Documentation

X2C® v6.6.4780
Free Edition

September 26, 2026

© Linz Center of Mechatronics GmbH

Contents

I Installation 6

1 Software versions 6

2 Setup with *Scilab* & *Xcos* support 6
- 2.1 Installation . . . 6
- 2.2 Uninstallation . . . 6

3 Configuration of *Code Composer Studio* 7
- 3.1 Install the TI v16.9.5 compiler . . . 7
- 3.2 *Texas Instruments* target processor types . . . 7
  - 3.2.1 Supported processors families . . . 7
  - 3.2.2 Change target processor in *Code Composer Studio* . . . 8
- 3.3 Change predefined Symbols . . . 8

4 Configuration of *MPLAB® X* 9
- 4.1 Install the XC16 compiler . . . 9
- 4.2 *Microchip* target processor types . . . 9
  - 4.2.1 Supported processors families . . . 9
  - 4.2.2 Change target processor in *MPLAB X* . . . 9
- 4.3 Change predefined Symbols . . . 10

II General 11

5 Introduction to *X2C®* 11
- 5.1 General structure of X2C . . . 11
- 5.2 Connection between frame program and model . . . 11
- 5.3 Boolean data representation . . . 12
- 5.4 Fixed point data representation . . . 13
  - 5.4.1 Standard signals . . . 13
  - 5.4.2 Unlimited/Unbalanced signals . . . 14
- 5.5 Floating point data representation . . . 14
  - 5.5.1 Standard signals . . . 14
  - 5.5.2 Unlimited/Unbalanced signals . . . 15
- 5.6 Angular Signals . . . 15
- 5.7 Simulation . . . 16
- 5.8 Restrictions . . . 16
  - 5.8.1 Algebraic loops . . . 16
  - 5.8.2 Connection of blocks with different implementations . . . 17
- 5.9 Model Identifier . . . 17
  - 5.9.1 Communicator and LNet services . . . 17
  - 5.9.2 Method . . . 17

6 Basic structure of the C Code 18
- 6.1 Main.c . . . 18
- 6.2 Hardware.c . . . 18

7 Testing 19
- 7.1 JUnit tests . . . 19
- 7.2 CUnit tests . . . 19

8 Coding Conventions **20**
8.1 Language . . . 20
8.2 General naming conventions . . . 20
8.3 Naming of files . . . 20
8.4 Naming of functions and methods . . . 20
8.5 Naming of macros . . . 20
8.6 Naming of variables . . . 20
8.7 Naming of type definitions . . . 21
8.8 Naming of model parameters . . . 21
8.9 Naming of X2C blocks . . . 21
8.10 Formatting . . . 21
8.10.1 Intending . . . 21
8.10.2 Braces . . . 21
8.10.3 Spaces . . . 22
8.10.4 Return statement . . . 22
8.10.5 Example . . . 22
8.11 Source and header files . . . 23
8.12 File headers . . . 23
8.13 Global definitions . . . 23
8.14 Template files . . . 24
8.15 Include order of header files . . . 25
8.16 Hardware registers . . . 25
8.17 Doxygen documentation . . . 25
8.18 List of commonly used abbreviations . . . 26

9 MISRA-C 2004 compliance **27**
9.1 Applied rules . . . 27

10 *Scilab* & *Xcos* user notes **28**
10.1 Xcos Diagram . . . 28

III Utilities **29**

11 Communicator **29**
11.1 *Xcos Communicator* start . . . 29
11.2 Standalone *Communicator* start . . . 29
11.3 Basic functions of the *Communicator* . . . 29
11.4 Settings . . . 33
11.5 Change parameters on the target with the *Communicator* . . . 34

12 Scope **35**

13 Block Generator **37**
13.1 Block properties . . . 37
13.2 Implementation properties . . . 39
13.2.1 Advanced Implementation Settings . . . 40
13.3 Save or load a block . . . 40

IV Add-Ons **42**

14 Code For Speed Optimizer (CFSO) **42**
14.1 Data Collection . . . 42
14.1.1 Implementations using a function link . . . 43
14.2 Code Generation . . . 43

15 Dashboard **44**
15.1 Configuration File . . . 44
15.2 Common properties . . . 45
15.3 Element types . . . 46
15.3.1 Block-Parameter . . . 46
15.3.2 Variables . . . 47
15.4 Model Override . . . 48

V How-To **49**

16 *X2C®* code generation with *Scilab* **49**

17 Loading and building the demo application Blinky in *Code Composer Studio* **51**

18 Loading and building the demo application Blinky in *MPLAB® X* **52**

19 Loading and building the demo application Blinky in *STM32CubeIDE* **54**

20 Loading and building the demo application Blinky in *DAVE IDE* **57**

21 The creation of an external project-specific *X2C®* block **61**
21.1 The creation of the basic structure . . . 61
21.2 Coding the source file . . . 64
21.3 Coding the conversion function . . . 65
21.3.1 A conversion function in Java . . . 66
21.3.2 A conversion function in JavaScript . . . 67
21.3.3 A conversion function in Python . . . 67
21.3.4 A conversion on target function (CoT) in C . . . 67
21.4 Finalizing the block in Scilab . . . 68
21.5 The block in an Xcos model . . . 68
21.6 The block in Code Composer Studio . . . 68

22 Setup *X2C®* for use in a *B&R® Automation Studio®* project **69**
22.1 Configuration . . . 69
22.2 Logical View . . . 71
22.3 Software Configuration . . . 72
22.4 Communication configuration . . . 72

VI Libraries **73**

23 Basic **73**
CommunicatorStart (Xcos only) . . . 74
CreateDocumentation . . . 75
Interact (Xcos only) . . . 76
ModelTransformation (Xcos only) . . . 77

24 Control **78**
AdaptivePT1 . . . 79
D . . . 80
Delay . . . 81
DLimit . . . 82
DT1 . . . 83
I . . . 84
ILimit . . . 85
P . . . 86
PI . . . 87
PID . . . 88
PIDLimit . . . 89
PILimit . . . 90
PLimit . . . 91
PT1 . . . 92
TDSystemO1 . . . 93
TDSystemO2 . . . 94
TF1 . . . 95
TF2 . . . 96
uI . . . 97

25 Filter **98**
ACDC . . . 99
AdaptiveNotch . . . 100
BandpassBiQ . . . 101
BandstopBiQ . . . 102
Bilin . . . 103
Biquad . . . 104
FundFreq . . . 105
HighpassBiQ . . . 106
IIR . . . 107
LowpassBiQ . . . 108
MinMaxPeriodic . . . 109
NotchBiQ . . . 110

26 General **111**
And . . . 112
AutoSwitch . . . 113
Constant . . . 114
Gain . . . 115
Inport . . . 116
Int2Real . . . 117
Limitation . . . 118
LookupTable . . . 119
LookupTable1D . . . 120
LookupTable2D . . . 121
LookupTable3D . . . 122
LoopBreaker . . . 123
ManualSwitch . . . 124
Maximum . . . 125
Minimum . . . 126
Not . . . 127
Or . . . 128

Outport . . . 129
RateLimiter . . . 130
Real2Int . . . 131
Saturation . . . 132
SaveSignal . . . 133
Selector . . . 134
Sequencer . . . 135
Sin2Limiter . . . 136
Sin3Gen . . . 137
SinGen . . . 138
TypeConv . . . 139
uConstant . . . 140
uGain . . . 141
uRateLimiter . . . 142
uSaveSignal . . . 143
Xor . . . 144

27 Math **145**
Abs . . . 146
Acos . . . 147
Add . . . 148
Asin . . . 149
Atan2 . . . 150
Average . . . 151
Cos . . . 152
Div . . . 153
Exp . . . 154
L2Norm . . . 155
Mult . . . 156
Negation . . . 157
Sign . . . 158
Sin . . . 159
Sqrt . . . 160
Sub . . . 161
Sum . . . 162
uAdd . . . 164
uSub . . . 165

Part I
Installation

1 Software versions

Following software versions were tested for full *X2C*® functionality:

Software	Version
Required:	
Scilab (www.scilab.org)	6.1.1
Optional (for standalone operation):	
Java Runtime Environment	Java SE 8 / ojdkbuild 13 JRE
Optional (for documentation):	
MiKTeX (www.miktex.org)	2.9
Doxygen (www.doxygen.org)	1.9.0
Graphviz (www.graphviz.org)	2.38
Optional (for programming):	
TI *Code Composer Studio*	11.x
TI Code Generation Tools	c2000_16.9.5.LTS / arm_16.9.4.LTS
TI TivaWare	TivaWare_C_Series-2.x.x.xxx
ST *STM32CubeIDE*	1.15.x
Infineon *DAVE IDE*	4.x.x
Microchip *MPLAB*® *X*	6.xx
Microchip XC16	1.xx

Different versions of these programs may work but without warranty.

2 Setup with *Scilab* & *Xcos* support

2.1 Installation

1. Open *Scilab* and with the *File Browser* navigate to <X2C_ROOT>\System\Scilab\Scripts. Right click on **setup.sce** and click *Execute in Scilab*.

2. Restart *Scilab*

3. The setup command creates a *X2C* configuration file which will automatically load *X2C* libraries and palettes at startup of *Scilab* .

2.2 Uninstallation

1. Open *Scilab* and execute the command `initX2C(%f)` in the *Scilab* console.

2. Restart *Scilab*

3. Once above command was executed, the *X2C* configuration file is deleted and *Scilab* will not load any *X2C* libraries or palettes anymore.

For the unlikely event that Scilab freezes at startup and remains in a deadlock state, the deinstallation can be done manually by deleting the file **scilab.ini** located in the Scilab home directory (for Windows typically C:\Users\<your user name>\AppData\Roaming\Scilab\scilab-6.x.x).

3 Configuration of *Code Composer Studio*

3.1 Install the TI v16.9.5 compiler

It is necessary to use the compiler version TI v16.9.5 in *Code Composer Studio* in combination with *X2C* . Navigate to **Project** → **Properties** click **General** and in the **Advanced settings** area see what compiler versions are available. It is necessary to use the compiler version **TI v16.9.5**. If this version is not selectable go to **Help** → **Install new Software** and in the **Work with** drop down menu choose **Code Generation Tools Update**. In the section **TI Compiler Updates** find *C2800 Compiler Tools Version 16.9.5* and mark it as seen in Figure 1. Click **Next** and install the update. Now go back to the Project Properties and change the compiler.

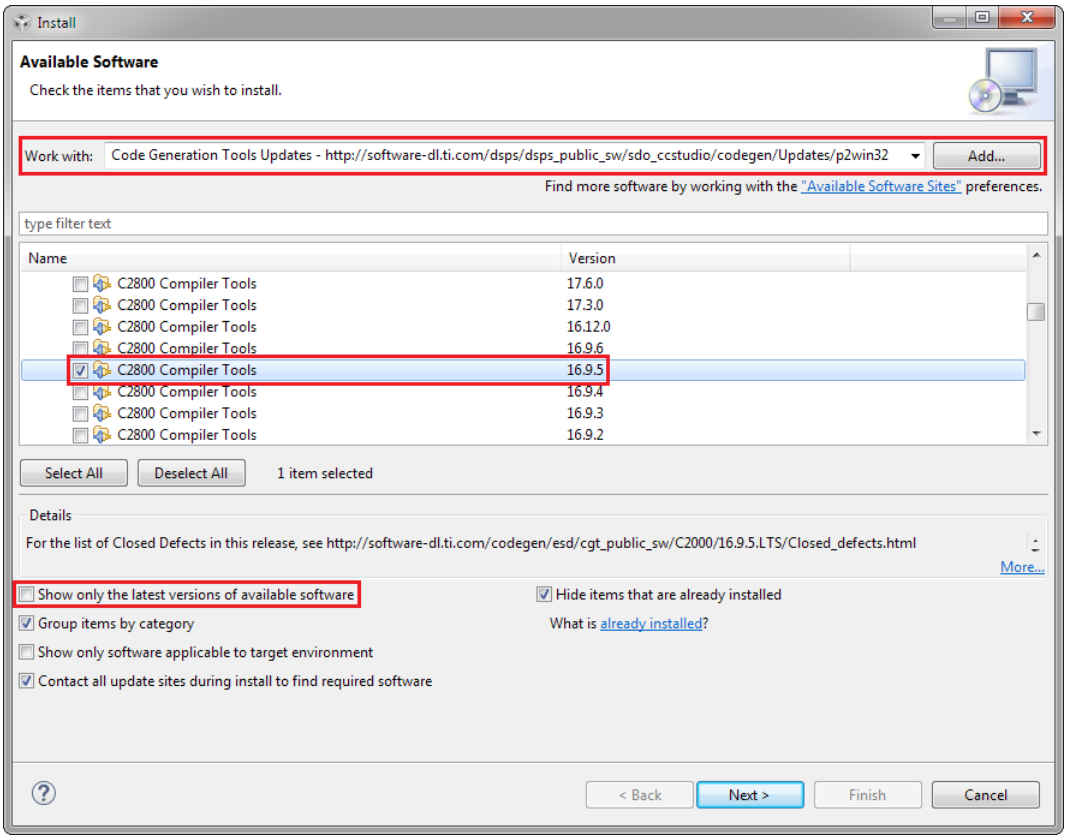


Figure 1: *Code Composer Studio* Compiler Download

3.2 *Texas Instruments* target processor types

3.2.1 Supported processors families

Currently the following *Texas Instruments* processor families are supported by *X2C* .

- TI C28x 32-Bit CPU
- TI TM4C12x 32-Bit CPU (ARM Cortex-M4 core)

3.2.2 Change target processor in *Code Composer Studio*

In the following section file names may vary with different processor types.

1. Import the *Blinky* demo application in *Code Composer Studio* (see Section 17 for more information).

2. Change the *Predefined symbols* (see Section 3.3) suitable for the used processor type.

3. With the *OS* file browser navigate to the *controlSUITE* subdirectory device_support and search for your processor type (e.g. C:\ti\controlSUITE\device_support\f2806x\v130\F2806x_headers).

4. Copy the folders *cmd, include* and *source* into the project directory <PROJECT_DIRECTORY>\TexasInstruments and replace the existing folders from the processor used in the *Blinkdy* demo application.

5. In *Code Composer Studio* open the **F28xxx_Device.h** file in the folder <PROJECT_DIRECTORY>\TexasInstruments\include. In the section *User To Select Target Device* search for your processor and change the 0 to `TARGET`. An example is shown in Figure 2.

```
#define   DSP28_28067P     0
#define   DSP28_28067UP    0
#define   DSP28_28067PZ    0
#define   DSP28_28067UPZ   0

#define   DSP28_28068P     0
#define   DSP28_28068UP    0
#define   DSP28_28068PZ    0
#define   DSP28_28068UPZ   0

#define   DSP28_28069P     0
#define   DSP28_28069UP    0
#define   DSP28_28069PZ    0
#define   DSP28_28069UPZ   TARGET
```

Figure 2: Change processor type in the *device.h* file

6. In *Code Composer Studio* open the files *Hardware.h* and *X2cDataTypes.h* and adapt the file names in the *include* directives for the *F28xxx_Device.h* file.

3.3 Change predefined Symbols

The *X2C* project uses predefined symbols to give the preprocessor information before compiling the project. Navigate to **Project** → **Properties** open **Build** → **C2000 Compiler** → **Predefined Symbols**.
Currently three different processor families can be choosen

- `__GENERIC_TI_C28X__` for *Texas Instruments* Processors
- `__GENERIC_ARM_ARMV7__` for *ARM* Processors
- `__GENERIC_MICROCHIP_DSPIC__` for Microchip Processors

The definition

- `__CUSTOM_DATATYPE_DEFINITIONS__`

is needed to avoid compiler warnings caused by multiple *typedefs*.

In addition the definition

- `SCOPE_SIZE=8000`

like seen in Figure 3 needs to be made. The value of *Scope Size* is changeable and depends on the intended application and the used target processor. In the *Blinky* demo applications these values are already defined.

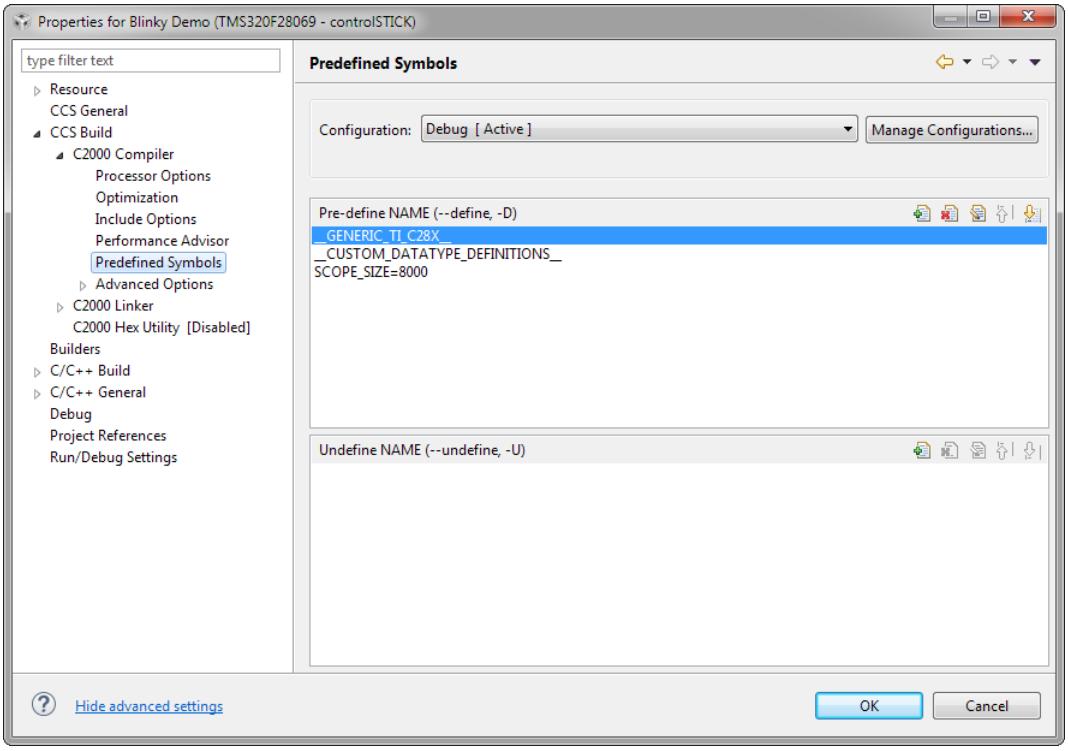


Figure 3: Predefined symbols for generic processor type in *Code Composer Studio*

4 Configuration of *MPLAB® X*

4.1 Install the XC16 compiler

When working with *MPLAB X* the compiler to build the project has to be installed manually. Which compiler is needed depends on the used processor type. In the demo application *Blinky* the *xc 16 v1.21* compiler from the *Microchip* web page (`http://www.microchip.com/pagehandler/en_us/devtools/mplabxc/`) can be used.

4.2 *Microchip* target processor types

4.2.1 Supported processors families

Currently the following *Microchip* processor families are supported by *X2C* .

- dsPIC 16-Bit CPU

4.2.2 Change target processor in *MPLAB X*

Right click on the **Project** → **Properties**. In the *Configuration* area *Devices* can be picked in the drop down menu. Click **OK** to save the changes.

4.3 Change predefined Symbols

The *X2C* project uses *predefined Symbols* to give the preprocessor information before compiling the project. In the the sample project these symbols are already defined. Right click on the **Projectname** → **Properties** → **XC16 Global Options** → **xc16-gcc** to eventually change them. In the section *Define C macros* a list of defines is available as seen in Figure 4. Depending on the used target processor three different processor families can be choosen

- `__GENERIC_TI_C28X__` for *Texas Instruments* Processors
- `__GENERIC_ARM_ARMV7__` for *ARM* Processors
- `__GENERIC_MICROCHIP_DSPIC__` for Microchip Processors

In addition the definition

- `SCOPE_SIZE=5000`

like seen in Figure 4 needs to be made. The value of *Scope Size* is changeable and depends on the intended application and the used target processor. In the *Blinky* demo applications these values are already defined.

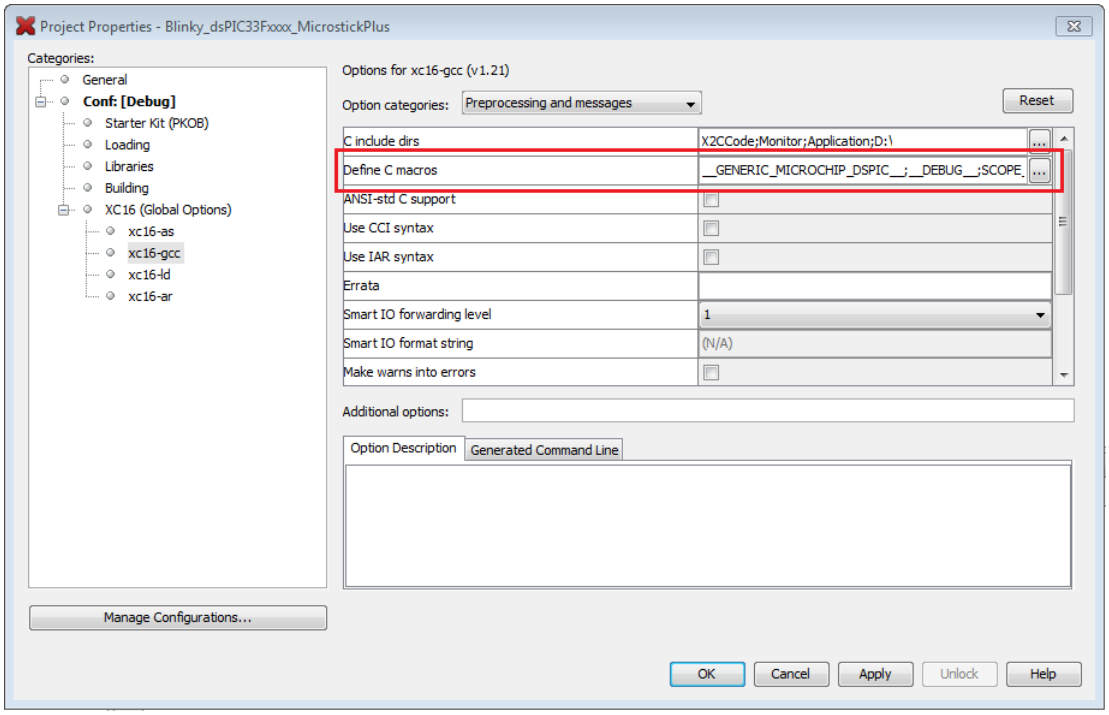


Figure 4: Predefined Symbols in *MPLAB X*

Part II
General

5 Introduction to *X2C®*

5.1 General structure of X2C

X2C can be seen as a two layer application, as it is depicted in figure 5.

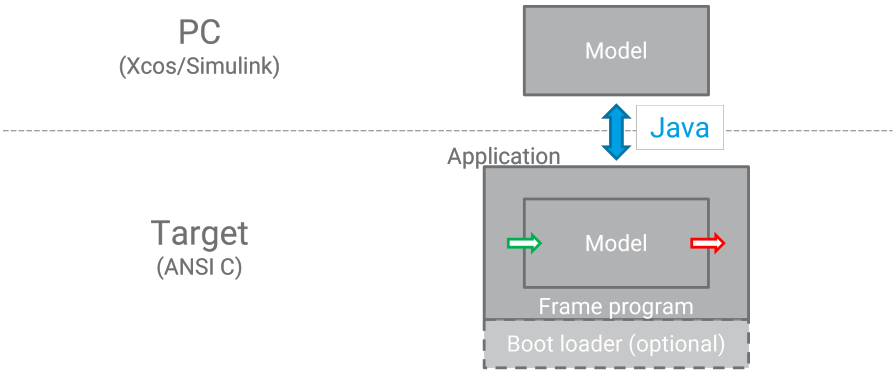


Figure 5: General X2C structure

The top layer is the abstraction layer located on the PC. Either in Xcos or in Simulink the model-based development takes place. This means the user designs the control algorithm graphically.
Based on the graphical model X2C generates C-code with its java tool Communicator. The generated C-code together with the so called frame program form the application. Since the model-based design is independent of hardware, the generated model code is (more or less) too. The model code is called from the frame program, which is manually coded and handles the initialization and configuration of the hardware peripherals and provides the hardware signals for the application.
In the commercial version a bootloader is available. With this bootloader programming of the target can be executed over Serial/CAN/Ethernet interface, so no programming device is needed.

5.2 Connection between frame program and model

As already mentioned in the previous section, the generated code from the model is embedded in the manually coded frame program. For the most minimalist application imaginable (no communication between target and PC, no hardware peripherals used by the target), the frame program consists of only two function calls:

- X2C_Init() once at startup
- X2C_Update() cyclically with the sample time defined in the model

The schematic structure of such a minimal frame program can be seen in figure 6a, where the generated code by X2C is marked in light blue.

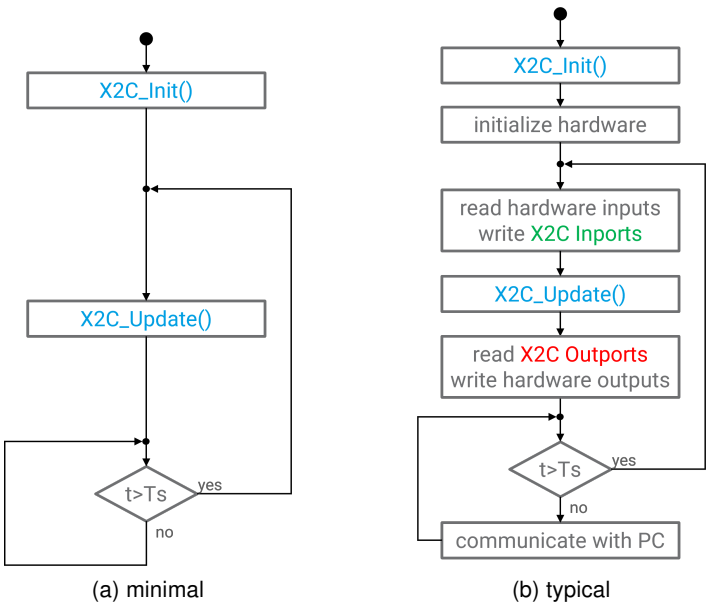


Figure 6: Frame program structure

However, in real life applications hardware peripherals will be used, and in most cases an X2C communication should be possible. Therefore, the startup process will be enhanced with an initialization routine of the target's peripherals, such as ADC, Timer, Interrupt, PWM, etc. and communication interfaces such as UART, CAN, Ethernet, etc.
The cyclic part of the frame program will be enhanced too. Before the X2C_Update() function will be called, the hardware inputs are read, and the values are assigned to the corresponding X2C Inports. For example, given an X2C Inport labelled as *AnIn1* in the model and the analog-to-digital conversion result available as *ADCResult[0]* variable in the frame program, the code would be:

```
x2cModel.inports.bAnIn1 = ADCResult[0];
```

After the X2C_Update() function call the computed X2C Outports are read and the values put to the corresponding hardware outputs. For example, given an X2C Outport labelled *LED1* and the target's name for port A hardware register is *GPADAT* the code for controlling a LED connected to pin 4 of hardware port A could be:

```
if (*x2cModel.outports.bLED1 != 0) {
    GPADAT |= 0x0008;
} else {
    GPADAT &= 0xFFF7;
}
```

While the target waits for the next cycle the X2C communication can be executed. It enables online parameter tuning and signal monitoring with X2C Communicator and Scope.
These enhancements are depicted in figure 6b.
Typically, the cyclic part is handled in a high priority interrupt service routine and the communication with the host PC is handled in a low priority task, e.g. the while(1) loop.

5.3 Boolean data representation

Boolean data is based on the header *stdbool.h* introduced in C99 in the C Standard Library for the C programming language.

Bool	
Implementation type	Boolean
Format	C99
Minimum value	0 (false)
Maximum value	1 (true)

5.4 Fixed point data representation

5.4.1 Standard signals

Standard signals are symmetrically scaled about zero and their scaling range is]-1...1[. In a case of an overflow the signal will be limited to 1 (or -1 respectively). For example a subtraction of a signal with value 0.5 from a signal with value -0.7 will lead to a signal with value -1.
Depending on the chosen implementation the values are handled in one of the following formats:

FiP8	
Implementation type	8-bit fixed point
Format	Q7
Minimum value	$-0.992\,187\,500$
Maximum value	$0.992\,187\,500$
Resolution	$0.007\,812\,500$

FiP16	
Implementation type	16-bit fixed point
Format	Q15
Minimum value	$-0.999\,969\,482\,421\,875$
Maximum value	$0.999\,969\,482\,421\,875$
Resolution	$0.000\,030\,517\,578\,125$

FiP32	
Implementation type	32-bit fixed point
Format	Q31
Minimum value	$-0.999\,999\,999\,534\,339$
Maximum value	$0.999\,999\,999\,534\,339$
Resolution	$0.000\,000\,000\,465\,661$

5.4.2 Unlimited/Unbalanced signals

The scaling of unlimited/unbalanced signals is [-1...1[. While the standard signals omit one value to achieve a symmetrical value range, the unlimited (or also called unbalanced) signals utilize the full value range which leads to a slightly unbalanced value range. As the name implies unlimited signals won't be limited. In fact unlimited signals utilize wrapping/overflow functions of the DSP. For example a subtraction of a signal with value 0.5 from a signal with value -0.7 will lead to a signal with value 0.8.
So the primary use of the unlimited signal is as angular signal where (-1...1) corresponds to (-π...π).
Depending on the chosen implementation the values are handled in one of the following formats:

FiP8	
Implementation type	8-bit fixed point
Format	Q7
Minimum value	$-1.000\,000\,000$
Maximum value	$0.992\,187\,500$
Resolution	$0.007\,812\,500$

FiP16	
Implementation type	16-bit fixed point
Format	Q15
Minimum value	$-1.000\,000\,000\,000\,000$
Maximum value	$0.999\,969\,482\,421\,875$
Resolution	$0.000\,030\,517\,578\,125$

FiP32	
Implementation type	32-bit fixed point
Format	Q31
Minimum value	$-1.000\,000\,000\,000\,000$
Maximum value	$0.999\,999\,999\,534\,339$
Resolution	$0.000\,000\,000\,465\,661$

5.5 Floating point data representation

5.5.1 Standard signals

The standard signals in floating point format are not restricted, the full value range according to the IEEE 754 standard is available.

Float32	
Implementation type	32-bit floating point
Format	IEEE 754
Minimum value	$-3.402823466385288598117041834845 2e+38$
Maximum value	$3.402823466385288598117041834845 2e+38$
Resolution	$\pm 1.175494350822287507968736537222 2e-38$ (normalized)

Float64	
Implementation type	64-bit floating point
Format	IEEE 754
Minimum value	$-1.797693134862315708145274237317e+308$
Maximum value	$1.797693134862315708145274237317e+308$
Resolution	$\pm 2.225073858507201383090232717332 4e-308$ (normalized)

5.5.2 Unlimited/Unbalanced signals

Contrary to their names the unlimited/unbalanced signals in floating point format are limited to $[-\pi, +\pi]$. All other properties are similar to the unlimited signals in fixed point format.

Float32	
Implementation type	32-bit floating point
Format	IEEE 754
Minimum value	$-3.14159265358979323846264338327 95$
Maximum value	$3.14159265358979323846264338327 95$
Resolution	$\pm 1.175494350822287507968736537222 2e-38$ (normalized)

Float64	
Implementation type	64-bit floating point
Format	IEEE 754
Minimum value	$-3.14159265358979323846264338327 95$
Maximum value	$3.14159265358979323846264338327 95$
Resolution	$\pm 2.225073858507201383090232717332 4e-308$ (normalized)

5.6 Angular Signals

The unlimited/unbalanced signals are typically used for angular signals, especially in the proprietary X2C *MotorControl* library. An angular signal is needed for the field oriented control of permanent magnet synchronous motors. For such a case, the zero value has to be defined.

For the X2C *MotorControl* library the zero value is the angle where the Back-EMF voltage of the first motor phase has its negative zero-crossing (see figure 7).

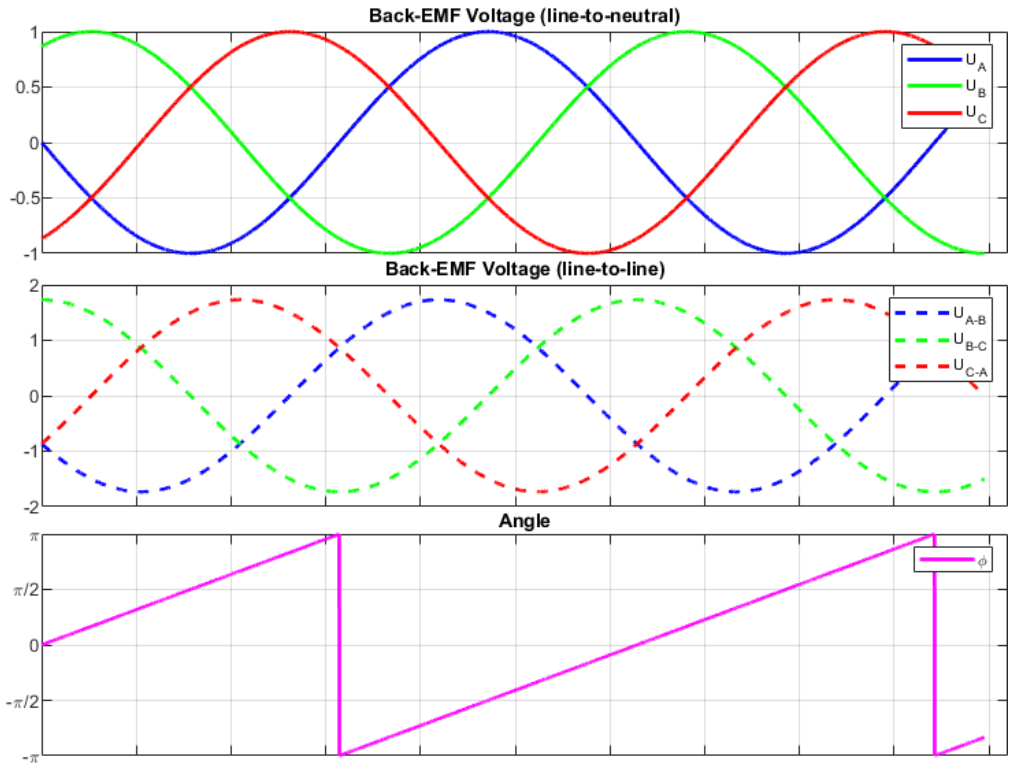


Figure 7: Angle in respect to Back-EMF voltage

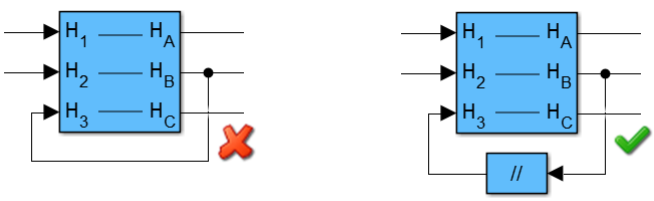
5.7 Simulation

It is possible to simulate a *X2C*-model in *Xcos* . All blocks except hardware dependent ones (e.g. Inport, Outport) contain glue code which are linked with the corresponding source code of the *X2C*-block. This enables a simulation of a *X2C*-model in *Xcos* . Hence testing of DSP control algorithms is possible without the need for a DSP connected to the simulation PC.

5.8 Restrictions

5.8.1 Algebraic loops

Algebraic loops as depicted in Figure 8a are not possible due to a execution order problem. Therefore a block is required which breaks the loop at a specific position. This can be achieved by inserting a block with no direct feedthrough functionality, e.g. Block: LoopBreaker from the X2C *General* library (see Figure 8b).



(a) forbidden (b) allowed

Figure 8: Algebraic loops

5.8.2 Connection of blocks with different implementations

Though blocks with different implementations are allowed (and computed correctly) in the same model, connections of ports with different datatypes are not permitted. The conversion blocks Block: TypeConv, Block: Int2Real and Block: Real2Int can be used to resolve datatype incompatibilities.

5.9 Model Identifier

The Model identifier is an 32-bit value to uniquely identify an *X2C* Model based on its

- Blocks and the used Implementation
- Block connections
- Block label names
- Block Parameter identifiers

However, the initial Block parameter values are *NOT* part of the Model identifier. So, if you only change the initial Block parameters, the Model identifier will not change. The Model identifier is added to the generated *X2C* files and may be used by the user application as well.

5.9.1 Communicator and LNet services

The *X2C* Communicator is able to check the Model Identifier on the target against the currently loaded *X2C* Model in the Communicator itself. To enable this feature, the user application must implement the LNet Model Identifier services. Once these services are available on the target, the Communicator checks the Model Identifiers on every connect. In case of a Model Identifier mismatch, the Communicator outputs a warning in the log window and shows a warning dialog. The latter can be disabled in the Communicator settings. If the target does not support the Model Identifier feature, the Communicator outputs a warning in the log window as well.

5.9.2 Method

The Model Identifier uses a CRC-32 checksum.
To add the elements in the correct order, the following steps are required:

1. Update the Parameter Identifiers of all Blocks in the Model so that each Block has a valid Parameter ID
2. Sort all Blocks ascending by the Parameter ID
3. Repeat the following sequence for each block
   (a) Get the unique Block label (containing all parent sub-systems) and add the ASCII value of each character to the checksum, starting with the first character
   (b) Get the 16-bit Block Implementation identifier and add the value as 2 8-bit values, LSB first
   (c) Get the 16-bit Parameter ID and add the value as 2 8-bit values, LSB first
   (d) Get the Parameter ID of the Blocks connected to each Inport, starting with the first Inport, and add each 16-bit Parameter ID as 2 8-bit values, LSB first

6 Basic structure of the C Code

When setting up a new project with *X2C* code generation it is necessary to configure the hardware on the target system. The following section provides basic information how the *Blinky* demo applications are structured. With this understanding one should be able to adapt hardware configuration for further projects.
In the following the *Blinky* demo application in combination with the *TI Piccolo F28069 ControlSTICK* and the *TMS320F28069* Processor is used for demonstration.
Info: The *.c files listed below need to be updated in case of changes in the *Xcos* model configuration.

6.1 Main.c

- **initInterruptVector()** defined in *Hardware.c* configures the target specific interrupts.
- **initSerial()** defined in *Hardware.c* initializes the serial interface.
- **initHardware()** defined in *Hardware.c* here the peripheral devices such as *Watchdog, GPIO Ports, ADCs, Timers* and others are defined.
- **X2C_init()** defined in *X2C.c* calls the initialization functions of the *X2C* blocks.
- The **while(1)** loop is mainly used for the serial communication.
- The **mainTask()** function is the key structure of the project. Here the connection between *Outports* and *Inports* are defined and the *X2C_Update()* function (defined in *X2C.c*) is called. The basic structure of the *mainTask()* is

  1. Assign Inports
  2. Call *X2C_Update()*
  3. Update Outports

  The *mainTask()* function is usually called by an *Interrupt Service Routine (Isr)* which can be triggered by multiple sources.
  Example: In the *Blinky Demo Application* after each *ADC* conversion cycle the *ADCIsr* calls the *mainTask()* function.
- **KICK_DOG** resets the *Watchdog* timer periodically. During operation this timer continuously counts a certain time span (configured in *Hardware.c*). If the application has an unexpected failure *KICK_DOG* can not be called and the *Watchdog* timer exceeds its limit and therefore the target reboots. If the operation executes as expected the *Watchdog* timer is within the limit and can be reseted by *KICK_DOG* without any further actions.

6.2 Hardware.c

In this file all connections and peripheral device settings should be made.

- In **initHardware()** all peripheral function should be initialized.
  - Watchdog timer
  - GPIO Ports
  - Interrupt initialization
  - ADC, Timer, PWM and all the other peripheral devices
- **initSerial()** initializes the serial interface on the target. The settings made here should match with the setting made in the *Communicator* described in Section 11.

7 Testing

7.1 JUnit tests

To minimize the risk of software bugs most parts of X2C are tested. The Java core of X2C is tested wit JUnit tests.

7.2 CUnit tests

Much care is also taken of testing the C-code of the blocks. These so called CUnit tests are conducted directly on the target. In Figure 9 the test setting can be seen.

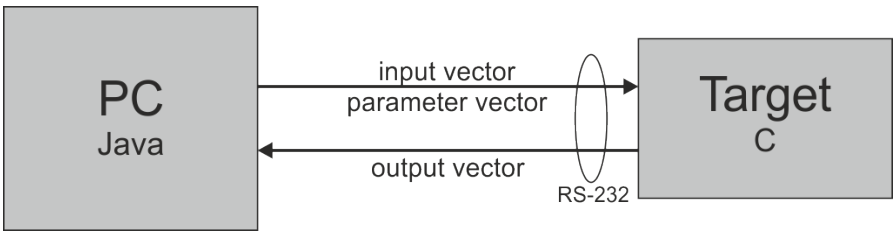


Figure 9: CUnit test setting

In the test environment on the PC a input vector and a parameter vector, if necessary, are defined and sent to the target via serial interface. On the target the update function of the block under test is executed. The resulting output vector is transferred back to the PC where it is compared with a reference output vector. If the difference between the actual and the reference output vector is below a specified limit the test is marked as passed. Otherwise an error is reported.
Basically, the generation of the test vectors are done with one or a combination of these methods:

Equivalence class testing To reduce the number of test items they are grouped into classes with same behavior. Only one member of each equivalence class is used as entry for the test vector. Two simple equivalence classes could be positive and negative numbers.

Boundary testing The entries for the test vectors are chosen in such a way that they lie below and above critical boundaries. For example, typical values for a FiP16 implementation would be -32768, -32767, -1, 0, 1, 32767.

Back-to-back testing For complex blocks this method is used. The vectors are generated by simulating a block or model with the same functionality in Simulink or Xcos.

Several different targets are used for testing. The test reports can be found in the library documentation *Library.source.pdf* in the directory *<X2C_ROOT>\Library*.

8 Coding Conventions

8.1 Language

The native language of X2C is English. Hence all documentation, file names, variables, comments in source files, etc. should be in English.

8.2 General naming conventions

- Unless otherwise stated, all names should use the camel case notation. A definition of camel case can be found on http://en.wikipedia.org/wiki/CamelCase. The type of camel case (upper or lower) depends on the type of name, see sections below.

  Examples: `ThisIsUpperCamelCasing.java`, `showLowerCamelCaseExample()`

- Only meaningful names should be used. Abbreviations should be avoided, unless they are widely used.
- Non-ASCII characters should be avoided. Also the space character schould not be used.
- Due to a character limitation in Scilab 5, names with more than 27 characters should be avoided.
- Names should not start with a number.
- Hungarian notation should not be used.

8.3 Naming of files

In general files should have a meaningful name. If abbreviations are used, easy understandable ones should be used. Upper camel case is recommended.
Examples: `Hardware.c`, `SystemControl.c`, `GlobalDefines.h`

8.4 Naming of functions and methods

Function and method names should contain a verb to describe the action of the function. The verb is placed first and is in lower case and subsequent nouns start with a capital letter (lower camel case).
Examples: `readADC()`, `setPWM()`

8.5 Naming of macros

Macros should be written in capital letters. Macro names which contain mutiple words should use a underscore as separator (screaming snake case).
Examples: `DISABLE_PWM`, `NO_ERROR`

8.6 Naming of variables

Based on the proposed convention from Sun Microsystems following guidelines should be considered:
"Variables are in mixed case with a lowercase first letter. Internal words start with capital letters. Variable names should be short yet meaningful. The choice of a variable name should be mnemonic- that is, designed to indicate to the casual observer the intent of its use. One-character variable names should be avoided except for temporary "throwaway"

variables. Common names for temporary variables are i, j, k, m, and n for integers; c, d, and e for characters."
Examples: `int16_t i; float32_t myWidth;`

8.7 Naming of type definitions

Type definitions as well as structure definitions should be written in upper camel case to distinguish them from variables.
Example:

```
typedef uint16_t MyType;

struct MyStruct
{
    uint16_t member1;
    int32_t member2;
};

struct MyStruct myStruct1;
struct MyStruct myStruct2;
```

8.8 Naming of model parameters

Parameter and variable names in Matlab or Scilab should follow lower snake case notation. This means the first word can either start with a lower or upper case letter, all subsequent words have to start with a lower case letter, and the words are separated by underscores.
Examples: `i_ref, n_max, k_T, U_dc_max`

8.9 Naming of X2C blocks

Block and Subsystem/Superblock names in Matlab or Scilab should follow upper camel case notation. Exceptions are words with abbreviations, then a underscore character as separator is allowed to increase readability.
Examples: `CurrentController, OffsetAngle, AnIn1, DigOut1, i_Phase1, m_Phase1, iq_Ref`

8.10 Formatting

8.10.1 Intending

For code intending **four spaces** should be used, tabs should be avoided.

8.10.2 Braces

Braces should be placed according to the Allman style. *"This style puts the brace associated with a control statement on the next line, indented to the same level as the control statement. Statements within the braces are indented to the next level."*
Though braces in separate lines increase the line count they have some advantages in terms of identification of code blocks, selecting statements by pre-processor macros, commenting out statements without breaking the code, etc.
An exception is the do-while loop, where the braces should be on the same line as the statements to make it easy to see that the while statement belongs to the loop.

8.10.3 Spaces

Between a statement and its opening parentheses a space should be inserted.
A space is also suggested after the comma between two function parameters, variables, array elements, etc.
Before and after an operator, relation symbol, etc. a space should be inserted.

8.10.4 Return statement

The return argument shall be enclosed in parentheses.

8.10.5 Example

Listing 1: Formatting example

```
/**
 * @brief Function to read ADC results.
 *
 * @param result Array with ADC results
 * @param method Variable which averaging method information
 * @return Number of iterations
 */
int16_t readInputs(uint16_t result[], int8_t method)
{
    int16_t x = 0, i = 0;

    if (method < 0)
    {
        readADC(result[0], result[1]);
    }
    else if (method > 0)
    {
       readADC(result[2], result[3]);
    }
    else
    {
        /* Do nothing in the else branch */
    }

    for (i = 0;  i < MAX_SIZE; i++)
    {
        x++;
    }

    while (x < 0)
    {
        x = x + 2;
    }

    do {
        x = x - i;
    } while (x > 0);

    return (x);
}
```

8.11 Source and header files

Every *.c source file should have a corresponding *.h header file. A minimalistic header with prototype definitions is sufficient.
Due to MISRA rule 8.1 (see MISRA-C 2004 compliance) it is required to have prototypes for every function, including static ones. To avoid conflicts between global and static function prototypes when including header files, the following rule shall apply:

- Global function prototypes should be located in its header file
- Static function prototypes should be located at the beginning of its source file

8.12 File headers

X2C developers should use following headers in the source files.

Listing 2: *.c header template

```
/*
 * $Revision: 4131 $
 * $Date:: 2026-03-16 17:20:39 +0100#$
 *
 * ===== CONFIDENTIAL =====
 * The content of this file is confidential according to the X2C
    Licence Terms and Conditions.
 *
 * Copyright (c) 2026, Linz Center of Mechatronics GmbH (LCM), web:
    www.lcm.at
 * All rights reserved.
 */
```

Listing 3: *.h header template

```
/**
 * @file
 * @brief Enter here a brief description
 */
/*
 * $Revision: 4131 $
 * $Date:: 2026-03-16 17:20:39 +0100#$
 *
 * ===== CONFIDENTIAL =====
 * The content of this file is confidential according to the X2C
    Licence Terms and Conditions.
 *
 * Copyright (c) 2026, Linz Center of Mechatronics GmbH (LCM), web:
    www.lcm.at
 * All rights reserved.
 */
```

8.13 Global definitions

Definition of macros which are used in more than one source file should be placed in `GlobalDefines.h`. Macros only used in one file should be defined in the source file in which they are used.
Globally needed variables should be defined in `GlobalDefines.c` and declared in `GlobalDefines.h`. This way all global variables are at one place and can be referenced from every file which has the `GlobalDefines.h` header file included. Exceptions are global

variables used in firmware modules/drivers which can not be substitued without major drawbacks.
Example:

Listing 4: GlobalDefines.c

```c
#include "GlobalDefines.h"

/*****************************************************************/
/* Global Variables */
/*****************************************************************/
uint16_t errorState  = NO_ERROR;  /* Error Message */
uint32_t moduleState = NO_ERROR;  /* Module Status */
FIStates fiState = RESET_STATE;   /* State of frequency inverter */
```

Listing 5: GlobalDefines.h

```c
#ifndef GLOBALDEFINES_H
    #define GLOBALDEFINES_H

    #include "Target.h"
    #include "X2C.h"

    /*****************************************************************/
    /* Global Variables */
    /*****************************************************************/
    extern uint16_t errorState;   /* Error Message */
    extern uint32_t moduleState;  /* Module Status */
    extern FIStates fiState;      /* State of frequency inverter */

#endif
```

8.14 Template files

For an easy orientation standard file names should be used in X2C projects:

- `Config.h`: Basic configuration settings for frame program.
- `Main.c`: Frame program main file.
- `Hardware.c`: Hardware configuration and initialization.
- `GlobalDefines.*`: Files with globally needed definitions and variables.
- `SystemControl.c`: File with startup sequence of power electronics and error handling.
- `InterruptControl.c` Interrupt handling, especially interrupt vector table and interrupt service routines.
- `InputControl.c`: Handling of analog and digital inputs.
- `OutputControl.c`: Handling of analog and digital outputs.
- `CANControl.c`: Configuration, initialization and functions of a CAN interface.

Of course, if files contain a lot of code it is recommended to split the file into several ones to maintain comprehensibility.
Example: Splitting of `InputControl.c` in `AnalogInputControl.c`, `DigitalInputControl.c` and `HallSensorControl.c`

8.15 Include order of header files

To avoid conflicts and missing dependencies header files should be included in following order:

1. System headers
2. Application headers
3. Header of current source file

Example:

Listing 6: Main.c

```
#include "VersionInfo.h"
#include "GlobalDefines.h"
#include "Hardware.h"
#include "SystemControl.h"
#include "InputControl.h"
#include "OutputControl.h"
#include "Main.h"
```

8.16 Hardware registers

To maintain comprehensibility and to allow interchangeability of source files hardware (DSP) registers should be accessed via macros.
Example:

```
#define SET_LED      (GPBSET = 0x00000004U)

/* some code */
SET_LED;
/* some more code */
```

instead of

```
/* some code */
GPBSET = 0x00000004;
/* some more code */
```

8.17 Doxygen documentation

The C source code should be documented with doxygen commands so that a documentation can be easily created. Javadoc style is advised, this means a comment block starts with two *'s, like this

```
/**
 * some documentation text
 */
```

Additionaly, the Javadoc style @ character should be used for doxygen tags.
Examples: `@brief`, `@param[in,out]`, `@return`, etc.

To enable documentation, the line

```
/** @file */
```

must be in the file containing the code to be documented, preferable in the *.h header file.

8.18 List of commonly used abbreviations

Abbreviations	
AC	Alternate Current
ADC	Analog to Digital Converter
CAN	Controller Area Network
DC	Direct Current
DSP	Digital Signal Processor
EEPROM	Electrically Erasable Programmable Read-Only Memory
FOC	Field Oriented Control
I2C	Inter-Integrated Circuit
PWM	Pulse Width Modulation
RAM	Random Access Memory
ROM	Read-Only Memory
SPI	Serial Peripheral Interface
UART	Universal Asynchronous Receiver / Transmitter
USB	Universal Serial Bus

9 MISRA-C 2004 compliance

The rules of the Motor Industry Software Reliability Association MISRA should be followed as much as possible. Some major rules are:

- **MISRA-C:2004 2.2/R:** Source code shall only use /* ... */ style comments.
- **MISRA-C:2004 19.4/R:** C macros shall only expand to a braced initialiser, a constant, a string literal, a parenthesised expression, a type qualifier, a storage class specifier, or a do-while-zero construct.
- **MISRA-C:2004 8.12/R:** When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialisation.
- **MISRA-C:2004 18.4/R:** Unions shall not be used.
- **MISRA-C:2004 16.3/R:** Identifiers shall be given for all of the parameters in a function prototype declaration.
- **MISRA-C:2004 19.15/R:** Precautions shall be taken in order to prevent the contents of a header file being included twice.
- **MISRA-C:2004 19.1/A:** #include statements in a file should only be preceded by other preprocessor directives or comments.
- **MISRA-C:2004 8.1/R:** Functions shall have prototype declarations and the prototype shall be visible at both the function definition and call.

9.1 Applied rules

```
/* Enable MISRA-C:2004 checking (all rules) */
--check_misra="all"

/* Rule violation handling */
--misra_advisory="warning"
--misra_required="warning"

/* Exceptions */
--check_misra="-1.1"  /* (MISRA-C:2004 1.1/R)   Ensure strict ANSI C mode (-ps) is enabled */
--check_misra="-12.7" /* (MISRA-C:2004 12.7/R)  Bitwise operators shall not be applied to operands whose underlying type is signed */
--check_misra="-19.7" /* (MISRA-C:2004 19.7/A)  A function should be used in preference to a function-like macro */
--check_misra="-5.7"  /* (MISRA-C:2004 5.7/A)   No identifier name should be reused */
```

Listing 7: MISRA.opt

10 *Scilab* & *Xcos* user notes

10.1 Xcos Diagram

Only blocks from the X2C *Xcos* libraries can be used to construct an X2C model. *Xcos* native blocks are permitted on the left side (input) of an X2C inport block and on the right side (output) of an X2C outport block for simulation purposes.
This limitation does not apply for the following *Xcos* event handling blocks since these are required for the X2C sample time and for X2C blocks that have an event input:

- CLOCK_c to define a base sample time for the X2C model and time dependent X2C blocks
- freq_div to define a different sample time derived from the base sample time for time dependent X2C blocks
- CLKFROM and CLKGOTO to simplify event signal routing

There are several additional exceptions for *Xcos* elements that can be used with or within X2C blocks.

Structual element exceptions

- Super block

Simulation element exceptions

- SCOPE elements from the Sinks palette

Part III
Utilities

11 Communicator

The *Communicator* is the interface between the target system and the model in *Xcos* . It is used to generate the C code files out of the model. Furthermore it is used to transfer data between the computer and the target. When started the *Communicator* is connected with the model via *RMI* interface and via serial interface with the target (DSP).

11.1 *Xcos Communicator* start

As described in Section 16 the *Communicator* is started out of an open *Xcos* model with the buttons *start Communicator*. The button *Transform model and push to Communicator* loads the model file (*.xml*) into the *Communicator*. Changes in the model structure can be made and pushed to the *Communicator* by double clicking on the button *Transform model and push to Communicator*.

11.2 Standalone *Communicator* start

If there are no intended changes in the model structure it is possible to start the *Communicator* without *Xcos* . In <X2C_ROOT>\System\Java double click on **Communicator.jar**. In the open *Communicator* go to **Model** → **Load Model** and browse to your project directory. In the *X2CCode* folder choose the model (*.xml*) file and open it. In the *Status* tab check the *Log* area if the model has been loaded successfully.

11.3 Basic functions of the *Communicator*

The *Communicator* is structured into the menu bar (1) the toolbar (2) and four main tabs (3).

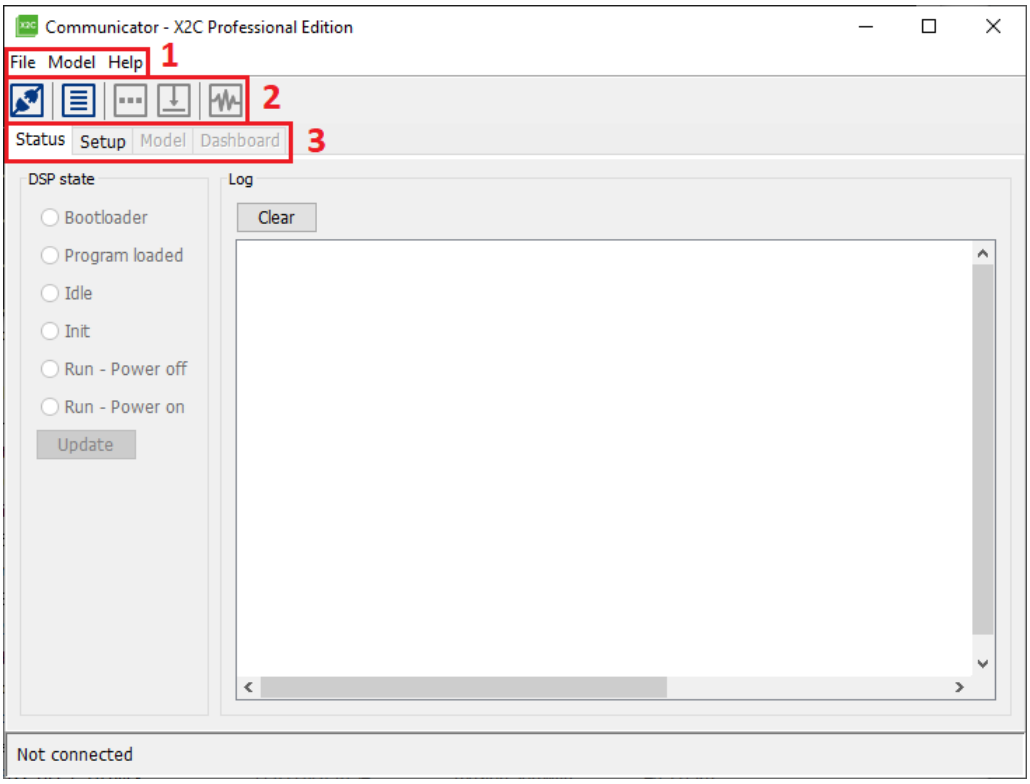


Figure 10: Basic structure of the *Communicator*

1. Menu bar
   - In the **File** menu the settings can be modified, loaded and saved.
   - In the **Model** menu a new *.xml* file can be loaded and saved.
2. Toolbar
   - With the button **Connect to Target** the *Communicator* can be connected respectively disconnected from the target.
   - **Create Code** generates the *X2C* C code files out of the *X2C* model. Changes in the *Model* tab like *Sample time* require new code creation.
   - **Create RTOS Code** was moved to *Settings*. See 11.4 for details.
   - In the **Download settings** the *Binary* and *MAP* files can be selected. These files are usually generated during the build process and are used by the *Communicator* for the following purposes:
     (a) The binary is required to download an application to the target system. This feature is not available in the free edition.
     (b) The MAP file is optional and is required only to view global variables in the X2C Scope. See 12 for details.
   - The **Download application to target** function is only available in the Professional edition of *X2C* . This function provides application download functionality without any use of external programing devices.
   - The **Scope** button starts an oscilloscope like environment for plotting signals and variables of the running target. For more information see Section 12.
3. Tabs
   - In the **Status** tab there are two main areas as seen in Figure 11. In *DSP state* the current status of the connected target is shown. The following states are possible:

- The **Bootloader** state indicates that no valid/working application is present on the target system. Bootloader and Application download are avilable in the Professional edition only.
- The **Program loaded** state indicates an application is present and running.
- In the **Idle** state only the communication between target and computer is active. All controller functions are inactive furthermore the *Outports* values are static.
- The **Init** state calls the initialization functions of all *X2C* blocks. In this state all signals an variables are reset to their initial values.
- **Run - Power off** means the application is running normally but the power supply of the power electronics (e.g. frequency converter) is off.
- In **Run - Power on** the system is fully active.

Info: In the *Blinky* demo application the last four states cannot be changed because they are only useful for engine control applications.
The *Log* area shows status updates and error messages and can be cleared with the *Clear* button.

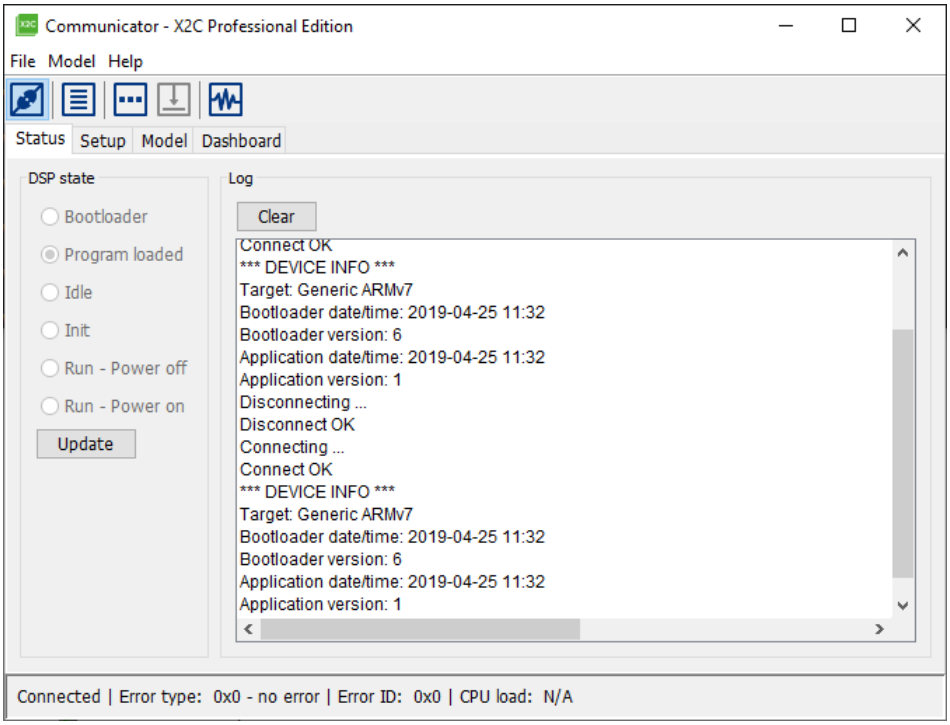


Figure 11: *Communicator* status

- The **Setup** tab is for the interface configuration. To change the settings the *Communicator* needs to be disconnected from the target.
  There are a few ways to connect with the target. One can choose between *Serial*, *USB* and *PCAN* interface. The setting made here need to be compatible with the target configurations.
  In the *Protocol* area the *LNet Node ID* can be set. By default this value is set to 1. Since the *Communicator* can be used with more than one target each target is defined with an unique *LNet Node ID*.

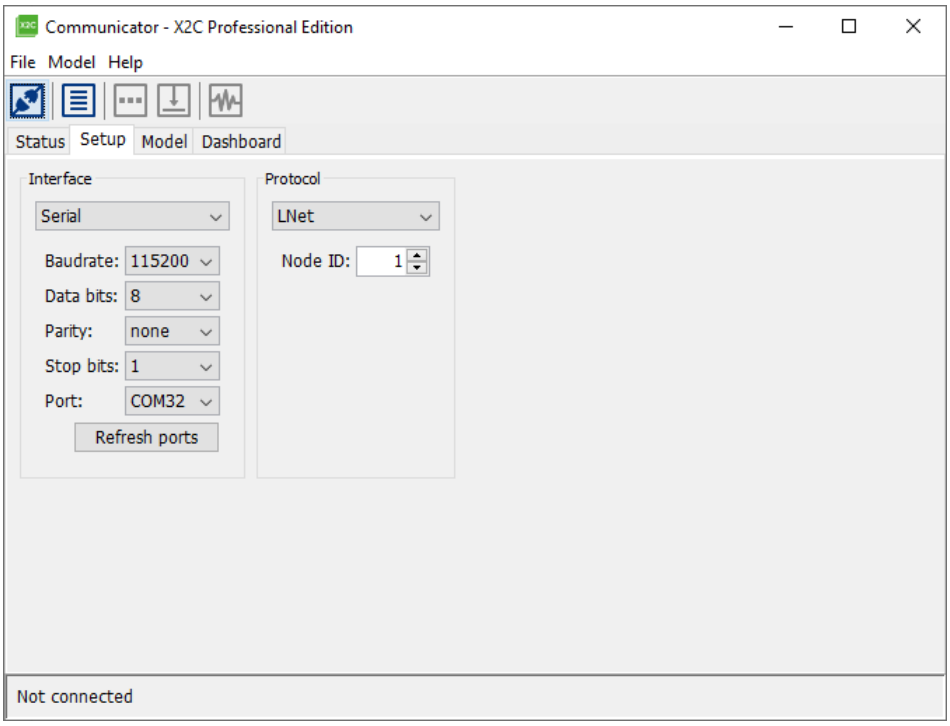


Figure 12: *Communicator* setup

- In the **Model** tab the *Model structure* area provides a list of the used blocks in the *Scilab* model. Jump to Section 11.5 to see how variables can be changed through the *Model structure* settings.
  The *Model properties* settings need to be made before code generation.
  - The *Sample time* can be changed in the *Xcos* model by changing the values in the *CLOCK* block. After double clicking on *transform model and push to Communicator* the sample time in the *Communicator* is updated. After clicking on *Analyze* the sample in the *Communicator* is updated.
    Note: Changes of the sample time made in the model need to be compatible with the defined sample time on the target.
  - *Use Scope* was moved to *Settings*. See 11.4 for details.
  - *Use Parameter ID for block data transfer* was moved to *Settings*. See 11.4 for details.

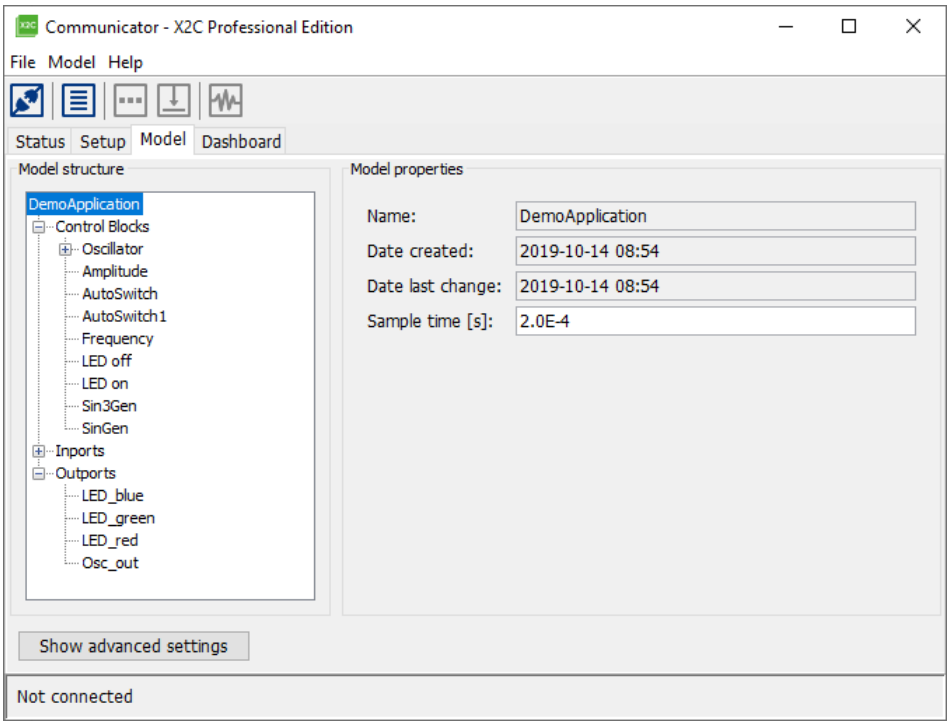


Figure 13: *Communicator Model*

11.4 Settings

The Settings can be found in the *File* menu. The Settings view is divided into 4 tabs.

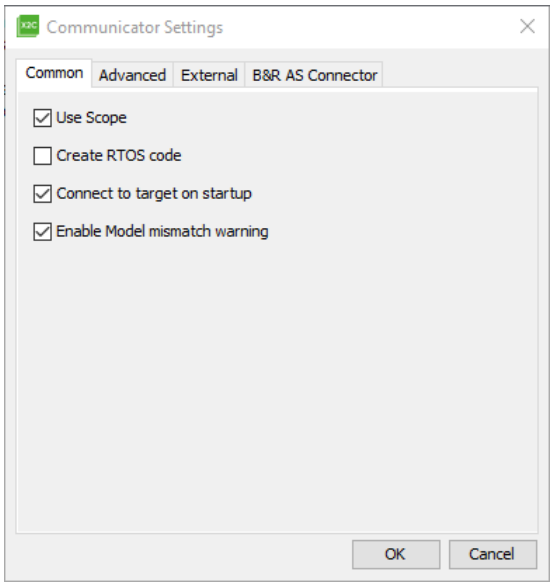


Figure 14: *Communicator Settings*

Common

- **Use Scope** [enabled]
  Use Scope defines if the scope application (see Section 12) can be used when connected with the target. When disabled the target processor is relieved due to less communication effort.

- **Create RTOS code** [disabled]
  Generates code which is optimized for real time operations.
- **Connect to target on startup** [enabled]
  Tries to connect to the target when the Communicator starts. If this option is enabled, the previously used communication setup is being used.
- **Enable Model mismatch warning** [enabled]
  If this option is enabled, a warning dialog is shown if the target Model ID differs from the Communicator Model ID. If the Model ID feature is not supported by the target of if no Model is present in the Communicator, this option is ignored.

Advanced

- **Create Signals code** [disabled]
  Creates a file containing lists with internal *X2C* signals. These signals are Inports, Outports and Block Outports.
- **Create & compile HotInt code** [disabled]
  Generates HotInt specific files. After successful generation, the HotInt project (Microsoft®Visual Studio) is being compiled. The latest version being found is used for compilation. Supported Visual Studio versions:
  - Visual Studio 2013
  - Visual Studio 2012
- **Enable Code For Speed Optimizer** [disabled]
  Allows the usage of the Code For Speed Optimizer. Refer to 14 for details.

External

Allows to select executables which are executed

- before code generation (Pre-Generate)
- after code geneeration (Post-Generate)

B&R AS Connector

This tab allows to configure the *X2C* Connector for *B&R Automation Studio* . Refer to 22 for details.

11.5 Change parameters on the target with the *Communicator*

If all connection are set up properly there are two ways of changing the parameter values on the running target.

1. In the *Communicator* click on **Model**. In the Model structure area all the Block properties are listed and can be changed by double clicking on them.
2. In the *Xcos* model double click on the blocks and change the values.

12 Scope

The *Scope* application is a very useful device for monitoring signals and variables on the running target. It allows an easy observation in an oscilloscope like environment. The *Scope* is structured into four main areas as seen in Figure 15.

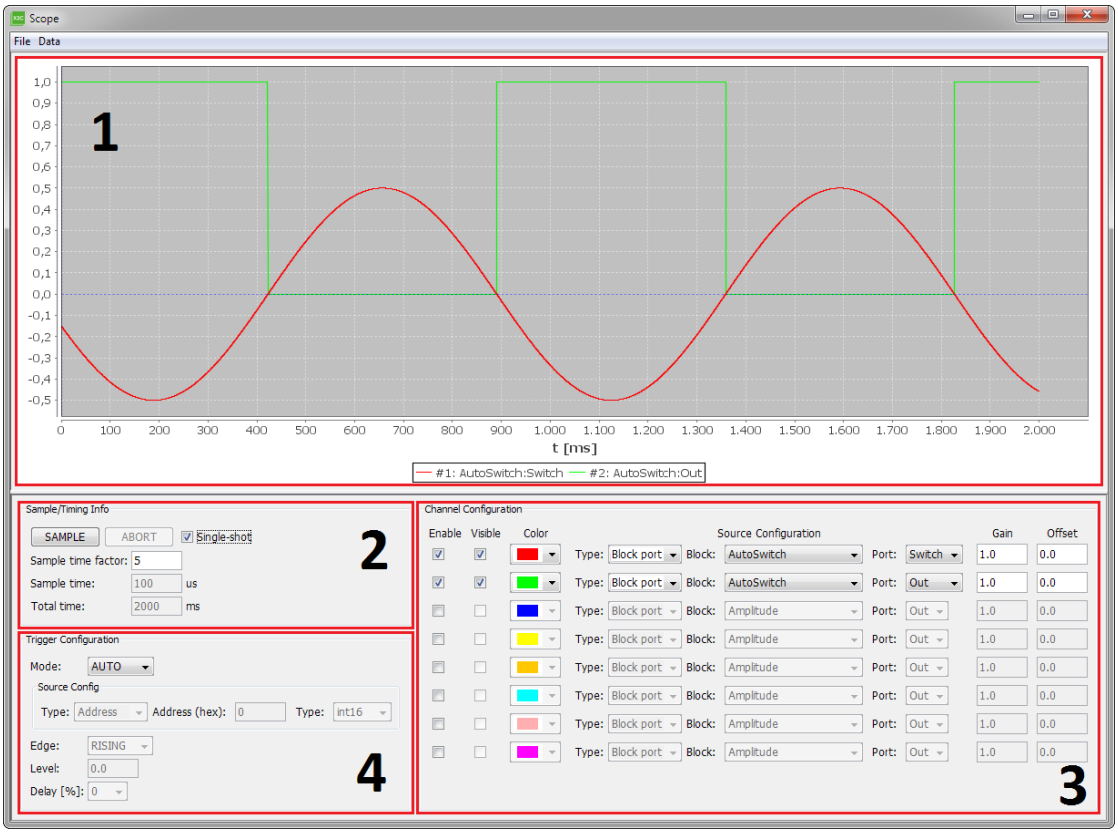


Figure 15: Communicator Scope

1. The **Plot area** shows the selected values at a time based abscissa (x-coordinate) in milliseconds and scaled from -1.0 to 1.0 at the ordinate (y-coordinate). Furthermore there is a legend a the bottom of the plot area showing the color of each channel.

2. In the **Sample/Timing Info** section options of the time axis can be made. The oscilloscope is started with the button *SAMPLE*. When the option *Single-shot* is marked only one time period (*see Total time*) is shown in the plot area. When unmarked the plot area continuously plots the received values from the target. Due to time delay in data transfer it is possible that there a missing values between two plot cycles.
   The plot process can be stopped with the button *ABORT*.
   With the option *Sample time factor* the time axis can be scaled. Factor 1 means every value (Time between two values is *Sample Time*) is plotted in the plot area. As example factor 5 means every fifth value is used, therefore a longer time span can be plotted at the time axis.

3. The **Channel Configuration** configures which signal is shown at the plot area. There are eight channels that can be plotted simultaneously. Mark *Enable* to configure one channel. In the *Type* menu *Address, Block Port* and *I/O port* can be chosen. *I/O ports* are the links between the target peripheries and the *X2C* model. The *Block Port* are

signals used in the *X2C* model.
When fixed point data representation is selected all signal are scaled to values between -1.0 and 1.0, therefore one might use the option *Gain* or *Offset* to plot the signal in real scale.

4. The **Trigger Configuration** is divided in the options *NORMAL* and *AUTO*. When option *AUTO* is chosen no specific trigger is set. In this configuration signal values are continuously transfer and plotted. This can lead to moving graphs especially when periodic signals are observed.

   Choose *NORMAL* to set up a trigger.

   (a) In *Source Config* choose a signal which should work as trigger source.

   (b) With *Edge* the trigger only checks rising respectively falling edges of the source signal.

   (c) The *Level* and *Delay* options move the trigger point in vertical and time direction.

   Example: Trigger the harmonic sine wave *u* from a *SinGen* block.
   As trigger source the signal itself is used. When the trigger is delayed in time a vertical marker indicates the position. The effect of the settings *Level* with a value of 0.2 and *Edge* for *FALLING* can be seen in Figure 16.

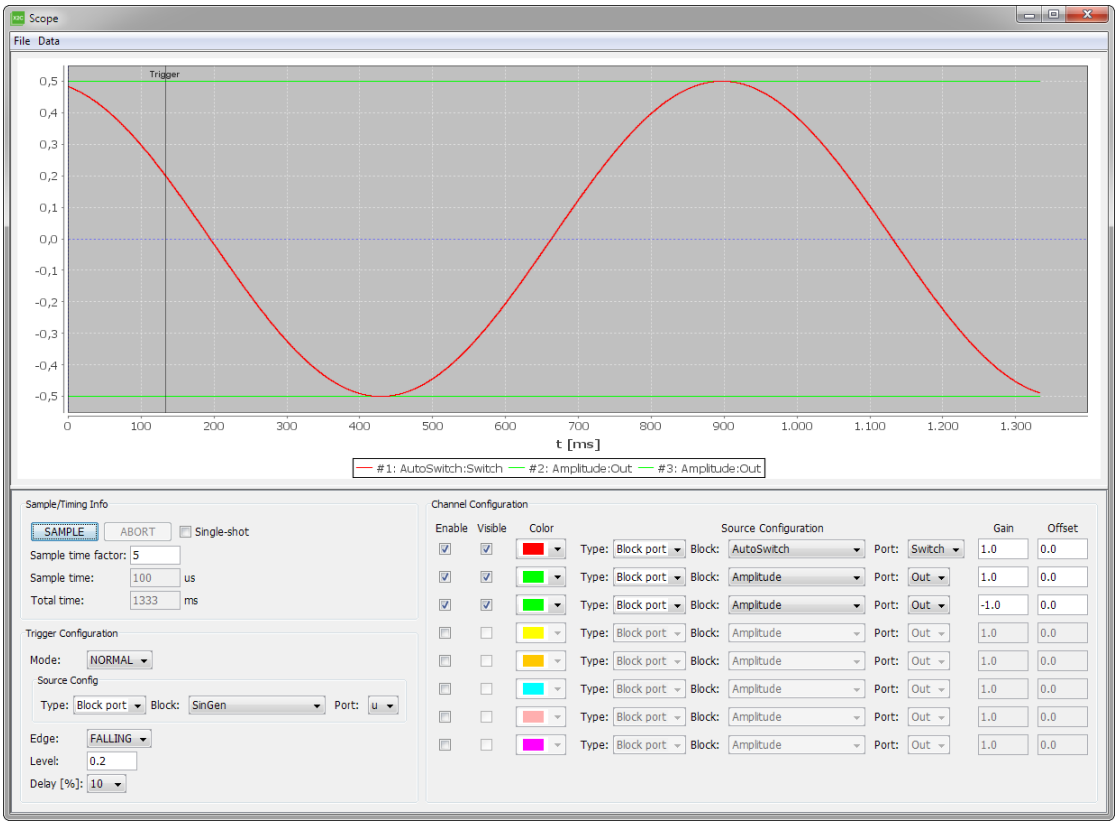


Figure 16: Scope Trigger using Example

13 Block Generator

The Block Generator is used to create new blocks, load and/or edit previously saved blocks. The following, essential parameters define the block function:

13.1 Block properties

Name

Each block within a library must have a unique name.

Library type

The library type selection is done via the 'Change configuration' button.
An internal library block will be stored within the *X2C* structure where only the library name is required. The internal library name can be selected via a dropdown menu.
When the block is saved, the files will be automatically saved into the correct directories.
An external (or project specific) block is stored within its project structure and requires the user to enter a library name and pre-namespace identifier. Both, the library name & pre-namespace, is enterd via text fields.
When the block is saved, a window will appear, which allows to select the project directory for this project specific block (only directory selection is possible) The Block Generator tries to save the files in the following structure (directories will be created automatically if they don't exist):
<selected directory>\Library\<library name>\

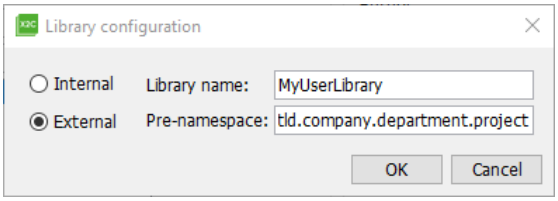


Figure 17: *Block Generator* External library settings

Identifier

Every block needs an unique identifier (ID) across all libraries to ensure proper functionality if a project uses blocks from different libraries. *ID* must not exceed 4095 and should be a value < 4000 for internal blocks and a value $\geq$ 4000 for external blocks.

Additional LaTeX information file

In case of having a LaTeX file with additional block information the name of the file can be set.

Mask In- & Outports

Every block can have several inports & outports. Each in-/outport must have an unique name.

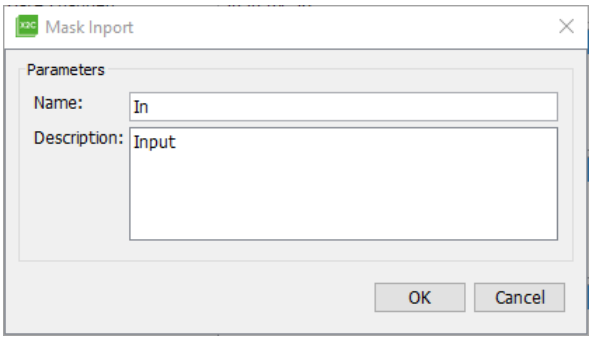


Figure 18: *Block Generator* Mask Inport

Mask Parameters

Every block can have several mask parameters. Each mask parameter must have an unique name. *Prompt* will be displayed next to the value input of this parameter. *Data type* decides between an input field for type *Double* or a dropdown menu for type *ComboBox*.
In case of type *Double* you can select the *Default value* for this parameter.
Type *ComboBox* lets you add/remove items which can be selected by the user if this parameter value should be changed. The default value is defined by selecting one out of the entries.
Visible makes this parameter visible, *Changeable* en- or disables this parameter.

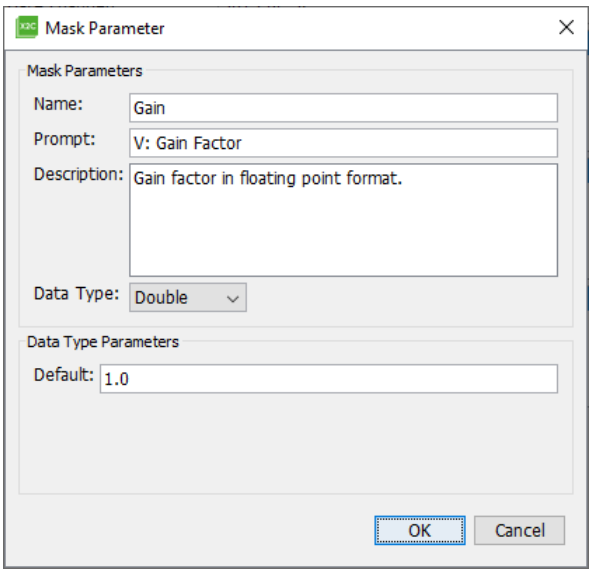


Figure 19: *Block Generator* Mask Parameter

Visualizations

Visualizations are used to represent the block within a model. *Command* contains the language specific commands to represent the block.

13.2 Implementation properties

Every block must have at least 1 Implementation. Each Implementation has its Init-, Update-, Save- and Load functions (C) and Conversion functions (Java/Python/JavaScript).

Name

Each implementation must have a unique name within a block. The Implementation name is used for C- & Java code file name.

Controller In- & Outports

The Controller In- & Outport names can be selected but not edited. The names are defined by the block's Mask In- & Outports. Only the data type must be selected for each In-/Outport.

Controller Parameter

Each Controller Parameter must have an unique name within its Implementation. The data type and default value can be selected. Also the ability to download/upload the parameter can be defined by using *Load & Save Enable* checkbox.
The *Array* option allows the Parameter to contain a data array instead of a scalar value. Two array configurations are possible:

- Modifyable
  Enabled by setting the *Array* checkbox and *Array size*. This type of array has a fixed size and is located in volatile memory. Due to volatile nature the data can be up- and downloaded.

- Fixed
  Enabled by setting both, the *Array* and *Array in flash memory* checkboxes. The size of this type of array is not fixed because it is generated on Model generation. It is located in non-volatile memory and therefore cannot be up- and downloaded.

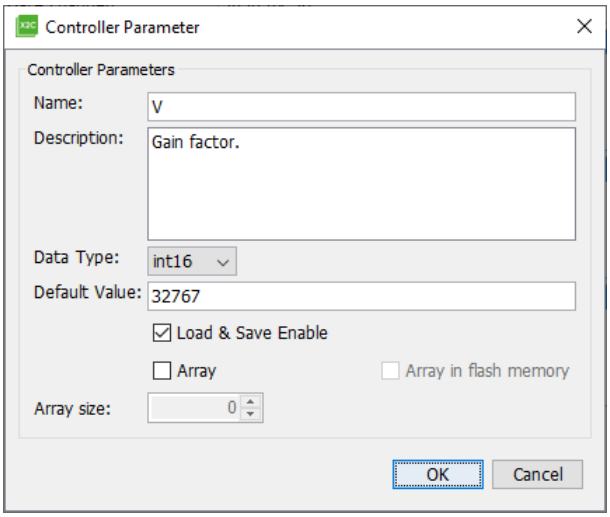


Figure 20: *Block Generator* Controller Parameter

Conversion function type

The Conversion function type can be selected by using the dropdown menu. If *Java*, *Python* or *JavaScript* is selected, the Block Generator will generate a conversion function template file for this block which needs to be manually filled with block-specific conversion calculations, otherwise no conversion function file will be created.

13.2.1 Advanced Implementation Settings

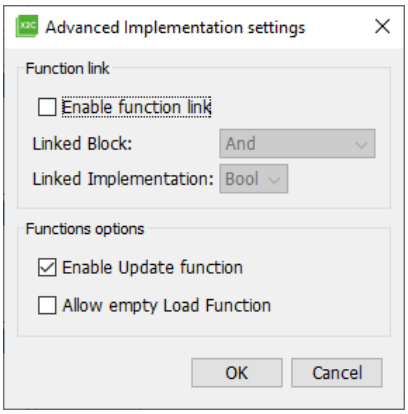


Figure 21: *Block Generator* Advanced Implementation Settings

Function Link

If this option is enabled, the *Init-*, *Update-*, *Load-* and *Save* Block functions are used from an other Block Implementation. A Block within the same *X2C* library may be selected as link source.

Update Function

This option is enabled by default. If disabled, the Implementation does not have a (cyclic) Update function.

Allow empty Load function

Forces the Block Generator to create a *Load* function even if the Implementation doesn't contain any transmittable parameters. The user customizable part of the *Load* function may be used to change the *Load* functions' default behaviour.

13.3 Save or load a block

Saving & loading is done via the *File* menu in the menu bar.

Saving

If the selected block is member of an internal library, the Block Generator automatically uses the correct library root directory.
In case of an external library block type, the user is prompted a directory selection window, in which the project directory can be selected. The library root directory is now located in: <user directory selection>\<Library>\<library name>.

Each library is organized in this structure:

- Controller: Directory with the C-code source files (*.c, *.h).
- Conversion: Directory with the Java, Python or JavaScript conversion files.
- Doc: Directory with files needed for the (auto-generated) documentation.
- Scilab: This directory contains the *Xcos* library files as well as the interfaces functions and the files need for simulation in *Xcos* .
- XML: Configuration files (*.xml) contain all block parameters and are located in this directory.

Part IV
Add-Ons

14 Code For Speed Optimizer (CFSO)

The CFSO allows the optimization of the *X2C* Update function(s) to decrease the execution time. TO achieve this, the CFSO, if activated, replaces the *X2C* Block function calls by the *X2C* Block Update code itself to remove the overhead of C function calls. The optimizer collects the required information from the Block-Implementation C source code.
It can be enabled in the Advanced *Communicator* settings tab.

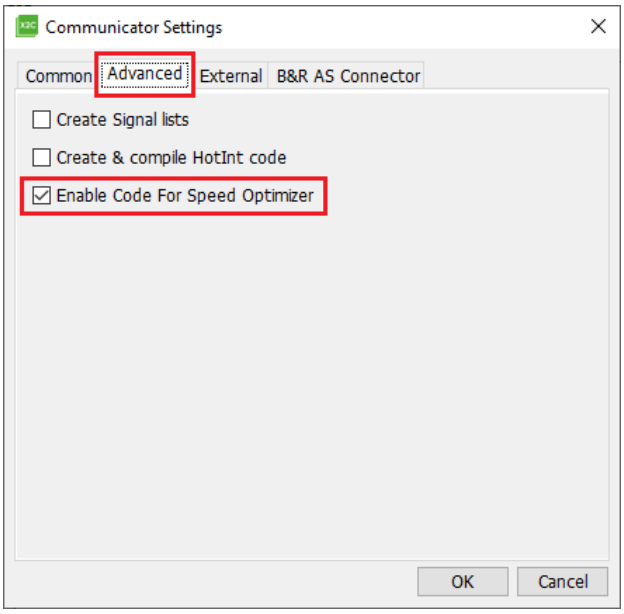


Figure 22: CFSO in Advanced *Communicator* settings

This feature is only available in editions **Plus** and **Professional**.

14.1 Data Collection

The CFSO collects the data from the C source for each Block-Implementation type. If the source is not available, the Implementation is tagged to use the usual Update function call instead and the next Implementation type is processed. Otherwise the following Implementation specific elements are extraced from the C source:

- Preprocessor code
- Includes
- Define names
- Update function code

During data extraction, every occurence of the Implementation Pointer variable is replaced by a placeholder. The data collection procedure is repeated for every Implementation type in the Model. In the further course the *X2C* Code Generator will replace the placeholder by the current Block variable.

14.1.1 Implementations using a function link

This kind of Implementation shares the Update function code with an other Implementation. In this case the Preprocessor code, Includes and Define names from both Implementations are merged. The Update function code is retrieved from the linked Implementation.

14.2 Code Generation

Once all the Implementation information is collected, the *X2C* Code Generator replaces every Update function call, except for the Implementations of which the sources are not available. Following elements are added at the beginning of the generated *X2C* C file:

- An informational line beyond the header to inform the user that the CFSO is active
- All collected Implementation C Includes

For each Update function the follwoing content is added in the following order:

1. Preprocessor code
2. Implementation Update function
3. Undefine C macros (#undef) for all possible defined C macros

The placeholders, which were set during the data collection procedure, are replaced by the appropriate Block Implementation variable. In case of an Implementation with unavailable C source, the usual Update function call is used. A comment is added above the relevant Update function call for information.

15 Dashboard

The Dashboard allows the user to view and modify selected elements. Each element features various, configurable properties for a user friendly representation of each data value. The elements can be uploaded or downloaded - depending on the respective type.
A download can be done for each indivdual, editable entry by changing its value or for all editable values by using the 'Download Editable Values' button.
The upload procedure is started by the 'Start Update' button and uploads ALL uploadable values at a defined interval.

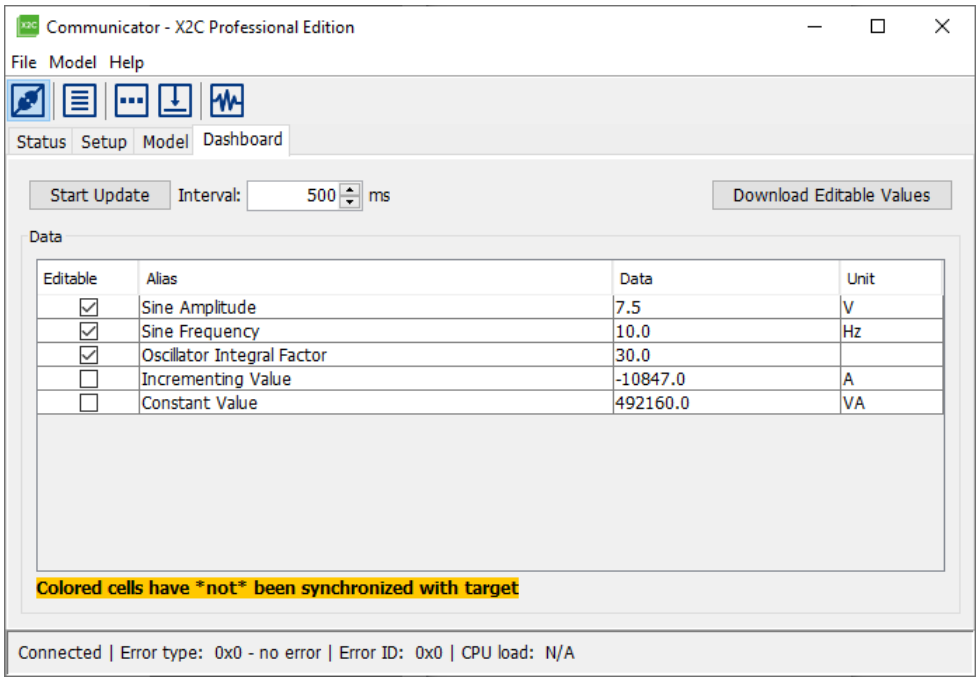


Figure 23: Communicator Dashboard View

This feature is only available in editions **Plus** and **Professional**.

15.1 Configuration File

The Dashboard configuration is stored in a properties file. Currently there is no GUI support to setup a Dashboard configuration. Instead, the user has to setup the file.
Configuration file templates are available to make the configuration process easier. These templates can be found in directory `/Templates/Dashboard`.

A Dashboard configuration can be loaded or cleared via the Communicator menus. Two options are available to either load a new configuration or clear the current configuration.

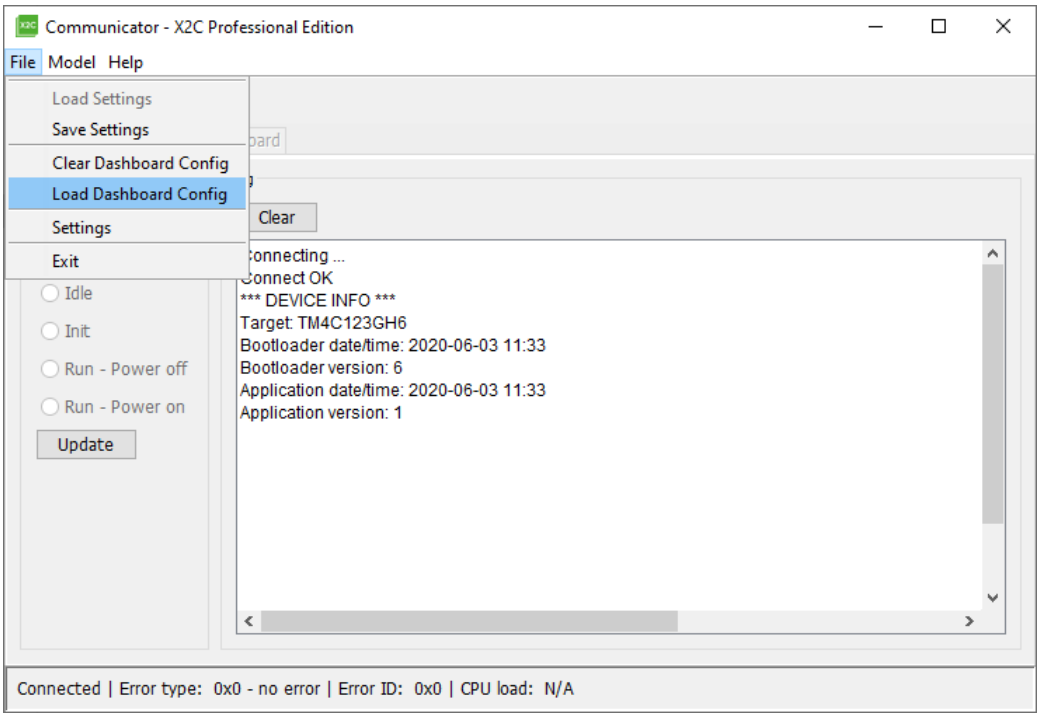


Figure 24: Communicator Dashboard configuration

These elements are usually contained in a configuration file:

Property

A property is defined by a key and value. The syntax is
`KEY=VALUE`

Example:
`MODEL=MyModel.xml`

Comment

The hash symbol # introduces a comment line. Every line starting with a hash symbol is ignored by the configuration file parser. Example:
`# This line is a comment`

15.2 Common properties

Each Dashboard element does have common properties independent of its type.

Alias name

If this property is set, the alias name will be displayed instead of the element name. Otherwise the element name is shown.

Property name: `ALIAS`

Example:
`ALIAS=Show this text instead`

Offset & Gain

Applies Offset and Gain factors. This allows, for example, the transformation between physical unit and raw value. If one or both values are not set, the appropriate default value is used. The default for GAIN is 1.0, the default for OFFSET is 0.0.

Property name: `OFFSET, GAIN`

Example:
```
OFFSET=0.5
GAIN=2.0
```

ATTENTION:
The Offset and Gain calculation depends on the transfer direction (UPLOAD or DOWNLOAD).

DOWNLOAD (data is transferred from Dashboard to the target):
$TargetValue = DashboardValue * GAIN + OFFSET$

UPLOAD (data is transferred from the target to the Dashboard):
$DashboardValue = (TargetValue - OFFSET)/GAIN$

Unit

Defines a physical unit. If this property is not set, no unit will be used.

Property name: `UNIT`

Example:
```
UNIT=MyUnitName
```

15.3 Element types

15.3.1 Block-Parameter

Allows to define an *X2C* Block and one of its Mask Parameters to be downloaded to the target. Following additional properties have to be set to identify the selected *X2C* Block:

- BLOCK
  X2C Block name
- PARAMETER
  Mask Parameter name of *X2C* Block
- MIN
  Defines the lower limit for the value
  If this value is not defined or empty, no lower limit is active.
- MAX
  Defines the upper limit for the value
  If this value is not defined or empty, no upper limit is active.
- VALUE
  Sets the initial Dashboard value
  If this value is not defined, the current Mask Parameter from the Block is used

This element type can currently be used for **download only**.

A complete example for a Block-Parameter element. This first 5 properties are Block-Parameter-type specific, the rest are common properties.

```
BLOCK=MyConstant
PARAMETER=Value
VALUE=5.0
MIN=1.0
MAX=12.0
ALIAS=Voltage
UNIT=V
GAIN=300.0
OFFSET=10.0
```

15.3.2 Variables

Allows to define a global symbol (variable) to be uploaded from the target and displayed in the Dashboard. Following additional properties have to be set:

- VARIABLE
  Variable name
- TYPE
  Data type of the variable

The data type can be one out of the following values:
int8, uint8, int16, uint16, int32, uint32, int64, uint64
The Boolean type is currently not supported. This element type can currently be used for **upload only**.

ATTENTION:
This feature requires a MAP file to be loaded in the Communicator because the target memory addresses for each global variable is extracted from the MAP file generated by the vendors' toolchain. If no MAP file is loaded, the variables will be displayed in the Dashboard View but the 'Start Update' button will remain inactive. If one or more invalid variable names are encountered in the Dashboard configuration file, an error message is shown in the Communicator log for every invalid variable name. Trying to use the 'Start Update' button will result in an error message showing the first invalid variable name.

A complete example for a Variable element. The first 2 properties are Variable-type specific, the rest are common properties.

```
VARIABLE=MyGlobalVar
TYPE=int16
ALIAS=Current
UNIT=A
GAIN=100.0
OFFSET=0.0
```

15.4 Model Override

This property allows the Communicator to replace the currently loaded *X2C* Model or load a new Model at all. The property is optional and can be omitted if not used but must be defined at the beginning of the Dashboard configuration file if used. The value can be a filename or a fully qualified file name containing the complete path.

Property name: `MODEL`
Example:
`MODEL=MyModelFile.xml`
`MODEL=C:\MyProject\Dashboard.txt`

The Communicator log window displays a message if loading the Model from the configuration file was successful.

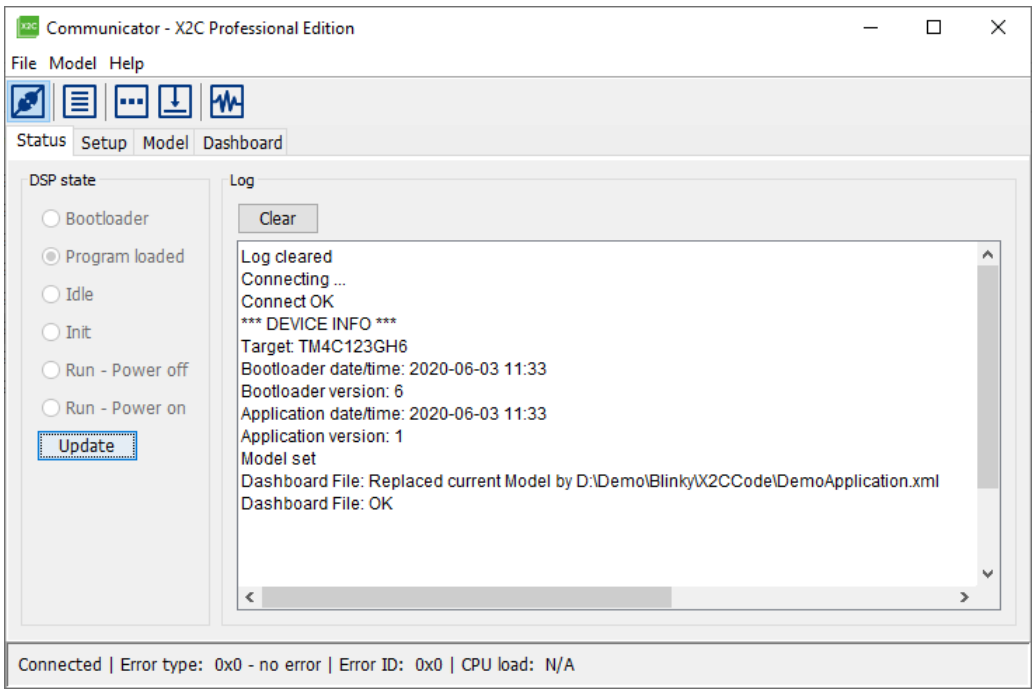


Figure 25: Communicator Dashboard Model log entry

Part V
How-To

16 *X2C®* code generation with *Scilab*

The following section describes *X2C* code generation of a *Xcos* model based on the *Blinky* demo application.

1. Open *Scilab* and in the file browser navigate to your project directory (e.g. <X2C_ROOT>\DemoApplication\Blinky_TI_TMS320F28069_controlSTICK\X2CCode).

2. Double click on **DemoApplication.zcos**. The example project contains a few blocks used to demonstrate the basic function of *X2C* (see Figure 26). The *Inport* and *Outport* blocks define the interface between the generated *X2C* code and the peripheral functions (e.g. ADC or GPIO Pins) on the target. For details about each block function read *X2Copen.Doc.pdf* in the documentation folder of the *X2C* directory.

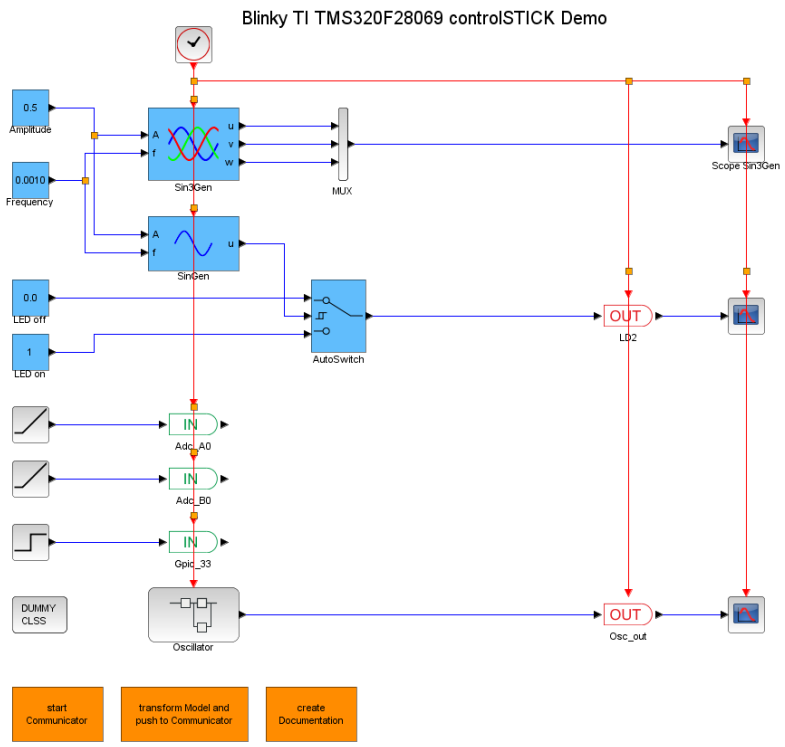


Figure 26: *Blinky* demo application in *Scilab*

3. Double click on **start Communicator** (for more information about the *Communicator* see Section 11). Some details of the current actions of the *Communicator* are shown in the *Log* area of the *Communicator* window and the *Scilab* command line:

```
Starting Communicator
done
Successfully connected to Communicator
```

4. Double click on **Transform model and push to Communicator** and check the pop-up window for the end of the transformation process.

5. Click **Create Code** in the *Communicator*. Now the files *X2C.h* and *X2C.c* are generated in the <PROJECT_ROOT>\X2CCode directory and the Log screen should contain the lines:

```
[...]
Model updated
Model XML file write: OK
Create code successful.
```

6. The *C* code for the *X2C* application has been created. Depending on the used target start the programming tool (e.g. *Code Composer Studio* , *STM32CubeIDE* or *MPLAB X*) and import the *Blinky* demo application project as described in Section 17, or 18 respectively. Follow the instructions on how to configure and download the application to the target.

17 Loading and building the demo application Blinky in *Code Composer Studio*

The demo application *Blinky* is intended to be used with a *TI F28069 Piccolo controlSTICK*.

1. Connect the *TI F28069 Piccolo controlSTICK* to the computer.

2. Open *Code Composer Studio* (choose workspace directory as you like). Now click **Project** → **Import Existing CCS Eclipse Project**. Browse to the location of the *Blinky* project (<X2C_ROOT>\DemoApplication\Blinky_TI_TMS320F28069_controlSTICK). Click **Finish** to import the project.

3. In the *Code Composer Studio* file structure of the *Blinky* demo project there are two virtual folders *Blocks* and *Core*, which should be linked directly to the *X2C* directory. To ensure this go to **Project** → **Properties** drop down **Resource** and click **Linked Resources**. Double click on folder **X2C_ROOT** and set the correct link to your X2C installation directory (<X2C_ROOT>). After hitting **OK** two times there should not be any warning signs (like shown in Figure 27) at the icons for the linked files in the Blocks and Core folders.

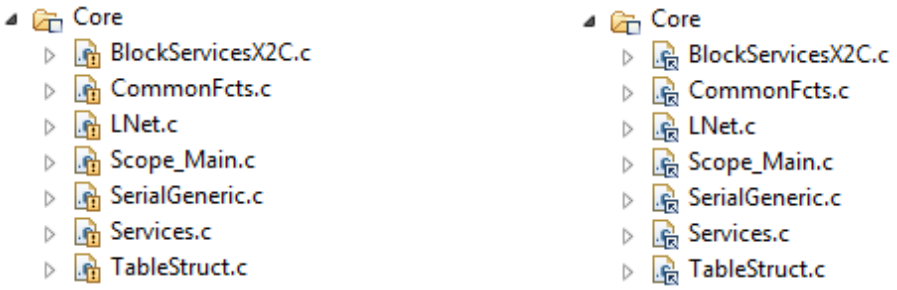


Figure 27: *Code Composer Studio* invalid (left) and valid (right) *X2C* root directory

4. The generated code from *X2C* is located in the folder <X2C_ROOT>\DemoApplication\Blinky_TI_TMS320F28069_controlSTICK\X2CCode. To check if code generation went fine go to the X2CCode folder and open *X2C.c*. Make sure time and date of code generation is plausible.

5. Build the project in *Code Composer Studio* by clicking **Project** → **Build all** or by clicking on the **Hammer** symbol as seen in Figure 28 at the top of the screen. Check for errors while building in the console at the bottom of the screen.

Figure 28: *Code Composer Studio* build and debug buttons

6. If your target is connected to the computer click **Run** → **Debug** or click on the *Bug* symbol as seen in Figure 28 at the top. The program is now transferred to the target and can be started with the **green arrow** button at the top.

7. After starting the program the on-board LED of the *TI F28069 Piccolo controlSTICK* should be blinking!

18 Loading and building the demo application Blinky in *MPLAB® X*

The demo application *Blinky* is build for the combination of the *Microstick II* with the *dsPIC33FJ128MC802* processor and the *MicrostickPlus* developer board (for details see www.microstick.com).
Info: To download a new application only the *Microstick II* needs to be connected with the computer.

1. Connect the *Microstick II* with the computer.

2. Open *MPLAB X* and click **File** → **Open Project**. Browse to the location of the *Blinky* demo application in the *X2C* directory <X2C_ROOT>\DemoApplication\. . . \Blinky_Microchip_dsPIC33Fxxxx_MicrostickPlus. Click **Open Project**.

3. In the case the demo application is copied/moved to a different location, the include paths have to be adapted. To ensure the compiler uses the correct path variables right click on the **Projectname** → **Properties** → **XC16 Global Options** → **xc16-gcc**. In the drop down menu **Option categories** choose **Preprocessing and messages**. Click on the dots beside *C include dirs*. There are relative paths to the needed include files listed as seen in Figure 29. Correct the links by double clicking on the path variables.
Info: Only the links to the *Library* and *Controller* path need to be updated.

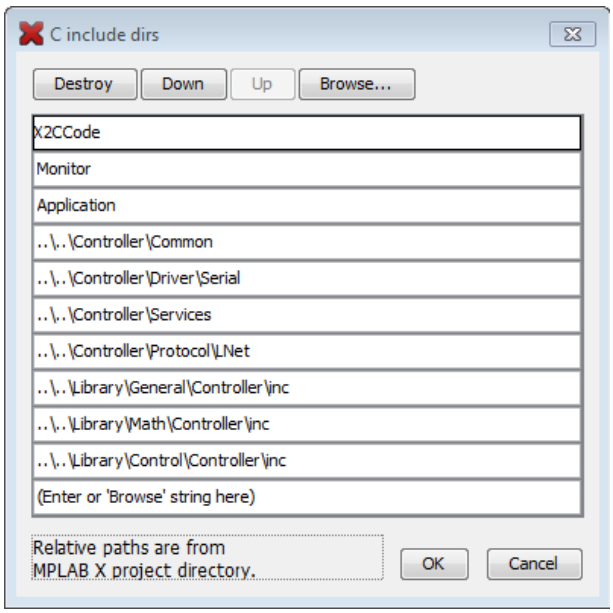


Figure 29: Default path variables for the include files

4. Go to **Run** → **Clean and Build Main Project** or click the *hammer with brush* button as seen in Figure 30. After building there should be a message `BUILD SUCCESSFUL` in the message area at the bottom of the screen.

Figure 30: *MPLAB X Clean and Build Main Project* button

5. If the build process was successful go to **Run** → **Run Main Project** or click the *Green*

Arrow button as seen in Figure 30. If there is a message similar to *MICROSTICK not Found* try to select the *Starter Kits (PKOB)* item which represents your board.

6. After starting the program the LED (RB12) on the *MicrostickPlus Board* should be blinking!

19 Loading and building the demo application Blinky in *STM32CubeIDE*

The demo application *Blinky* is intended to be used with the *ST STM32F072RB Nucleo* or the *ST STM32G474RE Nucleo* kit.

1. Connect the ST development kit with the computer. You may have to install the ST-Link USB driver (available on www.stm.com) to get the board recognized by your operating system.

2. Open *STM32CubeIDE* and click **File** → **Import...**. Select **General/Existing Projects into Workspace** and clickt **Next**. Select the root directory of the *Blinky* project (either <X2C_ROOT>\DemoApplication\Blinky_ST_STM32F072RB_Nucleo or <X2C_ROOT>\DemoApplication\Blinky_ST_STM32G474RE_Nucleo). Click **Finish** to import the project.

3. In the *STM32CubeIDE* file structure of the *Blinky* demo project are two virtual folders *X2C-Blocks* and *X2C-Core*, which are linked relatively to the *X2C* directory. If the *Blinky* demo project is copied/moved to a different location, the resource property *X2C_ROOT* as seen in Figure 31 and the build variable *X2CDirPath* as seen in Figure 32 have to be adapted.

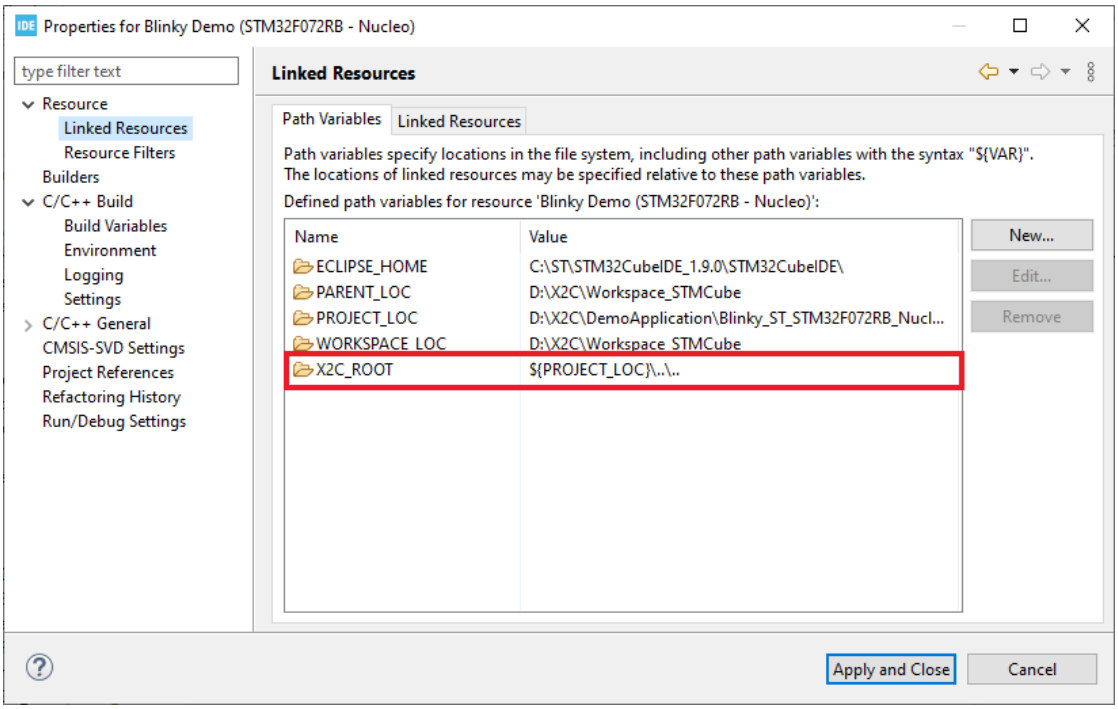


Figure 31: *STM32CubeIDE* resources settings

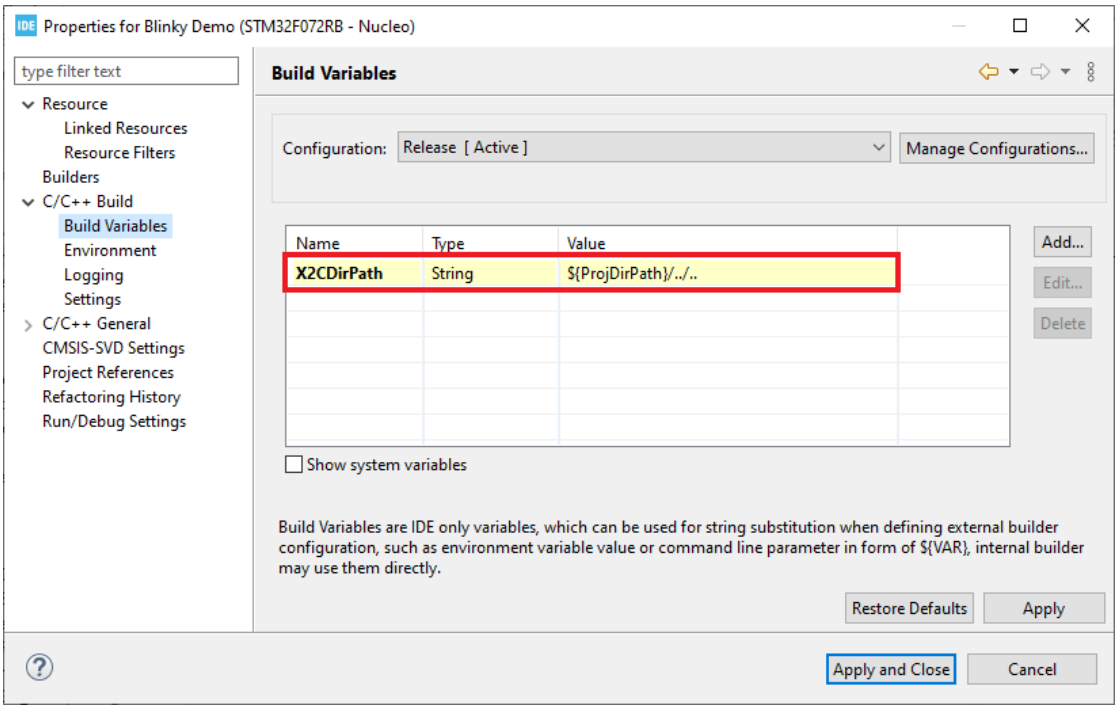


Figure 32: *STM32CubeIDE* build variables setting

To open shown windows go to **Project** → **Properties**.

4. The generated code from *X2C* is located in the X2CCode folder (eg. <X2C_ROOT>\DemoApplication\Blinky_ST_STM32F072RB_Nucleo\X2CCode). To check if code generation went fine go to the X2CCode folder and open *X2C.c*. Make sure time and date of code generation are plausible.

5. Before building the project the first time some hardware specific code has to be generated. Open the STM32CubeMX configuration window with a double-click on the **.ioc* file in STM32CubIDE's project explorer. Then the code can be generated with **Project** → **Generate Code** or by clicking on the *Code Generation* icon as seen in Figure 33.

Figure 33: *STM32CubeIDE* menu icons - Code Generation

6. Build the project in *STM32CubeIDE* by clicking **Project** → **Build Project** or by clicking on the *Build* icon as seen in Figure 34. Check for errors while building in the console at the bottom of the screen.

Figure 34: *STM32CubeIDE* menu icons - Build

7. If your target is connected to the computer click **Run** → **Run** or click on the *Run* icon as seen in Figure 35. The program is now transferred to the target and is automatically started.

Figure 35: *STM32CubeIDE* menu icons - Run

8. After starting the program the green on-board LED of the ST development kit should be blinking!

20 Loading and building the demo application Blinky in *DAVE IDE*

The demo application *Blinky* is intended to be used with the *Infineon XMC4700 Relax Kit*.

1. Connect the Infineon development kit with the computer. Use a Micro-USB cable to connect the USB port of the computer with the USB port labeled X101 on the kit.

2. Open *DAVE IDE* and click **File** → **Import...**. Select **General/Existing Projects into Workspace** and clickt **Next**. Select the root directory of the *Blinky* project (<X2C_ROOT>\DemoApplication\Blinky_Infineon_XMC4700_RelaxKit). Click **Finish** to import the project.

3. In the *DAVE IDE* file structure of the *Blinky* demo project are two virtual folders *X2C-Blocks* and *X2C-Core*, which are linked relatively to the *X2C* directory. If the *Blinky* demo project is copied/moved to a different location, the resource property *X2C_ROOT* as seen in Figure 36 and the build variable *X2CDirPath* as seen in Figure 37 have to be adapted.

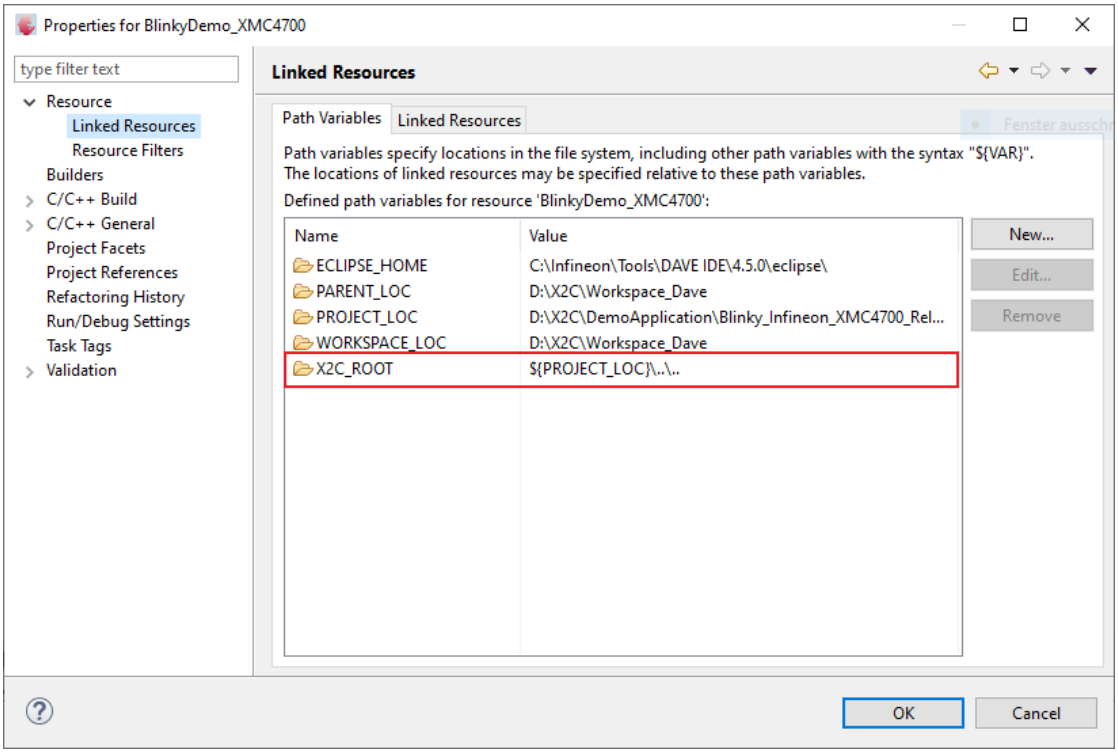


Figure 36: *DAVE IDE* resources settings

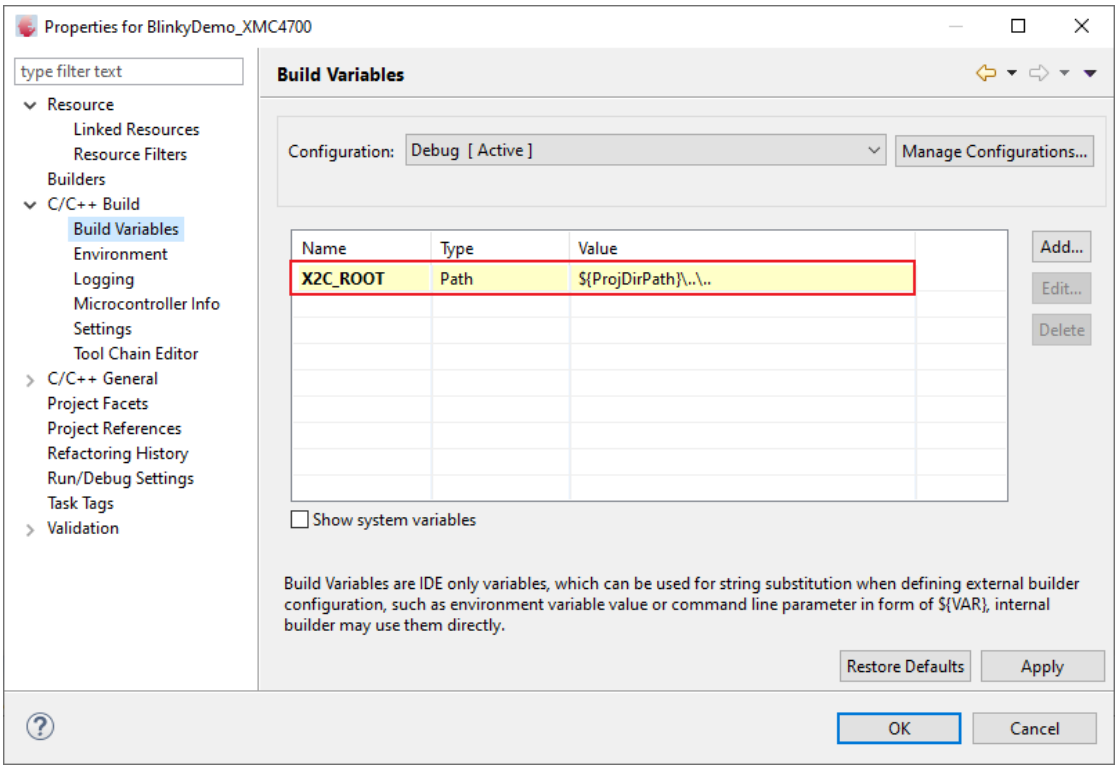


Figure 37: *DAVE IDE* build variables setting

To open shown windows go to **Project** → **Properties**.

4. The generated code from *X2C* is located in the X2CCode folder (eg. <X2C_ROOT>\DemoApplication\Blinky_Infineon_XMC4700_RelaxKit). To check if code generation went fine go to the X2CCode folder and open *X2C.c*. Make sure time and date of code generation are plausible.

5. Before building the project the first time some hardware specific code has to be generated. Right-click on the project name in the project tree to open the context menu and select **Set Active Project**. This activates the menu icons which are needed for code generation and project building.

   The hardware specific code can be generated with **DAVE** → **Generate Code** or by clicking on the *Generate Code* icon as seen in Figure 38.

Figure 38: *DAVE IDE* menu icons - Code Generation

6. Build the project in *DAVE IDE* by clicking **Project** → **Build Active Project** or by clicking on the *Build Active Project* icon as seen in Figure 39. Check for errors while building in the console at the bottom of the screen.

Figure 39: *DAVE IDE* menu icons - Build

7. If your target is connected to the computer click on the *Debug* icon as seen in Figure 40.

Figure 40: *DAVE IDE* menu icons - Debug

When the debug session is started for the first time a *Debug Configurations* window will open. Double-click on the *GDB SEGGER J-Link Debugging* entry on the left hand side. This will create a debug configuration as depicted in Figure 41.

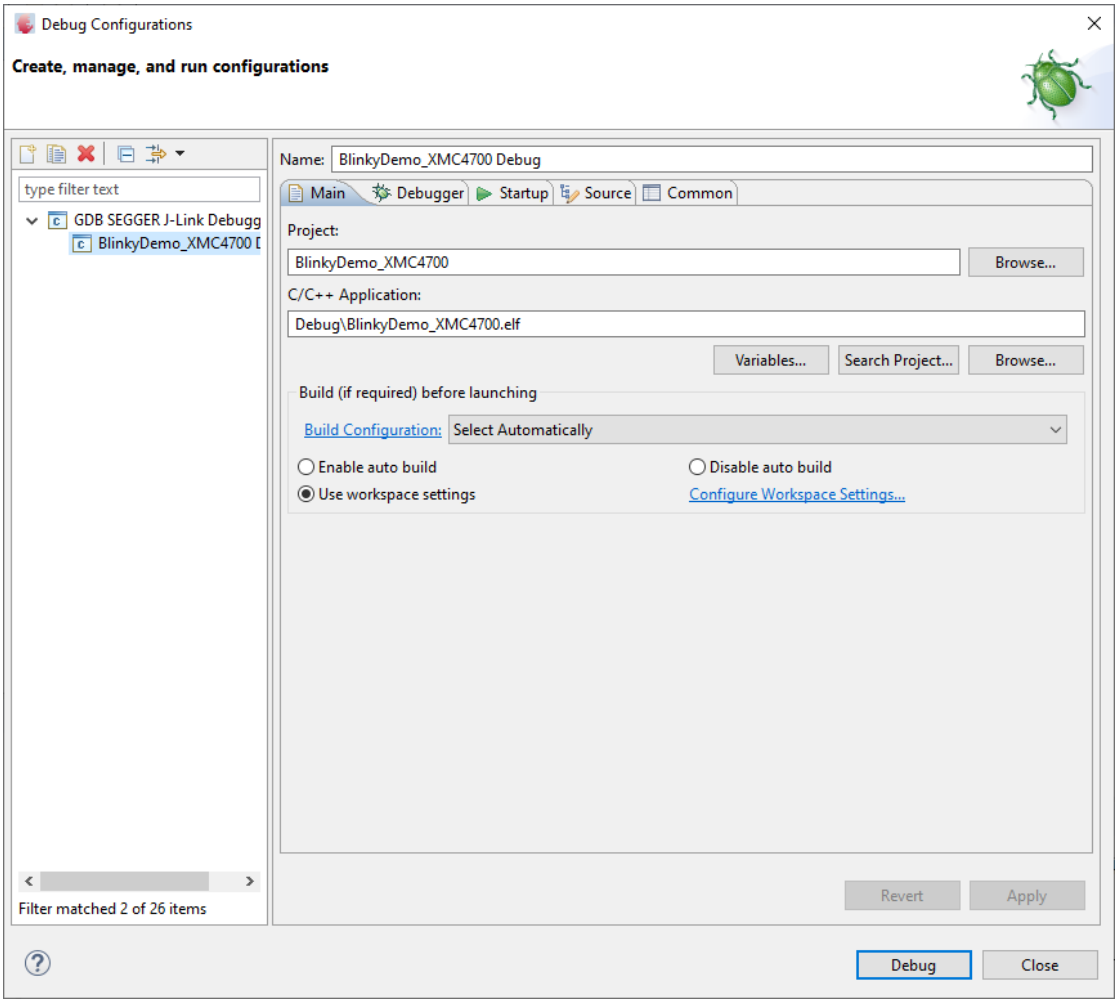


Figure 41: *DAVE IDE* Debug configuration

A click on **Debug** will start the debug session. This means the application will be transferred to the target and *DAVE IDE* will switch to the debug perspective.

8. To start the execution of the application press **F8** or click on the *Resume* icon as seen in Figure 42.

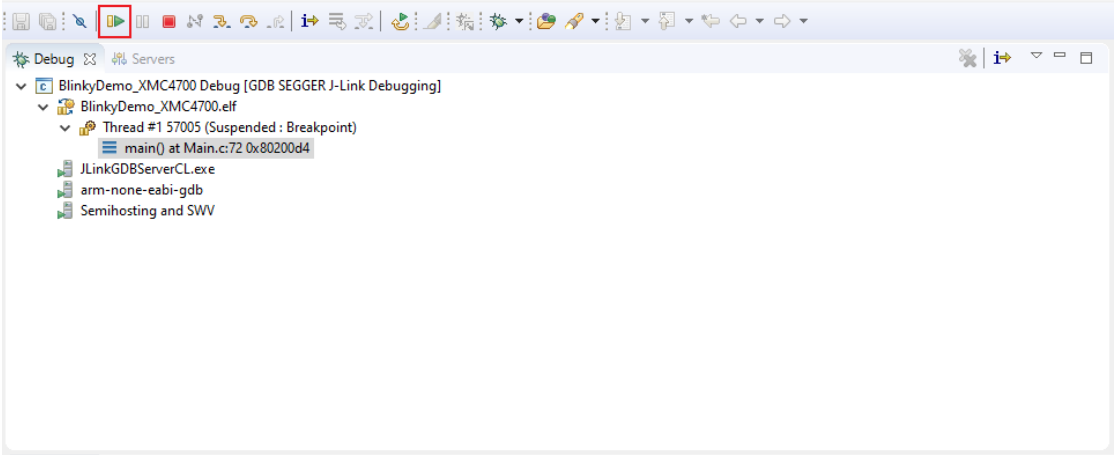


Figure 42: *DAVE IDE* Debug view

9. After starting the program the red on-board LED1 of the Infineon Relax Kit should be blinking!

21 The creation of an external project-specific *X2C®* block

The creation of an external project-specific *X2C* block is summarized in four steps by the subsequent four subsections.
The last two subsections explain what has to be considered when using a project-specific block.

21.1 The creation of the basic structure

The first step in the creation of an external project-specific *X2C* block is the creation of its basic structure using *X2C Block Generator* previously described in Section 13. *X2C Block Generator* can be initiated by executing the `BlockGenerator.jar` file located in `<X2CRoot>\System\Java\`. Once executed, *X2C Block Generator* opens up in the form of the GUI shown in Figure 43 whose options are described hereafter.

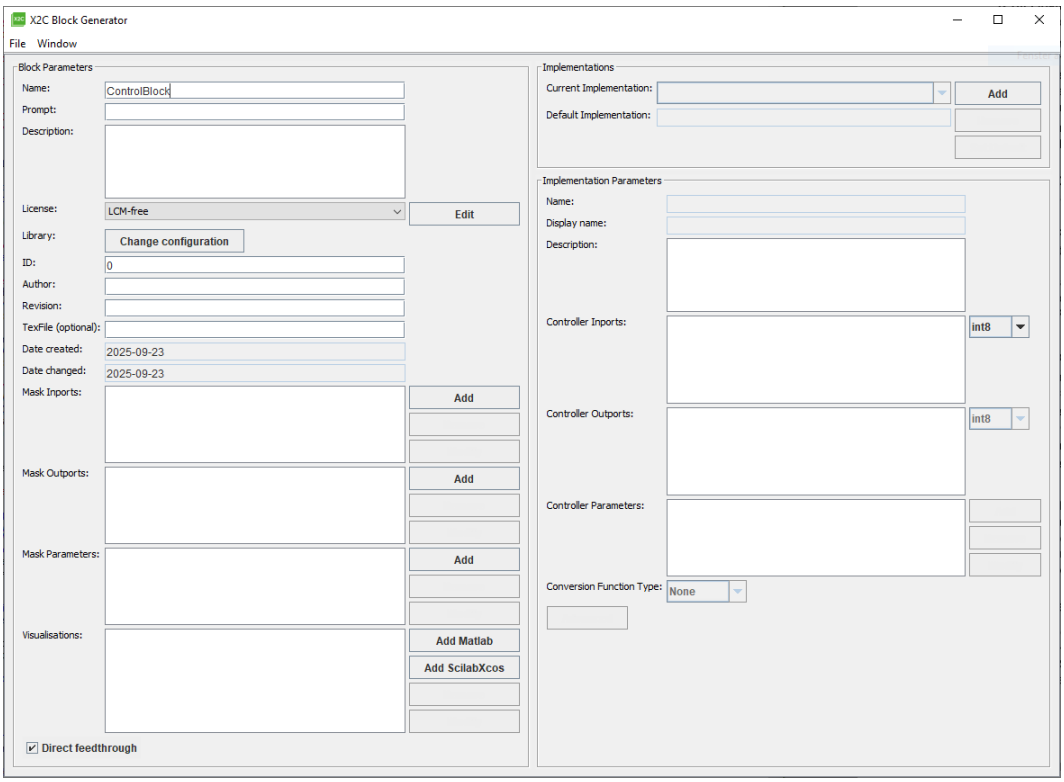


Figure 43: The initial appearance of the *X2C Block Generator* GUI

Block Parameters

- **Name** of the block needs to be uniquely specified for the unambiguous identification of the block in the library and the Simulink or Scilab Xcos model.
- **Prompt** is displayed in the help of the block in Scilab Xcos.
- **Description** of the block is displayed in the dialog window of the block in both Simulink and Scilab Xcos and should describe the purpose of the block.
- **License** can be selected from the drop-down menu that contains various options or it can be manually defined by clicking on the **Edit** push-button.
- **Library** needs to be specified by clicking on the **Change Configuration** push-button that opens up the *Library Configuration* window in which the **External** radio-button should be selected. By doing so, the **Library Name** and **Pre-Namespace** text-fields

become available. It is mandatory to save the library in the `<ProjectRoot>` folder (i.e. the folder that contains the `X2CCode` folder).

  - **Library Name** should be uniquely specified.
  - **Pre-Namespace** is used for Java conversion functions and can be arbitrarily selected. For example the library name can be entered here as well.

- **ID** needs to be an unique integer number between 4000 and 4095, including those two limits.
- **Author** of the block should be specified.
- **Revision** should be specified to enable the distinction between different software versions of the block.
- **TEX File** containing a description of the block in the LATEX format that can be optionally created for the documentation of the block. When writing display math, it is advisable not to use environments provided by the `amsmath` package. Instead, `displaymath`, `equation`, `array`, and `eqnarray` should be used for consistent rendering of display equations by JLATEX in the generated `.html` file. The name of the `.tex` file should differ from the name of the block. (Internal convention is `<BlockName>_Info.tex`).
- **Date Created** is automatically specified upon the creation of the block.
- **Date Changed** is automatically specified every time the block is changed.
- **Mask Inports** are user interfaces that need to be defined for all inputs of the block by clicking on the **Add** push-button that opens up the *Mask Inport* window in which the **Name** and **Description** text-fields under the **Parameters** section become available. Every mask inport can also be removed or edited by clicking on the **Remove** or **Modify** push-button, respectively.

  - **Parameters**
    * **Name** is displayed in the documentation of the block.
    * **Description** is displayed in the documentation of the block.

- **Mask Outports** are user interfaces that need to be defined for all outputs of the block by clicking on the **Add** push-button that opens up the *Mask Outport* window in which the **Name** and **Description** text-fields under the **Parameters** section become available. Every mask outport can also be removed or edited by clicking on the **Remove** or **Modify** push-button, respectively.

  - **Parameters**
    * **Name** is displayed in the documentation of the block.
    * **Description** is displayed in the documentation of the block.

- **Mask Parameters** are user interfaces that need to be defined for all parameters of the block by clicking on the **Add** push-button that opens up the *Mask Parameter* window in which the **Name**, **Prompt**, and **Description** text-fields together with the **Data Type** drop-down menu under the **Parameters** section become available. Every mask parameter can also be removed or edited by clicking on the **Remove** or **Modify** push-button, respectively.

  - **Parameters**
    * **Name** is displayed in the documentation of the block.
    * **Prompt** is displayed in the dialog window of the block in both Simulink and Scilab Xcos.

    * **Description** is displayed in the documentation of the block and the dialog window of the block in both Simulink and Scilab Xcos.
    * **Data Type** can be selected from the drop-down menu to be either **Double** or **ComboBox** as well as **Visible** and/or **Changeable** by the available check-boxes.
  - **Data Type Parameters**
    * **Default** value of the parameter needs to be specified in the case of **Data Type: Double**.
    * **Add** or **Remove** push-buttons become available for adding items in the case of **Data Type: ComboBox**.

- **Visualizations** of the block can be separately defined for Simulink by clicking on the **Add Matlab** push-button as well as for Scilab Xcos by clicking on the **Add ScilabXcos** push-button.
  - **Visualization Parameters** for Simulink
    * **Command** can contain labels of the block and its input(s) and output(s) that can be defined as:
      ```
      disp('\fontsize12<BlockLabel>', 'texmode', 'on')
      port_label('input', '<InputNumber>', '<InputLabel>',
      'texmode', 'on')
      port_label('output', '<OutputNumber>', '<OutputLabel>',
      'texmode', 'on')
      ```
  - **Visualization Parameters** for Scilab Xcos
    * **Define Section** defines options of the block in the `x2c_<BlockName>.sci` script between `// ++ BlockGenerator: Define Section` and `// - BlockGenerator: Define Section`.
    * **Block Icon Default Size** defines the size of the block in Scilab Xcos units in the `x2c_<BlockName>.sci` script between `// ++ BlockGenerator: BlockIconDefaultSize` and `// - BlockGenerator: BlockIconDefaultSize`.
    * **Plot Section** defines options of the block in the `x2c_<BlockName>.sci` script between `// ++ BlockGenerator: Style` and `// - BlockGenerator: Style`.
    * **Style** defines options of the block in the `starter.sce` script between `// ++ BlockGenerator: Plot Section` and `// - BlockGenerator: Plot Section`.

- **Direct Feedthrough** should be generally checked. It should be unchecked when the block serves as a breaker of an algebraic loop.

Implementations

- **Current Implementation** serves as an identifier of the current implementation and can be arbitrarily named, whereby multiple implementations are possible. The internal naming convention of LCM defines **Bool** (Boolean), **FiP8** (8 Bit Fixed Point), **FiP16** (16 Bit Fixed Point), **FiP32** (32 Bit Fixed Point), **Float32** (32 Bit Floating Point), and **Float64** (64 Bit Floating Point) as implementations. A new implementation can be defined by clicking on the **Add** push-button as well as removed by clicking on the **Remove** button.
- **Default Implementation** can be selected by choosing the desired implementation from the **Current Implementation** drop-down menu (if multiple available) and clicking on the **Set Default** push-button.

Implementation Parameters

- **Name** is automatically taken from the **Current Implementation** drop-down menu and every change in the text-field directly renames the selected implementation in the **Current Implementation** drop-down menu. The internal naming convention of LCM defines **Bool**, **FiP8**, **FiP16**, **FiP32**, **Float32**, and **Float64** as names.
- **Display Name** of the current implementation is displayed in the dialog window of the block in the **Used Implementation** drop-down menu. The internal naming convention of LCM defines **Boolean**, **8 Bit Fixed Point**, **16 Bit Fixed Point**, **32 Bit Fixed Point**, **32 Bit Floating Point**, and **64 Bit Floating Point** as display names. This name is used for naming of the corresponding C and Java files.
- **Description** of the current implementation in the documentation of the block. The internal naming convention of LCM defines **Boolean Implementation**, **8 Bit Fixed Point Implementation**, **16 Bit Fixed Point Implementation**, **32 Bit Fixed Point Implementation**, **32 Bit Floating Point Implementation**, or **64 Bit Floating Point Implementation** as descriptions.
- **Controller Inputs** of the current implementation need to be defined to be defined with respect to **Mask Inports**, whereby the data type can be selected from the drop-down menu.
- **Controller Outputs** of the current implementation need to be defined to be defined with respect to **Mask Outports**, whereby the data type can be selected from the drop-down menu.
- **Controller Parameters** of the current implementation need to be defined with respect to **Mask Outports** together with any additional internal parameter can also be defined) by clicking on the **Add** push-button that opens up the *Controller Parameter* window in which the **Name**, **Description**, and **Default Value** text-fields together with the **Data Type** drop-down menu and the **Load & Save Enable** and **Array** check-boxes under the **Controller Parameters** section become available. Every controller parameter can also be removed or edited by clicking on the **Remove** or **Modify** push-button, respectively.
  - **Controller Parameters**
    - **Name** is displayed in the dialog window of the block in both Simulink and Scilab Xcos and denotes a parameter of the block.
    - **Description** is displayed in the dialog window of the block in both Simulink and Scilab Xcos and should describe the parameter of the block.
    - **Data Type** determines the data type that is to be used on the target.
    - **Default Value** sets the default value of the parameter.
- **Conversion Function Type** can be selected from the drop-down menu between **Java**, **Python**, and **JavaScript**. After selecting the conversion function type, a template of the conversion function used for the conversion of mask parameters, which are of the type `double`, into the data type of the current implementation is generated. The template needs to be manually adapted. If there are no mask parameters, no conversion function is necessary.

21.2 Coding the source file

The functionality of the block needs to be manually implemented in the automatically generated `<BlockName>_<Implementation>.c` source file located in `<ProjectRoot>\Library\<LibraryName>\Controller\src\`. In that source file, everything between each pair of the comment tags `/* USERCODE-BEGIN: ... */` and `/* USERCODE-END: ... */` stays

unchanged even if the source file is regenerated by *X2C Block Generator*. Between the first pair of such comments, namely `/* USERCODE-BEGIN:Description */` and `/* USERCODE-END:Description */`, a short description of the block can be written in the form of comments. The definitions of the input(s) and the output(s) as well as parameters, variables, constants, and if necessary, the inclusions of header files should be defined between the `PreProcessor`-tags as

```
/* USERCODE-BEGIN:PreProcessor */
  #include "<HeaderFile>.h"
  #define <INPUT>      (*block->><InportName>)
  #define <OUTPUT>     (block->><OutportName>)
  #define <PARAMETER>  (block->><ParameterName>)
  #define <VARIABLE>   (block->><ParameterName>)
  #define <CONSTANT>   <HexadecimalValue>
/* USERCODE-END:PreProcessor */
```

The functionality of the block should be defined between the `UpdateFnc`-tags and can be demonstrated by an example in a `FiP16` implementation as

```
/* USERCODE-BEGIN:UpdateFnc */
  int32_t <Result>;
  <Result>   = (int32_t)<INPUT> * (int32_t)<VARIABLE>;
  <Result> >>= <PARAMETER>;
  <Result>  += (int32_t)<CONSTANT>;
  if (<Result> > INT16_MAX)
  {
      <Result> = INT16_MAX;
  }
  else
  {
      if (<Result> < -INT16_MAX)
      {
          <Result> = -INT16_MAX;
      }
  }
  <OUTPUT> = (int16_t)<Result>;
/* USERCODE-END:UpdateFnc */
```

The initial values of used variables can be specified between the `InitFnc`-tags as

```
/* USERCODE-BEGIN:InitFnc */
  <Variable> = 0;
/* USERCODE-END:InitFnc */
```

Between `/* USERCODE-BEGIN:LoadFnc */` and `/* USERCODE-END:LoadFnc */` and between `/* USERCODE-BEGIN:SaveFnc */` and `/* USERCODE-END:SaveFnc */` the user can insert code which is executed when the save-function of the block (download of the implementation parameters from the host to the target) or the load-function of the block (upload of the implementation parameters to the host from the target) is called. Usually these sections are left empty.

21.3 Coding the conversion function

If the block uses mask parameters, the conversion function, which converts the mask parameters into controller parameters, has to be implemented in the corresponding programming language as well. The template of the conversion function is located in `<ProjectRoot>\Library\<LibraryName>\Conversion\<ConversionType>\`.

21.3.1 A conversion function in Java

In the case of Java, the conversion function of the block needs to be manually implemented in the automatically generated `ConvFnc_<BlockName>_<Implementation>.java` file located in `<ProjectRoot>\Library\<LibraryName>\Conversion\Java\src\...`. Furthermore, **Eclipse** needs to be installed on the system and the project imported into the workspace via the menu bar as **File** ⇒ **Import...** ⇒ **General** ⇒ **Existing Project**. Under **Project** ⇒ **Build Automatically** needs to be enabled and by right-clicking on the project in the project tree, under **Properties** ⇒ **Java Compiler** ⇒ **Compiler Compliance Level** needs to be set to **1.6**. In the above mentioned file, everything between each pair of the comments `// USERCODE-BEGIN:` and `// USERCODE-END:` stays unchanged even if the source file is regenerated by *X2C Block Generator*. User-defined Java classes like

```
// USERCODE-BEGIN:Imports
  import at.lcm.x2c.utils.QFormat;
// USERCODE-END:Imports
```

can be loaded between the `Imports`-tags, while the conversion of the controller parameters from the mask into the designated implementation needs to be written between the `ConvMaskToImplementation`-tags and can be demonstrated by an example in a `FiP16` implementation as

```
// USERCODE-BEGIN:ConvMaskToImplementation
  double <ControlParameter>;
  final int BITS = 16;

  // Getting the value of the mask parameter:
  <ControlParameter> =
      Double.valueOf(<MaskParameter>MaskVal.getValue());

  // Calculating the shift factor:
  <ShiftFactor>CtrVal.setReal(0, 0,
      Double.valueOf(QFormat.getQFormat(<ControlParameter>, BITS,
      true)));

  // Calculating the Q-value:
  <ControlParameter>CtrVal.setReal(0, 0,
      Double.valueOf(QFormat.getQValue(<ControlParameter>,
      (int)(<ShiftFactor>CtrVal.getReal(0, 0)), BITS, true)));
// USERCODE-END:ConvMaskToImplementation
```

The conversion of the controller parameters from the designated implementation into the mask needs to be written between the `ConvImplementationToMask`-tags as

```
// USERCODE-BEGIN:ConvImplementationToMask
  double <ControlParameter>, <ShiftFactor>;
  final int BITS = 16;

  // Getting the Q-value:
  <ControlParameter> = <ControlParameter>CtrVal.getReal(0, 0);

  // Getting the shift factor:
  <ShiftFactor> = <ShiftFactor>CtrVal.getReal(0, 0);

  // Calculating the value of the mask parameter:
  <MaskParameter>MaskData.setReal(0, 0,
      QFormat.getDecValue((long)<ControlParameter>,
      (int)<ShiftFactor>, BITS, true));
// USERCODE-END:ConvImplementationToMask
```

Eventually a JAR file has to be created to make the conversion function available in Scilab/Xcos. There are multiple ways to create a JAR file. One could work with *.jardesc files in Eclipse, use Apache Ant tasks or simply build the JAR file on the command line.
Taking the command line approach as example, one has to change into the <LibraryDirectory>\Conversion\Java\bin directory and execute following command:

```
jar cf ..\<LibraryName>.jar .\
```

21.3.2 A conversion function in JavaScript

In the case of JavaScript, two JavaScript script files are automatically generated in `<ProjectRoot>\Library\<LibraryName>\Conversion\JavaScript\src\...`, where one is used for the conversion of the mask parameters to the implementation parameters, while the other one is used for the conversion of the implementation parameters to the mask parameters. The conversion of the mask parameters to the implementation parameters needs to be manually implemented in `ConvertMask2Imp_<BlockName>_<Implementation>.js`, while the conversion of the implementation parameters to the mask parameters needs to be implemented in `ConvertImp2Mask_<BlockName>_<Implementation>.js`. Both files contain `/* USERCODE-BEGIN:Description */` and `/* USERCODE-END:Description */` as well as `/* USERCODE-BEGIN:ExternalModules */` and `/* USERCODE-END:ExternalModules */`, where between the first pair a short description of the conversion function can be written, while between the second pair external modules can be imported. The conversion of the mask parameters to the implementation parameters comes between `/* USERCODE-BEGIN:Convert */` and `/* USERCODE-END:Convert */` in the first file, while the conversion of the implementation parameters to the mask parameters comes between `/* USERCODE-BEGIN:Revert */` and `/* USERCODE-END:Revert */` in the second file.

21.3.3 A conversion function in Python

Similar to the case of JavaScript, in the case of Python, two Python scripts are automatically generated in `<ProjectRoot>\Library\<LibraryName>\Conversion\Python\src\...`, where the `ConvertMask2Imp_<BlockName>_<Implementation>.py` script is used for the conversion of the mask parameters to the implementation parameters, while the `ConvertImp2Mask_<BlockName>_<Implementation>.py` script is used for the conversion of the implementation parameters to the mask parameters. The scripts are structurally identical to the above described JavaScript script files containing `# USERCODE-BEGIN: ...` and `# USERCODE-END: ...` sections, where the only difference is the scripting language.

21.3.4 A conversion on target function (CoT) in C

If it should be possible to convert the blocks mask parameters on the target directly, the CoT (Conversion on Target) conversion function has to be coded as well. The template for the CoT conversion function is located in `<ProjectRoot>\Library\<LibraryName>\Conversion\C\src\`.
The conversion of the mask parameters to the controller parameters of the designated implementation needs to be written between the `Conversion`-tags, see following example for the Gain_FiP16 implementation.

```
/* USERCODE-BEGIN:Conversion */
    /* Calculating the shift factor: */
    impParam->sfr = getQFormat(maskParam->Gain, 16);
    /* Calculating the Q-value: */
    impParam->V = getQx16Value(maskParam->Gain, impParam->sfr);
/* USERCODE-END:Conversion */
```

21.4 Finalizing the block in Scilab

In Scilab execute the command

```
createXcosBlock('<LibraryName>', '<BlockName>', '<ProjectRoot>')
```

21.5 The block in an Xcos model

When the project-specific block is used in an Xcos project, the corresponding palette has to be loaded prior opening the Xcos model. To simplify this task, the script `initProject.sce` can be used.
Please follow these steps when working with Scilab/Xcos:

1. Start Scilab.
2. Change into the `<ProjectRoot>\X2CCode` directory.
3. Execute the `initProject.sce` script for loading project-specific palettes. If not present, copy the file from any of the X2C demo applications located in `<X2CRoot>\DemoApplication`.
4. Open the projects Xcos model, the project-specific library should be visible in the palette browser now.

21.6 The block in Code Composer Studio

After including the project in Code Composer Studio, the library should be listed in the project tree. Before compiling, it is necessary to exclude the **Matlab** and **Conversion** folders from the build.

22 Setup *X2C*® for use in a *B&R*® *Automation Studio*® project

X2C allows the user to modify an already created *Automation Studio* project for use with *X2C* and generate the corresponding code. Therefore several configuration parameters have to be set.

22.1 Configuration

- Setup a new *Automation Studio* project or use an existing one
- Head to the Communicator settings and configure the the *Automation Studio* specific parts

  1. Enable the "Create Automation Studio code" option
  2. Select the project APJ file. After a valid project is selected, the configuration menu is updated and contains all project configurations. Further Settings elements will be enabled and updated.
  3. Select a configuration, a CPU module and a Application task class. This becomes the task in which the *X2C* Update function is called.
  4. Enable the Communication option in case TCP/IP communication should be used for online *X2C* Block parameter updates or Scope usage. After enabling, the task class for the *X2C* communication task and the TCP port can be selected.
  5. The "Use library" option allows you to use *X2C* as *Automation Studio* library block.
  6. Sets the *X2C* Scope buffer size. This buffer is used to cache the sampled data if the Scope feature is used.

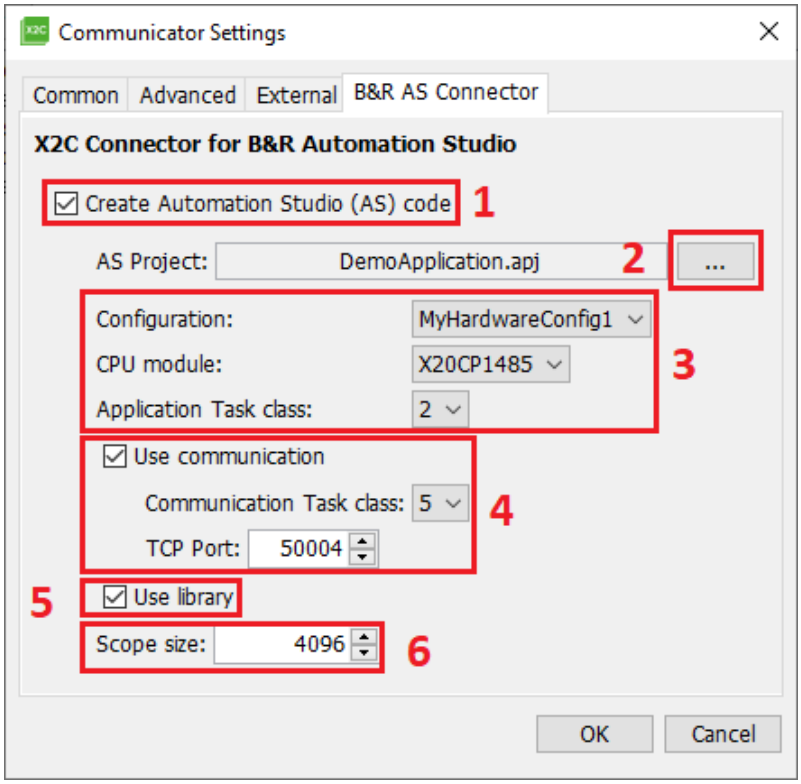


Figure 44: *X2C* Connector for *B&R Automation Studio* settings

- Confirm the changes by pressing the OK button

- Generate the *X2C* code as usual

All required *X2C* and *Automation Studio* files will be created. The Automation Studio configuration files are being updated to include the newly generated *X2C* code.

22.2 Logical View

Now we can head over to *Automation Studio* to check the modifications and build the project. *Automation Studio* may ask you to reload changed components.
The "Logical View" should have been updated by *X2C* content as well as the Software configuration file in the "Configuration View". In case of enabled Communication, the *B&R* library "AsTCP", required for TCP/IP communications, is added to your project libraries. Therefore your *Automation Studio* project may look like this:

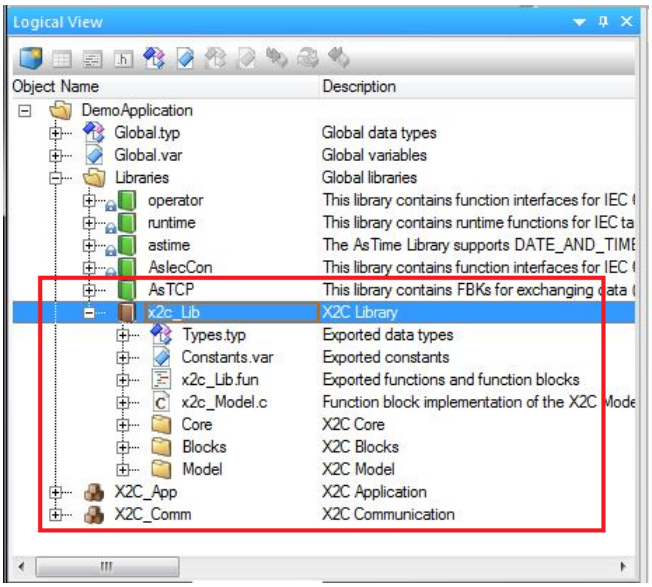


Figure 45: *Automation Studio* Logical View

This example shows the added

- *B&R* AsTCP library
- *X2C* as *Automation Studio* library
- *X2C* application program package
- *X2C* communication program package

22.3 Software Configuration

The software configuration file shows the newly added programs and libraries.

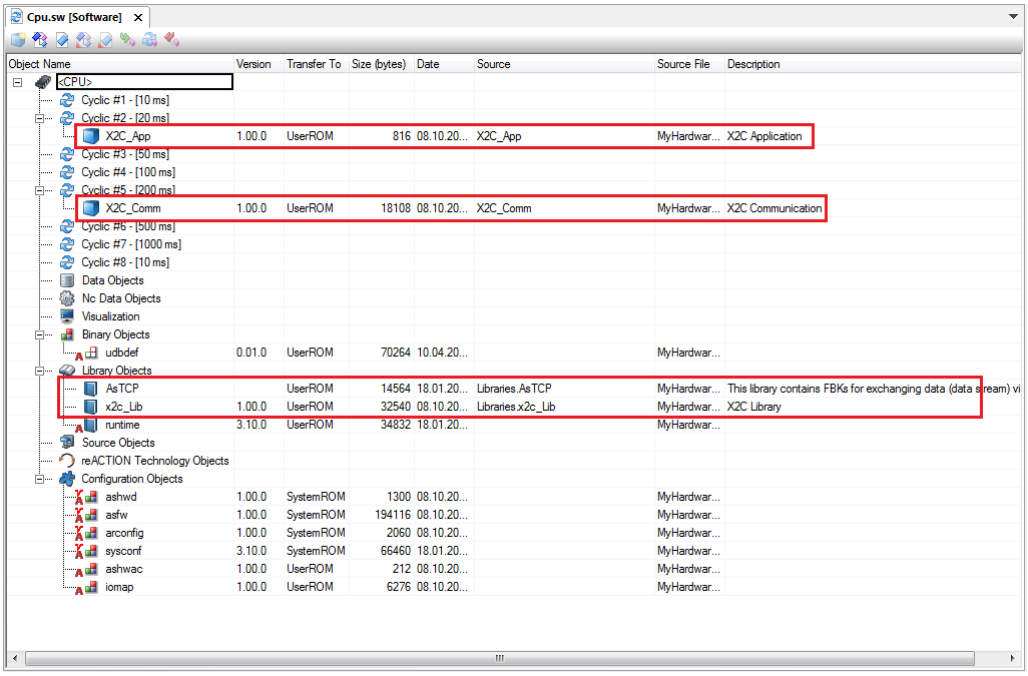


Figure 46: *Automation Studio* Software Configuration

22.4 Communication configuration

In case of enabled communication don't forget to configure the Ethernet and TCP/IP settings for your hardware in *Automation Studio* .

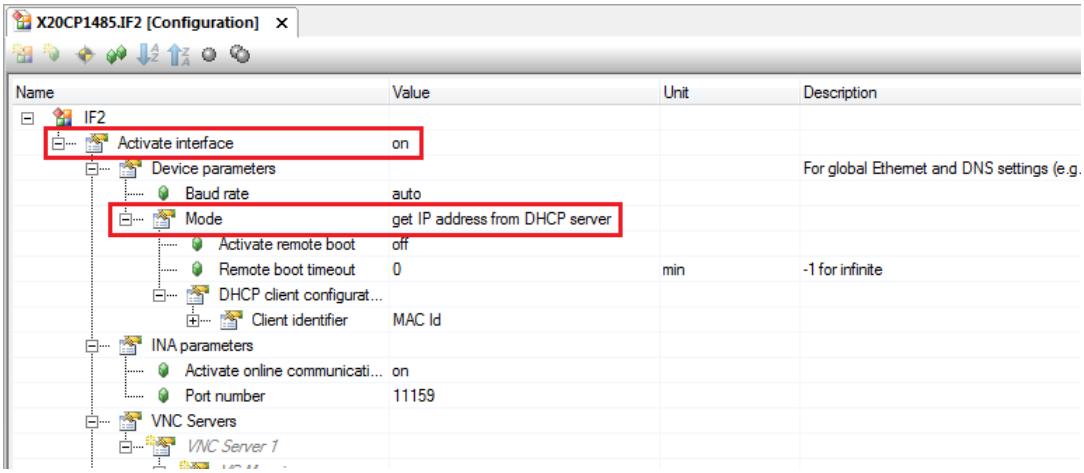


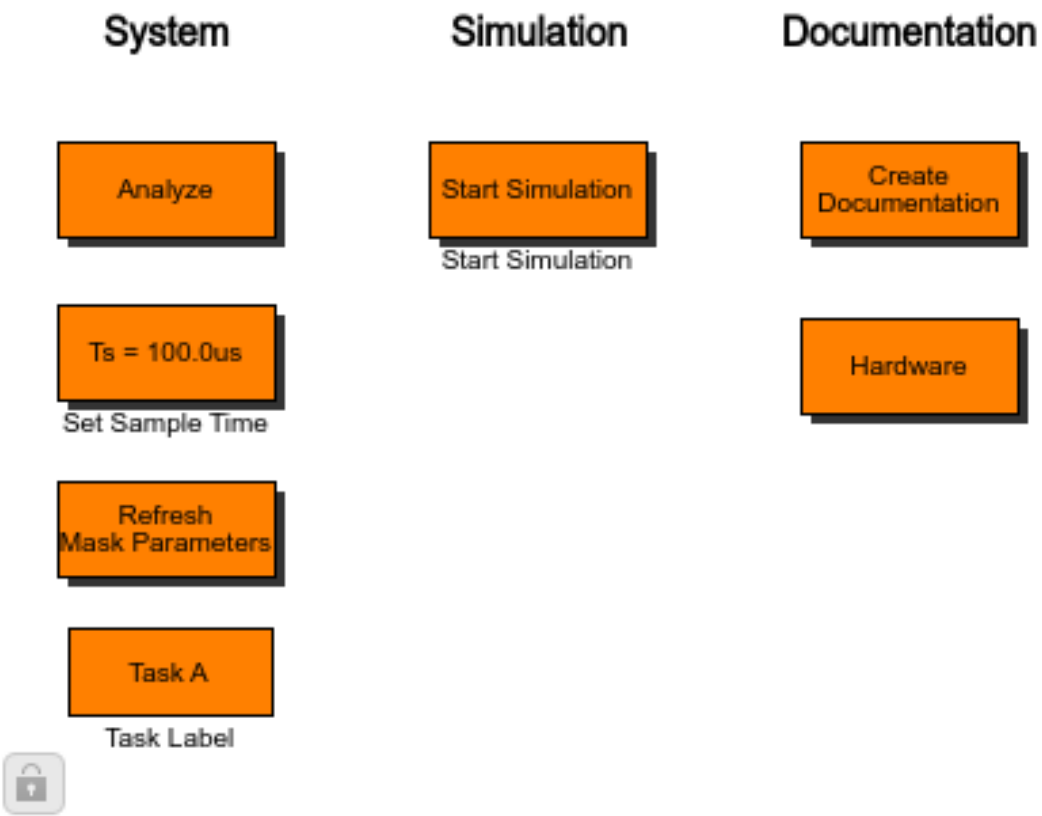
Figure 47: *Automation Studio* Ethernet configuration using DHCP (X20CP1485)

The *X2C* integration is now complete and the *Automation Studio* project is ready to be built.

Part VI
Libraries

23 Basic

Virtual blocks to configure basic code generation, simulation and help generation settings.



Block: CommunicatorStart (Xcos only)

Description:

ith a double-click on this block, the X2C Communicator will start. The Communicator will connect to Scilab via Java Remote Method Invocation (Java RMI) to enable data exchange between Scilab and X2C.

Block: CreateDocumentation

Description:

With a double-click on this block, a documentation with all relevant information about the current X2C project can be created. The document will be created in *<ProjectDir>\Doc* and will be called *ProjectDocumentation_<NameOfModel>.pdf*.

Parameters

User documentation: It is possible to add user specific documentation to the project documentation. When creating the project documentation, the *X2CCode* directory is scanned for the specifed tex file (*UserDoc.tex* by default). If the file is present, it will be included in the documentation.

Test reports: If the checkbox *Add test reports* is selected, test reports of the C-Unit tests of the used X2C blocks are added to the project documentation.

Requirements

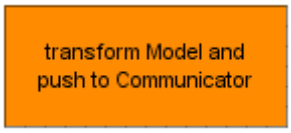
In order to generate a documentation in PDF-format a TeX-compiler (e.g. *MiKTeX*) has to be installed. The software tools *Doxygen* and *Graphviz* are recommended to get documentation of the C-code.

Block: Interact (Xcos only)

Description:

When double-clicked, the block executes the Scilab script *doInteraction.sci*, if available. By customizing the script, the user can execute simple tasks, e.g. setting block parameters, read parameter values, display status information, etc., repeatedly and effortlessly.

Block: ModelTransformation (Xcos only)

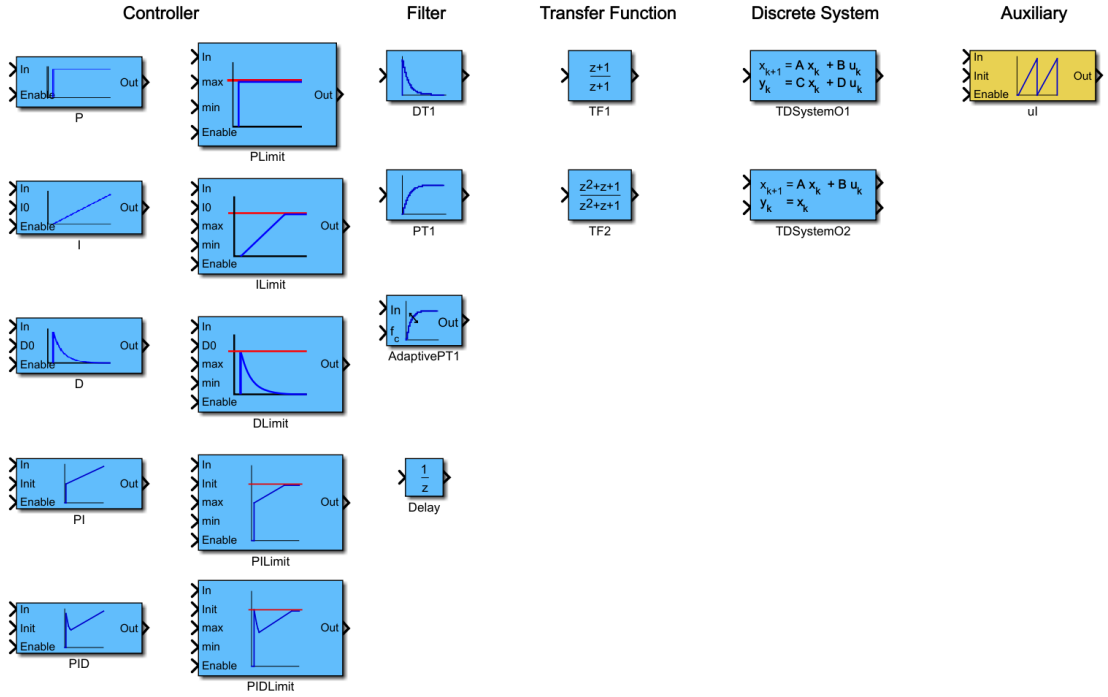


Description:

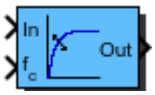
With a double-click on this block, the Xcos model will be analyzed. All relevant information for code generation are gathered, transfered to the Communicator and written to the model XML file.

24 Control

This library contains blocks for building control algorithms.



Block: AdaptivePT1



First order low pass with adaptive cut off frequency:
G(s) = V/(s/(2*pi*fc) + 1)

Inports	
In	Input In(k)
fc	Cutoff frequency

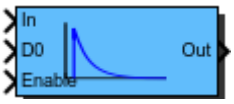
Outports	
Out	Output Out(k)

Mask Parameters		
Name	**ID**	**Description**
V	1	Gain
fmax	2	Maximum frequency [Hz] (not used in floating point implementations)
ts_fact	3	Multiplication factor of base sampling time (in integer format)
method	4	Discretization method

Implementations:

FiP8	8 Bit Fixed Point Implementation
FiP16	16 Bit Fixed Point Implementation
FiP32	32 Bit Fixed Point Implementation
Float32	32 Bit Floating Point Implementation
Float64	64 Bit Floating Point Implementation

Block: D



D Controller:
G(s) = Kd*s/(s/(2*pi*fc) + 1)

Inports	
In	Control error input
D0	Derivational value which is loaded during initialization. (Usually this inport is connect to the In inport.)
Enable	Enable == 0: Deactivation of block; Out set to 0 Enable 0->1: Preload of derivational part Enable == 1: Activation of block

Outports	
Out	Control value

Mask Parameters		
Name	**ID**	**Description**
Kd	1	Derivative Factor
fc	2	Cutoff frequency of realization low pass
ts_fact	3	Multiplication factor of base sampling time (in integer format)

Implementations:

FiP16 16 Bit Fixed Point Implementation
FiP32 32 Bit Fixed Point Implementation
Float32 32 Bit Floating Point Implementation
Float64 64 Bit Floating Point Implementation

Block: Delay

Output delay by one sample time interval.

Inports	
In	Input In(k)

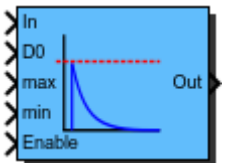
Outports	
Out	Output Out(k)=In(k-1)

Mask Parameters		
Name	**ID**	**Description**
ts_fact	1	Multiplication factor of base sampling time (in integer format)

Implementations:

Bool	Boolean Integration
FiP16	16 Bit Fixed Point Implementation
FiP32	32 Bit Fixed Point Implementation
Float32	32 Bit Floating Point Implementation
Float64	64 Bit Floating Point Implementation

Block: DLimit



D Controller with Output Limitation:
G(s) = Kd*s/(s/(2*pi*fc) + 1)

Inports	
In	Control error input
D0	Derivational value which is loaded during initialization. (Usually this inport is connect to the In inport.)
max	Maximum output value
min	Minimum output value
Enable	Enable == 0: Deactivation of block; Out set to 0 Enable == 1: Activation of block

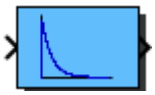
Outports	
Out	Control value

Mask Parameters		
Name	**ID**	**Description**
Kd	1	Derivative Factor
fc	2	Cutoff frequency of realization low pass
ts_fact	3	Multiplication factor of base sampling time (in integer format)

Implementations:

FiP16 16 Bit Fixed Point Implementation
FiP32 32 Bit Fixed Point Implementation
Float32 32 Bit Floating Point Implementation
Float64 64 Bit Floating Point Implementation

Block: DT1



First order high pass:
G(s) = V*s/(s/w + 1)

Inports	
In	Input In(k)

Outports	
Out	Output Out(k)

Mask Parameters		
Name	**ID**	**Description**
V	1	Gain
fc	2	Cut off frequency of low pass filter
ts_fact	3	Multiplication factor of base sampling time (in integer format)
method	4	Discretization method

Implementations:

FiP8	8 Bit Fixed Point Implementation
FiP16	16 Bit Fixed Point Implementation
FiP32	32 Bit Fixed Point Implementation
Float32	32 Bit Floating Point Implementation
Float64	64 Bit Floating Point Implementation

Block: I

I controller:
G(s) = Ki/s = 1/(Ti*s)

Inports	
In	Control error input
I0	Integral value which is loaded at initialization function call
Enable	Enable == 0: Deactivation of block; Out set to 0 Enable 0->1: Preload of integral part Enable == 1: Activation of block

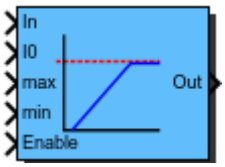
Outports	
Out	Control value

Mask Parameters		
Name	**ID**	**Description**
Ki	1	Integral Factor
ts_fact	2	Multiplication factor of base sampling time (in integer format)

Implementations:

FiP16 16 Bit Fixed Point Implementation
FiP32 32 Bit Fixed Point Implementation
Float32 32 Bit Floating Point Implementation
Float64 64 Bit Floating Point Implementation

Block: ILimit



I controller with output limitation:
G(s) = Ki/s = 1/(Ti*s)

Inports	
In	Control error input
I0	Integral value which is loaded at initialization function call
max	Maximum output value
min	Minimum output value
Enable	Enable == 0: Deactivation of block; Out set to 0 Enable 0->1: Preload of integral part Enable == 1: Activation of block

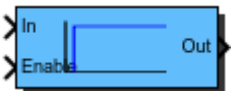
Outports	
Out	Control value

Mask Parameters		
Name	**ID**	**Description**
Ki	1	Integral Factor
ts_fact	2	Multiplication factor of base sampling time (in integer format)

Implementations:

FiP16 16 Bit Fixed Point Implementation
FiP32 32 Bit Fixed Point Implementation
Float32 32 Bit Floating Point Implementation
Float64 64 Bit Floating Point Implementation

Block: P



P controller:
G(s) = Kp

Inports	
In	Control error input
Enable	Enable == 0: Deactivation of block; Out set to 0 Enable == 1: Activation of block

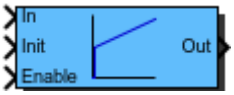
Outports	
Out	Control value

Mask Parameters		
Name	**ID**	**Description**
Kp	1	Proportional Factor

Implementations:

FiP16 16 Bit Fixed Point Implementation
FiP32 32 Bit Fixed Point Implementation
Float32 32 Bit Floating Point Implementation
Float64 64 Bit Floating Point Implementation

Block: PI



PI controller:
G(s) = Kp + Ki/s

Inports	
In	Control error input
Init	Initial value of output Out when enabled
Enable	Enable == 0: Deactivation of block; Out set to 0 Enable 0->1: Preload of integral part Enable == 1: Activation of block

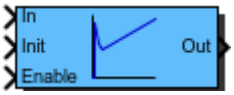
Outports	
Out	Control value

Mask Parameters		
Name	**ID**	**Description**
Kp	1	Proportional Factor
Ki	2	Integral Factor
initHandling	3	Determines what the signal provided on Init input is used for
ts_fact	4	Multiplication factor of base sampling time (in integer format)

Implementations:

FiP8 8 Bit Fixed Point Implementation
FiP16 16 Bit Fixed Point Implementation
FiP32 32 Bit Fixed Point Implementation
Float32 32 Bit Floating Point Implementation
Float64 64 Bit Floating Point Implementation

Block: PID



PID controller:
G(s) = Kp + Ki/s + Kd*s/(s/(2*pi*fc) + 1)

Inports	
In	Control error input
Init	Initial value of output Out when enabled
Enable	Enable == 0: Deactivation of block; Out set to 0 Enable 0->1: Preload of integral part Enable == 1: Activation of block

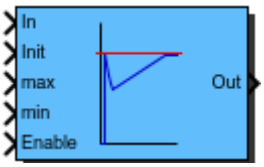
Outports	
Out	Control value

Mask Parameters		
Name	**ID**	**Description**
Kp	1	Proportional Factor
Ki	2	Integral Factor
Kd	3	Derivative Factor
fc	4	Cutoff frequency of realization low pass
initHandling	5	Determines what the signal provided on Init input is used for
ts_fact	6	Multiplication factor of base sampling time (in integer format)

Implementations:

FiP8	8 Bit Fixed Point Implementation
FiP16	16 Bit Fixed Point Implementation
FiP32	32 Bit Fixed Point Implementation
Float32	32 Bit Floating Point Implementation
Float64	64 Bit Floating Point Implementation

Block: PIDLimit



PID Controller with Output Limitation and anti wind-up method clamping:
G(s) = Kp + Ki/s + Kd*s/(s/(2*pi*fc) + 1)

Inports	
In	Control error input
Init	Initial value of output Out when enabled
max	Maximum output value
min	Minimum output value
Enable	Enable == 0: Deactivation of block; Out set to 0 Enable 0->1: Preload of integral part Enable == 1: Activation of block

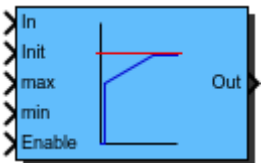
Outports	
Out	Control value

Mask Parameters		
Name	**ID**	**Description**
Kp	1	Proportional Factor
Ki	2	Integral Factor
Kd	3	Derivative Factor
fc	4	Cutoff frequency of realization low pass
initHandling	5	Determines what the signal provided on Init input is used for
ts_fact	6	Multiplication factor of base sampling time (in integer format)

Implementations:

FiP8	8 Bit Fixed Point Implementation
FiP16	16 Bit Fixed Point Implementation
FiP32	32 Bit Fixed Point Implementation
Float32	32 Bit Floating Point Implementation
Float64	64 Bit Floating Point Implementation

Block: PILimit



PI controller with output limitation and anti wind-up method clamping:
G(s) = Kp + Ki/s

Inports	
In	Control error input
Init	Initial value of output Out when enabled
max	Maximum output value
min	Minimum output value
Enable	Enable == 0: Deactivation of block; Out set to 0 Enable 0->1: Preload of integral part Enable == 1: Activation of block

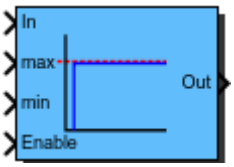
Outports	
Out	Control value

Mask Parameters		
Name	**ID**	**Description**
Kp	1	Proportional Factor
Ki	2	Integral Factor
initHandling	3	Determines what the signal provided on Init input is used for
ts_fact	4	Multiplication factor of base sampling time (in integer format)

Implementations:

FiP8	8 Bit Fixed Point Implementation
FiP16	16 Bit Fixed Point Implementation
FiP32	32 Bit Fixed Point Implementation
Float32	32 Bit Floating Point Implementation
Float64	64 Bit Floating Point Implementation

Block: PLimit



P controller with output limitation:
G(s) = Kp

Inports	
In	Control error input
max	Maximum output value
min	Minimum output value
Enable	Enable == 0: Deactivation of block; Out set to 0 Enable == 1: Activation of block

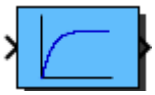
Outports	
Out	Control value

Mask Parameters		
Name	**ID**	**Description**
Kp	1	Proportional Factor

Implementations:

FiP16 16 Bit Fixed Point Implementation
FiP32 32 Bit Fixed Point Implementation
Float32 32 Bit Floating Point Implementation
Float64 64 Bit Floating Point Implementation

Block: PT1



First order low pass:
G(s) = V/(s/w + 1)

Inports	
In	Input In(k)

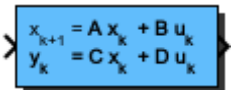
Outports	
Out	Output Out(k)

Mask Parameters		
Name	**ID**	**Description**
V	1	Gain
fc	2	Cut off frequency of low pass filter
ts_fact	3	Multiplication factor of base sampling time (in integer format)
method	4	Discretization method

Implementations:

FiP8	8 Bit Fixed Point Implementation
FiP16	16 Bit Fixed Point Implementation
FiP32	32 Bit Fixed Point Implementation
Float32	32 Bit Floating Point Implementation
Float64	64 Bit Floating Point Implementation

Block: TDSystemO1



1st order time discrete system with one input and one output.

Inports	
In	Input #1

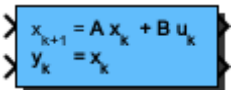
Outports	
Out	Output #1

Mask Parameters		
Name	**ID**	**Description**
A	1	State matrix A
B	2	Input matrix B
C	3	Output matrix C
D	4	Feedthrough matrix D

Implementations:

FiP8	8 Bit Fixed Point Implementation
FiP16	16 Bit Fixed Point Implementation
FiP32	32 Bit Fixed Point Implementation
Float32	32 Bit Floating Point Implementation
Float64	64 Bit Floating Point Implementation

Block: TDSystemO2



2nd order time discrete system with two inputs and two outputs.

Inports	
In1	Input #1
In2	Input #2

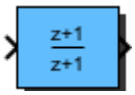
Outports	
Out1	Output #1
Out2	Output #2

Mask Parameters		
Name	**ID**	**Description**
A	1	State matrix A
B	2	Input matrix B

Implementations:

FiP8	8 Bit Fixed Point Implementation
FiP16	16 Bit Fixed Point Implementation
FiP32	32 Bit Fixed Point Implementation
Float32	32 Bit Floating Point Implementation
Float64	64 Bit Floating Point Implementation

Block: TF1



First order transfer function:
G(z) = (b1.z + b0) / (z + a0)

Inports	
In	Input In(k)

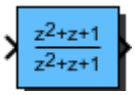
Outports	
Out	Output Out(k)

Mask Parameters		
Name	**ID**	**Description**
b1	1	b1
b0	2	b0
a0	3	a0
ts_fact	4	Multiplication factor of base sampling time (in integer format)

Implementations:

FiP8	8 Bit Fixed Point Implementation
FiP16	16 Bit Fixed Point Implementation
FiP32	32 Bit Fixed Point Implementation
Float32	32 Bit Floating Point Implementation
Float64	64 Bit Floating Point Implementation

Block: TF2



Second order transfer function:
$G(z) = (b2.z^2 + b1.z + b0) / (z^2 + a1.z + a0)$

Inports	
In	Input In(k)

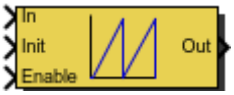
Outports	
Out	Output Out(k)

Mask Parameters		
Name	**ID**	**Description**
b2	1	b2
b1	2	b1
b0	3	b0
a1	4	a1
a0	5	a0
ts_fact	6	Multiplication factor of base sampling time (in integer format)

Implementations:

FiP16	16 Bit Fixed Point Implementation
FiP8	8 Bit Fixed Point Implementation
FiP32	32 Bit Fixed Point Implementation
Float32	32 Bit Floating Point Implementation
Float64	64 Bit Floating Point Implementation

Block: uI



Integrator for angle signals:
G(s) = Ki/s = 1/(Ti*s)

Inports	
In	Control error input
Init	Value which is loaded at initialization function call
Enable	Enable == 0: Deactivation of block; Out is set to 0. Enable 0->1: Preload of integral part. Enable == 1: Activation of block

Outports	
Out	Integrator output

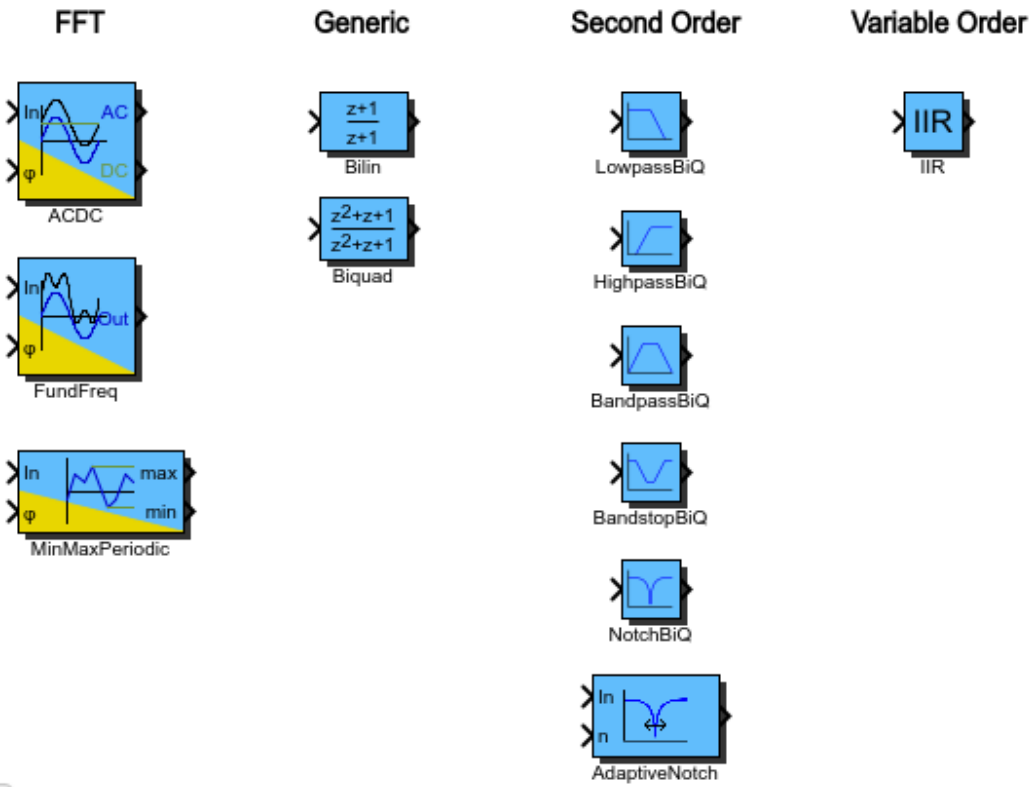
Mask Parameters		
Name	**ID**	**Description**
Ki	1	Integral Factor
ts_fact	2	Multiplication factor of base sampling time (in integer format)

Implementations:

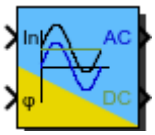
FiP8	8 Bit Fixed Point Implementation
FiP16	16 Bit Fixed Point Implementation
FiP32	32 Bit Fixed Point Implementation
Float32	32 Bit Floating Point Implementation
Float64	64 Bit Floating Point Implementation

25 Filter

Library with various kinds of digital filter blocks.



Block: ACDC



Calculation of AC and DC part of input signal.

Inports	
In	Input signal
phi	Angle signal

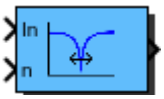
Outports	
AC	AC part of input signal
DC	DC part of input signal

Mask Parameters		
Name	**ID**	**Description**
ts_fact	1	Multiplication factor of base sampling time (in integer format)

Implementations:

FiP16	16 Bit Fixed Point Implementation
FiP32	32 Bit Fixed Point Implementation
Float32	32 Bit Floating Point Implementation
Float64	64 Bit Floating Point Implementation

Block: AdaptiveNotch



Notch filter with variable notch frequency

Inports	
In	Input signal
n	Speed input = notch frequency

Outports	
Out	Filtered output signal

Mask Parameters		
Name	**ID**	**Description**
Q	1	Q-Factor of the notch filter
n_thresh	2	Speed threshold for activating the filter
n_max	3	Maximum revolutions per minute (Not used in floating point implementations)
p	4	Number of pole pairs
ts_fact	5	Multiplication factor of base sampling time (in integer format)
method	6	Discretization method

Implementations:

FiP16 16 Bit Fixed Point Implementation
FiP32 32 Bit Fixed Point Implementation
Float32 32 Bit Floating Point Implementation
Float64 64 Bit Floating Point Implementation

Block: BandpassBiQ

Calculates the filter coefficients for a second order bandpass and performs filtering on input signal.
Second order transfer function used:
$H(z) = (b0.z^2 + b1.z + b2) / (z^2 + a1.z + a2)$

Inports	
In	Input In(k)

Outports	
Out	Output Out(k)

Mask Parameters		
Name	**ID**	**Description**
characteristic	1	The filter characteristic is one of Butterworth (=Bessel) or Chebyshev
fc1	2	Lower cut-off frequency in Hz
fc2	3	Upper cut-off frequency in Hz
rc	4	Attenuation at cut-off frequencies
ts_fact	5	Multiplication factor of base sampling time (in integer format)

Implementations:

FiP16 16 Bit Fixed Point Implementation
FiP32 32 Bit Fixed Point Implementation
Float32 32 Bit Floating Point Implementation
Float64 64 Bit Floating Point Implementation

Block: BandstopBiQ

Calculates the filter coefficients for a second order bandstop and performs filtering on input signal.
Second order transfer function used:
$H(z) = (b0.z^2 + b1.z + b2) / (z^2 + a1.z + a2)$

Inports	
In	Input In(k)

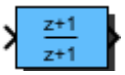
Outports	
Out	Output Out(k)

Mask Parameters		
Name	**ID**	**Description**
characteristic	1	The filter characteristic is one of Butterworth (=Bessel) or Chebyshev
fc1	2	Lower cut-off frequency in Hz
fc2	3	Upper cut-off frequency in Hz
rc	4	Attenuation at cut-off frequencies
ts_fact	5	Multiplication factor of base sampling time (in integer format)

Implementations:

FiP16 16 Bit Fixed Point Implementation
FiP32 32 Bit Fixed Point Implementation
Float32 32 Bit Floating Point Implementation
Float64 64 Bit Floating Point Implementation

Block: Bilin



First order transfer function:
H(z) = (b0.z + b1) / (a0.z + a1)

Inports	
In	Input In(k)

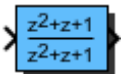
Outports	
Out	Output Out(k)

Mask Parameters		
Name	**ID**	**Description**
b0	1	Filter coefficient b0
b1	2	Filter coefficient b1
a0	3	Filter coefficient a0
a1	4	Filter coefficient a1
ts_fact	5	Multiplication factor of base sampling time (in integer format)

Implementations:

FiP16 16 Bit Fixed Point Implementation
FiP32 32 Bit Fixed Point Implementation
Float32 32 Bit Floating Point Implementation
Float64 64 Bit Floating Point Implementation

Block: Biquad



Second order transfer function:
$H(z) = (b0.z^2 + b1.z + b2) / (a0.z^2 + a1.z + a2)$

Inports	
In	Input In(k)

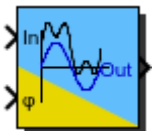
Outports	
Out	Output Out(k)

Mask Parameters		
Name	**ID**	**Description**
b0	1	Filter coefficient b0
b1	2	Filter coefficient b1
b2	3	Filter coefficient b2
a0	4	Filter coefficient a0
a1	5	Filter coefficient a1
a2	6	Filter coefficient a2
ts_fact	7	Multiplication factor of base sampling time (in integer format)

Implementations:

FiP16 16 Bit Fixed Point Implementation
FiP32 32 Bit Fixed Point Implementation
Float32 32 Bit Floating Point Implementation
Float64 64 Bit Floating Point Implementation

Block: FundFreq



Calculation of the first harmonic (fundamental frequency) of the input signal.

Inports	
In	Input signal
phi	Angle signal

Outports	
Out	First harmonic

Mask Parameters		
Name	**ID**	**Description**
ts_fact	1	Multiplication factor of base sampling time (in integer format)

Implementations:

FiP16	16 Bit Fixed Point Implementation
FiP32	32 Bit Fixed Point Implementation
Float32	32 Bit Floating Point Implementation
Float64	64 Bit Floating Point Implementation

Block: HighpassBiQ

Calculates the filter coefficients for a second order highpass and performs filtering on input signal.
Second order transfer function used:
$H(z) = (b0.z^2 + b1.z + b2) / (z^2 + a1.z + a2)$

Inports	
In	Input In(k)

Outports	
Out	Output Out(k)

Mask Parameters		
Name	**ID**	**Description**
characteristic	1	The filter characteristic is one of Butterworth, Chebyshev I, Chebyshev II (also known as inverse Chebyshev) or Bessel
fc	2	Cut-off frequency in Hz
rp	3	The passband ripple is only used and verified with 'Chebyshev I' filters. Must be a positive number
rs	4	The stopband ripple is only used and verified with 'Chebyshev II' filters. Must be a positive number
ts_fact	5	Multiplication factor of base sampling time (in integer format)

Implementations:

FiP16 16 Bit Fixed Point Implementation
FiP32 32 Bit Fixed Point Implementation
Float32 32 Bit Floating Point Implementation
Float64 64 Bit Floating Point Implementation

Block: IIR

Transfer function:
H(z) = (b0.z^n + b1.z + ...+ bn) / (a0.z^n + a1.z + ... + an)
The maximum filter order is 32.
The order of the denominator has to be greater than or equal to the order of the numerator.

Inports	
In	Input In(k)

Outports	
Out	Output Out(k)

Mask Parameters		
Name	**ID**	**Description**
CoeffB	1	Array with filter coefficients of numerator
CoeffA	2	Array with filter coefficients of denominator
ts_fact	3	Multiplication factor of base sampling time (in integer format)

Implementations:

FiP16 16 Bit Fixed Point Implementation
FiP32 32 Bit Fixed Point Implementation
Float32 32 Bit Floating Point Implementation
Float64 64 Bit Floating Point Implementation

Block: LowpassBiQ

Calculates the filter coefficients for a second order lowpass and performs filtering on input signal.
Second order transfer function used:
$H(z) = (b0.z^2 + b1.z + b2) / (z^2 + a1.z + a2)$

Inports	
In	Input In(k)

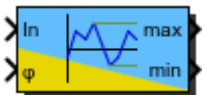
Outports	
Out	Output Out(k)

Mask Parameters		
Name	**ID**	**Description**
characteristic	1	The filter characteristic is one of Butterworth, Chebyshev I, Chebyshev II (also known as inverse Chebyshev) or Bessel
fc	2	Cut-off frequency in Hz
rp	3	The passband ripple is only used and verified with 'Chebyshev I' filters. Must be a positive number
rs	4	The stopband ripple is only used and verified with 'Chebyshev II' filters. Must be a positive number
ts_fact	5	Multiplication factor of base sampling time (in integer format)

Implementations:

FiP16	16 Bit Fixed Point Implementation
FiP32	32 Bit Fixed Point Implementation
Float32	32 Bit Floating Point Implementation
Float64	64 Bit Floating Point Implementation

Block: MinMaxPeriodic



Outputs the minimum and maximum of the input signal over one period of the (angle) signal phi.

Inports	
In	Input signal
phi	Angle signal

Outports	
max	Maximum of input signal
min	Minimum of input signal

Implementations:

FiP16 16 Bit Fixed Point Implementation
FiP32 32 Bit Fixed Point Implementation
Float32 32 Bit Floating Point Implementation
Float64 64 Bit Floating Point Implementation

Block: NotchBiQ

Calculates the filter coefficients for a second order notch filter and performs filtering on input signal.
Second order transfer function used:
$H(z) = (b0.z^2 + b1.z + b2) / (z^2 + a1.z + a2)$

Inports	
In	Input In(k)

Outports	
Out	Output Out(k)

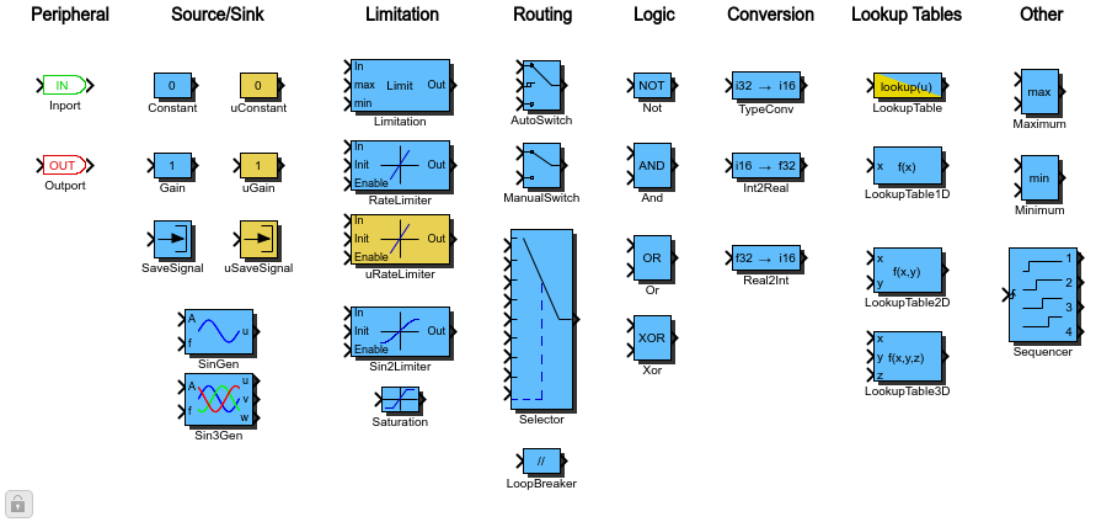
Mask Parameters		
Name	**ID**	**Description**
method	1	Selection between bilinear and matched Z-transform discretization method
fn	2	Notching frequency in Hz
Q	3	Defines a quality factor of the filter
ts_fact	4	Multiplication factor of base sampling time (in integer format)

Implementations:

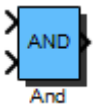
FiP16 16 Bit Fixed Point Implementation
FiP32 32 Bit Fixed Point Implementation
Float32 32 Bit Floating Point Implementation
Float64 64 Bit Floating Point Implementation

26 General

Library with general purpose blocks.



Block: And



Logical AND block.

Inports	
In1	
In2	

Outports	
Out	

Implementations:

Bool Boolean Implementation

Block: AutoSwitch

Switch between In1 and In3 dependent on Switch signal:
Switch signal rising: Switch >= Threshold up –> Out = In1
Switch signal falling: Switch < Threshold down –> Out = In3

Inports	
In1	Input #1
Switch	Input #2: Threshold signal
In3	Input #3

Outports	
Out	Either value of input #1 or input #3 dependent on value of input #2

Mask Parameters		
Name	**ID**	**Description**
Thresh_up	1	Threshold level for rising switch signal
Thresh_down	2	Threshold level for falling switch signal

Implementations:

FiP16 16 Bit Fixed Point Implementation
FiP32 32 Bit Fixed Point Implementation
Float32 32 Bit Floating Point Implementation
Float64 64 Bit Floating Point Implementation

Block: Constant

Constant value.

Outports	
Out	Constant output

Mask Parameters		
Name	**ID**	**Description**
Value	1	Constant factor

Implementations:

Bool	Boolean Implementation
Int8	8 Bit Integer Implementation
Int16	16 Bit Integer Implementation
Int32	32 Bit Integer Implementation
FiP8	8 Bit Fixed Point Implementation
FiP16	16 Bit Fixed Point Implementation
FiP32	32 Bit Fixed Point Implementation
Float32	32 Bit Floating Point Implementation
Float64	64 Bit Floating Point Implementation

Block: Gain

Amplification of input by gain factor.

Inports	
In	Input

Outports	
Out	Amplified input

Mask Parameters		
Name	**ID**	**Description**
Gain	1	Gain factor in floating point format

Implementations:

FiP8	8 Bit Fixed Point Implementation
FiP16	16 Bit Fixed Point Implementation
FiP32	32 Bit Fixed Point Implementation
Float32	32 Bit Floating Point Implementation
Float64	64 Bit Floating Point Implementation

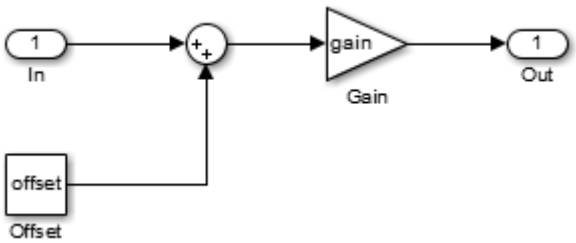
Block: Inport

Inports	
IN	Signal from frame program

Mask Parameters	
ts_fact	Multiplication factor of base sampling time (in integer format)
Gain	Gain value used in simulation
Offset	Offset value used in simulation

Description:

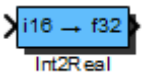
Serves as interface to the frame program. The input of this block is intended for simulation purposes and can be left unconnected if not used. Also the parameters *Gain* and *Offset* are only used during simulation. The schematic for simulation can be seen in the figure below.
Note: Currently, *Gain* and *Offset* parameters are only available in Matlab/Simulink.



Data Types:

bool	Boolean
int8	8 Bit Fixed Point
int16	16 Bit Fixed Point
int32	32 Bit Fixed Point
float32	32 Bit Floating Point
float64	64 Bit Floating Point

Block: Int2Real



Conversion block from integer (fixed point) datatypes to real (floating point) datatypes.
Out = In * Scale

Inports	
In	Integer input

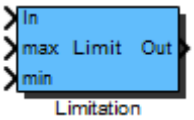
Outports	
Out	Real output

Mask Parameters		
Name	**ID**	**Description**
Scale	1	Scaling factor from integer to real

Implementations:

FiP8_Float32	8 Bit Fixed Point to 32 Bit Floating Point Implementation
FiP16_Float32	16 Bit Fixed Point to 32 Bit Floating Point Implementation
FiP32_Float32	32 Bit Fixed Point to 32 Bit Floating Point Implementation
FiP8_Float64	8 Bit Fixed Point to 64 Bit Floating Point Implementation
FiP16_Float64	16 Bit Fixed Point to 64 Bit Floating Point Implementation
FiP32_Float64	32 Bit Fixed Point to 64 Bit Floating Point Implementation
Bool_Float32	Boolean to 32 Bit Floating Point Implementation
Bool_Float64	Boolean to 64 Bit Floating Point Implementation

Block: Limitation



Limits the input signal to min and max_sci.
Caution: For correct computation the upper limit max has to be greater than the lower limit min!

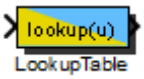
Inports	
In	Input signal
max	Upper limit
min	Lower limit

Outports	
Out	Limited input signal

Implementations:

FiP16	16 Bit Fixed Point Implementation
FiP32	32 Bit Fixed Point Implementation
Float32	32 Bit Floating Point Implementation
Float64	64 Bit Floating Point Implementation

Block: LookupTable



Look-up Table with 256+1 values.
Note: 257th value is used for preventing index overflow during interpolation.
-> for periodic signals the 257th value should be set equal to 1st value
-> for non-periodic signals the 257th value should be set equal to 256th value

Inports	
In	Table index

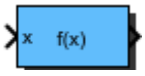
Outports	
Out	Table output

Mask Parameters		
Name	**ID**	**Description**
Lookup	1	Look-up Table

Implementations:

FiP8	8 Bit Fixed Point Implementation
FiP16	16 Bit Fixed Point Implementation
FiP32	32 Bit Fixed Point Implementation

Block: LookupTable1D



One dimensional look-up table with selectable number of data points.
Table data must be an array of size DimX. If one of the RAM implementations is used, DimX must not exceed 257.
If input is out of specified range, output will be cut off (no extrapolation).

Inports	
x	Table index in x direction

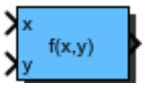
Outports	
Out	Table output

Mask Parameters		
Name	**ID**	**Description**
TableData	1	Look-up table data
DimX	2	Number of data points in x-direction
x_min	3	Minimum input value of x-dimension for look-up table
x_max	4	Maximum input value of x-dimension for look-up table

Implementations:

FiP16	64 Bit Floating Point Implementation with look-up table located in RAM
FiP32	32 Bit Fixed Point Implementation with look-up table located in Flash memory
Float32	32 Bit Floating Point Implementation with look-up table located in Flash memory
Float64	64 Bit Floating Point Implementation with look-up table located in Flash memory
FiP16_RAM	16 Bit Fixed Point Implementation with look-up table located in RAM
FiP32_RAM	32 Bit Fixed Point Implementation with look-up table located in RAM
Float32_RAM	32 Bit Floating Point Implementation with look-up table located in RAM
Float64_RAM	64 Bit Floating Point Implementation with look-up table located in RAM

Block: LookupTable2D



Two dimensional look-up table with selectable number of data points.
Table data must be an array of size DimX*DimY.
If input is out of specified range, output will be cut off (no extrapolation).

Inports	
x	Table index in x direction
y	Table index in y direction

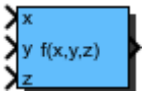
Outports	
Out	Table output

Mask Parameters		
Name	**ID**	**Description**
TableData	1	Look-up table data
DimX	2	Number of data points in x-direction
x_min	3	Minimum input value of x-dimension for look-up table
x_max	4	Maximum input value of x-dimension for look-up table
DimY	5	Number of data points in y-direction
y_min	6	Minimum input value of y-dimension for look-up table
y_max	7	Maximum input value of y-dimension for look-up table

Implementations:

FiP16 16 Bit Fixed Point Implementation
FiP32 32 Bit Fixed Point Implementation
Float32 32 Bit Floating Point Implementation
Float64 64 Bit Floating Point Implementation

Block: LookupTable3D



Three dimensional look-up table with selectable number of data points.
Table data must be an array of size DimX*DimY*DimZ.
If input is out of specified range, output will be cut off (no extrapolation).

Inports	
x	Table index in x direction
y	Table index in y direction
z	Table index in z direction

Outports	
Out	Table output

Mask Parameters		
Name	**ID**	**Description**
TableData	1	Look-up table data
DimX	2	Number of data points in x-direction
x_min	3	Minimum input value of x-dimension for look-up table
x_max	4	Maximum input value of x-dimension for look-up table
DimY	5	Number of data points in y-direction
y_max	6	Maximum input value of y-dimension for look-up table
y_min	7	Minimum input value of y-dimension for look-up table
DimZ	8	Number of data points in z-direction
z_min	9	Minimum input value of z-dimension for look-up table
z_max	10	Maximum input value of z-dimension for look-up table

Implementations:

FiP16 16 Bit Fixed Point Implementation
FiP32 32 Bit Fixed Point Implementation
Float32 32 Bit Floating Point Implementation
Float64 64 Bit Floating Point Implementation

Block: LoopBreaker

Block to break algebraic loops.

Inports	
In	Input In(k)

Outports	
Out	Output Out(k)=In(k-1)

Implementations:

Bool	Boolean Integration
FiP16	16 Bit Fixed Point Implementation
FiP32	32 Bit Fixed Point Implementation
Float32	32 Bit Floating Point Implementation
Float64	64 Bit Floating Point Implementation

Block: ManualSwitch

Toggling between inputs by double-clicking on block.

Inports	
In1	Input #1
In2	Input #2

Outports	
Out	

Mask Parameters		
Name	**ID**	**Description**
Toggle	1	Toggle

Implementations:

Bool	Boolean Implementation
FiP8	8 Bit Fixed Point Implementation
FiP16	16 Bit Fixed Point Implementation
FiP32	32 Bit Fixed Point Implementation
Float32	32 Bit Floating Point Implementation
Float64	64 Bit Floating Point Implementation

Block: Maximum

Outputs the greater value of the two input signals.

Inports	
In1	Input #1
In2	Input #2

Outports	
Out	Maximum of Input #1 and Input #2

Implementations:

FiP16 16 Bit Fixed Point Implementation
FiP32 32 Bit Fixed Point Implementation
Float32 32 Bit Floating Point Implementation
Float64 64 Bit Floating Point Implementation

Block: Minimum

Outputs the lesser value of the two input signals.

Inports	
In1	Input #1
In2	Input #2

Outports	
Out	Minimum of Input #1 and Input #2

Implementations:

FiP16 16 Bit Fixed Point Implementation
FiP32 32 Bit Fixed Point Implementation
Float32 32 Bit Floating Point Implementation
Float64 64 Bit Floating Point Implementation

Block: Not

Logical inverter block.

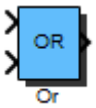
Inports	
In	

Outports	
Out	

Implementations:

Bool Boolean Implementation

Block: Or



Logical OR block.

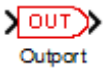
Inports	
In1	
In2	

Outports	
Out	

Implementations:

Bool Boolean Implementation

Block: Outport

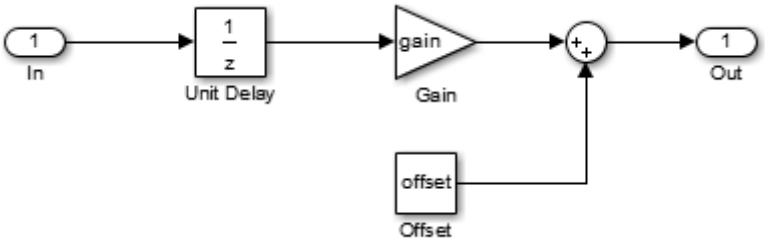


Outports	
OUT	Signal to frame program

Mask Parameters	
ts_fact	Multiplication factor of base sampling time (in integer format)
Gain	Gain value used in simulation
Offset	Offset value used in simulation

Description:

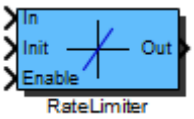
Serves as interface to the frame program. The output of this block is intended for simulation purposes and can be left unconnected if not used. Also the parameters *Gain*, and *Offset* are only used during simulation. The schematic for simulation can be seen in the figure below. The Unit Delay block is only used during simulation and should reflect the time delay caused by a discrete controller.
Note: Currently, *Gain* and *Offset* parameters are only available in Matlab/Simulink.



Data Types:

bool	Boolean
int8	8 Bit Fixed Point
int16	16 Bit Fixed Point
int32	32 Bit Fixed Point
float32	32 Bit Floating Point
float64	64 Bit Floating Point

Block: RateLimiter



Limitation of rising and falling rate.
Function of Enable:
0: rate limiting disabled, signal is passed through
1: rate limiting enabled, signal is rate limited
0->1: preload of output with value from init input

Inports	
In	
Init	Value which is loaded at rising flanke of enable signal
Enable	Enable == 0: Deactivation of block; Out is set to In. Enable != 0: Activation of block; Out is rate limited. Enable 0->1: Preloading of output; Out is set to value of Init input

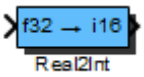
Outports	
Out	

Mask Parameters		
Name	**ID**	**Description**
Tr	1	Rising time in seconds. Slew rate will be 1/Tr
Tf	2	Falling time in seconds. Slew rate will be 1/Tf
ts_fact	3	Multiplication factor of base sampling time (in integer format)

Implementations:

FiP16 16 Bit Fixed Point Implementation
FiP32 32 Bit Fixed Point Implementation
Float32 32 Bit Floating Point Implementation
Float64 64 Bit Floating Point Implementation

Block: Real2Int



Conversion block from real (floating point) datatypes to integer (fixed point) datatypes.
Out = In / Scale

Inports	
In	Real input

Outports	
Out	Integer output

Mask Parameters		
Name	**ID**	**Description**
Scale	1	Scaling factor from real to integer

Implementations:

Float32_FiP8	32 Floating Point to 8 Bit Fixed Point Implementation
Float32_FiP16	32 Floating Point to 16 Bit Fixed Point Implementation
Float32_FiP32	32 Floating Point to 32 Bit Fixed Point Implementation
Float64_FiP8	64 Floating Point to 8 Bit Fixed Point Implementation
Float64_FiP16	64 Floating Point to 16 Bit Fixed Point Implementation
Float64_FiP32	64 Floating Point to 32 Bit Fixed Point Implementation
Float32_Bool	32 Floating Point to Boolean Implementation
Float64_Bool	64 Floating Point to Boolean Implementation

Block: Saturation

Saturation of output to adjustable upper and lower limit.

Inports	
In	Input

Outports	
Out	Limited output

Mask Parameters		
Name	**ID**	**Description**
max	1	Upper Limit
min	2	Lower Limit

Implementations:

FiP16 16 Bit Fixed Point Implementation
FiP32 32 Bit Fixed Point Implementation
Float32 32 Bit Floating Point Implementation
Float64 64 Bit Floating Point Implementation

Block: SaveSignal

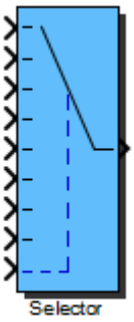
Makes the incoming signal accessible for reading with parameter numbers.

Inports	
In	Input signal to be saved

Implementations:

Bool	Boolean Implementation
FiP8	8 Bit Fixed Point Implementation
FiP16	16 Bit Fixed Point Implementation
FiP32	32 Bit Fixed Point Implementation
Float32	32 Bit Floating Point Implementation
Float64	64 Bit Floating Point Implementation

Block: Selector



Passing through of input signal selected by the select inport:
Select = 0 (DSP): Out = In0
Select = 1 (DSP): Out = In1
...
Select = 7 (DSP): Out = In7

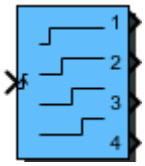
Inports	
In0	Input #0
In1	Input #1
In2	Input #2
In3	Input #3
In4	Input #4
In5	Input #5
In6	Input #6
In7	Input #7
Select	Input select

Outports	
Out	Selected input signal

Implementations:

FiP8	8 Bit Fixed Point Implementation
FiP16	16 Bit Fixed Point Implementation
FiP32	32 Bit Fixed Point Implementation
Float32	32 Bit Floating Point Implementation
Float64	64 Bit Floating Point Implementation

Block: Sequencer



Generation of time delayed (enable) sequence.

Inports	
Start	Start signal. Rising flank triggers sequence

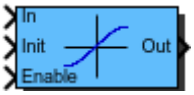
Outports	
Out1	Output #1
Out2	Output #2
Out3	Output #3
Out4	Output #4

Mask Parameters		
Name	**ID**	**Description**
Delay1	1	Time delay for output 1
Delay2	2	Time delay for output 2
Delay3	3	Time delay for output 3
Delay4	4	Time delay for output 4
ts_fact	5	Multiplication factor of base sampling time (in integer format)

Implementations:

Bool	Boolean Implementation
FiP8	8 Bit Fixed Point Implementation
FiP16	16 Bit Fixed Point Implementation
FiP32	32 Bit Fixed Point Implementation
Float32	32 Bit Floating Point Implementation
Float64	64 Bit Floating Point Implementation

Block: Sin2Limiter



Limitation of rising and falling rate with sin^2 characteristic.
Note: A running limitation process can not be interrupted!
Function of Enable:
0: rate limiting disabled, signal is set to zero
1: rate limiting enabled, signal is rate limited
0->1: preload of output with value from init input

Inports	
In	
Init	Value which is loaded at rising flanke of enable signal
Enable	Enable == 0: Deactivation of block; Out is set to zero. Enable != 0: Activation of block; Out is rate limited. Enable 0->1: Preloading of output; Out is set to value of Init input

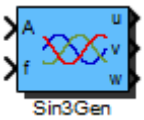
Outports	
Out	

Mask Parameters		
Name	**ID**	**Description**
Tr	1	Rising time in seconds. Slew rate will be 1/Tr
Tf	2	Falling time in seconds. Slew rate will be 1/Tf
ts_fact	3	Multiplication factor of base sampling time (in integer format)

Implementations:

FiP16 16 Bit Fixed Point Implementation
FiP32 32 Bit Fixed Point Implementation
Float32 32 Bit Floating Point Implementation
Float64 64 Bit Floating Point Implementation

Block: Sin3Gen



Generation of a 3 sine waves with amplitude (A) and frequency (f).

Inports	
A	Amplitude
f	Frequency

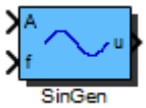
Outports	
u	Sine wave output phase u
v	Sine wave output phase v
w	Sine wave output phase w

Mask Parameters		
Name	**ID**	**Description**
fmax	1	Maximum Frequency in Hz
Offset	2	Offset
ts_fact	3	Multiplication factor of base sampling time (in integer format)

Implementations:

FiP16	16 Bit Fixed Point Implementation
FiP32	32 Bit Fixed Point Implementation
Float32	32 Bit Floating Point Implementation
Float64	64 Bit Floating Point Implementation

Block: SinGen



Generation of a sine wave with amplitude (A) and frequency (f).

Inports	
A	Amplitude
f	Frequency

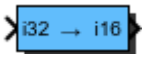
Outports	
u	Sine wave output

Mask Parameters		
Name	**ID**	**Description**
fmax	1	Maximum Frequency in Hz
Offset	2	Offset
Phase	3	Phase [-Pi..Pi]
ts_fact	4	Multiplication factor of base sampling time (in integer format)

Implementations:

FiP16 16 Bit Fixed Point Implementation
FiP32 32 Bit Fixed Point Implementation
Float32 32 Bit Floating Point Implementation
Float64 64 Bit Floating Point Implementation

Block: TypeConv



Data Type Conversion

Inports	
In	

Outports	
Out	

Implementations:

FiP8_16	8 to 16 Bit Fixed Point Implementation
FiP8_32	8 to 32 Bit Fixed Point Implementation
FiP16_8	16 to 8 Bit Fixed Point Implementation
FiP16_32	16 to 32 Bit Fixed Point Implementation
FiP32_8	32 to 8 Bit Fixed Point Implementation
FiP32_16	32 to 16 Bit Fixed Point Implementation
Bool_FiP16	Boolean to 16 Bit Fixed Point Implementation
Bool_FiP32	Boolean to 32 Bit Fixed Point Implementation
FiP16_Bool	16 Bit Fixed Point to Boolean Implementation
FiP32_Bool	32 Bit Fixed Point to Boolean Implementation

Block: uConstant

Constant value.

Outports	
Out	Constant output

Mask Parameters		
Name	**ID**	**Description**
Value	1	Constant factor

Implementations:

Bool	Boolean Integration
Int8	8 Bit Integer Implementation
Int16	16 Bit Integer Implementation
Int32	32 Bit Integer Implementation
FiP8	8 Bit Fixed Point Implementation
FiP16	16 Bit Fixed Point Implementation
FiP32	32 Bit Fixed Point Implementation
Float32	32 Bit Floating Point Implementation
Float64	64 Bit Floating Point Implementation

Block: uGain

Amplification of input by gain factor with output wrapping.

Inports	
In	Input

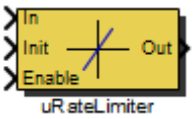
Outports	
Out	Amplified input

Mask Parameters		
Name	**ID**	**Description**
Gain	1	Gain factor in floating point format

Implementations:

FiP8	8 Bit Fixed Point Implementation
FiP16	16 Bit Fixed Point Implementation
FiP32	32 Bit Fixed Point Implementation
Float32	32 Bit Floating Point Implementation
Float64	64 Bit Floating Point Implementation

Block: uRateLimiter



Limitation of rising and falling rate.
Function of Enable:
0: rate limiting disabled, signal is passed through
1: rate limiting enabled, signal is rate limited
0->1: preload of output with value from init input

Inports	
In	
Init	Value which is loaded at rising flanke of enable signal
Enable	Enable == 0: Deactivation of block; Out is set to In. Enable != 0: Activation of block; Out is rate limited. Enable 0->1: Preloading of output; Out is set to value of Init input

Outports	
Out	

Mask Parameters		
Name	**ID**	**Description**
Tr	1	Rising time in seconds. Slew rate will be 1/Tr
Tf	2	Falling time in seconds. Slew rate will be 1/Tf
ts_fact	3	Multiplication factor of base sampling time (in integer format)

Implementations:

FiP16	16 Bit Fixed Point Implementation
FiP32	32 Bit Fixed Point Implementation
Float32	32 Bit Floating Point Implementation
Float64	64 Bit Floating Point Implementation

Block: uSaveSignal

Makes the incoming signal accessible for reading with parameter numbers.

Inports	
In	Input signal to be saved

Implementations:

Bool	Boolean Implementation
FiP8	8 Bit Fixed Point Implementation
FiP16	16 Bit Fixed Point Implementation
FiP32	32 Bit Fixed Point Implementation
Float32	32 Bit Floating Point Implementation
Float64	64 Bit Floating Point Implementation

Block: Xor

Logical XOR block.

Inports	
In1	
In2	

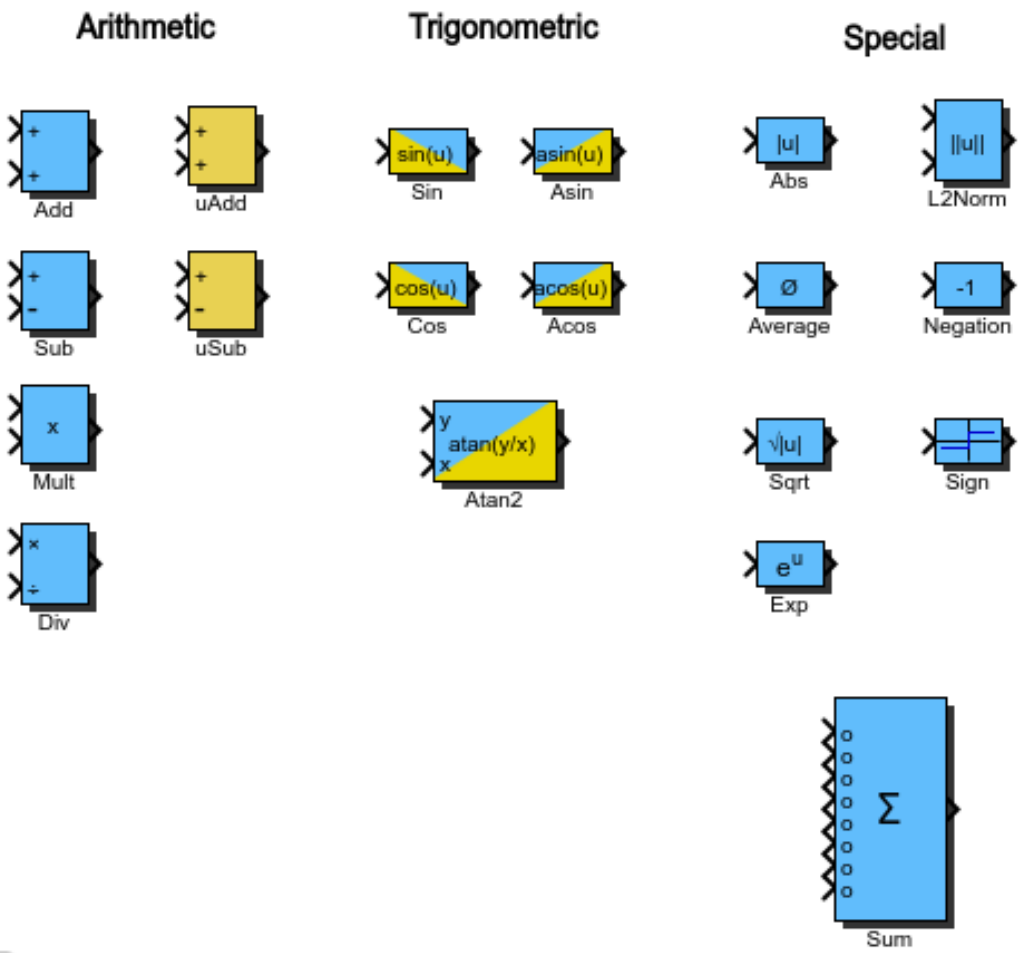
Outports	
Out	

Implementations:

Bool Boolean Implementation

27 Math

This library contains blocks for computation of common mathematical functions.



Block: Abs

Calculation of absolute value of input.

Inports	
In	Input u

Outports	
Out	Absolute value of u

Implementations:

FiP8	8 Bit Fixed Point Implementation
FiP16	16 Bit Fixed Point Implementation
FiP32	32 Bit Fixed Point Implementation
Float32	32 Bit Floating Point Implementation
Float64	64 Bit Floating Point Implementation

Block: Acos

Arccosine computation of input value.

Inports	
In	Input u

Outports	
Out	Result of sin(u)

Implementations:

FiP8	8 Bit Fixed Point Implementation
FiP16	16 Bit Fixed Point Implementation
FiP32	32 Bit Fixed Point Implementation
Float32	32 Bit Floating Point Implementation
Float64	64 Bit Floating Point Implementation

Block: Add

Addition of input 1 and input 2.

Inports	
In1	Addend 1
In2	Addend 2

Outports	
Out	Sum

Implementations:

FiP8	8 Bit Fixed Point Implementation
FiP16	16 Bit Fixed Point Implementation
FiP32	32 Bit Fixed Point Implementation
Float32	32 Bit Floating Point Implementation
Float64	64 Bit Floating Point Implementation

Block: Asin

Arcsine computation of input value.

Inports	
In	Input u

Outports	
Out	Result of sin(u)

Implementations:

FiP8	8 Bit Fixed Point Implementation
FiP16	16 Bit Fixed Point Implementation
FiP32	32 Bit Fixed Point Implementation
Float32	32 Bit Floating Point Implementation
Float64	64 Bit Floating Point Implementation

Block: Atan2

Computation of the angle between the inputs x and y.

Inports	
y	
x	

Outports	
Out	Result of atan2(y/x)

Implementations:

FiP8	8 Bit Fixed Point Implementation
FiP16	16 Bit Fixed Point Implementation
FiP32	32 Bit Fixed Point Implementation
Float32	32 Bit Floating Point Implementation
Float64	64 Bit Floating Point Implementation

Block: Average

Calculation of moving average value over n numbers.

Inports	
In	Input value

Outports	
Out	Averaged value

Mask Parameters		
Name	**ID**	**Description**
n	1	Number of points to be averaged over
ts_fact	2	Multiplication factor of base sampling time (in integer format)

Implementations:

FiP8 8 Bit Fixed Point Implementation
FiP16 16 Bit Fixed Point Implementation
FiP32 32 Bit Fixed Point Implementation
Float32 32 Bit Floating Point Implementation
Float64 64 Bit Floating Point Implementation

Block: Cos

Cosine computation of input value.

Inports	
In	Input u

Outports	
Out	Result of cos(u)

Implementations:

FiP8	8 Bit Fixed Point Implementation
FiP16	16 Bit Fixed Point Implementation
FiP32	32 Bit Fixed Point Implementation
Float32	32 Bit Floating Point Implementation
Float64	64 Bit Floating Point Implementation

Block: Div

Division of input Num by input Den.

Inports	
Num	Dividend (Numerator)
Den	Divisor (Denominator)

Outports	
Out	Quotient

Implementations:

FiP8	8 Bit Fixed Point Implementation
FiP16	16 Bit Fixed Point Implementation
FiP32	32 Bit Fixed Point Implementation
Float32	32 Bit Floating Point Implementation
Float64	64 Bit Floating Point Implementation

Block: Exp

Computation of the exponential of the input.

Inports	
In	Input u

Outports	
Out	Result of exp(u)

Implementations:

FiP8	8 Bit Fixed Point Implementation
FiP16	16 Bit Fixed Point Implementation
FiP32	32 Bit Fixed Point Implementation
Float32	32 Bit Floating Point Implementation
Float64	64 Bit Floating Point Implementation

Block: L2Norm

Calculation of L2-norm (euclidean norm).

Inports	
u1	Input u1
u2	Input u2

Outports	
Out	Euclidean norm of u1 and u2

Implementations:

FiP8	8 Bit Fixed Point Implementation
FiP16	16 Bit Fixed Point Implementation
FiP32	32 Bit Fixed Point Implementation
Float32	32 Bit Floating Point Implementation
Float64	64 Bit Floating Point Implementation

Block: Mult

Multiplication of input 1 with input 2.

Inports	
In1	Multiplicand 1
In2	Multiplicand 2

Outports	
Out	Product

Implementations:

FiP8	8 Bit Fixed Point Implementation
FiP16	16 Bit Fixed Point Implementation
FiP32	32 Bit Fixed Point Implementation
Float32	32 Bit Floating Point Implementation
Float64	64 Bit Floating Point Implementation

Block: Negation

Negation of input signal.

Inports	
In	Input

Outports	
Out	Negated input value

Implementations:

FiP8	8 Bit Fixed Point Implementation
FiP16	16 Bit Fixed Point Implementation
FiP32	32 Bit Fixed Point Implementation
Float32	32 Bit Floating Point Implementation
Float64	64 Bit Floating Point Implementation

Block: Sign

Signum function.

Inports	
In	Input u

Outports	
Out	Value corresponding to sign of u

Implementations:

FiP8	8 Bit Fixed Point Implementation
FiP16	16 Bit Fixed Point Implementation
FiP32	32 Bit Fixed Point Implementation
Float32	32 Bit Floating Point Implementation
Float64	64 Bit Floating Point Implementation

Block: Sin

Sine computation of input value.

Inports	
In	Input u

Outports	
Out	Result of sin(u)

Implementations:

FiP8	8 Bit Fixed Point Implementation
FiP16	16 Bit Fixed Point Implementation
FiP32	32 Bit Fixed Point Implementation
Float32	32 Bit Floating Point Implementation
Float64	64 Bit Floating Point Implementation

Block: Sqrt

Square root computation of absolute input value.

Inports	
In	Input u

Outports	
Out	Result of sqrt(|u|)

Implementations:

FiP8	8 Bit Fixed Point Implementation
FiP16	16 Bit Fixed Point Implementation
FiP32	32 Bit Fixed Point Implementation
Float32	32 Bit Floating Point Implementation
Float64	64 Bit Floating Point Implementation

Block: Sub

Subtraction of input Minus from input Plus.

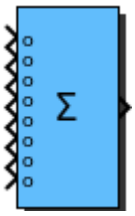
Inports	
Plus	Minuend
Minus	Subtrahend

Outports	
Out	Difference

Implementations:

FiP8	8 Bit Fixed Point Implementation
FiP16	16 Bit Fixed Point Implementation
FiP32	32 Bit Fixed Point Implementation
Float32	32 Bit Floating Point Implementation
Float64	64 Bit Floating Point Implementation

Block: Sum



Sum of inputs:
+ ... Input will be added to result.
- ... Input will be subtracted from result.
0 ... Input will be ignored.

Inports	
In1	Input #1
In2	Input #2
In3	Input #3
In4	Input #4
In5	Input #5
In6	Input #6
In7	Input #7
In8	Input #8

Outports	
Out	Result

Mask Parameters		
Name	**ID**	**Description**
In1	1	Input #1
In2	2	Input #2
In3	3	Input #3
In4	4	Input #4
In5	5	Input #5
In6	6	Input #6
In7	7	Input #7
In8	8	Input #8

Implementations:

FiP8	8 Bit Fixed Point Implementation
FiP16	16 Bit Fixed Point Implementation
FiP32	32 Bit Fixed Point Implementation
Float32	32 Bit Floating Point Implementation
Float64	64 Bit Floating Point Implementation

Block: uAdd

Addition of input 1 and input 2 with output wrapping.

Inports	
In1	Addend 1
In2	Addend 2

Outports	
Out	Sum

Implementations:

FiP8	8 Bit Fixed Point Implementation
FiP16	16 Bit Fixed Point Implementation
FiP32	32 Bit Fixed Point Implementation
Float32	32 Bit Floating Point Implementation
Float64	64 Bit Floating Point Implementation

Block: uSub

Subtraction of input Minus from input Plus with output wrapping.

Inports	
Plus	Minuend
Minus	Subtrahend

Outports	
Out	Difference

Implementations:

FiP8	8 Bit Fixed Point Implementation
FiP16	16 Bit Fixed Point Implementation
FiP32	32 Bit Fixed Point Implementation
Float32	32 Bit Floating Point Implementation
Float64	64 Bit Floating Point Implementation