An Axiometrix Solutions Brand

imc Application Module

Manual

Edition 11 - 2022-07-08

imc Test & Measurement GmbH • Voltastr. 5 • 13355 Berlin • Germany

Disclaimer of liability

Copyright

Open Source Software Licenses

Table of Contents

1 General introduction

1.1 Before you start

Dear user.

1. The software you have obtained, as well as the associated manual are directed toward competent and instructed users. If you notice any discrepancies, we request that you contact our Hotline 4.
2. Updates during software development can cause portions of the manual to become outdated. If you notice any discrepancies, we request that you contact our Hotline.
3. Please contact our Hotline if you find descriptions in the manual which you believe could be misunderstood and thereby lead to personal injury.
4. Read the license agreement. By using the software, you agree to the terms and conditions of the license agreement.

1.2 imc Customer Support / Hotline

If you have problems or questions, please contact our Customer Support/Hotline:

imc Test & Measurement GmbH

Hotline	(Germany):	**+49 30 467090-26**
E-Mail:	hotline@imc-tm.de	
Internet:	https://www.imc-tm.com	

International partners

For our international partners see https://www.imc-tm.com/distributors/.

Tip for ensuring quick processing of your questions:

If you contact us **you would help us**, if you know the **serial number of your devices** and the **version info of the software**. This documentation should also be on hand.

- The device's serial number appears on the nameplate.
- The program version designation is available in the About-Dialog.

Product Improvement and change requests

Please help us to improve our documentation and products:

- Have you found any errors in the software, or would you suggest any changes?
- Would any change to the mechanical structure improve the operation of the device?
- Are there any terms or explanations in the manual or the technical data which are confusing?
- What amendments or enhancements would you suggest?

Our Customer Support 4 will be happy to receive your feedback.

1.3 Legal notices

Quality Management

imc Test & Measurement GmbH holds DIN-EN-ISO-9001 certification since May 1995. You can download the CE Certification, current certificates and information about the imc quality system on our website: https://www.imc-tm.com/quality-assurance/.

imc Warranty

Subject to the general terms and conditions of imc Test & Measurement GmbH.

Liability restrictions

All specifications and notes in this document are subject to applicable standards and regulations, and reflect the state of the art well as accumulated years of knowledge and experience. The contents of this document have been carefully checked for consistency with the hardware and the software systems described. Nevertheless, it is impossible to completely rule out inconsistencies, so that we decline to offer any guarantee of total conformity. We reserve the right to make technical modifications of the systems.

The manufacturer declines any liability for damage arising from:

- failure to comply with the provided documentation,
- inappropriate use of the equipment.

2 imc Application Module

The imc Application Module serves to **integrate measurement channels from "third party" devices or systems** into an imc CRONOS*compact* respectively imc CRONOS*flex* system via standard hardware interfaces.

Examples of possible channel sources include:

- Specialized complex sensors
- "third party" devices
- Bus systems (e.g. fieldbusses)

The standard interfaces supported include, in particular:

- Ethernet
- serial interfaces (RS-232, RS-485, RS-422)

The systems to be integrated are typically user-customized or dedicated devices by third-party manufacturers. The integration is achieved by means of a standard hardware module (APPMOD), which comes with a dedicated processor for which a custom application is programmed. This program is either created by imc on commission or can also be created and implemented by qualified partners or trained users provided with specialized development tools.

This user-specific hardware and software expansion is supported by the device software (imc STUDIO). A special version of the device software is not necessary.

Characteristics:

- encapsulated, custom hardware + software solution, embedded in a standard system
- Standard system with complete software support
- Flexible support by unaltered standard operating software
- Standard hardware component
- Stand-alone, autonomous system environment

3 Functioning

The imc Application Module amounts to a subsystem within the imc CRONOS system family, on which an independent processor runs the user-specific application.

The application communicates with the external devices via the interfaces (Ethernet or serial ports) provided by the hardware module, where the module is able both to receive and transmit data.

Data are exchanged with the measurement device by means of the following mechanisms:

- channels ("FIFO-channels")
- pv-variables ("process vector")
- Display variables

Each of these variables can be defined either as input or output variables.

The channels are managed under the heading of the "Fieldbus channels" and are incorporated into the usual mechanisms such as Measurement Start/Stop and Trigger, just like all other channels. The channel data can be either equidistant, or thy can have a non-equidistant time-stamp format.

What is critical is that incorporation of the channels in the mechanisms is accomplished in a way which generates continuous streaming data. In this case, they are synchronized, and can be equidistant, so that they can also be subjected to operations in imc Online FAMOS.

Definitions

Application archive:	Configuration of an Application module. The configuration file has the file extension ".appmod". Application archives are stored packed as a ZIP-file and can be saved to a folder independently of the experiment. The archive currently in use is saved with the experiment.

4 Workflow

Process of development by imc, selected partners or trained customers

- Setup of development environment: Eclipse
- Programming of the application in C++
- Generation of a finished compilation as a zipped *.appmod file

Utilization by the user

- Use of the zipped *.appmod file with "any" unaltered standard imc STUDIO software
- Subsequent to concrete configuration (experiment) of the device:
  Selection of the specific application by specifying the storage location of *.appmod
  (even a USB-stick, for example)
- Flexible selection among different applications is also possible
- The application's code from *.appmod is embedded in the experiment, so it may not be present at runtime, or when the experiment is loaded.
- Application is either permanently programmed or can be parameterized by the user by means of a specific dialog / menu.

Debugging, Service, Diagnostics

- At runtime, a "Console" is available in the target system via a separate service interface ("SERVICE", RS232, 3.5mm jack), by means of which debugging and logging info can be recorded via the PC in support of diagnostics and application development purposes.
- For this purpose, the development environment (Eclipse) is NOT needed, but only the standard console program on the PC!

5 Setting Up

5.1 Prerequisites

Hardware prerequisites:

- imc CRONOS*compact* (CRC)
- imc CRONOS*flex* (CRFX)
- imc BUSDAQ*flex* (BUSFX)

Software prerequisites:

- imc STUDIO 4.0R1 or higher

Note

The use of process vector variables 7 are necessary for most applications. For the use of pv-variables imc Online FAMOS Professional must be provided in the device.

5.2 Installation Guide: Development Environment

The conditions delineated below apply to installation of the individual development environment components. As a matter of principle, the developer PC should be equipped with a sufficiently up-to-date operating system. Java does not need to be installed. If needed, a JAVA version (included in the product package) can be installed.

The remarks below assume that the compiler, ant, etc. are all installed in the target paths suggested in the respective Setup-programs.

1. Run the Batch-file "Install-IDE.bat" (as of firmware Version (imc DEVICES) 2.11R1). The installation will be in the folder "*C:\imc*\crossgcc\".

Note: The installation files may not be run from the Desktop folder.

2. Application development file for the installed version of firmware (imc DEVICES)

   Install "*Products\imc DEVICES\Tools\imcAppMod\imcAppModDevSetup.exe*" from the installation medium. Change the default path to "*C:\imc\imcAppMod*". The path in which the files are installed will be needed later when developing the modules.

Note

It is possible to install multiple versions; in this case these must be located in different paths:
C:\imc\imcAppMod_2_11R1,
C:\imc\imcAppMod_2_12R1, etc. When the modules are compiled, the respective path must be entered in Build.properties.

5.3 Setting up the development environment and adding the Demo Apps

Proceed as follows to set up the development environment with the complementary demonstration applications:

1. In the Eclipse "Workspace Launcher", set the Workspace and set the view to the Workbench:

- Enter "C:\Test234", for instance, in the Workspace box. Do **NOT** use "c:\imc\imcAppMod" or "c:\imc\imcAppMod\DemoApp"!
- Click on *OK*
- You can activate notifications to Eclipse Community. This will not affect how to run the product.
- Workbench

2. Import Application module-**Reference** projects to Eclipse:

1. Menu *File* > *Import*
2. Select "*General*" > "*Existing Projects into Workspace*" and click on "*Next*".
3. Insert "*C:\imc\imcAppMod*" at "*Select root directory:*" and click on "*Browse*".
4. Click on "*Finish*"

3. Import Application module-**Demo** projects to Eclipse:

1. Menu File > *Import*
2. Select "*General*" > "*Existing Projects into Workspace*" and click on "*Next*"
3. Insert "*C:\imc\imcAppMod\DemoApps*" at "*Select root directory:*" a click on "*Browse*"
4. Click on "*Finish*"

4. If applicable, modify the projects' "*build.properties*":

If applicable, move from the lower tab "*Build*" to the tab "*build.properties*"

1. In the Project Explorer, double-click on "*Kelvimat*" > "***build.properties***"; modify "***build.properties***"; see the example below
2. In the Project Explorer, double-click on Project Explorer "*DisplayApp*" > "***build.properties***" ; modify "***build.properties***"; see the example below
3. (through 7.) Do the same for all projects: Project Explorer "*IENASend12App*", "*FifoReaderDemoApp*", "*RS422DemoRApp*", "*Template*", "*Template_en*"

Example: Modify the entries denoted by bold font:

"build.properties"

```
##############################################################
# This variable must be modified if installation to
# any other folder is performed than
# the default folder: c:\imc\imcAppMod
imcAppModdir      = C:/imc/imcAppMod
##############################################################

ant_dir           = C:/imc/crossgcc/ant-1.9.7
ant_prg           = ${ant_dir}/bin/imc_ant.bat
svn_prg           = C:/Programme/Subversion/bin/svn.exe
nmake_prg         = ${msvs6_dir}/VC98/Bin/Nmake.exe
gmake_prg         = C:/imc/crossgcc//cygwin-2.9.0/bin/make.exe

#crosstools
crossgcc.path     = C:/imc/crossgcc/cygwin-2.9.0/
crossgcc.cygwin   = ${crossgcc.path}/bin
```

5. Save all files:

- Menu *File* > "*Save All*"

6. Build the Application module-zip-archive:

- Menu *Project* > "*Build All*"

7. Use the Windows-Explorer to verify whether the Appmod Zip-Archive has been created. Under each of the paths below, a respective ZIP-file should have been created:

1. C:\imc\imcAppMod\DemoApps\DisplayApp
2. C:\imc\imcAppMod\DemoApps\FifoReaderDemoApp
3. C:\imc\imcAppMod\DemoApps\IENASend12App
4. C:\imc\imcAppMod\DemoApps\KelviMatApp
5. C:\imc\imcAppMod\DemoApps\RS422DemoRApp
6. C:\imc\imcAppMod\DemoApps\Template
7. C:\imc\imcAppMod\DemoApps\Template_en

8. It is recommended to **disable** "*Build Automatically*" and **enable** "*Save before Build*" in the Eclipse settings.

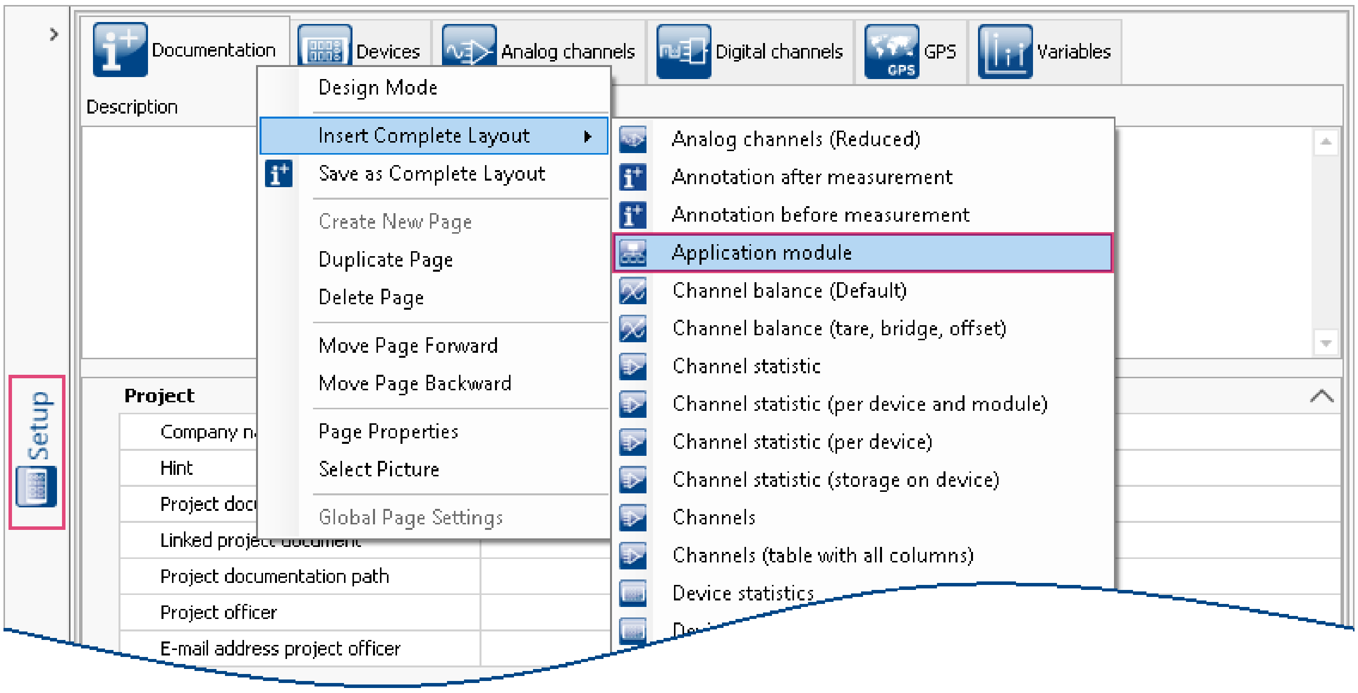
6 imc Application Module Assistant

The method of developing an Application archive is very different from that of using it.

This chapter only describes the user side of imc Application Module. Information on development is provided in the following chapter.

6.1 Assistant in imc STUDIO

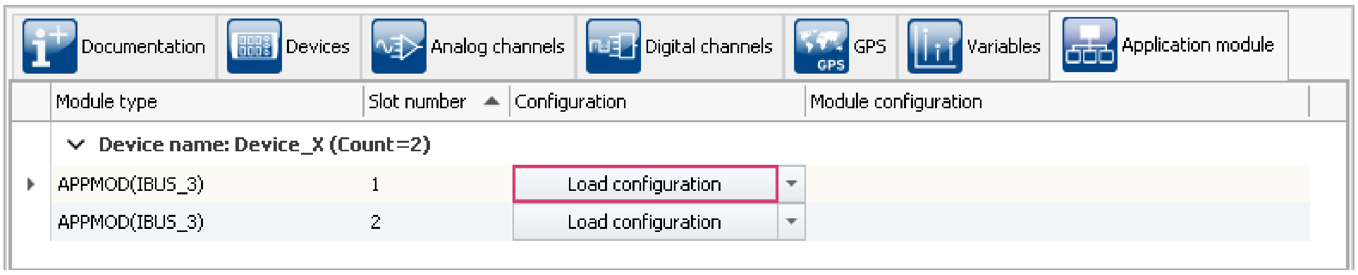
The Assistant for configuring the Application module is called from the *Layout Repository*:



Calling the Assistant

6.1.1 Loading a Applikation

To load a model, click on "*Load configuration*". Then a dialog window for selecting a model appears.



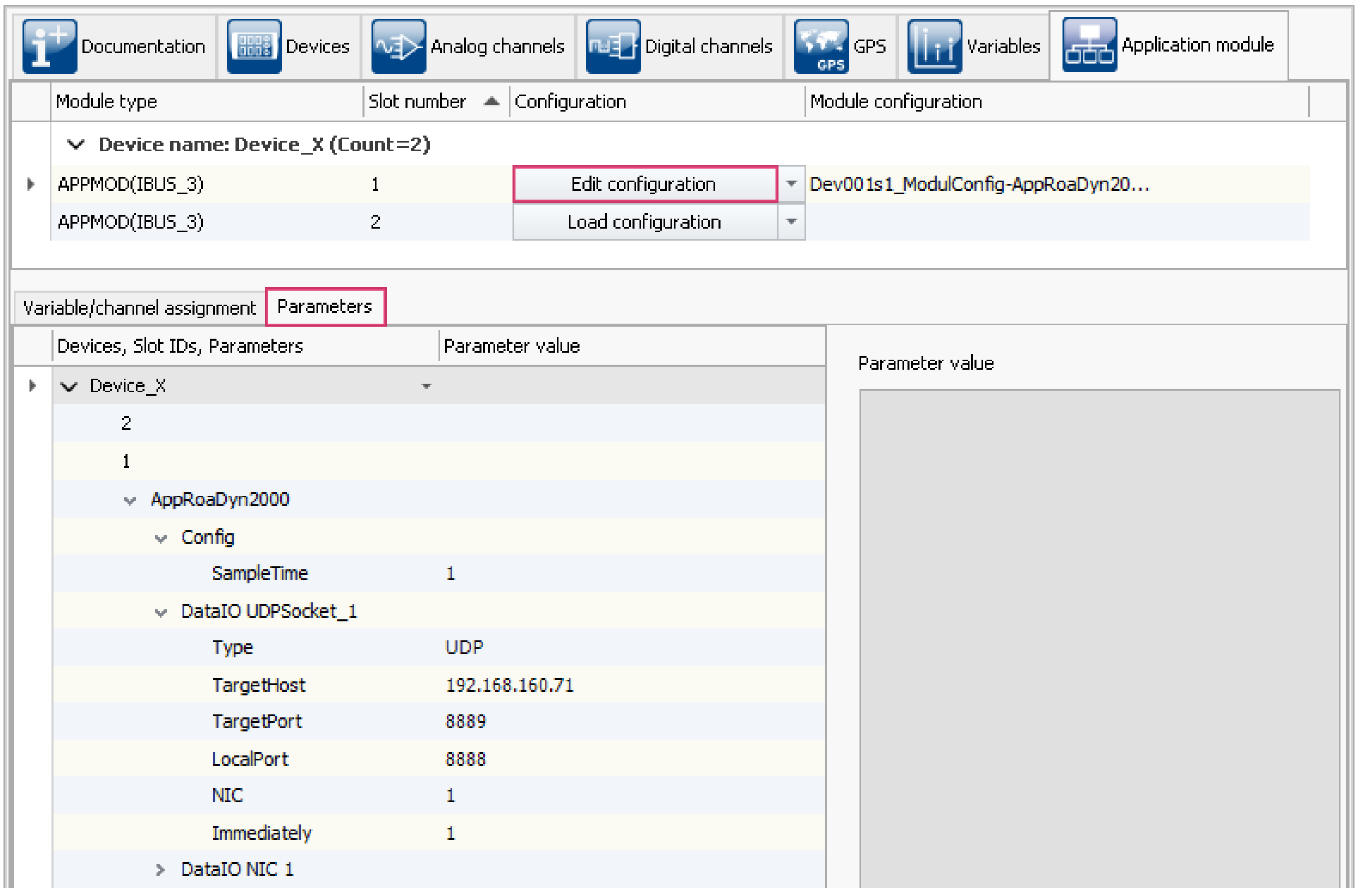
Module type	Slot number	Configuration	Module configuration
Device name: Device_X (Count=2)			
APPMOD(IBUS_3)	1	Load configuration	
APPMOD(IBUS_3)	2	Load configuration	

Load a configuration for the Application module

6.1.2 Editing the Module Configuration

Depending on how the developer created the Application, any configuration options 13 provided can be edited in the following dialog.

Select the tab *Parameters* to open the configuration tree.



Module type	Slot number	Configuration	Module configuration
Device name: Device_X (Count=2)			
APPMOD(IBUS_3)	1	Edit configuration	Dev001s1_ModulConfig-AppRoaDyn20...
APPMOD(IBUS_3)	2	Load configuration	

Devices, Slot IDs, Parameters	Parameter value
Device_X	
2	
1	
AppRoaDyn2000	
Config	
SampleTime	1
DataIO UDPSocket_1	
Type	UDP
TargetHost	192.168.160.71
TargetPort	8889
LocalPort	8888
NIC	1
Immediately	1
DataIO NIC 1	

Parameter value

Configuration window

6.2 Module Configuration

6.2.1 Configuration Possibilities

Important

The developer determines the scope of configuration possibilities. For this reason, a description of each Application by the developer is necessary, which includes descriptions of all parameters and options.

Direct exchange with components of the imc device is enabled by:

- Devices channels (analog, digital, fieldbus, etc.)
- Process vector variables
- Display variables

Note

Note that there can only be either inputs OR outputs. With outputs (-Out), resources are created. With inputs, the system makes note that it is to read from the resources. The direction is determined by the developer.

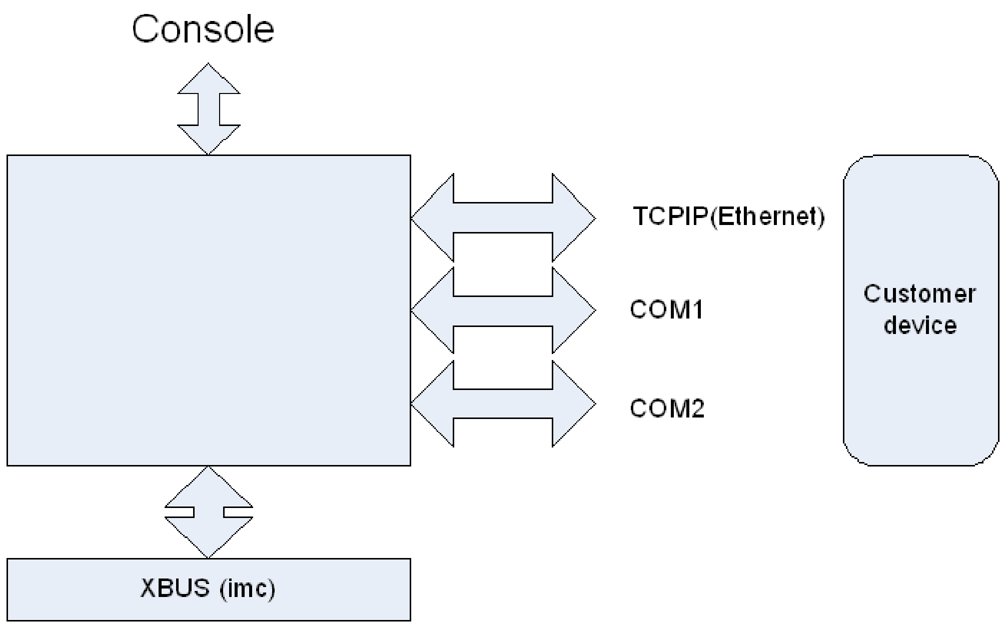
Configuration possibilities	Description
File (in preparation)	File IO: exchange with developer defined file. Parameter: • Name: Name of the file used for exchange with the module. The name is normally specified by the developer.
Serial Interface (ComPort)	Serial IO: Connection settings via the serial interface. Parameters: • Name: Name of the interface concerned. System name ("/dev/ttyPSC1"), or a number (1 for COM1; 2 for COM2). • Bit rate: Bits per second • Data bits: Data bit count • Stop bits: Stop bit count • Flowcontrol • Parity
TCP (outgoing, bidirectional)	TCP IO: Connection settings via TCP. Parameter: • Host (TargetHost): IP of the target configuration • Port (TargetPort): Port of the target configuration • Address: IP of the local network configuration • Netmask: Subnet mask • Gateway: Gateway if required
UDP (bidirectional)	UDP IO: Connection settings via UDP. Parameter: • Port: source port • Host (TargetHost): IP of the target configuration • Address: IP of local network configuration • Netmask: Subnet mask

6.2.2 Entries

Entries	Description
Display variable	Parameter: • Name: Sets the name of the Display variable used.
Process vector	Along with the name, three additional parameters are set: Parameter: • Name: Name of the process vector variable used • Unit: Process vector variable unit • Offset • Factor: Calculation factor • Init: Initialize value
Channel	Parameter: • Name: channel name. If the name is left empty, no IMC resource is created; any IMC resource already created for the channel is deleted along with the name. • comment: Comment box • Y-Axis Unit: Y-axis unit () • SampleTime: Should retain the default value. • SampleMode: (time stamped or equidistant). Should retain the default value.

7 Quick Guide: Developing an imc Application Module

Application module structure:



Depending on how the system is equipped, in the user's version some interfaces may not have terminals

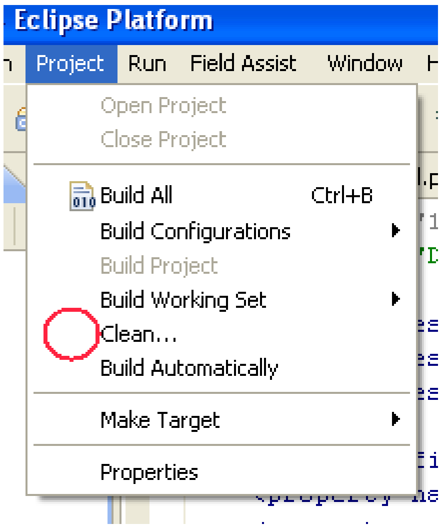
An Application module can use serial interfaces (COM1, COM2) and UDP, TCP (via the existing network interface).

The following conditions must be met:

- Development environment: Eclipse, Toolchains (Compiler, Linker)
  (a corresponding package is available)
- The developer extension matching the firmware version (imc DEVICES) must be installed on the application developer's PC.
  (imcAppModDevSetup.exe 9 on the associated imc STUDIO installation medium)
- The developer's device must be equipped with the same module model as the one which will later be used by the customer.
- The Application module in the developer's device must be equipped with a console which allows application output (display of the trace information) to be viewed.
  The setting on the PC's Hyperterminal, Terraterm, Putty etc. is:
  speed: 115200, *byte size*: 8, *parity*: none, *stop bits*: 1

Proceed as follows to create an Application module:

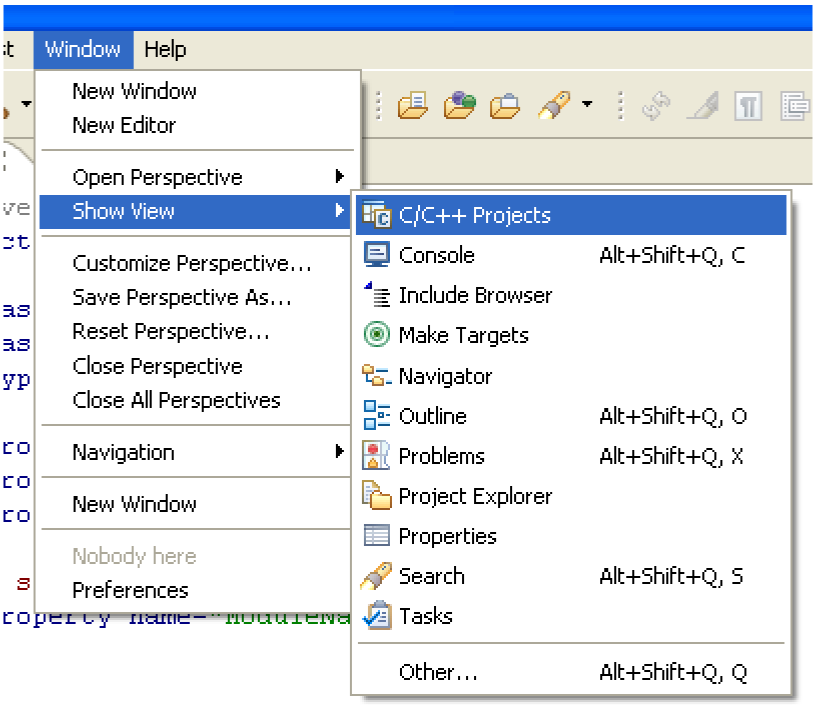
Comment: The sample projects shown are located in the subfolder *DemoApps* in the developer extension installation folder.
(*default: \C:\Program Files\imc\imcAppMod*)



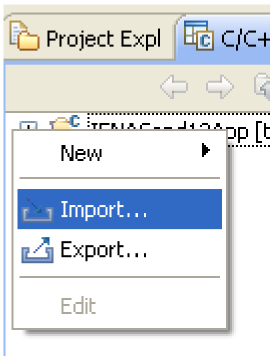
Copy the template project (this is always installed together with the developer extension for the Application module) to a new project folder.

Create a new workspace using Eclipse (the option *Build Automatically* should be deactivated).

Activate the C++ view:



Add the new project using "Import".

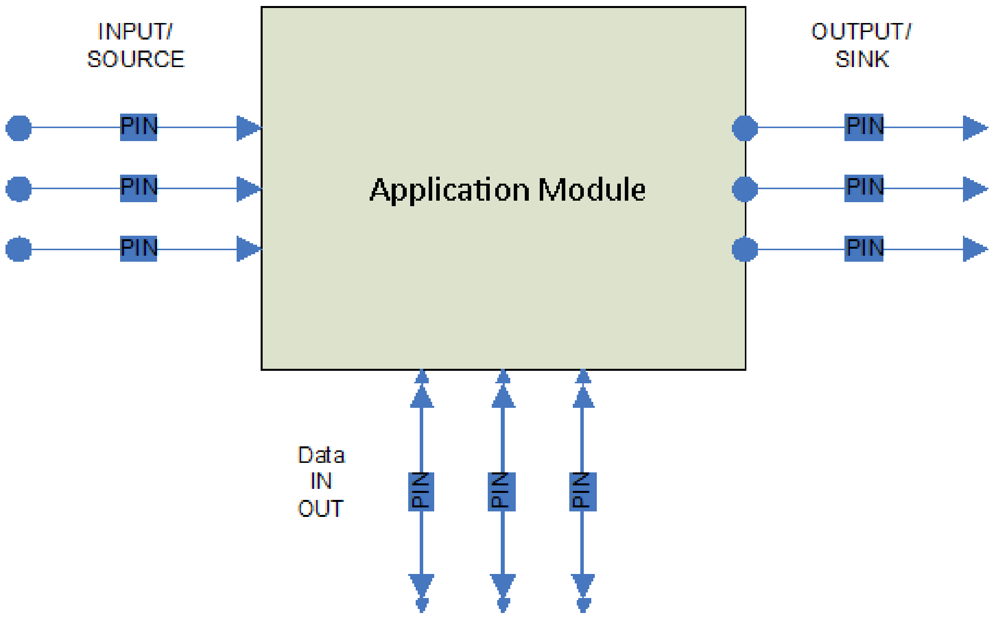


1. It is advised to rename the project after importation. Initially, it is inserted under the name "*Template*". This is done using Eclipse and the rename function.
2. In the file "*build.properties",* the variable "imcAppModdir" must be changed to the developer extension's installation path if it differs from the default setting "C:/Program Files/imc/imcAppMod". (Example see "*Setting up the development environment* 10")
3. Match the file name in the subfolder "*src*" to the name of the target application (here, a short name should be chosen, such as *LightControlApp)*.
   The following files must be renamed:
   - model/Template.pcdcl -> model/LightControlApp.pcdcl
   - model/Template.mpdcl -> model/ LightControlApp.mpdcl
   - src/template.cpp -> src/lightcontrolapp.cpp
   - src/template.h -> src/lightcontrolapp.h
4. Change the file build.xml. Here, the Property(Variable) "ModulName" must be set accordingly:
   <property name="ModuleName" value="*LightControlApp*"/>
5. Rename the class names in the Include-file and C++ source file:
   src/lightcontrolapp.h, src/lightcontrolapp.h
   Template -> *LightControlApp*
6. Create the pin configuration.
   The pin configuration is performed in the renamed "*Template.pcdcl"* file. Here, the process vector variables, channels and display variables which will be used in the module are declared. Additionally, data input and data output channels (COMPort, UDP, etc.) can be defined here, which will later be configured by the user.
7. Setting the module parameters
   The module parameters are set in the "*Template.mpdcl"* file. Here, it is possible to define parameters which can be changed during the measurement's runtime (Tunable Parameter) and for which process vector variables are not suitable. These parameters can then be transferred to the running module by means of the Assistant. Declarations to make include Name, Type and the Initial value. It is possible to make comments in the file (see the sample files on this subject).
8. Implementation of the module
   (Editing, compiling, ...).
9. Create an experiment in imc STUDIO, load it and configure the module.
10. Check function ...
    Start measurements; in case of malfunctioning it may be necessary to restart the device, or at least to use the Assistant to repeat uploading of the program created. If the measurement does not start, find the error in the console.
    Remarks: At present it is only possible to perform error analysis by means of the data imported from the device that will be connected (or a simulation of it), or by text output that is integrated by the application developer.

11. Debug (as in 8), adapt the experiment if necessary and then repeat uploading of the module...
12. If development of the module is completed, then Eclipse is used to create the "Release" version of the Application module.
13. Restart the device.
14. Next, the Application module is configured using the Application module assistant.
15. A test measurement is performed. If it functions as anticipated, the development process of the version of the Application module (application) is completed.
    Comment: Malfunctioning may occur in a variety of ways. For this reason, the application developer must also develop a testing strategy which ensures correct functioning (simulation of the customer devices, test signals, etc).

7.1 Pin Configuration

Principle outline:



Structure of the pin configuration

A "pin" generally comes with a direction:

1. SINK (Output). The data flow in the direction of the "Device"/"PC" from the Application module.
2. SOURCE (Input). The data flow from the device to the module.

Pins should be defined for all resources which the user must configure in the experiment. For example, if it is necessary to select and configure a serial interface, then it must be declared as data input/output. Only then is it possible for the user to make communication parameter settings.

Any imc resources which are used, such as channels, display- and process-vector variables, must be declared via "PINs" in order to be used. The user can decide later what their names are in the experiment, or with which imc resources they are linked (so-called "wiring" of the PINs).

A "Pin" which points to imc resources (PVV, display and channels) can only have one direction (either "SOURCE" or "SINK"), bi-directionality is not possible here).

Note

Process vector variables that are not assigned to a reader in a measurement configuration are not created in the device. If you create pins for process vector variables that are not read by any instance at a later time, the pin is still available. This enables the application to function independently of the evaluation of other modules.

7.1.1 Declaration of the Pin Configuration

```
# IMC AppMod input/output declarations
; comment
; Keywords are not upper-/lower case sensitive
; Values are upper-/lower case sensitive

Pin Name="PVVin" IOType="Source" PinType="PVVar"
    Name="PVVin"
    ValueType=INT32
    Offset=0
    Factor=1
    Unit=" V"
End

Pin Name="PVVout" IOType="Sink" PinType="PVVar"
    Name="PVVout"
    ValueType=INT32
    Offset=0
    Factor=1
    Unit=" V"
    Init="1234"
End

Pin Name="PVVout" IOType="Sink" PinType="PVVar"
    Name="PVVout"
    ValueType=FLOAT
    Offset=0
    Factor=1
    Unit=" V"
    Init="12.34"
End

Pin Name="ToBeSampled" IOType="Source" PinType=Channel
    Name="imcNameOfChannel"
    ValueType="INT16"
    idsampletime=20
    inputmode="TIMESTAMPED"
    inputstate=1
    isuint=false
    unit="C"
    yscalefactor=1
    yscaletype=1
    yoffset=0
    yminvalue=-100
    ymaxvalue=100
    yref1value=-1
    yref2value=40
End
```

```
Pin Name="Kanal_s" IOType="Sink" PinType=Channel
    Name="Kanal_s"
    ValueType="INT16"
    sampletime=10000.0
    inputmode="SAMPLED"
    atoriginal=true
    inputstate=1
    isuint=false
    unit="C"
    yscalefactor=1
    yscaletype=1
    yoffset=0
    yminvalue=-100
    ymaxvalue=100
    yref1value=-1
    yref2value=40
    DecoderBlock="dG90YWwgMzIKNCBkcnd4ci14ci14IDIgd3d3LWRhdGEgd3d3LWRhdGEgNDA5NiBPY3QgIDcgMTE6dG90YWwgMzI
    KNCBkcnd4ci14ci14IDIgd3d3LWRhdGEgd3d3LWRhdGEgNDA5NiBPY3QgIDcgMTE6"
End
```

Pin configuration	Beschreibung
PinName (Pin Name="...")	A pin always has a name by which it is known to the Application module (here: 'Name = "Device"'). Using this name, the application developer can access the resource (pin) in the application (see sample projects).
PinTyp (PinType="....")	The application developer specifies a PinType (PinType examples provided in the sample projects). This determines what resource is concerned. PVVar (Process vector variable), DisplayVariable, Channel or DataIO (files, serial interfaces, etc.) are not possible here. This specification cannot be altered by the user by means of the Assistant.
(IOType="SINK")	For process vector variables, display variables and channels, a direction is determined (IOType: "SOURCE" or "SINK"). With DataIO only, this specification is purely of an informal nature. With this the application developer can indicate to the user the main data flow direction at this terminal (pin). The user is not able to change this information.
Resource name (Name="Resource")	The resource name is a suggestion or indication as to the data type involved.
Pin Parameter	Each pin type comes with a specialized parameter which the user can adapt to the experiment. With the channels, however, the application developer decides how the data are used. This must be implemented with the application. Only the application developer knows which values are useful to the user and thus which ones the user should enter.
Pin parameters for process vector variables	These parameters are only used if a process vector variable is declared a "SINK". Otherwise, these values are determined by the instance of imc STUDIO which creates the process vector variables.
Value type	Value type declares data type of the process vector variables. The available choices are: "INT16", "INT32", "TIFLOAT", "FLOAT", "ASCII" and "BIT16"(only channel pins). **Note 1**: If communication with Online FAMOS is to be conducted by means of process vector variables, then this is only possible with process vector variables of the types TIFLOAT and INT32. Similarly, the Assistant will only list those process vector variables with the SOURCE PVV which match a ***Value type***. **Note 2**: It only makes sense to use value type "ASCII" with time stamped channels. Any arbitrary sequences of data may be written. The user is not limited to zero-terminated C-Strings; i.e. the length of the block is stated. **Note 3**: Value type "BIT16" is used for so-called port channels (channels which transfer 16 individual bits in the word instead of one 16-bit value).
Offset and Factor	*Offset* and *Factor* determine the resolution of the integer data types. For Float variables, these settings are not applicable.

Pin configuration	Beschreibung
Unit	*Unit* is an information element and determines the unit which is to be displayed as a string along with the process vector variables.
Pin parameters for channels	• **comment**: a comment on the resource. This field can be used to transfer information on the channel to the module. • **valuetype**: declares the channels' data type. The available choices are: "INT16", "INT32", "FLOAT", "TIFLOAT", "UINT16", "UINT32" and "DOUBLE64". Remark: valuetype "UINT16" and "UINT32" should, however, actually not be used; if possible use isuint, in order to define an unsigned integer. • **callbackonsampling**: This determines whether the application is called by means of a method when sample values are generated. • **monitoring**: The channel is declared as monitor channel. Only "SINK" channels can be declared as monitor channels and can be set to true or false. • **inputstate**: This determines whether a channel is active or passive. If a channel is set as passive, then it is no longer available as a resource in imc STUDIO. In general, it is not necessary for it to be set to 0 (passive). • **isuint**: This is used to set whether a channel is to work with unsigned integer values. The • **inputmode**: This specifies whether the input consists of time-stamped sample values or of one continuous data flow. For time-stamped values, use "TIMESTAMPED" and for continuously sample values "SAMPLED". • **atoriginal**: This sets the operating type for continuous sampled values. If this parameter is set to true, then the program itself writes to the sampled values to the channel FIFO storage; otherwise, the framework retrieves a sample from a cache and ensures on its own that sufficient data are written to the channel. • **synchronized**: Synchronized is used with the inputmode="SAMPLED" in order to ensure an equidistant data stream if the incoming data stream is not synchronized with the device. • **sampletime and idsampletime**: idsampletime defines the sampling rate by means of a value which looks up the actual sampling rate in a table. Instead of idsampletime, however, it is possible to use sampletime to specify the sampling rate in microseconds (as a floating point number), but the specified value is converted to the corresponding ID value. • **DecoderBlock**: For decoding a channel a decoder block is given. This is then attached to a channel resource. **Note**: The block must be set in "" and in one line. Multiple lines are not possible!

Possible sampling rates and their ID values

ID	Sampling rate	Sampling rate as floating point number
1	"10 µs"	10.0
2	"20 µs"	20.0
3	"50 µs"	50.0
4	"100 µs"	100.0
5	"200 µs"	200.0
6	"500 µs"	500.0
7	"1 ms"	1000.0
8	"2 ms"	2000.0
9	"5 ms"	5000.0
10	"10 ms"	10000.0
11	"20 ms"	20000.0
12	"50 ms"	50000.0
13	"100 ms"	100000.0
14	"200 ms"	200000.0
15	"500 ms"	500000.0
16	"1 s"	1000000.0
17	"2 s"	2000000.0
18	"5 s"	5000000.0
19	"10 s"	10000000.0
20	"20 s"	20000000.0
21	"30 s"	30000000.0
22	"1 min"	60000000.0
23	"2 min"	120000000.0
24	"5 min"	300000000.0
25	"10 min"	600000000.0
26	"20 min"	200000000.0
27	"30 min"	800000000.0
28	"1 h"	3600000000.0

Note

At the present time any sample time settings < 200 µs are rejected by the pin declaration compiler and are replaced with a sampling time of 200 µs.

- **Unit:** *Unit* determines the unit displayed as a string along with the channel data.
- **yoffset:** Offset
- Display scaling - *Yscaletype*, *yscalefactor, yoffset*, *yminvalue*, *ymaxvalue*, *yref1value* and *yref2value* configure the default scaling of the channel's curve window.
  yscaletype determines how the scaling is calculated:
  1. uses yscalefactor without offset.
  2. uses yscalefactor and yoffset.
  3. uses the two reference values yref1value and yref2value.
  4. uses the value limits yminvalue and ymaxvalue.

7.2 Sampling Types

Using the parameter *inputmode*, it is possible to set the sampling types *SAMPLED* and *TIMESTAMPED* (see *inputmode)*. By means of the two parameters *atoriginal* and *synchronized*, there are three possibilities for setting the sampling type *SAMPLED*.

Thus, there are the following ways to use channels (input, meaning the data flow proceeds from outside into the device) with their FIFO buffers.

1. The values arriving from the outside are written along with a time stamp to the channel info.
   The is activated for a channel**PIN** by:
   Inputmode = TIMESTAMPED .
2. Data arriving from the outside, but which are not synchronized to the device. In order to form an equidistant and synchronized data stream, these are not written directly to the FIFO, but instead the incoming sample is saved and extracted and the time which conforms to synchronization.
   The is activated for a channel**PIN** by:
   Inputmode = SAMPLED .
3. Data arriving from the outside batch-wise but which is equidistantly sampled and synchronized with the imc device. These data are then written directly to the FIFO buffer; no further synchronization or similar process is performed.
   The is activated for a channel**PIN** by:
   Inputmode = SAMPLED and *atoriginal=true* .
4. Data arriving from the outside from which an occasional value can be lost are supplied with substitute vales which are then written to the FIFO buffer.
   The is activated for a channel**PIN** by:
   Inputmode = SAMPLED und *synchronizedl=true* .

7.3 Module Parameters

7.3.1 Function of the Module Parameters

The module parameters are an extra part of configuring an application.

The module parameters are divided into three groups:

- Parameters which can be changed during the measurement (so-called "tunable" parameters).
- Parameters which are only significant at the measurement start.
- Parameters which describe the interfaces used.

For the "tunable" parameters there is a function which informs the application of any changes to one or more parameters. The values supplied to a parameter are used as the initial value when the measurement starts.

Parameter groups can be joined up into blocks.

Parameters which configure interfaces are not "tunable".

7.3.2 Declaration of the Module Parameters

```
// decl module parameter
[Blockname]
// Block of Variables, which may be changed during measurement.

// scalar type
IMC_IEEE_FLOAT fValue = 101.55
IMC_DOUBLE dVal1 = 155.0
IMC_INT8 i8Val = -10
IMC_UINT8 ui8Val = 10
IMC_INT16 i16Val = -255
IMC_UINT16 ui16Val = 255
IMC_INT32 i32Test = -100
IMC_UINT32 ui32Test = 100
IMC_STRING test2[255] = "hello"
IMC_BOOL bTest = IMC_TRUE
// test complex value
IMC_COMPLEX complexVal=(1 2)

// vector
IMC_UINT32 vect1 = [[1 2 3 4]]
IMC_UINT32 vect2 = [[1] [2] [3]]
// vector of complex values
IMC_COMPLEX complexVect1 = [[(1 1) (2 2) (3 3)]]
// New: string - vector
// each string-value in the vector may have a maximum of 10 characters
// String values longer than 10 characters are truncated
IMC_STRING sVectTest1[10] = [["value1" "value2"]]
// Each string-value has a maximum length of _MAXPATH
// (generally 255 characters)
IMC_STRING sVectTest2 = [["value1" "value2"]]

// matrix
IMC_UINT32 matrix1 = [[1 2 3] [4 5 6]]
IMC_COMPLEX complexMatrix = [[(1 1) (2 2) (3 3)] [(4 4) (5 5) (6 6)]]

// New: string-matrices
// Each string-value has a maximum length of _MAXPATH
// (generally 255 characters)
IMC_STRING sMatTest1 = [["value1" "value2"] ["value3" "value3"]]
// Each string-value in the matrix may have a maximum 10 characters
// String values which are longer are truncated to 10 characters
IMC_STRING sMatTest2[10] = [ ["value1"] ["value2"] ]

[Config]
// Block of parameters which are not "tunable"
IMC_INT8 i8Val = -10
IMC_UINT8 ui8Val = 10

[DataIO Name1]
Type=COM
Number=1
Bitrate=19200
Databits=8
Stopbits=1
Parity=none
Flowcontrol=none

[DataIO Name2]
Type=FILE
Name=filesname
Immediately=true

[DataIO Name3]
Type=TCP
TargetHost="192.168.160.51"
TargetPort=8080

[DataIO Name4]
Type=UDP
TargetHost="192.168.160.51"
TargetPort=8081
LocalPort=8082
NIC=1
```

```
[DataIO Name5]
Type=TCPServer
LocalPort=12345

[DataIO NIC 1]
IPAddress = "192.168.160.14"
Netmask = "255.255.255.128"
Gateway = ""
```

7.3.3 Declaration of the General Module Parameters

Parameters must always be created under a block. There are a few reserved block names:

- [Config] for the block of not "tunable" parameters.
- All block names which begin with DataIO (e.g. [DataIO Name1234])

All parameters must be commenced with a data type designation.

A parameter declaration appears as follows:

IMC_UINT32 ui32Test = 100

The data type (here: IMC_UINT32), the parameter name (here: ui32test) and the start value (here: 100).

The following data types are available (for scalars, matrices and vectors):

1. IMC_INT8 and IMC_UINT8
2. IMC_INT16 and IMC_UINT16
3. IMC_INT32 and IMC_UINT32
4. IMC_IEEE_FLOAT
5. IMC_BOOL

Note: Within the DataIO-blocks, the types are not specified, since they are determined by the respective parameter.

Access to the parameters is accomplished as the following example illustrates.

Example Access to the parameters

```
IMC_STRING sGroup("IENA");
IMC_STRING sParam("Key");

Node * pNode = GetModulParam(sGroup, sParam);
if (NULL == pNode) {
    TRACE(
        "IENASend12App::OnNewConfiguration(): GetModulParam() failed to fetch <Key>\n"
    );
    break;
}
objParameter& objParamKey = *pNode;
IMC_UINT16  m_IENA_Key = 0;
try {
    m_IENA_Key = objParamKey;
} catch (int err) {
    // bad cast …
}
```

Vector and matrix parameters are a special case:

Example Example of a matrix consisting of string values

```
IMC_STRING sGroup("IENA");
IMC_STRING sParam("szMatrixTest ");
Node* pNode = GetModulParam(sGroup, sParam);
objParameter& ObjParam_szMatrixTest = pNode;
unsigned int rows = ObjParam_szMatrixTest.getDimArrayFirst();
unsigned int cols = ObjParam_szMatrixTest.getDimArraySec();

for (; idxRow < rows; idxRow++) {
    unsigned int idxCol = 0;
    for (; idxCol < cols; idxCol++) {
        const char *szValue = (const char *)ObjParam_szMatrixTest[idxRow][idxCol];
    }
}
```

For the DataIO-blocks, a table (map) is set up, which summarizes the groups accordingly. The IO-interfaces to which they are connected are also available at the reference determined there; for more on this topic, see the sample projects (DemoApps).

7.3.4 Declaration of the DataIO Module Parameters

For the DataIO parameter, the following parameter groups exist:

Parameter	Description
1. Type UDP	a. ***TargetHost*** defines the device with which to communicate. Here, IP-numbers are to be used. The system does not decode names. The parameter is a string. b. ***TargetPort:*** Target port number (1-65535) to which the UDP-packages are to be sent. c. ***LocalPort:*** Port number (1-65535) from which the UDP-packages are to be sent and responses are received. Note: TargetPort and LocalPort can have the same number. d. ***NIC:*** Number of the network interface used. At the present time, there is only one (i.e.: the only permitted number is "1").
2. Type TCP	a. **TargetHost** defines the device with which to communicate. Here, IP-numbers are to be used. The system does not decode names. The parameter is a string. b. **TargetPort**: Target port number (1-65535) of the TCP port to which to connect.
3. Type COM	a. ***Number:*** States the number of the serial interface (1 for COM1, 2 for COM2). At the present time, only two serial interfaces are available. b. ***Bit rate****:* Bit rate to be used for the data transfer speed (9600, 19200, 115200). The standard values are available, see technical data 58. c. ***Databits:*** Databits specifies the number of data bits in a data byte of data transfer (the available choices are 5,6,7 and 8. Typically, 8 is used.) d. ***Stopbits***: Number of stop bits to be used (1 or 2 are available). The stop bit length of 1.5 is not supported at the present time. e. ***Parity****:* Specifies the Parity bit type. This setting is not possible in conjunction with 8 data bits. Here, "even", "odd" and "none" can be used. f. ***Flowcontrol****:* The flow control setting; the choices are either software flow control (the use of Xon/Xoff symbols) or hardware flow control (RTS and CTS lines), or none ("crtscts", "xonxoff" and "none").
4. Type File	a. ***Name:*** Name of the file to be used. Files included in the download of an application are placed in "/tmp" (e.g.: /tmp/MyTestFile.txt).
5. Type NIC	a. ***IPAddress:*** Configuration of the local IP-address of the network interface. b. ***Netmask:*** Configuration of the network interface's net mask. c. ***Gateway:*** If communication is to proceed via gateways, then the first route must be entered here.

Note

As part of the declaration of the UDP and TCP settings, a NIC type with default values is created automatically. If these values are used, it is possible to skip listing them separately in the declaration (IP address: "192.168.160.14", "255.255.255.0", no Gateway("")).

The type NIC always has the name "DataIO NIC 1", where the digits represent the NIC's number. At the present time, only one network interface is available.

For the DataIO types 1-3, there are the additional parameters "**Immediate**" and "**Comment**" available. If "**Immediate**" is set to "true", then the respective DataIO channel is activated upon importing the configuration. If "Immediate" is omitted or explicitly set to "false", then the program is forced to open the channel. "Comment" is simply a comment on the respective DataIO channel. The type NIC always causes configuration of the network interface addressed.

7.4 Setup of an imc Application Module/Application

Module operation always follows the sequence given below:

1. Integration and initialization of the module.
2. Performing the configuration, preparation for measurement with the connected measurement devices, etc.
3. Completing configuration after performing channel configuration.
4. Wait for measurement to begin.
5. Receiving and recording measurement data.
6. Sampling from canal pv-variables and sending to connected device.
7. If necessary, "STOP" message processing if a user has pressed the "STOP" button during measurement.
8. If necessary, processing of the "START" message if a user triggers a new start after measurement has automatically ended.
9. Deinitialization and end.

Ensure that the part of the module created by the application developer is entirely controlled by the Application module framework. In this regard the methods in the module instance (callback function) are accessed according to the operating status. At present it is not possible to start background processes in parallel (Threads/Tasks).

7.4.1 Register Class

In order to carry out registration, a registration class must be created for each Application module in the application framework. A global instance is then needed from this registration. This is necessary so that an Application module in the application framework can be registered for runtime. Only after this has been done can the application framework access the Application module and call the module methods.

In general, it is formulated as follows: ("Template" is replaced by the module class name):

```
class Template_register
{
public:
    Template_register(){
        IdxAppMod::RegisterAppMod(Template::ModCreate);
#ifdef _DEBUG
std::cerr << std::endl << "Template_register(): Registered Template" << std::endl << std::endl;
#endif
    }
    ~Template_register(){};

private:
    Template_register(Template_register &);            // NoImpl
    Template_register & operator= (Template_register &);//NoImpl
};
```

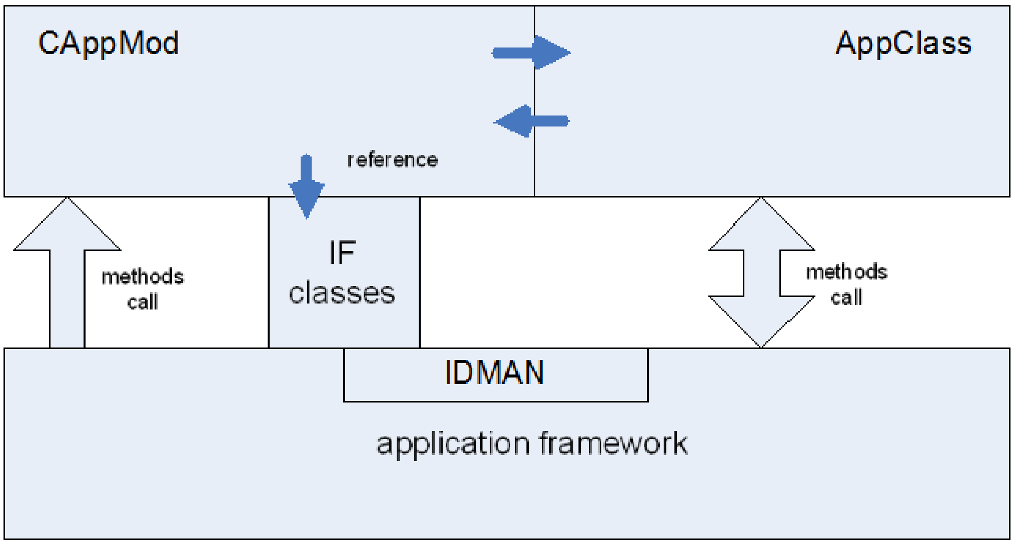
The Application module class is registered with a global instance as follows:

```
Template_register JustToReg;
```

7.4.2 The Class CAppMod

CAppMod is available as the interface class. The application developer adds the application class. The application developer then equips this with its functionality. The CAppMod class serves the framework as an interface to the Application module. In this the class of the Application module must overload several of the CAppMod methods. For those methods that do not need to be overloaded, the CAppMod possesses simple method stubs which ensure smooth running with the rest of the device during operation.

The module class of the Application module is registered in the application framework by means of the registration class mentioned above.



7.4.3 Status Return of Methods

With some exceptions, all methods of a Application module deliver a result value. This might be notification of a configuration error, status notifications, etc. The following return values are currently possible:

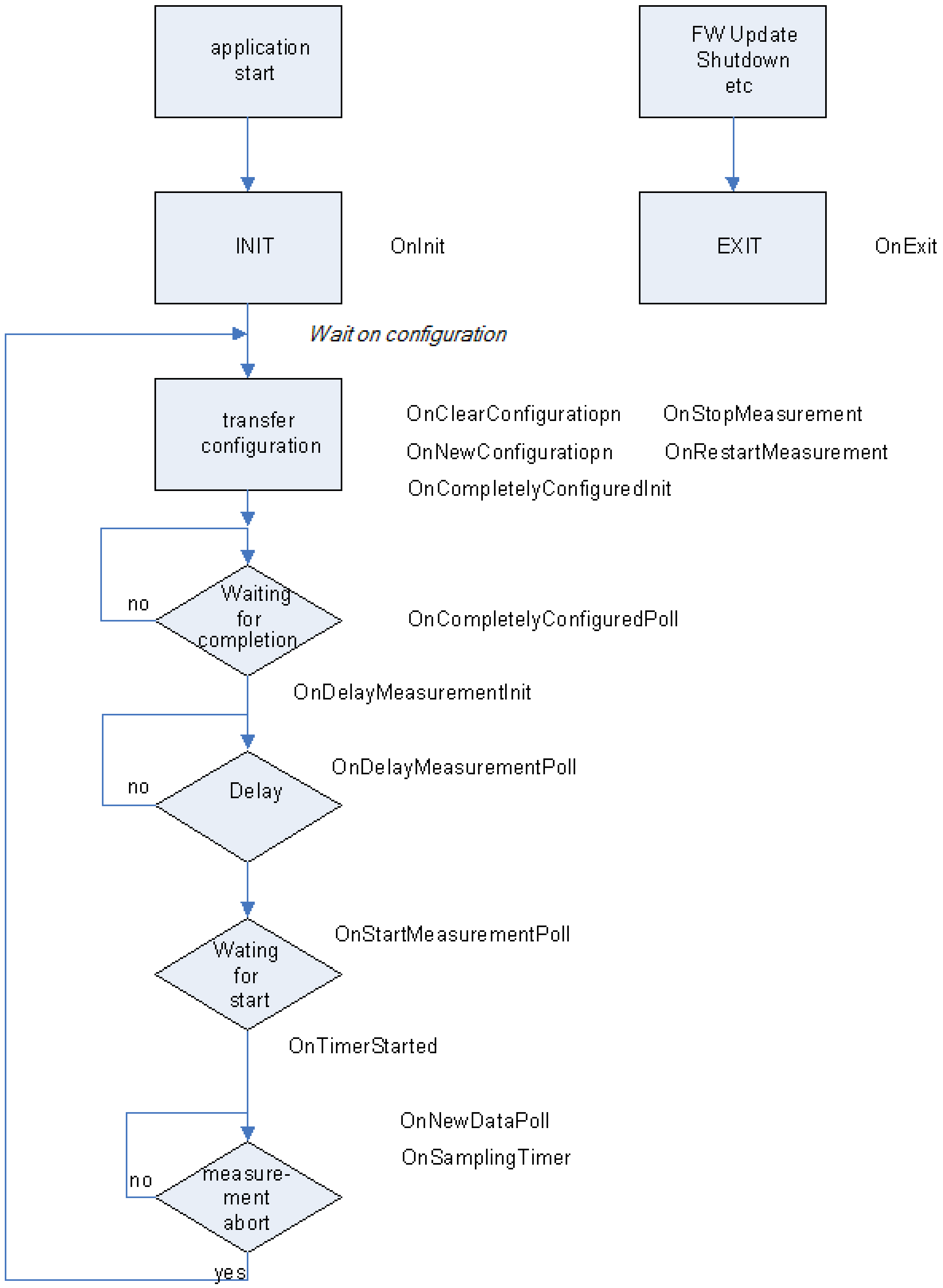
Return values

```
typedef enum _eAppModError
{
    // general
    APP_CONTINUE_POLL       =     1,    ///< Still need to wait
    APP_NO_ERROR            =     0,    ///< No error
    APP_SUCCESS             =     0,    ///< Operation successful.
    APP_FAILURE             = -   1,    ///< generic failure..
    APP_NOT_IMPLEMENTED     = -   2,
    // Config
    APP_BAD_CFG_VERSION     = -  10,    ///< Version of configuration has been detected to be
incompatible !
    APP_BAD_CFG             = -  11,    ///< Configuration is bad !
    // Hardware
    APP_HW_FAILED           = -  20,    ///< General problem with Harware
    APP_HW_INIT_FAILED      = -  21,    ///< Init of Hardware failed
    APP_HW_NOT_INITIALIZED = -  22,    ///< Hardware has not been initialized
    // Application specific results
    APP_MOD_ERROR_BASE      = -  32,    ///< Base of application specific results.
    APP_MOD_ERROR_FIRST     = -  33,    ///< first of application spec. results.
.
.

    APP_MOD_ERROR_LAST      = -  47,    ///< last of application specific results.
    // -
    APP_ERROR_LAST          = -  47     ///< LAST
    } eAppModError;
```

Note

The error code specific to the application is depicted in the general imc error list. *APP_MOD_ERROR_BASE* is assigned to -5432 (depicted 5432) and *APP_MOD_ERROR_LAST* to –5447. By means of the method **SignalOnlineError** (see below) the user can also be notified of an error even during measurement.

7.5 Flowchart of an Application



7.5.1 CAppMod Class Methods

```
IMC_STRING GetApplicationName();
```

GetApplicationName is called in order to obtain the name of the Application module. This method is implemented by the application developer.

```
APP_PIN_MAP * GetPinMap();
```

GetMap is a method that is defined and implemented by the CappMod. GetMap is a utility function for the application developer and delivers a pointer to the loaded Pin configuration. The usual methods for a STL MAP are available. The name of a "PIN name" used in the PIN declaration serves as the key.

```
APP_DATAIO_MAP * GetDataIOMap();
```

GetDataIOMap is a method which is defined and implemented by means of CappMod. GetDataIOMap is a service function for the developer, which returns a pointer to the DataIO parameter imported. The typical STL MAP methods are available. The name of the "DataIO"-parameter which had been used in the module parameter declaration serves as the key (e.g.: "DataIO NAME4").

```
Node * GetModulParam(IMC_STRING & sGroupName, IMC_STRING & sParamName);
```

GetModulParam allows accessing the module parameter database.

GetModulParam is a utility function for the application developer.

```
IMC_BOOL      GetRealtimeModeNeeded( void );
```

GetRealtimeModeNeeded is called by the framework in order to determine whether the operating system must be set up in the corresponding module.

```
eAppModError OnInit(void) = 0;
eAppModError OnExit(void) = 0;
```

OnInit is called when the Application module has been completely instantiated. This takes place one time during the module start. In the examples a reference to the PIN MAP is requested.

OnExit is called when the Application module is ended. It is called a maximum of three times.

These methods serve to carry out the functions at the start and end of the runtime. This allows e.g. a notification to be sent to a connected device.

Both methods are to be implemented by the application developer.

```
eAppModError OnNewConfiguration(void);
```

OnNewConfiguration is called by the framework when a new module configuration has been received and processed. The Application module can now for its part read the configuration and save and process this according to need. The forms chosen here have not been determined. The framework takes care of the deallocation in the resources contained in the PIN MAP.

```
eAppModError OnClearConfiguration(void);
```

OnClearConfiguration is called by the framework in order to show that the current configuration is no longer valid. such references to resources in the PIN MAP must be discarded. This method can and is called any number of times. The application alone must ensure that the resources are not returned twice (e.g. thus: free(pszMeldung);pszMeldung = NULL;.

`void OnCompletelyConfiguredInit();`

OnCompletlyConfiguredInit is called by the framework when all configurations have been loaded and processed and the whole configuration of the system is now in the system itself and this can be read by the module. Because after the module configuration the channel configuration is still given to the module, further action may be necessary.

This method is only to be implemented when preparations must be made for final processing of the entire configuration.

The configuration objects come in the following order:

1. IDMAN configuration (fills the idmanresourcepool)
2. Module configuration (Pin configuration and module parameters as well as their start values and the module executable itself.
3. Channel configuration

Following channel configuration the method pairs are called.

`eAppModError OnCompletelyConfiguredPoll();`

OnCompletlyConfiguredPoll is called by the framework for as long as APP_CONTINUE_POLL is returned. This method must regularly return again so that the framework and, if necessary, the timeout of the system software can be reset. Processing of the channel configuration is only confirmed to the basic system when APP_SUCCESS or an error notification occurs.

`void OnDelayMeasurementInit();`

OnDelayMeasurementInit is called by the framework in order to notify the Application module that work can still be carried out that is necessary for preparation of the measurement task. The necessary initializations are also to be carried out.

`eAppModError OnDelayMeasurementPoll();`

OnDelayMeasurementPoll is exceuted so that the Application module carries out the final work necessary to prepare for measurement. Confirmation of preparedness for measurement is only sent once APP_SUCCESS or an error notification occurs.

This method pair is only to be implemented if it is needed.

`eAppModError OnStartMeasurementPoll();`

OnDelayMeasurementPoll is called while waiting for the actual start of measurement. Here, for example, data can be received from a connected measurement device (e.g. if the device immediately begins to send data while it is establishing a connection).

Comment: possibly initializations must take place with OnDelayMeasurementInit.

`void OnNewDataPoll(const PAI_TIMERTICK & TimeTick) = 0;`

OnNewDataPoll is called at least one time per sampling interval by the framework during measurement. Regular operations are carried out here. Nevertheless, this method should never need longer than the shortest sampling time. This method is to be implemented by the application developer.

`void OnSamplingTimer(void *pObject, const PAI_TIMERTICK &);`

OnSamplingTimer is called by the framework when a time interval has been reached. The time interval is requested by means of AddSamplingTimer. In this case the object (pObject) is given as indentification.

`eAppModError OnStartTriggerDetected(int);`
`eAppModError OnStopTriggerDetected(int, IMC_BOOL);`

OnStopTriggerDetected and OnStartTriggerDetected are called by the framework when a stop or start trigger is recognized. The Application module can carry out an action, e.g. send a signal to a connected device.

Because a channel can start as well as stop a trigger, it is important to check for each channel whether the stop trigger is being caused by the beginning or end. As long as a channel is started and stopped by different triggers, the falling edge is regarded as the stop trigger (the rising edge has already set off the start trigger)

`void OnStopMeasurement(const PAI_TIMERTICK &);`

OnStopMeasurerment is called by the framework when the system user has pushed the stop button during measurement.

`void OnRestartMeasurement(const PAI_TIMERTICK &);`

OnRestartMeasurement is called by the framework when the system user activates the the start button after the stop button.

`eAppModError OnSampleGenerated(FBI_CHANNEL &, const PAI_TIMERTICK &);`

With OnSampleGenerated the user is informed that a sample value is being generated for one channel. The channel in question and the time stamp are generated. The channel and time stamp are transmitted.

`eAppModError OnSyncCreate(FBI_CHANNEL &, const PAI_TIMERTICK &);`

OnSyncCreate informs the application that measurement data has was not written in time for one channel by the Application module and a substitute value has been written. The channel object in question and the time stamp are transmitted.

With these two methods, steps can be taken to counter further measurement data failures.

`eAppModError OnStampedChannelCheck(FBI_CHANNEL &, const PAI_TIMERTICK &);`

OStampedChannelCheck informs the application that for a time-stamped channel, the measurement data have not been written by the application in time. The channel object in question and the time stamp are transmitted.

```
eAppModError SignalChangeOfModuleParam();
eAppModError SignalChangeOfModuleParam(Node &);
eAppModError SignalChangeOfModuleParam(IMC_VECTOR<Node *> &);
```

These methods are called by the framework when the the user has changed the parameters of the module. Notification of individual updates, lists or both of these can be given at the same time.

```
void AddSamplingTimer( const PAI_TIMERTICK& SamplingTimeInTicks, void*
pObject );
```

This method is implemented in CappMod and serves to register a "timer" in the framework. During measurement, the method OnSamplingTimer is then called. pObject serves exclusively the purpose of identification on the part of the module which assigns this value.

Note

This Timer setting applies for the entire duration of the measurement. It is not possible to "un-register". Instead, it happens automatically with the next measurement initialization.

Once a registered SamplingTimer is no longer needed, it can easily be detected by means of the pObject pointer, and in the case of OnSamplingTimer it can be exited without additional processing.

`void RequestExtendedTimeout();`

This method allows the Application module to extend timeout. This can be used, for example, during the configuration operation in OnNewConfiguration when complex operations are necessary with a connected device. Use of this should be avoided whenever possible.

`void SignalOnlineError(eAppModError Error);`

With this the application provides notification during measurement that an error has occurred. It is up to the application developer whether the application continues to write data. The framework itself will continue to call the appropriate methods of the application.

7.6 Pin MAP

Objects of the class CAppPin are filed by name. The CAppPin class is the basic class for the "Pins".

The most important methods of these classes:

Reading and writing values from/to a Pin:

```
/* reader */
virtual IMC_UINT32 readAscii();
virtual IMC_UINT16 readBit1();
virtual IMC_UINT16 readBit16();
virtual IMC_INT16  readInt16();
virtual IMC_INT32  readInt32();
virtual IMC_IEEE_FLOAT readFloat();
virtual IMC_IEEE_FLOAT readTiFloat();
/* writer */
virtual IMC_BOOL writeAscii  ( char *         value );
virtual IMC_BOOL writeBit1   ( IMC_UINT16     value );
virtual IMC_BOOL writeBit16  ( IMC_UINT16     value );
virtual IMC_BOOL writeInt16  ( IMC_INT16      value );
virtual IMC_BOOL writeInt32  ( IMC_INT32      value );
virtual IMC_BOOL writeFloat  ( IMC_IEEE_FLOAT value );
virtual IMC_BOOL writeTiFloat( IMC_IEEE_FLOAT value );
```

Note

At present these methods are supported exclusively by PVVar PINs. Of these, display variables know ***readFloat*** and ***writeFloat***.

The following functions are used for writing data to channels. The timestamp is only for channels with time-stamped values of importance. In all, a dummy is used. The value in this case is ignored.

Writing data to channels

```
/* writer with timestamp */
virtual IMC_BOOL writeBit    (const CAppModTimerTick &, IMC_UINT16);
virtual IMC_BOOL writeInt16  (const CAppModTimerTick &, IMC_INT16);
virtual IMC_BOOL writeInt32  (const CAppModTimerTick &, IMC_INT32);
virtual IMC_BOOL writeFloat  (const CAppModTimerTick &, IMC_IEEE_FLOAT);
virtual IMC_BOOL writeAscii  (const CAppModTimerTick &, const char *, IMC_UNIT16_lentgh);
```

Methods for writing multiple samples in one call:

```
virtual IMC_BOOL writeBit    (const CAppModTimerTick &, CAppModSampleInt16);
virtual IMC_BOOL writeInt16  (const CAppModTimerTick &, CAppModSampleInt16);
virtual IMC_BOOL writeInt32  (const CAppModTimerTick &, CAppModSampleInt32);
virtual IMC_BOOL writeFloat  (const CAppModTimerTick &, CAppModSampleFloat);
```

Note

To identify the "invalid" call, the above methods return false; if a value has been read or written successfully, true is reported.

1. To write multiple samples, proceed as follows: Create instance of the respective CAppModSampleXXX and size (new CAppModSampleXXX (N);)
2. Sampled values by means of
   putSamples (IMC_XXX * pSamples, int at, int n);
   putSample (IMC_XXX Sample, int at);
   in the instance.
3. Write them to the channel (FIFO) using the respective writing method at the pin.
   (PinEntry. writeXXX (Tick, CAppModSampleXXXInstance);)

The following methods are available for reading from ChannelFIFOs:

```
// Get FIFO State
virtual eAppModReaderState  getFIFOReaderState (void);
// FIFO readaccess
virtual eAppModSampleResult readAscii(AppModSampleTSAsciiVector&,IMC_UINT32 maxNum = 1) = 0;
virtual eAppModSampleResult readBit1(CAppModSampleInt16 &, IMC_BOOL exact = IMC_FALSE) = 0;
virtual eAppModSampleResult readBit1(AppModSampleTSInt16Vector&, IMC_UINT32 maxNum = 1) = 0;
virtual eAppModSampleResult readInt16(CAppModSampleInt16 &, IMC_BOOL exact = IMC_FALSE) = 0;
virtual eAppModSampleResult readInt16(AppModSampleTSInt16Vector& ,IMC_UINT32 maxNum = 1) = 0;
virtual eAppModSampleResult readInt32(CAppModSampleInt32 &, IMC_BOOL exact = IMC_FALSE) = 0;
virtual eAppModSampleResult readInt32(AppModSampleTSInt32Vector& ,IMC_UINT32 maxNum = 1) = 0;
virtual eAppModSampleResult readFloat(CAppModSampleFloat &, IMC_BOOL exact = IMC_FALSE) = 0;
virtual eAppModSampleResult readFloat(AppModSampleTSFloatVector& ,IMC_UINT32 maxNum = 1) = 0;
// FIFO Utilities
virtual IMC_INT16   getTriggerTime(CAppModTimerTick & TriggerTime) = 0;
virtual IMC_INT32   getPreTrigger() = 0;
virtual IMC_INT16   getTrigger() = 0;
virtual IMC_INT16   getFull() = 0;
virtual IMC_INT16   getReady() = 0;
virtual IMC_INT16   setReady() = 0;
virtual size_t      reSync() = 0;
```

GetResource(...);

```
virtual void GetResource(PAPP_CHANNEL &);
virtual void GetResource(PAPP_PVV &);
virtual void GetResource(PAPP_DAC &);
virtual void GetResource(void * & id);
```

Delivers by reference a pointer to the respective resource.

1. In the case of a process vector variable, the pointer on the instance of the PinConfig object (the PinConfig object allows access to the configuration data of the process vector variable in question).
2. In the case of a channel the pointer is detected on the instance of the PinConfig object (the PinConfig object allows access to the configuration data of of the channel in question).

GetName();

Delivers the name of the Pin.

ConnectType();

Delivers the Pin type.

GetType()

Delivers the direction (SINK or SOURCE).

GetValueType()

Delivers the data type (INT16, UINT16, etc.).

By means of the Pin object from the **PinMap** the above methods for writing data can be directly written in an imcResource. For writing to the channel resources, those methods should be used that are also rendered with a time stamp. This is only important for time stamped channels. For all other channel operation modes the transferred value is ignored.

It is important that for process variables and for channels only those methods are used that correspond to the data type. Also important here is that during reading of a PVVariable (which is written by an Online FAMOS program) the values are read as *TIFLOAT*. The method in question recalculates the value in an *IEEE Float* with which the application can then use for calculation.

When reading from FIFOS (channels), be sure to note that the method ***getFIFOReaderState*** must be continually called at the respective channel object. This is the only way to ensure that the system treats a FIFO correctly. This applies to all ChannelPins which are registered by means of a reference to an existing resource.

At the present time, for time-stamped channels, only one sample value per call can be read.

The methods for reading from the FiFo return the following result values

```
APPMODSAMPLE_LESS_READ =  1, ///< Not enough data in fifo.
APPMODSAMPLE_NOERROR   =  0, ///< Operation OK
APPMODSAMPLE_INVALID   = -1, ///< invalid Parameter (i.e. pos and amount out of range)
APPMODSAMPLE_NOT_READ  = -2, ///< inform caller , no data read
APPMODSAMPLE_FULL      = -3, ///< inform caller, fifo full
APPMODSAMPLE_NOMEM     = -4, ///< No memory available for sample
APPMODSAMPLE_OVERFLOW  = -5, ///< A FIFO Overrun occured
APPMODSAMPLE_LAST      = -6
```

- ***LESS_READ*** means that the requested number of sample values can not yet be found in the FIFO. Example: an attempt is made to read 100 sample values from the FIFO (instance of the class(es) CappModSample...(100)). *LESS_READ* provides a notification that fewer than 100 sample values have been written to the object.
- ***NOT_READ*** announces that the method read... has been called with *exact=IMC_TRUE*. In this case, fewer than 100 samples were in the FIFO and no sample values were read. *NOT_READ* is also returned in other situations when no sample values have been transferred to the object. If the object was filled completely, NOERROR is returned.
- ***FULL*** is returned if the system returns *FIFO_FULL* upon importing with *exact=IMC_TRUE*. No further data are written to the FIFO buffer in that case. The residual data in the FIFO must be retrieved using *exact=IMC_FALSE*. This is the only way to ensure that the channel's FIFO buffer is completely emptied and that the status is set correctly in the device.
- ***NOMEM*** is returned if insufficient memory is available while reading.
- ***OVERFLOW*** indicates that the FIFO has overflown; more data were written to the FIFO than can be retrieved during the buffer duration. The FIFO must then be synchronized to the channel object by means of reSync. reSync reports the number of sample values lost. In the case of a time-stamped channel, only the number of FIFO words is reported.

GetFIFOReaderState returns the following messages by means of *eAppModReaderState*:

- ***APPMOD_FIFO_ERROR*** : Announces an error state.
- ***APPMOD_FIFO_IDLE:*** The FIFO is in its IDLE state and no further sample values can be read. In this state, it is even NOT PERMITTED to try to retrieve sample values from the FIFO.
- ***APPMOD_FIFO_OVERRUN:*** The FIFO's state following an overflow. This state is only exited once reSync() has been called.
- ***APPMOD_FIFO_DONE***: At the end of the data recording (measurement stop, stop trigger), the system is briefly in the state DONE. This means that all data have been retrieved from the FIFO. The system waits for the FIFO to return to IDLE mode.
- ***APPMOD_FIFO_TRIGGERED***: The FIFO is in the state in which data can be read.

7.7 DataIO MAP

The **DataIO Map** (*GetDataIOMap()*) behaves the same way as the **PinMap**. Like the PinMap, it is an STLMAP. In contrast to the PinMap, it contains instances of the class CappModDataIO. Accordingly it is queried with "find".

Querying the ***DataIOMap*** for a DataIO entry ("DataIO Name3") returns a reference (Pointer) to an instance of the class CappModDataIO; (Note: the DataIOMap is an STL Map).

The CappModDataIO provides the user with the following methods:

`void GetResource(PCComm &, ParamVect * &);`

Returns a pointer to the instance of the associated CComm object, as well as a vector with string pairs.

`void GetResource(PCComm &);`

Returns a pointer to the instance of the associated CComm object.

`const IMC_STRING GetDataIOName();`

Returns the name of the DataIO channel.

`const IMC_STRING GetDataIOType();`

Returns a DataIO channel's type ("UDP", "TCP", etc).

`const IMC_STRING GetDataIOConnect();`

Returns a string describing the DataIO channel's connection.

`const IMC_STRING GetDataIOComment();`

Returns the DataIO channel's comment.

7.7.1 DataIO Class (CCom)

The Ccomm class creates a uniform control for data input and output. This is a factory class. On the one hand communication objects register with it. Using an associated name, CComm recognizes the type and creates an appropriate object. Four types are currently available:

Files, serial interfaces (COMPORT), UDP Socket and TCP Socket.

All methods deliver back a result. Enumeration eCommError describes the possible result values. The following error codes are depicted:

```
COMM_OK          =   0
COMM_NO_SPACE    =  -1
COMM_LESS_SPACE  =  -2
COMM_NO_DATA     =  -3
COMM_LESS_DATA   =  -4
COMM_BAD_NAME    =  -5
COMM_BUSY        =  -6
COMM_INVALID     =  -7
COMM_NO_CHAN     =  -8
COMM_CLOSED      =  -9
COMM_BAD_TYPE    = -10
COMM_READ_FAILED = -11
COMM_GENERAL     = -12
COMM_CANNOT_OPEN = -13
COMM_WRITE_FAILED= -14
COMM_SEEK_FAILED = -15
COMM_SOCKET_FAILED = -16
COMM_BAD_ADDRESS   = -17
COMM_BIND_FAILED   = -18
COMM_CONNECT_FAILED= -19
COMM_READ_TO_LARGE = -20
COMM_READ_TIMEOUT  = -21
```

7.7.2 Description of the CCom Class Methods

`eCommError open();`

Open the file/socket/COMPort. The necessary parameters are given when the object is created. These must be called on PINs if the channel has not been opened using the setting immediate.

```
eCommError reopen(const IMC_STRING &, const ParamVect &);
eCommError reopen(const IMC_STRING &, const IMC_STRING &, const ParamVect&);
```

The file/socket, etc. is opened again with the same object instance (the old channel is closed beforehand). A change in the communication type is in this case possible.

`eCommError close(void);`

Closes a communication channel

```
eCommError read (IMC_UINT32 rsize, void * , IMC_UINT32 & rReadSize);
eCommError read (IMC_UINT32 rsize, void * , IMC_STRING & , IMC_UINT32 & rRead);
```

Both methods read information out of the communication channel without waiting. When the channel cannot deliver the amount of data demanded, COMM_LESS_DATA is reported. Data is never waited for. The sender host is noted in the transmitted string IMC_STRING (only with UDP).

```
eCommError readwithto(IMC_UINT32 rsize, void * pDatabuffer, IMC_UINT32 &, IMC_UINT32
timeout );
eCommError readwithto(IMC_UINT32 rsize, void * pDatabuffer, IMC_STRING &, IMC_UINT32
&, IMC_UINT32 timeout );
```

Two methods for reading using a timeout. Waiting for data takes place for the stipulated period of time. If insufficient data is received before timeout, COMM_READ_TIMEOUT is reported back. For IMC_STRING parameter, the same period applies that has been given for the other "read" methods.

```
eCommError BuffLevel();
```

If the methods for reading from a communication channel encounter insufficient data or are terminated due to timeout, it is possible to request *BuffLevel* how much data could already be read.

```
eCommError write(IMC_UINT32 wsize, void * , IMC_UINT32 & rWritten);
eCommError write(IMC_UINT32 wsize, void * , IMC_STRING & , IMC_UINT32 & rWritten);
```

Methods for writing data to a communication channel.

```
eCommError seek(IMC_UINT32 type, IMC_INT32 offset);
```

Controls the specified position in a file. This method cannot be used on ComPorts and sockets.

```
eCommError flush(void);
```

Empties the temporary buffer of the data channel.

```
CHECK_RESULT check(void);
```

Checks whether reading or writing can be carried out. This eliminates the need to block a reading/writing operation. Furthermore, blocking a reading/writing operation is not allowed as otherwise the Application module framework would no longer work correctly.

`CHECK_RESULT` is a data structure reporting on the current state of the communication channel:

```
typedef struct {
    IMC_BOOL readerready;
    IMC_BOOL writerready;
    IMC_BOOL except;
    eCommError eResult;
}CHECK_RESULT;
```

```
eCommError SetChannelPar(const sParamsValue &);
```

Sets the parameters of the communication channel whenever this is necessary following the creation of an object (e.g. setting an ARP entry in the system). `sParamsValue` is the ICM_STRING pair that respectively sets both the parameter as IMC_STRING and the value as IMC_STRING.

Parameters for COM ports:

- bitrate (50-230400 in common steps. As of firmware version (imc DEVICES) 2.8R6 also 14400 and 28800)
- bytesize (5,6,7,8)
- stopbits(1,2)
- parity("none", "even", "odd")
- flowcontrol ("none", "crtscts", "xonxoff")

Parameters for UDP sockets:

- setarp (IPAdresse:MACAdresse)
- broadcast (on/off)

7.8 Control Block

For advanced access to hardware resources of the module there is a control block class that can be accessed by the application.

At present the following functionalities are available:

- FPGA version request
- FPGA variant request
- Clock basis request for the output clock generator
- Setting the output clock
- Activating and deactivating the output signal unit
- Control of seven output lines

About methods of overloading:

`IMC_BOOL getFPGAVersion(IMC_UINT16 & Version);`
Version request.

`IMC_BOOL getFPGAVariant(IMC_UINT16 & Variant);`
Variant request.

`IMC_BOOL getClockBase(IMC_UINT32 & ClockBase);`
Clock basis request for the output clock generator

`IMC_BOOL setCLOCKRate(IMC_UINT32 ClockRate);`
Setting the output clock synchronously for devices.

`IMC_BOOL setAPPSIGBit(IMC_UINT8 BitNo, IMC_BOOL bOn);`
Setting an output line (setting follows with the rising edge of the clock).

`IMC_BOOL enableAPPSIG();`
Activating the output signal unit

`IMC_BOOL disableAPPSIG();`
Deactivating the output signal unit

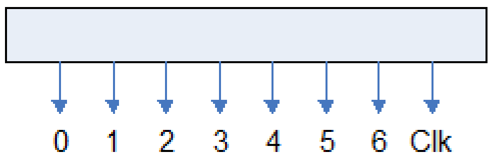
`IMC_BOOL selectRS485(IMC_UINT8 PortNo, IMC_BOOL brs485enable);`
Activating RS485 mode of the serial interface. This is only available using an Application module equipped for RS485.

`IMC_BOOL enableRS485Sender(IMC_UINT8 PortNo, bEnable);`
Activation of the RS485 transmission cable.

Note

If RS485 is only for reading (RS422), it is not necessary to activate the RS485 module. This is only needed for sending. In this case only RS232 bit rates will be available.

7.8.1 Using the Output Unit



If it is intended to use the output unit, the unit must be activated by enableAPPSIG. Only after this has been done will the clock signal be outputted and the control signal given outside. The signals should be set in advance in such a way that they assume the idle position; corresponding pullups and pulldowns connected to the lines must be taken into account.

7.9 SSI-Controller

At this time, the SSI-controller is only available on the PBUS_MPC_1 hardware. If the module you obtained has a different module installed, and you require the SSI-controller, then contact us. There are two SSI-controllers available (specification SSIPort: 1 or 2).

If the functionality is not available, an error message is returned.

7.9.1 Operation of the SSI-controller

The SSI-controller is operated using the following set of control block methods. When processing is successful, then **APP_SUCCESS** is returned.

```
eAppModError selectSSIController(IMC_UINT8 SSIPort);
```

This switches the SSI-controller to the line drivers. Please observe the notes presented below.

```
eAppModError deselectSSIController(IMC_UINT8 SSIPort);
```

This disconnects the SSI-controller from the line drivers. This should be done whenever the SSI-controller is no longer to be used.

```
eAppModError configSSIController(IMC_UINT8 SSIPort,
        IMC_UINT32 sampling rate, IMC_UINT16 SizeOfSample,
        IMC_UINT16 ValidBitsOfSample, IMC_UINT32 interface clock rate,
        IMC_SSI_CLOCKTYPE SSIMode);
```

This is for configuring the SSI-controller.

With SSIPort, the SSI-Controller is selected.

With Sampling Rate, the rate at which a connected sensor's values is sampled is set. Note that this always applies to both SSI-controllers and and also that always the last sampling rate set is used.

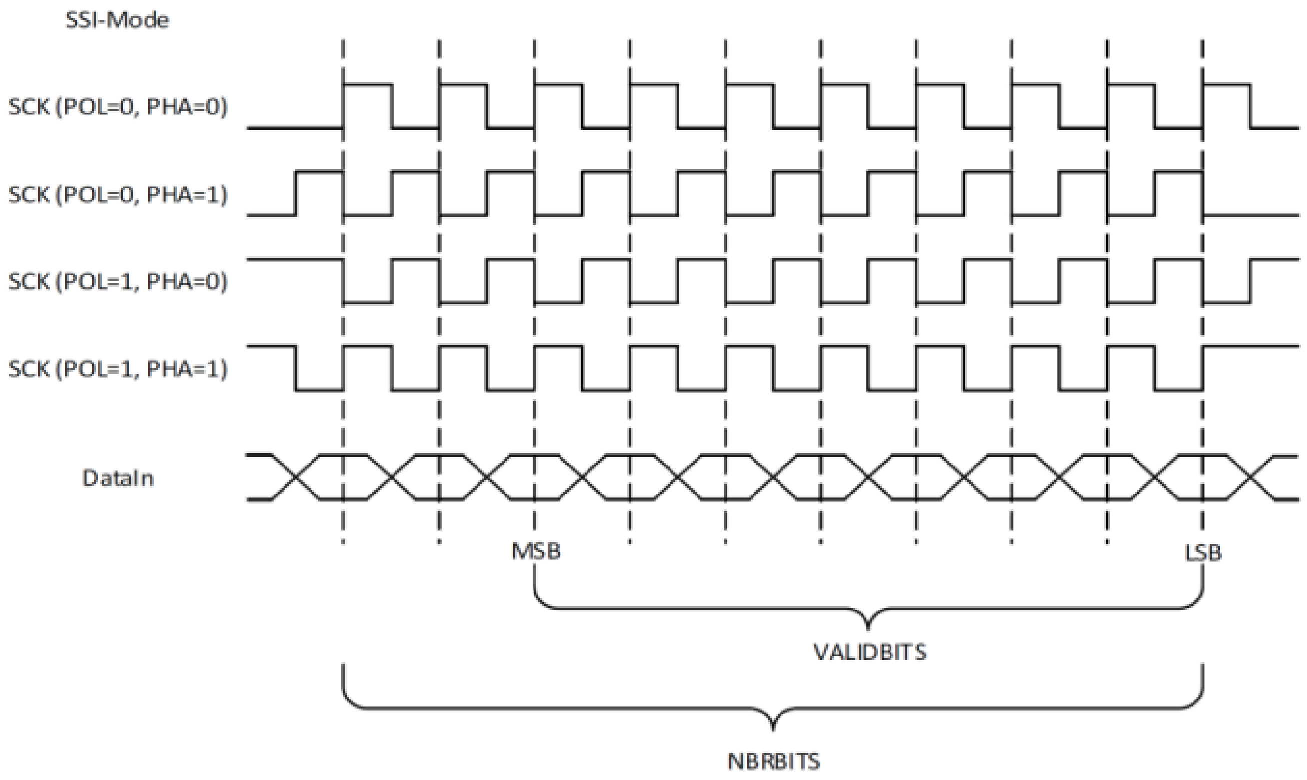
SizeOfSample specifies the number of bits which are to be retrieved from the connected sensor for each sampling process.

ValidBitsOfSample specifies the number of valid bits in a sample. This setting's value may only be less than or equal to SizeOfSample.

Interface Clock Rate sets the bit clock rate at which the sensor is queried.

```
typedef enum tag_IMC_SSI_CLOCKTYPE {
     IMC_SSI_CLOCK_POL0_PHASE0 = 0x0,
     IMC_SSI_CLOCK_POL0_PHASE1 = 0x1,
     IMC_SSI_CLOCK_POL1_PHASE0 = 0x2,
     IMC_SSI_CLOCK_POL1_PHASE1 = 0x3,
} IMC_SSI_CLOCKTYPE;
```

SSIMode specifies when the sensor outputs the data on the data line. See the following table:



```
eAppModError activateSSISender(IMC_UINT8 SSIPort);
```

Here, the SSI-controller is instructed to retrieve samples from a sensor at the interface clock rate. Generally this is done once upon starting the measurement. Note in this case that no sample is generated at the time T0. This method can also be called with the SSIPort at 0, in which case both SSI-controllers are instructed accordingly.

```
eAppModError deactivateSSISender (IMC_UINT8 SSIPort);
```

With this, the SSI-controller is instructed to stop polling the sensor. This method only returns once any sample call in progress has been completed.

```
eAppModError getSSISamples(IMC_UINT8 SSIPort, IMC_UINT16 MaxSamples,
        IMC_SSI_SAMPLE pSampleArray[],
        IMC_UINT16 & NumberOfSamples);
```

Polls the sample values accumulated in the SSI-controller.

SSIPort names the SSI-controller to be queried.

MaxSamples specifies how much space is in the SampleArray to be transferred.

pSampleArray transfers an array into which sample values are to be written.

NumberOfSamples is filled with the number of sample values actually read.

If a FIFO-overflow is detected, the reading of the FIFO is interrupted and an APP_MOD_HWFIFO_OVERRUN is reported. In that case, the NumberOfSamples reflects the number of sample value read up to that time.

7.9.2 Structure of IMC_SSI_SAMPLE

```
typedef union tag_IMC_SSI_SAMPLE {
    IMC_SSI_SAMPLE_BLOCK  Sample;
    IMC_SSI_SAMPLE_BUFFER Buffer;
} IMC_SSI_SAMPLE ;

typedef struct tag_IMC_SSI_SAMPLE_BLOCK {
    IMC_UINT16 DateBlock;
    IMC_UINT32 TimeBlock;
    IMC_UINT32 TenNanoSeconds;
    IMC_UINT32 Sample;
} GNUC_PACKED2 IMC_SSI_SAMPLE_BLOCK;
```

DateBlock and TimeBlock are BCD-encoded

Example

DateBlock(Year,Month): 0x1404,
TimeBlock(Day,Hour,Minute,Second): 0x28112213)

The application itself must mask the "invalid" bits of a sample; the SSI-controller does not handle this.

7.9.3 Note on initialization

Due to the activation of the SSI-controller, the following sequence in the initialization must be adhered to. Please also note that a part of the RS485 methods must be used.

```
CAppMod_CB & rCB = getAppModCB();
rCB.configSSIController(1,20,9,7,500000,IMC_SSI_CLOCK_POL1_PHASE1);
rCB.selectRS485(1,true);
rCB.enableRS485Sender(1,true);
rCB.enableRS485FullDuplex(1,true);
rCB.selectSSIController(1);
```

Note

- It is important for selectSSIControler to be called last.
- If the terminator function present in the output driver is to be used, then the enableRS485Terminator method is to be called as described above.

7.10 Module Parameter

As a principle, module parameters of the application are organized into groups. For the reading of a module parameter by the framework, a reference to the generic data type "node" is first created by the framework method.

"GetModulParam(IMC_STRING& sGroupName, IMC_STRING& sParamName)" and objParameter& objParamKey = *pNode). Once this has been done, the value of the module parameter can be read. A corresponding cast operation is carried out by the assignment operator (e.g. if an appropriate cast operator can't be assigned, an exception is triggered; bad cast).

See the example below for a module parameter within the parameter group "IENNA" with the name "Key" and the data type "IMC_UINT16".

Example Module parameter

```
IMC_STRING sGroup("IENA");
IMC_STRING sParam("Key");
Node * pNode = GetModulParam(sGroup, sParam);
if (NULL == pNode) {
    TRACE(
        "IENASend12App::OnNewConfiguration(): GetModulParam() failed to fetch <Key>\n"
    );
    break;
}
objParameter& objParamKey = *pNode;
IMC_UINT16  m_IENA_Key = 0;
try {
    m_IENA_Key = objParamKey;
} catch (int err) {
    // bad cast …
}
```

Special case – vector and matrix parameter: Example of a matrix consisting of string values

Example **Matrix**

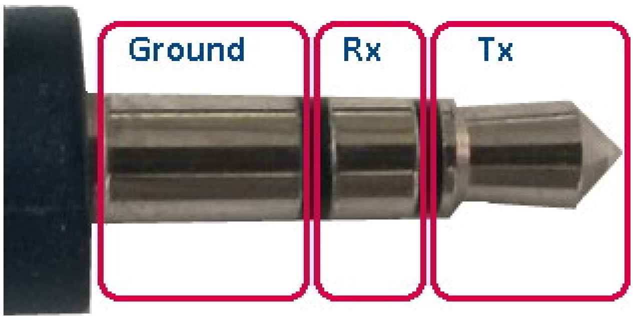
```
    IMC_STRING sGroup("IENA");
    IMC_STRING sParam("szMatrixTest ");
    Node* pNode = GetModulParam(sGroup, sParam);
    objParameter& ObjParam_szMatrixTest = pNode;
    unsigned int rows = ObjParam_szMatrixTest.getDimArrayFirst();
    unsigned int cols = ObjParam_szMatrixTest.getDimArraySec();

    for (; idxRow < rows; idxRow++) {
        unsigned int idxCol = 0;
        for (; idxCol < cols; idxCol++) {
            const char *szValue = (const char *)ObjParam_szMatrixTest[idxRow][idxCol];
        }
    }
```

8 Debug connector

For the output of debug messages from an application, a small 3.5mm jack socket is mounted on the front panel of the Application module. This is connected to a COM port of the PC via a suitable cable (also a USB serial adapter can be used).

By means of a terminal emulator (e. g. Putty), which is set to 115200bps, 8N1, flow control switched off (neither RTS/CTS nor XON/XOFF), the outputs can then be viewed.

If such a line is not available, it can be made according to the following diagram:



wired to female DSUB-9
Pin 5 -> Ground
Pin 2 -> Tx
Pin 3 -> Tx

9 Tutorial

This tutorial presents the procedure for creating an application using the Application module.

A new application is created on the base of a template. In this example, the conversion of a Celsius measurement to Kelvin is demonstrated.

9.1 Installation

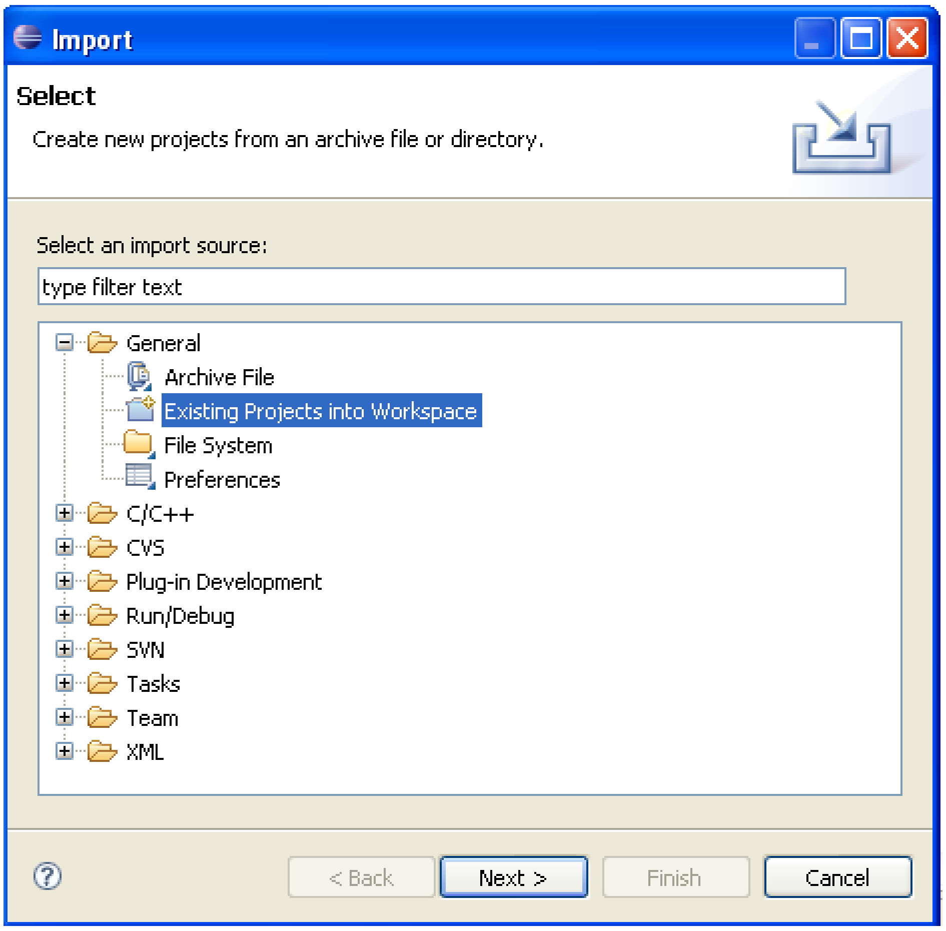
Install the development environment according to these Instructions 9.

Note

For English-language operating systems, you must use an installation folder path which does not contain any spaces.

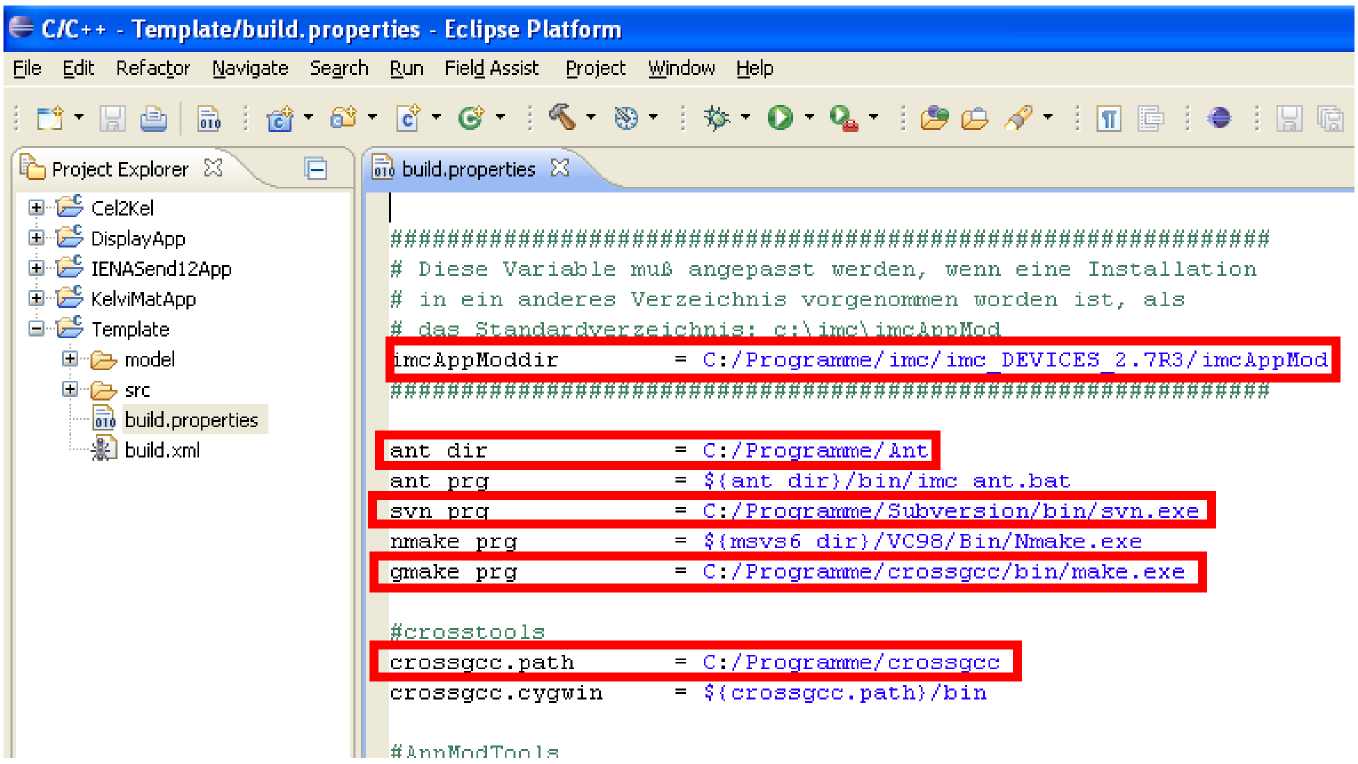
9.2 Opening Projects in Eclipse

- Copy the **sample models** in your Eclipse workspace
- The sample models are found in the folder where you previously installed "*imcAppModDevSetup* 46".
- Start **Eclipse**
- Deactivate "*Build Automatically*". See also here 16.
- Insert the sample projects into your Eclipse workspace.
- To do this, use the Import function -> "Existing Projects into Workspace" and select your workspace



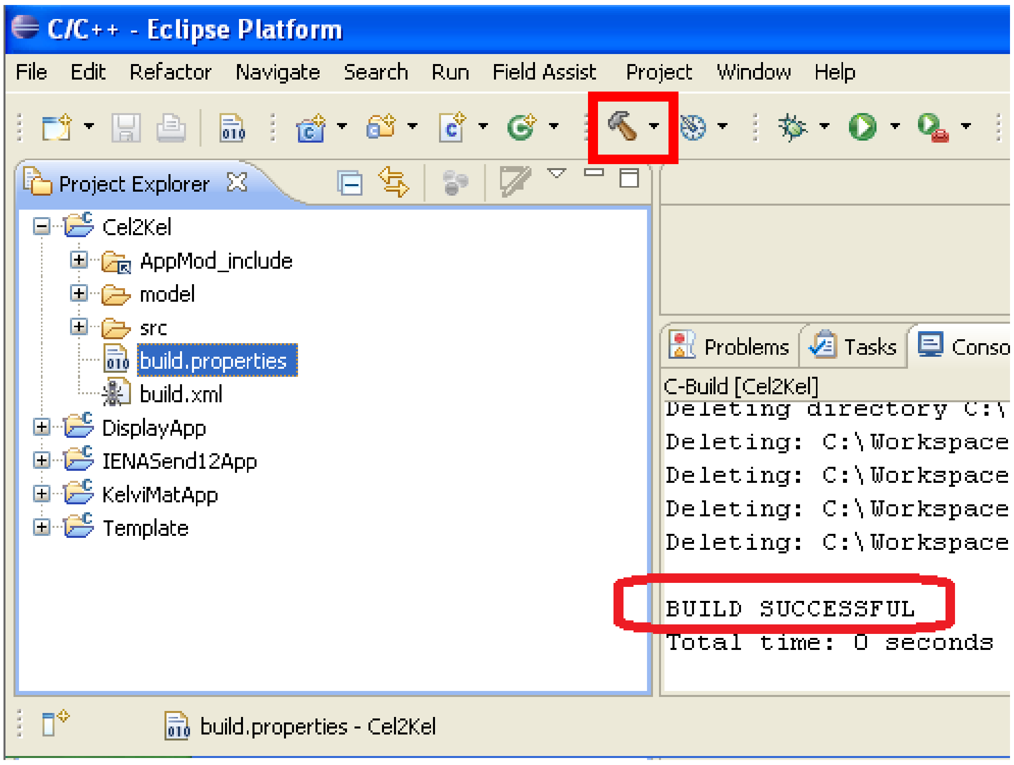
9.3 Adapting the Template Path

- Under *Template*, open the file *"build.properties"*
- Adapt the paths for the objects highlighted in red in accordance with your installation:


C/C++ - Template/build.properties - Eclipse Platform

```
##############################################################
# Diese Variable muß angepasst werden, wenn eine Installation
# in ein anderes Verzeichnis vorgenommen worden ist, als
# das Standardverzeichnis: c:\imc\imcAppMod
imcAppModdir          = C:/Programme/imc/imc_DEVICES_2.7R3/imcAppMod
##############################################################

ant dir               = C:/Programme/Ant
ant prg               = ${ant dir}/bin/imc ant.bat
svn prg               = C:/Programme/Subversion/bin/svn.exe
nmake prg             = ${msvs6 dir}/VC98/Bin/Nmake.exe
gmake prg             = C:/Programme/crossgcc/bin/make.exe

#crosstools
crossgcc.path         = C:/Programme/crossgcc
crossgcc.cygwin       = ${crossgcc.path}/bin

#AppModTools
```

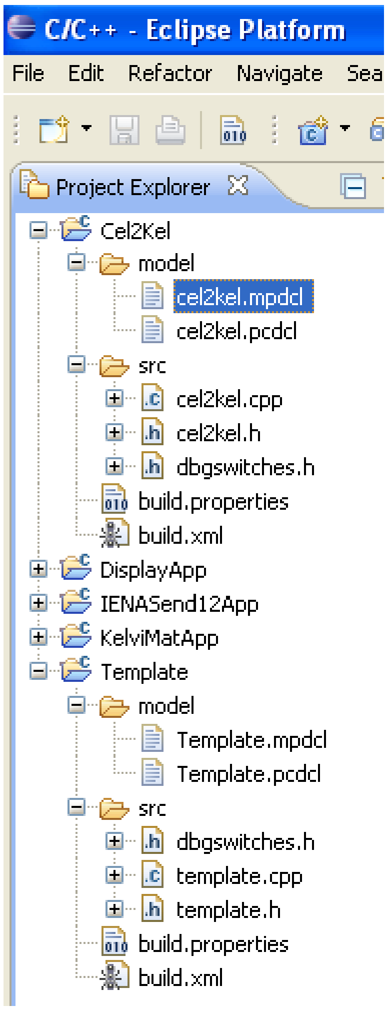
Once call paths have been corrected, you can build the template. To do this, click on the hammer icon:


C/C++ - Eclipse Platform

```
C-Build [Cel2Kel]
Deleting directory C:\
Deleting: C:\Workspace
Deleting: C:\Workspace
Deleting: C:\Workspace
Deleting: C:\Workspace

BUILD SUCCESSFUL
Total time: 0 seconds
```

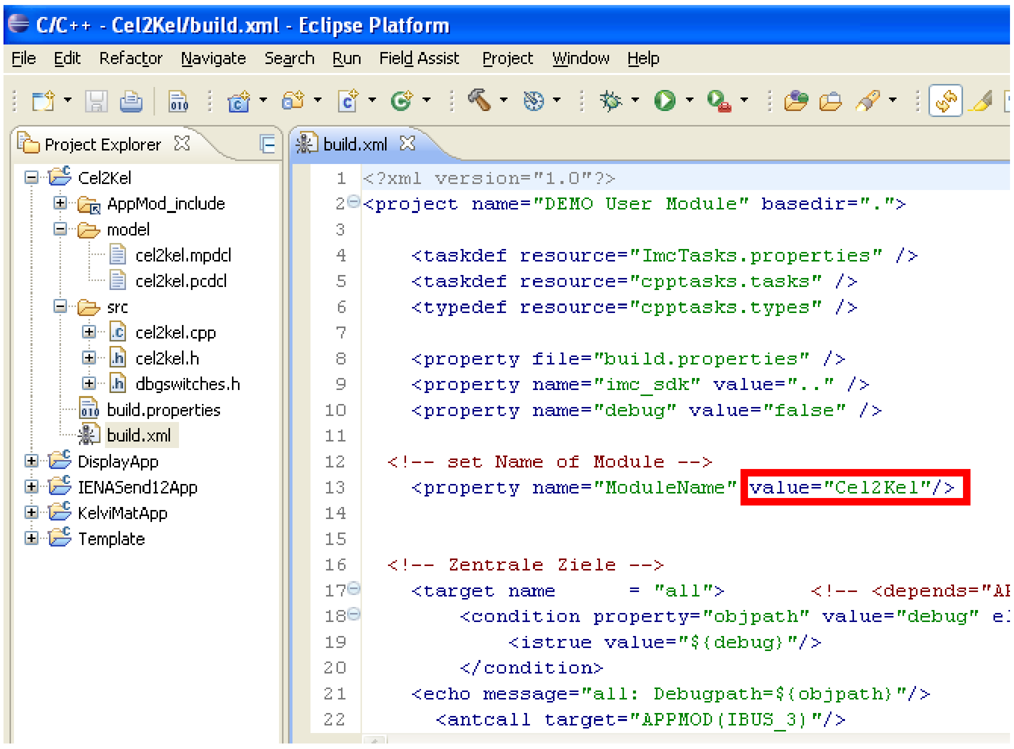
- If the build was successful, the message "BUILD SUCCESFUL" appears in the Eclipse console.

9.4 Renaming the Template for our Project



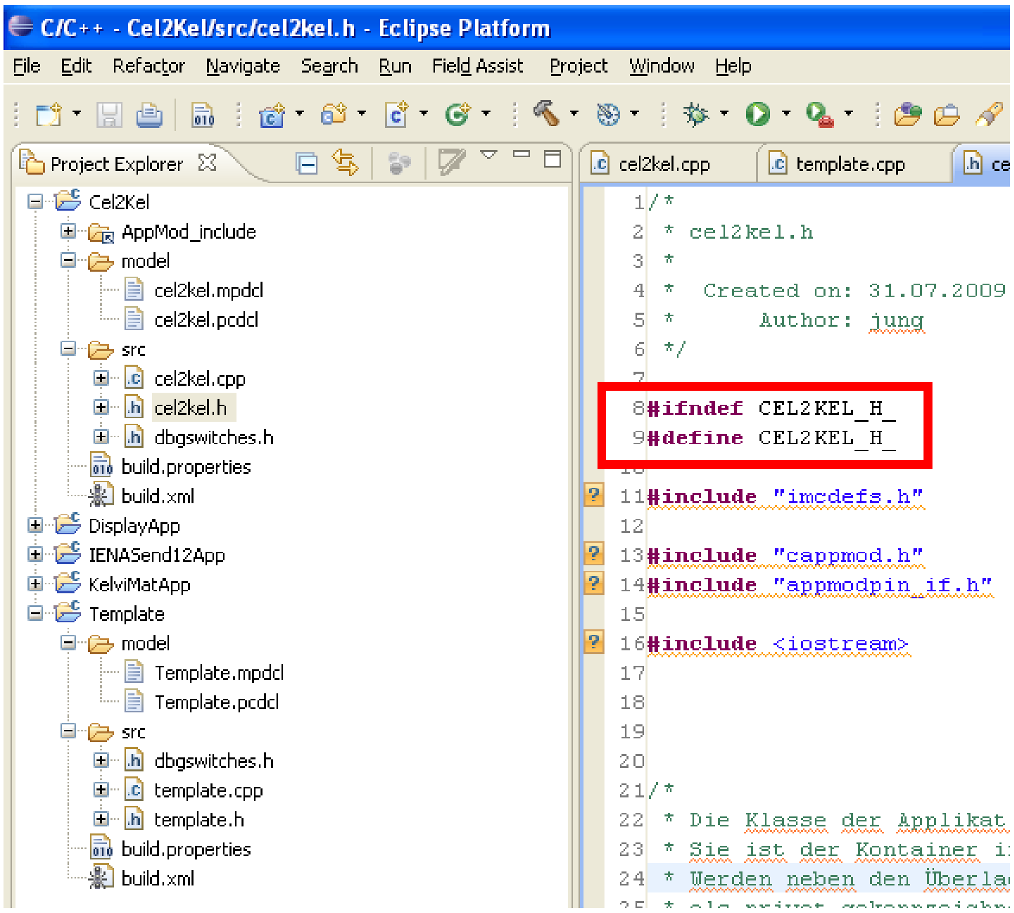
- Copy the "*Template*" project in Eclipse and rename it to "*Cel2Kel*".
- Rename the following project files in the project *Cel2Kel*:
  - *Template.mpdcl -> cel2kel.mpdcl*
  - *Template.pcdcl -> cel2kel.pcdcl*
  - *Template.ccpl -> cel2kel.ccp*
  - *Template.h -> cel2kel.h*

- Change the red shaded entry in the file *build.xml* to *Cel2Kel*



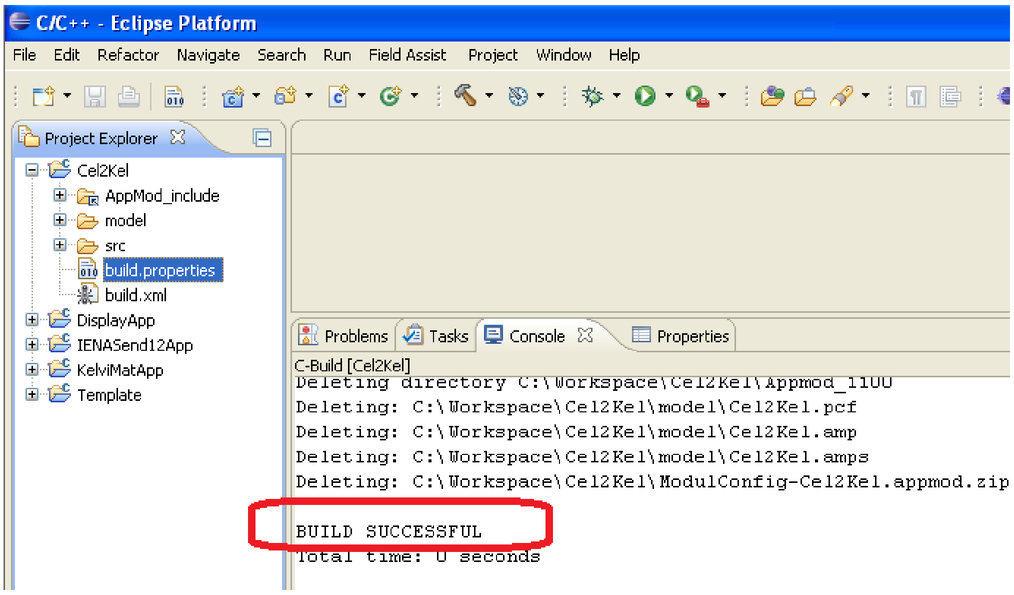
```
<?xml version="1.0"?>
<project name="DEMO User Module" basedir=".">

    <taskdef resource="ImcTasks.properties" />
    <taskdef resource="cpptasks.tasks" />
    <typedef resource="cpptasks.types" />

    <property file="build.properties" />
    <property name="imc_sdk" value=".." />
    <property name="debug" value="false" />

  <!-- set Name of Module -->
    <property name="ModuleName" value="Cel2Kel"/>


  <!-- Zentrale Ziele -->
    <target name      = "all">       <!-- <depends="AP
        <condition property="objpath" value="debug" el
            <istrue value="${debug}"/>
        </condition>
    <echo message="all: Debugpath=${objpath}"/>
      <antcall target="APPMOD(IBUS_3)"/>
```

- Ersetzen Sie in der Datei *cel2kel.cpp* alle "*Template*" Einträge (sowohl "#include", Klassen und aufrufe) durch "*cel2kel*"



```
/*
 * cel2kel.h
 *
 *  Created on: 31.07.2009
 *      Author: jung
 */

#ifndef CEL2KEL_H_
#define CEL2KEL_H_

#include "imcdefs.h"

#include "cappmod.h"
#include "appmodpin_if.h"

#include <iostream>




/*
 * Die Klasse der Applikat
 * Sie ist der Kontainer i
 * Werden neben den Überla
```

- Wenn die Umbenennung erfolgreich war, können Sie das Projekt kompilieren. Klicken Sie hierfür auf das Hammersymbol.
- Wenn der Bau erfolgreich war, erscheint in der Eclipse Konsole die Nachricht "BUILD SUCCESFUL"

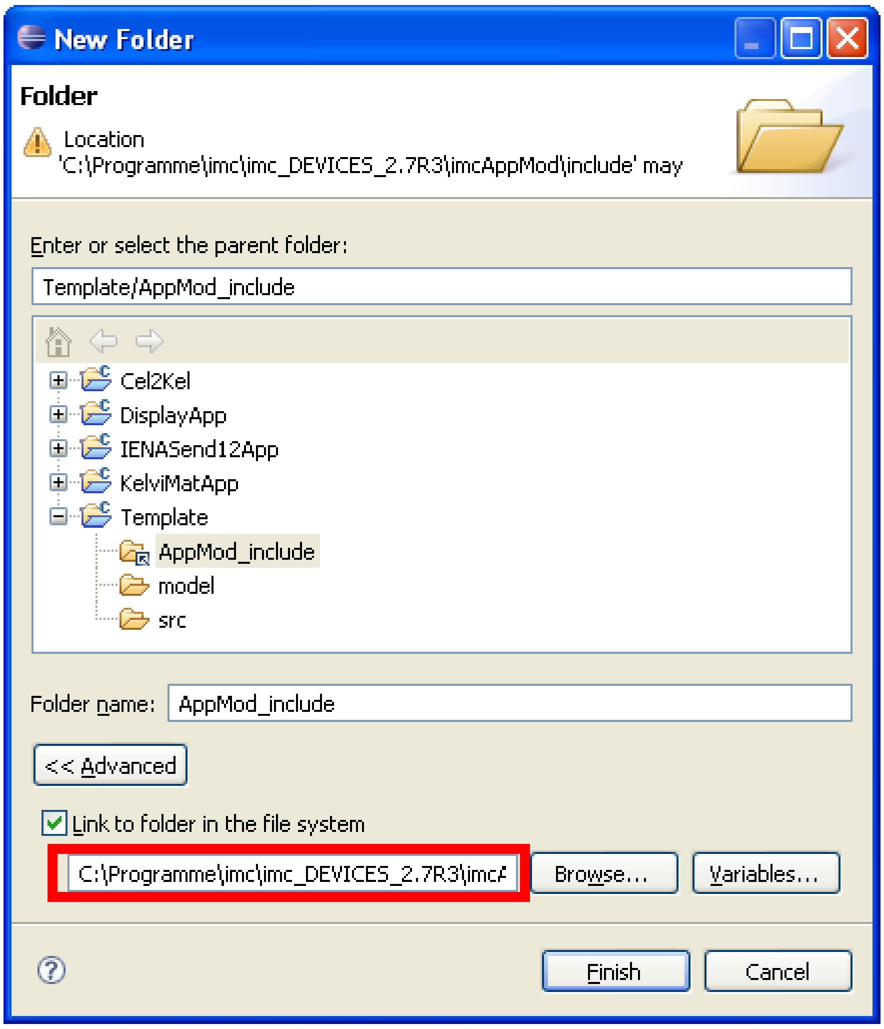


C-Build [Cel2Kel]

```
Deleting directory C:\Workspace\Cel2Kel\Appmod_1100
Deleting: C:\Workspace\Cel2Kel\model\Cel2Kel.pcf
Deleting: C:\Workspace\Cel2Kel\model\Cel2Kel.amp
Deleting: C:\Workspace\Cel2Kel\model\Cel2Kel.amps
Deleting: C:\Workspace\Cel2Kel\ModulConfig-Cel2Kel.appmod.zip

BUILD SUCCESSFUL
Total time: 0 seconds
```

9.5 Realizing our Project

- Add a new "*Link*" folder to the project. Apply the settings shown in the image below:


New Folder

Folder

Location 'C:\Programme\imc\imc_DEVICES_2.7R3\imcAppMod\include' may

Enter or select the parent folder:

Template/AppMod_include

Cel2Kel
DisplayApp
IENASend12App
KelviMatApp
Template
AppMod_include
model
src

Folder name: AppMod_include

<< Advanced

☑ Link to folder in the file system

C:\Programme\imc\imc_DEVICES_2.7R3\imcA

- **Step 1: Adding functions**
  - Shaded red: Select the installation folder for "imcAppModDevSetup", including "...\imcAppMod\include".
  - By means of this link, you have access to finished functions and definitions of variables.

- **Step 2: Defining Inputs/Outputs (cel2kel.pcdcl)**
  - In the file **cel2kel.pcdcl**, two channels are defined.

```
# IMC AppMod Ein Ausgabe Deklarationen
# Kommentar
; Kommentar
; Schlüsselworte beachten Groß und Kleisnchreibung nicht.
; Werte werden mit Groß und Kleinschreibung unterstützt

Pin Name="ToBeSampled" IOType="Source" PinType=PVVar
    Name="pv.Kanal_00_01"
    ValueType="INT16"
    unit=" C"
End

Pin Name="Temperatur_Out" IOType="Sink" PinType=Channel
    Name="Out"
    ValueType="Float"
    idsampletime=15
    inputmode="SAMPLED"
    atoriginal=false
    isuint=false
    unit=" K"
    yscalefactor=1
    yscaletype=1
    yoffset=0
    yminvalue=-1000
    ymaxvalue=1000
    yref1value=-1
    yref2value=40
End
```

- **Step 3: Tunable Parameter (cel2kel.mpdcl)**
  - In the file **cel2kel.mpdcl**, a tunable parameter is generated.

```
[input]
IMC_IEEE_FLOAT ReFactor = 273.0
```

- **Step 4: Definition of pointers and auxiliary variables (cel2kel.h)**
  - In the file **cel2kel.h** changes were made in the *Local Variables* area.

```
#ifndef CEL2KEL_H_
#define CEL2KEL_H_

#include "imcdefs.h"
#include "cappmod.h"
#include "appmodpin_if.h"
#include <iostream>

/*
 * Die Klasse der Applikation als Ableitung der CAppMod.
 * Sie ist der Container in dem die Applikation angelegt wird.
 * Werden neben den Überladenen Methoden weitere implementiert, so sollten diese
 * als privat gekennzeichnet werden.
 * Das Framework ruft nur die Methoden auf, die in der Basisklasse (CAppMod) bereits
 * derklariert worden sind.
 * Für den Großteil der Methoden bestehen in der Basisklasse Methodenrümpfe, die den
 * normalen Ablauf mit dem Gerät sicherstellen.
 */

class Cel2Kel : public CAppMod
{
public:
 /*
  * Diese dienen zur Überprüfung der Kompatibilität und Integrität
  * eines Moduls zur Laufzeit
  */

    IMC_UINT32  GetModuleMagic(void){return MODULEMAGIC;};
    IMC_UINT32  GetModuleVersion(void){return MODVERSION;};

 /*
  * Destruktor
  */
    virtual ~Cel2Kel();
 /*
  * Dieser Konstruktor wird nach der Registrierung der Applikationsklasse
  * (siehe"_register" Klasse) aufgerufen und übermittelt den Zugriff auf den IDMAN.
  * Sie sollte gegebenenfalls Membervariablen initialisieren. (Bsp template.c)
  */
```

```
    Cel2Kel();
 /*
  * Liefert dem Framework den Namen der Applikation. Dieser ist nicht zu
  * verwechseln mit einem Dateinamen.
  */

    virtual IMC_STRING GetApplicationName();
 /*
  * Diese Methode dient der Instantiierung der Applikation durch das Framework
  * (Sie ist so wie in template.c gezeigt zu implementieren).
  */

    static CAppMod * ModCreate();
 /*
  * Wird zur allgemeinen Initialisierung durch das Framework aufgerufen.
  * Sie wird jedoch nur einmal zur Laufzeit aufgerufen. Sie wird nur nach
  * dem Starten bei der ersten Messvorbereitung bei Verwendung der Applikation
  * aufgerufen.
  */

    eAppModError OnInit(void);
 /*
  * Wird aufgerufen, wenn die Applikation beendet werden soll.
  */

    eAppModError OnExit(void);
 /*
  * Diese Methode wird aufgerufen, wenn eine neue Konfiguration verarbeitet
  * worden ist. In dieser Methode kann nun die PinKonfiguration auslesen.
  * Die eigenen Datenstrukturen werden entsprechend eingerichtet.
  * (z.B. übertragen in eigene "Arrays" oder "Variablen")
  */

    virtual eAppModError OnNewConfiguration(void);
 /*
  * Bevor eine Konfiguration verarbeitet wird, wird die alte Konfiguration gelöscht.
  * Dabei wird OnClearConfiguration vom Framework aufgerufen.
  * Diese kann mehrfach aufgerufen werden.
  */

    virtual eAppModError OnClearConfiguration(void);
 /*
  * OnNewDataPoll wird kontinuierlich durch das Framework aufgerufen. Allerdings ist
  * keine feste zeitliche Regel vorhanden.
  * Die Methode darf keine Wartezyklen beinhalten
  */

    void OnNewDataPoll(const CAppModTimerTick & TimeTick);
 /*
  * OnSamplingTimer wird durch das Framework aufgerufen,
  * wenn das Zeitintervall, das mit AddSamplingTimer angemeldet worden ist,
  * abgelaufen ist.
  */

    void OnSamplingTimer(void * pObject, const CAppModTimerTick & Tick);
 /*
  * Das Framework wird die folgenden Methoden aufrufen, wenn auf einem der
  * konfigurierten Kanäle ein Trigger gemeldet worden ist
  */

    eAppModError OnStartTriggerDetected(int TriggerNo);
 ///< called on StartTrigger

    eAppModError OnStopTriggerDetected(int TriggerNo, IMC_BOOL GlobalState);
 ///< called in StopTrigger

    void  OnTimerStarted();
 /*
  * OnTimerStarted meldet den Start der Messung
  */
private:
 /*
  * Lokale Variablen
  */
    bool m_first;
 /*
  * Referenz zu der PinTabelle (Pin["NAME"])
  */
    APP_PIN_MAP     * m_pPinMap;
 /*
```

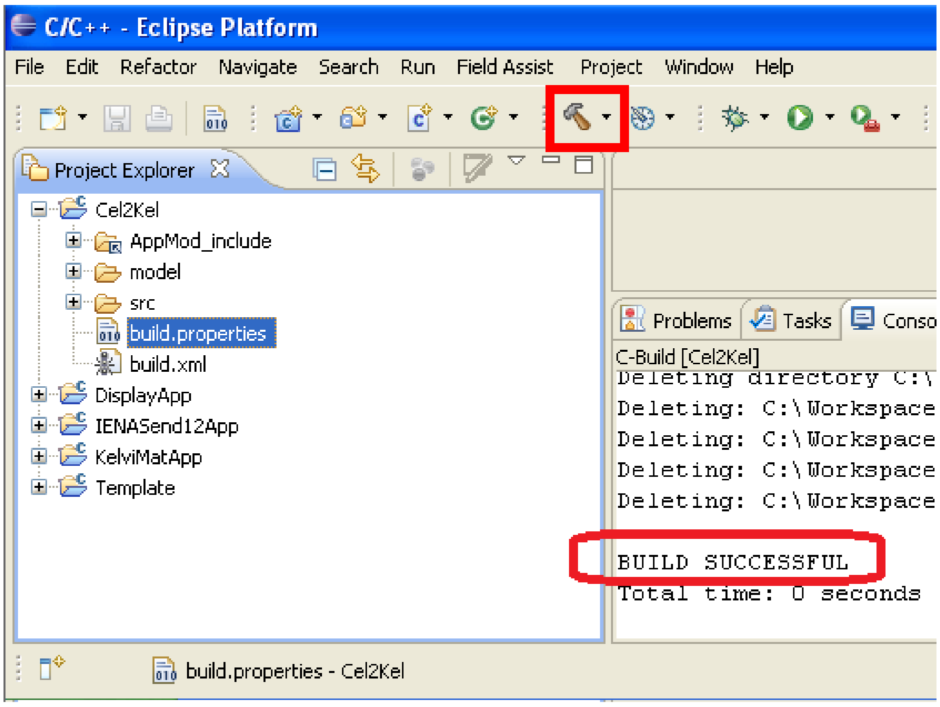
```
  * Verweise auf spezielle "PIN"'s
  */

    CAppModPin  * m_pToBeSampled;    /* PVVar ToBeSampled */
    IMC_BOOL      m_ToBeSampled_Triggered;
                               /* Vermerkt Triggerstatus des ToBeSampled Kanals */
    CAppModPin  * m_pTemperatur_Out;
    IMC_BOOL      m_ToTemperatur_Out_Triggered;
    CAppModPin  * m_pDevice;     /* Verweis auf einen Datein Ein/Ausgabe Kanal */
    PAPP_CHANNEL  m_papc;
};

/*
 * Die "*_register" Klasse dient der Registrierung der Appkikationsklasse beim Framework.
 * Ohne diese kann das Framework die Applikation nicht aufrufen.
 * Es ist notwendig eine globale Instanz der "_register" Klasse anzulegen.
 * (siehe template.cpp)
 */

class Cel2Kel_register
{
public:
 Cel2Kel_register(){
        CAppMod::RegisterAppMod(Cel2Kel::ModCreate);
#ifdef _DEBUG
        std::cerr << std::endl << "Cel2Kel_register(): Registered Cel2Kel" << std::endl /
 << std::endl ;
#endif

}
    ~Cel2Kel_register(){};
private:
 Cel2Kel_register(Cel2Kel_register &);              // NoImpl
 Cel2Kel_register & operator= (Cel2Kel_register &); // NoImpl
};

#endif /* CEL2KEL_H_ */
```

- **Step 5: Add functionality (cel2kel.cpp)**
  - Adopt the following changes

```
#include "ccommtypes.h"
#include "ccommcom.h"
#include "cel2kel.h"
#include <iostream>

/*
 * Die folgende Deklaration registriert die Applikation beim Framework.
 * Die Definition der "_register"-Klasse und die Bekanntmachung
 * ist zwingend erforderlich.
 */

Cel2Kel_register JustToReg;
/*
 * Konstruktor der Applikation
 */

Cel2Kel::Cel2Kel():
    CAppMod()
{
}

/*
 * Dient dem Framework als Relaismedhode
 */

CAppMod * Cel2Kel::ModCreate()

{
    return new Cel2Kel();
}

/*
 * Destruktor
 */

Cel2Kel::~Cel2Kel()
{
}
```

```
IMC_STRING Cel2Kel::GetApplicationName()
{
    IMC_STRING Cel2Kel = "Cel2Kel";
    return Cel2Kel;
}

/*
 * Wird durch das Framework bei Applikationsstart aufgerufen.
 */

CAppMod::eAppModError Cel2Kel::OnInit()
{
    std::cerr << "Cel2Kel::OnInit()" << std::endl;
    m_pPinMap = GetPinMap();
    return APP_SUCCESS;
}

/*
 * Wird durch das Framework aufgerufen, wenn die Applikation beendet wird.
 */

CAppMod::eAppModError Cel2Kel::OnExit()
{
    std::cerr << "Cel2Kel::OnExit()" << std::endl;
    return APP_SUCCESS;
}

/*
 * Wird durch das Framework aufgerufen, wenn die Konfiguration verarbeitet wurde
 * und anschließend durch die Applikation ausgewertet werden kann.
 */
CAppMod::eAppModError Cel2Kel::OnNewConfiguration(void)
{
    eAppModError eRes = APP_BAD_CFG;
    std::cerr << "Cel2Kel::OnNewConfiguration()" << std::endl;
    // clean and reset pin handle of Channel ToBeSampled

    m_pToBeSampled = NULL;
    m_pTemperatur_Out = NULL;
    m_first= true;

    try{
        do{
            APP_PIN_MAP::iterator it;
            APP_PIN_MAP & Map = *(m_pPinMap); // reference to PinMap.
            /*
             * Rufe PIN "ToBeSampled" aus der PIN Map ab.
             */
            it = Map.find("Temperatur_Out");
            if(Map.end()== it )
                break;
            /*
             * Referenz speichern und Status initialisieren.
             */
            m_pTemperatur_Out = it->second;
            m_ToTemperatur_Out_Triggered = IMC_FALSE;

            it = Map.find("ToBeSampled");
            if(Map.end()== it )
                break;
            /*
             * Referenz speichern und Status initialisieren.
             */
            m_pToBeSampled = it->second;
            m_ToBeSampled_Triggered = IMC_FALSE;
            /*
             * Hier wird die Abtastzeit abgerufen.
             */
            APP_CHANNEL * papc;
            m_pTemperatur_Out->GetResource(papc);
            m_pTemperatur_Out->GetResource(m_papc);
            CAppModTimerTick PaiSamplingTimeinterval((IMC_UINT16) /*papc->XAxes.iIdSampleTime*/
15);

            AddSamplingTimer( PaiSamplingTimeinterval, (void *) 1 );
            eRes = APP_SUCCESS;

        }while(false);

    }catch(...){
        eRes = APP_BAD_CFG;
    }
    return eRes;
}

CAppMod::eAppModError Cel2Kel::OnClearConfiguration(void)
{
    char sCmd[128];
    IMC_INT32 wsize;
    PCComm  pComPort;
    std::cerr << "Cel2Kel::OnClearConfiguration()" << std::endl;
    // clean and reset pin handle of Channel A0
    m_pToBeSampled = NULL;
    m_pTemperatur_Out = NULL;
    m_papc = NULL;
    return APP_SUCCESS;
}

/*
 * Bei Messstart zeigt diese Methode den Beginn.
 * Hier im Cel2Kel ist bereits durch das Framework alles erledigt worden.
 */

void Cel2Kel::OnTimerStarted()
{

}

/*
 * Diese Methode wird ständig aufgerufen. Mindestens ein Mal pro Abtastintervall.
 */

void Cel2Kel::OnNewDataPoll(const CAppModTimerTick & Tick)
{
    if(m_first){
        IMC_IEEE_FLOAT f16In = (IMC_IEEE_FLOAT) m_pToBeSampled->readInt16() * m_papc-
>YAxes.dScaleFactor + m_papc->YAxes.dOffset;
        printf("f16In: %f\n", f16In);

        f16In =f16In*0.06199+ 273;
        printf("f16In: %f\n", f16In);

        const CAppModTimerTick DummyTick;
        m_pTemperatur_Out->writeFloat(DummyTick, f16In );
        m_first=false;
    }

}

/*
 * Wird aufgerufen, damit die Applikation die Gelegenheit hat,
 * bei Auslösung eines Starttriggers Arbeiten zu erledigen, die in
 * solchen Fällen zu erledigen sind (z.B. Meldungen an angeschlossene
 * Geräte zu senden).
 */

CAppMod::eAppModError Cel2Kel::OnStartTriggerDetected(int TriggerNo)
{
       return APP_SUCCESS;
}

/*
 * Wird aufgerufen, damit die Applikation die Gelegenheit hat,
 * bei Auslösung eines Stoptriggers Arbeiten zu erledigen, die in
 * solchen Fällen zu erledigen sind (z.B. Meldungen an angeschlossene
 * Geräte zu senden).
 */

CAppMod::eAppModError Cel2Kel::OnStopTriggerDetected(int TriggerNo, IMC_BOOL GlobalState)
{
    return APP_SUCCESS;
}

/*
 * Wird für jedes angemeldete Abtastintervall aufgerufen.
 */
void Cel2Kel::OnSamplingTimer(void * pObject, const CAppModTimerTick & Tick)
{
    IMC_UINT32 hash = (IMC_UINT32) pObject; // on Cel2Kel this is just a 32bit integer.

    /* wie eine identifizierung der Abtastzeitintervalle geschieht, liegt
     * bei der Applikation. Ein weg wäre, einen Hashwert zu speicher.
     * Eine andere Variante wäre, der Registrierung einen INDEX in eine
     * Tabelle zu verwenden.
     * Entscheident ist dabei lediglich, dass die verwendeten Werte
     * eindeutig sind.
     */

    if(1 == hash){
        IMC_IEEE_FLOAT f16In = (IMC_IEEE_FLOAT) m_pToBeSampled->readInt16()  * m_papc-
>YAxes.dScaleFactor + m_papc->YAxes.dOffset;
        printf("f16In: %f\n", f16In); // <- Debug Ausgabe

        f16In =f16In*0.06199+ 273; // <- Umrechung
        printf("f16In: %f\n", f16In); // <- Debug Ausgabe

        const CAppModTimerTick DummyTick;
        m_pTemperatur_Out->writeFloat(DummyTick, f16In ); // <- Ausgabe Kanal
    }
}
```

- **Step 6: Compile**:
  - Once all changes to the project have been made, compile the project by clicking on the hammer symbol.



```
C-Build [Cel2Kel]
Deleting directory C:\
Deleting: C:\Workspace
Deleting: C:\Workspace
Deleting: C:\Workspace
Deleting: C:\Workspace

BUILD SUCCESSFUL
Total time: 0 seconds
```

Note

If you changed the Standard Software (imc STUDIO) on which the operation of your project is based, the project must be re-compiled. This is absolutely necessary when upgrading to a higher revision number, e.g. imc STUDIO 5.0R1 to 5.0R3.

9.6 Expanding the Project

Up to now, only the tunable parameter has been defined. In the next step, an offset control could be achieved.

10 Technical Specs - imc APPMOD

Embedded Processor		
Parameter	**Value**	**Remarks**
Embedded processor	Freescale Power PC MPC5200B Core CLK 384 MHz	
RAM	64 MB 48 MB	total memory available for the application
Flash	16 MB	only for the operation system
Operating system	Linux	

General		
Parameter	**Value**	**Remarks**
Interfaces	1x Ethernet interface and 1x serial interface 3.5 mm jack plug	Specific applications can each use exactly one of the two interfaces. Simultaneous use of both interfaces requires a system with two interfaces. service-jack (RS232, 115 kBaud, Tx, Rx, GND) console for development, debugging
Module width	requires 1 slot	fixed installation, ex factory
Modularity	order option	upon request
Max. amount of interfaces in one system	3 8 1 2 3 5	totally in one CRFX base unit totally in one CRC system totally in one BUSFX-4 system totally in one BUSFX-6 system totally in one BUSFX-8 system totally in one BUSFX-12 system

Ethernet Variant		
Parameter	**Value**	**Remarks**
Terminals / Nodes	1	
Terminal connectors	1x RJ45	
Topology	bus	
Transfer protocol	TCP / IP	IEEE Norm 802.3
Transfer medium	Ethernet	
Data flow direction	sending/receiving	
Baud rate	100 MBit 10 MBit	100BaseT (Half- and Full-duplex) 10BaseT (Half- and Full-duplex) Auto-sensing
Isolation strength	60 V	to system ground (CHASSIS)

Serial interface variant		
Parameter	**Value**	**Remarks**
Terminals / Nodes	1	
Terminal connectors	1x DSUB-9	
Baud rate	300, 1200, 2400, 4800, 9600, *14400*, 19200, *28800*, 38400, 57600, 115200, 230400	special bit-rates: 14400 and 28800
Isolation Isolation strength	galvanically isolated 60 V	to system ground (CHASSIS) nominal working voltage
Operation modes	RS 232 RS 485 / RS 422	flexibly configurable: multi-protocol transceiver
RS232 mode		
Parameter	**Value**	**Remarks**
Topology	point-to-point	
Transfer protocol	RS232	
Signal type	Tx, Rx, GND CTS, RTS	Basis signals Handshake, flow control
Data flow direction	sending/receiving	
Byte format	7 or 8 data bits, 1 or 2 stop bits, none/odd/even parity	
Flow control	XON/XOFF, RTS/CTS	
RS485/422 mode		
Topology	Bus	
Transfer protocol	RS485	compatible to RS422
Operating mode	Half- and Full-duplex	activated via software
Signal type	2x Tx, 2x Rx, GND	basis signals, differential
Data flow direction	sending/receiving	
Termination	120 Ω	activated via software

11 Pin configuration

RS 232

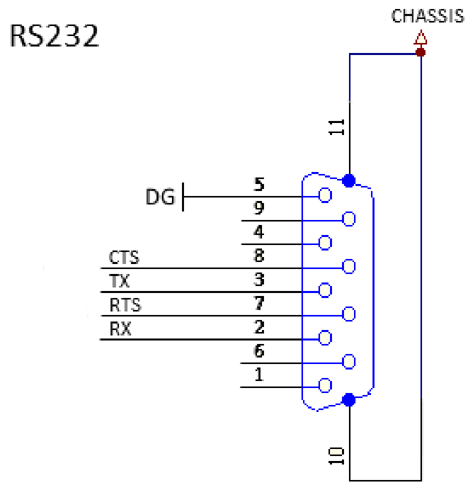
Signal	PIN
n.c.	1
RX	2
TX	3
n.c.	4
DG	5
n.c.	6
RTS	7
CTS	8
n.c.	9

RS 422 / RS 485 Full-Duplex

Signal	PIN
Rx+	2
Rx-	8
TX+	3
Tx-	7

RS 485 Half-Duplex

Signal	PIN
+D	3
-D	7



Service-interface jack plug 3,5mm

(RS232, 115 kBaud, Tx, Rx, GND)

Signal	DSUB9 Pin	jack plug 3,5 mm
RX	2	TIP (L)
TX	3	RING (R)
GND	5	Shield

Index