PRELIMINARY
Internal Use Only

Motor Control Library Set for the MPC564xL

User Reference Manual

N/A
Rev. 0.1
08/2011

freescale.com

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
08/2011	0.1	Initial release for rev 0.91

CONTENTS

1 License Agreement **1**

2 Introduction **6**
2.1 MC Library Architecture Overview . . . 6
2.2 Motor Control Library Set for the MPC564xL General Information . . . 7
2.3 Supported Compilers . . . 7
2.4 Matlab Integration . . . 7
2.5 Installation . . . 8
2.6 Library File Structure . . . 10
2.7 Library Integration into a Green Hills Multi Development Environment . . . 12
2.8 Library Integration into a Wind River Compiler Environment . . . 16
2.9 Library Integration into a Freescale CodeWarrior Development Environment . . . 19
2.10 MC Library Testing . . . 21

3 GFLIB **24**
3.1 Function API Overview . . . 24
3.2 GFLIB_Sin . . . 26
3.3 GFLIB_Cos . . . 29
3.4 GFLIB_Tan . . . 32
3.5 GFLIB_Asin . . . 36
3.6 GFLIB_Acos . . . 40
3.7 GFLIB_Atan . . . 44
3.8 GFLIB_AtanYX . . . 48
3.9 GFLIB_AtanYXShifted . . . 50
3.10 GFLIB_Sqrt . . . 54
3.11 GFLIB_Sign . . . 56
3.12 GFLIB_Lut1D . . . 58
3.13 GFLIB_Lut2D . . . 62
3.14 GFLIB_Hyst . . . 68
3.15 GFLIB_Ramp . . . 71
3.16 GFLIB_Limit . . . 74
3.17 GFLIB_LowerLimit . . . 76
3.18 GFLIB_UpperLimit . . . 78
3.19 GFLIB_VectorLimit . . . 80
3.20 GFLIB_IntegratorTR . . . 83
3.21 GFLIB_ControllerPIr . . . 85
3.22 GFLIB_ControllerPIrAW . . . 88
3.23 GFLIB_ControllerPIp . . . 91

3.24 GFLIB_ControllerPIpAW 94

4 GMCLIB **97**
4.1 Function API Overview 97
4.2 GMCLIB_Clark 98
4.3 GMCLIB_ClarkInv 100
4.4 GMCLIB_Park 103
4.5 GMCLIB_ParkInv 105
4.6 GMCLIB_ElimDcBusRip 107
4.7 GMCLIB_DecouplingPMSM 111
4.8 GMCLIB_SvmStd 115

5 GDFLIB **127**
5.1 Function API Overview 127
5.2 GDFLIB_FilterIIR1Init 128
5.3 GDFLIB_FilterIIR1 130
5.4 GDFLIB_FilterIIR2Init 133
5.5 GDFLIB_FilterIIR2 135
5.6 GDFLIB_FilterFIRInit 138
5.7 GDFLIB_FilterFIR 141
5.8 GDFLIB_FilterMAInit 144
5.9 GDFLIB_FilterMA 146

6 Data Types **148**
6.1 Defined in <SWLIBS_Typedefs.h> 148

7 Compound Data Types **149**
7.1 FILTER_IIR1_COEFF_T 152
7.2 FILTER_IIR2_COEFF_T 153
7.3 GDFLIB_FILTER_IIR1_T 154
7.4 GDFLIB_FILTER_IIR2_T 155
7.5 GDFLIB_FILTER_MA_T 156
7.6 GDFLIB_FILTERFIR_PARAM_T 157
7.7 GDFLIB_FILTERFIR_STATE_T 158
7.8 GFLIB_ACOS_TAYLOR_T 159
7.9 GFLIB_ASIN_TAYLOR_T 160
7.10 GFLIB_ATAN_TAYLOR_T 161
7.11 GFLIB_ATANYXSHIFTED_T 162
7.12 GFLIB_CONTROLLER_PI_P_T 163
7.13 GFLIB_CONTROLLER_PI_R_T 164
7.14 GFLIB_CONTROLLER_PIAW_P_T 165
7.15 GFLIB_CONTROLLER_PIAW_R_T 166
7.16 GFLIB_COSTLR_T 167
7.17 GFLIB_HYST_T 168
7.18 GFLIB_INTEGRATOR_TR_T 169
7.19 GFLIB_LIMIT_T 170
7.20 GFLIB_LOWERLIMIT_T 171
7.21 GFLIB_LUT1D_T 172

7.22 GFLIB_LUT2D_T . . . 173
7.23 GFLIB_RAMP_T . . . 174
7.24 GFLIB_SINTLR_T . . . 175
7.25 GFLIB_TANTLR_T . . . 176
7.26 GFLIB_UPPERLIMIT_T . . . 177
7.27 GFLIB_VECTORLIMIT_T . . . 178
7.28 GMCLIB_DECOUPLINGPMSM_T . . . 179
7.29 GMCLIB_ELIMDCBUSRIP_T . . . 180
7.30 SWLIBS_2Syst . . . 181
7.31 SWLIBS_3Syst . . . 182

8 Macro Definitions 183
8.1 Macro Definitions Overview . . . 183
8.2 GDFLIB_FilterFIRInit . . . 187
8.3 GDFLIB_FilterFIR . . . 188
8.4 GDFLIB_FilterIIR1Init . . . 189
8.5 GDFLIB_FilterIIR1 . . . 190
8.6 GDFLIB_FILTER_IIR1_DEFAULT . . . 191
8.7 GDFLIB_FilterIIR2Init . . . 192
8.8 GDFLIB_FilterIIR2 . . . 193
8.9 GDFLIB_FILTER_IIR2_DEFAULT . . . 194
8.10 GDFLIB_FilterMAInit . . . 195
8.11 GDFLIB_FilterMA . . . 196
8.12 GDFLIB_FILTER_MA_DEFAULT . . . 197
8.13 GFLIB_Acos . . . 198
8.14 GFLIB_Asin . . . 199
8.15 GFLIB_Atan . . . 200
8.16 GFLIB_AtanYX . . . 201
8.17 GFLIB_AtanXY . . . 202
8.18 GFLIB_AtanYXShifted . . . 203
8.19 GFLIB_ControllerPIp . . . 204
8.20 GFLIB_CONTROLLER_PI_P_DEFAULT . . . 205
8.21 GFLIB_ControllerPIpAW . . . 206
8.22 GFLIB_CONTROLLER_PIAW_P_DEFAULT . . . 207
8.23 GFLIB_ControllerPIr . . . 208
8.24 GFLIB_CONTROLLER_PI_R_DEFAULT . . . 209
8.25 GFLIB_ControllerPIrAW . . . 210
8.26 GFLIB_CONTROLLER_PIAW_R_DEFAULT . . . 211
8.27 GFLIB_Cos . . . 212
8.28 GFLIB_Hyst . . . 213
8.29 GFLIB_HYST_DEFAULT . . . 214
8.30 GFLIB_IntegratorTR . . . 215
8.31 GFLIB_INTEGRATOR_TR_DEFAULT . . . 216
8.32 GFLIB_Limit . . . 217
8.33 GFLIB_LowerLimit . . . 218
8.34 GFLIB_Lut1D . . . 219
8.35 GFLIB_Lut2D . . . 220

8.36 GFLIB_Ramp . . . 221
8.37 GFLIB_RAMP_DEFAULT . . . 222
8.38 GFLIB_Sign . . . 223
8.39 GFLIB_Sin . . . 224
8.40 GFLIB_Sqrt . . . 225
8.41 GFLIB_TAN_LIMIT . . . 226
8.42 GFLIB_Tan . . . 227
8.43 GFLIB_UpperLimit . . . 228
8.44 GFLIB_VectorLimit . . . 229
8.45 GMCLIB_Clark . . . 230
8.46 GMCLIB_ClarkInv . . . 231
8.47 GMCLIB_DecouplingPMSM . . . 232
8.48 GMCLIB_DECOUPLINGPMSM_DEFAULT . . . 233
8.49 GMCLIB_ElimDcBusRip . . . 234
8.50 GMCLIB_ELIMDCBUSRIP_DEFAULT . . . 235
8.51 GMCLIB_Park . . . 236
8.52 GMCLIB_ParkInv . . . 237
8.53 GMCLIB_SvmStd . . . 238
8.54 MATH_FIX32 . . . 239
8.55 MATH_FIX16 . . . 240
8.56 MATH_FLOAT . . . 241
8.57 MATH_DOUBLE . . . 242
8.58 USE_ARITHMETIC . . . 243
8.59 SFRACT_MIN . . . 244
8.60 SFRACT_MAX . . . 245
8.61 FRACT_MIN . . . 246
8.62 FRACT_MAX . . . 247
8.63 INT16_MAX . . . 248
8.64 INT16_MIN . . . 249
8.65 INT32_MAX . . . 250
8.66 INT32_MIN . . . 251
8.67 F16TOINT16 . . . 252
8.68 F32TOINT16 . . . 253
8.69 F64TOINT16 . . . 254
8.70 F16TOINT32 . . . 255
8.71 F32TOINT32 . . . 256
8.72 F64TOINT32 . . . 257
8.73 F16TOINT64 . . . 258
8.74 F32TOINT64 . . . 259
8.75 F64TOINT64 . . . 260
8.76 INT16TOF16 . . . 261
8.77 INT16TOF32 . . . 262
8.78 INT32TOF16 . . . 263
8.79 INT32TOF32 . . . 264
8.80 INT64TOF16 . . . 265
8.81 INT64TOF32 . . . 266
8.82 F16_1_DIVBY_SQRT3 . . . 267

8.83 F32_1_DIVBY_SQRT3 268
8.84 F16_SQRT3_DIVBY_2 269
8.85 F32_SQRT3_DIVBY_2 270
8.86 FRAC16 271
8.87 FRAC32 272
8.88 FLOAT_MIN 273
8.89 FLOAT_MAX 274
8.90 FLOAT_DIVBY_SQRT3 275
8.91 FLOAT_SQRT3_DIVBY_2 276
8.92 FLOAT_SQRT3_DIVBY_4 277
8.93 FLOAT_SQRT3_DIVBY_4_CORRECTION 278
8.94 FLOAT_2_PI 279
8.95 FLOAT_PI 280
8.96 FLOAT_PI_DIVBY_2 281
8.97 FLOAT_PI_SINGLE_CORRECTION 282
8.98 FLOAT_PI_CORRECTION 283
8.99 FLOAT_PI_DIVBY_4 284
8.100 FLOAT_4_DIVBY_PI 285
8.101 FLOAT_0_5 286
8.102 FLOAT_PLUS_1 287
8.103 FLOAT_MINUS_1 288
8.104 inline 289
8.105 _ARG2 290
8.106 VA_NUM_ARGS 291
8.107 macro_dispatcher 292
8.108 macro_dispatcher_ 293
8.109 macro_dispatcher__ 294
8.110 FALSE 295
8.111 TRUE 296

CHAPTER 1: LICENSE AGREEMENT

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

FREESCALE SOFTWARE LICENSE AGREEMENT 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 (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 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.

LICENSE GRANT: Your rights to use this Software vary depending upon the type of license model you ordered from Freescale. The type of license will be set forth in the Freescale Quotation document which formed the basis of your order or, with respect to evaluation licenses, in the correspondence with Freescale confirming your request for a copy of the Software for evaluation. If you do not remember this information, you may contact the party from whom you obtained the Software or us at www.freescale.com.

Definitions: For purposes of this Agreement the following terms are defined as set forth below:

"Authorized System" means Licensee's ECU product containing a Freescale processor or other semiconductor product supplied directly or indirectly from Freescale.

"Confidential Information" means any information disclosed by Freescale to the Licensee and, if disclosed in writing or in some other tangible form, that is marked at the time of disclosure as being "Confidential" or "Proprietary" or with words of similar import; provided, that Software provided in source code format will be deemed Confidential Information whether or not identified as such in writing or otherwise. Confidential Information does not include any information that: (a) is, or becomes, publicly known through no wrongful act on the Licensee's part; (b) is already known to the Licensee, or becomes lawfully known to the Licensee without restriction on disclosure; or (c) is independently developed by the Licensee.

"Customer Product Line" means the customer product line specified in the Quotation Document.

"Customer Target Project" means the one customer Target Project for one automotive vehicle manufacturer specified in the Quotation Document.

"Limited Derivative Works" means Licensee may make derivative works only to the extent the copyrighted material provided by Freescale is not modified. For the avoidance of doubt, Licensee explicitly has the right to: 1) add source code to source code provided by Freescale without modification to the original source code; 2) compile any source code into object code,

or 3) integrate the Freescale source code into its ECU without modifying the original source code.

"Quotation Document" means the Freescale document offering the goods and/or services which formed the basis of your order.

"Target Product" means the Freescale processor or other semiconductor product set forth in the Quotation Document.

"Target Product Family" means the Freescale processor family or other semiconductor product family set forth in the Quotation Document.

I. Non-Production License Models:

a. Evaluation License Freescale grants to you, free of charge, a personal, non-transferable, non-exclusive, revocable, royalty-free, license to use the Software for ninety (90) consecutive days from the date of your acceptance as indicated by clicking the "I accept" button below, internally, solely for the purpose of performing evaluation and testing of the Software, solely for automotive use, and solely on a Freescale Target Product. This license does not include the right to reproduce, distribute, sell, lease, sublicense or transfer all or any part of the Software, or to use the Software on non-Freescale integrated circuit devices, or to use the Software for any production purposes whatsoever. Freescale reserves all rights not expressly granted in this Agreement. 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.

b. Development License: Subject to payment of the fees set forth in the Quotation Document, Freescale grants to you a personal, non-transferable, non-exclusive, revocable, license for two years from the date of your acceptance as indicated by clicking the "I accept" button below, unless a different license term is specified in the Quotation Document ("Term"), (1) to use the Software on one Target Product Family for execution on a maximum of 500 sample Authorized Systems, (2) to reproduce the Software, (3) to prepare Limited Derivative Works of the Licensed Software (4) to distribute the Software and Limited Derivative Works thereof to automotive customers, and (5) to sublicense to automotive customers the right to use the distributed Software as included within the Authorized System solely for purposes of testing the Authorized System during the Term. Freescale reserves all rights not expressly granted in this Agreement. 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. You are solely responsible for systems you design using the Software.

II. Production License Models:

a. Project License: Subject to payment of the fees set forth in the Quotation Document, exclusively in conjunction with Licensee's development and sale of a Authorized System, Freescale grants to you the non-exclusive, non-transferable right (1) to use the Software in the Target Project on the Target Product, (2) to reproduce the Software, (3) to prepare Limited Derivative Works of the Software, (4) to distribute the Software and Limited Derivative Works thereof as part of a Authorized System, and (5) to sublicense to others the right to use the distributed Software as included within the Authorized System. 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.

b. Product Line License: Subject to payment of the fees set forth in the Quotation Document, exclusively in conjunction with Licensee's development and sale of a Authorized System, Freescale grants to you, the non-exclusive, non-transferable right (1) to use the Software in one Target Product Family in one Customer Product Line, (2) to reproduce the Software, (3) to prepare Limited Derivative Works of the Software, (4) to distribute the Software and Limited Derivative Works thereof as part of a Authorized System, and (5) to sublicense to others the right to use the distributed Software as included within the Authorized System. 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.

c. Family Multi-Project License: Subject to payment of the fees set forth in the Quotation Document, exclusively in conjunction with Licensee's development and sale of a Authorized System, Freescale grants to you, the non-exclusive, non-transferable right (1) to use the Software in one Target Product Family, (2) to reproduce the Software, (3) to prepare Limited Derivative Works of the Software, (4) to distribute the Software and Limited Derivative Works thereof as part of a Authorized System, and (5) to sublicense to others the right to use the distributed Software as included within the Authorized System. 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.

d. Volume License: Subject to payment of the fees set forth in the Quotation Document and the prior taking of a valid Development License for the same Target Product Family, exclusively in conjunction with Licensee's development and sale of a Authorized System, Freescale grants to you, the non-exclusive, non-transferable right (1) to use the Software in one Target Product Family on a maximum of the number of Authorized Systems set forth Quotation Document, (2) to reproduce the Software, (3) to prepare Limited Derivative Works of the Software, (4) to distribute the Software and Limited Derivative Works thereof as part of a Authorized System, and (5) to sublicense to others the right to use the distributed Software as included within the Authorized System. 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.

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. Except as otherwise agreed by the parties under separate agreement, 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.

NO WARRANTY. TO THE MAXIMUM EXTENT PERMITTED BY LAW, FREESCALE EXPRESSLY DISCLAIMS ANY WARRANTY FOR THE SOFTWARE. THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT. YOU ASSUME THE ENTIRE RISK ARISING OUT OF THE USE OR PERFORMANCE OF THE SOFTWARE, OR ANY SYSTEMS YOU DESIGN USING THE SOFTWARE (IF ANY). NOTHING IN THIS AGREEMENT MAY BE CONSTRUED AS A WARRANTY OR REPRESENTATION BY FREESCALE THAT THE SOFTWARE OR ANY DERIVATIVE WORK DEVELOPED WITH OR INCORPORATING THE SOFTWARE WILL BE FREE FROM INFRINGEMENT OF THE INTELLECTUAL PROPERTY RIGHTS OF THIRD PARTIES.

INDEMNITY. You agree to fully defend and indemnify Freescale from any and all claims, liabilities, and costs (including reasonable attorney's fees) related to (1) your use (including your sublicensee's use, if permitted) of the Software or (2) your violation of the terms and conditions of this Agreement.

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 DISCLAIMED 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, in which the failure of products could lead directly to death, personal injury or severe physical or environmental damage ("High Risk Activities"). You specifically represent and warrant that you will not use the Software or any derivative work of the Software for High Risk Activities.

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.

CONFIDENTIAL INFORMATION. The Licensee agrees to retain the Confidential Information in confidence perpetually, with respect to Software in source code form (human readable), or for a period of 3 years from the date of receipt of the Confidential Information, with respect to all other Confidential Information. During this period the Licensee may not disclose Freescale's Confidential Information to any third party except where such disclosure is necessary in order to achieve the purpose of the license because the automotive customer requires disclosure in conjunction with the testing or purchase of an Authorized System and who have executed written agreements obligating them to protect such Confidential Information. During this period Licensee will also limit dissemination of the Confidential Information to its employees or contractors who have a need to know of the Confidential Information and who have executed written agreements obligating them to protect such Confidential Information, and may not use the Freescale's Confidential Information for any purpose not authorized by this Agreement. The Licensee further agrees to use the same degree of care, but no less than a reasonable degree of care, with Freescale's Confidential Information as it would with its own confidential information. The Licensee may disclose Confidential Information as required by a court or under operation of law or order provided that the Licensee notifies Freescale of such requirement prior to disclosure, that the Licensee discloses only that information required, and that the Licensee allows Freescale the opportunity to object to such court or other legal body requiring such disclosure.

Freescale Automotive Software License Agreement, v1.02, June 2011

CHAPTER 2: INTRODUCTION

2.1 MC Library Architecture Overview

The aim of this document is to describe the Motor Control Library Set for the MPC564xL Power PC. It describes the components of the library, its behaviour and interaction, the API and steps needed to integrate the library into the customer project.

Figure 2-1: MCLib Components Structure.

The Motor Control Library Set for the MPC564xL consists of three sub-libraries, functionally connected as depicted in Fig. 2-1.

The Motor Control Library Set for the MPC564xL libraries are as follows:

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

As can be seen in Fig. 2-1, the Motor Control Library Set for the MPC564xL libraries form the layer architecture where only the GFLIB and GDLIB libraries are completely independent and can be used as stand-alone. The GMCLIB library depends on GFLIB and GDFLIB, hence cannot be used as stand-alone.

2.2 Motor Control Library Set for the MPC564xL General Information

The Motor Control Library Set for the MPC564xL was developed to fully support the embedded floating point unit available in the MPC564xLz4 core. With the exception of those functions where the mathematical principle limits the input or output values, all functions support the full single precision floating point data range. In compliance with IEEE 754-1985, the full single precision floating point data range is defined by the interval $(-2^{128}; 2^{128})$ and the minimum positive normalized value is 2^{-126}.

The Motor Control Library Set for the MPC564xL was tested using two different test methods. To test the precision of each function implementation, the testing based on Matlab reference models was used. This release was tested using the Matlab R2009a version. To test the implementation on the embedded side, the target-in-loop testing was performed on the Motor Control Library Set for the MPC564xL. This release was tested using the SFIO toolbox version 2.2 and Matlab R2009a version.

2.3 Supported Compilers

The Motor Control Library Set for the MPC564xL is written in ANSI-C99 ISO/IEC 9899:1999 standard language. The library was built and tested using the following compilers:

1. GreenHills MULTI for PPC version 5.1.7
2. Wind River Compiler version 5.8.0
3. CodeWarrior Development Studio for Freescale MPC55xx, MPC56xx version 2.7

The library is delivered in a library module "MCLIB_MPC564xL_v0.91.a" for the Green Hills compiler, library module "MCLIB_MPC564xL_v0.91.a for the Wind River compiler" and in two library modules "MCLIB_MPC564xL_v0.91.$suffix$" for the CodeWarrior compiler, with suffixes ".PPCEABI.V.SC.a" for signed and ".PPCEABI.V.UC.a" for unsigned char data type. Together with the pre-compiled library modules, these are all the necessary header files. The interfaces to the algorithms included in this library have been combined into a single public interface header file for each respective sub-library, i.e. gflib.h, gdflib.h and gmclib.h. This was done to simplify the number of files required for inclusion by application programs. Refer to the specific algorithm sections of this document for details on the software Application Programming Interface (API), definitions and functionality provided for the individual algorithms.

2.4 Matlab Integration

In addition to the Motor Control Library Set for the MPC564xL library modules, the Bit Accurate Models (BAM) are delivered in the installation package. These models can be used in the Matlab Simulink Toolbox to model the behaviour of each function in real implementation. Each model consists of these three files:

1. <libID>_<funcID>.BAM (e.g. GFLIB_Acos.BAM): Contains the Bit Accurate Model (BAM)) which can be included in the user Matlab Simulink model and refers to the S-function C file, and the S-function executable file.

2. <libID>_<funcID>_SF.c (e.g. GFLIB_Acos_SF.c): The S-function C file which calls a simple legacy function during the simulation.

3. <libID>_<funcID>_SF.mexw32 (e.g. GFLIB_Acos_SF.mexw32): Contains the compiled MATLAB S-function executable file created from the function C source file.

To include the Bit Accurate Model in the user Simulink model, simply copy the BAM into the Simulink model. Note that the BAM parameters need to be properly set up. The structure of the Matlab files delivered with the Motor Control Library Set for the MPC564xL is depicted in Fig. 2-2.

Figure 2-2: MCLib Matlab File Structure.

2.5 Installation

The Motor Control Library Set for the MPC564xL is delivered as a single executable file. To install the Motor Control Library Set for the MPC564xL on a user computer, it is necessary to run the installation file and follow these steps:

1. Accept the license agreement

Figure 2-3: MCLib installation - step 1. Highlighted "releaseID" identifies the actual release number, which is MCLIB for MPC564xL

2. Select the destination directory

Figure 2-4: MCLib installation - step 2. Highlighted "releaseID" identifies the actual release number, which is MCLIB for MPC564xL

3. Select Finish to end the installation

Figure 2-5: MCLib installation - step 3. Highlighted "releaseID" identifies the actual release number, which is MCLIB for MPC564xL

2.6 Library File Structure

After a successful installation, the Motor Control Library Set for the MPC564xL is added by default into the "C:\Program Files\Freescale\MCLIB for MPC564xL" subfolder. This folder will contain other nested subfolders and files required by the Motor Control Library Set for the MPC564xL , as shown in Fig 2-6.

Figure 2-6: MCLib directory structure.

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

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

In order to integrate the Motor Control Library Set for the MPC564xL into a new Green Hills compiler based project, the steps described in section 2.7 must be performed. In order to integrate the Motor Control Library Set for the MPC564xL into a new Wind River Compiler based project, the steps described in section 2.8 must be performed. Similarly, for integration of the Motor Control Library Set for the MPC564xL into a new CodeWarrior based project, the steps described in section 2.9 must be performed. The header file structure of the Motor Control Library Set for the MPC564xL is depicted in 2-7.

Figure 2-7: MCLib File Structure.

2.7 Library Integration into a Green Hills Multi Development Environment

The Motor Control Library Set for the MPC564xL is added into a new Greenhills Multi project using the following steps:

1. Open a new empty C project in the Greenhills Multi IDE. See the Greenhills Multi user manual for instructions.

2. Once you have successfully created and opened a new C project, right click on the project file *.gpj in the GHS Multi Project Manager. Select *<Set Build Options...>* from the pop-up menu, as shown in Fig. 2-8

Figure 2-8: Project build options.

3. In the *<Basic Options>* tab of the *<Build Options>* window, expand section *<Project>* and double click on the item *<Include Directories (-I)>*. Here, the directory where the builder should look for all the project header files, including the library header files, shall be specified as shown in Fig. 2-9. Considering default settings, the following path shall be added: *"C:\Program Files\Freescale\MCLIB for MPC564xL\include"*.

4. While still in the *<Project>* section, double click on *<Libraries (-l)>* and enter a path and a pre-compiled library object file into the dialogue box, as shown in Fig. 2-10.

Figure 2-9: Adding a path to the library header files.

Considering default settings, you should add *"C:\Program Files\Freescale\MCLIB for MPC564xL\lib\MCLIB_for_MPC564xL_v0.91.a"*

At this point, the Motor Control Library Set for the MPC564xL is linked with the user project file, and hence the library functions can be exploited and flawlessly compiled/linked with the user application.

5. In order to use the library functions, the library master header files must be included into the application source code, as shown in Fig. 2-11. This is done using the pre-processor directive *#include "<libID>.h"*, where <libID> can be *gflib* or *gdflib* or *gmclib*, depending on which library is to be employed.

   The master header files contain several additional header files that are needed for the Motor Control Library Set for the MPC564xL integration into any user application. They include the "SWLIBS_Typedefs.h" header file which contains all general purpose data type definitions, "SWLIBS_Inlines.h" containing all general maths functions that are designed as inline functions, and "SWLIBS_Defines.h" containing common macro definitions. Secondly, each master header file includes all library specific header files, each one containing the function prototypes and special data type definitions required by the corresponding source module.

Figure 2-10: Adding a path to a library object file.

Figure 2-11: Opening the C editor for editing the application source code.

Figure 2-12: Using the pre-processor directive to include library master header files.

Figure 2-13: Example of a GHS based application with integrated MCLib.

2.8 Library Integration into a Wind River Compiler Environment

As the Wind River Compiler is provided with the Eclipse based IDE called Wind River Workbench, the following instruction will describe the steps needed to add the Motor Control Library Set for the MPC564xL into a new project. All described steps assume that there is an existing Wind River Workbench project (MPC564xL_sample_application name is used in the pictures below).

In order to integrate the Motor Control Library Set for the MPC564xL into an existing Wind River Workbench project, it is necessery to provide the Wind River Workbench IDE with access paths to the Motor Control Library files. The following files shall be added to the user project:

- Library binary files located in the directory

  "C:\Program Files\Freescale\MCLIB for MPC564xL\lib" (note: this is the default location and may be modified during library installation)

- Header files located in the directory

  "C:\Program Files\Freescale\MCLIB for MPC564xL\include" (note: this is the default location and may be modified during library installation)

To add these files, choose your project in *Project Explorer* in the left tab, and from the Project menu select *Properties* as described in Fig. 2-14.

Figure 2-14: Selection of the project property in the Wind River Workbench.

In the *Properties* window, select *Build Properties* from the left-hand list and choose the *Build Path* tab as described in Fig. 2-15.

Figure 2-15: Selecting the include paths in the Wind River Workbench.

By selecting the *Add..* buton, add the directories, where the builder should look for all the header files, including the library file. Considering the default settings, the paths shown in Fig. 2-16 and Fig. 2-17 shall be added.

Figure 2-16: Including the header files directory to the Wind River Workbench.

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

Figure 2-17: Including the library file directory to the Wind River Workbench.

The master header files contain several additional header files that are needed for the Motor Control Library Set for the MPC564xL integration into any user application. They include the "SWLIBS_Typedefs.h" header file which contains all general purpose data type definitions, "SWLIBS_Inlines.h" containing all general maths functions that are designed as inline functions, and "SWLIBS_Defines.h" containing common macro definitions. Secondly, each master header file includes all library specific header files, each one containing the function prototypes and special data type definitions required by the corresponding source module.

After completing this procedure, the Wind River Workbench project shall be ready for using the Motor Control Library Set for the MPC564xL.

2.9 Library Integration into a Freescale CodeWarrior Development Environment

The release provides two library versions for the CodeWarrior environment. The library versions are different in the compilation options that were set while the library was compiled:

- the library file with the suffix: *.PPCEABI.V.SC.a*, compilation options used: Power Architecture Embedded ABI, VLE instructions set, char type is signed
- the library file with the suffix: *.PPCEABI.V.UC.a*, compilation options used: Power Architecture EABI, VLE instruction set, char type is unsigned

The selection of the library version should follow those compilation options set for the CodeWarrior project. If the Motor Control Library Set for the MPC564xL options and project options are different, warnings would be reported.

In order to use the Motor Control Library Set for the MPC564xL within a CodeWarrior GUI environment, it is necessary to provide the CodeWarrior GUI with information on access paths and to add the appropriate Motor Control Library Set for the MPC564xL files to the project. The following files shall be added to the project:

- Library binary files located in the directory

  "C:\Program Files\Freescale\MCLIB for MPC564xL\lib" (note: this is the default location and may be modified during library installation)

- Header files located in the directory

  "C:\Program Files\Freescale\MCLIB for MPC564xL\include" (note: this is the default location and may be modified during library installation)

An example of the procedure of how to add Motor Control Library Set for the MPC564xL files to a CodeWarrior project is described in the step-by-step instructions below:

1. Create a new or open an existing CodeWarrior project, for which Motor Control Library Set for the MPC564xL use is intended.
2. Add the library binary files and the header files to the CodeWarrior project. Files can be added by the use of mouse drag-and-drop or by selecting the "Add Files" action from the mouse right button menu in the project window. The CodeWarrior GUI may open a new window with information on new paths added to the project Access Paths list.
3. Now the Motor Control Library Set for the MPC564xL is ready for use.

After completing the procedure, the CodeWarrior project window may look like that in Figure 2-18

Figure 2-18: MCLib integration into a CodeWarrior project.

2.10 MC Library Testing

In order to validate the implementation of the Motor Control Library Set for the MPC564xL , the comparison of results from the MATLAB Reference Model and outputs from the tested library function is used. To ensure the Motor Control Library Set for the MPC564xL precison, two test methods are used:

- Matlab Simulink Toolbox based testing (refer to 2.10.1 for more details).
- Target-in-loop based testing (refer to 2.10.2 for detailed information).

The Fig. 2-19 shows the testing principle:

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

Results from the UUT and Reference Model are saved in the final report, together with the calculated error which is simply the difference between the output value from the Reference Model and the output value from the UUT, recalculated to an equivalent precision +/-1LSB in 16-bit arithmetic.

Figure 2-19: Principle of MC library testing.

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

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

Each function is considered tested if the required accuracy during deterministic, stochastic and boundary tests has been achieved. The following two subchapters 2.10.1 and 2.10.2 describe the differences between MCLIB testing based on BAM models and in-target-loop testing based on SFIO models.

2.10.1 MC Library Testing based on the MATLAB Simulink Toolbox

An example of the testing principle based on the BAM is depicted in the Clark transformation function (Fig. 2-20). The Bit Accurate Model contains the binary MEX-file built from the GMCLIB_Clark function using the MATLAB compiler. This file is called inside the BAM model, see Fig. 2-22. the Reference Model of the Clark transformation is not included in the MATLAB Simulink Toolbox and hence its mathematical representation had to be created. A detailed scheme of the Clark RM is in Fig. 2-21.

Figure 2-20: Testing of the GMCLIB_Clark function based on the MATLAB Simulink Toolbox.

Figure 2-21: Reference Model of the GMCLIB_Clark function.

Figure 2-22: Bit Accurate Model parameters of the Clark transformation.

2.10.2 MC Library In-Target-Loop Testing based on the SFIO Toolbox

The testing method in Fig. 2-23 is the similar to that described in the previous chapter with exception that the BAM model is replaced by the SFIO model. The SFIO Toolbox realizes the bridge between Matlab and the Embedded target. During testing, the function GMCLIB_Clark is called directly from the application running on the target MCU. Unlike testing based on MATLAB, the in-target-loop method verifies that the implementation of the Motor Control Library Set for the MPC564xL functions works correctly on the target MCU. Moreover, the SFIO application running on the processor is used to measure performance of the functions.

Figure 2-23: In-Target-Loop testing example based on the SFIO Toolbox.

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

CHAPTER 3: GFLIB

3.1 Function API Overview

- tMclib GFLIB_Sin (tMclib s32In, const GFLIB_SINTLR_T *const pParam)
  Sine function - Taylor polynomial approximation
- tMclib GFLIB_Cos (tMclib s32In, const GFLIB_COSTLR_T *const pParam)
  Cosine function - Minimax polynomial approximation
- tMclib GFLIB_Tan (tMclib s32In, const GFLIB_TANTLR_T *const pParam)
  Tangent function - Rational polynomial approximation
- tMclib GFLIB_Asin (tMclib s32In, const GFLIB_ASIN_TAYLOR_T *const pParam)
  Arcsine function - Square root based polynomial approximation
- tMclib GFLIB_Acos (tMclib s32In, const GFLIB_ACOS_TAYLOR_T *const pParam)
  Arccosine function - Square root based polynomial approximation
- tMclib GFLIB_Atan (tMclib s32In, const GFLIB_ATAN_TAYLOR_T *const pParam)
  Arctangent function - Rational polynomial approximation
- tMclib GFLIB_AtanYX (tMclib s32InY, tMclib s32InX)
  The function calculates the angle between the positive x-axis and the direction of a vector given by the (x, y) coordinates
- tMclib GFLIB_AtanYXShifted (tMclib s32InY, tMclib s32InX, const GFLIB_ATANYXSHIFTED_-T *const pParam)
  The function calculates the angle of two sine waves shifted in phase to each other
- tMclib GFLIB_Sqrt (register tMclib s32In)
  Square root function
- tMclib GFLIB_Sign (tMclib s32In)
  Sign function
- tMclib GFLIB_Lut1D (tMclib s32In, const GFLIB_LUT1D_T *const pParam)
  The function performs a linear interpolation over an arbitrary data set
- tMclib GFLIB_Lut2D (tMclib s32In1, tMclib s32In2, const GFLIB_LUT2D_T *const pPa-ram)
  The function performs a bilinear interpolation over an arbitrary data set
- tMclib GFLIB_Hyst (tMclib s32In, GFLIB_HYST_T * pParam)

Hysteresis function

- tMclib GFLIB_Ramp (tMclib s32In, GFLIB_RAMP_T * pParam)
  The function calculates the up/down ramp with the step increment/decrement defined in the pParam structure

- tMclib GFLIB_Limit (tMclib s32In, const GFLIB_LIMIT_T *const pParam)
  The function limits the input value to the defined upper and lower limits

- tMclib GFLIB_LowerLimit (tMclib s32In, const GFLIB_LOWERLIMIT_T *const pParam)
  The function limits the input value by the defined lower limit

- tMclib GFLIB_UpperLimit (tMclib s32In, const GFLIB_UPPERLIMIT_T *const pParam)
  The function 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)
  The function limits the magnitude of the input vector

- tMclib GFLIB_IntegratorTR (tMclib s32In, GFLIB_INTEGRATOR_TR_T * pParam)
  The function calculates a discrete implementation of the integrator (sum), discretized using a trapezoidal (Bilinear) transformation

- tMclib GFLIB_ControllerPIr (tMclib s32InErr, GFLIB_CONTROLLER_PI_R_T * pParam)
  Recurrent form Proportional-Integral controller without integrator anti-windup functionality

- tMclib GFLIB_ControllerPIrAW (tMclib s32InErr, GFLIB_CONTROLLER_PIAW_R_T * pParam)
  Recurrent form Proportional-Integral controller with implemented integrator anti-windup functionality

- tMclib GFLIB_ControllerPIp (tMclib s32InErr, GFLIB_CONTROLLER_PI_P_T * pParam)
  The function calculates the parallel form of the Proportional-Integral (PI) controller

- tMclib GFLIB_ControllerPIpAW (tMclib s32InErr, GFLIB_CONTROLLER_PIAW_P_T * pParam)
  The function calculates the parallel form of the Proportional-Integral (PI) controller with implemented integral anti-windup functionality

3.2 GFLIB_Sin

3.2.1 Declaration:

```
tMclib GFLIB_SinANSIC(tMclib 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 approximation coefficients. Using the function alias GFLIB_Sin, default coefficients are used.
tMclib	s32In	in	Input argument is a single precision floating point number that contains an angle in radians between $[-\pi, \pi]$.

3.2.4 Return:

The function returns the sine of the input argument as a single precision floating point number.

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 the calculation of a standard trigonometric *sine* function $\sin(x)$, using the floating point optimized minimax polynomial approximation. The following equation describes the chosen approach of calculating the *sine* function:

$$\sin(s32In) = s32In + s32In^3 \cdot a_2 + s32In^5 \cdot a_1 + s32In^7 \cdot a_0 \tag{3.1}$$

where $s32In$ is the input angle. Equation (3.1) can be rewritten into the following form:

$$\sin(s32In) = s32In + s32In^3 \cdot (a_2 + (s32In^2 \cdot (a_1 + s32In^2 \cdot a_0)) \tag{3.2}$$

The floating point optimized minimax approximation is chosen as sufficient in order to achieve the best ratio between calculation accuracy and execution speed. Optimized floating

point minimax approximation of *sine* function reaches the best precision in the $\left[-\frac{\pi}{2}, \frac{\pi}{2}\right]$ range. To calculate the values in the interval $\left[-\pi, -\frac{\pi}{2}\right)$ and $\left(\frac{\pi}{2}, \pi\right]$ the following equation can be used:

$$\sin(s32In) = \sin(\pi - s32In) \tag{3.3}$$

for interval $\left(\frac{\pi}{2}, \pi\right]$

$$\sin(s32In) = -\sin(\pi + s32In) \tag{3.4}$$

for interval $\left[-\pi, -\frac{\pi}{2}\right)$ Equations (3.3) and (3.4) can be transformed to the following formula:

$$\sin(s32In) = \begin{cases} sin(\pi + s32In) & \text{if} \quad -\pi \leq \text{s32In} < -\frac{\pi}{2} \\ \sin(s32In) & \text{if} \quad -\frac{\pi}{2} \leq \text{s32In} \leq \frac{\pi}{2} \\ \sin(\pi - s32In) & \text{if} \quad \frac{\pi}{2} < \text{s32In} < \pi \end{cases} \tag{3.5}$$

The floating point optimized approximation coefficients used for GFLIB_Sin calculation are noted in table 3-2.

Table 3-2: Approximation polynomial coefficients:

a_0	a_1	a_2
-0.000195168955	0.00833220803	-0.166666567325592

Fig. 3-1 depicts a floating point *sine* function generated from Matlab and the approximated value of the *sine* function obtained from GFLIB_Sin, plus their difference. The course of calculation accuracy as a function of the input angle can be observed from this figure.

3.2.6 Note:

The input angle (s32In) is in single precision floating point format considering the input angle directly in radians. The polynomial coefficients are stored in a locally defined structure with three members, the call of which is masked by the function alias GFLIB_Sin. The polynomial coefficients can be calculated by the user, and in such a case the full GFLIB_SinANSIC function call with a pointer to the newly defined coefficients shall be used instead of the function alias. As during the computation the irrational value of π is used for subtraction, the correction constant is used to increase the calculation precision in boundary intervals. This correction constant is equal to the difference of π computed in double and in single precision.

3.2.7 Reentrancy:

The function is reentrant.

3.2.8 Code Example:

Figure 3-1: sin(x) vs. GFLIB_Sin(s32In)

```
#include "gflib.h"

tFloat s32Angle;
tFloat s32Output;

void main(void)
{
    // input angle = 1.5707963 => pi/2
    s32Angle  = (tFloat)(1.5707963);

    // output should be 1
    s32Output = GFLIB_Sin(s32Angle);
}
```

3.2.9 Performance:

Table 3-3: GFLIB_Sin function performance

Code size [bytes] GHS/CW/DIAB	90/104/110
Data size [bytes] GHS/CW/DIAB	12/12/12
Execution clock cycles max [clk] GHS/CW/DIAB	47/53/62
Execution clock cycles avg [clk] GHS/CW/DIAB	44/49/56
Execution clock cycles min [clk] GHS/CW/DIAB	41/45/52

3.3 GFLIB_Cos

3.3.1 Declaration:

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

3.3.2 Alias:

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

3.3.3 Arguments:

Table 3-4: Function parameters

Type	Name	Dir.	Description
const GFLIB_COSTLR_T *const	pParam	in	Pointer to an array of approximation coefficients. Using the function alias GFLIB_Cos, default coefficients are used.
tMclib	s32In	in	Input argument is a single precision floating point number that contains an angle in radians between $[-\pi, \pi]$.

3.3.4 Return:

The function returns the cosine of the input argument as a single precision floating point number.

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 the calculation of a standard trigonometric *cosine* function $\cos(x)$, using the floating point optimized minimax polynomial approximation of the *sine* function. The following two equations describe the chosen approach of calculating the *sine* function:

$$\sin(s32In) = s32In + s32In^3 \cdot a_2 + s32In^5 \cdot a_1 + s32In^7 \cdot a_0 \tag{3.6}$$

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

where $a_0 \ldots a_2$ are coefficients of the approximation polynomial

Equations (3.6) and (3.7) can be further rewritten into the following form:

$$\cos(s32In) = (s32In + \frac{\pi}{2}) + (s32In + \frac{\pi}{2})^3 \cdot (a_2 + (s32In + \frac{\pi}{2})^2 \cdot (a_1 + (s32In + \frac{\pi}{2})^2 \cdot a_0)) \quad (3.8)$$

The floating point optimized minimax approximation is chosen as sufficient in order to achieve the best ratio between calculation accuracy and execution speed. Optimized floating point minimax approximation of the *sine* function reaches the best precision in the $\left[-\frac{\pi}{2}, \frac{\pi}{2}\right]$ range. To calculate the values in the interval $\left[-\pi, -\frac{\pi}{2}\right)$ and $\left(\frac{\pi}{2}, \pi\right]$ the following equation can be used:

$$\sin(s32In) = \sin(\pi - s32In) \quad (3.9)$$

for interval $\left(\frac{\pi}{2}, \pi\right]$

$$\sin(s32In) = -\sin(\pi + s32In) \quad (3.10)$$

for interval $\left[-\pi, -\frac{\pi}{2}\right)$

$$\cos(-s32In) = \cos(s32In) \quad (3.11)$$

Considering (3.7) and (3.11) , the previously defined interval $\left[-\frac{\pi}{2}, \frac{\pi}{2}\right]$ is transformed to the interval $[-\pi, 0]$, and intervals $\left(\frac{\pi}{2}, \pi\right]$ and $\left[-\pi, -\frac{\pi}{2}\right)$ are transformed to the interval $(0, \pi]$. Equations (3.7) , (3.9) and (3.10) are finally transformed to the following formula:

$$\cos(s32In) = \begin{cases} \cos(s32In + \frac{\pi}{2}) & \text{if} \quad -1 \leq \text{s32In} < 0 \\ \cos(\frac{\pi}{2} - s32In) & \text{if} \quad 0 \leq \text{s32In} < 1 \end{cases} \quad (3.12)$$

The floating point optimized approximation coefficients used for GFLIB_Cos calculation are noted in table 3-5.

Table 3-5: Approximation polynomial coefficients:

a_0	a_1	a_2
-0.000195168955	0.00833220803	-0.166666567325592

Fig. 3-2 depicts a floating point *cosine* function generated from Matlab and the approximated value of the *cosine* function obtained from GFLIB_Cos, plus their difference. The course of calculation accuracy as a function of the input angle can be observed from this figure.

3.3.6 Note:

The input angle (s32In) is in single precision floating point format considering the input angle directly in radians. The polynomial coefficients are stored in a locally defined structure with three members, the call of which is masked by the function alias GFLIB_Cos. The polynomial coefficients can be calculated by the user, and in such a case the full GFLIB_CosANSIC function call with a pointer to the newly defined coefficients shall be used instead of the function alias. As during the computation the irrational value of $\frac{\pi}{2}$ is used for subtraction, the correction constant is used to increase the calculation precision in boundary intervals. This correction constant is equal to the difference of $\frac{\pi}{2}$ computed in double and in single precision.

Figure 3-2: cos(x) vs. GFLIB_Cos(s32In)

3.3.7 Reentrancy:

The function is reentrant.

3.3.8 Code Example:

```
#include "gflib.h"

tFloat s32Angle;
tFloat s32Output;

void main(void)
{
    // input angle = 0.785398163
    s32Angle  = (tFloat)(0.785398163);

    // output should be 1
    s32Output = GFLIB_Cos(s32Angle);
}
```

3.3.9 Performance:

Table 3-6: GFLIB_Cos function performance

Code size [bytes] GHS/CW/DIAB	104/98/78
Data size [bytes] GHS/CW/DIAB	12/12/12
Execution clock cycles max [clk] GHS/CW/DIAB	60/57/40
Execution clock cycles avg [clk] GHS/CW/DIAB	44/39/38
Execution clock cycles min [clk] GHS/CW/DIAB	36/38/37

3.4 GFLIB_Tan

3.4.1 Declaration:

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

3.4.2 Alias:

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

3.4.3 Arguments:

Table 3-7: Function parameters

Type	Name	Dir.	Description
const GFLIB_TANTLR_T *const	pParam	in	Pointer to an array of approximation coefficients. Using the function alias GFLIB_Tan, default coefficients are used.
tMclib	s32In	in	Input argument is a single precision floating point number that contains an angle in radians between $(-\pi, \pi)$.

3.4.4 Return:

The function returns $\tan(s32In)$ as a single precision floating point number.

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 the calculation of a standard trigonometric *tangent* function $\tan(x)$, using the rational 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:

$$\tan(x) \equiv \frac{\sin(x)}{\cos(x)} \tag{3.13}$$

Because both $\sin(x)$ and $\cos(x)$ are defined in 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 $n = \pm 1, \pm 3, \pm 5 \ldots$. The graph of $\tan(x)$ is shown in Fig. 3-3.

Figure 3-3: Course of the function GFLIB_Tan

As can be seen from Fig. 3-3, tangent values are identical for angles in the ranges:

1. $\left[-\frac{\pi}{2}, 0\right)$ and $\left[\frac{\pi}{2}, \pi\right)$
2. $\left(-\pi, -\frac{\pi}{2}\right)$ and $\left[0, \frac{\pi}{2}\right)$

Moreover, it can be observed from Fig. 3-3 that the course of the $\tan(x)$ function output for angles in interval 1. is identical, but with the opposite sign, to output for angles in interval 2. Also, the following equation (3.14) is valid for the *tangent* function:

$$\tan(x) = \frac{1}{\tan(\frac{\pi}{2} - x)} \text{for} \frac{\pi}{4} < x < \frac{\pi}{2} \tag{3.14}$$

As the approximation reaches the best precision in range $\left[-\frac{\pi}{4}, \frac{\pi}{4}\right)$, the range reduction is applied to whole interval dividing it into four sub-intervals also using the (3.15) equation:

$$\tan(-x) = -\tan(x) \tag{3.15}$$

$$\tan(s32In) = \begin{cases} tan(s32In) & \text{if} \quad 0 \leq \text{s32In} < \frac{\pi}{4} \\ \frac{1}{\tan(\frac{\pi}{2} - s32In)} & \text{if} \quad \frac{\pi}{4} \leq \text{s32In} < \frac{\pi}{2} \\ -\frac{1}{\tan(s32In - \frac{\pi}{2})} & \text{if} \quad \frac{\pi}{2} \leq \text{s32In} < \frac{3\pi}{4} \\ \tan(\pi - s32In) & \text{if} \quad \frac{3\pi}{4} \leq \text{s32In} \leq \pi \end{cases} \tag{3.16}$$

Finally due to central symmetry of the $\tan$ function around the $\frac{3\pi}{4}$ point, the (3.16) is transformed to (3.17) equation:

$$\tan(s32In) = \begin{cases} tan(s32In) & \text{if} \quad 0 \leq \text{s32In} < \frac{\pi}{4} \\ \frac{1}{\tan(s32In - \frac{\pi}{2})} & \text{if} \quad \frac{\pi}{4} < \text{s32In} \leq \frac{3\pi}{4} \\ \tan(\pi - s32In) & \text{if} \quad \frac{\pi}{2} < \text{s32In} \leq \frac{3\pi}{4} \end{cases} \tag{3.17}$$

The rational polynomial approximation used actually approximates the $\tan(x \cdot \frac{\pi}{4})$, thus the input value has to be multiplied by the $\frac{4}{\pi}$ coefficient.

Finally, the sign is added to the computed value in interval $(-\pi, 0)$. To keep the approximation error within range, the output value is limited to 1000 (defined by the GFLIB_TAN_LIMIT constant) and dividing by zero is handled by a special case. The floating point optimized approximation coefficients used for GFLIB_Tan calculation are noted in table 3-4.

Table 3-8: Approximation polynomial coefficients:

a_0	a_1	a_2	a_3
211.8493696641210	-12.5288887278448	269.7350131214120	-71.4145309347748

Fig. 3-4 depicts a floating point *tangent* function generated from Matlab and the approximated value of the *tangent* function obtained from GFLIB_Tan, plus their difference. The course of calculation accuracy as a function of the input angle can be observed from this figure.

3.4.6 Note:

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. The approximation precision is guaranteed for interval $-((\pi) - \frac{\pi}{1024}) < s32In < ((\pi) - \frac{\pi}{1024})$. Outside this interval, the function returns the tangent of the input angle, however, the approximation error might be outside the range.

Figure 3-4: tan(x) vs. GFLIB_Tan(s32In)

3.4.7 Reentrancy:

The function is reentrant.

3.4.8 Code Example:

```
#include "gflib.h"

tFloat s32Angle;
tFloat s32Output;

void main(void)
{
    // input angle = pi/4
    s32Angle  = FRAC32(0.78539816);

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

3.4.9 Performance:

Table 3-9: GFLIB_Tan function performance

Code size [bytes] GHS/CW/DIAB	240/270/256
Data size [bytes] GHS/CW/DIAB	16/16/16
Execution clock cycles max [clk] GHS/CW/DIAB	106/107/118
Execution clock cycles avg [clk] GHS/CW/DIAB	88/89/105
Execution clock cycles min [clk] GHS/CW/DIAB	67/70/80

3.5 GFLIB_Asin

3.5.1 Declaration:

```
tMclib GFLIB_AsinANSIC(tMclib 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 3-10: Function parameters

Type	Name	Dir.	Description
tMclib	s32In	in	Input argument is a 32-bit number that contains a single precision floating point value.
const GFLIB_ASIN_TAYLOR_T *const	pParam	in	Pointer to an array of approximation coefficients. The function alias GFLIB_Asin uses the default coefficients.

3.5.4 Return:

The function returns $\arcsin(s32In)$ as a single precision floating point number.

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 the calculation of the standard inverse trigonometric *arcsine* function $\arcsin(x)$, using the combination of a square root

based polynomial approximation and a Taylor approximation. Function $\arcsin(x)$ takes the ratio of the length of the opposite side to the length of the hypotenuse and returns the angle.

Figure 3-5: Course of the function GFLIB_Asin

The computation algorithm for $arcsin(x)$ uses the symmetry of the $arcsin(x)$ function around the point $(0, 0)$, which allows the calculation of the function values just in interval $[0, 1)$, and to calculation of the function values in the interval $[-1, 0)$, use the simple formula:

$$y_{[-1,0)} = -y_{[0,1]}, \tag{3.18}$$

where:

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

For the interval $[0, 0.4)$, the algorithm uses a Taylor polynomial approximation as follows:

$$arcsin(s32In) = s32In + a_0 \cdot s32In^3 + a_1 \cdot s32In^5 + a_2 \cdot s32In^7 \tag{3.19}$$

For the interval $[0.4, 1)$, the algorithm uses a polynomial approximation as follows.

$$arcsin(s32In) = a_4 - \sqrt{1 - s32In} \cdot (a_3 + a_2 \cdot s32In + a_1 \cdot s32In^2 + a_0 \cdot s32In^3) \tag{3.20}$$

To calculate the values of the $arcsin(x)$ function in interval $[-1, 0)$, the symmetry of the $arcsin(x)$ function is used and the value is determined by the following formula:

$$arcsin(s32In)_{[-1,0)} = -sin(s32In)_{[0,1]}, \tag{3.21}$$

Taylor polynomial approximation coefficients used for GFLIB_Asin calculation in interval $[0, 0.4)$, are noted in table 3-11.

Table 3-11: Approximation polynomial coefficients:

a_0	a_1	a_2
0.16666666666	0.075	0.044642857

Table 3-12: Approximation polynomial coefficients:

a_0	a_1	a_2	a_3	a_4
-0.018725300	0.074261000	-0.212114380	1.570732421	1.570796326

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

In table 3-12 there are the approximation coefficients used for GFLIB_Asin calculation in interval $[0.4, 1)$.

Fig. 3-6 depicts the floating point *arcsine* function generated from Matlab, the approximated value of the *arcsine* function obtained from the GFLIB_Asin function, and their difference.

3.5.6 Note:

The input value is assumed to be in the interval $[-1, 1]$. The function uses data stored in the gflibAsinCoef structure. This is supplied to the GFLIB_AsinANSIC function as the argument which is masked in the function alias GFLIB_Asin. The polynomial coefficients can be calculated by the user and in such case the GFLIB_AsinANSIC function shall be called with a pointer to the newly defined coefficients used, instead of the function alias.

3.5.7 Reentrancy:

The function is reentrant.

3.5.8 Code Example:

```
#include "gflib.h"

tFloat s32Input;
tFloat s32Angle;

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

    // output should be 1.570796 => pi/2
    s32Angle = GFLIB_Asin(s32Input);
}
```

3.5.9 Performance:

Table 3-13: GFLIB_Asin function performance

Code size [bytes] GHS/CW/DIAB	182/170/142
Data size [bytes] GHS/CW/DIAB	32/32/32
Execution clock cycles max [clk] GHS/CW/DIAB	139/127/119
Execution clock cycles avg [clk] GHS/CW/DIAB	110/107/91
Execution clock cycles min [clk] GHS/CW/DIAB	90/75/61

3.6 GFLIB_Acos

3.6.1 Declaration:

```
tMclib GFLIB_AcosANSIC(tMclib 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 3-14: Function parameters

Type	Name	Dir.	Description
tMclib	s32In	in	Input argument is a 32-bit number that contains a single precision floating point value.
const GFLIB_ACOS_TAYLOR_T *const	pParam	in	Pointer to an array of approximation coefficients. The function alias GFLIB_Acos uses the default coefficients.

3.6.4 Return:

The function returns $\arccos(s32In)$ as a single precision floating point number.

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 the calculation of the standard inverse trigonometric *arccosine* function $\arccos(x)$, using the square root based 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 computation algorithm for $arccos(x)$ uses the symmetry of the $arcsin(x)$ function around the point $(0, 0)$, which allows the calculation of the function values just in interval $[0, 1)$, and to calculate the function values in the interval $[-1, 0)$ use the simple formula:

$$y_{[-1,0)} = -y_{[0,1]}, \tag{3.22}$$

where:

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

Figure 3-7: Course of the function GFLIB_Acos

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

For the interval $[0, 1]$, the algorithm uses a polynomial approximation as follows:

$$arcsin(s32In) = a_4 - \sqrt{1 - s32In} \cdot (a_3 + a_2 \cdot s32In + a_1 \cdot s32In^2 + a_0 \cdot s32In^3) \tag{3.23}$$

To calculate the values of the $arcsin(x)$ function in interval $[-1, 0)$, the symmetry of the $arcsin(x)$ function is used and the value is determined by the following formula:

$$arcsin(s32In)_{[-1,0)} = -arcsin(s32In)_{[0,1]}, \tag{3.24}$$

The $arccos(x)$ values are then calculated by the simple formula:

$$arccos(x) = \pi/2 - arcsin(x), \tag{3.25}$$

Polynomial approximation coefficients used for GFLIB_Acos calculation are noted in table 3-15.

Table 3-15: Approximation polynomial coefficients:

a_0	a_1	a_2	a_3	a_4
-0.018725300	0.074261000	-0.212114380	1.570732421	1.570796326

The functions *arcsine* and *arccosine* are similar, therefore the GFLIB_Acos function uses the same polynomial coefficients as the GFLIB_Asin function.

Fig. 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.

3.6.6 Note:

The input value is assumed to be in the interval $[-1, 1]$. The function uses data stored in the gflibAcosCoef structure. This is supplied to the GFLIB_AcosANSIC function as the second

Figure 3-8: acos(x) vs. GFLIB_Acos(s32In)

argument which is masked in the function alias GFLIB_Acos. In that case the default polynomial coefficients are used. The polynomial coefficients can be calculated by the user and in such case the GFLIB_AcosANSIC function shall be called with a pointer to the newly defined coefficients used, instead of the function alias.

3.6.7 Reentrancy:

The function is reentrant.

3.6.8 Code Example:

```
#include "gflib.h"

tFloat s32Input;
tFloat s32Angle;

void main(void)
{
    // input s32Input = 0
    s32Input  = (tFloat)(0);

    // output should be 1,570796 => pi/2
    s32Angle = GFLIB_Acos(s32Input);
}
```

3.6.9 Performance:

Table 3-16: GFLIB_Acos function performance

Code size [bytes] GHS/CW/DIAB	130/124/110
Data size [bytes] GHS/CW/DIAB	20/20/20
Execution clock cycles max [clk] GHS/CW/DIAB	130/109/123
Execution clock cycles avg [clk] GHS/CW/DIAB	117/106/120
Execution clock cycles min [clk] GHS/CW/DIAB	101/90/110

3.7 GFLIB_Atan

3.7.1 Declaration:

```
tMclib GFLIB_AtanANSIC(tMclib 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 3-17: Function parameters

Type	Name	Dir.	Description
const GFLIB_ATAN_TAYLOR_T *const	pParam	in	Pointer to an array of rational polynomial coefficients. Using the function alias GFLIB_Atan, default coefficients are used.
tMclib	s32In	in	Input argument is a single precision floating point number between $\left(-2^{128}, 2^{128}\right)$.

3.7.4 Return:

The function returns the atan of the input argument as a single precision floating point number that contains the angle in radians between $\left(-\frac{\pi}{2}, \frac{\pi}{2}\right)$.

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 the calculation of a standard trigonometric *arctangent* function $\arctan(x)$, using the rational 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 Fig. 3-9.

As can be observed from Fig. 3-9, the value of the $\arctan(x)$ function for interval $(-\infty, 0)$ is identical, but with the opposite sign, to the output of the function for $[0, \infty)$. Therefore, the approximation of the *arctangent* function over the entire defined range of input values can be simplified to the approximation for a ratio in the range $[0, \infty)$, and then, depending on the input sign, the result will be negated. In order to increase the accuracy of approximation without the need for a higher order polynomial, the interval $[0, \infty)$ is further divided into two intervals: $[0, 1]$

Figure 3-9: Course of the function GFLIB_Atan

and $(1, \infty)$. For the first interval $[0, 1]$ the rational polynomial approximation directly uses the input value, while for the second interval $(1, \infty)$ the reciprocal input parameter is used as the input value for rational polynomial approximation, and the output value is subtracted from $\frac{\pi}{2}$.

The GFLIB_Atan function uses the rational polynomial approximation as defined in the following equation:

$$s32Out = \frac{s32In \cdot (a_1 + s32In^2(a_2 + a_3 \cdot s32In^2))}{(a_4 + s32In^2(a_5 + s32In^2(a_6 + s32In^2)))} \tag{3.26}$$

The rational polynomial approximation coefficients used for GFLIB_Atan calculation, are noted in table 3-18.

Table 3-18: Rational polynomial approximation coefficients

a_1	a_2	a_3	a_4	a_5	a_6
48.70107004	49.53262637	9.40604244	65.76631639	21.58787406	48.70107004

Fig. 3-10 depicts a floating point *arctangent* function generated from Matlab and the approximated value of the *arctangent* function obtained from GFLIB_Atan, plus their difference. The course of calculation accuracy as a function of the input value can be observed from this figure.

3.7.6 Note:

The input value is assumed to be in the interval $(-2^{128}, 2^{128})$. The approximation coefficients are stored in a locally defined structure, the call of which is masked by the function alias GFLIB_Atan. The polynomial coefficients can be calculated by the user and in such a case the full GFLIB_AtanANSIC function call with a pointer to the newly defined coefficients shall be used instead of the function alias.

3.7.7 Reentrancy:

The function is reentrant.

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

3.7.8 Code Example:

```
#include "gflib.h"

tFloat s32Input;
tFloat s32Angle;

void main(void)
{
    // input ratio = 1
    s32Input  = (tFloat)(1);

    // output angle should be 0.7853981634 => pi/4
    s32Angle = GFLIB_Atan(s32Input);
}
```

3.7.9 Performance:

Table 3-19: GFLIB_Atan function performance

Code size [bytes] GHS/CW/DIAB	136/126/136
Data size [bytes] GHS/CW/DIAB	24/24/24
Execution clock cycles max [clk] GHS/CW/DIAB	111/105/111
Execution clock cycles avg [clk] GHS/CW/DIAB	89/96/92
Execution clock cycles min [clk] GHS/CW/DIAB	67/88/78

3.8 GFLIB_AtanYX

3.8.1 Declaration:

```
tMclib GFLIB_AtanYXANSIC(tMclib s32InY, tMclib s32InX)
```

3.8.2 Alias:

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

3.8.3 Alias:

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

3.8.4 Arguments:

Table 3-20: Function parameters

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

3.8.5 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.

3.8.6 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 one, `s32InX`, is the abscissa (the x coordinate). Both the input parameters are assumed to be in the single precision floating point range of $(-2^{128}, 2^{128})$. The computed angle is in the range of $[-\pi, \pi]$. The counter-clockwise direction is assumed to be positive, and thus a positive angle will be computed if the provided ordinate (`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, and if necessary the final angle addition is prepared or the sign of the input value is corrected. In the second step, the vector ordinate is divided by the vector abscissa (y/x) to obtain the tangent value of the angle offset. The angle

offset is computed by applying the ordinary arc tangent function. The sum of the angle addition and the angle offset within a single quarter form the angle to be computed. The function will return 0 if both input arguments are 0.

3.8.7 Note:

The function calls the GFLIB_Atan function. The computed value is within the range of $[-\pi, \pi]$.

3.8.8 Reentrancy:

The function is reentrant.

3.8.9 Code Example:

```
#include "gflib.h"

tFloat s32InY;
tFloat s32InX;
tFloat s32Ang;

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

    // Output angle should be 45 deg = PI/4 rad = 0.7853981634
    s32Ang = GFLIB_AtanYX(s32InY, s32InX);
}
```

3.8.10 Performance:

Table 3-21: GFLIB_AtanYX function performance

Code size [bytes] GHS/CW/DIAB	190/166/176
Data size [bytes] GHS/CW/DIAB	0/0/0
Execution clock cycles max [clk] GHS/CW/DIAB	177/160/199
Execution clock cycles avg [clk] GHS/CW/DIAB	159/141/176
Execution clock cycles min [clk] GHS/CW/DIAB	20/36/45

3.9 GFLIB_AtanYXShifted

3.9.1 Declaration:

```
tMclib GFLIB_AtanYXShiftedANSIC(tMclib s32InY, tMclib 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 3-22: Function parameters

Type	Name	Dir.	Description
tMclib	s32InY	in	The value of the first signal, assumed to be $sin(\theta)$.
tMclib	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 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:

$$\begin{aligned} y &= sin(\theta) \\ x &= sin(\theta + \Delta\theta) \end{aligned} \tag{3.27}$$

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 (see below). 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 with regards to execution time and accuracy. In order to use the function, the following necessary steps need to be completed:

- define $\Delta\theta$ and 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

The function uses the following algorithm for computing the angle:

$$\begin{aligned} b &= \frac{1}{2cos\frac{\Delta\theta}{2}}(y+x) \\ a &= \frac{1}{2sin\frac{\Delta\theta}{2}}(x-y) \\ \theta &= \text{AtanYX}(b,a) - (\Delta\theta/2 - \theta_{Offset}) \end{aligned} \tag{3.28}$$

where:

- x, y are respectively, the `s32InX`, and `s32InY`
- θ is the angle to be computed by the function, see the previous equation
- $\Delta\theta$ is the phase difference between the x, y signals, see the previous equation
- a, b intermediate variables
- θ_{Offset} is the additional phase shift in radians, the computed angle will be $\theta + \theta_{Offset}$

The algorithm can be easily justified by proving the trigonometric identity:

$$\tan(\theta + \Delta\theta) = \frac{(y+x)cos\frac{\Delta\theta}{2}}{(x-y)sin\frac{\Delta\theta}{2}} \tag{3.29}$$

The function initialization parameters can be calculated as shown in the following Matlab code:

```
function [KY, KX, THETAADJ] = atanyxshiftedpar(dthdeg, thoffsetdeg)
// ATANYXSHIFTEDPAR calculation of parameters for atanyxshifted() function
//
// [KY, KX, NY, NX, THETAADJ] = atanyxshiftedpar(dthdeg, thoffsetdeg)
//
// dthdeg = phase shift (delta theta) between sine waves in degrees
// thoffsetdeg = angle offset (theta offset) in degrees
// KY - multiplication coefficient of y signal
// KX - multiplication coefficient of x signal
```

```
// THETAADJ - adjusting angle in radians, scaled from [-pi, pi) to [-1, 1)

    if (dthdeg < -180) || (dthdeg >= 180)
        error('atanyxshiftedpar: dthdeg out of range');
    end
    if (thoffsetdeg < -180) || (thoffsetdeg >= 180)
        error('atanyxshiftedpar: thoffsetdeg out of range');
    end

    dth2 = ((dthdeg/2)/180*pi);
    thoffset = (thoffsetdeg/180*pi);
    KY = 1/(2*cos(dth2));
    KX = 1/(2*sin(dth2));
    THETAADJ = dthdeg/2 - thoffsetdeg;

    if THETAADJ >= 180
        THETAADJ = THETAADJ - 360;
    elseif THETAADJ < -180
        THETAADJ = THETAADJ + 360;
    end

    THETAADJ = THETAADJ/180;

return;
```

While applying the function, some general guidelines should be considered as stated below. At some values of the phase shift, and particularly at a phase shift approaching -180, 0 or 180 degrees, the algorithm may become numerically unstable, causing any error, contributed by input signal imperfections or through finite precision arithmetic, to be magnified significantly. Therefore, some care should be taken to avoid error wherever possible. The detailed error analysis of the algorithm is beyond the scope of this documentation, however, general guidelines are provided. There are several sources of error in the function:

- error of the supplied signal values due to the finite resolution of the AD conversion
- error contributed by higher order harmonics appearing in the input signals
- computational error of the multiplication due to the finite length of registers
- error of the phase shift $\Delta\theta$ representation in the finite precision arithmetic and in the values
- error due to differences in signal amplitudes

It should be noted that the function requires both signals to have the same amplitude. To minimize the output error, the amplitude of both signals should have the same amplitude. The function has been tested to be reliable at a phase shift in the range of [-179, -1] and [1, 179] degrees for perfectly sinusoidal input signals. Beyond this range, the function operates correctly, however, the output error can be beyond the guaranteed value. In a real application, an error, contributed by an AD conversion and by higher order harmonics of the input signals, should be also taken into account.

3.9.6 Note:

The function calls the GFLIB_AtanYX function. The function accuracy is guaranteed for a phase shift in the range of [-179, -1] and [1, 179] degrees at perfect input signals.

3.9.7 Reentrancy:

The function is reentrant.

3.9.8 Code Example:

```
#include "gflib.h"

tFloat s32InY;
tFloat s32InX;
tFloat s32Ang;
GFLIB_ATANYXSHIFTED_T Param;

void main(void)
{
    //dtheta = 69.33deg, thetaoffset = 10deg
    //KY = 1/(2*cos((69.33/2)/180*pi))= 0.6079089139
    //KX = 1/(2*sin((69.33/2)/180*pi))= 0.8790788409

    //THETAADJ = 10*pi/180 = 0.1745329252

    Param.s32Ky = (tFloat)(0.6079089139);
    Param.s32Kx = (tFloat)(0.8790788409);
    Param.s32ThetaAdj = (tFloat)(0.1745329252);

    // theta = 15 deg
    // Y = sin(theta) = 0.2588190
    // X = sin(theta + dtheta) = 0.9951074
    s32InY = (tFloat)(0.2588190);
    s32InX = (tFloat)(0.9951074);
    // Output angle should be 0.138888889 rad
    s32Ang = GFLIB_AtanYXShifted(s32InY, s32InX, &Param);
}
```

3.9.9 Performance:

Table 3-23: GFLIB_AtanYXShifted function performance

Code size [bytes] GHS/CW/DIAB	98/104/114
Data size [bytes] GHS/CW/DIAB	0/0/0
Execution clock cycles max [clk] GHS/CW/DIAB	221/229/253
Execution clock cycles avg [clk] GHS/CW/DIAB	210/196/225
Execution clock cycles min [clk] GHS/CW/DIAB	187/167/201

3.10 GFLIB_Sqrt

3.10.1 Declaration:

```
tMclib GFLIB_SqrtANSIC(register tMclib s32In)
```

3.10.2 Alias:

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

3.10.3 Arguments:

Table 3-24: Function parameters

Type	Name	Dir.	Description
register tMclib	s32In	in	The input value.

3.10.4 Return:

The function returns the square root of the input value. The return value is in single precision floating point format.

3.10.5 Description:

The GFLIB_Sqrt function computes the square root of the input value. The computations of square root is based on the H/W implementation of this operation directly in the MCU EFPU unit. As there is no standardized C-based transcription for this operation, this implementation of a square root operation directly uses the assembler instruction. To protect the computation of a square root from negative values, a separate condition is implemented. In such case (input value is negative), the maximum negative single precision floating point value is returned.

3.10.6 Note:

This implementation violates the MISRA-C standards due to reason of efficiency.

3.10.7 Reentrancy:

The function is reentrant.

3.10.8 Code Example:

```
#include "gflib.h"

tFloat s32In;
tFloat s32Out;

void main(void)
{
    // s32In = 0.5
    s32In  = (tFloat)(0.5);

    // s32Out should be 0.70710678
    s32Out = GFLIB_Sqrt(s32In);

}
```

3.10.9 Performance:

Table 3-25: GFLIB_Sqrt function performance

Code size [bytes] GHS/CW/DIAB	48/26/46
Data size [bytes] GHS/CW/DIAB	0/0/0
Execution clock cycles max [clk] GHS/CW/DIAB	36/27/35
Execution clock cycles avg [clk] GHS/CW/DIAB	31/26/33
Execution clock cycles min [clk] GHS/CW/DIAB	29/26/33

3.11 GFLIB_Sign

3.11.1 Declaration:

```
tMclib GFLIB_SignANSIC(tMclib s32In)
```

3.11.2 Alias:

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

3.11.3 Arguments:

Table 3-26: Function parameters

Type	Name	Dir.	Description
tMclib	s32In	in	Input argument is a single precision floating point value.

3.11.4 Return:

The GFLIB_Sign function returns the sign of the input argument.

3.11.5 Description:

The GFLIB_Sign function calculates the sign of the input argument. The function return value is computed as follows. If the input value is negative, then the return value will be set to `"-1"`, if the input value is zero, then the function returns `"0"`, otherwise if the input value is greater than zero, the return value will be `"1"`. In mathematical terms, the function works according to the equation below:

$$y_{out} = \begin{cases} 1 & \text{if } x_{in} > 0 \\ 0 & \text{if } x_{in} = 0 \\ -1 & \text{if } x_{in} < 0 \end{cases} \tag{3.30}$$

where:

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

3.11.6 Note:

The input and the output are in single precision floating point data format.

3.11.7 Reentrancy:

The function is reentrant.

3.11.8 Code Example:

```
#include "gflib.h"

tFloat s32In;
tFloat s32Out;

void main(void)
{
  s32In = -0.8635; // input value = -0.8635
  s32Out = GFLIB_Sign(s32In); // output should be -1
  s32In = 0.23635; // input value = 0.23635
  s32Out = GFLIB_Sign(s32In); // output should be 1
  s32In = 0; // input value = 0
  s32Out = GFLIB_Sign(s32In); // output should be 0
  for(;;); // loop forever
}
```

3.11.9 Performance:

Table 3-27: GFLIB_Sign function performance

Code size [bytes] GHS/CW/DIAB	28/26/30
Data size [bytes] GHS/CW/DIAB	0/0/0
Execution clock cycles max [clk] GHS/CW/DIAB	20/19/20
Execution clock cycles avg [clk] GHS/CW/DIAB	18/15/16
Execution clock cycles min [clk] GHS/CW/DIAB	14/13/13

3.12 GFLIB_Lut1D

3.12.1 Declaration:

```
tMclib GFLIB_Lut1DANSIC(tMclib 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 3-28: Function parameters

Type	Name	Dir.	Description
tMclib	s32In	in	The abscissa for which 1D interpolation is performed. Input value is in single precision floating data format.
const GFLIB_LUT1D_T *const	pParam	in	Pointer to the parameters structure with parameters of the look-up table function.

3.12.4 Return:

The function returns the interpolated value. The output value is in single precision floating point format.

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:

$$y = y_1 + \frac{y_2 - y_1}{x_2 - x_1} \cdot (x - x_1) \tag{3.31}$$

where:

- y is the interpolated value
- y_1 and y_2 are the function values at the the beginning and the end of the researched interpolation interval
- x_1 and x_2 are the values on the x-axis which determine the beginning and the end of the interpolation interval

- the x is the input value provided to the function in the `s32In` argument

The interpolating intervals are defined in the table provided by the `ps32Table` member of the parameters structure. The table contains ordinate values consecutively over the entire interpolating range. The abscissa values are assumed to be defined implicitly by a single interpolating interval length. The abscissa value is equal to the multiplication of the interpolating index and the interpolating interval length. For example, a table contains 4 interpolating points. The interpolating interval length in this example is equal to $2^{-1} = 0.5$. It should be noted that the `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 or, even, does not have to have zero in its range. A special algorithm is used to make the computation efficient, however, under some additional assumptions, as provided below:

- 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 abscissa provided for interpolation is 32 bits long

The algorithm performs the following steps:

1. Count the equidistant interpolating interval, in which the linear interpolation will be performed:

$$\texttt{fSegments} = \texttt{(tFloat)}(1 << (\texttt{s32ShamOffset} + 1)) \tag{3.32}$$

   where $fSegments$ is the interval length and the `pParam->s32ShamOffset` is the shift amount provided in the parameters structure as the member `s32ShamOffset`. The operator « represents the binary arithmetic left shift.

2. Compute the position of the interpolated values in the table:

$$\texttt{s32Intvl} = \texttt{(tS32)}((\texttt{fSegments}) \cdot (\texttt{s32In})) \tag{3.33}$$

   where `s32Intvl` is the position of the first ordinate value at the beginning of the interpolating interval, the `fSegments` is the equidistant interpolating interval and the `s32In` is the input value provided to the function as an argument. The casting to `tS32` data type is a final operation of the equation which cuts the fractional part of the value and then indicates the position of the first ordinate value in the table.

3. Compute the interpolated value by the linear interpolation between the ordinates read from the table at the start and the end of the computed interval. The computed abscissa offset is used for the linear interpolation.

$$\begin{aligned} y_1 &= (\texttt{32-bit data at address pTable}) + \texttt{s32Intvl} \\ y_2 &= (\texttt{32-bit data at address pTable}) + (\texttt{s32Intvl} + 1) \\ \Delta x &= \texttt{(tFloat)}(x \cdot 2^n) - \texttt{(tU32)}(x \cdot 2^n) \\ y &= y_1 + (y_2 - y_1) \cdot \Delta x \end{aligned} \tag{3.34}$$

   where y, $y1$ and $y2$ are, respectively, the interpolated value, the ordinate at the start of the interpolating interval, the ordinate at the end of the interpolating interval. The `pTable` is the address provided in the parameters structure `pParam->s32Table` and the `x` is the input value provided to the function as an `s32In` argument. It should be noted that due to assumption of equidistant data points, division by the interval length is avoided.

The address of the table with the data, the `pTable`, shall be defined by the parameter structure member `pParam->ps32Table`. The offset amount can be provided by the following formula:

$$\texttt{s32ShamOffset} = |n| \tag{3.35}$$

where n is the integer defining the length of the interpolating interval in the range of 1, 2, ... 29. This simple way to calculate the interpolating abscissa offset is a consequence of assuming that all interpolating interval lengths equal 2^{-n}. It should be noted that the input abscissa value can be positive or negative. If it is positive then the ordinate values are read as in the ordinary data array, that is, at or after the data pointer provided in the parameters structure `pParam->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 Warning:

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 a linear interpolation.

3.12.8 Reentrancy:

The function is reentrant.

3.12.9 Code Example:

```
#include "gflib.h"

tFloat s32In;
tFloat s32Out;
GFLIB_LUT1D_T trMyLut1D;
tFloat pTable1D[17] = {
  0.1, 0.2, 0.3, 0.256, 0.289, 0.311, 0.412, 0.468, 0.247, 0.123, 0, -0.145,
  -0.235, -0.589, 0.666, 0.789, 0.819
};

void main(void)
{
  // Setting parameters for Lut1D function
  trMyLut1D.ps32Table = &(pTable1D[0]);
  trMyLut1D.s32ShamOffset = 3;

  s32In = 0.2398; // input value = 0.2398
  s32Out = GFLIB_Lut1D(s32In,&trMyLut1D); // output should be 0.2836144
  s32In = 0.4051; // input value = 0.4051
  s32Out = GFLIB_Lut1D(s32In,&trMyLut1D); // output should be 0.4389696
  s32In = 0.9921; // input value = 0.9921
  s32Out = GFLIB_Lut1D(s32In,&trMyLut1D); // output should be 0.815208
  s32In = 0.6667; // input value = 0.6667
```

```
  s32Out = GFLIB_Lut1D(s32In,&trMyLut1D); // output should be -0.096744
  s32In = 0.73356; // input value = 0.73356
  s32Out = GFLIB_Lut1D(s32In,&trMyLut1D); // output should be -0.2113264
  for(;;); // loop forever
}
```

3.12.10 Performance:

Table 3-29: GFLIB_Lut1D function performance

Code size [bytes] GHS/CW/DIAB	72/96/78
Data size [bytes] GHS/CW/DIAB	0/0/0
Execution clock cycles max [clk] GHS/CW/DIAB	54/47/52
Execution clock cycles avg [clk] GHS/CW/DIAB	43/44/50
Execution clock cycles min [clk] GHS/CW/DIAB	35/44/50

3.13 GFLIB_Lut2D

3.13.1 Declaration:

```
tMclib GFLIB_Lut2DANSIC(tMclib s32In1, tMclib s32In2, const GFLIB_LUT2D_T *const
pParam)
```

3.13.2 Alias:

```
#define GFLIB_Lut2D(x, y, pParam_xy) \
        GFLIB_Lut2DANSIC((x), (y), (pParam_xy))
```

3.13.3 Arguments:

Table 3-30: Function parameters

Type	Name	Dir.	Description
tMclib	s32In1	in	First input variable for which 2D interpolation is performed. Input value is in single precision floating data format.
tMclib	s32In2	in	Second input variable for which 2D interpolation is performed. Input value is in single precision floating data format.
const GFLIB_LUT2D_T *const	pParam	in	Pointer to the parameters structure with parameters of the two dimensional look-up table function.

3.13.4 Return:

The function returns the interpolated value. The output value is in single precision floating point format.

3.13.5 Description:

The GFLIB_Lut2D function performs a two dimensional linear interpolation over a 2D table of data. The following interpolation formulas are used:

$$f[x, y_1] = f[x_1, y_1] + \frac{f[x_2, y_1] - f[x_1, y_1]}{x_2 - x_1} \cdot (x - x_1) \tag{3.36}$$

$$f[x, y_2] = f[x_1, y_2] + \frac{f[x_2, y_2] - f[x_1, y_2]}{x_2 - x_1} \cdot (x - x_1) \tag{3.37}$$

$$f[x,y] = f[x,y_1] + \frac{f[x,y_2] - f[x,y_1]}{y_2 - y_1} \cdot (y - y_1) \tag{3.38}$$

where:

- $f[x,y]$ is the interpolated value
- $f[x,y_1]$ and $f[x,y_2]$ are the ordinate values at, respectively, the beginning and the end of the final interpolating interval
- x_1, x_2, y_1 and y_2 are the area values, respectively, the interpolated area
- the x, y are the input values provided to the function in the $s32In1$ and $s32In2$ arguments

The interpolating intervals are defined in the table provided by the `ps32Table` member of the parameters structure. The table contains ordinate values consecutively over the entire interpolating range. The abscissa values are assumed to be defined implicitly by a single interpolating interval length. The abscissa value is equal to the multiplication of the interpolating index and the interpolating interval length. For example, a table contains 4 interpolating points. The interpolating interval length in this example is equal to $2^{-1} = 0.5$. It should be noted that the `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 or, even, does not have to have zero in its range. A special algorithm is used to make the computation efficient, however, under some additional assumptions as provided below:

- 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 abscissa provided for interpolation is 32 bits long

The bilinear interpolation performs the following steps:

1. Count the equidistant interpolating intervals, in which the bilinear interpolation will be performed:

$$\begin{aligned} \texttt{fSegmentsX} &= \texttt{(tFloat)}(1 << (\texttt{s32ShamOffset1} + 1)) \\ \texttt{fSegmentsY} &= \texttt{(tFloat)}(1 << (\texttt{s32ShamOffset2} + 1)) \end{aligned} \tag{3.39}$$

   where `fSegmentsX` and `fSegmentsY` are the interval lengths, the `pParam->s32ShamOffset1` `pParam->s32ShamOffset2` are the shift amounts provided in the parameters structure as the members `s32ShamOffset1`, `s32ShamOffset2`. The operator "«" represents the binary arithmetic left shift. The explicit casting to `tFloat` data type is required because parameters `s32ShamOffset1`,2 are integer numbers which have to be stored as float variables, and it then indicates the position of the first interpolated value in the X-axis and the Y-axis in the table.

2. Compute the position of the interpolated values in the table:

$$\begin{aligned} \texttt{s32IntvlX} &= \texttt{(tS32)}((\texttt{fSegmentsX}) \cdot (\texttt{s32In1})) \\ \texttt{s32IntvlY} &= \texttt{(tS32)}((\texttt{fSegmentsY}) \cdot (\texttt{s32In2})) \end{aligned} \tag{3.40}$$

   where `s32IntvlX` and `s32IntvlY` are the coordinates of the first ordinate value at the beginning of the interpolating interval, the `fSegmentsX` and `fSegmentsY` are the equidistant

interpolating intervals in the both axes and the `s32In1`, `s32In2` are the input values provided to the function as arguments. The casting to `tS32` data type is a final operation of the equation which cuts the fractional part of the value and then indicates the position of the first interpolated value in the X-axis and the Y-axis in the table.

3. Count the number of the interpolation samples in the X-axis:

$$\texttt{s32Xrange} = \texttt{((1«pParam->s32ShamOffset1+1)+1)} \tag{3.41}$$

4. Compute the pointer to the first interpolated value z_11 in the table:

$$\texttt{psfIntvl} = \texttt{(32-bit data at address pTable)} + \texttt{s32IntvlX} + (\texttt{s32Xrange} \cdot \texttt{s32IntvlY}) \tag{3.42}$$

where the `pTable` is the address provided in the parameters structure `pParam->s32Table`.

5. Compute the interpolated value by the linear interpolation in the X-axis of the y_1 value between the ordinates `x_1` and `x_2` read from the table at the start and the end of the computed interval in the X-axis. The computed abscissa offset is used for the linear interpolation:

$$z_1 = z_{11} + \frac{(x - x_1) \cdot (z_{21} - z_{11})}{x_2 - x_1} \tag{3.43}$$

reduced to (division by the interval length is avoided):

$$z_1 = z_{11} + (z_{21} - z_{11}) \cdot (\texttt{(tFloat)}(x \cdot 2^n) - \texttt{(tU32)}(x \cdot 2^n)) \tag{3.44}$$

interpretation in C language is following:

$$\begin{aligned} \texttt{fDX} &= \texttt{s32In1} \cdot \texttt{fSegmentsX-s32IntvlX} \\ \texttt{fZ1} &= (\texttt{*psfIntvl}) + (((\texttt{*(psfIntvl+1))-(*psfIntvl)}) \cdot (\texttt{fDX})) \end{aligned} \tag{3.45}$$

6. The same computation as shown above in the X-axis by the y_2 value:

$$z_2 = z_{12} + \frac{(x - x_1) \cdot (z_{22} - z_{12})}{x_2 - x_1} \tag{3.46}$$

reduced to (division by the interval length is avoided):

$$z_2 = z_{12} + (z_{22} - z_{12}) \cdot (\texttt{(tFloat)}(x \cdot 2^n) - \texttt{(tU32)}(x \cdot 2^n)) \tag{3.47}$$

interpretation in C language is the following:

$$\begin{aligned} \texttt{fZ2} = (\texttt{*(psfIntvl+s32Xrange)}) + (((\texttt{*(psfIntvl+s32Xrange+1))} \\ \texttt{-(*(psfIntvl+s32Xrange)))} \cdot (\texttt{fDX})) \end{aligned} \tag{3.48}$$

7. Final linear interpolation in the Y-axis:

$$z = z_1 + \frac{(y - y_1) \cdot (z_2 - z_1)}{y_2 - y_1} \tag{3.49}$$

reduced to (division by the interval length is avoided):

$$z = z_1 + (z_2 - z_1) \cdot (\texttt{(tFloat)}(y \cdot 2^n) - \texttt{(tU32)}(y \cdot 2^n)) \tag{3.50}$$

interpretation in C language is the following:

$$\texttt{return(fZ1+((fZ2-fZ1)} \cdot \texttt{(s32In2} \cdot \texttt{fSegmentsY-s32IntvlY)))} \tag{3.51}$$

The shift amounts shall be provided in the parameters structure `pParam->s32ShamOffset1` and `pParam->s32ShamOffset2`. The address of the table with the data, the `pTable`, shall be defined by the parameter structure member `pParam->ps32Table`. The offset amounts `s32ShamOffset_12` can be provided by the following formulas:

$$\begin{aligned} \texttt{s32ShamOffset1} &= |n_1| \\ \texttt{s32ShamOffset2} &= |n_2| \end{aligned} \tag{3.52}$$

where n_1 and n_2 are the integers defining the length of the interpolating interval in the range of 1, 2, ... 29. This simple way to calculate the interpolating abscissa offsets is the consequence of assuming that all interpolating interval lengths equal 2^{-n1} and 2^{-n2}. It should be noted that the input abscissa values can be positive or negative. If they are positive then the ordinate values are read as in the ordinary data array, that are at or after the data pointer provided in the parameters structure `pParam->ps32Table`. However, if they are negative, then the ordinate values are read from the memory, which is located behind the `pParam->ps32Table` pointer.

3.13.6 Warning:

The function does not check whether the input values are 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.

3.13.7 Note:

The function performs a bilinear interpolation.

3.13.8 Reentrancy:

The function is reentrant.

3.13.9 Code Example:

```
#include "gflib.h"

tFloat s32In1;
tFloat s32In2;
tFloat s32Out;
GFLIB_LUT2D_T trMyLut2D;
const tFloat IGBT_2D[289] =
{
  0.15,0.20,0.25,0.30,0.35,0.40,0.45,0.50,0.55,
  0.60,0.65,0.70,0.75,0.80,0.85,0.90,0.95,  // f[X0,Y0-Y16]
  0.15,0.20,0.25,0.30,0.35,0.40,0.45,0.50,0.55,
  0.60,0.65,0.70,0.75,0.80,0.85,0.90,0.95,  // f[X1,Y0-Y16]
  0.15,0.20,0.25,0.30,0.35,0.40,0.45,0.50,0.55,
  0.60,0.65,0.70,0.75,0.80,0.85,0.90,0.95,  // f[X2,Y0-Y16]
  0.15,0.20,0.25,0.30,0.35,0.40,0.45,0.50,0.55,
  0.60,0.65,0.70,0.75,0.80,0.85,0.90,0.95,  // f[X3,Y0-Y16]
  0.15,0.20,0.25,0.30,0.35,0.40,0.45,0.50,0.55,
  0.60,0.65,0.70,0.75,0.80,0.85,0.90,0.95,  // f[X4,Y0-Y16]
  0.15,0.20,0.25,0.30,0.35,0.4259,0.89565,0.50,
```

```
  0.55,0.60,0.65,0.70,0.75,0.80,0.85,0.90,0.95, // f[X5,Y0-Y16]
  0.15,0.20,0.25,0.30,0.35,0.68742,0.12345,0.50,
  0.55,0.60,0.65,0.70,0.75,0.80,0.85,0.90,0.95,// f[X6,Y0-Y16]
  0.15,0.20,0.25,0.30,0.35,0.40,0.45,0.50,0.55,
  0.60,0.65,0.70,0.75,0.80,0.85,0.90,0.95,  // f[X7,Y0-Y16]
  0.15,0.20,0.25,0.30,0.35,0.40,0.45,0.50,0.55,
  0.60,0.65,0.70,0.75,0.80,0.85,0.90,0.95,  // f[X8,Y0-Y16]
  0.15,0.20,0.25,0.30,0.35,0.40,0.45,0.50,0.55,
  0.60,0.65,0.70,0.75,0.80,0.85,0.90,0.95,  // f[X9,Y0-Y16]
  0.15,0.20,0.25,0.30,0.35,0.40,0.45,0.50,0.55,
  0.60,0.65,0.70,0.75,0.80,0.85,0.90,0.95,  // f[X10,Y0-Y16]
  0.15,0.20,0.25,0.30,0.35,0.40,0.45,0.50,0.55,
  0.60,0.65,0.70,0.75,0.80,0.85,0.90,0.95,  // f[X11,Y0-Y16]
  0.15,0.20,0.25,0.30,0.35,0.40,0.45,0.50,0.55,
  0.60,0.65,0.70,0.75,0.80,0.85,0.90,0.95,  // f[X12,Y0-Y16]
  0.15,0.20,0.25,0.30,0.35,0.40,0.45,0.50,0.55,
  0.60,0.65,0.70,0.75,0.80,0.85,0.90,0.95,  // f[X13,Y0-Y16]
  0.15,0.20,0.25,0.30,0.35,0.40,0.45,0.50,0.55,
  0.60,0.65,0.70,0.75,0.80,0.85,0.90,0.95,  // f[X14,Y0-Y16]
  0.15,0.20,0.25,0.30,0.35,0.40,0.45,0.50,0.55,
  0.60,0.65,0.70,0.75,0.80,0.85,0.90,0.95,  // f[X15,Y0-Y16]
  0.15,0.20,0.25,0.30,0.35,0.40,0.45,0.50,0.55,
  0.60,0.65,0.70,0.75,0.80,0.85,0.90,0.95   // f[X16,Y0-Y16]
};

void main(void)
{
  GFLIB_LUT2D_T Param; // parameters of the values
  // The interpolating interval length = 2^-(n+1) = 2^-4 = 0.0625, so the
  // X_offset for first input and Y_offset for second input is 3. The
  // interpolating table pointer is defined to point to the first
  // element (IGBT_2D[0]).
  Param.ps32Table = &(IGBT_2D [0]);
  Param.s32ShamOffset1 = 3; // offset for s32In1
  Param.s32ShamOffset2 = 3; // offset for s32In2

  s32In1 = 0.3368;
  s32In2 = 0.6781;
  s32Out = GFLIB_Lut2D(s32In1, s32In2, &Param); // output should be ~ s32Out = 0.419434
  s32In1 = 0.7326;
  s32In2 = 0.7326;
  s32Out = GFLIB_Lut2D(s32In1, s32In2, &Param); // output should be ~ s32Out = 0.736092
  s32In1 = 0.1265;
  s32In2 = 0.8655;
  s32Out = GFLIB_Lut2D(s32In1, s32In2, &Param); // output should be ~ s32Out = 0.251199
  s32In1 = 0.9654;
  s32In2 = 0.2480;
  s32Out = GFLIB_Lut2D(s32In1, s32In2, &Param); // output should be ~ s32Out = 0.092231
  s32In1 = 0.6235;
  s32In2 = 0.4356;
  s32Out = GFLIB_Lut2D(s32In1, s32In2, &Param); // output should be ~ s32Out = 0.648817
  s32In1 = 0.2222;
  s32In2 = 0.9678;
  s32Out = GFLIB_Lut2D(s32In1, s32In2, &Param); // output should be ~ s32Out = 0.327752
  s32In1 = 0.1999;
  s32In2 = 0.2987;
  s32Out = GFLIB_Lut2D(s32In1, s32In2, &Param); // output should be ~ s32Out = 0.309917
  s32In1 = 0.8945;
```

```
  s32In2 = 0.4356;
  s32Out = GFLIB_Lut2D(s32In1, s32In2, &Param); // output should be ~ s32Out = 0.865604
  for(;;); // loop forever
}
```

3.13.10 Performance:

Table 3-31: GFLIB_Lut2D function performance

Code size [bytes] GHS/CW/DIAB	146/210/166
Data size [bytes] GHS/CW/DIAB	0/0/0
Execution clock cycles max [clk] GHS/CW/DIAB	70/86/81
Execution clock cycles avg [clk] GHS/CW/DIAB	68/81/79
Execution clock cycles min [clk] GHS/CW/DIAB	65/77/79

3.14 GFLIB_Hyst

3.14.1 Declaration:

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

3.14.2 Alias:

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

3.14.3 Arguments:

Table 3-32: Function parameters

Type	Name	Dir.	Description
tMclib	s32In	in	Input value is in single precision floating data format.
GFLIB_HYST_T *	pParam	in/out	Pointer to the structure with parameters and states of the hysteresis function. Arguments of the structure contain a single precision floating point values.

3.14.4 Return:

The function returns the value of the hysteresis output, which is equal to either `s32OutValOn` or `s32OutValOff` depending on the value of the input and the state of the function output in the previous calculation step. The output value is in single precision floating point format.

3.14.5 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 the 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 `s32HystOn`, then the output value is equal to `s32OutValOn`. On the other hand, when the input value is lower than the lower threshold `s32HystOff`, then the output value is equal to `s32OutValOff`. When the input value is between these two threshold values then the output retains its value (the previous state).

$$s32OutState(k) = \begin{cases} s32OutValOn & \text{if} \quad s32In \geq s32HystOn \\ s32OutValOff & \text{if} \quad s32In \leq s32HystOff \\ s32OutState(k-1) & \text{otherwise} \end{cases} \tag{3.53}$$

A graphical description of GFLIB_Hyst functionality is shown in Fig. 3-11.

Figure 3-11: Hysteresis function

3.14.6 Warning:

For correct functionality, the threshold `s32HystOn` value must be greater than the `s32HystOff` value.

3.14.7 Note:

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

3.14.8 Reentrancy:

The function is reentrant.

3.14.9 Code Example:

```
#include "gflib.h"

tFloat s32In;
tFloat s32Out;
GFLIB_HYST_T trMyHyst = GFLIB_HYST_DEFAULT; // Definition of one hysteresis instance

void main(void)
{
  // Setting parameters for hysteresis
  trMyHyst.s32HystOn = 0.1289;
  trMyHyst.s32HystOff = -0.3634;
  trMyHyst.s32OutValOn = 0.589;
  trMyHyst.s32OutValOff = -0.123;
  trMyHyst.s32OutState = -0.3333;
```

```
  s32In  = -0.41115; // input value = -0.41115
  s32Out = GFLIB_Hyst(s32In,&trMyHyst); // output should be -0.123
  s32In  = 0.0165; // input value = 0.0165
  s32Out = GFLIB_Hyst(s32In,&trMyHyst); // output should be -0.3333
  s32In  = 0.295; // input value = 0.295
  s32Out = GFLIB_Hyst(s32In,&trMyHyst); // output should be 0.589
  for(;;); // loop forever
}
```

3.14.10 Performance:

Table 3-33: GFLIB_Hyst function performance

Code size [bytes] GHS/CW/DIAB	36/36/32
Data size [bytes] GHS/CW/DIAB	0/0/0
Execution clock cycles max [clk] GHS/CW/DIAB	37/33/28
Execution clock cycles avg [clk] GHS/CW/DIAB	34/28/26
Execution clock cycles min [clk] GHS/CW/DIAB	27/27/24

3.15 GFLIB_Ramp

3.15.1 Declaration:

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

3.15.2 Alias:

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

3.15.3 Arguments:

Table 3-34: Function parameters

Type	Name	Dir.	Description
tMclib	s32In	in	Input argument representing the desired output value. Input value is in single precision floating data format.
GFLIB_RAMP_T *	pParam	in/out	Pointer to the ramp parameters structure. Arguments of the structure contain single precision floating point values.

3.15.4 Return:

The function returns a value in single precision floating point format, which represents the actual ramp output. This value can be also described as a Ramp state value increased/decreased by a slope in order to achieve the desired input value.

3.15.5 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.

The ramp function is performed as follows:

$$s32Out = \begin{cases} (s32State + s32RampUp) & \text{if } (s32In \geq s32State) \\ (s32State - s32RampDown) & \text{if } (s32In < s32State) \\ (s32In) & \text{if } ((s32State + s32RampUp) \geq s32In) \\ (s32In) & \text{if } ((s32State - s32RampDown) \leq s32In) \end{cases} \tag{3.54}$$

Functionality of the implemented ramp algorithm can be explained with the use of Fig. 3-12

Figure 3-12: GFLIB_Ramp functionality

3.15.6 Note:

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

3.15.7 Reentrancy:

The function is reentrant.

3.15.8 Code Example:

```
#include "gflib.h"

tFloat s32In;
tFloat s32Out;
GFLIB_RAMP_T trMyRamp = GFLIB_RAMP_DEFAULT; // definition of one ramp instance

void main(void)
{
  // Setting parameters for ramp
  trMyRamp.s32State = 0.1625;
  trMyRamp.s32RampUp = 0.5;
  trMyRamp.s32RampDown = 0.11;
```

```
  s32In = 0.005; // input value = 0.005
  s32Out = GFLIB_Ramp(s32In,&trMyRamp); // output should be 0.0525
  s32In = 0.8; // input value = 0.8
  s32Out = GFLIB_Ramp(s32In,&trMyRamp); // output should be 0.6625
  s32In = 0.1625; // input value = 0.1625
  s32Out = GFLIB_Ramp(s32In,&trMyRamp); // output should be 0.1625
  s32In = 0.385; // input value = 0.385
  s32Out = GFLIB_Ramp(s32In,&trMyRamp); // output should be 0.385
  for(;;); // loop forever
}
```

3.15.9 Performance:

Table 3-35: GFLIB_Ramp function performance

Code size [bytes] GHS/CW/DIAB	66/74/64
Data size [bytes] GHS/CW/DIAB	0/0/0
Execution clock cycles max [clk] GHS/CW/DIAB	48/48/37
Execution clock cycles avg [clk] GHS/CW/DIAB	38/42/32
Execution clock cycles min [clk] GHS/CW/DIAB	37/37/30

3.16 GFLIB_Limit

3.16.1 Declaration:

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

3.16.2 Alias:

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

3.16.3 Arguments:

Table 3-36: Function parameters

Type	Name	Dir.	Description
tMclib	s32In	in	Input value is in single precision floating data format.
const GFLIB_LIMIT_T *const	pParam	in	Pointer to the limits structure. Arguments of the structure contain single precision floating point values.

3.16.4 Return:

The function returns the input value, or the upper or lower limit if the input value is beyond these limits. The output value is in single precision floating point format.

3.16.5 Description:

The GFLIB_Limit function tests whether the input value is within the upper and lower limits. If so, the input value will be returned. If the input value is above the upper limit, the upper limit will be returned. 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`.

3.16.6 Note:

The function assumes that the upper limit `s32UpperLimit` is greater than the lower limit `s32LowerLimit`. Otherwise, the function returns an undefined value.

3.16.7 Reentrancy:

The function is reentrant.

3.16.8 Code Example:

```
#include "gflib.h"

tFloat s32In;
tFloat s32Out;
GFLIB_LIMIT_T trMyLimit;

void main(void)
{
  // Setting parameters for limit function
  trMyLimit.s32LowerLimit = -0.236;
  trMyLimit.s32UpperLimit = 0.859;

  s32In = -598.1256; // input value = -598.1256
  s32Out = GFLIB_Limit(s32In, &trMyLimit); // output should be -0.236
  s32In = 0.6568; // input value = 0.6568
  s32Out = GFLIB_Limit(s32In, &trMyLimit); // output should be 0.6568
  s32In = 158.99; // input value = 158.99
  s32Out = GFLIB_Limit(s32In, &trMyLimit); // output should be 0.859
  for(;;); // loop forever
}
```

3.16.9 Performance:

Table 3-37: GFLIB_Limit function performance

Code size [bytes] GHS/CW/DIAB	26/42/28
Data size [bytes] GHS/CW/DIAB	0/0/0
Execution clock cycles max [clk] GHS/CW/DIAB	34/36/25
Execution clock cycles avg [clk] GHS/CW/DIAB	21/28/18
Execution clock cycles min [clk] GHS/CW/DIAB	15/23/13

3.17 GFLIB_LowerLimit

3.17.1 Declaration:

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

3.17.2 Alias:

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

3.17.3 Arguments:

Table 3-38: Function parameters

Type	Name	Dir.	Description
tMclib	s32In	in	Input value is in single precision floating data format.
const GFLIB_-LOWERLIMIT_T *const	pParam	in	Pointer to the limits structure. Argument of the structure contains a single precision floating point value.

3.17.4 Return:

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

3.17.5 Description:

The GFLIB_LowerLimit 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 parameter `s32LowerLimit` can be found in the limits structure, supplied to the function as a pointer `pParam`.

3.17.6 Note:

The input and the output are in single precision floating point data format.

3.17.7 Reentrancy:

The function is reentrant.

3.17.8 Code Example:

```
#include "gflib.h"

tFloat s32In;
tFloat s32Out;
GFLIB_LOWERLIMIT_T trMyLowerLimit;

void main(void)
{
  // Setting parameters for lowerlimit function
  trMyLowerLimit.s32LowerLimit = -0.3658;

  s32In = -0.81254; // input value = -0.81254
  s32Out = GFLIB_LowerLimit(s32In, &trMyLowerLimit); // output should be -0.3658
  s32In = 0.81254; // input value = 0.81254
  s32Out = GFLIB_LowerLimit(s32In, &trMyLowerLimit); // output should be 0.81254
  s32In = 0; // input value = 0
  s32Out = GFLIB_LowerLimit(s32In, &trMyLowerLimit); // output should be 0
  for(;;); // loop forever
}
```

3.17.9 Performance:

Table 3-39: GFLIB_LowerLimit function performance

Code size [bytes] GHS/CW/DIAB	12/26/14
Data size [bytes] GHS/CW/DIAB	0/0/0
Execution clock cycles max [clk] GHS/CW/DIAB	17/24/15
Execution clock cycles avg [clk] GHS/CW/DIAB	15/22/13
Execution clock cycles min [clk] GHS/CW/DIAB	14/21/12

3.18 GFLIB_UpperLimit

3.18.1 Declaration:

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

3.18.2 Alias:

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

3.18.3 Arguments:

Table 3-40: Function parameters

Type	Name	Dir.	Description
tMclib	s32In	in	Input value is in single precision floating data format.
const GFLIB_-UPPERLIMIT_T *const	pParam	in	Pointer to the limits structure. Argument of the structure contains a single precision floating point value.

3.18.4 Return:

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

3.18.5 Description:

The GFLIB_UpperLimit function tests whether the input value is below the upper limit. If so, the input value will be returned. Otherwise, if the input value is above the upper limit, the upper limit will be returned. The upper limit `s32UpperLimit` can be found in the parameters structure, supplied to the function as a pointer `pParam`.

3.18.6 Note:

The input and the output are in single precision floating point data format.

3.18.7 Reentrancy:

The function is reentrant.

3.18.8 Code Example:

```
#include "gflib.h"

tFloat s32In;
tFloat s32Out;
GFLIB_UPPERLIMIT_T trMyUpperLimit;

void main(void)
{
  // Setting parameters for upperlimit function
  trMyUpperLimit.s32UpperLimit = 0.2563;

  s32In = 0.78954; // input value = 0.78954
  s32Out = GFLIB_UpperLimit(s32In, &trMyUpperLimit); // output should be 0.2563
  s32In = -0.2114; // input value = -0.2114
  s32Out = GFLIB_UpperLimit(s32In, &trMyUpperLimit); // output should be -0.2114
  s32In = 0; // input value = 0
  s32Out = GFLIB_UpperLimit(s32In, &trMyUpperLimit); // output should be 0
  for(;;); // loop forever
}
```

3.18.9 Performance:

Table 3-41: GFLIB_UpperLimit function performance

Code size [bytes] GHS/CW/DIAB	12/26/14
Data size [bytes] GHS/CW/DIAB	0/0/0
Execution clock cycles max [clk] GHS/CW/DIAB	18/27/22
Execution clock cycles avg [clk] GHS/CW/DIAB	17/22/13
Execution clock cycles min [clk] GHS/CW/DIAB	15/21/12

3.19 GFLIB_VectorLimit

3.19.1 Declaration:

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

3.19.2 Alias:

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

3.19.3 Arguments:

Table 3-42: Function parameters

Type	Name	Dir.	Description
SWLIBS_2Syst *const	pOut	out	Pointer to the structure of the limited output vector. Arguments of the structure contain single precision floating point values.
const SWLIBS_2Syst *const	pIn	in	Pointer to the structure of the input vector. Arguments of the structure contain single precision floating point values.
const GFLIB_VECTORLIMIT_T *const	pParam	in	Pointer to the parameters structure. Arguments of the structure contain single precision floating point values.

3.19.4 Return:

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

3.19.5 Description:

The GFLIB_VectorLimit function limits the magnitude of the input vector, keeping its direction unchanged. The maximum magnitude of the input vector in the following equation is defined by the letter "L". Limitation is performed as follows:

$$y_{out} = \begin{cases} \dfrac{y_{in}}{\sqrt{x_{in}^2 + y_{in}^2}} \cdot L & \text{if } \sqrt{x_{in}^2 + y_{in}^2} > L \\ y_{in} & \text{if } \sqrt{x_{in}^2 + y_{in}^2} \leq L \end{cases} \tag{3.55}$$

$$x_{out} = \begin{cases} \dfrac{x_{in}}{\sqrt{x_{in}^2 + y_{in}^2}} \cdot L & \text{if } \sqrt{x_{in}^2 + y_{in}^2} > L \\ x_{in} & \text{if } \sqrt{x_{in}^2 + y_{in}^2} \leq L \end{cases} \tag{3.56}$$

Where:

- x_{in}, y_{in} and x_{out}, y_{out} are the co-ordinates of the input and output vector, respectively
- L is the maximum magnitude of the vector

The input vector co-ordinates are defined by the structure pointed to by the `pIn` parameter, and the output vector co-ordinates shall be found in the structure pointed to by the `pOut` parameter. The maximum vector magnitude `s32Lim` is defined in the parameters structure pointed to by the `pParam` function parameter. A graphical interpretation of the function can be seen in the figure below.

Figure 3-13: Graphical interpretation of the GFLIB_VectorLimit function.

If an actual limitation occurs, the function will return the logical true `"TRUE"`, otherwise the logical false `"FALSE"` will be returned.

3.19.6 Warning:

The maximum vector magnitude in the parameters structure, the `pParam->s32Lim`, must be positive and equal to or greater than `"0"`, otherwise the result is undefined. The function does not check for the valid range of the parameter `pParam->s32Lim`.

3.19.7 Note:

The function calls the MCLIB square root routine GFLIB_Sqrt.

3.19.8 Reentrancy:

The function is reentrant.

3.19.9 Code Example:

```
#include "gflib.h"

SWLIBS_2Syst InVect={0,0};
SWLIBS_2Syst OutVect={0,0};
tBool bLim;
GFLIB_VECTORLIMIT_T trMyVectorLimit;

void main(void)
{
  InVect.s32Arg1 = 0.5; // input vector Arg1 = 0.5
  InVect.s32Arg2 = 0.5; // input vector Arg2 = 0.5
  trMyVectorLimit.s32Lim = 0.707106781; // the maximum magnitude of the input vector
  bLim = GFLIB_VectorLimit(&OutVect,&InVect,&trMyVectorLimit);
         // output vector should be Arg1 = 0.5 and Arg2 = 0.5
  InVect.s32Arg1 = 0.3; // input vector Arg1 = 0.3
  InVect.s32Arg2 = 0.3; // input vector Arg2 = 0.3
  trMyVectorLimit.s32Lim = 0.003; // the maximum magnitude of the input vector
  bLim = GFLIB_VectorLimit(&OutVect,&InVect,&trMyVectorLimit);
         // output vector should be Arg1 = 0.0021213205 and Arg2 = 0.0021213205
  InVect.s32Arg1 = 0.0002; // input vector Arg1 = 0.0002
  InVect.s32Arg2 = 0.0003; // input vector Arg2 = 0.0003
  trMyVectorLimit.s32Lim = 0.00024; // the maximum magnitude of the input vector
  bLim = GFLIB_VectorLimit(&OutVect,&InVect,&trMyVectorLimit);
         // output vector should be Arg1 = 0.000133128 and Arg2 = 0.000199692
  for(;;); // loop forever
}
```

3.19.10 Performance:

Table 3-43: GFLIB_VectorLimit function performance

Code size [bytes] GHS/CW/DIAB	126/96/90
Data size [bytes] GHS/CW/DIAB	0/0/0
Execution clock cycles max [clk] GHS/CW/DIAB	121/95/121
Execution clock cycles avg [clk] GHS/CW/DIAB	108/91/116
Execution clock cycles min [clk] GHS/CW/DIAB	79/75/90

3.20 GFLIB_IntegratorTR

3.20.1 Declaration:

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

3.20.2 Alias:

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

3.20.3 Arguments:

Table 3-44: Function parameters

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

3.20.4 Return:

The function returns a single precision floating point value, which represents the actual integrated value of the input signal.

3.20.5 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:

$$u(t) = \int_0^t e(t)\,dt \tag{3.57}$$

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

$$H(s) = \frac{U(s)}{E(s)} = \frac{1}{s} \tag{3.58}$$

Transforming equation (3.58) into a digital time domain using the Bilinear transformation, leads to the following transfer function:

$$\mathcal{Z}\{H(s)\} = \frac{U(z)}{E(z)} = \frac{T_s + T_s z^{-1}}{2 - 2z^{-1}} \tag{3.59}$$

where T_s is the sampling period of the system. The discrete implementation of the digital transfer function (3.59) is as follows:

$$u(k) = u(k-1) + e(k) \cdot \frac{T_s}{2} + e(k-1) \cdot \frac{T_s}{2} \tag{3.60}$$

3.20.6 Note:

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

3.20.7 Reentrancy:

The function is reentrant.

3.20.8 Code Example:

```
#include "gflib.h"

tFloat s32In;
tFloat 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,
    trMyIntegrator.s32C1          = (tFloat)(100e-6/2);

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

    // output should be 2.5e-5
    s32Out = GFLIB_IntegratorTR(s32In,&trMyIntegrator);
}
```

3.20.9 Performance:

Table 3-45: GFLIB_IntegratorTR function performance

Code size [bytes] GHS/CW/DIAB	36/28/30
Data size [bytes] GHS/CW/DIAB	0/0/0
Execution clock cycles max [clk] GHS/CW/DIAB	28/25/36
Execution clock cycles avg [clk] GHS/CW/DIAB	27/23/31
Execution clock cycles min [clk] GHS/CW/DIAB	26/23/26

3.21 GFLIB_ControllerPIr

3.21.1 Declaration:

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

3.21.2 Alias:

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

3.21.3 Arguments:

Table 3-46: Function parameters

Type	Name	Dir.	Description
tMclib	s32InErr	in	Input error signal to the controller as a single precision floating point value.
GFLIB_CONTROLLER_PI_R_T *	pParam	in/out	Pointer to the controller parameters structure.

3.21.4 Return:

The function returns a single precision floating point value, representing the signal to be applied to the controlled system so that the input error is forced to zero.

3.21.5 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:

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

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

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

Transforming equation (3.62) into a discrete time domain leads to the following equation:

$$u(k) = u(k-1) + e(k) \cdot CC1 + e(k-1) \cdot CC2 \tag{3.63}$$

where Kp is proportional gain, Ki 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 3-47.

Table 3-47: Calculation of coefficients CC1 and CC2 using various discretization methods.

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

3.21.6 Note:

All controller parameters and states can be reset during declaration using the GFLIB_CONTROLLER_PI_R_DEFAULT macro. As the PI controller also contains the integration part, the output result is saddled by cumulative error. To enumerate the computation error in one calculation cycle, the internal accumulator is not used for testing purposes and is replaced by output from the previous calculation step of the reference model.

3.21.7 Reentrancy:

The function is reentrant.

3.21.8 Code Example:

```
#include "gflib.h"

tFloat s32InErr;
tFloat s32Output;

GFLIB_CONTROLLER_PI_R_T trMyPI = GFLIB_CONTROLLER_PI_R_DEFAULT;
```

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

    // controller parameters
    trMyPI.s32CC1sc            = (tFloat)(0.01);
    trMyPI.s32CC2sc            = (tFloat)(0.02);

    // output should be 5e-3
    s32Output = GFLIB_ControllerPIr(s32InErr,&trMyPI);
}
```

3.21.9 Performance:

Table 3-48: GFLIB_ControllerPIr function performance

Code size [bytes] GHS/CW/DIAB	40/30/28
Data size [bytes] GHS/CW/DIAB	0/0/0
Execution clock cycles max [clk] GHS/CW/DIAB	34/27/31
Execution clock cycles avg [clk] GHS/CW/DIAB	33/25/26
Execution clock cycles min [clk] GHS/CW/DIAB	31/25/26

3.22 GFLIB_ControllerPIrAW

3.22.1 Declaration:

```
tMclib GFLIB_ControllerPIrAWANSIC(tMclib s32InErr, GFLIB_CONTROLLER_PIAW_R_T
*pParam)
```

3.22.2 Alias:

```
#define GFLIB_ControllerPIrAW(s32InErr, pParam) \
        GFLIB_ControllerPIrAWANSIC(s32InErr, pParam)
```

3.22.3 Arguments:

Table 3-49: Function parameters

Type	Name	Dir.	Description
tMclib	s32InErr	in	Input error signal to the controller in single precision floating point data format.
GFLIB_CONTROLLER_-PIAW_R_T *	pParam	in/out	Pointer to the controller parameters structure

3.22.4 Return:

The function returns a single precision floating point value, representing the signal to be applied to the controlled system so that the input error is forced to zero.

3.22.5 Description:

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

The function GFLIB_ControllerPIrAWANSIC calculates a standard recurrent form of the Proportional-Integral controller, with integral anti-windup. The continuous time domain representation of the PI controller is defined as:

$$u(t) = e(t) \cdot K_P + K_I \int_0^t e(t)\, dt \tag{3.64}$$

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

$$H(s) = \frac{U(s)}{E(s)} = \frac{K_P + sK_I}{s} \tag{3.65}$$

Transforming equation (3.65) into a discrete time domain leads to the following equation:

$$u(k) = u(k-1) + e(k) \cdot CC1 + e(k-1) \cdot CC2 \tag{3.66}$$

where Kp is proportional gain, Ki 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 the controller coefficients calculated depending on the discretization method used, as shown in table 3-50.

Table 3-50: Calculation of coefficients CC1 and CC2 using various discretization methods.

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

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

$$u_f(k) = \begin{cases} \text{s32UpperLimit} & \rightarrow \quad u_f(k) \geq \text{s32UpperLimit} \\ u_f(k) & \rightarrow \quad \text{s32LowerLimit} < u_f(k) < \text{s32UpperLimit} \\ \text{s32LowerLimit} & \rightarrow \quad u_f(k) \leq \text{s32LowerLimit} \end{cases} \tag{3.67}$$

The bounds are described by a limitation element equation (3.67). When the bounds are exceeded, the non-linear saturation characteristic will take effect and influence the dynamic behaviour. The described limitation is implemented in the internal controller accumulator (limitation during the calculation). 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_R_DEFAULT macro. As the PI controller also contains the integration part, the output result is saddled by cumulative error. To enumerate the computation error in one calculation cycle, the internal accumulator is not used for testing purposes and is replaced by output from the previous calculation step of the reference model.

3.22.7 Reentrancy:

The function is reentrant.

3.22.8 Code Example:

```
#include "gflib.h"

tFloat s32InErr;
tFloat s32Output;
```

```
GFLIB_CONTROLLER_PIAW_R_T trMyPI = GFLIB_CONTROLLER_PIAW_R_DEFAULT;

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

    // controller parameters
    trMyPI.s32CC1sc            = (tFloat)(0.01);
    trMyPI.s32CC2sc            = (tFloat)(0.02);
    trMyPI.s32UpperLimit       = (tFloat)(1.0);
    trMyPI.s32LowerLimit       = (tFloat)(-1.0);

    // output should be 5e-3
    s32Output = GFLIB_ControllerPIrAW(s32InErr,&trMyPI);
}
```

3.22.9 Performance:

Table 3-51: GFLIB_ControllerPIrAW function performance

Code size [bytes] GHS/CW/DIAB	74/62/70
Data size [bytes] GHS/CW/DIAB	0/0/0
Execution clock cycles max [clk] GHS/CW/DIAB	63/47/52
Execution clock cycles avg [clk] GHS/CW/DIAB	54/40/46
Execution clock cycles min [clk] GHS/CW/DIAB	43/36/44

3.23 GFLIB_ControllerPIp

3.23.1 Declaration:

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

3.23.2 Alias:

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

3.23.3 Arguments:

Table 3-52: Function parameters

Type	Name	Dir.	Description
tMclib	s32InErr	in	Input error signal to the controller in single precision floating format.
GFLIB_CONTROLLER_PI_P_T *	pParam	in/out	Pointer to the controller parameters structure.

3.23.4 Return:

The function returns a single precision floating point value, representing the signal to be applied to the controlled system so that the input error is forced to zero.

3.23.5 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. The PI algorithm in the continuous time domain can be described as:

$$u(t) = e(t) \cdot K_P + K_I \int_0^t e(t)\, dt \tag{3.68}$$

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

Eq. (3.68) can be described using the Laplace transformation as follows:

$$H(s) = \frac{U(s)}{E(s)} = K_P + K_I \frac{1}{s} \tag{3.69}$$

The proportional part of equation (3.69) is transformed into the discrete time domain simply as:

$$u_P(k) = K_P \cdot e(k) \tag{3.70}$$

Transforming the integral part of equation (3.69) into a discrete time domain using the Bilinear method, also known as trapezoidal approximation, leads to the following equation:

$$u_I(k) = u_I(k-1) + e(k) \cdot \frac{K_I T_s}{2} + e(k-1) \cdot \frac{K_I T_s}{2} \tag{3.71}$$

where T_s [sec] is the sampling time.

3.23.6 Note:

All controller parameters and states can be reset during declaration using the GFLIB_CONTROLLER_PI_P_DEFAULT macro. As the PI controller also contains the integration part, the output result is saddled by cumulative error. To enumerate the computation error in one calculation cycle, the integrator value from the previous step is not used for testing purposes and is replaced by the reference model integrator value in the previous step.

3.23.7 Reentrancy:

The function is reentrant.

3.23.8 Code Example:

```
#include "gflib.h"

tFloat s32InErr;
tFloat s32Output;

GFLIB_CONTROLLER_PI_P_T trMyPI = GFLIB_CONTROLLER_PI_P_DEFAULT;

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

    // controller parameters
    trMyPI.s32PropGain          = (tFloat)(0.01);
    trMyPI.s32IntegGain         = (tFloat)(0.02);
    trMyPI.s32IntegPartK_1      = 0;
```

```
    // output should be 1.5e-2
    s32Output = GFLIB_ControllerPIp(s32InErr,&trMyPI);
}
```

3.23.9 Performance:

Table 3-53: GFLIB_ControllerPIp function performance

Code size [bytes] GHS/CW/DIAB	42/34/46
Data size [bytes] GHS/CW/DIAB	0/0/0
Execution clock cycles max [clk] GHS/CW/DIAB	43/25/39
Execution clock cycles avg [clk] GHS/CW/DIAB	41/23/37
Execution clock cycles min [clk] GHS/CW/DIAB	34/23/37

3.24 GFLIB_ControllerPIpAW

3.24.1 Declaration:

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

3.24.2 Alias:

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

3.24.3 Arguments:

Table 3-54: Function parameters

Type	Name	Dir.	Description
tMclib	s32InErr	in	Input error signal to the controller is a single precision floating point data type.
GFLIB_CONTROLLER_-PIAW_P_T *	pParam	in/out	Pointer to the controller parameters structure.

3.24.4 Return:

The function returns a single precision floating point value, representing the signal to be applied to the controlled system so that the input error is forced to zero.

3.24.5 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 controller output. The PI algorithm in the continuous time domain

can be described as:

$$u(t) = e(t) \cdot K_P + K_I \int_0^t e(t)\, dt \tag{3.72}$$

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

Eq. (3.72) can be described using the Laplace transformation as follows:

$$H(s) = \frac{U(s)}{E(s)} = K_P + K_I \frac{1}{s} \tag{3.73}$$

The proportional part of equation (3.73) is transformed into the discrete time domain simply as:

$$u_P(k) = K_P \cdot e(k) \tag{3.74}$$

Transforming the integral part of equation (3.73) into a discrete time domain using the Bilinear method, also known as trapezoidal approximation, leads to the following equation:

$$u_I(k) = u_I(k-1) + e(k) \cdot \frac{K_I T_s}{2} + e(k-1) \cdot \frac{K_I T_s}{2} \tag{3.75}$$

where T_s [sec] is the sampling time.

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 (3.76). 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 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.

3.24.6 Note:

All controller parameters and states can be reset during declaration using the GFLIB_CONTROLLER_PIAW_P_DEFAULT macro. As the PI controller contains also the integration part, the output result is saddled by cumulative error. To enumerate the computation error in one calculation cycle, the integrator value from the previous step is not used for testing purposes and is replaced by the reference model integrator value in the previous step.

3.24.7 Reentrancy:

The function is reentrant.

3.24.8 Code Example:

```
#include "gflib.h"

tFloat s32InErr;
tFloat s32Output;

GFLIB_CONTROLLER_PIAW_P_T trMyPI = GFLIB_CONTROLLER_PIAW_P_DEFAULT;

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

    // controller parameters
    trMyPI.s32PropGain         = (tFloat)(0.01);
    trMyPI.s32IntegGain        = (tFloat)(0.02);
    trMyPI.s32IntegPartK_1     = 0;
    trMyPI.s32UpperLimit       = (tFloat)(1.0);
    trMyPI.s32LowerLimit       = (tFloat)(-1.0);

    // output should be 1.5e-2
    s32Output = GFLIB_ControllerPIpAW(s32InErr,&trMyPI);
}
```

3.24.9 Performance:

Table 3-55: GFLIB_ControllerPIpAW function performance

Code size [bytes] GHS/CW/DIAB	134/118/98
Data size [bytes] GHS/CW/DIAB	0/0/0
Execution clock cycles max [clk] GHS/CW/DIAB	76/71/68
Execution clock cycles avg [clk] GHS/CW/DIAB	69/64/61
Execution clock cycles min [clk] GHS/CW/DIAB	48/51/48

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, tMclib s32AngularVel, const GMCLIB_DECOUPLINGPMSM_-T *const pParam)
  This function 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 $(\alpha - \beta)$. Arguments of the structure contain single precision floating point values.
const SWLIBS_3Syst *const	pIn	in	Pointer to the structure containing data of the three-phase stationary system $(sA - sB - sC)$. Arguments of the structure contain single precision floating point values.

4.2.4 Return:

Function returns no value.

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 in full range single precision floating point format.

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)
{
  tr32Abc.s32Arg1 = 0.707106781;  // input phase A = sin(45) = 0.707106781
  tr32Abc.s32Arg2 = 0.258819045;  // input phase B = sin(45 + 120) = 0.258819045
  tr32Abc.s32Arg3 = -0.965925826; // input phase C = sin(45 - 120) = -0.965925826
  GMCLIB_Clark(&tr32AlBe,&tr32Abc); // output should be:
                                    // tr32AlBe.s32Arg1 ~ alpha = 0.707106781
                                    // tr32AlBe.s32Arg2 ~ beta  = 0.707106781
  tr32Abc.s32Arg1 = 0.0; // input phase A = 0.0
  tr32Abc.s32Arg2 = 0.5; // input phase B = 0.5
  tr32Abc.s32Arg3 = 0.9; // input phase C = 0.9
  GMCLIB_Clark(&tr32AlBe,&tr32Abc); // output should be:
                                    // tr32AlBe.s32Arg1 ~ alpha = 0
                                    // tr32AlBe.s32Arg2 ~ beta = -0.230940108
  tr32Abc.s32Arg1 = 0.35; // input phase A = 0.35
  tr32Abc.s32Arg2 = 0.8;  // input phase B = 0.8
  tr32Abc.s32Arg3 = 0.42; // input phase C = 0.42
  GMCLIB_Clark(&tr32AlBe,&tr32Abc); // output should be:
                                    // tr32AlBe.s32Arg1 ~ alpha = 0.35
                                    // tr32AlBe.s32Arg2 ~ beta = 0.219393102
  for(;;); // loop forever
}
```

4.2.9 Performance:

Table 4-2: GMCLIB_Clark function performance

Code size [bytes] GHS/CW/DIAB	36/42/28
Data size [bytes] GHS/CW/DIAB	0/0/0
Execution clock cycles max [clk] GHS/CW/DIAB	37/31/22
Execution clock cycles avg [clk] GHS/CW/DIAB	27/29/20
Execution clock cycles min [clk] GHS/CW/DIAB	24/29/20

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 $(sA - sB - sC)$. Arguments of the structure contain single precision floating point values.
const SWLIBS_2Syst *const	pIn	in	Pointer to the structure containing data of the two-phase stationary orthogonal system $(\alpha - \beta)$. Arguments of the structure contain single precision floating point values.

4.3.4 Return:

Function returns no value.

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 in full range single precision floating point format.

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)
{
  tr32AlBe.s32Arg1 = 0.707106781; // input phase alpha = 0.707106781
  tr32AlBe.s32Arg2 = 0.258819045; // input phase beta = 0.258819045
  GMCLIB_ClarkInv(&tr32Abc,&tr32AlBe); // output should be:
                                       // tr32Abc.s32Arg1 ~ phA = 0.707106781
                                       // tr32Abc.s32Arg2 ~ phB = -0.129409523
                                       // tr32Abc.s32Arg3 ~ phC = -0.577697258
  tr32AlBe.s32Arg1 = 0;   // input phase alpha = 0
  tr32AlBe.s32Arg2 = 0.5; // input phase beta  = 0.5
  GMCLIB_ClarkInv(&tr32Abc,&tr32AlBe); // output should be:
                                       // tr32Abc.s32Arg1 ~ phA = 0
                                       // tr32Abc.s32Arg2 ~ phB = 0.433012702
                                       // tr32Abc.s32Arg3 ~ phC = -0.433012702
  tr32AlBe.s32Arg1 = 0.35; // input phase alpha = 0.35
  tr32AlBe.s32Arg2 = 0.8;  // input phase beta = 0.8
  GMCLIB_ClarkInv(&tr32Abc,&tr32AlBe); // output should be:
                                       // tr32Abc.s32Arg1 ~ phA = 0.35
                                       // tr32Abc.s32Arg2 ~ phB = 0.517820323
                                       // tr32Abc.s32Arg3 ~ phC = -0.867820323
  for(;;); // loop forever
}
```

4.3.9 Performance:

Table 4-4: GMCLIB_ClarkInv function performance

Code size [bytes] GHS/CW/DIAB	54/58/54
Data size [bytes] GHS/CW/DIAB	0/0/0
Execution clock cycles max [clk] GHS/CW/DIAB	38/36/31
Execution clock cycles avg [clk] GHS/CW/DIAB	33/35/30
Execution clock cycles min [clk] GHS/CW/DIAB	28/28/30

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:

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 in single precision floating point data format.

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  = (tFloat)(0.866025403);
    tr32Angle.s32Arg2  = (tFloat)(0.5);

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

    // output should be:
    // tr32Dq.s32Arg1 ~ d = 0.627880613
    // tr32Dq.s32Arg2 ~ q = 0.220479472
    GMCLIB_Park(&tr32Dq,&tr32Angle,&tr32AlBe);
}
```

4.4.9 Performance:

Table 4-6: GMCLIB_Park function performance

Code size [bytes] GHS/CW/DIAB	54/38/50
Data size [bytes] GHS/CW/DIAB	0/0/0
Execution clock cycles max [clk] GHS/CW/DIAB	50/27/55
Execution clock cycles avg [clk] GHS/CW/DIAB	48/24/53
Execution clock cycles min [clk] GHS/CW/DIAB	39/23/53

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:

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 in single precision floating point data format.

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  = (tFloat)(0.866025403);
    tr32Angle.s32Arg2  = (tFloat)(0.5);

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

    // output should be:
    // tr32AlBe.s32Arg1 ~ alpha = -0.495119387
    // tr32AlBe.s32Arg2 ~ beta  = 0.433521139
    GMCLIB_ParkInv(&tr32AlBe,&tr32Angle,&tr32Dq);
}
```

4.5.9 Performance:

Table 4-8: GMCLIB_ParkInv function performance

Code size [bytes] GHS/CW/DIAB	54/52/50
Data size [bytes] GHS/CW/DIAB	0/0/0
Execution clock cycles max [clk] GHS/CW/DIAB	41/38/56
Execution clock cycles avg [clk] GHS/CW/DIAB	40/36/52
Execution clock cycles min [clk] GHS/CW/DIAB	39/25/46

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. Arguments of the structure contain single precision floating point values.
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. Arguments of the structure contain single precision floating point values.
const GMCLIB_ELIMDCBUSRIP_T *const	pParam	in	Pointer to the parameters structure. Arguments of the structure contain single precision floating point values.

4.6.4 Return:

Function returns no value.

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) with only the 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 `pIn`, and the u_{α}^{*} and u_{β}^{*} respectively to the `s32Arg1` and `s32Arg2` members of the output structure `pOut`. It should be noted that although the modulation index (see the parameters structure `pParam`, the `s32ModIndex` member) 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.

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"

SWLIBS_2Syst f32AB;
SWLIBS_2Syst f32OutAB;
GMCLIB_ELIMDCBUSRIP_T trMyElimDcBusRip = GMCLIB_ELIMDCBUSRIP_DEFAULT;

void main(void)
{
  trMyElimDcBusRip.s32ModIndex = 0.866025404; // inverse modulation coefficient for
                                              // standard space vector modulation
  // Input voltage vector 15V @ angle 30deg
  f32AB.s32Arg1 = 12.99038106; // alpha component of input voltage vector = 12.99[V]
  f32AB.s32Arg2 = 7.5; // beta component of input voltage vector = 7.5[V]
  trMyElimDcBusRip.s32ArgDcBusMsr = 17; // value of "measured" DC bus voltage 17V
  GMCLIB_ElimDcBusRip(&f32OutAB,&f32AB,&trMyElimDcBusRip);
```

```
        // output should be:
        // alpha component of the output vector ~ f32OutAB.s32Arg1 = 1
        // beta component of the output vector ~ f32OutAB.s32Arg2 = 0.764140062
  // Input voltage vector 14V @ angle 54deg
  f32AB.s32Arg1 = 8.228993532; // alpha component of input voltage vector = 8.228993532[V]
  f32AB.s32Arg2 = 11.32623792; // beta component of input voltage vector = 11.32623792[V]
  trMyElimDcBusRip.s32ArgDcBusMsr = 25; // value of "measured" DC bus voltage 25V
  GMCLIB_ElimDcBusRip(&f32OutAB,&f32AB,&trMyElimDcBusRip);
        // output should be:
        // alpha component of the output vector ~ f32OutAB.s32Arg1 = 0.570121396
        // beta component of the output vector ~ f32OutAB.s32Arg2 = 0.784704782
  for(;;); // loop forever
}
```

4.6.9 Performance:

Table 4-10: GMCLIB_ElimDcBusRip function performance

Code size [bytes] GHS/CW/DIAB	160/152/156
Data size [bytes] GHS/CW/DIAB	0/0/0
Execution clock cycles max [clk] GHS/CW/DIAB	104/95/112
Execution clock cycles avg [clk] GHS/CW/DIAB	67/86/100
Execution clock cycles min [clk] GHS/CW/DIAB	92/76/82

4.7 GMCLIB_DecouplingPMSM

4.7.1 Declaration:

```
void GMCLIB_DecouplingPMSMANSIC(SWLIBS_2Syst *pUdqDec, const SWLIBS_2Syst *const
pUdq, const SWLIBS_2Syst *const pIdq, tMclib 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.
tMclib	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 L_D and L_Q coefficients (see the detailed section of the documentation).

4.7.4 Return:

Function returns no value.

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 very popular for field oriented control structures because both controllable quantities, current and voltage, are DC values. This allows employing only simple controllers to force the machine currents into the defined states.

The voltage equations of this model can be obtained by transforming the motor three phase voltage equations into a quadrature phase rotational frame, which is aligned and rotates synchronously with the rotor. Such a transformation, after some mathematical corrections, yields the following set of equations, describing the quadrature phase model of a PMSM motor, in a synchronous reference frame:

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

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:

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

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 for 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 for 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:

$$\begin{aligned} u_{d_{dec}} &= u_d - \omega_e \cdot L_q \cdot i_q \\ u_{q_{dec}} &= u_q + \omega_e \cdot L_d \cdot i_d \end{aligned} \tag{4.18}$$

4.7.6 Note:

All parameters can be reset during declaration using the GMCLIB_DECOUPLINGPMSM_-DEFAULT macro. All inputs and parameters contain single precision floating point data type values.

4.7.7 Reentrancy:

The function is reentrant.

4.7.8 Code Example:

```
#include "gmclib.h"

SWLIBS_2Syst trUsDQ;
SWLIBS_2Syst trIsDQ;
SWLIBS_2Syst trUsDecDQ;
tFloat s32We;
GMCLIB_DECOUPLINGPMSM_T trMyDecouplingPMSM = GMCLIB_DECOUPLINGPMSM_DEFAULT;

void main(void)
{
  trMyDecouplingPMSM.s32LD = 50e-3;  // LD inductance = 50mH
  trMyDecouplingPMSM.s32LQ = 100e-3; // LQ inductance = 100mH
  trUsDQ.s32Arg1 = 5.0;  // D quantity of input voltage vector 5[V]
  trUsDQ.s32Arg2 = 10.0; // Q quantity of input voltage vector 10[V]
  trIsDQ.s32Arg1 = 6.0;  // D quantity of measured current vector 6[A]
  trIsDQ.s32Arg2 = 4.0;  // Q quantity of measured current vector 4[A]
  s32We = 100.0;         // rotor angular velocity
  GMCLIB_DecouplingPMSM(&trUsDecDQ,&trUsDQ,&trIsDQ,s32We,&trMyDecouplingPMSM);
                         // output should be:
                         // trUsDecDQ.s32Arg1 ~= -15[V]
                         // trUsDecDQ.s32Arg2 ~= 70[V]
  trMyDecouplingPMSM.s32LD = 50e-3;  // LD inductance = 50mH
  trMyDecouplingPMSM.s32LQ = 100e-3; // LQ inductance = 100mH
  trUsDQ.s32Arg1 = 5.6;   // D quantity of input voltage vector 5[V]
```

```
  trUsDQ.s32Arg2 = 13.87; // Q quantity of input voltage vector 10[V]
  trIsDQ.s32Arg1 = 1.298; // D quantity of measured current vector 6[A]
  trIsDQ.s32Arg2 = 1.999; // Q quantity of measured current vector 4[A]
  s32We = 109.0;          // rotor angular velocity
  GMCLIB_DecouplingPMSM(&trUsDecDQ,&trUsDQ,&trIsDQ,s32We,&trMyDecouplingPMSM);
                           // output should be:
                           // trUsDecDQ.s32Arg1 ~= -16.1891[V]
                           // trUsDecDQ.s32Arg2 ~= 20.9441[V]
  for(;;); // loop forever
}
```

4.7.9 Performance:

Table 4-12: GMCLIB_DecouplingPMSM function performance

Code size [bytes] GHS/CW/DIAB	54/52/54
Data size [bytes] GHS/CW/DIAB	0/0/0
Execution clock cycles max [clk] GHS/CW/DIAB	62/37/63
Execution clock cycles avg [clk] GHS/CW/DIAB	60/35/58
Execution clock cycles min [clk] GHS/CW/DIAB	51/32/56

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 Fig. 4-1. Top and bottom switches work in a complementary mode; i.e., if the top switch (S_{At}) is ON, then the corresponding bottom switch (S_{Ab}) is OFF, and vice versa. Considering that value 1 is assigned to the ON state of the top switch, and value 0 is assigned to the ON state of the bottom switch, the switching vector, $[a, b, c]^T$ can be defined. Creating such a vector allows a numerical definition of all possible switching states. In a three-phase power stage configuration (as shown in Fig. 4-1), eight possible switching states (detailed in Fig. 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.

$$U_\alpha = \frac{2}{3}\left(U_a - \frac{U_b}{2} - \frac{U_c}{2}\right) \tag{4.19}$$

$$U_\beta = \frac{2}{3}\left(0 + \frac{\sqrt{3}U_b}{2} - \frac{\sqrt{3}U_c}{2}\right) \tag{4.20}$$

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.

Fig. 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 the approximation of the reference stator voltage vector U_S with an appropriate combination of the switching patterns composed of basic space vectors. A graphical explanation of this objective is shown in 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:

$$T = T_{60} + T_0 + T_{null} \tag{4.21}$$

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

$$U_s = \frac{T_{60}}{T} \cdot U_{60} + \frac{T_0}{T} \cdot U_0 \tag{4.22}$$

where T_{60} and T_0 are the respective duty-cycle ratios for which the basic space vectors U_{60} and U_0 should be applied within the time period T. T $_{null}$ is the course of time for which the null vectors O_{000} and O_{111} are applied. Those duty-cycle ratios can be calculated using equations:

$$u_\beta = \frac{T_{60}}{T} \cdot |U_{60}| \cdot \sin(60°) \tag{4.23}$$

$$u_\alpha = \frac{T_0}{T} \cdot |U_0| + \frac{u_\beta}{\tan(60°)} \tag{4.24}$$

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 $\sin(60°)$ and $\tan(60°)$ by their quantities $\frac{2}{\sqrt{3}}$ and $\sqrt{3}$, respectively, eq. (4.23) and eq. (4.24) can be rearranged for the unknown duty-cycle ratios $\frac{T_{60}}{T}$ and $\frac{T_0}{T}$:

$$\frac{T_{60}}{T} = u_\beta \tag{4.25}$$

$$U_S = \frac{T_{120}}{T} \cdot U_{120} + \frac{T_{60}}{T} \cdot U_{60} \tag{4.26}$$

Sector II is depicted in 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:

$$T = T_{120} + T_{60} + Tnull \tag{4.27}$$

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

$$U_s = \frac{T_{120}}{T} * U_{120} + \frac{T_{60}}{T} * U_{60} \tag{4.28}$$

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. A graphical representation of the auxiliary components is shown in 4-6.

Figure 4-6: Detail of the Voltage Vector Projection in Sector II

The equations describing those auxiliary time-duration components are:

$$\frac{\sin 30°}{\sin 120°} = \frac{A}{u_\beta} \tag{4.29}$$

$$\frac{\sin 60^\circ}{\sin 60^\circ} = \frac{B}{u_\alpha} \tag{4.30}$$

Eq. (4.28) and eq. (4.29) have been formed using the sine rule. These equations can be rearranged for the calculation of the auxiliary time-duration components A and B. This is done simply by substitution of the trigonometric terms $\sin(30^\circ), \sin(120^\circ)$ and $\sin(60^\circ)$ by their numerical representations $\frac{1}{2}$, $\frac{\sqrt{3}}{2}$ and $\frac{1}{\sqrt{3}}$, respectively.

$$A = \frac{1}{\sqrt{3}} \cdot u_\beta \tag{4.31}$$

$$B = u_\alpha \tag{4.32}$$

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 eq. (4.32) and eq. (4.33), as follows:

$$\frac{T_{120}}{T} \cdot |U_{120}| = A - B \tag{4.33}$$

$$\frac{T_{60}}{T} \cdot |U_{60}| = A + B \tag{4.34}$$

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:

$$\frac{T_{120}}{T} = \frac{1}{2}\left(u_\beta - \sqrt{3} \cdot u_\alpha\right) \tag{4.35}$$

$$\frac{T_{60}}{T} = \frac{1}{2}\left(u_\beta + \sqrt{3} \cdot u_\alpha\right) \tag{4.36}$$

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:

$$X = u_\beta \tag{4.37}$$

$$Y = \frac{1}{2} \cdot \left(u_\beta + \sqrt{3} \cdot u_\alpha\right) \tag{4.38}$$

$$Z = \frac{1}{2} \cdot \left(u_\beta - \sqrt{3} \cdot u_\alpha\right) \tag{4.39}$$

To simplify the calculation, two expressions, t_1 and t_2, are introduced. These expressions represent duty-cycle ratios of the basic space vectors in the respective sector; e.g., for the first sector, t_1 and t_2 represent duty-cycle ratios of the basic space vectors U_{60} and U_0; for the second sector, t_1 and t_2 represent duty-cycle ratios of the basic space vectors U_{120} and U_{60}, etc. For each sector, the relation between the t_1 and t_2 and variables X ((4.37)), Y ((4.38)) and Z ((4.39)), are listed in Table 4-16.

For the determination of auxiliary variables X eq. (4.37), Y eq. (4.38) and Z eq. (4.39), the sector number is required. This information can be obtained by several approaches. One

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

approach discussed here requires the use of a modified Inverse Clark Transformation to transform the direct (α) and quadrature (β) components into a balanced three-phase quantity u_{ref1}, u_{ref2} and u_{ref3}, used for a straightforward calculation of the sector number, to be shown later.

$$u_{ref1} = u_\beta \tag{4.40}$$

$$u_{ref2} = \frac{1}{2} \cdot \left(-u_\beta + \sqrt{3} \cdot u_\alpha\right) \tag{4.41}$$

$$u_{ref3} = \frac{1}{2} \cdot \left(-u_\beta - \sqrt{3} * u_\alpha\right) \tag{4.42}$$

The modified Inverse Clark Transformation projects the quadrature (u_β) component into u_{ref1}, as shown in 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 (u_a) and Quadrature (u_b) 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 4-9, can be a numerical solution of the approach shown in 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 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}; refer to 4-8. If these quantities are used as the inputs to the Sector Identification Tree, the product of those comparisons will be Sector I. Using the same approach identifies Sector II, if the stator reference voltage vector is located according

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

Figure 4-9: Identification of the Sector Number

to the one shown in 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:

$$t_1 = \frac{T - t_1 - t_2}{2} \tag{4.43}$$

$$t_2 = t_1 + t_1 \tag{4.44}$$

$$t_3 = t_2 + t_2 \tag{4.45}$$

where T is the switching period, t_1 and t_2 are the duty-cycle ratios (see table. 4-16) of the basic space vectors, given for the respective sector. Eq. (4.39), eq. (4.40) and eq. (4.41) 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 resides in applying the basic voltage vectors U_{XXX} and O_{XXX} for a certain time in such a way that the main vector, generated by the Pulse Width Modulation approach for the period T, is equal to the original stator reference voltage vector U_S. This provides a great variability of the arrangement of the basic vectors during the PWM period T. Those vectors might be arranged either to lower switching losses or to achieve diverse results, such as centre-aligned PWM, edge-aligned PWM or a minimal number of switching states. A brief discussion of the widely-used centre-aligned PWM follows. Generating the centre-aligned PWM pattern is accomplished practically by comparing the threshold levels, pwm_a, pwm_b and pwm_c with a free-running up-down counter. The timer counts to 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 4-10

4.8.6 Note:

As during the computation the irrational value of $\frac{\sqrt{3}}{4}$ is used for calculation, the correction constant is used to increase the calculation precision in boundary intervals. This correction constant is equal to the difference of $\frac{\sqrt{3}}{4}$ computed in double and in single precision.

4.8.7 Reentrancy:

The function is reentrant.

Figure 4-10: Standard Space Vector Modulation Technique - Centre-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       = (tFloat)(12.99/U_MAX);
    tr32InVoltage.s32Arg2       = (tFloat)(7.5/U_MAX);

    // output pwm dutycycles stored in structure referenced by tr32PwmABC
    // pwmA dutycycle    = (tFloat)(0.9999888)
    // pwmb dutycycle    = (tFloat)(0.5000111)
    // pwmc dutycycle    = (tFloat)(0.0000111)
    // svmSector         = 0x1 [sector]
    u32SvmSector = GMCLIB_SvmStd(&tr32PwmABC,&tr32InVoltage);
}
```

4.8.9 Performance:

Table 4-18: GMCLIB_SvmStd function performance

Code size [bytes] GHS/CW/DIAB	730/562/580
Data size [bytes] GHS/CW/DIAB	0/0/0
Execution clock cycles max [clk] GHS/CW/DIAB	97/110/111
Execution clock cycles avg [clk] GHS/CW/DIAB	80/96/88
Execution clock cycles min [clk] GHS/CW/DIAB	63/80/78

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
- tMclib GDFLIB_FilterIIR1 (tMclib s32In, GDFLIB_FILTER_IIR1_T * pParam)
  This function implements a Direct Form I of first order IIR filter
- void GDFLIB_FilterIIR2Init (GDFLIB_FILTER_IIR2_T * pParam)
  This function clears internal filter buffers used in function GDFLIB_FilterIIR2
- tMclib GDFLIB_FilterIIR2 (tMclib s32In, GDFLIB_FILTER_IIR2_T * pParam)
  This function implements a Direct Form I of second order IIR filter
- void GDFLIB_FilterFIRInit (const GDFLIB_FILTERFIR_PARAM_T *const pParam, GDFLIB_-FILTERFIR_STATE_T *const pState, tMclib * ps32InBuf)
  The function performs initialization for the GDFLIB_FilterFIR function
- tMclib GDFLIB_FilterFIR (tMclib s32In, const GDFLIB_FILTERFIR_PARAM_T *const pParam, GDFLIB_FILTERFIR_STATE_T *const pState)
  The function performs a single iteration of an FIR filter
- void GDFLIB_FilterMAInit (GDFLIB_FILTER_MA_T * pParam)
  This function clears the internal filter accumulator
- tMclib GDFLIB_FilterMA (tMclib 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 a filter structure with filter buffer and filter parameters. Arguments of the structure contain single precision floating point values.

5.2.4 Return:

Function returns no value.

5.2.5 Description:

This function clears the internal buffers of a first order IIR filter. The function 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"

GDFLIB_FILTER_IIR1_T trMyFilterIIR1 = GDFLIB_FILTER_IIR1_DEFAULT;

void main(void)
{
  // Filter coefficients
  trMyFilterIIR1.trFiltCoeff.s32B0 = 0.030468747091254;
  trMyFilterIIR1.trFiltCoeff.s32B1 = 0.030468747091254;
  trMyFilterIIR1.trFiltCoeff.s32A1 = -0.939062505817492;
  GDFLIB_FilterIIR1Init(&trMyFilterIIR1);
  for(;;); // loop forever
}
```

5.2.9 Performance:

Table 5-2: GDFLIB_FilterIIR1Init function performance

Code size [bytes] GHS/CW/DIAB	8/8/8
Data size [bytes] GHS/CW/DIAB	0/0/0
Execution clock cycles max [clk] GHS/CW/DIAB	6/6/6
Execution clock cycles avg [clk] GHS/CW/DIAB	6/6/6
Execution clock cycles min [clk] GHS/CW/DIAB	6/6/6

5.3 GDFLIB_FilterIIR1

5.3.1 Declaration:

```
tMclib GDFLIB_FilterIIR1ANSIC(tMclib 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 a filter structure with a filter buffer and filter parameters. Arguments of the structure contain single precision floating point values.
tMclib	s32In	in	Value of the input signal to be filtered in step (k). Input is a 32-bit number that contains a single precision floating point value.

5.3.4 Return:

The function returns a 32-bit value in single precision floating point format, representing the filtered value of the input signal in step (k).

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 eq. (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 eq. (5.2), must be transformed into a time difference equation as follows:

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

where: $x[k]$ is the input signal, $y[k]$ is the output signal, a_i and b_i are the filter coefficients. 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 lower order sections, i.e. first or second order. The main difference between DF-I and DF-II implementations of an IIR filter is in the number of delay buffers and in the number of guard bits required to handle the potential overflow. The DF-II implementation requires less delay buffers than DF-I, hence less data memory is utilized. On the other hand, since the poles come first in the DF-II realization, the signal entering the state delay-line typically requires a larger dynamic range than the output signal $y(k)$. Therefore, overflow can occur at the delay-line input of the DF-II implementation, unlike in the DF-I implementation.

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 Fig. 5-1 can be designed to meet the requirements for the first order Low Pass (LPF) or High Pass (HPF) filters. Filter coefficients can be calculated using various tools, for example, the Matlab *butter* function.

5.3.6 Warning:

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.

5.3.7 Note:

To enumerate the computation error in one calculation cycle, the internal accumulator is not used for testing purposes and is replaced by output from the previous calculation step of the Matlab precise reference model.

5.3.8 Reentrancy:

The function is reentrant.

5.3.9 Code Example:

```
#include "gdflib.h"

tFloat fInput;
tFloat fOutput;
GDFLIB_FILTER_IIR1_T trMyFilterIIR1 = GDFLIB_FILTER_IIR1_DEFAULT;

void main(void)
{
  // Filter coefficients
  trMyFilterIIR1.trFiltCoeff.s32B0 = 0.030468747091254;
  trMyFilterIIR1.trFiltCoeff.s32B1 = 0.030468747091254;
  trMyFilterIIR1.trFiltCoeff.s32A1 = -0.939062505817492;
  GDFLIB_FilterIIR1Init(&trMyFilterIIR1);
  fInput = 0.25; // input value = 0.25
  fOutput = GDFLIB_FilterIIR1(fInput, &trMyFilterIIR1); // output should be 0.0076171868
  fInput = 0.052; // input value = 0.052
  fOutput = GDFLIB_FilterIIR1(fInput, &trMyFilterIIR1); // output should be 0.016354576
  fInput = 0.351; // input value = 0.351
  fOutput = GDFLIB_FilterIIR1(fInput, &trMyFilterIIR1); // output should be 0.027636874
  fInput = 0.6523; // input value = 0.6523
  fOutput = GDFLIB_FilterIIR1(fInput, &trMyFilterIIR1); // output should be 0.056522049
  fInput = 0.7825; // input value = 0.7825
  fOutput = GDFLIB_FilterIIR1(fInput, &trMyFilterIIR1); // output should be 0.096794292
  for(;;); // loop forever
}
```

5.3.10 Performance:

Table 5-4: GDFLIB_FilterIIR1 function performance

Code size [bytes] GHS/CW/DIAB	46/46/40
Data size [bytes] GHS/CW/DIAB	0/0/0
Execution clock cycles max [clk] GHS/CW/DIAB	43/32/33
Execution clock cycles avg [clk] GHS/CW/DIAB	42/30/31
Execution clock cycles min [clk] GHS/CW/DIAB	34/30/31

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 5-5: Function parameters

Type	Name	Dir.	Description
GDFLIB_FILTER_IIR2_T *	pParam	in/out	Pointer to a filter structure with a filter buffer and filter parameters. Arguments of the structure contain single precision floating point values.

5.4.4 Return:

Function returns no value.

5.4.5 Description:

This function clears the internal buffers of a second order IIR filter. The function 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"

GDFLIB_FILTER_IIR2_T trMyFilterIIR2 = GDFLIB_FILTER_IIR2_DEFAULT;

void main(void)
{
  // Filter coefficients
  trMyFilterIIR2.trFiltCoeff.s32B0 = 0.066122101544579;
  trMyFilterIIR2.trFiltCoeff.s32B1 = 0.0;
  trMyFilterIIR2.trFiltCoeff.s32B2 = -0.066122101544579;
  trMyFilterIIR2.trFiltCoeff.s32A1 = -1.776189018043779;
  trMyFilterIIR2.trFiltCoeff.s32A2 = 0.867755796910841;
  GDFLIB_FilterIIR2Init(&trMyFilterIIR2);
  for(;;); // loop forever
}
```

5.4.9 Performance:

Table 5-6: GDFLIB_FilterIIR2Init function performance

Code size [bytes] GHS/CW/DIAB	12/12/12
Data size [bytes] GHS/CW/DIAB	0/0/0
Execution clock cycles max [clk] GHS/CW/DIAB	9/9/9
Execution clock cycles avg [clk] GHS/CW/DIAB	9/9/9
Execution clock cycles min [clk] GHS/CW/DIAB	9/9/9

5.5 GDFLIB_FilterIIR2

5.5.1 Declaration:

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

5.5.2 Alias:

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

5.5.3 Arguments:

Table 5-7: Function parameters

Type	Name	Dir.	Description
GDFLIB_FILTER_IIR2_T *	pParam	in/out	Pointer to a filter structure with a filter buffer and filter parameters. Arguments of the structure contain single precision floating point values.
tMclib	s32In	in	Value of the input signal to be filtered in step (k). Input is a 32-bit number that contains a single precision floating point value.

5.5.4 Return:

The function returns a 32-bit value in single precision floating point format, representing the filtered value of the input signal in step (k).

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 eq. (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 eq. (5.5), must be transformed into a time difference equation as follows:

$$y(k) = b_0 x(k) + b_1 x(k-1) + b_2 x(k-2) - a_1 y(k-1) - a_2 y(k-2) \tag{5.6}$$

where: $x[k]$ is the input signal, $y[k]$ is the output signal, a_i and b_i are the filter coefficients. 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 lower order sections, i.e. first or second order. The main difference between DF-I and DF-II implementations of an IIR filter is in the number of delay buffers and in the number of guard bits required to handle the potential overflow. The DF-II implementation requires less delay buffers than DF-I, hence less data memory is utilized. On the other hand, since the poles come first in the DF-II realization, the signal entering the state delay-line typically requires a larger dynamic range than the output signal $y(k)$. Therefore, overflow can occur at the delay-line input of the DF-II implementation, unlike in the DF-I implementation.

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

The coefficients of the filter depicted in Fig. 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.

5.5.6 Warning:

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.

5.5.7 Note:

To enumerate the computation error in one calculation cycle, the internal accumulator is not used for testing purposes and is replaced by output from the previous calculation step of the Matlab precise reference model.

5.5.8 Reentrancy:

The function is reentrant.

5.5.9 Code Example:

```
#include "gdflib.h"

tFloat fInput;
tFloat fOutput;
GDFLIB_FILTER_IIR2_T trMyFilterIIR2 = GDFLIB_FILTER_IIR2_DEFAULT;

void main(void)
{
  // Filter coefficients
  trMyFilterIIR2.trFiltCoeff.s32B0 = 0.066122101544579;
  trMyFilterIIR2.trFiltCoeff.s32B1 = 0.0;
  trMyFilterIIR2.trFiltCoeff.s32B2 = -0.066122101544579;
  trMyFilterIIR2.trFiltCoeff.s32A1 = -1.776189018043779;
  trMyFilterIIR2.trFiltCoeff.s32A2 = 0.867755796910841;
  GDFLIB_FilterIIR2Init(&trMyFilterIIR2);
  fInput = 0.25; // input value = 0.25
  fOutput = GDFLIB_FilterIIR2(fInput,&trMyFilterIIR2); // output should be 0.016530525
  fInput = 0.052; // input value = 0.052
  fOutput = GDFLIB_FilterIIR2(fInput,&trMyFilterIIR2); // output should be 0.032799684
  fInput = 0.351; // input value = 0.351
  fOutput = GDFLIB_FilterIIR2(fInput,&trMyFilterIIR2); // output should be 0.050592314
  fInput = 0.6523; // input value = 0.6523
  fOutput = GDFLIB_FilterIIR2(fInput,&trMyFilterIIR2); // output should be 0.10109249
  fInput = 0.7825; // input value = 0.7825
  fOutput = GDFLIB_FilterIIR2(fInput,&trMyFilterIIR2); // output should be 0.16418928
  for(;;); // loop forever
}
```

5.5.10 Performance:

Table 5-8: GDFLIB_FilterIIR2 function performance

Code size [bytes] GHS/CW/DIAB	82/74/64
Data size [bytes] GHS/CW/DIAB	0/0/0
Execution clock cycles max [clk] GHS/CW/DIAB	52/45/60
Execution clock cycles avg [clk] GHS/CW/DIAB	51/43/58
Execution clock cycles min [clk] GHS/CW/DIAB	49/43/58

5.6 GDFLIB_FilterFIRInit

5.6.1 Declaration:

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

5.6.2 Alias:

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

5.6.3 Arguments:

Table 5-9: Function parameters

Type	Name	Dir.	Description
const GDFLIB_FILTERFIR_PARAM_T *const	pParam	in	Pointer to a parameters structure.
GDFLIB_FILTERFIR_STATE_T *const	pState	out	Pointer to a state structure.
tMclib *	ps32InBuf	in/out	Pointer to a buffer for storing filter input signal values, must point to a R/W memory region and must be the filter order + 1 long.

5.6.4 Return:

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 Warning:

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 the number of the filter taps long. The number of taps in a filter is equal to the filter order + 1. There is no restriction as to the location of the parameters structure as long as it is readable.

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;

tFloat as32InBuf[FIR_NUMTAPS_MAX];
tFloat as32CoefBuf[FIR_NUMTAPS_MAX];

#define OUT_LEN 16

void main(void)
{
    int ii;
    tFloat 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++] = (tFloat)(-0.997590551617365);
    as32CoefBuf[ii++] = (tFloat)(-0.995837825348525);
    as32CoefBuf[ii++] = (tFloat)(0.0095364856578114);
```

```
    as32CoefBuf[ii++] = (tFloat)(0.0199709259997918);
    as32CoefBuf[ii++] = (tFloat)(-0.962045819946586);
    as32CoefBuf[ii++] = (tFloat)(-0.930427167532233);
    as32CoefBuf[ii++] = (tFloat)(0.137360839237161);
    as32CoefBuf[ii++] = (tFloat)(0.447230387221663);
    as32CoefBuf[ii++] = (tFloat)(0.447230387221663);
    as32CoefBuf[ii++] = (tFloat)(0.137360839237161);
    as32CoefBuf[ii++] = (tFloat)(-0.30427167532233);
    as32CoefBuf[ii++] = (tFloat)(-0.962045819946586);
    as32CoefBuf[ii++] = (tFloat)(0.199709259997918);
    as32CoefBuf[ii++] = (tFloat)(0.0095364856578114);
    as32CoefBuf[ii++] = (tFloat)(-0.995837825348525);
    as32CoefBuf[ii++] = (tFloat)(-0.997590551617365);

    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((tFloat)(1), &Param, &State);
    }

    // as32Out contains step response of the filter
}
```

5.6.10 Performance:

Table 5-10: GDFLIB_FilterFIRInit function performance

Code size [bytes] GHS/CW/DIAB	90/30/24
Data size [bytes] GHS/CW/DIAB	0/0/0
Execution clock cycles max [clk] GHS/CW/DIAB	34/34/34
Execution clock cycles avg [clk] GHS/CW/DIAB	34/34/34
Execution clock cycles min [clk] GHS/CW/DIAB	34/34/34

5.7 GDFLIB_FilterFIR

5.7.1 Declaration:

```
tMclib GDFLIB_FilterFIRANSIC(tMclib 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 5-11: Function parameters

Type	Name	Dir.	Description
tMclib	s32In	in	Input value.
const GDFLIB_-FILTERFIR_PARAM_T *const	pParam	in	Pointer to a parameter structure.
GDFLIB_FILTERFIR_-STATE_T *const	pState	in/out	Pointer to a 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:

$$y[n] = h_0 x[n] + h_1 x[n-1] + \cdots + h_N x[n-N] \tag{5.7}$$

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. It should be noted, that the number of taps of the filter is $N+1$ in this case. 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 Warning:

The count of maximal clock cycles depends on the selected order of the filter. Minimal clock cycles, noted in table 5-12, are valid for the first order FIR 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;

tFloat as32InBuf[FIR_NUMTAPS_MAX];
tFloat as32CoefBuf[FIR_NUMTAPS_MAX];

#define OUT_LEN 16

void main(void)
{
    int ii;
    tFloat 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++] = (tFloat)(-0.997590551617365);
    as32CoefBuf[ii++] = (tFloat)(-0.995837825348525);
    as32CoefBuf[ii++] = (tFloat)(0.0095364856578114);
    as32CoefBuf[ii++] = (tFloat)(0.0199709259997918);
    as32CoefBuf[ii++] = (tFloat)(-0.962045819946586);
    as32CoefBuf[ii++] = (tFloat)(-0.930427167532233);
    as32CoefBuf[ii++] = (tFloat)(0.137360839237161);
    as32CoefBuf[ii++] = (tFloat)(0.447230387221663);
    as32CoefBuf[ii++] = (tFloat)(0.447230387221663);
    as32CoefBuf[ii++] = (tFloat)(0.137360839237161);
    as32CoefBuf[ii++] = (tFloat)(-0.30427167532233);
    as32CoefBuf[ii++] = (tFloat)(-0.962045819946586);
    as32CoefBuf[ii++] = (tFloat)(0.199709259997918);
    as32CoefBuf[ii++] = (tFloat)(0.0095364856578114);
    as32CoefBuf[ii++] = (tFloat)(-0.995837825348525);
    as32CoefBuf[ii++] = (tFloat)(-0.997590551617365);

    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((tFloat)(1.0), &Param, &State);
    }

    // as32Out contains step response of the filter
}
```

5.7.10 Performance:

Table 5-12: GDFLIB_FilterFIR function performance

Code size [bytes] GHS/CW/DIAB	318/430/126
Data size [bytes] GHS/CW/DIAB	0/0/0
Execution clock cycles max [clk] GHS/CW/DIAB	93/93/93
Execution clock cycles avg [clk] GHS/CW/DIAB	82/82/82
Execution clock cycles min [clk] GHS/CW/DIAB	81/81/81

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 5-13: Function parameters

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

5.8.4 Return:

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 5-14: GDFLIB_FilterMAInit function performance

Code size [bytes] GHS/CW/DIAB	6/6/6
Data size [bytes] GHS/CW/DIAB	0/0/0
Execution clock cycles max [clk] GHS/CW/DIAB	6/6/6
Execution clock cycles avg [clk] GHS/CW/DIAB	6/6/6
Execution clock cycles min [clk] GHS/CW/DIAB	6/6/6

5.9 GDFLIB_FilterMA

5.9.1 Declaration:

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

5.9.2 Alias:

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

5.9.3 Arguments:

Table 5-15: Function parameters

Type	Name	Dir.	Description
tMclib	s32In	in	Value of the input signal to be filtered in step (k). The value is a single precision floating point data type.
GDFLIB_FILTER_MA_T *	pParam	in/out	Pointer to the filter structure with a filter accumulator and filter parameters

5.9.4 Return:

The function returns a single precision floating point value, 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"

tFloat s32Input;
tFloat s32Output;

GDFLIB_FILTER_MA_T trMyMA = GDFLIB_FILTER_MA_DEFAULT;

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

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

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

5.9.9 Performance:

Table 5-16: GDFLIB_FilterMA function performance

Code size [bytes] GHS/CW/DIAB	36/44/28
Data size [bytes] GHS/CW/DIAB	0/0/0
Execution clock cycles max [clk] GHS/CW/DIAB	48/48/36
Execution clock cycles avg [clk] GHS/CW/DIAB	43/43/34
Execution clock cycles min [clk] GHS/CW/DIAB	41/41/34

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	signed 16-bit fractional Q1.15 type
typedef tS32	tFrac32	signed 32-bit fractional Q1.31 type
typedef float	tFloat	single precision float type
typedef double	tDouble	double precision float type
typedef float	tMclib	32-bit float type

CHAPTER 7: COMPOUND DATA TYPES

7.0.1 Compound Data Types Overview

- FILTER_IIR1_COEFF_T
  Sub-structure containing filter coefficients.
- FILTER_IIR2_COEFF_T
  Sub-structure 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_T
  Structure containing approximation coefficients for calculation of the arccos function.
- GFLIB_ASIN_TAYLOR_T
  Structure containing approximation coefficients for calculation of the arcsin function.
- GFLIB_ATAN_TAYLOR_T
  Structure containing the approximation coefficients.
- GFLIB_ATANYXSHIFTED_T
  Structure containing the parameter for the 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 parallel form PI controller with anti-windup.

- GFLIB_CONTROLLER_PIAW_R_T

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

- GFLIB_COSTLR_T

  Structure containing one array of three single precision floating point coefficients for floating point optimized minimax approximation.

- 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_LUT2D_T

  Structure containing 2D look-up table parameters.

- GFLIB_RAMP_T

  Structure containing increment/decrement coefficients and state value for the ramp function implemented in GFLIB_Ramp.

- GFLIB_SINTLR_T

  Structure containing one array of three single precision floating point coefficients for floating point optimized minimax approximation.

- GFLIB_TANTLR_T

  Structure containing four rational polynomial coefficients.

- 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:

Sub-structure containing filter coefficients.

7.1.2 Compound Type Members:

Table 7-1: Members description

Type	Name	Description
tMclib	s32B0	B0 coefficient of an IIR1 filter. The parameter is in full range single precision floating point format.
tMclib	s32B1	B1 coefficient of an IIR1 filter. The parameter is in full range single precision floating point format.
tMclib	s32A1	A1 coefficient of an IIR1 filter. The parameter is in full range single precision floating point format.

7.2 FILTER_IIR2_COEFF_T

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

7.2.1 Description:

Sub-structure containing filter coefficients.

7.2.2 Compound Type Members:

Table 7-2: Members description

Type	Name	Description
tMclib	s32B0	B0 coefficient of an IIR2 filter. The parameter is in full range single precision floating point format.
tMclib	s32B1	B1 coefficient of an IIR2 filter. The parameter is in full range single precision floating point format.
tMclib	s32B2	B2 coefficient of an IIR2 filter. The parameter is in full range single precision floating point format.
tMclib	s32A1	A1 coefficient of an IIR2 filter. The parameter is in full range single precision floating point format.
tMclib	s32A2	A2 coefficient of an IIR2 filter. The parameter is in full range single precision floating point format.

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.
tMclib	s32FiltBufferX[2]	Input buffer of an IIR1 filter. The input values are in full range single precision floating point format.
tMclib	s32FiltBufferY[2]	Internal accumulator buffer. The values are in full range single precision floating point format.

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.
tMclib	s32FiltBufferX[3]	Input buffer of an IIR2 filter. The input values are in full range single precision floating point format.
tMclib	s32FiltBufferY[3]	Internal accumulator buffer. The values are in full range single precision floating point format.

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
tMclib	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 tMclib *	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.
tMclib *	ps32InBuf	Pointer to the input buffer.

7.8 GFLIB_ACOS_TAYLOR_T

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

7.8.1 Description:

Structure containing approximation coefficients for calculation of the arccos function. The polynomial approximation is done using the square root function of the input parameter. The essential assumption of this approach is a fast H/W based calculation of the square root operation. The approximation polynomial coefficients assume an input argument in the $[-\pi, \pi]$ interval.

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, i.e. GFLIB_AcosANSIC

7.8.2 Compound Type Members:

Table 7-8: Members description

Type	Name	Description
tFloat	s32a[5]	Array of appoximation coefficients

7.9 GFLIB_ASIN_TAYLOR_T

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

7.9.1 Description:

Structure containing approximation coefficients for calculation of the arcsin function. The polynomial approximation is done using the square root function of the input parameter. The essential assumption of this approach is a fast H/W based calculation of the square root operation. The approximation polynomial coefficients assume an input argument in the $[-pi, pi]$ interval.

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, i.e. GFLIB_AsinANSIC

7.9.2 Compound Type Members:

Table 7-9: Members description

Type	Name	Description
tFloat	s32a[8]	Array of approximation coefficients

7.10 GFLIB_ATAN_TAYLOR_T

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

7.10.1 Description:

Structure containing the approximation coefficients. The approximation uses the rational polynomial approximation which is based on the division of two rational polynomials. The essential assumption of this approach is the fast calculation of a two floating point values division. The approximation polynomial coefficients assume the input argument in the interval $\left(-2^{128}, 2^{128}\right)$.

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

7.10.2 Compound Type Members:

Table 7-10: Members description

Type	Name	Description
const tFloat	s32a[6]	Array of coeficients for polynomial approximation

7.11 GFLIB_ATANYXSHIFTED_T

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

7.11.1 Description:

Structure containing the parameter for the AtanYXShifted function.

7.11.2 Compound Type Members:

Table 7-11: Members description

Type	Name	Description
tFloat	s32Ky	Multiplication coefficient for the y-signal.
tFloat	s32Kx	Multiplication coefficient for the x-signal.
tFloat	s32ThetaAdj	Adjusting angle.

7.12 GFLIB_CONTROLLER_PI_P_T

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

7.12.1 Description:

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

7.12.2 Compound Type Members:

Table 7-12: Members description

Type	Name	Description
tFloat	s32PropGain	Proportional Gain, single precision floating point format
tFloat	s32IntegGain	Integral Gain, single precision floating point format
tFloat	s32IntegPartK_1	State variable integral part at step k-1, single precision floating point format
tFloat	s32InK_1	State variable input error at step k-1, single precision floating point format

7.13 GFLIB_CONTROLLER_PI_R_T

```
#include <GFLIB_ControllerPIr.h>
```

7.13.1 Description:

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

7.13.2 Compound Type Members:

Table 7-13: Members description

Type	Name	Description
tFloat	s32CC1sc	CC1 coefficient, single precision floating point format
tFloat	s32CC2sc	CC2 coefficient, single precision floating point format
tFloat	s32Acc	Internal controller accumulator, single precision floating point format
tFloat	s32InErrK1	Controller input from the previous calculation step, single precision floating point format

7.14 GFLIB_CONTROLLER_PIAW_P_T

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

7.14.1 Description:

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

7.14.2 Compound Type Members:

Table 7-14: Members description

Type	Name	Description
tFloat	s32PropGain	Proportional Gain, single precision floating point format
tFloat	s32IntegGain	Integral Gain, single precision floating point format
tFloat	s32LowerLimit	Lower Limit of the controller, single precision floating point format
tFloat	s32UpperLimit	Upper Limit of the controller, single precision floating point format
tFloat	s32IntegPartK_1	State variable integral part at step k-1, single precision floating point format
tFloat	s32InK_1	State variable input error at step k-1, single precision floating point format
tU16	u16LimitFlag	Limitation flag, if set to 1, the controller output has reached either the UpperLimit or LowerLimit

7.15 GFLIB_CONTROLLER_PIAW_R_T

```
#include <GFLIB_ControllerPIrAW.h>
```

7.15.1 Description:

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

7.15.2 Compound Type Members:

Table 7-15: Members description

Type	Name	Description
tFloat	s32CC1sc	CC1 coefficient, single precision floating point format
tFloat	s32CC2sc	CC2 coefficient, single precision floating point format
tFloat	s32Acc	Internal controller accumulator, single precision floating point format
tFloat	s32InErrK1	Controller input from the previous calculation step, single precision floating point format
tFloat	s32UpperLimit	Upper Limit of the controller, single precision floating point format
tFloat	s32LowerLimit	Lower Limit of the controller, single precision floating point format

7.16 GFLIB_COSTLR_T

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

7.16.1 Description:

Structure containing one array of three single precision floating point coefficients for floating point optimized minimax approximation. 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, i.e. GFLIB_CosANSIC.

7.16.2 Compound Type Members:

Table 7-16: Members description

Type	Name	Description
tFloat	s32A[3]	Array of three single precision floating point elements for storing coefficients of the floating point optimized minimax approximation polynomial.

7.17 GFLIB_HYST_T

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

7.17.1 Description:

Structure containing parameters and states for the hysteresis function implemented in GFLIB_-Hyst.

7.17.2 Compound Type Members:

Table 7-17: Members description

Type	Name	Description
tMclib	s32HystOn	Value determining the upper threshold. The parameter is in full range single precision floating point format.
tMclib	s32HystOff	Value determining the lower threshold. The parameter is in full range single precision floating point format.
tMclib	s32OutValOn	Value of the output when input is higher than the upper threshold. The parameter is in full range single precision floating point format.
tMclib	s32OutValOff	Value of the output when input is the lower than the lower threshold. The parameter is in full range single precision floating point format.
tMclib	s32OutState	Actual state of the output. The parameter is in full range single precision floating point format.

7.18 GFLIB_INTEGRATOR_TR_T

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

7.18.1 Description:

Structure containing integrator parameters and coefficients.

7.18.2 Compound Type Members:

Table 7-18: Members description

Type	Name	Description
tFloat	s32State	integrator state value, single precision floating point format
tFloat	s32InK1	input value in step k-1, single precision floating point format
tFloat	s32C1	integrator coefficient, single precision floating point format

7.19 GFLIB_LIMIT_T

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

7.19.1 Description:

Structure containing the limits.

7.19.2 Compound Type Members:

Table 7-19: Members description

Type	Name	Description
tMclib	s32LowerLimit	Value determining the lower limit threshold. The parameter is in full range single precision floating point format.
tMclib	s32UpperLimit	Value determining the upper limit threshold. The parameter is in full range single precision floating point format.

7.20 GFLIB_LOWERLIMIT_T

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

7.20.1 Description:

Structure containing the lower limit.

7.20.2 Compound Type Members:

Table 7-20: Members description

Type	Name	Description
tMclib	s32LowerLimit	Value determining the lower limit threshold. The parameter is in full range single precision floating point format.

7.21 GFLIB_LUT1D_T

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

7.21.1 Description:

Structure containing look-up table parameters.

7.21.2 Compound Type Members:

Table 7-21: Members description

Type	Name	Description
tS32	s32ShamOffset	Shift amount for extracting the fractional offset within an interpolated interval. The value is a 32-bit signed integer number.
const tFloat *	ps32Table	Table holding ordinate values of interpolating intervals. The ordinate values are in full range single precision floating point format.

7.22 GFLIB_LUT2D_T

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

7.22.1 Description:

Structure containing 2D look-up table parameters.

7.22.2 Compound Type Members:

Table 7-22: Members description

Type	Name	Description
tS32	s32ShamOffset1	X-axis shift amount for extracting fractional offset within an interpolated interval. The value is a 32-bit signed integer number.
tS32	s32ShamOffset2	Y-axis shift amount for extracting fractional offset within an interpolated interval. The value is a 32-bit signed integer number.
const tFloat *	ps32Table	Table holding 2D ordinate values of interpolating intervals. The ordinate values are in full range single precision floating point format.

7.23 GFLIB_RAMP_T

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

7.23.1 Description:

Structure containing increment/decrement coefficients and state value for the ramp function implemented in GFLIB_Ramp.

7.23.2 Compound Type Members:

Table 7-23: Members description

Type	Name	Description
tMclib	s32State	Ramp state value. The parameter is in full range single precision floating point format.
tMclib	s32RampUp	Ramp up increment coefficient. The parameter is in full range single precision floating point format.
tMclib	s32RampDown	Ramp down increment (decrement) coefficient. The parameter is in full range single precision floating point format.

7.24 GFLIB_SINTLR_T

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

7.24.1 Description:

Structure containing one array of three single precision floating point coefficients for floating point optimized minimax approximation. 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, i.e. GFLIB_SinANSIC.

7.24.2 Compound Type Members:

Table 7-24: Members description

Type	Name	Description
tFloat	s32A[3]	Array of three single precision floating point elements for storing coefficients of the approximation polynomial.

7.25 GFLIB_TANTLR_T

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

7.25.1 Description:

Structure containing four rational polynomial coefficients. By calling the function alias GFLIB_-Tan, default values of the coefficients are used. Approximation coefficients can be modified by the user and in such a case the full function call shall be used, i.e. GFLIB_TanANSIC.

7.25.2 Compound Type Members:

Table 7-25: Members description

Type	Name	Description
tFloat	s32A[4]	Polynomial coefficient for fractional approximation in single precision floating point format

7.26 GFLIB_UPPERLIMIT_T

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

7.26.1 Description:

Structure containing the upper limit.

7.26.2 Compound Type Members:

Table 7-26: Members description

Type	Name	Description
tMclib	s32UpperLimit	Value determining the upper limit threshold. The parameter is in full range single precision floating point format.

7.27 GFLIB_VECTORLIMIT_T

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

7.27.1 Description:

Structure containing the limits.

7.27.2 Compound Type Members:

Table 7-27: Members description

Type	Name	Description
tMclib	s32Lim	The maximum magnitude of the input vector. The parameter is in full range single precision floating point format. The defined magnitude must be positive and equal to or greater than 0.

7.28 GMCLIB_DECOUPLINGPMSM_T

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

7.28.1 Description:

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

7.28.2 Compound Type Members:

Table 7-28: Members description

Type	Name	Description
tFloat	s32LD	L_D inductance [H]. The parameter is in full range single precision floating point format.
tFloat	s32LQ	L_Q inductance [H]. The parameter is in full range single precision floating point format.

7.29 GMCLIB_ELIMDCBUSRIP_T

```
#include <GMCLIB_ElimDcBusRip.h>
```

7.29.1 Description:

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

7.29.2 Compound Type Members:

Table 7-29: Members description

Type	Name	Description
tMclib	s32ModIndex	Inverse Modulation Index. The parameter is in full range single precision floating point format. The modulation index must be equal to or greater than 0.
tMclib	s32ArgDcBusMsr	Measured DC bus voltage. The parameter is in full range single precision floating point format. The measured voltage must be positive and equal to or greater than 0.

7.30 SWLIBS_2Syst

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

7.30.1 Description:

Structure data type for two axis input/output variables.

7.30.2 Compound Type Members:

Table 7-30: Members description

Type	Name	Description
tMclib	s32Arg1	First argument
tMclib	s32Arg2	Second argument

7.31 SWLIBS_3Syst

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

7.31.1 Description:

Structure data type for three axis input/output variables.

7.31.2 Compound Type Members:

Table 7-31: Members description

Type	Name	Description
tMclib	s32Arg1	First argument
tMclib	s32Arg2	Second argument
tMclib	s32Arg3	Third argument

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_AtanXY(x, y)
- 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_ControllerPIrAW(s32InErr, pParam)
- GFLIB_CONTROLLER_PIAW_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_Lut2D(x, y, pParam_xy)
- GFLIB_Ramp(in, pParam)
- GFLIB_RAMP_DEFAULT
- GFLIB_Sign(x)
- GFLIB_Sin(s32In)
- GFLIB_Sqrt(s32In)
- GFLIB_TAN_LIMIT
- GFLIB_Tan(s32In)
- GFLIB_UpperLimit(s32In, pParam)
- 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)

- MATH_FIX32
- MATH_FIX16
- MATH_FLOAT
- MATH_DOUBLE
- USE_ARITHMETIC
- SFRACT_MIN
- SFRACT_MAX
- FRACT_MIN
- FRACT_MAX
- INT16_MAX
- INT16_MIN
- INT32_MAX
- INT32_MIN
- 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
- FRAC16(x)
- FRAC32(x)
- FLOAT_MIN
- FLOAT_MAX
- FLOAT_DIVBY_SQRT3
- FLOAT_SQRT3_DIVBY_2
- FLOAT_SQRT3_DIVBY_4
- FLOAT_SQRT3_DIVBY_4_CORRECTION
- FLOAT_2_PI
- FLOAT_PI
- FLOAT_PI_DIVBY_2
- FLOAT_PI_SINGLE_CORRECTION
- FLOAT_PI_CORRECTION
- FLOAT_PI_DIVBY_4
- FLOAT_4_DIVBY_PI
- FLOAT_0_5
- FLOAT_PLUS_1
- FLOAT_MINUS_1
- inline
- _ARG2(_0, _1, _2, _3, ...)
- VA_NUM_ARGS(...)
- macro_dispatcher(func, ...)
- macro_dispatcher_(func, nargs)
- macro_dispatcher__(func, nargs)
- 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.

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.

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.

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_AtanXY

```
#include <GFLIB_AtanYX.h>
```

8.17.1 Macro Definition:

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

8.17.2 Description:

Function alias for the GFLIB_AtanYXANSIC function.

8.18 GFLIB_AtanYXShifted

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

8.18.1 Macro Definition:

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

8.18.2 Description:

Function alias for the GFLIB_AtanYXShiftedANSIC function.

8.19 GFLIB_ControllerPIp

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

8.19.1 Macro Definition:

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

8.19.2 Description:

Function alias for the GFLIB_ControllerPIpANSIC function.

8.20 GFLIB_CONTROLLER_PI_P_DEFAULT

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

8.20.1 Macro Definition:

```
#define GFLIB_CONTROLLER_PI_P_DEFAULT        0,0,0,0
```

8.20.2 Description:

Macro containing default values of the parallel PI controller structure.

8.21 GFLIB_ControllerPIpAW

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

8.21.1 Macro Definition:

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

8.21.2 Description:

Function alias for the GFLIB_ControllerPIpAWANSIC function.

8.22 GFLIB_CONTROLLER_PIAW_P_DEFAULT

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

8.22.1 Macro Definition:

```
#define GFLIB_CONTROLLER_PIAW_P_DEFAULT        0,0,FLOAT_MIN,FLOAT_MAX,0,0,0
```

8.22.2 Description:

Macro containing default values for the structure of parameters of the parallel PI controller with anti-windup.

8.23 GFLIB_ControllerPIr

```
#include <GFLIB_ControllerPIr.h>
```

8.23.1 Macro Definition:

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

8.23.2 Description:

Function alias for the GFLIB_ControllerPIrANSIC function.

8.24 GFLIB_CONTROLLER_PI_R_DEFAULT

```
#include <GFLIB_ControllerPIr.h>
```

8.24.1 Macro Definition:

```
#define GFLIB_CONTROLLER_PI_R_DEFAULT        0,0,0,0
```

8.24.2 Description:

Macro containing default values of the recurrent form PI controller structure.

8.25 GFLIB_ControllerPIrAW

```
#include <GFLIB_ControllerPIrAW.h>
```

8.25.1 Macro Definition:

```
#define GFLIB_ControllerPIrAW(s32InErr, pParam) \
        GFLIB_ControllerPIrAWANSIC(s32InErr, pParam)
```

8.25.2 Description:

Function alias for the GFLIB_ControllerPIrAWANSIC function.

8.26 GFLIB_CONTROLLER_PIAW_R_DEFAULT

```
#include <GFLIB_ControllerPIrAW.h>
```

8.26.1 Macro Definition:

```
#define GFLIB_CONTROLLER_PIAW_R_DEFAULT        0,0,0,0,FLOAT_MIN,FLOAT_MAX
```

8.26.2 Description:

Macro containing default values of the recurrent form PI controller with anti-windup.

8.27 GFLIB_Cos

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

8.27.1 Macro Definition:

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

8.27.2 Description:

Function alias for the GFLIB_CosANSIC function.

8.28 GFLIB_Hyst

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

8.28.1 Macro Definition:

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

8.28.2 Description:

Function alias for the GFLIB_HystANSIC function.

8.29 GFLIB_HYST_DEFAULT

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

8.29.1 Macro Definition:

```
#define GFLIB_HYST_DEFAULT        0,0,0,0,0
```

8.29.2 Description:

Macro containing default values of the hysteresis function structure.

8.30 GFLIB_IntegratorTR

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

8.30.1 Macro Definition:

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

8.30.2 Description:

Function alias for the GFLIB_IntegratorTRANSIC function.

8.31 GFLIB_INTEGRATOR_TR_DEFAULT

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

8.31.1 Macro Definition:

```
#define GFLIB_INTEGRATOR_TR_DEFAULT        0,0,0
```

8.31.2 Description:

Macro containing default values of the integrator structure.

8.32 GFLIB_Limit

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

8.32.1 Macro Definition:

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

8.32.2 Description:

Function alias for the GFLIB_LimitANSIC function.

8.33 GFLIB_LowerLimit

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

8.33.1 Macro Definition:

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

8.33.2 Description:

Function alias for the GFLIB_LowerLimitANSIC function.

8.34 GFLIB_Lut1D

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

8.34.1 Macro Definition:

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

8.34.2 Description:

Function alias for the GFLIB_Lut1DANSIC function.

8.35 GFLIB_Lut2D

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

8.35.1 Macro Definition:

```
#define GFLIB_Lut2D(x, y, pParam_xy) \
        GFLIB_Lut2DANSIC((x), (y), (pParam_xy))
```

8.35.2 Description:

Function alias for GFLIB_Lut2DANSIC function.

8.36 GFLIB_Ramp

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

8.36.1 Macro Definition:

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

8.36.2 Description:

Function alias for the GFLIB_RampANSIC function.

8.37 GFLIB_RAMP_DEFAULT

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

8.37.1 Macro Definition:

```
#define GFLIB_RAMP_DEFAULT        0,0,0
```

8.37.2 Description:

Macro containing default values of the Ramp structure.

8.38 GFLIB_Sign

```
#include <GFLIB_Sign.h>
```

8.38.1 Macro Definition:

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

8.38.2 Description:

Function alias for the GFLIB_SignANSIC function.

8.39 GFLIB_Sin

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

8.39.1 Macro Definition:

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

8.39.2 Description:

Function alias for the GFLIB_SinANSIC function.

8.40 GFLIB_Sqrt

```
#include <GFLIB_Sqrt.h>
```

8.40.1 Macro Definition:

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

8.40.2 Description:

Function alias for the GFLIB_SqrtANSIC function.

8.41 GFLIB_TAN_LIMIT

```
#include <GFLIB_Tan.c>
```

8.41.1 Macro Definition:

```
#define GFLIB_TAN_LIMIT        1000
```

8.41.2 Description:

Max output value for the tangent function.

8.42 GFLIB_Tan

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

8.42.1 Macro Definition:

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

8.42.2 Description:

Function alias for the GFLIB_TanANSIC function.

8.43 GFLIB_UpperLimit

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

8.43.1 Macro Definition:

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

8.43.2 Description:

Function alias for the GFLIB_UpperLimitANSIC function.

8.44 GFLIB_VectorLimit

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

8.44.1 Macro Definition:

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

8.44.2 Description:

Function alias for the GFLIB_VectorLimitANSIC function.

8.45 GMCLIB_Clark

```
#include <GMCLIB_Clark.h>
```

8.45.1 Macro Definition:

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

8.45.2 Description:

Function alias for the GMCLIB_ClarkANSIC function.

8.46 GMCLIB_ClarkInv

```
#include <GMCLIB_ClarkInv.h>
```

8.46.1 Macro Definition:

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

8.46.2 Description:

Function alias for the GMCLIB_ClarkInvANSIC function.

8.47 GMCLIB_DecouplingPMSM

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

8.47.1 Macro Definition:

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

8.47.2 Description:

Function alias for the GMCLIB_DecouplingPMSMANSIC function.

8.48 GMCLIB_DECOUPLINGPMSM_DEFAULT

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

8.48.1 Macro Definition:

```
#define GMCLIB_DECOUPLINGPMSM_DEFAULT        0,0,0,0
```

8.48.2 Description:

Macro containing reset values of the parameters for the decoupling algorithm implemented in the GMCLIB_DecouplingPMSM function.

8.49 GMCLIB_ElimDcBusRip

```
#include <GMCLIB_ElimDcBusRip.h>
```

8.49.1 Macro Definition:

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

8.49.2 Description:

Function alias for the GMCLIB_ElimDcBusRipANSIC function.

8.50 GMCLIB_ELIMDCBUSRIP_DEFAULT

```
#include <GMCLIB_ElimDcBusRip.h>
```

8.50.1 Macro Definition:

```
#define GMCLIB_ELIMDCBUSRIP_DEFAULT        0,0
```

8.50.2 Description:

Macro containing default values for the parameter structure of the GMCLIB_ElimDcBusRip function.

8.51 GMCLIB_Park

```
#include <GMCLIB_Park.h>
```

8.51.1 Macro Definition:

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

8.51.2 Description:

Function alias for the GMCLIB_ParkANSIC function.

8.52 GMCLIB_ParkInv

```
#include <GMCLIB_ParkInv.h>
```

8.52.1 Macro Definition:

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

8.52.2 Description:

Function alias for the GMCLIB_ParkInvANSIC function.

8.53 GMCLIB_SvmStd

```
#include <GMCLIB_SvmStd.h>
```

8.53.1 Macro Definition:

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

8.53.2 Description:

Function alias for the GMCLIB_SvmStdANSIC function.

8.54 MATH_FIX32

```
#include <SWLIBS_Config.h>
```

8.54.1 Macro Definition:

```
#define MATH_FIX32        1
```

8.54.2 Description:

32-bit fix point arithmetic selection switch.

8.55 MATH_FIX16

```
#include <SWLIBS_Config.h>
```

8.55.1 Macro Definition:

```
#define MATH_FIX16        2
```

8.55.2 Description:

16-bit fix point arithmetic selection switch.

8.56 MATH_FLOAT

```
#include <SWLIBS_Config.h>
```

8.56.1 Macro Definition:

```
#define MATH_FLOAT        3
```

8.56.2 Description:

Single precision floating point arithmetic selection switch.

8.57 MATH_DOUBLE

```
#include <SWLIBS_Config.h>
```

8.57.1 Macro Definition:

```
#define MATH_DOUBLE        4
```

8.57.2 Description:

Double precision floating point arithmetic selection switch.

8.58 USE_ARITHMETIC

```
#include <SWLIBS_Config.h>
```

8.58.1 Macro Definition:

```
#define USE_ARITHMETIC        MATH_FLOAT
```

8.58.2 Description:

Arithmetics selector

8.59 SFRACT_MIN

```
#include <SWLIBS_Defines.h>
```

8.59.1 Macro Definition:

```
#define SFRACT_MIN        (-1.0)
```

8.59.2 Description:

Constant representing the maximal negative value of a signed 16-bit fixed point fractional number eq -1.0

8.60 SFRACT_MAX

```
#include <SWLIBS_Defines.h>
```

8.60.1 Macro Definition:

```
#define SFRACT_MAX        (0.999969482421875)
```

8.60.2 Description:

Constant representing the maximal positive value of a signed 16-bit fixed point fractional number eq 0.999969482421875

8.61 FRACT_MIN

```
#include <SWLIBS_Defines.h>
```

8.61.1 Macro Definition:

```
#define FRACT_MIN        (-1.0)
```

8.61.2 Description:

Constant representing the maximal negative value of signed 32-bit fixed point fractional number eq -1.0

8.62 FRACT_MAX

```
#include <SWLIBS_Defines.h>
```

8.62.1 Macro Definition:

```
#define FRACT_MAX        (0.9999999995343387126922607421875)
```

8.62.2 Description:

Constant representing the maximal positive value of a signed 32-bit fixed point fractional number eq 0.9999999995343387126922607421875

8.63 INT16_MAX

```
#include <SWLIBS_Defines.h>
```

8.63.1 Macro Definition:

```
#define INT16_MAX        ((tS16) 0x7fff)
```

8.63.2 Description:

Constant representing the maximal positive value of a signed 16-bit fixed point integer number eq $2^{16-1} - 1 =$ 0x7fff.

8.64 INT16_MIN

```
#include <SWLIBS_Defines.h>
```

8.64.1 Macro Definition:

```
#define INT16_MIN        ((tS16) 0x8000)
```

8.64.2 Description:

Constant representing the maximal negative value of a signed 16-bit fixed point integer number eq $-2^{16-1} =$ 0x8000.

8.65 INT32_MAX

```
#include <SWLIBS_Defines.h>
```

8.65.1 Macro Definition:

```
#define INT32_MAX        ((tS32) 0x7fffffff)
```

8.65.2 Description:

Constant representing the maximal positive value of a signed 32-bit fixed point integer number eq $2^{32-1} - 1 =$ 0x7fff ffff.

8.66 INT32_MIN

```
#include <SWLIBS_Defines.h>
```

8.66.1 Macro Definition:

```
#define INT32_MIN        ((tS32) 0x80000000)
```

8.66.2 Description:

Constant representing the maximal negative value of a signed 32-bit fixed point integer number eq $-2^{32-1} =$ 0x8000 0000.

8.67 F16TOINT16

```
#include <SWLIBS_Defines.h>
```

8.67.1 Macro Definition:

```
#define F16TOINT16(x) \
        ((tS16) (x))
```

8.67.2 Description:

Type casting - signed 16-bit fractional value cast to a signed integer 16-bit integer.

8.68 F32TOINT16

```
#include <SWLIBS_Defines.h>
```

8.68.1 Macro Definition:

```
#define F32TOINT16(x) \
        ((tS16) (x))
```

8.68.2 Description:

Type casting - lower 16 bits of a signed 32-bit fractional value cast to a signed 16-bit integer.

8.69 F64TOINT16

```
#include <SWLIBS_Defines.h>
```

8.69.1 Macro Definition:

```
#define F64TOINT16(x) \
        ((tS16) (x))
```

8.69.2 Description:

Type casting - lower 16 bits of a signed 64-bit fractional value cast to a signed 16-bit integer.

8.70 F16TOINT32

```
#include <SWLIBS_Defines.h>
```

8.70.1 Macro Definition:

```
#define F16TOINT32(x) \
        ((tS32) (x))
```

8.70.2 Description:

Type casting - a signed 16-bit fractional value cast to a signed 32-bit integer, the value placed at the lower 16 bits of the 32-bit result.

8.71 F32TOINT32

```
#include <SWLIBS_Defines.h>
```

8.71.1 Macro Definition:

```
#define F32TOINT32(x) \
        ((tS32) (x))
```

8.71.2 Description:

Type casting - signed 32-bit fractional value cast to a signed 32-bit integer.

8.72 F64TOINT32

```
#include <SWLIBS_Defines.h>
```

8.72.1 Macro Definition:

```
#define F64TOINT32(x) \
        ((tS32) (x))
```

8.72.2 Description:

Type casting - lower 32 bits of a signed 64-bit fractional value cast to a signed 32-bit integer.

8.73 F16TOINT64

```
#include <SWLIBS_Defines.h>
```

8.73.1 Macro Definition:

```
#define F16TOINT64(x) \
        ((tS64) (x))
```

8.73.2 Description:

Type casting - signed 16-bit fractional value cast to a signed 64-bit integer, the value placed at the lower 16 bits of the 64-bit result.

8.74 F32TOINT64

```
#include <SWLIBS_Defines.h>
```

8.74.1 Macro Definition:

```
#define F32TOINT64(x) \
        ((tS64) (x))
```

8.74.2 Description:

Type casting - signed 32-bit fractional value cast to a signed 64-bit integer, the value placed at the lower 32 bits of the 64-bit result.

8.75 F64TOINT64

```
#include <SWLIBS_Defines.h>
```

8.75.1 Macro Definition:

```
#define F64TOINT64(x) \
        ((tS64) (x))
```

8.75.2 Description:

Type casting - signed 64-bit fractional value cast to a signed 64-bit integer.

8.76 INT16TOF16

```
#include <SWLIBS_Defines.h>
```

8.76.1 Macro Definition:

```
#define INT16TOF16(x) \
        ((tFrac16) (x))
```

8.76.2 Description:

Type casting - signed 16-bit integer value cast to a signed 16-bit fractional.

8.77 INT16TOF32

```
#include <SWLIBS_Defines.h>
```

8.77.1 Macro Definition:

```
#define INT16TOF32(x) \
        ((tFrac32) (x))
```

8.77.2 Description:

Type casting - signed 16-bit integer value cast to a signed 32-bit fractional, the value placed at the lower 16 bits of the 32-bit result.

8.78 INT32TOF16

```
#include <SWLIBS_Defines.h>
```

8.78.1 Macro Definition:

```
#define INT32TOF16(x) \
        ((tFrac16) (x))
```

8.78.2 Description:

Type casting - lower 16 bits of a signed 32-bit integer value cast to a signed 16-bit fractional.

8.79 INT32TOF32

```
#include <SWLIBS_Defines.h>
```

8.79.1 Macro Definition:

```
#define INT32TOF32(x) \
        ((tFrac32) (x))
```

8.79.2 Description:

Type casting - signed 32-bit integer value cast to a signed 32-bit fractional.

8.80 INT64TOF16

```
#include <SWLIBS_Defines.h>
```

8.80.1 Macro Definition:

```
#define INT64TOF16(x) \
        ((tFrac16) (x))
```

8.80.2 Description:

Type casting - lower 16 bits of a signed 64-bit integer value cast to a signed 16-bit fractional.

8.81 INT64TOF32

```
#include <SWLIBS_Defines.h>
```

8.81.1 Macro Definition:

```
#define INT64TOF32(x) \
        ((tFrac32) (x))
```

8.81.2 Description:

Type casting - lower 32 bits of a signed 64-bit integer value cast to a signed 32-bit fractional.

8.82 F16_1_DIVBY_SQRT3

```
#include <SWLIBS_Defines.h>
```

8.82.1 Macro Definition:

```
#define F16_1_DIVBY_SQRT3        ((tFrac16) 0x49E7)
```

8.82.2 Description:

One over sqrt(3) with a 16-bit result, the result rounded for a better precision,i.e. $round(1/sqrt(3)*2^{15})$

8.83 F32_1_DIVBY_SQRT3

```
#include <SWLIBS_Defines.h>
```

8.83.1 Macro Definition:

```
#define F32_1_DIVBY_SQRT3        ((tFrac32) 0x49E69D16)
```

8.83.2 Description:

One over sqrt(3) with a 32-bit result, the result rounded for a better precision, i.e. $round(1/sqrt(3)*2^{31})$

8.84 F16_SQRT3_DIVBY_2

```
#include <SWLIBS_Defines.h>
```

8.84.1 Macro Definition:

```
#define F16_SQRT3_DIVBY_2        ((tFrac16) 0x6EDA)
```

8.84.2 Description:

Sqrt(3) divided by two with a 16-bit result, the result rounded for a better precision, i.e. $round(sqrt(3)/2*2^{15})$

8.85 F32_SQRT3_DIVBY_2

```
#include <SWLIBS_Defines.h>
```

8.85.1 Macro Definition:

```
#define F32_SQRT3_DIVBY_2        ((tFrac32) 0x6ED9EBA1)
```

8.85.2 Description:

Sqrt(3) divided by two with a 32-bit result, the result rounded for a better precision, i.e. $round(sqrt(3)/2*2^{31})$

8.86 FRAC16

```
#include <SWLIBS_Defines.h>
```

8.86.1 Macro Definition:

```
#define FRAC16(x) \
        ((tFrac16) ((x) < (SFRACT_MAX) ? ((x) >= SFRACT_MIN ? (x)*0x8000 :
0x8000) :  0x7fff))
```

8.86.2 Description:

Macro converting a signed fractional [-1,1) number into a 16-bit fixed point number in format Q1.15.

8.87 FRAC32

```
#include <SWLIBS_Defines.h>
```

8.87.1 Macro Definition:

```
#define FRAC32(x) \
        ((tFrac32) ((x) < (FRACT_MAX) ? ((x) >= FRACT_MIN ? (x)*0x80000000 :
0x80000000) :  0x7fffffff))
```

8.87.2 Description:

Macro converting a signed fractional [-1,1) number into a 32-bit fixed point number in format Q1.31.

8.88 FLOAT_MIN

```
#include <SWLIBS_Defines.h>
```

8.88.1 Macro Definition:

```
#define FLOAT_MIN        ((tFloat) -3.4E+38)
```

8.88.2 Description:

Constant representing the maximal negative value of the 32-bit float type.

8.89 FLOAT_MAX

```
#include <SWLIBS_Defines.h>
```

8.89.1 Macro Definition:

```
#define FLOAT_MAX        ((tFloat) 3.4E+38)
```

8.89.2 Description:

Constant representing the maximal positive value of the 32-bit float type.

8.90 FLOAT_DIVBY_SQRT3

```
#include <SWLIBS_Defines.h>
```

8.90.1 Macro Definition:

```
#define FLOAT_DIVBY_SQRT3        ((tFloat) 0.5773502691896258)
```

8.90.2 Description:

One over sqrt(3) in single precision floating point format.

8.91 FLOAT_SQRT3_DIVBY_2

```
#include <SWLIBS_Defines.h>
```

8.91.1 Macro Definition:

```
#define FLOAT_SQRT3_DIVBY_2        ((tFloat) 0.866025403784439)
```

8.91.2 Description:

Sqrt(3) divided by two in single precision floating point format.

8.92 FLOAT_SQRT3_DIVBY_4

```
#include <SWLIBS_Defines.h>
```

8.92.1 Macro Definition:

```
#define FLOAT_SQRT3_DIVBY_4        ((tFloat) 0.4330127018922190)
```

8.92.2 Description:

Sqrt(3) divided by four in single precision floating point format.

8.93 FLOAT_SQRT3_DIVBY_4_CORRECTION

```
#include <SWLIBS_Defines.h>
```

8.93.1 Macro Definition:

```
#define FLOAT_SQRT3_DIVBY_4_CORRECTION        ((tFloat)0)
```

8.93.2 Description:

Sqrt(3) divided by four correction constant.

8.94 FLOAT_2_PI

```
#include <SWLIBS_Defines.h>
```

8.94.1 Macro Definition:

```
#define FLOAT_2_PI        ((tFloat) 6.28318530717958)
```

8.94.2 Description:

2*Pi in single precision floating point format.

8.95 FLOAT_PI

```
#include <SWLIBS_Defines.h>
```

8.95.1 Macro Definition:

```
#define FLOAT_PI        ((tFloat) 3.14159265358979)
```

8.95.2 Description:

Pi in single precision floating point format.

8.96 FLOAT_PI_DIVBY_2

```
#include <SWLIBS_Defines.h>
```

8.96.1 Macro Definition:

```
#define FLOAT_PI_DIVBY_2        ((tFloat) 1.57079632679490)
```

8.96.2 Description:

Pi/2 in single precision floating point format.

8.97 FLOAT_PI_SINGLE_CORRECTION

```
#include <SWLIBS_Defines.h>
```

8.97.1 Macro Definition:

```
#define FLOAT_PI_SINGLE_CORRECTION        ((tFloat)4.37102068E-8)
```

8.97.2 Description:

Double to single precision correction constant for Pi, equal to (PI(Double) - PI(Single))

8.98 FLOAT_PI_CORRECTION

```
#include <SWLIBS_Defines.h>
```

8.98.1 Macro Definition:

```
#define FLOAT_PI_CORRECTION        ((tFloat)8.74204136E-8)
```

8.98.2 Description:

Double to single precision correction constant for Pi, equal to (2*(PI(Double) - PI(Single))

8.99 FLOAT_PI_DIVBY_4

```
#include <SWLIBS_Defines.h>
```

8.99.1 Macro Definition:

```
#define FLOAT_PI_DIVBY_4        ((tFloat) 0.7853981633974480)
```

8.99.2 Description:

Pi/4 in single precision floating point format.

8.100 FLOAT_4_DIVBY_PI

```
#include <SWLIBS_Defines.h>
```

8.100.1 Macro Definition:

```
#define FLOAT_4_DIVBY_PI        ((tFloat) 1.2732395447351600)
```

8.100.2 Description:

Number four divided by PI in single precision floating point format.

8.101 FLOAT_0_5

```
#include <SWLIBS_Defines.h>
```

8.101.1 Macro Definition:

```
#define FLOAT_0_5        ((tFloat) 0.5)
```

8.101.2 Description:

Value 0.5 in single precision floating point format.

8.102 FLOAT_PLUS_1

```
#include <SWLIBS_Defines.h>
```

8.102.1 Macro Definition:

```
#define FLOAT_PLUS_1        ((tFloat) 1)
```

8.102.2 Description:

Value 1 in single precision floating point format.

8.103 FLOAT_MINUS_1

```
#include <SWLIBS_Defines.h>
```

8.103.1 Macro Definition:

```
#define FLOAT_MINUS_1        ((tFloat) -1)
```

8.103.2 Description:

Value -1 in single precision floating point format.

8.104 inline

```
#include <SWLIBS_Inlines.h>
```

8.104.1 Macro Definition:

```
#define inline        __inline__
```

8.104.2 Description:

8.105 _ARG2

```
#include <SWLIBS_MacroDisp.h>
```

8.105.1 Macro Definition:

```
#define _ARG2(_0, _1, _2, _3, ...)  \
        _3
```

8.105.2 Description:

8.106 VA_NUM_ARGS

```
#include <SWLIBS_MacroDisp.h>
```

8.106.1 Macro Definition:

```
#define VA_NUM_ARGS(...)  \
        _ARG2(__VA_ARGS__, 3, 2, 1, 0)
```

8.106.2 Description:

8.107 macro_dispatcher

```
#include <SWLIBS_MacroDisp.h>
```

8.107.1 Macro Definition:

```
#define macro_dispatcher(func, ...)  \
        macro_dispatcher_(func, VA_NUM_ARGS(__VA_ARGS__))
```

8.107.2 Description:

8.108 macro_dispatcher_

```
#include <SWLIBS_MacroDisp.h>
```

8.108.1 Macro Definition:

```
#define macro_dispatcher_(func, nargs) \
        macro_dispatcher__(func, nargs)
```

8.108.2 Description:

8.109 macro_dispatcher__

```
#include <SWLIBS_MacroDisp.h>
```

8.109.1 Macro Definition:

```
#define macro_dispatcher__(func, nargs) \
        func##_Dsptchr_## nargs
```

8.109.2 Description:

8.110 FALSE

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

8.110.1 Macro Definition:

```
#define FALSE        ((tBool)0)
```

8.110.2 Description:

Boolean type FALSE constant.

8.111 TRUE

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

8.111.1 Macro Definition:

```
#define TRUE        ((tBool)1)
```

8.111.2 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

RoHS-compliant and/or Pb-free versions of Freescale products have the functionality and electrical characteristics of their non-RoHS-compliant and/or non-Pb-free counterparts. For further information, see www.freescale.com or contact your Freescale sales representative.

For information on Freescale's Environmental Products program, go to www.freescale.com/epp

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

Freescale Semiconductor reserves the right to make changes without further notice to any products herein. Freescale Semiconductor makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale Semiconductor assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. "Typical" parameters that may be provided in Freescale Semiconductor data sheets and/or specifications can and do vary in different applications and actual performance may vary over time. All operating parameters, including "Typicals", must be validated for each customer application by customer's technical experts. Freescale Semiconductor does not convey any license under its patent rights nor the rights of others. Freescale Semiconductor products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Freescale Semiconductor product could create a situation where personal injury or death may occur. Should Buyer purchase or use Freescale Semiconductor products for any such unintended or unauthorized application, Buyer shall indemnify and hold Freescale Semiconductor and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Freescale Semiconductor was negligent regarding the design or manufacture of the part.

Freescale™ and the Freescale logo are trademarks of Freescale Semiconductor, Inc. All other product or service names are the property of their respective owners.

©Freescale Semiconductor, Inc. 2009. All rights reserved.

N/A
Rev. 0.1, 08/2011