GMCLIB User's Guide

ARM® Cortex® M7

Contents

Chapter 1
Library

1.1 Introduction

1.1.1 Overview

This user's guide describes the General Motor Control Library (GMCLIB) for the family of ARM Cortex M7 core-based microcontrollers. This library contains optimized functions.

1.1.2 Data types

GMCLIB supports several data types: (un)signed integer, fractional , and accumulator. The integer data types are useful for general-purpose computation; they are familiar to the MPU and MCU programmers. The fractional data types enable powerful numeric and digital-signal-processing algorithms to be implemented. The accumulator data type is a combination of both; that means it has the integer and fractional portions.

The following list shows the integer types defined in the libraries:

- Unsigned 16-bit integer—<0 ; 65535> with the minimum resolution of 1

- Signed 16-bit integer—<-32768 ; 32767> with the minimum resolution of 1

- Unsigned 32-bit integer—<0 ; 4294967295> with the minimum resolution of 1

- Signed 32-bit integer—<-2147483648 ; 2147483647> with the minimum resolution of 1

The following list shows the fractional types defined in the libraries:

- Fixed-point 16-bit fractional—$<-1 ; 1 - 2^{-15}>$ with the minimum resolution of 2^{-15}

- Fixed-point 32-bit fractional—$<-1 ; 1 - 2^{-31}>$ with the minimum resolution of 2^{-31}

The following list shows the accumulator types defined in the libraries:

- Fixed-point 16-bit accumulator—$<-256.0 ; 256.0 - 2^{-7}>$ with the minimum resolution of 2^{-7}

- Fixed-point 32-bit accumulator—$<-65536.0 ; 65536.0 - 2^{-15}>$ with the minimum resolution of 2^{-15}

1.1.3 API definition

GMCLIB uses the types mentioned in the previous section. To enable simple usage of the algorithms, their names use set prefixes and postfixes to distinguish the functions' versions. See the following example:

```
f32Result = MLIB_Mac_F32lss(f32Accum, f16Mult1, f16Mult2);
```

where the function is compiled from four parts:

- MLIB—this is the library prefix

- Mac—the function name—Multiply-Accumulate

- F32—the function output type

- lss—the types of the function inputs; if all the inputs have the same type as the output, the inputs are not marked

The input and output types are described in the following table:

Table 1. Input/output types

Type	Output	Input
frac16_t	F16	s
frac32_t	F32	l
acc32_t	A32	a

1.1.4 Supported compilers

GMCLIB for the ARM Cortex M7 core is written in C language or assembly language with C-callable interface depending on the specific function. The library is built and tested using the following compilers:

- MCUXpresso IDE

- IAR Embedded Workbench

- Keil µVision

For the MCUXpresso IDE, the library is delivered in the *gmclib.a* file.

For the Kinetis Design Studio, the library is delivered in the *gmclib.a* file.

For the IAR Embedded Workbench, the library is delivered in the *gmclib.a* file.

For the Keil µVision, the library is delivered in the *gmclib.lib* file.

The interfaces to the algorithms included in this library are combined into a single public interface include file, *gmclib.h*. This is done to lower the number of files required to be included in your application.

1.1.5 Library configuration

GMCLIB for the ARM Cortex M7 core is written in C language or assembly language with C-callable interface depending on the specific function. Some functions from this library are inline type, which are compiled together with project using this library. The optimization level for inline function is usually defined by the specific compiler setting. It can cause an issue especially when high optimization level is set. Therefore the optimization level for all inline assembly written functions is defined by compiler pragmas using macros. The configuration header file *RTCESL_cfg.h* is located in: *specific library folder\MLIB\Include*. The optimization level can be changed by modifying the macro value for specific compiler. In case of any change the library functionality is not guaranteed.

1.1.6 Special issues

1. The equations describing the algorithms are symbolic. If there is positive 1, the number is the closest number to 1 that the resolution of the used fractional type allows. If there are maximum or minimum values mentioned, check the range allowed by the type of the particular function version.

2. The library functions that round the result (the API contains Rnd) round to nearest (half up).

3. This RTCESL requires the DSP extension for some saturation functions. If the core does not support the DSP extension feature the assembler code of the RTCESL will not be buildable. For example the core1 of the LPC55s69 has no DSP extension.

1.2 Library integration into project (MCUXpresso IDE)

This section provides a step-by-step guide on how to quickly and easily include GMCLIB into any MCUXpresso SDK example or new SDK project using MCUXpresso IDE. The SDK based project uses RTCESL from SDK package.

High-speed functions execution suppport

Some RT (or other) platforms contain high-speed functions execution support by relocating all functions from the default Flash memory location to the RAM location for much faster code access. The feature is important especially for devices with a slow Flash interface. This section shows how to turn the RAM optimization feature support on and off.

1. In the MCUXpresso SDK project name node or on the left-hand side, click Properties or select Project > Properties from the menu. A project properties dialog appears.

2. Expand the C/C++ Build node and select Settings. See Figure 1.

3. On the right-hand side, under the MCU C Compiler node, click the Preprocessor node. See Figure 1 .
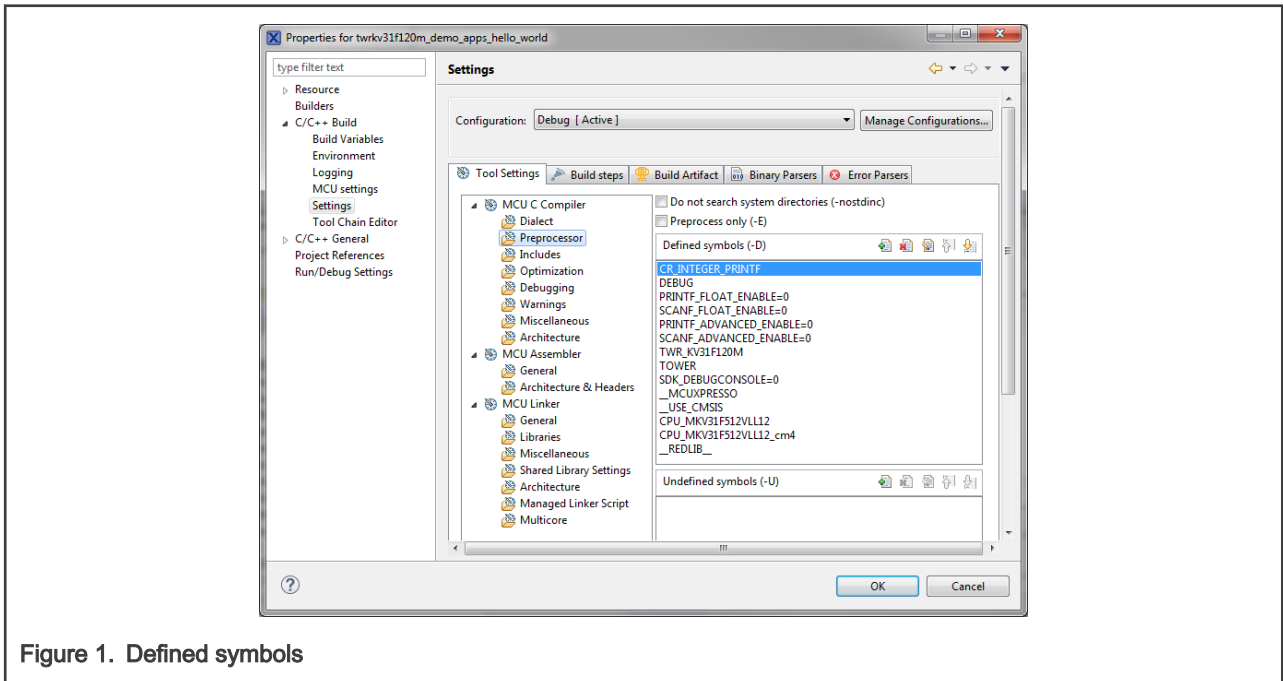
Figure 1. Defined symbols

4. On the right-hand side of the dialog, click the Add... icon located next to the Defined symbols (-D) title.

5. In the dialog that appears (see Figure 2), type the following:

   • **RAM_RELOCATION** — to turn the RAM optimization feature support on

   If the define is defined, all RTCEL functions are put to the RAM.
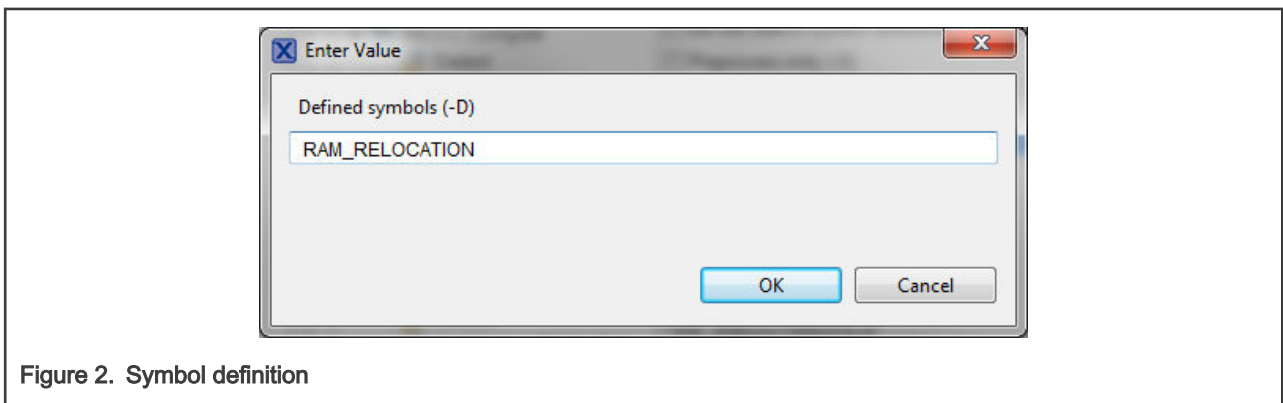
Figure 2. Symbol definition

6. Click OK in the dialog.

7. Click OK in the main dialog.

The RAM_RELOCATION macro places the `__RAMFUNC(RAM)` atribute in front of each function declaration.

Adding RTCESL component to project

The MCUXpresso SDK package is necessary to add any example or new project and RTCESL component. In case the package has not been downloaded go to mcuxpresso.nxp.com, build the final MCUXpresso SDK package for required board and download it.

After package is dowloaded, open the MCUXpresso IDE and drag&drop the SDK package in zip format to the Installed SDK window of the MCUXpresso IDE. After SDK package is dropped the mesage accepting window appears as can be show in following figure.
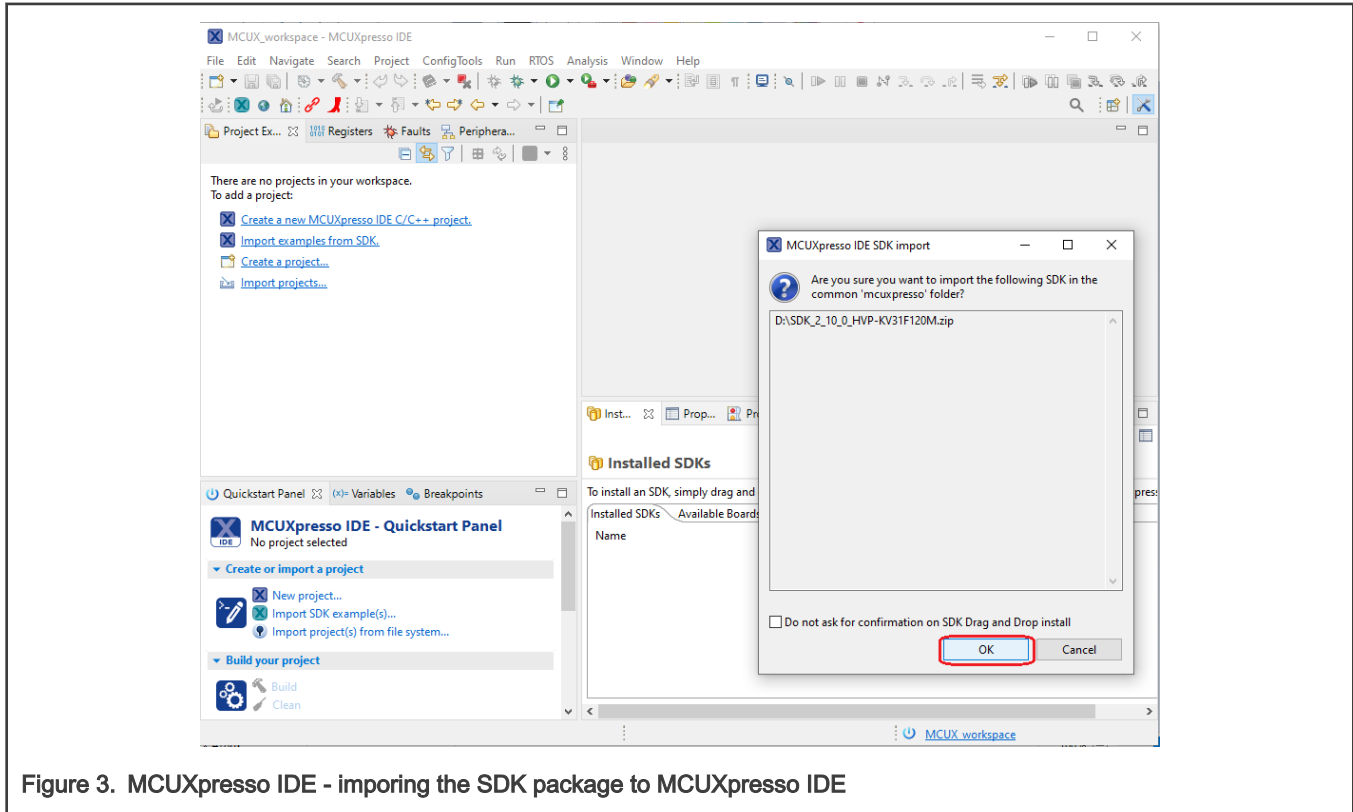
Figure 3. MCUXpresso IDE - imporing the SDK package to MCUXpresso IDE

Click OK to confirm the SDK package import. Find the Quickstart panel in left bottom part of the MCUXpresso IDE and click New project... item or Import SDK example(s)... to add rtcesl component to the project.
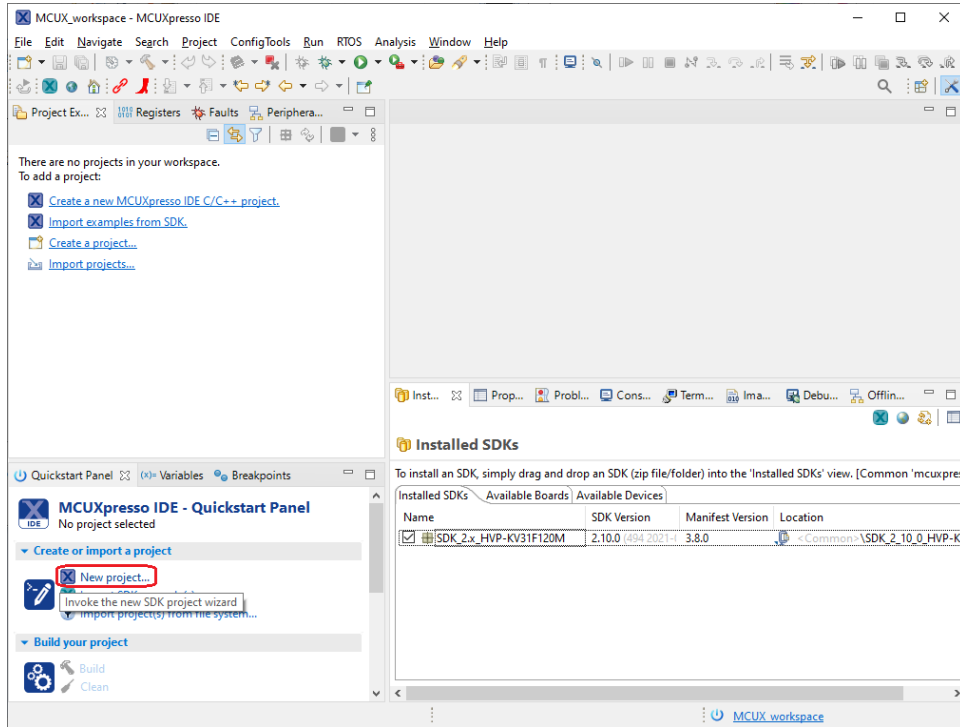
Figure 4. MCUXpresso IDE - create new project or Import SDK example(s)

Then select your board, and clik Next button.
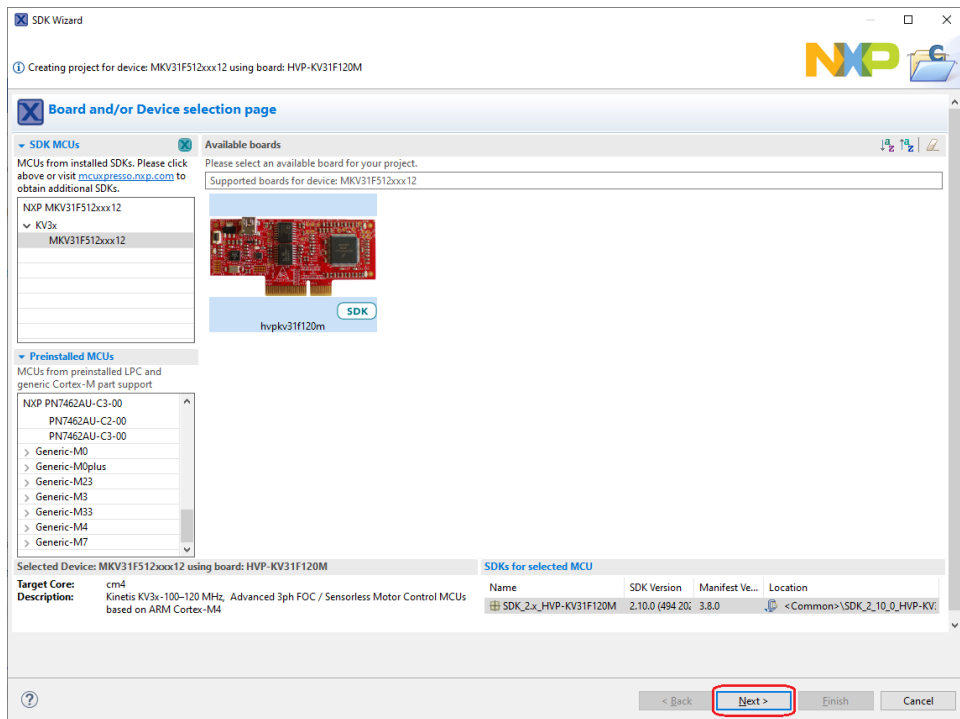
Figure 5. MCUXpresso IDE - selecting the board

Find the Middleware tab in the Components part of the window and click on the checkbox to be the rtcesl component ticked. Last step is to click the Finish button and wait for project creating with all RTCESL libraries and include paths.
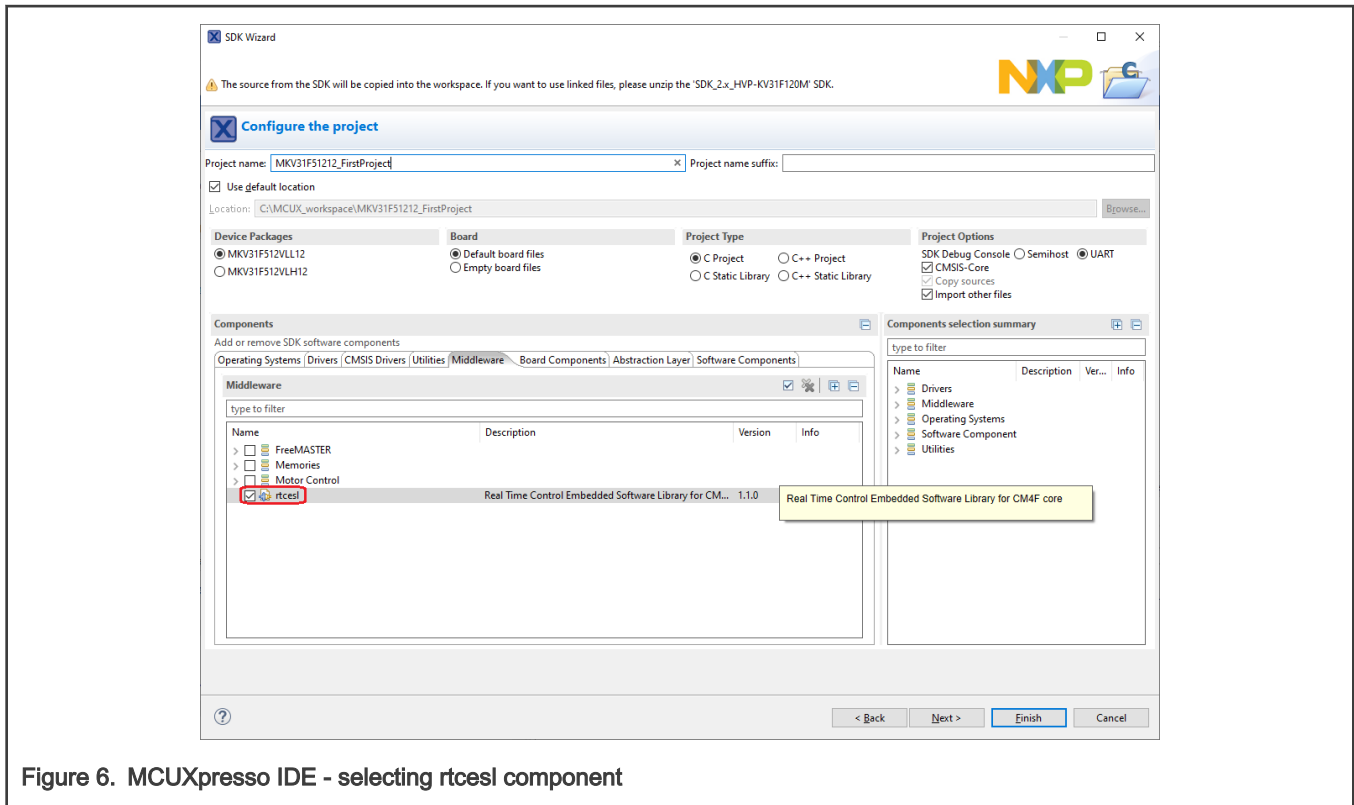
Figure 6. MCUXpresso IDE - selecting rtcesl component

Type the #include syntax into the code where you want to call the library functions. In the left-hand dialog, open the required .c file. After the file opens, include the following lines into the #include section:

```
#include "mlib.h"
#include "gflib.h"
#include "gmclib.h"
```

When you click the Build icon (hammer), the project is compiled without errors.

1.3 Library integration into project (Keil μVision)

This section provides a step-by-step guide on how to quickly and easily include GMCLIB into an empty project or any MCUXpresso SDK example or demo application projects using Keil μVision. This example uses the default installation path (C:\NXP\RTCESL\CM7_RTCESL_4.7_KEIL). If you have a different installation path, use that path instead. If any MCUXpresso SDK project is intended to use (for example hello_world project) go to Linking the files into the project chapter otherwise read next chapter.

NXP pack installation for new project (without MCUXpresso SDK)

This example uses the NXP MKV58F1M0xxx22 part, and the default installation path (C:\NXP\RTCESL\CM7_RTCESL_4.7_KEIL) is supposed. If the compiler has never been used to create any NXP MCU-based projects before, check whether the NXP MCU pack for the particular device is installed. Follow these steps:

1. Launch Keil μVision.

2. In the main menu, go to Project > Manage > Pack Installer….

3. In the left-hand dialog (under the Devices tab), expand the All Devices > Freescale (NXP) node.

4. Look for a line called "KVxx Series" and click it.

5. In the right-hand dialog (under the Packs tab), expand the Device Specific node.

6. Look for a node called "Keil::Kinetis_KVxx_DFP." If there are the Install or Update options, click the button to install/ update the package. See Figure 7.

7. When installed, the button has the "Up to date" title. Now close the Pack Installer.
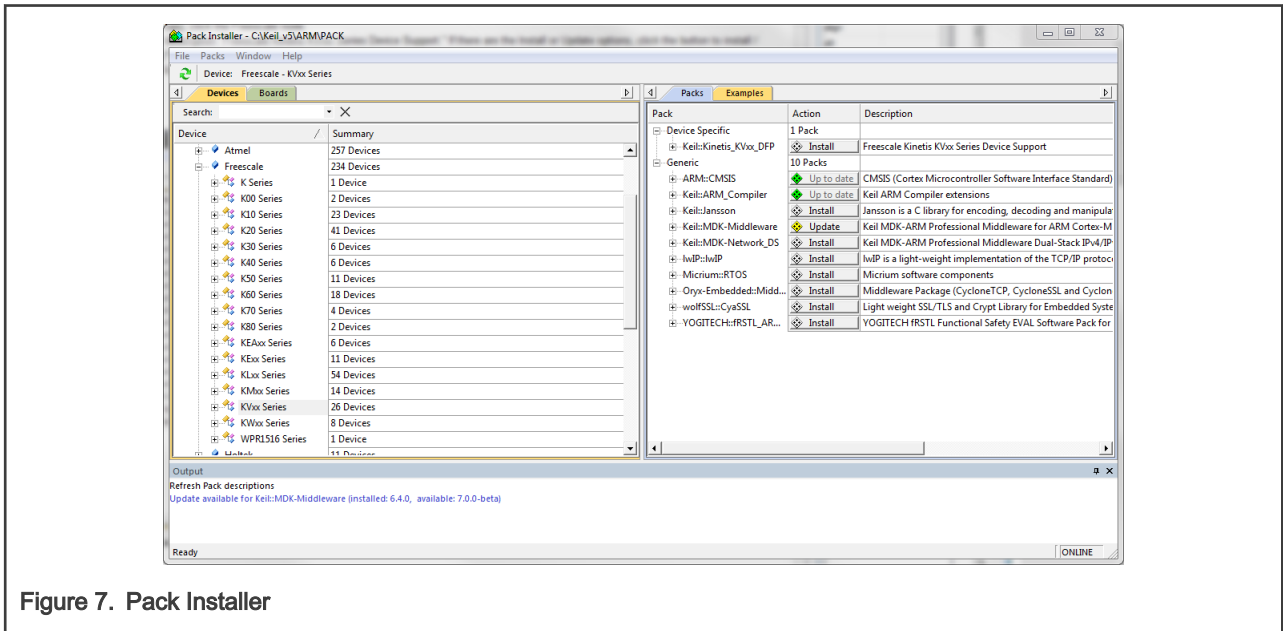
Figure 7. Pack Installer

New project (without MCUXpresso SDK)

To start working on an application, create a new project. If the project already exists and is opened, skip to the next section. Follow these steps to create a new project:

1. Launch Keil μVision.

2. In the main menu, select Project > New μVision Project…, and the Create New Project dialog appears.

3. Navigate to the folder where you want to create the project, for example C:\KeilProjects\MyProject01. Type the name of the project, for example MyProject01. Click Save. See Figure 8.
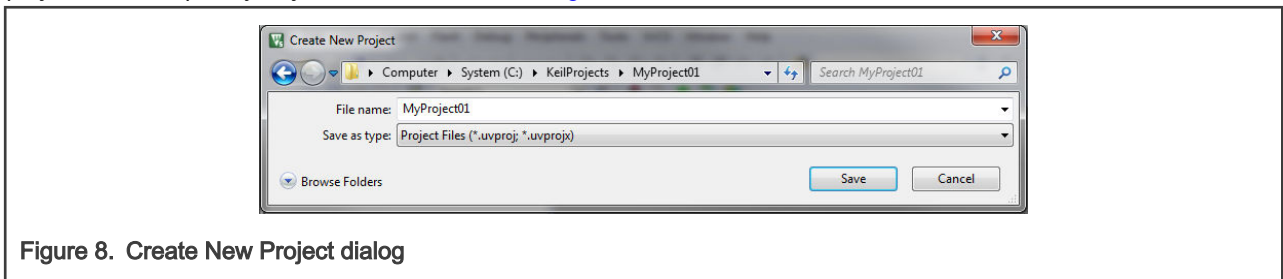
Figure 8. Create New Project dialog

4. In the next dialog, select the Software Packs in the very first box.

5. Type '' into the Search box, so that the device list is reduced to the devices.

6. Expand the node.

7. Click the MKV58F1M0xxx22 node, and then click OK. See Figure 9.
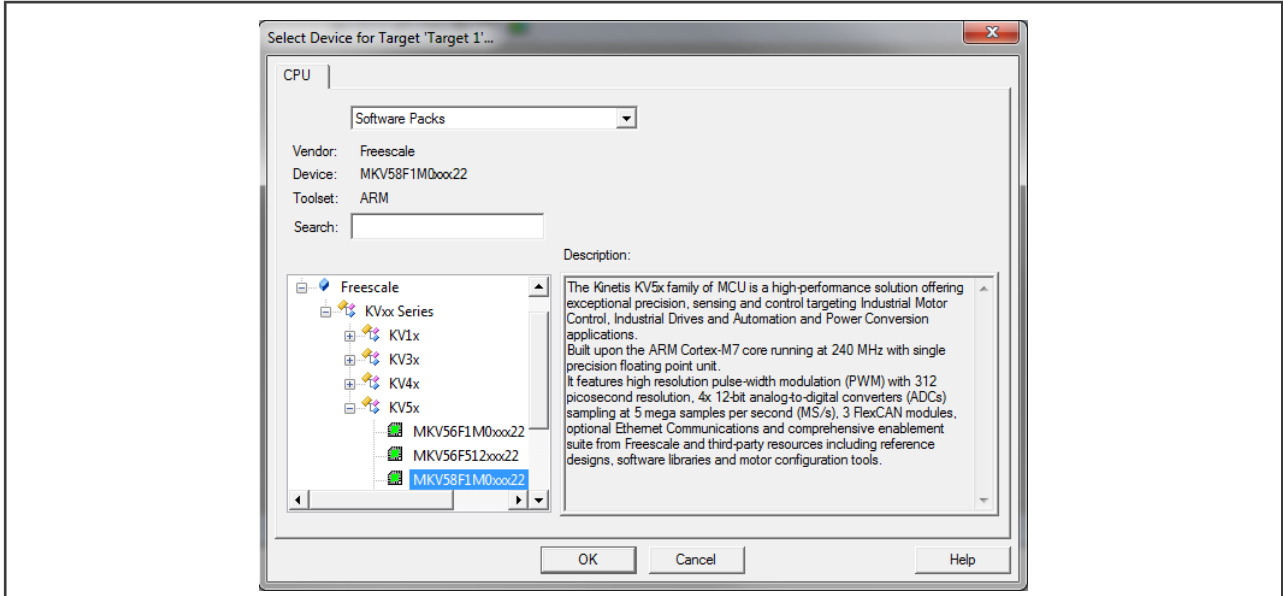
Figure 9. Select Device dialog

8. In the next dialog, expand the Device node, and tick the box next to the Startup node. See Figure 10.

9. Expand the CMSIS node, and tick the box next to the CORE node.
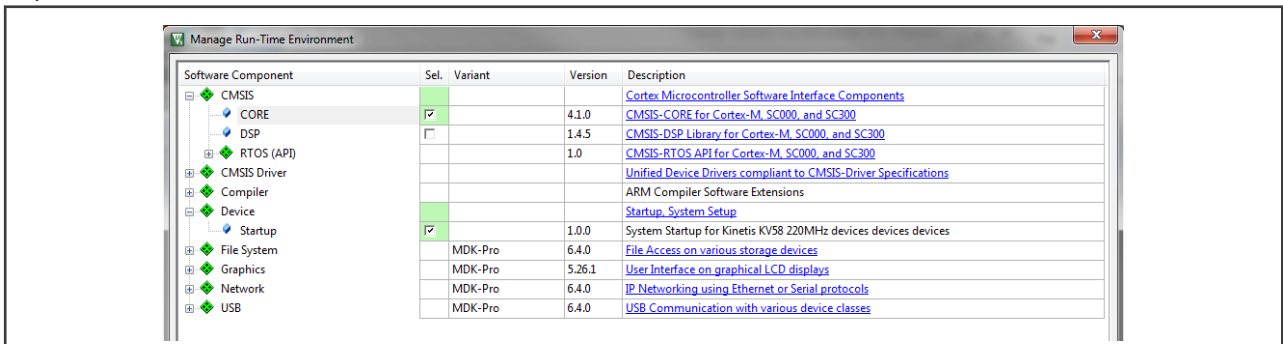
Figure 10. Manage Run-Time Environment dialog

10. Click OK, and a new project is created. The new project is now visible in the left-hand part of Keil μVision. See Figure 11.
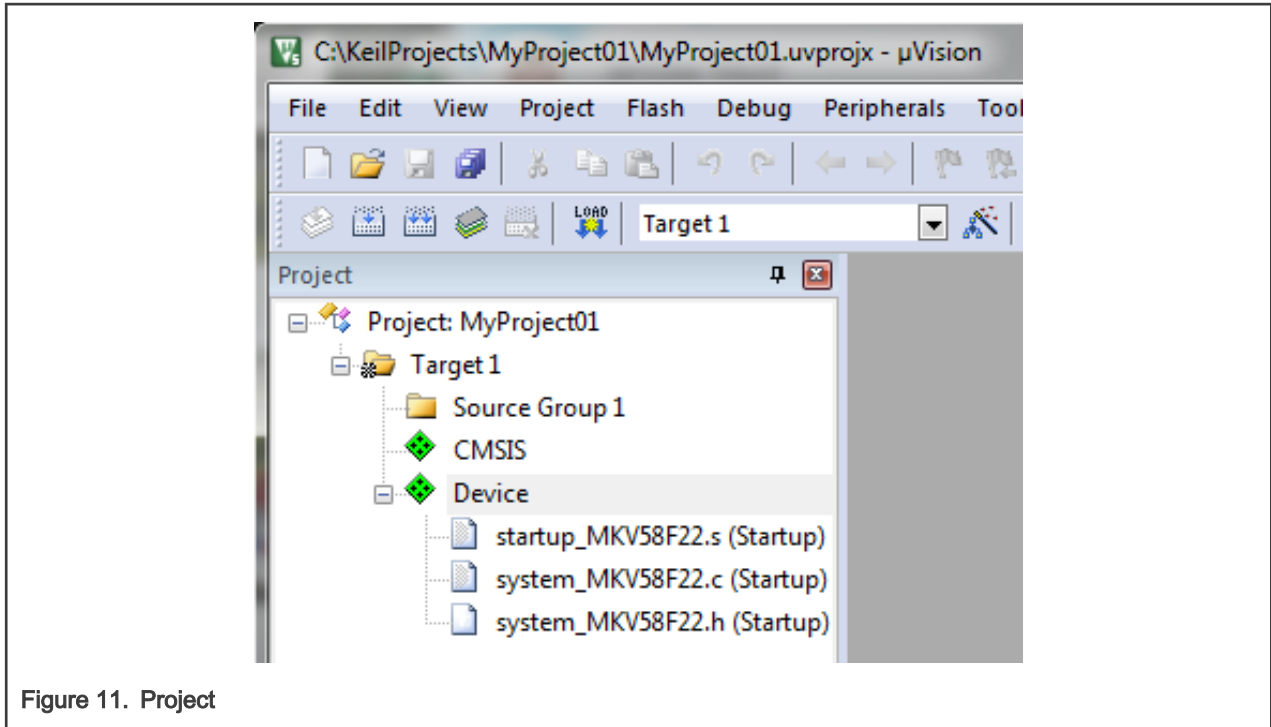
Figure 11. Project

11. In the main menu, go to Project > Options for Target 'Target1'…, and a dialog appears.

12. Select the Target tab.

13. Select Not Used in the Floating Point Hardware option. See Figure 11.

Figure 12. FPU

High-speed functions execution support

Some RT (or other) platforms contain high-speed functions execution support by relocating all functions from the default Flash memory location to the RAM location for much faster code access. The feature is important especially for devices with a slow Flash interface. This section shows how to turn the RAM optimization feature support on and off.

1. In the main menu, go to Project > Options for Target 'Target1'…, and a dialog appears.

2. Select the C/C++ tab. See #unique_18.

3. In the Include Preprocessor Symbols text box, type the following:

- **RAM_RELOCATION** — to turn the RAM optimization feature support on

If the define is defined, all RTCEL functions are put to the RAM.
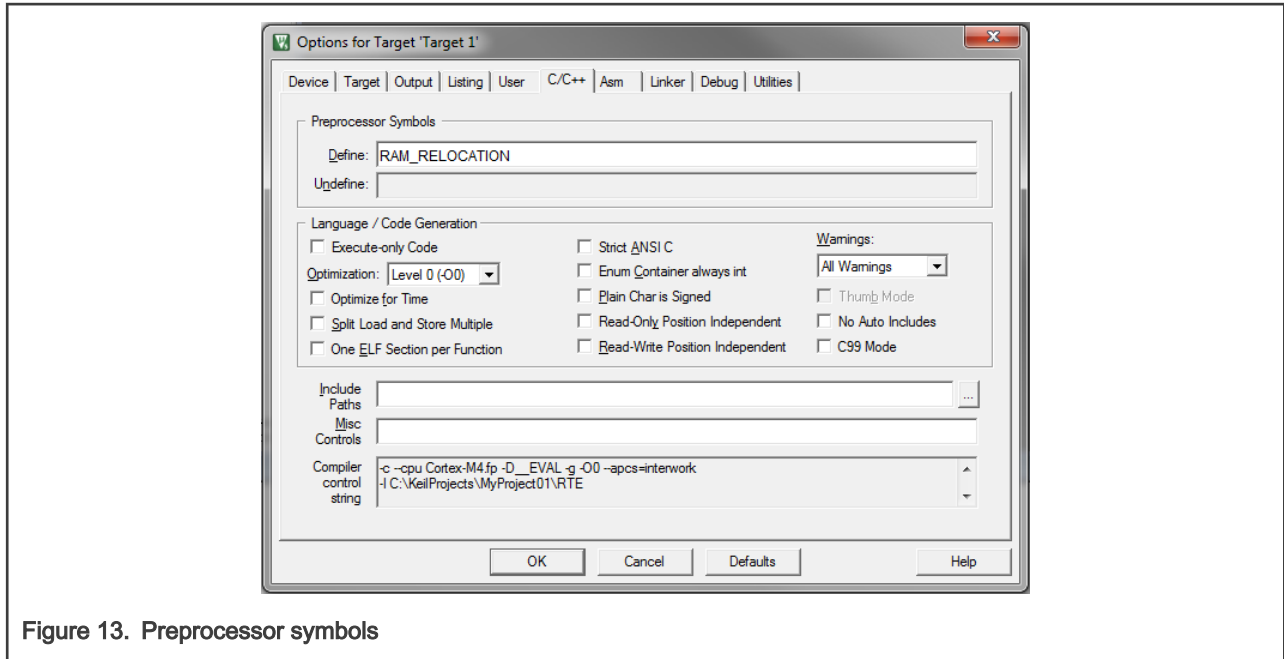
Figure 13. Preprocessor symbols

4. Click OK in the main dialog.

The RAM_RELOCATION macro places the `__attribute__ ((section ("ram")))` atribute in front of each function declaration.

Linking the files into the project

GMCLIB requires MLIB and GFLIB to be included too. The following steps show how to include all dependent modules.

To include the library files in the project, create groups and add them.

1. Right-click the Target 1 node in the left-hand part of the Project tree, and select Add Group… from the menu. A new group with the name New Group is added.

2. Click the newly created group, and press F2 to rename it to RTCESL.

3. Right-click the RTCESL node, and select Add Existing Files to Group 'RTCESL'… from the menu.

4. Navigate into the library installation folder C:\NXP\RTCESL\CM7_RTCESL_4.7_KEIL\MLIB\Include, and select the *mlib.h* file. If the file does not appear, set the Files of type filter to Text file. Click Add. See Figure 14.
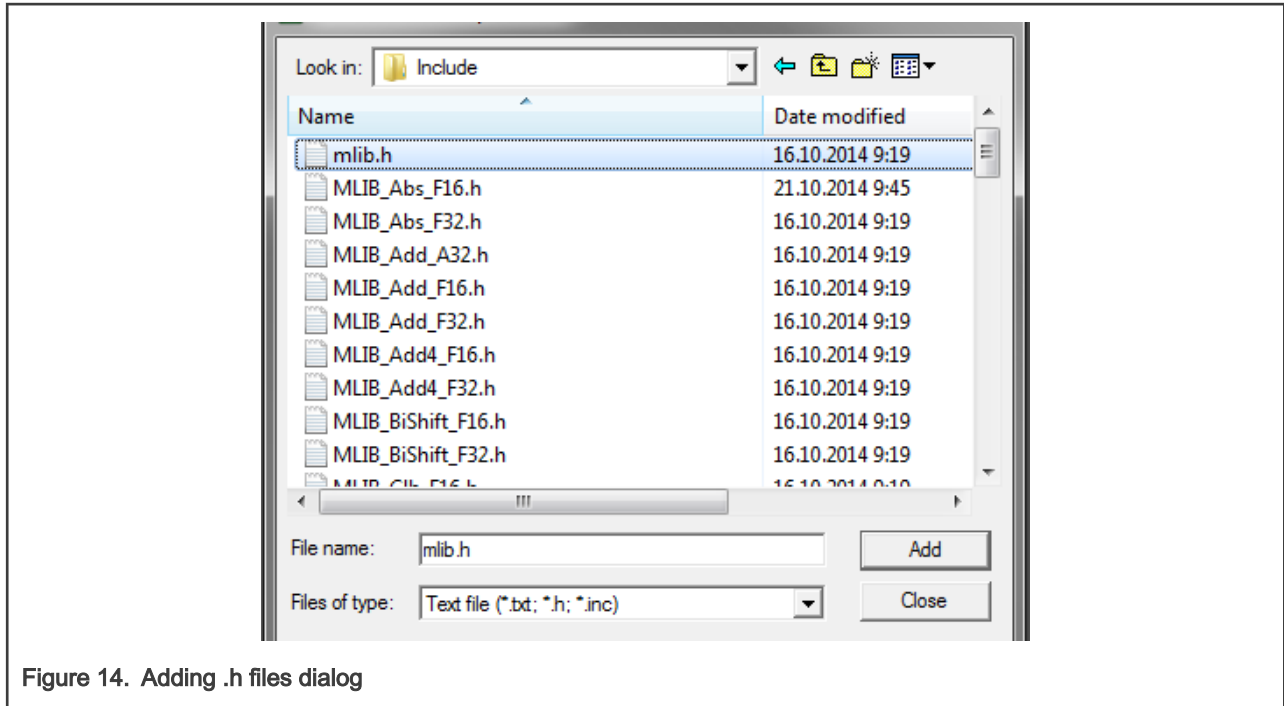
Figure 14. Adding .h files dialog

5. Navigate to the parent folder C:\NXP\RTCESL\CM7_RTCESL_4.7_KEIL\MLIB, and select the *mlib.lib* file. If the file does not appear, set the Files of type filter to Library file. Click Add. See Figure 15.
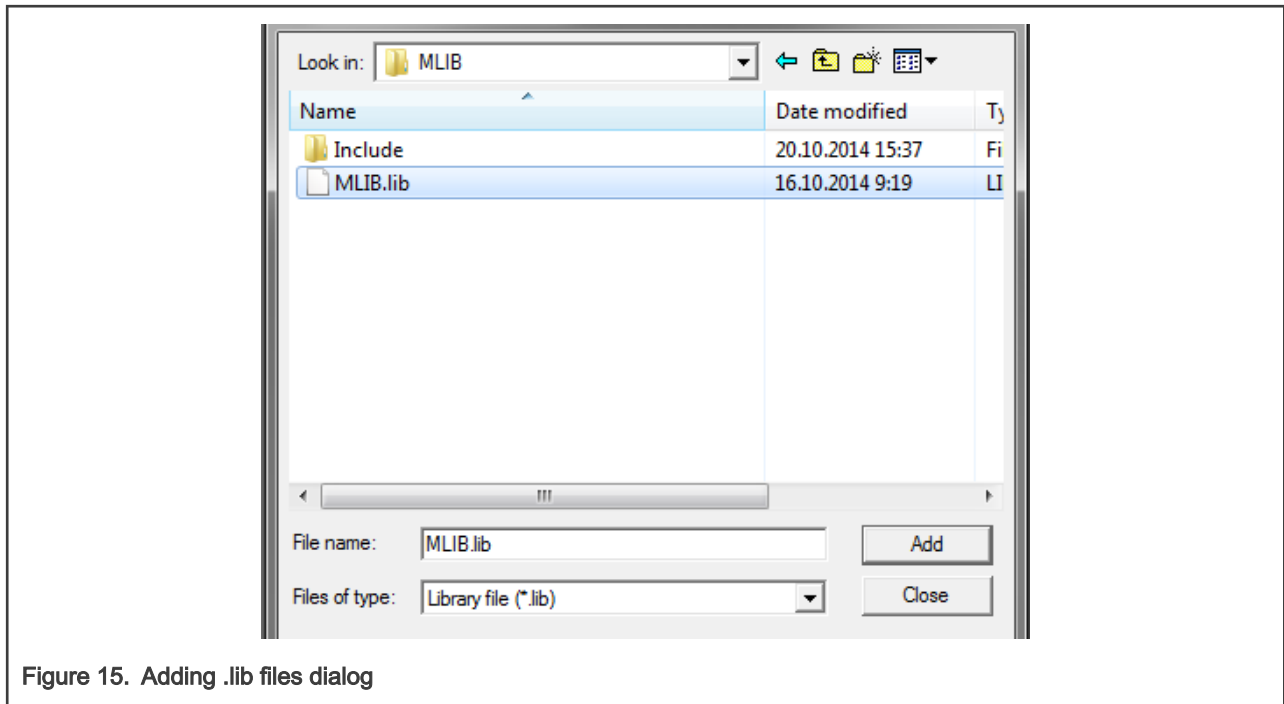
Figure 15. Adding .lib files dialog

6. Navigate into the library installation folder C:\NXP\RTCESL\CM7_RTCESL_4.7_KEIL\GFLIB\Include, and select the *gflib.h* file. If the file does not appear, set the Files of type filter to Text file. Click Add.

7. Navigate to the parent folder C:\NXP\RTCESL\CM7_RTCESL_4.7_KEIL\GFLIB, and select the *gflib.lib* file. If the file does not appear, set the Files of type filter to Library file. Click Add.

8. Navigate into the library installation folder C:\NXP\RTCESL\CM7_RTCESL_4.7_KEIL\GMCLIB\Include, and select the *gmclib.h* file. If the file does not appear, set the Files of type filter to Text file. Click Add.

9. Navigate to the parent folder C:\NXP\RTCESL\CM7_RTCESL_4.7_KEIL\GMCLIB, and select the *gmclib.lib* file. If the file does not appear, set the Files of type filter to Library file. Click Add.

10. Now, all necessary files are in the project tree; see Figure 16. Click Close.
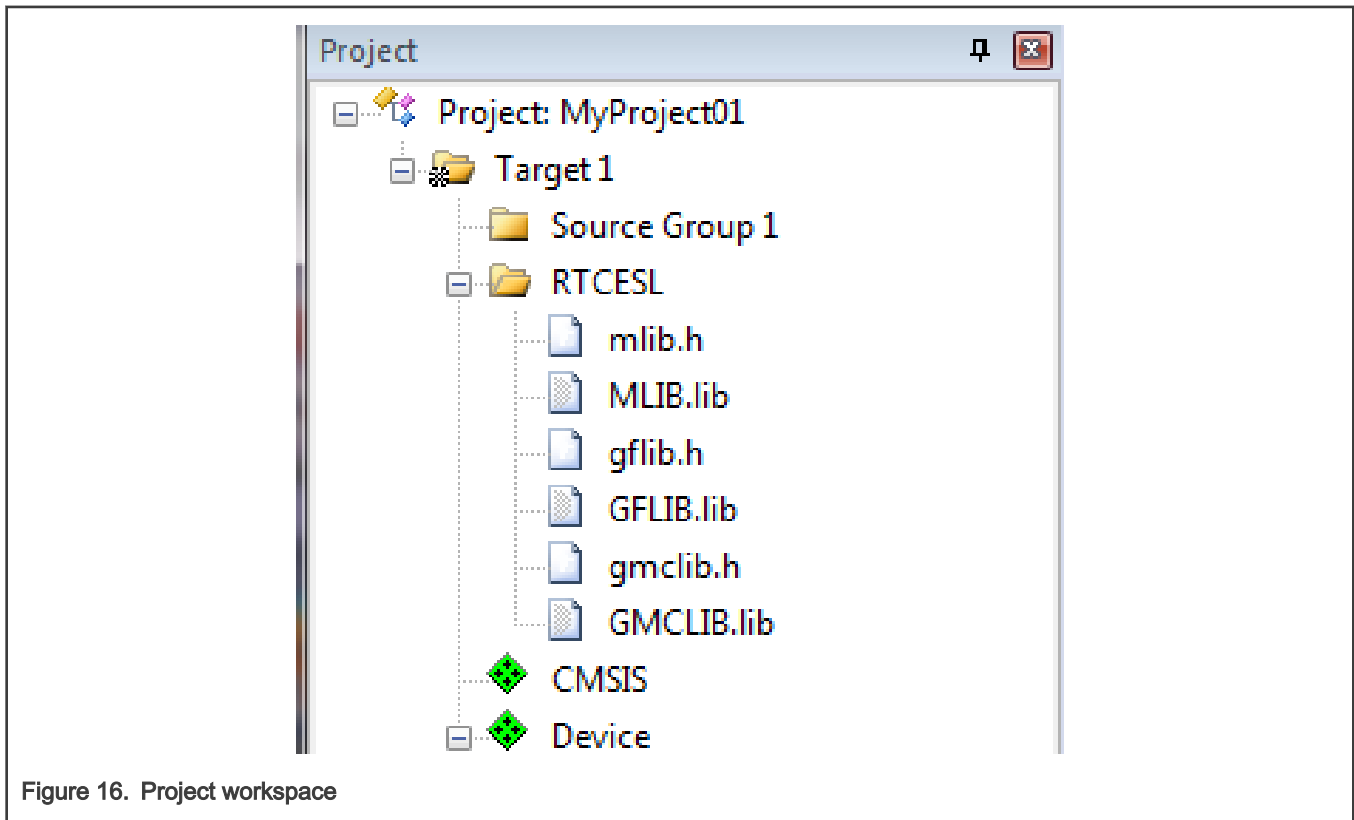
Figure 16. Project workspace

Library path setup

The following steps show the inclusion of all dependent modules.

1. In the main menu, go to Project > Options for Target 'Target1'…, and a dialog appears.

2. Select the C/C++ tab. See Figure 17.

3. In the Include Paths text box, type the following paths (if there are more paths, they must be separated by ';') or add them by clicking the … button next to the text box:

   • "C:\NXP\RTCESL\CM7_RTCESL_4.7_KEIL\MLIB\Include"

   • "C:\NXP\RTCESL\CM7_RTCESL_4.7_KEIL\GFLIB\Include"

   • "C:\NXP\RTCESL\CM7_RTCESL_4.7_KEIL\GMCLIB\Include"

4. Click OK.

5. Click OK in the main dialog.
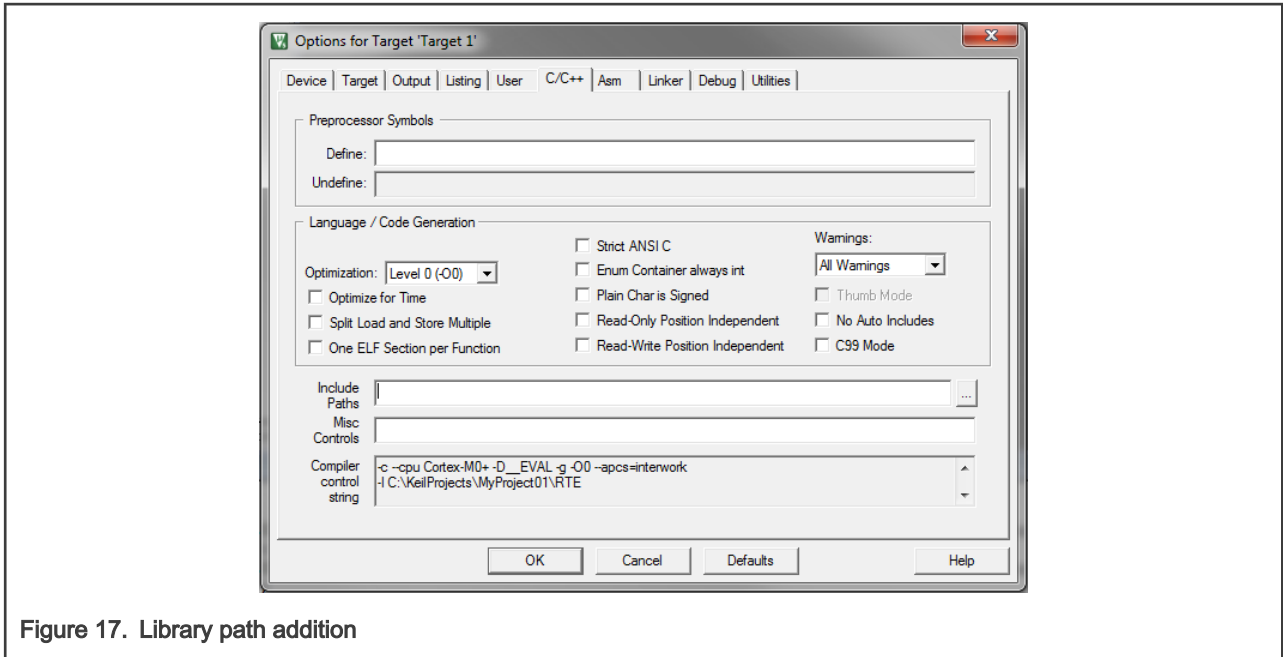
Figure 17. Library path addition

Type the #include syntax into the code. Include the library into a source file. In the new project, it is necessary to create a source file:

1. Right-click the Source Group 1 node, and Add New Item to Group 'Source Group 1'… from the menu.

2. Select the C File (.c) option, and type a name of the file into the Name box, for example '*main.c*'. See Figure 18.
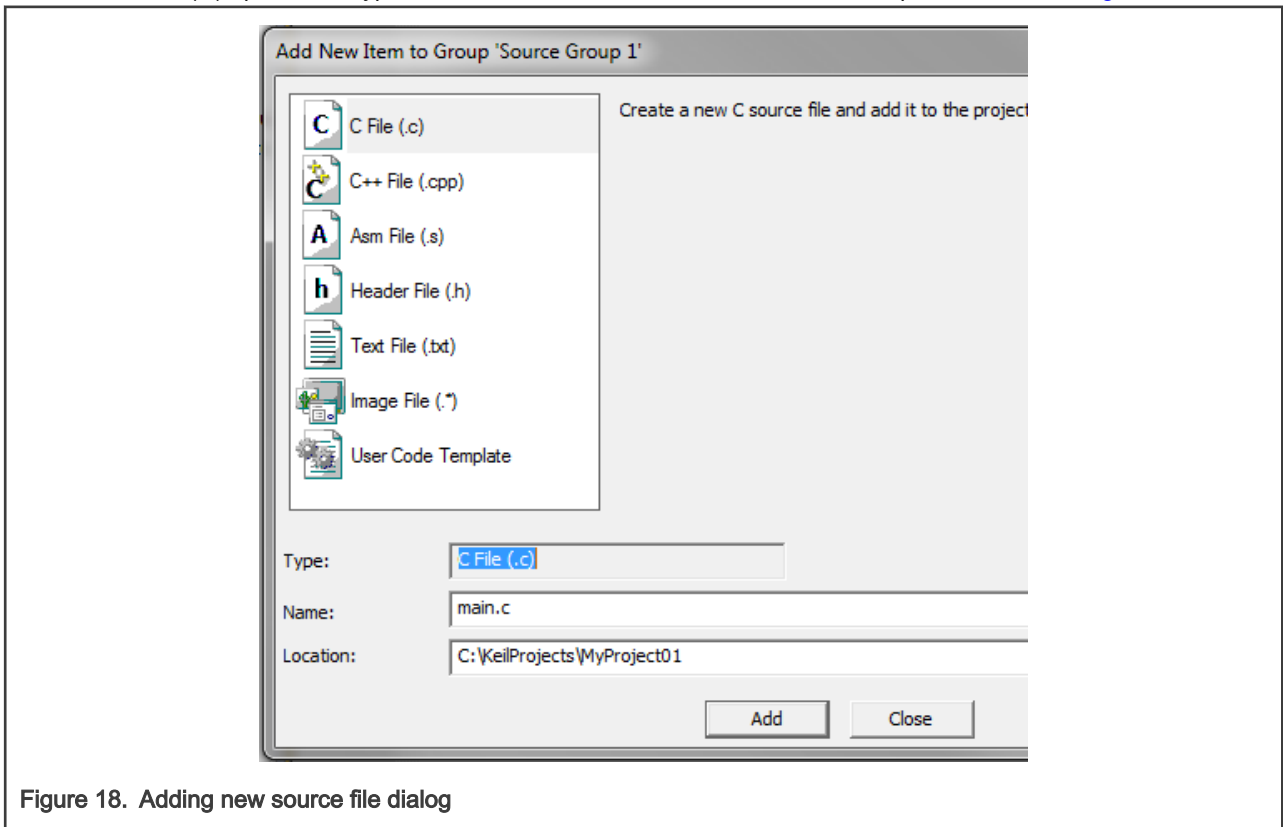
Figure 18. Adding new source file dialog

3. Click Add, and a new source file is created and opened up.

4. In the opened source file, include the following lines into the #include section, and create a main function:

```
#include "mlib.h"
#include "gflib.h"
#include "gmclib.h"

int main(void)
{
  while(1);
}
```

When you click the Build (F7) icon, the project will be compiled without errors.

1.4 Library integration into project (IAR Embedded Workbench)

This section provides a step-by-step guide on how to quickly and easily include the GMCLIB into an empty project or any MCUXpresso SDK example or demo application projects using IAR Embedded Workbench. This example uses the default installation path (C:\NXP\RTCESL\CM7_RTCESL_4.7_IAR). If you have a different installation path, use that path instead. If any MCUXpresso SDK project is intended to use (for example hello_world project) go to Linking the files into the project chapter otherwise read next chapter.

New project (without MCUXpresso SDK)

This example uses the NXP MKV58F1M0xxx22 part, and the default installation path (C:\NXP\RTCESL\CM7_RTCESL_4.7_IAR) is supposed. To start working on an application, create a new project. If the project already exists and is opened, skip to the next section. Perform these steps to create a new project:

1. Launch IAR Embedded Workbench.

2. In the main menu, select Project > Create New Project… so that the "Create New Project" dialog appears. See Figure 19.
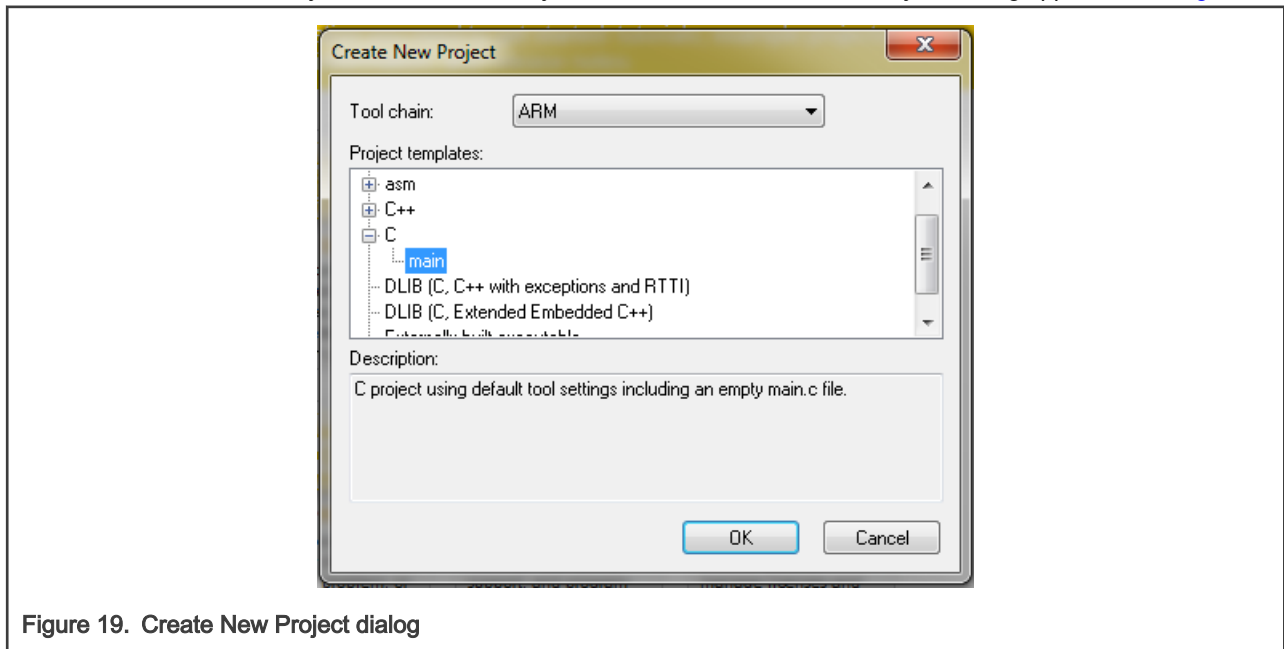
Figure 19. Create New Project dialog

3. Expand the C node in the tree, and select the "main" node. Click OK.

4. Navigate to the folder where you want to create the project, for example, C:\IARProjects\MyProject01. Type the name of the project, for example, MyProject01. Click Save, and a new project is created. The new project is now visible in the left-hand part of IAR Embedded Workbench. See Figure 20.
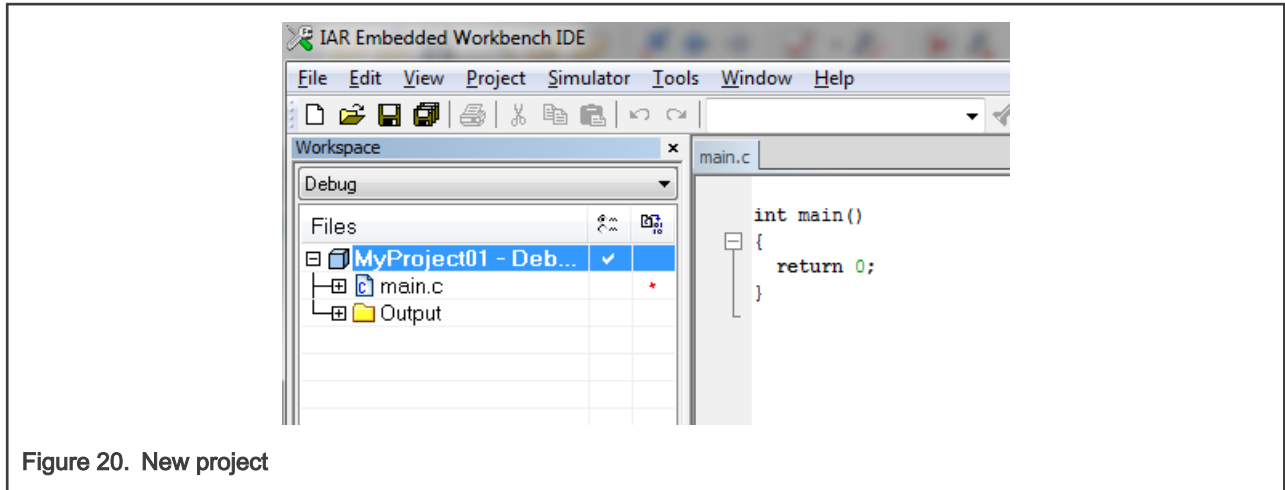
Figure 20. New project

5. In the main menu, go to Project > Options…, and a dialog appears.

6. In the Target tab, select the Device option, and click the button next to the dialog to select the MCU. In this example, select NXP > KV5x > NXP MKV58F1M0xxx22. Select None in the FPU option. Click OK. See Figure 21.
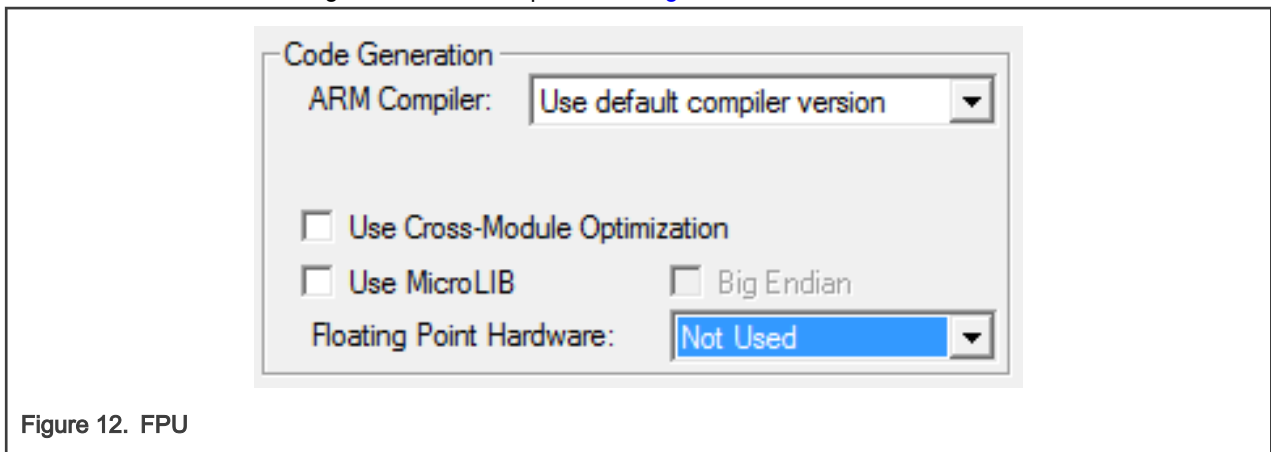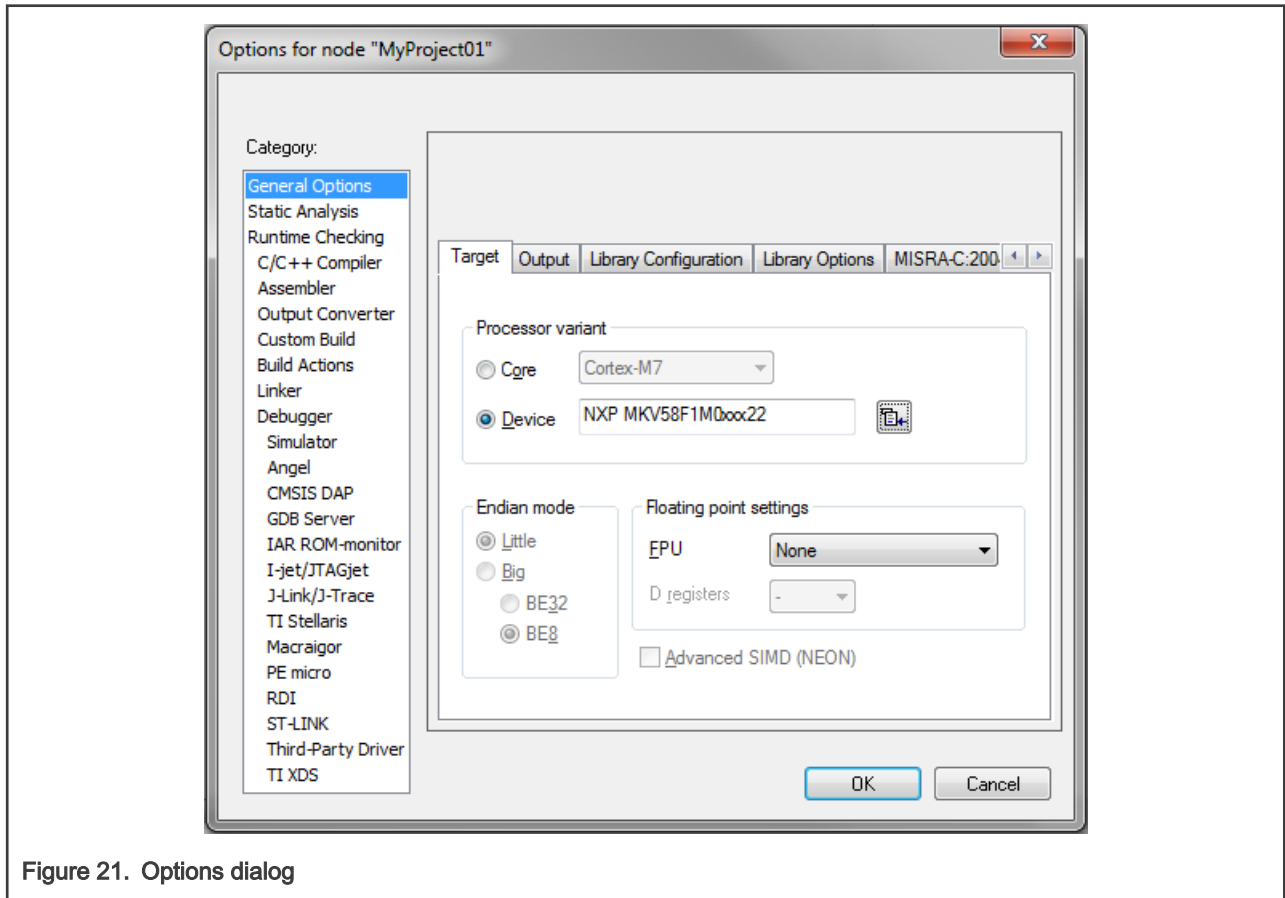

Figure 21. Options dialog

High-speed functions execution suppport

Some RT (or other) platforms contain high-speed functions execution support by relocating all functions from the default Flash memory location to the RAM location for much faster code access. The feature is important especially for devices with a slow Flash interface. This section shows how to turn the RAM optimization feature support on and off.

1. In the main menu, go to Project > Options…, and a dialog appears.

2. In the left-hand side column, select C/C++ Compiler.

3. In the right-hand side of the dialog, click the Preprocessor tab (it can be hidden on the right; use the arrow icons for navigation).

4. In the text box (in Defined symbols: (one per line)), type the following (See Figure 22):

   • **RAM_RELOCATION** — to turn the RAM optimization feature support on

If the define is defined, all RTCEL functions are put to the RAM.
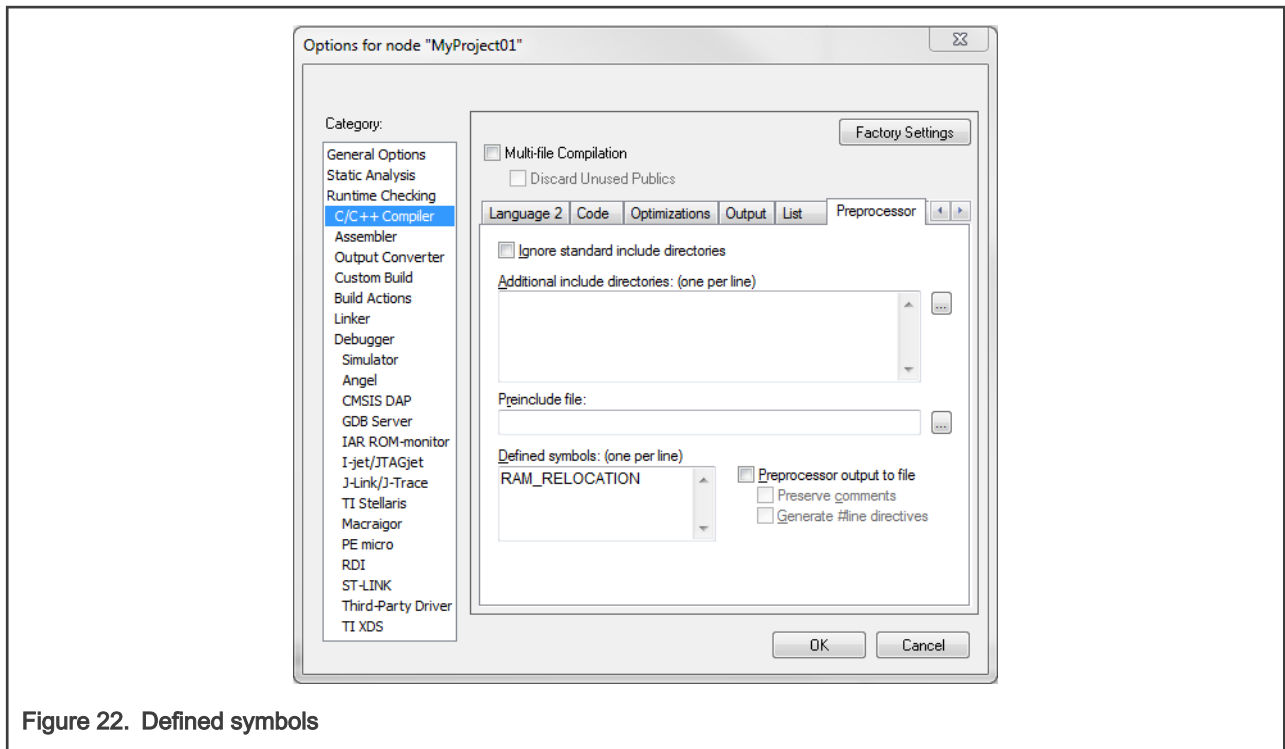
Figure 22. Defined symbols

5. Click OK in the main dialog.

The RAM_RELOCATION macro places the `__ramfunc` atribute in front of each function declaration.

Library path variable

To make the library integration easier, create a variable that will hold the information about the library path.

1. In the main menu, go to Tools > Configure Custom Argument Variables…, and a dialog appears.

2. Click the New Group button, and another dialog appears. In this dialog, type the name of the group PATH, and click OK. See Figure 23.
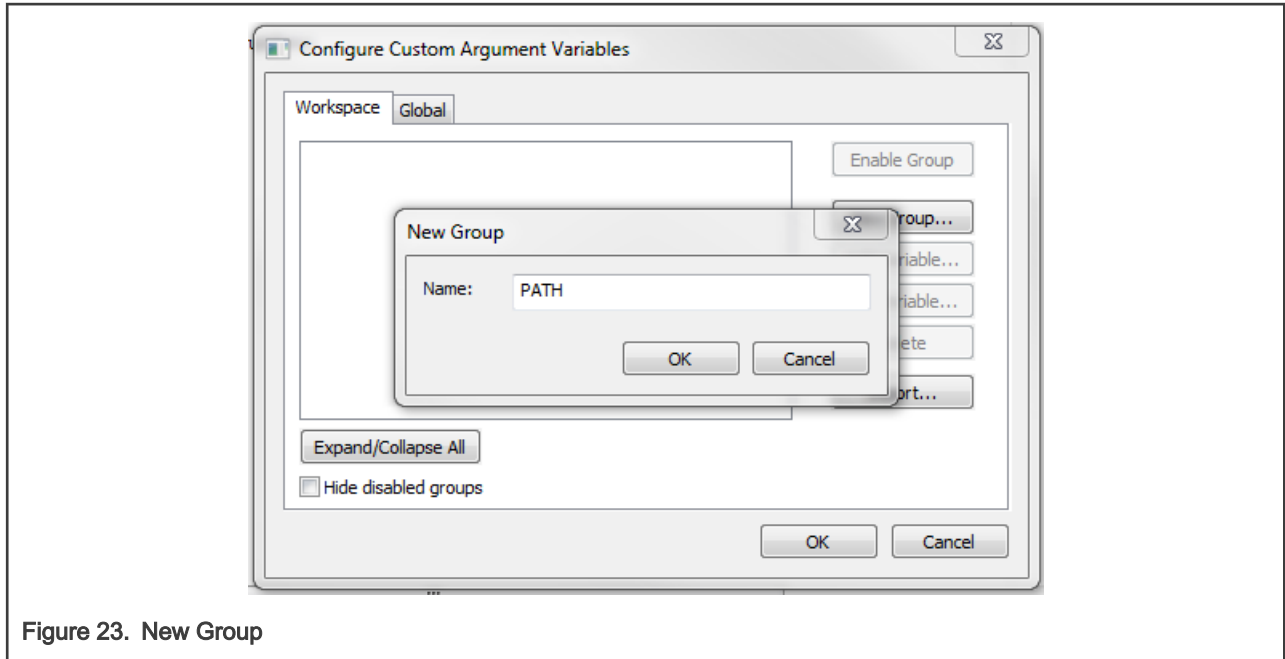
Figure 23. New Group

3. Click on the newly created group, and click the Add Variable button. A dialog appears.

4. Type this name: RTCESL_LOC

5. To set up the value, look for the library by clicking the '…' button, or just type the installation path into the box: C:\NXP\RTCESL\CM7_RTCESL_4.7_IAR. Click OK.

6. In the main dialog, click OK. See Figure 24.
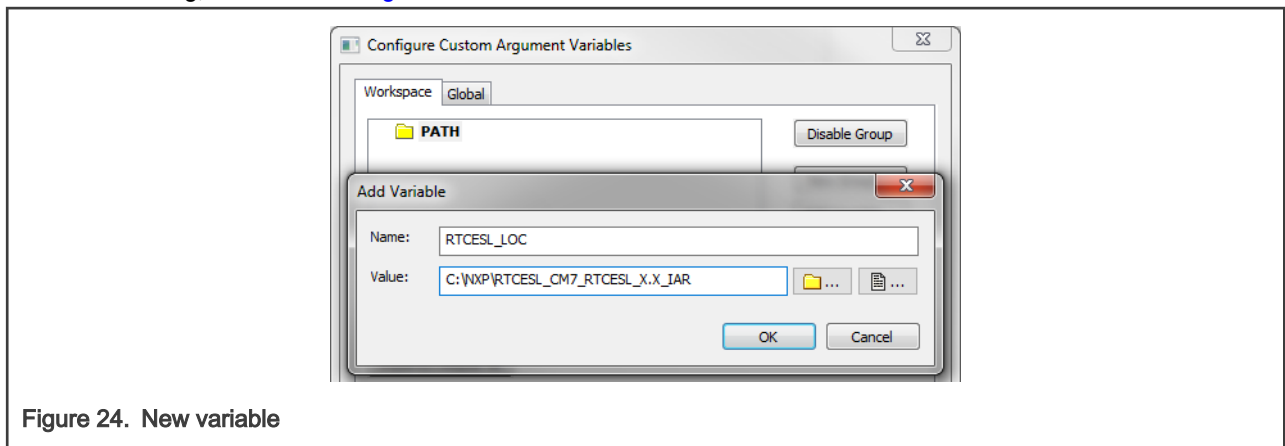
Figure 24. New variable

Linking the files into the project

GMCLIB requires MLIB and GFLIB to be included too. The following steps show the inclusion of all dependent modules.

To include the library files into the project, create groups and add them.

1. Go to the main menu Project > Add Group…

2. Type RTCESL, and click OK.

3. Click on the newly created node RTCESL, go to Project > Add Group…, and create a MLIB subgroup.

4. Click on the newly created node MLIB, and go to the main menu Project > Add Files… See Figure 26.

5. Navigate into the library installation folder C:\NXP\RTCESL\CM7_RTCESL_4.7_IAR\MLIB\Include, and select the *mlib.h* file. (If the file does not appear, set the file-type filter to Source Files.) Click Open. See Figure 25.

6. Navigate into the library installation folder C:\NXP\RTCESL\CM7_RTCESL_4.7_IAR\MLIB, and select the *mlib.a* file. If the file does not appear, set the file-type filter to Library / Object files. Click Open.
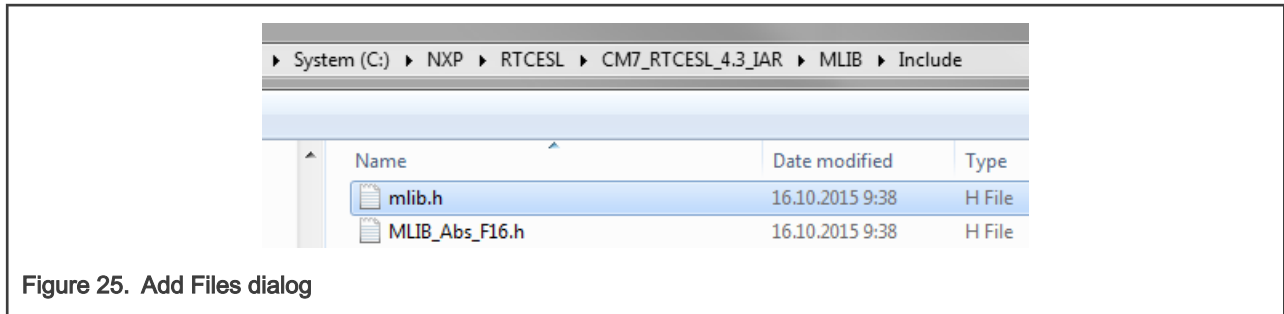
Figure 25. Add Files dialog

7. Click on the RTCESL node, go to Project > Add Group…, and create a GFLIB subgroup.

8. Click on the newly created node GFLIB, and go to the main menu Project > Add Files….

9. Navigate into the library installation folder C:\NXP\RTCESL\CM7_RTCESL_4.7_IAR\GFLIB\Include, and select the *gflib.h* file. (If the file does not appear, set the file-type filter to Source Files.) Click Open.

10. Navigate into the library installation folder C:\NXP\RTCESL\CM7_RTCESL_4.7_IAR\GFLIB, and select the *gflib.a* file. If the file does not appear, set the file-type filter to Library / Object files. Click Open.

11. Click on the RTCESL node, go to Project > Add Group…, and create a GMCLIB subgroup.

12. Click on the newly created node GMCLIB, and go to the main menu Project > Add Files….

13. Navigate into the library installation folder C:\NXP\RTCESL\CM7_RTCESL_4.7_IAR\GMCLIB\Include, and select the *gmclib.h* file. If the file does not appear, set the file-type filter to Source Files. Click Open.

14. Navigate into the library installation folder C:\NXP\RTCESL\CM7_RTCESL_4.7_IAR\GMCLIB, and select the *gmclib.a* file. If the file does not appear, set the file-type filter to Library / Object files. Click Open.

15. Now you will see the files added in the workspace. See Figure 26.
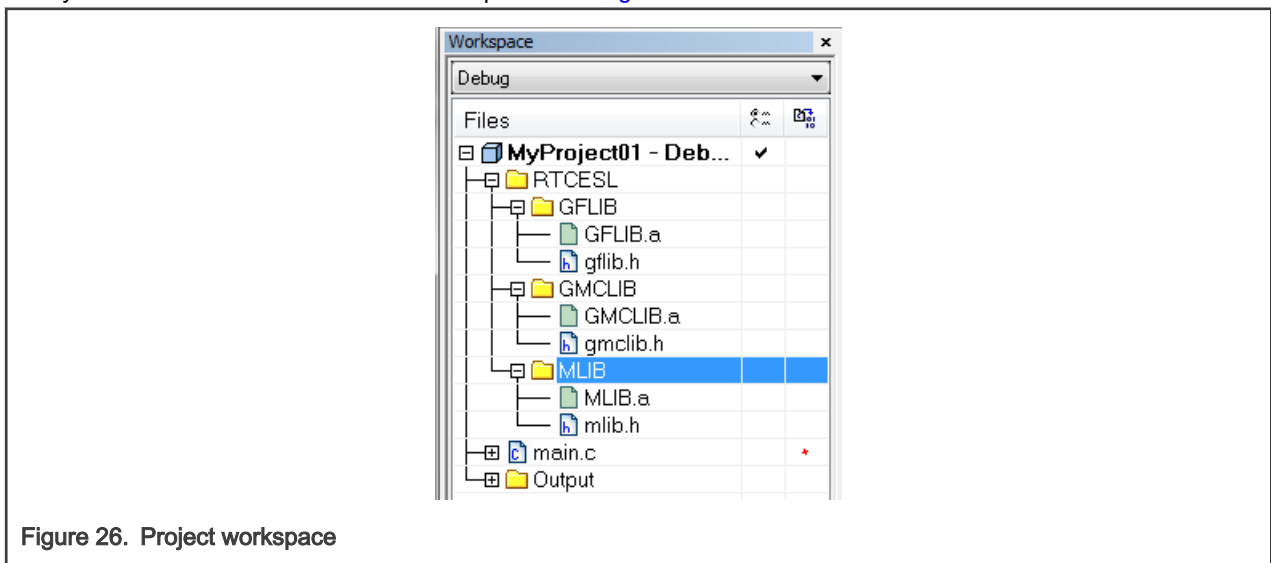
Figure 26. Project workspace

Library path setup

The following steps show the inclusion of all dependent modules:

1. In the main menu, go to Project > Options…, and a dialog appears.

2. In the left-hand column, select C/C++ Compiler.

3. In the right-hand part of the dialog, click on the Preprocessor tab (it can be hidden in the right; use the arrow icons for navigation).

4. In the text box (at the Additional include directories title), type the following folder (using the created variable):

   - $RTCESL_LOC$\MLIB\Include
   - $RTCESL_LOC$\GFLIB\Include
   - $RTCESL_LOC$\GMCLIB\Include

5. Click OK in the main dialog. See Figure 27.
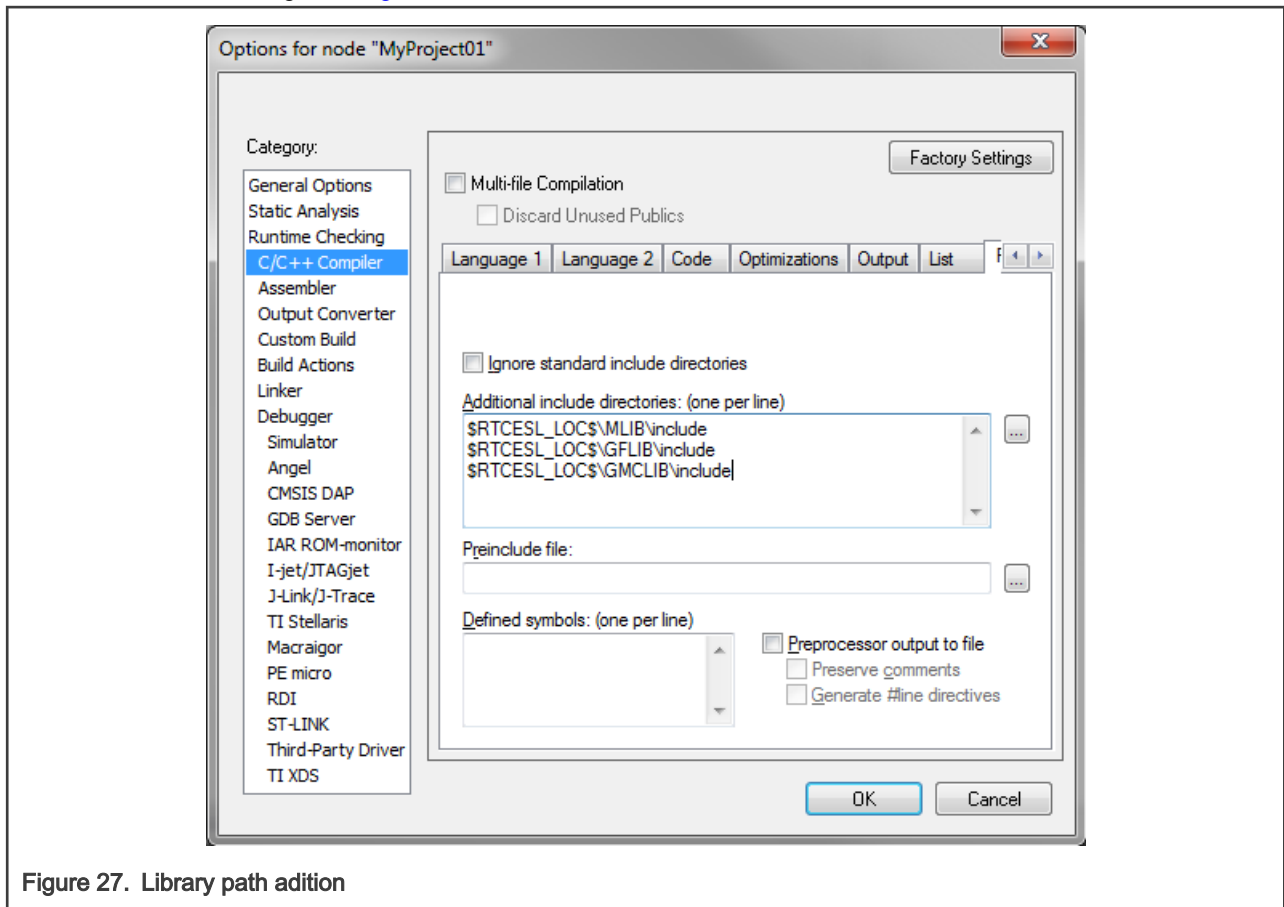
Figure 27. Library path adition

Type the #include syntax into the code. Include the library included into the *main.c* file. In the workspace tree, double-click the *main.c* file. After the *main.c* file opens up, include the following lines into the #include section:

```
#include "mlib.h"
#include "gflib.h"
#include "gmclib.h"
```

When you click the Make icon, the project will be compiled without errors.

Chapter 2
Algorithms in detail

2.1 GMCLIB_Clark

The GMCLIB_Clark function calculates the Clarke transformation, which is used to transform values (flux, voltage, current) from the three-phase coordinate system to the two-phase (α-β) orthogonal coordinate system, according to the following equations:

$$\alpha = a$$

$$\beta = \frac{1}{\sqrt{3}} b - \frac{1}{\sqrt{3}} c$$

2.1.1 Available versions

This function is available in the following versions:

- Fractional output - the output is the fractional portion of the result; the result is within the range <-1 ; 1). The result may saturate.

The available versions of the GMCLIB_Clark function are shown in the following table:

Table 2. Function versions

Function name	Input type	Output type	Result type
GMCLIB_Clark_F16	GMCLIB_3COOR_T_F16 *	GMCLIB_2COOR_ALBE_T_F16 *	void
	Clarke transformation of a 16-bit fractional three-phase system input to a 16-bit fractional two-phase system. The input and output are within the fractional range <-1 ; 1).		

2.1.2 Declaration

The available GMCLIB_Clark functions have the following declarations:

```
void GMCLIB_Clark_F16(const GMCLIB_3COOR_T_F16 *psIn, GMCLIB_2COOR_ALBE_T_F16 *psOut)
```

2.1.3 Function use

The use of the GMCLIB_Clark function is shown in the following examples:

Fixed-point version:

```
#include "gmclib.h"

static GMCLIB_2COOR_ALBE_T_F16 sAlphaBeta;
static GMCLIB_3COOR_T_F16 sAbc;

void Isr(void);

void main(void)
{
  /* ABC structure initialization */
  sAbc.f16A = FRAC16(0.0);
  sAbc.f16B = FRAC16(0.0);
  sAbc.f16C = FRAC16(0.0);
```

```
}

/* Periodical function or interrupt */
void Isr(void)
{
  /* Clarke Transformation calculation */
  GMCLIB_Clark_F16(&sAbc, &sAlphaBeta);
}
```

2.2 GMCLIB_ClarkInv

The GMCLIB_ClarkInv function calculates the Clarke transformation, which is used to transform values (flux, voltage, current) from the two-phase (α-β) orthogonal coordinate system to the three-phase coordinate system, according to the following equations:

$$a = \alpha$$

$$b = -\frac{1}{2}\alpha + \frac{\sqrt{3}}{2}\beta$$

$$c = -(a+b)$$

2.2.1 Available versions

This function is available in the following versions:

- Fractional output - the output is the fractional portion of the result; the result is within the range <-1 ; 1). The result may saturate.

The available versions of the GMCLIB_ClarkInv function are shown in the following table:

Table 3. Function versions

Function name	Input type	Output type	Result type
GMCLIB_ClarkInv_F16	GMCLIB_2COOR_ALBE_T_F16 *	GMCLIB_3COOR_T_F16 *	void
	Inverse Clarke transformation with a 16-bit fractional two-phase system input and a 16-bit fractional three-phase output. The input and output are within the fractional range <-1 ; 1).		

2.2.2 Declaration

The available GMCLIB_ClarkInv functions have the following declarations:

```
void GMCLIB_ClarkInv_F16(const GMCLIB_2COOR_ALBE_T_F16 *psIn, GMCLIB_3COOR_T_F16 *psOut)
```

2.2.3 Function use

The use of the GMCLIB_ClarkInv function is shown in the following examples:

Fixed-point version:

```
#include "gmclib.h"

static GMCLIB_2COOR_ALBE_T_F16 sAlphaBeta;
static GMCLIB_3COOR_T_F16 sAbc;
```

```
void Isr(void);

void main(void)
{
  /* Alpha, Beta structure initialization */
  sAlphaBeta.f16Alpha = FRAC16(0.0);
  sAlphaBeta.f16Beta = FRAC16(0.0);
}

/* Periodical function or interrupt */
void Isr(void)
{
  /* Inverse Clarke Transformation calculation */
  GMCLIB_ClarkInv_F16(&sAlphaBeta, &sAbc);
}
```

2.3 GMCLIB_Park

The GMCLIB_Park function calculates the Park transformation, which transforms values (flux, voltage, current) from the stationary two-phase (α-β) orthogonal coordinate system to the rotating two-phase (d-q) orthogonal coordinate system, according to the following equations:

$$d = \alpha \cdot \cos(\theta) + \beta \cdot \sin(\theta)$$

$$q = \beta \cdot \cos(\theta) - \alpha \cdot \sin(\theta)$$

where:

- θ is the position (angle)

2.3.1 Available versions

This function is available in the following versions:

- Fractional output - the output is the fractional portion of the result; the result is within the range <-1 ; 1). The result may saturate.

The available versions of the GMCLIB_Park function are shown in the following table:

Table 4. Function versions

Function name	Input type		Output type	Result type
GMCLIB_Park_F16	GMCLIB_2COOR_ALBE_T_F16 *		GMCLIB_2COOR_DQ_T_F16 *	void
	GMCLIB_2COOR_SINCOS_T_F16 *			
	The Park transformation of a 16-bit fractional two-phase stationary system input to a 16-bit fractional two-phase rotating system, using a 16-bit fractional angle two-component (sin / cos) position information. The inputs and the output are within the fractional range <-1 ; 1).			

2.3.2 Declaration

The available GMCLIB_Park functions have the following declarations:

```
void GMCLIB_Park_F16(const GMCLIB_2COOR_ALBE_T_F16 *psIn, const GMCLIB_2COOR_SINCOS_T_F16
*psAnglePos, GMCLIB_2COOR_DQ_T_F16 *psOut)
```

2.3.3 Function use

The use of the GMCLIB_Park function is shown in the following examples:

Fixed-point version:

```
#include "gmclib.h"

static GMCLIB_2COOR_ALBE_T_F16 sAlphaBeta;
static GMCLIB_2COOR_DQ_T_F16 sDQ;
static GMCLIB_2COOR_SINCOS_T_F16 sAngle;

void Isr(void);

void main(void)
{
  /* Alpha, Beta structure initialization */
  sAlphaBeta.f16Alpha = FRAC16(0.0);
  sAlphaBeta.f16Beta = FRAC16(0.0);

  /* Angle structure initialization */
  sAngle.f16Sin = FRAC16(0.0);
  sAngle.f16Cos = FRAC16(1.0);
}

/* Periodical function or interrupt */
void Isr(void)
{
  /* Park Transformation calculation */
  GMCLIB_Park_F16(&sAlphaBeta, &sAngle, &sDQ);
}
```

2.4 GMCLIB_ParkInv

The GMCLIB_ParkInv function calculates the Park transformation, which transforms values (flux, voltage, current) from the rotating two-phase (d-q) orthogonal coordinate system to the stationary two-phase (α-β) coordinate system, according to the following equations:

$$\alpha = d \cdot \cos(\theta) - q \cdot \sin(\theta)$$

$$\beta = d \cdot \sin(\theta) + q \cdot \cos(\theta)$$

where:

- θ is the position (angle)

2.4.1 Available versions

This function is available in the following versions:

- Fractional output - the output is the fractional portion of the result; the result is within the range <-1 ; 1). The result may saturate.

The available versions of the GMCLIB_ParkInv function are shown in the following table:

Table 5. Function versions

Function name	Input type	Output type	Result type
GMCLIB_ParkInv_F16	GMCLIB_2COOR_DQ_T_F16 *	GMCLIB_2COOR_ALBE_T_F16 *	void
	GMCLIB_2COOR_SINCOS_T_F16 *		
	Inverse Park transformation of a 16-bit fractional two-phase rotating system input to a 16-bit fractional two-phase stationary system, using a 16-bit fractional angle two-component (sin / cos) position information. The inputs and the output are within the fractional range <-1 ; 1).		

2.4.2 Declaration

The available GMCLIB_ParkInv functions have the following declarations:

```
void GMCLIB_ParkInv_F16(const GMCLIB_2COOR_DQ_T_F16 *psIn, const GMCLIB_2COOR_SINCOS_T_F16
*psAnglePos, GMCLIB_2COOR_ALBE_T_F16 *psOut)
```

2.4.3 Function use

The use of the GMCLIB_ParkInv function is shown in the following examples:

Fixed-point version:

```
#include "gmclib.h"

static GMCLIB_2COOR_ALBE_T_F16 sAlphaBeta;
static GMCLIB_2COOR_DQ_T_F16 sDQ;
static GMCLIB_2COOR_SINCOS_T_F16 sAngle;

void Isr(void);

void main(void)
{
  /* D, Q structure initialization */
  sDQ.f16D = FRAC16(0.0);
  sDQ.f16Q = FRAC16(0.0);

  /* Angle structure initialization */
  sAngle.f16Sin = FRAC16(0.0);
  sAngle.f16Cos = FRAC16(1.0);
}

/* Periodical function or interrupt */
void Isr(void)
{
  /* Inverse Park Transformation calculation */
  GMCLIB_ParkInv_F16(&sDQ, &sAngle, &sAlphaBeta);
}
```

2.5 GMCLIB_DecouplingPMSM

The GMCLIB_DecouplingPMSM function calculates the cross-coupling voltages to eliminate the d-q axis coupling that causes nonlinearity of the control.

The d-q model of the motor contains cross-coupling voltage that causes nonlinearity of the control. Figure 1 represents the d-q model of the motor that can be described using the following equations, where the underlined portion is the cross-coupling voltage:

$$u_d = R_s \cdot i_d + L_d \frac{d}{dt} i_d + \underline{L_q \cdot \omega_{el} \cdot i_q}$$
$$u_q = R_s \cdot i_q + L_q \frac{d}{dt} i_q - \underline{L_d \cdot \omega_{el} \cdot i_d} + \omega_{el} \cdot \psi_r$$

where:

- u_d, u_q are the d and q voltages
- i_d, i_q are the d and q currents
- R_s is the stator winding resistance
- L_d, L_q are the stator winding d and q inductances
- ω_{el} is the electrical angular speed
- ψ_r is the rotor flux constant
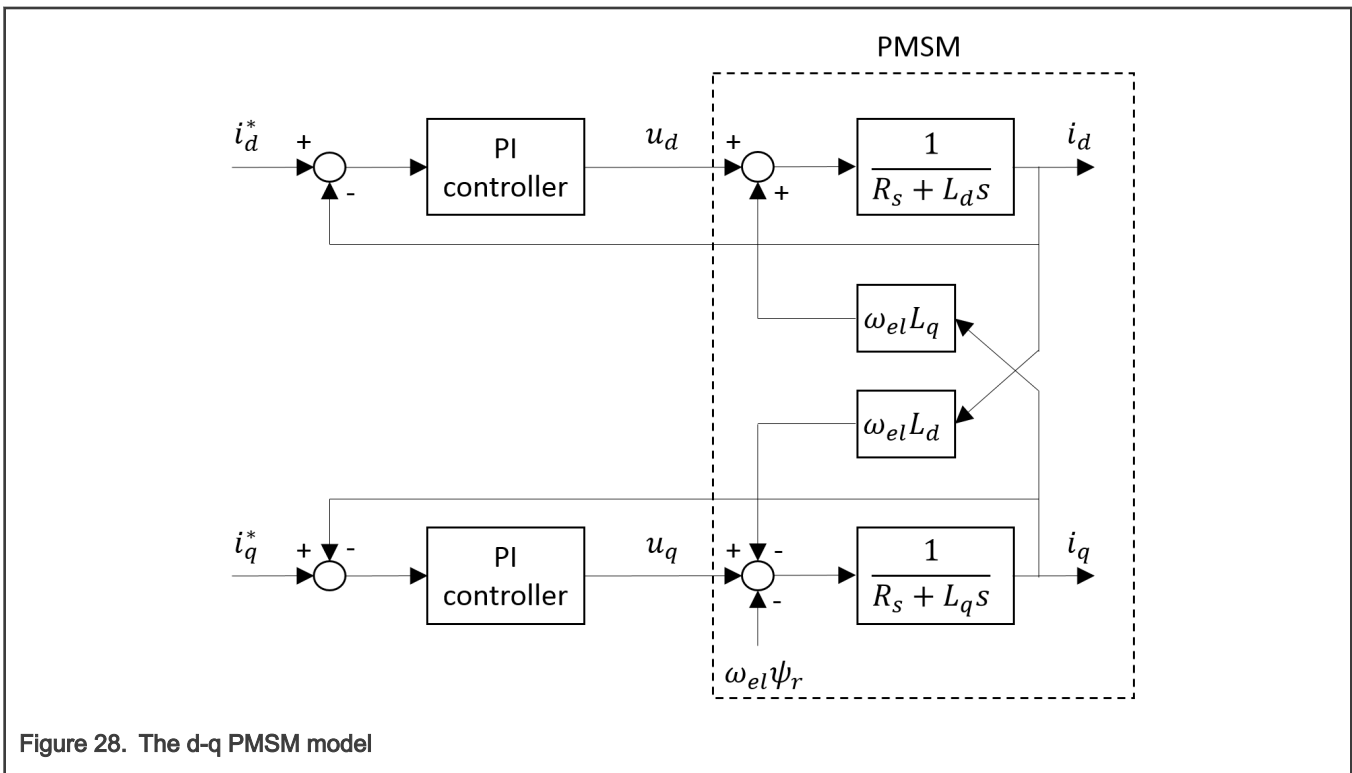
Figure 28. The d-q PMSM model

To eliminate the nonlinearity, the cross-coupling voltage is calculated using the GMCLIB_DecouplingPMSM algorithm, and feedforwarded to the d and q voltages. The decoupling algorithm is calculated using the following equations:

$$u_{ddec} = u_d - L_q \cdot \omega_{el} \cdot i_q$$
$$u_{qdec} = u_q + L_d \cdot \omega_{el} \cdot i_d$$

where:

- u_d, u_q are the d and q voltages; inputs to the algorithm

- u_{ddec}, u_{qdec} are the d and q decoupled voltages; outputs from the algorithm

The fractional representation of the d-component equation is as follows:

$$u_{ddec} = u_d - \omega_{el} \cdot i_q \Big(L_q \cdot \omega_{el\_max} \cdot \frac{i_{max}}{u_{max}} \Big)$$
$$k_q = L_q \cdot \omega_{el\_max} \cdot \frac{i_{max}}{u_{max}}$$
$$u_{ddec} = u_d - \omega_{el} \cdot i_q \cdot k_q$$

The fractional representation of the q-component equation is as follows:

$$u_{qdec} = u_q + \omega_{el} \cdot i_d \Big(L_d \cdot \omega_{el\_max} \cdot \frac{i_{max}}{u_{max}} \Big)$$
$$k_d = L_d \cdot \omega_{el\_max} \cdot \frac{i_{max}}{u_{max}}$$
$$u_{qdec} = u_q + \omega_{el} \cdot i_d \cdot k_d$$

where:

- k_d, k_q are the scaling coefficients
- i_{max} is the maximum current
- u_{max} is the maximum voltage
- ω_{el_max} is the maximum electrical speed

The k_d and k_q parameters must be set up properly.

The principle of the algorithm is depicted in Figure 2:

Figure 29. Algorithm diagram

2.5.1 Available versions

This function is available in the following versions:

- Fractional output - the output is the fractional portion of the result; the result is within the range <-1 ; 1). The result may saturate. The parameters use the accumulator types.

The available versions of the GMCLIB_DecouplingPMSM function are shown in the following table:

Table 6. Function versions

Function name	Input/output type		Result type
GMCLIB_DecouplingPMSM_F16	**Input**	GMCLIB_2COOR_DQ_T_F16 *	void
		GMCLIB_2COOR_DQ_T_F16 *	
		frac16_t	
	Parameters	GMCLIB_DECOUPLINGPMSM_T_A32 *	
	Output	GMCLIB_2COOR_DQ_T_F16 *	
	The PMSM decoupling with a 16-bit fractional d-q voltage, current inputs, and a 16-bit fractional electrical speed input. The parameters are 32-bit accumulator types. The output is a 16-bit fractional decoupled d-q voltage. The inputs and the output are within the range <-1 ; 1).		

2.5.2 GMCLIB_DECOUPLINGPMSM_T_A32 type description

Variable name	Input type	Description
a32KdGain	acc32_t	Direct axis decoupling parameter. The parameter is within the range <0 ; 65536.0)
a32KqGain	acc32_t	Quadrature axis decoupling parameter. The parameter is within the range <0 ; 65536.0)

2.5.3 Declaration

The available GMCLIB_DecouplingPMSM functions have the following declarations:

```
void GMCLIB_DecouplingPMSM_F16(const GMCLIB_2COOR_DQ_T_F16 *psUDQ, const GMCLIB_2COOR_DQ_T_F16
*psIDQ, frac16_t f16SpeedEl, const GMCLIB_DECOUPLINGPMSM_T_A32 *psParam,
GMCLIB_2COOR_DQ_T_F16 *psUDQDec)
```

2.5.4 Function use

The use of the GMCLIB_DecouplingPMSM function is shown in the following examples:

Fixed-point version:

```
#include "gmclib.h"

static GMCLIB_2COOR_DQ_T_F16 sVoltageDQ;
static GMCLIB_2COOR_DQ_T_F16 sCurrentDQ;
static frac16_t f16AngularSpeed;
static GMCLIB_DECOUPLINGPMSM_T_A32 sDecouplingParam;
static GMCLIB_2COOR_DQ_T_F16 sVoltageDQDecoupled;

void Isr(void);

void main(void)
{
```

```
    /* Voltage D, Q structure initialization */
    sVoltageDQ.f16D = FRAC16(0.0);
    sVoltageDQ.f16Q = FRAC16(0.0);

    /* Current D, Q structure initialization */
    sCurrentDQ.f16D = FRAC16(0.0);
    sCurrentDQ.f16Q = FRAC16(0.0);

    /* Speed initialization */
    f16AngularSpeed = FRAC16(0.0);

    /* Motor parameters for decoupling Kd = 40, Kq = 20  */
    sDecouplingParam.a32KdGain = ACC32(40.0);
    sDecouplingParam.a32KqGain = ACC32(20.0);
}

/* Periodical function or interrupt */
void Isr(void)
{
    /* Decoupling calculation */
    GMCLIB_DecouplingPMSM_F16(&sVoltageDQ, &sCurrentDQ, f16AngularSpeed,
&sDecouplingParam, &sVoltageDQDecoupled);
}
```

2.6 GMCLIB_DTCompLut1D

The GMCLIB_DTCompLut1D function implements dead-time-compensation algorithms that return error voltages from measured currents, knowing the inverter nonlinearity in the Look-Up Table (LUT). The aim of the GMCLIB_DTCompLut1D function is to make the inverter and the whole control loop more linear, especially around low-duty cycles, where the dead-time effect is dominant. The error voltage obtained from GMCLIB_DTCompLut1D has the most important impact in the low-motor-speed region, where the motor-supply voltage is low. The next operation is to transform the error voltages from the three-phase (a, b, c) system of coordinates to the two-phase (α, β) orthogonal system using the Clarke transformation. The multiplication by U_{DCBus} voltages is then processed and the output-compensation voltages are obtained after adding input voltages. The principle of the GMCLIB_DTCompLut1D function is shown in Figure 30.
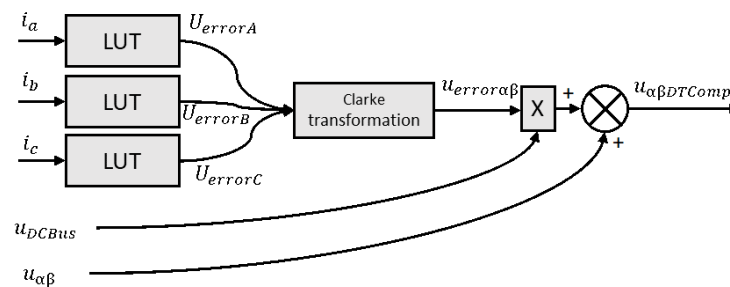
Figure 30. GMCLIB_DTCompLut1D function block scheme

Adding the GMCLIB_DTCompLut1D function into your application has the following essentials. Each inverter introduces the total error voltage, which is caused by the dead-time and current-clamping effects and the transistor voltage drop. The actual inverter output voltage is lower than the required voltage. The total error voltage depends on the actual phase current. The example of the inverter error characteristic is shown in Figure 31.
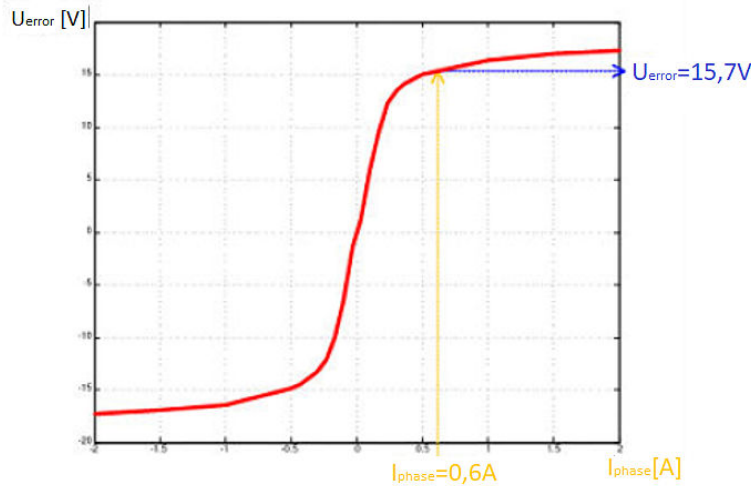
Figure 31. Inverter error characteristic example - voltage error dependency on phase current

The inverter error characteristic is assigned to the GMCLIB_DTCompLut1D algorithm as the LUT, which simply adds the error-voltage vector to the input-voltage vector. The data for the LUT should be measured from a real inverter for the best compensation result. The target of the measurement is the voltage-error-to-phase-current dependency, as shown in Figure 32.
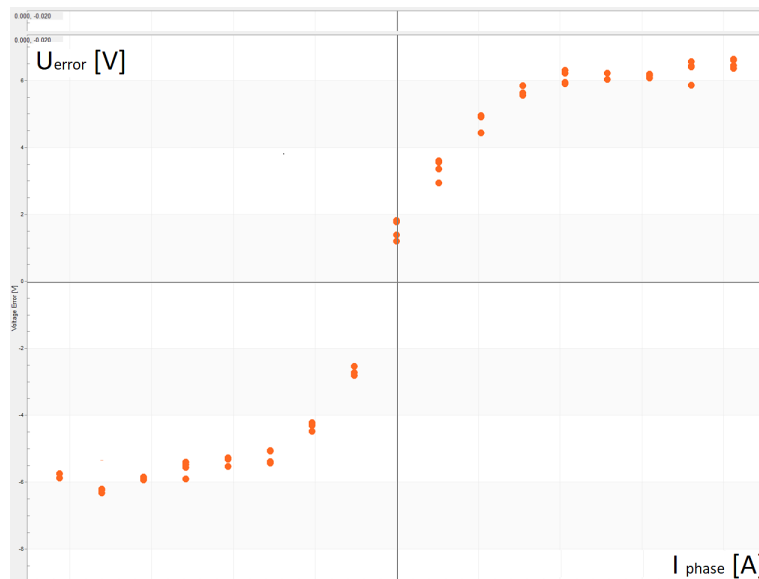
Figure 32. Measured inverter voltage error data example for LUT measured for phase current points

The LUT is defined by the table pointer and tableSize parameters. The table output is the error-voltage vector from the phase-current input vector. The GMCLIB_DTCompLut1D algorithm processes the compensation voltage according to the following equations:

$$u_{Aerror} = GFLIB\_Lut1D\_F16(i_a, Table, \quad TableSize)$$
$$u_{Berror} = GFLIB\_Lut1D\_F16(i_b, Table, \quad TableSize)$$
$$u_{Cerror} = GFLIB\_Lut1D\_F16(i_c, Table, \quad TableSize)$$

Figure 33. Equations to get the error voltages componnents

where:

- u_{Aerror}, $_{Berror}$, and u_{Cerror} are the error-voltage components of the error-voltage vector.
- i_a, i_b, and i_c are the current components of the measured-current vector.
- The LUT contains the inverter measured characteristic. See GMCLIB_DTCOMPLUT1D_T_F16 for details.
- The tableSize parameter is the size of the LUT parameter. See GMCLIB_DTCOMPLUT1D_T_F16 for details.
- The GFLIB_Lut1D_F16 parameter is the LUT function (GMCLIB_DTCompLut1D).

When the error voltages are calculated, the GMCLIB_DTCompLut1D algorithm continues using the following equations:

$$u_{\beta DTComp} = u_\alpha + (2 \times u_{AError} - u_{BError} - u_{CError}) \times \frac{u_{dcbus}}{3}$$

$$u_{\beta DTComp} = u_\beta + (u_{BError} - u_{CError}) \times \frac{\sqrt{3} \times u_{dcbus}}{3}$$

Figure 34. DTCompLUT1D function equations

where:

- $u_{\alpha DTComp}$ and $u_{\beta DTComp}$ are the compensation voltage components and outputs of the GMCLIB_DTCompLut1D function.
- u_α, and u_β are the voltage components of the input-voltage vector.
- u_{Aerror}, u_{Berror}, and u_{Cerror} are the error-voltage components of the error-voltage vector.
- u_{DCBus} is the DC-Bus voltage.

At the end of the GMCLIB_DTCompLut1D algorithm, the error-compensation-voltage componenets (α and β) are added to the input voltage components (α and β). The addition with saturation is used to avoid overflow and the resuls of the GMCLIB_DTCompLut1D function are the dead-time-compensated voltage components ($U_{\alpha DTComp}$ and $U_{\beta DTComp}$) as inputs for the SVMShifted function.
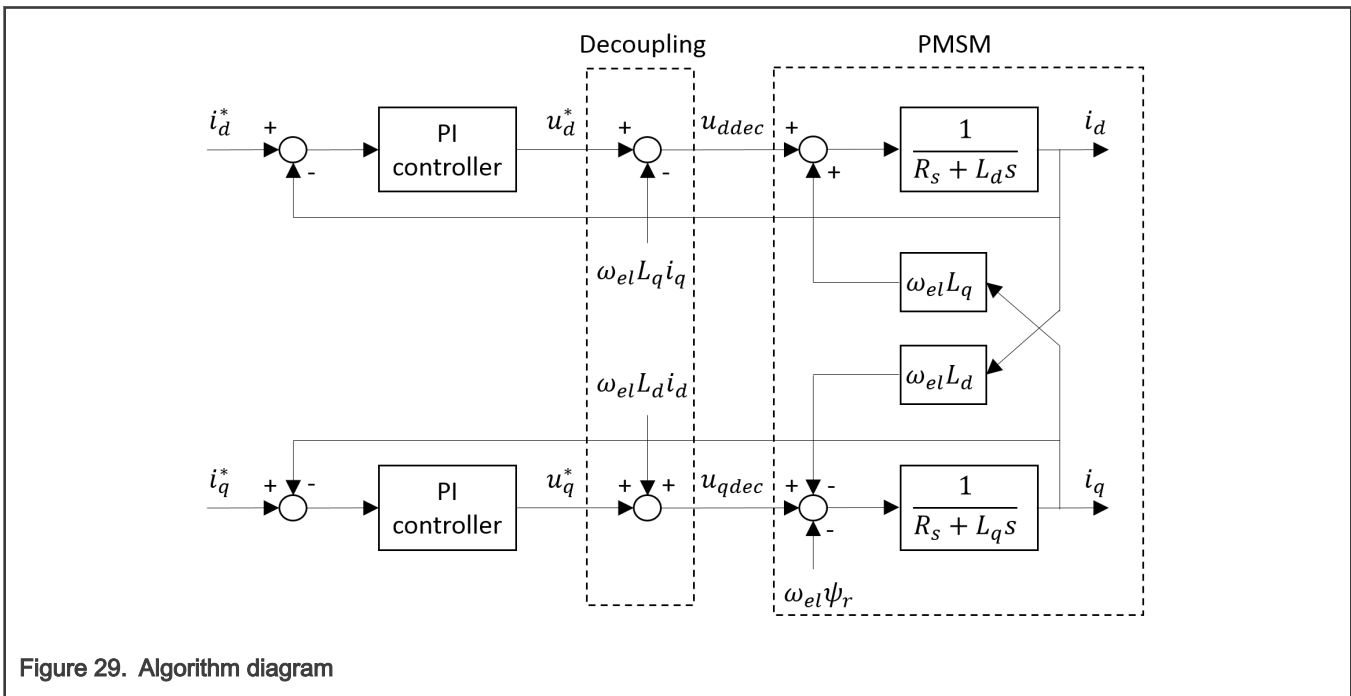
2.6.1 Available versions

This function is available in the following versions:

- Fractional output - the output is the fractional portion of the result. The result is in the range of **<-1 ; 1)**. The result may saturate. The parameter inludes the table and table size used the table types.

Table 7. Function versions

Function name	Input/output type		Result type
GMCLIB_DTCompLut1D_F16	Inputs	GMCLIB_3COOR_T_F16 * GMCLIB_2COOR_ALBE_T_F16 *, frac16_t	void
	Parameter	GMCLIB_DTCOMPLUT1D_T_F16 *, *	
	Output	GMCLIB_2COOR_ALBE_T_F16 *	
	The first input argument is the structure of current components represented by the 16-bit fractional values. It contains the abscissas for which the 1-D interpolations are performed. The second input argument is the structure of voltage components represented by the two-phase (α, β) orthogonal coordinate system. The last input argument is the measured 16-bit fractional DC-Bus voltage input.		

Table continues on the next page...

Table 7. Function versions

Function name	Input/output type	Result type
	The parameter structure consists of two members of the GMCLIB_DTCOMPLUT1D_T_F16 structure type. The first parameter is the pointer to the LUT and the second parameter is the size of the LUT. The ouput argument is the strucrure of compensated voltage components represented by the two-phase (α, β) orthogonal coordinate system, targeted for the SVMShifted function.	

2.6.2 GMCLIB_DTCOMPLUT1D_T_F16 type description

Variable name	Input type	Description
pf16Table	frac16_t *	The pointer to a LUT, which contains the 16-bit fractional values of the LUT.
u16TableSize	uint16_t	The size of the LUT parameter is in the range of <1 ; 15>. It means that the parameter is log2 of the number of points + 1.

2.6.3 Declaration

The available GMCLIB_DTCompLut1D functions have the following declarations:

```
GMCLIB_DTCompLut1D_F16(const GMCLIB_3COOR_T_F16 *psIABC, const GMCLIB_2COOR_ALBE_T_F16 *psUAlBe,
frac16_tf16UDCBus, const GMCLIB_DTCOMPLUT1D_T_F16 *psParam, GMCLIB_2COOR_ALBE_T_F16 *psUAlBeDTComp)
```

2.6.4 Function use

The use of the GMCLIB_DTCompLut1D function is shown in the following example:

Fixed-point version:

```c
#include "gmclib.h"
static GMCLIB_DTCOMPLUT1D_T_F16 sParam;
static GMCLIB_3COOR_T_F16 sIABC;
static GMCLIB_2COOR_ALBE_T_F16 sUAlBe, sUAlBeComp;
static frac16_t f16UDCBus;
static frac16_t f16Table[9] = {FRAC16(-0.7), FRAC16(-0.65), FRAC16(-0.55),
FRAC16(0.2), FRAC16(0.0),
                                    FRAC16(0.2), FRAC16(-0.8), FRAC16(0.91),
FRAC16(0.99)};
void Isr(void);

void main(void)
{
    /* ABC currents structure initialization */
    sIABC.f16A = FRAC16(0.1);
    sIABC.f16B = FRAC16(0.3);
    sIABC.f16C = FRAC16(-0.2);
    /* Alpha Bete voltages structure initialization */
    sUAlBe.f16Alpha = FRAC16(0.25);
    sUAlBe.f16Beta  = FRAC16(0.75);
    /* DC Bus voltage initialization */
```

```
        f16UDCBus = FRAC16(0.9);
        /* DC Bus voltage initialization */
        sParam.pf16Table = f16Table;
        sParam.u16TableSize = 3;
}
/* Periodical function or interrupt */
void Isr(void)
{
        /* Dead-Time compensation calculation */
        GMCLIB_DTCompLut1D_F16(&sIABC, &sUAlBe, f16UDCBus, &sParam, &sUAlBeComp);
}
```

2.7 GMCLIB_ElimDcBusRipFOC

The GMCLIB_ElimDcBusRipFOC function is used for the correct PWM duty cycle output calculation, based on the measured DC-bus voltage. The side effect is the elimination of the the DC-bus voltage ripple in the output PWM duty cycle. This function is meant to be used with a space vector modulation, whose modulation index (with respect to the DC-bus voltage) is an inverse square root of 3.

The general equation to calculate the duty cycle for the above-mentioned space vector modulation is as follows:

$$U_{PWM} = \frac{u_{FOC}}{u_{dcbus}} \cdot \sqrt{3}$$

where:

- U_{PWM} is the duty cycle output
- u_{FOC} is the real FOC voltage
- u_{dcbus} is the real measured DC-bus voltage

Using the previous equations, the GMCLIB_ElimDcBusRipFOC function compensates an amplitude of the direct-α and the quadrature-β component of the stator-reference voltage vector, using the formula shown in the following equations:

$$U_\alpha^* = \begin{cases} 0, & U_\alpha = 0 \ \wedge \ U_{dcbus} = 0 \\ 1, & U_\alpha \geq 0 \ \wedge \ |U_\alpha| \geq \frac{U_{dcbus}}{\sqrt{3}} \\ -1, & U_\alpha < 0 \ \wedge \ |U_\alpha| \geq \frac{U_{dcbus}}{\sqrt{3}} \\ \frac{U_\alpha}{U_{dcbus}} \cdot \sqrt{3}, & \text{else} \end{cases}$$

$$U_\beta^* = \begin{cases} 0, & U_\beta = 0 \ \wedge \ U_{dcbus} = 0 \\ 1, & U_\beta \geq 0 \ \wedge \ |U_\beta| \geq \frac{U_{dcbus}}{\sqrt{3}} \\ -1, & U_\beta < 0 \ \wedge \ |U_\beta| \geq \frac{U_{dcbus}}{\sqrt{3}} \\ \frac{U_\beta}{U_{dcbus}} \cdot \sqrt{3}, & \text{else} \end{cases}$$

where:

- U_α^* is the direct-α duty cycle ratio
- U_β^* is the quadrature-β duty cycle ratio
- U_α is the direct-α voltage
- U_β is the quadrature-β voltage

If the fractional arithmetic is used, the FOC and DC-bus voltages have their scales, which take place in GMCLIB_ElimDcBusRipFOC_Eq1; the equation is as follows:

$$U_{PWM} = \frac{U_{FOC} \cdot U_{FOC\_max}}{U_{dcbus} \cdot U_{dcbus\_max}} \cdot \sqrt{3}$$

where:

- U_{FOC} is the scaled FOC voltage
- U_{dcbus} is the scaled measured DC-bus voltage
- U_{FOC_max} is the FOC voltage scale
- U_{dcbus_max} is the DC-bus voltage scale

If this algorithm is used with the space vector modulation with the ratio of square root equal to 3, then the FOC voltage scale is expressed as follows :

$$U_{FOC\_max} = \frac{U_{dcbus\_max}}{\sqrt{3}}$$

The equation can be simplified as follows:

$$U_{PWM} = \frac{U_{FOC} \frac{U_{dcbus\_max}}{\sqrt{3}}}{U_{dcbus} \cdot U_{dcbus\_max}} \cdot \sqrt{3} = \frac{U_{FOC}}{U_{dcbus}}$$

The GMCLIB_ElimDcBusRipFOC function compensates an amplitude of the direct-α and the quadrature-β component of the stator-reference voltage vector in the fractional arithmetic, using the formula shown in the following equations:

$$U_{\alpha}* = \begin{cases} 0, & U_{\alpha} = 0 \ \wedge \ U_{dcbus} = 0 \\ 1, & U_{\alpha} > 0 \ \wedge \ |U_{\alpha}| \geq U_{dcbus} \\ -1, & U_{\alpha} < 0 \ \wedge \ |U_{\alpha}| \geq U_{dcbus} \\ \frac{U_{\alpha}}{U_{dcbus}}, & \text{else} \end{cases}$$

$$U_{\beta}* = \begin{cases} 0, & U_{\beta} = 0 \ \wedge \ U_{dcbus} = 0 \\ 1, & U_{\beta} > 0 \ \wedge \ |U_{\beta}| \geq U_{dcbus} \\ -1, & U_{\beta} < 0 \ \wedge \ |U_{\beta}| \geq U_{dcbus} \\ \frac{U_{\beta}}{U_{dcbus}}, & \text{else} \end{cases}$$

where:

- $U_{\alpha}*$ is the direct-α duty cycle ratio
- $U_{\beta}*$ is the quadrature-β duty cycle ratio
- U_{α} is the direct-α voltage
- U_{β} is the quadrature-β voltage

The GMCLIB_ElimDcBusRipFOC function can be used in general motor-control applications, and it provides elimination of the voltage ripple on the DC-bus of the power stage. Figure 1 shows the results of the DC-bus ripple elimination, while compensating the ripples of the rectified voltage using a three-phase uncontrolled rectifier.

Figure 35. Results of the DC-bus voltage ripple elimination

2.7.1 Available versions

This function is available in the following versions:

- Fractional output - the output is the fractional portion of the result; the result is within the range <-1 ; 1). The result may saturate.

The available versions of the GMCLIB_ElimDcBusRipFOC function are shown in the following table:

Table 8. Function versions

Function name	Input type	Output type	Result type
GMCLIB_ElimDcBusRipFOC_F16	frac16_t	GMCLIB_2COOR_ALBE_T_F16 *	void
	GMCLIB_2COOR_ALBE_T_F16 *		
	Compensation of a 16-bit fractional two-phase system input to a 16-bit fractional two-phase system, using a 16-bit fractional DC-bus voltage information. The DC-bus voltage input is within the fractional range <0 ; 1); the stationary (α-β) voltage input and the output are within the fractional range <-1 ; 1).		

2.7.2 Declaration

The available GMCLIB_ElimDcBusRipFOC functions have the following declarations:

```
void GMCLIB_ElimDcBusRipFOC_F16(frac16_t f16UDCBus, const GMCLIB_2COOR_ALBE_T_F16 *psUAlBe,
GMCLIB_2COOR_ALBE_T_F16 *psUAlBeComp)
```

2.7.3 Function use

The use of the GMCLIB_ElimDcBusRipFOC function is shown in the following example:

```c
#include "gmclib.h"

static frac16_t f16UDcBus;
static GMCLIB_2COOR_ALBE_T_F16 sUAlBe;
static GMCLIB_2COOR_ALBE_T_F16 sUAlBeComp;

void Isr(void);

void main(void)
{
  /* Voltage Alpha, Beta structure initialization */
  sUAlBe.f16Alpha = FRAC16(0.0);
  sUAlBe.f16Beta = FRAC16(0.0);

  /* DC bus voltage initialization */
  f16UDcBus = FRAC16(0.8);
}

/* Periodical function or interrupt */
void Isr(void)
{
  /* FOC Ripple elimination calculation */
  GMCLIB_ElimDcBusRipFOC_F16(f16UDcBus, &sUAlBe, &sUAlBeComp);
}
```

2.8 GMCLIB_ElimDcBusRip

The GMCLIB_ElimDcBusRip function is used for a correct PWM duty cycle output calculation, based on the measured DC-bus voltage. The side effect is the elimination of the the DC-bus voltage ripple in the output PWM duty cycle. This function can be used with any kind of space vector modulation; it has an additional input - the modulation index (with respect to the DC-bus voltage).

The general equation to calculate the duty cycle is as follows:

$$U_{PWM} = \frac{u_{FOC}}{u_{dcbus}} \cdot i_{mod}$$

where:

- U_{PWM} is the duty cycle output
- u_{FOC} is the real FOC voltage
- u_{dcbus} is the real measured DC-bus voltage
- i_{mod} is the space vector modulation index

Using the previous equations, the GMCLIB_ElimDcBusRip function compensates an amplitude of the direct-α and the quadrature-β component of the stator-reference voltage vector, using the formula shown in the following equations:

$$U_\alpha* = \begin{cases} 0, & U_\alpha = 0 \ \wedge \ U_{dcbus} = 0 \ \vee \ i_{mod} = 0 \\ 1, & U_\alpha > 0 \ \wedge \ |U_\alpha| \geq \frac{U_{dcbus}}{i_{mod}} \ \wedge \ i_{mod} > 0 \\ -1, & U_\alpha < 0 \ \wedge \ |U_\alpha| \geq \frac{U_{dcbus}}{i_{mod}} \ \wedge \ i_{mod} > 0 \\ \frac{U_\alpha}{U_{dcbus}} \cdot i_{mod}, & i_{mod} > 0 \end{cases}$$

$$U_\beta* = \begin{cases} 0, & U_\beta = 0 \ \wedge \ U_{dcbus} = 0 \ \vee \ i_{mod} = 0 \\ 1, & U_\beta > 0 \ \wedge \ |U_\beta| \geq \frac{U_{dcbus}}{i_{mod}} \ \wedge \ i_{mod} > 0 \\ -1, & U_\beta < 0 \ \wedge \ |U_\beta| \geq \frac{U_{dcbus}}{i_{mod}} \ \wedge \ i_{mod} > 0 \\ \frac{U_\beta}{U_{dcbus}} \cdot i_{mod}, & i_{mod} > 0 \end{cases}$$

where:

- $U_\alpha*$ is the direct-α duty cycle ratio

- $U_\beta*$ is the quadrature-β duty cycle ratio

- U_α is the direct-α voltage

- U_β is the quadrature-β voltage

If the fractional arithmetic is used, the FOC and DC-bus voltages have their scales, which take place in GMCLIB_ElimDcBusRipFOC_Eq1; the equation is as follows:

$$U_{PWM} = \frac{U_{FOC} \cdot U_{FOC\_max}}{U_{dcbus} \cdot U_{dcbus\_max}} \cdot i_{mod} = \frac{U_{FOC}}{U_{dcbus}} \cdot \frac{U_{FOC\_max}}{U_{dcbus\_max}} \cdot i_{mod}$$

where:

- U_{FOC} is the scaled FOC voltage

- U_{dcbus} is the scaled measured DC-bus voltage

- U_{FOC_max} is the FOC voltage scale

- U_{dcbus_max} is the DC-bus voltage scale

Thus, the modulation index in the fractional representation is expressed as follows :

$$i_{modfr} = \frac{U_{FOC\_max}}{U_{dcbus\_max}} \cdot i_{mod}$$

where:

- i_{modfr} is the space vector modulation index in the fractional arithmetic

The GMCLIB_ElimDcBusRip function compensates an amplitude of the direct-α and the quadrature-β component of the stator-reference voltage vector in the fractional arithmetic, using the formula shown in the following equations:

$$U_\alpha* = \begin{cases} 0, & U_\alpha = 0 \ \wedge \ U_{dcbus} = 0 \ \vee \ i_{modfr} = 0 \\ 1, & U_\alpha > 0 \ \wedge \ |U_\alpha| \geq \frac{U_{dcbus}}{i_{modfr}} \ \wedge \ i_{modfr} > 0 \\ -1, & U_\alpha < 0 \ \wedge \ |U_\alpha| \geq \frac{U_{dcbus}}{i_{modfr}} \ \wedge \ i_{modfr} > 0 \\ \frac{U_\alpha}{U_{dcbus}} \cdot i_{modfr}, & i_{modfr} > 0 \end{cases}$$

$$U_\beta* = \begin{cases} 0, & U_\beta = 0 \ \wedge \ U_{dcbus} = 0 \ \vee \ i_{modfr} = 0 \\ 1, & U_\beta > 0 \ \wedge \ |U_\beta| \geq \frac{U_{dcbus}}{i_{modfr}} \ \wedge \ i_{modfr} > 0 \\ -1, & U_\beta < 0 \ \wedge \ |U_\beta| \geq \frac{U_{dcbus}}{i_{modfr}} \ \wedge \ i_{modfr} > 0 \\ \frac{U_\beta}{U_{dcbus}} \cdot i_{modfr}, & i_{modfr} > 0 \end{cases}$$

where:

- $U_\alpha*$ is the direct-α duty cycle ratio

- $U_\beta*$ is the quadrature-β duty cycle ratio

- U_α is the direct-α voltage

- U_β is the quadrature-β voltage

The GMCLIB_ElimDcBusRip function can be used in general motor-control applications, and it provides elimination of the voltage ripple on the DC-bus of the power stage. Figure 1 shows the results of the DC-bus ripple elimination, while compensating the ripples of the rectified voltage, using a three-phase uncontrolled rectifier.
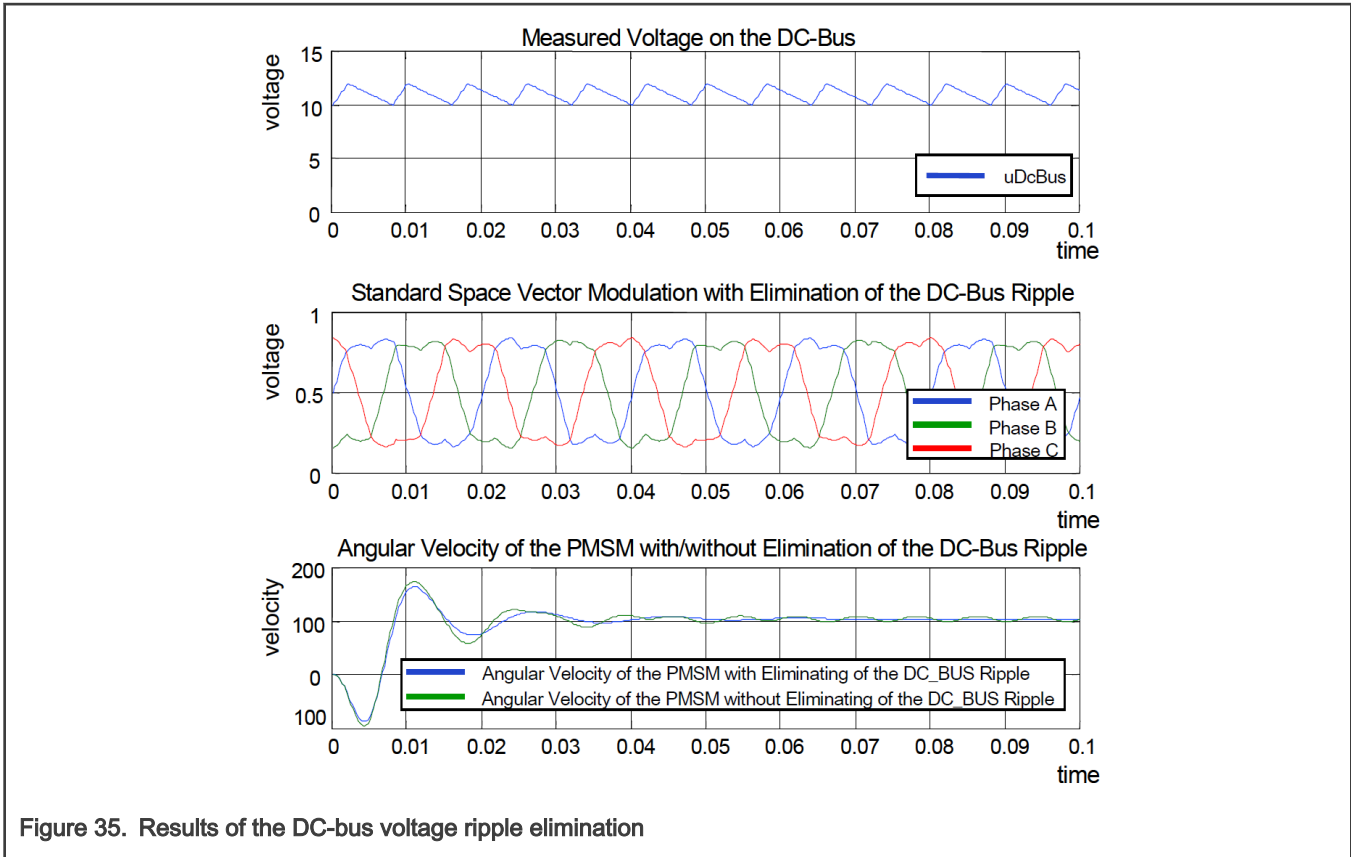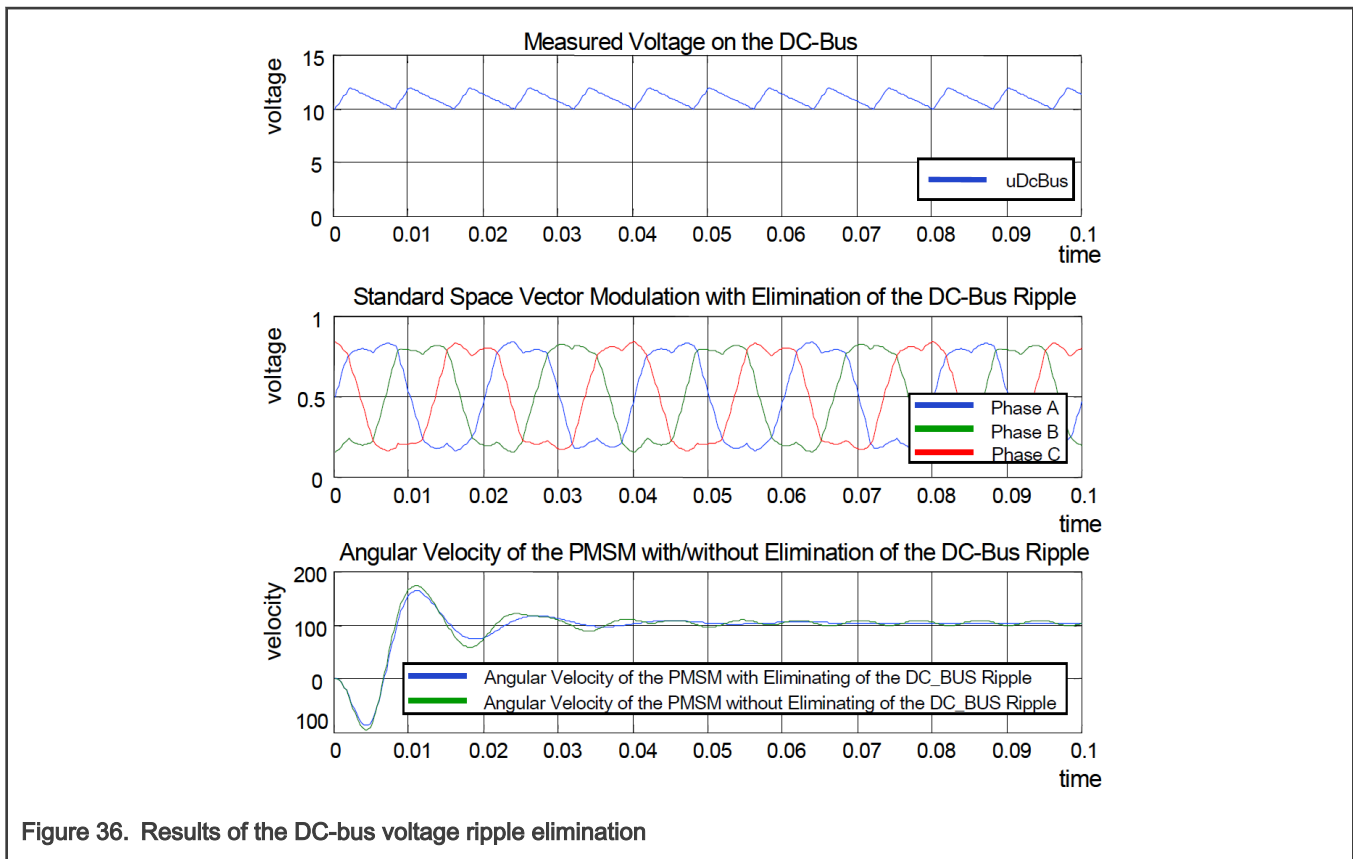

Figure 36. Results of the DC-bus voltage ripple elimination

2.8.1 Available versions

This function is available in the following versions:

- Fractional output - the output is the fractional portion of the result; the result is within the range <-1 ; 1). The result may saturate. The modulation index is a non-negative accumulator type value.

The available versions of the GMCLIB_ElimDcBusRip function are shown in the following table:

Table 9. Function versions

Function name	Input type	Output type	Result type
GMCLIB_ElimDcBusRip_F16sas	frac16_t	GMCLIB_2COOR_ALBE_T_F16 *	void
	acc32_t		
	GMCLIB_2COOR_ALBE_T_F16 *		
	Compensation of a 16-bit fractional two-phase system input to a 16-bit fractional two-phase system using a 16-bit fractional DC-bus voltage information and a 32-bit accumulator modulation index. The DC-bus voltage input is within the fractional range <0 ; 1); the modulation index is a non-negative value; the stationary (α-β) voltage input and output are within the fractional range <-1 ; 1).		

2.8.2 Declaration

The available GMCLIB_ElimDcBusRip functions have the following declarations:

```
void GMCLIB_ElimDcBusRip_F16sas(frac16_t f16UDCBus, acc32_t a32IdxMod, const GMCLIB_2COOR_ALBE_T_F16
*psUAlBeComp, GMCLIB_2COOR_ALBE_T_F16 *psUAlBe)
```

2.8.3 Function use

The use of the GMCLIB_ElimDcBusRip function is shown in the following example:

```
#include "gmclib.h"

static frac16_t f16UDcBus;
static acc32_t a32IdxMod;
static GMCLIB_2COOR_ALBE_T_F16 sUAlBe;
static GMCLIB_2COOR_ALBE_T_F16 sUAlBeComp;

void Isr(void);

void main(void)
{
  /* Voltage Alpha, Beta structure initialization */
  sUAlBe.f16Alpha = FRAC16(0.0);
  sUAlBe.f16Beta = FRAC16(0.0);

  /* SVM modulation index */
  a32IdxMod = ACC32(1.3);

  /* DC bus voltage initialization */
  f16UDcBus = FRAC16(0.8);
}

/* Periodical function or interrupt */
void Isr(void)
{
  /* Ripple elimination calculation */
```

```
    GMCLIB_ElimDcBusRip_F16sas(f16UDcBus, a32IdxMod, &sUAlBe, &sUAlBeComp);
}
```

2.9 GMCLIB_SvmStdShifted

The GMCLIB_SvmStdShifted function is based on the GMCLIB_SvmStd function and calculates the appropriate duty-cycle ratios, which are needed to generate the given stator-reference voltage vector using a special standard space-vector modulation technique. The GMCLIB_SvmStdShifted function enables the single-shunt measurement and current reconstruction and provides the data sturctures to configure the ADC and PWM peripherals.

The PWM signal generation by the GMCLIB_SvmStdShifted function applies a four-voltage vector in each PWM period:

- Two of them are inactive – vectors V0 (all switch elements are OFF – 000 states) and V7 (all switch elements are ON – 111 states).

- Two of them are active vectors that generate the motor power. See the table in Figure 37.

Two phase currents are normally available as the DC-Bus current (i_{dcb}) during the active voltage vectors in each PWM period. It is possible to reconstruct all phase currents by measuring two different samples of i_{dcb} in each PWM period. The i_{dcb} current is 0 during V0 and V7. It can be used for offset measurement.

SVM Sector	Active Voltage Vectors	DC-bus Current i_{dcb}
Sector I	V_1 (100)	$+I_{SA}$
	V_2 (110)	$-I_{SC}$
Sector II	V_3 (010)	$+I_{SB}$
	V_2 (110)	$-I_{SC}$
Sector III	V_3 (010)	$+I_{SB}$
	V_4 (011)	$-I_{SA}$
Sector IV	V_5 (001)	$+I_{SC}$
	V_4 (011)	$-I_{SA}$
Sector V	V_5 (001)	$+I_{SC}$
	V_6 (101)	$-I_{SB}$
Sector VI	V_1 (100)	$+I_{SA}$
	V_6 (101)	$-I_{SB}$
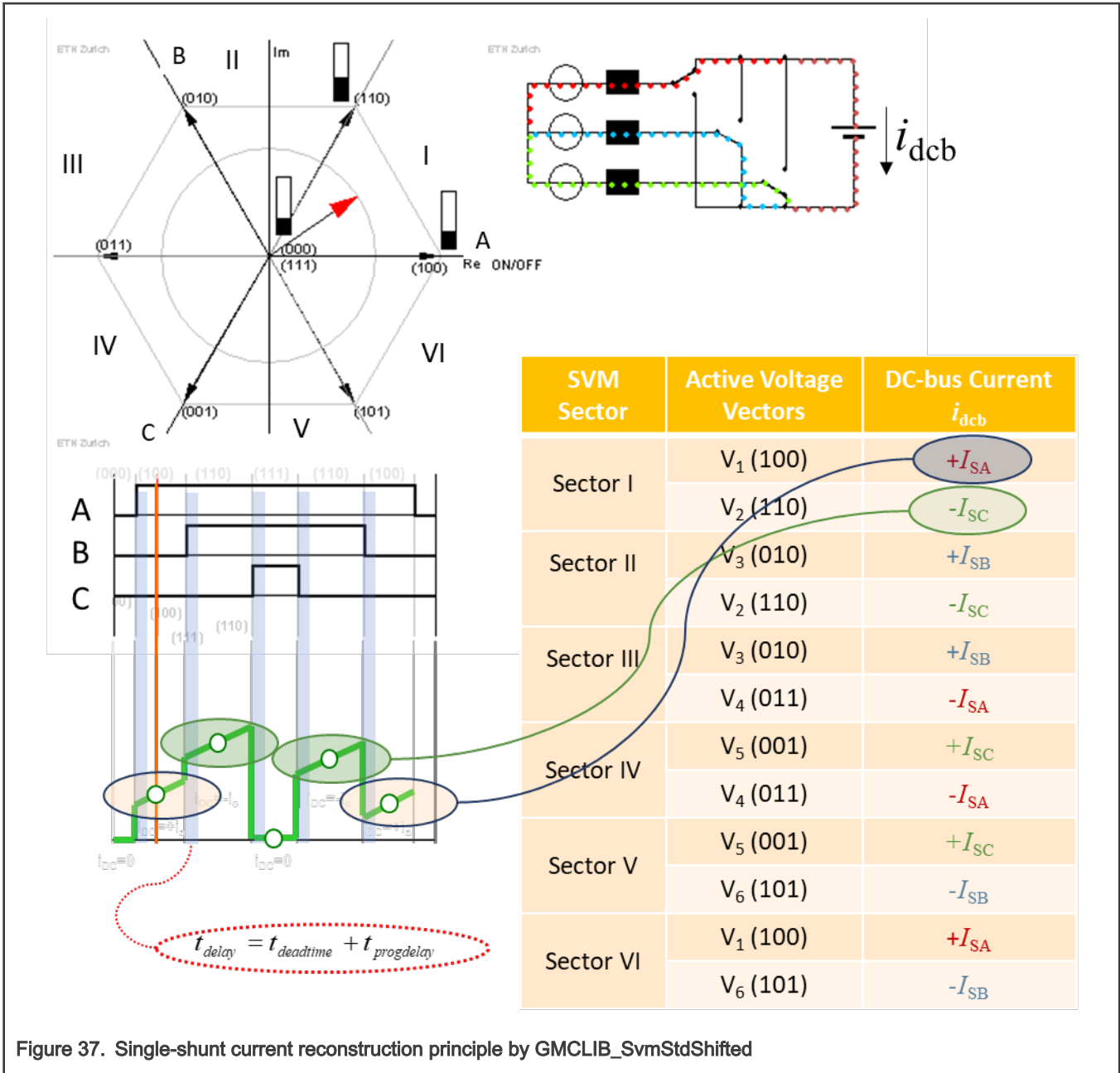
Figure 37. Single-shunt current reconstruction principle by GMCLIB_SvmStdShifted

Two different phase-current samples cannot be taken when:

- **The voltage vector is crossing the SVM sector border.** Only one sample can be taken.

- **The modulation index is low.** The sampling intervals are too short and no current samples can be taken.

There are many solutions available for these problems. For this function, the shifted-PWM method was used:

- The ON/OFF times are modified (shifted), if necessary.

- The duty cycles are preserved (the applied stator voltage is the same).

1. Passing Active Vector

Cannot take two phase current samples

A
B
C

Standard center-aligned PWM

A
B
C

Shifted-PWM

Two phase current samples can be taken

Figure 38. Problematic current reconstruction cases
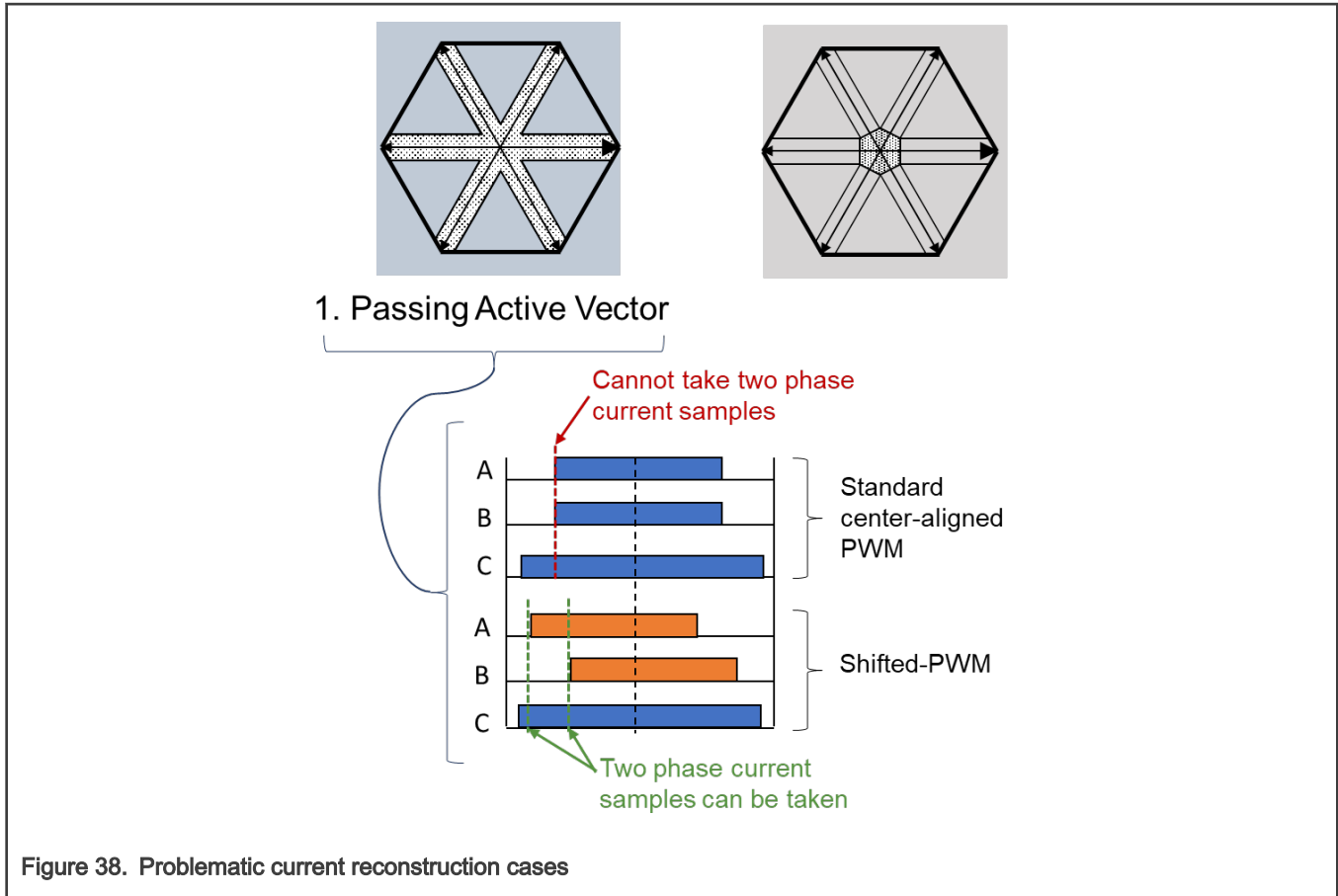
A different shifting strategy is applied for both critical cases:

1. **Passing active vector:**

   • Freezes the center edge.

   • Moves one critical edge.

   • It is used for higher modulation indexes.

2. **Low modulation indexes:**

   • Freezes the center edge.

   • Moves both side edges in opposite directions.

   • It is used for low modulation indexes.

The right method is selected within the SVM algorithm and shifts are applied by the PWM driver.
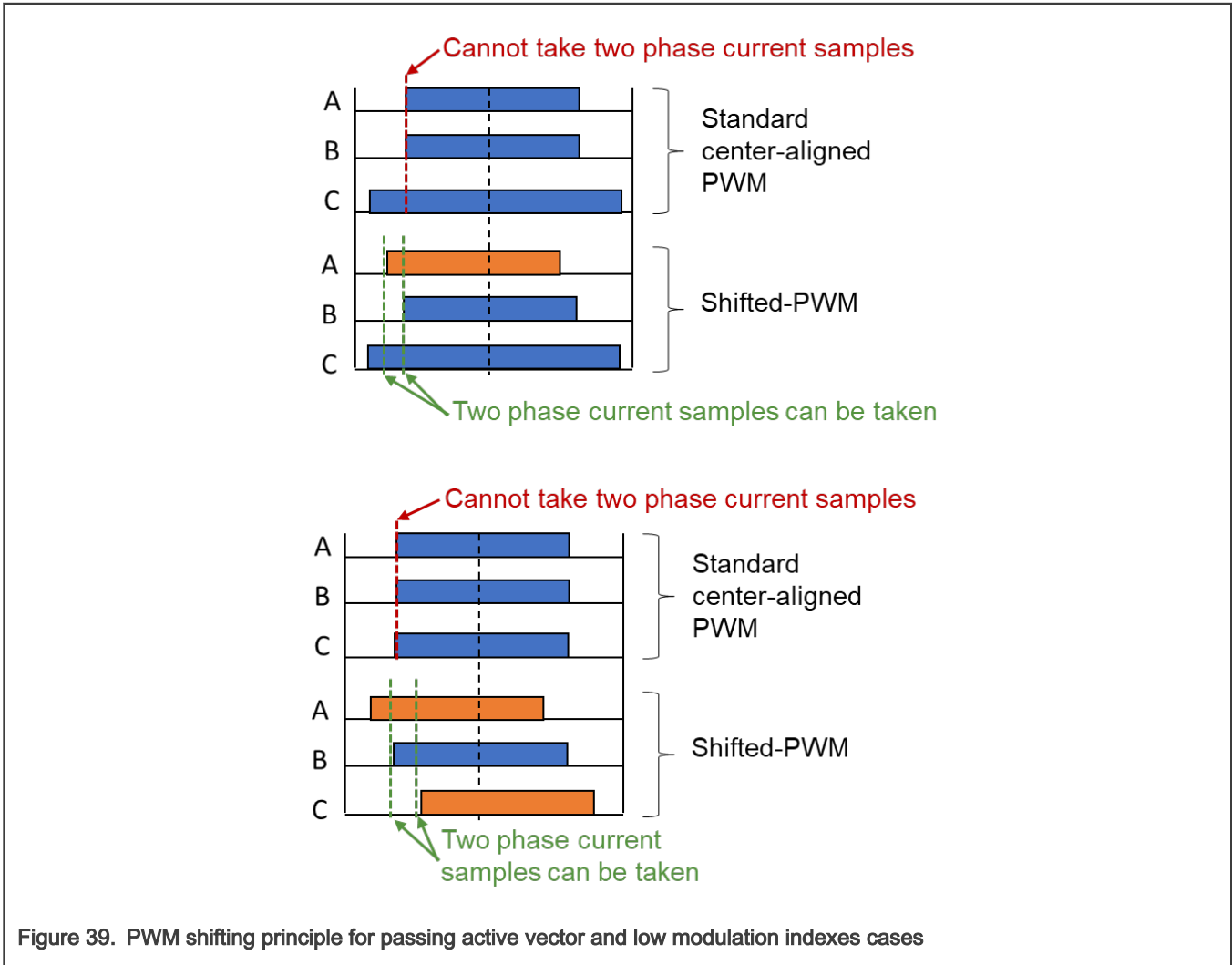
Figure 39. PWM shifting principle for passing active vector and low modulation indexes cases

2.9.1 Available versions

This function is available in the following versions:

- Fractional output - the duty-cycle outputs are the fractional portion of the result. The result is within the range of <0 ; 1). The result may saturate.

Table 10. Function versions

Function name	Input/Output type		Result type
GMCLIB_SvmStdShiffted_F16	Inputs	GMCLIB_2COOR_ALBE_T_F16 *,	void
	Parameter	GMCLIB_SVMSTDSHIFTED_T_F16 *	
	Outputs	GMCLIB_ADC_CONFIG_T_F16 * , GMCLIB_PWM_CONFIG_T_F16 *	
	Standard shiffted space vector modulation with a 16-bit fractional stationary (α-β) input. The parameter is pointed to by an input pointer. The ouputs are pointed to by output pointers. The result type is a void type.		

2.9.1.1 GMCLIB_SVMSTDSHIFTED_T_F16 type description

The input to configure the structure for the GMCLIB_SvmStdShifted function.

Variable name	Input type	Description
f16LowerLim	frac16_t	Low PWM duty-cycle limit. It influences any duty-cycle output in the sDuty structure of the PWM configuration structure. This parameter must be lower than f16UpperLim and of positive value within the range of <0 ; 1) as well. It is set by the user.
f16UpperLim	uint16_t	High PWM duty-cycle limit. It influences any duty-cycle output in the sDuty structure of the PWM configuration structure. This parameter must be higher than f16LowerLim and the value must be within the range of <0 ; 1) as well. It is set by the user.
f16MinT1T2	uint16_t	Minimum T1 or T2 time for a sufficient shift. It influences any shift output in the sShift structure of the PWM configuration structure. This parameter must be of a positive value and within the range of <0 ; 1). It is set by the user.

2.9.2 GMCLIB_ADC_CONFIG_T_F16 type description

The output structure to configure the ADC peripheral.

Variable name	Input type	Description
ui16SectorSVM	uint16_t	The output sector is an integer value within the range of <1 ; 6>. It is calculated by the algorithm and targeted to configure the ADC module.
f16SmplFirstEdge	frac16_t	The output delay for the first edge sample measurement. It is a 16-bit fractional value within the range of <-1 ; 1). It is calculated by the algorithm and targeted to configure the ADC module.
f16SmplSecondEdge	frac16_t	The output delay for the second sample measurement. It is a 16-bit fractional value within the range of <-1 ; 1). It is calculated by the algorithm and targeted to configure the ADC module.
eSmplOnePh	GMCLIB_PHASE_INDEX_T	The output value sets the first sample channel (phase) index. It is assigned from the algorithms and the enumeration value marks phase A, B, or C.
eSmplTwoPh	GMCLIB_PHASE_INDEX_T	The output value sets the second sample channel (phase) index. It is assigned from the algorithms and the enumeration value marks phase A, B, or C.
eCalcPh	GMCLIB_PHASE_INDEX_T	The output value sets the calculated sample channel (phase) index. It is assigned from the algorithms and the enumeration value marks phase A, B, or C.

2.9.3 GMCLIB_PWM_CONFIG_T_F16 type description

The output structure to configure the PWM peripheral.

Variable name		Input type	Description
sDuty	f16A	frac16_t	The phase-A duty cycle is a 16-bit fractional value within the range of <0 ; 1). It is calculated by the algorithms and targeted to configure the PWM module. It may be limited by upper or lower limits.
	f16B	frac16_t	The phase-B duty cycle is a 16-bit fractional value within the range of <0 ; 1). It is calculated by the algorithms and targeted to configure the PWM module. It may be limited by upper or lower limits.
	f16C	frac16_t	The phase-C duty cycle is a 16-bit fractional value within the range of <0 ; 1). It is calculated by the algorithms and targeted to configure the PWM module. It may be limited by upper or lower limits.
sShift	f16A	frac16_t	The phase-A shift is a 16-bit fractional value within the range of <-1 ; 1). It is calculated by the algorithms and targeted to configure the PWM module.
	f16B	frac16_t	The phase-B shift is a 16-bit fractional value within the range of <-1 ; 1). It is calculated by the algorithms and targeted to configure the PWM module.
	f16C	frac16_t	The phase-C shift is a 16-bit fractional value within the range of <-1 ; 1). It is calculated by the algorithms and targeted to configure the PWM module.

2.9.4 GMCLIB_PHASE_INDEX_T type description

The enum data type for labeling the phases in the GMCLIB_ADC_CONFIG_T_F16 structure.

Name	Value	Description
kPhaseA	0U	Phase A
kPhaseB	1U	Phase B
kPhaseC	2U	Phase C

2.9.5 Declaration

The available GMCLIB_SvmStdShifted function has the following declaration:

```
void GMCLIB_SvmStdShifted_F16(const GMCLIB_2COOR_ALBE_T_F16  *psIn, const GMCLIB_SVMSTDSHIFTED_T_F16
*psParam, GMCLIB_ADC_CONFIG_T_F16 *psCfgMeas, GMCLIB_PWM_CONFIG_T_F16 *psCfgPWM)
```

2.9.6 Function use

The use of the GMCLIB_SvmStdShifted function is shown in the following example:

Fixed-point version:

```
#include "gmclib.h"

static GMCLIB_2COOR_ALBE_T_F16 sAlphaBetaIn;
static GMCLIB_SVMSTDSHIFTED_T_F16 sParam;
static GMCLIB_ADC_CONFIG_T_F16 sCfgMeas;
static GMCLIB_PWM_CONFIG_T_F16 sCfgPWM;

void Isr(void);
```

```
void main(void)
{
    /* Alpha, Beta voltage inputs */
    sAlphaBetaIn.f16Alpha = FRAC16(0.25);
    sAlphaBetaIn.f16Beta  = FRAC16(0.1)
    /* Set SvmStdShifted parameter */
    sParam.f16LowerLim = FRAC16(0.01);
    sParam.f16UpperLim = FRAC16(0.9);
    sParam.f16MinT1T2  = FRAC16(0.1);
}
/* Periodical function or interrupt */
void Isr(void)
{
    /* SVM calculation update the output structures sCfgMeas and sCfPWM
    to confugure the ADC and PWM modules */
    GMCLIB_SvmStdShifted_F16(&sAlphaBetaIn, &sParam, &sCfgMeas, &sCfgPWM);
}
```

2.10 GMCLIB_SvmStd

The GMCLIB_SvmStd function calculates the appropriate duty-cycle ratios, which are needed for generation of the given stator-reference voltage vector, using a special standard space vector modulation technique.

The GMCLIB_SvmStd function for calculating the duty-cycle ratios is widely used in modern electric drives. This function calculates the appropriate duty-cycle ratios, which are needed for generating the given stator reference voltage vector, using a special space vector modulation technique, called standard space vector modulation.

The basic principle of the standard space vector modulation technique can be explained using the power stage diagram shown in Figure 1.
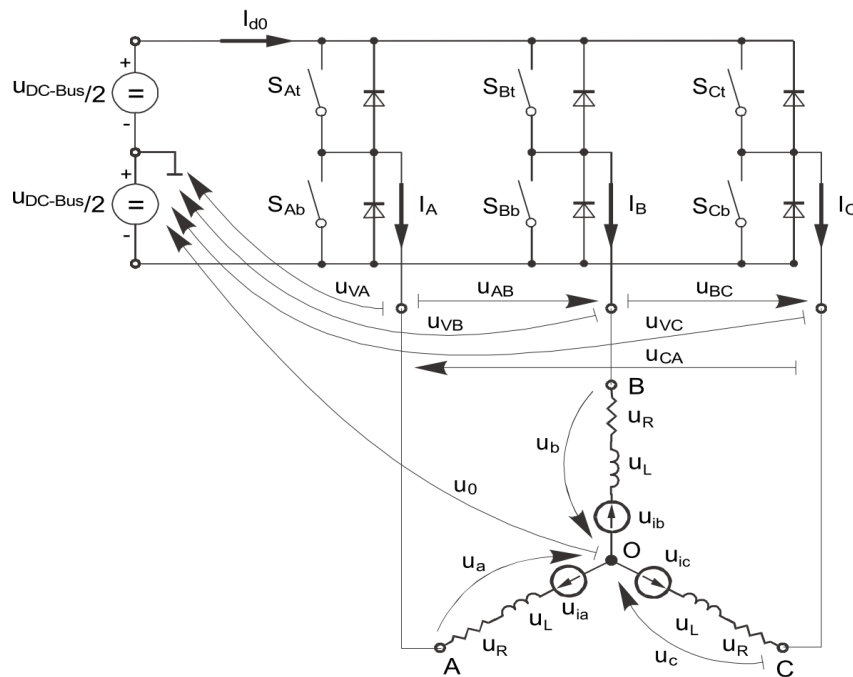
Figure 40. Power stage schematic diagram

The top and bottom switches are working in a complementary mode; for example, if the top switch S_{At} is on, then the corresponding bottom switch S_{Ab} is off, and vice versa. Considering that the 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 of such vector allows for numerical definition of all possible switching states. Phase-to-phase voltages can then be expressed in terms of the following states:

$$
\begin{bmatrix} U_{AB} \\ U_{BC} \\ U_{CA} \end{bmatrix} = U_{DCBus} \begin{bmatrix} 1 & -1 & 0 \\ 0 & 1 & -1 \\ -1 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} a \\ b \\ c \end{bmatrix}
$$

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

Assuming that the motor is completely symmetrical, it is possible to write a matrix equation, which expresses the motor phase voltages shown in GMCLIB_SvmStd_Eq1.

$$
\begin{bmatrix} U_a \\ U_b \\ U_c \end{bmatrix} = \frac{U_{DCBus}}{3} \begin{bmatrix} 2 & -1 & -1 \\ -1 & 2 & -1 \\ -1 & -1 & 2 \end{bmatrix} \cdot \begin{bmatrix} a \\ b \\ c \end{bmatrix}
$$

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

Table 11. 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 the direct-α and the quadrature-β components of the two-phase orthogonal coordinate system, describing the three-phase stator voltages, are expressed using the Clark transformation, arranged in a matrix form:

$$
\begin{bmatrix} U_\alpha \\ U_\beta \end{bmatrix} = \frac{2}{3} \begin{bmatrix} 1 & -\frac{1}{2} & -\frac{1}{2} \\ 0 & \frac{\sqrt{3}}{2} & -\frac{\sqrt{3}}{2} \end{bmatrix} \cdot \begin{bmatrix} U_a \\ U_b \\ U_c \end{bmatrix}
$$

The three-phase stator voltages - U_a, U_b, and U_c, are transformed using the Clark transformation into the direct-α and the quadrature-β components of the two-phase orthogonal coordinate system. The transformation results are listed in Table 2.

Table 12. 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

Table continues on the next page...

Table 12. Switching patterns and space vectors (continued)

A	B	C	U_α	U_β	Vector
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 2 depicts the basic feasible switching states (vectors). There are six nonzero vectors - U_0, U_{60}, U_{120}, U_{180}, U_{240}, and U_{300}, and two zero vectors - O_{111} and O_{000}, usable for switching. Therefore, the principle of the standard space vector modulation lies in applying the appropriate switching states for a certain time, and thus generating a voltage vector identical to the reference one.
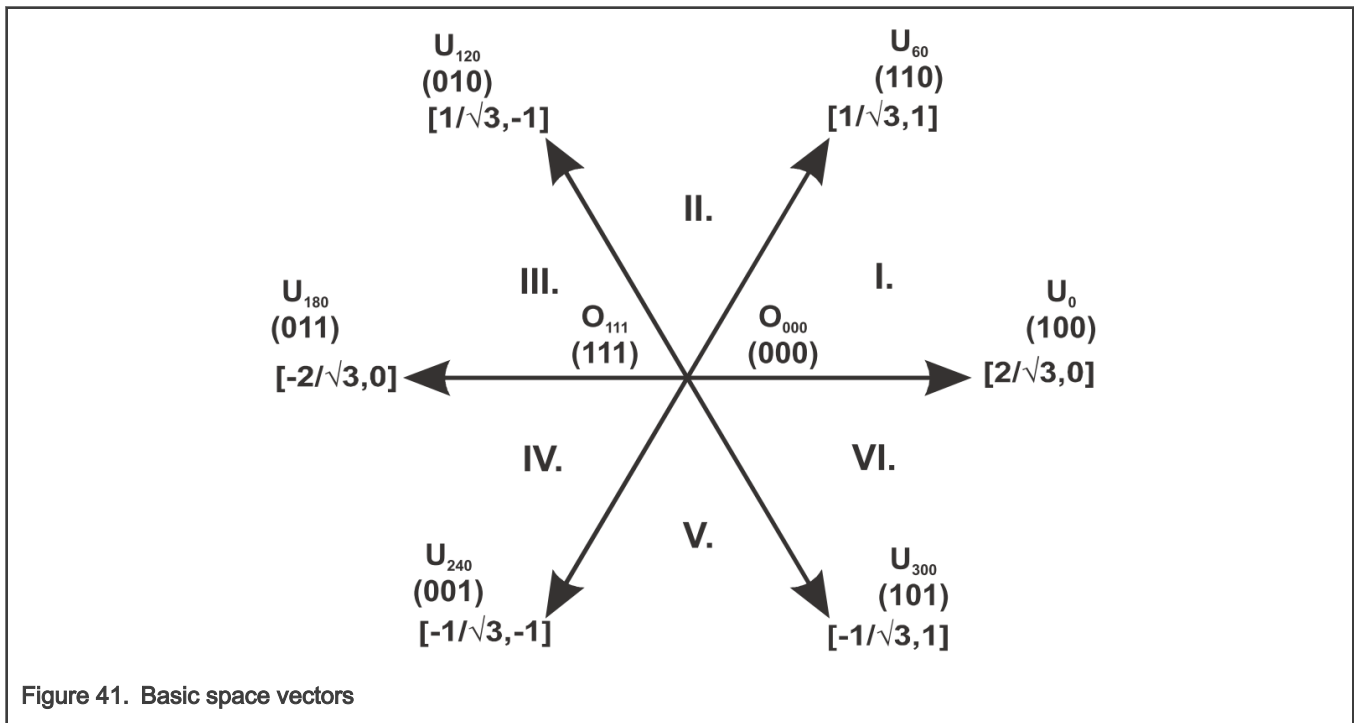
Figure 41. Basic space vectors

Referring to this principle, the 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 3 and Figure 4.
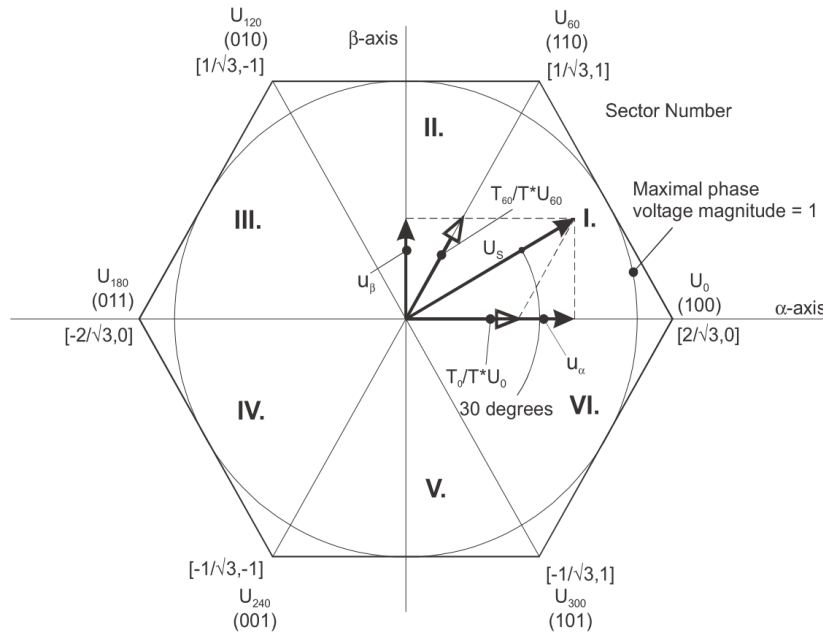
Figure 42. Projection of reference voltage vector in the respective sector

The stator reference voltage vector U_S is phase-advanced by 30° from the direct-α, and thus can be generated with an appropriate combination of the adjacent basic switching states U_0 and U_{60}. These figures also indicate the resultant direct-α and quadrature-β components for space vectors U_0 and U_{60}.
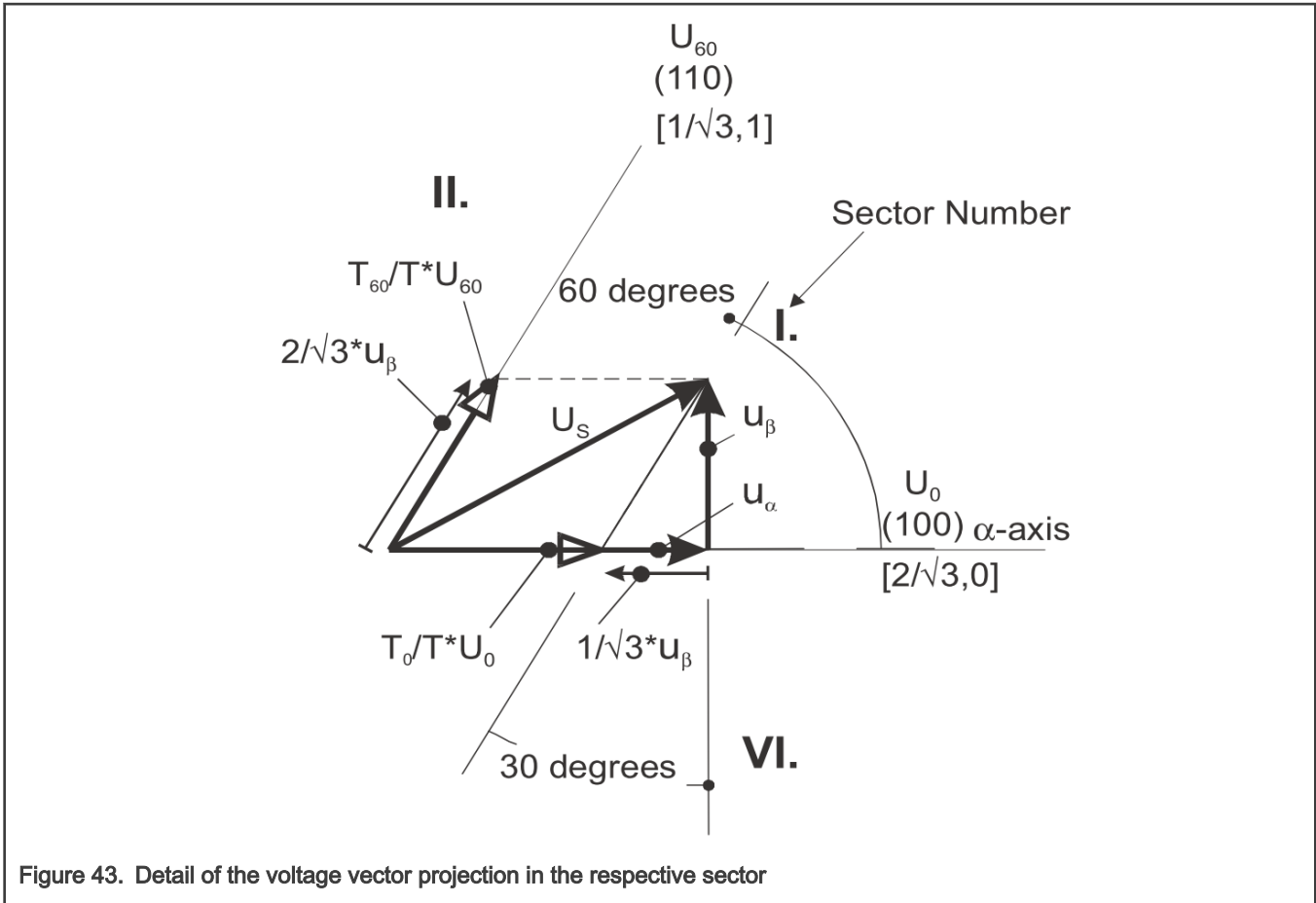
Figure 43. Detail of the voltage vector projection in the respective sector

In this case, the reference stator voltage vector U_S is located in sector I, and can be generated using the appropriate duty-cycle ratios of the basic switching states U_0 and U_{60}. The principal equations concerning this vector location are as follows:

$$T = T_{60} + T_0 + T_{null}$$
$$U_S = \frac{T_{60}}{T} \cdot U_{60} + \frac{T_0}{T} \cdot U_0$$

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

$$u_\beta = \frac{T_{60}}{T} \cdot |U_{60}| \cdot \sin60°$$
$$u_\alpha = \frac{T_0}{T} \cdot |U_0| + \frac{u_\beta}{\tan60°}$$

Considering that normalized magnitudes of basic space vectors are $|U_{60}| = |U_0| = 2 / \sqrt{3}$, and by the substitution of the trigonometric expressions sin 60° and tan 60° by their quantities 2 / √3, and √3, respectively, the GMCLIB_SvmStd_Eq5 can be rearranged for the unknown duty-cycle ratios T_{60} / T and T_0 / T as follows:

$$\frac{T_{60}}{T} = u_\beta$$
$$U_S = \frac{T_{120}}{T} \cdot U_{120} + \frac{T_{60}}{T} \cdot U_{60}$$

Sector II is depicted in GMCLIB_SvmStd_Img5. In this particular case, the reference stator voltage vector U_S is generated using the appropriate duty-cycle ratios of the basic switching states T_{60} and T_{120}. The basic equations describing this sector are as follows:

$$T = T_{120} + T_{60} + T_{null}$$

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

where T_{120} and T_{60} are the respective duty-cycle ratios, for which the basic space vectors U_{120} and U_{60} should be applied within the time period T. T_{null} is the 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 GMCLIB_SvmStd_Img6.
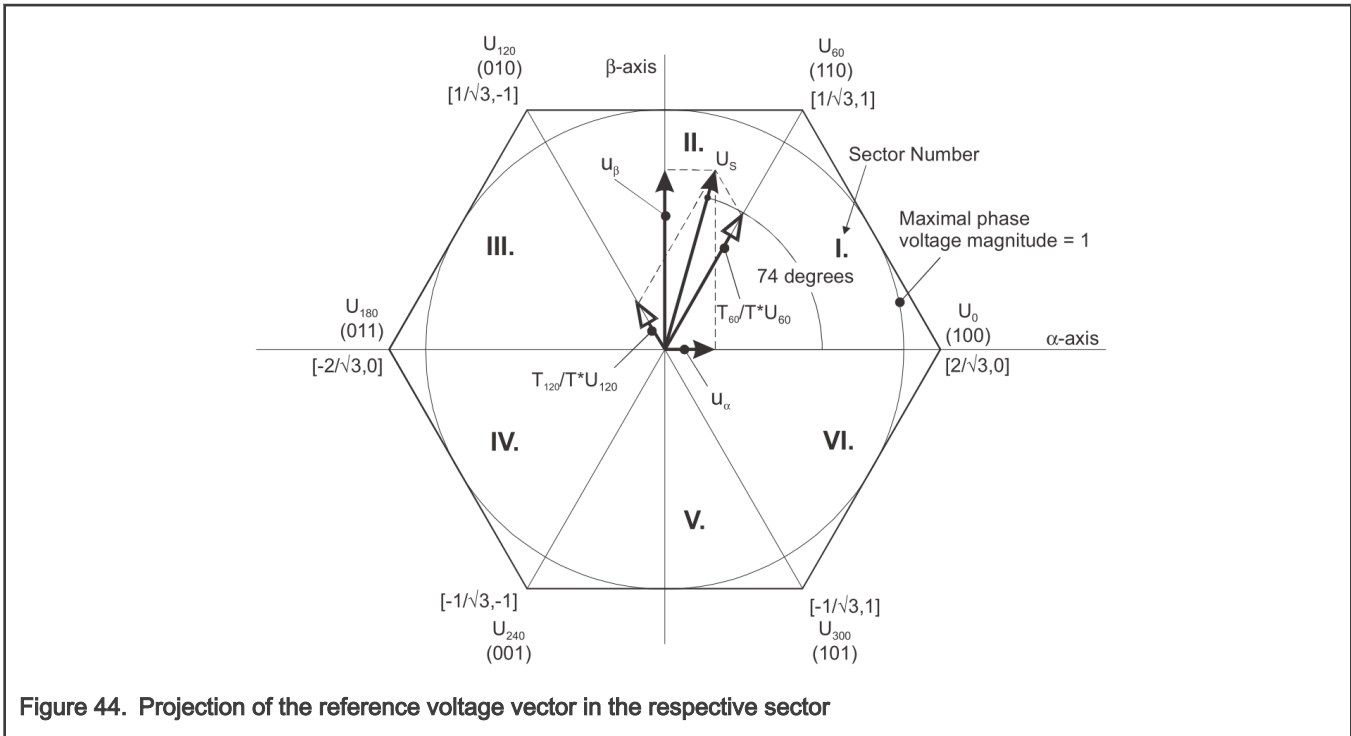
Figure 44. Projection of the reference voltage vector in the respective sector
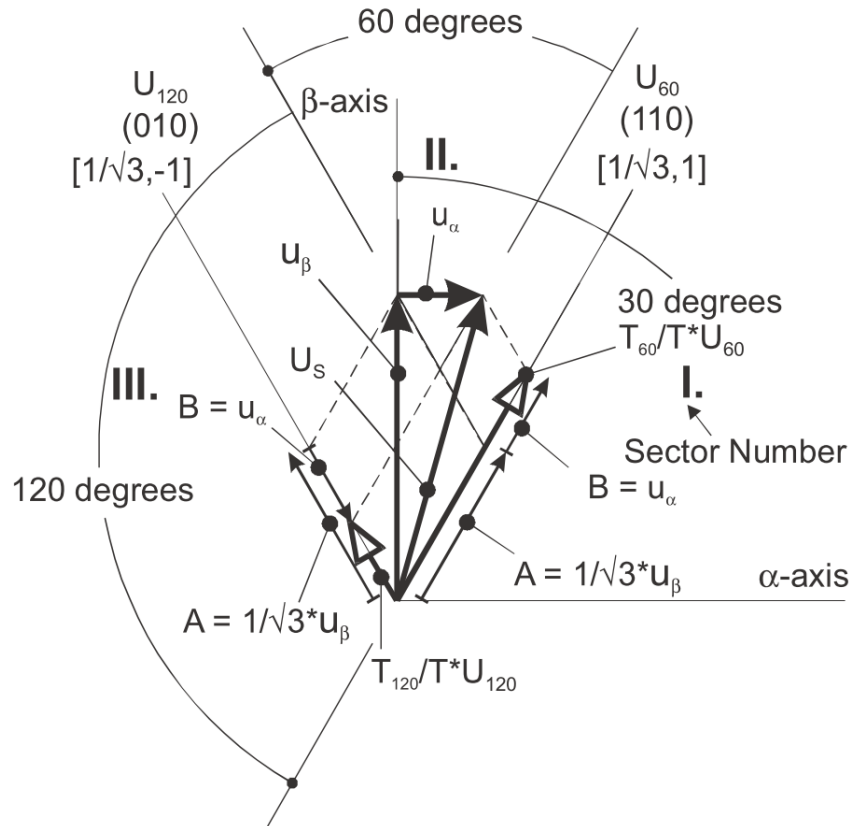
Figure 45. Detail of the voltage vector projection in the respective sector

The equations describing those auxiliary time-duration components are as follows:

$$\frac{\sin 30^{\circ}}{\sin 120^{\circ}} = \frac{A}{u_{\beta}}$$
$$\frac{\sin 60^{\circ}}{\sin 60^{\circ}} = \frac{B}{u_{\alpha}}$$

Equations in GMCLIB_SvmStd_Eq8 have been created using the sine rule.

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 GMCLIB_SvmStd_Eq9 as follows:

$$A = \frac{1}{\sqrt{3}} \cdot u_{\beta}$$
$$B = u_{\alpha}$$

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

$$\frac{T_{120}}{T} \cdot |U_{120}| = (A - B)$$
$$\frac{T_{60}}{T} \cdot |U_{60}| = (A + B)$$

The duty-cycle ratios in the remaining sectors can be derived using the same approach. The resulting equations will be similar to those derived for sector I and sector II.

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

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

To depict the duty-cycle ratios of the basic space vectors for all sectors, we define:

- Three auxiliary variables:

$$X = u_\beta$$

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

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

- Two expressions - t_1 and t_2, which generally represent the duty-cycle ratios of the basic space vectors in the respective sector (for example, for the first sector, t_1 and t_2), represent duty-cycle ratios of the basic space vectors U_{60} and U_0; for the second sector, t_1 and t_2 represent duty-cycle ratios of the basic space vectors U_{120} and U_{60}, and so on.

The expressions t_1 and t_2, in terms of auxiliary variables X, Y, and Z for each sector, are listed in Table 3.

Table 13. 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 using several approaches. The approach discussed here requires the use of modified Inverse Clark transformation to transform the direct-α and quadrature-β components into balanced three-phase quantities u_{ref1}, u_{ref2}, and u_{ref3}, used for straightforward calculation of the sector number, to be shown later.

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

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

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

The modified Inverse Clark transformation projects the quadrature-u_β component into u_{ref1}, as shown in GMCLIB_SvmStd_Img7 and GMCLIB_SvmStd_Img8, whereas voltages generated by the conventional Inverse Clark transformation project the direct-u_α component into u_{ref1}.
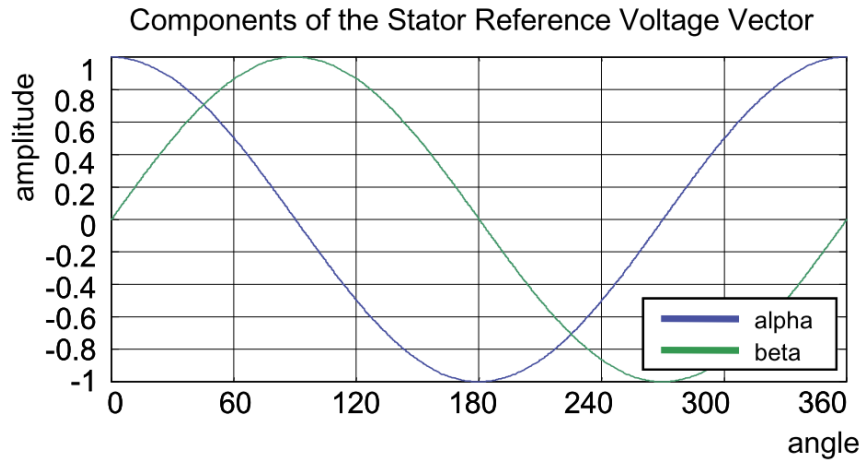
Figure 46. Direct-u_a and quadrature-u_b components of the stator reference voltage

GMCLIB_SvmStd_Img7 depicts the direct-u_α and quadrature-u_β components of the stator reference voltage vector U_S, which were calculated using equations $u_\alpha = \cos \vartheta$ and $u_\beta = \sin \vartheta$, respectively.
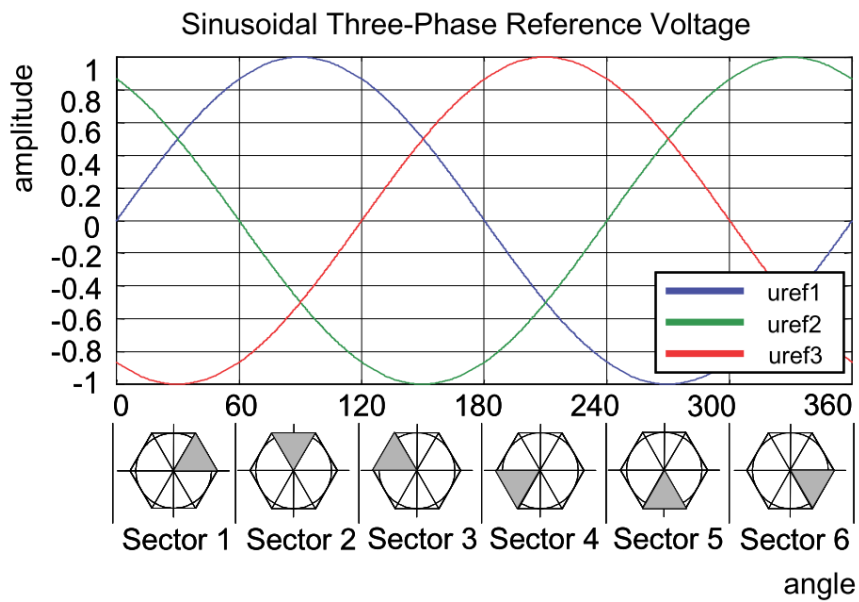
Figure 47. Reference voltages U_{ref1}, U_{ref2}, and U_{ref3}

The sector identification tree shown in GMCLIB_SvmStd_Img9 can be a numerical solution of the approach shown in GMCLIB_SvmStd_Img8.

Figure 48. Identification of the sector number
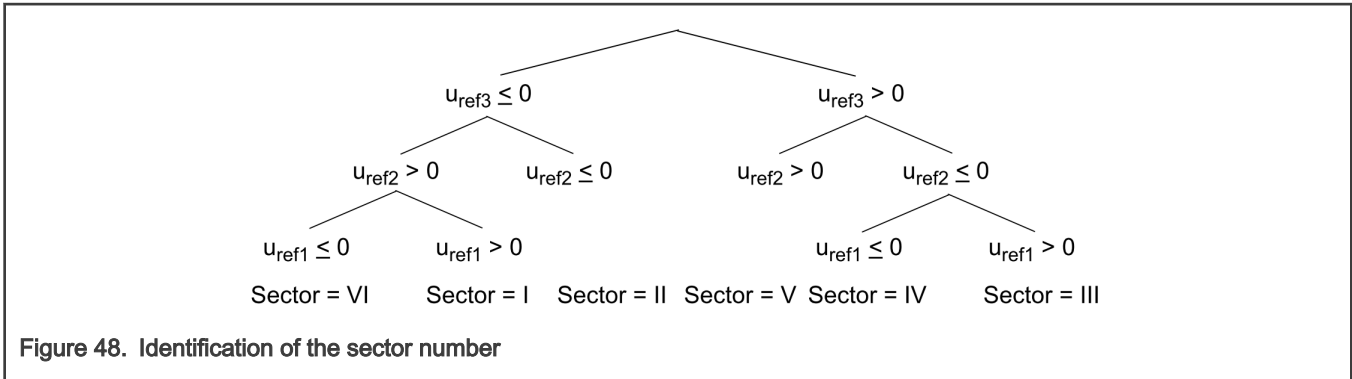
In the worst case, at least three simple comparisons are required to precisely identify the sector of the stator reference voltage vector. For example, if the stator reference voltage vector is located as shown in GMCLIB_SvmStd_Img3, the stator-reference voltage vector is phase-advanced by 30° from the direct α-axis, which results in the positive quantities of u_{ref1} and u_{ref2}, and the negative quantity of u_{ref3}; see GMCLIB_SvmStd_Img8. If these quantities are used as the inputs for the sector identification tree, the product of those comparisons will be sector I. The same approach identifies sector II, if the stator-reference voltage vector is located as shown in GMCLIB_SvmStd_Img5. The variables t_1, t_2, and t_3, which represent the switching duty-cycle ratios of the respective three-phase system, are calculated according to the following equations:

$$t_1 = \frac{T - t\_1 - t\_2}{2}$$
$$t_2 = t_1 + t\_1$$
$$t_3 = t_2 + t\_2$$

where T is the switching period, and t_1 and t_2 are the duty-cycle ratios of the basic space vectors given for the respective sector; Table 3, GMCLIB_SvmStd_Eq3, and GMCLIB_SvmStd_Eq15 are specific solely to the standard space vector modulation technique; 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 a view of the position of the stator reference voltage vector; see .

Table 14. 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 of applying the basic voltage vectors U_{XXX} and O_{XXX} for certain time, in such a way that the main vector generated by the pulse width modulation approach for the period T is equal to the original stator reference voltage vector U_S. This provides a great variability of arrangement of the basic vectors during the PWM period T. These vectors might be arranged either to lower the 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 by comparing the threshold levels pwm_a, pwm_b, and pwm_c with a free-running up-down counter. The timer counts to one, and then down to zero. 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 GMCLIB_SvmStd_Img10.

Figure 49. Standard space vector modulation technique — center-aligned PWM
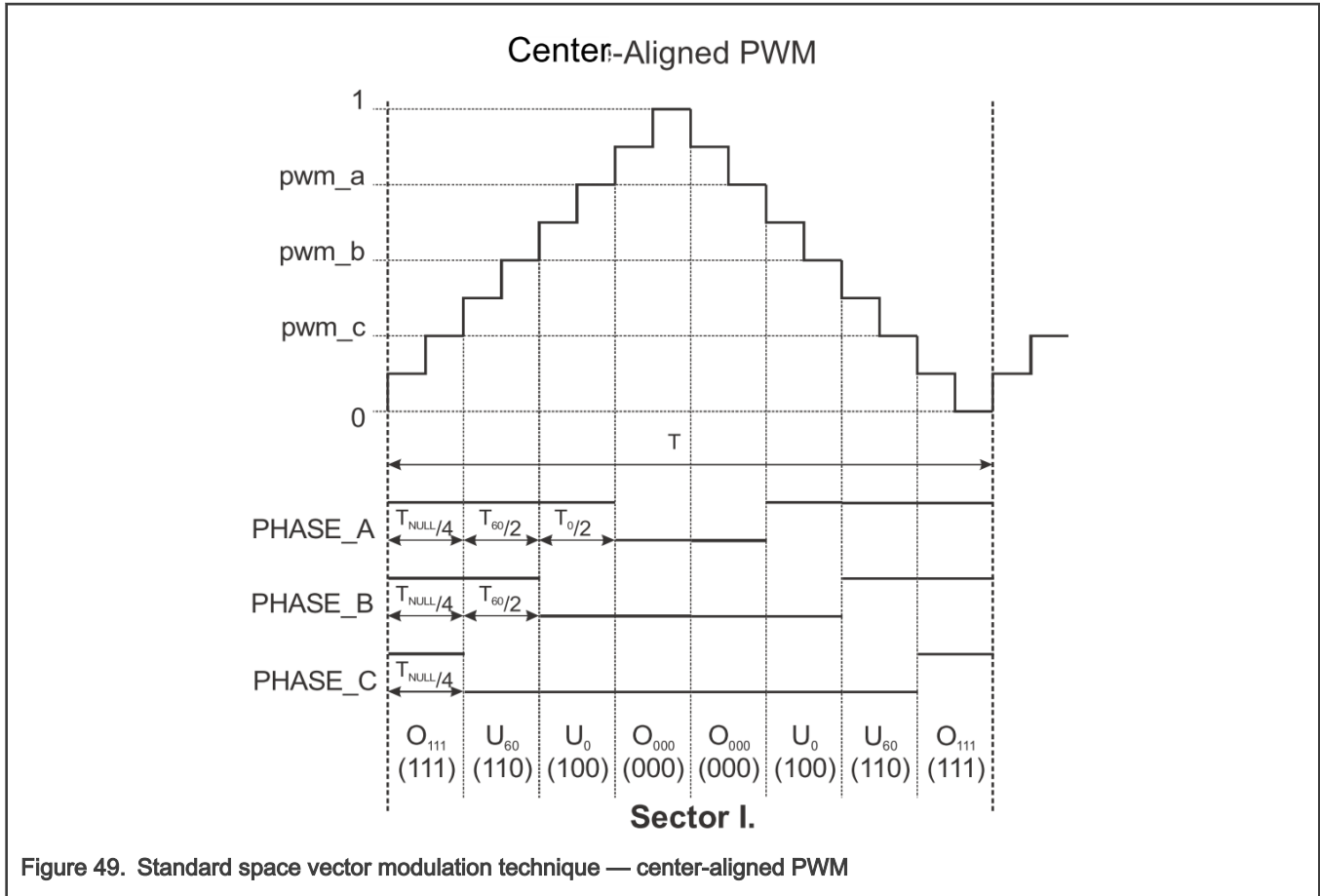
GMCLIB_SvmStd_Img11 shows the waveforms of the duty-cycle ratios, calculated using standard space vector modulation.

For the accurate calculation of the duty-cycle ratios, direct-α, and quadrature-β components of the stator reference voltage vector, it must be considered that the duty cycle cannot be higher than one (100 %); in other words, the assumption

$$\sqrt{\alpha^2 + \beta^2} \leq 1$$

must be met.

Figure 50. Standard space vector modulation technique

2.10.1 Available versions

This function is available in the following versions:

- Fractional output - the output is the fractional portion of the result; the result is within the range <0 ; 1). The result may saturate.

The available versions of the GMCLIB_SvmStd function are shown in the following table.

Table 15. Function versions

Function name	Input type	Output type	Result type
GMCLIB_SvmStd_F16	GMCLIB_2COOR_ALBE_T_F16 *	GMCLIB_3COOR_T_F16 *	uint16_t
	Standard space vector modulation with a 16-bit fractional stationary (α-β) input and a 16-bit fractional three-phase output. The result type is a 16-bit unsigned integer, which indicates the actual SVM sector. The input is within the range <-1 ; 1); the output duty cycle is within the range <0 ; 1). The output sector is an integer value within the range <1 ; 6>.		

2.10.2 Declaration

The available GMCLIB_SvmStd functions have the following declarations:

```
uint16_t GMCLIB_SvmStd_F16(const GMCLIB_2COOR_ALBE_T_F16 *psIn, GMCLIB_3COOR_T_F16 *psOut)
```

2.10.3 Function use

The use of the GMCLIB_SvmStd function is shown in the following example:

```
#include "gmclib.h"

static uint16_t u16Sector;
static GMCLIB_2COOR_ALBE_T_F16 sAlphaBeta;
static GMCLIB_3COOR_T_F16 sAbc;

void Isr(void);

void main(void)
{
  /* Alpha, Beta structure initialization */
  sAlphaBeta.f16Alpha = FRAC16(0.0);
  sAlphaBeta.f16Beta = FRAC16(0.0);
}

/* Periodical function or interrupt */
void Isr(void)
{
  /* SVM calculation */
  u16Sector = GMCLIB_SvmStd_F16(&sAlphaBeta, &sAbc);
}
```

2.11 GMCLIB_SvmIct

The GMCLIB_SvmIct function calculates the appropriate duty-cycle ratios, which are needed for generation of the given stator-reference voltage vector using the general sinusoidal modulation technique.

The GMCLIB_SvmIct function calculates the appropriate duty-cycle ratios, needed for generation of the given stator reference voltage vector using the conventional Inverse Clark transformation. Finding the sector in which the reference stator voltage vector U_S resides is similar to GMCLIB_SvmStd. This is achieved by first converting the direct-α and the quadrature-β components of the reference stator voltage vector U_S into the balanced three-phase quantities u_{ref1}, u_{ref2}, and u_{ref3} using the modified Inverse Clark transformation:

$$u_{ref1} = u_\beta$$
$$u_{ref2} = \frac{-u_\beta + \sqrt{3} \cdot u_\alpha}{2}$$
$$u_{ref3} = \frac{-u_\beta - \sqrt{3} \cdot u_\alpha}{2}$$

The calculation of the sector number is based on comparing the three-phase reference voltages u_{ref1}, u_{ref2}, and u_{ref3} with zero. This computation is described by the following set of rules:

$$a = \begin{cases} 1, & u_{ref1} > 0 \\ 0, & \text{else} \end{cases}$$
$$b = \begin{cases} 2, & u_{ref2} > 0 \\ 0, & \text{else} \end{cases}$$
$$c = \begin{cases} 4, & u_{ref3} > 0 \\ 0, & \text{else} \end{cases}$$

After passing these rules, the modified sector numbers are then derived using the following formula:

$$sector* = a + b + c$$

The sector numbers determined by this formula must be further transformed to correspond to those determined by the sector identification tree. The transformation which meets this requirement is shown in the following table:

Table 16. Transformation of the sectors

Sector*	1	2	3	4	5	6
Sector	2	6	1	4	3	5

Use the Inverse Clark transformation for transforming values such as flux, voltage, and current from an orthogonal rotating coordination system (u_α, u_β) to a three-phase rotating coordination system (u_a, u_b, and u_c). The original equations of the Inverse Clark transformation are scaled here to provide the duty-cycle ratios in the range <0 ; 1). These scaled duty cycle ratios pwm_a, pwm_b, and pwm_c can be used directly by the registers of the PWM block.

$$pwm\_a = 0.5 + \frac{u_\alpha}{2}$$

$$pwm\_b = 0.5 + \frac{-u_\alpha + \sqrt{3} \cdot u_\beta}{4}$$

$$pwm\_c = 0.5 + \frac{-u_\alpha - \sqrt{3} \cdot u_\beta}{4}$$

The following figure shows the waveforms of the duty-cycle ratios calculated using the Inverse Clark transformation.

Figure 51. Inverse Clark transform modulation technique
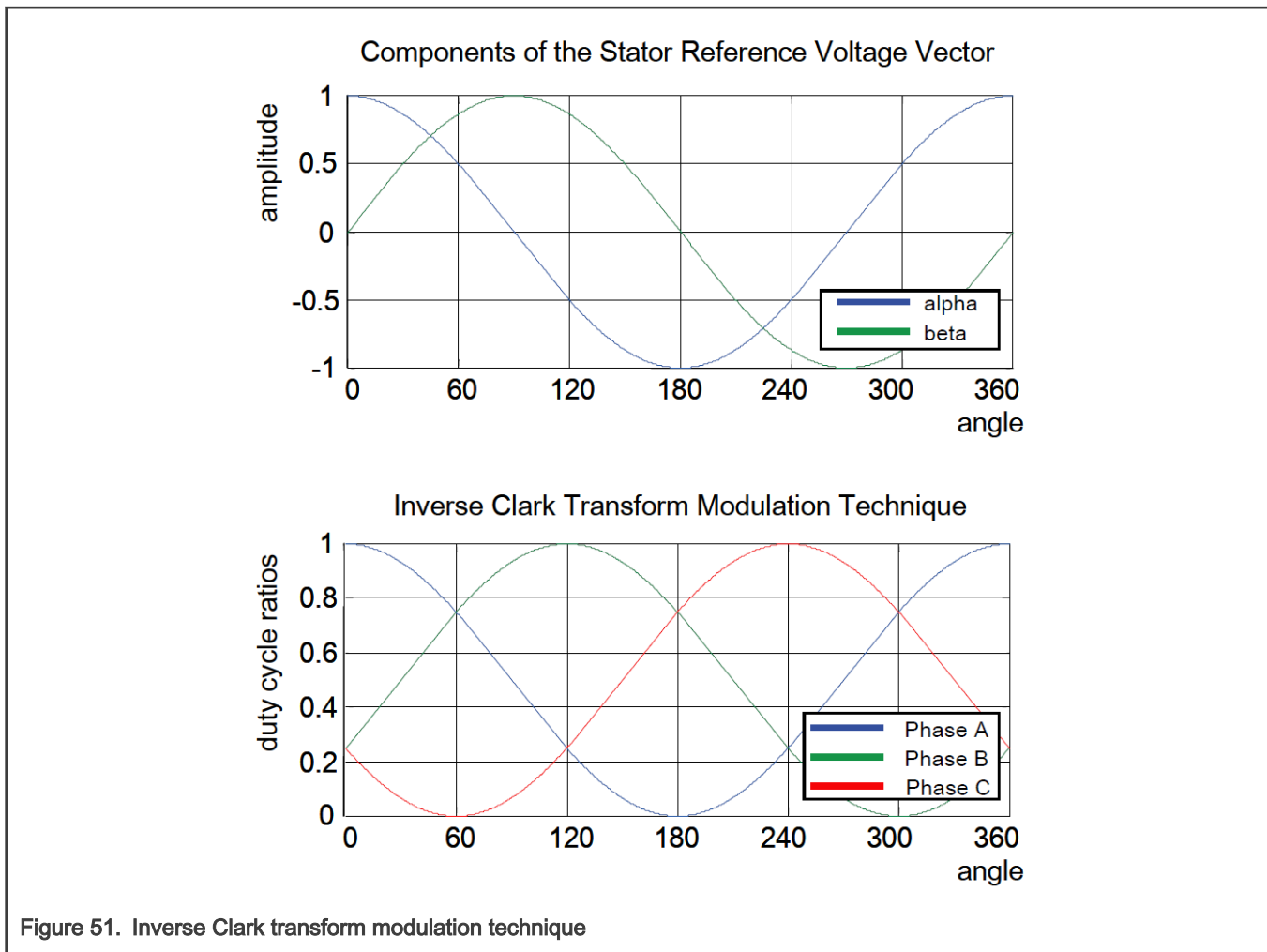
For an accurate calculation of the duty-cycle ratios and the direct-α and quadrature-β components of the stator reference voltage vector, the duty cycle cannot be higher than one (100 %); in other words, the assumption
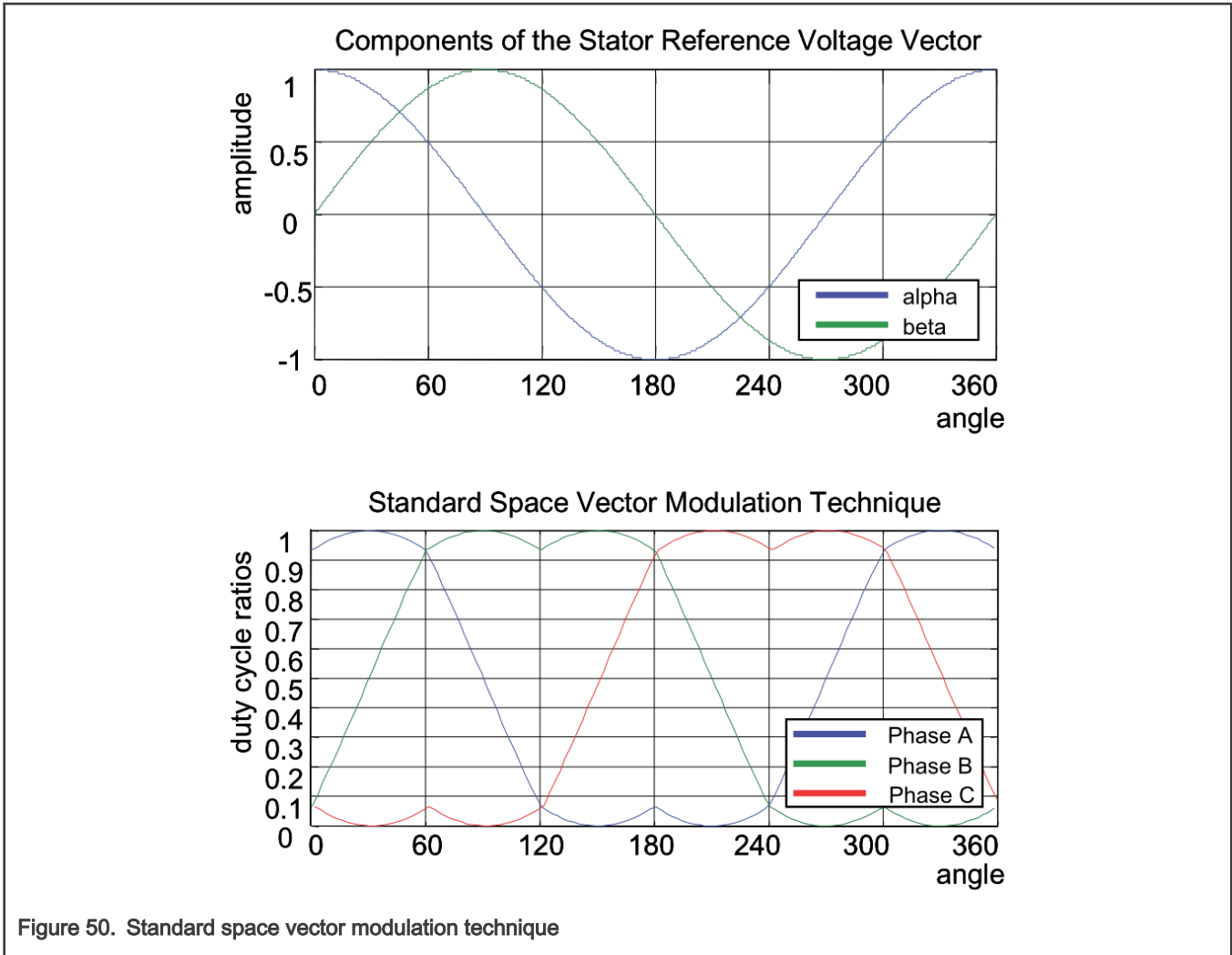
$$\sqrt{\alpha^2 + \beta^2} \leq 1$$

must be met.

2.11.1 Available versions

This function is available in the following versions:

- Fractional output - the output is the fractional portion of the result; the result is within the range <0 ; 1). The result may saturate.

The available versions of the GMCLIB_SvmIct function are shown in the following table:

Table 17. Function versions

Function name	Input type	Output type	Result type
GMCLIB_SvmIct_F16	GMCLIB_2COOR_ALBE_T_F16 *	GMCLIB_3COOR_T_F16 *	uint16_t
	General sinusoidal space vector modulation with a 16-bit fractional stationary (α-β) input and a 16-bit fractional three-phase output. The result type is a 16-bit unsigned integer, which indicates the actual SVM sector. The input is within the range <-1 ; 1); the output duty cycle is within the range <0 ; 1). The output sector is an integer value within the range <1 ; 6>.		

2.11.2 Declaration

The available GMCLIB_SvmIct functions have the following declarations:

```
uint16_t GMCLIB_SvmIct_F16(const GMCLIB_2COOR_ALBE_T_F16 *psIn, GMCLIB_3COOR_T_F16 *psOut)
```

2.11.3 Function use

The use of the GMCLIB_SvmIct function is shown in the following example:

```
#include "gmclib.h"

static uint16_t u16Sector;
static GMCLIB_2COOR_ALBE_T_F16 sAlphaBeta;
static GMCLIB_3COOR_T_F16 sAbc;

void Isr(void);

void main(void)
{
  /* Alpha, Beta structure initialization */
  sAlphaBeta.f16Alpha = FRAC16(0.0);
  sAlphaBeta.f16Beta = FRAC16(0.0);
}

/* Periodical function or interrupt */
void Isr(void)
{
  /* SVM calculation */
  u16Sector = GMCLIB_SvmIct_F16(&sAlphaBeta, &sAbc);
}
```

2.12 GMCLIB_SvmU0n

The GMCLIB_SvmU0nfunction calculates the appropriate duty-cycle ratios, which are needed for generation of the given stator-reference voltage vector using the general sinusoidal modulation technique.

The GMCLIB_SvmU0n function for calculating of duty-cycle ratios is widely used in modern electric drives. This function calculates the appropriate duty-cycle ratios, which are needed for generating the given stator reference voltage vector using a special space vector modulation technique called space vector modulation with O_{000} nulls, where only one type of null vector O_{000} is used (all bottom switches are turned on in the invertor).

The derivation approach of the space vector modulation technique with O_{000} nulls is in many aspects identical to the approach presented in GMCLIB_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 the respective phases:

$$t_1 = 0$$
$$t_2 = t_1 + t\_1$$
$$t_3 = t_2 + t\_2$$

where T is the switching period, and t_1 and t_2 are the duty-cycle ratios of the basic space vectors that are defined for the respective sector in Table 2-7.

The generally used center-aligned PWM is discussed briefly in the following sections. 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 GMCLIB_SvmU0n_Img1).

Figure 52. Space vector modulation technique with O_{000} nulls — center-aligned PWM

Figure GMCLIB_SvmU0n_Img1 shows calculated waveforms of the duty cycle ratios using space vector modulation with O_{000} nulls.

For an accurate calculation of the duty-cycle ratios, direct-α, and quadrature-β components of the stator reference voltage vector, consider that the duty cycle cannot be higher than one (100 %); in other words, the assumption

$$\sqrt{\alpha^2 + \beta^2} \leq 1$$

must be met.

Figure 53. Space vector modulation technique with O_{000} nulls

2.12.1 Available versions

This function is available in the following versions:

- Fractional output - the output is the fractional portion of the result; the result is within the range <0 ; 1). The result may saturate.

The available versions of the GMCLIB_SvmU0n function are shown in the following table:

Table 18. Function versions

Function name	Input type	Output type	Result type
GMCLIB_SvmU0n_F16	GMCLIB_2COOR_ALBE_T_F16 *	GMCLIB_3COOR_T_F16 *	uint16_t
	General sinusoidal space vector modulation with a 16-bit fractional stationary (α-β) input, and a 16-bit fractional three-phase output. The result type is a 16-bit unsigned integer, which indicates the actual		

Table continues on the next page...

Table 18. Function versions

Function name	Input type	Output type	Result type
	SVM sector. The input is within the range <-1 ; 1); the output duty cycle is within the range <0 ; 1). The output sector is an integer value within the range <1 ; 6>.		

2.12.2 Declaration

The available GMCLIB_SvmU0n functions have the following declarations:

```
uint16_t GMCLIB_SvmU0n_F16(const GMCLIB_2COOR_ALBE_T_F16 *psIn, GMCLIB_3COOR_T_F16 *psOut)
```

2.12.3 Function use

The use of the GMCLIB_SvmU0n function is shown in the following example:

```
#include "gmclib.h"

static uint16_t u16Sector;
static GMCLIB_2COOR_ALBE_T_F16 sAlphaBeta;
static GMCLIB_3COOR_T_F16 sAbc;

void Isr(void);

void main(void)
{
  /* Alpha, Beta structure initialization */
  sAlphaBeta.f16Alpha = FRAC16(0.0);
  sAlphaBeta.f16Beta = FRAC16(0.0);
}

/* Periodical function or interrupt */
void Isr(void)
{
  /* SVM calculation */
  u16Sector = GMCLIB_SvmU0n_F16(&sAlphaBeta, &sAbc);
}
```

2.13 GMCLIB_SvmU7n

The GMCLIB_SvmU7n function calculates the appropriate duty-cycle ratios, which are needed for generation of the given stator-reference voltage vector, using the general sinusoidal modulation technique.

The GMCLIB_SvmU7n function for calculating the duty-cycle ratios is widely used in modern electric drives. This function calculates the appropriate duty-cycle ratios, which are needed for generating the given stator reference voltage vector using a special space vector modulation technique called space vector modulation with O_{111} nulls, where only one type of null vector O_{111} is used (all top switches are turned on in the invertor).

The derivation approach of the space vector modulation technique with O_{111} nulls is identical (in many aspects) to the approach presented in GMCLIB_SvmStd. However, a distinct difference lies in the definition of variables t_1, t_2, and t_3 that represent switching duty-cycle ratios of the respective phases:

$$t_1 = T - t\_1 - t\_2$$
$$t_2 = t_1 + t\_1$$
$$t_3 = t_2 + t\_2$$

where T is the switching period, and t_1 and t_2 are the duty-cycle ratios of the basic space vectors defined for the respective sector in Table 2-7.

The generally-used center-aligned PWM is discussed briefly in the following sections. Generating the center-aligned PWM pattern is accomplished 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 GMCLIB_SvmU7n_Img1).
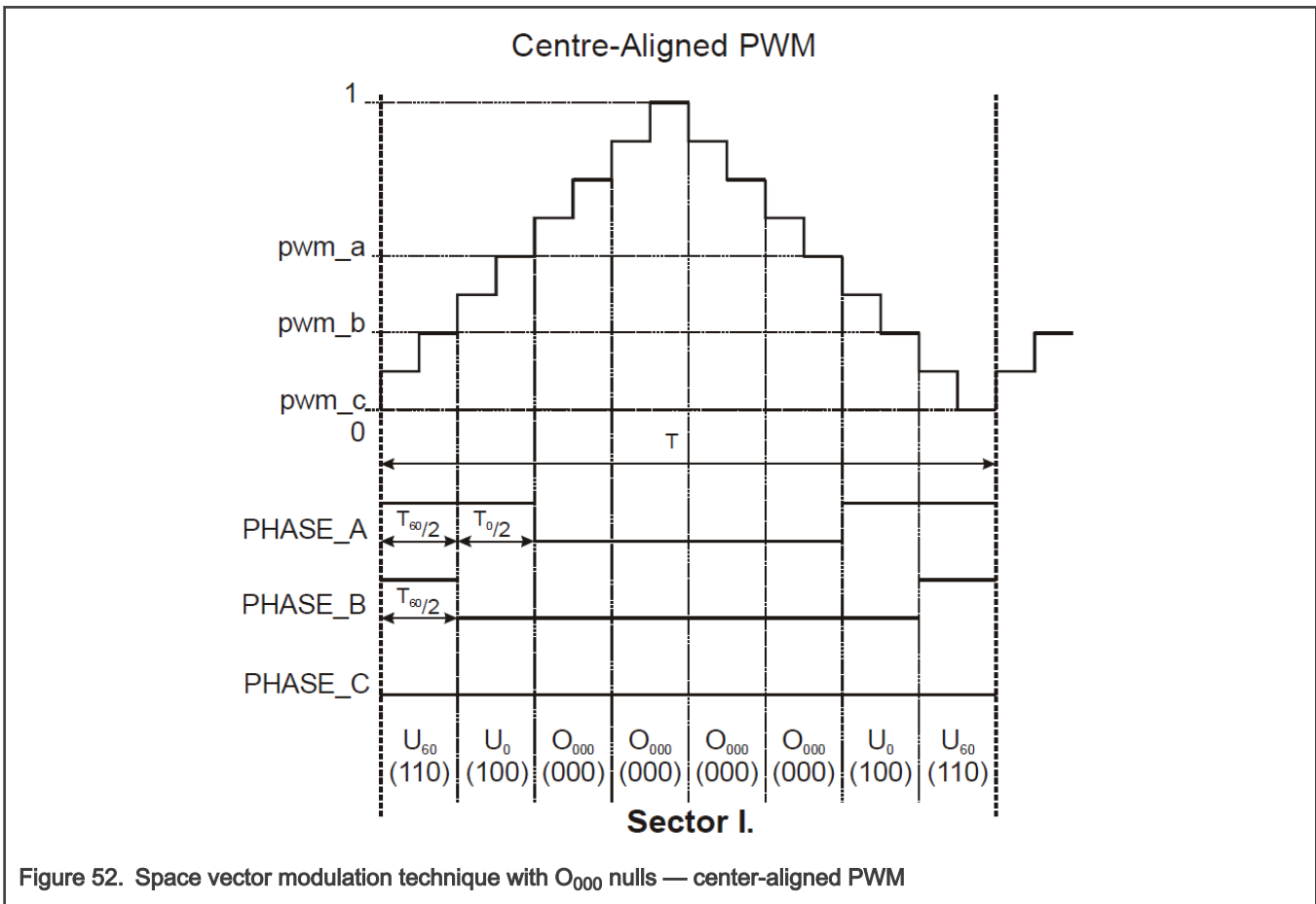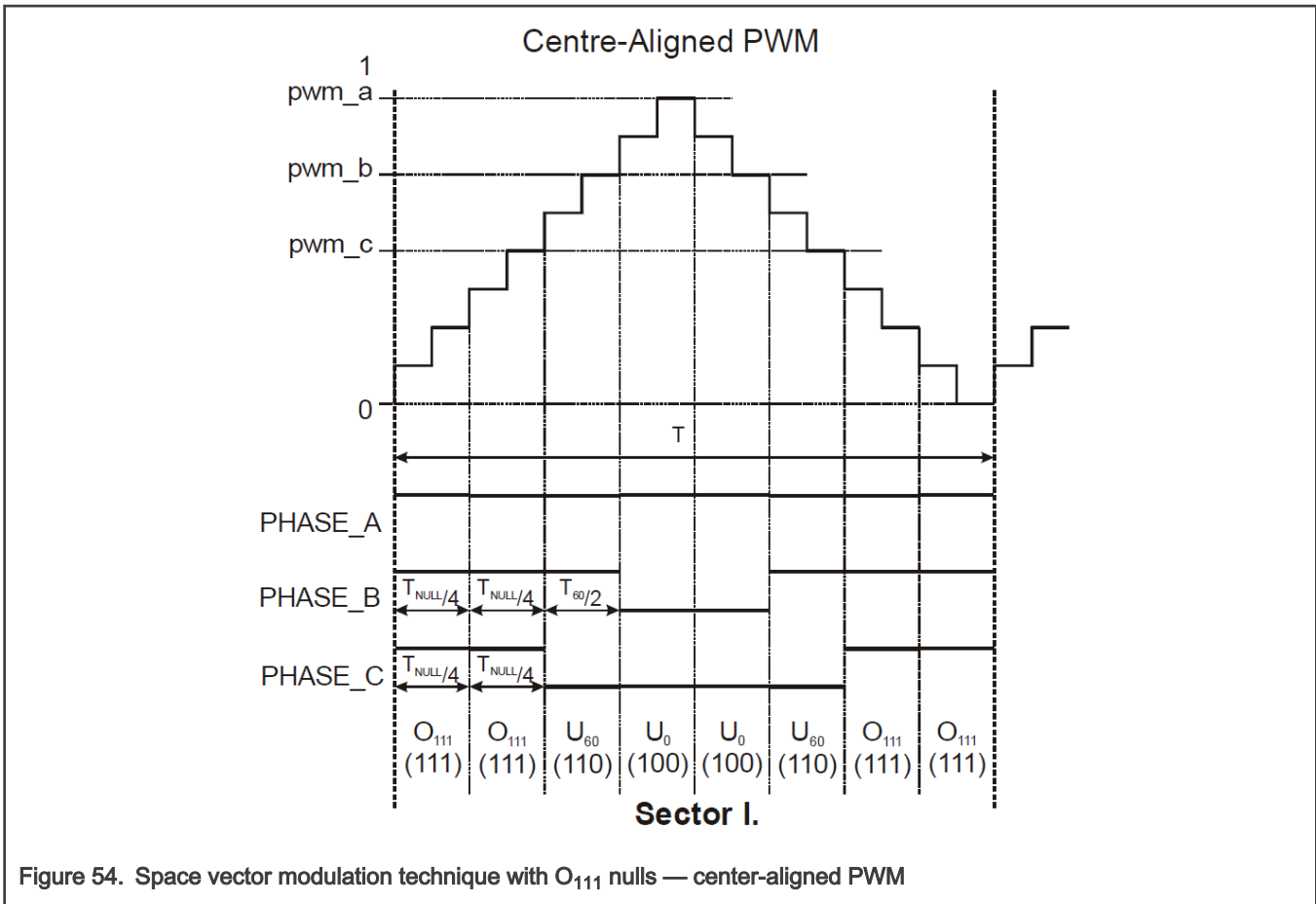

Figure 54. Space vector modulation technique with O_{111} nulls — center-aligned PWM

Figure GMCLIB_SvmU7n_Img1 shows calculated waveforms of the duty-cycle ratios using Space Vector Modulation with O_{111} nulls.

For an accurate calculation of the duty-cycle ratios, direct-α, and quadrature-β components of the stator reference voltage vector, it must be considered that the duty cycle cannot be higher than one (100 %); in other words, the assumption
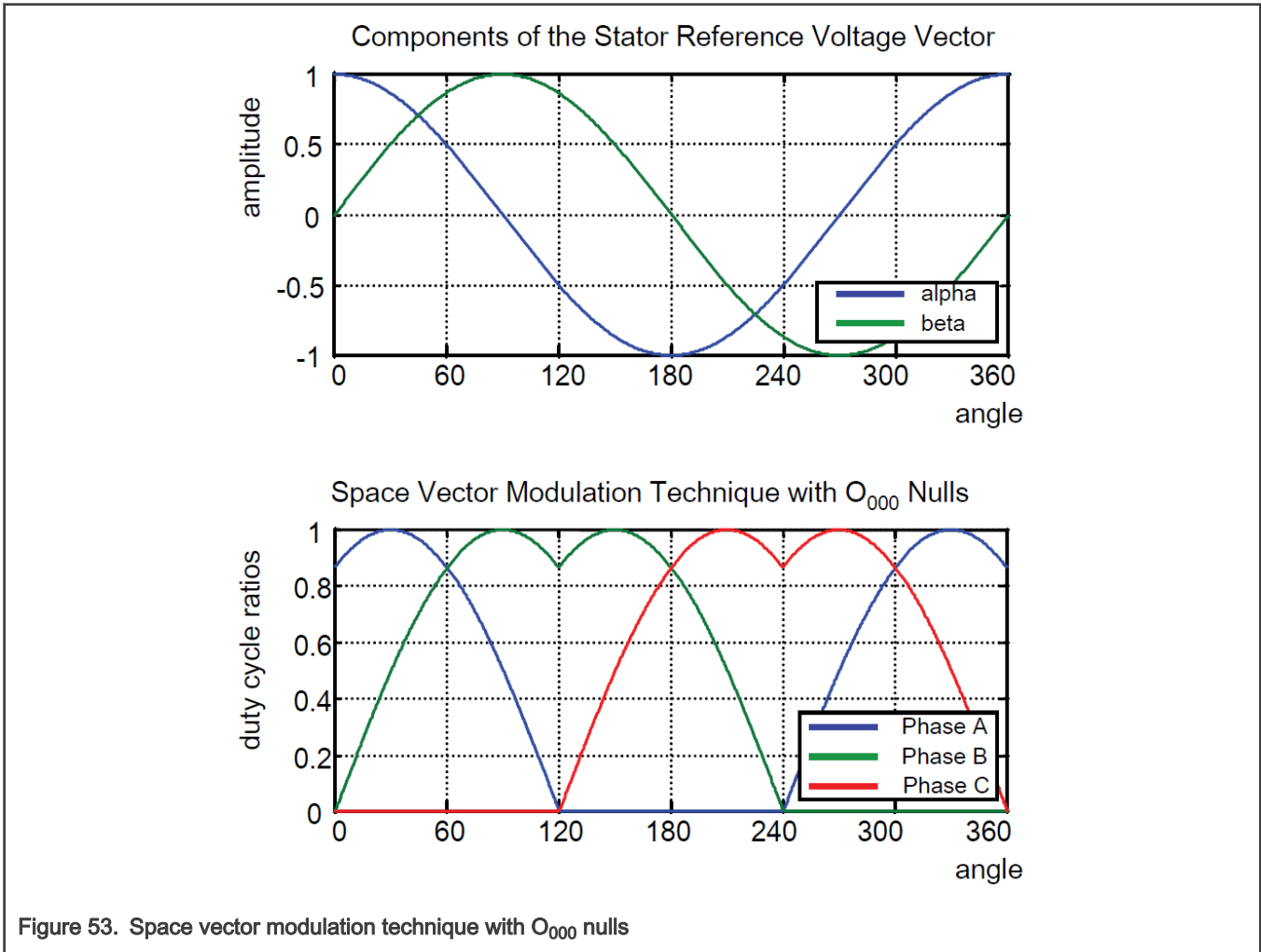
$$\sqrt{\alpha^2 + \beta^2} \leq 1$$

must be met.
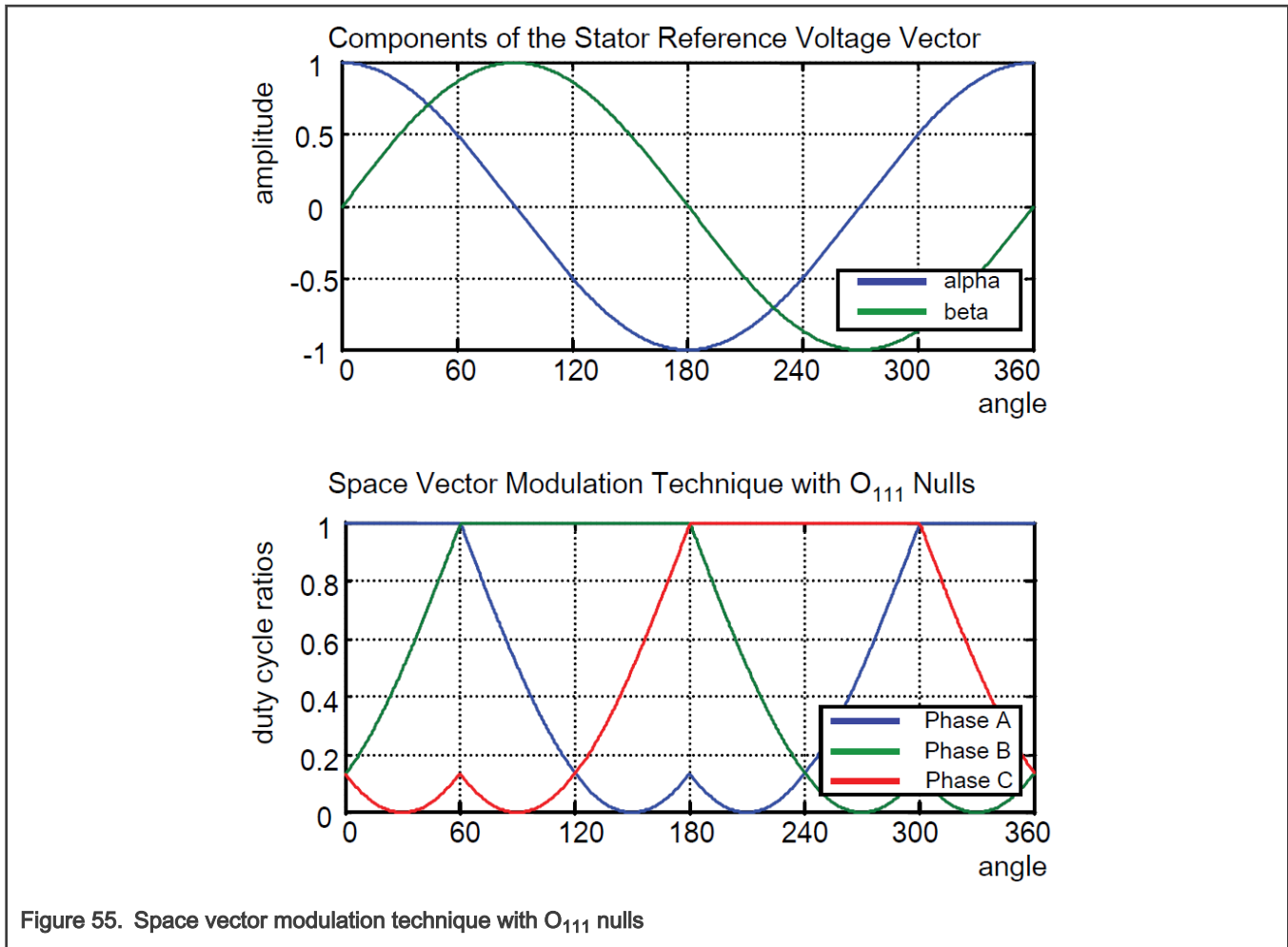
Figure 55. Space vector modulation technique with O_{111} nulls

2.13.1 Available versions

This function is available in the following versions:

 • Fractional output - the output is the fractional portion of the result; the result is within the range <0 ; 1). The result may saturate.

The available versions of the GMCLIB_SvmU7n function are shown in the following table:

Table 19. Function versions

Function name	Input type	Output type	Result type
GMCLIB_SvmU7n_F16	GMCLIB_2COOR_ALBE_T_F16 *	GMCLIB_3COOR_T_F16 *	uint16_t
	General sinusoidal space vector modulation with a 16-bit fractional stationary (α-β) input and a 16-bit fractional three-phase output. The result type is a 16-bit unsigned integer, which indicates the actual SVM sector. The input is within the range <-1 ; 1); the output duty cycle is within the range <0 ; 1). The output sector is an integer value within the range <1 ; 6>.		

2.13.2 Declaration

The available GMCLIB_SvmU7n functions have the following declarations:

```
uint16_t GMCLIB_SvmU7n_F16(const GMCLIB_2COOR_ALBE_T_F16 *psIn, GMCLIB_3COOR_T_F16 *psOut)
```

2.13.3 Function use

The use of the GMCLIB_SvmU7n function is shown in the following example:

```
#include "gmclib.h"

static uint16_t u16Sector;
static GMCLIB_2COOR_ALBE_T_F16 sAlphaBeta;
static GMCLIB_3COOR_T_F16 sAbc;

void Isr(void);

void main(void)
{
  /* Alpha, Beta structure initialization */
  sAlphaBeta.f16Alpha = FRAC16(0.0);
  sAlphaBeta.f16Beta = FRAC16(0.0);
}

/* Periodical function or interrupt */
void Isr(void)
{
  /* SVM calculation */
  u16Sector = GMCLIB_SvmU7n_F16(&sAlphaBeta, &sAbc);
}
```

2.14 GMCLIB_SvmDpwm

The GMCLIB_SvmDpwmfunction calculates the appropriate duty-cycle ratios needed for the generation of the given stator-reference voltage vector using the general non-sinusoidal modulation technique. The GMCLIB_SvmDpwmfunction is a subset of the GMCLIB_SvmExDpwmfunction and includes a power factor angle input. Both functions are identical if φ = 0.

The GMCLIB_SvmDpwm function belongs to the discontinuous PWM modulation techniques for 3-phase voltage inverters. The advantages of the discontinuous PWM technique are lower switching loses, but, on the other hand, it can cause higher harmonic distortion at low modulation indexes. The current sensing at low modulation indexes is more complicated and less precise when compared with the symmetrical modulation techniques like GMCLIB_SvmStd. Therefore, the discontinuous and continous SVM are usually combined together.

Finding the sector in which the reference stator voltage vector U_S resides is similar to GMCLIB_SvmStd. This is achieved by converting the direct-α and quadrature-β components of the reference stator voltage vector U_S into the balanced 3-phase quantities u_{ref1}, u_{ref2}, and u_{ref3} using the modified Inverse Clarke transformation:

$$u_{ref1} = u_\beta$$
$$u_{ref2} = \frac{\sqrt{3} \cdot u_\alpha - u_\beta}{2}$$
$$u_{ref3} = \frac{-\sqrt{3} \cdot u_\alpha - u_\beta}{2}$$

The sector calculation is based on comparing the 3-phase reference voltages u_{ref1}, u_{ref2}, and u_{ref3} with zero. This computation is described by the following figure:

Figure 56. Identification of the sector number

The knowledge of the sector is necessary for the current sensing especially when shunt resistors are used. The GMCLIB_SvmDpwm function does not require the sector directly, but it requires the portion identification explained in the following. The Inverse Clarke transformation converts the u_α, u_β voltage components of the reference stator voltage vector U_S to 3-phase voltage components u_a, u_b, and u_c. The portion identification selects the portion from the u_a, u_b, and u_c voltages, based on the following conditions.

Figure 57. Identification of the portion number
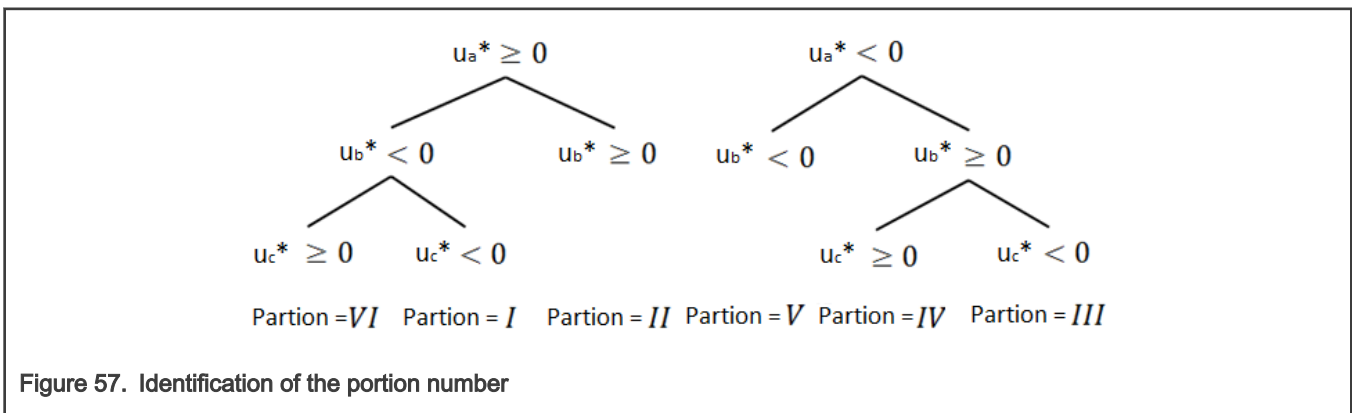
Finally, the corresponding duty cycle is selected according to the portion from the column of the following table.

Table 20. Duty cycle calculation from portions

Portions	I	II	III	IV	V	VI
Voltage boundaries	U_{330}, U_{30}	U_{30}, U_{90}	U_{90}, U_{150}	U_{150}, U_{210}	U_{210}, U_{270}	U_{270}, U_{330}
pwm_a	1	$0 - u_{ref3}$	$1 + u_{ref2}$	0	$1 - u_{ref3}$	$0 + u_{ref2}$
pwm_b	$1 - u_{ref2}$	$0 + u_{ref1} = u_\beta$	1	$0 - u_{ref2}$	$1 + u_{ref1} = 1 + u_\beta$	0
pwm_c	$1 + u_{ref3}$	0	$1 - u_{ref1} = 1 - u_\beta$	$0 + u_{ref3}$	1	$0 - u_{ref1} = 0 - u_\beta$

2.14.1 Available versions

This function is available in the following versions:

- Fractional output - the output is the fractional portion of the result; the result is within the range <0 ; 1). The result may saturate.

The available versions of the GMCLIB_SvmDpwm function are shown in the following table:

Table 21. Function versions

Function name	Input type	Output type	Result type
GMCLIB_SvmDpwm_F16	GMCLIB_2COOR_ALBE_T_F16 *	GMCLIB_3COOR_T_F16 *	uint16_t
	Standard discontinuous PWM with a 16-bit fractional stationary (α-β) input, and a 16-bit fractional 3-phase output. The result type is a 16-bit unsigned integer, which indicates the actual SVM sector. The input is within the range <-1 ; 1); the output duty cycle is within the range <0 ; 1). The output sector is an integer value within the range <1 ; 6>.		

2.14.2 Declaration

The available GMCLIB_SvmDpwm functions have the following declarations:

```
uint16_t GMCLIB_SvmDpwm_F16(const GMCLIB_2COOR_ALBE_T_F16 *psIn, GMCLIB_3COOR_T_F16 *psOut)
```

2.14.3 Function use

The use of the GMCLIB_SvmDpwm function is shown in the following example:

```
#include "gmclib.h"

static uint16_t u16Sector;
static GMCLIB_2COOR_ALBE_T_F16 sAlphaBeta;
static GMCLIB_3COOR_T_F16 sAbc;

void Isr(void);

void main(void)
{
  /* Alpha, Beta structure initialization */
  sAlphaBeta.f16Alpha = FRAC16(0.0);
  sAlphaBeta.f16Beta = FRAC16(0.0);

/* Periodical function or interrupt */
}

  void Isr(void)
{
  /* Standard Discountinues PWM SVM calculation */
  u16Sector = GMCLIB_SvmGenDpwm_F16(&sAlphaBeta, &sAbc);
}
```

2.15 GMCLIB_SvmExDpwm

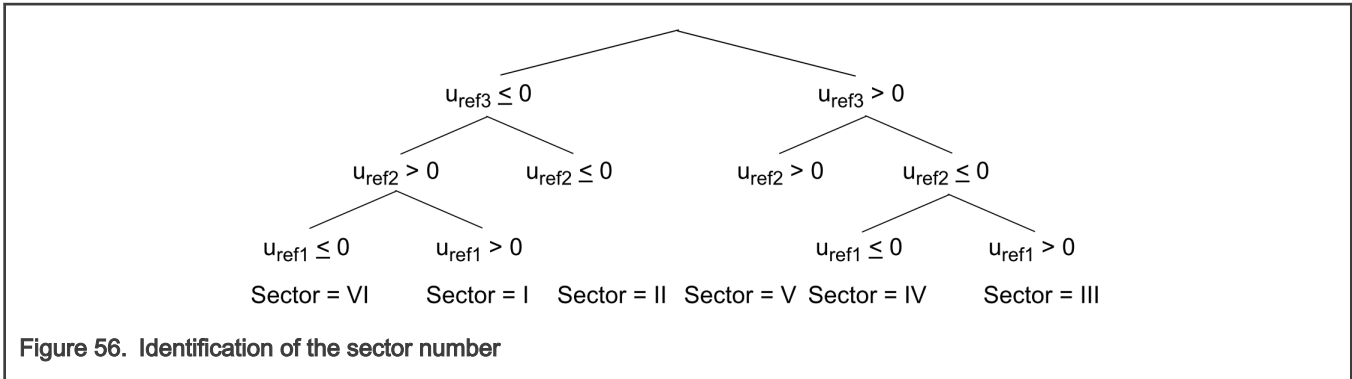The GMCLIB_SvmExDpwmfunction calculates the appropriate duty-cycle ratios needed for the generation of the given stator-reference voltage vector using the general non-sinusoidal modulation technique. The GMCLIB_SvmExDpwmfunction is a superset of the GMCLIB_SvmDpwmfunction without the power factor angle input.

The GMCLIB_SvmExDpwm function belongs to the discontinuous PWM modulation techniques for a 3-phase voltage inverter. The advantages of the discontinuous PWM technique are lower switching loses, but, on the other hand, it can cause higher harmonic distortion at low modulation indexes. The current sensing at low modulation indexes is more complicated and less precise when compared to the symmetrical modulation techniques like GMCLIB_SvmStd. Therefore, the discontinuous and continuous SVM are usually combined together.

Finding the sector in which the reference stator voltage vector U_S resides is similar to GMCLIB_SvmStd. This is achieved by converting the direct-α and quadrature-β components of the reference stator voltage vector U_S into the balanced 3-phase quantities u_{ref1}, u_{ref2}, and u_{ref3} using the modified Inverse Clarke transformation:

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

$$u_{ref2} = \frac{\sqrt{3} \cdot u_\alpha - u_\beta}{2}$$

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

The sector calculation is based on comparing the 3-phase reference voltages u_{ref1}, u_{ref2}, and u_{ref3} with zero. This computation is described by the following figure:

Figure 58. Identification of the sector number

The knowledge of the sector is necessary for the current sensing especially when shunt resistors are used. The GMCLIB_SvmExDpwm function does not require the sector directly, but it requires the portion identification explained in following text. The Park transformation uses the phase shift of the generated phase voltages and currents - φ angle to rotate the reference stator voltage vector U_S to U_S^* with the u_α^*, u_β^* components. The inverse Clarke transformation converts the u_α^*, u_β^* voltage components to 3-phase voltage components u_a^*, u_b^*, and u_c^*. The portion identification selects the portion from the u_a^*, u_b^*, and u_c^* voltages based on the following conditions.
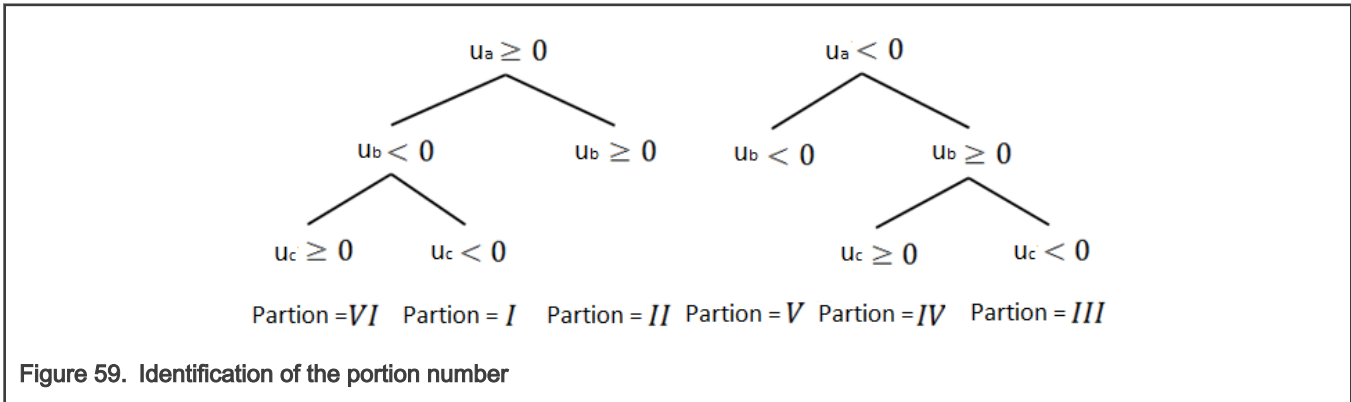
Figure 59. Identification of the portion number

Finally, the corresponding duty cycle is selected according to the portion from the column of the following table.

Table 22. Duty cycle calculation from portions

Portions	I	II	III	IV	V	VI
Voltage boundaries	U_{330}, U_{30}	U_{30}, U_{90}	U_{90}, U_{150}	U_{150}, U_{210}	U_{210}, U_{270}	U_{270}, U_{330}
pwm_a	1	$0 - u_{ref3}$	$1 + u_{ref2}$	0	$1 - u_{ref3}$	$0 + u_{ref2}$

Table continues on the next page...

Table 22. Duty cycle calculation from portions (continued)

| pwm_b | 1 - u_{ref2} | 0 + u_{ref1} = u_β | 1 | 0 - u_{ref2} | 1 + u_{ref1} = 1 + u_β | 0 |
| pwm_c | 1 + u_{ref3} | 0 | 1 - u_{ref1} = 1 - u_β | 0 + u_{ref3} | 1 | 0 - u_{ref1} = 0 - u_β |

2.15.1 Available versions

This function is available in the following versions:

- Fractional output - the output is the fractional portion of the result; the result is within the range <0 ; 1). The result may saturate.

The available versions of the GMCLIB_SvmExDpwm function are shown in the following table:

Table 23. Function versions

Function name	Input type		Output type	Result type
GMCLIB_SvmExDpwm_F16	GMCLIB_2COOR_ALBE_T_F16 *		GMCLIB_3COOR_T_F16 *	uint16_t
	GMCLIB_2COOR_SINCOS_T_F16 *			
	Extended discontinuous PWM with a 16-bit fractional stationary (α-β) input, the second input using a 16-bit fractional (sin(φ) / cos(φ)) structure of φ angle (-1/6 ; 1/6) in fraction corresponding (-π/6 ; π/6) in radians - angle of the power factor, it is a phase shift of the generated phase voltages and currents and a 16-bit fractional 3-phase output. The result type is a 16-bit unsigned integer which indicates the actual SVM sector. The input is within the range <-1 ; 1); the output duty cycle is within the range <0 ; 1). The output sector is an integer value within the range <1 ; 6>.			

2.15.2 Declaration

The available GMCLIB_SvmExDpwm functions have the following declarations:

```
uint16_t GMCLIB_SvmExDpwm_F16(const GMCLIB_2COOR_ALBE_T_F16 *psIn,const GMCLIB_2COOR_SINCOS_T_F16
*psAngle, GMCLIB_3COOR_T_F16 *psOut)
```

2.15.3 Function use

The use of the GMCLIB_SvmExDpwm function is shown in the following example:

```
#include "gmclib.h"

static uint16_t u16Sector;
static GMCLIB_2COOR_ALBE_T_F16 sAlphaBeta;
static GMCLIB_2COOR_SINCOS_T_F16 sAlphaBeta;
static GMCLIB_3COOR_T_F16 sAbc;

void Isr(void);

void main(void)
{
  /* Alpha, Beta structure initialization */
  sAlphaBeta.f16Alpha = FRAC16(0.0);
  sAlphaBeta.f16Beta = FRAC16(0.0);

  /* Power factor angle structure initialization */
  sAngle.f16Cos = FRAC16(1.0);
  sAngle.f16Sin = FRAC16(0.0);
}
```

```
/* Periodical function or interrupt */
void Isr(void)
{
  /* Extended Discountinues PWM calculation */
  u16Sector = GMCLIB_SvmExDpwm_F16(&sAlphaBeta, &sAngle, &sAbc);
}
```

Appendix A
Library types

A.1 bool_t

The bool_t type is a logical 16-bit type. It is able to store the boolean variables with two states: TRUE (1) or FALSE (0). Its definition is as follows:

```
typedef unsigned short bool_t;
```

The following figure shows the way in which the data is stored by this type:

Table 24. Data storage

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Value	Unused															Logical
TRUE	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
	0				0				0				1			
FALSE	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	0				0				0				0			

To store a logical value as bool_t, use the FALSE or TRUE macros.

A.2 uint8_t

The uint8_t type is an unsigned 8-bit integer type. It is able to store the variables within the range <0 ; 255>. Its definition is as follows:

```
typedef unsigned char uint8_t;
```

The following figure shows the way in which the data is stored by this type:

Table 25. Data storage

	7	6	5	4	3	2	1	0
Value	Integer							
255	1	1	1	1	1	1	1	1
	F				F			

Table continues on the next page...

Table 25. Data storage (continued)

11	0	0	0	0	1	0	1	1
	0				B			
124	0	1	1	1	1	1	0	0
	7				C			
159	1	0	0	1	1	1	1	1
	9				F			

A.3 uint16_t

The uint16_t type is an unsigned 16-bit integer type. It is able to store the variables within the range <0 ; 65535>. Its definition is as follows:

```
typedef unsigned short uint16_t;
```

The following figure shows the way in which the data is stored by this type:

Table 26. Data storage

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Value	Integer															
65535	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	F				F				F				F			
5	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1
	0				0				0				5			
15518	0	0	1	1	1	1	0	0	1	0	0	1	1	1	1	0
	3				C				9				E			
40768	1	0	0	1	1	1	1	1	0	1	0	0	0	0	0	0
	9				F				4				0			

A.4 uint32_t

The uint32_t type is an unsigned 32-bit integer type. It is able to store the variables within the range <0 ; 4294967295>. Its definition is as follows:

```
typedef unsigned long uint32_t;
```

The following figure shows the way in which the data is stored by this type:

Table 27. Data storage

Value	31	24	23	16	15	8	7	0
	Integer							
4294967295	F	F	F	F	F	F	F	F
2147483648	8	0	0	0	0	0	0	0
55977296	0	3	5	6	2	5	5	0
3451051828	C	D	B	2	D	F	3	4

A.5 int8_t

The int8_t type is a signed 8-bit integer type. It is able to store the variables within the range <-128 ; 127>. Its definition is as follows:

```
typedef char int8_t;
```

The following figure shows the way in which the data is stored by this type:

Table 28. Data storage

Value	7	6	5	4	3	2	1	0
	Sign	Integer						
127	0	1	1	1	1	1	1	1
	7				F			
-128	1	0	0	0	0	0	0	0
	8				0			
60	0	0	1	1	1	1	0	0
	3				C			

Table continues on the next page...

Table 28. Data storage (continued)

-97	1	0	0	1	1	1	1	1
	9				F			

A.6 int16_t

The int16_t type is a signed 16-bit integer type. It is able to store the variables within the range <-32768 ; 32767>. Its definition is as follows:

```
typedef short int16_t;
```

The following figure shows the way in which the data is stored by this type:

Table 29. Data storage

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Value	Sign	Integer														
32767	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	7				F				F				F			
-32768	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	8				0				0				0			
15518	0	0	1	1	1	1	0	0	1	0	0	1	1	1	1	0
	3				C				9				E			
-24768	1	0	0	1	1	1	1	1	0	1	0	0	0	0	0	0
	9				F				4				0			

A.7 int32_t

The int32_t type is a signed 32-bit integer type. It is able to store the variables within the range <-2147483648 ; 2147483647>. Its definition is as follows:

```
typedef long int32_t;
```

The following figure shows the way in which the data is stored by this type:

Table 30. Data storage

Table continues on the next page...

Table 30. Data storage (continued)

Value	31		24	23		16	15		8	7		0
	S				Integer							
2147483647	7	F		F	F		F	F		F		F
-2147483648	8	0		0	0		0	0		0		0
55977296	0	3		5	6		2	5		5		0
-843915468	C	D		B	2		D	F		3		4

A.8 frac8_t

The frac8_t type is a signed 8-bit fractional type. It is able to store the variables within the range <-1 ; 1). Its definition is as follows:

```
typedef char frac8_t;
```

The following figure shows the way in which the data is stored by this type:

Table 31. Data storage

Value	7	6	5	4	3	2	1	0
	Sign				Fractional			
0.99219	0	1	1	1	1	1	1	1
		7				F		
-1.0	1	0	0	0	0	0	0	0
		8				0		
0.46875	0	0	1	1	1	1	0	0
		3				C		
-0.75781	1	0	0	1	1	1	1	1
		9				F		

To store a real number as frac8_t, use the FRAC8 macro.

A.9 frac16_t

The frac16_t type is a signed 16-bit fractional type. It is able to store the variables within the range <-1 ; 1). Its definition is as follows:

```
typedef short frac16_t;
```

The following figure shows the way in which the data is stored by this type:

Table 32. Data storage

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Value	Sign	Fractional														
0.99997	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	7				F				F				F			
-1.0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	8				0				0				0			
0.47357	0	0	1	1	1	1	0	0	1	0	0	1	1	1	1	0
	3				C				9				E			
-0.75586	1	0	0	1	1	1	1	1	0	1	0	0	0	0	0	0
	9				F				4				0			

To store a real number as frac16_t, use the FRAC16 macro.

A.10 frac32_t

The frac32_t type is a signed 32-bit fractional type. It is able to store the variables within the range <-1 ; 1). Its definition is as follows:

```
typedef long frac32_t;
```

The following figure shows the way in which the data is stored by this type:

Table 33. Data storage

	31		24	23		16	15		8	7		0
Value	S	Fractional										
0.9999999995	7	F	F	F	F	F	F	F				

Table continues on the next page...

Table 33. Data storage (continued)

-1.0	8	0	0	0	0	0	0	0
0.02606645970	0	3	5	6	2	5	5	0
-0.3929787632	C	D	B	2	D	F	3	4

To store a real number as frac32_t, use the FRAC32 macro.

A.11 acc16_t

The acc16_t type is a signed 16-bit fractional type. It is able to store the variables within the range <-256 ; 256). Its definition is as follows:

```
typedef short acc16_t;
```

The following figure shows the way in which the data is stored by this type:

Table 34. Data storage

Value	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	Sign	Integer								Fractional						
255.9921875	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
		7			F				F				F			
-256.0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
		8			0				0				0			
1.0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
		0			0				8				0			
-1.0	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0
		F			F				8				0			
13.7890625	0	0	0	0	0	1	1	0	1	1	1	0	0	1	0	1
		0			6				E				5			
-89.71875	1	1	0	1	0	0	1	1	0	0	1	0	0	1	0	0
		D			3				2				4			

To store a real number as acc16_t, use the ACC16 macro.

A.12 acc32_t

The acc32_t type is a signed 32-bit accumulator type. It is able to store the variables within the range <-65536 ; 65536). Its definition is as follows:

```
typedef long acc32_t;
```

The following figure shows the way in which the data is stored by this type:

Table 35. Data storage

	31		24 23		16 15		8 7	0
Value	S	Integer			Fractional			
65535.999969	7	F	F	F	F	F	F	F
-65536.0	8	0	0	0	0	0	0	0
1.0	0	0	0	0	8	0	0	0
-1.0	F	F	F	F	8	0	0	0
23.789734	0	0	0	B	E	5	1	6
-1171.306793	F	D	B	6	5	8	B	C

To store a real number as acc32_t, use the ACC32 macro.

A.13 float_t

The float_t type is a signed 32-bit single precision floating-point type, defined by IEEE 754. It is able to store the full precision (normalized) finite variables within the range <-3.40282 $\cdot$ 10^{38} ; 3.40282 $\cdot$ 10^{38}) with the minimum resolution of 2^{-23}. The smallest normalized number is ±1.17549 $\cdot$ 10^{-38}. Nevertheless, the denormalized numbers (with reduced precision) reach yet lower values, from ±1.40130 $\cdot$ 10^{-45} to ±1.17549 $\cdot$ 10^{-38}. The standard also defines the additional values:

- Negative zero
- Infinity
- Negative infinity
- Not a number

The 32-bit type is composed of:

- Sign (bit 31)
- Exponent (bits 23 to 30)
- Mantissa (bits 0 to 22)

The conversion of the number is straighforward. The sign of the number is stored in bit 31. The binary exponent is decoded as an integer from bits 23 to 30 by subtracting 127. The mantissa (fraction) is stored in bits 0 to 22. An invisible leading bit (it is not

actually stored) with value 1.0 is placed in front; therefore, bit 23 has a value of 0.5, bit 22 has a value 0.25, and so on. As a result, the mantissa has a value between 1.0 and 2. If the exponent reaches -127 (binary 00000000), the leading 1.0 is no longer used to enable the gradual underflow.

The float_t type definition is as follows:

```
typedef float float_t;
```

The following figure shows the way in which the data is stored by this type:

Table 36. Data storage - normalized values

Value	S (31)	Exponent (24 23)	Mantissa (16 15 ... 8 7 ... 0)
$(2.0 - 2^{-23}) \cdot 2^{127}$ ≈ $3.40282 \cdot 10^{38}$	0	1 1 1 1 1 1 1 0 (7F)	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 (7F FF FF FF)
$-(2.0 - 2^{-23}) \cdot 2^{127}$ ≈ $-3.40282 \cdot 10^{38}$	1	1 1 1 1 1 1 1 0 (FF)	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 (7F FF FF FF)
2^{-126} ≈ $1.17549 \cdot 10^{-38}$	0	0 0 0 0 0 0 0 1 (00)	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 (80 00 00 00)
-2^{-126} ≈ $-1.17549 \cdot 10^{-38}$	1	0 0 0 0 0 0 0 1 (80)	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 (80 00 00 00)
1.0	0	0 1 1 1 1 1 1 1 (3F)	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 (80 00 00 00)
-1.0	1	0 1 1 1 1 1 1 1 (BF)	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 (80 00 00 00)
π ≈ 3.1415927	0	1 0 0 0 0 0 0 0 (40)	1 0 0 1 0 0 1 0 0 0 0 1 1 1 1 1 1 0 1 1 0 1 1 (49 0F DB)
-20810.086	1	1 0 0 0 1 1 0 1 (C6)	0 1 0 0 0 0 1 0 1 0 0 1 0 1 0 0 0 0 1 0 1 1 0 0 (A2 94 2C)

Table continues on the next page...

Table 36. Data storage - normalized values (continued)

Table 37. Data storage - denormalized values

Value	31		24	23		16	15		8	7		0
	S	Exponent					Mantissa					
0.0	0	0 0 0 0 0 0 0		0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0								
		0	0		0	0	0	0	0	0		
-0.0	1	0 0 0 0 0 0 0		0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0								
		8	0		0	0	0	0	0	0		
$(1.0 - 2^{-23}) \cdot 2^{-126}$ $\approx 1.17549 \cdot 10^{-38}$	0	0 0 0 0 0 0 0		0	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1								
		0	0		7	F	F	F	F	F		
$-(1.0 - 2^{-23}) \cdot 2^{-126}$ $\approx -1.17549 \cdot 10^{-38}$	1	0 0 0 0 0 0 0		0	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1								
		8	0		7	F	F	F	F	F		
$2^{-1} \cdot 2^{-126}$ $\approx 5.87747 \cdot 10^{-39}$	0	0 0 0 0 0 0 0		0	1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0								
		0	0		4	0	0	0	0	0		
$-2^{-1} \cdot 2^{-126}$ $\approx -5.87747 \cdot 10^{-39}$	1	0 0 0 0 0 0 0		0	1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0								
		8	0		4	0	0	0	0	0		
$2^{-23} \cdot 2^{-126}$ $\approx 1.40130 \cdot 10^{-45}$	0	0 0 0 0 0 0 0		0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1								
		0	0		0	0	0	0	0	1		
$-2^{-23} \cdot 2^{-126}$ $\approx -1.40130 \cdot 10^{-45}$	1	0 0 0 0 0 0 0		0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1								
		8	0		0	0	0	0	0	1		

Table 38. Data storage - special values

		31	24	23	16	15	8	7	0

A.14 GMCLIB_3COOR_T_F16

The GMCLIB_3COOR_T_F16 structure type corresponds to the three-phase stationary coordinate system, based on the A, B, and C components. Each member is of the frac16_t data type. The structure definition is as follows:

```
typedef struct
{
    frac16_t f16A;
    frac16_t f16B;
    frac16_t f16C;
} GMCLIB_3COOR_T_F16;
```

The structure description is as follows:

Table 39. GMCLIB_3COOR_T_F16 members description

Type	Name	Description
frac16_t	f16A	A component; 16-bit fractional type
frac16_t	f16B	B component; 16-bit fractional type
frac16_t	f16C	C component; 16-bit fractional type

A.15 GMCLIB_3COOR_T_FLT

The GMCLIB_3COOR_T_FLT structure type corresponds to the three-phase stationary coordinate system, based on the A, B, and C components. Each member is of the float_t data type. The structure definition is as follows:

```
typedef struct
{
    float_t fltA;
    float_t fltB;
```

```
    float_t fltC;
} GMCLIB_3COOR_T_FLT;
```

The structure description is as follows:

Table 40. GMCLIB_3COOR_T_FLT members description

Type	Name	Description
float_t	fltA	A component; 32-bit single precision floating-point type
float_t	fltB	B component; 32-bit single precision floating-point type
float_t	fltC	C component; 32-bit single precision floating-point type

A.16 GMCLIB_2COOR_AB_T_F16

The GMCLIB_2COOR_AB_T_F16 structure type corresponds to the general two-phase stationary coordinate system, based on the A and B orthogonal components. Each member is of the frac16_t data type. The structure definition is as follows:

```
typedef struct
{
    frac16_t f16A;
    frac16_t f16B;
} GMCLIB_2COOR_AB_T_F16;
```

The structure description is as follows:

Table 41. GMCLIB_2COOR_AB_T_F16 members description

Type	Name	Description
frac16_t	f16A	A-component; 16-bit fractional type
frac16_t	f16B	B-component; 16-bit fractional type

A.17 GMCLIB_2COOR_AB_T_F32

The GMCLIB_2COOR_AB_T_F32 structure type corresponds to the general two-phase stationary coordinate system, based on the A and B orthogonal components. Each member is of the frac32_t data type. The structure definition is as follows:

```
typedef struc
{
    frac32_t f32Alpha;
    frac32_t f32Beta;
} GMCLIB_2COOR_AB_T_F32;
```

The structure description is as follows:

Table 42. GMCLIB_2COOR_AB_T_F32 members description

Type	Name	Description
frac32_t	f32A	A component; 32-bit fractional type
frac32_t	f32B	B component; 32-bit fractional type

A.18 GMCLIB_2COOR_ALBE_T_F16

The GMCLIB_2COOR_ALBE_T_F16 structure type corresponds to the two-phase stationary coordinate system, based on the Alpha and Beta orthogonal components. Each member is of the frac16_t data type. The structure definition is as follows:

```
typedef struct
{
    frac16_t f16Alpha;
    frac16_t f16Beta;
} GMCLIB_2COOR_ALBE_T_F16;
```

The structure description is as follows:

Table 43. GMCLIB_2COOR_ALBE_T_F16 members description

Type	Name	Description
frac16_t	f16Apha	α-component; 16-bit fractional type
frac16_t	f16Beta	β-component; 16-bit fractional type

A.19 GMCLIB_2COOR_ALBE_T_FLT

The GMCLIB_2COOR_ALBE_T_FLT structure type corresponds to the two-phase stationary coordinate system based on the Alpha and Beta orthogonal components. Each member is of the float_t data type. The structure definition is as follows:

```
typedef struct
{
    float_t fltAlpha;
    float_t fltBeta;
} GMCLIB_2COOR_ALBE_T_FLT;
```

The structure description is as follows:

Table 44. GMCLIB_2COOR_ALBE_T_FLT members description

Type	Name	Description
float_t	fltApha	α-component; 32-bit single precision floating-point type
float_t	fltBeta	β-component; 32-bit single precision floating-point type

A.20 GMCLIB_2COOR_DQ_T_F16

The GMCLIB_2COOR_DQ_T_F16 structure type corresponds to the two-phase rotating coordinate system, based on the D and Q orthogonal components. Each member is of the frac16_t data type. The structure definition is as follows:

```
typedef struct
{
    frac16_t f16D;
    frac16_t f16Q;
} GMCLIB_2COOR_DQ_T_F16;
```

The structure description is as follows:

Table 45. GMCLIB_2COOR_DQ_T_F16 members description

Type	Name	Description
frac16_t	f16D	D-component; 16-bit fractional type
frac16_t	f16Q	Q-component; 16-bit fractional type

A.21 GMCLIB_2COOR_DQ_T_F32

The GMCLIB_2COOR_DQ_T_F32 structure type corresponds to the two-phase rotating coordinate system, based on the D and Q orthogonal components. Each member is of the frac32_t data type. The structure definition is as follows:

```
typedef struct
{
    frac32_t f32D;
    frac32_t f32Q;
} GMCLIB_2COOR_DQ_T_F32;
```

The structure description is as follows:

Table 46. GMCLIB_2COOR_DQ_T_F32 members description

Type	Name	Description
frac32_t	f32D	D-component; 32-bit fractional type
frac32_t	f32Q	Q-component; 32-bit fractional type

A.22 GMCLIB_2COOR_DQ_T_FLT

The GMCLIB_2COOR_DQ_T_FLT structure type corresponds to the two-phase rotating coordinate system, based on the D and Q orthogonal components. Each member is of the float_t data type. The structure definition is as follows:

```
typedef struct
{
    float_t fltD;
    float_t fltQ;
} GMCLIB_2COOR_DQ_T_FLT;
```

The structure description is as follows:

Table 47. GMCLIB_2COOR_DQ_T_FLT members description

Type	Name	Description
float_t	fltD	D-component; 32-bit single precision floating-point type
float_t	fltQ	Q-component; 32-bit single precision floating-point type

A.23 GMCLIB_2COOR_SINCOS_T_F16

The GMCLIB_2COOR_SINCOS_T_F16 structure type corresponds to the two-phase coordinate system, based on the Sin and Cos components of a certain angle. Each member is of the frac16_t data type. The structure definition is as follows:

```
typedef struct
{
    frac16_t f16Sin;
```

```
    frac16_t f16Cos;
} GMCLIB_2COOR_SINCOS_T_F16;
```

The structure description is as follows:

Table 48. GMCLIB_2COOR_SINCOS_T_F16 members description

Type	Name	Description
frac16_t	f16Sin	Sin component; 16-bit fractional type
frac16_t	f16Cos	Cos component; 16-bit fractional type

A.24 GMCLIB_2COOR_SINCOS_T_FLT

The GMCLIB_2COOR_SINCOS_T_FLT structure type corresponds to the two-phase coordinate system, based on the Sin and Cos components of a certain angle. Each member is of the float_t data type. The structure definition is as follows:

```
typedef struct
{
    float_t fltSin;
    float_t fltCos;
} GMCLIB_2COOR_SINCOS_T_FLT;
```

The structure description is as follows:

Table 49. GMCLIB_2COOR_SINCOS_T_FLT members description

Type	Name	Description
float_t	fltSin	Sin component; 32-bit single precision floating-point type
float_t	fltCos	Cos component; 32-bit single precision floating-point type

A.25 FALSE

The FALSE macro serves to write a correct value standing for the logical FALSE value of the bool_t type. Its definition is as follows:

```
#define FALSE    ((bool_t)0)
```

```
#include "mlib.h"

static bool_t bVal;

void main(void)
{
  bVal = FALSE;              /* bVal = FALSE */
}
```

A.26 TRUE

The TRUE macro serves to write a correct value standing for the logical TRUE value of the bool_t type. Its definition is as follows:

```
#define TRUE    ((bool_t)1)
```

```
#include "mlib.h"

static bool_t bVal;

void main(void)
{
  bVal = TRUE;                /* bVal = TRUE */
}
```

A.27 FRAC8

The FRAC8 macro serves to convert a real number to the frac8_t type. Its definition is as follows:

```
#define FRAC8(x) ((frac8_t)((x) < 0.9921875 ? ((x) >= -1 ? (x)*0x80 : 0x80) : 0x7F))
```

The input is multiplied by 128 (=2^7). The output is limited to the range <0x80 ; 0x7F>, which corresponds to <-1.0 ; 1.0-2^{-7}>.

```
#include "mlib.h"

static frac8_t f8Val;

void main(void)
{
  f8Val = FRAC8(0.187);               /* f8Val = 0.187 */
}
```

A.28 FRAC16

The FRAC16 macro serves to convert a real number to the frac16_t type. Its definition is as follows:

```
#define FRAC16(x) ((frac16_t)((x) < 0.999969482421875 ? ((x) >= -1 ? (x)*0x8000 : 0x8000) : 0x7FFF))
```

The input is multiplied by 32768 (=2^{15}). The output is limited to the range <0x8000 ; 0x7FFF>, which corresponds to <-1.0 ; 1.0-2^{-15}>.

```
#include "mlib.h"

static frac16_t f16Val;

void main(void)
{
  f16Val = FRAC16(0.736);                 /* f16Val = 0.736 */
}
```

A.29 FRAC32

The FRAC32 macro serves to convert a real number to the frac32_t type. Its definition is as follows:

```
#define FRAC32(x) ((frac32_t)((x) < 1 ? ((x) >= -1 ? (x)*0x80000000 : 0x80000000) : 0x7FFFFFFF))
```

The input is multiplied by 2147483648 (=2^{31}). The output is limited to the range <0x80000000 ; 0x7FFFFFFF>, which corresponds to <-1.0 ; 1.0-2^{-31}>.

```
#include "mlib.h"

static frac32_t f32Val;

void main(void)
{
  f32Val = FRAC32(-0.1735667);                  /* f32Val = -0.1735667 */
}
```

A.30 ACC16

The ACC16 macro serves to convert a real number to the acc16_t type. Its definition is as follows:

```
#define ACC16(x) ((acc16_t)((x) < 255.9921875 ? ((x) >= -256 ? (x)*0x80 : 0x8000) : 0x7FFF))
```

The input is multiplied by 128 (=2^7). The output is limited to the range <0x8000 ; 0x7FFF> that corresponds to <-256.0 ; 255.9921875>.

```
#include "mlib.h"

static acc16_t a16Val;

void main(void)
{
  a16Val = ACC16(19.45627);              /* a16Val = 19.45627 */
}
```

A.31 ACC32

The ACC32 macro serves to convert a real number to the acc32_t type. Its definition is as follows:

```
#define ACC32(x) ((acc32_t)((x) < 65535.999969482421875 ? ((x) >= -65536 ? (x)*0x8000 : 0x80000000) :
0x7FFFFFFF))
```

The input is multiplied by 32768 (=2^{15}). The output is limited to the range <0x80000000 ; 0x7FFFFFFF>, which corresponds to <-65536.0 ; 65536.0-2^{-15}>.

```
#include "mlib.h"

static acc32_t a32Val;

void main(void)
```

```
{
  a32Val = ACC32(-13.654437);               /* a32Val = -13.654437 */
}
```