Set of General Math and Motor Control Functions for Cortex M4 Core

User Reference Manual

MCLIBCORETXM4UG
Rev. 1
6/2011

freescale semiconductor

Page intentionally left blank.

Page intentionally left blank.

The following revision history table summarizes changes contained in this document.

Table 0-1: Revision History

Date	Revision Label	Description
6/2011	1	Initial release for r1.2

CONTENTS

CHAPTER 1: LICENSE AGREEMENT

IMPORTANT. Read the following Freescale Semiconductor Software License Agreement ("Agreement") completely. By selecting "I accept the terms of the license agreement" and clicking the "Next" button at the end of the page, you indicate that you accept the terms of this Agreement. You may then install the software.

FREESCALE SEMICONDUCTOR SOFTWARE LICENSE AGREEMENT [SOFTWARE FOR: Set of General Math and Motor Control Functions for Cortex M4 Core]

This is a legal agreement between you (either as an individual or as an authorized representative of your employer) and Freescale Semiconductor, Inc. ("Freescale"). It concerns your rights to use this file and any accompanying written materials produced by Freescale (the "Software"). In consideration for Freescale allowing you to access the Software, you are agreeing to be bound by the terms of this Agreement. If you do not agree to all of the terms of this Agreement, do not install or download the Software. If you change your mind later, stop using the Software and delete all copies of the Software in your possession or control. Any copies of the Software that you have already distributed, where permitted, and do not destroy will continue to be governed by this Agreement. Your prior use will also continue to be governed by this Agreement. Please note that 3rd party products, including but not limited to software ("3rd Party Products"), may be distributed in conjunction with the Software. This Agreement does not apply to those 3rd Party Products, which will be subject to their own licensing terms.

LICENSE GRANTS. Your license to the Software and applicable restrictions vary depending on the nature of the Software provided. Review the following grants carefully to ensure your compliance.

IF SOFTWARE PROVIDED IN SOURCE FORM. Freescale grants to you the non-exclusive, non-transferable right (1) to use the Software exclusively in conjunction with a development platform from Freescale or other development, prototype, or production platform utilizing at least one Cortex M4 core based processor from Freescale ("Exclusive Use"), (2) to reproduce the Software as necessary to accomplish the Exclusive Use, (3) to prepare derivative works of the Software as necessary to accomplish the Exclusive Use, (4) to distribute the Software and derivative works thereof in object (machine-readable) form only as integrated with a development platform from Freescale or other development, prototype, or production platform utilizing at least one Cortex M4 core based processor from Freescale, and (5) to sublicense to others the right to use the distributed Software. You must prohibit your sublicensees from translating, reverse engineering, decompiling, or disassembling the Software except to the extent applicable law specifically prohibits such restriction. If you violate any of the terms or restrictions of this Agreement, Freescale may immediately terminate this Agreement, and require that you stop using and delete all copies of the Software in your possession or control.

IF SOFTWARE PROVIDED IN OBJECT FORM ONLY. Freescale grants to you the non-exclusive, non-transferable right (1) to use the Software exclusively in conjunction with a development platform from Freescale or other development, prototype, or production platform utilizing at least one Cortex M4 core based processor from Freescale ("Exclusive Use"), (2) to reproduce the Software as necessary to accomplish the Exclusive Use, (3) to distribute the Software only as integrated with a development platform from Freescale or other development, prototype, or production platform utilizing at least one Cortex M4 core based processor from Freescale, and (4) to sublicense to others the right to use the distributed Software. The Software is provided to you only in object (machine-readable) form. You may exercise the rights above only with respect to such object form. You may not translate, reverse engineer, decompile, or disassemble the Software except to the extent applicable law specifically prohibits such restriction. In addition, you must prohibit your sublicensees from doing the same. If you violate any of the terms or restrictions of this Agreement, Freescale may immediately terminate this Agreement, and require that you stop using and delete all copies of the Software in your possession or control.

FOR TOOLS. Freescale grants to you the non-exclusive, non-transferable right (1) to use the Software exclusively in conjunction with a development platform from Freescale ("Exclusive Use"), and (2) to reproduce the Software. The Software is provided to you only in object (machine-readable) form. You may not distribute or sublicense the Software to others. You may exercise the rights above only with respect to such object form. You may not translate, reverse engineer, decompile, or disassemble the Software except to the extent applicable law specifically prohibits such restriction. If you violate any of the terms or restrictions of this Agreement, Freescale may immediately terminate this Agreement, and require that you stop using and delete all copies of the Software in your possession or control.

COPYRIGHT. The Software is licensed to you, not sold. Freescale owns the Software, and United States copyright laws and international treaty provisions protect the Software. Therefore, you must treat the Software like any other copyrighted material (e.g., a book or musical recording). You may not use or copy the Software for any other purpose than what is described in this Agreement. Except as expressly provided herein, Freescale does not grant to you any express or implied rights under any Freescale or third party patents, copyrights, trademarks, or trade secrets. Additionally, you must reproduce and apply any copyright or other proprietary rights notices included on or embedded in the Software to any copies or derivative works made thereof, in whole or in part, if any.

SUPPORT. Freescale is NOT obligated to provide any support, upgrades or new releases of the Software. If you wish, you may contact Freescale and report problems and provide suggestions regarding the Software. Freescale has no obligation whatsoever to respond in any way to such a problem report or suggestion. Freescale may make changes to the Software at any time, without any obligation to notify or provide updated versions of the Software to you.

LIMITED WARRANTY ON MEDIA. Freescale warrants that the media on which the Software is recorded will be free from defects in materials and workmanship under normal use for a period of 90 days from the date of purchase as evidenced by a copy of the receipt. Freescale's entire liability and your exclusive remedy under this warranty will be replacement of the defective media returned to Freescale with a copy of the receipt. Freescale will have no responsibility to replace any media damaged by accident, abuse or misapplication. This warranty extends only to you and may be invoked only by you for your customers. Freescale will not accept warranty returns from your customers.

NO ADDITIONAL WARRANTY. EXCEPT FOR THE LIMITED WARRANTY ON MEDIA PROVIDED ABOVE, THE SOFTWARE AND 3RD PARTY PRODUCTS, IF ANY, ARE PROVIDED "AS IS". YOUR USE OF THE SOFTWARE OR 3RD PARTY PRODUCTS IS AT YOUR SOLE RISK. SHOULD THE SOFTWARE OR ANY 3RD PARTY PRODUCT PROVE DEFECTIVE, YOU (AND NOT FREESCALE OR ANY FREESCALE REPRESENTATIVE) ASSUME

THE ENTIRE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION. FREESCALE EXPRESSLY DISCLAIMS ALL WARRANTIES WITH RESPECT TO THE SOFTWARE AND 3RD PARTY PRODUCTS, WHETHER SUCH WARRANTIES ARE EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. YOU EXPRESSLY ASSUME ALL LIABILITIES AND RISKS, FOR ANYONE'S USE OR OPERATION OF ANY APPLICATION PROGRAMS YOU MAY CREATE WITH THE SOFTWARE.

INDEMNITY. Freescale will defend, at its expense, any suits asserted against you based upon a claim that the Software as provided by Freescale infringes a U.S. patent or copyright or misappropriates a trade secret, and pay costs and damages finally awarded based upon such suit, if you: (1) promptly notify Freescale in writing as soon as reasonably practicable after you first become aware of the claim of infringement or misappropriation, but in no event later than 15 days of the date on which you first received notice of the claim; and (2) at Freescale's request and expense, give Freescale sole control of the suit and all requested assistance for defense of the suit. Freescale will not be liable for any settlement made without its written consent. If the use or sale of any Software component program licensed under this Agreement is enjoined as a result of such suit, Freescale at its option and at no expense to you, will: (1) obtain for you the right to use such program consistent with the license granted in this Agreement for the affected program; (2) substitute an equivalent program and extend this indemnity thereto; or (3) accept the return of the program and refund the portion of the license fee for such component program less reasonable charge for prior use. If an infringement or misappropriation claim related to the Software is alleged prior to completion of delivery, Freescale has the right to decline to make further shipments notwithstanding any other provision of this Agreement. This indemnity does not extend to any suit based upon any infringement or alleged infringement arising from any program furnished by Freescale that is: (1) altered in any way by you or any third party if the alleged infringement would not have occurred but for such alteration; (2) combined with any other products or elements not furnished by Freescale if the alleged infringement would not have occurred but for such combination; (3) designed or manufactured in accordance with your designs, specifications or instructions if the alleged infringement would not have occurred but for such designs, specifications or instructions; or (4) designed or manufactured in compliance with standards issued by any public or private standards body if the alleged infringement would not have occurred but for compliance with such standards. In no event will Freescale indemnify you or be liable in any way for royalties payable based on a per use basis, or any royalty basis other than a reasonable royalty based upon revenue derived by Freescale from your license of the Software.

THE INDEMNITY PROVIDED IN THIS SECTION IS THE SOLE, EXCLUSIVE, AND ENTIRE LIABILITY OF FREESCALE AND THE REMEDIES PROVIDED IN THIS SECTION SHALL BE YOUR EXCLUSIVE REMEDIES AGAINST FREESCALE FOR PATENT OR COPYRIGHT INFRINGEMENT OR TRADE SECRET MISAPPROPRIATION AND IS PROVIDED IN LIEU OF ALL WARRANTIES, EXPRESS, IMPLIED OR STATUTORY IN REGARD THERETO, INCLUDING, WITHOUT LIMITATION, THE WARRANTY AGAINST INFRINGEMENT SPECIFIED IN THE UNIFORM COMMERCIAL CODE.

LIMITATION OF LIABILITY. IN NO EVENT WILL FREESCALE BE LIABLE, WHETHER IN CONTRACT, TORT, OR OTHERWISE, FOR ANY INCIDENTAL, SPECIAL, INDIRECT, CONSEQUENTIAL OR PUNITIVE DAMAGES, INCLUDING, BUT NOT LIMITED TO, DAMAGES FOR ANY LOSS OF USE, LOSS OF TIME, INCONVENIENCE, COMMERCIAL LOSS, OR LOST PROFITS, SAVINGS, OR REVENUES TO THE FULL EXTENT SUCH MAY BE DIS-

CLAIMED BY LAW.

COMPLIANCE WITH LAWS; EXPORT RESTRICTIONS. You must use the Software in accordance with all applicable U.S. laws, regulations and statutes. You agree that neither you nor your licensees (if any) intend to or will, directly or indirectly, export or transmit the Software to any country in violation of U.S. export restrictions.

GOVERNMENT USE. Use of the Software and any corresponding documentation, if any, is provided with RESTRICTED RIGHTS. Use, duplication or disclosure by the Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(l) and (2) of the Commercial Computer Software–Restricted Rights at 48 CFR 52.227-19, as applicable. Manufacturer is Freescale Semiconductor, Inc., 6501 William Cannon Drive West, Austin, TX, 78735.

HIGH RISK ACTIVITIES. You acknowledge that the Software is not fault tolerant and is not designed, manufactured or intended by Freescale for incorporation into products intended for use or resale in on-line control equipment in hazardous, dangerous to life or potentially life-threatening environments requiring fail-safe performance, such as in the operation of nuclear facilities, aircraft navigation or communication systems, air traffic control, direct life support machines or weapons systems.

CHOICE OF LAW; VENUE; LIMITATIONS. You agree that the statutes and laws of the United States and the State of Texas, USA, without regard to conflicts of laws principles, will apply to all matters relating to this Agreement or the Software, and you agree that any litigation will be subject to the exclusive jurisdiction of the state or federal courts in Texas, USA. You agree that regardless of any statute or law to the contrary, any claim or cause of action arising out of or related to this Agreement or the Software must be filed within one (1) year after such claim or cause of action arose or be forever barred.

PRODUCT LABELING. You are not authorized to use any Freescale trademarks, brand names, or logos.

ENTIRE AGREEMENT. This Agreement constitutes the entire agreement between you and Freescale regarding the subject matter of this Agreement, and supersedes all prior communications, negotiations, understandings, agreements or representations, either written or oral, if any. This Agreement may only be amended in written form, executed by you and Freescale.

SEVERABILITY. If any provision of this Agreement is held for any reason to be invalid or unenforceable, then the remaining provisions of this Agreement will be unimpaired and, unless a modification or replacement of the invalid or unenforceable provision is further held to deprive you or Freescale of a material benefit, in which case the Agreement will immediately terminate, the invalid or unenforceable provision will be replaced with a provision that is valid and enforceable and that comes closest to the intention underlying the invalid or unenforceable provision.

NO WAIVER. The waiver by Freescale of any breach of any provision of this Agreement will not operate or be construed as a waiver of any other or a subsequent breach of the same or a different provision.

CHAPTER 2: INTRODUCTION

2.1 MC Library Architecture Overview

The purpose of this document is to describe the MCLIB for the Freescale Cortex M4 core-based microcontrollers. It describes the components of the library, its behaviour and interaction, the API, and steps needed for integration of the library to the project.

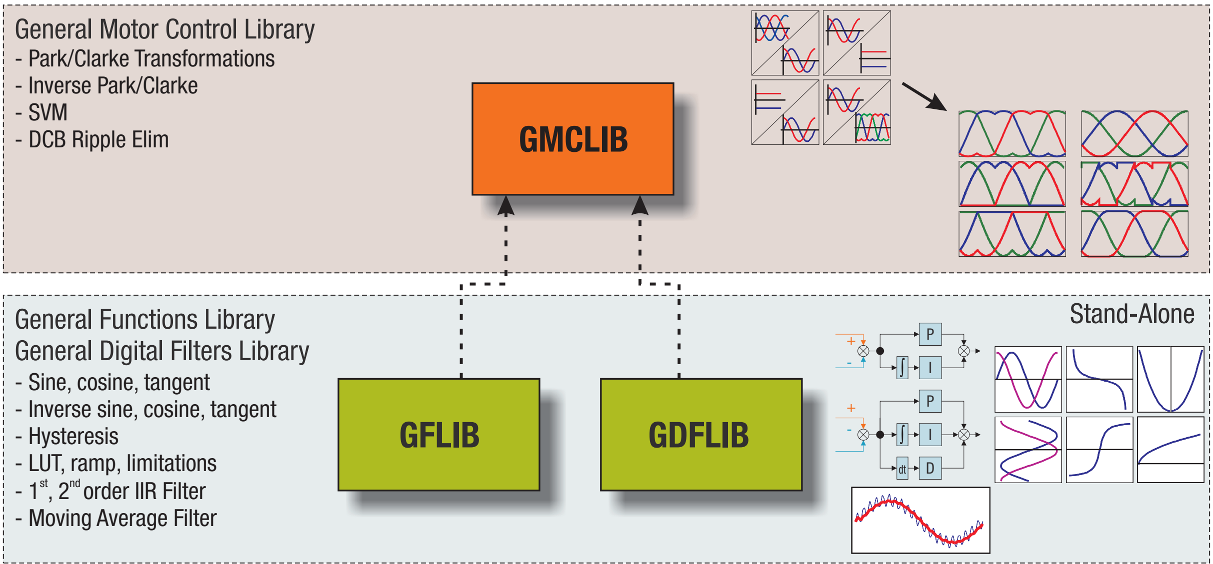


Figure 2-1: MCLIB components structure

The MCLIB consists of three sub libraries, functionally connected as depicted in Figure 2-1. The set of functions consists of following groups:

- General Function Library (GFLIB) - basic trigonometric and general maths functions such as sin, cos, tan, hyst, limit, and so on.
- General Digital Filters Library (GDFLIB) - digital IIR and FIR filters designed to be used in a motor control application
- General Motor Control Library (GMCLIB) - standard algorithms used for motor control such as Clarke/Park transformations, Space Vector Modulation, and so on.

As can be seen in Figure 2-1, the MCLIB libraries form the layer architecture where only the GFLIB and GDLIB libraries are completely independent and can stand-alone. The GMCLIB library depends on GFLIB and GDFLIB, can not stand-alone.

2.2 Supported Compilers

The library was built and tested using the IAR Compiler and Metrowerks CodeWarrior Compiler. The interfaces to the algorithms included in this library have been combined into a single public interface header file for each respective sub-library, that is: gflib.h, gdflib.h, and gmclib.h. This was done in order to simplify the number of files required for inclusion by application programs. See the specific algorithm sections of this document for details on the software Application Programming Interface (API), definitions, and functionality provided for the individual algorithms.

2.3 Installation

The MCLIB is delivered as a single executable file. To install the MCLIB on a user computer, run the installation file and follow these steps. Highlighted "ReleaseID" identifies the actual release number, which is Cortex M4 FSLESL r1.1.

1. Accept the license agreement.

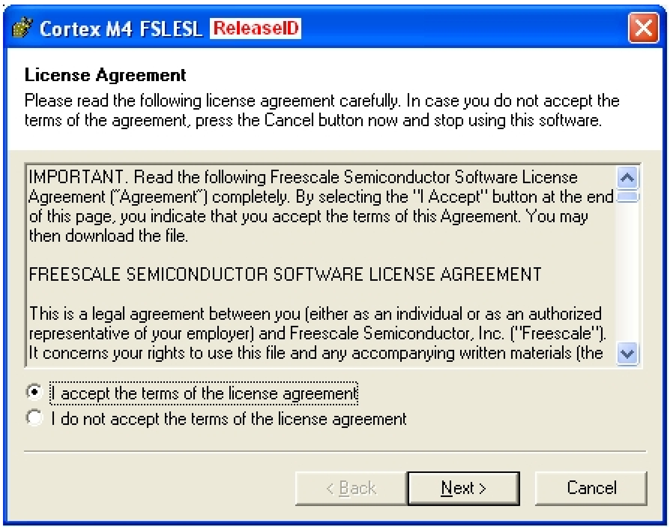


Figure 2-2: Step 1.

2. Select the destination directory.

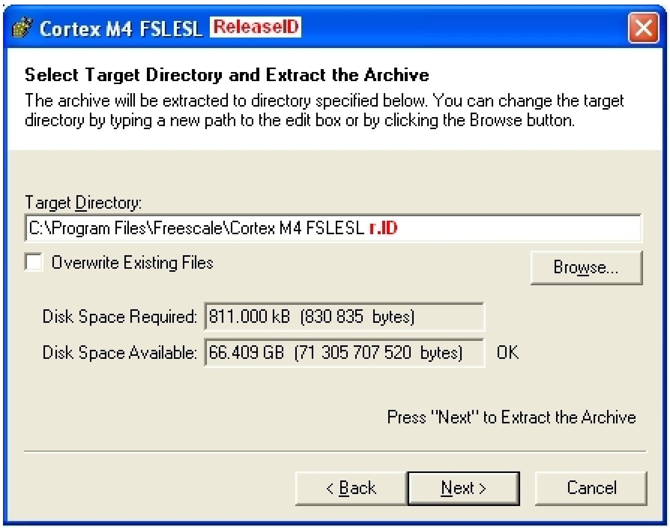


Figure 2-3: Step 2.

3. Select "Finish" to end the installation.

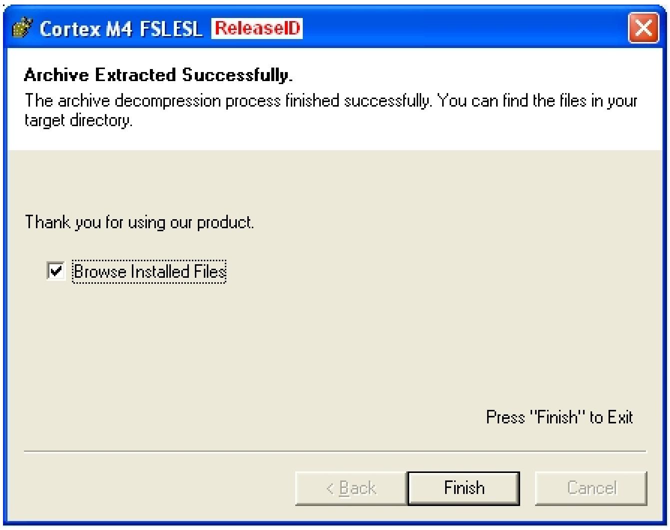
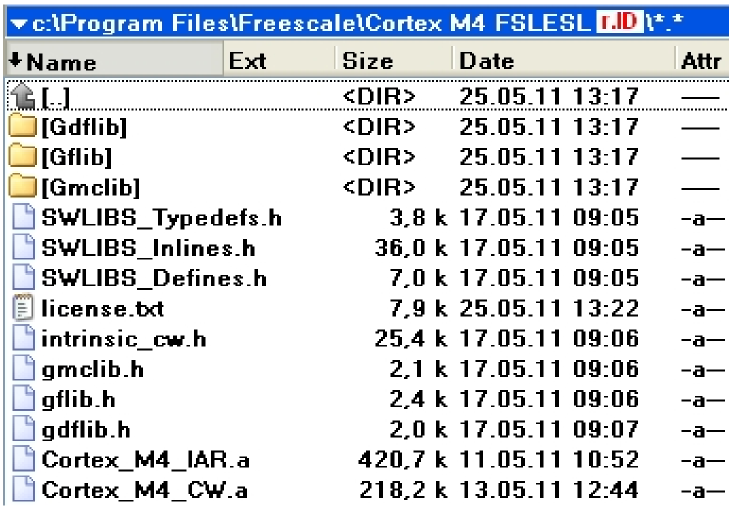


Figure 2-4: Step 3.

2.4 Library File Structure

After a successful installation, the Motor Control Library is placed by default into the: "C:\Program Files\Freescale\Cortex M4 FSLESL r1.1" subfolder. This folder will contain other nested sub-folders and files required by the Motor Control Library, as shown in Figure 2-5.



Name	Ext	Size	Date	Attr
[..]		<DIR>	25.05.11 13:17	——
[Gdflib]		<DIR>	25.05.11 13:17	——
[Gflib]		<DIR>	25.05.11 13:17	——
[Gmclib]		<DIR>	25.05.11 13:17	——
SWLIBS_Typedefs.h		3,8 k	17.05.11 09:05	-a—
SWLIBS_Inlines.h		36,0 k	17.05.11 09:05	-a—
SWLIBS_Defines.h		7,0 k	17.05.11 09:05	-a—
license.txt		7,9 k	25.05.11 13:22	-a—
intrinsic_cw.h		25,4 k	17.05.11 09:06	-a—
gmclib.h		2,1 k	17.05.11 09:06	-a—
gflib.h		2,4 k	17.05.11 09:06	-a—
gdflib.h		2,0 k	17.05.11 09:07	-a—
Cortex_M4_IAR.a		420,7 k	11.05.11 10:52	-a—
Cortex_M4_CW.a		218,2 k	13.05.11 12:44	-a—

Figure 2-5: MCLIB directory structure.

The installed directories/files include:

- Three directories - containing header files for each function
- Header files (*.h) which each contain the list of functions from relevant libraries (such as gdflib, gflib and gmclib.h). SWLIBS header files include definitions and math functions
- Library files (*.a) - containing all compiled function algorithms
- license.txt contains license agreement

In order to integrate the Motor Control Library into a new Cortex M4 core based project, follow the steps described in Section 2.5 or 2.6.

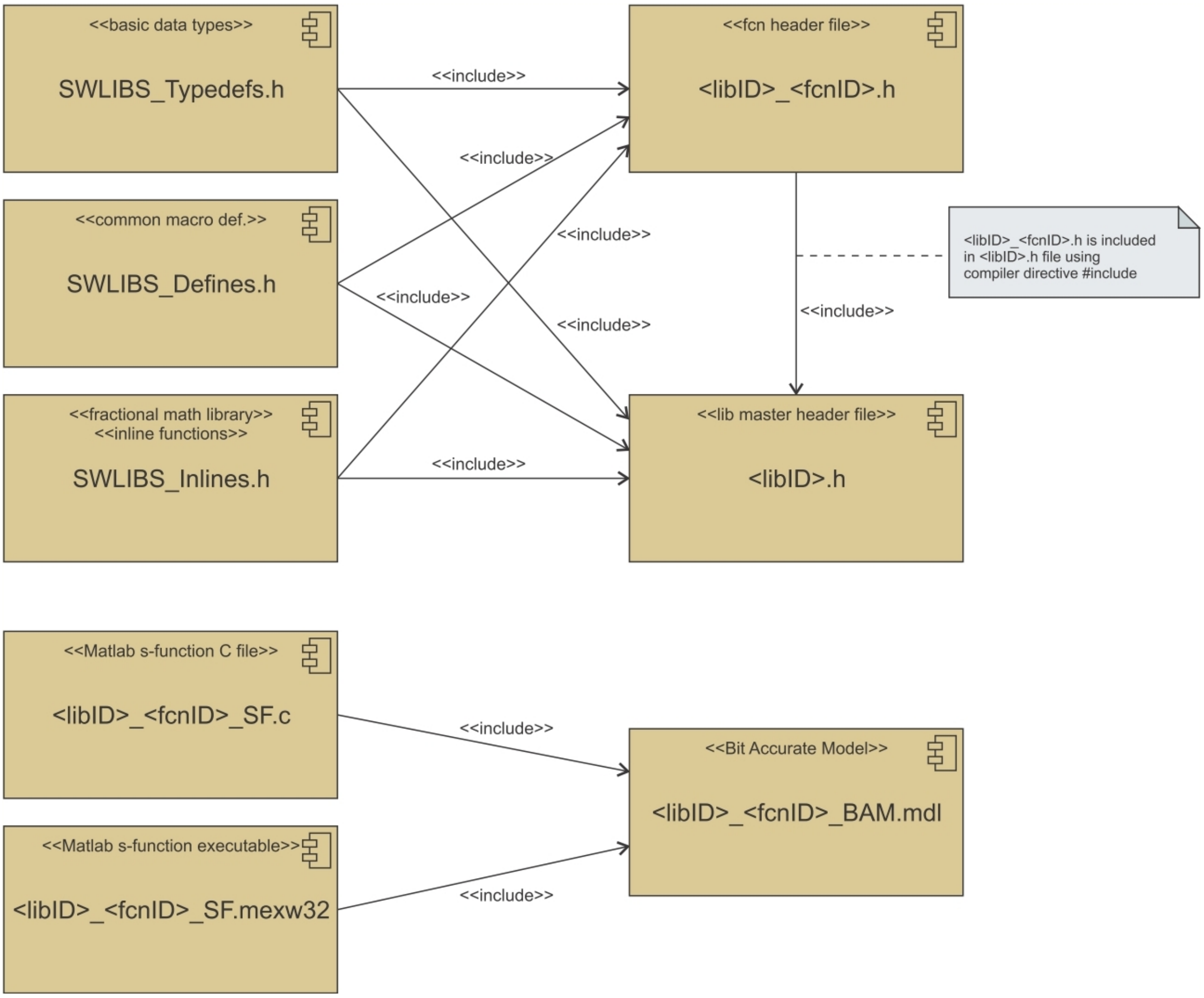


Figure 2-6: MCLIB file structure.

2.5 Library Integration into an IAR Embedded Workbench IDE

The Motor Control Library is added into an IAR Embedded Workbench IDE project by performing the following steps:

1. Open a new empty C project in the IAR Embedded Workbench IDE.
2. Add a new group by right-clicking inside the workspace section, then choose *Add* > *Add Group*. Insert name GDFLIB. Repeat to create the next two groups: GFLIB, GMCLIB.

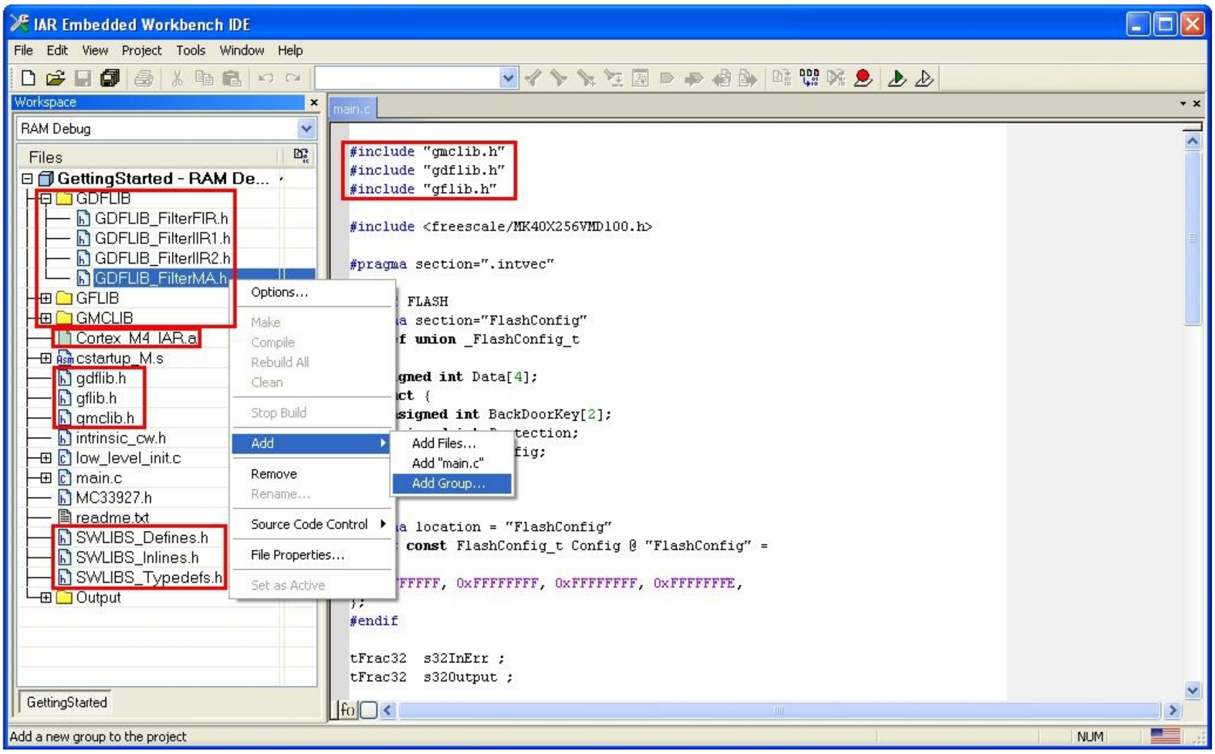


Figure 2-7: Project build

3. Open the directory where you installed the libraries. The default path is "C:\Program Files\Freescale\Cortex M4 FSLESL r1.1". Add all files into each group by right click then choosing *Add* > *Add files*. Select all files from each subdirectory (GDFLIB, GFLIB, or GMCLIB). Do not forget to add header files: *gdflib.h*, *gflib.h* and *gmclib.h* to the root level for each library. Also add the library file *Cortex_M4_IAR.h*. All files added into IAR are outlined in red in Figure 2-9.
4. Write pre processor directives *#include "gdflib.h"*, *#include "gflif.h"* and *#include "gmclib.h"* at the beginning of the *main.c* file.
5. Set up the paths for header files by choosing *Project* > *Options* from the menu (or press alt+F7). Choose the category *C/C++ Compiler* and select the *Preprocessor* tab. If the default installation path was used insert the following lines:
   c:\Program Files\Freescale\Cortex M4 FSLESL r1.1\
   c:\Program Files\Freescale\Cortex M4 FSLESL r1.1\GDFLIB\

c:\Program Files\Freescale\Cortex M4 FSLESL r1.1\GFLIB\
c:\Program Files\Freescale\Cortex M4 FSLESL r1.1\GMCLIB\
If the default installation path was not use, replace it with you own installation path.

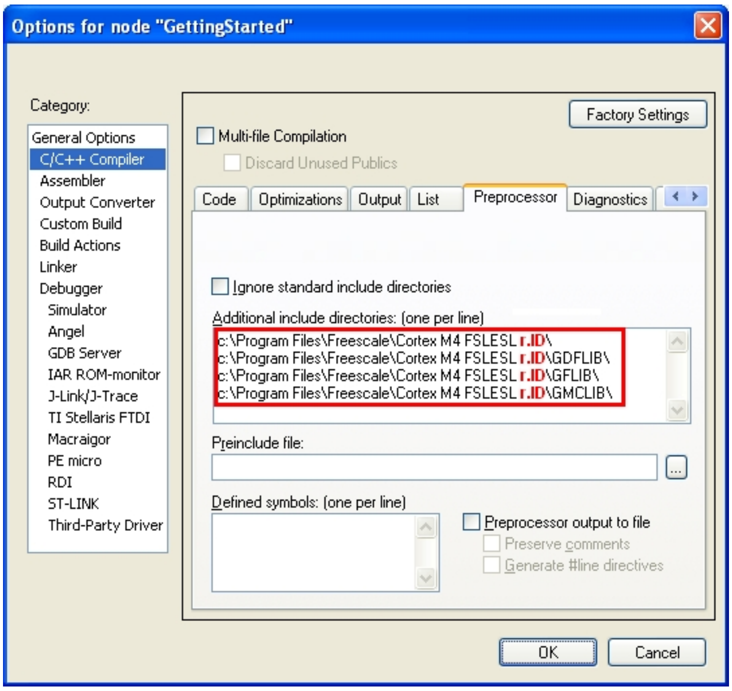


Figure 2-8: Project options.

6. The Motor Control Library is now ready for use.

2.6 Library Integration into an Metrowerks CodeWarrior Development Studio

The Motor Control Library is added into a Metrowerks CodeWarrior Development Studio project by performing the following steps:

1. Open a new empty C project in the CodeWarrior Development Studio.

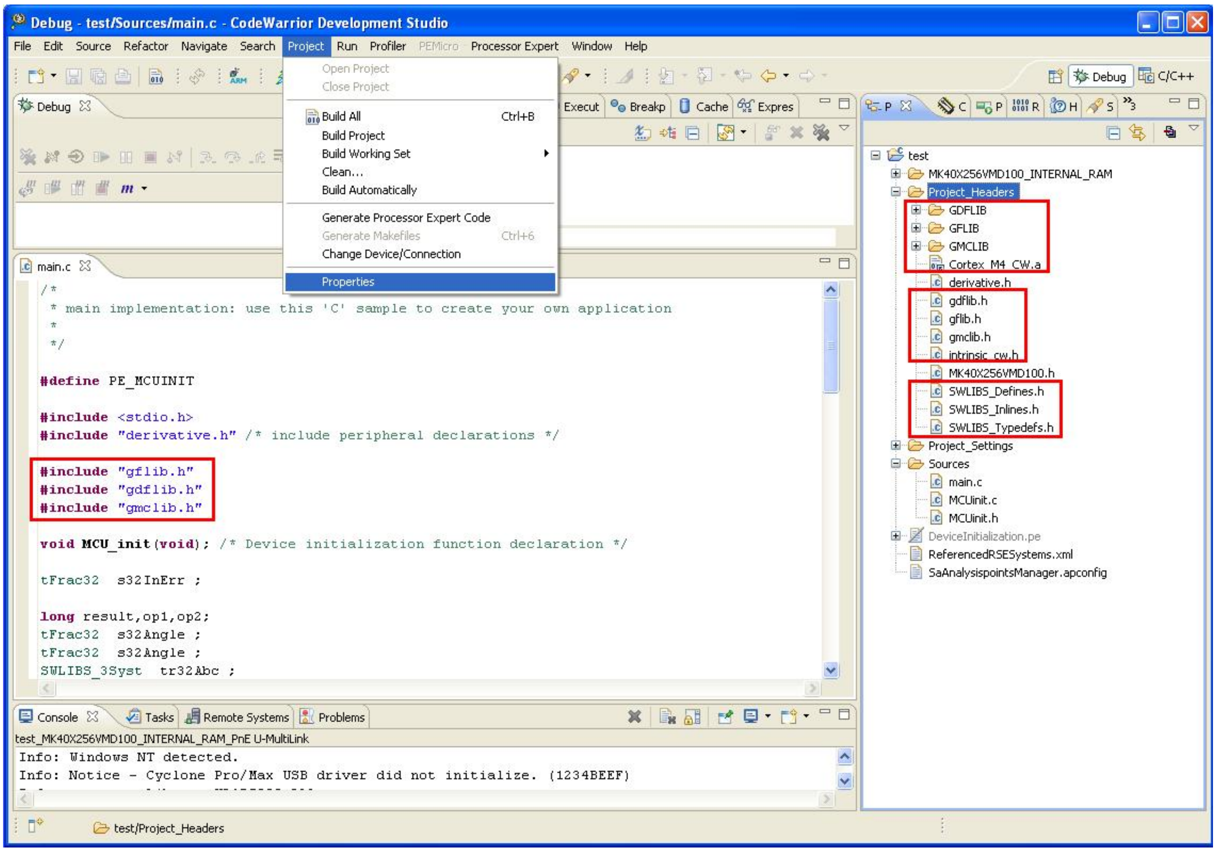


Figure 2-9: Project build.

2. Open the directory where you installed the libraries. The default path is "C:\Program Files\Freescale\Cortex M4 FSLESL r1.1". Drag and drop files into the workspace from each subdirectory (GDFLIB, GFLIB, or GMCLIB). Do not forget to add header files: *gdflib.h*, *gflib.h*, and *gmclib.h* to the root level for each library. Also add the library file: *Cortex_M4_CW.a*. All files added into CW are marked outlined in red in Figure 2-9.

3. Write pre processor directives *#include "gdflib.h"*, *#include "gflif.h"*, and *#include "gmclib.h"* at the beginning of the *main.c* file.

4. Set up the paths for header files and the library file by choosing *Project > Properties* from the menu. Expand setting *C/C++ General*, choose the subsetting *Paths and Symbols*, select the *Includes* tab, and select *C Source File*. If the default installation path was used, click *Add...* and insert the following lines one at the time:

c:\Program Files\Freescale\Cortex M4 FSLESL r1.1\
c:\Program Files\Freescale\Cortex M4 FSLESL r1.1\GDFLIB\
c:\Program Files\Freescale\Cortex M4 FSLESL r1.1\GFLIB\
c:\Program Files\Freescale\Cortex M4 FSLESL r1.1\GMCLIB\
If the default installation path was not use, replace it with you own installation path.

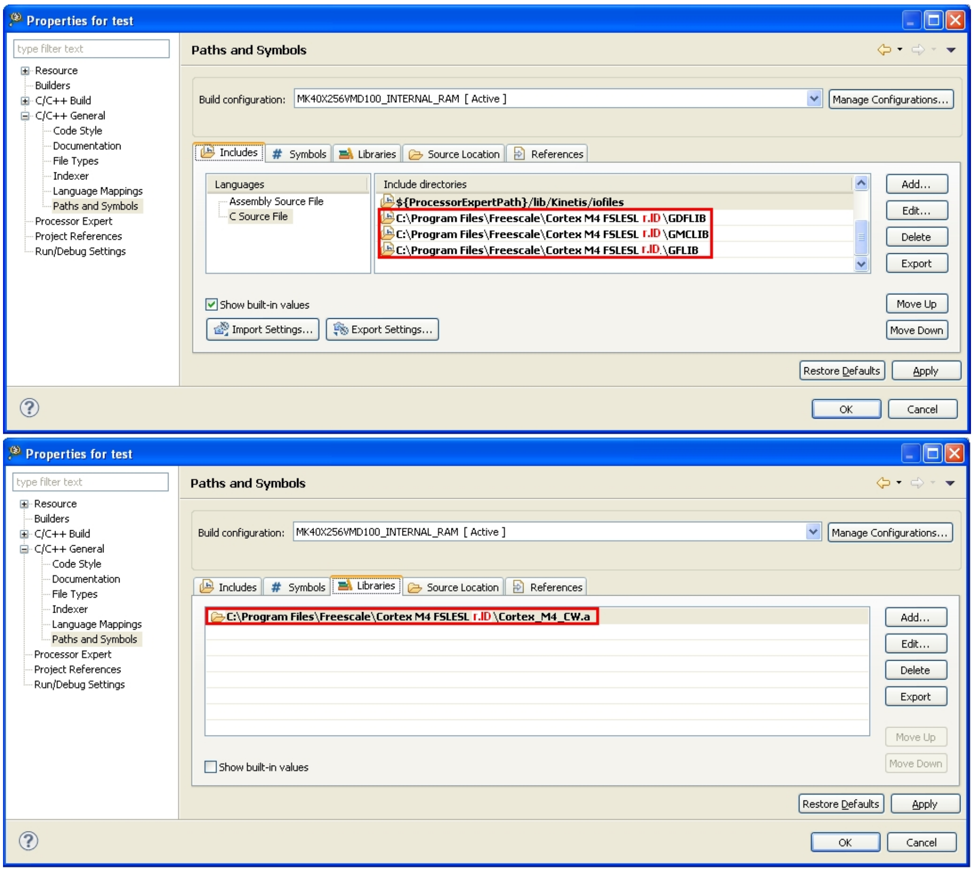


Figure 2-10: Project options.

5. Select the *Libraries* tab and click *Add...*. Write:
   c:\ Program Files\ Freescale\ Cortex M4 FSLESL r1.1\ GDFLIB\ Cortex_M4_CW.a.
   All modified items are marked outlined in red in Figure 2-9.

6. The Motor Control Library is now ready for use.

CHAPTER 3: GFLIB

3.1 Function API Overview

- tFrac32 GFLIB_Sin (tFrac32 s32In, const GFLIB_SINTLR_T *const pParam)
  Sine function - Taylor polynomial approximation
- tFrac32 GFLIB_Cos (tFrac32 s32In, const GFLIB_COSTLR_T *const pParam)
  Cosine function - Taylor polynomial approximation
- tFrac32 GFLIB_Tan (tFrac32 s32In, const GFLIB_TANTLR_T *const pParam)
  Tangent function - Piece-wise polynomial approximation
- tFrac32 GFLIB_Asin (tFrac32 s32In, const GFLIB_ASIN_TAYLOR_T *const pParam)
  Arcsine function - Piece-wise polynomial approximation
- tFrac32 GFLIB_Acos (tFrac32 s32In, const GFLIB_ACOS_TAYLOR_T *const pParam)
  Arccosine function - Piece-wise polynomial approximation
- tFrac32 GFLIB_Atan (tFrac32 s32In, const GFLIB_ATAN_TAYLOR_T *const pParam)
  Arctangent function - Taylor polynomial approximation
- tS32 GFLIB_AtanYX (tS32 s32InY, tS32 s32InX)
  Calculates the angle between the positive x-axis and the direction of a vector given by the (x, y) coordinates
- tFrac32 GFLIB_AtanYXShifted (tFrac32 s32InY, tFrac32 s32InX, const GFLIB_ATANYXSHIFTED_-T *const pParam)
  Calculates the angle of two sine waves shifted in phase one to each other
- tFrac32 GFLIB_Sqrt (tFrac32 s32In)
  Square root function
- tFrac32 GFLIB_Sign (tFrac32 s32In)
  Sign function
- tFrac32 GFLIB_Lut1D (tFrac32 s32In, const GFLIB_LUT1D_T *const pParam)
  Performs a linear interpolation over an arbitrary data set
- tFrac32 GFLIB_Hyst (tFrac32 s32In, GFLIB_HYST_T * pParam)
  Hysteresis function
- tFrac32 GFLIB_Ramp (tFrac32 s32In, GFLIB_RAMP_T * pParam)
  Calculates the up/down ramp with the step increment/decrement defined in the pParam

structure

- tFrac32 GFLIB_Limit (tFrac32 s32In, const GFLIB_LIMIT_T *const pParam)
  Limits the input value to the defined upper and lower limits
- tFrac32 GFLIB_LowerLimit (tFrac32 s32In, const GFLIB_LOWERLIMIT_T *const pParam)
  Limits the input value by the defined lower limit
- tFrac32 GFLIB_UpperLimit (tFrac32 s32In, const GFLIB_UPPERLIMIT_T *const pParam)
  Limits the input value by the defined upper limit
- tBool GFLIB_VectorLimit (SWLIBS_2Syst *const pOut, const SWLIBS_2Syst *const pIn, const GFLIB_VECTORLIMIT_T *const pParam)
  Limits magnitude of the input vector
- tFrac32 GFLIB_IntegratorTR (tFrac32 s32In, GFLIB_INTEGRATOR_TR_T * pParam)
  Calculates a discrete implementation of the integrator (sum), discretized using a trapezoidal (Bilinear) transformation
- tFrac32 GFLIB_ControllerPIr (tFrac32 s32InErr, GFLIB_CONTROLLER_PI_R_T * pParam)
  Recurrent form Proportional-Integral controller without integrator anti-windup functionality
- tFrac32 GFLIB_ControllerPIp (tFrac32 s32InErr, GFLIB_CONTROLLER_PI_P_T * pParam)
  Calculates the parallel form of the Proportional-Integral (PI) controller
- tFrac32 GFLIB_ControllerPIpAW (tFrac32 s32InErr, GFLIB_CONTROLLER_PIAW_P_T * pParam)
  Calculates the parallel form of the Proportional-Integral (PI) controller with implemented integral anti-windup functionality

3.2 GFLIB_Sin

3.2.1 Declaration

```
tFrac32 GFLIB_SinANSIC(tFrac32 s32In, const GFLIB_SINTLR_T *const pParam)
```

3.2.2 Alias

```
#define GFLIB_Sin(s32In) \
        GFLIB_SinANSIC(s32In, &gflibSinCoef)
```

3.2.3 Arguments

Table 3-1: Function parameters

Type	Name	Dir.	Description
const GFLIB_SINTLR_T *const	pParam	in	Pointer to an array of Taylor coefficients. Using the function alias GFLIB_Sin, default coefficients are used.
tFrac32	s32In	in	Input argument is a 32-bit number that contains an angle in radians between $[-\pi, \pi)$ normalized between $[-1, 1)$.

3.2.4 Return

The function returns the sine of the input argument as a fixed point 32-bit number, normalized between $[-1, 1)$.

3.2.5 Description

The GFLIB_SinANSIC function, denoting ANSI-C compatible source code implementation, can be called via the function alias GFLIB_Sin.

The GFLIB_Sin function provides a computational method for calculation of a standard trigonometric *sine* function $\sin(x)$, using the 9th order Taylor polynomial approximation. The Taylor polynomial approximation of a *sine* function is described as follows:

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} + \cdots = \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n+1}}{(2n+1)!} \tag{3.1}$$

where x is the input angle.

The 9th order polynomial approximation was chosen for its ability to achieve the best ratio between calculation accuracy and speed of calculation. The goal is to have the output error within 3LSB on the upper 16 bits of the 32-bit result. Because the GFLIB_Sin function is implemented with consideration to fixed point fractional arithmetic, all variables are normalized to fit into the $[-1, 1)$ range. Therefore, in order to cast the fractional value of the input angle s32In $[-1, 1)$ into the correct range $[-\pi, \pi)$, the input s32In must be multiplied by π. The fixed point fractional implementation of the GFLIB_Sin function, using 9th order Taylor approximation, is given as follows:

$$\sin(\pi \cdot s32In) = (\pi \cdot s32In) - \frac{(\pi \cdot s32In)^3}{3!} + \frac{(\pi \cdot s32In)^5}{5!} - \frac{(\pi \cdot s32In)^7}{7!} + \frac{(\pi \cdot s32In)^9}{9!} \tag{3.2}$$

The 9th order polynomial approximation of the sine function has a high accuracy in the range $\left[-\frac{\pi}{2}, \frac{\pi}{2}\right)$ of the argument, but in wider ranges the calculation error quickly increases. To minimize the error without having to use a higher order polynomial, the symmetry of the sine function $\sin(x) = \sin(\pi - x)$ is utilized. Therefore, the input argument is transferred to be always in the range $\left[-\frac{\pi}{2}, \frac{\pi}{2}\right)$ and the Taylor polynomial is calculated only in the range of the argument $\left[-\frac{\pi}{2}, \frac{\pi}{2}\right)$. To make calculations more precise (because in calculations the value used is the input angle rounded to a 16-bit fractional number), the given argument value s32In (that is to be transferred into the range $[-0.5, 0.5)$ due to the *sine* function symmetry) is shifted by 1 bit to the left (multiplied by 2). Then, the value of $s32In^2$, used in the calculations, is in the range $[-1, 1)$ instead of $[-0.25, 0.25)$. Shifting the input value by 1 bit to the left will increase the accuracy of the calculated $\sin(\pi \cdot s32In)$ function. Implementing such a scale on the approximation function described by Equation (3.2), results in the following:

$$\begin{aligned} &\sin\left(s32In \cdot 2 \cdot \tfrac{\pi}{2}\right) = \ldots \\ &\ldots = \left(\frac{\left(s32In \cdot 2 \cdot \frac{\pi}{2}\right)}{2} - \frac{\left(s32In \cdot 2 \cdot \frac{\pi}{2}\right)^3}{3! \cdot 2} + \frac{\left(s32In \cdot 2 \cdot \frac{\pi}{2}\right)^5}{5! \cdot 2} - \frac{\left(s32In \cdot 2 \cdot \frac{\pi}{2}\right)^7}{7! \cdot 2} + \frac{\left(s32In \cdot 2 \cdot \frac{\pi}{2}\right)^9}{9! \cdot 2}\right) \cdot 2 \end{aligned} \tag{3.3}$$

Equation (3.3) can be further rewritten into the following form:

$$\begin{aligned} \sin\left(s32In \cdot \pi\right) &= (s32In \cdot 2)(a_1 + \\ &+ (s32In \cdot 2)^2(a_2 + \\ &+ (s32In \cdot 2)^2(a_3 + \\ &+ (s32In \cdot 2)^2(a_4 + \\ &+ (s32In \cdot 2)^2(a_5)))))) \cdot 2 \end{aligned} \tag{3.4}$$

where $a_1 \ldots a_5$ are coefficients of the approximation polynomial, which are calculated as follows (represented as 32-bit signed fractional numbers):

$$\begin{aligned} a_1 &= \frac{\left(\frac{\pi}{2}\right)}{2} &= 0.785398163397448 &\Rightarrow \frac{\left(\frac{\pi}{2}\right)}{2} \cdot 2^{31} &= 0x6487ED51 \\ a_3 &= -\frac{\left(\frac{\pi}{2}\right)^3}{3! \cdot 2} &= -0.322982048753123 &\Rightarrow -\frac{\left(\frac{\pi}{2}\right)^3}{3! \cdot 2} \cdot 2^{31} &= 0xD6A88634 \\ a_5 &= \frac{\left(\frac{\pi}{2}\right)^5}{5! \cdot 2} &= 0.0398463131230835 &\Rightarrow \frac{\left(\frac{\pi}{2}\right)^5}{5! \cdot 2} \cdot 2^{31} &= 0x0519AF1A \\ a_7 &= -\frac{\left(\frac{\pi}{2}\right)^7}{7! \cdot 2} &= -0.00234087706765934 &\Rightarrow -\frac{\left(\frac{\pi}{2}\right)^7}{7! \cdot 2} \cdot 2^{31} &= 0xFFB34B4D \\ a_9 &= \frac{\left(\frac{\pi}{2}\right)^9}{9! \cdot 2} &= 8.02205923936799e^{-005} &\Rightarrow \frac{\left(\frac{\pi}{2}\right)^9}{9! \cdot 2} \cdot 2^{31} &= 0x0002A0F0 \end{aligned} \tag{3.5}$$

Therefore, the resulting equation has the following form:

$$\begin{aligned} \sin\left(s32In \cdot \pi\right) &= (s32In \cdot 2)(0x6487ED51 + \\ &+ (s32In \cdot 2)^2(0xD6A88634 + \\ &+ (s32In \cdot 2)^2(0x0519AF1A + \\ &+ (s32In \cdot 2)^2(0xFFB34B4D + \\ &+ (s32In \cdot 2)^2(0x0002A0F0)))))) \cdot 2 \end{aligned} \tag{3.6}$$

Figure 3-1 depicts a floating point *sine* function generated from Matlab and the approximated value of the *sine* function obtained from GFLIB_Sin, as well as their difference. The course of calculation accuracy as a function of the input angle can be observed from this figure. The achieved accuracy with consideration to the 9th order Taylor approximation and described fixed point scaling, is less than 3LSB on the upper 16 bits of the 32-bit result.

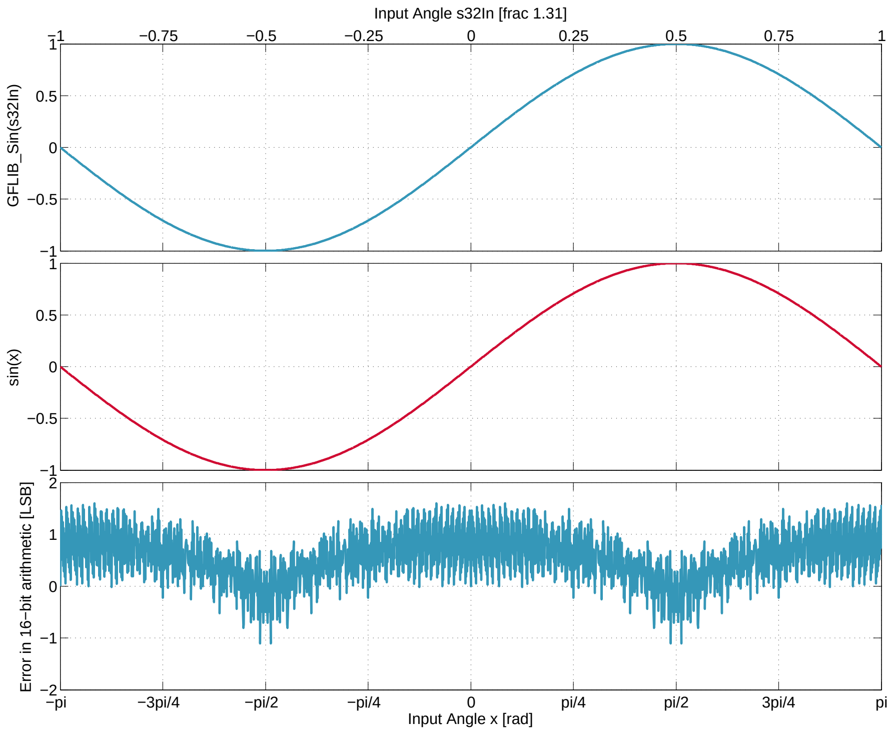


Figure 3-1: sin(x) versus GFLIB_Sin(s32In)

3.2.6 Note

The input angle (s32In) is normalized into the range $[-1, 1)$. The polynomial coefficients are stored in a locally-defined structure with five members, the call of which is masked by the function alias GFLIB_Sin. The polynomial coefficients can be calculated by the user, and in which case the full GFLIB_SinANSIC function call with a pointer to the newly-defined coefficients shall be used instead of the function alias.

3.2.7 Reentrancy

The function is reentrant.

3.2.8 Code Example

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

```
tFrac32 s32Angle;
tFrac32 s32Output;

void main(void)
{
    // input angle = 0.5 => pi/2
    s32Angle  = FRAC32(0.5);

    // output should be 0x7FFF8000
    s32Output = GFLIB_Sin(s32Angle);
}
```

3.2.9 Performance

Table 3-2: GFLIB_Sin function performance

Code size [bytes] IAR	88
Data size [bytes] IAR	0
Execution clock cycles max [clk] IAR	44
Execution clock cycles min [clk] IAR	44

3.3 GFLIB_Cos

3.3.1 Declaration

```
tFrac32 GFLIB_CosANSIC(tFrac32 s32In, const GFLIB_COSTLR_T *const pParam)
```

3.3.2 Alias

```
#define GFLIB_Cos(s32In) \
        GFLIB_CosANSIC(s32In, &gflibCosCoef)
```

3.3.3 Arguments

Table 3-3: Function parameters

Type	Name	Dir.	Description
const GFLIB_COSTLR_T *const	pParam	in	Pointer to an array of Taylor coefficients. Using the function alias GFLIB_Cos, default coefficients are used.
tFrac32	s32In	in	Input argument is a 32-bit number that contains an angle in radians between $[-\pi, \pi)$ normalized between $[-1, 1)$.

3.3.4 Return

The function returns the cos of the input argument as a fixed point 32-bit number, normalized between $[-1, 1)$.

3.3.5 Description

The GFLIB_CosANSIC function, denoting ANSI-C compatible source code implementation, can be called via the function alias GFLIB_Cos.

The GFLIB_Cos function provides a computational method for calculation of a standard trigonometric *cosine* function $\cos(x)$, using the 9th order Taylor polynomial approximation of the *sine* function. The following two equations describe the chosen approach of calculating the *cosine* function:

$$\cos(s32In) = \sin(\frac{\pi}{2} + s32In) = \sin(x) \tag{3.7}$$

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} + \cdots = \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n+1}}{(2n+1)!} \tag{3.8}$$

The 9th order polynomial approximation is chosen as sufficient an order to achieve the best ratio between calculation accuracy and speed of calculation. The goal is to have the output error within 3LSB on the upper 16 bits of the 32-bit result. Because the GFLIB_Sin function is implemented with consideration to fixed point fractional arithmetic, all variables are normalized to fit into the $[-1, 1)$ range. Therefore, in order to cast the fractional value of the input angle s32In $[-1, 1)$ into the correct range $[-\pi, \pi)$, the input s32In must be multiplied by π. The fixed point fractional implementation of the GFLIB_Sin function, using 9th order Taylor approximation, is given as follows:

$$\sin(\pi \cdot s32In) = (\pi \cdot s32In) - \frac{(\pi \cdot s32In)^3}{3!} + \frac{(\pi \cdot s32In)^5}{5!} - \frac{(\pi \cdot s32In)^7}{7!} + \frac{(\pi \cdot s32In)^9}{9!} \tag{3.9}$$

The 9th order polynomial approximation of the sine function has a high accuracy in the range $\left[-\frac{\pi}{2}, \frac{\pi}{2}\right)$ of the argument, but the calculation error is quickly increases in wider ranges. To minimize the error without having to use a higher order polynomial, the symmetry of the sine function $\sin(x) = \sin(\pi - x)$ is utilized. Therefore, the input argument is transferred to be always in the range $\left[-\frac{\pi}{2}, \frac{\pi}{2}\right)$ and the Taylor polynomial is calculated only in the range of the argument $\left[-\frac{\pi}{2}, \frac{\pi}{2}\right)$.

To make calculations more precise (because in calculations the value used is the input angle rounded to a 16-bit fractional number), the given argument value s32In (that is to be transferred into the range $[-0.5, 0.5)$ due to the *sine* function symmetry) is shifted by 1 bit to the left (multiplied by 2). Then, the value of $s32In^2$, when used in the calculations, is in the range $[-1, 1)$ instead of $[-0.25, 0.25]$. Shifting the input value by 1 bit to the left will increase the accuracy of the calculated $\sin(\pi \cdot s32In)$ function. Implementing such a scale on the approximation function described by Equation (3.9), results in the following:

$$\begin{aligned} &\sin\left(s32In \cdot 2 \cdot \tfrac{\pi}{2}\right) = \ldots \\ &\ldots = \left(\frac{\left(s32In \cdot 2 \cdot \frac{\pi}{2}\right)}{2} - \frac{\left(s32In \cdot 2 \cdot \frac{\pi}{2}\right)^3}{3! \cdot 2} + \frac{\left(s32In \cdot 2 \cdot \frac{\pi}{2}\right)^5}{5! \cdot 2} - \frac{\left(s32In \cdot 2 \cdot \frac{\pi}{2}\right)^7}{7! \cdot 2} + \frac{\left(s32In \cdot 2 \cdot \frac{\pi}{2}\right)^9}{9! \cdot 2}\right) \cdot 2 \end{aligned} \tag{3.10}$$

Equation (3.10) can be further rewritten into the following form:

$$\begin{aligned} \sin(s32In \cdot \pi) &= (s32In \cdot 2)(a_1 + \\ &+ (s32In \cdot 2)^2(a_2 + \\ &+ (s32In \cdot 2)^2(a_3 + \\ &+ (s32In \cdot 2)^2(a_4 + \\ &+ (s32In \cdot 2)^2(a_5)))))) \cdot 2 \end{aligned} \tag{3.11}$$

where $a_1 \ldots a_5$ are coefficients of the approximation polynomial, which are calculated as follows (represented as 32-bit signed fractional numbers):

$$\begin{aligned} a_1 &= \frac{\left(\frac{\pi}{2}\right)}{2} &&= 0.785398163397448 &&\Rightarrow \frac{\left(\frac{\pi}{2}\right)}{2} \cdot 2^{31} &&= 0x6487ED51 \\ a_3 &= -\frac{\left(\frac{\pi}{2}\right)^3}{3! \cdot 2} &&= -0.322982048753123 &&\Rightarrow -\frac{\left(\frac{\pi}{2}\right)^3}{3! \cdot 2} \cdot 2^{31} &&= 0xD6A88634 \\ a_5 &= \frac{\left(\frac{\pi}{2}\right)^5}{5! \cdot 2} &&= 0.0398463131230835 &&\Rightarrow \frac{\left(\frac{\pi}{2}\right)^5}{5! \cdot 2} \cdot 2^{31} &&= 0x0519AF1A \\ a_7 &= -\frac{\left(\frac{\pi}{2}\right)^7}{7! \cdot 2} &&= -0.00234087706765934 &&\Rightarrow -\frac{\left(\frac{\pi}{2}\right)^7}{7! \cdot 2} \cdot 2^{31} &&= 0xFFB34B4D \\ a_9 &= \frac{\left(\frac{\pi}{2}\right)^9}{9! \cdot 2} &&= 8.02205923936799e^{-005} &&\Rightarrow \frac{\left(\frac{\pi}{2}\right)^9}{9! \cdot 2} \cdot 2^{31} &&= 0x0002A0F0 \end{aligned} \tag{3.12}$$

Therefore, the resulting equation has the following form:

$$\begin{aligned}\sin(s32In\cdot\pi) &= (s32In\cdot 2)(0x6487ED51+\\ &+ (s32In\cdot 2)^2(0xD6A88634+\\ &+ (s32In\cdot 2)^2(0x0519AF1A+\\ &+ (s32In\cdot 2)^2(0xFFB34B4D+\\ &+ (s32In\cdot 2)^2(0x0002A0F0)))))\cdot 2\end{aligned} \tag{3.13}$$

Figure 3-2 depicts a floating point *cosine* function generated from Matlab and the approximated value of the *cosine* function obtained from GFLIB_Cos, as well as their difference. The course of calculation accuracy as a function of the input angle can be observed from this figure. The achieved accuracy with consideration to the 9th order Taylor approximation and described fixed point scaling is less than 3LSB on the upper 16 bits of the 32-bit result.

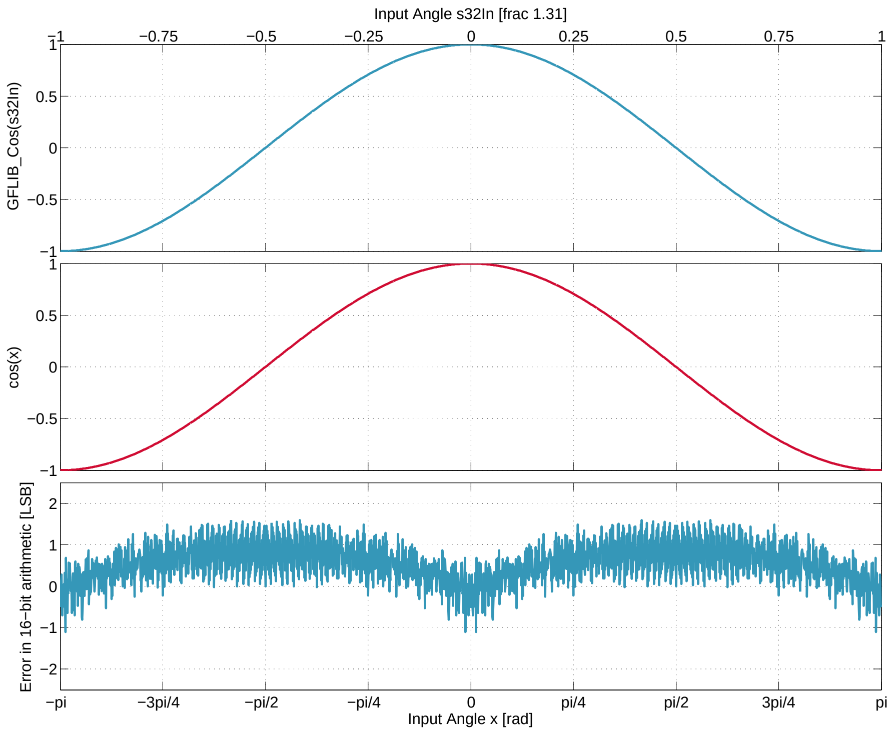


Figure 3-2: cos(x) versus GFLIB_Cos(s32In)

3.3.6 Note

The input angle (s32In) is normalized into the range $[-1, 1)$. The polynomial coefficients are stored in a locally-defined structure with five members, the call of which is masked by the function alias GFLIB_Cos. The polynomial coefficients can be calculated by the user, in which case the GFLIB_CosANSIC function call with a pointer to the newly defined coefficients shall be used instead of the function alias.

3.3.7 Reentrancy

The function is reentrant.

3.3.8 Code Example

```
#include "gflib.h"

tFrac32 s32Angle;
tFrac32 s32Output;

void main(void)
{
    // input angle = 0.25 => pi/4
    s32Angle  = FRAC32(0.25);

    // output should be 0x5A824000
    s32Output = GFLIB_Cos(s32Angle);
}
```

3.3.9 Performance

Table 3-4: GFLIB_Cos function performance

Code size [bytes] IAR	92
Data size [bytes] IAR	0
Execution clock cycles max [clk] IAR	44
Execution clock cycles min [clk] IAR	44

3.4 GFLIB_Tan

3.4.1 Declaration

```
tFrac32 GFLIB_TanANSIC(tFrac32 s32In, const GFLIB_TANTLR_T *const pParam)
```

3.4.2 Alias

```
#define GFLIB_Tan(s32In) \
        GFLIB_TanANSIC(s32In, &gflibTanCoef)
```

3.4.3 Arguments

Table 3-5: Function parameters

Type	Name	Dir.	Description
const GFLIB_TANTLR_T *const	pParam	in	Pointer to an array of Taylor coefficients. Using the function alias GFLIB_Tan, default coefficients are used.
tFrac32	s32In	in	Input argument is a 32-bit number that contains an angle in radians between $[-\pi, \pi)$ normalized between $[-1, 1)$.

3.4.4 Return

The function returns $\tan(\pi \cdot s32In)$ as a fixed point 32-bit number, normalized between $[-1, 1)$.

3.4.5 Description

The GFLIB_TanANSIC function, denoting ANSI-C compatible source code implementation, can be called via the function alias GFLIB_Tan.

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

$$\tan(x) \equiv \frac{\sin(x)}{\cos(x)} \tag{3.14}$$

Because both $\sin(x)$ and $\cos(x)$ are defined on interval $[-\pi, \pi)$, function $\tan(x)$ is equal to zero when $\sin(x) = 0$ and is equal to infinity when $\cos(x) = 0$. Therefore, the *tangent* function has asymptotes at $n \cdot \frac{\pi}{2}$ for $n = \pm1, \pm3, \pm5 \ldots$. The graph of $\tan(x)$ is shown in Figure 3-3.

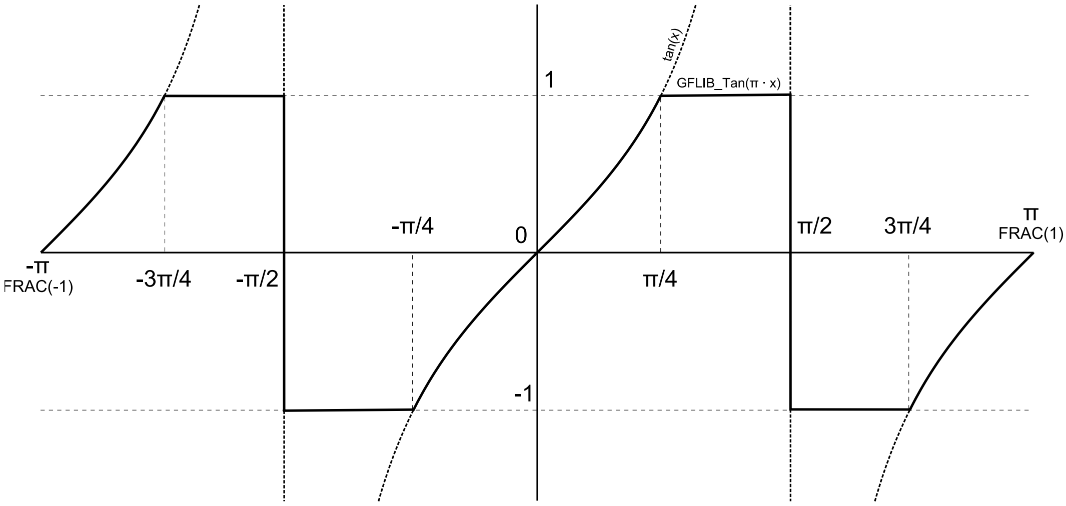


Figure 3-3: Course of the function GFLIB_Tan

The GFLIB_Tan function is implemented with consideration to fixed point fractional arithmetic, hence all tangent values falling beyond $[-1, 1)$ are truncated to -1 and 1 respectively. This truncation is applied for angles in the ranges $[-\frac{3\pi}{4}, -\frac{\pi}{4})$ and $[\frac{\pi}{4}, \frac{3\pi}{4})$. As can be further seen from Figure 3-3, tangent values are identical for angles in the ranges:

1. $[-\frac{\pi}{4}, 0)$ and $[\frac{3\pi}{4}, \pi)$
2. $[-\pi, -\frac{3\pi}{4})$ and $[0, \frac{\pi}{4})$

Moreover, it can be observed from Figure 3-3 that the course of the $\tan(x)$ function output for angles in the first range. is identical, but with the opposite sign, to output for angles in second range. Therefore, the approximation of the *tangent* function over the entire defined range of input angles can be simplified to an approximation for angles in the range $[0, \frac{\pi}{4})$, and then, depending on the input angle, the result will be negated. In order to increase the accuracy of approximation without the need for a higher order polynomial, the interval $[0, \frac{\pi}{4})$ is further divided into eight equally-spaced subintervals, with polynomial approximation done for each interval respectively. Such a division results in eight sets of polynomial coefficients. Moreover, it allows for the use of a polynomial of only the 4th order to achieve an accuracy of less than 0.5LSB (on the upper 16 bits of 32-bit results) across the full range of input angles.

The GFLIB_Tan function uses fixed point fractional arithmetic, so to cast the fractional value of the input angle s32In $[-1, 1)$ into the correct range $[-\pi, \pi)$, the fixed point input angle s32In must be multiplied by π. Then the fixed point fractional implementation of the approximation polynomial, used for calculation of each sub sector, is defined as follows:

$$\text{s32Dump} = a_1 \cdot s32In^3 + a_2 \cdot s32In^2 + a_3 \cdot s32In + a_4 \tag{3.15}$$

$$\tan(\pi \cdot s32In) = \begin{cases} \text{s32Dump} & \text{if} \quad -1 \leq \text{s32In} < -0.5 \quad \text{or} \quad 0 \leq \text{s32In} < 0.5 \\ \text{-s32Dump} & \text{if} \quad -0.5 \leq \text{s32In} < 0 \quad \text{or} \quad 0.5 \leq \text{s32In} < 1 \end{cases} \tag{3.16}$$

The division of the $[0, \frac{\pi}{4})$ interval into eight subintervals, with polynomial coefficients calculated for each subinterval, is noted in Table 3-6. Polynomial coefficients were obtained using the

Matlab fitting function, where a polynomial of the 4th order was used for the fitting of each respective subinterval.

Table 3-6: Integer Polynomial coefficients for each interval

Interval	a_1	a_2	a_3	a_4
$\langle 0, \frac{\pi}{32})$	86016	256000	105668608	105498624
$\langle \frac{\pi}{32}, \frac{2\pi}{32})$	92160	786432	107732992	318550016
$\langle \frac{2\pi}{32}, \frac{3\pi}{32})$	106496	1380352	112027648	537917440
$\langle \frac{3\pi}{32}, \frac{4\pi}{32})$	133120	2091008	118910976	768380928
$\langle \frac{4\pi}{32}, \frac{5\pi}{32})$	174080	3000320	128995328	1015683072
$\langle \frac{5\pi}{32}, \frac{6\pi}{32})$	239616	4225024	143284224	1287151616
$\langle \frac{6\pi}{32}, \frac{7\pi}{32})$	348160	5963776	163397632	1592680448
$\langle \frac{7\pi}{32}, \frac{8\pi}{32})$	536576	8568832	192008192	1946363904

Figure 3-4 depicts a floating point *tangent* function generated from Matlab and the approximated value of the *tangent* function obtained from GFLIB_Tan, as well as their difference. The course of calculation accuracy as a function of the input angle can be observed from this figure. The achieved accuracy, with consideration to the 4th order piece-wise polynomial approximation and described fixed point scaling, is less than 0.5LSB on the upper 16 bits of the 32-bit result.

3.4.6 Note

The input angle (s32In) is normalized into the range $[-1, 1)$. The polynomial coefficients are stored in a locally-defined structure, the call of which is masked by the function alias GFLIB_Tan. The polynomial coefficients can be calculated by the user and in such a case the full GFLIB_TanANSIC function call with a pointer to the newly-defined coefficients shall be used instead of the function alias.

3.4.7 Reentrancy

The function is reentrant.

3.4.8 Code Example

```
#include "gflib.h"

tFrac32 s32Angle;
tFrac32 s32Output;

void main(void)
{
    // input angle = 0.25 => pi/4
```

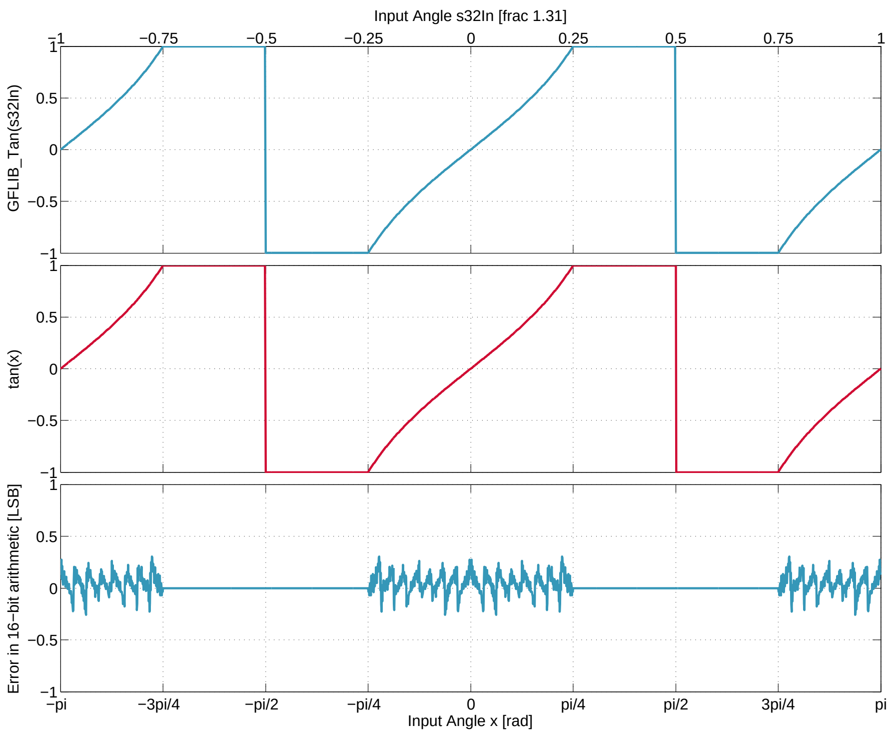


Figure 3-4: tan(x) versus GFLIB_Tan(s32In)

```
    s32Angle   = FRAC32(0.25);

    // output should be 0x7FFFFFFF = 1
    s32Output  = GFLIB_Tan(s32Angle);
}
```

3.4.9 Performance

Table 3-7: GFLIB_Tan function performance

Code size [bytes] IAR	232
Data size [bytes] IAR	0
Execution clock cycles max [clk] IAR	63
Execution clock cycles min [clk] IAR	63

3.5 GFLIB_Asin

3.5.1 Declaration

```
tFrac32 GFLIB_AsinANSIC(tFrac32 s32In, const GFLIB_ASIN_TAYLOR_T *const pParam)
```

3.5.2 Alias

```
#define GFLIB_Asin(x) \
        GFLIB_AsinANSIC((x), &gflibAsinCoef)
```

3.5.3 Arguments

Table 3-8: Function parameters

Type	Name	Dir.	Description
const GFLIB_ASIN_TAYLOR_T *const	pParam	in	Pointer to an array of Taylor coefficients. Using the function alias GFLIB_Asin, default coefficients are used.
tFrac32	s32In	in	Input argument is a 32-bit number that contains a value between $[-1, 1)$.

3.5.4 Return

The function returns $\frac{\arcsin(s32In)}{\pi}$ as a fixed point 32-bit number, normalized between $[-1, 1)$.

3.5.5 Description

The GFLIB_AsinANSIC function, denoting ANSI-C compatible source code implementation, can be called via the function alias GFLIB_Asin.

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

The computational algorithm uses the symmetry of the $arcsin(x)$ function around the point $(0, \pi/2)$, which allows for computing the function values in the interval $[0, 1)$ and to compute the function values in the interval $[-1, 0)$ by the simple formula:

$$y_{[-1,0)} = -y_{[1,0)}, \tag{3.17}$$

where:

- $y_{[-1,0)}$ is the $arcsin(x)$ function value in the interval $[-1, 0)$

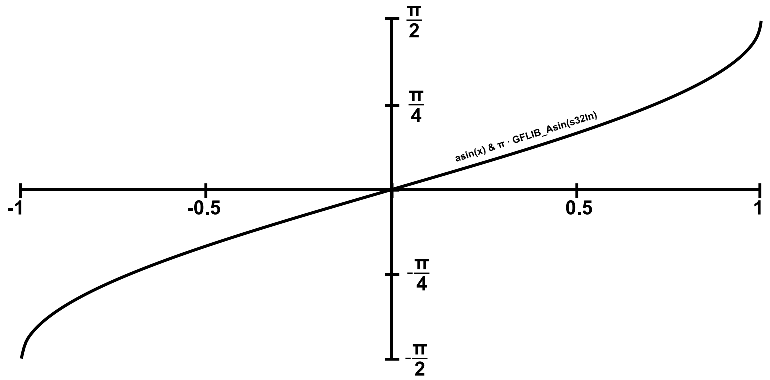


Figure 3-5: Course of the function GFLIB_Asin

- $y_{[1,0)}$ is the $arcsin(x)$ function value in the interval $[1, 0)$

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 $(0, 0.5]$:

$$\arcsin(\sqrt{1-x}) = \frac{\pi}{2} - \arcsin(\sqrt{x}) \tag{3.18}$$

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:

$$\text{s32Dump} = a_1 \cdot s32In^4 + a_2 \cdot s32In^3 + a_3 \cdot s32In^2 + a_4 \cdot s32In + a_5 \tag{3.19}$$

$$\arcsin(s32In) = \begin{cases} \text{-s32Dump} & \text{if} \quad -1 \leq \text{s32In} < 0 \\ \text{s32Dump} & \text{if} \quad 0 \leq \text{s32In} < 1 \end{cases} \tag{3.20}$$

The division of the $[0, 1)$ interval into two subintervals, with polynomial coefficients calculated for each subinterval, is noted in Table 3-9.

Table 3-9: Integer Polynomial coefficients for each interval

Interval	a_1	a_2	a_3	a_4	a_5
$\langle 0, \frac{1}{2})$	91918582	66340080	9729967	682829947	12751
$\langle \frac{1}{2}, 1)$	-52453538	-36708911	-15136243	-964576326	1073652175

Polynomial coefficients were obtained using the Matlab fitting function, where a polynomial of the 5th order fit each respective subinterval. 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))
```

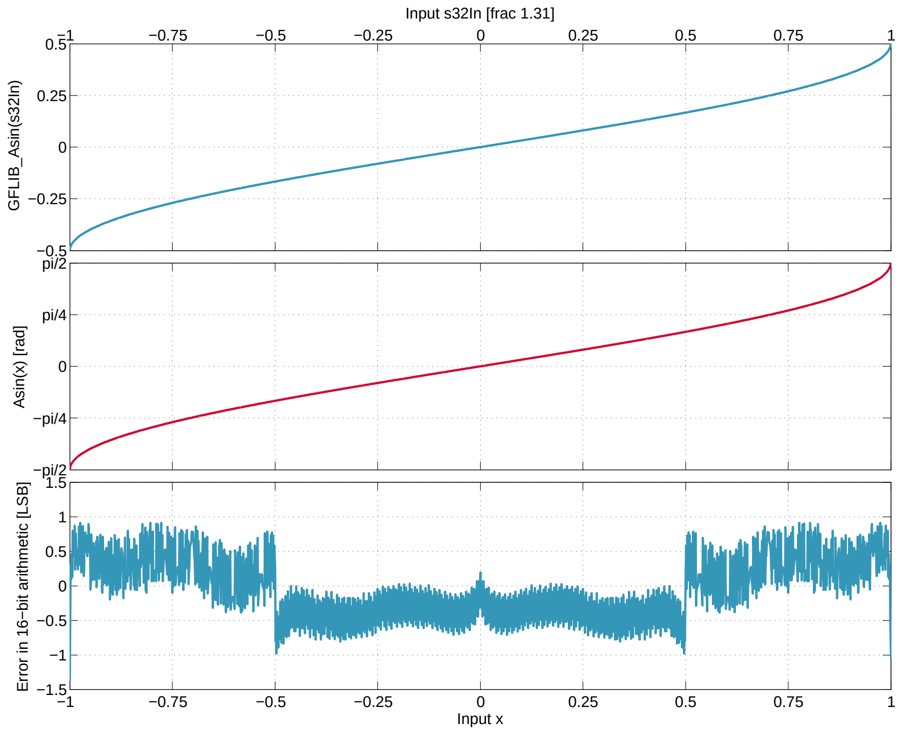


Figure 3-6: asin(x) vs. GFLIB_Asin(s32In)

Figure 3-6 depicts a floating point *arcsine* function generated from Matlab and the approximated value of the *arcsine* function obtained from GFLIB_Asin, as well as their difference. The course of calculation accuracy as a function of the input ratio can be observed from this figure. The achieved accuracy, with consideration to the 5th order piece-wise polynomial approximation and described fixed point scaling, is less than 1.3LSB on the upper 16 bits of the 32-bit result.

3.5.6 Note

The output angle is normalized into the range $[-0.5, 0.5)$. The polynomial coefficients are stored in a locally-defined structure, the call of which is masked by the function alias GFLIB_Asin. The polynomial coefficients can be calculated by the user, in which a case the full GFLIB_-AsinANSIC function call with a pointer to the newly defined coefficients shall be used instead of the function alias.

3.5.7 Reentrancy

The function is reentrant.

3.5.8 Code Example

```
#include "gflib.h"

tFrac32 s32Input;
tFrac32 s32Angle;

void main(void)
{
    // input s32Input = 1
    s32Input  = FRAC32(1);

    // output should be 0x3FFE A1CF = 0.49995 => pi/2
    s32Angle = GFLIB_Asin(s32Input);
}
```

3.5.9 Performance

Table 3-10: GFLIB_Asin function performance

Code size [bytes] IAR	180
Data size [bytes] IAR	40
Execution clock cycles max [clk] IAR	155
Execution clock cycles min [clk] IAR	68

3.6 GFLIB_Acos

3.6.1 Declaration

```
tFrac32 GFLIB_AcosANSIC(tFrac32 s32In, const GFLIB_ACOS_TAYLOR_T *const pParam)
```

3.6.2 Alias

```
#define GFLIB_Acos(x) \
        GFLIB_AcosANSIC((x), &gflibAcosCoef)
```

3.6.3 Arguments

Table 3-11: Function parameters

Type	Name	Dir.	Description
tFrac32	s32In	in	Input argument is a 32-bit number that contains a value between $[-1, 1)$.
const GFLIB_ACOS_TAYLOR_T *const	pParam	in	Pointer to an array of Taylor coefficients. The function alias GFLIB_Acos uses the default coefficients.

3.6.4 Return

The function returns $\frac{\arccos(s32In)}{\pi}$ as a fixed point 32-bit number normalized between $[0, 1)$.

3.6.5 Description

The GFLIB_AcosANSIC function, denoting ANSI-C compatible source code implementation, can be called via the function alias GFLIB_Acos.

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.

The computational algorithm uses the symmetry of the $arccos(x)$ function around the point $(0, \pi/2)$, which allows for computing the function values in the interval $[0, 1)$ and to compute the function values in the interval $[-1, 0)$ by the simple formula:

$$y_{[-1,0)} = \pi/2 + y_{[0,1)}, \tag{3.21}$$

where:

- $y_{[-1,0)}$ is the $arccos(x)$ function value in the interval $[-1, 0)$

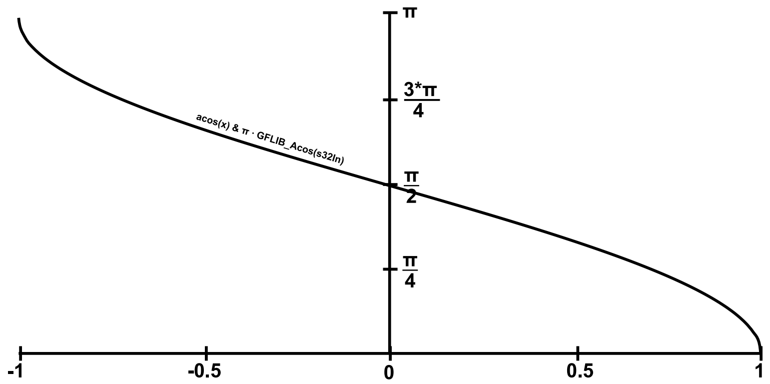


Figure 3-7: Course of the function GFLIB_Acos

- $y_{[0,1)}$ is the $arccos(x)$ function value in the interval $[0, 1)$

Additionally, because the $arccos(x)$ 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 $(0, 0.5]$:

$$\arccos(\sqrt{1-x}) = \frac{\pi}{2} - \arccos(\sqrt{x}) \tag{3.22}$$

In this way, the computation of the $arccos(x)$ 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 a polynomial approximation as follows:

$$\text{s32Dump} = a_1 \cdot s32In^4 + a_2 \cdot s32In^3 + a_3 \cdot s32In^2 + a_4 \cdot s32In + a_5 \tag{3.23}$$

$$\arccos(s32In) = \begin{cases} \text{-s32Dump} & \text{if} \quad -1 \leq \text{s32In} < 0 \\ \text{s32Dump} & \text{if} \quad 0 \leq \text{s32In} < 1 \end{cases} \tag{3.24}$$

The division of the $[0, 1)$ interval into two subintervals, with polynomial coefficients calculated for each subinterval, is noted in Table 3-12.

Table 3-12: Integer Polynomial coefficients for each interval

Interval	a_1	a_2	a_3	a_4	a_5
$\langle 0, \frac{1}{2})$	91918582	66340080	9729967	682829947	12751
$\langle \frac{1}{2}, 1)$	-52453538	-36708911	-15136243	-964576326	1073652175

The implementation of the GFLIB_Acos is almost the same as in the function GFLIB_Asin. However, the output of the GFLIB_Acos is corrected as follows:

$$\text{s32Dump} = FRAC32(0.5) - s32Dump \tag{3.25}$$

The polynomial coefficients were obtained using the Matlab fitting function, where a polynomial of 5th order was used for the fitting of each respective subinterval. Because the functions *arcsine* and *arccosine* are similar, the GFLIB_Acos function uses the same polynomial coefficients as the GFLIB_Asin function.

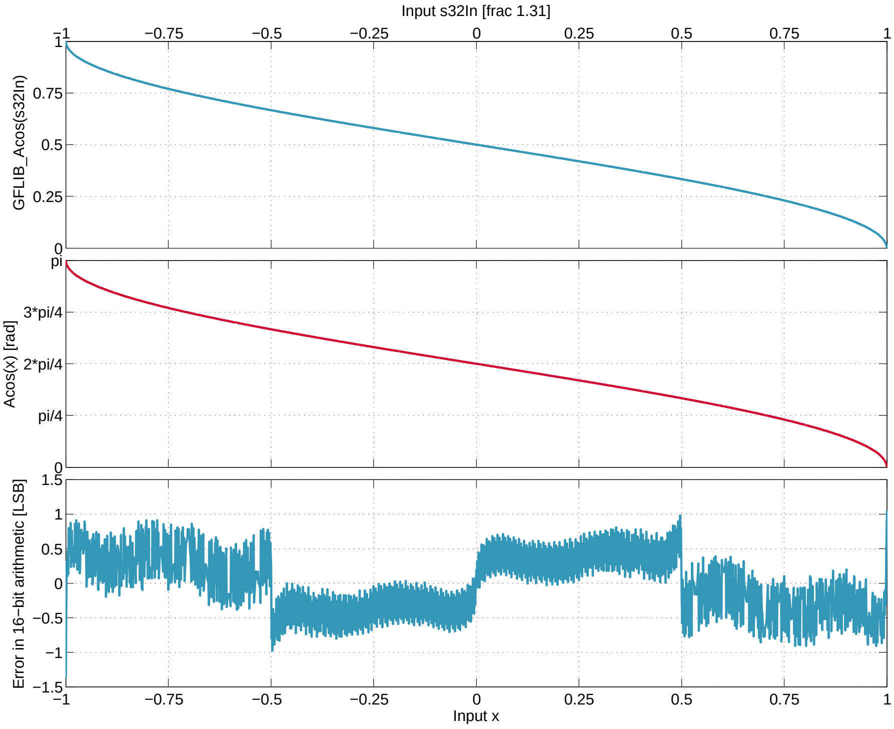


Figure 3-8: acos(x) versus GFLIB_Acos(s32In)

Figure 3-8 depicts the floating point *arccosine* function generated from Matlab, the approximated value of the *arccosine* function obtained from the GFLIB_Acos function, and their difference. The achieved accuracy is better than 1.3LSB on the upper 16 bits of the 32-bit result.

3.6.6 Note

The output angle is normalized into the range $[0, 1)$. The function uses data stored in the gflibAcosCoef structure, which is supplied to the GFLIB_Acos function as the argument, which is masked in the function alias GFLIB_Acos. The polynomial coefficients can be calculated by the user in which case the GFLIB_AcosANSIC function call with a pointer to the newly defined coefficients shall be used instead of the function alias.

3.6.7 Reentrancy

The function is reentrant.

3.6.8 Code Example

```
#include "gflib.h"

tFrac32 s32Input;
tFrac32 s32Angle;
```

```
void main(void)
{
    // input s32Input = 0
    s32Input   = FRAC32(0);

    // output should be 0x400031EF = 0.5 => pi/2
    s32Angle = GFLIB_Acos(s32Input);
}
```

3.6.9 Performance

Table 3-13: GFLIB_Acos function performance

Code size [bytes] IAR	188
Data size [bytes] IAR	0
Execution clock cycles max [clk] IAR	157
Execution clock cycles min [clk] IAR	70

3.7 GFLIB_Atan

3.7.1 Declaration

```
tFrac32 GFLIB_AtanANSIC(tFrac32 s32In, const GFLIB_ATAN_TAYLOR_T *const pParam)
```

3.7.2 Alias

```
#define GFLIB_Atan(x) \
        GFLIB_AtanANSIC((x), &gflibAtanCoef)
```

3.7.3 Arguments

Table 3-14: Function parameters

Type	Name	Dir.	Description
const GFLIB_ATAN_TAYLOR_T *const	pParam	in	Pointer to an array of Taylor coefficients. Using the function alias GFLIB_Atan, default coefficients are used.
tFrac32	s32In	in	Input argument is a 32-bit number between $[-1, 1)$.

3.7.4 Return

The function returns the atan of the input argument as a fixed point 32-bit number that contains the angle in radians between $\left[-\frac{\pi}{4}, \frac{\pi}{4}\right)$, normalized between $[-0.25, 0.25)$.

3.7.5 Description

The GFLIB_AtanANSIC function, denoting ANSI-C compatible source code implementation, can be called via the function alias GFLIB_Atan.

The GFLIB_Atan function provides a computational method for calculation of a standard trigonometric *arctangent* function $\arctan(x)$, using the piece-wise polynomial approximation. Function $\arctan(x)$ takes a ratio and returns the angle of two sides of a right-angled triangle. The ratio is the length of the side opposite the angle divided by the length of the side adjacent to the angle. The graph of $\arctan(x)$ is shown in Figure 3-9.

The GFLIB_Atan function is implemented with consideration to fixed point fractional arithmetic. As can be further seen from Figure 3-9, the arctangent values are identical for the input ranges $[-1, 0)$ and $[0, 1)$. Figure 3-9 also shows that the course of the $\arctan(x)$ function output for a ratio in interval 1. is identical, but with the opposite sign, to the output for a ratio in second interval. Therefore, the approximation of the *arctangent* function over the entire defined range of input ratios can be simplified to the approximation for a ratio in range $[0, 1)$, and then, depending on the input ratio, the result will be negated. In order to increase the accuracy of approximation without the need for a higher order polynomial, the interval $[0, 1)$ is further divided

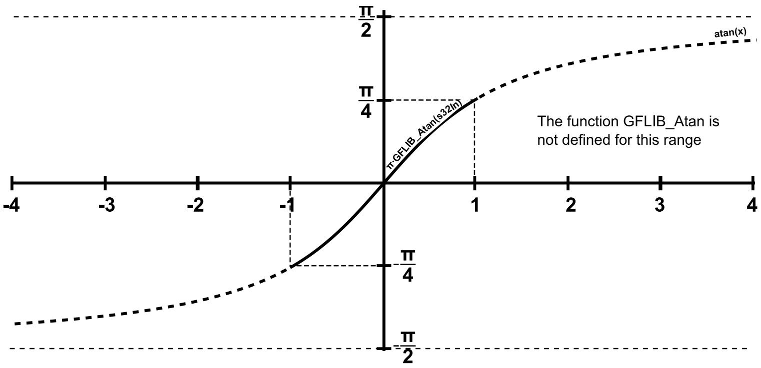


Figure 3-9: Course of the function GFLIB_Atan

into eight equally spaced sub intervals, and polynomial approximation is done for each interval respectively. Such a division results in eight sets of polynomial coefficients. Moreover, it allows for using a polynomial of only the 4th order to achieve an accuracy of less than 0.5LSB (on the upper 16 bits of 32-bit results) across the entire range of input ratios.

The GFLIB_Atan function uses fixed point fractional arithmetic, so to cast the fractional value of the output angle $[-0.25, 0.25)$ into the correct range $\left[-\frac{\pi}{4}, \frac{\pi}{4}\right)$, the fixed point output angle can be multiplied by π for an angle in radians. Then, the fixed point fractional implementation of the approximation polynomial, used for calculation of each sub sector, is defined as follows:

$$\text{s32Dump} = a_1 \cdot s32In^3 + a_2 \cdot s32In^2 + a_3 \cdot s32In + a_4 \tag{3.26}$$

$$\arctan(s32In) = \begin{cases} \text{s32Dump} & \text{if} \quad 0 \leq \text{s32In} < 1 \\ \text{-s32Dump} & \text{if} \quad -1 \leq \text{s32In} < 0 \end{cases} \tag{3.27}$$

The division of the $[0, 1)$ interval into eight subintervals, with polynomial coefficients calculated for each subinterval, is noted in Table 3-15. Polynomial coefficients were obtained using the Matlab fitting function, where a polynomial of the 4th order was used for the fitting of each respective subinterval.

Figure 3-10 depicts a floating point *arctangent* function generated from Matlab and the approximated value of the *arctangent* function obtained from GFLIB_Atan, and their difference. The course of calculation accuracy as a function of the input value can be observed from this figure. The achieved accuracy, with consideration to the 4th order piece-wise polynomial approximation and described fixed point scaling, is less than 0.5LSB on the upper 16 bits of the 32-bit result.

3.7.6 Note

The output angle is normalized into the range $[-0.25, 0.25)$. The polynomial coefficients are stored in locally defined structure, the call of which is masked by the function alias GFLIB_-Atan. The polynomial coefficients can be calculated by the user, in which case the full GFLIB_-AtanANSIC function call with a pointer to the newly defined coefficients shall be used instead of the function alias.

Table 3-15: Integer Polynomial coefficients for each interval:

Interval	a_1	a_2	a_3	a_4
$\langle 0, \frac{1}{8})$	-53248	-165888	42557440	42668032
$\langle \frac{1}{8}, \frac{2}{8})$	-45056	-464896	41271296	126697472
$\langle \frac{2}{8}, \frac{3}{8})$	-30720	-690176	38922240	207040512
$\langle \frac{3}{8}, \frac{4}{8})$	-14336	-821248	35858432	281909248
$\langle \frac{4}{8}, \frac{5}{8})$	-2048	-866304	32454656	350251008
$\langle \frac{5}{8}, \frac{6}{8})$	8192	-845824	29009920	411703296
$\langle \frac{6}{8}, \frac{7}{8})$	12288	-786432	25735168	466407424
$\langle \frac{7}{8}, 1)$	14336	-708608	22738944	514828288

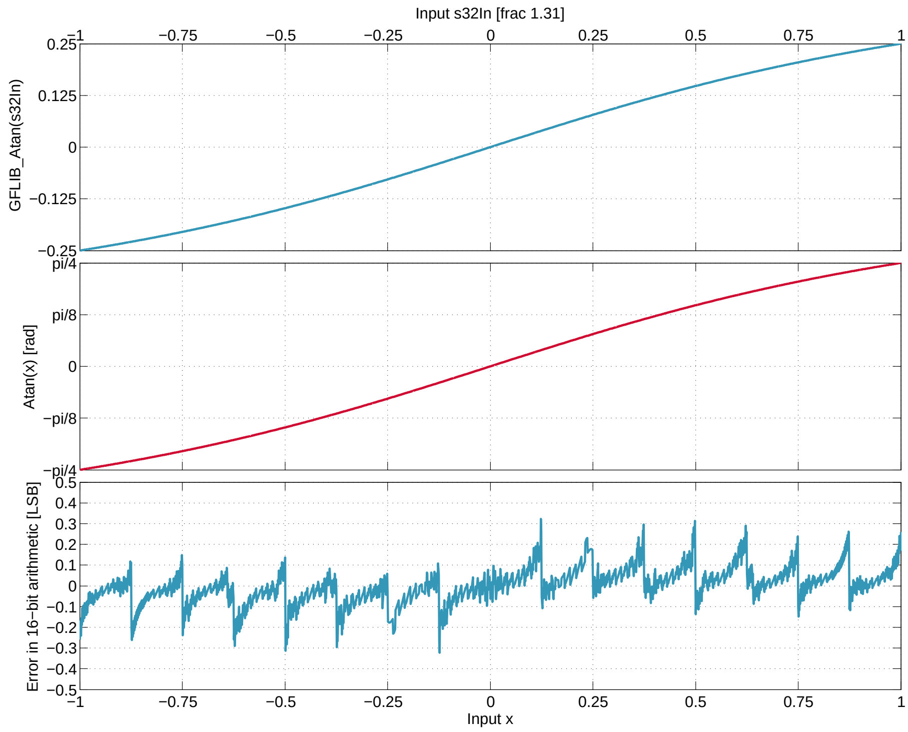


Figure 3-10: atan(x) vs. GFLIB_Atan(s32In)

3.7.7 Reentrancy

The function is reentrant.

3.7.8 Code Example

```
#include "gflib.h"

tFrac32 s32Input;
tFrac32 s32Angle;

void main(void)
{
    // input ratio = 0x7FFFFFFF => 1
    s32Input  = FRAC32(1);

    // output angle should be 0x1FFFBD7F = 0.249999 => pi/4
    s32Angle = GFLIB_Atan(s32Input);
}
```

3.7.9 Performance

Table 3-16: GFLIB_Atan function performance

Code size [bytes] IAR	180
Data size [bytes] IAR	40
Execution clock cycles max [clk] IAR	55
Execution clock cycles min [clk] IAR	52

3.8 GFLIB_AtanYX

3.8.1 Declaration

```
tS32 GFLIB_AtanYXANSIC(tS32 s32InY, tS32 s32InX)
```

3.8.2 Alias

```
#define GFLIB_AtanYX(y, x) \
        GFLIB_AtanYXANSIC((y), (x))
```

3.8.3 Arguments

Table 3-17: Function parameters

Type	Name	Dir.	Description
tS32	s32InY	in	The ordinate of the input vector (y coordinate).
tS32	s32InX	in	The abscissa of the input vector (x coordinate).

3.8.4 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.

3.8.5 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, `s32InY`, is the ordinate (the y coordinate) and the second parameter, `s32InX`, is the abscissa (the x coordinate). Both the input parameters are assumed to be in the fractional range of $[-1,1)$. The computed angle is limited by the fractional range of $[-1,1)$, which corresponds to the real range of $[-\pi,\pi)$. The counter-clockwise direction is assumed to be positive and thus a positive angle will be computed if the provided ordinate (s32InY) 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 arc tangent 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 function, the GFLIB_AtanYX function correctly places the calculated angle within the whole fractional range of $[-1, 1)$, which corresponds to the real angle range of $[-\pi, \pi)$.

3.8.6 Note

The function calls the GFLIB_Atan function. The computed value is within the range of $[-1, 1)$.

3.8.7 Reentrancy

The function is reentrant.

3.8.8 Code Example

```
#include "gflib.h"

tFrac32 s32InY;
tFrac32 s32InX;
tFrac32 s32Ang;

void main(void)
{
    s32InY = FRAC32(0.5);
    s32InX = FRAC32(0.5);

    // Angle 45 deg = PI/4 rad = 0.25 = 0x20000000
    // output should be close to 0x20001000
    s32Ang = GFLIB_AtanYX(s32InY, s32InX);

    return;
}
```

3.8.9 Performance

Table 3-18: GFLIB_AtanYX function performance

Code size [bytes] IAR	196
Data size [bytes] IAR	0
Execution clock cycles max [clk] IAR	177
Execution clock cycles min [clk] IAR	98

3.9 GFLIB_AtanYXShifted

3.9.1 Declaration

```
tFrac32 GFLIB_AtanYXShiftedANSIC(tFrac32 s32InY, tFrac32 s32InX, const
GFLIB_ATANYXSHIFTED_T *const pParam)
```

3.9.2 Alias

```
#define GFLIB_AtanYXShifted(y, x, p) \
        GFLIB_AtanYXShiftedANSIC((y), (x), (p))
```

3.9.3 Arguments

Table 3-19: Function parameters

Type	Name	Dir.	Description
tFrac32	s32InY	in	The value of the first signal, assumed to be $sin(\theta)$.
tFrac32	s32InX	in	The value of the second signal, assumed to be $sin(\theta + \Delta\theta)$.
const GFLIB_ATANYXSHIFTED_T *const	pParam	in	The parameters for the function.

3.9.4 Return

The function returns the angle of two sine waves shifted in phase one to each other.

3.9.5 Description

The function calculates the angle of two sinusoidal signals, one shifted in phase to the other. The phase shift between sinusoidal signals does not have to be $\pi/2$ and can be any value. It is assumed that the arguments of the function are as follows:

$$\begin{aligned} y &= sin(\theta) \\ x &= sin(\theta + \Delta\theta) \end{aligned} \tag{3.28}$$

where:

- x, y are respectively, the `s32InX`, and `s32InY` 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. The GFLIB_AtanYXShifted 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, need to be computed by means of the provided Matlab function. If $\Delta\theta = \pi/2$ or $\Delta\theta = -\pi/2$, then the function is similar to the GFLIB_AtanYX function, however, the GFLIB_AtanYX function in this case is more effective in execution time and accuracy. In order to use the function, the following 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 at the end of GFLIB_AtanYXShifted)

The function uses the following algorithm for computing the angle:

$$\begin{aligned} b &= \frac{S}{2cos\frac{\Delta\theta}{2}}(y+x) \\ a &= \frac{S}{2sin\frac{\Delta\theta}{2}}(x-y) \\ \theta &= \text{AtanYX}(b,a) - (\Delta\theta/2 - \theta_{Offset}) \end{aligned} \tag{3.29}$$

where:

- x, y are respectively, the `s32InX`, and `s32InY`
- θ is the angle to be computed by the function
- $\Delta\theta$ is the phase difference between the x, y signals
- 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:

$$\tan(\theta + \Delta\theta) = \frac{(y+x)cos\frac{\Delta\theta}{2}}{(x-y)sin\frac{\Delta\theta}{2}} \tag{3.30}$$

For the purposes of fractional arithmetic, the algorithm is implemented such that additional values are used as shown in the following equation:

$$
\begin{aligned}
&\frac{S}{2cos\frac{\Delta\theta}{2}} = C_y = K_y 2^{N_y} \\
&\frac{S}{2sin\frac{\Delta\theta}{2}} = C_x = K_x 2^{N_x} \\
&\theta_{adj} = \Delta\theta/2 - \theta_{offset}
\end{aligned}
\tag{3.31}
$$

where:

- C_y, C_x are the algorithm coefficients for y and x signals
- K_y is the multiplication coefficient of the y signal, represented by the parameters structure member `pParam->s32Ky`
- K_x is the multiplication coefficient of the x signal, represented by the parameters structure member `pParam->s32Kx`
- N_y is the scaling coefficient of the y signal, represented the by parameters structure member `pParam->s32Ny`
- N_x is the 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->s32ThetaAdj`

The multiplication and scaling coefficients, as well as the adjusting angle, shall be defined in a parameters structure provided as the function input parameter. The function uses 16-bit fractional arithmetic for multiplication, therefore the 16 least significant bits of the input values and the K_y, K_x multiplication coefficients are ignored. 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. At some values of the phase shift, and particularly at phase shift approaching -180, 0, or 180 degrees, the algorithm may become numerically unstable. This will cause any error, whether contributed by input signal imperfections or through finite precision arithmetic, to be magnified significantly. Therefore, care should be taken to avoid error where possible. The detailed error analysis of the algorithm is complicated, 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 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 also be taken into account.

3.9.6 Note

The function calls the GFLIB_AtanYX 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.

3.9.7 Reentrancy

The function is reentrant.

3.9.8 Code Example

```
#include "gflib.h"

tFrac32 s32InY;
tFrac32 s32InX;
tFrac32 s32Ang;
GFLIB_ATANYXSHIFTED_T 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 = 0.137027777777778

    Param.s32Ky = FRAC32(0.60789036201452440);
    Param.s32Kx = FRAC32(0.87905201358520957);
    Param.s32Ny = 0;
    Param.s32Nx = 0;
    Param.s32ThetaAdj = FRAC32(0.137027777777778);

    // theta = 15 deg
    // Y = sin(theta) = 0.2588190
    // X = sin(theta + dtheta) = 0.9951074
    s32InY = FRAC32(0.2588190);
    s32InX = FRAC32(0.9951074);
    s32Ang = GFLIB_AtanYXShifted(s32InY, s32InX, &Param);

    // s32Ang contains 0x11c6cdfc, the theoretical value is 0x11c71c72

    return;
}
```

3.9.9 Performance

Table 3-20: GFLIB_AtanYXShifted function performance

Code size [bytes] IAR	196
Data size [bytes] IAR	0
Execution clock cycles max [clk] IAR	252
Execution clock cycles min [clk] IAR	182

3.10 GFLIB_Sqrt

3.10.1 Declaration

```
tFrac32 GFLIB_SqrtANSIC(tFrac32 s32In)
```

3.10.2 Alias

```
#define GFLIB_Sqrt(s32In) \
        GFLIB_SqrtANSIC(s32In)
```

3.10.3 Arguments

Table 3-21: Function parameters

Type	Name	Dir.	Description
tFrac32	s32In	in	The input value.

3.10.4 Return

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

3.10.5 Description

The GFLIB_Sqrt function computes the square root of the input value. The computations are made by a simple iterative testing of each bit starting with the most significant one. In total, 15 iterations are made, each performing the following steps:

1. Add a single testing bit to the tentative square root value.
2. Square the tentative square root value and test whether it is greater or lower than the input value.
3. If greater, clear the bit in the tentative square root value.
4. Shift the single testing bit right.

Figure 3-11 depicts a floating point *sqrt(x)* function generated from Matlab, the calculated value of the *sqrt* function obtained from GFLIB_Sqrt, and their difference. The course of calculation accuracy as a function of the input value can be observed from this figure. The computed value is equal to or is 1LSB (on the upper 16 bits of the 32-bit result; 1LSB $\equiv 2^{-15}$) less than the true square root value. In order to obtain a value with a 0.5LSB (16bit) accuracy, additional iterations are required.

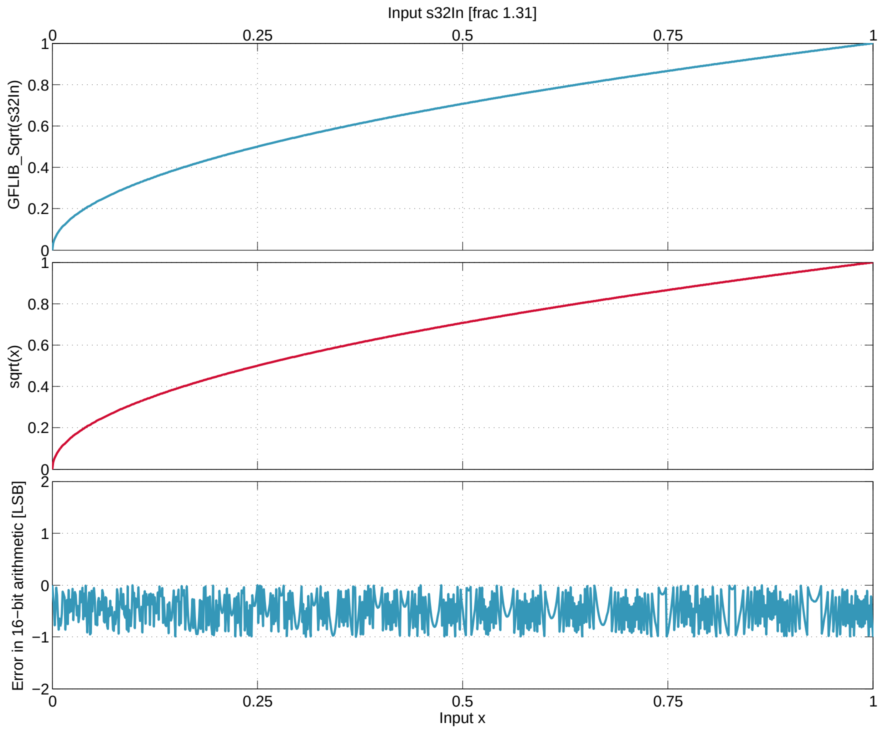


Figure 3-11: real sqrt(x) versus GFLIB_Sqrt(s32In)

3.10.6 Note

The input value is limited to the range $[0, 1)$, the computed value is undefined if not within this range.

3.10.7 Reentrancy

The function is reentrant.

3.10.8 Code Example

```
#include "gflib.h"

tFrac32 s32In;
tFrac32 s32Out;

void main(void)
{
    // s32In = 0.5 = 0x40000000
    s32In  = FRAC32(0.5);

    // s32Out should be 0x5a820000
    s32Out = GFLIB_Sqrt(s32In);

    return;
```

```
}
```

3.10.9 Performance

Table 3-22: GFLIB_Sqrt function performance

Code size [bytes] IAR	234
Data size [bytes] IAR	0
Execution clock cycles max [clk] IAR	88
Execution clock cycles min [clk] IAR	88

3.11 GFLIB_Sign

3.11.1 Declaration

```
tFrac32 GFLIB_SignANSIC(tFrac32 s32In)
```

3.11.2 Alias

```
#define GFLIB_Sign(x) \
        GFLIB_SignANSIC(x)
```

3.11.3 Arguments

Table 3-23: Function parameters

Type	Name	Dir.	Description
tFrac32	s32In	in	Input signal fraction

3.11.4 Return

The GFLIB_Sign function returns the sign of the argument.

3.11.5 Description

The function GFLIB_Sign calculates the sign of the input argument. If the the input value is negative, then the return value will be set to -1 (0x80000000 hex). If the input value is zero, then the return value will be set to 0 (0x0 hex). If the input value is greater than zero, the return value will be almost 1 (0x7fffffff hex). In mathematical terms:

$$y_{out} = \begin{cases} 1 & \text{if } x_{in} > 0 \\ 0 & \text{if } x_{in} = 0 \\ -1 & \text{if } x_{in} < 0 \end{cases} \tag{3.32}$$

where:

- y_{out} is the return value
- x_{in} is the input value provided as `s32In` parameter

3.11.6 Note

The function can be used as an argument of the conditional statement.

3.11.7 Reentrancy

The function is reentrant.

3.11.8 Code Example

```
#include "gflib.h"

tFrac32 s32In;
tFrac32 s32Out;

void main(void)
{
    // input value = 0.5
    // output should be 0x7FFFFFFF
    s32In  = FRAC32(0.5);
    s32Out = GFLIB_Sign(s32In);

    // input value = 0.0
    // output should be 0x00000000
    s32In  = FRAC32(0.0);
    s32Out = GFLIB_Sign(s32In);

    // input value = -0.5
    // output should be 0x80000000
    s32In  = FRAC32(-0.5);
    s32Out = GFLIB_Sign(s32In);

    return;
}
```

3.11.9 Performance

Table 3-24: GFLIB_Sign function performance

Code size [bytes] IAR	22
Data size [bytes] IAR	0
Execution clock cycles max [clk] IAR	14
Execution clock cycles min [clk] IAR	11

3.12 GFLIB_Lut1D

3.12.1 Declaration

```
tFrac32 GFLIB_Lut1DANSIC(tFrac32 s32In, const GFLIB_LUT1D_T *const pParam)
```

3.12.2 Alias

```
#define GFLIB_Lut1D(x, pParam) \
        GFLIB_Lut1DANSIC((x), (pParam))
```

3.12.3 Arguments

Table 3-25: Function parameters

Type	Name	Dir.	Description
tFrac32	s32In	in	The abscissa for which 1D interpolation is performed.
const GFLIB_LUT1D_T *const	pParam	in	Pointer to the parameters structure.

3.12.4 Return

The GFLIB_Lut1D returns the interpolated value from the table with 16-bit accuracy.

3.12.5 Description

The GFLIB_Lut1D function performs a 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:

$$y = y_1 + \frac{y_2 - y_1}{x_2 - x_1} \cdot (x - x_1) \tag{3.33}$$

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 `s32In` argument

The interpolating intervals are defined in the table (GFLIB_Lut1D parameter) provided by the `ps32Table` member of the parameters structure. The table (GFLIB_Lut1D parameter) 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 `ps32Table` 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 3-26: GFLIB_Lut1D example table

ordinate (y)	interpolating index	abscissa (x)
-0.5	-1	$-1 * (2^{-1})$
`ps32Table` $\rightarrow 0.0$	0	$0 * (2^{-1})$
0.25	1	$1 * (2^{-1})$
0.5	N/A	$2 * (2^{-1})$

Table 3-26 contains four interpolating points. The interpolating interval length in this example is equal to 2^{-1}. The `ps32Table` 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 `ps32Table` 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 and does not have to have zero in its range. However, a special algorithm is used to make the computation efficient, under some additional assumptions, as provided below:

- the values of the interpolated function are 32 bits long
- the length of each interpolating interval is equal to 2^{-n}, where n is an integer in the range of 1, 2, ... 29
- the provided abscissa for interpolation is 32 bits long

The algorithm performs the following steps:

1. Compute the index representing the interval, in which the linear interpolation will be performed:

$$I = x >> s_{Interval} \tag{3.34}$$

   where I is the interval index and the $s_{Interval}$ the shift amount provided in the parameters structure as the member `s32ShamIntvl`. The operator $>>$ represents the binary arithmetic right shift.

2. Compute the abscissa offset within an interpolating interval:

$$\Delta x = x << s_{Offset} \ \texttt{\&}\ \texttt{0x7fffffff} \tag{3.35}$$

   where Δx is the abscissa offset within an interval and the s_{Offset} is the shift amount provided in the parameters structure. The operators $<<$ and $\&$ represent, respectively, the binary left shift and the bitwise logical conjunction. It should be noted that the computation represents the extraction of some least significant bits of the x with the sign bit cleared.

3. Compute the interpolated value by the linear interpolation between the 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.

$$\begin{aligned} y_1 &= (\texttt{32-bit data at address pTable}) + 4 \cdot I \\ y_2 &= (\texttt{32-bit data at address pTable}) + 4 \cdot (I+1) \\ y &= y_1 + (y_2 - y_1) \cdot \Delta x \end{aligned} \tag{3.36}$$

where y, y_1 ,and y_2 are, respectively, the interpolated value, the ordinate at the start of the interpolating interval, and the ordinate at the end of the interpolating interval. The `pTable` is the address provided in the parameters structure `pParam->as32Table`. It should be noted that due to assumption of equidistant data points, division by the interval length is avoided.

Computations are performed with a 16-bit accuracy. Specifically, the 16 least significant bits are ignored in all multiplications. The shift amounts shall be provided in the parameters structure (`pParam->s32ShamOffset`, `pParam->s32ShamIntvl`). The address of the table with the data, the `pTable`, shall be defined by the parameter structure member `pParam->ps32Table`. The shift amounts, the $s_{Interval}$ and s_{Offset}, can be computed with the following formulas:

$$\begin{aligned} s_{Interval} &= 31 - |n| \\ s_{Offset} &= |n| \end{aligned} \tag{3.37}$$

where n is the integer defining the length of the interpolating interval in the range of -1, -2, ... -29. The computation of the abscissa offset and the interval index can be viewed also in the following way. The input abscissa value can be divided into two parts. The interval index is composed of the first n most significant bits of the 32-bit word, after the bit sign. The rest of the bits form the abscissa offset within the interpolating interval. This simple way to calculate the interpolating interval index and the abscissa offset is the consequence of assuming that all interpolating interval lengths equal 2^{-n}. 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->ps32Table`). However, if it is negative, then the ordinate values are read from the memory, which is located behind the `pParam->ps32Table` pointer.

3.12.6 Caution

The function does not check whether the input abscissa value is within the range allowed by the interpolating data table (`pParam->ps32Table`). If the computed interval index points to data outside the provided data table, then the interpolation will be computed with invalid data.

3.12.7 Note

The function performs the linear interpolation with 16-bit accuracy.

3.12.8 Reentrancy

The function is reentrant.

3.12.9 Code Example

```
#include "gflib.h"

GFLIB_LUT1D_T Param;

// The interpolating interval length is 2^-1
// The interpolating table pointer is defined to point to
// the element 1 (pTable1[1]).  The interpolating index can be
// then a value of -1, 0, 1.
tFrac32 pTable1[4] = {
    1073741824,     // interpolating index = -1, abscissa = -2^-1
    1342177280,     // interpolating index =  0, abscissa =  0.0
    1610612736,     // interpolating index =  1, abscissa =  2^-1
    1879048192      // interpolating index N/A,  abscissa =  1.0
                    // N/A = Not Applicable
};

volatile tFrac32 x;
volatile tFrac32 y;

int
main(int argc, char *argv[])
{
    Param.ps32Table = &(pTable1[1]);
    Param.s32ShamOffset = 1;
    Param.s32ShamIntvl = 31 - 1;

    x = 0;
    y = GFLIB_Lut1D(x, &Param);     // y = 0x50000000

    x = 536870912;
    y = GFLIB_Lut1D(x, &Param);     // y = 0x58000000

    x = 1610612736;
    y = GFLIB_Lut1D(x, &Param);     // y = 0x68000000

    x = 3758096384;
    y = GFLIB_Lut1D(x, &Param);     // y = 0x48000000

    return 0;
}
```

3.12.10 Performance

Table 3-27: GFLIB_Lut1D function performance

Code size [bytes] IAR	62
Data size [bytes] IAR	0
Execution clock cycles max [clk] IAR	30
Execution clock cycles min [clk] IAR	30

3.13 GFLIB_Hyst

3.13.1 Declaration

```
tFrac32 GFLIB_HystANSIC(tFrac32 s32In, GFLIB_HYST_T *pParam)
```

3.13.2 Alias

```
#define GFLIB_Hyst(s32In, pParam) \
        GFLIB_HystANSIC(s32In, pParam)
```

3.13.3 Arguments

Table 3-28: Function parameters

Type	Name	Dir.	Description
tFrac32	s32In	in	Input signal in the form of a 32-bit fixed point fractional number, normalized between $[-1, 1)$.
GFLIB_HYST_T *	pParam	in	Pointer to the structure with parameters and states of the hysteresis function.

3.13.4 Return

The function returns the value of hysteresis output, which is equal to either *s32OutValOn* or *s32OutValOff* depending on the value of 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 $[-1, 1)$.

3.13.5 Description

The GFLIB_HystANSIC function, denoting ANSI-C compatible source code implementation, can be called via the function alias GFLIB_Hyst.

The GFLIB_Hyst function provides a computational method for calculation of a hysteresis (relay) function. The function switches the output between the two predefined values stored in the *s32OutValOn* and *s32OutValOff* members of structure GFLIB_HYST_T. When the value of the input is higher than the upper threshold *s32OutValOn*, then the output value is equal to *s32OutValOn*. When the input value is lower than the lower threshold *s32OutValOff*, then the output value is equal to *s32OutValOff*. When the input value is between the two threshold values then the output retains its value.

$$s32OutState(k) = \begin{cases} s32OutValOn & \text{if} \quad s32In \geq s32HystOn \\ s32OutValOff & \text{if} \quad s32In \leq s32HystOff \\ s32OutState(k-1) & \text{otherwise} \end{cases} \tag{3.38}$$

A graphical description of GFLIB_Hyst functionality is shown in Figure 3-12.

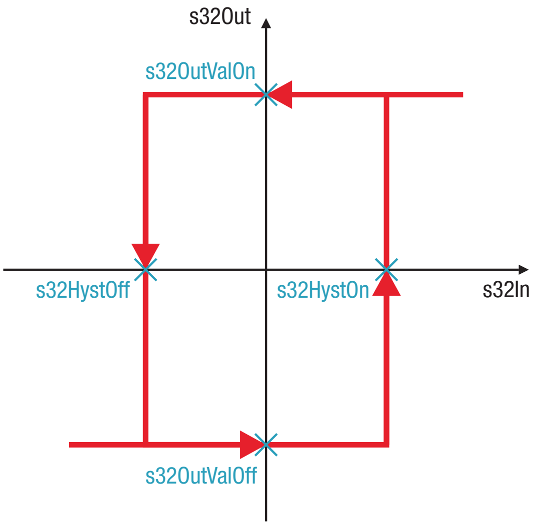


Figure 3-12: Hysteresis function

3.13.6 Caution

For correct functionality, s32HystOn must be greater than s32HystOff.

3.13.7 Note

All parameters and states used by the function can be reset during declaration using the GFLIB_HYST_DEFAULT macro.

3.13.8 Reentrancy

The function is reentrant.

3.13.9 Code Example

```
#include "gflib.h"

tFrac32 s32In;
tFrac32 s32Out;

// Definition of one hysteresis instance
GFLIB_HYST_T trMyHyst = GFLIB_HYST_DEFAULT;

void main(void)
{
    // Setting parameters for hysteresis
    trMyHyst.s32HystOn      = FRAC32(0.3);
    trMyHyst.s32HystOff     = FRAC32(-0.3);
    trMyHyst.s32OutValOn    = FRAC32(0.5);
    trMyHyst.s32OutValOff   = FRAC32(-0.5);
```

```
    // input value = 0.5
    s32In   = FRAC32(0.5);

    // output should be 0x40000000
    s32Out  = GFLIB_Hyst(s32In,&trMyHyst);
}
```

3.13.10 Performance

Table 3-29: GFLIB_Hyst function performance

Code size [bytes] IAR	24
Data size [bytes] IAR	0
Execution clock cycles max [clk] IAR	13
Execution clock cycles min [clk] IAR	13

3.14 GFLIB_Ramp

3.14.1 Declaration

```
tFrac32 GFLIB_RampANSIC(tFrac32 s32In, GFLIB_RAMP_T *pParam)
```

3.14.2 Alias

```
#define GFLIB_Ramp(in, pParam) \
        GFLIB_RampANSIC(in, pParam)
```

3.14.3 Arguments

Table 3-30: Function parameters

Type	Name	Dir.	Description
GFLIB_RAMP_T *	pParam	in/out	Pointer to the ramp parameters structure
tFrac32	s32In	in	Input argument representing the desired output value.

3.14.4 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.

3.14.5 Description

The GFLIB_RampANSIC function, denoting ANSI-C compatible source code implementation, can be called via the function alias GFLIB_Ramp.

If the desired (input) value is greater than the ramp output value, the function adds the `s32RampUp` value to the actual output value. The output cannot be greater than the desired value.

If the desired value is lower than the actual value, the function subtracts the `s32RampDown` value from the actual value. The output cannot be lower than the desired value.

Functionality of the implemented ramp algorithm can be explained with use of Figure 3-13.

3.14.6 Note

All parameters and states used by the function can be reset during declaration using the GFLIB_RAMP_DEFAULT macro.

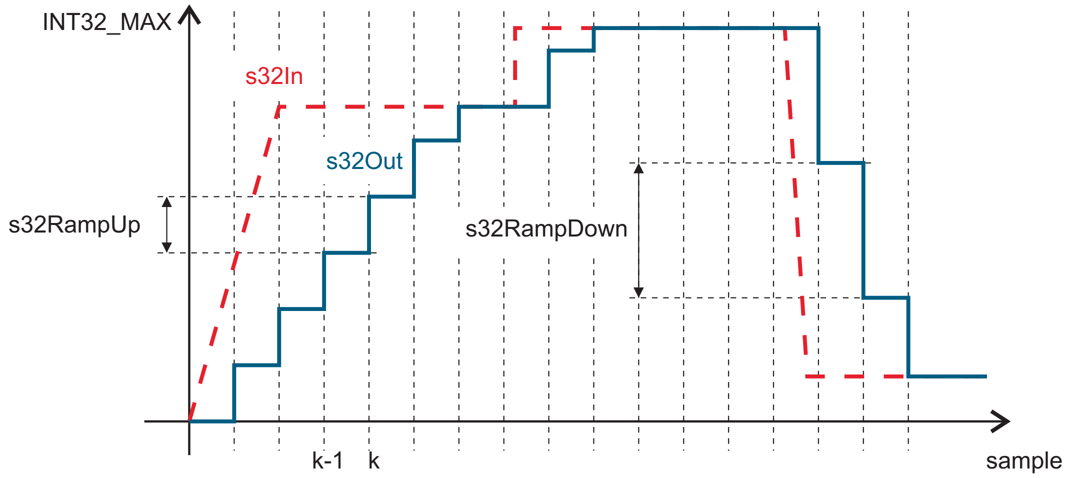


Figure 3-13: GFLIB_Ramp functionality

3.14.7 Reentrancy

The function is reentrant.

3.14.8 Code Example

```
#include "gflib.h"

tFrac32 s32In;
tFrac32 s32Out;

// Definition of one ramp instance
GFLIB_RAMP_T trMyRamp = GFLIB_RAMP_DEFAULT;

void main(void)
{
    // Setting parameters for hysteresis
    trMyRamp.s32RampUp      = 214748364;
    trMyRamp.s32RampDown    = 71582788;

    // input value = 0.5
    s32In   = FRAC32(0.5);

    // output should be 0xCCCCCCC
    s32Out  = GFLIB_Ramp(s32In,&trMyRamp);
}
```

3.14.9 Performance

Table 3-31: GFLIB_Ramp function performance

Code size [bytes] IAR	48
Data size [bytes] IAR	0
Execution clock cycles max [clk] IAR	27
Execution clock cycles min [clk] IAR	23

3.15 GFLIB_Limit

3.15.1 Declaration

```
tFrac32 GFLIB_LimitANSIC(tFrac32 s32In, const GFLIB_LIMIT_T *const pParam)
```

3.15.2 Alias

```
#define GFLIB_Limit(s32In, pParam) \
        GFLIB_LimitANSIC((s32In), (pParam))
```

3.15.3 Arguments

Table 3-32: Function parameters

Type	Name	Dir.	Description
tFrac32	s32In	in	Input value.
const GFLIB_LIMIT_T *const	pParam	in	Pointer to the limits structure.

3.15.4 Return

GFLIB_Limit returns the input value, or the upper or lower limit if the input value is beyond these limits.

3.15.5 Description

The 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. Similarly, 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`.

3.15.6 Note

The function assumes that the upper limit is greater than the lower limit. Otherwise, the function returns an undefined value.

3.15.7 Reentrancy

The function is reentrant.

3.15.8 Code Example

```
#include "gflib.h"

tFrac32 x;
tFrac32 y;
GFLIB_LIMIT_T LimitParam;

void main(void)
{
    // 0.5 = 0x40000000
    LimitParam.s32LowerLimit = FRAC32(-0.5);

    // -0.5 = 0xc0000000
    LimitParam.s32UpperLimit = FRAC32(0.5);

    // x = 0.75 = 0x60000000
    x   = FRAC32(0.75);
    // y should be 0x40000000
    y = GFLIB_Limit(x, &LimitParam);

    // x = -0.75 = 0xa0000000
    x   = FRAC32(-0.75);
    // y should be 0xc0000000
    y = GFLIB_Limit(x, &LimitParam);

    // x = 0.25 = 0x20000000
    x   = FRAC32(0.25);
    // y should be 0x20000000
    y = GFLIB_Limit(x, &LimitParam);

    return;
}
```

3.15.9 Performance

Table 3-33: GFLIB_Limit function performance

Code size [bytes] IAR	20
Data size [bytes] IAR	0
Execution clock cycles max [clk] IAR	11
Execution clock cycles min [clk] IAR	11

3.16 GFLIB_LowerLimit

3.16.1 Declaration

```
tFrac32 GFLIB_LowerLimitANSIC(tFrac32 s32In, const GFLIB_LOWERLIMIT_T *const
pParam)
```

3.16.2 Alias

```
#define GFLIB_LowerLimit(s32In, pParam) \
        GFLIB_LowerLimitANSIC((s32In), (pParam))
```

3.16.3 Arguments

Table 3-34: Function parameters

Type	Name	Dir.	Description
tFrac32	s32In	in	Input value.
const GFLIB_LOWERLIMIT_T *const	pParam	in	Pointer to the limits structure.

3.16.4 Return

GFLIB_LowerLimit returns the input value, or the lower limit if the input value is lower than the lower limit.

3.16.5 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 can be found in the limits structure, supplied to the function as a pointer `pParam`.

3.16.6 Note

The function can be used as an argument of the conditional statement.

3.16.7 Reentrancy

The function is reentrant.

3.16.8 Code Example

```
#include "gflib.h"

tFrac32 x;
tFrac32 y;
GFLIB_LOWERLIMIT_T LowerLimitParam;

void main(void)
{
    // 0.5 = 0x40000000
    LowerLimitParam.s32LowerLimit = FRAC32(0.5);

    // x = 0.75 = 0x60000000
    x  = FRAC32(0.75);
    // y should be 0x60000000
    y = GFLIB_LowerLimit(x, &LowerLimitParam);

    // x = -0.75 = 0xa0000000
    x  = FRAC32(-0.75);
    // y should be 0x40000000
    y = GFLIB_LowerLimit(x, &LowerLimitParam);

    // x = 0.25 = 0x20000000
    x  = FRAC32(0.25);
    // y should be 0x40000000
    y = GFLIB_LowerLimit(x, &LowerLimitParam);

    return;
}
```

3.16.9 Performance

Table 3-35: GFLIB_LowerLimit function performance

Code size [bytes] IAR	10
Data size [bytes] IAR	0
Execution clock cycles max [clk] IAR	11
Execution clock cycles min [clk] IAR	11

3.17 GFLIB_UpperLimit

3.17.1 Declaration

```
tFrac32 GFLIB_UpperLimitANSIC(tFrac32 s32In, const GFLIB_UPPERLIMIT_T *const
pParam)
```

3.17.2 Alias

```
#define GFLIB_UpperLimit(s32In, pParam) \
        GFLIB_UpperLimitANSIC((s32In),(pParam))
```

3.17.3 Arguments

Table 3-36: Function parameters

Type	Name	Dir.	Description
tFrac32	s32In	in	Input value.
const GFLIB_UPPERLIMIT_T *const	pParam	in	Pointer to the limits structure.

3.17.4 Return

GFLIB_UpperLimit returns the input value, or the upper limit if the input value is larger than the upper limit.

3.17.5 Description

The 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 limits can be found in the limits structure, supplied to the function as a pointer `pParam`.

3.17.6 Note

The function can be used as an argument of the conditional statement.

3.17.7 Reentrancy

The function is reentrant.

3.17.8 Code Example

```
#include "gflib.h"

tFrac32 x;
tFrac32 y;
GFLIB_UPPERLIMIT_T UpperLimitParam;

void main(void)
{
    // 0.5 = 0x40000000
    UpperLimitParam.s32UpperLimit = FRAC32(0.5);

    // x = 0.75 = 0x60000000
    x  = FRAC32(0.75);
    // y should be 0x40000000
    y = GFLIB_UpperLimit(x, &UpperLimitParam);

    // x = -0.75 = 0xa0000000
    x  = FRAC32(-0.75);
    // y should be 0xa0000000
    y = GFLIB_UpperLimit(x, &UpperLimitParam);

    // x = 0.25 = 0x20000000
    x  = FRAC32(0.25);
    // y should be 0x20000000
    y = GFLIB_UpperLimit(x, &UpperLimitParam);

    return;
}
```

3.17.9 Performance

Table 3-37: GFLIB_UpperLimit function performance

Code size [bytes] IAR	10
Data size [bytes] IAR	0
Execution clock cycles max [clk] IAR	6
Execution clock cycles min [clk] IAR	6

3.18 GFLIB_VectorLimit

3.18.1 Declaration

```
tBool GFLIB_VectorLimitANSIC(SWLIBS_2Syst *const pOut, const SWLIBS_2Syst *const
pIn, const GFLIB_VECTORLIMIT_T *const pParam)
```

3.18.2 Alias

```
#define GFLIB_VectorLimit(s32Out, s32In, pParam) \
        GFLIB_VectorLimitANSIC((s32Out), (s32In), (pParam))
```

3.18.3 Arguments

Table 3-38: Function parameters

Type	Name	Dir.	Description
SWLIBS_2Syst *const	pOut	out	Pointer to the structure of the limited output vector.
const SWLIBS_2Syst *const	pIn	in	Pointer to the structure of the input vector.
const GFLIB_VECTORLIMIT_T *const	pParam	in	Pointer to the parameters structure.

3.18.4 Return

The function will return true (`TRUE`) if the input vector is limited, or false (`FALSE`) otherwise.

3.18.5 Description

The GFLIB_VectorLimit function limits the magnitude of the input vector, keeping its direction unchanged. Limitation is performed as follows:

$$y_{out} = \begin{cases} \frac{y_{in}}{\sqrt{x_{in}^2 + y_{in}^2}} \cdot L & \text{if } \sqrt{x_{in}^2 + y_{in}^2} > L \\ y_{in} & \text{if } \sqrt{x_{in}^2 + y_{in}^2} \leq L \end{cases} \tag{3.39}$$

$$x_{out} = \begin{cases} \frac{x_{in}}{\sqrt{x_{in}^2 + y_{in}^2}} \cdot L & \text{if } \sqrt{x_{in}^2 + y_{in}^2} > L \\ x_{in} & \text{if } \sqrt{x_{in}^2 + y_{in}^2} \leq L \end{cases} \tag{3.40}$$

Where:

- x_{in}, y_{in}, and x_{out}, y_{out} are the coordinates of the input and output vector, respectively

- L is the maximum magnitude of the vector

The input vector coordinates are defined by the structure pointed to by the `pIn` parameter, and the output vector coordinates 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 3-14

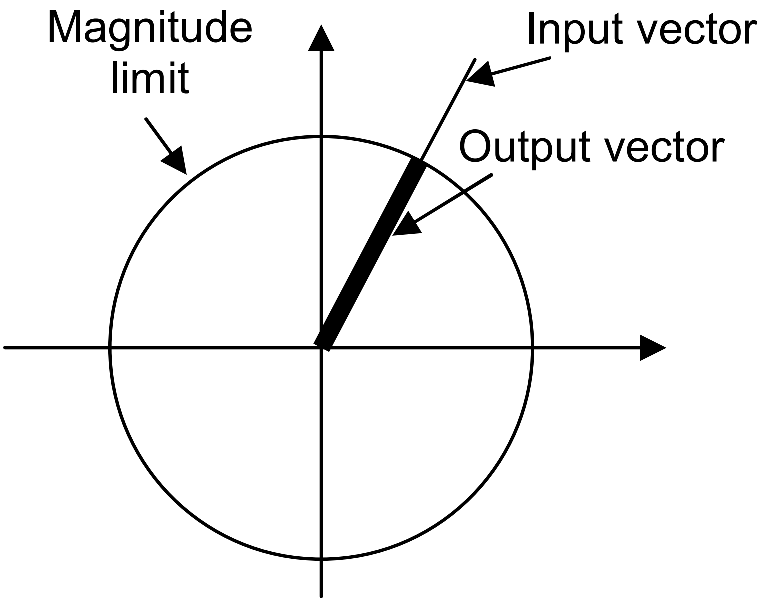


Figure 3-14: Graphical interpretation of the GFLIB_VectorLimit function

If an actual limitation occurs, the function will return the logical true (TRUE), otherwise the logical false will be returned (FALSE). For computational reasons, 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 the logical true (`TRUE`) in this case. Also, the 16 least significant bits of the maximum vector magnitude in the parameters structure, the `pParam->s32Lim`, are ignored. This means that the defined magnitude must be equal to or greater than 2^{-15}, otherwise the result is undefined.

3.18.6 Caution

The maximum vector magnitude in the parameters structure, the `pParam->s32Lim`, must be positive and equal to or greater than 2^{-15}, otherwise the result is undefined. The function does not check for the valid range of `pParam->s32Lim`.

3.18.7 Note

The function calls the square root routine (`GFLIB_Sqrt`).

3.18.8 Reentrancy

The function is reentrant.

3.18.9 Code Example

```
#include "gflib.h"

GFLIB_VECTORLIMIT_T VectorLimitParam;
SWLIBS_2Syst In = { 0, 0};
SWLIBS_2Syst Out = { 0, 0};
tBool bLim;

void main(void)
{
    VectorLimitParam.s32Lim = 0x20000000; // 2^-2
    In.s32Arg1 = 0x20000000; // 2^-2
    In.s32Arg2 = 0x20000000; // 2^-2
    bLim = GFLIB_VectorLimit(&Out, &In, &VectorLimitParam);
    // Calculations:
    // Len = sqrt(2)*2^-2
    // Len > 2^-2, limitation required
    // xout = 2^-2/Len * Lim = sqrt(2)/2 * 2^-2
    // yout = 2^-2/Len * Lim = sqrt(2)/2 * 2^-2
    // sqrt(2)/2*2^-2 = 0x16A09E66
    // The output should be:
    // Out.s32Arg1 = 0x16a08000
    // Out.s32Arg2 = 0x16a08000
    // bLim = TRUE

    return;
}
```

3.18.10 Performance

Table 3-39: GFLIB_VectorLimit function performance

Code size [bytes] IAR	236
Data size [bytes] IAR	0
Execution clock cycles max [clk] IAR	230
Execution clock cycles min [clk] IAR	39

3.19 GFLIB_IntegratorTR

3.19.1 Declaration

```
tFrac32 GFLIB_IntegratorTRANSIC(tFrac32 s32In, GFLIB_INTEGRATOR_TR_T *pParam)
```

3.19.2 Alias

```
#define GFLIB_IntegratorTR(in, pParam) \
        GFLIB_IntegratorTRANSIC(in, pParam)
```

3.19.3 Arguments

Table 3-40: Function parameters

Type	Name	Dir.	Description
tFrac32	s32In	in	Input argument to be integrated.
GFLIB_INTEGRATOR_-TR_T *	pParam	in/out	Pointer to the integrator parameters structure

3.19.4 Return

The function returns a 32-bit value in format Q1.31, which represents the actual integrated value of the input signal.

3.19.5 Description

The GFLIB_IntegratorTRANSIC function, denoting ANSI-C compatible source code implementation, can be called via the function alias GFLIB_IntegratorTR.

The function GFLIB_IntegratorTR implements a discrete integrator using trapezoidal (Bilinear) transformation. The continuous time domain representation of the integrator is defined as:

$$u(t) = \int_0^t e(t)\,dt \tag{3.41}$$

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

$$H(s) = \frac{U(s)}{E(s)} = \frac{1}{s} \tag{3.42}$$

Transforming Equation (3.42) into a digital time domain using Bilinear transformation, leads to the following transfer function:

$$\mathcal{Z}\{H(s)\} = \frac{U(z)}{E(z)} = \frac{T_s + T_s z^{-1}}{2 - 2z^{-1}} \tag{3.43}$$

where T_s is the sampling period of the system. The discrete implementation of the digital transfer function (3.43) is as follows:

$$u(k) = u(k-1) + e(k) \cdot \frac{T_s}{2} + e(k-1) \cdot \frac{T_s}{2} \tag{3.44}$$

Considering fractional maths implementation, the integrator input and output maximal values (scales) must be known. The discrete implementation is then given as follows:

$$u(k) = u(k-1) + e(k) \cdot \frac{T_s}{2}\frac{E_{MAX}}{U_{MAX}} + e(k-1) \cdot \frac{T_s}{2}\frac{E_{MAX}}{U_{MAX}} \tag{3.45}$$

where E_{MAX} is the input scale and U_{MAX} is the output scale. Integrator constant C1 is then defined as:

$$C1_f = \frac{T_s}{2}\frac{E_{MAX}}{U_{MAX}} \tag{3.46}$$

In order to implement the discrete form integrator as in (3.45) on a fixed point platform, the value of $C1_f$ coefficient must reside in a the fractional range $[-1, 1)$. Therefore, scaling must be introduced as follows:

$$s32C1 = C1_f \cdot 2^{-u16NShift} \tag{3.47}$$

The introduced scaling is chosen such that coefficient $s32C1$ 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:

$$u16NShift = \text{ceil}\left(\frac{\log(C1_f)}{\log(2)}\right) \tag{3.48}$$

3.19.6 Note

All parameters and states used by the function can be reset during declaration using the GFLIB_INTEGRATOR_TR_DEFAULT macro.

3.19.7 Reentrancy

The function is reentrant.

3.19.8 Code Example

```
#include "gflib.h"

tFrac32 s32In;
tFrac32 s32Out;

// Definition of one integrator instance
GFLIB_INTEGRATOR_TR_T trMyIntegrator = GFLIB_INTEGRATOR_TR_DEFAULT;

void main(void)
{
    // Setting parameters for integrator, Ts = 100e-6, E_MAX=U_MAX=1
    trMyIntegrator.s32C1          = FRAC32(100e-6/2);
    trMyIntegrator.u16NShift      = 0;
```

```
    // input value = 0.5
    s32In   = FRAC32(0.5);

    // output should be 0x0000D1B7
    s32Out  = GFLIB_IntegratorTR(s32In,&trMyIntegrator);
}
```

3.19.9 Performance

Table 3-41: GFLIB_IntegratorTR function performance

Code size [bytes] IAR	110
Data size [bytes] IAR	0
Execution clock cycles max [clk] IAR	40
Execution clock cycles min [clk] IAR	40

3.20 GFLIB_ControllerPIr

3.20.1 Declaration

```
tFrac32 GFLIB_ControllerPIrANSIC(tFrac32 s32InErr, GFLIB_CONTROLLER_PI_R_T
*pParam)
```

3.20.2 Alias

```
#define GFLIB_ControllerPIr(s32InErr, pParam) \
        GFLIB_ControllerPIrANSIC(s32InErr, pParam)
```

3.20.3 Arguments

Table 3-42: Function parameters

Type	Name	Dir.	Description
tFrac32	s32InErr	in	Input error signal to the controller is a 32-bit number normalized between $[-1, 1)$.
GFLIB_CONTROLLER_-PI_R_T *	pParam	in/out	Pointer to the controller parameters structure.

3.20.4 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.

3.20.5 Description

The GFLIB_ControllerPIrANSIC function, denoting ANSI-C compatible source code implementation, can be called via the function alias GFLIB_ControllerPIr.

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:

$$u(t) = e(t) \cdot K_P + K_I \int_0^t e(t)\, dt \tag{3.49}$$

The transfer function for this kind of PI controller, in a continuous time domain, is described using the Laplace transformation as follows:

$$H(s) = \frac{U(s)}{E(s)} = \frac{K_P + sK_I}{s} \tag{3.50}$$

Transforming equation (3.50) into a discrete time domain leads to the following equation:

$$u(k) = u(k-1) + e(k) \cdot CC1 + e(k-1) \cdot CC2 \tag{3.51}$$

where K_P is proportional gain, K_I is integral gain, Ts 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 3-43.

Table 3-43: Calculation of coefficients CC1 and CC2 using various discretization methods

	Trapezoidal	Bakward Rect.	Forward Rect.
$CC1 =$	$K_p + K_i \frac{T_s}{2}$	$K_p + K_i T_s$	K_p
$CC2 =$	$-K_p + K_i \frac{T_s}{2}$	$-K_p$	$-K_p + K_i T_s$

In order to implement the discrete equation of the controller (3.51) on the fixed point arithmetic platform, the maximal values (scales) of the input and output signals: E^{MAX} is maximal value of the controller input error signal U^{MAX} is maximal value of the controller output signal have to be known beforehand. 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:

$$e_f(k) = \frac{e(k)}{E^{MAX}} \tag{3.52}$$

$$u_f(k) = \frac{u(k)}{U^{MAX}} \tag{3.53}$$

The resulting controller discrete time domain equation in fixed point fractional representation is therefore given as:

$$u_f(k) \cdot U^{MAX} = u_f(k-1) \cdot U^{MAX} + e_f(k) \cdot E^{MAX} \cdot CC1 + e_f(k-1) \cdot E^{MAX} \cdot CC2 \tag{3.54}$$

which can be rearranged into the following form:

$$u_f(k) = u_f(k-1) + e_f(k) \cdot CC1_f + e_f(k-1) \cdot CC2_f \tag{3.55}$$

where

$$CC1_f = CC1 \cdot \frac{E^{MAX}}{U^{MAX}} \quad , \quad CC2_f = CC2 \cdot \frac{E^{MAX}}{U^{MAX}} \tag{3.56}$$

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

- E^{MAX} - maximal value of the controller input error signal
- U^{MAX} - maximal value of the controller output signal fractional range. Therefore, a scaling of $CC1_f, CC2_f$ is introduced as:

$$s32CC1sc = CC1_f \cdot 2^{-u16NShift} \tag{3.57}$$

$$s32CC2sc = CC2_f \cdot 2^{-u16NShift} \tag{3.58}$$

The introduced scaling shift $u16NShift$ is chosen so that both coefficients $s32CC1sc$, $s32CC2sc$ 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 $[-1, 1)$.

$$u16NShift = \max\left(\text{ceil}\left(\frac{\log(\text{abs}(CC1_f))}{\log(2)}\right), \text{ceil}\left(\frac{\log(\text{abs}(CC2_f))}{\log(2)}\right)\right) \tag{3.59}$$

The final, scaled, fractional equation of a reccurent PI controller on a 32-bit fixed point platform is therefore implemented as follows:

$$u_f(k) \cdot (2^{-u16NShift}) = u_f(k-1) \cdot (2^{-u16NShift}) + e_f(k) \cdot s32CC1sc + e_f(k-1) \cdot s32CC2sc \tag{3.60}$$

where:

- $u_f(k)$ - fractional representation $[-1, 1)$ of the controller output
- $e_f(k)$ - fractional representation $[-1, 1)$ of the controller input (error)
- $s32CC1sc$ - fractional representation $[-1, 1)$ of the 1st controller coefficient
- $s32CC2sc$ - fractional representation $[-1, 1)$ of the 2nd controller coefficient
- $u16NShift$ - in range $[0, 31]$ - is chosen such that both coefficients $s32CC1sc$ and $s32CC2sc$ are in the range $[-1, 1)$

3.20.6 Note

All controller parameters and states can be reset during declaration using the GFLIB_CONTROLLER_-PI_R_DEFAULT macro.

3.20.7 Reentrancy

The function is reentrant.

3.20.8 Code Example

```
#include "gflib.h"

tFrac32 s32InErr;
tFrac32 s32Output;

GFLIB_CONTROLLER_PI_R_T trMyPI = GFLIB_CONTROLLER_PI_R_DEFAULT;

void main(void)
{
    // input error = 0.25
    s32InErr  = FRAC32(0.25);
```

```
    // controller parameters
    trMyPI.s32CC1sc              = FRAC32(0.01);
    trMyPI.s32CC2sc              = FRAC32(0.02);
    trMyPI.u16NShift             = 1;

    // output should be 0x00A3D70A
    s32Output = GFLIB_ControllerPIr(s32InErr,&trMyPI);
}
```

3.20.9 Performance

Table 3-44: GFLIB_ControllerPIr function performance

Code size [bytes] IAR	114
Data size [bytes] IAR	0
Execution clock cycles max [clk] IAR	40
Execution clock cycles min [clk] IAR	40

3.21 GFLIB_ControllerPIp

3.21.1 Declaration

```
tFrac32 GFLIB_ControllerPIpANSIC(tFrac32 s32InErr, GFLIB_CONTROLLER_PI_P_T
*pParam)
```

3.21.2 Alias

```
#define GFLIB_ControllerPIp(s32InErr, pParam) \
        GFLIB_ControllerPIpANSIC(s32InErr, pParam)
```

3.21.3 Arguments

Table 3-45: Function parameters

Type	Name	Dir.	Description
tFrac32	s32InErr	in	Input error signal to the controller is a 32-bit number normalized between $[-1, 1)$.
GFLIB_CONTROLLER_-PI_P_T *	pParam	in/out	Pointer to the controller parameters structure.

3.21.4 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.

3.21.5 Description

The GFLIB_ControllerPIpANSIC function, denoting ANSI-C compatible implementation, can be called via the function alias GFLIB_ControllerPIp.

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_ControllerPIpANSIC 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. Nevertheless, the accumulator overflow is prevented by correct saturation of the controller output at maximal values: $[-1, 1)$ in fractional interpretation, or $\left[-2^{31}, 2^{31}-1\right)$ in integer interpretation. The PI algorithm in the continuous time domain can be described as:

$$u(t) = e(t) \cdot K_P + K_I \int_0^t e(t)\, dt \tag{3.61}$$

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 (3.61) can be described using the Laplace transformation as follows:

$$H(s) = \frac{U(s)}{E(s)} = K_P + K_I \frac{1}{s} \tag{3.62}$$

The proportional part of Equation (3.62) is transformed into the discrete time domain simply as:

$$u_P(k) = K_P \cdot e(k) \tag{3.63}$$

Transforming the integral part of Equation (3.62) into a discrete time domain using the Bilinear method, also known as trapezoidal approximation, leads to the following equation:

$$u_I(k) = u_I(k-1) + e(k) \cdot \frac{K_I T_s}{2} + e(k-1) \cdot \frac{K_I T_s}{2} \tag{3.64}$$

where T_s [sec] is the sampling time. 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 beforehand. 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:

$$e_f(k) = \frac{e(k)}{E^{MAX}} \tag{3.65}$$

$$u_f(k) = \frac{u(k)}{U^{MAX}} \tag{3.66}$$

Applying such scaling (normalization) on the proportional term of Equation (3.63) results in:

$$u_{Pf}(k) = e_f(k) \cdot K_{P\_sc} \tag{3.67}$$

where $K_{P\_sc} = K_P \cdot \frac{E^{MAX}}{U^{MAX}}$ is the proportional gain parameter considering input/output scaling. Analogically, scaling the integral term of Equation (3.64) results in:

$$u_{If}(k) = u_{If}(k-1) + K_{I\_sc} \cdot e_f(k) + K_{I\_sc} \cdot e_f(k-1) \tag{3.68}$$

where $K_{I\_sc} = \frac{K_I T_s}{2} \cdot \frac{E^{MAX}}{U^{MAX}}$ 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:

$$s32PropGain = K_{P\_sc} \cdot 2^{s16PropGainShift} \tag{3.69}$$

and

$$s32IntegGain = K_{I\_sc} \cdot 2^{s16IntegGainShift} \tag{3.70}$$

where

- $s16PropGain$ is the scaled value of proportional gain $[-1, 1)$
- $s16PropGainShift$ is the scaling shift for proportional gain $[-31, 31]$
- $s16IntegGain$ is the scaled value of integral gain $[-1, 1)$
- $s16IntegGainShift$ is the scaling shift for integral gain $[-31, 31]$

3.21.6 Note

All controller parameters and states can be reset during declaration using the GFLIB_CONTROLLER_PI_P_DEFAULT macro.

3.21.7 Reentrancy

The function is reentrant.

3.21.8 Code Example

```
#include "gflib.h"

tFrac32 s32InErr;
tFrac32 s32Output;

GFLIB_CONTROLLER_PI_P_T trMyPI = GFLIB_CONTROLLER_PI_P_DEFAULT;

void main(void)
{
    // input error = 0.25
    s32InErr  = FRAC32(0.25);

    // controller parameters
    trMyPI.s32PropGain        = FRAC32(0.01);
    trMyPI.s32IntegGain       = FRAC32(0.02);
    trMyPI.s16PropGainShift    = 1;
    trMyPI.s16IntegGainShift   = 1;
    trMyPI.s32IntegPartK_1    = 0;

    // output should be 0x01EB851E
    s32Output = GFLIB_ControllerPIp(s32InErr,&trMyPI);
}
```

3.21.9 Performance

Table 3-46: GFLIB_ControllerPIp function performance

Code size [bytes] IAR	190
Data size [bytes] IAR	0
Execution clock cycles max [clk] IAR	57
Execution clock cycles min [clk] IAR	57

3.22 GFLIB_ControllerPIpAW

3.22.1 Declaration

```
tFrac32 GFLIB_ControllerPIpAWANSIC(tFrac32 s32InErr, GFLIB_CONTROLLER_PIAW_P_T
*pParam)
```

3.22.2 Alias

```
#define GFLIB_ControllerPIpAW(s32InErr, pParam) \
        GFLIB_ControllerPIpAWANSIC(s32InErr, pParam)
```

3.22.3 Arguments

Table 3-47: Function parameters

Type	Name	Dir.	Description
tFrac32	s32InErr	in	Input error signal to the controller is a 32-bit number normalized between $[-1, 1)$.
GFLIB_CONTROLLER_-PIAW_P_T *	pParam	in/out	Pointer to the controller parameters structure.

3.22.4 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.

3.22.5 Description

The GFLIB_ControllerPIpAWANSIC function, denoting ANSI-C compatible implementation, can be called via the function alias GFLIB_ControllerPIpAW.

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_ControllerPIpAWANSIC 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 $(s32UpperLimit$ and $s32LowerLimit)$ 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 - GFLIB_CONTROLLER_PIAW_P_T. 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:

$$u(t) = e(t) \cdot K_P + K_I \int_0^t e(t)\, dt \tag{3.71}$$

where

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

Equation (3.71) can be described using the Laplace transformation as follows:

$$H(s) = \frac{U(s)}{E(s)} = K_P + K_I \frac{1}{s} \tag{3.72}$$

The proportional part of equation (3.72) is transformed into the discrete time domain simply as:

$$u_P(k) = K_P \cdot e(k) \tag{3.73}$$

Transforming the integral part of equation (3.72) into a discrete time domain using the Bilinear method, also known as trapezoidal approximation, leads to the following equation:

$$u_I(k) = u_I(k-1) + e(k) \cdot \frac{K_I T_s}{2} + e(k-1) \cdot \frac{K_I T_s}{2} \tag{3.74}$$

where T_s [sec] is the sampling time. 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:

$$e_f(k) = \frac{e(k)}{E^{MAX}} \tag{3.75}$$

$$u_f(k) = \frac{u(k)}{U^{MAX}} \tag{3.76}$$

Applying such scaling (normalization) on the proportional term of Equation (3.73) results in:

$$u_{Pf}(k) = e_f(k) \cdot K_{P\_sc} \tag{3.77}$$

where $K_{P\_sc} = K_P \cdot \frac{E^{MAX}}{U^{MAX}}$ is the proportional gain parameter considering input/output scaling. Analogically, scaling the integral term of Equation (3.74) results in:

$$u_{If}(k) = u_{If}(k-1) + K_{I\_sc} \cdot e_f(k) + K_{I\_sc} \cdot e_f(k-1) \tag{3.78}$$

where $K_{I\_sc} = \frac{K_I T_s}{2} \cdot \frac{E^{MAX}}{U^{MAX}}$ 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:

$$s32PropGain = K_{P\_sc} \cdot 2^{s16PropGainShift} \tag{3.79}$$

$$s32IntegGain = K_{I\_sc} \cdot 2^{s16IntegGainShift} \tag{3.80}$$

where

- $s16PropGain$ is the scaled value of proportional gain $[-1, 1)$
- $s16PropGainShift$ is the scaling shift for proportional gain $[-31, 31]$
- $s16IntegGain$ is the scaled value of integral gain $[-1, 1)$
- $s16IntegGainShift$ is the scaling shift for integral gain $[-31, 31]$

The output signal limitation is implemented in this controller. The actual output $u(k)$ is bounded not to exceed the given limit values $s32UpperLimit$, $s32LowerLimit$. This is due to either the bounded power of the actuator or to the physical constraints of the plant.

$$u_f(k) = \begin{cases} \text{s32UpperLimit} & \rightarrow \quad u_f(k) \geq \text{s32UpperLimit} \\ u_f(k) & \rightarrow \quad \text{s32LowerLimit} < u_f(k) < \text{s32UpperLimit} \\ \text{s32LowerLimit} & \rightarrow \quad u_f(k) \leq \text{s32LowerLimit} \end{cases}$$

The bounds are described by a limitation element Equation (3.81). When the bounds are exceeded the non-linear saturation characteristic will take effect and influence the dynamic behaviour. 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.

3.22.6 Note

All controller parameters and states can be reset during declaration using the GFLIB_CONTROLLER_-PIAW_P_DEFAULT macro.

3.22.7 Reentrancy

The function is reentrant.

3.22.8 Code Example

```
#include "gflib.h"

tFrac32 s32InErr;
tFrac32 s32Output;

GFLIB_CONTROLLER_PIAW_P_T trMyPI = GFLIB_CONTROLLER_PIAW_P_DEFAULT;

void main(void)
{
    // input error = 0.25
    s32InErr  = FRAC32(0.25);

    // controller parameters
    trMyPI.s32PropGain         = FRAC32(0.01);
    trMyPI.s32IntegGain        = FRAC32(0.02);
    trMyPI.s16PropGainShift     = 1;
    trMyPI.s16IntegGainShift    = 1;
    trMyPI.s32IntegPartK_1     = 0;
    trMyPI.s32UpperLimit       = FRAC32(1.0);
    trMyPI.s32LowerLimit       = FRAC32(-1.0);

    // output should be 0x01EB851E
    s32Output = GFLIB_ControllerPIpAW(s32InErr,&trMyPI);
}
```

3.22.9 Performance

Table 3-48: GFLIB_ControllerPIpAW function performance

Code size [bytes] IAR	242
Data size [bytes] IAR	0
Execution clock cycles max [clk] IAR	96
Execution clock cycles min [clk] IAR	77

CHAPTER 4: GMCLIB

4.1 Function API Overview

- void GMCLIB_Clark (SWLIBS_2Syst * pOut, const SWLIBS_3Syst *const pIn)
  Clarke Transformation algorithm implementation
- void GMCLIB_ClarkInv (SWLIBS_3Syst * pOut, const SWLIBS_2Syst *const pIn)
  Inverse Clarke Transformation algorithm implementation
- void GMCLIB_Park (SWLIBS_2Syst * pOut, const SWLIBS_2Syst *const pInAngle, const SWLIBS_2Syst *const pIn)
  Park Transformation algorithm implementation
- void GMCLIB_ParkInv (SWLIBS_2Syst * pOut, const SWLIBS_2Syst *const pInAngle, const SWLIBS_2Syst *const pIn)
  Inverse Park Transformation algorithm implementation
- void GMCLIB_ElimDcBusRip (SWLIBS_2Syst * pOut, const SWLIBS_2Syst *const pIn, const GMCLIB_ELIMDCBUSRIP_T *const pParam)
  Elimination of the DC bus voltage ripple
- void GMCLIB_DecouplingPMSM (SWLIBS_2Syst * pUdqDec, const SWLIBS_2Syst *const pUdq, const SWLIBS_2Syst *const pIdq, tFrac32 s32AngularVel, const GMCLIB_DECOUPLINGPMSM_T *const pParam)
  Calculates the cross-coupling voltages to eliminate the dq axis coupling causing non-linearity of the field oriented control
- tU32 GMCLIB_SvmStd (SWLIBS_3Syst * pOut, const SWLIBS_2Syst *const pIn)
  Function alias GMCLIB_SvmStd

4.2 GMCLIB_Clark

4.2.1 Declaration

```
void GMCLIB_ClarkANSIC(SWLIBS_2Syst *pOut, const SWLIBS_3Syst *const pIn)
```

4.2.2 Alias

```
#define GMCLIB_Clark(pOut, pIn) \
        GMCLIB_ClarkANSIC(pOut,pIn)
```

4.2.3 Arguments

Table 4-1: Function parameters

Type	Name	Dir.	Description
SWLIBS_2Syst *	pOut	out	Pointer to the structure containing data of the two-phase stationary orthogonal system $(\alpha - \beta)$.
const SWLIBS_3Syst *const	pIn	in	Pointer to the structure containing data of the three-phase stationary system $(sA - sB - sC)$.

4.2.4 Return

The function returns data type void.

4.2.5 Description

The GMCLIB_ClarkANSIC function, denoting ANSI-C compatible implementation, can be called via the function alias GMCLIB_Clark. The Clarke Transformation is used to transform values from the three-phase (A-B-C) coordinate system to the two-phase (alpha-beta) orthogonal coordinate system, according to the following equations:

$$i_\alpha = i_{sA} \tag{4.1}$$

$$i_\beta = (i_{sB} - i_{sC}) \cdot \frac{1}{\sqrt{3}} \tag{4.2}$$

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

4.2.6 Note

The inputs and the outputs are normalized to fit in the range $[-1, 1)$.

4.2.7 Reentrancy

The function is reentrant.

4.2.8 Code Example

```
#include "gmclib.h"

SWLIBS_3Syst tr32Abc;
SWLIBS_2Syst tr32AlBe;

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
    tr32Abc.s32Arg1  = FRAC32(0.707106781);
    tr32Abc.s32Arg2  = FRAC32(0.258819045);
    tr32Abc.s32Arg3  = FRAC32(-0.965925826);

    // output should be
    // tr32AlBe.s32Arg1 ~ alpha = 0x5A827999
    // tr32AlBe.s32Arg2 ~ beta  = 0x5A827999
    GMCLIB_Clark(&tr32AlBe,&tr32Abc);
}
```

4.2.9 Performance

Table 4-2: GMCLIB_Clark function performance

Code size [bytes] IAR	48
Data size [bytes] IAR	0
Execution clock cycles max [clk] IAR	29
Execution clock cycles min [clk] IAR	29

4.3 GMCLIB_ClarkInv

4.3.1 Declaration

```
void GMCLIB_ClarkInvANSIC(SWLIBS_3Syst *pOut, const SWLIBS_2Syst *const pIn)
```

4.3.2 Alias

```
#define GMCLIB_ClarkInv(pOut, pIn) \
        GMCLIB_ClarkInvANSIC(pOut, pIn)
```

4.3.3 Arguments

Table 4-3: Function parameters

Type	Name	Dir.	Description
SWLIBS_3Syst *	pOut	out	Pointer to the structure containing data of the three-phase stationary system $(sA - sB - sC)$.
const SWLIBS_2Syst *const	pIn	in	Pointer to the structure containing data of the two-phase stationary orthogonal system $(\alpha - \beta)$.

4.3.4 Return

The function returns data type void.

4.3.5 Description

The GMCLIB_ClarkInvANSIC function, denoting ANSI-C compatible implementation, can be called via the function alias GMCLIB_ClarkInv. The GMCLIB_ClarkInv function calculates the Inverse Clarke transformation, which is used to transform values from the two-phase $(\alpha - \beta)$ orthogonal coordinate system to the three-phase $(sA - sB - sC)$ coordinate system, according to these equations:

$$i_{sA} = i_{\alpha} \tag{4.3}$$

$$i_{sB} = -\frac{1}{2} \cdot i_{\alpha} + \frac{\sqrt{3}}{2} \cdot i_{\beta} \tag{4.4}$$

$$i_{sC} = -\frac{1}{2} \cdot i_{\alpha} - \frac{\sqrt{3}}{2} \cdot i_{\beta} \tag{4.5}$$

4.3.6 Note

The inputs and the outputs are normalized to fit in the range $[-1, 1)$.

4.3.7 Reentrancy

The function is reentrant.

4.3.8 Code Example

```
#include "gmclib.h"

SWLIBS_2Syst tr32AlBe;
SWLIBS_3Syst tr32Abc;

void main(void)
{
    // input phase alpha = sin(45) = 0.707106781
    // input phase beta  = cos(45) = 0.707106781
    tr32AlBe.s32Arg1  = FRAC32(0.707106781);
    tr32AlBe.s32Arg2  = FRAC32(0.707106781);

    // output should be
    // tr32Abc.s32Arg1 ~ phA = 0x5A827999
    // tr32Abc.s32Arg2 ~ phB = 0x2120FB83
    // tr32Abc.s32Arg3 ~ phC = 0x845C8AE5
    GMCLIB_ClarkInv(&tr32Abc,&tr32AlBe);
}
```

4.3.9 Performance

Table 4-4: GMCLIB_ClarkInv function performance

Code size [bytes] IAR	48
Data size [bytes] IAR	0
Execution clock cycles max [clk] IAR	25
Execution clock cycles min [clk] IAR	25

4.4 GMCLIB_Park

4.4.1 Declaration

```
void GMCLIB_ParkANSIC(SWLIBS_2Syst *pOut, const SWLIBS_2Syst *const pInAngle,
const SWLIBS_2Syst *const pIn)
```

4.4.2 Alias

```
#define GMCLIB_Park(pOut, pInAngle, pIn) \
        GMCLIB_ParkANSIC(pOut,pInAngle,pIn)
```

4.4.3 Arguments

Table 4-5: Function parameters

Type	Name	Dir.	Description
SWLIBS_2Syst *	pOut	out	Pointer to the structure containing data of the two-phase rotational orthogonal system $(d - q)$.
const SWLIBS_2Syst *const	pInAngle	in	Pointer to the structure where the values of the sine and cosine of the rotor position are stored.
const SWLIBS_2Syst *const	pIn	in	Pointer to the structure containing data of the two-phase stationary orthogonal system $(\alpha - \beta)$.

4.4.4 Return

The function returns data type void.

4.4.5 Description

The GMCLIB_ParkANSIC function, denoting ANSI-C compatible implementation, can be called via the function alias GMCLIB_Park. The GMCLIB_Park function calculates the Park Transformation, which transforms values (flux, voltage, current) from the two-phase $(\alpha - \beta)$ stationary orthogonal coordinate system to the two-phase $(d - q)$ rotational orthogonal coordinate system, according to these equations:

$$d = \cos(\theta_e) \cdot \alpha + sin(\theta_e) \cdot \beta \tag{4.6}$$

$$q = -\sin(\theta_e) \cdot \alpha + cos(\theta_e) \cdot \beta \tag{4.7}$$

where θ_e represents the electrical position of the rotor flux.

4.4.6 Note

The inputs and the outputs are normalized to fit in the range $[-1, 1)$.

4.4.7 Reentrancy

The function is reentrant.

4.4.8 Code Example

```
#include "gmclib.h"

SWLIBS_2Syst tr32Angle;
SWLIBS_2Syst tr32AlBe;
SWLIBS_2Syst tr32Dq;

void main(void)
{
    // input angle sin(60) = 0.866025403
    // input angle cos(60) = 0.5
    tr32Angle.s32Arg1  = FRAC32(0.866025403);
    tr32Angle.s32Arg2  = FRAC32(0.5);

    // input alpha = 0.123
    // input beta = 0.654
    tr32AlBe.s32Arg1  = FRAC32(0.123);
    tr32AlBe.s32Arg2  = FRAC32(0.654);

    // output should be
    // tr32Dq.s32Arg1 ~ d = 0x505E6455
    // tr32Dq.s32Arg2 ~ q = 0x1C38ABDC
    GMCLIB_Park(&tr32Dq,&tr32Angle,&tr32AlBe);
}
```

4.4.9 Performance

Table 4-6: GMCLIB_Park function performance

Code size [bytes] IAR	130
Data size [bytes] IAR	0
Execution clock cycles max [clk] IAR	74
Execution clock cycles min [clk] IAR	74

4.5 GMCLIB_ParkInv

4.5.1 Declaration

```
void GMCLIB_ParkInvANSIC(SWLIBS_2Syst *pOut, const SWLIBS_2Syst *const pInAngle,
const SWLIBS_2Syst *const pIn)
```

4.5.2 Alias

```
#define GMCLIB_ParkInv(pOut, pInAngle, pIn) \
        GMCLIB_ParkInvANSIC(pOut, pInAngle, pIn)
```

4.5.3 Arguments

Table 4-7: Function parameters

Type	Name	Dir.	Description
SWLIBS_2Syst *	pOut	out	Pointer to the structure containing data of the two-phase stationary orthogonal system $(\alpha - \beta)$.
const SWLIBS_2Syst *const	pInAngle	in	Pointer to the structure where the values of the sine and cosine of the rotor position are stored.
const SWLIBS_2Syst *const	pIn	in	Pointer to the structure containing data of the two-phase rotational orthogonal system $(d - q)$.

4.5.4 Return

The function returns data type void.

4.5.5 Description

The GMCLIB_ParkInvANSIC function, denoting ANSI-C compatible implementation, can be called via the function alias GMCLIB_ParkInv. The GMCLIB_ParkInv function calculates the Inverse Park Transformation, which transforms quantities (flux, voltage, current) from the two-phase $(d - q)$ rotational orthogonal coordinate system to the two-phase $(\alpha - \beta)$ stationary orthogonal coordinate system, according to these equations:

$$\alpha = \cos(\theta_e) \cdot d - \sin(\theta_e) \cdot q \tag{4.8}$$

$$\beta = \sin(\theta_e) \cdot d + \cos(\theta_e) \cdot q \tag{4.9}$$

4.5.6 Note

The inputs and the outputs are normalized to fit in the range $[-1, 1)$.

4.5.7 Reentrancy

The function is reentrant.

4.5.8 Code Example

```
#include "gmclib.h"

SWLIBS_2Syst tr32Angle;
SWLIBS_2Syst tr32Dq;
SWLIBS_2Syst tr32AlBe;

void main(void)
{
    // input angle sin(60) = 0.866025403
    // input angle cos(60) = 0.5
    tr32Angle.s32Arg1  = FRAC32(0.866025403);
    tr32Angle.s32Arg2  = FRAC32(0.5);

    // input d  = 0.123
    // input q  = 0.654
    tr32Dq.s32Arg1  = FRAC32(0.123);
    tr32Dq.s32Arg2  = FRAC32(0.654);

    // output should be
    // tr32AlBe.s32Arg1 ~ alpha = 0xBF601273
    // tr32AlBe.s32Arg2 ~ beta  = 0x377D9EE4
    GMCLIB_ParkInv(&tr32AlBe,&tr32Angle,&tr32Dq);
}
```

4.5.9 Performance

Table 4-8: GMCLIB_ParkInv function performance

Code size [bytes] IAR	130
Data size [bytes] IAR	0
Execution clock cycles max [clk] IAR	74
Execution clock cycles min [clk] IAR	74

4.6 GMCLIB_ElimDcBusRip

4.6.1 Declaration

```
void GMCLIB_ElimDcBusRipANSIC(SWLIBS_2Syst *pOut, const SWLIBS_2Syst *const pIn,
const GMCLIB_ELIMDCBUSRIP_T *const pParam)
```

4.6.2 Alias

```
#define GMCLIB_ElimDcBusRip(pOut, pIn, pParams) \
        GMCLIB_ElimDcBusRipANSIC(pOut, pIn, pParams)
```

4.6.3 Arguments

Table 4-9: Function parameters

Type	Name	Dir.	Description
SWLIBS_2Syst *	pOut	out	Pointer to the structure with direct (α) and quadrature (β) components of the required stator voltage vector re-calculated so as to compensate for voltage ripples on the DC bus.
const SWLIBS_2Syst *const	pIn	in	Pointer to the structure with direct (α) and quadrature (β) components of the required stator voltage vector before compensation of voltage ripples on the DC bus.
const GMCLIB_-ELIMDCBUSRIP_T *const	pParam	in	Pointer to the parameters structure.

4.6.4 Return

The function returns data type void.

4.6.5 Description

The GMCLIB_ElimDcBusRipANSIC function, denoting ANSI-C compatible implementation, can be called via the function alias GMCLIB_ElimDcBusRip. 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. Obviously, 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. For a better understanding, let's consider the following two simple examples:

Example 1

- amplitude of the required phase voltage $U_{req} = 50$[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 $U_{req} = 50$[V] on the inverter phase output:

$$U_{req\_new} = \frac{U_{req} \cdot U_{DC\_BUS\_MAX}}{U_{DC\_BUS\_ACTUAL}} = 50V \tag{4.10}$$

Example 2:

- amplitude of the required phase voltage $U_{req} = 50$[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 $U_{req} = 50$[V] on the inverter phase output:

$$U_{req\_new} = \frac{U_{req} \cdot U_{DC\_BUS\_MAX}}{U_{DC\_BUS\_ACTUAL}} = 55.5V \tag{4.11}$$

The imperfections of the DC bus voltage are compensated for by the modification of amplitudes of the direct- α and the quadrature- β components of the stator reference voltage vector. The following formulas are used:

- for the α-component:

$$u_\alpha^* = \begin{cases} \dfrac{s32ModIndex \cdot u_\alpha}{s32ArgDcBusMsr/2} & \text{if } abs\left(s32ModIndex \cdot u_\alpha\right) < \frac{s32ArgDcBusMsr}{2} \\ sign\left(u_\alpha\right) & \text{otherwise} \end{cases} \tag{4.12}$$

- for the β-component:

$$u_\beta^* = \begin{cases} \dfrac{s32ModIndex \cdot u_\beta}{s32ArgDcBusMsr/2} & \text{if } abs\left(s32ModIndex \cdot u_\beta\right) < \frac{s32ArgDcBusMsr}{2} \\ sign\left(u_\beta\right) & \text{otherwise} \end{cases} \tag{4.13}$$

where: $s32ModIndex$ is the inverse modulation index, $s32ArgDcBusMsr$ 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 $s32ModIndex$ and $s32ArgDcBusMsr$ are supplied to the function within the parameters structure through its members. The u_α, u_β correspond respectively to the `s32Arg1` and `s32Arg2` members of the input structure, and the u_α^* and u_β^* respectively to the `s32Arg1` and `s32Arg2` members of the output structure. It should be noted that although the modulation index (`s32ModIndex`) is assumed to be equal to or greater than zero, the possible values are restricted to those values resulting from the use of Space Vector Modulation techniques.

In order to correctly handle the discontinuity at $s32ArgDcBusMsr$ approaching 0, and for efficiency reasons, the function will assign 0 to the output duty cycle ratios if the $s32ArgDcBusMsr$ is below the threshold of 2^{-15}. In other words, the 16 least significant bits of the `s32DcBusMsr` are ignored. Additionally, the computed output of the $u_\alpha*$ and u_β^* components may be inaccurate in the 16 least significant bits.

4.6.6 Note

Both the inverse modulation index `pIn->s32ModIndex` and the measured DC bus voltage `pIn->s32DcBusMsr` must be equal to or greater than 0, otherwise the results are undefined.

4.6.7 Reentrancy

The function is reentrant.

4.6.8 Code Example

```
#include "gmclib.h"

#define U_MAX    (36.0)      // Voltage scale

SWLIBS_2Syst tr32InVoltage;
SWLIBS_2Syst tr32OutVoltage;

GMCLIB_ELIMDCBUSRIP_T trMyElimDCB = GMCLIB_ELIMDCBUSRIP_DEFAULT;

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.s32Arg1        = FRAC32(12.99/U_MAX);
    tr32InVoltage.s32Arg2        = FRAC32(7.5/U_MAX);

    // inverse modulation coefficient for standard space vector modulation
    trMyElimDCB.s32ModIndex      = FRAC32(0.866025403784439);

    // value of "measured" DC bus voltage 17V
    // When used in final application this randomly chosen value shall be
    // replaced by actual value of measured voltage on DC bus terminals of
    // the power inverter
    trMyElimDCB.s32ArgDcBusMsr   = FRAC32(17.0/U_MAX);

    // output should be
```

```
    // alpha component of the output voltage vector:
    //     (12.99/36)*0.8660/(17.0/36/2) = 1.3235 -> 1.0 -> 0x7fffffff
    // beta component of the output voltage vector :
    //     (7.5/36)*0.8660/(17.0/36/2) = 0.7641 -> 0x61cf5770
    // due to 16-bit accuracy the result will be 0x61cf8000

    GMCLIB_ElimDcBusRip(&tr32OutVoltage,&tr32InVoltage,&trMyElimDCB);

    return;
}
```

4.6.9 Performance

Table 4-10: GMCLIB_ElimDcBusRip function performance

Code size [bytes] IAR	246
Data size [bytes] IAR	0
Execution clock cycles max [clk] IAR	140
Execution clock cycles min [clk] IAR	36

4.7 GMCLIB_DecouplingPMSM

4.7.1 Declaration

```
void GMCLIB_DecouplingPMSMANSIC(SWLIBS_2Syst *pUdqDec, const SWLIBS_2Syst *const
pUdq, const SWLIBS_2Syst *const pIdq, tFrac32 s32AngularVel, const
GMCLIB_DECOUPLINGPMSM_T *const pParam)
```

4.7.2 Alias

```
#define GMCLIB_DecouplingPMSM(pUdqDec, pUdq, pIdq, s32AngularVel, pParam) \
        GMCLIB_DecouplingPMSMANSIC(pUdqDec,pUdq,pIdq,s32AngularVel,pParam)
```

4.7.3 Arguments

Table 4-11: Function parameters

Type	Name	Dir.	Description
SWLIBS_2Syst *	pUdqDec	out	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 *const	pUdq	in	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 *const	pIdq	in	Pointer to the structure containing direct (i_{df}) and quadrature (i_{qf}) components of the stator current vector measured on the motor terminals
tFrac32	s32AngularVel	in	Rotor angular velocity in rad/sec, referred to as (ω_{ef}) in the detailed section of the documentation
const GMCLIB_DECOUPLINGPMSM_T *const	pParam	in	Pointer to the structure containing k_{df} and k_{qf} coefficients and scale parameters (k_{d_shift}) and (k_{q_shift})

4.7.4 Return

The function returns data type void.

4.7.5 Description

The GMCLIB_DecouplingPMSMANSIC function, denoting ANSI-C compatible source code implementation, can be called via the function alias GMCLIB_DecouplingPMSM.

The quadrature phase model of a PMSM motor, in a synchronous reference frame, is popular for field-oriented control structures because both controllable quantities, current and voltage, are DC values. This allows for 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:

$$\begin{bmatrix} u_d \\ u_q \end{bmatrix} = \underbrace{R_s \begin{bmatrix} i_d \\ i_q \end{bmatrix} + \begin{bmatrix} L_d & 0 \\ 0 & L_q \end{bmatrix} \frac{\mathrm{d}}{\mathrm{d}t} \begin{bmatrix} i_d \\ i_q \end{bmatrix}}_{linear} + \underbrace{\omega_e \begin{bmatrix} -Lq \\ Ld \end{bmatrix} \begin{bmatrix} i_q \\ i_d \end{bmatrix}}_{cross-coupling} + \underbrace{\omega_e \psi_{pm} \begin{bmatrix} 0 \\ 1 \end{bmatrix}}_{backEMF} \tag{4.14}$$

It can bee seen that (4.14) 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 (4.14) 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:

$$\begin{aligned} u_d &= R_s i_d + L_d \tfrac{\mathrm{d}i_d}{\mathrm{d}t} \\ u_q &= R_s i_q + L_q \tfrac{\mathrm{d}i_q}{\mathrm{d}t} \end{aligned} \tag{4.15}$$

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 (4.14), using the feed-forward voltages $\overline{u}_{dq_{comp}}$ given from (4.14) as follows:

$$\begin{aligned} u_{d_{comp}} &= -\omega_e \cdot L_q \cdot i_q \\ u_{q_{comp}} &= \omega_e \cdot L_d \cdot i_d \end{aligned} \tag{4.16}$$

The feed-forward voltages $\overline{u}_{dq_{comp}}$ are added to the voltages generated by the current controllers $\overline{u}_{dq}$, which cover the R-L model. The resulting voltages represent the direct $u_{d_{dec}}$ 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 $\overline{u}_{dq_{comp}}$ and adds these to the input $\overline{u}_{dq}$ 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:

$$\begin{aligned} u_{d_{dec}} &= u_d + u_{d_{comp}} \\ u_{q_{dec}} &= u_q + u_{q_{comp}} \end{aligned} \tag{4.17}$$

where $\overline{u}_{dq}$ is the voltage vector calculated by the controllers (with the already-compensated back EMF component), $\overline{u}_{dq_{comp}}$ is the feed-forward compensating voltage vector described in (4.16), and $\overline{u}_{dq_{dec}}$ is the resulting decoupled voltage vector to be applied on the motor terminals. Substituting (4.16) into (4.17), and normalizing (4.17), results in the following set of equations:

$$\begin{aligned} u_{df_{dec}} \cdot U_{max} &= u_{df} \cdot U_{max} - \omega_{ef} \cdot \Omega_{max} \cdot L_q \cdot i_{qf} \cdot I_{max} \\ u_{qf_{dec}} \cdot U_{max} &= u_{qf} \cdot U_{max} + \omega_{ef} \cdot \Omega_{max} \cdot L_d \cdot i_{df} \cdot I_{max} \end{aligned} \tag{4.18}$$

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:

$$\begin{aligned} u_{df_{dec}} &= \frac{u_{d_{dec}}}{U_{max}} & u_{qf_{dec}} &= \frac{u_{q_{dec}}}{U_{max}} \\ u_{df} &= \frac{u_d}{U_{max}} & u_{qf} &= \frac{u_q}{U_{max}} \\ i_{df} &= \frac{i_d}{I_{max}} & i_{qf} &= \frac{i_q}{I_{max}} \\ \omega_{ef} &= \frac{\omega_e}{\Omega_{max}} \end{aligned} \tag{4.19}$$

Rearranging (4.18) results in:

$$\begin{aligned} u_{df_{dec}} &= u_{df} - \omega_{ef} \cdot i_{qf} \frac{L_q \cdot \Omega_{max} \cdot I_{max}}{U_{max}} = u_{df} - \omega_{ef} \cdot i_{qf} \cdot k_d \\ u_{qf_{dec}} &= u_{qf} + \omega_{ef} \cdot i_{df} \frac{L_d \cdot \Omega_{max} \cdot I_{max}}{U_{max}} = u_{qf} + \omega_{ef} \cdot i_{df} \cdot k_q \end{aligned} \tag{4.20}$$

where k_d and k_q are coefficients calculated as:

$$\begin{aligned} k_d &= L_q \cdot \Omega_{max} \cdot \frac{I_{max}}{U_{max}} \\ k_q &= L_d \cdot \Omega_{max} \cdot \frac{I_{max}}{U_{max}} \end{aligned} \tag{4.21}$$

Because function GMCLIB_DecouplingPMSM is implemented using the fractional arithmetics, 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:

$$\begin{aligned} k_{d\_shift} &= ceil\left(\frac{\log(k_d)}{\log(2)}\right) \\ k_{q\_shift} &= ceil\left(\frac{\log(k_q)}{\log(2)}\right) \end{aligned} \tag{4.22}$$

Using scaling coefficients (4.22), the fractional representation of coefficients k_d and k_q from (4.21) are derived as follows:

$$\begin{aligned} k_{df} &= k_d \cdot 2^{-k_{d\_shift}} \\ k_{qf} &= k_q \cdot 2^{-k_{q\_shift}} \end{aligned} \tag{4.23}$$

Substituting (4.21) - (4.23) into (4.20) results in the final form of the equation set, which is implemented in the GMCLIB_DecouplingPMSM function:

$$\begin{aligned} u_{df_{dec}} &= u_{df} - \omega_{ef} \cdot i_{qf} \cdot k_{df} \cdot 2^{k_{d\_shift}} \\ u_{qf_{dec}} &= u_{qf} + \omega_{ef} \cdot i_{df} \cdot k_{qf} \cdot 2^{k_{q\_shift}} \end{aligned} \tag{4.24}$$

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.

4.7.6 Note

All parameters can be reset during declaration using the GMCLIB_DECOUPLINGPMSM_-DEFAULT macro.

4.7.7 Reentrancy

The function is reentrant.

4.7.8 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

// Example of calculation of function scale coefficients (remove backslashes
// and copy paste into Matlab)
// L_D            = 50e-3;
// L_Q            = 100e-3;
// U_MAX          = 50;
// I_MAX          = 10;
// W_MAX          = 2000;
// k_d            = (L_Q*W_MAX*I_MAX/U_MAX)
// k_d_shift      = ceil(log(k_d)/log(2))      = 6
// k_df           = k_d * 2^(-k_d_shift)       = 0.625
// k_q            = (L_D*W_MAX*I_MAX/U_MAX)
// k_q_shift      = ceil(log(k_q)/log(2))      = 5
// k_qf           = k_q * 2^(-k_q_shift)       = 0.625

GMCLIB_DECOUPLINGPMSM_T trMyDec = GMCLIB_DECOUPLINGPMSM_DEFAULT;
SWLIBS_2Syst     trUsDQ;
SWLIBS_2Syst     trIsDQ;
SWLIBS_2Syst     trUsDecDQ;
tFrac32           s32We;

void main(void)
{
    // scaling coefficients of given decoupling algorithm
    trMyDec.s32Kd        = FRAC32(0.625);
    trMyDec.s16KdShift  = 6;
    trMyDec.s32Kq        = FRAC32(0.625);
    trMyDec.s16KqShift  = 5;
    trUsDQ.s32Arg1 = FRAC32(5.0/U_MAX);  // d quantity of input voltage vector 5[V]
    trUsDQ.s32Arg2 = FRAC32(10.0/U_MAX); // q quantity of input voltage vector 10[V]
```

```
    trIsDQ.s32Arg1 = FRAC32(6.0/I_MAX);  // d quantity of measured current vector 6[A]
    trIsDQ.s32Arg2 = FRAC32(4.0/I_MAX);  // q quantity of measured current vector 4[A]
    s32We          = FRAC32(100.0/W_MAX) // rotor angular velocity

    // Output should be
    // trUsDecDQ.s32Arg1 = 0xA666668C ~= -35[V]
    // trUsDecDQ.s32Arg1 = 0x66666659 ~=  40[V]
    GMCLIB_DecouplingPMSM(&trUsDecDQ,&trUsDq,&trIsDq,s32We,&trMyDec);
}
```

4.7.9 Performance

Table 4-12: GMCLIB_DecouplingPMSM function performance

Code size [bytes] IAR	196
Data size [bytes] IAR	0
Execution clock cycles max [clk] IAR	68
Execution clock cycles min [clk] IAR	68

4.8 GMCLIB_SvmStd

4.8.1 Declaration

```
tU32 GMCLIB_SvmStdANSIC(SWLIBS_3Syst *pOut, const SWLIBS_2Syst *const pIn)
```

4.8.2 Alias

```
#define GMCLIB_SvmStd(pOutput, pInput) \
        GMCLIB_SvmStdANSIC(pOutput,pInput)
```

4.8.3 Arguments

Table 4-13: Function parameters

Type	Name	Dir.	Description
const SWLIBS_2Syst *const	pIn	in	Pointer to the structure containing direct U_α and quadrature U_β components of the stator voltage vector
SWLIBS_3Syst *	pOut	out	Pointer to the structure containing calculated duty-cycle ratios of the the 3-Phase system

4.8.4 Return

The function returns a 32-bit value in format INT, representing the actual space sector which contains the stator reference vector U_s.

4.8.5 Description

The GMCLIB_SvmStdANSIC function, denoting ANSI-C compatible implementation, can be called via the function alias GMCLIB_SvmStd.

The 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 4-1. Top and bottom switches work in a complementary mode; in other words, 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 for a numerical definition of all possible switching states. In a three-phase power stage configuration (as shown in Figure 4-1), eight possible switching states (Figure 4-2) are feasible.

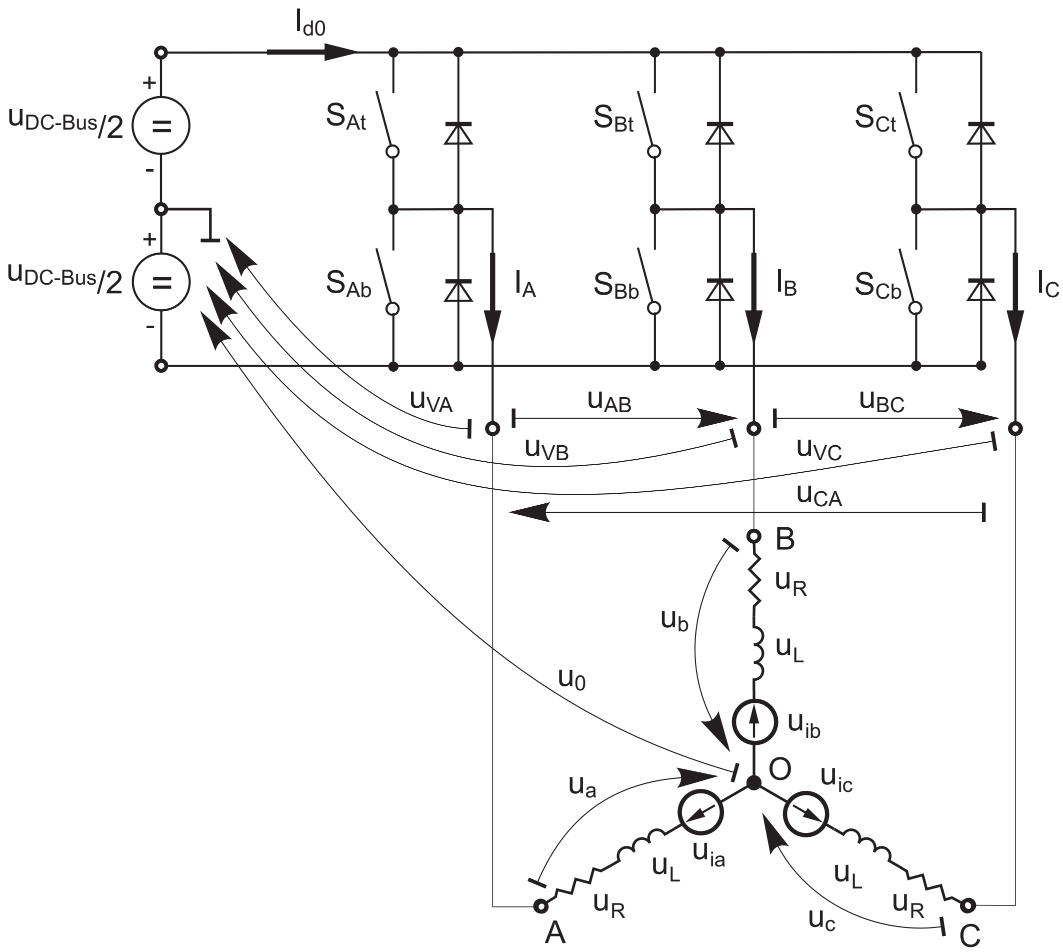


Figure 4-1: Power stage schematic diagram

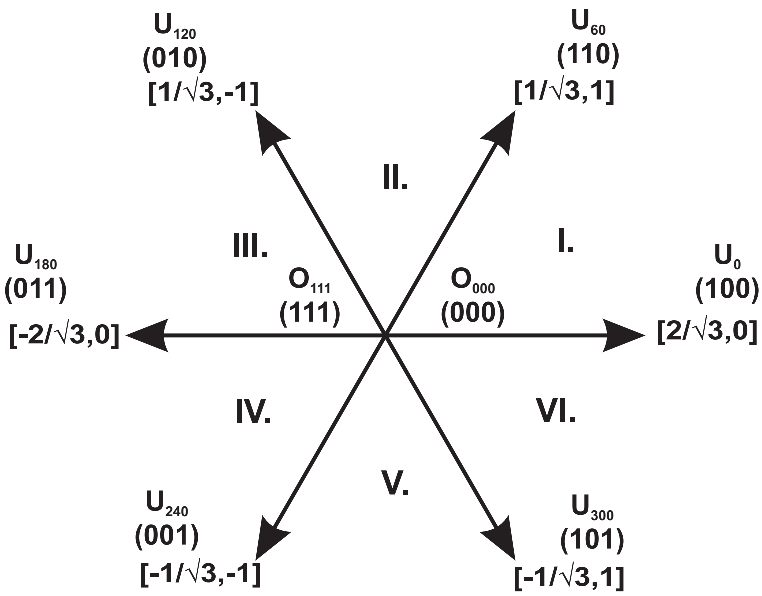


Figure 4-2: Basic space vectors

These states, together with the resulting instantaneous output line-to-line and phase voltages, are listed in Table 4-14.

Table 4-14: 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	$\frac{2}{3}U_{DCBus}$	$-\frac{1}{3}U_{DCBus}$	$-\frac{1}{3}U_{DCBus}$	U_{DCBus}	0	$-U_{DCBus}$	U_0
1	1	0	$\frac{1}{3}U_{DCBus}$	$\frac{1}{3}U_{DCBus}$	$-\frac{2}{3}U_{DCBus}$	0	U_{DCBus}	$-U_{DCBus}$	U_{60}
0	1	0	$-\frac{1}{3}U_{DCBus}$	$\frac{2}{3}U_{DCBus}$	$-\frac{1}{3}U_{DCBus}$	$-U_{DCBus}$	U_{DCBus}	0	U_{120}
0	1	1	$-\frac{2}{3}U_{DCBus}$	$\frac{1}{3}U_{DCBus}$	$\frac{1}{3}U_{DCBus}$	$-U_{DCBus}$	0	U_{DCBus}	U_{240}
0	0	1	$-\frac{1}{3}U_{DCBus}$	$-\frac{1}{3}U_{DCBus}$	$\frac{2}{3}U_{DCBus}$	0	$-U_{DCBus}$	U_{DCBus}	U_{300}
1	0	1	$\frac{1}{3}U_{DCBus}$	$-\frac{2}{3}U_{DCBus}$	$\frac{1}{3}U_{DCBus}$	U_{DCBus}	$-U_{DCBus}$	0	U_{360}
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.

$$U_\alpha = \frac{2}{3}\left(U_a - \frac{U_b}{2} - \frac{U_c}{2}\right) \tag{4.25}$$

$$U_\beta = \frac{2}{3}\left(0 + \frac{\sqrt{3}U_b}{2} - \frac{\sqrt{3}U_c}{2}\right) \tag{4.26}$$

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 4-15.

Figure 4-2 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.

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 Figures 4-3 and 4-4.

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:

$$T = T_{60} + T_0 + T_{null} \tag{4.27}$$

Table 4-15: Switching patterns and space vectors

a	b	c	U_α	U_β	Vector
0	0	0	0	0	O_{000}
1	0	0	$\frac{2}{3}U_{DCBus}$	0	U_0
1	1	0	$\frac{1}{3}U_{DCBus}$	$\frac{1}{\sqrt{3}}U_{DCBus}$	U_{60}
0	1	0	$-\frac{1}{3}U_{DCBus}$	$\frac{1}{\sqrt{3}}U_{DCBus}$	U_{120}
0	1	1	$-\frac{2}{3}U_{DCBus}$	0	U_{240}
0	0	1	$-\frac{1}{3}U_{DCBus}$	$-\frac{1}{\sqrt{3}}U_{DCBus}$	U_{300}
1	0	1	$\frac{1}{3}U_{DCBus}$	$-\frac{1}{\sqrt{3}}U_{DCBus}$	U_{360}
1	1	1	0	0	O_{111}

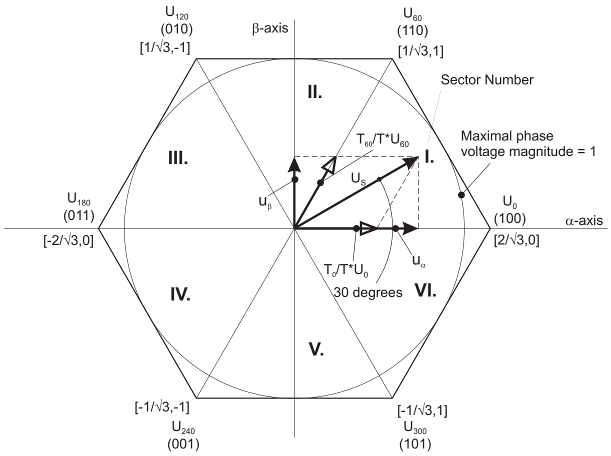


Figure 4-3: Projection of reference voltage vector in Sector I

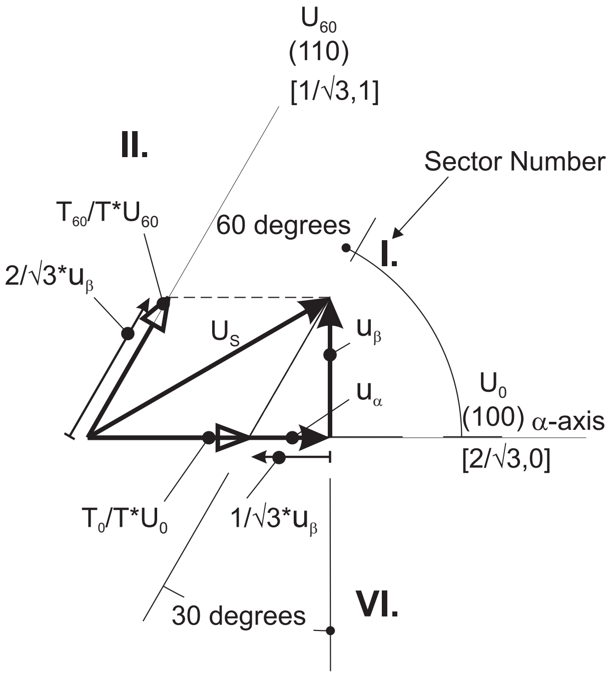


Figure 4-4: Detail of the voltage vector projection in Sector I

$$U_s = \frac{T_{60}}{T} \cdot U_{60} + \frac{T_0}{T} \cdot U_0 \tag{4.28}$$

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:

$$u_\beta = \frac{T_{60}}{T} cdot \left|U_{60}\right| \cdot \sin 60^\circ \tag{4.29}$$

$$u_\alpha = \frac{T_0}{T} \cdot \left|U_0\right| + \frac{u_\beta}{\tan 60^\circ} \tag{4.30}$$

Considering that the normalized magnitudes of the basic space vectors are $|U_{60}| = |U_0| = \frac{2}{\sqrt{3}}$ and by substitution of the trigonometric expressions $\sin 60^\circ$ and $\tan 60^\circ$ by their quantities $\frac{2}{\sqrt{3}}$ and $\sqrt{3}$, respectively, equations (4.29) and (4.30) can be rearranged for the unknown duty-cycle ratios $\frac{T_{60}}{T}$ and $\frac{T_0}{T}$:

$$\frac{T_{60}}{T} = u_\beta \tag{4.31}$$

$$U_S = \frac{T_{120}}{T} \cdot U_{120} + \frac{T_{60}}{T} \cdot U_{60} \tag{4.32}$$

Sector II is depicted in Figure 4-5. 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:

$$T = T_{120} + T_{60} + Tnull \tag{4.33}$$

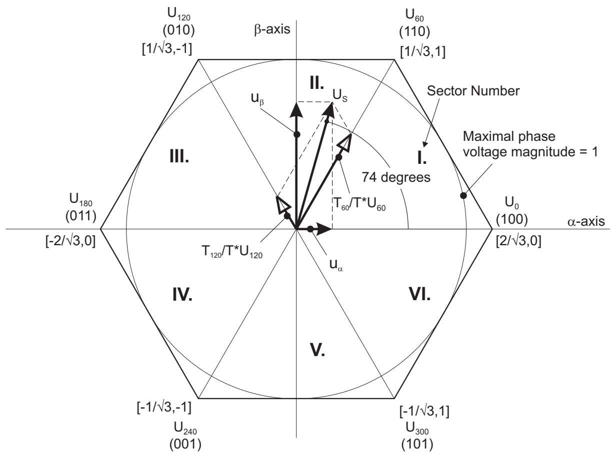


Figure 4-5: Projection of the Reference Voltage Vector in Sector II

$$U_s = \frac{T_{120}}{T} * U_{120} + \frac{T_{60}}{T} * U_{60} \tag{4.34}$$

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 4-6.

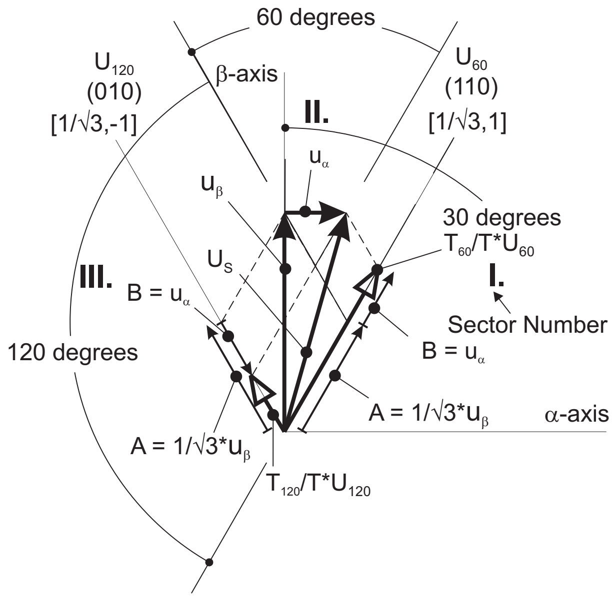


Figure 4-6: Detail of the voltage vector Projection in Sector II

The equations describing those auxiliary time-duration components are:

$$\frac{\sin 30^\circ}{\sin 120^\circ} = \frac{A}{u_\beta} \tag{4.35}$$

$$\frac{\sin 60^\circ}{\sin 60^\circ} = \frac{B}{u_\alpha} \tag{4.36}$$

Equations (4.34) and (4.35) 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 $\sin 30°, \sin 120°$ and $\sin 60°$ by their numerical representations $\frac{1}{2}$, $\frac{\sqrt{3}}{2}$ and $\frac{1}{\sqrt{3}}$, respectively.

$$A = \frac{1}{\sqrt{3}} \cdot u_\beta \tag{4.37}$$

$$B = u_\alpha \tag{4.38}$$

The resultant duty-cycle ratios, $\frac{T_{120}}{T}$ and $\frac{T_{60}}{T}$, are then expressed in terms of the auxiliary time-duration components defined by Equations (4.38) and (4.39), as follows:

$$\frac{T_{120}}{T} \cdot |U_{120}| = A - B \tag{4.39}$$

$$\frac{T_{60}}{T} \cdot |U_{60}| = A + B \tag{4.40}$$

With the help of these equations, and also considering the normalized magnitudes of the basic space vectors to be $|U_{120}| = |U_{60}| = \frac{2}{\sqrt{3}}$, the equations expressed for the unknown duty-cycle ratios of basic space vectors $\frac{T_{120}}{T}$ and $\frac{T_{60}}{T}$ can be written:

$$\frac{T_{120}}{T} = \frac{1}{2}\left(u_\beta - \sqrt{3} \cdot u_\alpha\right) \tag{4.41}$$

$$\frac{T_{60}}{T} = \frac{1}{2}\left(u_\beta + \sqrt{3} \cdot u_\alpha\right) \tag{4.42}$$

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:

$$X = u_\beta \tag{4.43}$$

$$Y = \frac{1}{2} \cdot \left(u_\beta + \sqrt{3} \cdot u_\alpha\right) \tag{4.44}$$

$$Z = \frac{1}{2} \cdot \left(u_\beta - \sqrt{3} \cdot u_\alpha\right) \tag{4.45}$$

Two expressions t_1 and t_2 generally represent duty-cycle ratios of the basic space vectors in the respective sector; for example 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}, and so on. For each sector, the expressions t_1 and t_2, in terms of auxiliary variables X, Y, and Z, are listed in Table 4-16.

The sector number is required for the determination of auxiliary variables X equation (4.43), Y equation (4.44) and Z equation (4.45), 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.

Table 4-16: 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

$$u_{ref1} = u_\beta \tag{4.46}$$

$$u_{ref2} = \frac{1}{2} \cdot \left(-u_\beta + \sqrt{3} \cdot u_\alpha \right) \tag{4.47}$$

$$u_{ref3} = \frac{1}{2} \cdot \left(-u_\beta - \sqrt{3} * u_\alpha \right) \tag{4.48}$$

The modified Inverse Clark Transformation projects the quadrature- u_β component into u_{ref1}, as shown in Figures 4-7 and 4-8, whereas voltages generated by the conventional Inverse Clark Transformation project the u_α component into u_{ref1}.

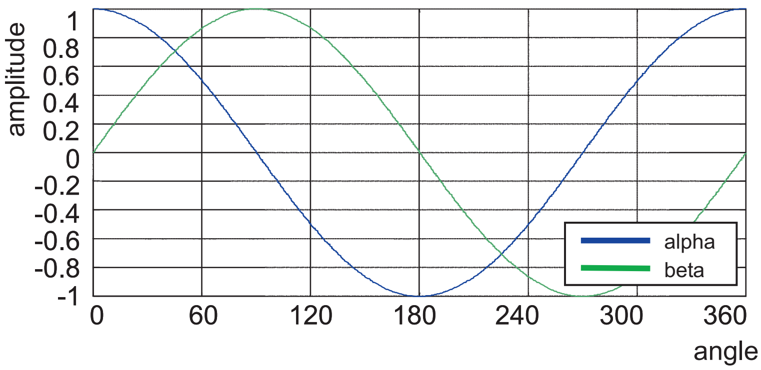


Figure 4-7: Direct-ua and Quadrature-ub components of stator reference voltage

Figure 4-7 depicts the u_α and u_β components of the stator reference voltage vector U_S that were calculated by the equations $u_\alpha = \cos\vartheta$ and $u_\beta = \sin\vartheta$, respectively.

The Sector Identification Tree, shown in Figure 4-9, can be a numerical solution of the approach shown in Figure 4-8.

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 4-3, 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}; see to Figure 4-8. If these quantities are used as the inputs to the Sector Identification Tree, the product of those comparisons will be Sector I. The same approach identifies Sector II if the stator reference voltage vector is located according to the one shown in Figure 4-6. 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:

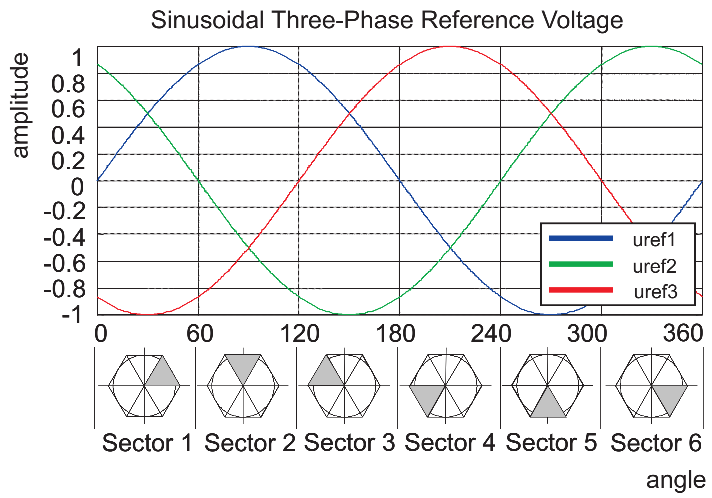


Figure 4-8: Reference voltages u_ref1, u_ref2 and u_ref3

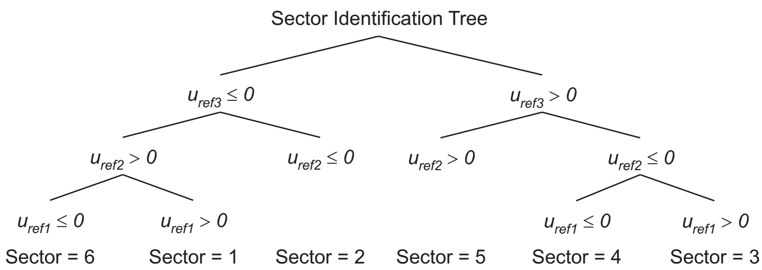


Figure 4-9: Identification of the sector number

$$t_1 = \frac{T - t\_1 - t\_2}{2} \tag{4.49}$$

$$t_2 = t_1 + t\_1 \tag{4.50}$$

$$t_3 = t_2 + t\_2 \tag{4.51}$$

where T is the switching period, t_1 and t_2 are the duty-cycle ratios of the basic space vectors. The vectors are given for the respective sector Equation (4.45), Equation (4.46) and Equation (4.47) 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 $t3$, 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 4-17.

Table 4-17: 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_b	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 center-aligned PWM, edge-aligned PWM or a minimal number of switching states. A brief discussion of the widely-used center-aligned PWM follows. The center-aligned PWM pattern is generated by comparing the threshold levels, pwm_a, pwm_b, and pwm_c with a free-running up-down counter. The timer counts to 1 (0x7FFF) and then down to 0 (0x0000). 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 4-10.

4.8.6 Note

There are several types of Space Vector Modulation Modulations which differ mainly by order of the vectors. This one is the most common.

4.8.7 Reentrancy

The function is reentrant.

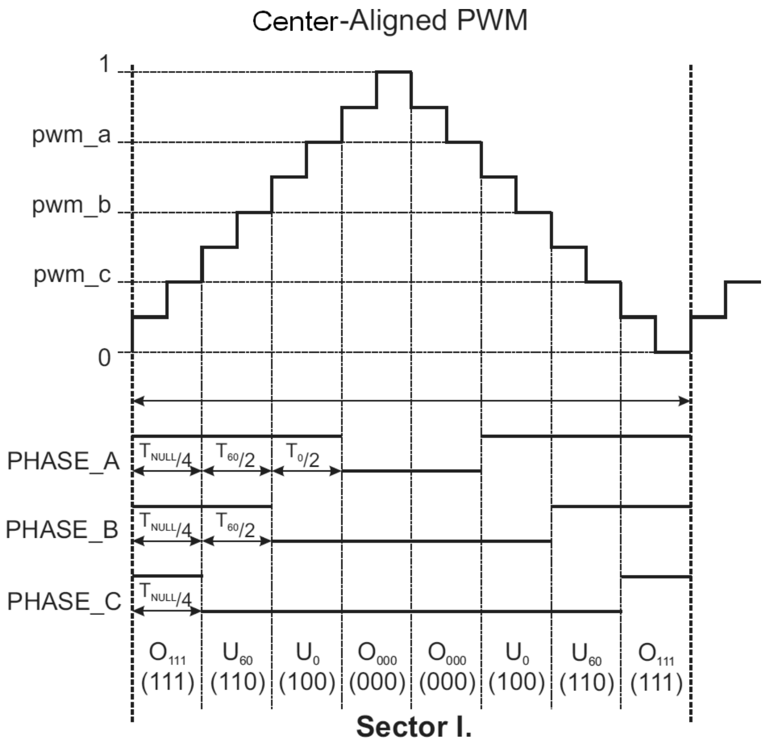


Figure 4-10: Standard Space Vector Modulation technique - center-aligned PWM

4.8.8 Code Example

```
#include "gmclib.h"
#define U_MAX 15

SWLIBS_2Syst tr32InVoltage;
SWLIBS_3Syst 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.s32Arg1       = FRAC32(12.99/U_MAX);
    tr32InVoltage.s32Arg2       = 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(&tr32PwmABC,&tr32InVoltage);
}
```

4.8.9 Performance

Table 4-18: GMCLIB_SvmStd function performance

Code size [bytes] IAR	130
Data size [bytes] IAR	0
Execution clock cycles max [clk] IAR	65
Execution clock cycles min [clk] IAR	59

Chapter 5: GDFLIB

5.1 Function API Overview

- void GDFLIB_FilterIIR1Init (GDFLIB_FILTER_IIR1_T * pParam)
  This function clears internal filter buffers used in function GDFLIB_FilterIIR1
- tFrac32 GDFLIB_FilterIIR1 (tFrac32 s32In, GDFLIB_FILTER_IIR1_T * pParam)
  This function implements a Direct Form I first order IIR filter
- void GDFLIB_FilterIIR2Init (GDFLIB_FILTER_IIR2_T * pParam)
  This function clears internal filter buffers used in function GDFLIB_FilterIIR2
- tFrac32 GDFLIB_FilterIIR2 (tFrac32 s32In, GDFLIB_FILTER_IIR2_T * pParam)
  This function implements a Direct Form I second order IIR filter
- void GDFLIB_FilterFIRInit (const GDFLIB_FILTERFIR_PARAM_T *const pParam, GDFLIB_FILTERFIR_STATE_T *const pState, tFrac32 * ps32InBuf)
  This function performs initialization for the GDFLIB_FilterFIR function
- tFrac32 GDFLIB_FilterFIR (tFrac32 s32In, const GDFLIB_FILTERFIR_PARAM_T *const pParam, GDFLIB_FILTERFIR_STATE_T *const pState)
  This function performs a single iteration of an FIR filter
- void GDFLIB_FilterMAInit (GDFLIB_FILTER_MA_T * pParam)
  This function clears the internal filter accumulator
- tFrac32 GDFLIB_FilterMA (tFrac32 s32In, GDFLIB_FILTER_MA_T * pParam)
  This function implements a moving average recursive filter

5.2 GDFLIB_FilterIIR1Init

5.2.1 Declaration

```
void GDFLIB_FilterIIR1InitANSIC(GDFLIB_FILTER_IIR1_T *pParam)
```

5.2.2 Alias

```
#define GDFLIB_FilterIIR1Init(pParam) \
        GDFLIB_FilterIIR1InitANSIC(pParam)
```

5.2.3 Arguments

Table 5-1: Function parameters

Type	Name	Dir.	Description
GDFLIB_FILTER_IIR1_T *	pParam	in/out	Pointer to filter structure with filter buffer and filter parameters

5.2.4 Return

The function returns data type void.

5.2.5 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.

5.2.6 Note

This function shall not be called together with GDFLIB_FilterIIR1 unless periodic clearing of filter buffers is required.

5.2.7 Reentrancy

The function is reentrant.

5.2.8 Code Example

```
#include "gdflib.h"

tFrac32 s32Input;
tFrac32 s32Output;

GDFLIB_FILTER_IIR1_T trMyIIR1 = GDFLIB_FILTER_IIR1_DEFAULT;
```

```
void main(void)
{
    // input value = 0.25
    s32Input  = FRAC32(0.25);

    // filter coefficients (LPF 100Hz, Ts=100e-6)
    trMyIIR1.trFiltCoeff.s32B0          = FRAC32(0.030468747091254/8);
    trMyIIR1.trFiltCoeff.s32B1          = FRAC32(0.030468747091254/8);
    trMyIIR1.trFiltCoeff.s32A1          = FRAC32(-0.939062505817492/8);
    GDFLIB_FilterIIR1Init(&trMyIIR1);
}
```

5.2.9 Performance

Table 5-2: GDFLIB_FilterIIR1 function performance

Code size [bytes] IAR	80
Data size [bytes] IAR	0
Execution clock cycles max [clk] IAR	37
Execution clock cycles min [clk] IAR	36

5.3 GDFLIB_FilterIIR1

5.3.1 Declaration

```
tFrac32 GDFLIB_FilterIIR1ANSIC(tFrac32 s32In, GDFLIB_FILTER_IIR1_T *pParam)
```

5.3.2 Alias

```
#define GDFLIB_FilterIIR1(s32In, pParam) \
        GDFLIB_FilterIIR1ANSIC(s32In,pParam)
```

5.3.3 Arguments

Table 5-3: Function parameters

Type	Name	Dir.	Description
GDFLIB_FILTER_IIR1_T *	pParam	in/out	Pointer to the filter structure with a filter buffer and filter parameters
tFrac32	s32In	in	Value of input signal to be filtered in step (k). The value is a 32-bit number in the 1.31 fractional format

5.3.4 Return

The function returns a 32-bit value in fractional format 1.31, representing the filtered value of the input signal in step (k).

5.3.5 Description

The GDFLIB_FilterIIR1ANSIC function, denoting ANSI-C compatible implementation, can be called via the function alias GDFLIB_FilterIIR1.

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:

$$H(z) = \frac{Y(z)}{X(z)} = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2} + \cdot + b_N z^{-N}}{1 + a_1 z^{-1} + a_2 z^{-2} + \cdot + a_N z^{-N}} \tag{5.1}$$

where N denotes the filter order. The first order IIR filter in the Z-domain is therefore given from Equation (5.1) as:

$$H(z) = \frac{Y(z)}{X(z)} = \frac{b_0 + b_1 z^{-1}}{1 + a_1 z^{-1}} \tag{5.2}$$

In order to implement the first order IIR filter on a microcontroller, the discrete time domain representation of the filter, described by Equation (5.2), must be transformed into a time difference equation as follows:

$$y(k) = b_0 x(k) + b_1 x(k-1) - a_1 y(k-1) \tag{5.3}$$

Equation (5.3) 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, that is. 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. 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.

Because there are two delay buffers necessary for both DF-I and DF-II implementation of the first order IIR filter, the DF-I implementation was chosen to be used in the GDFLIB_FilterIIR1 function.

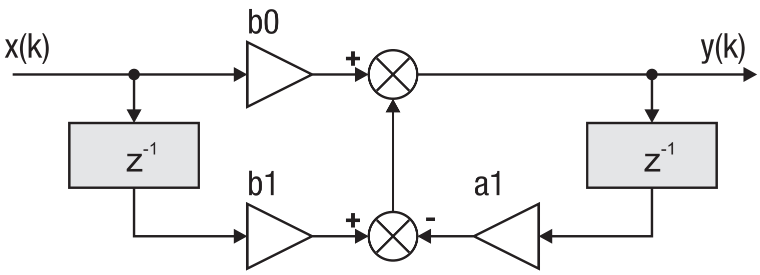


Figure 5-1: Direct Form 1 first order IIR filter

The coefficients of the filter depicted in Figure 5-1 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. In order to avoid overflow during the calculation of the GDFLIB_FilterIIR1 function, filter coefficients must be divided by eight. The coefficient quantization error due to finite precision arithmetic can be neglected in the case of a first order filter. Therefore, the calculation of coefficients can be done 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)

s32B0  = b(1)/8;
s32B1  = b(2)/8;
s32A1  = a(2)/8;
disp('Coefficients for GDFLIB_FilterIIR1 function:');
disp(['s32B0  = FRAC32(' num2str(s32B0) ')']);
disp(['s32B1  = FRAC32(' num2str(s32B1) ')']);
disp(['s32A1  = FRAC32(' num2str(s32A1) ')']);
```

5.3.6 Caution

Because of fixed point implementation, and to avoid overflow during the calculation of the GDFLIB_FilterIIR1 function, filter coefficients must be divided by eight. Function output is internally multiplied by eight to correct the coefficient scaling.

5.3.7 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 macro during instance declaration or by calling the GDFLIB_FilterIIR1Init function.

5.3.8 Reentrancy

The function is reentrant.

5.3.9 Code Example

```
#include "gdflib.h"

tFrac32 s32Input;
tFrac32 s32Output;

GDFLIB_FILTER_IIR1_T trMyIIR1 = GDFLIB_FILTER_IIR1_DEFAULT;

void main(void)
{
    // input value = 0.25
    s32Input  = FRAC32(0.25);

    // filter coefficients (LPF 100Hz, Ts=100e-6)
    trMyIIR1.trFiltCoeff.s32B0          = FRAC32(0.030468747091254/8);
    trMyIIR1.trFiltCoeff.s32B1          = FRAC32(0.030468747091254/8);
    trMyIIR1.trFiltCoeff.s32A1          = FRAC32(-0.939062505817492/8);
    GDFLIB_FilterIIR1Init(&trMyIIR1);

    // output should be 0x00F99998
    s32Output = GDFLIB_FilterIIR1(s32Input,&trMyIIR1);
}
```

5.4 GDFLIB_FilterIIR2Init

5.4.1 Declaration

```
void GDFLIB_FilterIIR2InitANSIC(GDFLIB_FILTER_IIR2_T *pParam)
```

5.4.2 Alias

```
#define GDFLIB_FilterIIR2Init(pParam) \
        GDFLIB_FilterIIR2InitANSIC(pParam)
```

5.4.3 Arguments

Table 5-4: Function parameters

Type	Name	Dir.	Description
GDFLIB_FILTER_IIR2_T *	pParam	in/out	Pointer to the filter structure with a filter buffer and filter parameters

5.4.4 Return

The function returns data type void.

5.4.5 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.

5.4.6 Note

This function shall not be called together with GDFLIB_FilterIIR2 unless periodic clearing of filter buffers is required.

5.4.7 Reentrancy

The function is reentrant.

5.4.8 Code Example

```
#include "gdflib.h"

tFrac32 s32Input;
tFrac32 s32Output;
```

```
GDFLIB_FILTER_IIR2_T trMyIIR2 = GDFLIB_FILTER_IIR2_DEFAULT;

void main(void)
{
    // input value = 0.25
    s32Input   = FRAC32(0.25);

    // filter coefficients (BPF 400-625Hz, Ts=100e-6)
    trMyIIR2.trFiltCoeff.s32B0          = FRAC32(0.066122101544579/8);
    trMyIIR2.trFiltCoeff.s32B1          = FRAC32(0.0);
    trMyIIR2.trFiltCoeff.s32B2          = FRAC32(-0.066122101544579/8);
    trMyIIR2.trFiltCoeff.s32A1          = FRAC32(-1.776189018043779/8);
    trMyIIR2.trFiltCoeff.s32A2          = FRAC32(0.867755796910841/8);
    GDFLIB_FilterIIR2Init(&trMyIIR2);
}
```

5.4.9 Performance

Table 5-5: GDFLIB_FilterIIR2Init function performance

Code size [bytes] IAR	120
Data size [bytes] IAR	0
Execution clock cycles max [clk] IAR	56
Execution clock cycles min [clk] IAR	55

5.5 GDFLIB_FilterIIR2

5.5.1 Declaration

```
tFrac32 GDFLIB_FilterIIR2ANSIC(tFrac32 s32In, GDFLIB_FILTER_IIR2_T *pParam)
```

5.5.2 Alias

```
#define GDFLIB_FilterIIR2(s32In, pParam) \
        GDFLIB_FilterIIR2ANSIC(s32In,pParam)
```

5.5.3 Arguments

Table 5-6: Function parameters

Type	Name	Dir.	Description
GDFLIB_FILTER_IIR2_T *	pParam	in/out	Pointer to the filter structure with a filter buffer and filter parameters
tFrac32	s32In	in	Value of input signal to be filtered in step (k). The value is a 32-bit number in the 1.31 fractional format

5.5.4 Return

The function returns a 32-bit value in fractional format 1.31, representing the filtered value of the input signal in step (k).

5.5.5 Description

The GDFLIB_FilterIIR2ANSIC function, denoting ANSI-C compatible implementation, can be called via the function alias GDFLIB_FilterIIR2.

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:

$$H(z) = \frac{Y(z)}{X(z)} = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2} + \cdot + b_N z^{-N}}{1 + a_1 z^{-1} + a_2 z^{-2} + \cdot + a_N z^{-N}} \tag{5.4}$$

where N denotes the filter order. The second order IIR filter in the Z-domain is therefore given from Equation (5.4) as:

$$H(z) = \frac{Y(z)}{X(z)} = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2}}{1 + a_1 z^{-1} + a_2 z^{-2}} \tag{5.5}$$

In order to implement the second order IIR filter on a microcontroller, the discrete time domain representation of the filter, described by Equation (5.5), must be transformed into a time difference equation as follows:

$$y(k) = b_0x(k) + b_1x(k-1) + b_2x(k-2) - a_1y(k-1) - a_2y(k-2) \tag{5.6}$$

Equation (5.6) 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, that is 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. 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.

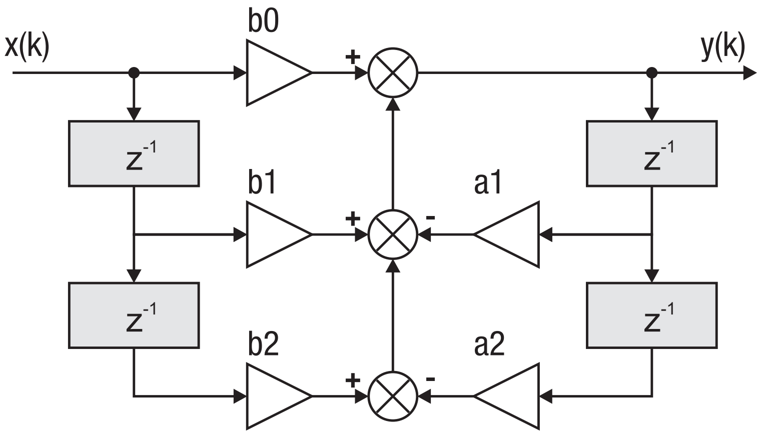


Figure 5-2: Direct Form 1 second order IIR filter

The coefficients of the filter depicted in Figure 5-2 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. In order to avoid overflow during the calculation of the GDFLIB_FilterIIR2 function, filter coefficients must be divided by eight. The coefficient quantization error due to finite precision arithmetic can be neglected in the case of a second order filter. Therefore, calculation of coefficients can be done 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)

s32B0 = b(1)/8;
s32B1 = b(2)/8;
s32B2 = b(3)/8;
```

```
s32A1 = a(2)/8;
s32A2 = a(3)/8;
disp (' Coefficients for GDFLIB_FilterIIR2 function :');
disp ([ 's32B0 = FRAC32(' num2str( s32B0 ) ')']);
disp ([ 's32B1 = FRAC32(' num2str( s32B1 ) ')']);
disp ([ 's32B2 = FRAC32(' num2str( s32B2 ) ')']);
disp ([ 's32A1 = FRAC32(' num2str( s32A1 ) ')']);
disp ([ 's32A2 = FRAC32(' num2str( s32A2 ) ')']);
```

5.5.6 Caution

Because of fixed point implementation, and to avoid overflow during the calculation of the GDFLIB_FilterIIR2 function, filter coefficients must be divided by eight. Function output is internally multiplied by eight to correct the coefficient scaling.

5.5.7 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 macro during instance declaration or by calling the GDFLIB_FilterIIR2Init function.

5.5.8 Reentrancy

The function is reentrant.

5.5.9 Code Example

```
#include "gdflib.h"

tFrac32 s32Input;
tFrac32 s32Output;

GDFLIB_FILTER_IIR2_T trMyIIR2 = GDFLIB_FILTER_IIR2_DEFAULT;

void main(void)
{
    // input value = 0.25
    s32Input  = FRAC32(0.25);

    // filter coefficients (BPF 400-625Hz, Ts=100e-6)
    trMyIIR2.trFiltCoeff.s32B0        = FRAC32(0.066122101544579/8);
    trMyIIR2.trFiltCoeff.s32B1        = FRAC32(0.0);
    trMyIIR2.trFiltCoeff.s32B2        = FRAC32(-0.066122101544579/8);
    trMyIIR2.trFiltCoeff.s32A1        = FRAC32(-1.776189018043779/8);
    trMyIIR2.trFiltCoeff.s32A2        = FRAC32(0.867755796910841/8);
    GDFLIB_FilterIIR2Init(&trMyIIR2);

    // output should be 0x0021DAC18
    s32Output = GDFLIB_FilterIIR2(s32Input,&trMyIIR2);
}
```

5.6 GDFLIB_FilterFIRInit

5.6.1 Declaration

```
void GDFLIB_FilterFIRInitANSIC(const GDFLIB_FILTERFIR_PARAM_T *const pParam,
GDFLIB_FILTERFIR_STATE_T *const pState, tFrac32 *ps32InBuf)
```

5.6.2 Alias

```
#define GDFLIB_FilterFIRInit(pParam, pState, pInBuf) \
        GDFLIB_FilterFIRInitANSIC((pParam), (pState), (pInBuf))
```

5.6.3 Arguments

Table 5-7: Function parameters

Type	Name	Dir.	Description
const GDFLIB_FILTERFIR_PARAM_T *const	pParam	in	Pointer to the parameters structure.
GDFLIB_FILTERFIR_STATE_T *const	pState	out	Pointer to the state structure.
tFrac32 *	ps32InBuf	in/out	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.

5.6.4 Return

The function returns data type void.

5.6.5 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.

5.6.6 Caution

No check is performed for R/W capability and the length of the input buffer (`pState->ps32InBuf`).

5.6.7 Note

The input buffer pointer (`State->ps32InBuf`) must point to a Read/Write memory region, which must be at least as long as the number of filter taps. 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.

5.6.8 Reentrancy

The function is reentrant only if the calling code is provided with a distinct instance of the structure pointed to by `pState`.

5.6.9 Code Example

```
#include "gdflib.h"

#define FIR_NUMTAPS 16
#define FIR_NUMTAPS_MAX 64
#define FIR_ORDER (FIR_NUMTAPS - 1)

GDFLIB_FILTERFIR_PARAM_T Param;
GDFLIB_FILTERFIR_STATE_T State;

tFrac32 as32InBuf[FIR_NUMTAPS_MAX];
tFrac32 as32CoefBuf[FIR_NUMTAPS_MAX];

#define OUT_LEN 16

void main(void)
{
    int ii;
    tFrac32 as32OutBuf[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;
    as32CoefBuf[ii++] = 0xFFB10C14;
    as32CoefBuf[ii++] = 0xFF779D25;
    as32CoefBuf[ii++] = 0x01387DD7;
```

```
    as32CoefBuf[ii++] = 0x028E6845;
    as32CoefBuf[ii++] = 0xFB245142;
    as32CoefBuf[ii++] = 0xF7183CC7;
    as32CoefBuf[ii++] = 0x11950A3C;
    as32CoefBuf[ii++] = 0x393ED867;
    as32CoefBuf[ii++] = 0x393ED867;
    as32CoefBuf[ii++] = 0x11950A3C;
    as32CoefBuf[ii++] = 0xF7183CC7;
    as32CoefBuf[ii++] = 0xFB245142;
    as32CoefBuf[ii++] = 0x028E6845;
    as32CoefBuf[ii++] = 0x01387DD7;
    as32CoefBuf[ii++] = 0xFF779D25;
    as32CoefBuf[ii++] = 0xFFB10C14;

    Param.u32Order = 15;
    Param.ps32CoefBuf = &as32CoefBuf[0];

    // Initialize FIR filter
    GDFLIB_FilterFIRInit(&Param, &State, &as32InBuf[0]);

    // Compute step response of the filter
    for(ii=0; ii < OUT_LEN; ii++)
    {
        as32OutBuf[ii] = GDFLIB_FilterFIR(FRAC32(1.0), &Param, &State);
    }

    // as32Out contains step response of the filter

    return;
}
```

5.6.10 Performance

Table 5-8: GDFLIB_FilterFIRInit function performance

Code size [bytes] IAR	80
Data size [bytes] IAR	0/0
Execution clock cycles max [clk] IAR	37
Execution clock cycles min [clk] IAR	36

5.7 GDFLIB_FilterFIR

5.7.1 Declaration

```
tFrac32 GDFLIB_FilterFIRANSIC(tFrac32 s32In, const GDFLIB_FILTERFIR_PARAM_T
*const pParam, GDFLIB_FILTERFIR_STATE_T *const pState)
```

5.7.2 Alias

```
#define GDFLIB_FilterFIR(s32In, pParam, pState) \
        GDFLIB_FilterFIRANSIC((s32In), (pParam), (pState))
```

5.7.3 Arguments

Table 5-9: Function parameters

Type	Name	Dir.	Description
tFrac32	s32In	in	Input value.
const GDFLIB_FILTERFIR_PARAM_T *const	pParam	in	Pointer to the parameter structure.
GDFLIB_FILTERFIR_STATE_T *const	pState	in/out	Pointer to the filter state structure.

5.7.4 Return

The value of a filtered signal after processing by an FIR filter.

5.7.5 Description

The function performs the operation of an FIR filter on a sample-by-sample basis. At each new input to the FIR filter, the function should be called, which will return a new filtered value. The FIR filter is defined by the following formula:

$$y[n] = h_0x[n] + h_1x[n-1] + \cdots + h_Nx[n-N] \tag{5.7}$$

where: $x[n]$ is the input signal, $y[n]$ is the output signal, h_i are the filter coefficients, and N is the filter order. The number of taps of the filter is $N+1$ in this case. The multiply and accumulate operations are performed with 32 accumulation guard bits present, which means that no saturation is performed during computations. However, if the final value cannot fit in the return data type, saturation may occur. It should be noted, although rather theoretically, that no saturation is performed on the accumulation guard bits and an overflow over the accumulation guard bits may occur. The function assumes that the filter order is at least one, which is equivalent to two taps. The filter also cannot contain more than ffffffff (hexadecimal) taps, which is equivalent to the order of fffffffe (hexadecimal). The input values are recorded by the function in the provided state structure in a circular buffer, pointed to by the state structure member `pState->ps32InBuf`. The buffer index is stored in `pState->u32Idx`, which points to the buffer element where a new input signal sample will be stored. The filter coefficients are stored in the parameter structure, in the structure member `pParam->ps32CoefBuf`. The first call to the function must be preceded by an initialization, which can be made through the GDFLIB_FilterFIRInit function. The GDFLIB_FilterFIRInit and then the GDFLIB_FilterFIR functions should be called with the same parameters.

5.7.6 Caution

The count of maximal clock cycles depends on the selected order of the filter.

5.7.7 Note

From the performance point of view, the function is designed to work with filters with a larger number of taps (equal order + 1). As a rule of thumb, if the number of taps is lower than 5, a different algorithm should be considered.

5.7.8 Reentrancy

The function is reentrant only if the calling code is provided with a distinct instance of the structure pointed to by `pState`.

5.7.9 Code Example

```
#include "gdflib.h"

#define FIR_NUMTAPS 16
#define FIR_NUMTAPS_MAX 64
#define FIR_ORDER (FIR_NUMTAPS - 1)

GDFLIB_FILTERFIR_PARAM_T Param;
GDFLIB_FILTERFIR_STATE_T State;

tFrac32 as32InBuf[FIR_NUMTAPS_MAX];
tFrac32 as32CoefBuf[FIR_NUMTAPS_MAX];

#define OUT_LEN 16

void main(void)
{
    int ii;
    tFrac32 as32OutBuf[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;
    as32CoefBuf[ii++] = 0xFFB10C14;
    as32CoefBuf[ii++] = 0xFF779D25;
    as32CoefBuf[ii++] = 0x01387DD7;
    as32CoefBuf[ii++] = 0x028E6845;
    as32CoefBuf[ii++] = 0xFB245142;
    as32CoefBuf[ii++] = 0xF7183CC7;
    as32CoefBuf[ii++] = 0x11950A3C;
    as32CoefBuf[ii++] = 0x393ED867;
    as32CoefBuf[ii++] = 0x393ED867;
    as32CoefBuf[ii++] = 0x11950A3C;
    as32CoefBuf[ii++] = 0xF7183CC7;
    as32CoefBuf[ii++] = 0xFB245142;
    as32CoefBuf[ii++] = 0x028E6845;
    as32CoefBuf[ii++] = 0x01387DD7;
    as32CoefBuf[ii++] = 0xFF779D25;
    as32CoefBuf[ii++] = 0xFFB10C14;

    Param.u32Order = 15;
    Param.ps32CoefBuf = &as32CoefBuf[0];

    // Initialize FIR filter
    GDFLIB_FilterFIRInit(&Param, &State, &as32InBuf[0]);

    // Compute step response of the filter
    for(ii=0; ii < OUT_LEN; ii++)
    {
        as32OutBuf[ii] = GDFLIB_FilterFIR(FRAC32(1.0), &Param, &State);
    }

    // as32Out contains step response of the filter

    return;
}
```

5.8 GDFLIB_FilterMAInit

5.8.1 Declaration

```
void GDFLIB_FilterMAInitANSIC(GDFLIB_FILTER_MA_T *pParam)
```

5.8.2 Alias

```
#define GDFLIB_FilterMAInit(pParam) \
        GDFLIB_FilterMAInitANSIC(pParam)
```

5.8.3 Arguments

Table 5-10: Function parameters

Type	Name	Dir.	Description
GDFLIB_FILTER_MA_T *	pParam	in/out	Pointer to the filter structure with a filter accumulator and filter parameters

5.8.4 Return

The function returns data type void.

5.8.5 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.

The size of the filter window (number of filtered points) shall be defined prior to this function call. The number of the filtered points is defined by assigning a value to the u16NSamples variable stored within the filter structure. This number represents the number of filtered points as a power of 2 as follows:

$$n_p = 2^{u16NSamples} \qquad 0 \leq u16NSamples \leq 31 \tag{5.8}$$

5.8.6 Note

This function shall not be called together with GDFLIB_FilterMA unless periodic clearing of filter buffers is required.

5.8.7 Reentrancy

The function is reentrant.

5.8.8 Code Example

```
#include "gdflib.h"

GDFLIB_FILTER_MA_T trMyMA = GDFLIB_FILTER_MA_DEFAULT;

void main(void)
{
    GDFLIB_FilterMAInit(&trMyMA);
}
```

5.8.9 Performance

Table 5-11: GDFLIB_FilterMAInit function performance

Code size [bytes] IAR	2/6
Data size [bytes] IAR	0/0
Execution clock cycles max [clk] IAR	10/10
Execution clock cycles min [clk] IAR	9/8

5.9 GDFLIB_FilterMA

5.9.1 Declaration

```
tFrac32 GDFLIB_FilterMAANSIC(tFrac32 s32In, GDFLIB_FILTER_MA_T *pParam)
```

5.9.2 Alias

```
#define GDFLIB_FilterMA(s32In, pParam) \
        GDFLIB_FilterMAANSIC(s32In, pParam)
```

5.9.3 Arguments

Table 5-12: Function parameters

Type	Name	Dir.	Description
tFrac32	s32In	in	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 *	pParam	in/out	Pointer to the filter structure with a filter accumulator and filter parameters.

5.9.4 Return

The function returns a 32-bit value in format Q1.31, representing the filtered value of the input signal in step (k).

5.9.5 Description

The GDFLIB_FilterMAANSIC function, denoting ANSI-C compatible implementation, can be called via the function alias GDFLIB_FilterMA.

This function calculates a recursive form of an average filter. The filter calculation consists of the following equations:

$$acc(k) = acc(k-1) + x(k) \tag{5.9}$$

$$y(k) = \frac{acc(k)}{n_p} \tag{5.10}$$

$$acc(k) \leftarrow acc(k) - y(k) \tag{5.11}$$

where x(k) is the actual value of the input signal, acc(k) is the internal filter accumulator, y(k) is the actual filter output and n_p is the number of points in the filtered window. The size of the filter window (number of filtered points) shall be defined prior to this function call. The number of the filtered points is defined by assigning a value to the u16NSamples variable stored within the filter structure. This number represents the number of filtered points as a power of 2 as follows:

$$n_p = 2^{u16NSamples} \qquad 0 \leq u16NSamples \leq 31 \tag{5.12}$$

5.9.6 Note

The size of the filter window (number of filtered points) must be defined prior to this function call and must be equal to or greater than 0, and equal to or smaller than 31 $(0 \leq u16NSamples \leq 31)$.

5.9.7 Reentrancy

The function is reentrant.

5.9.8 Code Example

```
#include "gdflib.h"

tFrac32 s32Input;
tFrac32 s32Output;

GDFLIB_FILTER_MA_T trMyMA = GDFLIB_FILTER_MA_DEFAULT;

void main(void)
{
    // input value = 0.25
    s32Input  = FRAC32(0.25);

    // filter window = 2^5 = 32 samples
    trMyMA.u16NSamples   = 5;
    GDFLIB_FilterMAInit(&trMyMA);

    // output should be 0x1000000 = FRAC32(0.0078125)
    s32Output = GDFLIB_FilterMA(s32Input,&trMyMA);
}
```

5.9.9 Performance

Table 5-13: GDFLIB_FilterMA function performance

Code size [bytes] IAR	80
Data size [bytes] IAR	0
Execution clock cycles max [clk] IAR	20
Execution clock cycles min [clk] IAR	20

CHAPTER 6: DATA TYPES

6.1 Defined in <SWLIBS_Typedefs.h>

Table 6-1: Data types description

Type	Name	Description
typedef unsigned char	tBool	basic boolean type
typedef unsigned char	tU8	unsigned 8-bit integer type
typedef signed char	tS8	signed 8-bit integer type
typedef unsigned short	tU16	unsigned 16-bit integer type
typedef signed short	tS16	signed 16-bit integer type
typedef unsigned int	tU32	unsigned 32-bit integer type
typedef signed int	tS32	signed 32-bit integer type
typedef unsigned long long	tU64	unsigned 64-bit integer type
typedef signed long long	tS64	signed 64-bit integer type
typedef tS16	tFrac16	16-bit signed fractional Q1.15 type
typedef tS32	tFrac32	32-bit signed fractional Q1.31 type

CHAPTER 7: COMPOUND DATA TYPES

7.0.1 Compound Data Types Overview

- FILTER_IIR1_COEFF_T

  Substructure containing filter coefficients

- FILTER_IIR2_COEFF_T

  Substructure containing filter coefficients

- GDFLIB_FILTER_IIR1_T

  Structure containing filter buffer and coefficients

- GDFLIB_FILTER_IIR2_T

  Structure containing filter buffer and coefficients

- GDFLIB_FILTER_MA_T

  Structure containing filter buffer and coefficients

- GDFLIB_FILTERFIR_PARAM_T

  Structure containing parameters of the filter

- GDFLIB_FILTERFIR_STATE_T

  Structure containing the current state of the filter

- GFLIB_ACOS_TAYLOR_COEF_T

  Structure containing five polynomial coefficients for one subinterval

- GFLIB_ACOS_TAYLOR_T

  Structure containing two substructures with polynomial coefficients to cover all subintervals

- GFLIB_ASIN_TAYLOR_COEF_T

  Structure containing five polynomial coefficients for one subinterval

- GFLIB_ASIN_TAYLOR_T

  Structure containing two substructures with polynomial coefficients to cover all subintervals

- GFLIB_ATAN_TAYLOR_COEF_T

  Structure containing four polynomial coefficients for one subinterval

- GFLIB_ATAN_TAYLOR_T

  Structure containing eight substructures with polynomial coefficients to cover all subintervals

- GFLIB_ATANYXSHIFTED_T

  Structure containing the parameter for the GFLIB_AtanYXShifted function

- GFLIB_CONTROLLER_PI_P_T

  Structure containing parameters and states of the the parallel form PI controller

- GFLIB_CONTROLLER_PI_R_T

  Structure containing parameters and states of the recurrent form PI controller

- GFLIB_CONTROLLER_PIAW_P_T

  Structure containing parameters and states of the the parallel form PI controller with anti-windup

- GFLIB_COSTLR_T

  Structure containing one array of five 32-bit elements for storing coefficients of a Taylor polynomial

- GFLIB_HYST_T

  Structure containing parameters and states for the hysteresis function implemented in GFLIB_Hyst

- GFLIB_INTEGRATOR_TR_T

  Structure containing integrator parameters and coefficients

- GFLIB_LIMIT_T

  Structure containing the limits

- GFLIB_LOWERLIMIT_T

  Structure containing the lower limit

- GFLIB_LUT1D_T

  Structure containing look-up table parameters

- GFLIB_RAMP_T

  Structure containing controller parameters and coefficients

- GFLIB_SINTLR_T

  Structure containing one array of five 32-bit elements for storing coefficients of a Taylor polynomial

- GFLIB_TAN_TAYLOR_COEF_T

  Structure containing four polynomial coefficients for one subinterval

- GFLIB_TANTLR_T

  Structure containing eight substructures with polynomial coefficients to cover all subintervals

- GFLIB_UPPERLIMIT_T

  Structure containing the upper limit

- GFLIB_VECTORLIMIT_T

  Structure containing the limits

- GMCLIB_DECOUPLINGPMSM_T

  Structure containing coefficients for calculation of the decoupling algorithm implemented in the GMCLIB_DecouplingPMSM function

- GMCLIB_ELIMDCBUSRIP_T

  Structure containing the PWM modulation index and the measured value of the DC bus voltage for calculation of the DC bus ripple elimination

- SWLIBS_2Syst

  Structure data type for two axis input/output variables

- SWLIBS_3Syst

  Structure data type for three axis input/output variables

7.1 FILTER_IIR1_COEFF_T

`#include <GDFLIB_FilterIIR1.h>`

7.1.1 Description

Substructure containing filter coefficients.

7.1.2 Compound Type Members

Table 7-1: Members description

Type	Name	Description
tFrac32	s32B0	b0 coefficient of an IIR1 filter, 32-bit
tFrac32	s32B1	b1 coefficient of an IIR1 filter, 32-bit
tFrac32	s32A1	a1 coefficient of an IIR1 filter, 32-bit

7.2 FILTER_IIR2_COEFF_T

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

7.2.1 Description

Substructure containing filter coefficients.

7.2.2 Compound Type Members

Table 7-2: Members description

Type	Name	Description
tFrac32	s32B0	b0 coefficient of an IIR2 filter, 32-bit
tFrac32	s32B1	b1 coefficient of an IIR2 filter, 32-bit
tFrac32	s32B2	b2 coefficient of an IIR2 filter, 32-bit
tFrac32	s32A1	a1 coefficient of an IIR2 filter, 32-bit
tFrac32	s32A2	a2 coefficient of an IIR2 filter, 32-bit

7.3 GDFLIB_FILTER_IIR1_T

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

7.3.1 Description

Structure containing filter buffer and coefficients.

7.3.2 Compound Type Members

Table 7-3: Members description

Type	Name	Description
FILTER_IIR1_-COEFF_T	trFiltCoeff	filter coefficients substructure
tFrac32	s32FiltBufferX[2]	input buffer of an IIR1 filter
tFrac32	s32FiltBufferY[2]	internal accumulator buffer

7.4 GDFLIB_FILTER_IIR2_T

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

7.4.1 Description

Structure containing filter buffer and coefficients.

7.4.2 Compound Type Members

Table 7-4: Members description

Type	Name	Description
FILTER_IIR2_-COEFF_T	trFiltCoeff	filter coefficients substructure
tFrac32	s32FiltBufferX[3]	input buffer of an IIR2 filter
tFrac32	s32FiltBufferY[3]	internal accumulator buffer

7.5 GDFLIB_FILTER_MA_T

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

7.5.1 Description

Structure containing filter buffer and coefficients.

7.5.2 Compound Type Members

Table 7-5: Members description

Type	Name	Description
tFrac32	s32Acc	filter accumulator
tU16	u16NSamples	number of samples for averaging, filter sample window [0,31]

7.6 GDFLIB_FILTERFIR_PARAM_T

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

7.6.1 Description

Structure containing parameters of the filter.

7.6.2 Compound Type Members

Table 7-6: Members description

Type	Name	Description
tU32	u32Order	FIR filter order, must be 1 or more.
const tFrac32 *	ps32CoefBuf	FIR filter coefficients buffer.

7.7 GDFLIB_FILTERFIR_STATE_T

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

7.7.1 Description

Structure containing the current state of the filter.

7.7.2 Compound Type Members

Table 7-7: Members description

Type	Name	Description
tU32	u32Idx	Input buffer index.
tFrac32 *	ps32InBuf	Pointer to the input buffer.

7.8 GFLIB_ACOS_TAYLOR_COEF_T

```
#include <GFLIB_Acos.h>
```

7.8.1 Description

Structure containing five polynomial coefficients for one subinterval. Output of $\arccos(s32In)$ for interval $[0, 1)$ of the input ratio is divided into two subsectors. Polynomial approximation is done using a 5th order polynomial for each subsector respectively. Five coefficients for a single subinterval are stored in this GFLIB_ACOS_TAYLOR_COEF_T structure.

7.8.2 Compound Type Members

Table 7-8: Members description

Type	Name	Description
const tFrac32	s32a[5]	Array of five 32-bit elements for storing coefficients of the piece-wise polynomial.

7.9 GFLIB_ACOS_TAYLOR_T

```
#include <GFLIB_Acos.h>
```

7.9.1 Description

Structure containing two substructures with polynomial coefficients to cover all subintervals. Output of $\arccos(s32In)$ for interval $[0, 1)$ of the input ratio is divided into two subsectors. Polynomial approximation is done using a 5th order polynomial, for each subsector respectively. Two arrays, each including five polynomial coefficients for each subinterval, are stored in this GFLIB_ACOS_TAYLOR_T structure.

By calling the function alias GFLIB_Acos, default values of the coefficients are used. Polynomial coefficients can be modified by the user and in such a case the full function call shall be used, that is GFLIB_AcosANSIC.

7.9.2 Compound Type Members

Table 7-9: Members description

Type	Name	Description
const GFLIB_ACOS_TAYLOR_COEF_T	GFLIB_ACOS_SECTOR[2]	Array of two elements for storing two subarrays (each subarray contains five 32-bit coefficients) for all subintervals.

7.10 GFLIB_ASIN_TAYLOR_COEF_T

```
#include <GFLIB_Asin.h>
```

7.10.1 Description

Structure containing five polynomial coefficients for one subinterval. Output of $\arcsin(s32In)$ for interval $[0, 1)$ of the input ratio is divided into two subsectors. Polynomial approximation is done using a 5th order polynomial for each subsector respectively. Five coefficients for a single subinterval are stored in this GFLIB_ASIN_TAYLOR_COEF_T structure.

7.10.2 Compound Type Members

Table 7-10: Members description

Type	Name	Description
const tFrac32	s32a[5]	Array of five 32-bit elements for storing coefficients of the piece-wise polynomial.

7.11 GFLIB_ASIN_TAYLOR_T

```
#include <GFLIB_Asin.h>
```

7.11.1 Description

Structure containing two substructures with polynomial coefficients to cover all subintervals. Output of $\arcsin(s32In)$ for interval $[0,1)$ of the input ratio is divided into two subsectors. Polynomial approximation is done using a 5th order polynomial, for each subsector respectively. Two arrays, each including five polynomial coefficients for each subinterval, are stored in this GFLIB_ASIN_TAYLOR_T structure.

By calling the function alias GFLIB_Asin, default values of the coefficients are used. Polynomial coefficients can be modified by the user and in such a case the full function call shall be used, that is GFLIB_AsinANSIC.

7.11.2 Compound Type Members

Table 7-11: Members description

Type	Name	Description
const GFLIB_ASIN_TAYLOR_COEF_T	GFLIB_ASIN_SECTOR[2]	Array of two elements for storing eight subarrays (each subarray contains four 32-bit coefficients) for all subintervals.

7.12 GFLIB_ATAN_TAYLOR_COEF_T

`#include <GFLIB_Atan.h>`

7.12.1 Description

Structure containing four polynomial coefficients for one subinterval. Output of $\arctan(s32In)$ for interval $[0,1)$ of the input ratio is divided into eight subsectors. Polynomial approximation is done using a 4th order polynomial, for each subsector respectively. Four coefficients for a single subinterval are stored in this GFLIB_ATAN_TAYLOR_COEF_T structure.

7.12.2 Compound Type Members

Table 7-12: Members description

Type	Name	Description
const tFrac32	s32a[4]	Array of five 32-bit elements for storing coefficients of the piece-wise polynomial.

7.13 GFLIB_ATAN_TAYLOR_T

```
#include <GFLIB_Atan.h>
```

7.13.1 Description

Structure containing eight substructures with polynomial coefficients to cover all subintervals. Output of $\arctan(s32In)$ for interval $[0, 1)$ of the input ratio is divided into eight subsectors. Polynomial approximation is done using a 4th order polynomial, for each subsector respectively. Eight arrays, each including four polynomial coefficients for each subinterval, are stored in this GFLIB_ATAN_TAYLOR_COEF_T structure.

By the calling function alias GFLIB_Atan, default values of the coefficients are used. Polynomial coefficients can be modified by the user and in such a case the full function call shall be used, that is GFLIB_AtanANSIC.

7.13.2 Compound Type Members

Table 7-13: Members description

Type	Name	Description
const GFLIB_ATAN_-TAYLOR_COEF_T	GFLIB_ATAN_SECTOR[8]	Array of eight elements for storing eight subarrays (each subarray contains four 32-bit coefficients) for all subintervals.

7.14 GFLIB_ATANYXSHIFTED_T

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

7.14.1 Description

Structure containing the parameter for the GFLIB_AtanYXShifted function.

7.14.2 Compound Type Members

Table 7-14: Members description

Type	Name	Description
tFrac32	s32Ky	Multiplication coefficient for the y-signal.
tFrac32	s32Kx	Multiplication coefficient for the x-signal.
tS32	s32Ny	Scaling coefficient for the y-signal.
tS32	s32Nx	Scaling coefficient for the x-signal.
tFrac32	s32ThetaAdj	Adjusting angle.

7.15 GFLIB_CONTROLLER_PI_P_T

```
#include <GFLIB_ControllerPIp.h>
```

7.15.1 Description

Structure containing parameters and states of the the parallel form PI controller.

7.15.2 Compound Type Members

Table 7-15: Members description

Type	Name	Description
tFrac32	s32PropGain	Proportional Gain, fractional format normalized to fit into $\left[-2^{31}, 2^{31}-1\right)$
tFrac32	s32IntegGain	Integral Gain, fractional format normalized to fit into $\left[-2^{31}, 2^{31}-1\right)$
tS16	s16PropGainShift	Proportional Gain Shift, integer format $[-31, 31]$
tS16	s16IntegGainShift	Integral Gain Shift, integer format $[-31, 31]$
tFrac32	s32IntegPartK_1	State variable integral part at step k-1
tFrac32	s32InK_1	State variable input error at step k-1

7.16 GFLIB_CONTROLLER_PI_R_T

#include <GFLIB_ControllerPIr.h>

7.16.1 Description

Structure containing parameters and states of the recurrent form PI controller.

7.16.2 Compound Type Members

Table 7-16: Members description

Type	Name	Description
tFrac32	s32CC1sc	CC1 coefficient, fractional format normalized to fit into $\left[-2^{31}, 2^{31}-1\right)$
tFrac32	s32CC2sc	CC2 coefficient, fractional format normalized to fit into $\left[-2^{31}, 2^{31}-1\right)$
tFrac32	s32Acc	Internal controller accumulator
tFrac32	s32InErrK1	Controller input from the previous calculation step
tU16	u16NShift	Scaling factor for controller coefficients, integer format $[0, 31]$

7.17 GFLIB_CONTROLLER_PIAW_P_T

```
#include <GFLIB_ControllerPIpAW.h>
```

7.17.1 Description

Structure containing parameters and states of the the parallel form PI controller with anti-windup.

7.17.2 Compound Type Members

Table 7-17: Members description

Type	Name	Description
tFrac32	s32PropGain	Proportional Gain, fractional format normalized to fit into $\left[-2^{31}, 2^{31}-1\right)$
tFrac32	s32IntegGain	Integral Gain, fractional format normalized to fit into $\left[-2^{31}, 2^{31}-1\right)$
tS16	s16PropGainShift	Proportional Gain Shift, integer format $[-31, 31]$
tS16	s16IntegGainShift	Integral Gain Shift, integer format $[-31, 31]$
tFrac32	s32LowerLimit	Lower Limit of the controller, fractional format normalized to fit into $\left[-2^{31}, 2^{31}-1\right)$
tFrac32	s32UpperLimit	Upper Limit of the controller, fractional format normalized to fit into $\left[-2^{31}, 2^{31}-1\right)$
tFrac32	s32IntegPartK_1	State variable integral part at step k-1
tFrac32	s32InK_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

7.18 GFLIB_COSTLR_T

```
#include <GFLIB_Cos.h>
```

7.18.1 Description

Structure containing one array of five 32-bit elements for storing coefficients of a Taylor polynomial. By calling the function alias GFLIB_Cos, default values of the coefficients are used. Polynomial coefficients can be modified by the user and in such a case the full function call shall be used, that is GFLIB_CosANSIC.

7.18.2 Compound Type Members

Table 7-18: Members description

Type	Name	Description
tFrac32	s32A[5]	Array of five 32-bit elements for storing coefficients of the Taylor polynomial.

7.19 GFLIB_HYST_T

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

7.19.1 Description

Structure containing parameters and states for the hysteresis function implemented in GFLIB_Hyst.

7.19.2 Compound Type Members

Table 7-19: Members description

Type	Name	Description
tFrac32	s32HystOn	Value determining the upper threshold; fractional format normalized to fit into $\left[-2^{31}, 2^{31}-1\right)$.
tFrac32	s32HystOff	Value determining the lower threshold; fractional format normalized to fit into $\left[-2^{31}, 2^{31}-1\right)$.
tFrac32	s32OutValOn	Value of the output when input is higher than the upper threshold; fractional format normalized to fit into $\left[-2^{31}, 2^{31}-1\right)$.
tFrac32	s32OutValOff	Value of the output when input is the lower than lower threshold; fractional format normalized to fit into $\left[-2^{31}, 2^{31}-1\right)$.
tFrac32	s32OutState	Actual state of the output; fractional format normalized to fit into $\left[-2^{31}, 2^{31}-1\right)$.

7.20 GFLIB_INTEGRATOR_TR_T

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

7.20.1 Description

Structure containing integrator parameters and coefficients.

7.20.2 Compound Type Members

Table 7-20: Members description

Type	Name	Description
tFrac32	s32State	Integrator state value
tFrac32	s32InK1	Input value in step k-1
tFrac32	s32C1	Integrator coefficient = $\frac{E_{MAX} \cdot T_s}{U_{MAX} \cdot 2} \cdot 2^{-u16NShift}$
tU16	u16NShift	Scaling factor for the integrator coefficient s32C1, integer format $[0, 31]$

7.21 GFLIB_LIMIT_T

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

7.21.1 Description

Structure containing the limits.

7.21.2 Compound Type Members

Table 7-21: Members description

Type	Name	Description
tFrac32	s32LowerLimit	Lower limit.
tFrac32	s32UpperLimit	Upper limit.

7.22 GFLIB_LOWERLIMIT_T

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

7.22.1 Description

Structure containing the lower limit.

7.22.2 Compound Type Members

Table 7-22: Members description

Type	Name	Description
tFrac32	s32LowerLimit	Lower limit.

7.23 GFLIB_LUT1D_T

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

7.23.1 Description

Structure containing look-up table parameters.

7.23.2 Compound Type Members

Table 7-23: Members description

Type	Name	Description
tS32	s32ShamOffset	Shift amount for extracting the fractional offset within an interpolated interval.
tS32	s32ShamIntvl	Shift amount for extracting the interval index of an interpolated interval.
const tFrac32 *	ps32Table	Table holding ordinate values of interpolating intervals.

7.24 GFLIB_RAMP_T

```
#include <GFLIB_Ramp.h>
```

7.24.1 Description

Structure containing controller parameters and coefficients.

7.24.2 Compound Type Members

Table 7-24: Members description

Type	Name	Description
tFrac32	s32State	Ramp state value
tFrac32	s32RampUp	Ramp up increment coefficient
tFrac32	s32RampDown	Ramp down increment(decrement) coefficient

7.25 GFLIB_SINTLR_T

```
#include <GFLIB_Sin.h>
```

7.25.1 Description

Structure containing one array of five 32-bit elements for storing coefficients of a Taylor polynomial. By calling the function alias GFLIB_Sin, default values of the coefficients are used. Polynomial coefficients can be modified by the user and in such a case the full function call shall be used, that is GFLIB_SinANSIC.

7.25.2 Compound Type Members

Table 7-25: Members description

Type	Name	Description
tFrac32	s32A[5]	Array of five 32-bit elements for storing coefficients of the Taylor polynomial.

7.26 GFLIB_TAN_TAYLOR_COEF_T

```
#include <GFLIB_Tan.h>
```

7.26.1 Description

Structure containing four polynomial coefficients for one subinterval. Output of $\tan(\pi \cdot \text{s32In})$ for interval $[0, \frac{\pi}{4})$ of the input angles is divided into eight subsectors. Polynomial approximation is done using a 4th order polynomial, for each subsector respectively. Four coefficients for a single subinterval are stored in this GFLIB_TAN_TAYLOR_COEF_T structure.

7.26.2 Compound Type Members

Table 7-26: Members description

Type	Name	Description
tFrac32	s32A[4]	Array of four 32-bit elements for storing the polynomial coefficients for one subinterval.

7.27 GFLIB_TANTLR_T

```
#include <GFLIB_Tan.h>
```

7.27.1 Description

Structure containing eight substructures with polynomial coefficients to cover all subintervals. Output of $\tan(\pi \cdot \text{s32In})$ for interval $[0, \frac{\pi}{4})$ of the input angles is divided into eight subsectors. Polynomial approximation is done using a 4th order polynomial, for each subsector respectively. Eight arrays, each including four polynomial coefficients for each subinterval, are stored in this GFLIB_TANTLR_T structure.

By calling the function alias GFLIB_Tan, default values of the coefficients are used. Polynomial coefficients can be modified by the user and in such a case the full function call shall be used, that is GFLIB_TanANSIC.

7.27.2 Compound Type Members

Table 7-27: Members description

Type	Name	Description
GFLIB_TAN_-TAYLOR_COEF_T	GFLIB_TAN_SECTOR[8]	Array of eight elements for storing eight subarrays (each subarray contains four 32-bit coefficients) for all subintervals.

7.28 GFLIB_UPPERLIMIT_T

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

7.28.1 Description

Structure containing the upper limit.

7.28.2 Compound Type Members

Table 7-28: Members description

Type	Name	Description
tFrac32	s32UpperLimit	Upper Limit.

7.29 GFLIB_VECTORLIMIT_T

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

7.29.1 Description

Structure containing the limits.

7.29.2 Compound Type Members

Table 7-29: Members description

Type	Name	Description
tFrac32	s32Lim	The maximum magnitude of the input vector. The defined magnitude must be positive and equal to or greater than 2^-15.

7.30 GMCLIB_DECOUPLINGPMSM_T

```
#include <GMCLIB_DecouplingPMSM.h>
```

7.30.1 Description

Structure containing coefficients for calculation of the decoupling algorithm implemented in the GMCLIB_DecouplingPMSM function.

7.30.2 Compound Type Members

Table 7-30: Members description

Type	Name	Description
tFrac32	s32Kd	Coefficient k_{df}, in fractional format normalized to fit into $[-2^{31}, 2^{31}-1)$
tS16	s16KdShift	Scaling coefficient k_{d_shift}, integer format $[-31, 31]$
tFrac32	s32Kq	Coefficient k_{qf}, in fractional format normalized to fit into $[-2^{31}, 2^{31}-1)$
tS16	s16KqShift	Scaling coefficient k_{q_shift}, integer format $[-31, 31]$

7.31 GMCLIB_ELIMDCBUSRIP_T

```
#include <GMCLIB_ElimDcBusRip.h>
```

7.31.1 Description

Structure containing the PWM modulation index and the measured value of the DC bus voltage for calculation of the DC bus ripple elimination.

7.31.2 Compound Type Members

Table 7-31: Members description

Type	Name	Description
tFrac32	s32ModIndex	Inverse Modulation Index, fractional format normalized to fit into $\left[-2^{31}, 2^{31}-1\right)$, must be 0 or positive.
tFrac32	s32ArgDcBusMsr	Measured DC bus voltage, fractional format normalized to fit into $\left[-2^{31}, 2^{31}-1\right)$, must be 0 or positive.

7.32 SWLIBS_2Syst

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

7.32.1 Description

Structure data type for two axis input/output variables.

7.32.2 Compound Type Members

Table 7-32: Members description

Type	Name	Description
tFrac32	s32Arg1	First argument, type signed 32-bit fractional
tFrac32	s32Arg2	Second argument, type signed 32-bit fractional

7.33 SWLIBS_3Syst

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

7.33.1 Description

Structure data type for three axis input/output variables.

7.33.2 Compound Type Members

Table 7-33: Members description

Type	Name	Description
tFrac32	s32Arg1	First argument, type signed 32-bit fractional
tFrac32	s32Arg2	Second argument, type signed 32-bit fractional
tFrac32	s32Arg3	Third argument, type signed 32-bit fractional

CHAPTER 8: MACRO DEFINITIONS

8.1 Macro Definitions Overview

- GDFLIB_FilterFIRInit(pParam, pState, pInBuf)
- GDFLIB_FilterFIR(s32In, pParam, pState)
- GDFLIB_FilterIIR1Init(pParam)
- GDFLIB_FilterIIR1(s32In, pParam)
- GDFLIB_FILTER_IIR1_DEFAULT
- GDFLIB_FilterIIR2Init(pParam)
- GDFLIB_FilterIIR2(s32In, pParam)
- GDFLIB_FILTER_IIR2_DEFAULT
- GDFLIB_FilterMAInit(pParam)
- GDFLIB_FilterMA(s32In, pParam)
- GDFLIB_FILTER_MA_DEFAULT
- GFLIB_Acos(x)
- GFLIB_Asin(x)
- GFLIB_Atan(x)
- GFLIB_AtanYX(y, x)
- GFLIB_AtanYXShifted(y, x, p)
- GFLIB_ControllerPIp(s32InErr, pParam)
- GFLIB_CONTROLLER_PI_P_DEFAULT
- GFLIB_ControllerPIpAW(s32InErr, pParam)
- GFLIB_CONTROLLER_PIAW_P_DEFAULT
- GFLIB_ControllerPIr(s32InErr, pParam)
- GFLIB_CONTROLLER_PI_R_DEFAULT
- GFLIB_Cos(s32In)

- GFLIB_Hyst(s32In, pParam)
- GFLIB_HYST_DEFAULT
- GFLIB_IntegratorTR(in, pParam)
- GFLIB_INTEGRATOR_TR_DEFAULT
- GFLIB_Limit(s32In, pParam)
- GFLIB_LowerLimit(s32In, pParam)
- GFLIB_Lut1D(x, pParam)
- GFLIB_Ramp(in, pParam)
- GFLIB_RAMP_DEFAULT
- GFLIB_Sign(x)
- GFLIB_Sin(s32In)
- GFLIB_Sqrt(s32In)
- GFLIB_Tan(s32In)
- GFLIB_UpperLimit(s32In, pParam)
- F32SQRT2BY2
- F32MULBY2(x, y)
- GFLIB_VectorLimit(s32Out, s32In, pParam)
- GMCLIB_Clark(pOut, pIn)
- GMCLIB_ClarkInv(pOut, pIn)
- GMCLIB_DecouplingPMSM(pUdqDec, pUdq, pIdq, s32AngularVel, pParam)
- GMCLIB_DECOUPLINGPMSM_DEFAULT
- GMCLIB_ElimDcBusRip(pOut, pIn, pParams)
- GMCLIB_ELIMDCBUSRIP_DEFAULT
- GMCLIB_Park(pOut, pInAngle, pIn)
- GMCLIB_ParkInv(pOut, pInAngle, pIn)
- GMCLIB_SvmStd(pOutput, pInput)
- USE_FRAC32_ARITHMETIC
- SFRACT_MIN
- SFRACT_MAX

- FRACT_MIN
- FRACT_MAX
- INT16_MAX
- INT16_MIN
- INT32_MAX
- INT32_MIN
- FRAC16(x)
- FRAC32(x)
- F16TOINT16(x)
- F32TOINT16(x)
- F64TOINT16(x)
- F16TOINT32(x)
- F32TOINT32(x)
- F64TOINT32(x)
- F16TOINT64(x)
- F32TOINT64(x)
- F64TOINT64(x)
- INT16TOF16(x)
- INT16TOF32(x)
- INT32TOF16(x)
- INT32TOF32(x)
- INT64TOF16(x)
- INT64TOF32(x)
- F16_1_DIVBY_SQRT3
- F32_1_DIVBY_SQRT3
- F16_SQRT3_DIVBY_2
- F32_SQRT3_DIVBY_2
- FALSE
- TRUE

8.2 GDFLIB_FilterFIRInit

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

8.2.1 Macro Definition

```
#define GDFLIB_FilterFIRInit(pParam, pState, pInBuf) \
        GDFLIB_FilterFIRInitANSIC((pParam), (pState), (pInBuf))
```

8.2.2 Description

Function alias for the GDFLIB_FilterFIRInitANSIC function.

8.3 GDFLIB_FilterFIR

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

8.3.1 Macro Definition

```
#define GDFLIB_FilterFIR(s32In, pParam, pState) \
        GDFLIB_FilterFIRANSIC((s32In), (pParam), (pState))
```

8.3.2 Description

Function alias for the GDFLIB_FilterFIRANSIC function.

8.4 GDFLIB_FilterIIR1Init

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

8.4.1 Macro Definition

```
#define GDFLIB_FilterIIR1Init(pParam) \
        GDFLIB_FilterIIR1InitANSIC(pParam)
```

8.4.2 Description

Function alias for the GDFLIB_FilterIIR1InitANSIC function.

8.5 GDFLIB_FilterIIR1

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

8.5.1 Macro Definition

```
#define GDFLIB_FilterIIR1(s32In, pParam) \
        GDFLIB_FilterIIR1ANSIC(s32In,pParam)
```

8.5.2 Description

Function alias for the GDFLIB_FilterIIR1ANSIC function.

8.6 GDFLIB_FILTER_IIR1_DEFAULT

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

8.6.1 Macro Definition

```
#define GDFLIB_FILTER_IIR1_DEFAULT        0,0,0,0,0
```

8.6.2 Description

Macro containing default values of the first order IIR filter structure.

8.7 GDFLIB_FilterIIR2Init

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

8.7.1 Macro Definition

```
#define GDFLIB_FilterIIR2Init(pParam) \
        GDFLIB_FilterIIR2InitANSIC(pParam)
```

8.7.2 Description

Function alias for the GDFLIB_FilterIIR2InitANSIC function.

8.8 GDFLIB_FilterIIR2

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

8.8.1 Macro Definition

```
#define GDFLIB_FilterIIR2(s32In, pParam) \
        GDFLIB_FilterIIR2ANSIC(s32In,pParam)
```

8.8.2 Description

Function alias for the GDFLIB_FilterIIR2ANSIC function.

8.9 GDFLIB_FILTER_IIR2_DEFAULT

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

8.9.1 Macro Definition

```
#define GDFLIB_FILTER_IIR2_DEFAULT        0,0,0,0,0,0,0,0,0
```

8.9.2 Description

Macro containing default values of the second order IIR filter structure.

8.10 GDFLIB_FilterMAInit

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

8.10.1 Macro Definition

```
#define GDFLIB_FilterMAInit(pParam) \
        GDFLIB_FilterMAInitANSIC(pParam)
```

8.10.2 Description

Function alias for the GDFLIB_FilterMAInitANSIC function.

8.11 GDFLIB_FilterMA

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

8.11.1 Macro Definition

```
#define GDFLIB_FilterMA(s32In, pParam) \
        GDFLIB_FilterMAANSIC(s32In, pParam)
```

8.11.2 Description

Function alias for the GDFLIB_FilterMAANSIC function.

8.12 GDFLIB_FILTER_MA_DEFAULT

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

8.12.1 Macro Definition

```
#define GDFLIB_FILTER_MA_DEFAULT        0,0
```

8.12.2 Description

Macro containing default values of the first order MA filter structure.

8.13 GFLIB_Acos

```
#include <GFLIB_Acos.h>
```

8.13.1 Macro Definition

```
#define GFLIB_Acos(x) \
        GFLIB_AcosANSIC((x), &gflibAcosCoef)
```

8.13.2 Description

Function alias for the GFLIB_AcosANSIC function.

8.14 GFLIB_Asin

```
#include <GFLIB_Asin.h>
```

8.14.1 Macro Definition

```
#define GFLIB_Asin(x) \
        GFLIB_AsinANSIC((x), &gflibAsinCoef)
```

8.14.2 Description

Function alias for the GFLIB_AsinANSIC function.

8.15 GFLIB_Atan

```
#include <GFLIB_Atan.h>
```

8.15.1 Macro Definition

```
#define GFLIB_Atan(x) \
        GFLIB_AtanANSIC((x), &gflibAtanCoef)
```

8.15.2 Description

Function alias for the GFLIB_AtanANSIC function.

8.16 GFLIB_AtanYX

```
#include <GFLIB_AtanYX.h>
```

8.16.1 Macro Definition

```
#define GFLIB_AtanYX(y, x) \
        GFLIB_AtanYXANSIC((y), (x))
```

8.16.2 Description

Function alias for the GFLIB_AtanYXANSIC function.

8.17 GFLIB_AtanYXShifted

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

8.17.1 Macro Definition

```
#define GFLIB_AtanYXShifted(y, x, p) \
        GFLIB_AtanYXShiftedANSIC((y), (x), (p))
```

8.17.2 Description

Function alias for the GFLIB_AtanYXShiftedANSIC function

8.18 GFLIB_ControllerPIp

```
#include <GFLIB_ControllerPIp.h>
```

8.18.1 Macro Definition

```
#define GFLIB_ControllerPIp(s32InErr, pParam) \
        GFLIB_ControllerPIpANSIC(s32InErr, pParam)
```

8.18.2 Description

Function alias for the GFLIB_ControllerPIpANSIC function.

8.19 GFLIB_CONTROLLER_PI_P_DEFAULT

```
#include <GFLIB_ControllerPIp.h>
```

8.19.1 Macro Definition

```
#define GFLIB_CONTROLLER_PI_P_DEFAULT        0,0,0,0,0,0
```

8.19.2 Description

Macro containing default values of the parallel PI controller structure.

8.20 GFLIB_ControllerPIpAW

```
#include <GFLIB_ControllerPIpAW.h>
```

8.20.1 Macro Definition

```
#define GFLIB_ControllerPIpAW(s32InErr, pParam) \
        GFLIB_ControllerPIpAWANSIC(s32InErr, pParam)
```

8.20.2 Description

Function alias for the GFLIB_ControllerPIpAWANSIC function.

8.21 GFLIB_CONTROLLER_PIAW_P_DEFAULT

```
#include <GFLIB_ControllerPIpAW.h>
```

8.21.1 Macro Definition

```
#define GFLIB_CONTROLLER_PIAW_P_DEFAULT        0,0,0,0,INT32_MIN,INT32_MAX,0,0,0
```

8.21.2 Description

Macro containing default values for the structure of parameters of the parallel PI controller with antiwindup.

8.22 GFLIB_ControllerPIr

```
#include <GFLIB_ControllerPIr.h>
```

8.22.1 Macro Definition

```
#define GFLIB_ControllerPIr(s32InErr, pParam) \
        GFLIB_ControllerPIrANSIC(s32InErr, pParam)
```

8.22.2 Description

Function alias for the GFLIB_ControllerPIrANSIC function.

8.23 GFLIB_CONTROLLER_PI_R_DEFAULT

```
#include <GFLIB_ControllerPIr.h>
```

8.23.1 Macro Definition

```
#define GFLIB_CONTROLLER_PI_R_DEFAULT        0,0,0,0,0
```

8.23.2 Description

Macro containing default values of the recurrent form PI controller structure.

8.24 GFLIB_Cos

```
#include <GFLIB_Cos.h>
```

8.24.1 Macro Definition

```
#define GFLIB_Cos(s32In) \
        GFLIB_CosANSIC(s32In, &gflibCosCoef)
```

8.24.2 Description

Function alias for the GFLIB_CosANSIC function.

8.25 GFLIB_Hyst

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

8.25.1 Macro Definition

```
#define GFLIB_Hyst(s32In, pParam) \
        GFLIB_HystANSIC(s32In, pParam)
```

8.25.2 Description

Function alias for the GFLIB_HystANSIC function.

8.26 GFLIB_HYST_DEFAULT

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

8.26.1 Macro Definition

```
#define GFLIB_HYST_DEFAULT        0,0,0,0,0
```

8.26.2 Description

Macro containing default values of the hysteresis function structure.

8.27 GFLIB_IntegratorTR

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

8.27.1 Macro Definition

```
#define GFLIB_IntegratorTR(in, pParam) \
        GFLIB_IntegratorTRANSIC(in, pParam)
```

8.27.2 Description

Function alias for the GFLIB_IntegratorTRANSIC function.

8.28 GFLIB_INTEGRATOR_TR_DEFAULT

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

8.28.1 Macro Definition

```
#define GFLIB_INTEGRATOR_TR_DEFAULT        0,0,0,0
```

8.28.2 Description

Macro containing default values of the integrator structure.

8.29 GFLIB_Limit

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

8.29.1 Macro Definition

```
#define GFLIB_Limit(s32In, pParam) \
        GFLIB_LimitANSIC((s32In), (pParam))
```

8.29.2 Description

Function alias for the GFLIB_LimitANSIC function.

8.30 GFLIB_LowerLimit

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

8.30.1 Macro Definition

```
#define GFLIB_LowerLimit(s32In, pParam) \
        GFLIB_LowerLimitANSIC((s32In), (pParam))
```

8.30.2 Description

Function alias for the GFLIB_LowerLimitANSIC function.

8.31 GFLIB_Lut1D

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

8.31.1 Macro Definition

```
#define GFLIB_Lut1D(x, pParam) \
        GFLIB_Lut1DANSIC((x), (pParam))
```

8.31.2 Description

Function alias for the GFLIB_Lut1DANSIC function.

8.32 GFLIB_Ramp

```
#include <GFLIB_Ramp.h>
```

8.32.1 Macro Definition

```
#define GFLIB_Ramp(in, pParam) \
        GFLIB_RampANSIC(in, pParam)
```

8.32.2 Description

Function alias for the GFLIB_RampANSIC function.

8.33 GFLIB_RAMP_DEFAULT

```
#include <GFLIB_Ramp.h>
```

8.33.1 Macro Definition

```
#define GFLIB_RAMP_DEFAULT        0,0,0
```

8.33.2 Description

Macro containing default values of the Ramp structure.

8.34 GFLIB_Sign

```
#include <GFLIB_Sign.h>
```

8.34.1 Macro Definition

```
#define GFLIB_Sign(x) \
        GFLIB_SignANSIC(x)
```

8.34.2 Description

Function alias for the GFLIB_SignANSIC function.

8.35 GFLIB_Sin

```
#include <GFLIB_Sin.h>
```

8.35.1 Macro Definition

```
#define GFLIB_Sin(s32In) \
        GFLIB_SinANSIC(s32In, &gflibSinCoef)
```

8.35.2 Description

Function alias for the GFLIB_SinANSIC function.

8.36 GFLIB_Sqrt

```
#include <GFLIB_Sqrt.h>
```

8.36.1 Macro Definition

```
#define GFLIB_Sqrt(s32In) \
        GFLIB_SqrtANSIC(s32In)
```

8.36.2 Description

Function alias for the GFLIB_SqrtANSIC function.

8.37 GFLIB_Tan

```
#include <GFLIB_Tan.h>
```

8.37.1 Macro Definition

```
#define GFLIB_Tan(s32In) \
        GFLIB_TanANSIC(s32In, &gflibTanCoef)
```

8.37.2 Description

Function alias for the GFLIB_TanANSIC function.

8.38 GFLIB_UpperLimit

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

8.38.1 Macro Definition

```
#define GFLIB_UpperLimit(s32In, pParam) \
        GFLIB_UpperLimitANSIC((s32In),(pParam))
```

8.38.2 Description

Function alias for the GFLIB_UpperLimitANSIC function.

8.39 F32SQRT2BY2

```
#include <GFLIB_VectorLimit.c>
```

8.39.1 Macro Definition

```
#define F32SQRT2BY2        0x5A82799A
```

8.39.2 Description

Local define for the GFLIB_VectorLimit function performing a 32-bit fractional multiplication with division by 2.

8.40 F32MULBY2

```
#include <GFLIB_VectorLimit.c>
```

8.40.1 Macro Definition

```
#define F32MULBY2(x, y) \
        ((tS32) ((((tS64) (x))*((tS64) (y)))>>32))
```

8.40.2 Description

Local define for the GFLIB_VectorLimit function holding a 32-bit $\sqrt{2}/2$.

8.41 GFLIB_VectorLimit

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

8.41.1 Macro Definition

```
#define GFLIB_VectorLimit(s32Out, s32In, pParam) \
        GFLIB_VectorLimitANSIC((s32Out), (s32In), (pParam))
```

8.41.2 Description

Function alias for the GFLIB_VectorLimitANSIC function.

8.42 GMCLIB_Clark

```
#include <GMCLIB_Clark.h>
```

8.42.1 Macro Definition

```
#define GMCLIB_Clark(pOut, pIn) \
        GMCLIB_ClarkANSIC(pOut,pIn)
```

8.42.2 Description

Function alias for the GMCLIB_ClarkANSIC function.

8.43 GMCLIB_ClarkInv

```
#include <GMCLIB_ClarkInv.h>
```

8.43.1 Macro Definition

```
#define GMCLIB_ClarkInv(pOut, pIn) \
        GMCLIB_ClarkInvANSIC(pOut, pIn)
```

8.43.2 Description

Function alias for the GMCLIB_ClarkInvANSIC function.

8.44 GMCLIB_DecouplingPMSM

```
#include <GMCLIB_DecouplingPMSM.h>
```

8.44.1 Macro Definition

```
#define GMCLIB_DecouplingPMSM(pUdqDec, pUdq, pIdq, s32AngularVel, pParam) \
        GMCLIB_DecouplingPMSMANSIC(pUdqDec,pUdq,pIdq,s32AngularVel,pParam)
```

8.44.2 Description

Function alias for the GMCLIB_DecouplingPMSMANSIC function.

8.45 GMCLIB_DECOUPLINGPMSM_DEFAULT

```
#include <GMCLIB_DecouplingPMSM.h>
```

8.45.1 Macro Definition

```
#define GMCLIB_DECOUPLINGPMSM_DEFAULT        0,0,0,0
```

8.45.2 Description

Macro containing reset values of the parameters for the decoupling algorithm implemented in the GMCLIB_DecouplingPMSM function.

8.46 GMCLIB_ElimDcBusRip

```
#include <GMCLIB_ElimDcBusRip.h>
```

8.46.1 Macro Definition

```
#define GMCLIB_ElimDcBusRip(pOut, pIn, pParams) \
        GMCLIB_ElimDcBusRipANSIC(pOut, pIn, pParams)
```

8.46.2 Description

Function alias for the GMCLIB_ElimDcBusRipANSIC function.

8.47 GMCLIB_ELIMDCBUSRIP_DEFAULT

```
#include <GMCLIB_ElimDcBusRip.h>
```

8.47.1 Macro Definition

```
#define GMCLIB_ELIMDCBUSRIP_DEFAULT        0,0
```

8.47.2 Description

Macro containing default values for the parameter structure of the GMCLIB_ElimDcBusRip function.

8.48 GMCLIB_Park

```
#include <GMCLIB_Park.h>
```

8.48.1 Macro Definition

```
#define GMCLIB_Park(pOut, pInAngle, pIn) \
        GMCLIB_ParkANSIC(pOut,pInAngle,pIn)
```

8.48.2 Description

Function alias for the GMCLIB_ParkANSIC function.

8.49 GMCLIB_ParkInv

```
#include <GMCLIB_ParkInv.h>
```

8.49.1 Macro Definition

```
#define GMCLIB_ParkInv(pOut, pInAngle, pIn) \
        GMCLIB_ParkInvANSIC(pOut, pInAngle, pIn)
```

8.49.2 Description

Function alias for the GMCLIB_ParkInvANSIC function.

8.50 GMCLIB_SvmStd

```
#include <GMCLIB_SvmStd.h>
```

8.50.1 Macro Definition

```
#define GMCLIB_SvmStd(pOutput, pInput) \
        GMCLIB_SvmStdANSIC(pOutput,pInput)
```

8.50.2 Description

Function alias for the GMCLIB_SvmStdANSIC function.

8.51 USE_FRAC32_ARITHMETIC

```
#include <SWLIBS_Defines.h>
```

8.51.1 Macro Definition

```
#define USE_FRAC32_ARITHMETIC
```

8.51.2 Description

If USE_ASM macro is enabled, then, if existing, an assembly version of the functions are used. If USE_FRAC32_ARITHMETIC macro is enabled, then 32 bit arithmetic is used for intermediate calculations, achieving the best possible calculation accuracy.

8.52 SFRACT_MIN

```
#include <SWLIBS_Defines.h>
```

8.52.1 Macro Definition

```
#define SFRACT_MIN         (-1.0)
```

8.52.2 Description

Constant representing the maximal negative value of a signed fixed point 16-bit fractional number equalling -1.0.

8.53 SFRACT_MAX

```
#include <SWLIBS_Defines.h>
```

8.53.1 Macro Definition

```
#define SFRACT_MAX        (0.999969482421875)
```

8.53.2 Description

Constant representing the maximal positive value of a signed fixed point 16-bit fractional number equalling 0.999969482421875.

8.54 FRACT_MIN

```
#include <SWLIBS_Defines.h>
```

8.54.1 Macro Definition

```
#define FRACT_MIN        (-1.0)
```

8.54.2 Description

Constant representing the maximal negative value of a signed fixed point 32-bit fractional number equalling -1.0.

8.55 FRACT_MAX

```
#include <SWLIBS_Defines.h>
```

8.55.1 Macro Definition

```
#define FRACT_MAX        (0.999999999534338712692260742187 5)
```

8.55.2 Description

Constant representing the maximal positive value of a signed fixed point 32-bit fractional number equalling 0.9999999995343387126922607421875.

8.56 INT16_MAX

```
#include <SWLIBS_Defines.h>
```

8.56.1 Macro Definition

```
#define INT16_MAX        ((tS16) 0x7fff)
```

8.56.2 Description

Constant representing the maximal positive value of a signed fixed point 16-bit integer number equalling $2^{16-1} - 1 =$ 0x7fff.

8.57 INT16_MIN

```
#include <SWLIBS_Defines.h>
```

8.57.1 Macro Definition

```
#define INT16_MIN        ((tS16) 0x8000)
```

8.57.2 Description

Constant representing the maximal negative value of a signed fixed point 16-bit integer number equalling -2^{16-1} = 0x8000.

8.58 INT32_MAX

```
#include <SWLIBS_Defines.h>
```

8.58.1 Macro Definition

```
#define INT32_MAX        ((tS32) 0x7fffffff)
```

8.58.2 Description

Constant representing the maximal positive value of a signed fixed point 32-bit integer number equalling $2^{32-1} - 1 =$ 0x7fff ffff.

8.59 INT32_MIN

```
#include <SWLIBS_Defines.h>
```

8.59.1 Macro Definition

```
#define INT32_MIN        ((tS32) 0x80000000)
```

8.59.2 Description

Constant representing the maximal negative value of a signed fixed point 32-bit integer number equalling -2^{32-1} = 0x8000 0000.

8.60 FRAC16

```
#include <SWLIBS_Defines.h>
```

8.60.1 Macro Definition

```
#define FRAC16(x) \
        ((tFrac16) ((x) < (SFRACT_MAX) ? ((x) >= SFRACT_MIN ? (x)*0x8000 :
0x8000) :  0x7fff))
```

8.60.2 Description

Macro converting a signed fractional [-1,1) number into a fixed point 16-bit number in the format Q1.15.

8.61 FRAC32

```
#include <SWLIBS_Defines.h>
```

8.61.1 Macro Definition

```
#define FRAC32(x) \
        ((tFrac32) ((x) < (FRACT_MAX) ? ((x) >= FRACT_MIN ? (x)*0x80000000 :
0x80000000) :  0x7fffffff))
```

8.61.2 Description

Macro converting a signed fractional [-1,1) number into a fixed point 32-bit number in the format Q1.31.

8.62 F16TOINT16

```
#include <SWLIBS_Defines.h>
```

8.62.1 Macro Definition

```
#define F16TOINT16(x) \
        ((tS16) (x))
```

8.62.2 Description

Type casting - a signed 16-bit fractional value cast as a signed 16-bit integer.

8.63 F32TOINT16

```
#include <SWLIBS_Defines.h>
```

8.63.1 Macro Definition

```
#define F32TOINT16(x) \
        ((tS16) (x))
```

8.63.2 Description

Type casting - the lower 16 bits of a signed fractional 32-bit fractional value cast as a signed integer 16-bit integer.

8.64 F64TOINT16

```
#include <SWLIBS_Defines.h>
```

8.64.1 Macro Definition

```
#define F64TOINT16(x) \
        ((tS16) (x))
```

8.64.2 Description

Type casting - the lower 16 bits of a signed fractional 64-bit fractional value cast as a signed integer 16-bit integer.

8.65 F16TOINT32

```
#include <SWLIBS_Defines.h>
```

8.65.1 Macro Definition

```
#define F16TOINT32(x) \
        ((tS32) (x))
```

8.65.2 Description

Type casting - a signed 16-bit fractional value cast as a signed 32-bit integer, the value placed at the lower 16 bits of the 32-bit result.

8.66 F32TOINT32

```
#include <SWLIBS_Defines.h>
```

8.66.1 Macro Definition

```
#define F32TOINT32(x) \
        ((tS32) (x))
```

8.66.2 Description

Type casting - a signed 32-bit fractional value cast as a signed 32-bit integer.

8.67 F64TOINT32

```
#include <SWLIBS_Defines.h>
```

8.67.1 Macro Definition

```
#define F64TOINT32(x) \
        ((tS32) (x))
```

8.67.2 Description

Type casting - lower 32 bits of a signed 64-bit fractional value cast as a signed 32-bit integer.

8.68 F16TOINT64

```
#include <SWLIBS_Defines.h>
```

8.68.1 Macro Definition

```
#define F16TOINT64(x) \
        ((tS64) (x))
```

8.68.2 Description

Type casting - a signed 16-bit fractional value cast as a signed 64-bit integer, the value placed at the lower 16 bits of the 64-bit result.

8.69 F32TOINT64

```
#include <SWLIBS_Defines.h>
```

8.69.1 Macro Definition

```
#define F32TOINT64(x) \
        ((tS64) (x))
```

8.69.2 Description

Type casting - a signed 32-bit fractional value cast as a signed 64-bit integer, the value placed at the lower 32 bits of the 64-bit result.

8.70 F64TOINT64

```
#include <SWLIBS_Defines.h>
```

8.70.1 Macro Definition

```
#define F64TOINT64(x) \
        ((tS64) (x))
```

8.70.2 Description

Type casting - a signed 64-bit fractional value cast as a signed 64-bit integer.

8.71 INT16TOF16

```
#include <SWLIBS_Defines.h>
```

8.71.1 Macro Definition

```
#define INT16TOF16(x) \
        ((tFrac16) (x))
```

8.71.2 Description

Type casting - a signed 16-bit integer value cast as a signed 16-bit fractional.

8.72 INT16TOF32

```
#include <SWLIBS_Defines.h>
```

8.72.1 Macro Definition

```
#define INT16TOF32(x) \
        ((tFrac32) (x))
```

8.72.2 Description

Type casting - a signed 16-bit integer value cast as a signed 32-bit fractional, the value placed at the lower 16 bits of the 32-bit result.

8.73 INT32TOF16

```
#include <SWLIBS_Defines.h>
```

8.73.1 Macro Definition

```
#define INT32TOF16(x) \
        ((tFrac16) (x))
```

8.73.2 Description

Type casting - the lower 16 bits of a signed 32-bit integer value cast as a signed 16-bit fractional.

8.74 INT32TOF32

```
#include <SWLIBS_Defines.h>
```

8.74.1 Macro Definition

```
#define INT32TOF32(x) \
        ((tFrac32) (x))
```

8.74.2 Description

Type casting - a signed 32-bit integer value cast as a signed fractional 32-bit fractional.

8.75 INT64TOF16

```
#include <SWLIBS_Defines.h>
```

8.75.1 Macro Definition

```
#define INT64TOF16(x) \
        ((tFrac16) (x))
```

8.75.2 Description

Type casting - the lower 16 bits of a signed 64-bit integer value cast as a signed 16-bit fractional.

8.76 INT64TOF32

```
#include <SWLIBS_Defines.h>
```

8.76.1 Macro Definition

```
#define INT64TOF32(x) \
        ((tFrac32) (x))
```

8.76.2 Description

Type casting - the lower 32 bits of a signed 64-bit integer value cast as a signed 32-bit fractional.

8.77 F16_1_DIVBY_SQRT3

```
#include <SWLIBS_Defines.h>
```

8.77.1 Macro Definition

```
#define F16_1_DIVBY_SQRT3        ((tFrac16) 0x49E7)
```

8.77.2 Description

One over $\sqrt{3}$ with a 16-bit result, the result rounded for a better precision, that is, $round(1/sqrt(3)*2^{15})$

8.78 F32_1_DIVBY_SQRT3

```
#include <SWLIBS_Defines.h>
```

8.78.1 Macro Definition

```
#define F32_1_DIVBY_SQRT3        ((tFrac32) 0x49E69D16)
```

8.78.2 Description

One over $\sqrt{3}$ with a 32-bit result, the result rounded for a better precision, that is, $round(1/sqrt(3)*2^{31})$

8.79 F16_SQRT3_DIVBY_2

```
#include <SWLIBS_Defines.h>
```

8.79.1 Macro Definition

```
#define F16_SQRT3_DIVBY_2        ((tFrac16) 0x6EDA)
```

8.79.2 Description

$\sqrt{3}/2$ with a 16-bit result, the result rounded for a better precision, that is, $round(sqrt(3)/2*2^{15})$

8.80 F32_SQRT3_DIVBY_2

```
#include <SWLIBS_Defines.h>
```

8.80.1 Macro Definition

```
#define F32_SQRT3_DIVBY_2        ((tFrac32) 0x6ED9EBA1)
```

8.80.2 Description

$\sqrt{3}/2$ with a 32-bit result, the result rounded for a better precision, that is, $round(sqrt(3)/2 * 2^{31})$

8.81 FALSE

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

8.81.1 Macro Definition

```
#define FALSE        ((tBool)0)
```

8.81.2 Description

Boolean type FALSE constant

8.82 TRUE

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

8.82.1 Macro Definition

```
#define TRUE        ((tBool)1)
```

8.82.2 Description

Boolean type TRUE constant.

*H*ow to Reach Us:

Home Page:
www.freescale.com

Web Support:
http://www.freescale.com/support

USA/Europe or Locations Not Listed:
Freescale Semiconductor, Inc.
Technical Information Center, EL516
2100 East Elliot Road
Tempe, Arizona 85284
1-800-521-6274 or +1-480-768-2130
www.freescale.com/support

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
www.freescale.com/support

Japan:
Freescale Semiconductor Japan Ltd. Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064
Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor China Ltd.
Exchange Building 23F
No. 118 Jianguo Road
Chaoyang District
Beijing 100022
China
+86 10 5879 8000
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447 or +1-303-675-2140
Fax: +1-303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

MCLIBCORETXM4UG
Rev. 1, 6/2011