freescale™
semiconductor

56F83xx
Motor Control Library

Reference Manual

Motor Control Library, 2nd Edition

for 56F83xx Motorola Hybrid Controllers

Rev. 2.0, 04/2005

freescale.com

Motor Control Library for 56F83xx Motorola Hybrid Controllers

Reference Manual — Rev. 2.0

Freescale Czech System Center
Roznov pod Radhostem, Czech Republic

The following revision history table summarizes changes contained in this document. For your convenience, the page number designators have been linked to the appropriate location.

Revision history

Date	Revision Level	Description	Page Number(s)
July 2003	1.0	Initial release	N/A
April 2005	2.0	Library files consolidation	N/A

Section 1. INTRODUCTION

Section 1. BASIC FUNCTIONS

Section 2. TRANSFORMATIONS

Section 3. CONTROLLERS

Section 4. RESOLVER PROCESSING

Section 5. MODULATION TECHNIQUES

Section 6. RAMP

Appendix A. References

Section 1. INTRODUCTION

1.1 Overview

This Reference Manual describes Motor Control Library for 56F83xx Motorola Hybrid Controllers. This library contains general-purpose algorithms that apply to motor control applications independent of the specific motor type. The library is supplied in both a binary form and as a source code. The binary form is unique by its simplicity to integrate with user application and source form enables enabling compile-time optimizations and custom-tailoring.

1.2 License

The Software is licensed to Licensee by Freescale Semiconductor Incorporated ("Freescale") for use under the terms of this Agreement. Freescale retains ownership of the Software. Freescale grants only the rights specifically granted in this Agreement and grants no other rights. Title to the Software, all copies thereof and all rights therein, including all rights in any intellectual property including patents, copyrights, and trade secrets applicable thereto, shall remain vested in Freescale.

For the Source, Freescale grants Licensee a personal, non-exclusive, non-assignable, revocable, royalty-free right to use, copy, and make derivatives of the Source solely in a development system environment in order to produce object code solely for operating on a Freescale semiconductor device having a central processing unit ("Derivative Object").

For the Object and Derivative Object, Freescale grants Licensee a personal, non-exclusive, non-assignable, revocable, royalty-free right to copy, use, and distribute the Object and the Derivative Object solely for operating on a Freescale semiconductor device having a central processing unit.

Licensee agrees to: (a) not use, modify, or copy the Software except as expressly provided herein, (b) not distribute, disclose, transfer, sell, assign, rent, lease, or otherwise make available the Software, any derivatives thereof, or this license to a third party except as expressly provided herein, (c) not remove obliterate, or otherwise defeat any copyright, trademark, patent or proprietary notices, related to the Software (d) not in any form export, re-export, resell, ship or divert or cause to be exported, re-exported, resold, shipped, or diverted, directly or indirectly, the Software or a direct product thereof to any country which the United States government or any agency thereof at the time of export or re-export requires an export license or other government approval without first obtaining such license or approval.

THE SOFTWARE IS PROVIDED ON AN "AS IS" BASIS AND WITHOUT WARRANTY OF ANY KIND INCLUDING (WITHOUT LIMITATION) ANY WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. IN NO EVENT SHALL FREESCALE BE LIABLE FOR ANY LIABILITY OR DAMAGES OF ANY KIND INCLUDING, WITHOUT LIMITATION, DIRECT OR INDIRECT

OR INCIDENTAL OR CONSEQUENTIAL OR PUNITIVE DAMAGES OR LOST PROFITS OR LOSS OF USE ARISING FROM USE OF THE SOFTWARE OR THE PRODUCT REGARDLESS OF THE FORM OF ACTION OR THEORY OF LIABILITY (INCLUDING WITHOUT LIMITATION, ACTION IN CONTRACT, NEGLIGENCE, OR PRODUCT LIABILITY) EVEN IF FREESCALE HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE. THIS DISCLAIMER OF WARRANTY EXTENDS TO LICENSEE OR USERS OF PRODUCTS AND IS IN LIEU OF ALL WARRANTIES WHETHER EXPRESS, IMPLIED, OR STATUTORY, INCLUDING IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR PARTICULAR PURPOSE.

Freescale does not represent or warrant that the Software is free of infringement of any third party patents, copyrights, trade secrets, or other intellectual property rights or that Freescale has the right to grant the licenses contained herein. Freescale does not represent or warrant that the Software is free of defect, or that it meets any particular requirements or need of the Licensee, or that it conforms to any documentation, or that it meets any standards.

Freescale shall not be responsible to maintain the Software, provide upgrades to the Software, or provide any field service of the Software. Freescale reserves the right to make changes to the Software without further notice to Licensee.

The Software is 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 Software could create a situation where personal injury or death may occur. Should Licensee purchase or use the Software for any such unintended or unauthorized application, Licensee shall indemnify and hold Freescale 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 was negligent regarding the design or manufacture of the Software.

The term of this Agreement is for as long as Licensee uses the Software for its intended purpose and is not in default of any provisions of this Agreement. Freescale may terminate this Agreement if Licensee is in default of any of the terms and conditions of this Agreement.

This Agreement shall be governed by and construed in accordance with the laws of the State of Arizona and can only be modified in a writing signed by both parties. Licensee agrees to jurisdiction and venue in the State of Arizona.

By using, modifying, installing, compiling, or copying the Software, Licensee acknowledges that this Agreement has been read and understood and agrees to be bound by its terms and conditions. Licensee agrees that this Agreement is the complete and exclusive statement of the agreement between Licensee and Freescale and supersedes any earlier proposal or prior arrangement, whether oral or written, and any other communications relative to the subject matter of this Agreement.

1.3 Supported Platforms

The Motor Control Library for 56F83xx Motorola Hybrid Controllers is written in C-language and assembler so it is compliant with all currently supported Freescale software tools. All algorithms are delivered in two library modules **MCLIB_56800ELDM.lib** and **MCLIB_56800ESDM.lib**, each module is intended for respective memory model. The **MCLIB_56800ELDM.lib** shall be integrated into a large data memory model project and **MCLIB_56800ESDM.lib** into a small data memory model project. The process of library integration into CodeWarrior project is discussed in **Section: 1.5 Library Integration**.

The interfaces to the algorithms included in this library have been combined into a single public interface include file, **mclib.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), defined and functionality provided for the individual algorithms.

1.4 Install Motor Control Library

The Motor Control Library is designed to assist developers in creating motor control applications. It contain numerous general and specific motor control algorithms, which can be used in connection with CodeWarrior developments tool for 56800E Hybrid Controllers.

If user want to fully use this library, the CodeWarrior tools should be installed prior the Motor Control Library. In case that Motor Control Library tool is installed while CodeWarrior is not present, users can only browse the installed software package, but will not be able to build, download and run code.

The installation itself consists of copying the required files to the destination hard drive, checking the presence of CodeWarrior and creating the shortcut under the Start->Programs menu.

NOTE: Each Motor Control Library release is installed in its own new directory named **MCLIB56F800E_rX.X** , where **X.X** denotes actual release number. This way of library installation enables users in maintaining older releases and projects and gives them free choice in selection of active library release.

To start the installation process, perform the following steps:

1. Execute **MCLIB56F800E_rXX.exe**
2. Follow the Motor Control Library software installation instructions on your screen.

After successful installation the motor control library is added by default into "**C:\Program Files\Freescale\MCLIB56F800E_rX.X**" subfolder. This folder will contain other nested subfolders and files required by Motor Control Library as shown in **Figure 1-1**.

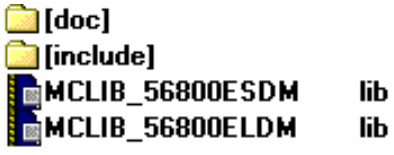

Figure 1-1. Installed subfolders and library files.

The list of all installed files and their brief description is given below:

...\MCLIB_56800ESDM.lib motor control library file for small data memory model,

...\MCLIB_56800ELDM.lib motor control library file for large data memory model,

...\doc\MCLIB_56F800E.pdf motor control library Reference Manual,

...\include\mclib.h unique motor control library header file. This file contains all motor control library function prototypes, user data types and other required constant definitions. This file must be included into any source file, where library functions or data objects shall be used.

In order to integrate Motor Control Library into a new CodeWarrior project the steps depicted in the next section must be performed.

1.5 Library Integration

The Motor Control Library is added into a new CodeWarrior project using the following steps:

1. Open new CodeWarrior project. From CodeWarrior menu choose **Project** > **New.** It will open project window as shown in **Figure 1-2**..

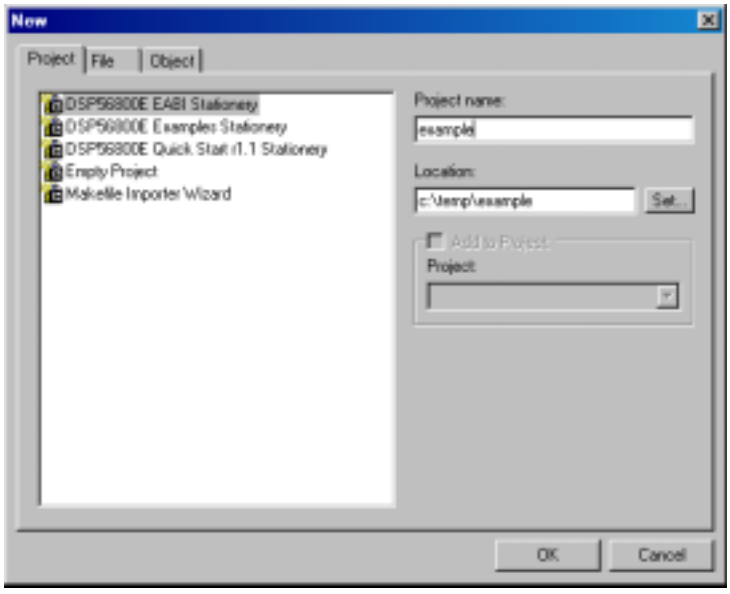

Figure 1-2. Selecting of new CodeWarrior project

Further select CodeWarrior project template, e.g. CodeWarrior DSP56800E EABI Stationery, type project name and click <**OK>.** After project name confirmation CodeWarrior enables user to choose device name and application type - see **Figure 1-3**.

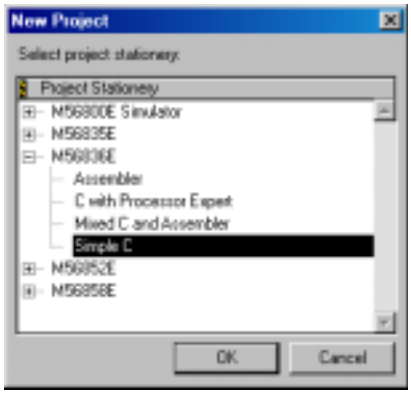

Figure 1-3. Selecting device name and application type.

2. Create “Motor Control” group in your new opened project. Note that this step is not mandatory, it is mentioned here just for purpose of maintaining file consistency in the CodeWarrior project window. From CodeWarrior menu choose **Project** > **Create Group...**, type *Motor Control* into appeared dialog window and click <**OK>.**

Figure 1-4. New project group creation

As a result of successful accomplishment of above step, the new **Motor Control** group has been created in the project window - see **Figure 1-5**.

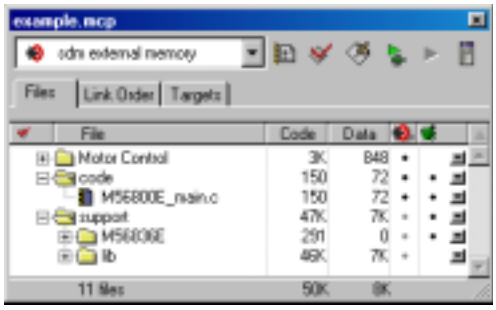

Figure 1-5. New project group created.

3. Reference appropriate **MCLIB_56800ExDM.lib** file in the project window. This can be achieved by dragging library file from proper library subfolder, e.g. **C:\Program Files\ Freescale\ MCLIB56F800E_rX.X\MCLIB_56800ESDM.lib ,** and dropping it under **Motor Control** group in the CodeWarrior project window as shown in **Figure 1-6**.

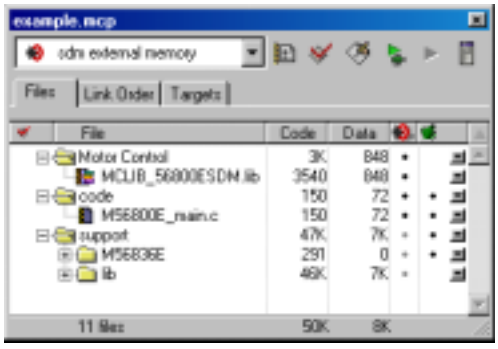

Figure 1-6. New project with referenced library files.

This step will also automatically add Motor Control Library path into project access paths, thus enabling users in exploiting library functions and consequential flawless project compilation and linking. The added path reference is shown in **Figure 1-7**.

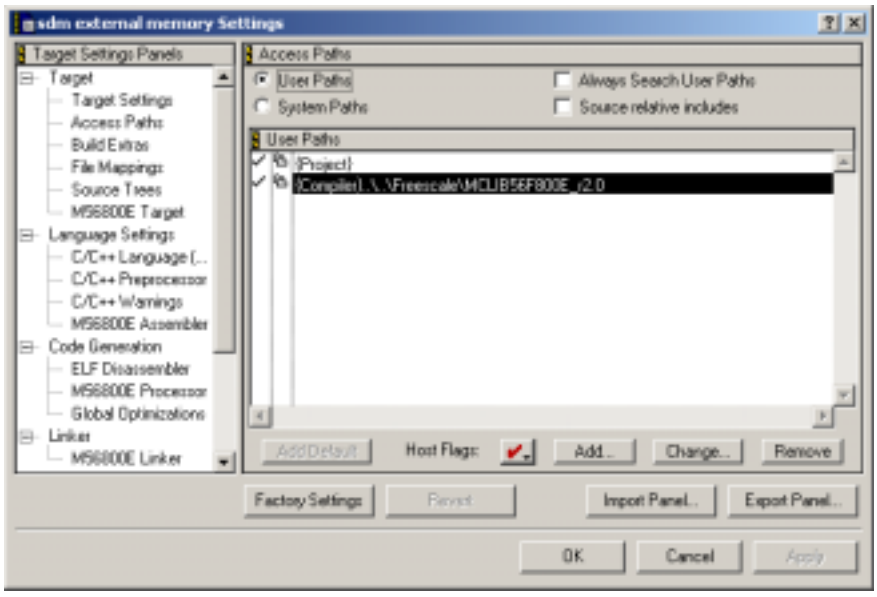

Figure 1-7. Automatically added project path.

1.6 API Definition

The following programming instructions must be added into a user application in order to use the Motor Control Library functions.

```
#include “mclib.h”
```

The “**mclib.h**” header file contains several additional header files that are needed for Motor Control Library integration into any user application. It includes **“mclib_types.h**” header file containing all general purpose data type definitions, see **Table 1-1**. Secondly, it includes all library specific header files; each one containing function prototypes and special data type definitions required by the corresponding source module.

1.7 User Definitions

Table 1-1. User generic definitions in mclib_types.h

Mnemonics	Size	Description
Word8	8 -bit	to represent 8-bit signed variable/value
UWord8	8 -bit	to represent 8-bit unsigned variable/value
Word16	16 -bit	to represent 16-bit signed variable/value
UWord16	16 -bit	to represent 16-bit unsigned variable/value
Word32	32 -bit	to represent 32-bit signed variable/value
UWord32	32 -bit	to represent 32-bit unsigned variable/value
Int8	8 -bit	to represent 8-bit signed variable/value
UInt8	8 -bit	to represent 8-bit unsigned variable/value
Int16	16 -bit	to represent 16-bit signed variable/value
UInt16	16 -bit	to represent 16-bit unsigned variable/value
Int32	32 -bit	to represent 32-bit signed variable/value
UInt32	32 -bit	to represent 32-bit unsigned variable/value
Frac16	16 -bit	to represent 16-bit signed variable/value
Frac32	32 -bit	to represent 32-bit signed variable/value
bool	16-bit	to represent boolean variable (true/false)
true	constant	represents true value
false	constant	represents false value
NULL	constant	represents NULL pointer
FRAC16()	macro	transform float value from <-1,1) range into fractional representation <-32768,32767>
FRAC32()	macro	transform float value from <-1,1) range into fractional representation <-2147483648,2147483648>

Table 1-2. User generic definitions in mclib_types.h

Name	Structure Members	Description
MC_3PhSyst	Frac16 a Frac16 b Frac16 c	three phase system
MC_2PhSyst	Frac16 alpha Frac16 beta	two phase system
MC_Angle	Frac16 sin Frac16 cos	sine and cosine components
MC_DqSyst	Frac16 d Frac16 q	generic DQ system
MC_PiParams	Frac16 propGain; Frac16 integGain; Frac32 integPartK_1; Frac16 posPiLimit; Frac16 negPiLimit; Word16 propGainSc; Word16 integGainSc; Word16 satFlag;	parameters of the PI1 and PI2 controllers

1.8 Library Functions and Macros

Each of the functions or macros in the MCLIB Library is described in further chapters. Each description consist of a number of subsections:

Synopsis This subsection gives the header files that should be included within a source file that references the function or macro. It also shows an appropriate declaration for the function or for a function that can be substituted for a macro. This declaration is not included in your program; only the header file(s) should be included.

Arguments This optional subsection describes input arguments to a function or macro.

Description This subsection is a description of the function or macro. It explains algorithms being used by functions or macro.

Returns This optional subsection describes the return value (if any) for the function or macro.

Range Issues This optional subsection specifies ranges of input variables.

Special Issues This optional subsection specifies a special assumptions that are mandatory for correct function calculation; e.g. saturation, rounding, etc.

Implementation This optional subsection specifies whether call into function generates library function call or macro expansion. This subsection also consist of one or more examples of the use of the function. The examples are often just fragments of code (not completed programs) for illustration purposes.

See Also This optional subsection provides a list of related functions or macros.

Performance This section specifies actual requirements of the function or macro in terms of required code memory, data memory and number of clock cycles to execute.

1.9 Library Benchmarks

Function Name	Code Size	Data Size	Execution Clocks[1]
MCLIB_Sin	44 words	48 words	84 cycles
MCLIB_Cos	11 words	56 words	107 cycles
MCLIB_Tan	62 words	48 words	65-107 cycles
MCLIB_Atan	40 words	48 words	77-79 cycles
MCLIB_AtanYX	106 words	0 words	81-177 cycles
MCLIB_Asin	66 words	40 words	104-218 cycles
MCLIB_Acos	8 words	70 words	124-238 cycles
MCLIB_Sqrt	68 words	48 words	105 cycles
MCLIB_SetRandSeed16	4 word	1 words	18 cycles
MCLIB_Rand16	21 word	1 words	91 cycles
MCLIB_GetSetSaturationMode	17 word	0 words	36 cycles
MCLIB_InitAtanYXShifted	19 words	0 words	44 words
MCLIB_AtanYXShifted	47 words	2 words	208-216 words
MCLIB_ClarkTrfm	14 word	0 words	61 cycles
MCLIB_ClarkTrfmInv	16 word	0 words	73 cycles
MCLIB_ParkTrfm	17 word	0 words	91 cycles
MCLIB_ParkTrfmInv	17 word	0 words	92 cycles
MCLIB_ControllerPI	52 word	0 words	82 cycles
MCLIB_InitTrackObsv	37 word	0 words	56 cycles
MCLIB_CalcTrackObsv	248 word	13 words	271-280 cycles
MCLIB_GetResPosition	15 word	0 words	29 cycles
MCLIB_GetResSpeed	4 word	0 words	17 cycles
MCLIB_GetResRevolutions	4 word	0 words	17 cycles
MCLIB_SetResPosition	15 word	0 words	28 cycles
MCLIB_SetResRevolutions	4 word	0 words	18 cycles

MCLIB_InitSinCos	116 word	0 words	790 cycles
MCLIB_RawSinCosPos	133 word	0 words	214-247 cycles
MCLIB_CalcSinCosObs	17 word	24 words	366 cycles
MCLIB_GetSinCosPos	15 word	0 words	28 cycles
MCLIB_GetSinCosSpeed	4 word	0 words	18 cycles
MCLIB_GetSinCosRev	4 word	0 words	18 cycles
MCLIB_SetSinCosPos	15 word	0 words	33 cycles
MCLIB_SetSinCosRev	4 word	0 words	21 cycles
MCLIB_SvmStd	138 word	0 words	91-104 cycles
MCLIB_SvmU0n	123 word	0 words	88-100 cycles
MCLIB_SvmU7n	131 word	0 words	89-101 cycles
MCLIB_SvmAlt	125 word	0 words	88-100 cycles
MCLIB_PwmIct	70 word	0 words	106-107 cycles
MCLIB_SvmSci	130 word	0 words	147-178 cycles
MCLIB_ElimDcBusRip	70 word	0 words	135-163 cycles

1. Number of clock cycles; on 56F83xx controllers one clock cycle equals to 16.7ns @ 60MHz IP Bus clock frequency

Section 1. BASIC FUNCTIONS

1.1 API Summary

Table 1-1. API Function Summary

Name	Arguments	Return	Description
MCLIB_Sin	Frac16 x	Frac16	Function calculates the sine value of the argument using piece-wise polynomial approximation method, see **Section: 1.2 MCLIB_Sin**
MCLIB_Cos	Frac16 x	Frac16	Function calculates the cosine value of the argument using piece-wise polynomial approximation method, see **Section: 1.3 MCLIB_Cos**
MCLIB_Sin2	Frac16 x	Frac16	Function calculates the sine value of the argument using lookup table, see **Section: 1.4 MCLIB_Sin2**
MCLIB_Cos2	Frac16 x	Frac16	Function calculates the cosine value of the argument using lookup table, see **Section: 1.5 MCLIB_Cos2**
MCLIB_Tan	Frac16 x	Frac16	Function calculates the tangent value of the argument using piece-wise polynomial approximation, see **Section: 1.6 MCLIB_Tan**
MCLIB_Atan	Frac16 x	Frac16	Function calculates the arcus tangent value of the argument using piece-wise polynomial approximation, see **Section: 1.7 MCLIB_Atan**
MCLIB_AtanYX	Frac16 y, Frac16 x	Frac16	Function calculates the arcus tangent value based on the provided x, y co-ordinates as arguments using division and piece-wise polynomial approximation, see **Section: 1.8 MCLIB_AtanYX**
MCLIB_Asin	Frac16 x	Frac16	Function calculates the arcus sine value of the argument using piece-wise polynomial approximation, see **Section: 1.9 MCLIB_Asin**
MCLIB_Acos	Frac16 x	Frac16	Function calculates the arcus sine value of the argument using piece-wise polynomial approximation, see **Section: 1.10 MCLIB_Acos**

Table 1-1. API Function Summary

Name	Arguments	Return	Description
MCLIB_Sqrt	Frac16 x	Frac16	Function calculates the square root value of the argument using piece-wise polynomial approximation with post-adjustment method, see **Section: 1.11 MCLIB_Sqrt**
MCLIB_SetRandSeed16	Frac16 x	none	Function sets new seed of random number generator, see function description is in the **Section: 1.12 MCLIB_SetRandSeed16**
MCLIB_Rand16	none	Frac16	Function generates random number, see function description in the **Section: 1.13 MCLIB_Rand16**
MCLIB_GetSetSaturationMode	bool saturationMode	bool	Function set new saturation mode and returns current one, see function description in the **Section: 1.14 MCLIB_GetSetSaturationMode**
MCLIB_InitAtanYXShifted	Frac16 Ky Frac16 Kx Frac16 Ny Frac16 Nx Frac16 ThetaAdj	void	Initializes internal state of the **MCLIB_AtanYXShifted** function, see function description in the **Section: 1.15 MCLIB_InitAtanYXShifted**
MCLIB_AtanYXShifted	Frac16 y Frac16 x	Frac16	Computes angle of two sine waves shifted in phase one to each other. See function description in the **Section: 1.16 MCLIB_AtanYXShifted**

1.2 MCLIB_Sin

1.2.1 Synopsis

#include"mclib.h"
Frac16 ***MCLIB_Sin*** *(Frac16 x)*

1.2.2 Arguments

x	in	input data value

1.2.3 Description

The **MCLIB_Sin** function calculates sin(pi*x) using piece-wise polynomial approximation.

1.2.4 Returns

The function returns result of sin(pi*x).

1.2.5 Range Issues

The input data value is in the range of [-1,1), which corresponds to the angle in the range of [-pi,pi). The output data value is in the range of [-1,1). It means that the function value of 0.5, which corresponds to pi/2, will be 0x7FFF and of -0.5, which corresponds to -pi/2, will be 0x8000.

1.2.6 Special Issues

The function requires the saturation mode to be set.

1.2.7 Implementation

The **MCLIB_Sin** is implemented as a function call.

Code Example 1-1. MCLIB_Sin

```
/* include function and data prototypes required by the module  */
#include "mclib.h"

/* input data value in range <-1,1) corresponds to <-pi,pi)     */
#define PIBY4 0.25              /* 0.25 equals to pi/4          */

void main (void)
{
   Frac16 x, y;

   __turn_on_sat();

   x= FRAC16(PIBY4);

   /* Compute the sine value                                    */
   y = MCLIB_Sin(x);             /* y should contain 23171      */
}
```

1.2.8 Performance

Table 1-2. Performance of the MCLIB_Sin function.

Code Size	44 words	
Data Size	48 words	
Execution Clocks	Min	84 cycles
	Max	84 cycles

1.3 MCLIB_Cos

1.3.1 Synopsis

#include ”mclib.h”
Frac16 ***MCLIB_Cos*** *(Frac16 x)*

1.3.2 Arguments

x	in	input data value

1.3.3 Description

The **MCLIB_Cos** function computes the cos(pi*x) using sine calculation. The sine calculation is made with use of piece-wise polynomial approximation.

1.3.4 Returns

The function returns cos(pi*x).

1.3.5 Range Issues

The input data value is in the range of [-1,1), which corresponds to the angle in the range of [-pi,pi) and the output data value is in the range [-1, 1). It means that the function value of 0, which corresponds to 0*pi=0, will be 0x7FFF and of 1, which corresponds to pi, will be 0x8000.

1.3.6 Special Issues

The function requires the saturation mode to be set.

The function is very short and for speed reason it may be reasonable to consider to create an inlined version of the function.

1.3.7 Implementation

The **MCLIB_Cos** is implemented as a function call.

Code Example 1-2. MCLIB_Cos

```
/* include function and data prototypes required by the module  */
#include "mclib.h"

/* input data value in range <-1,1) corresponds to <-pi,pi)     */
#define PIBY4 0.25            /* 0.25 equals to pi/4            */

void main (void)
{
   Frac16 x, y;

   __turn_on_sat();

   x= FRAC16(PIBY4);

   /* Compute the sine value                                     */
   y = MCLIB_Cos(x);          /* y should contain 23171         */
}
```

1.3.8 Performance

Table 1-3. Performance of the MCLIB_Cos function.

Code Size	11 words	
Data Size	0 words	
Execution Clocks	Min	107 cycles
	Max	107 cycles

1.4 MCLIB_Sin2

1.4.1 Synopsis

#include"mclib.h"
Frac16 ***MCLIB_Sin2*** *(Frac16 x)*

1.4.2 Arguments

x	in	input data value

1.4.3 Description

The **MCLIB_Sin2** function calculates sin(pi*x) using lookup table.

1.4.4 Returns

The function returns result of sin(pi*x).

1.4.5 Range Issues

The input data value is in the range of [-1,1), which corresponds to the angle in the range of [-pi,pi). The output data value is in the range of [-1,1). It means that the function value of 0.5, which corresponds to pi/2, will be 0x7FFF and of -0.5, which corresponds to -pi/2, will be 0x8000.

1.4.6 Special Issues

The function requires the saturation mode to be set.

1.4.7 Implementation

The **MCLIB_Sin2** is implemented as an inlined function .

Code Example 1-3. MCLIB_Sin2

```
/* include function and data prototypes required by the module  */
#include "mclib.h"

/* input data value in range <-1,1) corresponds to <-pi,pi)     */
#define PIBY4 0.25              /* 0.25 equals to pi/4          */

void main (void)
{
   Frac16 x, y;

   __turn_on_sat();

   x= FRAC16(PIBY4);

   /* Compute the sine value                                    */
   y = MCLIB_Sin2(x);             /* y should contain 23170     */
}
```

1.4.8 Performance

Table 1-4. Performance of the MCLIB_Sin2 function.

Code Size	285 words	
Data Size	0 words	
Execution Clocks	Min	38 cycles
	Max	38 cycles

1.5 MCLIB_Cos2

1.5.1 Synopsis

#include ”mclib.h”
Frac16 ***MCLIB_Cos2*** *(Frac16 x)*

1.5.2 Arguments

x	in	input data value

1.5.3 Description

The **MCLIB_Cos2** function computes the cos(pi*x) using sine calculation. The sine calculation is made with use of lookup table.

1.5.4 Returns

The function returns cos(pi*x).

1.5.5 Range Issues

The input data value is in the range of [-1,1), which corresponds to the angle in the range of [-pi,pi) and the output data value is in the range [-1, 1). It means that the function value of 0, which corresponds to 0*pi=0, will be 0x7FFF and of 1, which corresponds to pi, will be 0x8000.

1.5.6 Special Issues

The function requires the saturation mode to be set.

1.5.7 Implementation

The **MCLIB_Cos2** is implemented as an inlined function.

Code Example 1-4. MCLIB_Cos2

```
/* include function and data prototypes required by the module  */
#include "mclib.h"

/* input data value in range <-1,1) corresponds to <-pi,pi)     */
#define PIBY4 0.25            /* 0.25 equals to pi/4            */

void main (void)
{
   Frac16 x, y;

   __turn_on_sat();

   x= FRAC16(PIBY4);

   /* Compute the sine value                                     */
   y = MCLIB_Cos2(x);          /* y should contain 23170         */
}
```

1.5.8 Performance

Table 1-5. Performance of the MCLIB_Cos2 function.

Code Size	291 words	
Data Size	0 words	
Execution Clocks	Min	46 cycles
	Max	46 cycles

1.6 MCLIB_Tan

1.6.1 Synopsis

#include ”mclib.h”
Frac16 ***MCLIB_Tan*** *(Frac16 x)*

1.6.2 Arguments

x	in	input data value

1.6.3 Description

The **MCLIB_Tan** function computes the tan(pi*x) using piece-wise polynomial approximation.

Due to limits of fractional arithmetic all tangent values falling beyond [-1, 1), are truncated to -1 and 1 respectively. The exact values are provided in the table **Table 1-6**.

Table 1-6. MCLIB_Tan Function Values

Input argument value		Computed result	
Real	**Fractional (hex)**	**Real**	**Fractional (hex)**
$\left(-\pi, -\frac{3}{4}\cdot\pi\right)$	(8000, A000)	tan	tan
$-\frac{3}{4}\cdot\pi$	A000	1.0	7FFF
$\left(-\frac{3}{4}\cdot\pi, -\frac{1}{2}\cdot\pi\right)$	(A000, C000)	1.0	7FFF
$-\frac{1}{2}\cdot\pi$	C000	-1.0	8000
$\left(-\frac{1}{2}\cdot\pi, -\frac{1}{4}\cdot\pi\right)$	(C000, E000)	-1.0	8000
$-\frac{1}{4}\cdot\pi$	E000	-1.0	8000
$\left(-\frac{1}{4}\cdot\pi, \frac{1}{4}\cdot\pi\right)$	(E000, 2000)	tan	tan

Table 1-6. MCLIB_Tan Function Values

Input argument value		Computed result	
Real	**Fractional (hex)**	**Real**	**Fractional (hex)**
$\frac{1}{4} \cdot \pi$	2000	1.0	7FFF
$\left(\frac{1}{4} \cdot \pi, \frac{1}{2} \cdot \pi\right)$	(2000,4000)	1.0	7FFF
$\frac{1}{2} \cdot \pi$	4000	1.0	7FFF
$\left(\frac{1}{2} \cdot \pi, \frac{3}{2} \cdot \pi\right)$	(4000, 6000)	-1.0	8000
$\left(\frac{3}{4} \cdot \pi, \pi\right)$	(6000,7FFF)	tan	tan

1.6.4 Returns

The function returns tan(pi*x) with limits imposed by fractional arithmetic.

1.6.5 Range Issues

The input data value is in the range of [-1,1), which corresponds to the angle in the range of [-pi,pi). The output data value is in the range of [-1, 1), which means that function value of 0.25, which corresponds to pi/4, will be 0x7FFF and of -0.25, which corresponds to -pi/4, will be 0x8000.

If a tangent value is beyond the permissible range of [-1, 1), then it is truncated respectively to -1 and 1.

1.6.6 Special Issues

The function calculation is correct regardless of saturation mode.

1.6.7 Implementation

The **MCLIB_Tan** is implemented as a function call.

Code Example 1-5. MCLIB_Tan

```
/* include function and data prototypes required by the module  */
#include "mclib.h"

/* angle value of <-1,1) corresponds to <-pi,pi)                */
#define PIBY4     0.25             /* 0.25 equals to pi/4       */

void main (void)
{
   Frac16 x, y;

   __turn_on_sat();

   /* Compute the tangent value                                  */
   x = FRAC16(PIBY4);
   y = MCLIB_Tan(x);               /* y = 0x7FFF equals to 1     */
}
```

1.6.8 Performance

Table 1-7. Performance of the MCLIB_Tan function.

Code Size	62 words	
Data Size	56 words	
Execution Clocks	Min	65 cycles
	Max	107 cycles

1.7 MCLIB_Atan

1.7.1 Synopsis

#include ”mclib.h”
Frac16 ***MCLIB_Atan*** *(Frac16 x)*

1.7.2 Arguments

x	in	input data value

1.7.3 Description

The **MCLIB_Atan** function computes the atan(x)/(pi/4) using piece-wise polynomial approximation.

1.7.4 Returns

The function returns atan(x)/(pi/4).

1.7.5 Range Issues

The input data value is in the range of [-1,1). The output data value is in range [-0.25, 0.25), which corresponds to the angle in the range of [-pi/4,pi/4).

1.7.6 Special Issues

The function requires the saturation mode to be set.

1.7.7 Implementation

The **MCLIB_Atan** is implemented as a function call.

Code Example 1-6. MCLIB_Atan

```
/* include function and data prototypes required by the module  */
#include "mclib.h"

void main (void)
{
   Frac16 x, y;

  __turn_on_sat();

   x= FRAC16(1.0);

   /* Compute the arcus tangent value                            */
   y = MCLIB_Atan(x);                 /* y should contain 8192  */
}
```

1.7.8 Performance

Table 1-8. Performance of the MCLIB_Atan function.

Code Size	40 words	
Data Size	48 words	
Execution Clocks	Min	77 cycles
	Max	79 cycles

1.8 MCLIB_AtanYX

1.8.1 Synopsis

#include ”mclib.h”
Frac16 ***MCLIB_AtanYX*** *(Frac16 y, Frac16 x)*

1.8.2 Arguments

y	in	input data value, y co-ordinate
x	in	input data value, x co-ordinate

1.8.3 Description

The **MCLIB_AtanYX** function computes angle whose tangent is y/x using division of input arguments (y divided by x) and piece-wise polynomial approximation. This function is fractional counterpart of the well-known atan2() function defined in ANSI-C math library.

In opposite to the **MCLIB_Atan**, the function correctly places the calculated angle in the whole [-pi, pi) range, basing on signs of x and y co-ordinates provided as arguments.

If the x, y co-ordinates both are 0 (zero), then the function returns 0.25, which corresponds to pi/4.

1.8.4 Returns

The function returns angle whose tangent is y/x (arcus tangent from x, y co-ordinates).

1.8.5 Range Issues

The input data value is in the range of [-1,1). The output data value is in the range [-1, 1), which corresponds to the angle in the range of [-pi,pi).

1.8.6 Special Issues

The function calculates correct results regardless of the saturation mode.

1.8.7 Implementation

The **MCLIB_AtanYX** is implemented as a function call.

Code Example 1-7. MCLIB_AtanYX

```
/* include function and data prototypes required by the module  */
#include "mclib.h"

void main (void)
{
   Frac16 x, y, z;

   __turn_on_sat();

   x = FRAC16(0.5);
   y = FRAC16(1.0);

   /* Compute the arcus tangent value                            */
   z = MCLIB_AtanYX(y, x);             /* z should contain 11547 */
}
```

1.8.8 Performance

Table 1-9. Performance of the MCLIB_AtanYX function.

Code Size	106 words	
Data Size	0 words	
Execution Clocks	Min	81 cycles
	Max	177 cycles

1.9 MCLIB_Asin

1.9.1 Synopsis

#include ”mclib.h”
Frac16 ***MCLIB_Asin*** *(Frac16 x)*

1.9.2 Arguments

x	in	input data value

1.9.3 Description

The **MCLIB_Asin** function computes the asin(x)/(pi/2) using piece-wise polynomial approximation.

1.9.4 Returns

The function returns asin(x)/(pi/2).

1.9.5 Range Issues

The input data value is in the range of [-1,1). The output data value is in range [-0.5, 0.5), which corresponds to the angle in the range of [-pi/2,pi/2).

1.9.6 Special Issues

The function requires the saturation mode to be set.

1.9.7 Implementation

The **MCLIB_Asin** is implemented as a function call.

Code Example 1-8. MCLIB_Asin

```
/* include function and data prototypes required by the module  */
#include "mclib.h"

void main (void)
{
   Frac16 x, y;

   __turn_on_sat();

   x= FRAC16(0.5);

   /* Compute the arcus sine value                                */
   y = MCLIB_Asin(x);          /* y should contain 5462           */
}
```

1.9.8 Performance

Table 1-10. Performance of the MCLIB_Asin function.

Code Size	66 words	
Data Size	40 words	
Execution Clocks	Min	104 cycles
	Max	218 cycles

1.10 MCLIB_Acos

1.10.1 Synopsis

#include ”mclib.h”
Frac16 ***MCLIB_Acos*** *(Frac16 x)*

1.10.2 Arguments

x	in	input data value

1.10.3 Description

The **MCLIB_Acos** function computes the acos(x)/(pi/2) using the arcus sine calculation. The sine calculation is made with use of piece-wise polynomial approximation.

1.10.4 Returns

The function returns acos(x)/(pi/2).

1.10.5 Range Issues

The input data value is in the range of [-1,1). The output data value is in range [0,1), which corresponds to the angle in the range of [0,pi).

1.10.6 Special Issues

The function requires the saturation mode to be set.

The function is very short and for speed reasons it may be reasonable to create an inlined version of the function.

1.10.7 Implementation

The **MCLIB_Acos** is implemented as a function call.

Code Example 1-9. MCLIB_Acos

```
/* include function and data prototypes required by the module  */
#include "mclib.h"

void main (void)
{
   Frac16 x, y;

   __turn_on_sat();

   x= FRAC16(0.5);

   /* Compute the arcus sine value                                */
   y = MCLIB_Acos(x);              /* y should contain 10922     */
}
```

1.10.8 Performance

Table 1-11. Performance of the MCLIB_Acos function.

Code Size	8 words	
Data Size	0 words	
Execution Clocks	Min	124 cycles
	Max	238 cycles

1.11 MCLIB_Sqrt

1.11.1 Synopsis

#include ”mclib.h”
Frac16 ***MCLIB_Sqrt****(Frac32 x)*

1.11.2 Arguments

x	in	input data value

1.11.3 Description

The **MCLIB_Sqrt** function computes the square root of x using piece-wise polynomial approximation and post-adjustment of the least significant bits.

The function calculates the square root correctly for values equal to or larger than zero. For negative arguments the function may behave unpredictably.

1.11.4 Returns

For argument equal to or larger than zero the function returns the square root of argument, which is a number which if squared is the best approximation of the argument. The function rounds correctly the least significant bit.

For negative arguments the function returns undefined value.

1.11.5 Range Issues

The input data value is in the range of [0,1), expressed with 32-bits precision. The output data value is in the range of [0, 1) expressed with 16-bits precision.

1.11.6 Special Issues

The function calculates correct results regardless of the saturation mode.

1.11.7 Implementation

The **MCLIB_Sqrt** is implemented as a function call.

Code Example 1-10. MCLIB_Sqrt

```
/* include function and data prototypes required by the module  */
#include "mclib.h"

void main (void)
{
   Frac32 x;
   Frac16 y;

   __turn_on_sat();

   x= FRAC32(0.5);

   /* Compute the sqrt value                                    */
   y = MCLIB_Sqrt(x);               /* y should contain 23170   */
}
```

1.11.8 Performance

Table 1-12. Performance of the MCLIB_Sqrt function.

Code Size	68 words	
Data Size	70 words	
Execution Clocks	Min	105 cycles
	Max	105 cycles

1.12 MCLIB_SetRandSeed16

1.12.1 Synopsis

#include ”mclib.h”
*void **MCLIB_SetRandSeed16** (Frac16 x);*

1.12.2 Arguments

x	in	new seed of random number generator

1.12.3 Description

The **MCLIB_SetRandSeed16** function sets new seed of random number generator.

1.12.4 Implementation

The **MCLIB_SetRandSeed16** is implemented as a function call.

Code Example 1-11. MCLIB_SetRandSeed16

```
/* include function and data prototypes required by the module  */
#include "mclib.h"

/* main function call                                           */
void main (void)
{
  /* set new seed for random number generator                */
   MCLIB_SetRandSeed16(10000);
}
```

1.12.5 Performance

Table 1-13. Performance of the MCLIB_SetRandSeed16 function.

Code Size	4 words	
Data Size	1 words	
Execution Clocks	Min	18 cycles
	Max	18 cycles

1.13 MCLIB_Rand16

1.13.1 Synopsis

#include ”mclib.h”
Frac16 ***MCLIB_Rand16*** *(void);*

1.13.2 Description

The **MCLIB_Rand16** function returns random number, it is obtain by using linear congruential generator, which generate a sequence of integers *I1, I2, I3,* ..., by the recurrence relation

$$I_{j+1} = a \cdot I_j + c \qquad \textbf{(EQ 1-1.)}$$

where *a* and *c* are positive integer numbers (*a* = 31821 and *c* = 13849) called the multiplier and the increment respectively.

1.13.3 Return

The function returns 16-bit signed fractional value in the range -1 to 1.

1.13.4 Implementation

The **MCLIB_Rand16** is implemented as a function call.

Code Example 1-12. MCLIB_Rand16

```
/* include function and data prototypes required by the module  */
#include "mclib.h"

/* main function call                                           */
void main (void)
{
   Frac16 temp;

  /* generate random number                                     */
   temp = MCLIB_Rand16();
}
```

1.13.5 Performance

Table 1-14. Performance of the MCLIB_Rand16 function.

Code Size	21 words	
Data Size	1 words	
Execution Clocks	Min	91 cycles
	Max	91 cycles

1.14 MCLIB_GetSetSaturationMode

1.14.1 Synopsis

#include ”mclib.h”
Frac16 ***MCLIB_GetSetSaturationMode*** *(void);*

1.14.2 Arguments

saturationMode	in	new value of saturation bit of Operating Mode Register

1.14.3 Description

The **MCLIB_GetSetSaturationMode** function sets the value of saturation bit in Operating Mode Register (OMR) according to the input *saturationMode* parameter.

1.14.4 Return

The function returns current state of saturation bit in Operating Mode Register.

1.14.5 Implementation

The **MCLIB_GetSetSaturationMode** is implemented as a function call.

Code Example 1-13. MCLIB_GetSetSaturationMode

```
/* include function and data prototypes required by the module  */
#include "mclib.h"

/* main function call                                           */
void main (void)
{
  bool mode;

  /* set saturation and save old saturation state               */
  mode = true;
  mode = MCLIB_GetSetSaturationMode(mode);
}
```

1.14.6 Performance

Table 1-15. Performance of the MCLIB_GetSetSaturationMode function.

Code Size	17 words	
Data Size	0 words	
Execution Clocks	Min	36 cycles
	Max	36 cycles

1.15 MCLIB_InitAtanYXShifted

1.15.1 Synopsis

#include ”mclib.h”
void ***MCLIB_InitAtan2Shifted*** *(Frac16 ky, Frac16 kx, Frac16 ny, Frac16 nx, Frac16 thetaAdj);*

1.15.2 Arguments

ky	in	multiplication coefficient of y signal
kx	in	multiplication coefficient of x signal
ny	in	scaling coefficient of y signal
nx	in	scaling coefficient of x signal
thetaAdj	in	adjusting angle

1.15.3 Description

This function initializes internal variables of the **MCLIB_AtanYXShifted** function. The function should be called before the first call to the **MCLIB_AtanYXShifted** function.

The initialization parameters can be calculated according to algorithm provided as Matlab code, see **Code Example 1-15. Initialization Parameters Calculation in Matlab** on page 33.

1.15.4 Implementation

The **MCLIB_InitAtanYXShifted** function is implemented as a function call.

Code Example 1-14. MCLIB_InitAtanYXShifted

```
/* include function and data prototypes required by the module */
#include "mclib.h"

/* Define appropriate data                                      */
/* dtheta = 5deg, thetaoffset = 0deg                            */
#define NY                          0
#define NX                          4
#define KY                          FRAC16(0.50046106925577549)
#define KX                          FRAC16(0.71640268727196699)
#define THETAADJ                    FRAC16(0.01388549804687500)


void main (void)
{
   /* Initialize MCLIB_AtanYXShifted function                   */
   MCLIB_InitAtanYXShifted(KY, KX, NY, NX, THETAADJ);
}
```

1.15.5 Performance

Table 1-16. Performance of the MCLIB_InitAtanYXShifted function.

Code Size	19 words	
Data Size	0 words	
Execution Clocks	Min	44 cycles
	Max	44 cycles

1.16 MCLIB_AtanYXShifted

1.16.1 Synopsis

#include ”mclib.h”
Frac16 ***MCLIB_AtanYXShifted****(Frac16 y, Frac16 x)*

1.16.2 Arguments

y	in	input data value, equal to $\sin\theta$
x	in	input data value, equal to $\sin(\theta + \Delta\theta)$

1.16.3 Description

The **MCLIB_AtanYXShifted** function computes an angle assuming that the arguments denote the following values:

$$y = \sin\theta$$
$$x = \sin(\theta + \Delta\theta) \qquad \textbf{(EQ 1-2.)}$$

where:

y,x	signal values provided as arguments
θ	angle to be computed by the function
$\Delta\theta$	phase difference between y,x signals

If $\Delta\theta = \pi/2$ or $\Delta\theta = -\pi/2$ then the function is similar to the **MCLIB_AtanYX** function, however the **MCLIB_AtanYX** function in this case is more effective.

The function uses the following algorithm for computing the angle:

$$b = \frac{S}{2 \cdot \cos\frac{\Delta\theta}{2}} \cdot (y + x)$$
$$a = \frac{S}{2 \cdot \sin\frac{\Delta\theta}{2}} \cdot (x - y) \qquad \textbf{(EQ 1-3.)}$$
$$\theta = \text{atan2}(b,a) - \Delta\theta/2 + \theta_{offset}$$

where:

y,x	signal values provided as arguments as in **EQ 1-2**
θ	angle to be computed by the function as in **EQ 1-2**
θ_{offset}	angle offset
S	scaling coefficient to prevent from overflow over 1, S is nearly 1 and $S < 1$
a,b	intermediate variables

For the purposes of fractional conventions the algorithms is implemented so that some addition values are used as shown in the equation **EQ 1-4** below:

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

$$\frac{S}{2 \cdot \sin\frac{\Delta\theta}{2}} = K_x \cdot 2^{N_x} \qquad \textbf{(EQ 1-4.)}$$

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

where:

K_y	multiplication coefficient of y signal
N_y	scaling coefficient of y signal
K_x	multiplication coefficient of x signal
N_x	scaling coefficient of x signal
θ_{adj}	adjusting angle

The signal values need to be provided to the function as arguments. The phase difference $\Delta\theta$ need to be set through an initialization function see **Section 1.15, “MCLIB_InitAtanYXShifted”** .

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

Code Example 1-15. Initialization Parameters Calculation in Matlab

```
function [KY, KX, NY, NX, THETAADJ] = atan2shiftedpar(dthdeg, thoffsetdeg)
% ATAN2SHIFTEDPAR  calculation of parameters for atan2shifted() function
%
%   [NY, NX, KY, KX, THETAADJ] = atan2shiftedpar(dthdeg, thoffsetdeg)
%
%   dthdeg = phase shift between sine waves in degrees
%   thoffsetdeg = angle offset in degrees
%   NY       - scaling coefficient of y signal
%   NX       - scaling coefficient of x signal
%   KY       - multiplication coefficient of y signal
%   KX       - multiplication coefficient of x signal
%   THETAADJ - adjusting angle in radians, scaled from [-pi, pi) to [-1, 1)
    dth2 = ((dthdeg/2)/180*pi);
    thoffset = (thoffsetdeg/180*pi);
    CY = (1 - 2^-15)/(2*cos(dth2));
    CX = (1 - 2^-15)/(2*sin(dth2));

    if(abs(CY) >= 1) NY = ceil(log2(abs(CY)));
    else             NY = 0;
    end

    if(abs(CX) >= 1) NX = ceil(log2(abs(CX)));
    else             NX = 0;
    end

    KY = CY/2^NY;
    KX = CX/2^NX;

    THETAADJ = (dthdeg/2 - thoffsetdeg)/180;
```

For example if $\Delta\theta = 69{,}33°,\ \theta = 0°$ then:

```
dtheta = 69.33deg, thetaoffset = 0deg
CY = (1 - 2^-15)/(2*cos((69.33/2)/180*pi))= 0.60789036201452440
CX = (1 - 2^-15)/(2*sin((69.33/2)/180*pi))= 0.87905201358520957
NY = 0 (abs(CY) < 1)
NX = 0 (abs(CX) < 1)
KY = 0.60789/2^0 = 0.60789036201452440 = (Frac16) 19919
KX = 0.87905/2^0 = 0.87905201358520957 = (Frac16) 28805
THETAADJ = 0.19259643554687500 = (Frac16)  6311
```

The function requires that both signals have the same amplitude equal 1.0, if not an additional error is contributed, see **EQ 1-5**.

The algorithm may become numerically instable under some conditions. For example, if $\Delta\theta$ approaches 0, then the intermediate variable a cannot be computed due to discontinuity at 0.

Basing on theory of partial derivatives, one can derive more precise formula for expressing the error of the presented algorithm:

$$\varepsilon_{\theta_C} = \left\lceil \frac{0{,}318}{\sin(\Delta\theta_{mod}/2)} \cdot \varepsilon_{yx} + 0{,}159 \cdot \coth\Delta\theta_{mod} \cdot \varepsilon_A + \left(1 + \frac{0{,}318}{\sin\Delta\theta_{mod}}\right) \cdot \varepsilon_{\Delta\theta_C/2} + 1{,}23 \cdot LSB \right\rceil, \text{if } 0 < \Delta\theta_m < \pi/2$$

$$\varepsilon_{\theta_C} = \left\lceil \frac{0{,}318}{\cos(\Delta\theta_{mod}/2)} \cdot \varepsilon_{yx} + 0{,}159 \cdot \coth\Delta\theta_{mod} \cdot \varepsilon_A + \left(1 + \frac{0{,}318}{\sin\Delta\theta_{mod}}\right) \cdot \varepsilon_{\Delta\theta_C/2} + 1{,}23 \cdot LSB \right\rceil, \text{if } \pi/2 \le \Delta\theta_m < \pi \quad \textbf{(EQ 1-5)}$$

where:

LSB	Least Significant Bit, for 16-bits fractional numbers equal 2^{-15}
$\lceil \ldots \rceil$	ceiling function (the least integer larger or equal to)
ε_{θ_C}	error of computed angle expressed in LSB
ε_A	difference in signals amplitudes, absolute value, in LSB
ε_{yx}	error contributed by the y, x signals, $\varepsilon_y = \varepsilon_x = \varepsilon_{yx}$ in LSB
$\varepsilon_{\Delta\theta_C/2}$	measurement error of phase difference between signals divided by 2 plus error due to finite binary resolution
$\Delta\theta_{mod}$	phase difference $\Delta\theta$ modulo π

The signal error ε_{yx} includes an error due to limited resolution (for example of an ADC converter) and signal distortion by higher order harmonics.

For reference the derivation of error is provided at the end of this section.

It should be noticed that the error reaches its minimum at $\Delta\theta$ equal $\pi/2$ or $\pi/2$ and in this case the function becomes analogous to trivial arcus tangent function. On the other hand the error is approaching infinity with $\Delta\theta$ approaching 0 or π. Therefore the function is able to provide the most accurate results if $\Delta\theta$ does not differ much from $\pi/2$ or $-\pi/2$.

NOTE: Due to cyclic nature of angle ($\pi = -\pi$) even a small error may cause to shift the angle by $2 \cdot \pi$. For example if the correct angle is π (or 7FFF hex), due to contribution of various errors listed above, the computation may result in value of $-\pi$ (or 8000 hex). It should be noticed that due to random nature of errors it is not possible to detect such errors by the functions itself. It is advised to consider weather such behaviour may cause any computational problems in a final application.

Derivation of Numerical Error

The error of the computed angle consists of:

- error of values of provided signals due to finite resolution of ADC converter: ε_y and ε_x
- error contributed by higher order harmonics appearing in signal values: ε_y and ε_x (included in the variable)
- computational error of multiplication due to finite length of micro-controller registers : ε_b and ε_a
- error of representation of angle difference $\Delta\theta$ in finite precision arithmetic: $\varepsilon_{\Delta\theta_C/2}$
- error due to measurement of angle difference $\Delta\theta$: $\varepsilon_{\Delta\theta_C/2}$ (included in the variable)
- error due to differences in signal amplitudes

Using elementary mathematical functions and operators the computed angle can be expressed as:

$$\theta_C = \frac{1}{\pi} \cdot \operatorname{atan}\left(\tan\left(\frac{\Delta\theta_C}{2}\right) \cdot \frac{S \cdot A_y \cdot \sin(\Delta\theta_A) + S \cdot A_x \cdot \sin(\theta_A + \Delta\theta_A) + \varepsilon_y + \varepsilon_x + \varepsilon_b \cdot 2 \cdot \cos\frac{\Delta\theta_C}{2}}{S \cdot A_x \cdot \sin(\theta_A) - S \cdot A_x \cdot \sin(\theta_A + \Delta\theta_A) + \varepsilon_y - \varepsilon_x + \varepsilon_a \cdot 2 \cdot \sin\frac{\Delta\theta_C}{2}} \right) - \varepsilon_{\Delta\theta_C/2} \quad \textbf{(EQ 1-6)}$$

where:

A_y	amplitude of signal y, nearly 1.0
A_x	amplitude of signal x, nearly 1.0
θ_C	computed angle
θ_A	the actual angle
$\Delta\theta_A$	the actual phase difference between signals
ε_y	error contributed by the y signal, includes error contributed by ADC converter and higher order harmonics
ε_x	error contributed by the x signal, includes error contributed by ADC converter and higher order harmonics
$\varepsilon_{\Delta\theta_C/2}$	error contributed by the representation of $\Delta\theta_C/2$ in fractional format (numerical resolution) plus measurement error of $\Delta\theta_A$ divided by 2

ε_b error contributed by the multiplication to compute the intermediate variable b, see **EQ 1-3**

ε_a error contributed by the multiplication to compute the intermediate variable a, see **EQ 1-3**

The error of the computed angle is equal:

$$\delta\theta_C = \max_{\theta_A}\left(\left|\sum_u \frac{\partial\theta_A}{\partial u}\cdot\delta u\right|\right)\Bigg|_{\substack{\Delta\theta_C/2\\ \varepsilon_y=0\\ \varepsilon_x=0\\ \varepsilon_b=0\\ \varepsilon_a=0\\ A_y=1\\ A_x=1}} \qquad \textbf{(EQ 1-7.)}$$

$$u = \Delta\theta_C/2, \varepsilon_y, \varepsilon_x, \varepsilon_b, \varepsilon_a, A_y, A_x$$

where:

u independent variable

$\max_{\theta_A}$ maximum over all range of θ_A

δ partial difference operator

$\frac{\partial}{\partial}$ partial derivative operator

Furthermore one can assume:

$$\begin{gathered}
\varepsilon_y = \delta\varepsilon_y \geq 0\\
\varepsilon_x = \delta\varepsilon_x \geq 0\\
\varepsilon_b = \delta\varepsilon_b \geq 0\\
\varepsilon_a = \delta\varepsilon_a \geq 0\\
\varepsilon_{\Delta\theta_C/2} = \delta(\Delta\theta_C/2) \geq 0\\
\varepsilon_A = |A_y - A_x| \geq 0\\
\Delta\theta_C \cong \Delta\theta_A = \Delta\theta\\
S \cong 1\\
\varepsilon_{\theta_C} = \delta\theta_C\\
A_y \cong A_x = A_{yx} \cong 1,\ A_{yx} < 1
\end{gathered} \qquad \textbf{(EQ 1-8.)}$$

where:

ε_A error contributed by difference in signals amplitude

ε_{θ_C} the resulting error of computed angle

It should be noticed that the error contributed by signals (ε_y and ε_x), errors due to finite precision arithmetic (ε_b, ε_a and $\varepsilon_{\Delta\theta_C/2}$) are random in nature. The error due to unequal signal amplitudes ε_A is in opposite a systematic error[1].

Then the maximum error computed from partial differences is equal:

$$\varepsilon_{\theta_C} = \frac{1}{\pi} \cdot \max_{\theta_A}\left(\left|\frac{\sin(\theta_A + \Delta\theta)}{A_{yx} \cdot \sin\Delta\theta}\right| \cdot \varepsilon_x + \left|\frac{\sin\theta_A}{A_{yx} \cdot \sin\Delta\theta}\right| \cdot \varepsilon_y + \left|\frac{\cos\left(\theta_A + \frac{\Delta\theta}{2}\right)}{A_{yx}}\right| \cdot \varepsilon_b + \left|\frac{\sin\left(\theta_A + \frac{\Delta\theta}{2}\right)}{A_{yx}}\right| \cdot \varepsilon_a \right.$$

$$\left. + \left|\frac{\coth\Delta\theta}{2 \cdot A_{yx}}\right| \cdot \varepsilon_A + \left|\frac{\sin(2 \cdot \theta_A + \Delta\theta)}{\sin\Delta\theta}\right| \cdot \varepsilon_{\Delta\theta_C/2} \right) + \varepsilon_{\Delta\theta_C/2} \quad \textbf{(EQ 1-9)}$$

Apart of the error sources mentioned above, additional error is contributed by the **MCLIB_AtanYX** function. In case of the **MCLIB_AtanYX** function from the MCLIB library the computation error is equal one LSB.

The error of the y and x signals, the ε_y and ε_x appears due to finite resolution of A/D converter and higher order harmonics and is equal some amount of LSB.

The computation error due to multiplication in numerator and denominator in **EQ 1-3**, ε_b and ε_a, is equal half of LSB (each one).

The error due to finite, binary representation of $\Delta\theta_C/2$ is equal half of LSB. Additionally measurement error of phase angle equal some amount of LSB need to be taken into account.

1. The division into random and systematic errors is conventional, indeed the $\varepsilon_{\Delta\theta_C/2}$ error can be also treated as a systematic error as well as ε_A

The error due to difference in amplitude for each signal, the ε_A, is equal some amount of LSB. It should be noticed that the error appears only if the amplitudes of both signals differ.

After appropriate calculation and substitutions the upper boundary for the error becomes **EQ 1-5**.

1.16.4 Returns

The function returns an angle of two sine wave shifted in phase.

1.16.5 Range Issues

The input data value is in the range of [-1,1). The output data value is in the range [-1, 1), which corresponds to the angle in the range of [-pi,pi).

The computation error depends strongly on phase difference between the sine waves, which may cause numerical issues.

1.16.6 Special Issues

The function calls internally **MCLIB_AtanYX** function.

The function is not sensitive to saturation mode bit setting. However it should be noticed that the function actually uses the saturation mode bit. The saturation mode bit is stored and restored during a function call.

The function uses **REP** instruction to perform shifting with scaling coefficients N_y, N_x used as repetition amounts. Therefore large values of scaling coefficient may increase interrupt latency time.

1.16.7 Implementation

The **MCLIB_AtanYXShifted** is implemented as a function call with initialization.

Code Example 1-16. MCLIB_AtanYXShifted

```
/* include function and data prototypes required by the module */
#include "mclib.h"

/* Define appropriate data                                      */
/* dtheta = 5deg, thetaoffset = 0deg                            */
#define NY                          0
#define NX                          4
#define KY                          FRAC16(0.50046106925577549)
#define KX                          FRAC16(0.71640268727196699)
#define THETAADJ                    FRAC16(0.01388549804687500)

void main (void)
{
   Frac16 x, y, z;

   y = -7477;          /* ~= sin(-166.811)                      */
   x = -10229;         /* ~= sin(-166.811 + 5)                  */

   /* Initialize                                                */
   MCLIB_InitAtanYXShifted(KY, KX, NY, NX, THETAADJ);

   /* Compute angle                                             */
   z = MCLIB_AtanYXShifted(y, x);

   /* The result: -30366 -> -166.805 deg                        */
}
```

1.16.8 Performance

Table 1-17. Performance of the MCLIB_AtanYXShifted function.

Code Size	[1]47 words	
Data Size	2 words	
Execution Clocks	Min	208 cycles
	Max	216 cycles

1. NOTE: code size does not include the code of the **MCLIB_AtanYX** function.

Section 2. TRANSFORMATIONS

2.1 API Summary

Table 2-1. API Function Summary

Name	Arguments	Return	Description
MCLIB_ClarkTrfm	MC_2PhSyst *pAlphaBeta, MC_3PhSyst *p_abc	void	Function function calculates the Clarke Transformation algorithm according to equations listed in **Section: 2.2 MCLIB_ClarkTrfm**
MCLIB_ClarkTrfmInv	MC_3PhSyst *p_abc, MC_2PhSyst *pAlphaBeta	void	Function calculates the Inverse Clarke Transformation algorithm according to equations listed in **Section: 2.3 MCLIB_ClarkTrfmInv**
MCLIB_ParkTrfm	MC_DqSyst *pDQ MC_2PhSyst *pAlphaBeta, MC_Angle *psinCos	void	Function calculates the Park Transformation algorithm according to equations listed in **Section: 2.4 MCLIB_ParkTrfm**
MCLIB_ParkTrfmInv	MC_Angle *pSinCos, MC_DqSyst *pDQ, MC_2PhSyst *pAlphaBeta	void	Function calculates the Inverse Park Transformation algorithm according to equations listed in **Section: 2.5 MCLIB_ParkTrfmInv**

2.2 MCLIB_ClarkTrfm

2.2.1 Synopsis

#include ”mclib.h”
*void **MCLIB_ClarkTrfm** (MC_2PhSyst *pAlphaBeta, MC_3PhSyst *p_abc);*

2.2.2 Arguments

*pAlphaBeta	in	Pointer to structure containing data of two-phase rotating orthogonal system
*p_abc	out	Pointer to structure containing data of three-phase rotating system

2.2.3 Description

This function calculates the Clarke transformation, which is used for transforming values (flux, voltage, current) from a three-phase rotating coordinate system to an alpha-beta rotating orthogonal coordinate system according to these functions:

$$alpha = a \qquad \textbf{(EQ 2-1.)}$$

$$beta = \frac{1}{\sqrt{3}}a + \frac{2}{\sqrt{3}}b \qquad \textbf{(EQ 2-2.)}$$

2.2.4 Special Issues

The function calculates correct results only if the saturation flag is set. The flag don’t have to be set if input data are within unite circle.

2.2.5 Implementation

The **MCLIB_ClarkTrfm** function is implemented as a function call.

Code Example 2-1. MCLIB_ClarkTrfm

```
/* include function and data prototypes required by the module  */
#include "mclib.h"

/* main fucntion call                                           */
void main (void)
{
  /* define data                                                */
  MC_2PhSyst  alphaBeta;
  MC_3PhSyst  abc;

  /* load input variable by data                                */
  alphaBeta.alpha = FRAC16(1.0);     /* corresponds to 1.0      */
  alphaBeta.beta  = 0;               /* corresponds to 0.0      */
  /* compute Clark Inverse Transformation                       */
  MCLIB_ClarkTrfmInv (&abc, &alphaBeta);
  /* compute Clark Transformation                               */
  MCLIB_ClarkTrfm (&alphaBeta, &abc);
}
```

2.2.6 See Also

MCLIB_ClarkTrfmInv

2.2.7 Performance

Table 2-2. Performance of the MCLIB_ClarkTrfm function.

Code Size	14 words	
Data Size	0 words	
Execution Clocks	Min	61 cycles
	Max	61 cycles

2.3 MCLIB_ClarkTrfmInv

2.3.1 Synopsis

#include ”mclib.h”
void ***MCLIB_ClarkTrfmInv*** *(MC_3PhSyst *p_abc, MC_2PhSyst *pAlphaBeta);*

2.3.2 Arguments

*p_abc	in	Pointer to structure containing data of three-phase rotating system
*pAlphaBeta	out	Pointer to structure containing data of two-phase rotating orthogonal system

2.3.3 Description

The **MCLIB_ClarkTrfmInv** function calculates Inverse Clarke Transformation, which transforms values (flux, voltage, current) from an alpha-beta rotating orthogonal coordination system to a three-phase rotating coordination system, according to these equations:

$$a = alpha \qquad \textbf{(EQ 2-3.)}$$

$$b = -0.5 \times alpha + \frac{\sqrt{3}}{2} \times beta \qquad \textbf{(EQ 2-4.)}$$

$$c = -(a + b) \qquad \textbf{(EQ 2-5.)}$$

2.3.4 Special Issues

The function calculates correct results only if the saturation flag is set. The flag don’t have to be set if input data are within unite circle.

2.3.5 Implementation

The **MCLIB_ClarkTrfmInv** function is implemented as a function call.

Code Example 2-2. MCLIB_ClarkTrfmInv

```
/* include function and data prototypes required by the module  */
#include "mclib.h"

/* main fucntion call                                           */
void main (void)
{
  /* define data                                                */
  MC_2PhSyst  alphaBeta;
  MC_3PhSyst  abc;

  /* load input variable by data                                */
  alphaBeta.alpha = FRAC16(1.0);     /* corresponds to 1.0      */
  alphaBeta.beta  = 0;               /* corresponds to 0.0      */
  /* compute Clark Inverse Transformation                       */
  MCLIB_ClarkTrfmInv (&abc, &alphaBeta);
  /* compute Clark Transformation                               */
  MCLIB_ClarkTrfm (&alphaBeta, &abc);
}
```

2.3.6 See Also

MCLIB_ClarkTrfm

2.3.7 Performance

Table 2-3. Performance of the MCLIB_ClarkTrfmInv function.

Code Size	16 words	
Data Size	0 words	
Execution Clocks	Min	73 cycles
	Max	73 cycles

2.4 MCLIB_ParkTrfm

2.4.1 Synopsis

#include ”mclib.h”
*void **MCLIB_ParkTrfm** (MC_DqSyst *pDQ, MC_2PhSyst *pAlphaBeta, MC_Angle *pSinCos);*

2.4.2 Arguments

*pSinCos	in	Pointer to a structure where the values of sine and cosine are stored
*pAlphaBeta	in	Pointer to a structure containing data of two-phase rotating orthogonal system
*pDQ	out	Pointer to a structure containing data of DQ coordinate two-phase stationary orthogonal system

2.4.3 Description

This function calculates Park Transformation, which transforms values (flux, voltage, current) from an alpha-beta rotating orthogonal coordinate system to a d-q stationary orthogonal coordinate system according to these equations:

$$d = alpha \times \cos(theta) + beta \times \sin(theta) \qquad \textbf{(EQ 2-6.)}$$

$$q = beta \times \cos(theta) - alpha \times \sin(theta) \qquad \textbf{(EQ 2-7.)}$$

2.4.4 Special Issues

The function calculates correct results only if the saturation flag is set. The flag don’t have to be set if input data are within unite circle.

2.4.5 Implementation

The **MCLIB_ParkTrfm** function is implemented as a function call.

Code Example 2-3. MCLIB_ParkTrfm

```
/* include function and data prototypes required by the module  */
#include "mclib.h"

/* main function call                                           */
void main (void)
{
  /* define data                                                */
  MC_DqSyst     dqSystem;
  MC_2PhSyst    twoPhSystem;
  MC_Angle      angle;

  /* load input variable by data                                */
  twoPhSystem.alpha = FRAC16(1.0);    /* corresponds to 1.0     */
  twoPhSystem.beta  = 0;              /* corresponds to 0.0     */
  angle.sin = FRAC16(0.0);           /* corresponds to 0.0     */
  angle.cos  = FRAC16(1.0);           /* corresponds to 1.0     */

  /* computes Park Transformation                               */
  MCLIB_ParkTrfm    (&dqSystem, &twoPhSystem, &angle);
  /* computes Inverse Park Transformation                       */
  MCLIB_ParkTrfmInv (&twoPhSystem, &dqSystem, &angle);
}
```

2.4.6 See Also

MCLIB_ParkTrfmInv

2.4.7 Performance

Table 2-4. Performance of the MCLIB_ParkTrfm function.

Code Size	17 words	
Data Size	0 words	
Execution Clocks	Min	91 cycles
	Max	91 cycles

2.5 MCLIB_ParkTrfmInv

2.5.1 Synopsis

#include ”mclib.h”
*void **MCLIB_ParkTrfmInv** (MC_2PhSyst *pAlphaBeta, MC_DqSyst *pDQ, MC_Angle *pSinCos);*

2.5.2 Arguments

*pSinCos	in	Pointer to a structure where the values of sine and cosine are stored
*pDQ	in	Pointer to a structure containing data of dq coordinate two-phase stationary orthogonal system
*pAlphaBeta	out	Pointer to a structure containing data of two-phase rotating orthogonal system

2.5.3 Description

This function calculates Inverse Park Transformation, which transforms values (flux, voltage, current) from a d-q stationary orthogonal coordinate system to an alpha-beta rotating orthogonal coordinate system according to these equations:

$$alpha = d \times \cos(theta) - q \times \sin(theta)$$ **(EQ 2-8.)**

$$beta = d \times \sin(theta) + q \times \cos(theta)$$ **(EQ 2-9.)**

2.5.4 Special Issues

The function calculates correct results only if the saturation flag is set. The flag don’t have to be set if input data are within unite circle.

2.5.5 Implementation

The **MCLIB_ParkTrfmInv** function is implemented as a function call.

Code Example 2-4. MCLIB_ParkTrfmInv

```
/* include function and data prototypes required by the module  */
#include "mclib.h"

/* main function call                                           */
void main (void)
{
  /* define data                                                */
  MC_DqSyst     dqSystem;
  MC_2PhSyst    twoPhSystem;
  MC_Angle      angle;

  /* load input variable by data                                */
  twoPhSystem.alpha = FRAC16(1.0);    /* corresponds to 1.0     */
  twoPhSystem.beta  = 0;              /* corresponds to 0.0     */
  angle.sin = FRAC16(0.0);           /* corresponds to 0.0     */
  angle.cos  = FRAC16(1.0);           /* corresponds to 1.0     */

  /* computes Park Transformation                               */
  MCLIB_ParkTrfm    (&dqSystem, &twoPhSystem, &angle);
  /* computes Inverse Park Transformation                       */
  MCLIB_ParkTrfmInv (&twoPhSystem, &dqSystem, &angle);
}
```

2.5.6 See Also

MCLIB_ParkTrfm

2.5.7 Performance

Table 2-5. Performance of the MCLIB_ParkTrfmInv function.

Code Size	17 words	
Data Size	0 words	
Execution Clocks	Min	92 cycles
	Max	92 cycles

Section 3. CONTROLLERS

3.1 API Summary

Table 3-1. API Function Summary

Name	Arguments	Return	Description
MCLIB_ControllerPI	Frac16 desiredValue, Frac16 measuredValue, MC_PiParams *pParams	Frac16	Function calculates the Proportional-Integral (PI) algorithm according to equations listed in Section: 3.2 MCLIB_ControllerPI
MCLIB_ControllerPI2	Frac16 desiredValue, Frac16 measuredValue, MC_PiParams *pParams	Frac16	Function calculates the Proportional-Integral (PI) algorithm according to equations listed in Section: 3.3 MCLIB_ControllerPI2

3.2 MCLIB_ControllerPI

3.2.1 Synopsis

#include ”mclib.h”

Frac16 ***MCLIB_ControllerPI*** *(Frac16 desiredValue, Frac16 measuredValue, MC_PiParams *pParams);*

3.2.2 Arguments

desiredValue	in	desired value of the process
measuredValue	in	measured value of the process
*pParams	in/out	parameters of PI controller; the **MC_PiParams** data type is described in header file mclib_types.h.

3.2.3 Description

The **MCLIB_ControllerPI** function calculates the Proportional-Integral (PI) algorithm according to equations below:

The PI algorithm in continuous time domain:

$$u(t) = K\left[e(t) + \frac{1}{T_I}\int_0^{\tau} e(\tau)d\tau\right] \quad \textbf{(EQ 3-1.)}$$

$$e(t) = w(t) - y(t) \quad \textbf{(EQ 3-2.)}$$

The transfer function is shown below:

$$F(p) = K\left[1 + \frac{1}{T_I}\cdot\frac{1}{p}\right] = \frac{u(p)}{e(p)} \quad \textbf{(EQ 3-3.)}$$

The PI algorithm in discrete time domain in fractional arithmetic:

$$u_f(k) = K_{sc}\cdot e_f(k) + u_{If}(k-1) + K_{Isc}\cdot\frac{T}{T_I}\cdot e_f(k) \quad \textbf{(EQ 3-4.)}$$

$$e(k) = w(k) - y(k) \quad \textbf{(EQ 3-5.)}$$

where

$$u_f(k) = u(k)/u_{max} \quad \textbf{(EQ 3-6.)}$$

$$w_f(k) = w(k)/w_{max} \quad \textbf{(EQ 3-7.)}$$

$$y_f(k) = y(k)/y_{max} \quad \textbf{(EQ 3-8.)}$$

$$e_f(k) = e(k)/e_{max} \quad \textbf{(EQ 3-9.)}$$

$$K_{sc} = K \cdot \frac{e_{max}}{u_{max}} \quad \textbf{(EQ 3-10.)}$$

$$K_{Isc} = K \cdot \frac{T}{T_I} \cdot \frac{e_{max}}{u_{max}} \quad \textbf{(EQ 3-11.)}$$

The integral is approximated by Backward Euler method, also known as Backward Rectangular or right - hand approximation. For this method, $1/p$ is approximated by $u_I(k) = u_I(k-1) + T \cdot e(k)$

$$K_{sc} = ProportionalGain \cdot 2^{-ProportionalGainScale} \quad \textbf{(EQ 3-12.)}$$

$$K_{Isc} = IntegralGain \cdot 2^{-IntegralGainScale} \quad \textbf{(EQ 3-13.)}$$

$$0{,}5 < ProportionalGain < 1 \quad \textbf{(EQ 3-14.)}$$

$$0{,}5 < IntegralGain < 1 \quad \textbf{(EQ 3-15.)}$$

$$-14 < ProportionalGainScale < 14 \quad \textbf{(EQ 3-16.)}$$

$$-14 < IntegralGainScale < 14 \quad \textbf{(EQ 3-17.)}$$

The calculation of the scaling coefficients; ProportionalGainScale and IntegralGainScale can be performed using the following equations:

$$\frac{\log(0.5) - \log(ProportionalGain)}{\log 2} < ProportionalGainScale \quad \textbf{(EQ 3-18.)}$$

$$\frac{\log 1 - \log(ProportionalGain)}{\log 2} > ProportionalGainScale \quad \textbf{(EQ 3-19.)}$$

$$\frac{\log(0.5) - \log(IntegralGain)}{\log 2} < IntegralGainScale \quad \textbf{(EQ 3-20.)}$$

$$\frac{\log 1 - \log(IntegralGain)}{\log 2} > IntegralGainScale \quad \textbf{(EQ 3-21.)}$$

For example if $K = 0.05$ then scaling factor and scaling gain are given based on equations **EQ 3-18** to **EQ 3-21**.

$$\frac{\log(0.5) - \log(0.05)}{\log 2} < Scale < \frac{\log 1 - \log(0.05)}{\log 2}$$

$$3.3219 < Scale < 4.3219$$

From above equations the $Scale = 4$ and

$$Gain = K \cdot 2^{Scale} = 0.05 \cdot 2^4 = 0.8$$

Controller implements following limitations:

– The integral portion limitation (anti-wind-up effect)

$$NegativePILimit \le u_{If}(k) \le PositivePILimit \quad \textbf{(EQ 3-22.)}$$

– The controller output limitation

$$NegativePILimit \le u_{f}(k) \le PositivePILimit \quad \textbf{(EQ 3-23.)}$$

where:

$e_f(k)$	=	Input error in step k - discrete time domain fractional
$w_f(k)$	=	Desired value in step k - discrete time domain fractional
$y_f(k)$	=	Measured value in step k - discrete time domain fractional
$u_f(k)$	=	Controller output in step k - discrete time fractional
$u_{If}(k-1)$	=	Integral output portion in step k-1 - discrete time fractional
T_I	=	Integral time constant
T	=	Sampling time
t	=	Time
K	=	Controller gain
p	=	Laplace variable

3.2.4 Returns

The function returns the value representing the controller output in step [k]. The output value is limited by using PositivePILimit and NegativePILimit values of MC_PiParams data structure - see **Table 1-2**.

3.2.5 Range Issues

$$0.5 < ProportionalGain < 1$$

$$0.5 < IntegralGain < 1$$

$$-14 < IntegralProportionalGainScale < 14$$

$$) - 14 < ProportionalGainScale < 14$$

$$2^{-15} < K < 2^{15}$$

$$2^{-15} < K \cdot \frac{T}{T_I} < 2^{15}$$

3.2.6 Special Issues

The function calculates correct results regardless of the saturation mode.

3.2.7 Implementation

The **MCLIB_ControllerPI** is implemented as a function call.

Code Example 3-1. MCLIB_ControllerPI

```
/* include function and data prototypes required by the module  */
#include "mclib.h"

/* main function call                                           */
void main (void)
{
  /* define data                                                */
  MC_PiParams   PIparams;
  Frac16        desiredValue, measuredValue, PIoutput;

  /* load input variable by data                                */
  PIparams.propGain    = FRAC16(0.5);       /* 0.5*2-1=0.25     */
  PIparams.propGainSc  = 1;
  PIparams.integGainSc = 0;
  PIparams.integGain   = FRAC16(0.1);
  PIparams.posPiLimit  = FRAC16(1.0);
  PIparams.negPiLimit  = FRAC16(-1.0);
  PIparams.integPartK_1= 0;
  desiredValue  = FRAC16(1.0);
  measuredValue = FRAC16(0.0);;

  /* calculate output of the PI controller                      */
  PIoutput = MCLIB_ControllerPI (desiredValue, measuredValue, &PIparams);
}
```

3.2.8 Performance

Table 3-2. Performance of the MCLIB_ControllerPI function.

Code Size	52 words	
Data Size	0 words	
Execution Clocks	Min	82 cycles
	Max	82 cycles

3.3 MCLIB_ControllerPI2

3.3.1 Synopsis

#include "mclib.h"
Frac16 ***MCLIB_ControllerPI2*** *(Frac16 desiredValue, Frac16 measuredValue, MC_PiParams *pParams);*

3.3.2 Arguments

desiredValue	in	desired value of the process
measuredValue	in	measured value of the process (feedback)
*pParams	in/out	parameters of PI controller; the **MC_PiParams** data type is described in header file mclib_types.h.

3.3.3 Description

The **MCLIB_ControllerPI2** function calculates the Proportional-Integral (PI) algorithm according to equations below:

The PI algorithm in continuous time domain:

$$u(t) = K\left[e(t) + \frac{1}{T_I}\int_0^{\tau} e(\tau)d\tau\right] \quad \textbf{(EQ 3-24.)}$$

$$e(t) = w(t) - y(t) \quad \textbf{(EQ 3-25.)}$$

The transfer function is shown below:

$$F(p) = K\left[1 + \frac{1}{T_I}\cdot\frac{1}{p}\right] = \frac{u(p)}{e(p)} \quad \textbf{(EQ 3-26.)}$$

The PI algorithm in discrete time domain in fractional arithmetic:

$$u_f(k) = K_{sc}\cdot e_f(k) + u_{If}(k-1) + K_{Isc}\cdot\frac{T}{T_I}\cdot e_f(k) \quad \textbf{(EQ 3-27.)}$$

$$e(k) = w(k) - y(k) \quad \textbf{(EQ 3-28.)}$$

where

$$u_f(k) = u(k)/u_{max} \quad \textbf{(EQ 3-29.)}$$

$$w_f(k) = w(k)/w_{max} \quad \textbf{(EQ 3-30.)}$$

$$y_f(k) = y(k)/y_{max} \quad \textbf{(EQ 3-31.)}$$

$$e_f(k) = e(k)/e_{max} \quad \textbf{(EQ 3-32.)}$$

$$K_{sc} = K \cdot \frac{e_{max}}{u_{max}} \quad \textbf{(EQ 3-33.)}$$

$$K_{Isc} = K \cdot \frac{T}{T_I} \cdot \frac{e_{max}}{u_{max}} \quad \textbf{(EQ 3-34.)}$$

The integral is approximated by Backward Euler method, also known as Backward Rectangular or right - hand approximation. For this method, $1/p$ is approximated by $u_I(k) = u_I(k-1) + T \cdot e(k)$

$$K_{sc} = ProportionalGain \cdot 2^{-ProportionalGainScale} \quad \textbf{(EQ 3-35.)}$$

$$K_{Isc} = IntegralGain \cdot 2^{-IntegralGainScale} \quad \textbf{(EQ 3-36.)}$$

$$0{,}5 < ProportionalGain < 1 \quad \textbf{(EQ 3-37.)}$$

$$0{,}5 < IntegralGain < 1 \quad \textbf{(EQ 3-38.)}$$

$$-14 < ProportionalGainScale < 14 \quad \textbf{(EQ 3-39.)}$$

$$-14 < IntegralGainScale < 14 \quad \textbf{(EQ 3-40.)}$$

The calculation of the scaling coefficients; ProportionalGainScale and IntegralGainScale can be performed using the following equations:

$$\frac{\log(0.5) - \log(ProportionalGain)}{\log 2} < ProportionalGainScale \quad \textbf{(EQ 3-41.)}$$

$$\frac{\log 1 - \log(ProportionalGain)}{\log 2} > ProportionalGainScale \quad \textbf{(EQ 3-42.)}$$

$$\frac{\log(0.5) - \log(IntegralGain)}{\log 2} < IntegralGainScale \quad \textbf{(EQ 3-43.)}$$

$$\frac{\log 1 - \log(IntegralGain)}{\log 2} > IntegralGainScale \quad \textbf{(EQ 3-44.)}$$

For example if $K = 0.05$ then scaling factor and scaling gain are given based on equations **EQ 3-41** to **EQ 3-44**.

$$\frac{\log(0.5) - \log(0.05)}{\log 2} < Scale < \frac{\log 1 - \log(0.05)}{\log 2}$$

$$3.3219 < Scale < 4.3219$$

From above equations the $Scale = 4$ and

$$Gain = K \cdot 2^{Scale} = 0.05 \cdot 2^4 = 0.8$$

Controller implements following limitations:

– The integral portion limitation (anti-wind-up effect)

$$NegativePILimit \le u_{If}(k) \le PositivePILimit \quad \textbf{(EQ 3-45.)}$$

– The controller output limitation

$$NegativePILimit \le u_{f}(k) \le PositivePILimit \quad \textbf{(EQ 3-46.)}$$

where:

$e_f(k)$	=	Input error in step k - discrete time domain fractional
$w_f(k)$	=	Desired value in step k - discrete time domain fractional
$y_f(k)$	=	Measured value in step k - discrete time domain fractional
$u_f(k)$	=	Controller output in step k - discrete time domain in fractional
$u_{If}(k-1)$	=	Integral output portion in step k-1 - discrete time domain in fractional
T_I	=	Integral time constant
T	=	Sampling time
t	=	Time
K	=	Controller gain
p	=	Laplace variable

3.3.4 Returns

The function returns the value representing the controller output in step [k]. The output value is limited by using PositivePILimit and

NegativePILimit values set in MC_PiParams data structure - see **Table 1-2**.

The PI2 controller algorithms also returns the saturation flag. This flag named "saturationFlag" is the member of the structure of the PI controller parameters. If the PI2 controller output reaches the positive or negative limit the saturationFlag=1 otherwise saturationFlag=0.

3.3.5 Range Issues

$$0.5 < ProportionalGain < 1$$
$$0.5 < IntegralGain < 1$$
$$-14 < IntegralProportionalGainScale < 14$$
$$0 - 14 < ProportionalGainScale < 14$$
$$2^{-15} < K < 2^{15}$$
$$2^{-15} < K \cdot \frac{T}{T_I} < 2^{15}$$

3.3.6 Special Issues

The function calculates correct results regardless of the saturation mode.

3.3.7 Implementation

The **MCLIB_ControllerPI2** is implemented as a function call.

*Code Example 3-2. **MCLIB_ControllerPI2***

```
/* include function and data prototypes required by the module  */
#include "mclib.h"

/* main function call                                           */
void main (void)
{
  /* define data                                                */
  MC_PiParams   PIparams;
  Frac16        desiredValue, measuredValue, PIoutput;

  /* load input variable by data                                */
  PIparams.propGain    = FRAC16(0.5);       /* 0.5*2-1=0.25     */
  PIparams.propGainSc  = 1;
  PIparams.integGainSc = 0;
  PIparams.integGain   = FRAC16(0.1);
  PIparams.posPiLimit  = FRAC16(1.0);
  PIparams.negPiLimit  = FRAC16(-1.0);
  PIparams.integPartK_1= 0;
  desiredValue  = FRAC16(1.0);
  measuredValue = FRAC16(0.0);;

  /* calculate output of the PI controller                      */
  PIoutput = MCLIB_ControllerPI2 (desiredValue, measuredValue, &PIparams);
}
```

3.3.8 Performance

Table 3-3. Performance of the MCLIB_ControllerPI2 function.

Code Size	124 words	
Data Size	0 words	
Execution Clocks	Min	105 cycles
	Max	127 cycles

Section 4. RESOLVER PROCESSING

4.1 API Summary

Table 4-1. API Function Summary

Name	Arguments	Return	Description
MCLIB_InitTrackObsv	Frac16 k1_d, Frac16 k2_d, int k1_sc, int k2_sc	void	This function initializes internal variables of the *Angle Tracking Observer*. Refer to section **Section: 4.2 MCLIB_InitTrackObsv**
MCLIB_CalcTrackObsv	Frac16 sinA, Frac16 cosA	none	This function calculates *Angle Tracking Observer*. The observer is based on equations listed in **Section: 4.3 MCLIB_CalcTrackObsv**
MCLIB_GetResPosition	none	Frac16	This function returns an estimate of the rotor angle. Refer to **Section: 4.4 MCLIB_GetResPosition**
MCLIB_GetResSpeed	none	Frac32	This function returns the estimate of the actual rotor speed read from an internal variable of the *Angle Tracking Observer.* Refer to **Section: 4.5 MCLIB_GetResSpeed**
MCLIB_GetResRevolutions	none	int	This function returns the estimate of the rotor revolutions. Refer to **Section: 4.6 MCLIB_GetResRevolutions**
MCLIB_SetResPosition	Frac16 newPosition	none	This function is used to set a new angle to the current angle. Refer to **Section: 4.7 MCLIB_SetResPosition**
MCLIB_SetResRevolutions	int newRevolutions	none	This function sets the number of revolutions that is stored in the internal variable of the *Angle Tracking Observer* algorithm. Refer to **Section: 4.8 MCLIB_SetResRevolutions**

4.2 MCLIB_InitTrackObsv

4.2.1 Synopsis

#include ”mclib.h”
*void **MCLIB_InitTrackObsv** (Frac16 k1_d, Frac16 k2_d, int k1_sc, int k2_sc);*

4.2.2 Arguments

k1_d	in	scaled K1 coefficient of the *Angle Tracking Observer*
k2_d	in	scaled K2 coefficient of the *Angle Tracking Observer*
k1_sc	in	scaling factor of scaled K1 coefficient of the *Angle Tracking Observer*
k2_sc	in	scaling factor of scaled K2 coefficient of the *Angle Tracking Observer*

4.2.3 Description

This function initializes internal variables of the *Angle Tracking Observer*. It should be called in the initialization part of the user’s application software.

4.2.4 Implementation

The **MCLIB_InitTrackObsv** function is implemented as a function call.

Code Example 4-1. MCLIB_InitTrackObsv

```
/* include function and data prototypes required by the module  */
#include "mclib.h"

/* parameters of the Angle Tracking Observer                    */
/*************************************************************
 * F0 = 500/2/pi = 80 Hz, DAMPING = 0.84, Fs = 1/Ts = 8000Hz    *
 *************************************************************/
#define K1_D         FRAC16(0.63661977236758)
#define K2_D         FRAC16(0.84000000000000)
#define K1_SCALE     9
#define K2_SCALE     5

/* main function call                                           */
void main (void)
{
   /* initialize Angle Tracking Observer algorithm               */
   MCLIB_InitTrackObsv (K1_D, K2_D, K1_SCALE, K2_SCALE);
}
```

4.2.5 Performance

Table 4-2. Performance of the MCLIB_InitTrackObsv function.

Code Size	37 words	
Data Size	13 words	
Execution Clocks	Min	56 cycles
	Max	56 cycles

4.3 MCLIB_CalcTrackObsv

4.3.1 Synopsis

#include ”mclib.h”
*void **MCLIB_CalcTrackObsv** (Frac16 sinA, Frac16 cosA);*

4.3.2 Arguments

sinA	in	sample measured on the sine winding of the resolver sensor
cosA	in	sample measured on the co-sine winding of the resolver sensor

4.3.3 Description

This function calculates the *Angle Tracking Observer* algorithm. It is recommended to call this function at every sampling period, e.g. in the ADC End of Scan interrupt service routine. It requires two input arguments: the sine sample sinA and co-sine sample cosA. The practical implementation of the *Angle Tracking Observer* algorithm is described below.

The *Angle Tracking Observer* compares values of the resolver output signals U_{sin}, U_{cos} with their corresponding estimations $\hat{U}_{sin}$, $\hat{U}_{cos}$. As in any common closed-loop systems, the intent is to minimize observer error. The observer error is given here by subtraction of the estimated resolver rotor angle $\hat{\Theta}$ from the actual rotor angle Θ (see **Figure 4-1**).

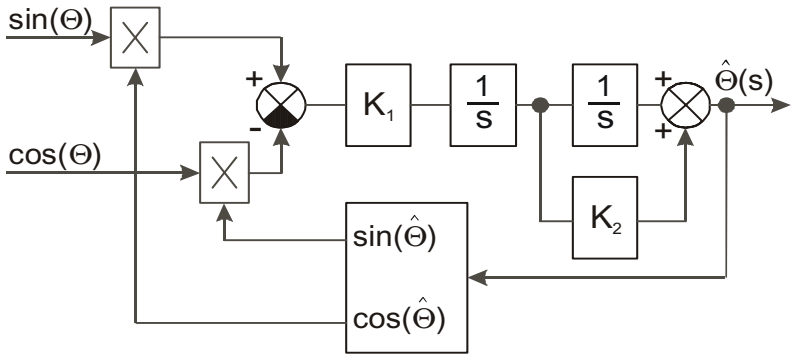

Figure 4-1. Block Scheme of the Angle

Note that mathematical expression of observer error is known as the formula of the difference of two angles:

$$\sin(\Theta - \hat{\Theta}) = \sin(\Theta)\cos(\hat{\Theta}) - \cos(\Theta)\sin(\hat{\Theta}) \qquad \textbf{(EQ 4-1.)}$$

The analog integrators in **Figure 4-1**, marked as 1/s, are replaced by an equivalent of the discrete-time integrator (see **Figure 4-2**), using the *Forward Euler* integration method.

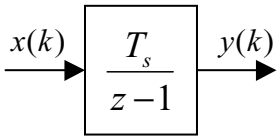

Figure 4-2. Discrete-Time

From the definition of this method, the analog integrator is approximated by a difference equation:

$$y(k+1) = y(k) + x(k) \cdot T_s \qquad \textbf{(EQ 4-2.)}$$

where $x(k)$ and $y(k)$ are input and output values in step k and T_s [s] is the sampling period. Index k represents the previous (old) value and index $k + 1$ the current (new) value. The transfer function corresponding to this difference equation is:

$$H(z) = \frac{T_s}{z-1} \qquad \textbf{(EQ 4-3.)}$$

The discrete-time block diagram of the *Angle Tracking Observer* is shown in **Figure 4-2**.

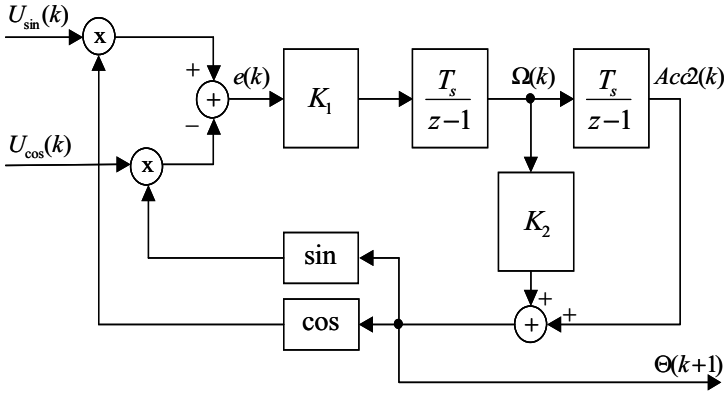

Figure 4-3. Block Scheme of Discrete-Time Tracking Observer

It should be noted that the loop with gain K_2 is predictive and together with *Acc2(k)* it provides for the estimation of the actual rotor angle in step *k+1*. The essential equations for implementation of the *Angle Tracking Observer*, according to block scheme in **Figure 4-3**, are as follows:

$$\Omega(k+1) = \Omega(k) + T_s \cdot e(k) \cdot K_1 \quad \textbf{(EQ 4-4.)}$$

$$Acc2(k+1) = Acc2(k) + T_s \cdot \Omega(k) \quad \textbf{(EQ 4-5.)}$$

$$\Theta(k+1) = K_2 \cdot \Omega(k+1) + Acc2(k+1) \quad \textbf{(EQ 4-6.)}$$

$$e(k+1) = U_{\sin}(k+1) \cdot \cos\Theta(k+1) - U_{\cos}(k+1) \cdot \sin\Theta(k+1) \quad \textbf{(EQ 4-7.)}$$

where: $K_1 = \omega_n^2$ and $K_2 = 2\zeta/\omega_n$ are coefficients of the *Angle Tracking Observer*,

$e(k)$ is observer error in step *k,*

ω_n and ζ are the natural freq. [rad s^{-1}] and damping factor [-],

T_S is the sampling period [s],

$\Omega(k+1)$ is the actual rotor speed [rad s^{-1}] in step $k+1$,

$Acc2(k+1)$ is the actual rotor angle [rad] without scaled addition of speed in step $k+1$,

$\Theta(k+1)$ is the actual rotor angle [rad] in step $k+1$,

$U_{\sin}(k+1)$ and $U_{\cos}(k+1)$ are sine and co-sine samples.

In equations **EQ 4-4** to **EQ 4-7**, there are coefficients and quantities that are greater than one (for example, the actual rotor speed $\Omega(k+1)$) or that are too small to be precisely represented by a 16 bit fractional value. Due to this fact a special transformation of equations **EQ 4-4** to **EQ 4-7** must be carried out in order to be successfully implemented using fractional arithmetic. This transformation is based on these steps.

Firstly, the actual rotor angle in the digital representation $\Theta_d(k+1)$[2] is scaled by π to fit into the range -1 to 1:

$$\Theta_d(k+1) = \frac{\Theta(k+1)}{\pi} \quad \textbf{(EQ 4-8.)}$$

2. Subscript ***d*** denotes a digital representation of the corresponding constant/variable.

Secondly, the discrete-time integrators are replaced by accumulators; i.e., the integrators are computed only as summations without multiplying the input value by sampling period T_S. In comparison with **EQ 4-2**, the accumulator is defined as $y(k+1) = y(k) + x(k)$, where $x(k)$ and $y(k)$ are input and output values in step k.

The last step of the transformation is that the coefficients of *Angle Tracking Observer* K_1 in equation **EQ 4-4** and K_2 in equation **EQ 4-6** are replaced by their scaled equivalents K_{1d} and K_{2d} to reflect the scaling of position by π and the elimination of sampling period T_S in the integrator computation.

Finally, after the transformation, the equations suitable for implementation on the DSP core are as follows:

$$\Omega_d(k+1) = \Omega_d(k) + e(k) \cdot K_{1d} \qquad \textbf{(EQ 4-9.)}$$

$$Acc2_d(k+1) = Acc2_d(k) + \Omega_d(k) \qquad \textbf{(EQ 4-10.)}$$

$$\Theta_d(k+1) = K_{2d} \cdot \Omega_d(k+1) + Acc2_d(k+1) \qquad \textbf{(EQ 4-11.)}$$

$$e(k+1) = U_{\sin}(k+1) \cdot \cos(\pi \cdot \Theta_d(k+1)) - U_{\cos}(k+1) \cdot \sin(\pi \cdot \Theta_d(k+1)), \qquad \textbf{(EQ 4-12.)}$$

where the scaled coefficients K_{1d} and K_{2d} can be expressed after the derivation by the formulas:

$$K_{1d} = \frac{1}{\pi} \cdot T_s^2 \cdot K_1 = \frac{1}{\pi} \cdot \omega_n^2 \cdot T_s^2 \qquad \textbf{(EQ 4-13.)}$$

$$K_{2d} = \frac{K_2}{T_s} = \frac{2 \cdot \zeta}{\omega_n \cdot T_s} \qquad \textbf{(EQ 4-14.)}$$

where:ω_n is the natural frequency [rad s^{-1}], ζ is the damping factor [-] and T_S is the sampling period [s].

There is a $1/\pi$ included in the K_{1d} coefficient as a result of scaling the rotor position by π and the sampling period T_S in K_{1d} and K_{2d} coefficients as a result of replacing discrete-time integrators by accumulators.

The relation between the digital rotor speed $\Omega_d(k+1)$ in the range –1 to 1 and the actual rotor speed $\Omega(k+1)$ in [rad/s] is:

$$\Omega_d(k+1) = \frac{\Omega(k+1) \cdot T_s}{\pi}. \qquad \textbf{(EQ 4-15.)}$$

Note that this expression can be directly derived from the comparison of equations **EQ 4-4** and **EQ 4-9**. **Table 4-3** shows the maximal and minimal rotor speed that corresponds to the digital rotor speed of 1 and 2^{-31}, respectively.

Table 4-3. Maximal and Minimal Rotor Speed

Sampling Frequency [kHz]	Maximal Rotor Speed [r.p.m.]	Minimal Rotor(1) Speed [r.p.m.]
16	480000	0.00022
8	240000	0.00011
4	120000	0.00005

1. The minimal measurable rotor speed is very low. In certain application, however, it is typically limited by noise conditions to 0.1 r.p.m.

The functionality of the *Angle Tracking Observer* can also be explained using an example of the constant rotor speed. If the observer error $e(k)$ is zero then the first accumulator, representing speed $\Omega_d(k+1)$, remains constant. At every sampling period a constant value (first accumulator) - angular difference passed during T_s - is added to the second accumulator, representing position $Acc2_d(k+1)$. Note that implementation must reflect angular position overflow at the $\pi/-\pi$ boundary.

Before the resolver driver functions are used, the user is required to define the *Angle Tracking Observer* coefficients K1_D, K1_SCALE and K2_D, K2_SCALE in the include file *resolver.h*. These coefficients can be calculated using expressions:

$$K1\_D = K_{1d} \cdot 2^{K1\_SCALE} \qquad \textbf{(EQ 4-16.)}$$

$$K2\_D = K_{2d} \cdot 2^{-K2\_SCALE} \qquad \textbf{(EQ 4-17.)}$$

where: K_{1d} and K_{2d} are coefficients given by equation **EQ 4-16** and **EQ 4-17** and $K1\_SCALE$, $K2\_SCALE$ are chosen in such a way that $K1\_D,\ K2\_D \in \langle 0.5, 1.0)$.

Both coefficients K_{1d} and K_{2d} are normalized using introduced transformations to fit in a 16-bit fractional format. Having assigned scaling coefficients, the multiplication by coefficient K_{1d} (K_{2d}) can be easily performed by multiplication with its normalized value K1_D (K2_D) and then by shifting the result right (left) accordingly to the number of bits given by K1_SCALE (K2_SCALE).

It follows an example of calculation of the *Angle Tracking Observer* coefficients in Matlab language:

```
function [K1_D,K2_D,K1_SCALE,K2_SCALE,er1,er2,K1,K2] = trackdemparF0,D,Fs);
%
% function [K1_D,K2_D,K1_SCALE,K2_SCALE,er1,er2] = trackdempar(F0,D,Fs);
%
% Function calculates parameters of tracking observer.
% Input parameter:
% F0 - natural frequency of tracking observer [Hz]
% D  - damping factor [-]
% Fs - sampling frequency [Hz]
%
% Output parameters are calculated according to these formulas:
% K1 = (2*pi*F0/Fs)^2 / pi
% K2 = 2*D/(2*pi*F0/Fs)
%
% Example:
% F0=100; D=0.2; Fs=8000;
% [K1_D,K2_D,K1_SCALE,K2_SCALE,er1,er2,K1,K2] = trackdempar(F0,D,Fs)

format long
Q = 1/(2*D);  % quality factor
K1 = (2*pi*F0/Fs)^2/pi;% "Omega0^2" = (2*pi*F0/Fs)^2
K2 = 2*D/(2*pi*F0/Fs);% "2*Delta/Omega0" = Delta/(2*pi*F0)*Fs
K1_SCALE = floor(log2(1/K1));
K2_SCALE = ceil(log2(K2));
K1_D = K1*2^K1_SCALE;
K2_D = K2/2^K2_SCALE;
er1 = K1-K1_D/2^K1_SCALE;   % test if calculation is OK, should be 0
er2 = K2-K2_D*2^K2_SCALE;   % test if calculation is OK, should be 0
```

4.3.4 Returns

The function returns an estimation of the actual rotor angle, speed and number of revolutions in internal variables of the *Angle Tracking Observer* algorithm. Then these internal variables can be read by accessory functions **MCLIB_GetResPosition**, **MCLIB_GetResSpeed**

and **MCLIB_GetResRevolutions** respectively, which are described in next subsections.

4.3.5 Range Issues

Input sine and cosine arguments must be scaled to the range -1 to 1, before passing to that function. Function internally controls saturation state.

4.3.6 Implementation

The **MCLIB_CalcTrackObsv** function is implemented as a function call.

Code Example 4-2. MCLIB_CalcTrackObsv

```
/* include function and data prototypes required by the module  */
#include "mclib.h"

/* parameters of the Angle Tracking Observer                    */
/****************************************************************
 * F0 = 500/2/pi = 80 Hz, DAMPING = 0.84, Fs = 1/Ts = 8000Hz    *
 ****************************************************************/
#define K1_D         FRAC16(0.63661977236758)
#define K2_D         FRAC16(0.84000000000000)
#define K1_SCALE     9
#define K2_SCALE     5

/* interrupt function prototypes                                */
void adc_isr (void);

/* main function call                                           */
void main (void)
{
  /* define data                                                */
   Frac16   est_angle, est_revolutions;
   Frac16   est_speed;

   /* initialize Angle Tracking Observer algorithm               */
   MCLIB_InitTrackObsv (K1_D, K2_D, K1_SCALE, K2_SCALE);

   /* main program cycle                                         */
   while (1)
   {
     /* this function is supposed to be called when interrupt at */
     /* the end of adc conversion when new measured resolver sine*/
     /* and cosine voltages are available - it is called in this */
     /* loop intentionally to simplify example                   */
     adc_isr ();

     /* read resolver estimations                                */
     est_angle       = MCLIB_GetResPosition    ();
     est_revolutions = MCLIB_GetResRevolutions ();

     /* angular speed returned by MCLIB_GetResSpeed() function is*/
     /* 32-bit fractional value in range <-1; 1) that equals     */
     /* to the actual angular speed <-Fs*30; Fs*30)[RPM], where  */
     /* Fs is sampling frequency.                                */
     /* In this example Fs = 8kHz; i.e., the angular speed       */
     /* varies from -240000 to 240000[RPM]. To transform this    */
     /* value to 16-bit range <-1, 1), corresponding to -5000 to */
     /* 5000[RPM], the software must perform multiplication of   */
     /* the returned 32-bit value by: 240000/5000 => 0.75*2^6    */
     est_speed=round(L_shl(L_mult_ls(MCLIB_GetResSpeed(),FRAC16(0.75)),6));
   }
}

/* interrupt function definitions                               */
void adc_isr (void)
{
```

```
  static int angle = 0;   /* actual angle of the resolver shaft */

  /* calculate angle tracking observer using emulated resolver  */
  /* sine and cosine samples                                    */
  MCLIB_CalcTrackObsv (MCLIB_Sin2 (angle), MCLIB_Cos2 (angle));
  angle++;
}
```

4.3.7 Performance

Table 4-4. Performance of the MCLIB_CalcTrackObsv function.

Code Size	248 words	
Data Size	14 words	
Execution Clocks	Min	271 cycles
	Max	280 cycles

4.4 MCLIB_GetResPosition

4.4.1 Synopsis

#include ”mclib.h”
Frac16 ***MCLIB_GetResPosition*** *(void);*

4.4.2 Description

This function returns an estimate of the actual rotor angle. Note that function **MCLIB_CalcTrackObsv** must be called prior to the call of this driver function.

4.4.3 Returns

The returned rotor angle position is the 16-bit signed fractional value in the range -1 to 1 corresponding -pi to pi.

4.4.4 Implementation

The *getResPosition* function is implemented as a function call.

Code Example 4-3. MCLIB_GetResPosition

```
/* include function and data prototypes required by the module  */
#include "mclib.h"

/* parameters of the Angle Tracking Observer                     */
/*****************************************************************
 * F0 = 500/2/pi = 80 Hz, DAMPING = 0.84, Fs = 1/Ts = 8000Hz    *
 *****************************************************************/
#define K1_D         FRAC16(0.63661977236758)
#define K2_D         FRAC16(0.84000000000000)
#define K1_SCALE     9
#define K2_SCALE     5

/* interrupt function prototypes                                 */
void adc_isr (void);

/* main fucntion call                                            */
void main (void)
{
  /* define data                                                 */
  Frac16est_angle;

   /* initialize Angle Tracking Observer algorithm               */
   MCLIB_InitTrackObsv (K1_D, K2_D, K1_SCALE, K2_SCALE);

   /* main program cycle                                         */
   while (1)
   {
     /* this function is supposed to be called when interrupt at */
     /* the end of adc conversion when new measured resolver sine*/
     /* and cosine voltages are available - it is called in this */
     /* loop intentionally to simplify example                   */
     adc_isr ();
     /* read resolver estimations                                */
     est_angle       = MCLIB_GetResPosition ();
   }
}

/* interrupt function definitions                                */
void adc_isr (void)
{
  static int angle = 0;   /* actual angle of the resolver shaft */

  /* calculate angle tracking observer using emulated resolver  */
  /* sine and cosine samples                                    */
  MCLIB_CalcTrackObsv (MCLIB_Sin2 (angle), MCLIB_Cos2 (angle));
  angle++;
}
```

4.4.5 See Also

MCLIB_GetResSpeed, **MCLIB_GetResRevolutions**

4.4.6 Performance

Table 4-5. Performance of the MCLIB_GetResPosition function.

Code Size	15 words	
Data Size	2 words	
Execution Clocks	Min	29 cycles
	Max	29 cycles

4.5 MCLIB_GetResSpeed

4.5.1 Synopsis

#include ”mclib.h”
Frac32 ***MCLIB_GetResSpeed*** *(void);*

4.5.2 Description

This function returns the estimate of the actual rotor speed read from an internal variable of the Angle Tracking Observer algorithm. Note that the function **MCLIB_CalcTrackObsv** must be called prior to the call of this driver function.

4.5.3 Returns

The function returns a 32-bit signed fractional value in the range -1 to 1. The relation between the returned digital rotor speed in step *k+1*, marked as $\Omega_d\,(k+1)$, and the actual rotor speed in step *k+1*, $\Omega(k+1)$ in [rad/s] is

$$\Omega_d\,(k+1) = \frac{\Omega(k+1)\cdot T_s}{\pi}, \qquad \textbf{(EQ 4-18.)}$$

where T_S [s] is the sampling period.

4.5.4 Implementation

The **MCLIB_GetResSpeed** function is implemented as a function call.

Code Example 4-4. MCLIB_GetResSpeed

```
/* include function and data prototypes required by the module  */
#include "mclib.h"

/* parameters of the Angle Tracking Observer                     */
/*****************************************************************
 * F0 = 500/2/pi = 80 Hz, DAMPING = 0.84, Fs = 1/Ts = 8000Hz    *
 *****************************************************************/
#define K1_D         FRAC16(0.63661977236758)
#define K2_D         FRAC16(0.84000000000000)
#define K1_SCALE     9
#define K2_SCALE     5

/* interrupt function prototypes                                 */
void adc_isr (void);

/* main fucntion call                                            */
void main (void)
{
  /* define data                                                 */
   Frac16  est_speed;

   /* initialize Angle Tracking Observer algorithm               */
   MCLIB_InitTrackObsv (K1_D, K2_D, K1_SCALE, K2_SCALE);

   /* main program cycle                                         */
   while (1)
   {
     /* this function is supposed to be called when interrupt at */
     /* the end of adc conversion when new measured resolver sine*/
     /* and cosine voltages are available - it is called in this */
     /* loop intentionally to simplify example                   */
     adc_isr ();

    /* angular speed returned by MCLIB_GetResSpeed() function is*/
    /* 32-bit fractional value in range <-1; 1) that equals     */
    /* to the actual angular speed <-Fs*30; Fs*30)[RPM], where  */
    /* Fs is sampling frequency.                                */
    /* In this example Fs = 8kHz; i.e., the angular speed       */
    /* varies from -240000 to 240000[RPM]. To transform this    */
    /* value to 16-bit range <-1, 1), corresponding to -5000 to */
    /* 5000[RPM], the software must perform multiplication of   */
    /* the returned 32-bit value by: 240000/5000 => 0.75*2^6    */
    est_speed=round(L_shl(L_mult_ls(MCLIB_GetResSpeed(),FRAC16(0.75)),6));
   }
}

/* interrupt function definitions                                */
void adc_isr (void)
{
  static int angle = 0;   /* actual angle of the resolver shaft */

  /* calculate angle tracking observer using emulated resolver  */
  /* sine and cosine samples                                    */
  MCLIB_CalcTrackObsv (MCLIB_Sin2 (angle), MCLIB_Cos2 (angle));
```

```
  angle++;
}
```

4.5.5 See Also

MCLIB_GetResPosition, **MCLIB_GetResRevolutions**

4.5.6 Performance

Table 4-6. Performance of the MCLIB_GetResSpeed function.

Code Size	4 words	
Data Size	2 words	
Execution Clocks	Min	17 cycles
	Max	17 cycles

4.6 MCLIB_GetResRevolutions

4.6.1 Synopsis

#include ”mclib.h”
int ***MCLIB_GetResRevolutions*** *(void);*

4.6.2 Description

This function returns the estimate of the rotor revolutions. Note that the function **MCLIB_CalcTrackObsv** must be called prior to the call of this driver function.

4.6.3 Returns

This function returns the actual number of rotor revolutions. The returned value is taken as a 16-bit signed integer (range -32768 to 32767).

4.6.4 Implementation

The **MCLIB_GetResRevolutions** function is implemented as a function call.

Code Example 4-5. MCLIB_GetResRevolutions

```
/* include function and data prototypes required by the module  */
#include "mclib.h"

/* parameters of the Angle Tracking Observer                     */
/*****************************************************************
 * F0 = 500/2/pi = 80 Hz, DAMPING = 0.84, Fs = 1/Ts = 8000Hz    *
 *****************************************************************/
#define K1_D         FRAC16(0.63661977236758)
#define K2_D         FRAC16(0.84000000000000)
#define K1_SCALE     9
#define K2_SCALE     5

/* interrupt function prototypes                                 */
void adc_isr (void);

/* main fucntion call                                            */
void main (void)
{
  /* define data                                                 */
  Frac16est_revolutions;

   /* initialize Angle Tracking Observer algorithm               */
   MCLIB_InitTrackObsv (K1_D, K2_D, K1_SCALE, K2_SCALE);

   /* main program cycle                                         */
   while (1)
   {
     /* this function is supposed to be called when interrupt at */
     /* the end of adc conversion when new measured resolver sine*/
     /* and cosine voltages are available - it is called in this */
     /* loop intentionally to simplify example                   */
     adc_isr ();
     /* read resolver estimations                                */
     est_revolutions = MCLIB_GetResRevolutions ();
   }
}

/* interrupt function definitions                                */
void adc_isr (void)
{
  static int angle = 0;   /* actual angle of the resolver shaft */

  /* calculate angle tracking observer using emulated resolver  */
  /* sine and cosine samples                                    */
  MCLIB_CalcTrackObsv (MCLIB_Sin2 (angle), MCLIB_Cos2 (angle));
  angle++;
}
```

4.6.5 See Also

MCLIB_GetResPosition, **MCLIB_GetResSpeed**

4.6.6 Performance

Table 4-7. Performance of the MCLIB_GetResRevolutions function.

Code Size	4 words	
Data Size	1 words	
Execution Clocks	Min	17 cycles
	Max	17 cycles

4.7 MCLIB_SetResPosition

4.7.1 Synopsis

#include ”mclib.h”
*void **MCLIB_SetResPosition** (Frac16 newPosition);*

4.7.2 Arguments

newPosition	in	new rotor angle

4.7.3 Description

This function is used to set a new angle to the current angle. The function is supposed to initialize the instantaneous rotor angle to zero or any other value which might be useful at the start-up of the application.

4.7.4 Range Issues

The passed argument newPosition is in the 16-bit fractional in range -1 to 1 corresponding -pi to pi.

4.7.5 Implementation

The **MCLIB_SetResPosition** function is implemented as a function call.

Code Example 4-6. MCLIB_SetResPosition

```
/* include function and data prototypes required by the module  */
#include "mclib.h"

/* parameters of the Angle Tracking Observer                     */
/*****************************************************************
 * F0 = 500/2/pi = 80 Hz, DAMPING = 0.84, Fs = 1/Ts = 8000Hz    *
 *****************************************************************/
#define K1_D         FRAC16(0.63661977236758)
#define K2_D         FRAC16(0.84000000000000)
#define K1_SCALE     9
#define K2_SCALE     5

/* main fucntion call                                            */
void main (void)
{
  /* define data                                                 */
  Frac16est_revolutions;

   /* initialize Angle Tracking Observer algorithm                */
   MCLIB_InitTrackObsv (K1_D, K2_D, K1_SCALE, K2_SCALE);
   MCLIB_SetResPosition(20000);
}
```

4.7.6 See Also

MCLIB_SetResRevolutions

4.7.7 Performance

Table 4-8. Performance of the MCLIB_SetResPosition function.

Code Size	15 words	
Data Size	1 words	
Execution Clocks	Min	28 cycles
	Max	28 cycles

4.8 MCLIB_SetResRevolutions

4.8.1 Synopsis

#include ”mclib.h”
*void **MCLIB_SetResRevolutions** (int newRevolutions);*

4.8.2 Arguments

newRevolutions	in	new number of revolutions

4.8.3 Description

This function sets the number of revolutions that is stored in the internal variable of the Angle Tracking Observer algorithm. It is usually used to set the number of revolutions to zero in the application initialization phase.

4.8.4 Implementation

The **MCLIB_SetResRevolutions** function is implemented as a function call.

Code Example 4-7. MCLIB_SetResRevolutions

```
/* include function and data prototypes required by the module  */
#include "mclib.h"

/* parameters of the Angle Tracking Observer                    */
/****************************************************************
 * F0 = 500/2/pi = 80 Hz, DAMPING = 0.84, Fs = 1/Ts = 8000Hz    *
 ****************************************************************/
#define K1_D         FRAC16(0.63661977236758)
#define K2_D         FRAC16(0.84000000000000)
#define K1_SCALE     9
#define K2_SCALE     5

/* main fucntion call                                           */
void main (void)
{
  /* define data                                                */
  Frac16est_revolutions;

   /* initialize Angle Tracking Observer algorithm               */
   MCLIB_InitTrackObsv     (K1_D, K2_D, K1_SCALE, K2_SCALE);
   MCLIB_SetResRevolutions (15); /* sets number of revolutions   */
}
```

4.8.5 See Also

MCLIB_SetResPosition

4.8.6 Performance

Table 4-9. Performance of the MCLIB_SetResRevolutions function.

Code Size	4 words	
Data Size	1 words	
Execution Clocks	Min	18 cycles
	Max	18 cycles

Section 5. MODULATION TECHNIQUES

5.1 API Summary

Table 5-1. API Function Summary

Name	Arguments	Return	Description
MCLIB_SvmStd	MC_2PhSyst *p_AlphaBeta, MC_3PhSyst *p_abc	int	This function calculates appropriate duty-cycle ratios, which are needed for generating the given stator reference voltage vector using a special Standard Space Vector Modulation technique. Output duty-cycle ratios are calculated according to equations given in **Section: 5.2 MCLIB_SvmStd**
MCLIB_SvmU0n	MC_2PhSyst *p_AlphaBeta, MC_3PhSyst *p_abc	int	This function calculates the appropriate duty-cycle ratios, needed for generating the given stator reference voltage vector. It uses the special Space Vector Modulation technique, termed Space Vector Modulation with O000 Nulls. Output duty-cycle ratios are calculated according to equations given in **Section: 5.3 MCLIB_SvmU0n**
MCLIB_SvmU7n	MC_2PhSyst *p_AlphaBeta, MC_3PhSyst *p_abc	int	This function calculates the appropriate duty-cycle ratios, needed for generating the given stator reference voltage vector. It uses the special Space Vector Modulation technique, termed Space Vector Modulation with O111 Ones. Output duty-cycle ratios are calculated according to equations given in **Section: 5.4 MCLIB_SvmU7n**
MCLIB_SvmAlt	MC_2PhSyst *p_AlphaBeta, MC_3PhSyst *p_abc	int	This function calculates appropriate duty-cycle ratios, which are needed for generating the given stator reference voltage vector using a special Standard Space Vector Modulation technique. Output duty-cycle ratios are calculated according to equations given in **Section: 5.5 MCLIB_SvmAlt**

Table 5-1. API Function Summary

Name	Arguments	Return	Description
MCLIB_PwmIct	MC_2PhSyst *p_AlphaBeta, MC_3PhSyst *p_abc	int	This function calculates appropriate duty-cycle ratios, which are needed for generating the given stator reference voltage vector using the General Sinusoidal Modulation technique. Output duty-cycle ratios are calculated according to equations given in **Section: 5.6 MCLIB_PwmIct**
MCLIB_SvmSci	MC_2PhSyst *p_AlphaBeta, MC_3PhSyst *p_abc	int	This function calculates appropriate duty-cycle ratios, which are needed for generating the given stator reference voltage vector using the General Sinusoidal Modulation with an injection of the third harmonic. Output duty-cycle ratios are calculated according to equations given in **Section: 5.7 MCLIB_SvmSci**
MCLIB_ElimDcBusRip	Frac16 invModIndex, Frac16 u_DcBusMsr, MC_2PhSyst *pInp_AlphaBeta, MC_2PhSyst *pOut_AlphaBeta	void	Function is used for elimination of the DC-Bus voltage ripple as described in **Section: 5.8 MCLIB_ElimDcBusRip**

5.2 *MCLIB_SvmStd*

5.2.1 Synopsis

#include ”mclib.h”
int ***MCLIB_SvmStd*** *(MC_2PhSyst *p_AlphaBeta, MC_3PhSyst *p_abc);*

5.2.2 Arguments

*p_AlphaBeta	in	Pointer to the structure containing direct (alpha/α) and quadrature (beta/β) components of the stator voltage vector; the MC_2PhSyst data type is described in **Table 1-2**.
*p_abc	out	Pointer to the structure containing calculated duty-cycle ratios of the 3-Phase system; the MC_3PhSyst data type is described in **Table 1-2**.

5.2.3 Description

This approach to calculating duty-cycle ratios is widely-used in modern electric drives. This function calculates appropriate duty-cycle ratios, which are needed for generating the given stator reference voltage vector using a special Space Vector Modulation technique, termed Standard Space Vector Modulation.

The basic principle of the Standard Space Vector Modulation Technique can be explained with the help of the power stage diagram in **Figure 5-1**.

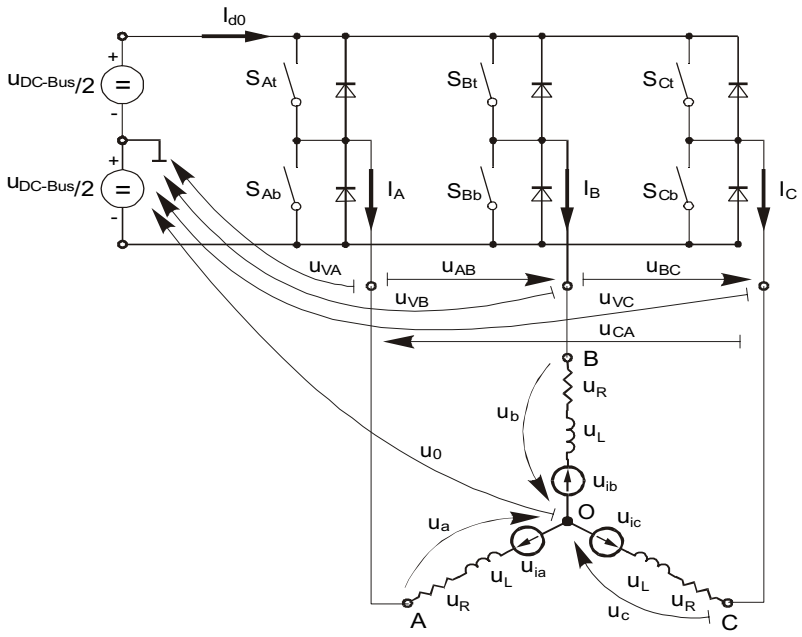

Figure 5-1. Power Stage Schematic Diagram

Top and bottom switches are working 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 numerical definition of all possible switching states. Phase-to-phase voltages can then be expressed in terms of these states:

$$\begin{bmatrix} U_{AB} \\ U_{BC} \\ U_{CA} \end{bmatrix} = U_{DCBus} \begin{bmatrix} 1 & -1 & 0 \\ 0 & 1 & -1 \\ -1 & 0 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} \qquad \textbf{(EQ 5-1.)}$$

where U_{DCBus} is the instantaneous voltage measured on the DCBus.

Assuming that the motor is ideally symmetrical, it's possible to write a matrix equation that expresses the motor phase voltages, shown in **EQ 5-1**.

$$\begin{bmatrix} U_a \\ U_b \\ U_c \end{bmatrix} = \frac{U_{DCBus}}{3} \begin{bmatrix} 2 & -1 & -1 \\ -1 & 2 & -1 \\ -1 & -1 & 2 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} \quad \textbf{(EQ 5-2.)}$$

In a 3-Phase power stage configuration, as shown in **Figure 5-1**, eight possible switching states (vectors), detailed in **Figure 5-2**, are feasible. These states, together with the resulting instantaneous output line-to-line and phase voltages, are listed in **Table 5-2**.

Table 5-2. 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	$2U_{DCBus}/3$	$-U_{DCBus}/3$	$-U_{DCBus}/3$	U_{DCBus}	0	$-U_{DCBus}$	U_0
1	1	0	$U_{DCBus}/3$	$U_{DCBus}/3$	$-2U_{DCBus}/3$	0	U_{DCBus}	$-U_{DCBus}$	U_{60}
0	1	0	$-U_{DCBus}/3$	$2U_{DCBus}/3$	$-U_{DCBus}/3$	$-U_{DCBus}$	U_{DCBus}	0	U_{120}
0	1	1	$-2U_{DCBus}/3$	$U_{DCBus}/3$	$U_{DCBus}/3$	$-U_{DCBus}$	0	U_{DCBus}	U_{240}
0	0	1	$-U_{DCBus}/3$	$-U_{DCBus}/3$	$2U_{DCBus}/3$	0	$-U_{DCBus}$	U_{DCBus}	U_{300}
1	0	1	$U_{DCBus}/3$	$-2U_{DCBus}/3$	$U_{DCBus}/3$	U_{DCBus}	$-U_{DCBus}$	0	U_{360}
1	1	1	0	0	0	0	0	0	O_{111}

The quantities of direct-α and quadrature-β components of the 2-Phase orthogonal coordinate system, describing the 3-Phase stator voltages, are expressed by the Clarke Transformation, arranged in a matrix form.

$$\begin{bmatrix} U_\alpha \\ U_\beta \end{bmatrix} = \frac{2}{3} \begin{bmatrix} 1 & -\frac{1}{2} & -\frac{1}{2} \\ 0 & \frac{\sqrt{3}}{2} & -\frac{\sqrt{3}}{2} \end{bmatrix} \begin{bmatrix} U_a \\ U_b \\ U_c \end{bmatrix} \quad \textbf{(EQ 5-3.)}$$

The 3-Phase stator voltages, U_a, U_b, and U_c, are transformed, using Clarke Transformation into direct-α and quadrature-β components of the

2-Phase orthogonal coordinate system. The transformation results are listed in **Table 5-3**

Table 5-3. Switching Patterns and Space Vectors

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

Figure 5-2 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.

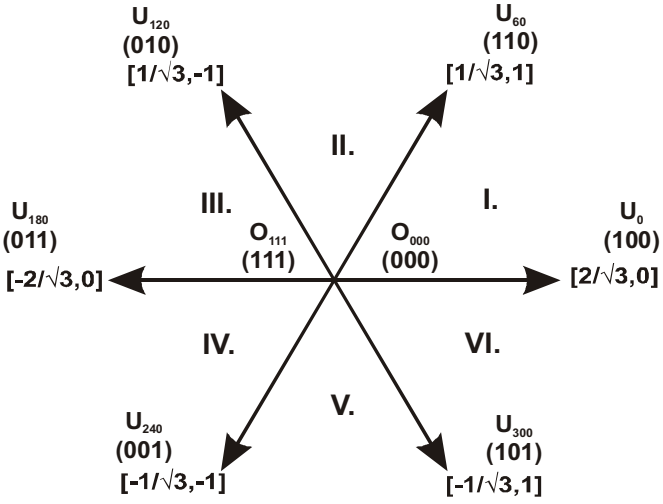

Figure 5-2. Basic Space Vectors

Referring to that principle, an objective of the Standard Space Vector Modulation is an approximation of the reference stator voltage vector U_S

with an appropriate combination of the switching patterns composed of basic space vectors. The graphical explanation of this objective is shown in **Figure 5-3** and **Figure 5-4**.

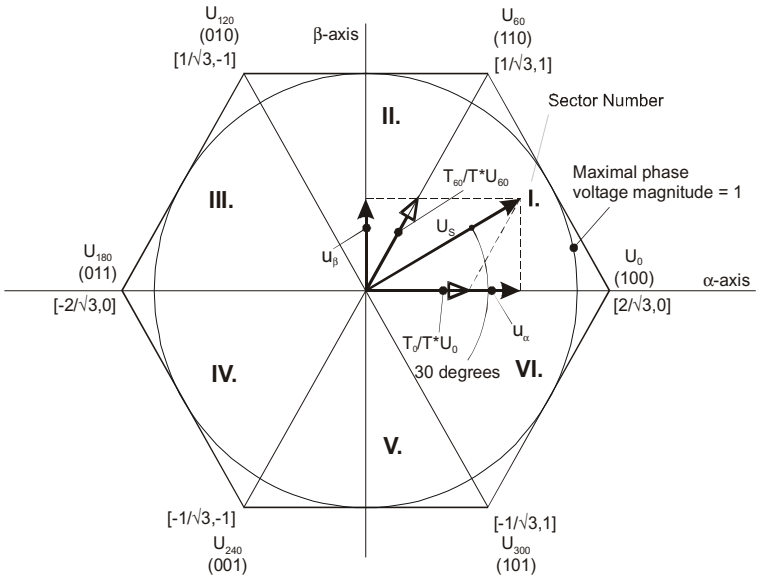

Figure 5-3. Projection of the Reference Voltage Vector in Sector I

The stator reference voltage vector U_S is phase-advanced by 30° from the direct-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 resultant direct-α and quadrature-β components for space vectors U_0 and U_{60}

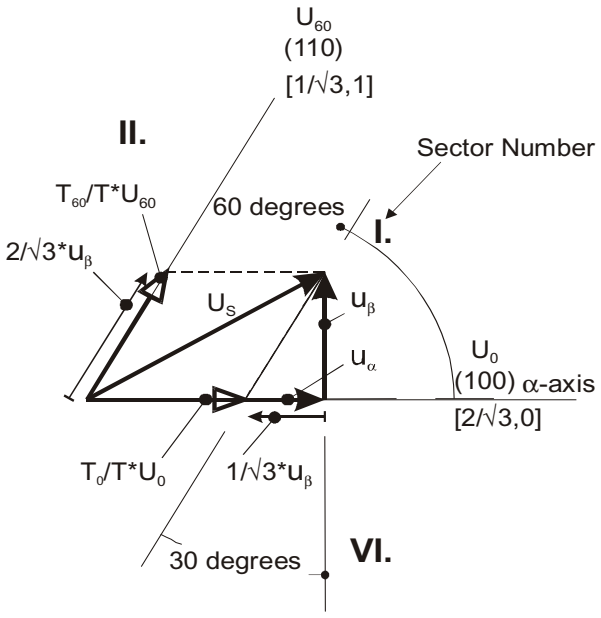

Figure 5-4. Detail of the Voltage Vector Projection in Sector I

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:

$$T = T_{60} + T_0 + T_{null} \quad \textbf{(EQ 5-4.)}$$

$$U_S = \frac{T_{60}}{T} \cdot U_{60} + \frac{T_0}{T} \cdot U_0 \quad \textbf{(EQ 5-5.)}$$

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:

$$u_\beta = \frac{T_{60}}{T} \cdot |U_{60}| \cdot \sin 60° \quad \textbf{(EQ 5-6.)}$$

$$u_\alpha = \frac{T_0}{T} \cdot |U_0| + \frac{u_\beta}{\tan 60°} \quad \textbf{(EQ 5-7.)}$$

Considering that normalized magnitudes of basic space vectors are |U60| = |U0| = $2/\sqrt{3}$ and by substitution of the trigonometric expressions sin60° and tan60° by their quantities $2/\sqrt{3}$ and $\sqrt{3}$, respectively, **EQ 5-6** and **EQ 5-7** can be rearranged for the unknown duty-cycle ratios T_{60}/T and T_0/T:

$$\frac{T_{60}}{T} = u_\beta \quad \textbf{(EQ 5-8.)}$$

$$\frac{T_0}{T} = \frac{1}{2} \cdot (\sqrt{3} \cdot u_\alpha - u_\beta) \quad \textbf{(EQ 5-9.)}$$

Sector II is depicted in **Figure 5-5**. 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:

$$T = T_{120} + T_{60} + T_{null} \quad \textbf{(EQ 5-10.)}$$

$$U_S = \frac{T_{120}}{T} \cdot U_{120} + \frac{T_{60}}{T} \cdot U_{60} \quad \textbf{(EQ 5-11.)}$$

where T_{120} and T_{60} are the respective duty-cycle ratios for which basic space vectors U_{120} and U_{60} 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. These resultant duty-cycle ratios are formed from the auxiliary components termed “A” and “B”. The graphical representation of the auxiliary components is shown in **Figure 5-6**.

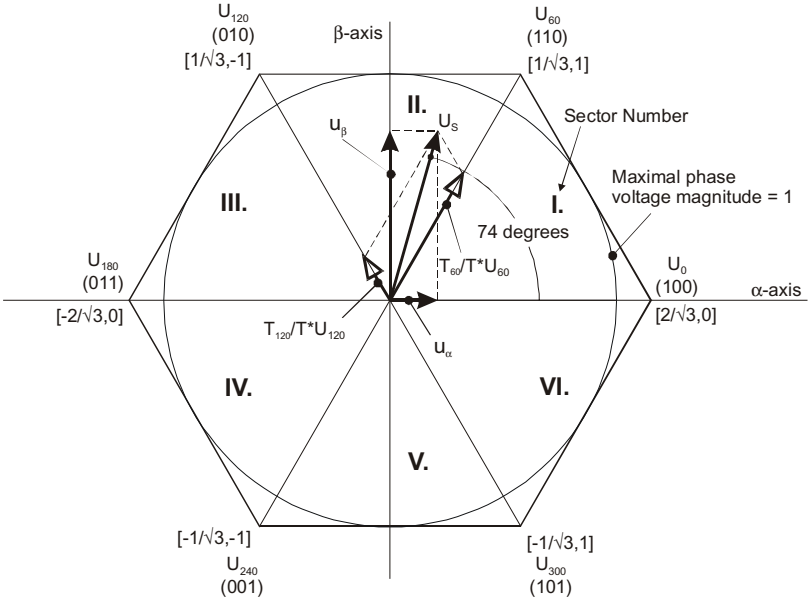

Figure 5-5. Projection of the Reference Voltage Vector in Sector II

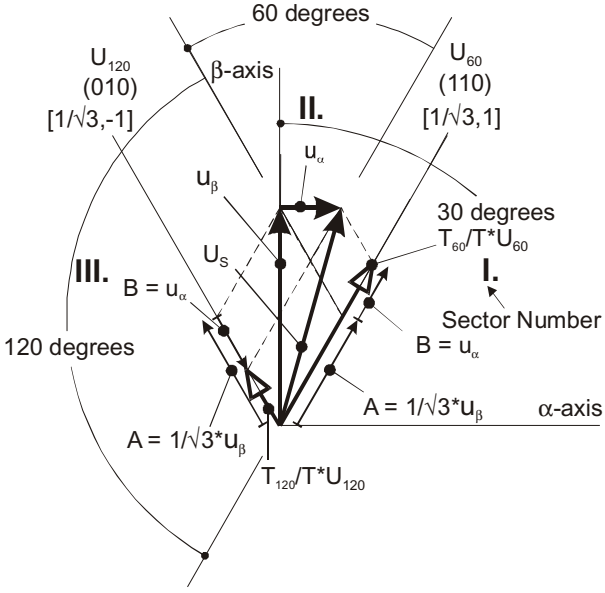

Figure 5-6. Detail of the Voltage Vector Projection in Sector II

The equations describing those auxiliary time-duration components are.

$$\frac{\sin 30°}{\sin 120°} = \frac{A}{u_\beta} \quad \textbf{(EQ 5-12.)}$$

$$\frac{\sin 60°}{\sin 60°} = \frac{B}{u_\alpha} \quad \textbf{(EQ 5-13.)}$$

EQ 5-12 and **EQ 5-13** have been formed using the Sinus 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 sin30°, sin120° and sin60° by their numerical representations 1/2, $\sqrt{3}/2$ and $1/\sqrt{3}$, respectively.

$$A = \frac{1}{\sqrt{3}} \cdot u_\beta \quad \textbf{(EQ 5-14.)}$$

$$B = u_\alpha \quad \textbf{(EQ 5-15.)}$$

The resultant duty-cycle ratios, T_{120}/T and T_{60}/T, are then expressed in terms of the auxiliary time-duration components defined by **EQ 5-16** and **EQ 5-17**, as follows:

$$\frac{T_{120}}{T} \cdot \left|U_{120}\right| = A - B \quad \textbf{(EQ 5-16.)}$$

$$\frac{T_{60}}{T} \cdot \left|U_{60}\right| = A + B \quad \textbf{(EQ 5-17.)}$$

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

$$\frac{T_{120}}{T} = \frac{1}{2} \cdot (u_\beta - \sqrt{3} \cdot u_\alpha) \quad \textbf{(EQ 5-18.)}$$

$$\frac{T_{60}}{T} = \frac{1}{2} \cdot (u_\beta + \sqrt{3} \cdot u_\alpha) \quad \textbf{(EQ 5-19.)}$$

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 basic space vectors for all sectors, we define:

- Three auxiliary variables:

  $X = u_\beta$

  $Y = 1/2 \cdot (u_\beta + \sqrt{3} \cdot u_\alpha)$

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

- Two expressions

  t_1

  t_2

  which generally represent duty-cycle ratios of basic space vectors in the respective sector; e.g., for the first sector, t_1 and t_2 represent duty-cycle ratios of basic space vectors U_{60} and U_0; for the second sector, t_1 and t_2 represent duty-cycle ratios of basic space vectors U_{120} and U_{60}, etc.

For each sector, the expressions t_1 and t_2, in terms of auxiliary variables X, Y and Z, are listed in **Table 5-4**.

Table 5-4. Determination of t_1 and t_2 Expressions

Sectors	U_0,U_{60}	U_{60},U_{120}	U_{120},U_{180}	U_{180},U_{240}	U_{240},U_{300}	U_{300},U_0
t_1	X	Y	-Y	Z	-Z	-X
t_2	-Z	Z	X	-X	-Y	Y

For the determination of auxiliary variables X, Y and Z, the Sector number is required. This information can be obtained by several approaches. One approach discussed here requires the use of modified Inverse Clark Transformation to transform the direct-α and quadrature-β

components into a balanced 3-Phase quantity u_{ref1}, u_{ref2} and u_{ref3}, used for straightforward calculation of the Sector number, to be shown later.

$$u_{ref1} = u_{\beta} \quad \textbf{(EQ 5-20.)}$$

$$u_{ref2} = \frac{-u_{\beta} + \sqrt{3} \cdot u_{\alpha}}{2} \quad \textbf{(EQ 5-21.)}$$

$$u_{ref3} = \frac{-u_{\beta} - \sqrt{3} \cdot u_{\alpha}}{2} \quad \textbf{(EQ 5-22.)}$$

The *modified* Inverse Clark Transformation projects the quadrature-u_{β} component into u_{ref1}, as shown in **Figure 5-7** and **Figure 5-8**, whereas voltages generated by the *conventional* Inverse Clark Transformation project the direct-u_{α} component into u_{ref1}.

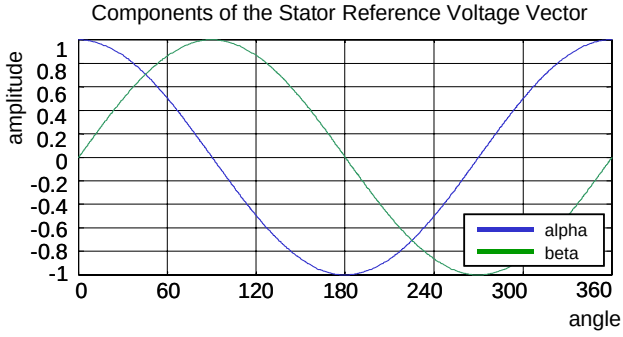

Figure 5-7. Direct-u_{α} and Quadrature-u_{β} Components of the Stator Reference Voltage

Figure 5-7 depicts the direct-u_{α} and quadrature-u_{β} components of the stator reference voltage vector U_S that were calculated by equations $u_{\alpha} = \cos \vartheta$ and $u_{\beta} = \sin \vartheta$, respectively.

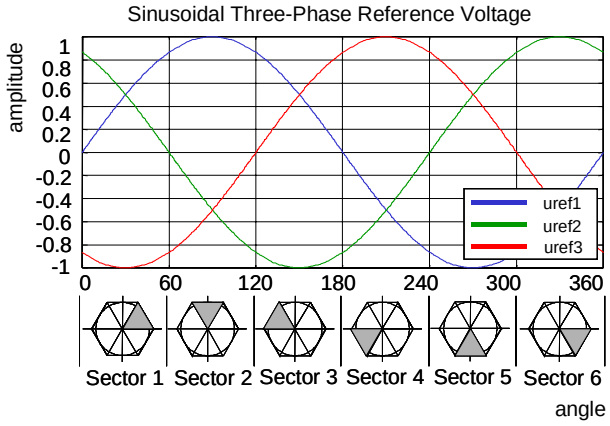

Figure 5-8. Reference Voltages u_{ref1}, u_{ref2} and u_{ref3}

The Sector Identification Tree, shown in **Figure 5-9**, can be a numerical solution of the approach shown in **Figure 5-8**.

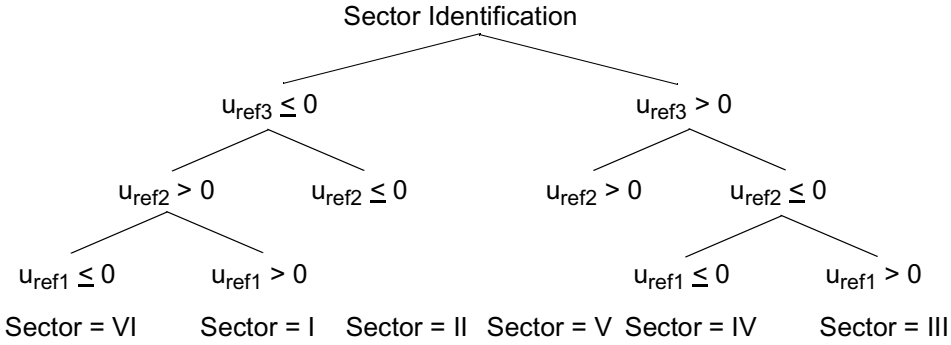

Figure 5-9. Identification of the Sector Number

It should be pointed out that, in the worst case, three simple comparisons are required to precisely identify the sector of the stator reference voltage vector. For example, if the stator reference voltage vector resides according to the one shown in **Figure 5-3**, the stator reference voltage vector is phase-advanced by 30° from the direct α-axis, which results in positive quantities of u_{ref1} and u_{ref2} and a negative quantity of u_{ref3}; refer to **Figure 5-8**. If these quantities are used as inputs to the Sector Identification Tree, the product of those comparisons will be Sector I. Using the same approach identifies Sector II, if the stator

reference voltage vector is located according to the one shown in **Figure 5-5**. The variables t_1, t_2 and t_3, representing switching duty-cycle ratios of the respective 3-Phase system, are given by the following equations:

$$t_1 = \frac{T - t\_1 - t\_2}{2} \quad \textbf{(EQ 5-23.)}$$

$$t_2 = t_1 + t\_1 \quad \textbf{(EQ 5-24.)}$$

$$t_3 = t_2 + t\_2 \quad \textbf{(EQ 5-25.)}$$

where T is the switching period, t_1 and t_2 are duty-cycle ratios of basic space vectors, given for the respective sector; **Table 5-4**. **EQ 5-23**, **EQ 5-24** and **EQ 5-25** are specific solely to the Standard Space Vector Modulation technique; consequently, other Space Vector Modulation techniques discussed later will require deriving different equations.

The next step is to assign the correct duty-cycle ratios, t_1, t_2 and t_3, to the respective motor phases. This is a simple task, accomplished in view of the position of the stator reference voltage vector; **Table 5-5**.

Table 5-5. Assignment of the Duty-Cycle Ratios to Motor Phases

Sectors	U_0,U_{60}	U_{60},U_{120}	U_{120},U_{180}	U_{180},U_{240}	U_{240},U_{300}	U_{300},U_0
pwm_a	t_3	t_2	t_1	t_1	t_2	t_3
pwm_b	t_2	t_3	t_3	t_2	t_1	t_1
pwm_c	t_1	t_1	t_2	t_3	t_3	t_2

The principle of the Space Vector Modulation technique consists in applying basic voltage vectors U_{XXX} and O_{XXX} for the certain time in such a way that the mean vector, generated by the Pulse Width Modulation approach for the period T, is equal to the original stator reference voltage vector U_S. This provides a great variability of the arrangement of basic vectors during the PWM period T. Those vectors might be arranged either to lower switching losses or to achieve diverse results, such as center-aligned PWM, edge-aligned PWM or a minimal number of switching states. A brief discussion of the widely-used center-aligned PWM follows.

Generating the center-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 **Figure 5-10**.

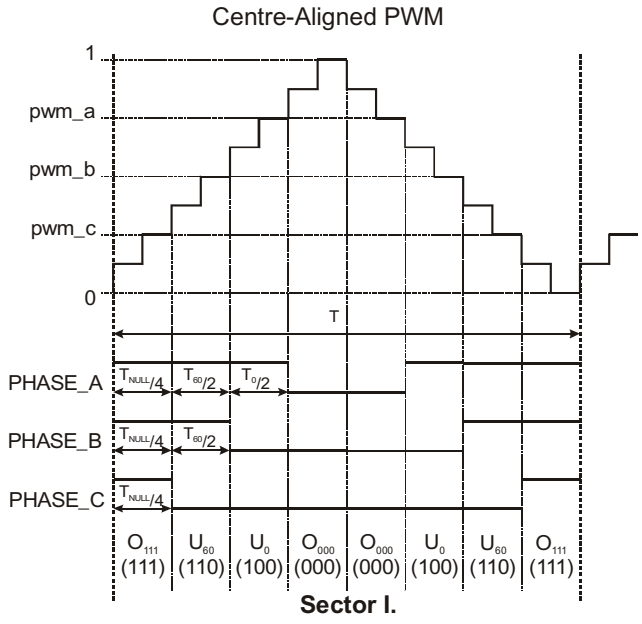

Figure 5-10. Standard Space Vector Modulation Technique - Center-Aligned PWM

Figure 5-11 graphically shows calculated waveforms of duty cycle ratios using Standard Space Vector Modulation.

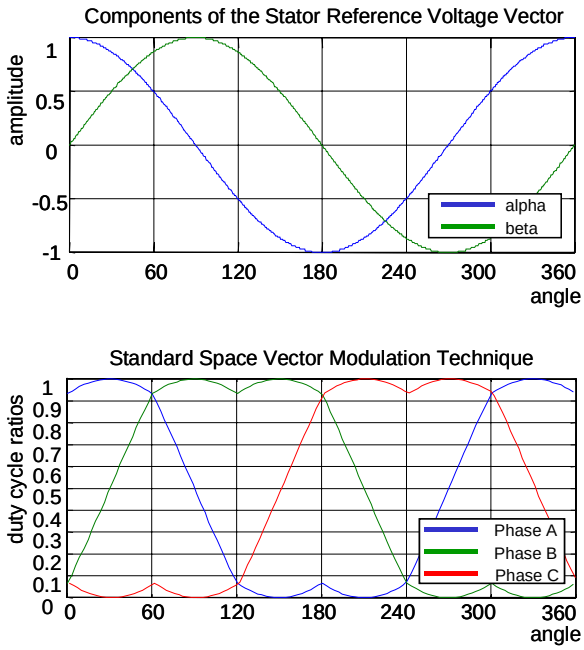

Figure 5-11. Standard Space Vector Modulation Technique

5.2.4 Returns

The function returns an integer value representing the Sector number in which the instantaneous stator reference voltage vector is located.

5.2.5 Range Issues

To provide an accurate calculation of the duty-cycle ratios, direct-a and quadrature-b components of the stator reference voltage vector must be considered as Q15 fractional numbers with their magnitude within the unit circle; i.e., the assumption $\sqrt{\alpha^2 + \beta^2} \leq 1$ must be met.

5.2.6 Special Issues

The **MCLIB_SvmStd** function is intended for periodical use; i.e., it might be called from a timer interrupt or a PWM updates interrupt. Referring to that, this function was programmed using assembler language with emphasis on maximizing the computational speed.

5.2.7 Implementation

The **MCLIB_SvmStd** function is implemented as a function call.

Code Example 5-1. MCLIB_SvmStd

```
/* include function and data prototypes required by the module  */
#include "mclib.h"

/* main function call                                           */
void main (void)
{
  /* define data                                                */
  MC_2PhSyst         twoPhSystem;
  MC_3PhSyst         threePhSystem;
  int                sector;

  /* load input variable by data                                */
  twoPhSystem.alpha = FRAC16(1.0);    /* corresponds to 1.0     */
  twoPhSystem.beta  = 0;              /* corresponds to 0.0     */

  /* computed duty cycles ratios and sector number              */
  sector = MCLIB_SvmStd (&twoPhSystem, &threePhSystem);
}
```

5.2.8 See Also

MCLIB_SvmStd, **MCLIB_SvmU0n**, **MCLIB_SvmU7n**, **MCLIB_SvmAlt**, **MCLIB_SvmSci**

5.2.9 Performance

Table 5-6. Performance of the MCLIB_SvmStd function.

Code Size	138 words	
Data Size	0 words	
Stack Size	4 words	
Execution Clocks	Min	91 cycles
	Max	104 cycles

5.3 MCLIB_SvmU0n

5.3.1 Synopsis

#include "mclib.h"
int ***MCLIB_SvmU0n*** *(MC_2PhSyst *p_AlphaBeta, MC_3PhSyst *p_abc);*

5.3.2 Arguments

*p_AlphaBeta	in	Pointer to the structure containing direct (alpha/α) and quadrature (beta/β) components of the stator voltage vector; the MC_2PhSyst data type is described in **Table 1-2**.
*p_abc	out	Pointer to the structure containing calculated duty-cycle ratios of the 3-Phase system; the MC_3PhSyst data type is described in **Table 1-2**.

5.3.3 Description

This function calculates the appropriate duty-cycle ratios, needed for generating the given stator reference voltage vector. It uses the special Space Vector Modulation technique, termed Space Vector Modulation with O_{000} Nulls.

The derivation approach of the Space Vector Modulation technique with O_{000} Nulls is identical, in many aspects, to the approach presented in in **Section 5.2, "MCLIB_SvmStd"** . However, a distinct difference lies in the definition of the variables t_1, t_2 and t_3 that represent switching duty-cycle ratios of respective phases:

$$t_1 = 0 \qquad \textbf{(EQ 5-26.)}$$

$$t_2 = t_1 + t\_1 \qquad \textbf{(EQ 5-27.)}$$

$$t_3 = t_2 + t\_2 \qquad \textbf{(EQ 5-28.)}$$

where T is the switching period and t_1 and t_2 are duty-cycle ratios of basic space vectors that are defined for the respective sector in **Table 5-4**.

The generally-used center-aligned PWM is discussed briefly in the following sections. Generating the center-aligned PWM pattern is accomplished practically by comparing threshold levels pwm_a, pwm_b,

and pwm_c with the free-running up-down counter. The timer counts up to 1 (0x7FFF) and then down to 0 (0x0000). It is supposed that, when a threshold level is larger than the timer value, the respective PWM output is active. Otherwise, it is inactive; see **Figure 5-12**.

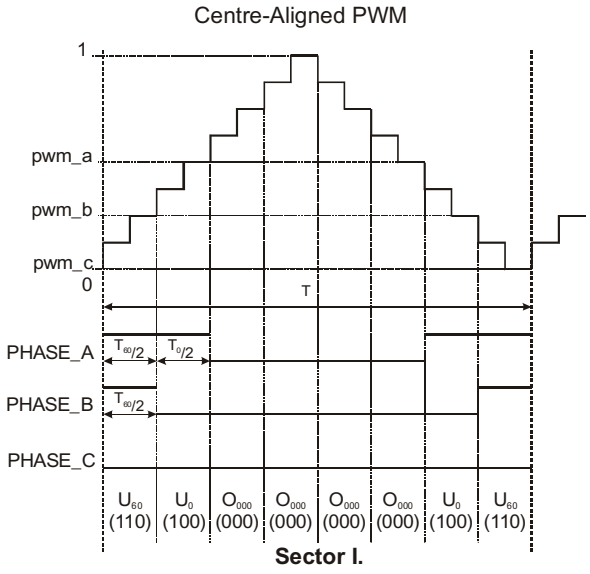

Figure 5-12. Space Vector Modulation Technique with O_{000} Nulls - Center-Aligned PWM

Figure 5-13 graphically shows calculated waveforms of duty cycle ratios shows using Space Vector Modulation with O000 Nulls.

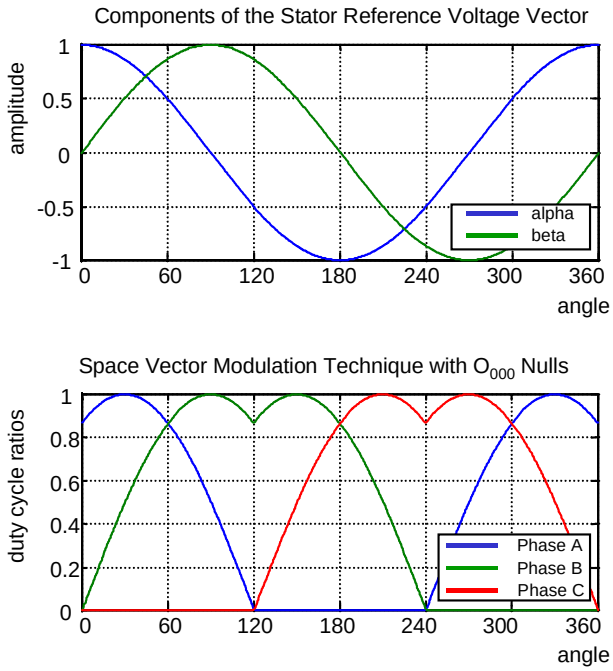

Figure 5-13. Space Vector Modulation Technique with O_{000} Nulls

5.3.4 Returns

The function returns an integer value representing the Sector number in which the instantaneous stator reference voltage vector is located.

5.3.5 Range Issues

To provide an accurate calculation of the duty-cycle ratios, direct-a and quadrature-b components of the stator reference voltage vector must be considered as Q15 fractional numbers with their magnitude within the unit circle; i.e., the assumption $\sqrt{\alpha^2 + \beta^2} \leq 1$ must be met.

5.3.6 Special Issues

The svmU0n function is intended for periodical use; i.e., it might be called from a timer interrupt or a PWM updates interrupt. Referring to that, this function was programmed using assembler language with emphasis on maximizing the computational speed.

5.3.7 Implementation

The **MCLIB_SvmU0n** function is implemented as a function call.

Code Example 5-2. MCLIB_SvmU0n

```
/* include function and data prototypes required by the module  */
#include "mclib.h"

/* main function call                                           */
void main (void)
{
  /* define data                                                */
  MC_2PhSyst         twoPhSystem;
  MC_3PhSyst         threePhSystem;
  int                sector;

  /* load input variable by data                                */
  twoPhSystem.alpha = FRAC16(1.0);    /* corresponds to 1.0     */
  twoPhSystem.beta  = 0;              /* corresponds to 0.0     */

  /* computed duty cycles ratios and sector number              */
  sector = MCLIB_SvmU0n (&twoPhSystem, &threePhSystem);
}
```

5.3.8 See Also

MCLIB_SvmStd, **MCLIB_SvmU7n**, **MCLIB_SvmAlt**, **MCLIB_PwmIct**, **MCLIB_SvmSci**

5.3.9 Performance

Table 5-7. Performance of the MCLIB_SvmU0n function.

Code Size	123 words	
Data Size	0 words	
Execution Clocks	Min	88 cycles
	Max	100 cycles

5.4 MCLIB_SvmU7n

5.4.1 Synopsis

#include "mclib.h"
int ***MCLIB_SvmU7n*** *(MC_2PhSyst *p_AlphaBeta, MC_3PhSyst *p_abc);*

5.4.2 Arguments

*p_AlphaBeta	in	Pointer to the structure containing direct (alpha/α) and quadrature (beta/β) components of the stator voltage vector; the MC_2PhSyst data type is described in **Table 1-2**.
*p_abc	out	Pointer to the structure containing calculated duty-cycle ratios of the 3-Phase system; the MC_3PhSyst data type is described in **Table 1-2**.

5.4.3 Description

This function calculates the appropriate duty-cycle ratios, needed for generating the given stator reference voltage vector. It uses the special Space Vector Modulation technique, termed Space Vector Modulation with O_{111} Nulls.

The derivation approach of the Space Vector Modulation technique with O_{111} Nulls is identical, in many aspects, to the approach presented in in **Section 5.2, "MCLIB_SvmStd"** . However, a distinct difference lies in the definition of the variables t_1, t_2 and t_3 that represent the switching duty-cycle ratios of the respective phases.

$$t_1 = T - t\_1 - t\_2 \quad \textbf{(EQ 5-29.)}$$

$$t_2 = t_1 + t\_1 \quad \textbf{(EQ 5-30.)}$$

$$t_3 = t_2 + t\_2 \quad \textbf{(EQ 5-31.)}$$

where T is the switching period, and t_1 and t_2 are duty-cycles ratios of the space vectors that are defined for the respective sector in **Table 5-4**.

Generating the center-aligned PWM pattern is accomplished practically by comparing the threshold levels pwm_a, pwm_b, and pwm_c with the free-running up-down counter. The timer counts up to 1 (0x7FFF) and then down to 0 (0x0000). It is supposed that when a threshold level is

larger than the timer value, the respective PWM output is active. Otherwise, it is inactive; see **Figure 5-14**.

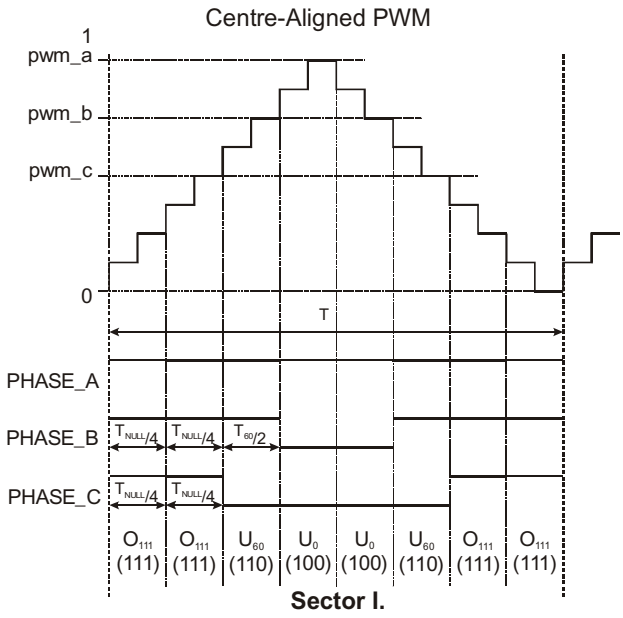

Figure 5-14. Space Vector Modulation Technique with O_{111} Nulls - Center-Aligned PWM

Figure 5-14 graphically shows calculated waveforms of duty cycle ratios shows using Space Vector Modulation with O_{111} Nulls.

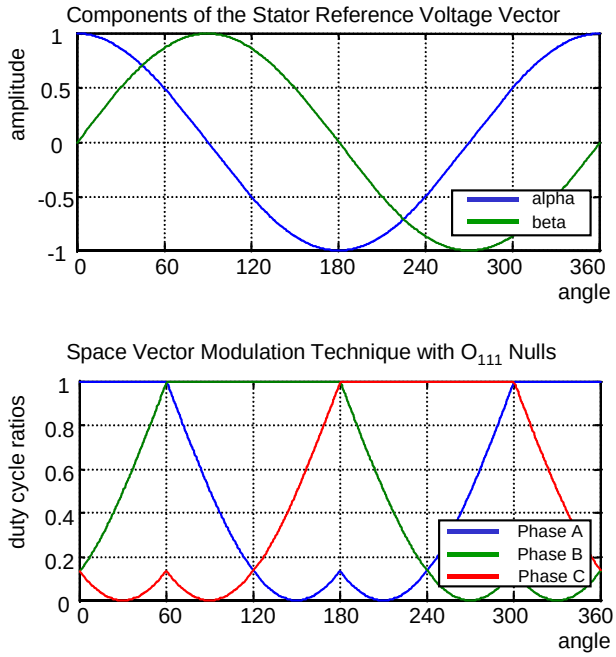

Figure 5-15. Space Vector Modulation with O_{111} Nulls

5.4.4 Returns

The function returns an integer value representing the Sector number in which the instantaneous stator reference voltage vector is located.

5.4.5 Range Issues

To provide an accurate calculation of the duty-cycle ratios, direct-a and quadrature-b components of the stator reference voltage vector must be considered as Q15 fractional numbers with their magnitude within the unit circle; i.e., the assumption $\sqrt{\alpha^2 + \beta^2} \leq 1$ must be met.

5.4.6 Special Issues

The **MCLIB_SvmU7n** function is intended for periodical use; i.e., it might be called from a timer interrupt or a PWM updates interrupt. Referring to that, this function was programmed using assembler language with emphasis on maximizing the computational speed.

5.4.7 Implementation

The **MCLIB_SvmU7n** function is implemented as a function call.

Code Example 5-3. MCLIB_SvmU7n

```
/* include function and data prototypes required by the module  */
#include "mclib.h"

/* main function call                                           */
void main (void)
{
  /* define data                                                */
  MC_2PhSyst         twoPhSystem;
  MC_3PhSyst         threePhSystem;
  int                sector;

  /* load input variable by data                                */
  twoPhSystem.alpha = FRAC16(1.0);    /* corresponds to 1.0     */
  twoPhSystem.beta  = 0;              /* corresponds to 0.0     */

  /* computed duty cycles ratios and sector number              */
  sector = MCLIB_SvmU7n (&twoPhSystem, &threePhSystem);
}
```

5.4.8 See Also

MCLIB_SvmStd, **MCLIB_SvmU0n**, **MCLIB_SvmAlt**, **MCLIB_PwmIct**, **MCLIB_SvmSci**

5.4.9 Performance

Table 5-8. Performance of the MCLIB_SvmU7n function.

Code Size	131 words	
Data Size	0 words	
Execution Clocks	Min	89 cycles
	Max	101 cycles

5.5 MCLIB_SvmAlt

5.5.1 Synopsis

#include ”mclib.h”
int ***MCLIB_SvmAlt*** *(MC_2PhSyst *p_AlphaBeta, MC_3PhSyst *p_abc);*

5.5.2 Arguments

*p_AlphaBeta	in	Pointer to the structure containing direct (alpha/α) and quadrature (beta/β) components of the stator voltage vector; the MC_2PhSyst data type is described in **Table 1-2**.
*p_abc	out	Pointer to the structure containing calculated duty-cycle ratios of the 3-Phase system; the MC_3PhSyst data type is described in **Table 1-2**.

5.5.3 Description

This function calculates the appropriate duty-cycle ratios, needed for generating the given stator reference voltage vector. It uses the special Space Vector Modulation technique, termed Space Vector Modulation with states O_{000} in even sectors and state O_{111} in odd sectors.

The derivation approach of this Space Vector Modulation technique is identical, in many aspects, to the approach presented in in **Section 5.2, “MCLIB_SvmStd”** . However, a distinct difference lies in the definition of the variables t1, t2 and t3 that represent the switching duty-cycle ratios of the respective phases. These variables are given for even sectors (2,4,6) by the same equations as those defined in **Section 5.3, “MCLIB_SvmU0n”** .

$$t_1 = 0 \qquad \textbf{(EQ 5-32.)}$$

$$t_2 = t_1 + t\_1 \qquad \textbf{(EQ 5-33.)}$$

$$t_3 = t_2 + t\_2 \qquad \textbf{(EQ 5-34.)}$$

For the odd sectors (1,3,5), these variables are given by equations that are identical to those defined in **Section 5.4, "MCLIB_SvmU7n"** .

$$t_1 = T - t\_1 - t\_2 \quad \textbf{(EQ 5-35.)}$$

$$t_2 = t_1 + t\_1 \quad \textbf{(EQ 5-36.)}$$

$$t_3 = t_2 + t\_2 \quad \textbf{(EQ 5-37.)}$$

where T is the switching period, t_1 and t_2 are duty-cycle ratios of the space vectors, which are defined for the respective sector in **Table 5-4**. Generating the center-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. This timer counts up to 1 (0x7FFF) and then down to 0 (0x0000). When a threshold level is larger than the counter value, the respective PWM output is active. Otherwise, it is inactive; see **Figure 5-16**.

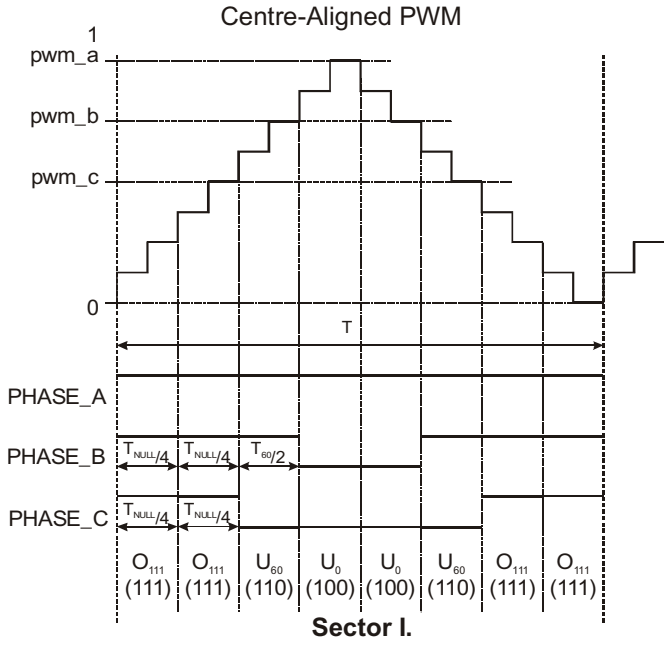

Figure 5-16. Space Vector Modulation Technique with Alternate Nulls - Center-Aligned PWM

Figure 5-17 graphically shows calculated waveforms of duty cycle ratios shows using Space Vector Modulation with states O_{000} in even sectors and state O_{111} in odd sectors.

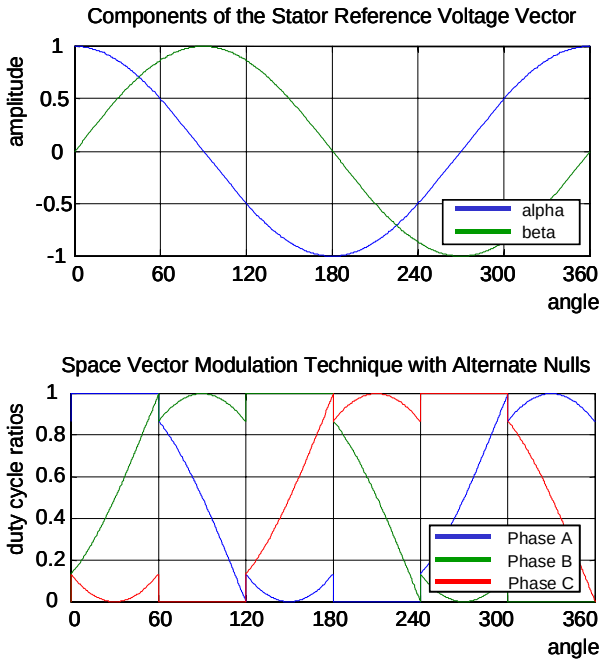

Figure 5-17. Space Vector Modulation Technique with Alternate Nulls

5.5.4 Returns

The function returns an integer value representing the Sector number in which the instantaneous stator reference voltage vector is located.

5.5.5 Range Issues

To provide an accurate calculation of the duty-cycle ratios, direct-a and quadrature-b components of the stator reference voltage vector must be considered as Q15 fractional numbers with their magnitude within the unit circle; i.e., the assumption $\sqrt{\alpha^2 + \beta^2} \leq 1$ must be met.

5.5.6 Special Issues

The svmAlt function is intended for periodical use; i.e., it might be called from a timer interrupt or a PWM updates interrupt. Referring to that, this function was programmed using assembler language with emphasis on maximizing the computational speed.

5.5.7 Implementation

The **MCLIB_SvmAlt** function is implemented as a function call.

Code Example 5-4. MCLIB_SvmAlt

```
/* include function and data prototypes required by the module  */
#include "mclib.h"

/* main function call                                           */
void main (void)
{
  /* define data                                                */
  MC_2PhSyst         twoPhSystem;
  MC_3PhSyst         threePhSystem;
  int                sector;

  /* load input variable by data                                */
  twoPhSystem.alpha = FRAC16(1.0);    /* corresponds to 1.0     */
  twoPhSystem.beta  = 0;              /* corresponds to 0.0     */

  /* computed duty cycles ratios and sector number              */
  sector = MCLIB_SvmAlt (&twoPhSystem, &threePhSystem);
}
```

5.5.8 See Also

MCLIB_SvmStd, **MCLIB_SvmU0n**, **MCLIB_SvmU7n**, **MCLIB_PwmIct**, **MCLIB_SvmSci**

5.5.9 Performance

Table 5-9. Performance of the MCLIB_SvmAlt function.

Code Size	125 words	
Data Size	0 words	
Execution Clocks	Min	88 cycles
	Max	100 cycles

5.6 MCLIB_PwmIct

5.6.1 Synopsis

#include ”mclib.h”
int ***MCLIB_PwmIct*** *(MC_2PhSyst *p_AlphaBeta, MC_3PhSyst *p_abc);*

5.6.2 Arguments

*p_AlphaBeta	in	Pointer to the structure containing direct (alpha/α) and quadrature (beta/β) components of the stator voltage vector; the MC_2PhSyst data type is described in **Table 1-2**.
*p_abc	out	Pointer to the structure containing calculated duty-cycle ratios of the 3-Phase system; the MC_3PhSyst data type is described in **Table 1-2**.

5.6.3 Description

This function calculates the appropriate duty-cycle ratios, needed for generating the given stator reference voltage vector with the help of conventional Inverse Clark transformation.

Finding the sector in which the reference stator voltage vector U_S resides is similar to that discussed in **Section 5.2, “MCLIB_SvmStd”** . This is achieved by first converting the direct-α and the quadrature-β components of the reference stator voltage vector U_S into balanced 3-Phase quantities u_{ref1}, u_{ref2} and u_{ref3}, using the modified Inverse Clark Transform:

$$u_{ref1} = u_{\beta} \qquad \textbf{(EQ 5-38.)}$$

$$u_{ref2} = \frac{-u_{\beta} + \sqrt{3} \cdot u_{\alpha}}{2} \qquad \textbf{(EQ 5-39.)}$$

$$u_{ref3} = \frac{-u_{\beta} - \sqrt{3} \cdot u_{\alpha}}{2} \qquad \textbf{(EQ 5-40.)}$$

The calculation of the sector number is based on comparing the 3-Phase reference voltages $uref_1$, $uref_2$ and $uref_3$ with zero. This computation can be described by the following set of rules:

$$a = \begin{cases} 1.0 & \text{if } u_{refl} > 0 \\ 0 & \text{otherwise} \end{cases} \quad \textbf{(EQ 5-41.)}$$

$$b = \begin{cases} 2.0 & \text{if } u_{ref2} > 0 \\ 0 & \text{otherwise} \end{cases} \quad \textbf{(EQ 5-42.)}$$

$$c = \begin{cases} 4.0 & \text{if } u_{ref3} > 0 \\ 0 & \text{otherwise} \end{cases} \quad \textbf{(EQ 5-43.)}$$

After passing these rules, modified sector numbers are then derived from the formula sector * = a + b + c

The sector numbers determined by this formula must be further transformed in order to correspond to those which would be determined by the Sector Identification Tree. The transformation, which meets this requirement, is shown in **Table 5-10**.

Table 5-10. Transformation of the Sectors

sector *	1	2	3	4	5	6
sector	2	6	1	4	3	5

The Inverse Clark Transformation might be used for transforming values such as flux, voltage and current from an orthogonal coordination system (uα, uβ) to a 3-Phase rotating coordination system (u_a, u_b and u_c). The original equations of the Inverse Clark Transformation are scaled here to provide duty-cycle ratios in the range 0 < pwm_x < 1, where x refers to the corresponding phases. These scaled duty-cycle

ratios pwm_a, pwm_b and pwm_c might be used directly by registers of the PWM block.

$$pwm\_a = 0.5 + \frac{u_\alpha}{2} \quad \textbf{(EQ 5-44.)}$$

$$pwm\_b = 0.5 + \frac{-u_\alpha + \sqrt{3} \cdot u_\beta}{4} \quad \textbf{(EQ 5-45.)}$$

$$pwm\_c = 0.5 + \frac{-u_\alpha - \sqrt{3} \cdot u_\beta}{4} \quad \textbf{(EQ 5-46.)}$$

Figure 5-18 graphically shows calculated waveforms of duty cycle ratios shows using Inverse Clark Transformation.

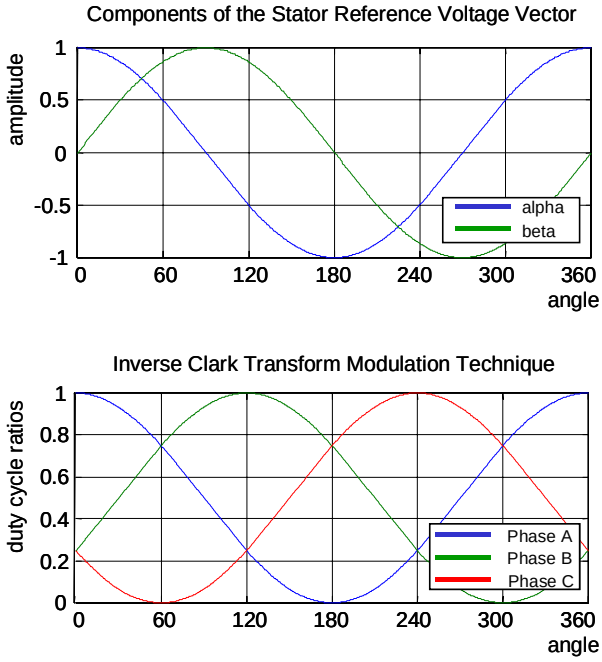

Figure 5-18. Inverse Clark Modulation Technique

5.6.4 Returns

The function returns an integer value representing the Sector number in which the instantaneous stator reference voltage vector is located.

5.6.5 Range Issues

To provide an accurate calculation of the duty-cycle ratios, direct-a and quadrature-b components of the stator reference voltage vector must be considered as Q15 fractional numbers with their magnitude within the unit circle; i.e., the assumption $\sqrt{\alpha^2 + \beta^2} \leq 1$ must be met.

5.6.6 Special Issues

The **MCLIB_PwmIct** function is intended for periodical use; i.e., it might be called from a timer interrupt or a PWM updates interrupt. Referring to that, this function was programmed using assembler language with emphasis on maximizing the computational speed.

5.6.7 Implementation

The **MCLIB_PwmIct** function is implemented as a function call.

Code Example 5-5. MCLIB_PwmIct

```
/* include function and data prototypes required by the module  */
#include "mclib.h"

/* main function call                                           */
void main (void)
{
  /* define data                                                */
  MC_2PhSyst         twoPhSystem;
  MC_3PhSyst         threePhSystem;
  int                sector;

  /* load input variable by data                                */
  twoPhSystem.alpha = FRAC16(1.0);    /* corresponds to 1.0     */
  twoPhSystem.beta  = 0;              /* corresponds to 0.0     */

  /* computed duty cycles ratios and sector number              */
  sector = MCLIB_PwmIct (&twoPhSystem, &threePhSystem);
}
```

5.6.8 See Also

MCLIB_SvmStd, **MCLIB_SvmU0n**, **MCLIB_SvmU7n**, **MCLIB_SvmAlt**, **MCLIB_SvmSci**

5.6.9 Performance

Table 5-11. Performance of the MCLIB_PwmIct function.

Code Size	70 words	
Data Size	7 words	
Execution Clocks	Min	106 cycles
	Max	107 cycles

5.7 MCLIB_SvmSci

5.7.1 Synopsis

#include "mclib.h"
int ***MCLIB_SvmSci*** *(MC_2PhSyst *p_AlphaBeta, MC_3PhSyst *p_abc);*

5.7.2 Arguments

*p_AlphaBeta	in	Pointer to the structure containing direct (alpha/α) and quadrature (beta/β) components of the stator voltage vector; the MC_2PhSyst data type is described in **Table 1-2**.
*p_abc	out	Pointer to the structure containing calculated duty-cycle ratios of the 3-Phase system; the MC_3PhSyst data type is described in **Table 1-2**.

5.7.3 Description

This function calculates the appropriate duty-cycle ratios, needed for generating the given stator reference voltage vector with the help of sinusoidal modulation with Sine-Cap Injection algorithm.

Finding the sector in which the reference stator voltage vector U_S resides is similar to that discussed in **Section 5.2, "MCLIB_SvmStd"** .

The balanced 3-Phase duty-cycle ratios may be calculated based on Sine Cap Injection algorithm in the following stages:

The calculation of basic duty-cycle ratios using the Inverse Clarke Transformation.

$$u_a = u_\alpha \qquad \textbf{(EQ 5-47.)}$$

$$u_b = \frac{-u_\alpha + \sqrt{3} \cdot u_\beta}{2} \qquad \textbf{(EQ 5-48.)}$$

$$u_c = \frac{-u_\alpha - \sqrt{3} \cdot u_\beta}{2} \qquad \textbf{(EQ 5-49.)}$$

An amplitude of the basic duty-cycle ratios u_a, u_b and u_c calculated by **EQ 5-47**, **EQ 5-48** and **EQ 5-49** is in the range [-1, 1].

Basic duty-cycle ratios are then multiplied by the coefficient 2/(√3).

$$u'_a = \frac{2}{\sqrt{3}} \cdot u_a \qquad \textbf{(EQ 5-50.)}$$

$$u'_b = \frac{2}{\sqrt{3}} \cdot u_b \qquad \textbf{(EQ 5-51.)}$$

$$u'_c = \frac{2}{\sqrt{3}} \cdot u_c \qquad \textbf{(EQ 5-52.)}$$

It should be noted that the values of these variables are within the range -2/(√3) <u′x <2/(√3); therefore, smart scaling of the fractional numbers must be utilized to provide fractional calculations with an adequate accuracy level. For more information about scaling, refer to the assembler source code of the described modulation function in module *svm.c*.

If values of variables u'_a, u'_b and u'_c exceed unity, they are stored in an auxiliary variable u_0. This variable will be called the Sine Cap Voltage variable. The procedure to obtain it can be mathematically defined by a series of three formula:

$$u_0 = \begin{cases} 1.0 - u'_a & \text{if } u'_a > 1.0 \\ -1.0 - u'_a & \text{if } u'_a < -1.0 \\ 0 & otherwise \end{cases} \qquad \textbf{(EQ 5-53.)}$$

$$u_0 = \begin{cases} 1.0 - u'_b & \text{if } u'_b > 1.0 \\ -1.0 - u'_b & \text{if } u'_b < -1.0 \\ 0 & otherwise \end{cases} \qquad \textbf{(EQ 5-54.)}$$

$$u_0 = \begin{cases} 1.0 - u'_c & \text{if } u'_c > 1.0 \\ -1.0 - u'_c & \text{if } u'_c < -1.0 \\ 0 & otherwise \end{cases} \qquad \textbf{(EQ 5-55.)}$$

Note that due to the 120° voltage phase shift, that is distinguishing for the balanced 3-Phase system, only one phase will contribute to the building of Sine-Cap Voltage u_0 at each time point.

Final duty-cycle ratios are then calculated by following equations:

$$pwm\_\mathrm{a} = \frac{1}{2} \cdot (u_0 + u'_a + 1) \qquad \textbf{(EQ 5-56.)}$$

$$pwm\_\mathrm{b} = \frac{1}{2} \cdot (u_0 + u'_b + 1) \qquad \textbf{(EQ 5-57.)}$$

$$pwm\_\mathrm{c} = \frac{1}{2} \cdot (u_0 + u'_c + 1) \qquad \textbf{(EQ 5-58.)}$$

Figure 5-19 graphically shows calculated waveforms of duty cycle ratios shows using sinusoidal modulation with Sine-Cap Injection algorithm.

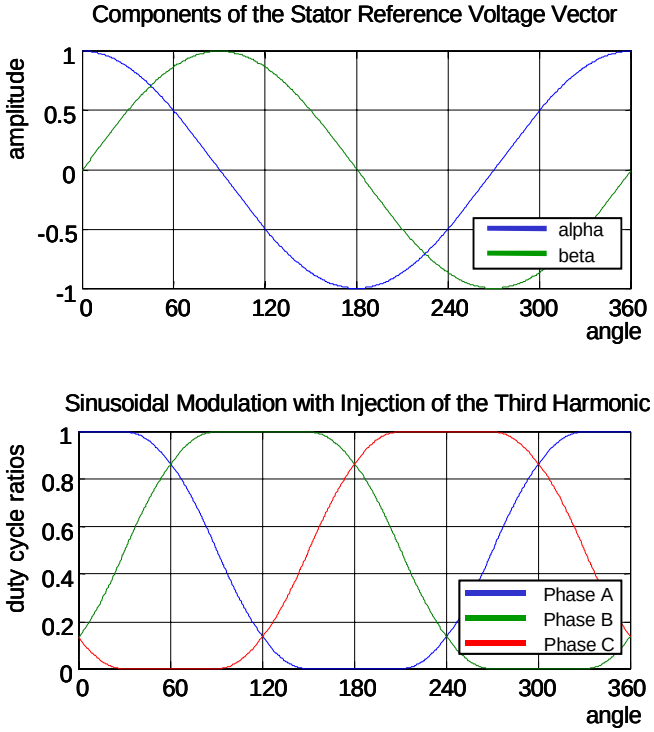

Figure 5-19. Sinusoidal Modulation with an Injection of the Third Harmonic

5.7.4 Returns

The function returns an integer value representing the Sector number in which the instantaneous stator reference voltage vector is located.

5.7.5 Range Issues

To provide an accurate calculation of the duty-cycle ratios, direct-a and quadrature-b components of the stator reference voltage vector must be considered as Q15 fractional numbers with their magnitude within the unit circle; i.e., the assumption $\sqrt{\alpha^2 + \beta^2} \leq 1$ must be met.

5.7.6 Special Issues

The **MCLIB_SvmSci** function is intended for periodical use; i.e., it might be called from a timer interrupt or a PWM updates interrupt. Referring to that, this function was programmed using assembler language with emphasis on maximizing the computational speed.

5.7.7 Design/Implementation

The **MCLIB_SvmSci** function is implemented as a function call.

Code Example 5-6. MCLIB_SvmSci

```
/* include function and data prototypes required by the module  */
#include "mclib.h"

/* main function call                                           */
void main (void)
{
  /* define data                                                */
  MC_2PhSyst         twoPhSystem;
  MC_3PhSyst         threePhSystem;
  int                sector;

  /* load input variable by data                                */
  twoPhSystem.alpha = FRAC16(1.0);    /* corresponds to 1.0     */
  twoPhSystem.beta  = 0;              /* corresponds to 0.0     */

  /* computed duty cycles ratios and sector number              */
  sector = MCLIB_SvmSci (&twoPhSystem, &threePhSystem);
}
```

5.7.8 See Also

MCLIB_SvmStd, **MCLIB_SvmU0n**, **MCLIB_SvmU7n**, **MCLIB_SvmAlt**, **MCLIB_PwmIct**

5.7.9 Performance

Table 5-12. Performance of the MCLIB_SvmSci function.

Code Size	130 words	
Data Size	7 words	
Execution Clocks	Min	147 cycles
	Max	178 cycles

5.8 MCLIB_ElimDcBusRip

5.8.1 Synopsis

#include ”mclib.h”
*void **MCLIB_SvmElimDcBusRip** (Frac16 invModIndex, Frac16 u_DcBusMsr, MC_2PhSyst *pInp_AlphaBeta, MC_2PhSyst *pOut_AlphaBeta);*

5.8.2 Arguments

invModIndex	in	Inverse Modulation Index; depends on the selected Modulation Technique
u_DcBusMsr	in	Measured DCBus voltage
*pInp_AlphaBeta	in	Pointer to structure with direct (alpha/α) and quadrature (beta/β) components of the stator voltage vector
*pOut_AlphaBeta	out	Pointer to structure with direct (alpha/α) and quadrature (beta/β) components of the stator voltage vector

5.8.3 Description

This function may be used in general motor control applications and provides the elimination of the voltage ripple on the DCBus of the power stage.

The function *svmElimDcBusRip* compensates an amplitude of the direct-α and the quadrature-β component of the stator reference voltage vector U_S for imperfections in the DCBus voltage. These imperfections are eliminated by the formulae shown in the following equations:

$$alpha^* = \begin{cases} \dfrac{invModIndex \cdot alpha}{u\_\text{DcBusMsr}/2} & \text{if } |invModIndex \cdot alpha| < \dfrac{u\_\text{DCBusMsr}}{2} \\ sign(alpha) \cdot 1.0 & \text{otherwise} \end{cases} \quad \textbf{(EQ 5-59.)}$$

$$beta^* = \begin{cases} \dfrac{invModIndex \cdot beta}{u\_\text{DcBusMsr}/2} & \text{if } |invModIndex \cdot beta| < \dfrac{u\_\text{DCBusMsr}}{2} \\ sign(beta) \cdot 1.0 & \text{otherwise} \end{cases} \quad \textbf{(EQ 5-60.)}$$

where the y = sign (x) function is defined as follows:

$$y = \begin{cases} 1.0 & \text{if } x \geq 0 \\ -1.0 & \text{otherwise} \end{cases} \quad \textbf{(EQ 5-61.)}$$

where x = alpha, beta are input duty-cycle ratios and alpha*, beta* are output duty-cycle ratios. Note that input duty-cycle ratios are referenced with the pointer **pInp_AlphaBeta* and output duty-cycle ratios are referenced with **pOut_AlphaBeta*.

Figure 5-20 graphically shows results of DCBus ripple elimination while compensating ripples of rectified voltage using 3-Phase Uncontrolled rectifier.

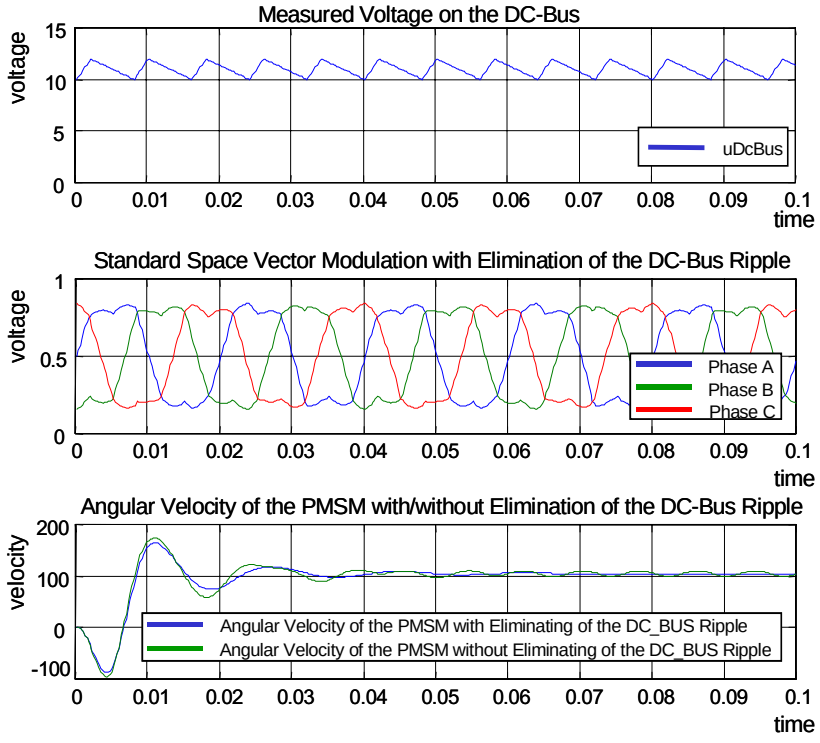

Figure 5-20. Results of the DCBus Voltage Ripple Elimination

5.8.4 Returns

The function returns an integer value representing the Sector number in which the instantaneous stator reference voltage vector is located.

5.8.5 Range Issues

To achieve proper functionality, arguments of this function must be within specified limits:

- invModIndex must be within a fractional range and positive: $0 \leq$ invModIndex ≤ 1 The value depends on the selected Modulation Technique; i.e., for Space Vector Modulation Techniques and Injection of the Third Harmonic, it is equal to 0,866025 and for the Inverse Clark Transformation, it is equal to 1,0.

- u_DcBusMsr must be within the fractional range and positive: $0 \leq$ u_DcBusMsr ≤ 1 that is equal to 0% - 100% of the maximum DCBus voltage.

- alpha, beta components of the stator reference voltage vector must be within a fractional range: $-u\_DcBusMsr/(2 \cdot invModIndex) \leq x \leq u\_DcBusMsr/(2 \cdot invModIndex)$, where x stands for alpha, beta. In the case where the inputs are out of the specified range, then the respective outputs alpha*, beta* will be saturated to their positive or negative maximal value, according to the sign of input components.

5.8.6 Special Issues

The **MCLIB_ElimDcBusRip** function is intended to be used periodically; e.g., called from within a timer interrupt period or the PWM updates interrupt.

5.8.7 Implementation

The **MCLIB_ElimDcBusRip** function is implemented as a function call.

Code Example 5-7. MCLIB_ElimDcBusRip

```
/* include function and data prototypes required by the module   */
#include "mclib.h"

/* main function call                                            */
void main (void)
{
  MC_2PhSyst inTwoPhSystem, outTwoPhSystem;
  Frac16     invModIndex;
  Frac16     dcBusMsr;

  /* load input variable by values                               */
  inTwoPhSystem.alpha= 16384;   /* equals to 0.5                 */
  inTwoPhSystem.beta = 0;       /* equals to 0.0                 */
  invModIndex = 28377;          /* equals to 0.866025            */
  dcBusMsr = 31129;             /* equals to 0.95                */
  /* eliminates voltage ripple on the DC-Bus                     */
  MCLIB_ElimDcBusRip (invModIndex,dcBusMsr,&inTwoPhSystem,&outTwoPhSystem);
}
```

5.8.8 Performance

Table 5-13. Performance of the MCLIB_ElimDcBusRip function.

Code Size	70 words	
Data Size	0 words	
Execution Clocks	Min	135 cycles
	Max	163 cycles

Section 6. RAMP

6.1 API Summary

Table 6-1. API Function Summary

Name	Arguments	Return	Description
MCLIB_RampGetValue	Frac32 incrementUp, Frac32 incrementDown, Frac32 *pActualValue, Frac32 *pRequestedValue	Frac32	Function generates the acceleration / decceleration ramp. The function computes the next value of acceleration / decceleration ramp related to the current acceleration / decceleration ramp direction. See **Section: 6.2 MCLIB_RampGetValue**

6.2 MCLIB_RampGetValue

6.2.1 Synopsis

#include ”mclib.h”
Frac32 ***MCLIB_RampGetValue*** *(Frac32 incrementUp, Frac32 incrementDown, const Frac32 *pActualValue, const Frac32 *pRequestedValue);*

6.2.2 Arguments

incrementUp	in	Increment step for direction up
incrementDown	in	Increment step for direction down
*pActualValue	in	Pointer to variable containing actual value
*pRequestedValue	in	Pointer to variable containing requested value

6.2.3 Description

The *rampGetValue* function performs a linear Ramp generation determined by input parameters as shown in **Figure 6-1**.

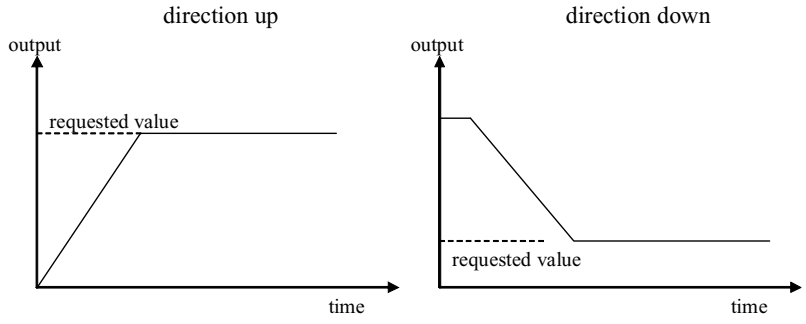

Figure 6-1. Ramp generation chart

6.2.4 Returns

If the *requestedValue* is greater than *actualValue,* the *rampGetValue* function returns *actualValue* + *incrementUp* until the maximum is reached, (maximum is *requestedValue*), at which point it will return *requestedValue*.

If the *requestedValue* is less than *actualValue,* the *rampGetValue* function returns *actualValue - incrementDown* until the minimum is reached, (minimum is *requestedValue*), at which point it will return *requestedValue*.

6.2.5 Implementation

The **MCLIB_RampGetValue** function is implemented as a function call.

Code Example 6-1. MCLIB_RampGetValue

```
/* include function and data prototypes required by the module  */
#include "mclib.h"

/* main function call                                           */
void main (void)
{
   /* define data                                               */
   Frac32 reqVal, actVal;

   /* load input variable by data                               */
   actVal = 5750;
   reqVal = 6000;

   MCLIB_RampGetValue(300, 100, &actVal, &reqVal);
   /* actVal will be 6000                                       */
}
```

6.2.6 Performance

Table 6-2. Performance of the MCLIB_RampGetValue function.

Code Size	30 words	
Data Size	0 words	
Execution Clocks	Min	55 cycles
	Max	55 cycles

Appendix A. References

1. DSP56800E 16-bit DSP Core Reference Manual; Motorola (DSP56800ERM/D)
2. DSP56F83x Family 16-Bit Digital Signal Processor Peripheral Manual; Motorola (DSP56F83xUM/D)

How to Reach Us:

USA/Europe/Locations not listed:
Freescale Semiconductor Literature Distribution
P.O. Box 5405, Denver, Colorado 80217
1-800-521-6274 or 480-768-2130

Japan:
Freescale Semiconductor Japan Ltd.
SPS, Technical Information Center
3-20-1, Minami-Azabu
Minato-ku
Tokyo 106-8573, Japan
81-3-3440-3569

Asia/Pacific:
Freescale Semiconductor H.K. Ltd.
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T. Hong Kong
852-26668334

Learn More:
For more information about Freescale Semiconductor products, please visit
http://www.freescale.com

UCANUM

freescale™ semiconductor