SILworX

C++ Function Block Manual

SAFETY NONSTOP

HIMA

Contact

HIMA contact details:

HIMA Paul Hildebrandt GmbH

P.O. Box 1261

68777 Brühl, Germany

Phone: +49 6202 709-0

Fax: +49 6202 709-107

E-mail: info@hima.com

Revision index	Revisions	Type of change	
		technical	editorial
5.00	First issue	X	X
5.01	Chapter 3.2 Required processor operating systems	X	X
5.02	Chapter 4.3.1 Stack Size	X	X

Table of Contents

1 Introduction

The present manual describes how to use and program the C++ function blocks. The manual provides information about the C++ language elements and the configuration in SILworX.

1.1 Structure of this Manual

This manual is organized in the following main chapters:

- Introduction
- Safety
- Product description
- Start-up
- Operation
- Appendix

1.2 Target Audience

This document addresses system planners, configuration engineers, programmers of automation devices and personnel authorized to implement, operate and maintain the devices and systems. Knowledge of programming in C++ is required.

1.3 Formatting Conventions

To ensure improved readability and comprehensibility, the following fonts are used in this document:

Bold:	To highlight important parts Names of buttons, menu functions and tabs that can be clicked and used in the programming tool.
Italics:	For parameters and system variables
`Courier`	Literal user inputs
RUN	Operating state are designated by capitals
Chapter 1.2.3	Cross references are hyperlinks even though they are not particularly marked. When the cursor hovers over a hyperlink, it changes its shape. Click the hyperlink to jump to the corresponding position.

Safety notes and operating tips are particularly marked.

1.3.1 Safety Notes

The safety notes are represented as described below.
These notes must absolutely be observed to reduce the risk to a minimum. The content is structured as follows:

- Signal word: warning, caution, notice
- Type and source of risk
- Consequences arising from non-observance
- Risk prevention

⚠ SIGNAL WORD

Type and source of risk!

Consequences arising from non-observance

Risk prevention

The signal words have the following meanings:

- Warning indicates hazardous situation which, if not avoided, could result in death or serious injury.
- Warning indicates hazardous situation which, if not avoided, could result in minor or modest injury.
- Notice indicates a hazardous situation which, if not avoided, could result in property damage.

NOTE

Type and source of damage!

Damage prevention

1.3.2 Operating Tips

Additional information is structured as presented in the following example:

i The text corresponding to the additional information is located here.

Useful tips and tricks appear as follows:

TIP The tip text is located here.

2 Safety

All safety information, notes and instructions specified in this document must be strictly observed.

The user must not make use of all technical possibilities provided by the C++ function blocks, see Chapter 3.3.3.

This document describes which rules have to be followed to create a C++ function block, which language elements may be used and the responsibility arising for the user.

The C++ function blocks may only be used if all guidelines and safety instructions are adhered to.

2.1 Intended Use

C++ function blocks may be used for safety-related operation in consultation with the test authority responsible for the final inspection!

WARNING

The user is responsible for guaranteeing the safety of the user program in accordance with IEC 61508!

An error in the function of a C++ function block can have immediate effects on any part of the calling user program. Effects on the simultaneously running user programs or other system components such as hardware or protocols are only possible indirectly via global variables.

3 Product Description

The C++ function blocks allow the user to embed their own C++ programs into the SILworX function block language (V5.34 and higher). The input variables and output variables of the C++ function blocks serve as an interface to the functions of the embedded C++ programs. A C++ function block can be used in SILworX as a typical function block.

Compared to the function block diagram, C++ function blocks allow for more flexible programming and higher performance. Additionally, a C++ source code from third-party tools (such as mathematics and simulation software) can be embedded into the SILworX function block diagram.

3.1 License

An activation code with the corresponding license option is required to create and change a C++ function block. Archives and projects including finished C++ function blocks may be used without license option. The activation code must then be specified in the SILworX license management.

3.2 Required HIMax/HIMatrix Operating System Versions

C++ function blocks can only be used in controllers with a suitable processor operating system, see the following table:

System	Processor operating system
HIMax	CPU OS V4.14 and higher
HIMatrix Layout 2	CPU OS V8.10 and higher
HIMatrix Layout 3	CPU OS V8.14 and higher

Table 1: Required Processor Operating Systems

3.3 Overview and Handling of C++ Function Blocks

This chapter offers an overview of the C++ function blocks' features and provides details on how to use the C++ function blocks.

3.3.1 Using C++ Function Blocks

- C++ function blocks can be created using SILworX.
- C++ function blocks can be used in the user program and in function blocks.
- Online changes (reload) of a program with C++ function blocks are not possible.
- SILworX functions such as archive, restore, copy, insert, delete, print or activate the write or know-how protection for C++ function blocks are available.
- The C++ function blocks can only be tested as black box via the input and output values, see Chapter 5.2.
- Users can import archives that contain C++ function blocks.
- C++ function blocks can be deleted.

3.3.2 Interfaces and Use of Variables

- The variables types VAR_Input, VAR_Output and VAR are used for the C++ function block interface.
- The value of VAR and VAR_Output variables is retained throughout an entire user program cycle.
- Only elementary data types (no arrays, structures or pointers) may be used for VAR_Input, VAR_Output and VAR.
- The C++ identifier for a variable is composed of a prefix and the variable name entered in SILworX. Characters not allowed for a C++ identifier and contained in a variable name are

replaced. Underscores as well. The characters used for the replacement are underscore followed by one to three alphabetic and numeric characters.

- Additional C++ source code and header files can be added. The elements defined in these files can be used within C++ function blocks.

3.3.3 Notes and Restrictions

The following section presents an overview of the most important restrictions:

- The use of global SILworX variables in C++ function blocks is not allowed.
- The use of SILworX function blocks in C++ function blocks is not allowed.
- The online test of C++ source code is not possible.
- The offline simulation of C++ source code is not possible.
- The class structure of C++ function blocks may only be changed using the SILworX interface declaration of VAR_Input, VAR_Output, VAR.

3.4 Basic Procedure for Creating C++ Function Blocks

The following steps describe the basic sequence for creating a C++ function block in SILworX.

1. Create a C++ function block in SILworX and specify the properties, see Chapter 4.1 and Chapter 4.2.
2. Configure the C++ function block in the editor. The input and output variables of the C++ function block and stack size are specified in the editor, see Chapter 4.3.
3. Export the C++ source file function body from the C++ function block, see Chapter 4.3.4.
4. Edit and extend the C++ source file function body with user-specific functions, see Chapter 5.1.
5. Import the extended C++ source file function bodies and, if existing, additional C++ source files into the C++ function block, see Chapter 4.3.5.

4 Configuration

This chapter describes how to configure the C++ function blocks and explains the functions of the C++ function block editor.

4.1 Creating C++ Function Blocks in SILworX

Follow these steps to add a C++ function block to a function block library.

If a library already exists, it can be used for the C++ function blocks as well.

To create a library for C++ function blocks

1. Choose the configuration and select **New** from the context menu or the action bar.
   ☑ The *New Object* dialog box is displayed.
2. Click **Library**.
3. Enter the name of the new library in the *Name* field.
4. Click **OK**.

A new library for the C++ function blocks is created as sub-element of configuration.

To create a C++ function block

1. Choose the new library and select **New** from the context menu or the action bar.
   ☑ The *New Object* dialog box is displayed.
2. Click **C++ Function Block Type**.
3. Enter the name of the new C++ function block type in the *Name* field.
4. Click **OK**.

A new C++ function block type is created as sub-element of the library.

4.2 Properties Dialog Box of the C++ Function Block

The properties of the C++ function blocks common to all POU types are set in the Properties dialog box. The properties of individual C++ function blocks are located in the C++ function block editor.

Parameter	Description	Changeable
Name	Name of a function block type such as specified in the structure tree, in the Drawing Area of the FBD Editor, and in the Function Blocks tab of the Object Panel. The name of the function block type must be unique. The name appears within the graphical object. The name of the function block type can also be altered in the Interface Viewer. Double-click the name in the graphical object and enter a new name. Press the Enter key to confirm the action.	Yes
Extendibility	Activate this parameter if the function block should be extendible.	Yes
Minimum Extendibility [pins]	This parameter defines the minimum number of inputs that can be displayed in the Drawing Area. This value corresponds to the default size.	Yes if Extendibility was activated.
Maximum Extendibility [pins]	This parameter defines the maximum number of inputs that can be displayed in the Drawing Area. The maximum extendibility depends on the number of inputs defined with VAR_INPUT. Use the *Extendibility* function to extend the POE in the Drawing Area to its maximum size.	Yes if Extendibility was activated.

Default Instance Name	Instance name of a function block type such as displayed in the Drawing Area of the FBD Editor. Each instance of the function block type may have its own name. If no user-specific name is assigned, the default instance name is used followed by a consecutive number «_n». The instance name appears outside on the upper border of the graphical object. The default instance name cannot be changed in the Interface Viewer.	Yes
Default Width	The default width defines the width of the function block type in grid units of the Drawing Area. The default value is 10. This means that the object width in the Drawing Area is equal to 10 grid units. Note that the height of the object is preset by SILworX. Default value: 10	Yes
Variant	Normal: The user-defined POU is unlocked and third parties have unrestricted access to the C++ source file. This is the default setting. Read-Only: With the read-only setting, the following restrictions apply compared to an unlocked C++ function block. ▪ The editor of a read-only C++ function block can be opened, but not altered. All edit and modification options in the editor are disabled. ▪ The properties dialog box of a read-only C++ function block can be opened, but not altered. Read-Only can neither be reversed nor can it be circumvented using the Copy and Paste or Archive and Restore functions. Before SILworX locks a C++ function block, the confirmation prompt must be confirmed with *Yes*. i Make sure that an unlocked backup of the C++ function block is available! The code created by the code generator for a read-only C++ function block is the same code created for an unlocked C++ function block (identical CRC). Unlocked C++ function blocks can thus be used during the test phase. After testing and completion, they can be locked without affecting the CRCs, which remain unchanged. Read-only C++ function blocks are highlighted in color in the FBD Editor to better distinguish them from unlocked function bocks.	Yes

Variant	Know-How Protection	The know-how protection is set to define the following restrictions compared to an unlocked C++ function block: ▪ The editor of a C++ function block with know-how protection cannot be opened. ▪ The Properties dialog box of a C++ function block with know-how protection only displays the function block name. This field cannot be edited. ▪ If a C++ function block is know-how protected, only the interface is printed out in the documentation and only the name appears in the properties. All other data are not accessible. Know-how protection can neither be reversed nor can it be circumvented using the Copy and Paste or Archive and Restore functions. Before SILworX locks a C++ function block, the confirmation prompt must be confirmed with *Yes*. i Make sure that an unlocked backup of the C++ function block is available! The code created by the code generator for a know-how protected POU is the same created for an unlocked C++ function block (identical CRC). Unlocked C++ function blocks can thus be used during the test phase. After testing and completion, they can be locked without affecting the CRCs, which remain unchanged. Know-how protected C++ function blocks are highlighted in color in the FBD Editor to better distinguish them from unlocked function bocks.	Yes

Table 2: Properties C++ Function Blocks

4.3 The C++ Function Block Editor

The C++ Function Block Editor contains the components for creating the C++ function blocks including the import and export functions for the C++ source code files.

4.3.1 Stack Size

In SILworX, a stack size of 4 kBytes is reserved for HIMax and HIMatrix controllers.
If the C++ function blocks require a higher stack size, this can be set in SILworX for each individual C++ function block.

4.3.1.1 Determining and Setting the Additional Stack Size

If a larger stack size is needed, the users must estimate the maximum number of bytes that the C++ functions called in the C++ function block will occupy on the stack.

To ensure maximum availability, HIMA recommends to include a sufficiently large reserve in the estimate of the maximum stack size.

If the defined stack size is not sufficient, a stack overflow may result when the C++ function block is called, causing, in turn, the user program to immediately enter the error stop and the controller to enter the safe state.

The following C++ source code characteristics, which may considerably affect and increase the stack size, must be taken into account when estimating the required stack size:

- Large arrays or structures as local variables
- Large structures as transition variables
- Nested C++ functions

HIMA recommends the following procedure for estimating the required stack size:

1. Determine which C++ function has the largest local and transfer variables within the C++ source code.
2. Within the C++ source code, determine the "deepest" call path among the various nested C++ functions that contain the aforementioned C++ function.
3. Determine the total size of all local and transition variables within this call path. In the C++ function blocks, enter this value as the stack size along with additional reserve as needed.
4. Perform a functional test in the HIMax or HIMatrix controller to ensure the C++ function block has sufficient stack size. The stack size cannot be checked during an offline simulation.

In SILworX, the additional stack size can be set for each individual C++ function block. The upper limit for the range corresponds to the maximum memory space allowed for the program. In practice, the memory space used for the stack is lower.

Parameter	Description
Stack Size (HIMatrix Layout 2)	Range of values: 0…1 048 064 Byte Standard: 0
Stack Size (HIMatrix Layout 3)	Range of values: 0…5 177 344 Byte Standard: 0
Stack Size (HIMax)	Range of values: 0…10 481 664 Byte Standard: 0

Table 3: Stack Size for HIMax/HIMatrix Controllers

4.3.2 Local Variables in the POU Editor

The **Local Variables** tab contains all the local variables that can be used in the current logic. This also includes the variables that have been defined as input or output variables of C++ function blocks.

The variable types VAR, VAR_INPUT and VAR_OUTPUT for programming C++ function blocks are available in the POU Editor.

4.3.3 Source Files

A C++ function block contains the following default source files:

- Header file "SRCUSR<type file name>.h"
- Cpp file "SRCUSR<type file name>_Content.cpp"

Optionally, additional source files can be imported, see Chapter 4.3.5.

4.3.4 Exporting Source Files from a C++ Function Block

The function **Export Source Files** is used to export all existing files of the C++ function block to the target directory (e.g., in the PADT). The exported C++ function bodies can be edited in a text editor and extended with user-defined functions.

To export source files

1. In the structure tree, select **Configuration** and open **Library**.
2. Right-click the **C++ function block** and select **Edit** from the context menu.
3. Click **Export Source Files** and select the target directory.
4. Click **OK** to confirm.

   ☑ If the target directory already contains files with the same name, the following options are provided in a dialog box.

   - *Copy and replace the previous file.*
   - *Do not copy.*
   - *Retain both files and rename the previous file.*

For each successfully exported file, a message is entered into the logbook.

The cpp file "SRCUSR<type file name>_Content.cpp.as_imported“ is created during the export (last imported .cpp-file)

4.3.5 Importing Source Files to a C++ Function Block

The **Import Source Files** function is used to read the source files previously exported back in the C++ function block. All source files existing in the C++ function block are deleted. In doing so, the C++ source files imported and extended with user-defined functions in the text editor are now integrated into the C++ function block.

The file names of the source files must have the extension .h or .cpp.

To import source files

1. In the structure tree, select **Configuration** and open **Library**.
2. Right-click the **C++ function block** and select **Edit** from the context menu.
3. Click **Import Source Files** and select the source directory.
4. Click **OK** to confirm.

   ☑ A dialog box appears informing that all existing source files will be deleted in the C++ function block. After this action, all source files are once again read in the C++ function block.

5. If the source files should be deleted and re-imported, click **OK** to confirm the action.

5 Creating C++ Source Code

This chapter describes the integration of user-defined C++ functions into the C++ source code function body and the rules that must be observed.

5.1 Integration of C++ Code

To implement the desired function in the C++ source code, the C++ source files in the user-defined sections are extended with the corresponding instructions.

Each user-defined section starts directly after a comment line such as:

```
// Start of user code section: <title>
```

The user-defined section ends directly before a comment line such as:

```
// End of user code section: <title>
```

The placeholder `<title>` stands for the respective section.

i The comment lines, which mark the user-defined sections, may not be altered.

5.1.1 C++ Function Blocks Interfaces for Function Block Language

A C++ function block is represented by a class in C++. Each function block instance is translated in an instance of this class.

The variables VAR or VAR_OUTPUT declared in SILworX appear in C++ as attributes of this class.

The calling of the C++ function block by another function block in SILworX corresponds to the calling of a function "CallFunctionContent" of the block's class in C++.

The variables VAR_INPUT declared in SILworX appear in C++ as parameters of this function.

5.1.2 Using C++ Function Block in Function Block Diagram

To use a C++ function block in function block diagram, drag the C++ function block from the *Function Blocks* tab onto the FBD Editor and establish the connections required.

5.1.3 Generating Code

No additional generation steps are required for a C++ function block. During standard code generation in SILworX, the *.cpp source files are compiled for a resource.

5.2 Testing and Diagnosis

To verify proper functioning of the C++ code, it must be tested in a suitable C++ development environment.

Additionally, during the SILworX online test, the finished C++ function block has to be tested within the system as black box, refer to the Safety Manual (HI 801 003 E) for more details. To check whether the C++ function block is behaving in accordance with the defined function, the inputs are controlled and the outputs are monitored.

SILworX does not perform any C++ code diagnosis.

5.3 C++ Language Scope

Only the language elements specified in this chapter may be used.

5.3.1 General C++ Language Elements

- Constants with supported data types
- Local variables with supported data types
- C++ global variables with supported data types
- Literals
- Comments
- if .. else
- switch
- while, do..while
- for
- function declarations and function calls
- sizeof
- assignments
- return
- break
- continue
- external
  for C++ functions and variables only, but no external declarations with specification of a programming language, e.g., external "C" typedef void (*CFunction)(void) is illeagal.
- #include
- #ifdef, #if .. #elif .. #else .. #endif
- #define
- #undef (for self-defined elements only)

5.3.2 Namespaces

A namespace is reserved for each C++ function block. This namespace is directly subordinated to the global namespace. The namespace identifier is composed of the prefix N_ followed by the class name of the C++ function block.

Example: namespace N_USRFunctionBlock1 { /* ... */ }

- A C++ function block must directly or indirectly subordinate all externally visible elements defined by the C++ function block to the namespace of the C++ function block. "Externally visible" means that the name of the element has an external linkage in accordance with the C++ standard. This also includes the namespaces, but the namespace of a C++ function block cannot be subordinated to itself.
- A C++ function block may define any subordinated namespaces in its reserved namespace.
- A C++ function block may define any alias names for namespaces, also in the global scope.
  Example: namespace FB1 = N_USRFunctionBlock1;
- A C++ function block may use using directives, even in the global scope.
  Example: using namespace N_USRFunctionBlock1;

5.3.3 C++ Global Variables

C++ local and global variables can be used in a C++ function block.

⚠ WARNING

During reload, the C++ global variables are overwritten with their new initial values.

- C++ global variables may be accessed (read, write) from all C++ function blocks of a program.
- The value of C++ global variables is retained between the user program cycles.
- The definition of each C++ global variable that is not initialized to zero, must contain the C++ global variable identifier followed by the X_USERDATA_INIT macro call and the C++ initialization string.

  Example:

  ```
  namespace N_USRFunctionBlock1 {
  int32 gLastError X_USERDATA_INIT = 1;
  myStruct gStruct X_USERDATA_INIT = { 5, true, 9.3 };
  }
  ```
- The C++ function block must not use the X_USERDATA_INIT macro for a declaration of C++ global variables, which is not a definition.
  Example: namespace N_USRFunctionBlock1 { extern int32 gLastError; }
- The definition of each C++ global variable initialized to zero, however, must contain the C++ global variable identifier followed by the macro call. Additionally, the X_USERDATA_ZERO macro, which can be used with no further details, exists for this specific case.
  Example:
  ```
  namespace N_USRFunctionBlock1 {
  int32 gLastError X_USERDATA_ZERO;
  myStruct gStruct X_USERDATA_ZERO;
  }
  ```
- The C++ function block must not use the X_USERDATA_ZERO macro for a declaration of the C++ global variables, which is not a definition.
  Example: namespace N_USRFunctionBlock1 { extern int32 gLastError; }

5.3.4 C++ Pointer

If reasonably possible, pointers should be avoided in accordance with IEC 61508.

For pointers, a C++ function block may only use the following values:

- The corresponding null pointer values.
- Objects addresses defining the same or another C++ function block.
- Function addresses defining the same or another C++ function block.

5.3.5 Includes

A C++ function block may use include instructions in the form of #include "file name", but each of these instructions has to meet the following conditions:

- The file name must be specified without directory path.
- The file name must identify an additional source file of the same or of another C++ function block.
- A C++ function block (A) can thus embed a source file from another C++ function block (B). To this end, each user program calling an instance of the one C++ function block (A) must also call at least one instance of the other C++ function block (B).

A C++ function block may use include instructions in the form of #include <Header>, but only the following headers are allowed:

float.h

limits.h

math.h

stddef.h

Additionally, a C++ function block may use the following sequence of include instructions in each additional source file (the sequence is generated by SILworX in the main source file "SRCUSR<type file name>_Content.cpp" as well):

```
#include "iec_types.h"

#include "iec_std_types.h"

#include "SRCUSR<type file name>.h"
```

If these instructions exist, they have to be located at the beginning of the source file (only comments and blank lines may precede). The specified order must be maintained and the sequence must be complete.

5.3.6 Data Types

The following table shows the matching data types as they are used in SILworX and C++, respectively . The corresponding alias names are also allowed in C++ source code.

i The corresponding assignment of alias name and SILworX data type is predefined. Theoretically, however, the assigned elementary C++ data type may change, in particular, for future projects.
If available and the size of a variable or the compatibility with SILworX data types are important factors, HIMA recommends to use the alias names defined by the PADT.

SILworX Data Type	Alias Names	C++ Data Type	Byte Size
BOOL	-	bool	1
BYTE and USINT	ubyte, uint8	unsigned char	1
WORD and UINT	uword, uint16	unsigned short	2
DWORD and UDINT	udword, uint32	unsigned long	4
LWORD and ULINT	uldword, uint64	unsigned long long	8
SINT	int8	char	1
INT	int16	short	2
DINT	int32	long	4
LINT and TIME	int64, time	long long	8
REAL	real	float	4
LREAL	lreal	double	8

Table 4: Data Types

For variables of BOOL data type, the bitmap 0x00 for false or 0x01 for true is the only valid internal representation. This is relevant, when the memory of a BOOL variable is altered by other means than by assigning this variable.

Void, structures, arrays, pointers, references and user-defined data types can be also used.

5.3.7 Functions and Operators

Compound expressions may be used.

5.3.7.1 Logical Operators

The following table specifies the logical operators such as defined in SILworX and C++.

SILworX Function	C++ Operator	Applicable SILworX Data Types
AND	&&	BOOL
OR	||	BOOL
NOT	!	BOOL

Table 5: Logical Operators

5.3.7.2 Bitwise Operators

The following table specifies the bitwise operators such as defined in SILworX and C++.

SILworX Function	C++ Operator	Applicable SILworX Data Types
AND	&	all ANY_ bit data types, except for BOOL
OR	|	all ANY_ bit data types, except for BOOL
NOT	~	all ANY_ bit data types, except for BOOL
XOR	^	all ANY_ bit data types, except for BOOL
SHR	>>	all ANY_ bit data types, except for BOOL
SHL	<<	all ANY_ bit data types, except for BOOL

Table 6: Bitwise Operators

5.3.7.3 Relational Operators

The following table specifies the relational operators such as defined in SILworX and C++.

SILworX Function	C++ Operator
GE	>=
GT	>
LE	<=
LT	<
EQ	==
NE	!=

Table 7: Relational Operators

5.3.7.4 Cast Operators

The following table specifies the cast operators such as defined in SILworX and C++.

SILworX Function	Alias Names	C++ Operator
AtoBOOL	-	(bool)
AtoBYTE	(ubyte), (uint8)	(unsigned char)
AtoWORD or AtoUINT	(uword), (uint16)	(unsigned short)
AtoDWORD or AtoUDINT	(udword), (uint32)	(unsigned long)
AtoLWORD or AtoULINT	(uldword), (uint64)	(unsigned long long)
AtoSINT	(int8)	(char)
AtoINT	(int16)	(short)
AtoDINT	(int32)	(long)
AtoLINT or AtoTIME	(int64), (time)	(long long)
AtoREAL	(real)	(float)
AtoLREAL	(lreal)	(double)

Table 8: Cast Operators

In C++, the trunk function is used for casting float or double to an integer data type. In contrast, with SILworX functions, the value is rounded in accordance with IEC 61131-3.

If the range of values of the target data type is exceeded when casting float or double to (unsigned) long long, a difference in the C++ code results compared to the corresponding SILworX function.

5.3.7.5 Arithmetic Operators

The following table specifies the arithmetic operators such as defined in SILworX and C++.

SILworX Function	C++ Operator
ADD	+
SUB	-
MUL	*
DIV	/
MOD	%

Table 9: Arithmetic Operators

5.3.7.6 Numeric Functions

The following table specifies the numeric functions such as defined in SILworX and C++.

SILworX Function	C++ Function
SQRT	sqrtf(float), sqrt(double)
ABS	abs(char), abs(short), labs(long), llabs(long long), fabsf(float), fabs(double)
SIN	sinf(float), sin(double)
ASIN	asinf(float), asin(double)
COS	cosf(float), cos(double)
ACOS	acosf(float), acos(double)
TAN	tanf(float), tan(double)
ATAN	atanf(float), atan(double)
EXPT	powf(float, float), pow(double, double)
EXP	expf(float), exp(double)
LN	logf(float), log(double)
LOG	log10f(float), log10(double)

Table 10: Numeric Functions

5.4 File Format and Legal Characters

Only text files with characters in accordance with UTF-8 encoding are allowed as C++ source files for a C++ function block.

The Unicode® code point of a character is specified as hexadecimal number with the prefix U+.

The characters U+0021 to U+007E are allowed throughout the source code of a C++ function block.

Explicitly, the following characters are allowed:

! " # $ % & ' ( ) * + , - . /

0 1 2 3 4 5 6 7 8 9 : ; < = > ?

@ A B C D E F G H I J K L M N O

P Q R S T U V W X Y Z [\ ] ^ _

` a b c d e f g h i j k l m n o

p q r s t u v w x y z { | } ~

Additionally, standard space (U+0020) and horizontal tab (U+0009) are allowed throughout the source code of a C++ function block.

The following additional characters U+00A1 through U+00FF are allowed within comments, e.g., « ± ° ¹ ² ³ ¼ ½ ¾ µ Ä Ö Ü ä ö ü ß

5.5 Rules for File Names

The following rules apply to the file names of additional source files of a C++ function block.

- The file name may only contain the following characters:
  0 1 2 3 4 5 6 7 8 9 A B C D E F G H I J K L M N O P Q R S T U V W X Y Z _ a b c d e f g h i j k l m n o p q r s t u v w x y z
- The file name must not contain white spaces (U+0020) and must have the extension h. or .cpp.
- The file name must not start with one of the following character sequences SRC, HIMA or IEC. This also applies for any usage of upper or lower case.
- The file name must not correspond to the name of a header file installed with SILworX. (These are, for example, the following files with the extension .h from the directory structure under target_codegen in the PADT installation: amdecl.h, ansi.h, asm.h, base.h, binders.h, cast.h, complex.h, config.h, cpl.h, ctype.h, errno.h, glu.h, hidpi.h, limits.h, lock.h, m68k.h, malloc.h, map.h, math.h, rs6000.h, stat.h, time.h, varargs.h, wchar.h)

5.6 Illegal C++ Language Elements

A C++ function block must not be used to change or circumvent the semantics of the C++ source code sections generated by SILworX.

A C++ function block may or must not use the following features:

- Dynamic Memory Management
- goto
- Inline Assembler
- How to add libraries in binary form
- Recursion
- Interrupts
- #pragma
- Runtime Type Information (RTTI)
  - dynamic_cast
  - typeid
  - Exception-Handling

5.6.1 Macros

A C++ function body may not add or delete macro definitions with macro identifier starting with one of the following prefixes:

_, AM_, AMDEF_, AMDECL_, DLL_, HIMA_, IEC_, LS_, SECTION_, SRCUSR_, TARGET_, X_

5.6.2 Include

A C++ function block may not shape an include directive using a macro.

Example (the following is not allowed):

```
#define INCFILE "incfile.h"
#include INCFILE
```

5.6.3 Data Types

A C++ function block must not use pointers that are described as "pointers to non-static class members" by the C++ standard.

A C++ function block must not dereference a null pointer.

A C++ function block must not use the long double data type.

A C++ function block must not define classes/unions within a function.

5.6.4 Functions

A C++ function block must not define any global functions named main, WinMain or DllMain. Not within a namespace as well!

5.6.5 Variables

A C++ function block must not define any static variables within a function.

A C++ function block must not define any static data elements in classes/unions.

5.6.6 Literals

A C++ function block must not use any string literals.

5.6.7 Operators

A C++ function block must not alter any data declared as const by SILworX.

5.6.8 Other Syntax

The source text of a C++ function block must not contain external declarations with specification of a programming language.

Examples (the following is not allowed):

extern "C" typedef void (*CFunction)(void);

extern "C" { void Func1(void); }

5.6.9 Use of SILworX Global Variables

A C++ function block must not access SILworX global variables, either with write or with read rights.

Appendix

Index of Tables

HI 801 319 E

HIMA Paul Hildebrandt GmbH
Albert-Bassermann-Str. 28 | 68782 Brühl, Germany
Phone +49 6202 709-0 | Telefax +49 6202 709-107
info@hima.com | www.hima.com

SAFETY
NONSTOP

For a detailed list of all our subsidiaries and representatives,
please visit our website: www.hima.com/contact