is no direct equivalence between the smoothing factor λ 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 λ 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 (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 71. 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);

    // ##### 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
*uParam);
```

Arguments

Table 72. 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_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 (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.

2.12.3.1 Function GDFLIB_FilterMAInit_F32

Declaration

```
void GDFLIB_FilterMAInit_F32(GDFLIB\_FILTER\_MA\_T\_F32 *pParam);
```

Arguments

Table 73. GDFLIB_FilterMAInit_F32 arguments

Type	Name	Direction	Description
GDFLIB_FILTER_MA_T_F32 *	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);
}
```

2.12.3.2 Function GDFLIB_FilterMAInit_F16

Declaration

```
void GDFLIB_FilterMAInit_F16(GDFLIB_FILTER_MA_T_F16 *pParam);
```

Arguments

Table 74. 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 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 (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.

2.12.4.1 Function GDFLIB_FilterMASetState_F32

Declaration

```
void GDFLIB_FilterMASetState_F32(tFrac32 f32FilterMAOut,  
GDFLIB\_FILTER\_MA\_T\_F32 *pParam);
```

Arguments

Table 75. GDFLIB_FilterMASetState_F32 arguments

Type	Name	Direction	Description
tFrac32	f32FilterMAOut	input	Required output of the FilterMA.
GDFLIB_FILTER_MA_T_F32 *	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;
    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);
}
```

2.12.4.2 Function GDFLIB_FilterMASetState_F16

Declaration

```
void GDFLIB_FilterMASetState_F16(tFrac16 f16FilterMAOut,  
GDFLIB\_FILTER\_MA\_T\_F16 *pParam);
```

Arguments

Table 76. 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.ul6NSamples = 5;
    tFrac16 f16FilterMAOut;

    // Initialize the GDFLIB_FilterMA state variable to a predefined value
    // Warning: The ul6NSamples 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);
}
```

2.13 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.

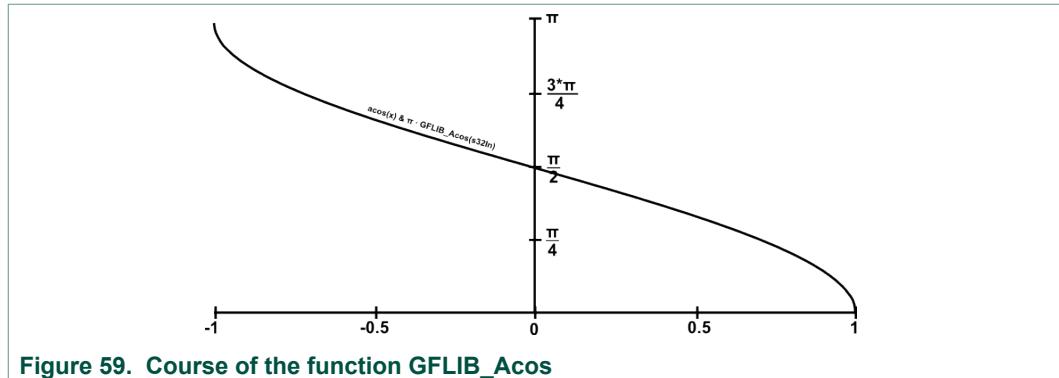


Figure 59. Course of the function GFLIB_Acos

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 77. GFLIB_Acos_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_ACOS_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_ACOS_DEFAULT_F32 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:

$$\frac{\sin^{-1}(fIn)}{\pi} = (a_0 + a_1 \cdot |fIn| + a_2 \cdot |fIn|^2 + a_3 \cdot |fIn|^3 + a_4 \cdot |fIn|^4) \cdot sign(fIn)$$

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 78](#).

Table 78. Integer polynomial coefficients for each interval

Interval	a_0	a_1	a_2	a_3	a_4
$<0, 1/2)$	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 (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 79. [GFLIB_Acos_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_ACOS_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_ACOS_DEFAULT_F16 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 80](#).

Table 80. 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

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:

$$fOut = \frac{\cos^{-1}(fIn)}{\pi} = \frac{1}{2} - \frac{\sin^{-1}(fIn)}{\pi}$$

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 (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) ;
}

```

2.14 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.

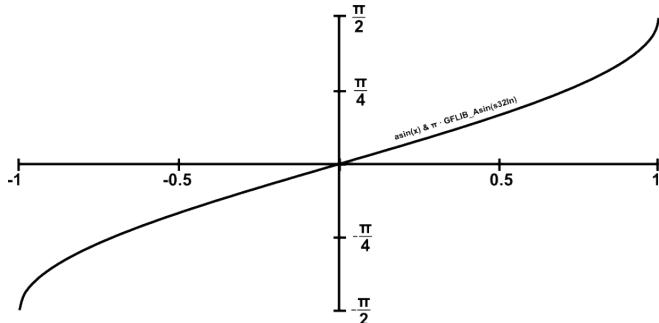


Figure 60. Course of the function GFLIB_Asin

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 81.** *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 82](#).

Table 82. *Integer polynomial coefficients for each interval*

Interval	a ₀	a ₁	a ₂	a ₃	a ₄
<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 (HardFault).

Code Example

```
#include "gflib.h"

tFrac32 f32Input;
tFrac32 f32Angle;

void main(void)
{
    // input f32Input = (1-(2^-31))
    f32Input = (tFrac32)(0x7FFFFFFF);

    // output should be 0x3FFFF5A7 = 0.4999987665
    f32Angle = GFLIB_Asin_F32(f32Input, GFLIB_ASIN_DEFAULT_F32);

    // output should be 0x3FFFF5A7 = 0.4999987665
    f32Angle = GFLIB_Asin(f32Input, GFLIB_ASIN_DEFAULT_F32, F32);
```

```

// ######
// Available only if 32-bit fractional implementation selected
// as default
// #####
// output should be 0x3FFFF5A7 = 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 83. 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:

$$fOut = \frac{\sin^{-1}(fIn)}{\pi} = (a_0 + a_1 \cdot |fIn| + a_2 \cdot |fIn|^2 + a_3 \cdot |fIn|^3 + a_4 \cdot |fIn|^4) \cdot \text{sign}(fIn)$$

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 84](#).

Table 84. 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 (HardFault).

Code Example

```
#include "gplib.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);

    // ##### Available only if 16-bit fractional implementation selected
    // as default
    // #####
    // output should be 0x3FAE = 0.4974975
    f16Angle = GFLIB_Asin(f16Input);
}

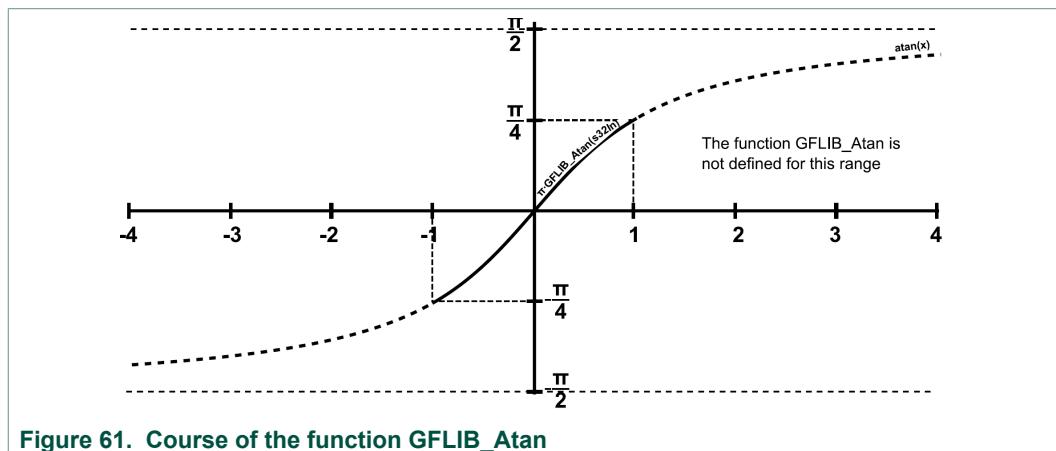
```

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 61](#).



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 85. 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>&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 π 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_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 86](#).

Table 86. Integer minimax polynomial coefficients for each interval

Interval	a_0	a_1	a_2
<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 (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 87. GFLIB_Atan_F16 arguments

Type	Name	Direction	Description
tFrac16	f16In	input	Input argument is a 16-bit number between [-1, 1].
const GFLIB_ATAN_T_F16 *const	pParam	input	Pointer to an array of minimax approximation coefficients. In case the default approximation coefficients are used, the &pParam must be replaced with GFLIB_ATAN_DEFAULT_F16 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 compute 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 π 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_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 88](#).

Table 88. 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 (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);
}

```

2.16 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 89. GFLIB_AtanYX_F32 arguments

Type	Name	Direction	Description
<a>tFrac32	f32InY	input	The ordinate of the input vector (y coordinate).
<a>tFrac32	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 [- π , π). 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 [- π , π].

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 90. `GFLIB_AtanYX_F16` arguments

Type	Name	Direction	Description
<code>tFrac16</code>	<code>f16InY</code>	<code>input</code>	The ordinate of the input vector (y coordinate).
<code>tFrac16</code>	<code>f16InX</code>	<code>input</code>	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 [- π , π). 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 [- π , π].

Note: The function calls the [GFLIB_Atan_F16](#) function. The computed value is within the range of [-1, 1).

Code Example

```
#include "gplib.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);
}
```

2.17 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:

$$y = \sin(\theta)$$

$$x = \sin(\theta + \Delta\theta)$$

Equation GFLIB_AtanYXShifted_Eq1

where:

- x, y are respectively, the InX and InY arguments
- θ 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 θ_{Offset} is added to the computed angle θ . The angle offset is an additional parameter, which can be used to set the zero of the θ axis. If θ_{Offset} is zero, then the angle computed by the function will be exactly θ .

Note: The input/output pointers must contain valid addresses, otherwise a fault may occur (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 91. `GFLIB_AtanYXShifted_F32` arguments

Type	Name	Direction	Description
<code>tFrac32</code>	<code>f32InY</code>	<code>input</code>	The value of the first signal, assumed to be $\sin(\theta)$.
<code>tFrac32</code>	<code>f32InX</code>	<code>input</code>	The value of the second signal, assumed to be $\sin(\theta + \Delta\theta)$.
const <code>GFLIB_ATANYXSHIFTED_T_F32</code> *	<code>pParam</code>	<code>input, output</code>	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 θ_{Offset} and the phase difference θ . 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 θ_{Offset} , the $\Delta\theta$ shall be known from the input sinusoidal signals, the θ_{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:

$$b = \frac{s}{2\cos(\frac{\Delta\theta}{2})} (y + x)$$

$$a = \frac{s}{2\sin(\frac{\Delta\theta}{2})} (x - y)$$

$$\theta = \tan^{-1}\left(\frac{y}{x}\right) - \left(\frac{\Delta\theta}{2} - \theta_{\text{offset}}\right)$$

Equation `GFLIB_AtanYXShifted_F32_Eq1`

where:

- x, y are respectively, the `f32InX`, and `f32InY`
- θ 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
- θ_{Offset} is the additional phase shift, the computed angle will be $\theta + \theta_{\text{Offset}}$

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:

$$\frac{S}{2\cos(\frac{\Delta\theta}{2})} = C_y = K_y \cdot 2^{N_y}$$

$$\frac{S}{2\sin(\frac{\Delta\theta}{2})} = C_x = K_x \cdot 2^{N_x}$$

$$\theta_{\text{adj}} = \frac{\Delta\theta}{2} - \theta_{\text{offset}}$$

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
- θ_{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 θ_{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 92. `GFLIB_AtanYXShifted_F16` arguments

Type	Name	Direction	Description
<code>tFrac16</code>	<code>f16InY</code>	<code>input</code>	The value of the first signal, assumed to be $\sin(\theta)$.
<code>tFrac16</code>	<code>f16InX</code>	<code>input</code>	The value of the second signal, assumed to be $\sin(\theta + \Delta\theta)$.
const <code>GFLIB_ATANYXSHIFTED_T_F16 *</code>	<code>pParam</code>	<code>input, output</code>	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 θ_{Offset} and the phase difference θ . 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 θ_{Offset} , the $\Delta\theta$ shall be known from the input sinusoidal signals, the θ_{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:

$$b = \frac{s}{2\cos(\frac{\Delta\theta}{2})} (y + x)$$

$$a = \frac{s}{2\sin(\frac{\Delta\theta}{2})} (x - y)$$

$$\theta = \tan^{-1}\left(\frac{y}{x}\right) - \left(\frac{\Delta\theta}{2} - \theta_{\text{offset}}\right)$$

Equation GFLIB_AtanYXShifted_F16_Eq1

where:

- x, y are respectively, the `f16InX`, and `f16InY`

- θ 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
- θ_{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
- θ_{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 θ_{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 "gflib.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 = 0.055555555555

    Param.f16Ky = FRAC16(0.60789036201452440);
    Param.f16Kx = FRAC16(0.87905201358520957);
    Param.s16Ny = 0;
    Param.s16Nx = 0;
    Param.f16ThetaAdj = FRAC16(0.055555555555);

    // 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);  
}
```

2.18 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:

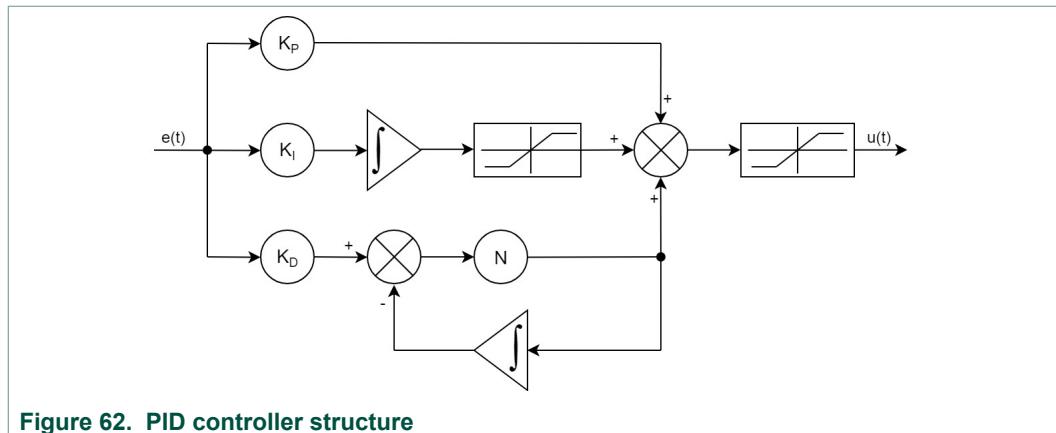
$$u(t) = K_P \cdot e(t) + K_I \int_0^t e(t) dt + K_D \frac{de(t)}{dt}$$

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 62](#).

**Figure 62.** PID controller structure

This system can be described using the Laplace transformation as follows:

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

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 an exception may occur (Data TLB Error, Data Storage, Alignment, Machine Check).

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 93. `GFLIB_ControllerPIDpAW_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_PID_P_AW_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_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:

$$u_{If}(k) = K_{D_SC} \cdot [e_f(k) - e_f(k-1)] + e^{T_s N} \cdot u_{Df}(k-1) \quad \text{where } K_{D_SC} = \frac{K_D}{T_s} \cdot (1 - e^{T_s N}) \cdot \frac{E^{MAX}}{U^{MAX}}$$

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 94. 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).
GFLIB_CONTROLLER_PID_P_AW_T_F16 *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:

$$u_{Df}(k) = K_{D_{SC}} \cdot [e_f(k) - e_f(k-1)] + e^{T_s N} \cdot u_{Df}(k-1) \quad \text{where} \quad K_{D_{SC}} = \frac{K_D}{T_s} \cdot (1 - e^{T_s N}) \cdot \frac{E^{MAX}}{U^{MAX}}$$

Equation [GFLIB_ControllerPIDpAW_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:

$$f16PropGain = K_{P_{SC}} \cdot 2^{s16PropGainShift} \quad \text{where} \quad s16PropGainShift = \lceil \log_2 K_{P_{SC}} \rceil$$

$$f16IntegGain = K_{I_{SC}} \cdot 2^{s16IntegGainShift} \quad \text{where} \quad s16IntegGainShift = \lceil \log_2 K_{I_{SC}} \rceil$$

$$f16DerivGain = K_{D_{SC}} \cdot 2^{s16DerivGainShift} \quad \text{where} \quad s16DerivGainShift = \lceil \log_2 K_{D_{SC}} \rceil$$

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.

$$f16FiltCoef = e^{T_s N}$$

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 "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);
}
```

2.18.3 Function GFLIB_ControllerPIDpAWInit

Description

This function clears the GFLIB_ControllerPIDpAW state variables.

Note: The input/output pointer must contain valid address, otherwise a fault may occur (HardFault).

Re-entrancy

The function is re-entrant for a different pCtrl.

2.18.3.1 Function GFLIB_ControllerPIDpAWInit_F32**Declaration**

```
void
GFLIB_ControllerPIDpAWInit_F32 (GFLIB_CONTROLLER_PID_P_AW_T_F32
*const pParam);
```

Arguments**Table 95. GFLIB_ControllerPIDpAWInit_F32 arguments**

Type	Name	Direction	Description
GFLIB_CONTROLLER_PID_P_AW_T_F32 *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);
}
```

2.18.3.2 Function GFLIB_ControllerPIDpAWInit_F16

Declaration

```
void
GFLIB_ControllerPIDpAWInit_F16(GFLIB\_CONTROLLER\_PID\_P\_AW\_T\_F16
*const pParam);
```

Arguments

Table 96. `GFLIB_ControllerPIDpAWInit_F16` arguments

Type	Name	Direction	Description
<code>GFLIB_CONTROLLER_PID_P_AW_T_F16 *const</code>	<code>pParam</code>	<code>input, output</code>	Pointer to the structure with <code>GFLIB_ControllerPIDpAW</code> state.

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);
}

```

2.18.4 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 valid address, otherwise a fault may occur (HardFault).

Re-entrancy

The function is re-entrant for a different pCtrl.

2.18.4.1 Function GFLIB_ControllerPIDpAWSetState_F32

Declaration

```
void GFLIB_ControllerPIDpAWSetState_F32(tFrac32
f32ControllerPIDpAWOut, GFLIB\_CONTROLLER\_PID\_P\_AW\_T\_F32 *const
pParam);
```

Arguments

Table 97. GFLIB_ControllerPIDpAWSetState_F32 arguments

Type	Name	Direction	Description
tFrac32	f32ControllerPIDpAWOut	input	Required output of the GFLIB_ControllerPIDpAW.
GFLIB_CONTROLLER_PID_P_AW_T_F32 *const	pParam	input, output	Pointer to the structure with GFLIB_ControllerPIDpAW state.

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);
}

```

2.18.4.2 Function GFLIB_ControllerPIDpAWSetState_F16

Declaration

```

void GFLIB_ControllerPIDpAWSetState_F16 (tFrac16
                                         f16ControllerPIDpAWOut, GFLIB_CONTROLLER_PID_P_AW_T_F16 *const
                                         pParam);

```

Arguments

Table 98. GFLIB_ControllerPIDpAWSetState_F16 arguments

Type	Name	Direction	Description
<u>tFrac16</u>	f16ControllerPIDpAWOut	input	Required output of the GFLIB_ControllerPIDpAW.
<u>GFLIB_CONTROLLER_PID_P_AW_T_F16</u> *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);
}
```

2.19 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 an exception may occur (Data TLB Error, Data Storage, Alignment, Machine Check).

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 99. `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 "gplib.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 100. 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_ControllerPIpInit

Description

This function clears the GFLIB_ControllerPIp state variables.

Note: The input/output pointer must contain valid address, otherwise a fault may occur (HardFault).

Re-entrancy

The function is re-entrant for a different pCtrl.

2.19.3.1 Function GFLIB_ControllerPIpInit_F32

Declaration

```
void GFLIB_ControllerPIpInit_F32(GFLIB_CONTROLLER_PI_P_T_F32
*const pParam);
```

Arguments

Table 101. GFLIB_ControllerPIpInit_F32 arguments

Type	Name	Direction	Description
GFLIB_CONTROLLER_PI_P_T_F32 *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);
}

```

2.19.3.2 Function GFLIB_ControllerPIpInit_F16

Declaration

```
void GFLIB_ControllerPIpInit_F16(GFLIB\_CONTROLLER\_PI\_P\_T\_F16
*const pParam);
```

Arguments

Table 102. GFLIB_ControllerPIpInit_F16 arguments

Type	Name	Direction	Description
GFLIB_CONTROLLER_PI_P_T_F16 *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  
    // (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 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 valid address, otherwise a fault may occur (HardFault).

Re-entrancy

The function is re-entrant for a different pCtrl.

2.19.4.1 Function [GFLIB_ControllerPIpSetState_F32](#)

Declaration

```
void GFLIB_ControllerPIpSetState_F32(tFrac32 f32ControllerPIpOut,  
GFLIB\_CONTROLLER\_PI\_P\_T\_F32 *const pParam);
```

Arguments

Table 103. `GFLIB_ControllerPIpSetState_F32` arguments

Type	Name	Direction	Description
<code>tFrac32</code>	<code>f32ControllerPIpOut</code>	<code>input</code>	Required output of the <code>GFLIB_ControllerPIp</code> .
<code>GFLIB_CONTROLLER_PI_P_T_F32 *const</code>	<code>pParam</code>	<code>input, output</code>	Pointer to the structure with <code>GFLIB_ControllerPIp</code> state.

Code Example

```
#include "gplib.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_ControllerPIpSetState_F16

Declaration

```
void GFLIB_ControllerPIpSetState_F16(tFrac16 f16ControllerPIpOut,
GFLIB_CONTROLLER_PI_P_T_F16 *const pParam);
```

Arguments

Table 104. GFLIB_ControllerPIpSetState_F16 arguments

Type	Name	Direction	Description
<u>tFrac16</u>	f16ControllerPIpOut	input	Required output of the GFLIB_ControllerPIp.
<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);
}
```

2.20 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 an exception may occur (Data TLB Error, Data Storage, Alignment, Machine Check).

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 105. `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:

$$u_{P_f}(k) = e_f(k) \cdot K_{P_SC} \text{ where } K_{P_SC} = K_P \cdot \frac{E^{MAX}}{U^{MAX}}$$

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:

$$u_{I_f}(k) = u_{I_f}(k-1) + K_{I_SC} \cdot e_f(k) + K_{I_SC} \cdot e_f(k-1) \quad \text{where } K_{I_SC} = \frac{K_I T_s}{2} \cdot \frac{E^{MAX}}{U^{MAX}}$$

Equation GFLIB_ControllerPIpAW_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:

$$u_f(k) = e_f(k) \cdot K_{P_SC} + u_{I_f}(k-1) + K_{I_SC} \cdot e_f(k) + K_{I_SC} \cdot e_f(k-1)$$

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 "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
    // (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 106. GFLIB_ControllerPIpAW_F16 arguments

Type	Name	Direction	Description
tFrac16	f16InErr	input	Input error signal to the controller is a 16-bit number normalized between [-1, 1].
GFLIB_CONTROLLER_PIAW_P_T_F16 *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_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:

$$u_{If}(k) = u_{If}(k-1) + K_{I_SC} \cdot e_f(k) + K_{I_SC} \cdot e_f(k-1) \text{ where } K_{I_SC} = \frac{K_I T_s}{2} \cdot \frac{E^{MAX}}{U^{MAX}}$$

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 "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);
}

```

2.20.3 Function GFLIB_ControllerPIpAWInit

Description

This function clears the GFLIB_ControllerPIpAW state variables.

Note: The input/output pointer must contain valid address, otherwise a fault may occur (HardFault).

Re-entrancy

The function is re-entrant for a different pCtrl.

2.20.3.1 Function GFLIB_ControllerPIpAWInit_F32

Declaration

```
void GFLIB_ControllerPIpAWInit_F32(GFLIB_CONTROLLER_PIAW_P_T_F32
*const pParam);
```

Arguments

Table 107. GFLIB_ControllerPIpAWInit_F32 arguments

Type	Name	Direction	Description
GFLIB_CONTROLLER_PIAW_P_T_F32 *const	pParam	input, output	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);
}
```

2.20.3.2 Function GFLIB_ControllerPIpAWInit_F16

Declaration

```
void GFLIB_ControllerPIpAWInit_F16(GFLIB\_CONTROLLER\_PIAW\_P\_T\_F16
*const pParam);
```

Arguments

Table 108. GFLIB_ControllerPIpAWInit_F16 arguments

Type	Name	Direction	Description
GFLIB_CONTROLLER_PIAW_P_T_F16 *const	pParam	input, output	Pointer to the structure with GFLIB_ControllerPIpAW 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 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 valid address, otherwise a fault may occur (HardFault).

Re-entrancy

The function is re-entrant for a different pCtrl.

2.20.4.1 Function [GFLIB_ControllerPIpAWSetState_F32](#)

Declaration

```
void GFLIB_ControllerPIpAWSetState_F32(tFrac32
f32ControllerPIpAWOut, GFLIB\_CONTROLLER\_PIAW\_P\_T\_F32 *const
pParam);
```

Arguments

Table 109. GFLIB_ControllerPIpAWSetState_F32 arguments

Type	Name	Direction	Description
tFrac32	f32ControllerPIpAWOut	input	Required output of the GFLIB_ControllerPIpAW.
GFLIB_CONTROLLER_PIAW_P_T_F32 *const	pParam	input, output	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);
}

```

2.20.4.2 Function GFLIB_ControllerPIpAWSetState_F16

Declaration

```
void GFLIB_ControllerPIpAWSetState_F16(tFrac16
f16ControllerPIpAWOut, GFLIB_CONTROLLER_PIAW_P_T_F16 *const
pParam);
```

Arguments

Table 110. GFLIB_ControllerPIpAWSetState_F16 arguments

Type	Name	Direction	Description
tFrac16	f16ControllerPIpAWOut	input	Required output of the GFLIB_ControllerPIpAW.
<u>GFLIB_CONTROLLER_PIAW_P_T_F16</u> *const	pParam	input, output	Pointer to the structure with GFLIB_ControllerPIpAW 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);
}
```

2.21 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 111](#).

Table 111. 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 an exception may occur (Data TLB Error, Data Storage, Alignment, Machine Check).

Re-entrancy

The function is re-entrant.

2.21.1 Function `GFLIB_ControllerPIr_F32`

Declaration

```
tFrac32 GFLIB_ControllerPIr_F32 (tFrac32 f32InErr,
GFLIB_CONTROLLER_PI_R_T_F32 *const pParam);
```

Arguments

Table 112. `GFLIB_ControllerPIr_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_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_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_ControllerPIr_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_ControllerPIr_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 "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);
}
```

2.21.2 Function `GFLIB_ControllerPlr_F16`

Declaration

```
tFrac16 GFLIB_ControllerPlr_F16(tFrac16 f16InErr,
GFLIB_CONTROLLER_PI_R_T_F16 *const pParam);
```

Arguments

Table 113. `GFLIB_ControllerPlr_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_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_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_ControllerPlr_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_ControllerPlr_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 "gplib.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_ControllerPIrInit`

Description

This function clears the `GFLIB_ControllerPIr` state variables.

Note: The input/output pointer must contain valid address, otherwise a fault may occur (HardFault).

Re-entrancy

The function is re-entrant for a different `pCtrl`.

2.21.3.1 Function `GFLIB_ControllerPIrInit_F32`

Declaration

```
void GFLIB_ControllerPIrInit_F32(GFLIB\_CONTROLLER\_PI\_R\_T\_F32
*const pParam);
```

Arguments

Table 114. `GFLIB_ControllerPIrInit_F32` arguments

Type	Name	Direction	Description
GFLIB_CONTROLLER_PI_R_T_F32 *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.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);
}

```

2.21.3.2 Function GFLIB_ControllerPIrInit_F16

Declaration

```
void GFLIB_ControllerPIrInit_F16(GFLIB\_CONTROLLER\_PI\_R\_T\_F16
*const pParam);
```

Arguments

Table 115. GFLIB_ControllerPIrInit_F16 arguments

Type	Name	Direction	Description
GFLIB_CONTROLLER_PI_R_T_F16 *const	pParam	input, output	Pointer to the structure with GFLIB_ControllerPIr state.

Code Example

```
#include "gplib.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 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 valid address, otherwise a fault may occur (HardFault).

Re-entrancy

The function is re-entrant for a different pCtrl.

2.21.4.1 Function GFLIB_ControllerPIrSetState_F32

Declaration

```
void GFLIB_ControllerPIrSetState_F32(tFrac32 f32ControllerPIrOut,
GFLIB\_CONTROLLER\_PI\_R\_T\_F32 *const pParam);
```

Arguments

Table 116. GFLIB_ControllerPIrSetState_F32 arguments

Type	Name	Direction	Description
tFrac32	f32ControllerPIrOut	input	Required output of the GFLIB_ControllerPIr.
GFLIB_CONTROLLER_PI_R_T_F32 *const	pParam	input, output	Pointer to the structure with GFLIB_ControllerPIr state.

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);
}
```

2.21.4.2 Function `GFLIB_ControllerPIrSetState_F16`

Declaration

```
void GFLIB_ControllerPIrSetState_F16(tFrac16 f16ControllerPIrOut,  
GFLIB_CONTROLLER_PI_R_T_F16 *const pParam);
```

Arguments

Table 117. `GFLIB_ControllerPIrSetState_F16` arguments

Type	Name	Direction	Description
<u>tFrac16</u>	f16ControllerPIrOut	input	Required output of the <code>GFLIB_ControllerPIr</code> .
<u>GFLIB_CONTROLLER_PI_R_T_F16</u> *const	pParam	input, output	Pointer to the structure with <code>GFLIB_ControllerPIr</code> 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.u16NShift = 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);
}

```

2.22 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 118](#).

Table 118. 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 an exception may occur (Data TLB Error, Data Storage, Alignment, Machine Check).

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 119. 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 *const</code>	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.

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_ControllerPIrAWSetState_F32(f32ControllerPIrAWOut, &trMyPI);

// Alternative 2: API call with implementation parameter
// (only one alternative shall be used).
GFLIB_ControllerPIrAWSsetState(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_ControllerPIrAWSsetState(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 120. GFLIB_ControllerPIrAW_F16 arguments

Type	Name	Direction	Description
tFrac16	f16InErr	input	Input error signal to the controller is a 16-bit number normalized between [-1, 1).

Type	Name	Direction	Description
GFLIB_CONTROLLER_PIAW_R_T_F16 *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).

$$u16NShift = \max\left(\left\lceil \frac{\log(\text{abs}(CC1_f))}{\log(2)} \right\rceil, \left\lceil \frac{\log(\text{abs}(CC2_f))}{\log(2)} \right\rceil\right)$$

Equation GFLIB_ControllerPIrAW_F16_Eq6

The final, scaled, fractional equation of the recurrent PI controller on a 16-bit fixed point platform is therefore implemented as follows:

$$u_f(k) \cdot 2^{u16NShift} = u_f(k-1) \cdot 2^{u16NShift} + e_f(k) \cdot f16CC1sc + e_f(k-1) \cdot f16CC2sc$$

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
- $u16NShift$ - 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 UpperLimit , LowerLimit . This is due to either the bounded power of the actuator or to the physical constraints of the plant.

$$u(k) = \begin{cases} f16UpperLimit & \text{if } u_f(k) \geq f16UpperLimit \\ u_f(k) & \text{if } f16LowerLimit < u_f(k) < f16UpperLimit \\ f16LowerLimit & \text{if } u_f(k) \leq f16LowerLimit \end{cases}$$

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.

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_ControllerPIrAWS.setState_F16(f16ControllerPIrAWOut, &trMyPI);

    // Alternative 2: API call with implementation parameter
    // (only one alternative shall be used).
    GFLIB_ControllerPIrAWS.setState(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_ControllerPIrAWS.setState(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_ControllerPIrAWInit

Description

This function clears the GFLIB_ControllerPIrAW state variables.

Note: The input/output pointer must contain valid address, otherwise a fault may occur (HardFault).

Re-entrancy

The function is re-entrant for a different pCtrl.

2.22.3.1 Function GFLIB_ControllerPIrAWInit_F32

Declaration

```
void GFLIB_ControllerPIrAWInit_F32(GFLIB\_CONTROLLER\_PIAW\_R\_T\_F32
*const pParam);
```

Arguments

Table 121. GFLIB_ControllerPIrAWInit_F32 arguments

Type	Name	Direction	Description
GFLIB_CONTROLLER_PIAW_R_T_F32 *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);
}
```

2.22.3.2 Function `GFLIB_ControllerPIrAWInit_F16`

Declaration

```
void GFLIB_ControllerPIrAWInit_F16(GFLIB\_CONTROLLER\_PIAW\_R\_T\_F16  
*const pParam);
```

Arguments

Table 122. `GFLIB_ControllerPIrAWInit_F16` arguments

Type	Name	Direction	Description
GFLIB_CONTROLLER_PIAW_R_T_F16 *const	pParam	input, output	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_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);
}

```

2.22.4 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 valid address, otherwise a fault may occur (HardFault).

Re-entrancy

The function is re-entrant for a different pCtrl.

2.22.4.1 Function **GFLIB_ControllerPIrAWSetState_F32**

Declaration

```
void GFLIB_ControllerPIrAWSetState_F32(tFrac32
f32ControllerPIrAWOut, GFLIB_CONTROLLER_PIAW_R_T_F32 *const
pParam);
```

Arguments

Table 123. **GFLIB_ControllerPIrAWSetState_F32** arguments

Type	Name	Direction	Description
tFrac32	f32ControllerPIrAWOut	input	Required output of the GFLIB_ControllerPIrAW.

Type	Name	Direction	Description
<code>GFLIB_CONTROLLER_PIAW_R_T_F32 *const</code>	pParam	<code>input, output</code>	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.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_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_ControllerPIrAWSetState_F16

Declaration

```

void GFLIB_ControllerPIrAWSetState_F16(tFrac16
f16ControllerPIrAWOut, GFLIB_CONTROLLER_PIAW_R_T_F16 *const
pParam);

```

Arguments

Table 124. GFLIB_ControllerPIrAWSetState_F16 arguments

Type	Name	Direction	Description
<u>tFrac16</u>	f16ControllerPIrAWOut	input	Required output of the GFLIB_ControllerPIrAW.
<u>GFLIB_CONTROLLER_PIAW_R_T_F16</u> *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);

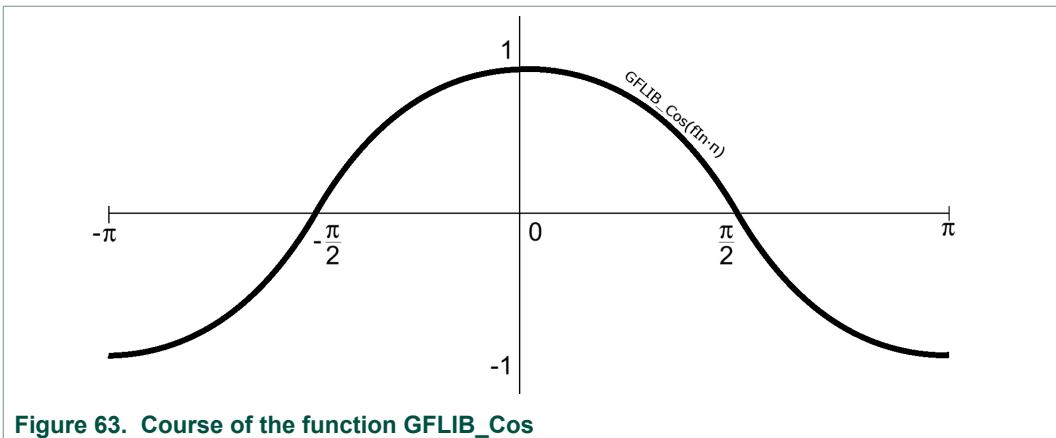
    // Alternative 3: API call with global configuration of implementation
// (only one alternative shall be used). This alternative is available
// only if 16-bit fractional implementation is selected as default.
f16Output = GFLIB_ControllerPIrAW(f16InErr, &trMyPI);
}
```

2.23 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 125. 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 GFLIB_COS_DEFAULT_F32 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:

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

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 (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 126. GFLIB_Cos_F16 arguments

Type	Name	Direction	Description
tFrac16	f16In	input	Input argument is a 16-bit number that contains an angle in radians from interval [-π, π) normalized between [-1, 1).
const GFLIB_COS_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_COS_DEFAULT_F16 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 (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);
}
```

2.24 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).

$$OutState(k) = \begin{cases} OutValOn & \text{if } In \geq HystOn \\ OutValOff & \text{if } In \leq HystOff \\ OutState(k-1) & \text{if otherwise} \end{cases}$$

Equation GFLIB_Hyst_Eq1

A graphical description of GFLIB_Hyst functionality is shown in [Figure 64](#).

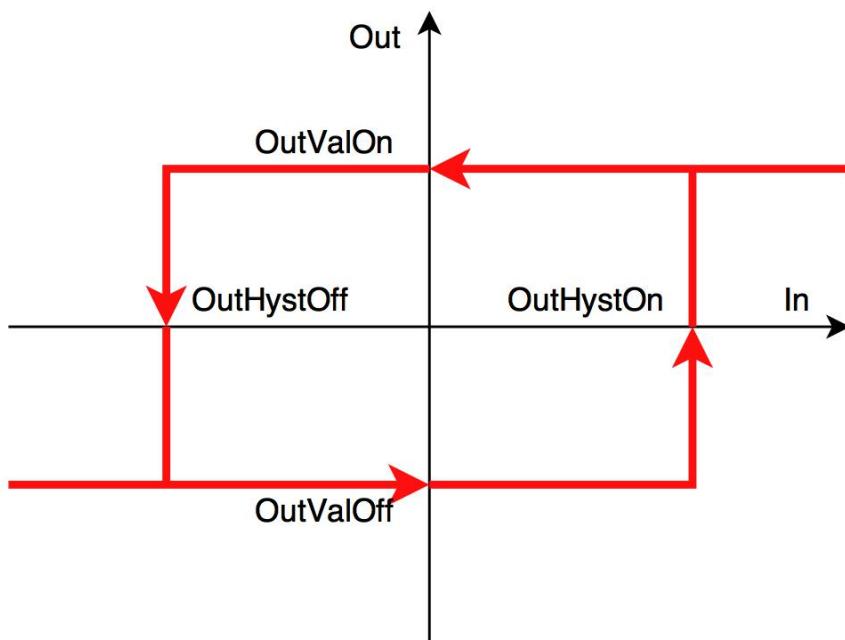


Figure 64. Hysteresis function

Note: The input pointer must contain a valid address otherwise a fault may occur (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 127. GFLIB_Hyst_F32 arguments**

Type	Name	Direction	Description
tFrac32	f32In	input	Input signal in the form of a 32-bit fixed point number, normalized between [-1, 1).
GFLIB_HYST_T_F32 *const	pParam	input, output	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 "gflib.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 128. GFLIB_Hyst_F16 arguments

Type	Name	Direction	Description
tFrac16	f16In	input	Input signal in the form of a 16-bit fixed point number, normalized between [-1, 1).
GFLIB_HYST_T_F16 *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);
}

```

2.25 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 (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 129. `GFLIB_IntegratorTR_F32` arguments

Type	Name	Direction	Description
<code>tFrac32</code>	<code>f32In</code>	<code>input</code>	Input argument to be integrated.
<code>GFLIB_INTEGRATOR_TR_T_F32 *const</code>	<code>pParam</code>	<code>input, output</code>	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:

$$u16NShift = \text{ceil}\left(\frac{\log(\text{abs}(C_1))}{\log(2)}\right)$$

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);

    // 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 130. `GFLIB_IntegratorTR_F16` arguments

Type	Name	Direction	Description
<code>tFrac16</code>	<code>f16In</code>	<code>input</code>	Input argument to be integrated.
<code>GFLIB_INTEGRATOR_TR_T_F16 *const</code>	<code>pParam</code>	<code>input, output</code>	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);

    // 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_IntegratorTRSetState

Description

This function initializes the GFLIB_IntegratorTR state variables to achieve the required output values.

Note: The input/output pointer must contain valid address, otherwise a fault may occur (HardFault).

Re-entrancy

The function is re-entrant for a different pCtrl.

2.25.3.1 Function GFLIB_IntegratorTRSetState_F32**Declaration**

```
void GFLIB_IntegratorTRSetState_F32(tFrac32 f32IntegratorTROut,  
GFLIB\_INTEGRATOR\_TR\_T\_F32 *const pParam);
```

Arguments**Table 131. 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 "gplib.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
}

```

2.25.3.2 Function GFLIB_IntegratorTRSetState_F16

Declaration

```
void GFLIB_IntegratorTRSetState_F16(tFrac16 f16IntegratorTROut,
GFLIB\_INTEGRATOR\_TR\_T\_F16 *const pParam);
```

Arguments

Table 132. GFLIB_IntegratorTRSetState_F16 arguments

Type	Name	Direction	Description
tFrac16	f16IntegratorTROut	input	Required output of the GFLIB_IntegratorTR.
GFLIB_INTEGRATOR_TR_T_F16 *const	pParam	input, output	Pointer to the structure with GFLIB_IntegratorTR 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.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
}

```

2.26 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 an exception may occur (Data TLB Error, Data Storage, Alignment, Machine Check).

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 133. 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 134. GFLIB_Limit_F16 arguments**

Type	Name	Direction	Description
<code>tFrac16</code>	<code>f16In</code>	<code>input</code>	Input value.
<code>const GFLIB_LIMIT_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 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 "gplib.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);
}
```

2.27 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 an exception may occur (Data TLB Error, Data Storage, Alignment, Machine Check).

Re-entrancy

The function is re-entrant.

2.27.1 Function `GFLIB_LowerLimit_F32`

Declaration

```
tFrac32 GFLIB_LowerLimit_F32(tFrac32 f32In, const
GFLIB_LOWERLIMIT_T_F32 *const pParam);
```

Arguments

Table 135. `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);
}
```

2.27.2 Function `GFLIB_LowerLimit_F16`

Declaration

```
tFrac16 GFLIB_LowerLimit_F16(tFrac16 f16In, const
GFLIB_LOWERLIMIT_T_F16 *const pParam);
```

Arguments

Table 136. `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</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.

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 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 an exception may occur (Data TLB Error, Data Storage, Alignment, Machine Check).*

Re-entrancy

The function is re-entrant.

2.28.1 Function GFLIB_Lut1D_F32

Declaration

```
tFrac32 GFLIB_Lut1D_F32(tFrac32 f32In, const GFLIB_LUT1D_T_F32
*const pParam);
```

Arguments

Table 137. GFLIB_Lut1D_F32 arguments

Type	Name	Direction	Description
tFrac32	f32In	input	The abscissa for which 1D interpolation is performed.
const GFLIB_LUT1D_T_F32 *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 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 138. GFLIB_Lut1D example table

ordinate (y)	interpolating index	abscissa (x)
-0.5	-1	-1*(2 ⁻¹)
0.0	0	0*(2 ⁻¹)
0.25	1	1*(2 ⁻¹)
0.5	N/A	2*(2 ⁻¹)

The [Table 138](#) contains 4 interpolating points (note four rows). The interpolating interval length in this example is equal to 2⁻¹. 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 < a, b >, 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 Δx :

$$\Delta x = \frac{x}{2^{uShamOffset}} - sIntvl$$

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.

$$y_1 = pTable[sIntvl]$$

$$y_2 = pTable[sIntvl + 1]$$

$$y = y_1 + (y_2 - y_1) \cdot \Delta x$$

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 [FRAC32_MAX](#). For a better understanding, please, see the extended code example.

Code Example

```
#include "gplib.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(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 0x6666666 ~ FRAC32(0.8)
    f32Out = GFLIB_Lut1D(f32In, &trf32MyLut1D, F32);

    // available only if 32-bit fractional implementation
    // selected as default
    // output should be 0x6666666 ~ 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,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 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);
}

```

2.28.2 Function GFLIB_Lut1D_F16

Declaration

```
tFrac16 GFLIB_Lut1D_F16(tFrac16 f16In, const GFLIB\_LUT1D\_T\_F16
*const pParam);
```

Arguments

Table 139. 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 140. GFLIB_Lut1D example table

ordinate (y)	interpolating index	abscissa (x)
-0.5	-1	-1*(2 ⁻¹)
0.0	0	0*(2 ⁻¹)
0.25	1	1*(2 ⁻¹)
0.5	N/A	2*(2 ⁻¹)

The [Table 140](#) contains 4 interpolating points (note four rows). The interpolating interval length in this example is equal to 2⁻¹. 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 Δx :

$$\Delta x = \frac{x}{2^{uShamOffset}} - sIntvl$$

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.

$$\begin{aligned}y_1 &= pTable[sIntvl] \\y_2 &= pTable[sIntvl + 1] \\y &= y_1 + (y_2 - y_1) \cdot \Delta x\end{aligned}$$

Equation GFLIB_Lut1D_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 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 followinf figure.

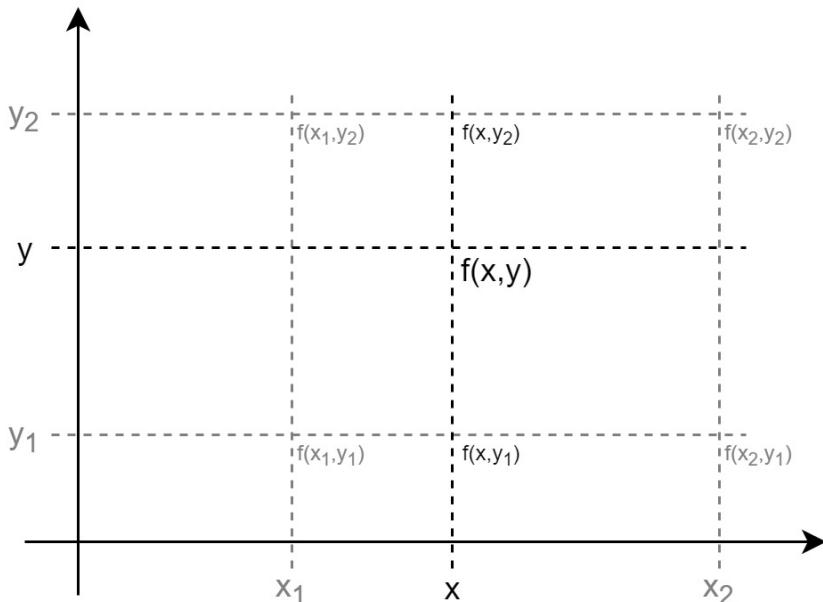


Figure 65. Interpolated area

Re-entrancy

The function is re-entrant.

2.29.1 Function `GFLIB_Lut2D_F32`

Declaration

```
tFrac32 GFLIB_Lut2D_F32(tFrac32 f32In1, tFrac32 f32In2, const
GFLIB_LUT2D_T_F32 *const pParam);
```

Arguments

Table 141. `GFLIB_Lut2D_F32` arguments

Type	Name	Direction	Description
<code>tFrac32</code>	<code>f32In1</code>	<code>input</code>	First input variable for which 2D interpolation is performed.
<code>tFrac32</code>	<code>f32In2</code>	<code>input</code>	Second input variable for which 2D interpolation is performed.
<code>const GFLIB_LUT2D_T_F32 *const</code>	<code>pParam</code>	<code>input</code>	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 142. `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:

$$\begin{aligned}f32In1_{min} &= \frac{-\text{OriginIdxX}}{2^{u16ShamOffset1}} \\f32In1_{max} &= f32In1_{min} + 1 \\f32In2_{min} &= \frac{-\text{OriginIdxY}}{2^{u16ShamOffset2}} \\f32In2_{max} &= f32In2_{min} + 1\end{aligned}$$

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 an exception may occur (Data TLB Error, Data Storage, Alignment, Machine Check).

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 0xCCCCCCC ~ FRAC32(0.1)
    f32Out = GFLIB_Lut2D_F32(f32In1, f32In2, &tr32tMyLut2D);

    // output should be 0xCCCCCCC ~ FRAC32(0.1)
    f32Out = GFLIB_Lut2D(f32In1, f32In2, &tr32tMyLut2D, F32);

    // available only if 32-bit fractional implementation
    // selected as default
    // output should be 0xCCCCCCC ~ FRAC32(0.1)
    f32Out = GFLIB_Lut2D(f32In1, f32In2, &tr32tMyLut2D);
```

```
}
```

2.29.2 Function `GFLIB_Lut2D_F16`

Declaration

```
tFrac16 GFLIB_Lut2D_F16(tFrac16 f16In1, tFrac16 f16In2, const
GFLIB_LUT2D_T_F16 *const pParam);
```

Arguments

Table 143. `GFLIB_Lut2D_F16` arguments

Type	Name	Direction	Description
<code>tFrac16</code>	<code>f16In1</code>	<code>input</code>	First input variable for which 2D interpolation is performed.
<code>tFrac16</code>	<code>f16In2</code>	<code>input</code>	Second input variable for which 2D interpolation is performed.
<code>const GFLIB_LUT2D_T_F16 *const</code>	<code>pParam</code>	<code>input</code>	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 144. `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:

$$\begin{aligned}f16In1_{min} &= \frac{-\text{OriginIdxX}}{2^{u16ShamOffset1}} \\f16In1_{max} &= f16In1_{min} + 1 \\f16In2_{min} &= \frac{-\text{OriginIdxY}}{2^{u16ShamOffset2}} \\f16In2_{max} &= f16In2_{min} + 1\end{aligned}$$

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 an exception may occur (Data TLB Error, Data Storage, Alignment, Machine Check).

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.u16ShamOffset1 = (tU16)2;
    tr16tMyLut2D.u16ShamOffset2 = (tU16)1;
    tr16tMyLut2D.pf16Table = &(pf16Table2D[10]);

    // input vector
    f16In1 = FRAC16(-0.5);
    f16In2 = FRAC16(-0.5);

    // output should be 0xFFFF ~ FRAC16(0.1)
    f16Out = GFLIB_Lut2D_F16(f16In1, f16In2, &tr16tMyLut2D);

    // output should be 0xFFFF ~ FRAC16(0.1)
    f16Out = GFLIB_Lut2D(f16In1, f16In2, &tr16tMyLut2D, F16);

    // available only if 16-bit fractional implementation
    // selected as default
    // output should be 0xFFFF ~ FRAC16(0.1)
    f16Out = GFLIB_Lut2D(f16In1, f16In2, &tr16tMyLut2D);
```

{}

2.30 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 66](#)

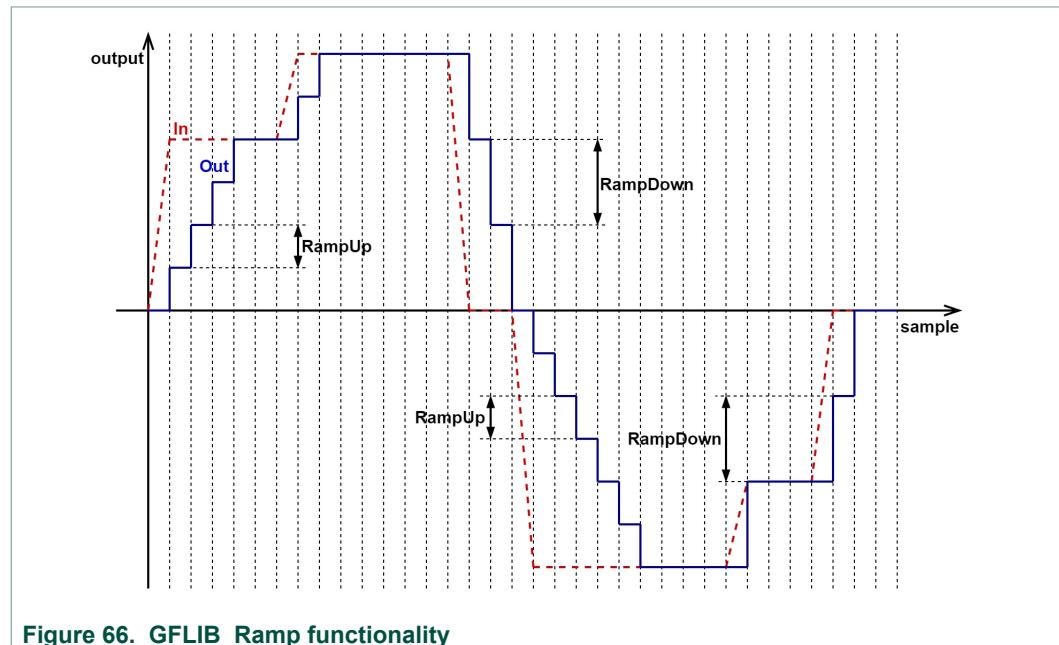


Figure 66. GFLIB_Ramp functionality

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 an exception may occur (Data TLB Error, Data Storage, Alignment, Machine Check).

Re-entrancy

The function is re-entrant.

2.30.1 Function GFLIB_Ramp_F32

Declaration

```
tFrac32 GFLIB_Ramp_F32(tFrac32 f32In, GFLIB_RAMP_T_F32 *const pParam);
```

Arguments

Table 145. GFLIB_Ramp_F32 arguments

Type	Name	Direction	Description
tFrac32	f32In	input	Input argument representing the desired output value.
GFLIB_RAMP_T_F32 *const	pParam	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);  
}
```

2.30.2 Function GFLIB_Ramp_F16

Declaration

```
tFrac16 GFLIB_Ramp_F16(tFrac16 f16In, GFLIB_RAMP_T_F16 *const  
pParam);
```

Arguments

Table 146. 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 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.

2.31.1 Function GFLIB_Sign_F32

Declaration

```
tFrac32 GFLIB_Sign_F32(tFrac32 f32In);
```

Arguments

Table 147. 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 "gplib.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);
}
```

2.31.2 Function GFLIB_Sign_F16**Declaration**

tFrac16 GFLIB_Sign_F16(tFrac16 f16In);

Arguments**Table 148. GFLIB_Sign_F16 arguments**

Type	Name	Direction	Description
tFrac16	f16In	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" (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 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.

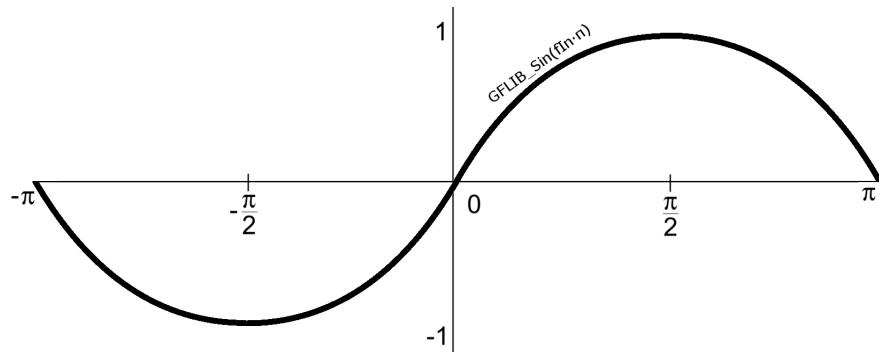


Figure 67. Course of the function GFLIB_Sin

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.

2.32.1 Function `GFLIB_Sin_F32`

Declaration

```
tFrac32 GFLIB_Sin_F32(tFrac32 f32In, const GFLIB_SIN_T_F32 *const pParam);
```

Arguments

Table 149. `GFLIB_Sin_F32` arguments

Type	Name	Direction	Description
<code>tFrac32</code>	<code>f32In</code>	<code>input</code>	Input argument is a 32-bit number that contains an angle in radians from interval $[-\pi, \pi]$ normalized between $[-1, 1]$.
<code>const GFLIB_SIN_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>&pParam</code> must be replaced with <code>GFLIB_SIN_DEFAULT_F32</code> 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 (HardFault).

Code Example

```
#include "gplib.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);
}
```

2.32.2 Function GFLIB_Sin_F16

Declaration

```
tFrac16 GFLIB_Sin_F16(tFrac16 f16In, const GFLIB_SIN_T_F16 *const pParam);
```

Arguments

Table 150. GFLIB_Sin_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 GFLIB_SIN_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_SIN_DEFAULT_F16 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 (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 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.

2.33.1 Function GFLIB_SinCos_F32

Declaration

```
void GFLIB_SinCos_F32(tFrac32 f32In, SWLIBS\_2Syst\_F32 *pOut,
const GFLIB\_SINCOS\_T\_F32 *const pParam);
```

Arguments

Table 151. GFLIB_SinCos_F32 arguments

Type	Name	Direction	Description
tFrac32	f32In	input	Input argument is a 32-bit number that contains an angle in radians from interval [- π , π) normalized between [-1, 1).
SWLIBS_2Syst_F32 *	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 GFLIB_SINCOS_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_SINCOS_DEFAULT_F32 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 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 (HardFault).

Code Example

```
#include "gplib.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);
}
```

2.33.2 Function GFLIB_SinCos_F16

Declaration

```
void GFLIB_SinCos_F16(tFrac16 f16In, SWLIBS_2Syst_F16 *pOut,
const GFLIB_SINCOS_T_F16 *const pParam);
```

Arguments

Table 152. GFLIB_SinCos_F16 arguments

Type	Name	Direction	Description
tFrac16	f16In	input	Input argument is a 16-bit number that contains an angle in radians from interval [-π, π] normalized between [-1, 1].
SWLIBS_2Syst_F16 *	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.

Type	Name	Direction	Description
const GFLIB_SINCOS_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_SINCOS_DEFAULT_F16 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 7th order Taylor polynomial approximation; the default polynomial coefficients are provided in the [GFLIB_SINCOS_DEFAULT_F16](#) structure.

Note: The input pointer must contain a valid address otherwise a fault may occur (HardFault).

Code Example

```
#include "gflib.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 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.

2.34.1 Function GFLIB_Sqrt_F32

Declaration

```
tFrac32 GFLIB_Sqrt_F32(tFrac32 f32In);
```

Arguments

Table 153. GFLIB_Sqrt_F32 arguments

Type	Name	Direction	Description
tFrac32	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.

Note: The valid input range is [0, 1). The function returns zero for negative inputs.

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);

}
```

2.34.2 Function `GFLIB_Sqrt_F16`

Declaration

```
tFrac16 GFLIB_Sqrt_F16(tFrac16 f16In);
```

Arguments

Table 154. `GFLIB_Sqrt_F16` arguments

Type	Name	Direction	Description
<code>tFrac16</code>	<code>f16In</code>	<code>input</code>	The input value.

Return

The function returns the square root of the input value. The return value is within the [0, 1) fraction range.

Note: The valid input range is [0, 1). The function returns zero for negative inputs.

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 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.

2.35.1 Function `GFLIB_Tan_F32`

Declaration

```
tFrac32 GFLIB_Tan_F32(tFrac32 f32In, const GFLIB_TAN_T_F32 *const pParam);
```

Arguments

Table 155. `GFLIB_Tan_F32` arguments

Type	Name	Direction	Description
<code>tFrac32</code>	<code>f32In</code>	<code>input</code>	Input argument is a 32-bit number that contains an angle in radians from interval $[-\pi, \pi]$ normalized between $[-1, 1]$.
<code>const GFLIB_TAN_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>&pParam</code> must be replaced with <code>GFLIB_TAN_DEFAULT_F32</code> 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.

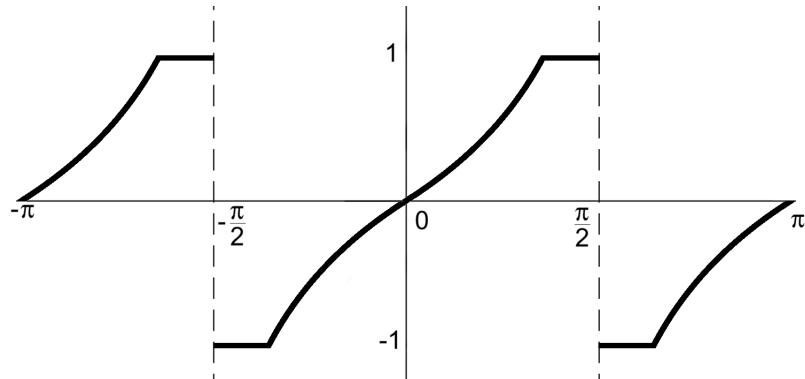


Figure 68. Course of the function `GFLIB_Tan_F32`

Note: The input pointer must contain a valid address otherwise a fault may occur (HardFault).

Code Example

```
#include "gplib.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);
}

```

2.35.2 Function GFLIB_Tan_F16

Declaration

```
tFrac16 GFLIB_Tan_F16(tFrac16 f16In, const GFLIB_TAN_T_F16 *const pParam);
```

Arguments

Table 156. 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 GFLIB_TAN_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_SIN_DEFAULT_F16 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.

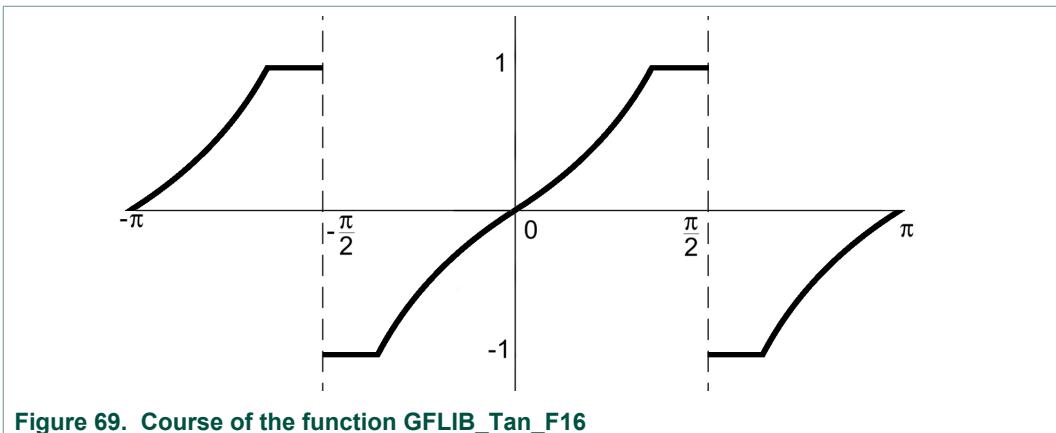


Figure 69. Course of the function GFLIB_Tan_F16

Note: The input pointer must contain a valid address otherwise a fault may occur (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 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 (HardFault).

Re-entrancy

The function is re-entrant.

2.36.1 Function `GFLIB_UpperLimit_F32`

Declaration

```
tFrac32 GFLIB_UpperLimit_F32(tFrac32 f32In, const
GFLIB_UPPERLIMIT_T_F32 *const pParam);
```

Arguments

Table 157. `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);
}
```

2.36.2 Function GFLIB_UpperLimit_F16

Declaration

```
tFrac16 GFLIB_UpperLimit_F16(tFrac16 f16In, const
GFLIB_UPPERLIMIT_T_F16 *const pParam);
```

Arguments

Table 158. GFLIB_UpperLimit_F16 arguments

Type	Name	Direction	Description
tFrac16	f16In	input	Input value.
const GFLIB_UPPERLIMIT_T_F16 *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.

Code Example

```
#include "gflib.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 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.

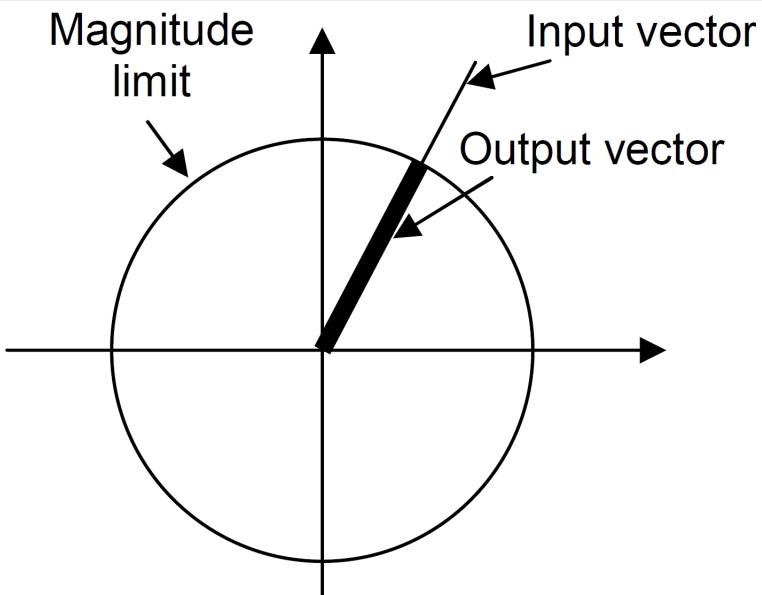


Figure 70. Graphical interpretation of the GFLIB_VectorLimit function.

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 (HardFault).

Re-entrancy

The function is re-entrant.

2.37.1 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 159. `GFLIB_VectorLimit_F32` arguments

Type	Name	Direction	Description
const SWLBS_2Syst_F32 *const	pIn	input	Pointer to the structure of the input vector.
SWLBS_2Syst_F32 *const	pOut	output	Pointer to the structure of the limited output vector.
const GFLIB_VECTORLIMIT_T_F32 *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.

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\(&f32pOut, &f32pIn, &f32trMyVectorLimit\);
}

```

2.37.2 Function [GFLIB_VectorLimit_F16](#)

Declaration

```
tBool GFLIB\_VectorLimit\_F16(SWLIBS\_2Syst\_F16 *const pOut, const
SWLIBS\_2Syst\_F16 *const pIn, const GFLIB\_VECTORLIMIT\_T\_F16 *const
pParam);
```

Arguments

Table 160. [GFLIB_VectorLimit_F16](#) arguments

Type	Name	Direction	Description
const SWLIBS_2Syst_F16 *const	pIn	input	Pointer to the structure of the input vector.
SWLIBS_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*.

Code Example

```
#include "gplib.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 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 (HardFault).

Re-entrancy

The function is re-entrant.

2.38.1 Function GFLIB_VMin_F32

Declaration

```
tU32 GFLIB_VMin_F32(const tFrac32 *pIn, tU32 u32N);
```

Arguments

Table 161. 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 "gflib.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)];
}
```

2.38.2 Function GFLIB_VMin_F16

Declaration

```
tU16 GFLIB_VMin_F16(const tFrac16 *pIn, tU16 u16N);
```

Arguments

Table 162. 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 "gflib.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)];
}
```

2.38.3 Function GFLIB_VMin4_F16**Declaration**

```
INLINE tU16 GFLIB_VMin4_F16(const tFrac16 *pIn);
```

Arguments**Table 163. GFLIB_VMin4_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 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 "gplib.h"

void main(void)
{
    tFrac16 pInputArray[4] = {6, 3, 1, 4};
    tFrac16 f16MinValue;

    f16MinValue = pInputArray[GFLIB_VMin4_F16(pInputArray)];
}
```

2.38.4 Function GFLIB_VMin5_F16

Declaration

INLINE `tU16 GFLIB_VMin5_F16(const tFrac16 *pIn);`

Arguments

Table 164. GFLIB_VMin5_F16 arguments

Type	Name	Direction	Description
const <code>tFrac16</code> *	<code>pIn</code>	<code>input</code>	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 "gplib.h"

void main(void)
{
    tFrac16 pInputArray[5] = {6, 3, 1, 4, 5};
    tFrac16 f16MinValue;

    f16MinValue = pInputArray[GFLIB_VMin5_F16(pInputArray)];
}
```

2.38.5 Function GFLIB_VMin6_F16

Declaration

INLINE `tU16 GFLIB_VMin6_F16(const tFrac16 *pIn);`

Arguments**Table 165. GFLIB_VMin6_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 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)];
}
```

2.38.6 Function GFLIB_VMin7_F16**Declaration**

```
INLINE tU16 GFLIB_VMin7_F16(const tFrac16 *pIn);
```

Arguments**Table 166. GFLIB_VMin7_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 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)];
}
```

2.38.7 Function GFLIB_VMin8_F16**Declaration**

INLINE [tU16](#) GFLIB_VMin8_F16(const [tFrac16](#) *pIn);

Arguments**Table 167. GFLIB_VMin8_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 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)];
}
```

2.38.8 Function GFLIB_VMin9_F16**Declaration**

INLINE [tU16](#) GFLIB_VMin9_F16(const [tFrac16](#) *pIn);

Arguments**Table 168. 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)];
}
```

2.38.9 Function GFLIB_VMin10_F16**Declaration**

```
INLINE tU16 GFLIB_VMin10_F16(const tFrac16 *pIn);
```

Arguments**Table 169. GFLIB_VMin10_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 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 "gplib.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)];
}
```

2.38.10 Function GFLIB_VMin11_F16

Declaration

```
INLINE tU16 GFLIB_VMin11_F16(const tFrac16 *pIn);
```

Arguments

Table 170. GFLIB_VMin11_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 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 "gplib.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)];
}
```

2.38.11 Function GFLIB_VMin12_F16

Declaration

```
INLINE tU16 GFLIB_VMin12_F16(const tFrac16 *pIn);
```

Arguments**Table 171.** `GFLIB_VMin12_F16` arguments

Type	Name	Direction	Description
const <code>tFrac16</code> *	<code>pIn</code>	<code>input</code>	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)];
}
```

2.38.12 Function GFLIB_VMin13_F16**Declaration**

```
INLINE tU16 GFLIB_VMin13_F16(const tFrac16 *pIn);
```

Arguments**Table 172.** `GFLIB_VMin13_F16` arguments

Type	Name	Direction	Description
const <code>tFrac16</code> *	<code>pIn</code>	<code>input</code>	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)];
}
```

2.38.13 Function GFLIB_VMin14_F16**Declaration**

INLINE `tU16 GFLIB_VMin14_F16(const tFrac16 *pIn);`

Arguments**Table 173. GFLIB_VMin14_F16 arguments**

Type	Name	Direction	Description
const <code>tFrac16 *</code>	<code>pIn</code>	<code>input</code>	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 "gplib.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)];
}
```

2.38.14 Function `GFLIB_VMin15_F16`

Declaration

```
INLINE tU16 GFLIB_VMin15_F16(const tFrac16 *pIn);
```

Arguments

Table 174. GFLIB_VMin15_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 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)];
}
```

2.38.15 Function `GFLIB_VMin16_F16`

Declaration

```
INLINE tU16 GFLIB_VMin16_F16(const tFrac16 *pIn);
```

Arguments

Table 175. GFLIB_VMin16_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 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)];
}
```

2.39 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 (α - β) 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 α 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 (HardFault).

Re-entrancy

The function is re-entrant.

2.39.1 Function GMCLIB_BetaProjection_F32

Declaration

```
void GMCLIB_BetaProjection_F32(SWLIBS\_2Syst\_F32 *const pOut,  
const SWLIBS\_3Syst\_F32 *const pIn);
```

Arguments

Table 176. GMCLIB_BetaProjection_F32 arguments

Type	Name	Direction	Description
const SWLIBS_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.
SWLIBS_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"

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 = 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);
}
```

2.39.2 Function GMCLIB_BetaProjection_F16

Declaration

```
void GMCLIB_BetaProjection_F16(SWLIBS\_2Syst\_F16 *const pOut,
const SWLIBS\_3Syst\_F16 *const pIn);
```

Arguments

Table 177. 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);
}
```

2.40 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 (HardFault).

Re-entrancy

The function is re-entrant.

2.40.1 Function GMCLIB_BetaProjection3Ph_F32

Declaration

```
void GMCLIB_BetaProjection3Ph_F32(SWLIBS\_3Syst\_F32 *const pOut,  
const SWLIBS\_3Syst\_F32 *const pIn);
```

Arguments

Table 178. GMCLIB_BetaProjection3Ph_F32 arguments

Type	Name	Direction	Description
const SWLIBS_3Syst_F32 *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.
SWLIBS_3Syst_F32 *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);
}

```

2.40.2 Function GMCLIB_BetaProjection3Ph_F16

Declaration

```
void GMCLIB_BetaProjection3Ph_F16(SWLIBS\_3Syst\_F16 *const pOut,
const SWLIBS\_3Syst\_F16 *const pIn);
```

Arguments

Table 179. GMCLIB_BetaProjection3Ph_F16 arguments

Type	Name	Direction	Description
const SWLIBS_3Syst_F16 *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.
SWLIBS_3Syst_F16 *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);
```

2.41 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 (α - β) orthogonal coordinate system, according to the following equations:

$$\begin{aligned} i_\alpha &= i_A \\ i_\beta &= (i_B - i_C) \cdot \frac{1}{\sqrt{3}} \end{aligned}$$

Equation GMCLIB_Clark_Eq1

where it is assumed that the axis A (axis of the first phase) and the axis α are in the same direction.

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

Re-entrancy

The function is re-entrant.

2.41.1 Function GMCLIB_Clark_F32

Declaration

```
void GMCLIB_Clark_F32(SWLIS_2Syst_F32 *const pOut, const
                      SWLIS_3Syst_F32 *const pIn);
```

Arguments

Table 180. GMCLIB_Clark_F32 arguments

Type	Name	Direction	Description
const SWLIS_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.
SWLIS_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"

SWLIS_3Syst_F32 f32trAbc;
SWLIS_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);
}

```

2.41.2 Function GMCLIB_Clark_F16

Declaration

```
void GMCLIB_Clark_F16(SWLIS_2Syst_F16 *const pOut, const
SWLIS_3Syst_F16 *const pIn);
```

Arguments

Table 181. GMCLIB_Clark_F16 arguments

Type	Name	Direction	Description
const SWLIS_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.
SWLIS_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"

SWLIS_3Syst_F16 f16trAbc;
SWLIS_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);
}

```

2.42 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 (α - β) orthogonal coordinate system to the three-phase (A-B-C) coordinate system, according to these equations:

$$i_A = i_\alpha$$

$$i_B = -\frac{1}{2} \cdot i_\alpha + \frac{\sqrt{3}}{2} \cdot i_\beta$$

$$i_C = -\frac{1}{2} \cdot i_\alpha - \frac{\sqrt{3}}{2} \cdot i_\beta$$

Equation GMCLIB_ClarkInv_Eq1

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

Re-entrancy

The function is re-entrant.

2.42.1 Function GMCLIB_ClarkInv_F32

Declaration

```
void GMCLIB_ClarkInv_F32(SWLIBS_3Syst_F32 *const pOut, const
                           SWLIBS_2Syst_F32 *const pIn);
```

Arguments**Table 182. GMCLIB_ClarkInv_F32 arguments**

Type	Name	Direction	Description
const SWLIBS_2Syst_F32 *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.
SWLIBS_3Syst_F32 *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);
```

2.42.2 Function GMCLIB_ClarkInv_F16

Declaration

```
void GMCLIB_ClarkInv_F16(SWLIBS\_3Syst\_F16 *const pOut, const
SWLIBS\_2Syst\_F16 *const pIn);
```

Arguments

Table 183. GMCLIB_ClarkInv_F16 arguments

Type	Name	Direction	Description
const SWLIBS_2Syst_F16 *const	pIn	input	Pointer to the structure containing data of the two-phase stationary orthogonal system (α - β). Arguments of the structure contain fixed point 16-bit values.
SWLIBS_3Syst_F16 *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);
```

{}

2.43 Function GMCLIB_DcouplingPMSM

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:

$$\begin{bmatrix} u_d \\ u_q \end{bmatrix} = R_s \cdot \begin{bmatrix} i_d \\ i_q \end{bmatrix} + \begin{bmatrix} L_d & 0 \\ 0 & L_q \end{bmatrix} \frac{di}{dt} \begin{bmatrix} i_d \\ i_q \end{bmatrix} + \omega_e \begin{bmatrix} -L_q & 0 \\ 0 & L_d \end{bmatrix} \begin{bmatrix} i_q \\ i_d \end{bmatrix} + \omega_e \psi_{pm} \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

Equation GMCLIB_DcouplingPMSM_Eq1

It can be seen that [GMCLIB_DcouplingPMSM_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_DcouplingPMSM_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:

$$\begin{aligned} u_d &= R_s i_d + L_d \frac{di_d}{dt} \\ u_q &= R_s i_q + L_q \frac{di_q}{dt} \end{aligned}$$

Equation GMCLIB_DcouplingPMSM_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_DcouplingPMSM_Eq1](#), using the feed-forward voltages u_{dq_comp} given from [GMCLIB_DcouplingPMSM_Eq1](#) as follows:

$$\begin{aligned} u_{d_{comp}} &= -\omega_e L_q i_q \\ u_{q_{comp}} &= \omega_e L_d i_d \end{aligned}$$

Equation GMCLIB_DcouplingPMSM_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_{q_dec} and quadrature u_{q_decq} 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:

$$\begin{aligned} u_{d_{dec}} &= u_d + u_{d_{comp}} \\ u_{q_{dec}} &= u_q + u_{q_{comp}} \end{aligned}$$

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 (HardFault).

Re-entrancy

The function is re-entrant.

2.43.1 Function GMCLIB_DecouplingPMSM_F32

Declaration

```
void GMCLIB_DecouplingPMSM_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 184. GMCLIB_DecouplingPMSM_F32 arguments

Type	Name	Direction	Description
SWLIBS_2Syst_F32 *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 SWLIBS_2Syst_F32 *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 SWLIBS_2Syst_F32 *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.
tFrac32	f32AngularVel	input	Rotor angular velocity in rad/sec, referred to as (ω_{ef}) in the detailed section of the documentation.
const GMCLIB_DECOUPLINGPMSM_T_F32 *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} , Ω_{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:

$$\begin{aligned} u_{df_{dec}} &= \frac{u_{df}}{U_{max}}, & u_{qf_{dec}} &= \frac{u_{qf}}{U_{max}}, & u_{df} &= \frac{u_d}{U_{max}}, & u_{qf} &= \frac{u_q}{U_{max}}, \\ i_{df} &= \frac{i_d}{I_{max}}, & i_{qf} &= \frac{i_q}{I_{max}}, & \omega_{ef} &= \frac{\omega_e}{\Omega_{max}} \end{aligned}$$

Equation GMCLIB_DcouplingPMSM_F32_Eq2

Further, rearranging [GMCLIB_DcouplingPMSM_F32_Eq1](#) results in:

$$\begin{aligned} u_{df_{dec}} &= u_{df} - \omega_{ef} \cdot i_{qf} \cdot \frac{\Omega_{max} \cdot L_q \cdot I_{max}}{U_{max}} = u_{df} - \omega_{ef} \cdot i_{qf} \cdot k_q \\ u_{qf_{dec}} &= u_{qf} + \omega_{ef} \cdot i_{df} \cdot \frac{\Omega_{max} \cdot L_d \cdot I_{max}}{U_{max}} = u_{qf} + \omega_{ef} \cdot i_{df} \cdot k_d \end{aligned}$$

Equation GMCLIB_DcouplingPMSM_F32_Eq3

where k_d and k_q are coefficients calculated as:

$$\begin{aligned} k_d &= L_q \cdot \Omega_{max} \cdot \frac{I_{max}}{U_{max}} \\ k_q &= L_d \cdot \Omega_{max} \cdot \frac{I_{max}}{U_{max}} \end{aligned}$$

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:

$$u_{df_{dec}} = u_{df} - \omega_{ef} \cdot i_{qf} \cdot k_{qf} \cdot 2^{k_{q_shift}}$$

$$u_{qf_{dec}} = u_{qf} + \omega_{ef} \cdot i_{df} \cdot k_{df} \cdot 2^{k_{d_shift}}$$

Equation GMCLIB_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_DcouplingPMSM_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);
}

```

2.43.2 Function GMCLIB_DecouplingPMSM_F16

Declaration

```
void GMCLIB_DecouplingPMSM_F16(SWLBS_2Syst_F16 *const pUdqDec,
const SWLBS_2Syst_F16 *const pUdq, const SWLBS_2Syst_F16 *const
pIdq, tFrac16 f16AngularVel, const GMCLIB_DECOUPLINGPMSM_T_F16
*const pParam);
```

Arguments

Table 185. GMCLIB_DecouplingPMSM_F16 arguments

Type	Name	Direction	Description
SWLBS_2Syst_F16 *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 SWLBS_2Syst_F16 *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 SWLBS_2Syst_F16 *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.
tFrac16	f16AngularVel	input	Rotor angular velocity in rad/sec, referred to as (ω_{ef}) in the detailed section of the documentation.
const GMCLIB_DECOUPLINGPMSM_T_F16 *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_DecouplingPMSM_eq3](#) into [GMCLIB_DecouplingPMSM_eq4](#), and normalizing [GMCLIB_DecouplingPMSM_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_DecouplingPMSM_F16_Eq1

where subscript f denotes the fractional representation of the respective quantity, and U_{max} , I_{max} , Ω_{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:

$$\begin{aligned} u_{df_{dec}} &= \frac{u_{d_{dec}}}{U_{max}}, & u_{qf_{dec}} &= \frac{u_{q_{dec}}}{U_{max}}, & u_{df} &= \frac{u_d}{U_{max}}, & u_{qf} &= \frac{u_q}{U_{max}}, \\ i_{df} &= \frac{i_d}{I_{max}}, & i_{qf} &= \frac{i_q}{I_{max}}, & \omega_{ef} &= \frac{\omega_e}{\Omega_{max}} \end{aligned}$$

Equation GMCLIB_DecouplingPMSM_F16_Eq2

Further, rearranging [GMCLIB_DecouplingPMSM_F16_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_DecouplingPMSM_F16_Eq3

where k_d and k_q are coefficients calculated as:

$$\begin{aligned} k_d &= L_q \cdot \Omega_{max} \cdot \frac{I_{max}}{U_{max}} \\ k_q &= L_d \cdot \Omega_{max} \cdot \frac{I_{max}}{U_{max}} \end{aligned}$$

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:

$$\begin{aligned} k_{d_shift} &= ceil\left(\frac{\log(k_d)}{\log(2)}\right) \\ k_{q_shift} &= ceil\left(\frac{\log(k_q)}{\log(2)}\right) \end{aligned}$$

Equation GMCLIB_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:

$$\begin{aligned} k_{df} &= k_d \cdot 2^{k_{d_shift}} \\ k_{qf} &= k_q \cdot 2^{k_{q_shift}} \end{aligned}$$

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);  
}
```

2.44 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 (α) and quadrature (β) 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 (HardFault).

Re-entrancy

The function is re-entrant.

2.44.1 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 186. GMCLIB_ElimDcBusRip_F32 arguments**

Type	Name	Direction	Description
SWLIBS_2Syst_F32 *const	pOut	output	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 SWLIBS_2Syst_F32 *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 GMCLIB_ELIMDCBUSRIP_T_F32 *const	pParam	input	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_{α} and u_{β} are the input voltages, and the u_{α}^* and u_{β}^* are the output duty-cycle ratios.

The f32ModIndex and f32ArgDCBusMsr are supplied to the function within the parameters structure through its members. The u_{α} , u_{β} correspond respectively to the f32Arg1 and f32Arg2 members of the input structure, and the u_{α}^* and u_{β}^* 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 $pIn->f32ArgDCBusMsr$ 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);
}

```

2.44.2 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 187. GMCLIB_ElimDcBusRip_F16 arguments

Type	Name	Direction	Description
SWLIBS_2Syst_F16 *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.
const SWLIBS_2Syst_F16 *const	pIn	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 GMCLIB_ELIMDCBUSRIP_T_F16 *const	pParam	input	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

$$u_{\alpha}^* = \begin{cases} \frac{f16ModIndex \cdot u_{\alpha}}{f16ArgDCBusMsr/2} & \text{if } abs(f16ModIndex \cdot u_{\alpha}) < \frac{f16ArgDCBusMsr}{2} \\ sign(u_{\alpha}) & \text{otherwise} \end{cases}$$

$$u_{\beta}^* = \begin{cases} \frac{f16ModIndex \cdot u_{\beta}}{f16ArgDCBusMsr/2} & \text{if } abs(f16ModIndex \cdot u_{\beta}) < \frac{f16ArgDCBusMsr}{2} \\ sign(u_{\beta}) & \text{otherwise} \end{cases}$$

Equation GMCLIB_ElimDcBusRip_F16_Eq3

where: f16ModIndex is the inverse modulation index, f16ArgDcBusMsr is the measured DC bus voltage, the u_{α} and u_{β} are the input voltages, and the u_{α}^* and u_{β}^* are the output duty-cycle ratios. The f16ModIndex and f16ArgDcBusMsr are supplied to the function within the parameters structure through its members. The u_{α} , u_{β} correspond respectively to the f16Arg1 and f16Arg2 members of the input structure, and the u_{α}^* and u_{β}^* 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);
}

```

2.45 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 (α - β) 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 θ_e represents the electrical position of the rotor flux.

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

Re-entrancy

The function is re-entrant.

2.45.1 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 188. GMCLIB_Park_F32 arguments

Type	Name	Direction	Description
SWLIBS_2Syst_F32 *	pOut	input, output	Pointer to the structure containing data of the two-phase rotational orthogonal system (d-q).
const SWLIBS_2Syst_F32 *const	pInAngle	input	Pointer to the structure where the values of the sine and cosine of the rotor position are stored.
const SWLIBS_2Syst_F32 *const	pIn	input	Pointer to the structure containing data of the two-phase stationary orthogonal system (\alpha- \beta).

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 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);
}

```

2.45.2 Function GMCLIB_Park_F16

Declaration

```
void GMCLIB_Park_F16(SWLIBLES\_2Syst\_F16 *pOut, const
SWLIBLES\_2Syst\_F16 *const pInAngle, const SWLIBLES\_2Syst\_F16 *const
pIn);
```

Arguments

Table 189. GMCLIB_Park_F16 arguments

Type	Name	Direction	Description
SWLIBLES_2Syst_F16 *	pOut	input, output	Pointer to the structure containing data of the two-phase rotational orthogonal system (d-q).
const SWLIBLES_2Syst_F16 *const	pInAngle	input	Pointer to the structure where the values of the sine and cosine of the rotor position are stored.
const SWLIBLES_2Syst_F16 *const	pIn	input	Pointer to the structure containing data of the two-phase stationary orthogonal system (\alpha- \beta).

Note: The inputs and the outputs are normalized to fit in the range [-1, 1].

Code Example

```

#include "gmclib.h"

SWLIBLES\_2Syst\_F16 tr16Angle;
SWLIBLES\_2Syst\_F16 tr16AlBe;
SWLIBLES\_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.f16Arg1 ~ d = 0x505E
// tr16Dq.f16Arg2 ~ q = 0x1C38
GMCLIB_Park(&tr16Dq,&tr16Angle,&tr16AlBe);
}
```

2.46 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 (α - β) 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 θ_e represents the electrical position of the rotor flux.

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

Re-entrancy

The function is re-entrant.

2.46.1 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 190. GMCLIB_ParkInv_F32 arguments**

Type	Name	Direction	Description
SWLIBS_2Syst_F32 *const	pOut	input, output	Pointer to the structure containing data of the two-phase stationary orthogonal system (\alpha- \beta).
const SWLIBS_2Syst_F32 *const	pInAngle	input	Pointer to the structure where the values of the sine and cosine of the rotor position are stored.
const SWLIBS_2Syst_F32 *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);
```

2.46.2 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 191. GMCLIB_ParkInv_F16 arguments

Type	Name	Direction	Description
SWLIBS_2Syst_F16 *const	pOut	input, output	Pointer to the structure containing data of the two-phase stationary orthogonal system (α - β).
const SWLIBS_2Syst_F16 *const	pInAngle	input	Pointer to the structure where the values of the sine and cosine of the rotor position are stored.
const SWLIBS_2Syst_F16 *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);
}
```

2.47 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.

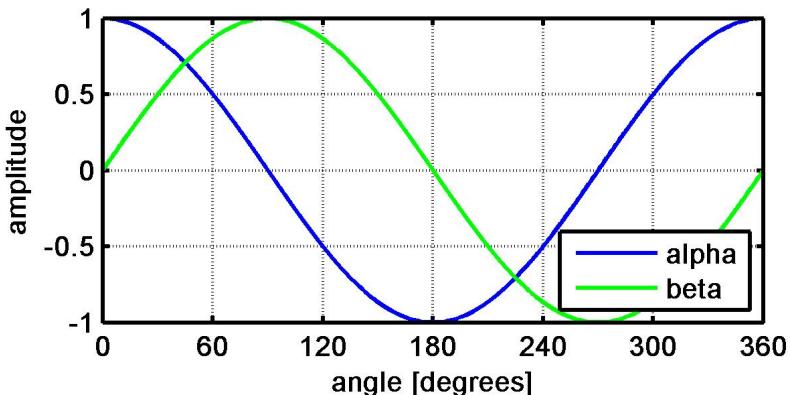


Figure 71. Input waveforms

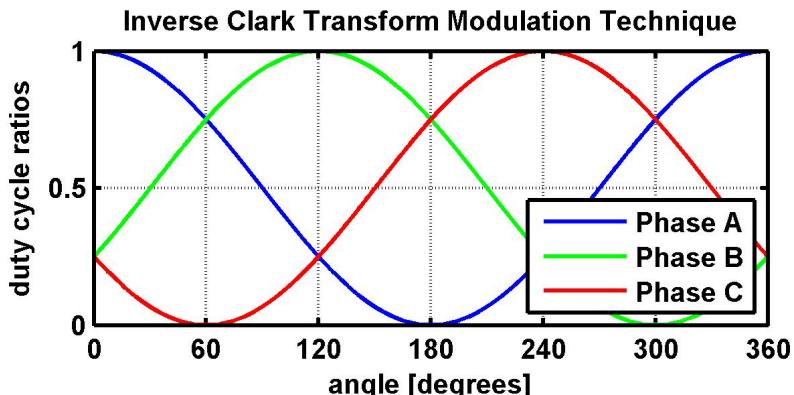


Figure 72. Output waveforms

Note: The input/output pointers must contain valid addresses, otherwise an exception may occur (Data TLB Error, Data Storage, Alignment, Machine Check).

Re-entrancy

The function is re-entrant.

2.47.1 Function GMCLIB_PwmIct_F32

Declaration

```
tU32 GMCLIB_PwmIct_F32(SWLIBS_3Syst_F32 *pOut, const
                           SWLIBS_2Syst_F32 *const pIn);
```

Arguments

Table 192. GMCLIB_PwmIct_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_a and quadrature U_b 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
}

```

2.47.2 Function GMCLIB_PwmIct_F16

Declaration

```
tU16 GMCLIB_PwmIct_F16(SWLIBS\_3Syst\_F16 *pOut, const
SWLIBS\_2Syst\_F16 *const pIn);
```

Arguments

Table 193. 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.
const SWLIBS_2Syst_F16 *const	pIn	input	Pointer to the structure containing direct U_α and quadrature U_β 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
}

```

2.48 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.

$$\begin{aligned}
 u_a &= u_\alpha \\
 u_b &= \frac{-u_\alpha + \sqrt{3} \cdot u_\beta}{2} \\
 u_c &= \frac{-u_\alpha - \sqrt{3} \cdot u_\beta}{2}
 \end{aligned}$$

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:

$$\begin{aligned} u_0 &= \begin{cases} 1-u'_a & \text{if } u'_a > 1 \\ -1-u'_a & \text{if } u'_a < -1 \\ 0 & \text{otherwise} \end{cases} \\ u_0 &= \begin{cases} 1-u'_b & \text{if } u'_b > 1 \\ -1-u'_b & \text{if } u'_b < -1 \\ 0 & \text{otherwise} \end{cases} \\ u_0 &= \begin{cases} 1-u'_c & \text{if } u'_c > 1 \\ -1-u'_c & \text{if } u'_c < -1 \\ 0 & \text{otherwise} \end{cases} \end{aligned}$$

Equation GMCLIB_SvmSci_Eq3

4. Due to the 120° 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:

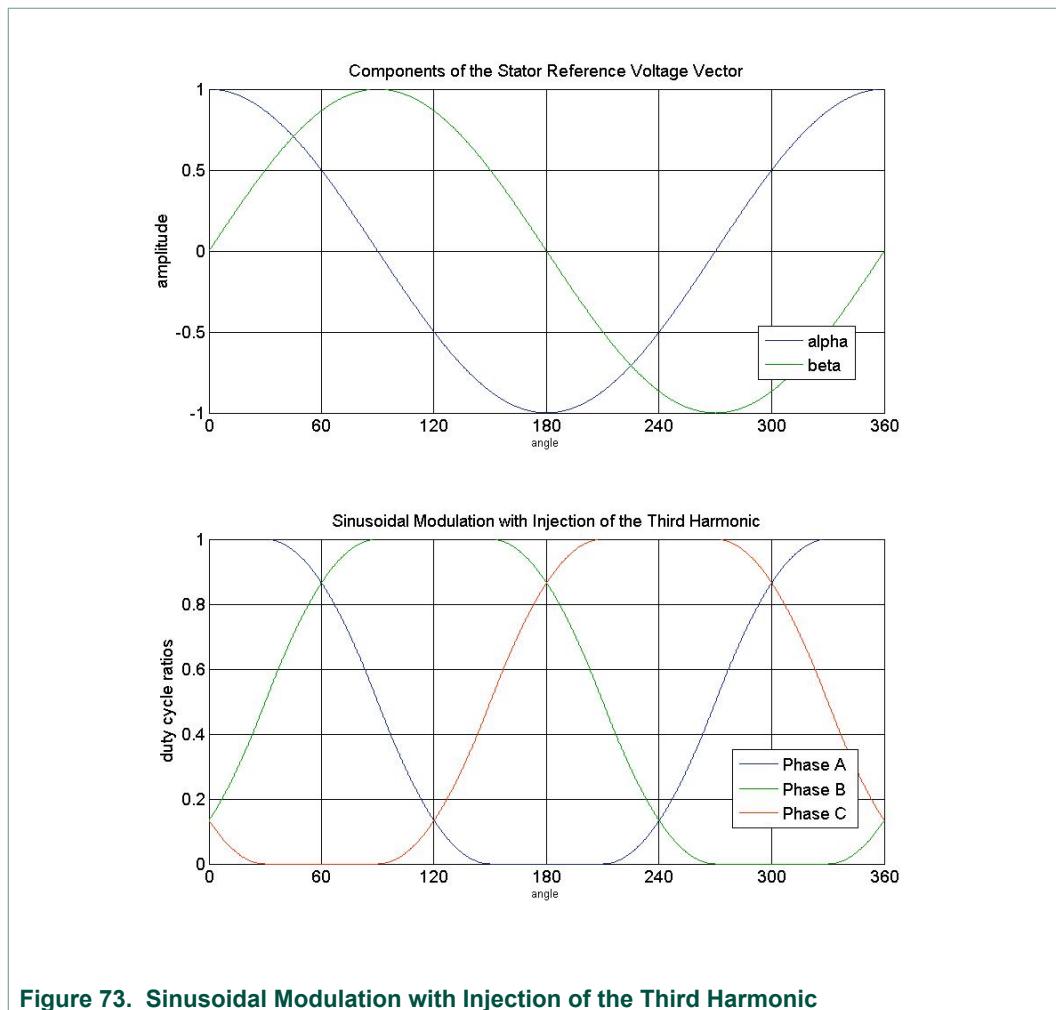
$$pwm_A = \frac{1}{2} (u_0 + u'_a + 1)$$

$$pwm_B = \frac{1}{2} (u_0 + u'_b + 1)$$

$$pwm_C = \frac{1}{2} (u_0 + u'_c + 1)$$

Equation GMCLIB_SvmSci_Eq4

[Figure 73](#) shows calculated waveforms of the duty-cycle ratios using the sinusoidal modulation with Sine Cap Injection algorithm.

**Figure 73. 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 an exception may occur (Data TLB Error, Data Storage, Alignment, Machine Check).

Re-entrancy

The function is re-entrant.

2.48.1 Function GMCLIB_SvmSci_F32

Declaration

```
tU32 GMCLIB_SvmSci_F32(SWLBS_3Syst_F32 *pOut, const
                           SWLBS_2Syst_F32 *const pIn);
```

Arguments**Table 194. GMCLIB_SvmSci_F32 arguments**

Type	Name	Direction	Description
SWLBS_3Syst_F32 *	pOut	input, output	Pointer to the structure containing calculated duty-cycle ratios of the 3-Phase system.
const SWLBS_2Syst_F32 *const	pln	input	Pointer to the structure containing direct U_α and quadrature U_β 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);
}
```

2.48.2 Function GMCLIB_SvmSci_F16

Declaration

```
tU16 GMCLIB_SvmSci_F16(SWLIBS_3Syst_F16 *pOut, const
SWLIBS_2Syst_F16 *const pIn);
```

Arguments

Table 195. GMCLIB_SvmSci_F16 arguments

Type	Name	Direction	Description
<code>SWLIBS_3Syst_F16 *</code>	<code>pOut</code>	<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>	<code>pIn</code>	<code>input</code>	Pointer to the structure containing direct U_α and quadrature U_β 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);
}
```

2.49 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 74](#).

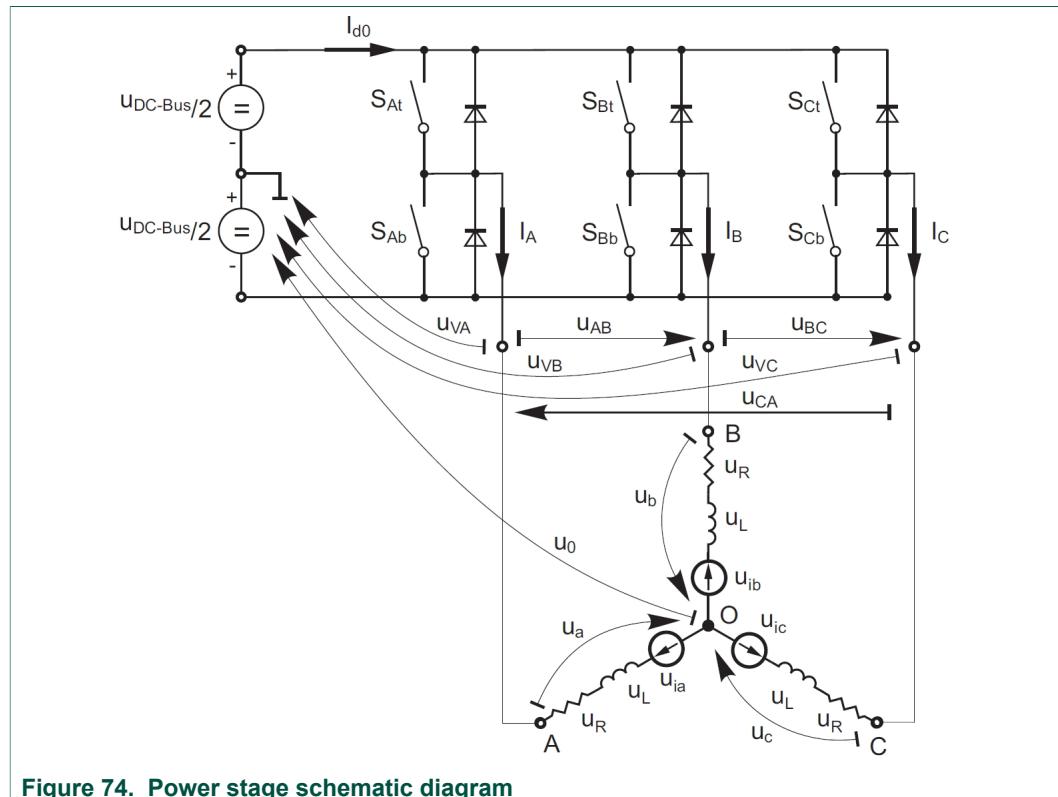


Figure 74. Power stage schematic diagram

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 74](#)), eight possible switching states (detailed in [Figure 75](#)) are feasible.

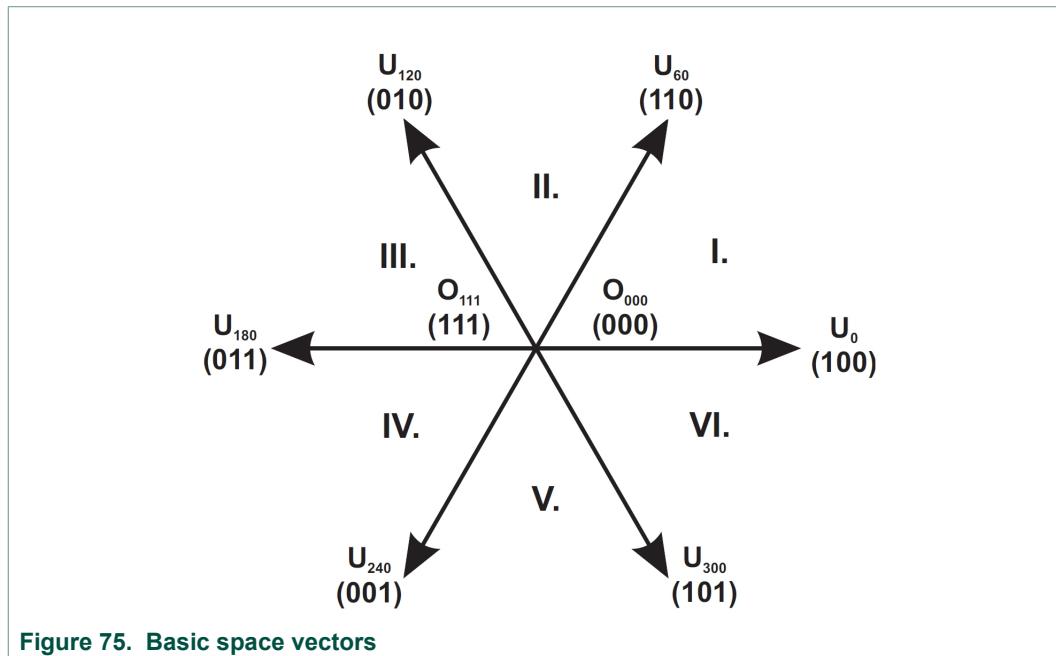


Figure 75. Basic space vectors

These states, together with the resulting instantaneous output line-to-line and phase voltages, are listed in [Table 196](#).

Table 196. 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*U _{DCBus}	-1/3*U _{DCBus}	-1/3*U _{DCBus}	U _{DCBus}	0	-U _{DCBus}	U ₀
1	1	0	1/3*U _{DCBus}	1/3*U _{DCBus}	-2/3*U _{DCBus}	0	U _{DCBus}	-U _{DCBus}	U ₆₀
0	1	0	-1/3*U _{DCBus}	2/3*U _{DCBus}	-1/3*U _{DCBus}	-U _{DCBus}	U _{DCBus}	0	U ₁₂₀
0	1	1	-2/3*U _{DCBus}	1/3*U _{DCBus}	1/3*U _{DCBus}	-U _{DCBus}	0	U _{DCBus}	U ₁₈₀
0	0	1	-1/3*U _{DCBus}	-1/3*U _{DCBus}	2/3*U _{DCBus}	0	-U _{DCBus}	U _{DCBus}	U ₂₄₀
1	0	1	1/3*U _{DCBus}	-2/3*U _{DCBus}	1/3*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_α and the quadrature-u_β 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

$$U_{\beta} = \frac{2}{3} \cdot \left(0 + \frac{\sqrt{3}U_b}{2} - \frac{\sqrt{3}U_c}{2} \right)$$

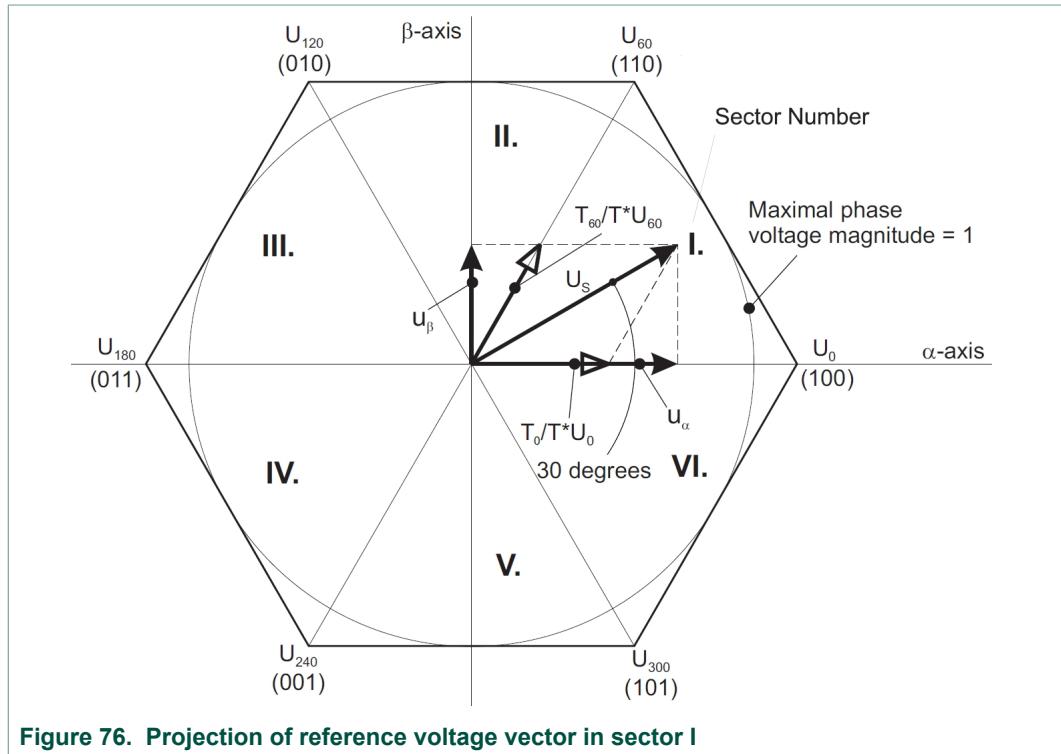
Equation GMCLIB_SvmStd_Eq2

The three-phase stator voltages, U_a , U_b , and U_c , are transformed using the Clarke Transformation into the U_α and the U_β components of the two-phase orthogonal coordinate system. The transformation results are listed in [Table 197](#).

Table 197. Switching patterns and space vectors

a	b	c	U_α	U_β	Vector
0	0	0	0	0	O_{000}
1	0	0	$2/3*U_{DCBus}$	0	U_0
1	1	0	$1/3*U_{DCBus}$	$1/\sqrt{3}*U_{DCBus}$	U_{60}
0	1	0	$-1/3*U_{DCBus}$	$1/\sqrt{3}*U_{DCBus}$	U_{120}
0	1	1	$-2/3*U_{DCBus}$	0	U_{180}
0	0	1	$-1/3*U_{DCBus}$	$-1/\sqrt{3}*U_{DCBus}$	U_{240}
1	0	1	$1/3*U_{DCBus}$	$-1/\sqrt{3}*U_{DCBus}$	U_{300}
1	1	1	0	0	O_{111}

[Figure 75](#) 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 76. 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 76](#) and [Figure 77](#).

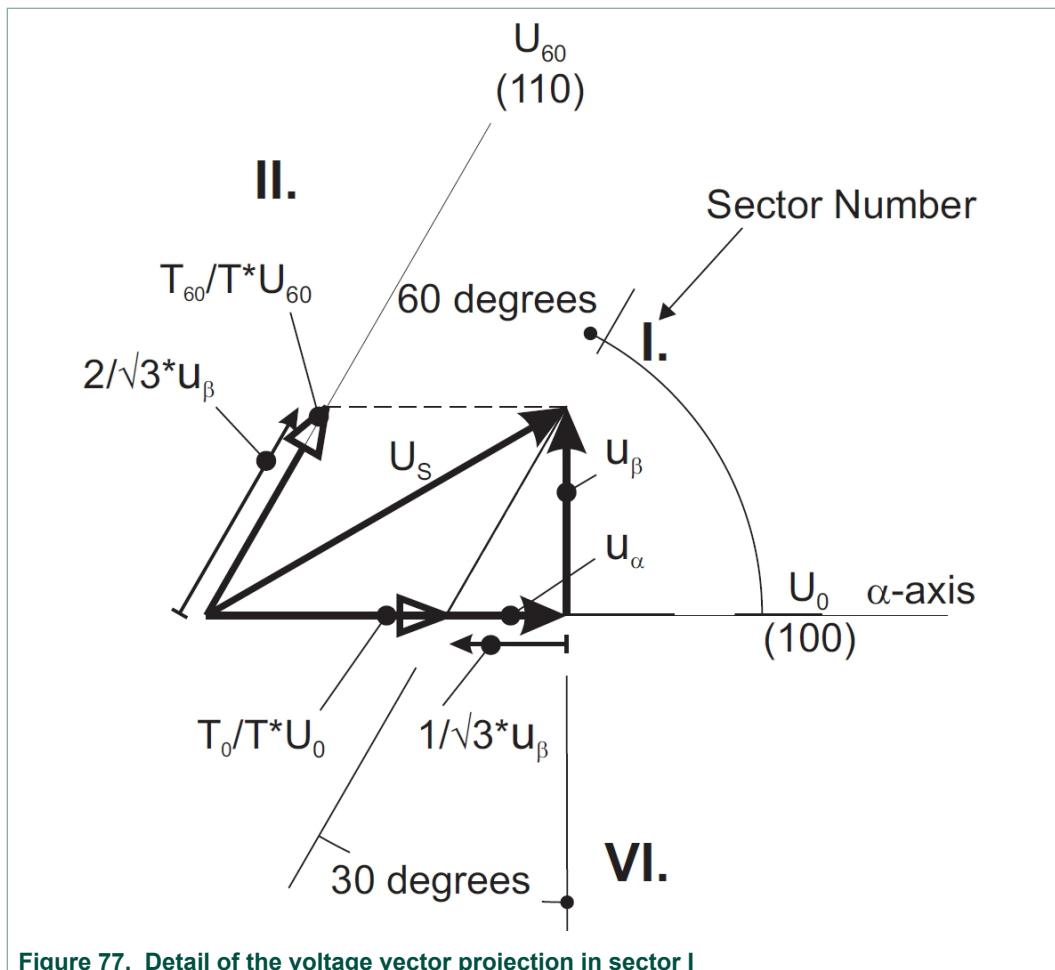


Figure 77. Detail of the voltage vector projection in sector I

The stator reference voltage vector U_S is phase-advanced by 30° from the axis- α 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_α and U_β 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_{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

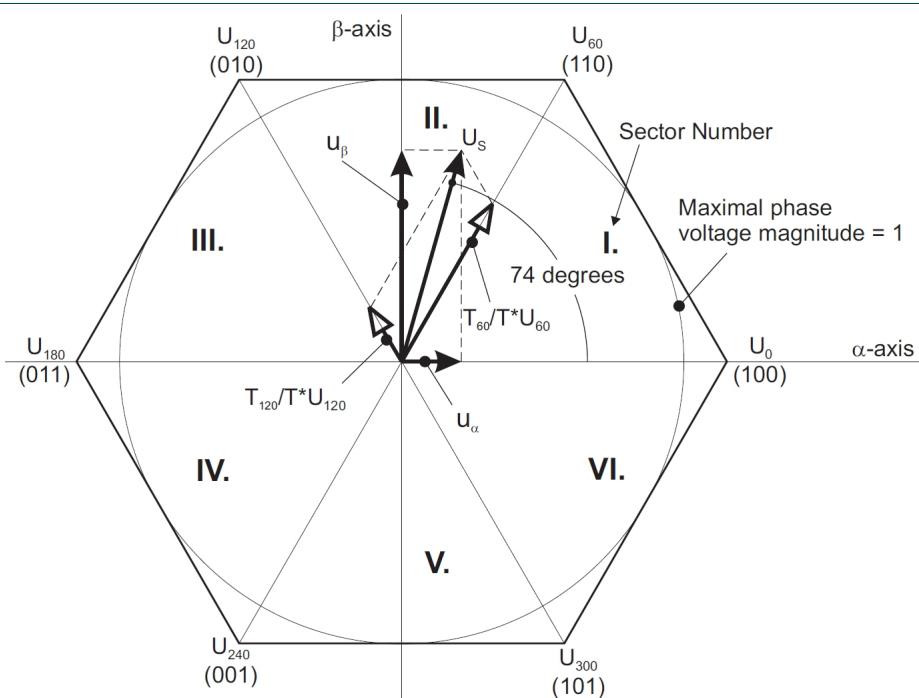


Figure 78. Projection of the reference voltage vector in sector II

Sector II is depicted in [Figure 78](#). 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 79](#).

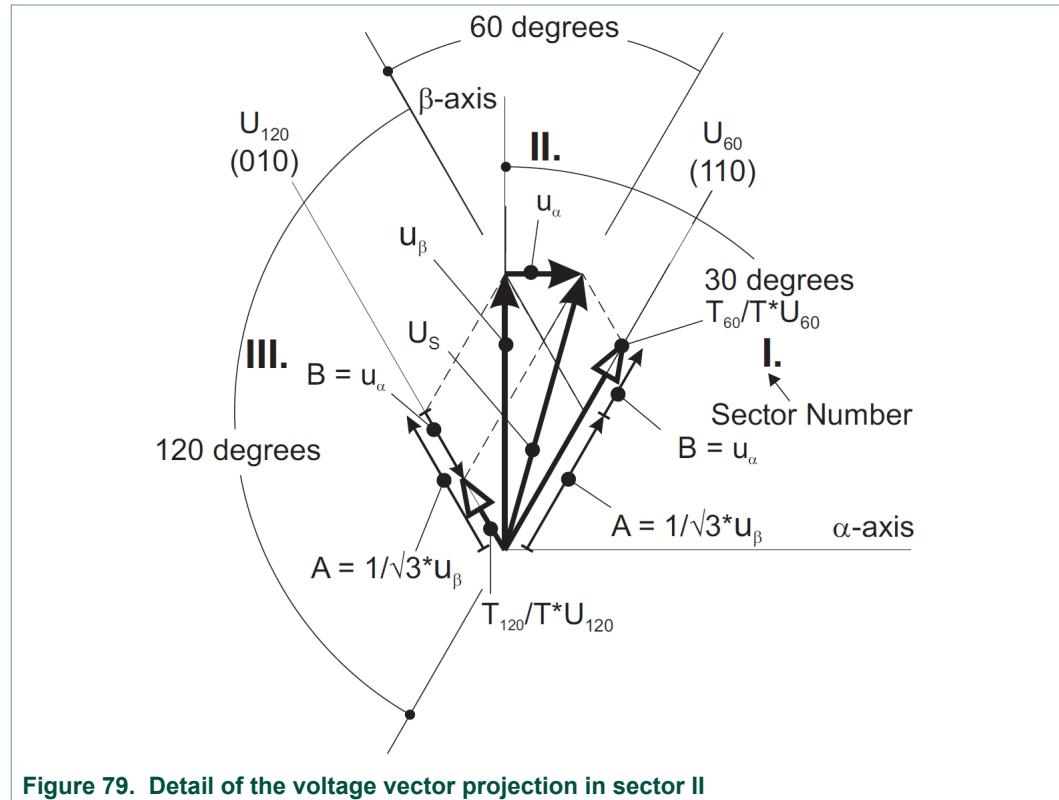


Figure 79. Detail of the voltage vector projection in sector II

The equations describing those auxiliary time-duration components are:

$$\frac{\sin^{-1}30^\circ}{\sin^{-1}120^\circ} = \frac{A}{u_a}$$

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 198](#).

Table 198. 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- α and quadrature- β 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

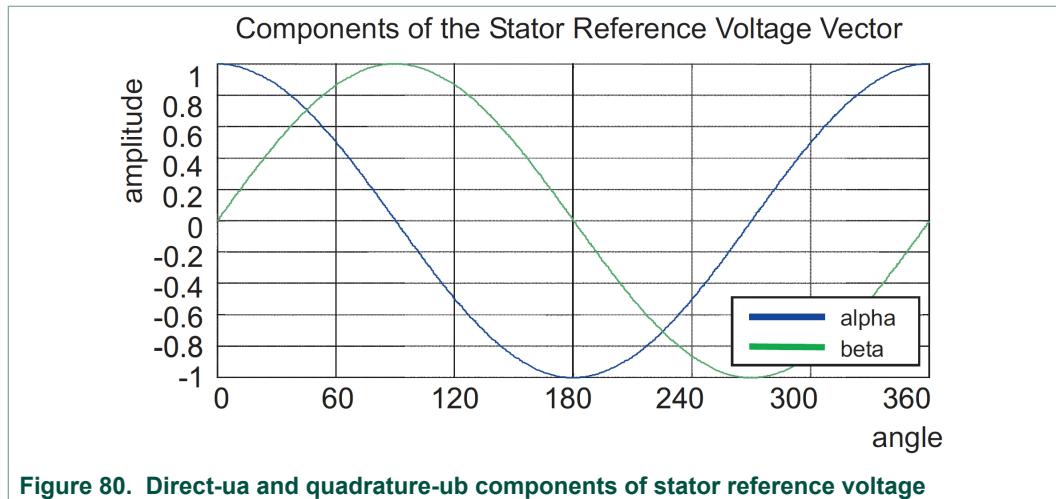
$$u_{ref2} = \frac{1}{2} (-u_\beta + \sqrt{3} \cdot u_\alpha)$$

Equation GMCLIB_SvmStd_Eq23

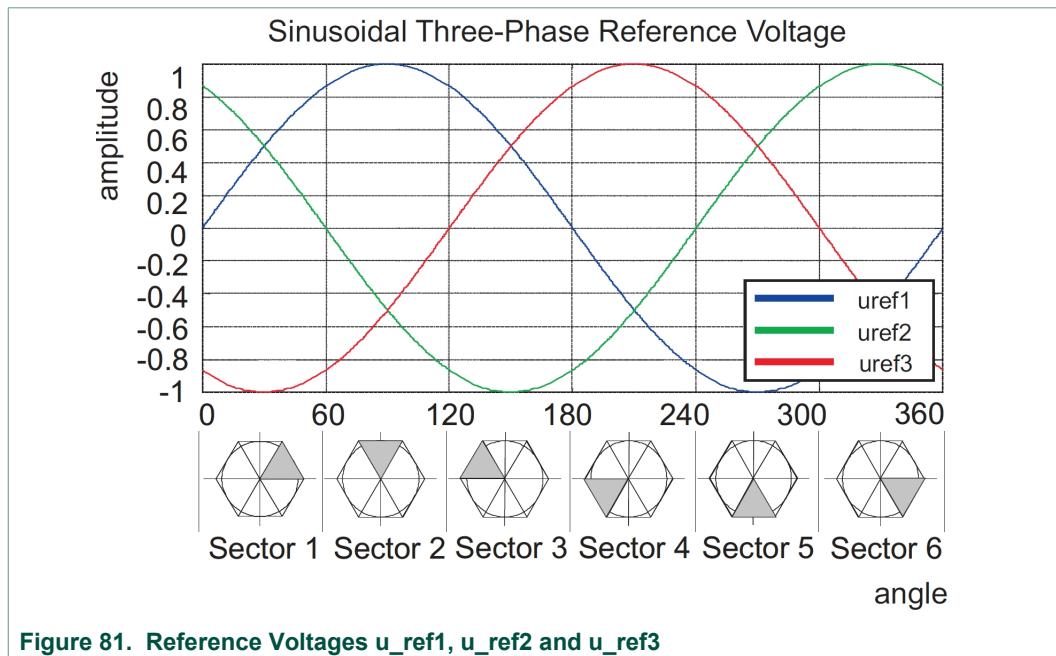
$$u_{ref3} = \frac{1}{2} (-u_\beta - \sqrt{3} \cdot u_\alpha)$$

Equation GMCLIB_SvmStd_Eq24

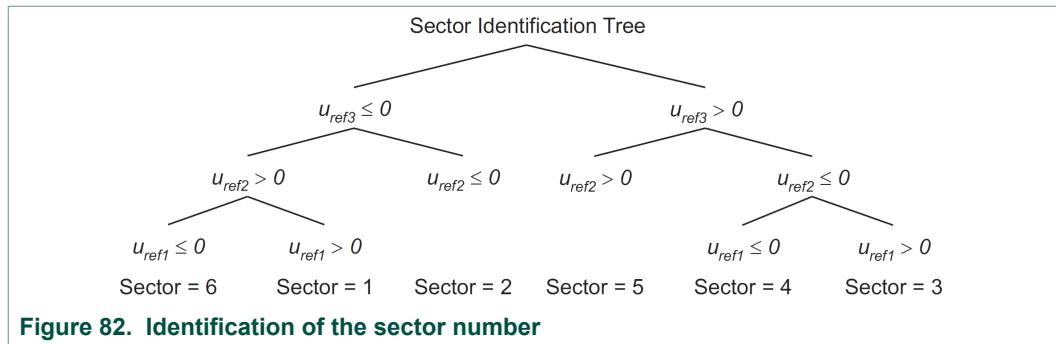
The modified Inverse Clark Transformation projects the quadrature- u_β component into u_{ref1} , as shown in [Figure 80](#) and [Figure 81](#), whereas voltages generated by the conventional Inverse Clark Transformation project the u_α component into u_{ref1} .

**Figure 80. Direct- u_α and quadrature- u_β components of stator reference voltage**

[Figure 80](#) depicts the u_α and u_β 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 81. Reference Voltages $u_{\text{ref}1}$, $u_{\text{ref}2}$ and $u_{\text{ref}3}$**

The Sector Identification Tree, shown in [Figure 82](#), can be a numerical solution of the approach shown in [Figure 81](#).



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 76](#), the stator reference voltage vector is phase-advanced by 30° from the α -axis, which results in the positive quantities of u_{ref1} and u_{ref2} and the negative quantity of u_{ref3} ; refer to [Figure 81](#). 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 79](#). 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

$$t_2 = t_1 + t_1$$

Equation GMCLIB_SvmStd_Eq26

$$t_3 = t_2 + t_2$$

Equation GMCLIB_SvmStd_Eq27

where T is the switching period, t_1 and t_2 are the duty-cycle ratios (see [Table 198](#)) 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 199](#).

Table 199. 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 _a	t_3	t_2	t_1	t_1	t_2	t_3
pwm _b	t_2	t_3	t_3	t_2	t_1	t_1
pwm _c	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 83](#)

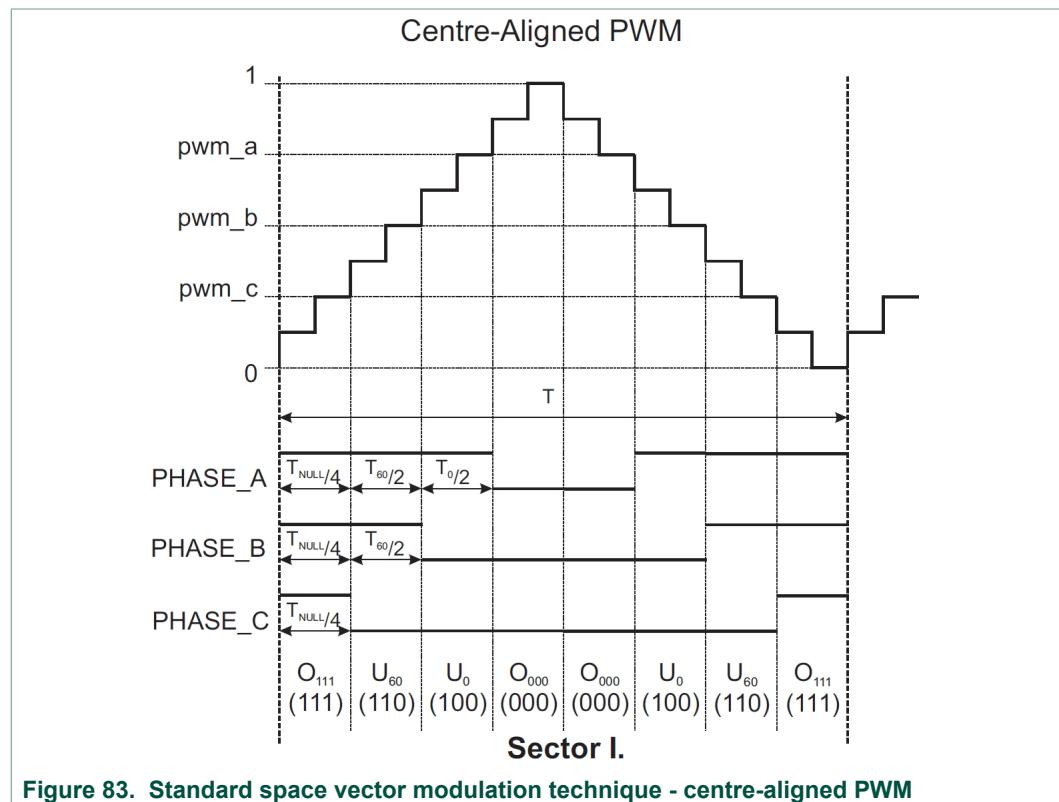


Figure 83. Standard space vector modulation technique - centre-aligned PWM

Note: The input/output pointers must contain valid addresses, otherwise an exception may occur (Data TLB Error, Data Storage, Alignment, Machine Check).

Re-entrancy

The function is re-entrant.

2.49.1 Function GMCLIB_SvmStd_F32

Declaration

```
tU32 GMCLIB_SvmStd_F32(SWLIBS_3Syst_F32 *pOut, const
                           SWLIBS_2Syst_F32 *const pIn);
```

Arguments**Table 200. GMCLIB_SvmStd_F32 arguments**

Type	Name	Direction	Description
SWLBS_3Syst_F32 *	pOut	input, output	Pointer to the structure containing calculated duty-cycle ratios of the 3-Phase system.
const SWLBS_2Syst_F32 *const	pln	input	Pointer to the structure containing direct U_α and quadrature U_β 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);
}
```

2.49.2 Function GMCLIB_SvmStd_F16

Declaration

```
tU16 GMCLIB_SvmStd_F16(SWLIBS_3Syst_F16 *pOut, const
SWLIBS_2Syst_F16 *const pIn);
```

Arguments

Table 201. 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_α and quadrature U_β 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);
}
```

2.50 Function GMCLIB_SvmU0n

This function calculates the duty-cycle ratios using the special Space Vector Modulation technique, termed Space Vector Modulation with O₀₀₀ 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 O₀₀₀ Nulls.

The derivation approach of the Space Vector Modulation technique with O₀₀₀ Nulls is identical, in many aspects, to the approach presented in [GMCLIB_SvmStd](#). However, a distinct difference lies in the definition of the variables t₁, t₂ and t₃ that represents switching duty-cycle ratios of the respective phases:

$$t_1 = 0$$

Equation GMCLIB_SvmU0n_Eq1

$$t_2 = t_1 + t_{-1}$$

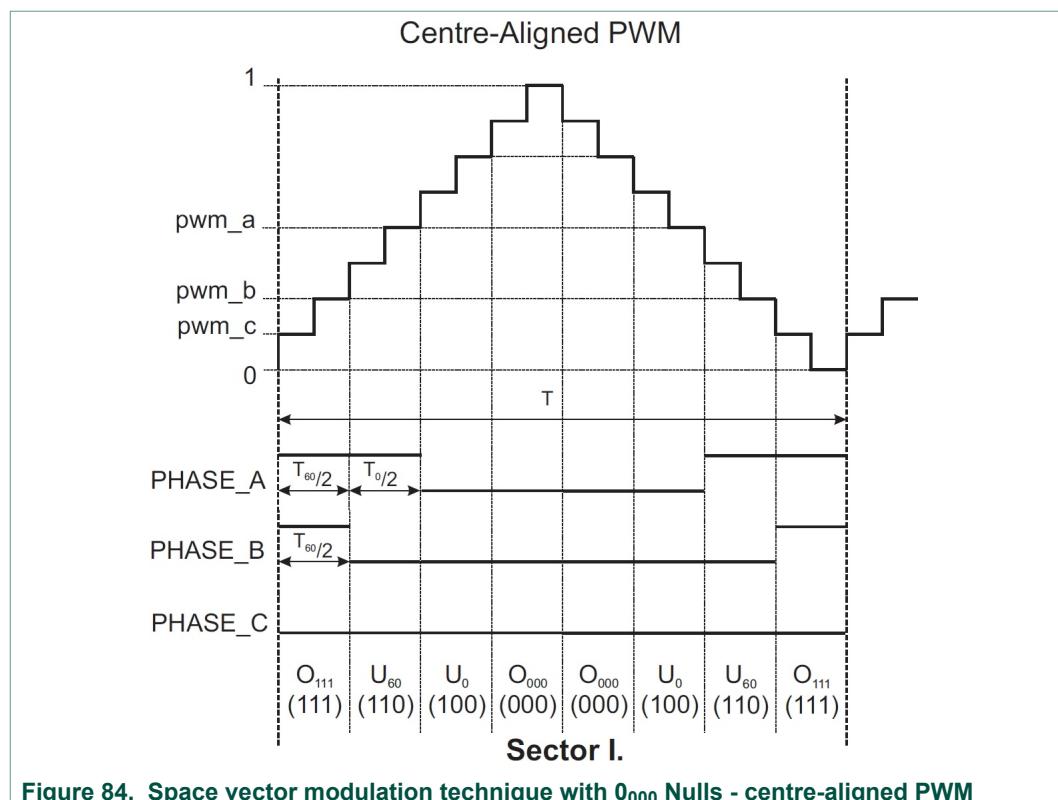
Equation GMCLIB_SvmU0n_Eq2

$$t_3 = t_2 + t_{-2}$$

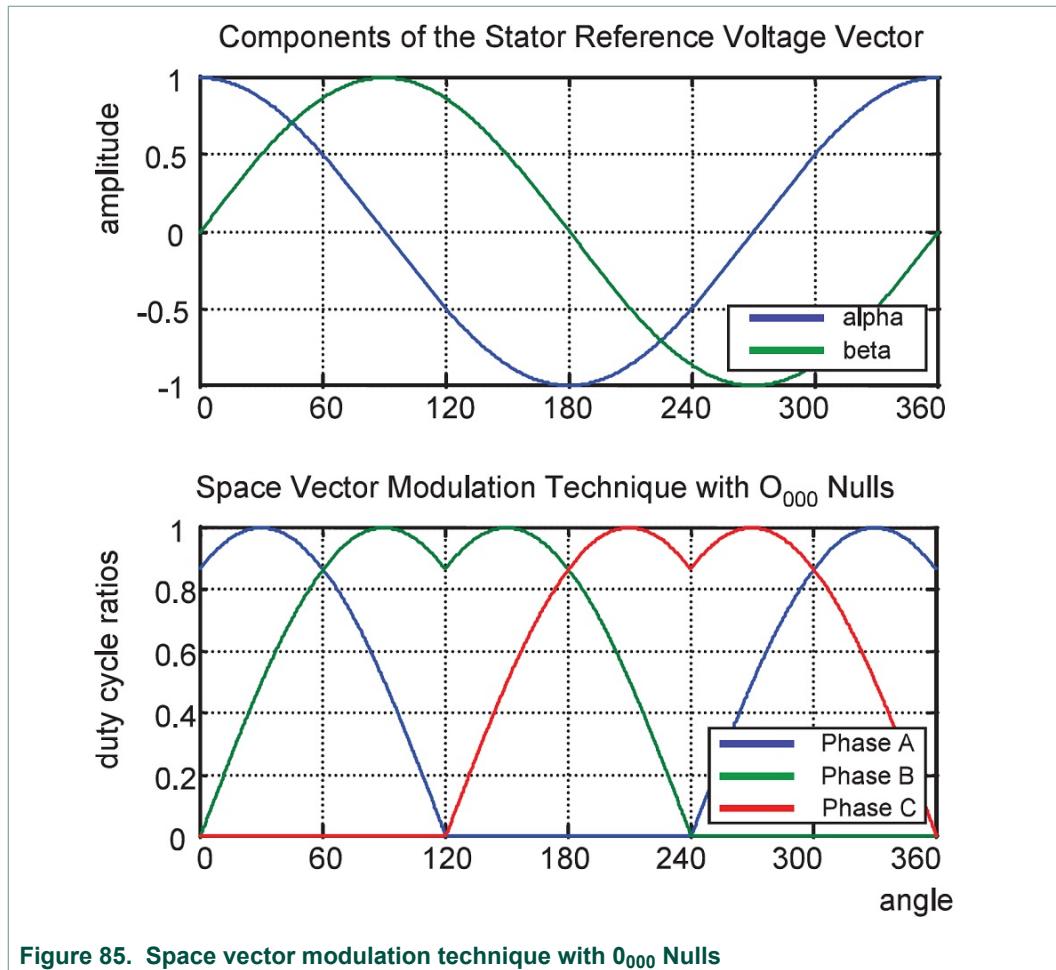
Equation GMCLIB_SvmU0n_Eq3

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 198](#).

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 84](#)



[Figure 85](#) shows calculated waveforms of the duty cycle ratios using Space Vector Modulation with O_{000} Nulls.

Figure 85. Space vector modulation technique with O_{000} Nulls

Note: The input/output pointers must contain valid addresses, otherwise an exception may occur (Data TLB Error, Data Storage, Alignment, Machine Check).

Re-entrancy

The function is re-entrant.

2.50.1 Function GMCLIB_SvmU0n_F32

Declaration

```
tU32 GMCLIB_SvmU0n_F32(SWLIBS_3Syst_F32 *pOut, const
                           SWLIBS_2Syst_F32 *const pIn);
```

Arguments

Table 202. 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_α and quadrature U_β 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);
```

2.50.2 Function GMCLIB_SvmU0n_F16**Declaration**

```
tU16 GMCLIB_SvmU0n_F16(SWLIBS_3Syst_F16 *pOut, const
SWLIBS_2Syst_F16 *const pIn);
```

Arguments**Table 203. GMCLIB_SvmUOn_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.
const SWLIBS_2Syst_F16 *const	pln	input	Pointer to the structure containing direct U_α and quadrature U_β 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);
}
```

2.51 Function GMCLIB_SvmU7n

This function calculates the duty-cycle ratios using the special Space Vector Modulation technique, termed Space Vector Modulation with O₁₁₁ 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 O₁₁₁ Ones.

The derivation approach of the Space Vector Modulation technique with O₁₁₁ Ones is identical, in many aspects, to the approach presented in [GMCLIB_SvmStd](#). However, a distinct difference lies in the definition of the variables t₁, t₂ and t₃ 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_{_1} and t_{_2} are duty-cycle ratios of basic space vectors that are defined for the respective sector in [Table 198](#).

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 86](#)

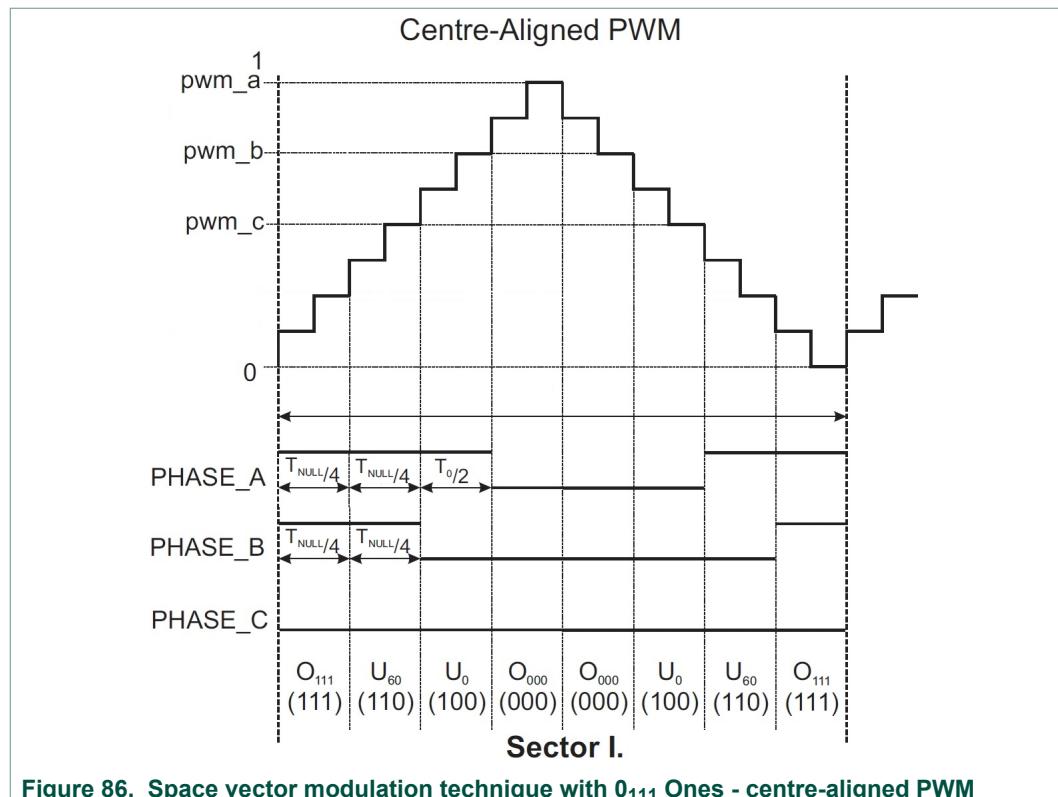
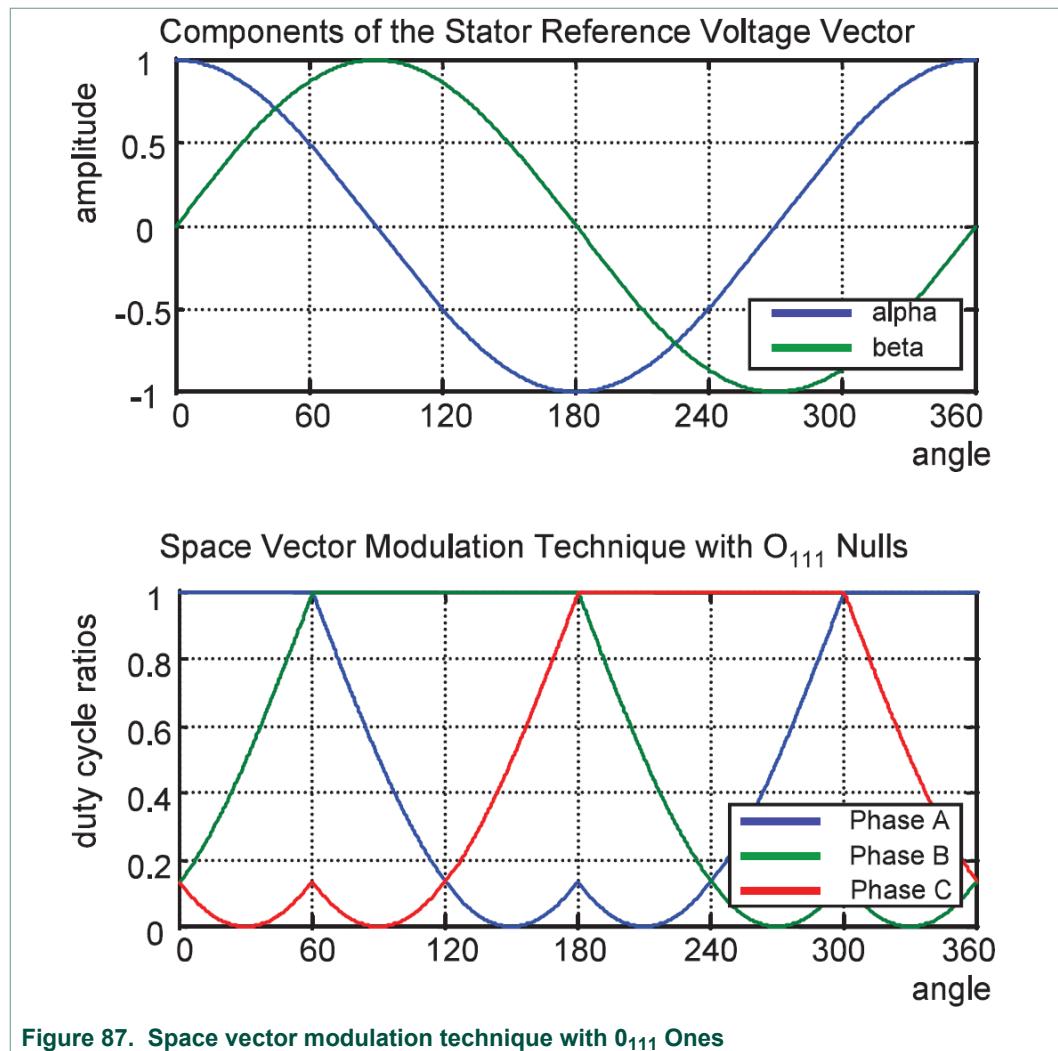


Figure 86. Space vector modulation technique with O_{111} Ones - centre-aligned PWM

Figure 87 shows calculated waveforms of the duty cycle ratios using Space Vector Modulation with O_{111} Ones.

Figure 87. Space vector modulation technique with O_{111} Ones

Note: The input/output pointers must contain valid addresses, otherwise an exception may occur (Data TLB Error, Data Storage, Alignment, Machine Check).

Re-entrancy

The function is re-entrant.

2.51.1 Function GMCLIB_SvmU7n_F32

Declaration

```
tU32 GMCLIB_SvmU7n_F32(SWLIBS\_3Syst\_F32 *pOut, const
SWLIBS\_2Syst\_F32 *const pIn);
```

Arguments

Table 204. GMCLIB_SvmU7n_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.

Type	Name	Direction	Description
const SWLIBS_2Syst_F32 *const	pln	input	Pointer to the structure containing direct U_α and quadrature U_β 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);
}
```

2.51.2 Function GMCLIB_SvmU7n_F16

Declaration

```
tU16 GMCLIB_SvmU7n_F16(SWLIBS_3Syst_F16 *pOut, const
                           SWLIBS_2Syst_F16 *const pIn);
```

Arguments

Table 205. 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_α and quadrature U_β 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);
}
```

2.52 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.

2.52.1 Function MLIB_Abs_F32

Declaration

```
INLINE tFrac32 MLIB_Abs_F32(register tFrac32 f32In);
```

Arguments

Table 206. MLIB_Abs_F32 arguments

Type	Name	Direction	Description
register tFrac32	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);
}
```

2.52.2 Function MLIB_Abs_F16

Declaration

INLINE `tFrac16` MLIB_Abs_F16(`register tFrac16 f16In`);

Arguments

Table 207. MLIB_Abs_F16 arguments

Type	Name	Direction	Description
<code>register tFrac16</code>	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);
}
```

2.53 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.

2.53.1 Function MLIB_AbsSat_F32**Declaration**

```
INLINE tFrac32 MLIB_AbsSat_F32(register tFrac32 f32In);
```

Arguments

Table 208. 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, 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);
}
```

2.53.2 Function MLIB_AbsSat_F16**Declaration**

```
INLINE tFrac16 MLIB_AbsSat_F16(register tFrac16 f16In);
```

Arguments**Table 209. MLIB_AbsSat_F16 arguments**

Type	Name	Direction	Description
register tFrac16	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, 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);
}
```

2.54 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.

2.54.1 Function MLIB_Add_F32

Declaration

```
INLINE tFrac32 MLIB_Add_F32(register tFrac32 f32In1, register  
tFrac32 f32In2);
```

Arguments

Table 210. MLIB_Add_F32 arguments

Type	Name	Direction	Description
register tFrac32	f32In1	input	First value to be add.
register tFrac32	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);
}

```

2.54.2 Function MLIB_Add_F16

Declaration

```
INLINE tFrac16 MLIB_Add_F16(register tFrac16 f16In1, register
tFrac16 f16In2);
```

Arguments

Table 211. MLIB_Add_F16 arguments

Type	Name	Direction	Description
register <u>tFrac16</u>	f16In1	input	First value to be add.
register <u>tFrac16</u>	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);
}

```

2.55 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.

2.55.1 Function `MLIB_AddSat_F32`

Declaration

```
INLINE tFrac32 MLIB_AddSat_F32(register tFrac32 f32In1, register
tFrac32 f32In2);
```

Arguments

Table 212. `MLIB_AddSat_F32` arguments

Type	Name	Direction	Description
register <code>tFrac32</code>	f32In1	input	First value to be add.
register <code>tFrac32</code>	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, 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);
}
```

2.55.2 Function MLIB_AddSat_F16

Declaration

```
INLINE tFrac16 MLIB_AddSat_F16(register tFrac16 f16In1, register
tFrac16 f16In2);
```

Arguments

Table 213. MLIB_AddSat_F16 arguments

Type	Name	Direction	Description
register tFrac16	f16In1	input	First value to be add.
register tFrac16	f16In2	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 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, 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);
}
```

2.56 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.

2.56.1 Function MLIB_Convert_F32F16

Declaration

```
INLINE tFrac32 MLIB_Convert_F32F16(register tFrac16 f16In1,  
register tFrac16 f16In2);
```

Arguments

Table 214. MLIB_Convert_F32F16 arguments

Type	Name	Direction	Description
register tFrac16	f16In1	input	Input value in 16-bit fractional format to be converted.
register tFrac16	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);
}

```

2.56.2 Function MLIB_Convert_F16F32

Declaration

```
INLINE tFrac16 MLIB_Convert_F16F32(register tFrac32 f32In1,  
register tFrac32 f32In2);
```

Arguments

Table 215. MLIB_Convert_F16F32 arguments

Type	Name	Direction	Description
register <u>tFrac32</u>	f32In1	input	Input value in 32-bit fractional format to be converted.
register <u>tFrac32</u>	f32In2	input	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);
}

```

2.57 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.

2.57.1 Function MLIB_ConvertPU_F32F16**Declaration**

```
INLINE tFrac32 MLIB_ConvertPU_F32F16(register tFrac16 f16In);
```

Arguments**Table 216. MLIB_ConvertPU_F32F16 arguments**

Type	Name	Direction	Description
register tFrac16	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);
}
```

2.57.2 Function MLIB_ConvertPU_F16F32**Declaration**

```
INLINE tFrac16 MLIB_ConvertPU_F16F32(register tFrac32 f32In);
```

Arguments

Table 217. 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);
}
```

2.58 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.

2.58.1 Function MLIB_Div_F32

Declaration

```
INLINE tFrac32 MLIB_Div_F32(register tFrac32 f32In1, register
tFrac32 f32In2);
```

Arguments**Table 218. 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);
}
```

2.58.2 Function MLIB_Div_F16

Declaration

```
INLINE tFrac16 MLIB_Div_F16(register tFrac16 f16In1, register
tFrac16 f16In2);
```

Arguments

Table 219. 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);
}
```

2.59 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.

2.59.1 Function MLIB_DivSat_F32

Declaration

```
INLINE tFrac32 MLIB_DivSat_F32(register tFrac32 f32In1, register tFrac32 f32In2);
```

Arguments

Table 220. MLIB_DivSat_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, 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);
}

```

2.59.2 Function MLIB_DivSat_F16

Declaration

```
INLINE tFrac16 MLIB_DivSat_F16(register tFrac16 f16In1, register tFrac16 f16In2);
```

Arguments

Table 221. 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);
}
```

2.60 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.

2.60.1 Function MLIB_Mac_F32**Declaration**

```
INLINE tFrac32 MLIB_Mac_F32(register tFrac32 f32In1, register
tFrac32 f32In2, register tFrac32 f32In3);
```

Arguments**Table 222. MLIB_Mac_F32 arguments**

Type	Name	Direction	Description
register tFrac32	f32In1	input	Input value to be add.

Type	Name	Direction	Description
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);
}
```

2.60.2 Function MLIB_Mac_F32F16F16

Declaration

```
INLINE tFrac32 MLIB_Mac_F32F16F16(register tFrac32 f32In1,  
register tFrac16 f16In2, register tFrac16 f16In3);
```

Arguments

Table 223. MLIB_Mac_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.

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);

}

```

2.60.3 Function MLIB_Mac_F16

Declaration

```
INLINE tFrac16 MLIB_Mac_F16(register tFrac16 f16In1, register
tFrac16 f16In2, register tFrac16 f16In3);
```

Arguments

Table 224. 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);
}

```

2.61 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.

2.61.1 Function [MLIB_MacSat_F32](#)

Declaration

```
INLINE tFrac32 MLIB\_MacSat\_F32(register tFrac32 f32In1, register
tFrac32 f32In2, register tFrac32 f32In3);
```

Arguments

Table 225. [MLIB_MacSat_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. 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);
}
```

2.61.2 Function MLIB_MacSat_F32F16F16**Declaration**

```
INLINE tFrac32 MLIB_MacSat_F32F16F16(register tFrac32 f32In1,
register tFrac16 f16In2, register tFrac16 f16In3);
```

Arguments

Table 226. 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);
}
```

2.61.3 Function MLIB_MacSat_F16

Declaration

```
INLINE tFrac16 MLIB_MacSat_F16(register tFrac16 f16In1, register
tFrac16 f16In2, register tFrac16 f16In3);
```

Arguments

Table 227. 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);
}

```

2.62 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.

2.62.1 Function `MLIB_Mnac_F32`

Declaration

```
INLINE tFrac32 MLIB_Mnac_F32(register tFrac32 f32In1, register
tFrac32 f32In2, register tFrac32 f32In3);
```

Arguments

Table 228. `MLIB_Mnac_F32` arguments

Type	Name	Direction	Description
register tFrac32	f32In1	input	Input value to be subtracted.
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 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);
}
```

2.62.2 Function MLIB_Mnac_F32F16F16

Declaration

```
INLINE tFrac32 MLIB_Mnac_F32F16F16(register tFrac32 f32In1,
register tFrac16 f16In2, register tFrac16 f16In3);
```

Arguments

Table 229. MLIB_Mnac_F32F16F16 arguments

Type	Name	Direction	Description
register tFrac32	f32In1	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 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);
}
```

2.62.3 Function MLIB_Mnac_F16

Declaration

```
INLINE tFrac16 MLIB_Mnac_F16(register tFrac16 f16In1, register
tFrac16 f16In2, register tFrac16 f16In3);
```

Arguments

Table 230. 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);
}

```

2.63 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.

2.63.1 Function `MLIB_Msu_F32`

Declaration

```
INLINE tFrac32 MLIB_Msu_F32(register tFrac32 f32In1, register
tFrac32 f32In2, register tFrac32 f32In3);
```

Arguments

Table 231. `MLIB_Msu_F32` arguments

Type	Name	Direction	Description
register tFrac32	f32In1	input	Input value from which to subtract.
register tFrac32	f32In2	input	First value to be multiplied.
register tFrac32	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);
}
```

2.63.2 Function MLIB_Msu_F32F16F16

Declaration

```
INLINE tFrac32 MLIB_Msu_F32F16F16(register tFrac32 f32In1,
register tFrac16 f16In2, register tFrac16 f16In3);
```

Arguments

Table 232. MLIB_Msu_F32F16F16 arguments

Type	Name	Direction	Description
register tFrac32	f32In1	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 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);
}
```

2.63.3 Function MLIB_Msu_F16

Declaration

```
INLINE tFrac16 MLIB_Msu_F16(register tFrac16 f16In1, register
tFrac16 f16In2, register tFrac16 f16In3);
```

Arguments

Table 233. 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);
}

```

2.64 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.

2.64.1 Function **MLIB_Mul_F32**

Declaration

```
INLINE tFrac32 MLIB_Mul_F32(register tFrac32 f32In1, register tFrac32 f32In2);
```

Arguments

Table 234. MLIB_Mul_F32 arguments

Type	Name	Direction	Description
register tFrac32	f32In1	input	Operand is a 32-bit number normalized between [-1,1].
register tFrac32	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: 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 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);
}
```

2.64.2 Function MLIB_Mul_F32F16F16

Declaration

```
INLINE tFrac32 MLIB_Mul_F32F16F16(register tFrac16 f16In1,
register tFrac16 f16In2);
```

Arguments

Table 235. 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: 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;
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);
}
```

2.64.3 Function MLIB_Mul_F16

Declaration

```
INLINE tFrac16 MLIB_Mul_F16(register tFrac16 f16In1, register
tFrac16 f16In2);
```

Arguments

Table 236. 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);
}
```

2.65 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.

2.65.1 Function MLIB_MulSat_F32

Declaration

```
INLINE tFrac32 MLIB_MulSat_F32(register tFrac32 f32In1, register tFrac32 f32In2);
```

Arguments

Table 237. MLIB_MulSat_F32 arguments

Type	Name	Direction	Description
register tFrac32	f32In1	input	Operand is a 32-bit number normalized between [-1,1].
register tFrac32	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 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);
}

```

2.65.2 Function MLIB_MulSat_F32F16F16

Declaration

```
INLINE tFrac32 MLIB_MulSat_F32F16F16(register tFrac16 f16In1,  
register tFrac16 f16In2);
```

Arguments

Table 238. MLIB_MulSat_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 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 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);
}
```

2.65.3 Function MLIB_MulSat_F16

Declaration

```
INLINE tFrac16 MLIB_MulSat_F16(register tFrac16 f16In1, register tFrac16 f16In2);
```

Arguments

Table 239. 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);
}
```

2.66 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.

2.66.1 Function MLIB_Neg_F32

Declaration

```
INLINE tFrac32 MLIB_Neg_F32(register tFrac32 f32In);
```

Arguments

Table 240. 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);
}
```

2.66.2 Function MLIB_Neg_F16

Declaration

```
INLINE tFrac16 MLIB_Neg_F16(register tFrac16 f16In);
```

Arguments

Table 241. 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);
}
```

2.67 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.

2.67.1 Function MLIB_NegSat_F32

Declaration

```
INLINE tFrac32 MLIB_NegSat_F32(register tFrac32 f32In);
```

Arguments

Table 242. MLIB_NegSat_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 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, 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);
```

```
}
```

2.67.2 Function MLIB_NegSat_F16

Declaration

```
INLINE tFrac16 MLIB_NegSat_F16(register tFrac16 f16In);
```

Arguments

Table 243. MLIB_NegSat_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 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, 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);
}
```

2.68 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.

2.68.1 Function MLIB_Norm_F32

Declaration

```
INLINE tU16 MLIB_Norm_F32(register tFrac32 f32In);
```

Arguments

Table 244. MLIB_Norm_F32 arguments

Type	Name	Direction	Description
register tFrac32	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);
}

```

2.68.2 Function `MLIB_Norm_F16`

Declaration

INLINE `tU16` `MLIB_Norm_F16(register tFrac16 f16In);`

Arguments

Table 245. `MLIB_Norm_F16` arguments

Type	Name	Direction	Description
register <code>tFrac16</code>	<code>f16In</code>	<code>input</code>	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);
}

```

2.69 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 246. MLIB_RndSat_F16F32 arguments

Type	Name	Direction	Description
register tFrac32	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);
}
```

2.70 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.

2.70.1 Function MLIB_Round_F32

Declaration

```
INLINE tFrac32 MLIB_Round_F32(register tFrac32 f32In1, register  
tU16 u16In2);
```

Arguments

Table 247. MLIB_Round_F32 arguments

Type	Name	Direction	Description
register tFrac32	f32In1	input	The value to be rounded.
register tU16	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);
}

```

2.70.2 Function MLIB_Round_F16

Declaration

```
INLINE tFrac16 MLIB_Round_F16(register tFrac16 f16In1, register
tU16 u16In2);
```

Arguments

Table 248. MLIB_Round_F16 arguments

Type	Name	Direction	Description
register tFrac16	f16In1	input	The value to be rounded.
register tU16	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);
}
```

2.71 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.

2.71.1 Function MLIB_ShBi_F32

Declaration

```
INLINE tFrac32 MLIB_ShBi_F32(register tFrac32 f32In1, register
tS16 s16In2);
```

Arguments

Table 249. MLIB_ShBi_F32 arguments

Type	Name	Direction	Description
register tFrac32	f32In1	input	First value to be shift.
register tS16	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_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);
}
```

2.71.2 Function MLIB_ShBi_F16

Declaration

```
INLINE tFrac16 MLIB_ShBi_F16(register tFrac16 f16In1, register
ts16 s16In2);
```

Arguments

Table 250. MLIB_ShBi_F16 arguments

Type	Name	Direction	Description
register tFrac16	f16In1	input	First value to be left shift.
register ts16	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);
}
```

2.72 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.

2.72.1 Function MLIB_ShBiSat_F32

Declaration

```
INLINE tFrac32 MLIB_ShBiSat_F32(register tFrac32 f32In1, register tS16 s16In2);
```

Arguments

Table 251. MLIB_ShBiSat_F32 arguments

Type	Name	Direction	Description
register tFrac32	f32In1	input	First value to be shift.
register tS16	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);
}

```

2.72.2 Function **MLIB_ShBiSat_F16**

Declaration

```
INLINE tFrac16 MLIB_ShBiSat_F16(register tFrac16 f16In1, register
ts16 s16In2);
```

Arguments

Table 252. MLIB_ShBiSat_F16 arguments

Type	Name	Direction	Description
register tFrac16	f16In1	input	First value to be left shift.
register ts16	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);
}

```

2.73 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.

2.73.1 Function MLIB_ShL_F32

Declaration

```
INLINE tFrac32 MLIB_ShL_F32(register tFrac32 f32In1, register tU16 u16In2);
```

Arguments

Table 253. 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);
}

```

2.73.2 Function `MLIB_ShL_F16`

Declaration

```
INLINE tFrac16 MLIB_ShL_F16(register tFrac16 f16In1, register
tU16 u16In2);
```

Arguments

Table 254. `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);
}

```

2.74 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.

2.74.1 Function MLIB_ShLSat_F32

Declaration

```
INLINE tFrac32 MLIB_ShLSat_F32(register tFrac32 f32In1, register
tU16 u16In2);
```

Arguments

Table 255. 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, 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);
}
```

2.74.2 Function MLIB_ShLSat_F16

Declaration

```
INLINE tFrac16 MLIB_ShLSat_F16(register tFrac16 f16In1, register
tU16 u16In2);
```

Arguments

Table 256. 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, 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);
}
```

2.75 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.

2.75.1 Function MLIB_ShR_F32

Declaration

```
INLINE tFrac32 MLIB_ShR_F32(register tFrac32 f32In1, register
tU16 u16In2);
```

Arguments**Table 257. MLIB_ShR_F32 arguments**

Type	Name	Direction	Description
register tFrac32	f32In1	input	First value to be right shift.
register tU16	u16In2	input	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);
}
```

2.75.2 Function MLIB_ShR_F16**Declaration**

```
INLINE tFrac16 MLIB_ShR_F16(register tFrac16 f16In1, register
tU16 u16In2);
```

Arguments**Table 258. MLIB_ShR_F16 arguments**

Type	Name	Direction	Description
register tFrac16	f16In1	input	First value to be right shift.
register tU16	u16In2	input	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);
}
```

2.76 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.

2.76.1 Function MLIB_Sub_F32

Declaration

```
INLINE tFrac32 MLIB_Sub_F32(register tFrac32 f32In1, register  
tFrac32 f32In2);
```

Arguments

Table 259. MLIB_Sub_F32 arguments

Type	Name	Direction	Description
register tFrac32	f32In1	input	Operand is a 32-bit number normalized between[-1,1).
register tFrac32	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);
}

```

2.76.2 Function MLIB_Sub_F16

Declaration

```
INLINE tFrac16 MLIB_Sub_F16(register tFrac16 f16In1, register tFrac16 f16In2);
```

Arguments

Table 260. MLIB_Sub_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

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);
}

```

2.77 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.

2.77.1 Function [MLIB_SubSat_F32](#)

Declaration

```
INLINE tFrac32 MLIB\_SubSat\_F32(register tFrac32 f32In1, register tFrac32 f32In2);
```

Arguments

Table 261. [MLIB_SubSat_F32](#) arguments

Type	Name	Direction	Description
register tFrac32	f32In1	input	Operand is a 32-bit number normalized between [-1,1].
register tFrac32	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 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);
}
```

2.77.2 Function MLIB_SubSat_F16

Declaration

```
INLINE tFrac16 MLIB_SubSat_F16(register tFrac16 f16In1, register
tFrac16 f16In2);
```

Arguments

Table 262. MLIB_SubSat_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

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);
}
```

2.78 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.

2.78.1 Function MLIB_VMac_F32

Declaration

```
INLINE tFrac32 MLIB_VMac_F32(register tFrac32 f32In1, register tFrac32 f32In2, register tFrac32 f32In3, register tFrac32 f32In4);
```

Arguments

Table 263. MLIB_VMac_F32 arguments

Type	Name	Direction	Description
register tFrac32	f32In1	input	First input value to first multiplication.
register tFrac32	f32In2	input	Second input value to first multiplication.
register tFrac32	f32In3	input	First input value to second multiplication.
register tFrac32	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);
}

```

2.78.2 Function [MLIB_VMac_F32F16F16](#)

Declaration

```
INLINE tFrac32 MLIB\_VMac\_F32F16F16(register tFrac16 f16In1,  
register tFrac16 f16In2, register tFrac16 f16In3, register  
tFrac16 f16In4);
```

Arguments

Table 264. [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);
}
```

2.78.3 Function MLIB_VMac_F16

Declaration

```
INLINE tFrac16 MLIB_VMac_F16(register tFrac16 f16In1, register
tFrac16 f16In2, register tFrac16 f16In3, register tFrac16
f16In4);
```

Arguments

Table 265. MLIB_VMac_F16 arguments

Type	Name	Direction	Description
register tFrac16	f16In1	input	First input value to first multiplication.

Type	Name	Direction	Description
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 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);
}
```

2.79 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 266. 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 S32K11x 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 267. 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 268. 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 S32K11x devices.

5.1 AMCLIB_BEMF_OBSRV_DQ_T_F16

Observer configuration structure.

Source File

```
#include <AMCLIB_BemfObsrvDQ.h>
```

Compound Type Members**Table 269. AMCLIB_BEMF_OBSRV_DQ_T_F16 members description**

Type	Name	Description
SWLIBS_2Syst_F16	pEObsrv	Estimated BEMF - D/Q.
SWLIBS_2Syst_F32	pIObsrv	Estimated current - D/Q.
GFLIB_CONTROLLER_PIAW_R_T_F16	pParamD	Observer parameters for D-axis controller.
GFLIB_CONTROLLER_PIAW_R_T_F16	pParamQ	Observer parameters for Q-axis controller.
SWLIBS_2Syst_F32	pIObsrvln_1L	Inputs of RL circuit at step k-1 - low word
SWLIBS_2Syst_F16	pIObsrvln_1H	Inputs of RL circuit at step k-1 - high word
tFrac16	f16IGain	Scaled RL circuit constant, fractional format normalized to fit into (- 2^{15} , $2^{15}-1$).
tFrac16	f16UGain	Scaled voltage cross-coupling constant, fractional format normalized to fit into (- 2^{15} , $2^{15}-1$).
tFrac16	f16WIGain	Scaled angular velocity cross-coupling constant, fractional format normalized to fit into (- 2^{15} , $2^{15}-1$).
tFrac16	f16EGain	Scaled back-EMF cross-coupling constant, fractional format normalized to fit into (- 2^{15} , $2^{15}-1$).
tS16	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 270. AMCLIB_BEMF_OBSRV_DQ_T_F32 members description**

Type	Name	Description
SWLIBS_2Syst_F32	pEObsrv	Estimated BEMF - D/Q.
SWLIBS_2Syst_F32	pIObsrv	Estimated current - D/Q.
GFLIB_CONTROLLER_PIAW_R_T_F32	pParamD	Observer parameters for D-axis controller.
GFLIB_CONTROLLER_PIAW_R_T_F32	pParamQ	Observer parameters for Q-axis controller.

Type	Name	Description
SWLIBS_2Syst_F32	pIObsrvln_1L	Inputs of RL circuit at step k-1 - low word
SWLIBS_2Syst_F16	pIObsrvln_1H	Inputs of RL circuit at step k-1 - high word
tFrac32	f32IGain	Scaled RL circuit constant, fractional format normalized to fit into (- 2^{31} , $2^{31}-1$).
tFrac32	f32UGain	Scaled voltage cross-coupling constant, fractional format normalized to fit into (- 2^{31} , $2^{31}-1$).
tFrac32	f32WIGain	Scaled angular velocity cross-coupling constant, fractional format normalized to fit into (- 2^{31} , $2^{31}-1$).
tFrac32	f32EGain	Scaled back-EMF cross-coupling constant, fractional format normalized to fit into (- 2^{31} , $2^{31}-1$).
tS16	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_CURRENT_LOOP_T_F16

CurrentLoop configuration structure.

Source File

```
#include <AMCLIB_CurrentLoop.h>
```

Compound Type Members

Table 271. AMCLIB_CURRENT_LOOP_T_F16 members description

Type	Name	Description
GFLIB_CONTROLLER_PIAW_R_T_F16	pPIrAWD	D-axis ControllerPIrAW paremeters structure.
GFLIB_CONTROLLER_PIAW_R_T_F16	pPIrAWQ	Q-axis ControllerPIrAW paremeters 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.

5.4 AMCLIB_CURRENT_LOOP_T_F32

CurrentLoop configuration structure.

Source File

```
#include <AMCLIB_CurrentLoop.h>
```

Compound Type Members

Table 272. AMCLIB_CURRENT_LOOP_T_F32 members description

Type	Name	Description
GFLIB_CONTROLLER_PIAW_R_T_F32	pPIrAWD	D-axis ControllerPIrAW paremeters structure.
GFLIB_CONTROLLER_PIAW_R_T_F32	pPIrAWQ	Q-axis ControllerPIrAW paremeters 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.

5.5 AMCLIB_FW_DEBUG_T_F16

FW configuration structure with debugging information.

Source File

```
#include <AMCLIB_FW.h>
```

Compound Type Members

Table 273. 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 AMCLIB_CURRENT_LOOP_T_F16 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.

5.6 AMCLIB_FW_DEBUG_T_F32

FW configuration structure with debugging information.

Source File

```
#include <AMCLIB_FW.h>
```

Compound Type Members

Table 274. 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.

5.7 AMCLIB_FW_SPEED_LOOP_DEBUG_T_F16

FWSpeedLoop configuration structure with debugging information.

Source File

```
#include <AMCLIB_FWSpeedLoop.h>
```

Compound Type Members

Table 275. AMCLIB_FW_SPEED_LOOP_DEBUG_T_F16 members description

Type	Name	Description
GDFLIB_FILTER_MA_T_F16	pFilterW	Velocity FilterMA parameters structure.
GDFLIB_FILTER_MA_T_F16	pFilterFW	Field weakening angle FilterMA parameters structure.
GFLIB_CONTROLLER_PIAW_P_T_F16	pPIpAWQ	Q-axis ControllerPIpAW parameters structure.
GFLIB_CONTROLLER_PIAW_P_T_F16	pPIpAWFW	Field weakening angle ControllerPIpAW parameters structure.
GFLIB_RAMP_T_F32	pRamp	Speed ramp parameters 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 AMCLIB_CURRENT_LOOP_T_F16 structure element pPIrAWQ.f16UpperLimit). The limit must be a positive value.
tFrac16	f16WReqFilt	SpeedLoop - Filtered value of the required angular velocity.
tFrac16	f16WErr	SpeedLoop - Velocity deviation.
tFrac16	f16IDQReqAmp	SpeedLoop - Filtered value of the measured angular velocity.
tFrac16	f16WFbckFilt	SpeedLoop - Required current amplitude.
tFrac16	f16IQErrSign	FW - Current deviation after sign elimination.
tFrac16	f16FWErr	FW - Current deviation.
tFrac16	f16IQErr	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.
tFrac16	f16FWSin	FW - Q-axis unity current phasor component.
tFrac16	f16FWCos	FW - D-axis unity current phasor component.

5.8 AMCLIB_FW_SPEED_LOOP_DEBUG_T_F32

FWSpeedLoop configuration structure with debugging information.

Source File

```
#include <AMCLIB_FWSpeedLoop.h>
```

Compound Type Members

Table 276. AMCLIB_FW_SPEED_LOOP_DEBUG_T_F32 members description

Type	Name	Description
GDFLIB_FILTER_MA_T_F32	pFilterW	Velocity FilterMA parameters structure.
GDFLIB_FILTER_MA_T_F32	pFilterFW	Field weakening angle FilterMA parameters structure.
GFLIB_CONTROLLER_PIAW_P_T_F32	pPIpAWQ	Q-axis ControllerPIpAW parameters structure.
GFLIB_CONTROLLER_PIAW_P_T_F32	pPIpAWFW	Field weakening angle ControllerPIpAW parameters structure.
GFLIB_RAMP_T_F32	pRamp	Speed ramp 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 (should point to the AMCLIB_CURRENT_LOOP_T_F32 structure element pPIrAWQ.f32UpperLimit). The limit must be a positive value.
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.
tFrac32	f32IDQReqAmp	SpeedLoop - Required current amplitude.
tFrac32	f32IQErrSign	FW - Current deviation after sign elimination.
tFrac32	f32FWErr	FW - Current deviation.
tFrac32	f32IQErr	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_SPEED_LOOP_T_F16

FWSpeedLoop configuration structure.

Source File

```
#include <AMCLIB_FWSpeedLoop.h>
```

Compound Type Members**Table 277. AMCLIB_FW_SPEED_LOOP_T_F16 members description**

Type	Name	Description
GDFLIB_FILTER_MA_T_F16	pFilterW	Velocity FilterMA paremeters structure.
GDFLIB_FILTER_MA_T_F16	pFilterFW	Field weakening angle FilterMA paremeters structure.
GFLIB_CONTROLLER_PIAW_P_T_F16	pPIpAWQ	Q-axis ControllerPIpAW paremeters structure.
GFLIB_CONTROLLER_PIAW_P_T_F16	pPIpAWFW	Field weakening angle ControllerPIpAW paremeters structure.
GFLIB_RAMP_T_F32	pRamp	Speed ramp 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 AMCLIB_CURRENT_LOOP_T_F16 structure element pPIrAWQ.f16UpperLimit). The limit must be a positive value.

5.10 AMCLIB_FW_SPEED_LOOP_T_F32

FWSpeedLoop configuration structure.

Source File

#include <AMCLIB_FWSpeedLoop.h>

Compound Type Members**Table 278. AMCLIB_FW_SPEED_LOOP_T_F32 members description**

Type	Name	Description
GDFLIB_FILTER_MA_T_F32	pFilterW	Velocity FilterMA paremeters structure.
GDFLIB_FILTER_MA_T_F32	pFilterFW	Field weakening angle FilterMA paremeters structure.
GFLIB_CONTROLLER_PIAW_P_T_F32	pPIpAWQ	Q-axis ControllerPIpAW paremeters structure.
GFLIB_CONTROLLER_PIAW_P_T_F32	pPIpAWFW	Field weakening angle ControllerPIpAW paremeters structure.
GFLIB_RAMP_T_F32	pRamp	Speed ramp paremeters structure.
tFrac32 *	pIQFbck	Pointer to the q-axis component of the feedback current in the two-phase rotational orthogonal system (d-q).

Type	Name	Description
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 AMCLIB_CURRENT_LOOP_T_F32 structure element pPIrAWQ.f32UpperLimit). The limit must be a positive value.

5.11 AMCLIB_FW_T_F16

FW configuration structure.

Source File

```
#include <AMCLIB_FW.h>
```

Compound Type Members

Table 279. AMCLIB_FW_T_F16 members description

Type	Name	Description
GDFLIB_FILTER_MA_T_F16	pFilterFW	Field weakening angle FilterMA parameters structure.
GFLIB_CONTROLLER_PIAW_P_T_F16	pPIpAWFW	Field weakening angle ControllerPIpAW parameters 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. Must be a positive value.

5.12 AMCLIB_FW_T_F32

FW configuration structure.

Source File

```
#include <AMCLIB_FW.h>
```

Compound Type Members

Table 280. AMCLIB_FW_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.

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. Must be a positive value.

5.13 AMCLIB_SPEED_LOOP_DEBUG_T_F16

SpeedLoop configuration structure with debugging information.

Source File

```
#include <AMCLIB_SpeedLoop.h>
```

Compound Type Members

Table 281. AMCLIB_SPEED_LOOP_DEBUG_T_F16 members description

Type	Name	Description
GDFLIB_FILTER_MA_T_F16	pFilterW	Velocity FilterMA paremeters structure.
GFLIB_CONTROLLER_PIAW_P_T_F16	pPIpAWQ	Velocity ControllerPIpAW paremeters structure.
GFLIB_RAMP_T_F32	pRamp	Speed ramp paremeters structure.
tFrac16	f16WReqFilt	SpeedLoop - Filtered value of the required angular velocity.
tFrac16	f16WErr	SpeedLoop - Velocity deviation.
tFrac16	f16WFbckFilt	SpeedLoop - Filtered value of the measured angular velocity.

5.14 AMCLIB_SPEED_LOOP_DEBUG_T_F32

SpeedLoop configuration structure with debugging information.

Source File

```
#include <AMCLIB_SpeedLoop.h>
```

Compound Type Members

Table 282. 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	pPIpAWQ	Velocity ControllerPIpAW paremeters structure.
GFLIB_RAMP_T_F32	pRamp	Speed ramp paremeters structure.

Type	Name	Description
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.

5.15 AMCLIB_SPEED_LOOP_T_F16

SpeedLoop configuration structure.

Source File

```
#include <AMCLIB_SpeedLoop.h>
```

Compound Type Members

Table 283. AMCLIB_SPEED_LOOP_T_F16 members description

Type	Name	Description
GDFLIB_FILTER_MA_T_F16	pFilterW	Velocity FilterMA paremeters structure.
GFLIB_CONTROLLER_PIAW_P_T_F16	pPIpAWQ	Velocity ControllerPIpAW paremeters structure.
GFLIB_RAMP_T_F32	pRamp	Speed ramp paremeters structure.

5.16 AMCLIB_SPEED_LOOP_T_F32

SpeedLoop configuration structure.

Source File

```
#include <AMCLIB_SpeedLoop.h>
```

Compound Type Members

Table 284. AMCLIB_SPEED_LOOP_T_F32 members description

Type	Name	Description
GDFLIB_FILTER_MA_T_F32	pFilterW	Velocity FilterMA paremeters structure.
GFLIB_CONTROLLER_PIAW_P_T_F32	pPIpAWQ	Velocity ControllerPIpAW paremeters structure.
GFLIB_RAMP_T_F32	pRamp	Speed ramp paremeters structure.

5.17 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 285.** AMCLIB_TRACK_OBSRV_T_F16 members description

Type	Name	Description
GFLIB_CONTROLLER_PIAW_R_T_F16	pParamPI	Observer PIrAW controller parameters.
GFLIB_INTEGRATOR_TR_T_F16	pParamInteg	Observer integrator parameters.

5.18 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 286.** AMCLIB_TRACK_OBSRV_T_F32 members description

Type	Name	Description
GFLIB_CONTROLLER_PIAW_R_T_F32	pParamPI	Observer PIrAW controller parameters.
GFLIB_INTEGRATOR_TR_T_F32	pParamInteg	Observer integrator parameters.

5.19 AMCLIB_WINDMILLING_T_F16

Observer configuration structure.

Source File

```
#include <AMCLIB_Windmilling.h>
```

Compound Type Members**Table 287.** AMCLIB_WINDMILLING_T_F16 members description

Type	Name	Description
tFrac16	f16ADCComp	A/D converter error compensation constant. This parameter is initialized by AMCLIB_WindmillingInit_F16 based on user's input.
AMCLIB_TRACK_OBSRV_T_F16	pParamATO	PI controller and integrator parameters and state. These parameters must be configured by the user before the first call of AMCLIB_Windmilling_F16 .
GDFLIB_FILTER_MA_T_F16	pDetMA	PLL lock detector internal state. Do not change. This sub-structure and all its parameters are automatically initialized by AMCLIB_WindmillingInit_F16 .

Type	Name	Description
tU8	u8DetNoiseCnt	PLL lock detector internal state. Do not change. This parameter is automatically initialized by AMCLIB_WindmillingInit_F16 .
tU8	u8DetSpin	PLL lock detector internal state. Do not change. This parameter is automatically initialized by AMCLIB_WindmillingInit_F16 .
tU16	u16DetHystCnt	PLL lock detector internal state. Do not change. This parameter is automatically initialized by AMCLIB_WindmillingInit_F16 .
tU16	u16DetStopCnt	PLL lock detector internal state. Do not change. This parameter is automatically initialized by AMCLIB_WindmillingInit_F16 .

5.20 AMCLIB_WINDMILLING_T_F32

Observer configuration structure.

Source File

```
#include <AMCLIB_Windmilling.h>
```

Compound Type Members

Table 288. AMCLIB_WINDMILLING_T_F32 members description

Type	Name	Description
tFrac32	f32ADCComp	A/D converter error compensation constant. This parameter is initialized by AMCLIB_WindmillingInit_F32 based on user's input.
AMCLIB_TRACK_OBSRV_T_F32	pParamATO	PI controller and integrator parameters and state. These parameters must be configured by the user before the first call of AMCLIB_Windmilling_F32 .
GDFLIB_FILTER_MA_T_F32	pDetMA	PLL lock detector internal state. Do not change. This sub-structure and all its parameters are automatically initialized by AMCLIB_WindmillingInit_F32 .
tU8	u8DetNoiseCnt	PLL lock detector internal state. Do not change. This parameter is automatically initialized by AMCLIB_WindmillingInit_F32 .
tU8	u8DetSpin	PLL lock detector internal state. Do not change. This parameter is automatically initialized by AMCLIB_WindmillingInit_F32 .
tU16	u16DetHystCnt	PLL lock detector internal state. Do not change. This parameter is automatically initialized by AMCLIB_WindmillingInit_F32 .
tU16	u16DetStopCnt	PLL lock detector internal state. Do not change. This parameter is automatically initialized by AMCLIB_WindmillingInit_F32 .

5.21 GDFLIB_FILTER_IIR1_COEFF_T_F16

Sub-structure containing filter coefficients.

Source File

```
#include <GDFLIB_FilterIIR1.h>
```

Compound Type Members

Table 289. 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^{15} , $2^{15}-1$).
tFrac16	f16B1	B1 coefficient of an IIR1 filter, fractional format normalized to fit into (- 2^{15} , $2^{15}-1$).
tFrac16	f16A1	A1 coefficient of an IIR1 filter, fractional format normalized to fit into (- 2^{15} , $2^{15}-1$).

5.22 GDFLIB_FILTER_IIR1_COEFF_T_F32

Sub-structure containing filter coefficients.

Source File

```
#include <GDFLIB_FilterIIR1.h>
```

Compound Type Members

Table 290. 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^{31} , $2^{31}-1$).
tFrac32	f32B1	B1 coefficient of an IIR1 filter, fractional format normalized to fit into (- 2^{31} , $2^{31}-1$).
tFrac32	f32A1	A1 coefficient of an IIR1 filter, fractional format normalized to fit into (- 2^{31} , $2^{31}-1$).

5.23 GDFLIB_FILTER_IIR1_T_F16

Structure containing filter buffer and coefficients.

Source File

```
#include <GDFLIB_FilterIIR1.h>
```

Compound Type Members**Table 291.** 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} , $2^{15}-1$).

5.24 GDFLIB_FILTER_IIR1_T_F32

Structure containing filter buffer and coefficients.

Source File

```
#include <GDFLIB_FilterIIR1.h>
```

Compound Type Members**Table 292.** 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^{31} , $2^{31}-1$).
tFrac32	f32FiltBufferY	Internal accumulator buffer, fractional format normalized to fit into (- 2^{31} , $2^{31}-1$).

5.25 GDFLIB_FILTER_IIR2_COEFF_T_F16

Sub-structure containing filter coefficients.

Source File

```
#include <GDFLIB_FilterIIR2.h>
```

Compound Type Members**Table 293.** 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^{15} , $2^{15}-1$).
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} , $2^{15}-1$).

Type	Name	Description
tFrac16	f16A2	A2 coefficient of an IIR2 filter, fractional format normalized to fit into (- 2^{15} , $2^{15}-1$).

5.26 GDFLIB_FILTER_IIR2_COEFF_T_F32

Sub-structure containing filter coefficients.

Source File

```
#include <GDFLIB_FilterIIR2.h>
```

Compound Type Members

Table 294. 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} , $2^{31}-1$).

5.27 GDFLIB_FILTER_IIR2_T_F16

Structure containing filter buffer and coefficients.

Source File

```
#include <GDFLIB_FilterIIR2.h>
```

Compound Type Members

Table 295. 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^{15} , $2^{15}-1$).
tFrac32	f32FiltBufferY	Internal accumulator buffer, fractional format normalized to fit into (- 2^{15} , $2^{15}-1$).

5.28 GDFLIB_FILTER_IIR2_T_F32

Structure containing filter buffer and coefficients.

Source File

```
#include <GDFLIB_FilterIIR2.h>
```

Compound Type Members**Table 296. 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 ³¹ , 2 ³¹ -1).
tFrac32	f32FiltBufferY	Internal accumulator buffer, fractional format normalized to fit into (- 2 ³¹ , 2 ³¹ -1).

5.29 GDFLIB_FILTER_MA_T_F16

Structure containing filter buffer and coefficients.

Source File

```
#include <GDFLIB_FilterMA.h>
```

Compound Type Members**Table 297. GDFLIB_FILTER_MA_T_F16 members description**

Type	Name	Description
tFrac32	f32Acc	State variable - filter accumulator.
tU16	u16NSamples	Recalculated smoothing factor [0, 15].

5.30 GDFLIB_FILTER_MA_T_F32

Structure containing filter buffer and coefficients.

Source File

```
#include <GDFLIB_FilterMA.h>
```

Compound Type Members**Table 298. GDFLIB_FILTER_MA_T_F32 members description**

Type	Name	Description
tFrac32	f32Acc	State variable - filter accumulator.
tU16	u16NSamples	Recalculated smoothing factor [0, 31].

5.31 GDFLIB_FILTERFIR_PARAM_T_F16

Structure containing parameters of the filter.

Source File

```
#include <GDFLIB_FilterFIR.h>
```

Compound Type Members**Table 299. GDFLIB_FILTERFIR_PARAM_T_F16 members description**

Type	Name	Description
tU16	u16Order	FIR filter order, must be in the interval [1, 32767].
const tFrac16 *	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.

5.32 GDFLIB_FILTERFIR_PARAM_T_F32

Structure containing parameters of the filter.

Source File

```
#include <GDFLIB_FilterFIR.h>
```

Compound Type Members**Table 300. GDFLIB_FILTERFIR_PARAM_T_F32 members description**

Type	Name	Description
tU32	u32Order	FIR filter order, must be in the interval [1, 32767].
const tFrac32 *	pCoefBuf	FIR filter coefficients buffer. The array stores (u32Order + 1) filter coefficients.

5.33 GDFLIB_FILTERFIR_STATE_T_F16

Structure containing the current state of the filter.

Source File

```
#include <GDFLIB_FilterFIR.h>
```

Compound Type Members**Table 301. 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.

5.34 GDFLIB_FILTERFIR_STATE_T_F32

Structure containing the current state of the filter.

Source File

```
#include <GDFLIB_FilterFIR.h>
```

Compound Type Members

Table 302. 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.

5.35 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 303. GFLIB_ACOS_T_F16 members description

Type	Name	Description
GFLIB_ACOS_TAYLOR_COEF_T_F16	GFLIB_ACOS_SECTOR	Array of two elements for storing two sub-arrays (each sub-array contains five 16-bit coefficients) for all sub-intervals.

5.36 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 304. GFLIB_ACOS_T_F32 members description

Type	Name	Description
GFLIB_ACOS_TAYLOR_COEF_T_F32	GFLIB_ACOS_SECTOR	Array of two elements for storing three sub-arrays (each sub-array contains five 32-bit coefficients) for all sub-intervals.

5.37 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 305. GFLIB_ACOS_TAYLOR_COEF_T_F16 members description

Type	Name	Description
const tFrac16	f16A	Array of five 16-bit elements for storing coefficients of the piece-wise polynomial.

5.38 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 306.** [GFLIB_ACOS_TAYLOR_COEF_T_F32](#) members description

Type	Name	Description
const tFrac32	f32A	Array of five 32-bit elements for storing coefficients of the piece-wise polynomial.

5.39 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 307.** [GFLIB_ASIN_T_F16](#) members description

Type	Name	Description
GFLIB_ASIN_TAYLOR_COEF_T_F16	GFLIB_ASIN_SECTOR	Default approximation coefficients datatype for arcsine approximation.

5.40 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 308.** [GFLIB_ASIN_T_F32](#) members description

Type	Name	Description
GFLIB_ASIN_TAYLOR_COEF_T_F32	GFLIB_ASIN_SECTOR	Default approximation coefficients datatype for arcsine approximation.

5.41 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 309. GFLIB_ASIN_TAYLOR_COEF_T_F16 members description

Type	Name	Description
const tFrac16	f16A	Array of approximation coefficients for piece-wise polynomial.

5.42 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 310. GFLIB_ASIN_TAYLOR_COEF_T_F32 members description

Type	Name	Description
const tFrac32	f32A	Array of five 32-bit elements for storing coefficients of the piece-wise polynomial.

5.43 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 311. GFLIB_ATAN_T_F16 members description

Type	Name	Description
const GFLIB_ATAN_TAYLOR_COEF_T_F16	GFLIB_ATAN_SECTOR	Structure containing eight sub-structures with polynomial coefficients to cover all sub-intervals.

5.44 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 312. GFLIB_ATAN_T_F32 members description

Type	Name	Description
const GFLIB_ATAN_TAYLOR_COEF_T_F32	GFLIB_ATAN_SECTOR	Structure containing eight sub-structures with polynomial coefficients to cover all sub-intervals.

5.45 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 313.** `GFLIB_ATAN_TAYLOR_COEF_T_F16` members description

Type	Name	Description
const tFrac16	f16A	Array of polynomial approximation coefficients for one sub-interval.

5.46 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 314.** `GFLIB_ATAN_TAYLOR_COEF_T_F32` members description

Type	Name	Description
const tFrac32	f32A	Array of minimax polynomial approximation coefficients for one sub-interval.

5.47 GFLIB_ATANYXSHIFTED_T_F16

Structure containing the parameter for the AtanYXShifted function.

Source File

```
#include <GFLIB_AtanYXShifted.h>
```

Compound Type Members**Table 315.** `GFLIB_ATANYXSHIFTED_T_F16` members description

Type	Name	Description
tFrac16	f16Ky	Multiplication coefficient for the y-signal.
tFrac16	f16Kx	Multiplication coefficient for the x-signal.
tS16	s16Ny	Scaling coefficient for the y-signal.
tS16	s16Nx	Scaling coefficient for the x-signal.
tFrac16	f16ThetaAdj	Adjusting angle.

5.48 GFLIB_ATANYXSHIFTED_T_F32

Structure containing the parameter for the AtanYXShifted function.

Source File

```
#include <GFLIB_AtanYXShifted.h>
```

Compound Type Members**Table 316.** GFLIB_ATANYXSHIFTED_T_F32 members description

Type	Name	Description
tFrac32	f32Ky	Multiplication coefficient for the y-signal.
tFrac32	f32Kx	Multiplication coefficient for the x-signal.
tS32	s32Ny	Scaling coefficient for the y-signal.
tS32	s32Nx	Scaling coefficient for the x-signal.
tFrac32	f32ThetaAdj	Adjusting angle.

5.49 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 317.** 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.

5.50 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 318. 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.

5.51 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 319. 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].

5.52 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 320. `GFLIB_CONTROLLER_PI_R_T_F32` members description

Type	Name	Description
tFrac32	f32CC1sc	CC1 coefficient, fractional format normalized to fit into (- 2^{31} , $2^{31}-1$).
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].

5.53 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 321. `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} , $2^{15}-1$).
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.

5.54 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 322.** 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.

5.55 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 323.** GFLIB_CONTROLLER_PIAW_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.
tFrac16	f16UpperLimit	Upper Limit of the controller, fractional format normalized to fit into (- 2^{15} , $2^{15}-1$).

Type	Name	Description
tFrac16	f16LowerLimit	Lower Limit of the controller, fractional format normalized to fit into (- 2^{15} , $2^{15}-1$).
tU16	u16NShift	Scaling bitwise shift applied to the controller coefficients, integer format [0, 15].

5.56 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 324. GFLIB_CONTROLLER_PIAW_R_T_F32 members description

Type	Name	Description
tFrac32	f32CC1sc	CC1 coefficient, fractional format normalized to fit into (- 2^{31} , $2^{31}-1$).
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.
tFrac32	f32UpperLimit	Upper Limit of the controller, fractional format normalized to fit into (- 2^{31} , $2^{31}-1$).
tFrac32	f32LowerLimit	Lower Limit of the controller, fractional format normalized to fit into (- 2^{31} , $2^{31}-1$).
tU16	u16NShift	Scaling bitwise shift applied to the controller coefficients, integer format [0, 31].

5.57 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 325. GFLIB_CONTROLLER_PID_P_AW_T_F16 members description

Type	Name	Description
tFrac16	f16PropGain	Proportional Gain, fractional format normalized to fit into (0, $2^{15}-1$).

Type	Name	Description
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.

5.58 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 326. 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].

Type	Name	Description
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.

5.59 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 327. GFLIB_COS_T_F16 members description

Type	Name	Description
tFrac16	f16A	Array of four 16-bit elements for storing coefficients of the Taylor polynomial.

5.60 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 328. GFLIB_COS_T_F32 members description

Type	Name	Description
tFrac32	f32A	Array of five 32-bit elements for storing coefficients of the Taylor polynomial.

5.61 GFLIB_HYST_T_F16

Structure containing parameters and states for the hysteresis function.

Source File

```
#include <GFLIB_Hyst.h>
```

Compound Type Members**Table 329.** GFLIB_HYST_T_F16 members description

Type	Name	Description
tFrac16	f16HystOn	Value determining the upper threshold.
tFrac16	f16HystOff	Value determining the lower threshold.
tFrac16	f16OutValOn	Value of the output when input is higher than the upper threshold.
tFrac16	f16OutValOff	Value of the output when input is lower than the lower threshold.
tFrac16	f16OutState	Actual state of the output.

5.62 GFLIB_HYST_T_F32

Structure containing parameters and states for the hysteresis function.

Source File

```
#include <GFLIB_Hyst.h>
```

Compound Type Members**Table 330.** 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.

5.63 GFLIB_INTEGRATOR_TR_T_F16

Structure containing integrator parameters and coefficients.

Source File

```
#include <GFLIB_IntegratorTR.h>
```

Compound Type Members**Table 331.** 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 _{MAX} / T _s) (U _{MAX} *2) ^(2^{-u16NShift}) .
tU16	u16NShift	Scaling bitwise shift applied to the integrator coefficient f16C1, integer format [0, 15].

5.64 GFLIB_INTEGRATOR_TR_T_F32

Structure containing integrator parameters and coefficients.

Source File

```
#include <GFLIB_IntegratorTR.h>
```

Compound Type Members

Table 332. 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 _{MAX} / T _s) (U _{MAX} *2) ^(2^{-u16NShift}) .
tU16	u16NShift	Scaling bitwise shift applied to the integrator coefficient f32C1, integer format [0, 15].

5.65 GFLIB_LIMIT_T_F16

Structure containing the limits.

Source File

```
#include <GFLIB_Limit.h>
```

Compound Type Members

Table 333. 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.

5.66 GFLIB_LIMIT_T_F32

Structure containing the limits.

Source File

```
#include <GFLIB_Limit.h>
```

Compound Type Members

Table 334. `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.

5.67 GFLIB_LOWERLIMIT_T_F16

Structure containing the lower limit.

Source File

```
#include <GFLIB_LowerLimit.h>
```

Compound Type Members

Table 335. `GFLIB_LOWERLIMIT_T_F16` members description

Type	Name	Description
tFrac16	f16LowerLimit	Value determining the lower limit threshold.

5.68 GFLIB_LOWERLIMIT_T_F32

Structure containing the lower limit.

Source File

```
#include <GFLIB_LowerLimit.h>
```

Compound Type Members

Table 336. `GFLIB_LOWERLIMIT_T_F32` members description

Type	Name	Description
tFrac32	f32LowerLimit	Value determining the lower limit threshold.

5.69 GFLIB_LUT1D_T_F16

Structure containing 1D look-up table parameters.

Source File

```
#include <GFLIB_Lut1D.h>
```

Compound Type Members

Table 337. `GFLIB_LUT1D_T_F16` members description

Type	Name	Description
tU16	u16ShamOffset	Shift amount for extracting the fractional offset within an interpolated interval.

Type	Name	Description
const tFrac16 *	pf16Table	Table holding ordinate values of interpolating intervals.

5.70 **GFLIB_LUT1D_T_F32**

Structure containing 1D look-up table parameters.

Source File

```
#include <GFLIB_Lut1D.h>
```

Compound Type Members

Table 338. 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.

5.71 **GFLIB_LUT2D_T_F16**

Structure containing 2D look-up table parameters.

Source File

```
#include <GFLIB_Lut2D.h>
```

Compound Type Members

Table 339. 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.

5.72 **GFLIB_LUT2D_T_F32**

Structure containing 2D look-up table parameters.

Source File

```
#include <GFLIB_Lut2D.h>
```

Compound Type Members**Table 340.** `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.

5.73 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 341.** `GFLIB_RAMP_T_F16` members description

Type	Name	Description
tFrac16	f16State	Ramp state value.
tFrac16	f16RampUp	Ramp up increment coefficient.
tFrac16	f16RampDown	Ramp decrement coefficient.

5.74 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 342.** `GFLIB_RAMP_T_F32` members description

Type	Name	Description
tFrac32	f32State	Ramp state value.
tFrac32	f32RampUp	Ramp up increment coefficient.
tFrac32	f32RampDown	Ramp decrement coefficient.

5.75 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 343.** GFLIB_SIN_T_F16 members description

Type	Name	Description
tFrac16	f16A	Array of four 16-bit elements for storing coefficients of the Taylor polynomial.

5.76 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 344.** GFLIB_SIN_T_F32 members description

Type	Name	Description
tFrac32	f32A	Array of five 32-bit elements for storing coefficients of the Taylor polynomial.

5.77 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 345.** GFLIB_SINCOS_T_F16 members description

Type	Name	Description
tFrac16	f16A	Array of four 16-bit elements for storing coefficients of the Taylor polynomial.

5.78 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 346.** **GFLIB_SINCOS_T_F32** members description

Type	Name	Description
tFrac32	f32A	Array of five 32-bit elements for storing coefficients of the Taylor polynomial.

5.79 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 347.** **GFLIB_TAN_T_F16** members description

Type	Name	Description
GFLIB_TAN_TAYLOR_COEF_T_F16	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.

5.80 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 348.** **GFLIB_TAN_T_F32** members description

Type	Name	Description
GFLIB_TAN_TAYLOR_COEF_T_F32	GFLIB_TAN_SECTOR	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.

5.81 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 349. GFLIB_TAN_TAYLOR_COEF_T_F16 members description

Type	Name	Description
const tFrac16	f16A	Structure containing four polynomial coefficients for one sub-interval.

5.82 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 350. GFLIB_TAN_TAYLOR_COEF_T_F32 members description

Type	Name	Description
const tFrac32	f32A	Structure containing four polynomial coefficients for one sub-interval.

5.83 GFLIB_UPPERLIMIT_T_F16

Structure containing the upper limit.

Source File

```
#include <GFLIB_UpperLimit.h>
```

Compound Type Members**Table 351.** `GFLIB_UPPERLIMIT_T_F16` members description

Type	Name	Description
tFrac16	f16UpperLimit	Value determining the upper limit threshold.

5.84 GFLIB_UPPERLIMIT_T_F32

Structure containing the upper limit.

Source File

```
#include <GFLIB_UpperLimit.h>
```

Compound Type Members**Table 352.** `GFLIB_UPPERLIMIT_T_F32` members description

Type	Name	Description
tFrac32	f32UpperLimit	Value determining the upper limit threshold.

5.85 GFLIB_VECTORLIMIT_T_F16

Structure containing the limit.

Source File

```
#include <GFLIB_VectorLimit.h>
```

Compound Type Members**Table 353.** `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.

5.86 GFLIB_VECTORLIMIT_T_F32

Structure containing the limit.

Source File

```
#include <GFLIB_VectorLimit.h>
```

Compound Type Members**Table 354.** `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.

5.87 GMCLIB_DECOUPLINGPMSM_T_F16

Structure containing coefficients for calculation of the decoupling.

Source File

```
#include <GMCLIB_DcouplingPMSM.h>
```

Compound Type Members

Table 355. GMCLIB_DECOUPLINGPMSM_T_F16 members description

Type	Name	Description
tFrac16	f16Kd	Coefficient k_{df} .
tS16	s16KdShift	Scaling coefficient k_{d_shift} .
tFrac16	f16Kq	Coefficient k_{qf} .
tS16	s16KqShift	Scaling coefficient k_{q_shift} .

5.88 GMCLIB_DECOUPLINGPMSM_T_F32

Structure containing coefficients for calculation of the decoupling.

Source File

```
#include <GMCLIB_DcouplingPMSM.h>
```

Compound Type Members

Table 356. GMCLIB_DECOUPLINGPMSM_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} .

5.89 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 357. GMCLIB_ELIMDCBUSRIP_T_F16 members description

Type	Name	Description
tFrac16	f16ArgDcBusMsr	Measured DC bus voltage.
tFrac16	f16ModIndex	Inverse Modulation Index.

5.90 GMCLIB_ELIxDCBUSRIP_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 358. GMCLIB_ELIxDCBUSRIP_T_F32 members description

Type	Name	Description
tFrac32	f32ArgDcBusMsr	Measured DC bus voltage.
tFrac32	f32ModIndex	Inverse Modulation Index.

5.91 SWLIBS_2Syst_F16

Array of two standard 16-bit fractional arguments.

Source File

```
#include <SWLIBS_Typedefs.h>
```

Compound Type Members

Table 359. SWLIBS_2Syst_F16 members description

Type	Name	Description
tFrac16	f16Arg1	First argument
tFrac16	f16Arg2	Second argument

5.92 SWLIBS_2Syst_F32

Array of two standard 32-bit fractional arguments.

Source File

```
#include <SWLIBS_Typedefs.h>
```

Compound Type Members

Table 360. SWLIBS_2Syst_F32 members description

Type	Name	Description
tFrac32	f32Arg1	First argument
tFrac32	f32Arg2	Second argument

5.93 SWLIBS_3Syst_F16

Array of three standard 16-bit fractional arguments.

Source File

```
#include <SWLIBS_Typedefs.h>
```

Compound Type Members**Table 361.** SWLIBS_3Syst_F16 members description

Type	Name	Description
tFrac16	f16Arg1	First argument
tFrac16	f16Arg2	Second argument
tFrac16	f16Arg3	Third argument

5.94 SWLIBS_3Syst_F32

Array of three standard 32-bit fractional arguments.

Source File

```
#include <SWLIBS_Typedefs.h>
```

Compound Type Members**Table 362.** SWLIBS_3Syst_F32 members description

Type	Name	Description
tFrac32	f32Arg1	First argument
tFrac32	f32Arg2	Second argument
tFrac32	f32Arg3	Third argument

5.95 SWLIBS_VERSION_T

Motor Control Library Set identification structure.

Source File

```
#include <SWLIBS_Version.h>
```

Compound Type Members**Table 363.** 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 S32K11x 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_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, }
```

6.4 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, }
```

6.5 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}
```

6.6 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}
```

6.7 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}
```

6.8 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}
```

6.9 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}
```

6.10 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}
```

6.11 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
```

6.12 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
```

6.13 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 }
```

6.14 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 }
```

6.15 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
```

6.16 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
```

6.17 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
```

6.18 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
```

6.19 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}}
```

6.20 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}}
```

6.21 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
```

6.22 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
```

6.23 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}}
```

6.24 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}}
```

6.25 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
```

6.26 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
```

6.27 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}
```

6.28 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}
```

6.29 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
```

6.30 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
```

6.31 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
```

6.32 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
```

6.33 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
```

6.34 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
```

6.35 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
```

6.36 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
```

6.37 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
```

6.38 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
```

6.39 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 &f32gplibAtanCoef
```

6.40 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 &f16gplibAtanCoef
```

6.41 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
```

6.42 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
```

6.43 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
```

6.44 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, (ts16) 0, (ts16) 0, (ts16) 0,  
(tFrac32) 0, (tFrac32) 0, (tFrac32) 0, (tFrac32) 0, (tu16) 0 }
```

6.45 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, (ts16) 0, (ts16) 0, (ts16) 0,  
(tFrac16) 0, (tFrac16) 0, (tFrac32) 0, (tFrac16) 0, (tFrac16) 0, (tu16) 0 }
```

6.46 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
```

6.47 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
```

6.48 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 }
```

6.49 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 }
```

6.50 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
```

6.51 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
```

6.52 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 }
```

6.53 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 }
```

6.54 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
```

6.55 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
```

6.56 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 }
```

6.57 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 }
```

6.58 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
```

6.59 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
```

6.60 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 }
```

6.61 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 }
```

6.62 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
```

6.63 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
```

6.64 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
```

6.65 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
```

6.66 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
```

6.67 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
```

6.68 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 }
```

6.69 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 }
```

6.70 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
```

6.71 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
```

6.72 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 }
```

6.73 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 }
```

6.74 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
```

6.75 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
```

6.76 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}
```

6.77 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}
```

6.78 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
```

6.79 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
```

6.80 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}
```

6.81 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}
```

6.82 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
```

6.83 **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
```

6.84 **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 }
```

6.85 **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 }
```

6.86 **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
```

6.87 **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
```

6.88 **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 }
```

6.89 **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 }
```

6.90 **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
```

6.91 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
```

6.92 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 }
```

6.93 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 }
```

6.94 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
```

6.95 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
```

6.96 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
```

6.97 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
```

6.98 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
```

6.99 **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
```

6.100 **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
```

6.101 **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
```

6.102 **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
```

6.103 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
```

6.104 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
```

6.105 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
```

6.106 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
```

6.107 **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
```

6.108 **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}
```

6.109 **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}
```

6.110 **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
```

6.111 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
```

6.112 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 }
```

6.113 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 }
```

6.114 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
```

6.115 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
```

6.116 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 }
```

6.117 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 }
```

6.118 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
```

6.119 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
```

6.120 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 }
```

6.121 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 }
```

6.122 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
```

6.123 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
```

6.124 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_OFF
```

6.125 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}
```

6.126 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)
```

6.127 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)
```

6.128 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)
```

6.129 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)
```

6.130 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)
```

6.131 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)
```

6.132 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)
```

6.133 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)
```

6.134 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)
```

6.135 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)
```

6.136 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)
```

6.137 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)
```

6.138 INT32_MAX

Constant representing the maximal positive value of a signed 32-bit fixed point integer number, equal to $2^{31}-1 = 0x7ffff ffff$.

Source File

```
#include <SWLIBS_Defines.h>
```

Macro Definition

```
#define INT32_MAX ((ts32) 0x7fffffff)
```

6.139 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)
```

6.140 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))
```

6.141 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))
```

6.142 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))
```

6.143 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))
```

6.144 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))
```

6.145 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))
```

6.146 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))
```

6.147 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))
```

6.148 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))
```

6.149 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))
```

6.150 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))
```

6.151 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))
```

6.152 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))
```

6.153 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))
```

6.154 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))
```

6.155 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))
```

6.156 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))
```

6.157 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))
```

6.158 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))
```

6.159 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)
```

6.160 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)
```

6.161 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)
```

6.162 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)
```

6.163 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) 0x6ED9EB1)
```

6.164 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)
```

6.165 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)
```

6.166 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)
```

6.167 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)
```

6.168 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)
```

6.169 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)
```

6.170 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)
```

6.171 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))
```

6.172 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))
```

6.173 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
```

6.174 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
```

6.175 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

1	Introduction	3	2.5.3.2	Function AMCLIB_FWSpeedLoopInit_F16 109	
1.1	Architecture Overview	3	2.5.4	Function AMCLIB_FWSpeedLoopSetState ... 111	
1.2	General Information	4	2.5.4.1	Function AMCLIB_FWSpeedLoopSetState_	
1.3	Multiple Implementation Support	4	F32	111	
1.4	Supported Compilers	6	2.5.4.2	Function AMCLIB_FWSpeedLoopSetState_	
1.5	MATLAB Integration	6	F16	114	
1.6	Installation	7	2.5.5	Function AMCLIB_FWSpeedLoopDebug 116	
1.7	Library File Structure	10	2.5.5.1	Function AMCLIB_FWSpeedLoopDebug_	
1.8	Integration Assumption	11	F32	117	
1.9	Library Integration into a Green Hills Multi Development Environment	11	2.5.5.2	Function AMCLIB_FWSpeedLoopDebug_	
1.10	Library Integration into a S32 Design Studio IDE for Arm based MCUs	14	F16	120	
1.11	Library Testing	17	2.6	Function AMCLIB_SpeedLoop	122
1.12	Functions Accuracy	20	2.6.1	Function AMCLIB_SpeedLoop_F32	123
2	Functions	22	2.6.2	Function AMCLIB_SpeedLoop_F16	126
2.1	Function index	22	2.6.3	Function AMCLIB_SpeedLoopInit	129
2.2	Function AMCLIB_BemfObsrvDQ	35	2.6.3.1	Function AMCLIB_SpeedLoopInit_F32	129
2.2.1	Function AMCLIB_BemfObsrvDQ_F32	37	2.6.3.2	Function AMCLIB_SpeedLoopInit_F16	131
2.2.2	Function AMCLIB_BemfObsrvDQ_F16	43	2.6.4	Function AMCLIB_SpeedLoopSetState	133
2.2.3	Function AMCLIB_BemfObsrvDQInit	48	2.6.4.1	Function AMCLIB_SpeedLoopSetState_	
2.2.3.1	Function AMCLIB_BemfObsrvDQInit_F32	49	F32	133	
2.2.3.2	Function AMCLIB_BemfObsrvDQInit_F16	51	2.6.4.2	Function AMCLIB_SpeedLoopSetState_	
2.2.4	Function AMCLIB_BemfObsrvDQSetState	53	F16	135	
2.2.4.1	Function AMCLIB_BemfObsrvDQSetState_		2.6.5	Function AMCLIB_SpeedLoopDebug	137
	F32		2.6.5.1	Function AMCLIB_SpeedLoopDebug_F32	137
2.2.4.2	Function AMCLIB_BemfObsrvDQSetState_		2.6.5.2	Function AMCLIB_SpeedLoopDebug_F16 139	
	F16		2.7	Function AMCLIB_TrackObsrv	142
2.3	Function AMCLIB_CurrentLoop	57	2.7.1	Function AMCLIB_TrackObsrv_F32	145
2.3.1	Function AMCLIB_CurrentLoop_F32	59	2.7.2	Function AMCLIB_TrackObsrv_F16	149
2.3.2	Function AMCLIB_CurrentLoop_F16	62	2.7.3	Function AMCLIB_TrackObsrvInit	153
2.3.3	Function AMCLIB_CurrentLoopInit	65	2.7.3.1	Function AMCLIB_TrackObsrvInit_F32	153
2.3.3.1	Function AMCLIB_CurrentLoopInit_F32	66	2.7.3.2	Function AMCLIB_TrackObsrvInit_F16	155
2.3.3.2	Function AMCLIB_CurrentLoopInit_F16	68	2.7.4	Function AMCLIB_TrackObsrvSetState	157
2.3.4	Function AMCLIB_CurrentLoopSetState	69	2.7.4.1	Function AMCLIB_TrackObsrvSetState_	
2.3.4.1	Function AMCLIB_CurrentLoopSetState_		F32	157	
	F32		2.7.4.2	Function AMCLIB_TrackObsrvSetState_	
	70		F16	159	
2.3.4.2	Function AMCLIB_CurrentLoopSetState_		2.8	Function AMCLIB_Windmilling	161
	F16		2.8.1	Function AMCLIB_Windmilling_F32	162
2.4	Function AMCLIB_FW	74	2.8.2	Function AMCLIB_Windmilling_F16	164
2.4.1	Function AMCLIB_FW_F32	77	2.8.3	Function AMCLIB_WindmillingInit	166
2.4.2	Function AMCLIB_FW_F16	80	2.8.3.1	Function AMCLIB_WindmillingInit_F32	167
2.4.3	Function AMCLIB_FWInit	83	2.8.3.2	Function AMCLIB_WindmillingInit_F16	168
2.4.3.1	Function AMCLIB_FWInit_F32	84	2.9	Function GDFLIB_FilterFIR	170
2.4.3.2	Function AMCLIB_FWInit_F16	86	2.9.1	Function GDFLIB_FilterFIR_F32	171
2.4.4	Function AMCLIB_FWSetState	87	2.9.2	Function GDFLIB_FilterFIR_F16	173
2.4.4.1	Function AMCLIB_FWSetState_F32	88	2.9.3	Function GDFLIB_FilterFIRInit	175
2.4.4.2	Function AMCLIB_FWSetState_F16	90	2.9.3.1	Function GDFLIB_FilterFIRInit_F32	175
2.4.5	Function AMCLIB_FWDebug	92	2.9.3.2	Function GDFLIB_FilterFIRInit_F16	178
2.4.5.1	Function AMCLIB_FWDebug_F32	93	2.10	Function GDFLIB_FilterIIR1	180
2.4.5.2	Function AMCLIB_FWDebug_F16	95	2.10.1	Function GDFLIB_FilterIIR1_F32	181
2.5	Function AMCLIB_FWSpeedLoop	97	2.10.2	Function GDFLIB_FilterIIR1_F16	183
2.5.1	Function AMCLIB_FWSpeedLoop_F32	99	2.10.3	Function GDFLIB_FilterIIR1Init	184
2.5.2	Function AMCLIB_FWSpeedLoop_F16	102	2.10.3.1	Function GDFLIB_FilterIIR1Init_F32	184
2.5.3	Function AMCLIB_FWSpeedLoopInit	106	2.10.3.2	Function GDFLIB_FilterIIR1Init_F16	185
2.5.3.1	Function AMCLIB_FWSpeedLoopInit_F32 107		2.11	Function GDFLIB_FilterIIR2	186
			2.11.1	Function GDFLIB_FilterIIR2_F32	187

2.11.2	Function GDFLIB_FilterIIR2_F16	189	2.20.4.2	Function GFLIB_ControllerPlpAWSetState_F16	266
2.11.3	Function GDFLIB_FilterIIR2Init	191	2.21	Function GFLIB_ControllerPlr	268
2.11.3.1	Function GDFLIB_FilterIIR2Init_F32	191	2.21.1	Function GFLIB_ControllerPlr_F32	268
2.11.3.2	Function GDFLIB_FilterIIR2Init_F16	192	2.21.2	Function GFLIB_ControllerPlr_F16	272
2.12	Function GDFLIB_FilterMA	192	2.21.3	Function GFLIB_ControllerPlrInit	275
2.12.1	Function GDFLIB_FilterMA_F32	193	2.21.3.1	Function GFLIB_ControllerPlrInit_F32	275
2.12.2	Function GDFLIB_FilterMA_F16	194	2.21.3.2	Function GFLIB_ControllerPlrInit_F16	276
2.12.3	Function GDFLIB_FilterMAInit	196	2.21.4	Function GFLIB_ControllerPlrSetState	278
2.12.3.1	Function GDFLIB_FilterMAInit_F32	196	2.21.4.1	Function GFLIB_ControllerPlrSetState_F32 ..	278
2.12.3.2	Function GDFLIB_FilterMAInit_F16	197	2.21.4.2	Function GFLIB_ControllerPlrSetState_F16 ..	280
2.12.4	Function GDFLIB_FilterMASetState	197	2.22	Function GFLIB_ControllerPlrAW	281
2.12.4.1	Function GDFLIB_FilterMASetState_F32 ..	198	2.22.1	Function GFLIB_ControllerPlrAW_F32 ..	282
2.12.4.2	Function GDFLIB_FilterMASetState_F16 ..	198	2.22.2	Function GFLIB_ControllerPlrAW_F16 ..	285
2.13	Function GFLIB_Acos	199	2.22.3	Function GFLIB_ControllerPlrAWInit	289
2.13.1	Function GFLIB_Acos_F32	200	2.22.3.1	Function GFLIB_ControllerPlrAWInit_F32 ..	289
2.13.2	Function GFLIB_Acos_F16	202	2.22.3.2	Function GFLIB_ControllerPlrAWInit_F16 ..	291
2.14	Function GFLIB_Asin	204	2.22.4	Function GFLIB_ControllerPlrAWSetState ..	292
2.14.1	Function GFLIB_Asin_F32	204	2.22.4.1	Function GFLIB_ControllerPlrAWSetState_	292
2.14.2	Function GFLIB_Asin_F16	207	2.22.4.2	Function GFLIB_ControllerPlrAWSetState_F32 ..	292
2.15	Function GFLIB_Atan	209	2.23	Function GFLIB_Cos	295
2.15.1	Function GFLIB_Atan_F32	210	2.23.1	Function GFLIB_Cos_F32	296
2.15.2	Function GFLIB_Atan_F16	211	2.23.2	Function GFLIB_Cos_F16	297
2.16	Function GFLIB_AtanYX	213	2.24	Function GFLIB_Hyst	298
2.16.1	Function GFLIB_AtanYX_F32	213	2.24.1	Function GFLIB_Hyst_F32	299
2.16.2	Function GFLIB_AtanYX_F16	215	2.24.2	Function GFLIB_Hyst_F16	301
2.17	Function GFLIB_AtanYXShifted	216	2.25	Function GFLIB_IntegratorTR	302
2.17.1	Function GFLIB_AtanYXShifted_F32 ..	217	2.25.1	Function GFLIB_IntegratorTR_F32	303
2.17.2	Function GFLIB_AtanYXShifted_F16 ..	221	2.25.2	Function GFLIB_IntegratorTR_F16	305
2.18	Function GFLIB_ControllerPIDpAW	225	2.25.3	Function GFLIB_IntegratorTRSetState	306
2.18.1	Function GFLIB_ControllerPIDpAW_F32 ..	227	2.25.3.1	Function GFLIB_IntegratorTRSetState_F32 ..	307
2.18.2	Function GFLIB_ControllerPIDpAW_F16 ..	230	2.25.3.2	Function GFLIB_IntegratorTRSetState_F16 ..	308
2.18.3	Function GFLIB_ControllerPIDpAWInit	233	2.26	Function GFLIB_Limit	309
2.18.3.1	Function GFLIB_ControllerPIDpAWInit_F32 ..	234	2.26.1	Function GFLIB_Limit_F32	309
2.18.3.2	Function GFLIB_ControllerPIDpAWInit_F16 ..	235	2.26.2	Function GFLIB_Limit_F16	310
2.18.4	Function GFLIB_ControllerPIDpAWSetState	237	2.27	Function GFLIB_LowerLimit	311
2.18.4.1	Function GFLIB_ControllerPIDpAWSetState_F32 ..	237	2.27.1	Function GFLIB_LowerLimit_F32	312
2.18.4.2	Function GFLIB_ControllerPIDpAWSetState_F16 ..	239	2.27.2	Function GFLIB_LowerLimit_F16	313
2.19	Function GFLIB_ControllerPlp	240	2.28	Function GFLIB_Lut1D	313
2.19.1	Function GFLIB_ControllerPlp_F32	242	2.28.1	Function GFLIB_Lut1D_F32	314
2.19.2	Function GFLIB_ControllerPlp_F16	244	2.28.2	Function GFLIB_Lut1D_F16	317
2.19.3	Function GFLIB_ControllerPlpInit	247	2.29	Function GFLIB_Lut2D	320
2.19.3.1	Function GFLIB_ControllerPlpInit_F32 ..	247	2.29.1	Function GFLIB_Lut2D_F32	322
2.19.3.2	Function GFLIB_ControllerPlpInit_F16 ..	249	2.29.2	Function GFLIB_Lut2D_F16	324
2.19.4	Function GFLIB_ControllerPlpSetState	250	2.30	Function GFLIB_Ramp	326
2.19.4.1	Function GFLIB_ControllerPlpSetState_F32 ..	250	2.30.1	Function GFLIB_Ramp_F32	327
2.19.4.2	Function GFLIB_ControllerPlpSetState_F16 ..	252	2.30.2	Function GFLIB_Ramp_F16	328
2.20	Function GFLIB_ControllerPlpAW	253	2.31	Function GFLIB_Sign	329
2.20.1	Function GFLIB_ControllerPlpAW_F32 ..	255	2.31.1	Function GFLIB_Sign_F32	329
2.20.2	Function GFLIB_ControllerPlpAW_F16 ..	258	2.31.2	Function GFLIB_Sign_F16	330
2.20.3	Function GFLIB_ControllerPlpAWInit	261	2.32	Function GFLIB_Sin	331
2.20.3.1	Function GFLIB_ControllerPlpAWInit_F32 ..	261	2.32.1	Function GFLIB_Sin_F32	332
2.20.3.2	Function GFLIB_ControllerPlpAWInit_F16 ..	263	2.32.2	Function GFLIB_Sin_F16	333
2.20.4	Function GFLIB_ControllerPlpAWSetState ..	264	2.33	Function GFLIB_SinCos	334
2.20.4.1	Function GFLIB_ControllerPlpAWSetState_F32 ..	264	2.33.1	Function GFLIB_SinCos_F32	334
			2.33.2	Function GFLIB_SinCos_F16	335
			2.34	Function GFLIB_Sqrt	336

2.34.1	Function GFLIB_Sqrt_F32	337	2.49.2	Function GMCLIB_SvmStd_F16	407
2.34.2	Function GFLIB_Sqrt_F16	338	2.50	Function GMCLIB_SvmU0n	408
2.35	Function GFLIB_Tan	338	2.50.1	Function GMCLIB_SvmU0n_F32	410
2.35.1	Function GFLIB_Tan_F32	339	2.50.2	Function GMCLIB_SvmU0n_F16	411
2.35.2	Function GFLIB_Tan_F16	340	2.51	Function GMCLIB_SvmU7n	413
2.36	Function GFLIB_UpperLimit	341	2.51.1	Function GMCLIB_SvmU7n_F32	415
2.36.1	Function GFLIB_UpperLimit_F32	342	2.51.2	Function GMCLIB_SvmU7n_F16	417
2.36.2	Function GFLIB_UpperLimit_F16	343	2.52	Function MLIB_Abs	418
2.37	Function GFLIB_VectorLimit	344	2.52.1	Function MLIB_Abs_F32	418
2.37.1	Function GFLIB_VectorLimit_F32	345	2.52.2	Function MLIB_Abs_F16	419
2.37.2	Function GFLIB_VectorLimit_F16	346	2.53	Function MLIB_AbsSat	420
2.38	Function GFLIB_VMin	347	2.53.1	Function MLIB_AbsSat_F32	420
2.38.1	Function GFLIB_VMin_F32	348	2.53.2	Function MLIB_AbsSat_F16	421
2.38.2	Function GFLIB_VMin_F16	348	2.54	Function MLIB_Add	422
2.38.3	Function GFLIB_VMin4_F16	349	2.54.1	Function MLIB_Add_F32	423
2.38.4	Function GFLIB_VMin5_F16	350	2.54.2	Function MLIB_Add_F16	424
2.38.5	Function GFLIB_VMin6_F16	350	2.55	Function MLIB_AddSat	425
2.38.6	Function GFLIB_VMin7_F16	351	2.55.1	Function MLIB_AddSat_F32	425
2.38.7	Function GFLIB_VMin8_F16	352	2.55.2	Function MLIB_AddSat_F16	426
2.38.8	Function GFLIB_VMin9_F16	352	2.56	Function MLIB_Convert	427
2.38.9	Function GFLIB_VMin10_F16	353	2.56.1	Function MLIB_Convert_F32F16	428
2.38.10	Function GFLIB_VMin11_F16	354	2.56.2	Function MLIB_Convert_F16F32	429
2.38.11	Function GFLIB_VMin12_F16	354	2.57	Function MLIB_ConvertPU	430
2.38.12	Function GFLIB_VMin13_F16	355	2.57.1	Function MLIB_ConvertPU_F32F16	430
2.38.13	Function GFLIB_VMin14_F16	356	2.57.2	Function MLIB_ConvertPU_F16F32	431
2.38.14	Function GFLIB_VMin15_F16	357	2.58	Function MLIB_Div	432
2.38.15	Function GFLIB_VMin16_F16	357	2.58.1	Function MLIB_Div_F32	432
2.39	Function GMCLIB_BetaProjection	358	2.58.2	Function MLIB_Div_F16	434
2.39.1	Function GMCLIB_BetaProjection_F32	359	2.59	Function MLIB_DivSat	435
2.39.2	Function GMCLIB_BetaProjection_F16	360	2.59.1	Function MLIB_DivSat_F32	435
2.40	Function GMCLIB_BetaProjection3Ph	361	2.59.2	Function MLIB_DivSat_F16	436
2.40.1	Function GMCLIB_BetaProjection3Ph_F32	361	2.60	Function MLIB_Mac	437
2.40.2	Function GMCLIB_BetaProjection3Ph_F16	362	2.60.1	Function MLIB_Mac_F32	437
2.41	Function GMCLIB_Clark	363	2.60.2	Function MLIB_Mac_F32F16F16	439
2.41.1	Function GMCLIB_Clark_F32	364	2.60.3	Function MLIB_Mac_F16	440
2.41.2	Function GMCLIB_Clark_F16	365	2.61	Function MLIB_MacSat	441
2.42	Function GMCLIB_ClarkInv	366	2.61.1	Function MLIB_MacSat_F32	441
2.42.1	Function GMCLIB_ClarkInv_F32	366	2.61.2	Function MLIB_MacSat_F32F16F16	442
2.42.2	Function GMCLIB_ClarkInv_F16	368	2.61.3	Function MLIB_MacSat_F16	444
2.43	Function GMCLIB_DecouplingPMSM	369	2.62	Function MLIB_Mnac	445
2.43.1	Function GMCLIB_DecouplingPMSM_F32	370	2.62.1	Function MLIB_Mnac_F32	445
2.43.2	Function GMCLIB_DecouplingPMSM_F16	373	2.62.2	Function MLIB_Mnac_F32F16F16	446
2.44	Function GMCLIB_ElimDcBusRip	376	2.62.3	Function MLIB_Mnac_F16	448
2.44.1	Function GMCLIB_ElimDcBusRip_F32	376	2.63	Function MLIB_Msu	449
2.44.2	Function GMCLIB_ElimDcBusRip_F16	379	2.63.1	Function MLIB_Msu_F32	449
2.45	Function GMCLIB_Park	381	2.63.2	Function MLIB_Msu_F32F16F16	450
2.45.1	Function GMCLIB_Park_F32	382	2.63.3	Function MLIB_Msu_F16	452
2.45.2	Function GMCLIB_Park_F16	383	2.64	Function MLIB_Mul	453
2.46	Function GMCLIB_ParkInv	384	2.64.1	Function MLIB_Mul_F32	453
2.46.1	Function GMCLIB_ParkInv_F32	384	2.64.2	Function MLIB_Mul_F32F16F16	454
2.46.2	Function GMCLIB_ParkInv_F16	386	2.64.3	Function MLIB_Mul_F16	455
2.47	Function GMCLIB_PwmIct	387	2.65	Function MLIB_MulSat	456
2.47.1	Function GMCLIB_PwmIct_F32	388	2.65.1	Function MLIB_MulSat_F32	457
2.47.2	Function GMCLIB_PwmIct_F16	389	2.65.2	Function MLIB_MulSat_F32F16F16	458
2.48	Function GMCLIB_SvmSci	390	2.65.3	Function MLIB_MulSat_F16	459
2.48.1	Function GMCLIB_SvmSci_F32	392	2.66	Function MLIB_Neg	460
2.48.2	Function GMCLIB_SvmSci_F16	394	2.66.1	Function MLIB_Neg_F32	460
2.49	Function GMCLIB_SvmStd	395	2.66.2	Function MLIB_Neg_F16	461
2.49.1	Function GMCLIB_SvmStd_F32	405	2.67	Function MLIB_NegSat	462

2.67.1	Function MLIB_NegSat_F32	463	5.18	AMCLIB_TRACK_OBSRV_T_F32	503
2.67.2	Function MLIB_NegSat_F16	464	5.19	AMCLIB_WINDMILLING_T_F16	503
2.68	Function MLIB_Norm	465	5.20	AMCLIB_WINDMILLING_T_F32	504
2.68.1	Function MLIB_Norm_F32	465	5.21	GDFLIB_FILTER_IIR1_COEFF_T_F16	505
2.68.2	Function MLIB_Norm_F16	466	5.22	GDFLIB_FILTER_IIR1_COEFF_T_F32	505
2.69	Function MLIB_RndSat_F16F32	467	5.23	GDFLIB_FILTER_IIR1_T_F16	505
2.70	Function MLIB_Round	467	5.24	GDFLIB_FILTER_IIR1_T_F32	506
2.70.1	Function MLIB_Round_F32	468	5.25	GDFLIB_FILTER_IIR2_COEFF_T_F16	506
2.70.2	Function MLIB_Round_F16	469	5.26	GDFLIB_FILTER_IIR2_COEFF_T_F32	507
2.71	Function MLIB_ShBi	470	5.27	GDFLIB_FILTER_IIR2_T_F16	507
2.71.1	Function MLIB_ShBi_F32	471	5.28	GDFLIB_FILTER_IIR2_T_F32	507
2.71.2	Function MLIB_ShBi_F16	472	5.29	GDFLIB_FILTER_MA_T_F16	508
2.72	Function MLIB_ShBiSat	473	5.30	GDFLIB_FILTER_MA_T_F32	508
2.72.1	Function MLIB_ShBiSat_F32	473	5.31	GDFLIB_FILTERFIR_PARAM_T_F16	508
2.72.2	Function MLIB_ShBiSat_F16	474	5.32	GDFLIB_FILTERFIR_PARAM_T_F32	509
2.73	Function MLIB_ShL	475	5.33	GDFLIB_FILTERFIR_STATE_T_F16	509
2.73.1	Function MLIB_ShL_F32	475	5.34	GDFLIB_FILTERFIR_STATE_T_F32	510
2.73.2	Function MLIB_ShL_F16	476	5.35	GFLIB_ACOS_T_F16	510
2.74	Function MLIB_ShLSat	477	5.36	GFLIB_ACOS_T_F32	510
2.74.1	Function MLIB_ShLSat_F32	477	5.37	GFLIB_ACOS_TAYLOR_COEF_T_F16	511
2.74.2	Function MLIB_ShLSat_F16	478	5.38	GFLIB_ACOS_TAYLOR_COEF_T_F32	511
2.75	Function MLIB_ShR	479	5.39	GFLIB_ASIN_T_F16	512
2.75.1	Function MLIB_ShR_F32	479	5.40	GFLIB_ASIN_T_F32	512
2.75.2	Function MLIB_ShR_F16	480	5.41	GFLIB_ASIN_TAYLOR_COEF_T_F16	513
2.76	Function MLIB_Sub	481	5.42	GFLIB_ASIN_TAYLOR_COEF_T_F32	513
2.76.1	Function MLIB_Sub_F32	482	5.43	GFLIB_ATAN_T_F16	513
2.76.2	Function MLIB_Sub_F16	483	5.44	GFLIB_ATAN_T_F32	514
2.77	Function MLIB_SubSat	484	5.45	GFLIB_ATAN_TAYLOR_COEF_T_F16	514
2.77.1	Function MLIB_SubSat_F32	484	5.46	GFLIB_ATAN_TAYLOR_COEF_T_F32	515
2.77.2	Function MLIB_SubSat_F16	485	5.47	GFLIB_ATANYXSHIFTED_T_F16	515
2.78	Function MLIB_VMac	486	5.48	GFLIB_ATANYXSHIFTED_T_F32	515
2.78.1	Function MLIB_VMac_F32	487	5.49	GFLIB_CONTROLLER_PI_P_T_F16	516
2.78.2	Function MLIB_VMac_F32F16F16	488	5.50	GFLIB_CONTROLLER_PI_P_T_F32	516
2.78.3	Function MLIB_VMac_F16	489	5.51	GFLIB_CONTROLLER_PI_R_T_F16	517
2.79	Function SWLIBS_GetVersion	491	5.52	GFLIB_CONTROLLER_PI_R_T_F32	517
3	Typedefs	491	5.53	GFLIB_CONTROLLER_PIAW_P_T_F16	518
4	Enums	492	5.54	GFLIB_CONTROLLER_PIAW_P_T_F32	518
4.1	AMCLIB_WINDMILLING_RET_T	492	5.55	GFLIB_CONTROLLER_PIAW_R_T_F16	519
4.2	tBool	492	5.56	GFLIB_CONTROLLER_PIAW_R_T_F32	520
5	Compound data types	492	5.57	GFLIB_CONTROLLER_PID_P_AW_T_F16	520
5.1	AMCLIB_BEMF_OBSRV_DQ_T_F16	492	5.58	GFLIB_CONTROLLER_PID_P_AW_T_F32	521
5.2	AMCLIB_BEMF_OBSRV_DQ_T_F32	493	5.59	GFLIB_COS_T_F16	522
5.3	AMCLIB_CURRENT_LOOP_T_F16	494	5.60	GFLIB_COS_T_F32	522
5.4	AMCLIB_CURRENT_LOOP_T_F32	494	5.61	GFLIB_HYST_T_F16	522
5.5	AMCLIB_FW_DEBUG_T_F16	495	5.62	GFLIB_HYST_T_F32	523
5.6	AMCLIB_FW_DEBUG_T_F32	496	5.63	GFLIB_INTEGRATOR_TR_T_F16	523
5.7	AMCLIB_FW_SPEED_LOOP_DEBUG_T_F16	496	5.64	GFLIB_INTEGRATOR_TR_T_F32	524
5.8	AMCLIB_FW_SPEED_LOOP_DEBUG_T_F32	497	5.65	GFLIB_LIMIT_T_F16	524
5.9	AMCLIB_FW_SPEED_LOOP_T_F16	498	5.66	GFLIB_LIMIT_T_F32	524
5.10	AMCLIB_FW_SPEED_LOOP_T_F32	499	5.67	GFLIB_LOWERLIMIT_T_F16	525
5.11	AMCLIB_FW_T_F16	500	5.68	GFLIB_LOWERLIMIT_T_F32	525
5.12	AMCLIB_FW_T_F32	500	5.69	GFLIB_LUT1D_T_F16	525
5.13	AMCLIB_SPEED_LOOP_DEBUG_T_F16	501	5.70	GFLIB_LUT1D_T_F32	526
5.14	AMCLIB_SPEED_LOOP_DEBUG_T_F32	501	5.71	GFLIB_LUT2D_T_F16	526
5.15	AMCLIB_SPEED_LOOP_T_F16	502	5.72	GFLIB_LUT2D_T_F32	526
5.16	AMCLIB_SPEED_LOOP_T_F32	502	5.73	GFLIB_RAMP_T_F16	527
5.17	AMCLIB_TRACK_OBSRV_T_F16	502	5.74	GFLIB_RAMP_T_F32	527
			5.75	GFLIB_SIN_T_F16	527
			5.76	GFLIB_SIN_T_F32	528

5.77	GFLIB_SINCOS_T_F16	528	6.34	GFLIB_ASIN_DEFAULT	543
5.78	GFLIB_SINCOS_T_F32	528	6.35	GFLIB_ASIN_DEFAULT_F32	543
5.79	GFLIB_TAN_T_F16	529	6.36	GFLIB_ASIN_DEFAULT_F16	544
5.80	GFLIB_TAN_T_F32	529	6.37	GFLIB_ATAN_T	544
5.81	GFLIB_TAN_TAYLOR_COEF_T_F16	530	6.38	GFLIB_ATAN_DEFAULT	544
5.82	GFLIB_TAN_TAYLOR_COEF_T_F32	530	6.39	GFLIB_ATAN_DEFAULT_F32	544
5.83	GFLIB_UPPERLIMIT_T_F16	530	6.40	GFLIB_ATAN_DEFAULT_F16	545
5.84	GFLIB_UPPERLIMIT_T_F32	531	6.41	GFLIB_ATANYXSHIFTED_T	545
5.85	GFLIB_VECTORLIMIT_T_F16	531	6.42	GFLIB_CONTROLLER_PID_P_AW_T	545
5.86	GFLIB_VECTORLIMIT_T_F32	531	6.43	GFLIB_CONTROLLER_PID_P_AW_	
5.87	GMCLIB_DECOUPLINGPMSM_T_F16	532		DEFAULT	545
5.88	GMCLIB_DECOUPLINGPMSM_T_F32	532	6.44	GFLIB_CONTROLLER_PID_P_AW_	
5.89	GMCLIB_ELIMDCBUSRIP_T_F16	532		DEFAULT_F32	546
5.90	GMCLIB_ELIMDCBUSRIP_T_F32	533	6.45	GFLIB_CONTROLLER_PID_P_AW_	
5.91	SWLIBS_2Syst_F16	533		DEFAULT_F16	546
5.92	SWLIBS_2Syst_F32	533	6.46	GFLIB_CONTROLLER_PI_P_T	546
5.93	SWLIBS_3Syst_F16	533	6.47	GFLIB_CONTROLLER_PI_P_DEFAULT	546
5.94	SWLIBS_3Syst_F32	534	6.48	GFLIB_CONTROLLER_PI_P_DEFAULT_	
5.95	SWLIBS_VERSION_T	534		F32	547
6	Macros	534	6.49	GFLIB_CONTROLLER_PI_P_DEFAULT_	
6.1	AMCLIB_BEMF_OBSRV_DQ_DEFAULT_			F16	547
	F32	535	6.50	GFLIB_CONTROLLER_PIAW_P_T	547
6.2	AMCLIB_BEMF_OBSRV_DQ_DEFAULT_			GFLIB_CONTROLLER_PIAW_P_	
	F16	535	6.51	DEFAULT	548
6.3	AMCLIB_CURRENT_LOOP_DEFAULT_			GFLIB_CONTROLLER_PIAW_P_	
	F32	535	6.52	DEFAULT_F32	548
6.4	AMCLIB_CURRENT_LOOP_DEFAULT_			GFLIB_CONTROLLER_PIAW_P_	
	F16	535	6.53	DEFAULT_F16	548
6.5	AMCLIB_FW_DEFAULT_F32	536	6.54	GFLIB_CONTROLLER_PI_R_T	548
6.6	AMCLIB_FW_DEFAULT_F16	536	6.55	GFLIB_CONTROLLER_PI_R_DEFAULT	549
6.7	AMCLIB_FW_SPEED_LOOP_DEFAULT_			GFLIB_CONTROLLER_PI_R_DEFAULT_	
	F32	536	6.56	F32	549
6.8	AMCLIB_FW_SPEED_LOOP_DEFAULT_			GFLIB_CONTROLLER_PI_R_DEFAULT_	
	F16	537	6.57	F16	549
6.9	AMCLIB_SPEED_LOOP_DEFAULT_F32	537	6.58	GFLIB_CONTROLLER_PIAW_R_T	549
6.10	AMCLIB_SPEED_LOOP_DEFAULT_F16	537	6.59	GFLIB_CONTROLLER_PIAW_R_	
6.11	AMCLIB_TRACK_OBSRV_T	537		DEFAULT	550
6.12	AMCLIB_TRACK_OBSRV_DEFAULT	538	6.60	GFLIB_CONTROLLER_PIAW_R_	
6.13	AMCLIB_TRACK_OBSRV_DEFAULT_F32	538		DEFAULT_F32	550
6.14	AMCLIB_TRACK_OBSRV_DEFAULT_F16	538	6.61	GFLIB_CONTROLLER_PIAW_R_	
6.15	GDFLIB_FILTERFIR_PARAM_T	539		DEFAULT_F16	550
6.16	GDFLIB_FILTERFIR_STATE_T	539	6.62	GFLIB_COS_T	550
6.17	GDFLIB_FILTER_IIR1_T	539	6.63	GFLIB_COS_DEFAULT	551
6.18	GDFLIB_FILTER_IIR1_DEFAULT	539	6.64	GFLIB_COS_DEFAULT_F32	551
6.19	GDFLIB_FILTER_IIR1_DEFAULT_F32	540	6.65	GFLIB_COS_DEFAULT_F16	551
6.20	GDFLIB_FILTER_IIR1_DEFAULT_F16	540	6.66	GFLIB_HYST_T	551
6.21	GDFLIB_FILTER_IIR2_T	540	6.67	GFLIB_HYST_DEFAULT	552
6.22	GDFLIB_FILTER_IIR2_DEFAULT	540	6.68	GFLIB_HYST_DEFAULT_F32	552
6.23	GDFLIB_FILTER_IIR2_DEFAULT_F32	541	6.69	GFLIB_HYST_DEFAULT_F16	552
6.24	GDFLIB_FILTER_IIR2_DEFAULT_F16	541	6.70	GFLIB_INTEGRATOR_TR_T	552
6.25	GDFLIB_FILTER_MA_T	541	6.71	GFLIB_INTEGRATOR_TR_DEFAULT	553
6.26	GDFLIB_FILTER_MA_DEFAULT	541	6.72	GFLIB_INTEGRATOR_TR_DEFAULT_F32	553
6.27	GDFLIB_FILTER_MA_DEFAULT_F32	542	6.73	GFLIB_INTEGRATOR_TR_DEFAULT_F16	553
6.28	GDFLIB_FILTER_MA_DEFAULT_F16	542	6.74	GFLIB_LIMIT_T	553
6.29	GFLIB_ACOS_T	542	6.75	GFLIB_LIMIT_DEFAULT	554
6.30	GFLIB_ACOS_DEFAULT	542	6.76	GFLIB_LIMIT_DEFAULT_F32	554
6.31	GFLIB_ACOS_DEFAULT_F32	543	6.77	GFLIB_LIMIT_DEFAULT_F16	554
6.32	GFLIB_ACOS_DEFAULT_F16	543	6.78	GFLIB_LOWERLIMIT_T	554
6.33	GFLIB_ASIN_T	543	6.79	GFLIB_LOWERLIMIT_DEFAULT	555

6.80	GFLIB_LOWERLIMIT_DEFAULT_F32	555	6.133	FRAC16_0_25	568
6.81	GFLIB_LOWERLIMIT_DEFAULT_F16	555	6.134	UINT16_MAX	568
6.82	GFLIB_LUT1D_T	555	6.135	INT16_MAX	568
6.83	GFLIB_LUT1D_DEFAULT	556	6.136	INT16_MIN	569
6.84	GFLIB_LUT1D_DEFAULT_F32	556	6.137	UINT32_MAX	569
6.85	GFLIB_LUT1D_DEFAULT_F16	556	6.138	INT32_MAX	569
6.86	GFLIB_LUT2D_T	556	6.139	INT32_MIN	569
6.87	GFLIB_LUT2D_DEFAULT	557	6.140	INT16TOINT32	570
6.88	GFLIB_LUT2D_DEFAULT_F32	557	6.141	INT32TOINT16	570
6.89	GFLIB_LUT2D_DEFAULT_F16	557	6.142	INT32TOINT64	570
6.90	GFLIB_RAMP_T	557	6.143	INT64TOINT32	570
6.91	GFLIB_RAMP_DEFAULT	558	6.144	F16TOINT16	571
6.92	GFLIB_RAMP_DEFAULT_F32	558	6.145	F32TOINT16	571
6.93	GFLIB_RAMP_DEFAULT_F16	558	6.146	F64TOINT16	571
6.94	GFLIB_SIN_T	558	6.147	F16TOINT32	571
6.95	GFLIB_SIN_DEFAULT	559	6.148	F32TOINT32	572
6.96	GFLIB_SIN_DEFAULT_F32	559	6.149	F64TOINT32	572
6.97	GFLIB_SIN_DEFAULT_F16	559	6.150	F16TOINT64	572
6.98	GFLIB_SINCOS_T	559	6.151	F32TOINT64	572
6.99	GFLIB_SINCOS_DEFAULT	560	6.152	F64TOINT64	573
6.100	GFLIB_SINCOS_DEFAULT_F32	560	6.153	INT16TOF16	573
6.101	GFLIB_SINCOS_DEFAULT_F16	560	6.154	INT16TOF32	573
6.102	GFLIB_TAN_T	560	6.155	INT32TOF16	573
6.103	GFLIB_TAN_DEFAULT	561	6.156	INT32TOF32	574
6.104	GFLIB_TAN_DEFAULT_F32	561	6.157	INT64TOF16	574
6.105	GFLIB_TAN_DEFAULT_F16	561	6.158	INT64TOF32	574
6.106	GFLIB_UPPERLIMIT_T	561	6.159	F16_1_DIVBY_SQRT3	574
6.107	GFLIB_UPPERLIMIT_DEFAULT	562	6.160	F32_1_DIVBY_SQRT3	574
6.108	GFLIB_UPPERLIMIT_DEFAULT_F32	562	6.161	F16_SQRT3_DIVBY_2	575
6.109	GFLIB_UPPERLIMIT_DEFAULT_F16	562	6.162	F16_SQRT3_DIVBY_4	575
6.110	GFLIB_VECTORLIMIT_T	562	6.163	F32_SQRT3_DIVBY_2	575
6.111	GFLIB_VECTORLIMIT_DEFAULT	563	6.164	F32_SQRT3_DIVBY_4	575
6.112	GFLIB_VECTORLIMIT_DEFAULT_F32	563	6.165	F16_SQRT2_DIVBY_2	576
6.113	GFLIB_VECTORLIMIT_DEFAULT_F16	563	6.166	F32_SQRT2_DIVBY_2	576
6.114	GMCLIB_DECOUPLINGPMSM_T	563	6.167	F16_1_DIVBY_3	576
6.115	GMCLIB_DECOUPLINGPMSM_DEFAULT	564	6.168	F32_1_DIVBY_3	576
6.116	GMCLIB_DECOUPLINGPMSM_DEFAULT_F32	564	6.169	F16_2_DIVBY_3	577
6.117	GMCLIB_DECOUPLINGPMSM_DEFAULT_F16	564	6.170	F32_2_DIVBY_3	577
6.118	GMCLIB_ELIMDCBUSRIP_T	564	6.171	FRAC16	577
6.119	GMCLIB_ELIMDCBUSRIP_DEFAULT	565	6.172	FRAC32	577
6.120	GMCLIB_ELIMDCBUSRIP_DEFAULT_F32	565	6.173	SWLIBS_2Syst	578
6.121	GMCLIB_ELIMDCBUSRIP_DEFAULT_F16	565	6.174	SWLIBS_3Syst	578
6.122	SWLIBS_SUPPORT_F32	565	6.175	SWLIBS_ID	578
6.123	SWLIBS_SUPPORT_F16	566			
6.124	SWLIBS_SUPPORT_FLT	566			
6.125	SWLIBS_SUPPORTED_				
	IMPLEMENTATION	566			
6.126	SFRACT_MIN	566			
6.127	SFRACT_MAX	567			
6.128	FRACT_MIN	567			
6.129	FRACT_MAX	567			
6.130	FRAC32_0_5	567			
6.131	FRAC16_0_5	568			
6.132	FRAC32_0_25	568			