An Axiometrix Solutions Brand

imc Applikations-Modul

Handbuch

Edition 11 - 08.07.2022

imc Test & Measurement GmbH • Voltastraße 5 • 13355 Berlin • Deutschland

Haftungsausschluss

Diese Dokumentation wurde mit großer Sorgfalt erstellt und auf Übereinstimmung mit der beschriebenen Hard- und Software geprüft. Dennoch können Abweichungen und Fehler nicht ausgeschlossen werden, sodass wir für die vollständige Übereinstimmung keine Gewähr übernehmen.

Technische Änderungen bleiben vorbehalten.

Copyright

Open Source Software Lizenzen

Einige Komponenten von imc-Produkten verwenden Software, die unter der GNU General Public License (GPL) lizenziert sind. Details finden Sie im About-Dialog.

Falls Sie eine Kopie der verwendeten GPL Sourcen erhalten möchten, setzen Sie sich bitte mit unserer Hotline in Verbindung.

Inhaltsverzeichnis

1 Allgemeine Einführung

1.1 Bevor Sie starten

Sehr geehrter Nutzer.

1. Die überlassene Software sowie das dazugehörige Handbuch sind für fachkundige und eingewiesene Benutzer ausgestaltet. Sollten sich Unstimmigkeiten ergeben, wenden Sie sich bitte an unsere Hotline 4.
2. Durch Updates in der fortschreitenden Softwareentwicklung können einzelne Passagen des Handbuchs überholt sein. Wenn Ihnen Abweichungen auffallen, wenden Sie sich bitte an unsere Hotline.
3. Wenden Sie sich bitte an unsere Hotline, wenn Sie aufgrund missverständlicher Regelungen oder Ausführungen des vorliegenden Handbuchs zu der Auffassung gelangen, dass Personenschäden zu befürchten sind.
4. Lesen Sie den Lizenzvertrag. Mit der Nutzung der Software, erkennen Sie die Bedingungen des Lizenzvertrags an.

1.2 imc Kundendienst / Hotline

Zur technischen Unterstützung steht Ihnen unser Kundendienst bzw. unsere Hotline zur Verfügung:

imc Test & Measurement GmbH

Hotline: **+49 30 467090-26**

E-Mail: hotline@imc-tm.de

Internet: https://www.imc-tm.de

Internationale Vertriebspartner

Die internationalen Vertriebspartner finden Sie im Internet unter https://www.imc-tm.de/distributoren/.

Hilfreich für Ihre Anfrage:

Sie helfen uns bei Anfragen, wenn Sie die **Seriennummer Ihrer Produkte**, sowie die **Versionsbezeichnung der Software** nennen können. Diese Dokumentation sollten Sie ebenfalls zur Hand haben.

- Die Seriennummer des Gerätes finden Sie z.B. auf dem Typ-Schild auf dem Gerät.
- Die Versionsbezeichnung der Software finden Sie in dem Info-Dialog.

Produktverbesserung und Änderungswünsche

Helfen Sie uns die Dokumentation und die Produkte zu verbessern:

- Sie haben einen Fehler in der Software gefunden oder einen Vorschlag für eine Änderung?
- Das Arbeiten mit dem Gerät könnte durch eine Änderung der Mechanik verbessert werden?
- Im Handbuch oder in den technischen Daten gibt es Begriffe oder Beschreibungen, die unverständlich sind?
- Welche Ergänzungen und Erweiterungen schlagen Sie vor?

Über eine Nachricht an unseren Kundendienst 4 würden wir uns freuen.

1.3 Rechtliche Hinweise

Qualitätsmanagement

imc Test & Measurement GmbH ist seit Mai 1995 DIN-EN-ISO-9001 zertifiziert. Aktuelle Zertifikate, Konformitätserklärungen und Informationen zu unserem Qualitätsmanagementsystem finden Sie unter:
https://www.imc-tm.de/qualitaetssicherung/.

imc Gewährleistung

Es gelten die Allgemeinen Geschäftsbedingungen der imc Test & Measurement GmbH.

Haftungsbeschränkung

Alle Angaben und Hinweise in diesem Dokument wurden unter Berücksichtigung der geltenden Normen und Vorschriften, dem Stand der Technik sowie unserer langjährigen Erkenntnisse und Erfahrungen zusammengestellt. Die Dokumentation wurde auf Übereinstimmung mit der beschriebenen Hard- und Software geprüft. Dennoch können Abweichungen und Fehler nicht ausgeschlossen werden, sodass wir für die vollständige Übereinstimmung keine Gewähr übernehmen. Technische Änderungen bleiben vorbehalten.

Der Hersteller übernimmt keine Haftung für Schäden aufgrund:

- Nichtbeachtung des Handbuches sowie der Ersten Schritte
- Nichtbestimmungsgemäßer Verwendung.

2 imc Applikations-Modul

Das imc Applikations-Modul dient dazu, **Messkanäle** in ein imc CRONOS*compac*t bzw. imc CRONOS*flex* System zu **integrieren**, die **von "externen" Geräten oder Systemen** über Standard Hardware-Schnittstellen geliefert werden.

Diese Quellen können etwa folgende sein:

- spezielle komplexe Sensoren
- "externe" Geräte
- Bussysteme (z.B. Feldbusse)

Die unterstützen Standard-Schnittstellen sind insbesondere:

- Ethernet
- serielle Schnittstellen (RS-232, RS-485, RS-422)

Die einzubindenden Systeme sind typischerweise anwenderspezifische bzw. dedizierte Geräte von Fremd-Herstellern. Die Integration erfolgt mittels eines Standard-Hardware-Moduls (APPMOD). Auf diesem steht ein dedizierter Prozessor zur Verfügung, für den eine spezielle Applikation programmiert wird. Diese wird entweder von imc als Auftragsarbeit erstellt, oder kann von qualifizierten Partnern bzw. speziell geschulten Anwendern mit zur Verfügung gestellten Entwicklungswerkzeugen implementiert werden.

Diese anwenderspezifische Hard- und Software-Erweiterung wird dabei von der Gerätesoftware (imc STUDIO) unterstützt. Eine spezielle Version der Gerätesoftware ist nicht nötig.

Besondere Merkmale:

- gekapselte Hardware + Software Spezial-Lösung, eingebettet in ein imc Standard System
- Standard-System mit vollständiger Software-Unterstützung
- flexibel unterstützt durch unveränderte Standard Geräte-Software
- Standard-Hardware Komponente
- Stand-alone fähige autarke System-Umgebung

3 Funktionsweise

Das imc Applikations-Modul stellt ein Subsystem innerhalb der imc CRONOS System Familie dar, auf dem ein autarker Prozessor die anwenderspezifische Applikation ausführt.

Die Applikation kommuniziert über die vom Hardware-Modul bereitgestellten Schnittstellen (Ethernet oder serielle Ports) mit den externen Geräten, wobei das Modul sowohl empfangen (Datenaufnahme) als auch senden kann.

Mit dem imc Messgerät werden Daten über die folgenden Mechanismen ausgetauscht:

- Kanäle ("FIFO-Kanäle")
- pv-Variablen ("Prozessvektor")
- Display-Variablen

Dabei kann jede dieser Größen jeweils entweder als Eingangs- oder Ausgangs-Größe definiert werden.

Die Kanäle und werden in der Rubrik der "Feldbus-Kanäle" verwaltet und sind in die üblichen Mechanismen wie Start / Stopp der Messung und Trigger eingebunden, wie alle anderen Kanäle auch. Sie können sowohl gleichförmig abgetastet sein, als auch ein nicht äquidistantes Zeitstempel-Format haben.

Entscheidend ist, dass die Einbindung in einer Weise erfolgt, dass kontinuierlich strömende Daten ("streaming data") erzeugt werden. Sie sind dabei synchron, können äquidistant sein, können also auch imc Online FAMOS verrechnet werden.

Begriffsdefinitionen

Applikationsarchiv: Konfiguration eines Applikations-Moduls. Die Konfigurationsdatei nutzt die Dateierweiterung ".appmod". Applikationsarchive werden in gepackter Form als ZIP-Datei gespeichert und können unabhängig vom Experiment in einem Ordner abgelegt werden. Das aktuell verwendete Archiv wird mit dem Experiment gespeichert.

4 Workflow

Entwicklungsprozess durch imc, ausgewählte Partner oder geschulte Kunden

- Einrichten der Entwicklungsumgebung: Eclipse
- Programmierung der Applikation in C++
- Erzeugen eines fertigen Komplilats, als gezippte *.appmod Datei

Anwendung durch den Benutzer

- Verwendung der gezippten *.appmod Datei mit "beliebiger" unveränderter Standard Software imc STUDIO
- im Zuge der konkreten Konfiguration des Geräts (Experiment):
  Auswahl der spezifischen Applikation durch Angabe des Speicherorts der *.appmod
  (auch z.B. vom USB-Stick)
- Auch flexible Auswahl unter mehreren Applikationen möglich
- Code der Applikation aus *.appmod wird ins Experiment eingebettet, muss also zur Laufzeit bzw. beim Laden des Experiments nicht vorhanden sein.
- Applikation ist entweder fest programmiert oder kann vom Benutzer über einen spezifischen Dialog / Menü parametriert werden.

Debugging, Service, Diagnose

- Zur Laufzeit im Zielsystem ist über eine separate Service-Schnittstelle ("SERVICE", RS232, 3.5mm Klinke) eine "Console" verfügbar, mit der etwa über den PC im Sinne eines "Fahrtenschreibers" Debug und Protokoll Informationen geloggt werden können, um im realen Betrieb Diagnose oder Weiterentwicklung der Applikation zu unterstützen
- Hierfür ist NICHT das Entwicklungssystem (Eclipse) nötig, sondern nur das Standard Konsolen Programm auf Ihrem PC!

5 Inbetriebnahme

5.1 Systemvoraussetzungen

Hardware Voraussetzungen:

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

Software Voraussetzungen:

- imc STUDIO 4.0R1 oder höher

Hinweis

Für die meisten Anwendungen werden Prozessvektor-Variablen 7 verwendet. Für die Nutzung der pv-Variablen muss im Gerät imc Online FAMOS Professional freigeschaltet sein.

5.2 Installationsanleitung: Entwicklungsumgebung

Es gelten folgende Voraussetzungen für die Installation der einzelnen Komponenten der Entwicklungsumgebung. Grundsätzlich sollte der Entwickler PC mit einem hinreichend aktuell gehaltenem Betriebssystem versehen sein. Java muss nicht installiert sein. Bei Bedarf kann eine JAVA Version (im Paket enthalten) installiert werden.

Es wird später davon ausgegangen, dass compiler, ant, etc an ihren, durch die Setup-Programme vorgeschlagenen, Zielpfade installiert werden.

1. Ausführen der Batch-Datei "Install-IDE.bat". (ab Firmware Version (imc DEVICES) 2.11R1) Die Installation erfolgt in "*C:\imc*\crossgcc\".

Hinweis: Die Installationsdateien dürfen nicht aus dem Desktop-Verzeichnis ausgeführt werden.

2. Applikationsentwicklungsdateien zur installierten Firmware Version (imc DEVICES)

   Installieren Sie "*Products\imc DEVICES\Tools\imcAppMod\imcAppModDevSetup.exe*" von dem imc STUDIO-Installationsmedium.
   Ändern Sie den vorgegebenen Standardpfad auf "*C:\imc\imcAppMod*". Der Pfad in den die Dateien installiert worden sind, wird bei der Entwicklung der Module benötigt.

Hinweis

Es können mehrere Versionen installiert werden, die müssen sich dann in verschiedenen Pfaden befinden: *C:\imc\imcAppMod_2_11R1,*
C:\imc\imcAppMod_2_12R1, etc. Beim Übersetzen der Module muss dann der betreffende Pfad in der Build.properties eingetragen werden.

5.3 Einrichten der Entwicklungsumgebung und hinzufügen der Demo Apps

Zum Einrichten der Entwicklungsumgebung mit den mitgelieferten Demonstrations Applikationen, gehen Sie folgendermaßen vor:

1. Im Eclipse "Workspace Launcher" den Workspace setzen und die Ansicht auf die Workbench wechseln:

- Ins Feld Workspace z.B. "C:\Test234" eintragen. Verwenden Sie **NICHT** "c:\imc\imcAppMod" oder "c:\imc\imcAppMod\DemoApp"!
- *OK* betätigen
- Mitteilungen an Eclipse Community können aktiviert werden. Dies spielt für die Ausführung keine Rolle.
- Workbench

2. Applikations-Moduls-**Referenz** Projekte in Eclipse importieren:

1. Menu *File* > *Import*
2. "*General*" > "*Existing Projects into Workspace*" auswählen und "*Next*" drücken
3. bei "*Select root directory:*" "*C:\imc\imcAppMod*" einfügen und "*Browse*" drücken
4. "*Finish*" drücken

3. Applikations-Moduls-**Demo** Projekte in Eclipse importieren:

1. Menu File > *Import*
2. "*General*" > "*Existing Projects into Workspace*" auswählen und "*Next*" drücken
3. bei "*Select root directory:*" "*C:\imc\imcAppMod\DemoApps*" einfügen und "*Browse*" drücken
4. "*Finish*" drücken

4. Die "*build.properties*" der Projekte falls nötig anpassen:.

Ggf. vom unteren Reiter "*Build*" auf Reiter "*build.properties*" wechseln

1. Project Explorer "*Kelvimat*" > "***build.properties***" doppelklicken; "***build.properties***" anpassen; siehe Beispiel unten
2. Project Explorer "*DisplayApp*" > "***build.properties***" doppelklicken; "***build.properties***" anpassen; siehe Beispiel unten
3. bis 7. Verfahren Sie so mit allen Projekten: Project Explorer "*IENASend12App*", "*FifoReaderDemoApp*", "*RS422DemoRApp*", "*Template*", "*Template_en*"

Beispiel: Passen Sie die fettmarkierten Einträge an:

"build.properties"

```
################################################################
# Diese Variable muss angepasst werden, wenn eine Installation
# in ein anderes Verzeichnis vorgenommen worden ist, als
# das Standardverzeichnis: 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. Alle Dateien speichern:

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

6. Die Applikations-Modul-Zip-Archive bauen:

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

7. Prüfen Sie mit dem Windows-Explorer, ob die Appmod Zip-Archive angelegt wurden. Jeweils unter den folgenden Pfaden sollten eine passende ZIP-Datei angelegt sein:

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. Es ist hilfreich in den Einstellungen von Eclipse "*Build Automatically*" zu **deaktivieren** und "*Save before Build*" zu **aktivieren**.

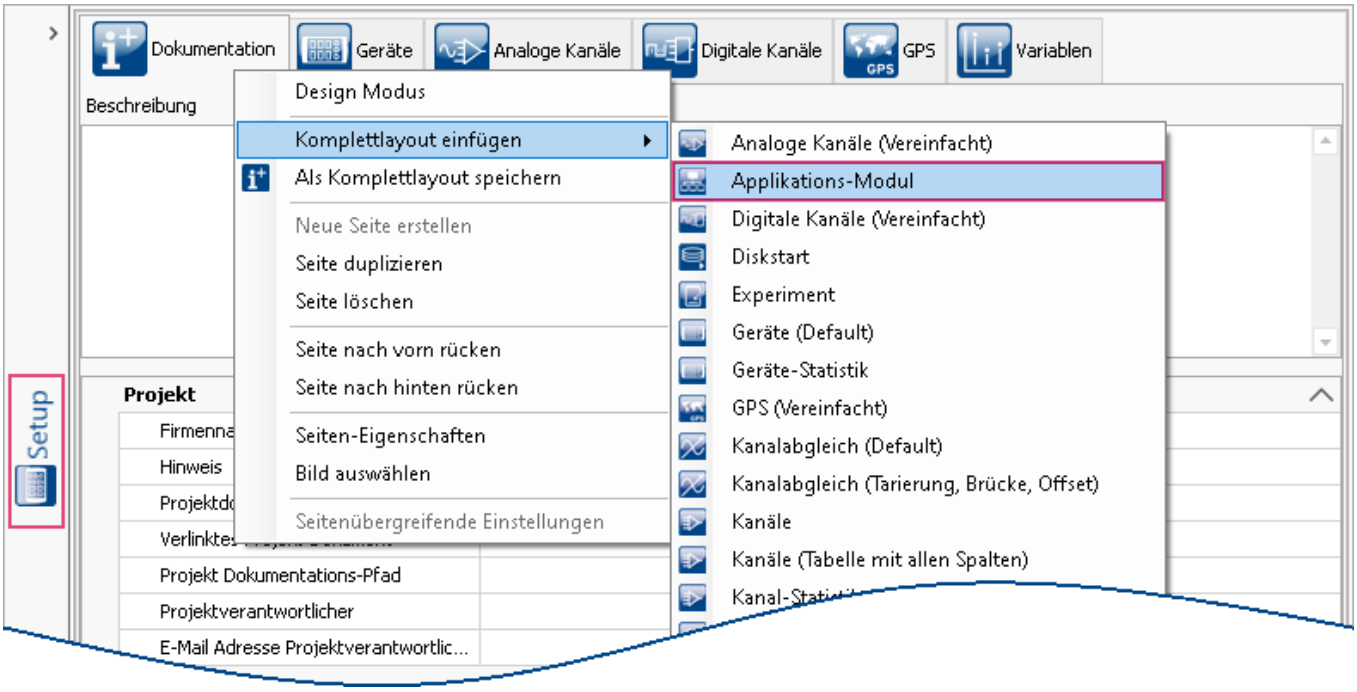
6 imc Applikations-Modul Assistent

Die Entwicklung eines Applikationsarchives unterscheidet sich stark von der Anwendung.

Dieses Kapitel beschreibt die Anwenderseite des imc Applikations-Moduls. Informationen zur Entwicklung finden Sie im folgenden Kapitel 16.

6.1 Assistent in imc STUDIO

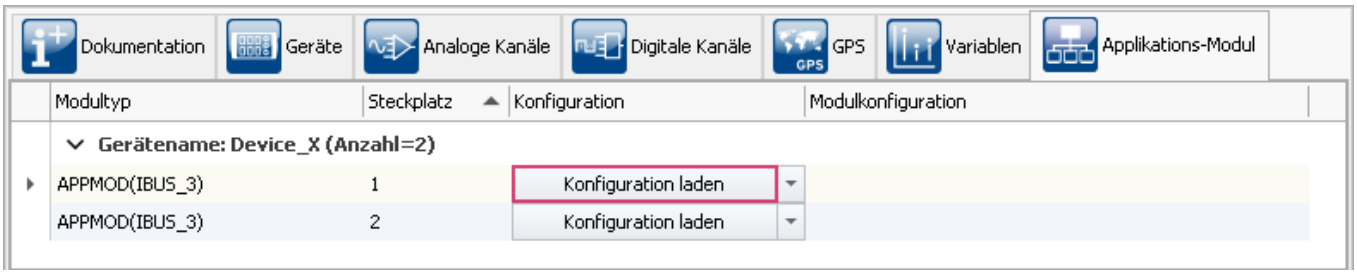
Der Assistent zur Konfiguration des Applikations-Moduls wird über die Layout-Ablage hinzugefügt:



Aufruf des Assistenten

6.1.1 Laden einer Applikation

Um eine Applikation zu laden klicken Sie beim auf "*Konfiguration laden*". Danach öffnet sich ein Dialogfenster zur Auswahl eines Modells.

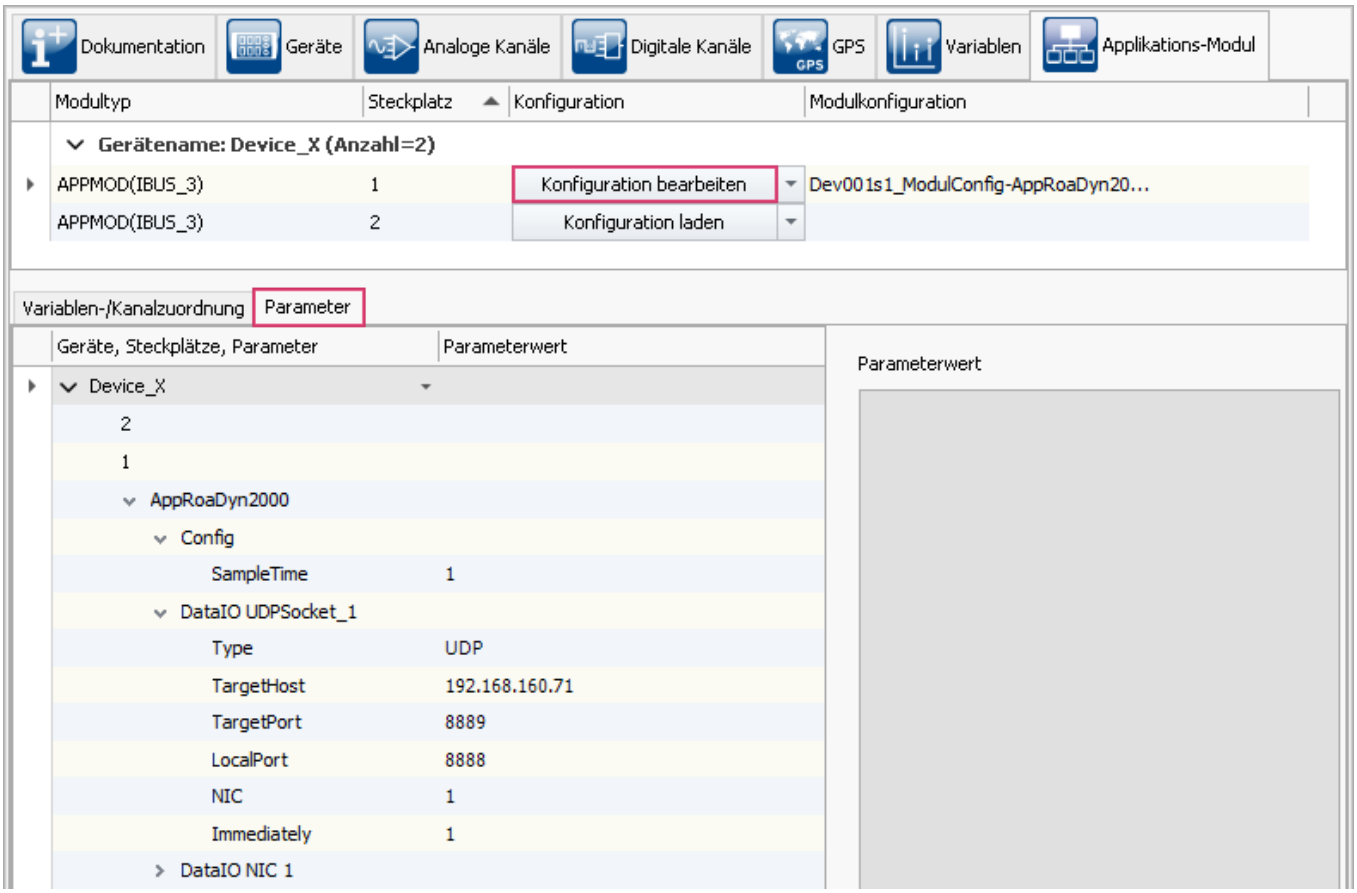


Laden einer Konfiguration für das Applikations-Modul

6.1.2 Bearbeiten der Modul-Konfiguration

Abhängig von der erstellten Applikation, sind vom Entwickler Konfigurationsmöglichkeiten 13 vorbereitet, die Sie im folgenden Dialog editieren können.

Wählen Sie dazu die Karte *Parameter* und klappen Sie die Zweige des Parameterbaumes auf:



Konfigurationsfenster

6.2 Modul-Konfiguration

6.2.1 Konfigurationsmöglichkeiten

Wichtig

Der Entwickler bestimmt den Umfang der Konfigurationsmöglichkeiten. Daher ist für jede Applikation eine Beschreibung des Entwicklers notwendig, in der die Parameter und Möglichkeiten beschrieben sind.

Einen direkter Austausch mit Komponenten des imc Gerätes wird ermöglicht mit:

- Geräte-Kanäle (analoge, digitale, Feldbusse, etc.)
- Prozessvektor-Variablen
- Display-Variablen

Hinweis

Zu beachten ist, dass es sich hier immer nur entweder um Eingänge ODER Ausgänge handelt. Bei Ausgängen (-Out) werden Ressourcen angelegt. Bei Eingängen wird lediglich hinterlegt, dass von den Ressourcen gelesen werden soll. Die Richtung legt der Entwickler fest.

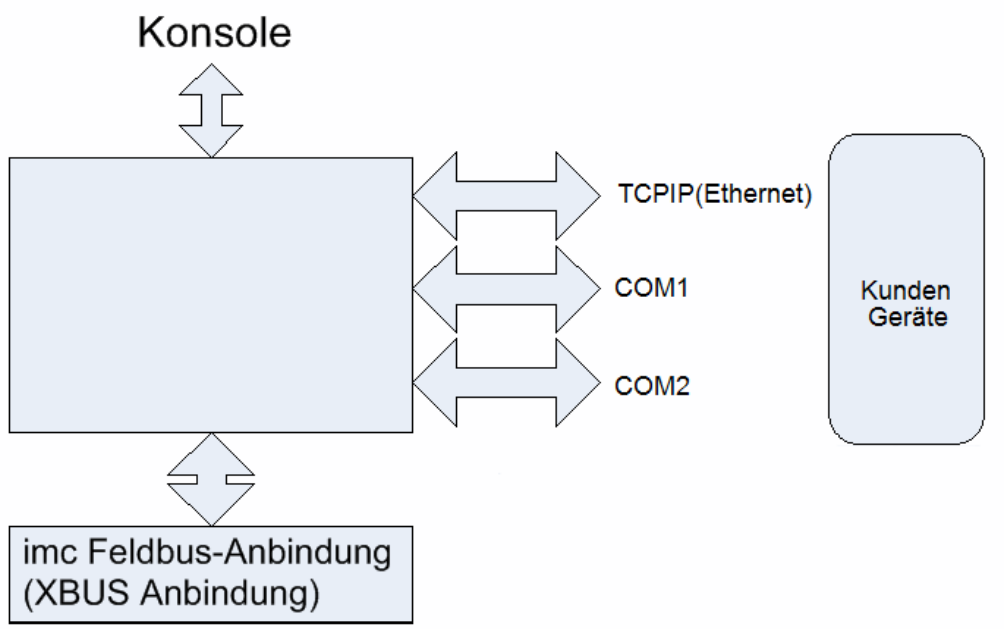
Konfigurationsmöglichkeiten	Beschreibung
Datei (in Vorbereitung)	Datei IO: Austausch mit einer vom Entwickler definierten Datei. Parameter: • Name: Name der Datei, die zum Austausch mit dem Modul verwendet wird. Der Name wird normalerweise vom Entwickler vorgegeben.
Serielle Schnittstelle (ComPort)	Serielle IO:Verbindungseinstellungen über die serielle Schnittstelle. Parameter: • Name: Name der betreffenden Schnittstelle. Systemname ("/dev/ttyPSC1",) oder eine Zahl (1 für COM1; 2 für COM2). • Bitrate: Geschwindigkeit Bit pro Sekunde • Databits: Anzahl der Datenbits • Stopbits: Anzahl der Stoppbits • Flowcontrol: Flusskontrolle() • Parity: Parität
TCP (ausgehend, bidirektional)	TCP IO: Verbindungseinstellungen über TCP. Parameter: • Host (TargetHost) IP der Zielkonfiguration • Port (TargetPort): Port der Zielkonfiguration • Address: IP der lokalen Netzwerkkonfiguration • Netmask: Subnetzmaske • Gateway: Gateway bei Bedarf
UDP (bidrektional)	UDP IO:Verbindungseinstellungen über UDP. Parameter: • Port: Quellport • Zielhost: IP des Empfängers • Address: IP der lokalen Netzwerkkonfiguration • Netmask: Subnetzmaske

6.2.2 Einträge

Einträge	Beschreibung
Display-Variable	Parameter: • Name: Der Name der verwendeten Display-Variable wird eingestellt.
Prozessvektor	Neben dem Namen werden drei weitere Parameter eingestellt: Die Einheit, die dargestellt werden soll (Unit), der Offset(Offset) und der Faktor(Factor). Parameter: • Name: Der Name der verwendeten Prozessvektor-Variable • Unit: Einheit der Prozessvektor-Variable • Offset: Offset • Factor: Verrechnungsfaktor • Init: Initialisierungswert
Kanal	Parameter: • Name: Kanalname. Wird der Name leer gelassen, wird keine imc Ressource angelegt; eine eventuell zu dem Kanal bereits angelegte imc-Ressource wird beim Löschen des Namens wieder entfernt. • comment: Kommentarfeld • Y-Axis Unit: Y-Achseneinheit () • SampleTime: Abtastzeit(). Sollten auf dem voreingestellten Wert bleiben. • SampleMode: Abtasttyp(). Sollten auf dem voreingestellten Wert bleiben.

7 Kurzanleitung: Entwicklung eines imc Applikations-Modul

Aufbau Applikations-Modul:



Je nach Ausstattung können bei der vorliegenden Version die Anschlüsse abweichen

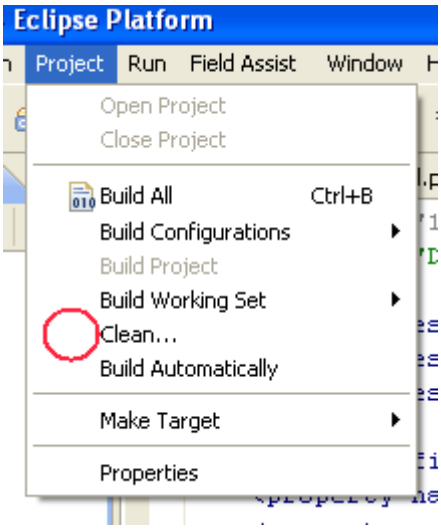
Ein Applikations-Modul kann serielle Schnittstellen (COM1, COM2) und UDP, TCP (über die vorhandene Netzwerkschnittstelle) verwenden. Das Applikations-Modul gehört zur Familie der Feldbus-Anbindungen bzw. XBUS Anbindungen und ist somit eine Ausstattungs-Option. Geräte können ab Werk mit diesen Modulen ausgerüstet werden. Ein nachträgliches Erweitern, Austauschen oder Umstecken durch den Benutzer ist nicht vorgesehen.

Es gelten die folgenden Voraussetzungen:

- Entwicklungsumgebung (eclipse, toolchains (Compiler, Linker)
- Die Entwicklererweiterung passend zu der installierten firmware Version (imc DEVICES) muss auf dem PC des Applikationsentwicklers installiert sein.
  (imcAppModDevSetup.exe 9 auf dem zugehörigen imc STUDIO Installationsmedium)
- Das Entwicklergerät muss mit demselben Modultyp ausgestattet sein, wie es später beim Kunden eingesetzt werden soll.
- Das Applikations-Modul im Entwicklergerät muss mit einer Konsole ausgestattet sein, damit die Ausgaben der Applikation (Anzeige der Trace-Informationen) beobachtet werden können.
  Einstellung am PC in Hyperterminal, Terraterm, Putty etc. ist hierzu:
  Speed: 115200, *Bytesize*: 8, *Parität*: keine, *Stopbits*: 1

Für die Erstellung eines Applikations-Moduls ist wie folgt vorzugehen:

Anmerkung: Die Beispiel-Projekte liegen im Unterverzeichnis *DemoApps* im Installationsverzeichnis der Entwicklererweiterung.
(*Standard: C:\Program Files\imc\imcAppMod*)



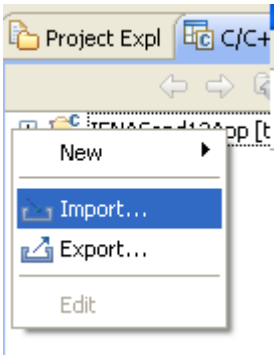
Kopieren des Template-Projektes (dieses wird immer mit der Entwicklererweiterung für das Applikations-Modul installiert) in ein neues Projekt-Verzeichnis.

Anlegen eines neuen Workspaces mit der Eclipse (die Option *Build Automatically* sollte deaktiviert werden).

In diesem neuen Workspace muss nun ein Referenzprojekt 48 hinzugefügt werden.

C:\imc\imcAppMod

Hinzufügen des neuen Projektes über "Import"

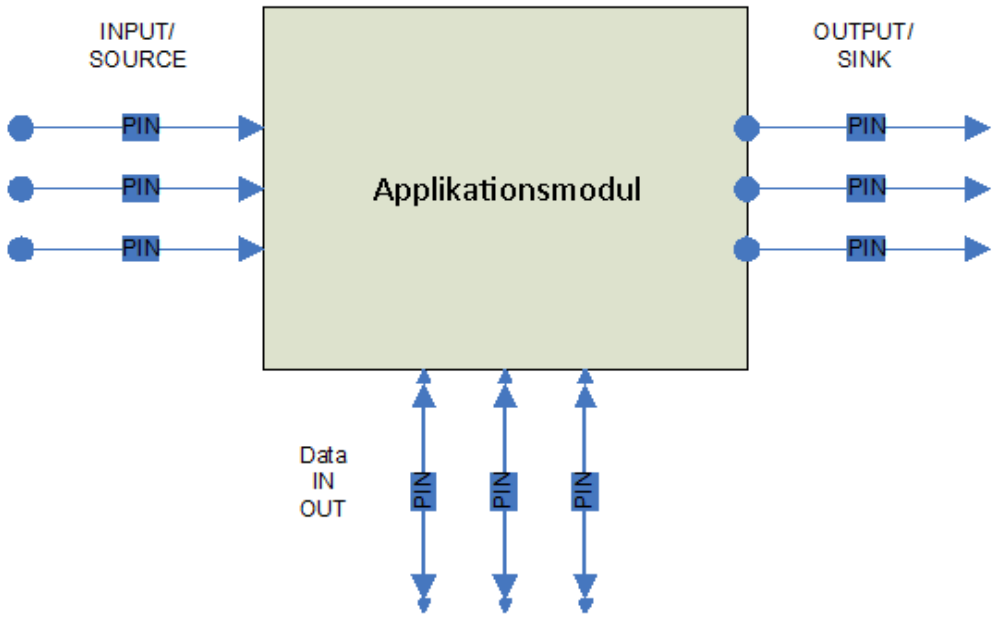


1. Es empfiehlt sich das Projekt nach dem Import umzubenennen. Es wird erstmals mit dem Namen "*Template*" eingefügt. Dies geschieht mit der Eclipse und der Renamefunktion.
2. In der Datei "*build.properties*" müssen die Buildpfade an den Installationspfad der Entwicklererweiterung 9 angepasst werden. (Beispiel siehe: "*Einrichten der Entwicklungsumgebung* 10")
3. Anpassen der Dateinamen im "*src*" Unterverzeichnis an den Namen der Zielapplikation (hier sollte ein kurzer Namen gewählt werden, wie z.B. *LightControlApp)*.
   Folgende Dateien sind umzubenennen:
   - 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. Anpassen der build.xml. Hier muss die Property (Variable) "ModulName" entsprechend gesetzt werden:
   <property name="ModuleName" value="*LightControlApp*"/>
5. Umbenennen der Klassennamen in der Include-Datei und C++Quelldatei:
   src/lightcontrolapp.h, src/lightcontrolapp.h
   Template -> *LightControlApp*

6. Erstellen der Pin-Konfiguration.
   Die Pin-Konfiguration wird innerhalb der umbenannten "*Template.pcdcl*" Datei vorgenommen. Hier werden die Prozessvektor-Variablen, Kanäle und Display-Variable deklariert, die im Modul verwendet werden sollen. Zusätzlich können hier Daten Ein- und Ausgabekanäle (COMPort, UDP, etc) definiert werden, die später durch den Anwender konfigurieren werden sollen.
7. Erstellung der ModulParameter
   Die ModulParameter werden innerhalb der "*Template.mpdcl*" Datei vorgenommen. Hier können Parameter definiert werden, die zur Laufzeit der Messung verändert werden können (TunableParameter) und für die sich Prozessvektor-Variablen nicht eignen. Diese Parameter können dann mit Hilfe des Assistenten zum laufenden Modul übertragen werden. Bei der Deklaration werden **Name**, **Typ** und der **Startwert** Angegeben. Kommentare in der Datei sind möglich (s. hierzu die Beispieldateien).
8. Implementation des Moduls
   (Editieren, kompilieren, editieren, …).
9. Erstellen eines Experimentes mit imc STUDIO, Laden und Konfigurieren des Moduls.
10. Funktion prüfen ...
    Messungen starten, je nach Fehlfunktion kann es nötig sein, dass das Gerät neu gestartet werden muss oder zumindest mit dem Assistenten das erstellte Programm erneut hochgeladen werden muss. Falls die Messung nicht startet erscheint der Fehler in der Konsole.
    Anmerkung: Derzeit kann die Fehleranalyse nur mittels der eingelesenen Daten aus dem Gerät, das angebunden werden soll(oder dessen Simulation) durchgeführt werden sowie per Textausgaben, die der Applikationsentwickler mit integriert.
11. Fehler beseitigen (wie 8), ggf. das Experiment anpassen und dann das Modul wieder hochladen ...
12. Wenn die Entwicklung des Modules abgeschlossen ist, wird mit der Eclipse dann die "Release" Version des Applikations-Moduls erstellt.
13. Gerät neu starten.
14. Das Applikations-Modul wird dann mit dem Applikations-Modul-Assistenten konfiguriert.
15. Eine Testmessung durchführen. Ist die Funktion wie vorgesehen, ist hier die Entwicklung dieser Version des Applikations-Moduls (Applikation) fertig.
    Anmerkung: Fehlfunktionen können sich auf die vielfältigste Weise zeigen. Hier muss sich der Applikant eine Teststrategie entwickeln, mit der die Funktion sichergestellt wird (Simulation der Kundengeräte, Testsignale, etc).

7.1 Pinkonfiguration

Grundsätzlicher Aufbau:



Aufbau der Pinkonfiguration

Ein "Pin" hat im Allgemeinen eine Richtung:

1. SINK (Output). Die Daten fließen in Richtung "Gerät"/"PC" aus dem Applikations-Modul.
2. SOURCE (INPUT). Die Daten fließen aus dem Gerät in das Modul.

Pins sollten für alle Ressourcen definiert werden, die der Anwender im Experiment konfigurieren muss. Ist es z.B. nötig eine serielle Schnittstelle auszuwählen und zu konfigurieren, dann muss diese als Daten Ein/Ausgabe deklariert werden. Nur dann ist es dem Anwender möglich die Kommunikationsparameter einzustellen.

Genutzte imc Ressourcen, wie Kanäle, Display- und Prozessvektor-Variablen, müssen, wenn sie genutzt werden sollen, über "PINs" deklariert werden. Der Anwender kann dann später festlegen, wie sie in seinem Experiment heißen bzw. mit welchen imc Ressourcen sie verbunden werden (sog. "Verdrahten" der Pins).

Ein "PIN", der auf imc Ressourcen (PVV, Display und Kanäle) verweist, kann nur eine Richtung (entweder "SOURCE" oder "SINK") haben (Bidirektional ist hier nicht möglich).

Hinweis

Prozessvektor-Variablen, die in einer Messkonfiguration keinen Leser zugewiesen bekommen, werden nicht im Gerät angelegt. Werden nun Pins für Prozessvektor-Variablen angelegt, die später von keiner Instanz gelesen werden, dann steht das Pin dennoch zur Verfügung. Dies dient dazu, damit die Applikation unabhängig von der Auswertung anderer Module funktionieren kann.

7.1.1 Deklaration der Pinkonfiguration

```
# imc AppMod Ein-/Ausgabe Deklarationen
; Kommentar
; Schlüsselworte beachten keine Groß und Kleinschreibung
; Werte werden mit Groß und Kleinschreibung unterstützt

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
```

Pinkonfiguration	Beschreibung
PinName (Pin Name="...")	Ein Pin hat immer einen Namen, so wie er für das Applikations-Modul heißt (hier 'Name="Device"'). Über diesen Namen kann der Applikant in seiner Applikation auf die Ressource (Pin) zugreifen (siehe Beispielprojekte).

Pinkonfiguration	Beschreibung
PinTyp (PinType="....")	Der Applikant gibt einen PinTyp vor (PinType Beispiele in den Beispielprojekten nachschauen). Mit ihm wird festgelegt, um welche Ressource es sich hier handelt. PVVar (Prozessvektor-Variable), Display-Variable, Channel (Kanäle) oder DataIO (Dateien, serielle Schnittstellen, etc) sind hier möglich. Diese Angabe kann der Anwender mit dem Assistenten nicht verändern.
Richtung(IOType="SINK")	Für Prozessvektor-Variable, Display-Variablen und Kanäle wird eine Richtung (IOType: "SOURCE" oder "SINK") festgelegt. Lediglich bei DataIO ist die Angabe rein informeller Natur. Der Applikant kann damit dem Anwender die Hauptflussrichtung der Daten auf diesem Anschluss(Pin) mitteilen. Der Anwender kann diese Information nicht verändern.
Ressourcenname (Name="Resource")	Der Ressourcenname ist ein Vorschlag und ggf. ein Hinweis, um welchen Datentyp es sich dabei handelt.
Pin Parameter	Jeder Pintyp hat für ihn spezielle Parameter, die der Anwender dem Experiment anpassen kann. Bei den Kanälen entscheidet der Applikant, auf welche Weise die Daten verwendet werden. Mit der Applikation muss dieses implementiert werden. Nur dem Applikant ist bekannt, welche Werte dabei für den Anwender auch sinnvoll sind und dieser eintragen darf/soll.
Pin Parameter für Prozessvektor-Variablen	Diese Parameter werden nur verwendet, wenn eine Prozessvektor-Variable als "SINK" deklariert wird. Im anderen Fall werden diese Werte von der Instanz in imc STUDIO, welche die Prozessvektor-Variable anlegt, festgelegt.
Valuetype	Valuetype deklariert den Datentyp der Prozessvektor-Variablen. Zur Verfügung stehen "INT16", "INT32", "TIFLOAT", "FLOAT", "ASCII" und "BIT16" (nur Kanal-Pins). **Anmerkung 1**: Wenn mit Online FAMOS über Prozessvektor-Variablen kommuniziert werden soll, dann ist dieses nur über Prozessvektor-Variablen des Typs TIFLOAT und INT32 möglich. Ebenso wird der Assistent nur die Prozessvektor-Variablen bei den SOURCE PVV auflisten, die zu einem ***Valuetype*** passen. **Anmerkung 2:** Der Valuetype "ASCII" ist nur sinnvoll bei zeitgestempelten Kanälen einsetzbar. Es können beliebige Datenfolgen geschrieben werden. Der Anwender ist nicht auf nullterminierte C-Strings beschränkt, d.h.: die Länge des Blockes wird angegeben. **Anmerkung 3**: Der Valuetype "BIT16" wird für sogenannte Port-Kanäle verwendet (Kanäle, die statt einen 16 Bit Wert 16 Einzelbits im Wort übertragen).
Offset und Factor	Mit *Offset* und *Factor* wird die Auflösung der Integerdatentypen angegeben. Bei Floatvariablen haben diese Werte keine Bedeutung.
Unit	*Unit* ist ein Informationselement und gibt die Einheit in Textform an, die bei der Darstellung der Prozessvektor-Variablen angezeigt werden soll.

Pinkonfiguration	Beschreibung
Pin Parameter für Kanäle	• **comment**: Mit comment wird ein Kommentar zu der Ressource gesetzt. Es ist möglich über dieses Kommentarfeld dem Modul den Kanal betreffend Informationen zu übergeben. • **valuetype**: valuetype deklariert den Datentyp des Kanals. Hier stehen "INT16", "INT32", "FLOAT", "TIFLOAT", "UINT16", "UINT32" und "DOUBLE64". Anmerkung: valuetype "UINT16" und "UINT32" sollten nach Möglichkeit nicht verwendet werden, stattdessen sollte isuint verwendet werden, um festzulegen, ob es sich um einen Vorzeichenbehafteten Integer handelt. • **callbackonsampling**: Über callbackonsampling wird festgelegt, ob die Applikation über eine Methode beim Erzeugen der Abtastwerte aufgerufen wird. • **monitoring**: Der Kanal wird als Monitorkanal markiert. Lediglich "SINK" Kanäle lassen sich als Monitorkanal deklarieren und können auf true oder false gesetzt werden. • **Inputstate**: Inputstate legt fest, ob ein Kanal aktiv oder passiv ist. Wird ein Kanal auf passive gesetzt, so steht er als Ressource in imc STUDIO nicht mehr zur Verfügung. Im Allgemeinen ist es nicht erforderlich, dass er auf 0 (passiv) gesetzt wird. • **isuint**: isuint wird verwendet, um festzulegen, ob ein Kanal mit vorzeichenlosen Integerwerten arbeitet. • **inputmode**: Mit inputmode wird festgelegt, ob es sich um Abtastwerte mit Zeitstempeln handelt oder ein kontinuierlicher Datenfluss ist. Für zeitgestempelte Werte wird "TIMESTAMPED" verwendet und für kontinuierliche Abtastwerte "SAMPLED". • **atoriginal**: Mit atoriginal wird die Betriebsart für die kontinuierlichen Abtastwerte eingestellt. Wird dieser Parameter auf true gesetzt, so schreibt die Applikation selbst die Abtastwerte in den Kanalfifo; ansonsten übernimmt das Framework eine Abtastung aus einem Zwischenspeicher und sorgt selbst dafür, dass genügend Daten in den Kanal geschrieben werden. • **Synchronized**: Synchronized wird mit inputmode="SAMPLED" verwendet, um einen äquidistanten Datenstrom zu gewährleisten, wenn der einkommende Datenstrom nicht mit dem Gerät synchron ist. • **sampletime und idsampletime**: idsampletime definiert die Abtastrate über einen Wert, der die eigentliche Abtastrate aus einer Tabelle entnimmt. Anstelle idsampletime kann jedoch mittels sampletime die Abtastrate in Mikrosekunden (als Fließkommazahl) angegeben werden; der angegebene Wert wird jedoch auf den zugehörigen ID Wert umgerechnet. • **DecoderBlock**: Zur Dekodierung eines Kanales wird ein Dekoder-Block mitgegeben. Dieser wird dann an eine Kanalresource angehängt. **Hinweis**: Der Block muss in "" und in eine Zeile gesetzt werden. Mehrzeilige Angaben sind nicht möglich!

Mögliche Abtastraten und ihr ID Wert

ID	Abtastrate	Wert Abtastrate (Fließkommazahl)
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

Hinweis

Derzeit wird eine Einstellung von Abtastzeiten < 200 µs durch den Pindeklerationskompiler abgelehnt und durch eine Abtastzeit von 200 µs ersetzt.

- **Unit**: Unit ist ein Informationselement und gibt die Einheit in Textform an, die bei der Darstellung der Kanaldaten angezeigt wird.
- **yoffset**: Offset
- Anzeigeskalierung - *Yscaletype, yscalefactor, yoffset, yminvalue, ymaxvalue, yref1value* und *yref2value* konfigurieren die Standardskalierung des Kurvenfensters des Kanals.
  yscaletype legt fest, wie die Skalierung berechnet wird:
  1. Verwendet *yscalefactor* ohne offset.
  2. Verwendet *yscalefactor* und yoffset.
  3. Verwendet die beiden Referenzwerte *yref1value* und *yref2value*.
  4. Verwendet die Grenzwerte *yminvalue* und *ymaxvalue*.

7.2 Abtastarten

Mit dem Parameter *inputmode* lässt sich die Abtastart *SAMPLED* und *TIMESTAMPED* einstellen (siehe *inputmode)*. Mit den beiden Parametern *atoriginal* und *synchronized* bestehen insgesamt vier Varianten die Abtastart *SAMPLED* einzustellen.

Es bestehen somit folgende Varianten, Kanäle (Eingabe, das heißt der Datenfluss geht von außen in das Gerät) mit ihren Fifos zu benutzen.

1. Die von außen kommenden Werte werden mit einem Zeitstempel in das Kanalfifo geschrieben.
   Aktiviert wird es bei einem Kanal**PIN** mit:
   Inputmode = TIMESTAMPED .
2. Von außen kommende Daten, die jedoch asynchron zum Gerät sind. Damit ein äquidistanter und synchroner Datenstrom entsteht, werden diese nicht direkt in das Fifo geschrieben, sondern "abgetastet", d.h. das einkommende Sample wird gespeichert und zum synchronen Zeitpunkt dann herausgeschrieben.
   Aktiviert wird es bei einem Kanal**PIN** mit:
   Inputmode = SAMPLED .
3. Von außen kommende Datum, die in Schüben kommen, jedoch bereits von ihrer Abtastung her äquidistant und synchron zum imc-Gerät sind. Diese werden dann direkt in den Fifo geschrieben, es wird keine weitere Snychronisation und ähnliches durchgeführt.
   Aktiviert wird es bei einem Kanal**PIN** mit:
   Inputmode = SAMPLED und *atoriginal=true* .
4. Von außen kommende Daten, bei denen gelegentlich ein Wert verloren gehen kann, werden mit Ersatzwerten versorgt, die dann in den Fifo geschrieben werden.
   Aktiviert wird es bei einem Kanal**PIN** mit:
   Inputmode = SAMPLED und *synchronizedl=true* .

7.3 Modulparameter

7.3.1 Funktion der Modulparameter

Die Modulparameter sind ein weiterer Teil der Konfiguration einer Applikation. Sie dienen zur Konfiguration der Applikation.

Die Modulparameter sind in drei Gruppen aufgeteilt:

- Parameter, die auch während der Messung verändert werden können (sogenannte "tunable" Parameter).
- Parameter, die nur beim Messstart von Bedeutung sind.
- Paramter, die die eingesetzten Schnittstellen beschreiben.

Für die "tunable" Parameter gibt es eine Funktion welche die Applikation von einer Änderung eines oder mehrerer Parameter informiert. Die Werte, die einem Parameter gegeben werden, legen den Initialwert zu Messbeginn fest.

Parametergruppen können in Blöcken zusammengelegt werden.

Parameter, die Schnittstellen konfigurieren sind nicht "tunable".

7.3.2 Deklaration der Modulparameter

```
// 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)]]
// Neu: string - Vector
// jeder string-Wert im Vector darf maximal 10 Zeichen lang sein
// String Werte, die länger sind, werden auf 10 Zeichen abgeschnitten
IMC_STRING sVectTest1[10] = [["value1" "value2"]]
// jeder string-Wert hat die maximale Länge von _MAXPATH
// (i.a. 255 Zeichen)
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)]]

// Neu: string-Matritzen
// jeder string-Wert hat die maximale Länge von _MAXPATH
// (i.a. 255 Zeichen)
IMC_STRING sMatTest1 = [["value1" "value2"] ["value3" "value3"]]
// jeder string-Wert in der Matrix darf maximal 10 Zeichen lang sein
// String Werte, die länger sind, werden auf 10 Zeichen abgeschnitten
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 Deklaration der allgemeinen Modulparameter

Paramter müssen immer unter einem Block angelegt werden. Es gibt einzelne reservierte Blocknamen:

- [Config] für den Block der nicht"tunable" Parameter.
- Alle Blocknamen die mit DataIO beginnen (z.B. [DataIO Name1234])

Alle Parameter müssen mit einer Datatypenbezeichnung eingeleitet werden.

Eine Parameterdeklaration hat dann folgendes aussehen:

IMC_UINT32 ui32Test = 100

Der Datentyp(hier IMC_UINT32), der Parametername(hier ui32test) und der Startwert(hier 100).

Folgende Datentypen stehen zur Verfügung (für Skalare, Matrizen und Vektoren):

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

Anmerkung: Innerhalb der DataIO-Blöcke wird auf die Typen verzichtet, da diese durch den jeweiligen Parameter festgelegt sind.

Der Zugriff auf die Parameter erfolgt wie folgendes Beispiel zeigt.

Beispiel Zugriff auf die 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 …
}
```

Sonderfall Vektor und Matrix Parameter:

Beispiel Beispiel einer Matrix bestehend aus String-Werten

```
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];
    }
}
```

Für die DataIO-Blöcke wird eine Tabelle (Map) aufgebaut, welche die Gruppen entsprechen zusammenfasst. Die damit verbunden IO-Schnittstellen stehen ebenso an der dort ermittelten Referenz zur Verfügung; siehe hierzu die Beispielprojekte (DemoApps).

7.3.4 Deklaration der DataIO Modulparameter

Für die DataIO Parameter gibt es folgende Gruppen von Parametern:

Parameter	Beschreibung
1. Typ UDP	a. ***TargetHost*** definiert das Gerät mit dem kommuniziert werden soll. Hier sind IP-Nummern zu verwenden. Eine Namensauflösung findet nicht statt. Der Parameter ist eine Zeichenkette. b. ***TargetPort:*** ZielPortnummer (1-65535) an den die UDP-Pakete geschickt werden sollen. c. ***LocalPort:*** Portnummer(1-65535) von dem die UDP-Pakete geschickt werden und antworten Empfangen werden. Hinweis: TargetPort und LocalPort können die selbe Nummer haben. d. ***NIC:*** Nummer der verwendeten Netzwerkschnittstelle. Derzeit gibt es nur eine (d.h.: als Nummer ist hier derzeit nur die "1" zulässig).
2. Typ TCP	a. ***TargetHost*** definiert das Gerät mit dem kommuniziert werden soll. Hier sind IP-Nummern zu verwenden. Eine Namensauflösung findet nicht statt. Der Parameter ist eine Zeichenkette. b. ***TargetPort:*** ZielPortnummer (1-65535) des TCP Ports mit dem verbunden werden soll.
3. Typ COM	a. ***Number:*** Benennt die Nummer der seriellen Schnittstelle (1 für COM1, 2 für COM2). Derzeit stehen zwei serielle Schnittstellen zur Verfügung. b. ***Bitrate****:* Bitrate, die für die Übertragungsgeschwindigkeit verwendet werden soll (9600, 19200, 115200). Es stehen die Standardwerte zur Verfügung, siehe technische Daten 60. c. ***Databits:*** Databits gibt die Anzahl der Datenbits in einem Datenbyte der Übertragung an (es stehen 5,6,7 und 8 zur Verfügung. Im allgemeinen wird die 8 eingesetzt. d. ***Stopbits***: Anzahl der Stoppbits, die verwendet werden sollen (1 und 2 stehen zur Verfügung). Die Stoppbitlänge von 1.5 wird derzeit nicht unterstützt. e. ***Parity****:* Gibt die Art des Paritätsbit an. Diese Einstellung geht nicht bei 8 Datenbits. Hier kann "even", "odd" und "none" verwendet werden. f. ***Flowcontrol****:* Einstellung der Flusskontrolle. Es kann zwischen Software-(der Einsatz der Xon/Xoff Symbole) oder Hardware-Flusskontrolle(Leitungen RTS und CTS) sowie keiner gewählt werden ("crtscts", "xonxoff" und "none").
4. Typ File	a. ***Name:*** Name der Datei, die verwendet werden soll. Dateien, die mit einer Applikation mit aufgespielt werden liegen in "/tmp" (z.B.: /tmp/MeineTestDatei.txt).
5. Typ NIC	a. ***IPAddress:*** Konfiguration der lokalen IP-Adresse der Netzwerkschnittstelle. b. ***Netmask:*** Konfiguration der Netzmaske der Netzwerkschnittstelle. c. ***Gateway:*** Wird über Gateways hinweg kommuniziert, so muss der erste Route hier eingetragen werden.

Hinweis

Bei der Deklaration der UDP und TCP Einstellungen wird ein NIC Typ mit Standardwerten automatisch angelegt. Werden diese Werte verwendet, kann auf ein gesondertes Aufführen in der Deklaration verzichtet werden (IP Adresse: "192.168.160.14", "255.255.255.0", kein Gateway("")).

Der Typ NIC trägt immer den Namen "DataIO NIC 1", wobei die Ziffer die Nummer des NIC's bedeutet. Derzeit steht nur eine Netzwerkschnittstelle zur Verfügung.

Für die DataIO Typen 1-3 steht noch der Parameter "**Immendiate**" und "**Comment**" zur Verfügung. Wird "**Immendiate**" auf "true" gesetzt, so wird der betreffende DataIO Kanal mit dem Einlesen der Konfiguration aktiviert. Wird "Immendiate" weggelassen oder explizit auf "false" gesetzt, dann obliegt es der Anwendung, den Kanal zu öffnen. "Comment" ist lediglich ein Kommentar zu dem betreffenden DataIO Kanal. Der Typ NIC führt immer zur Konfiguration der adressierten Netzwerkschnittstelle.

7.4 Aufbau eines imc Applikations-Modul/Applikation

Der Betrieb des Moduls folgt immer folgendem Ablauf:

1. Einbinden und Initialisierung des Moduls.
2. Verarbeiten der Konfiguration, Vorbereiten des Messbetriebs mit den angeschlossenen Geräten, etc.
3. Abschluss der Konfiguration nach Verarbeiten der Kanalkonfiguration.
4. Warten auf den Start der Messung
5. Empfang und Aufzeichnen von Messdaten
6. Abtasten von Kanal pv-Variablen und versenden an angeschlossene Geräte.
7. Ggf. Verarbeiten der "STOP" Meldung, wenn ein Anwender die "STOP" Taste während der Messung betätigt.
8. Ggf. Verarbeiten der "START" Meldung, wenn ein Benutzer nach automatischem Messende den erneuten "Start" auslöst.
9. Deinitialisierung und Beenden.

Dabei ist zu beachten, dass sich der vom Applikant erstellte Teil des Moduls komplett unter der Kontrolle des Applikations-Modul-Frameworks befindet. Hierzu werden Methoden in der Modulinstanz aufgerufen (Callback Funktionen), dem Betriebszustand entsprechend. Derzeit ist es nicht möglich parallele Hintergrund Prozesse (Threads/Tasks) zu starten.

7.4.1 Die Register Klasse

Für jedes Applikations-Modul ist eine Registrierungsklasse zur Registrierung beim Applikation-Framework zu erstellen. Von dieser Registrierungsklasse wird dann eine globale Instanz benötigt. Diese ist nötig, damit das Applikations-Modul beim Applikations-Framework zur Laufzeit angemeldet wird. Erst dann kann das Applikations-Framework auf das Applikations-Modul zugreifen und die Methoden des Moduls aufrufen.

Im Allgemeinen wird sie wie folgt formuliert ("Template" wird mit dem Modulklassennamen ersetzt):

```
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
};
```

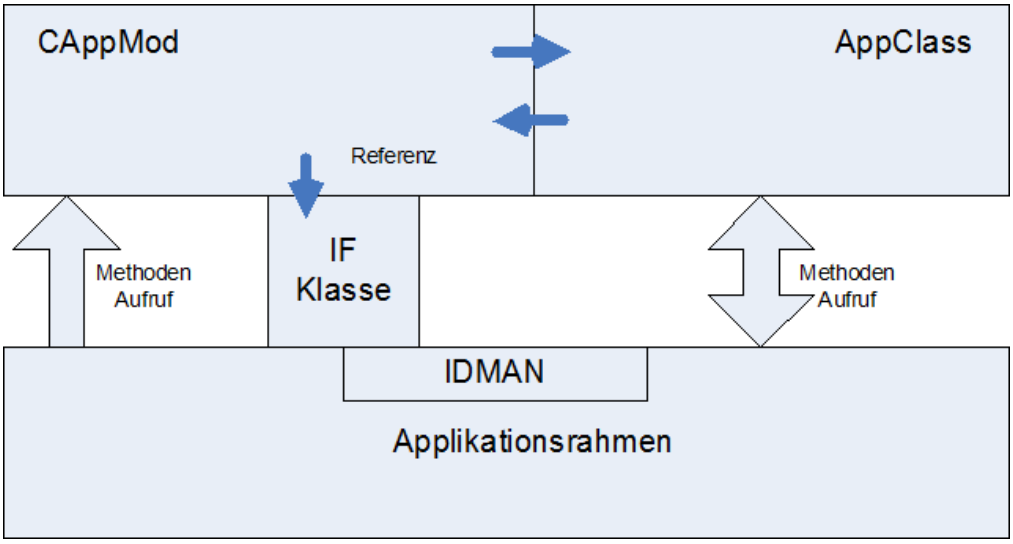
Registriert wird dann die Applikations-Modul-Klasse wie folgt mit einer globalen Instanz:

```
Template_register JustToReg;
```

7.4.2 Die Klasse CAppMod

Als "Interface"-Klasse steht die CAppMod zur Verfügung. Der Applikant leitet seine Applikationsklasse ab; diese stattet er dann mit seiner Funktionalität aus. Die CAppMod Klasse dient dem Framework als Schnittstelle zum Applikations-Modul. Die Klasse des Applikations-Modul muss dazu einige Methoden der CAppMod überladen. Für die Methoden, die nicht überladen werden müssen, besitzt CAppMod einfache Stub-Methoden, die dazu dienen den reibungslosen Ablauf mit dem restlichen Gerät während des Betriebes zu gewährleisten.

Durch die oben genannte Registrierklasse wird die Modulklasse des Applikations-Moduls beim Applikations-Framework(Applikationsrahmen) registriert.



7.4.3 Statusrückgaben von Methoden

Bis auf wenige Ausnahmen, liefern alle Methoden eines Applikations-Moduls einen Ergebniswert zurück. Das können Meldungen über Konfigurationsfehler, Statusmeldungen, etc sein. Folgende Rückgabewerte sind derzeit möglich:

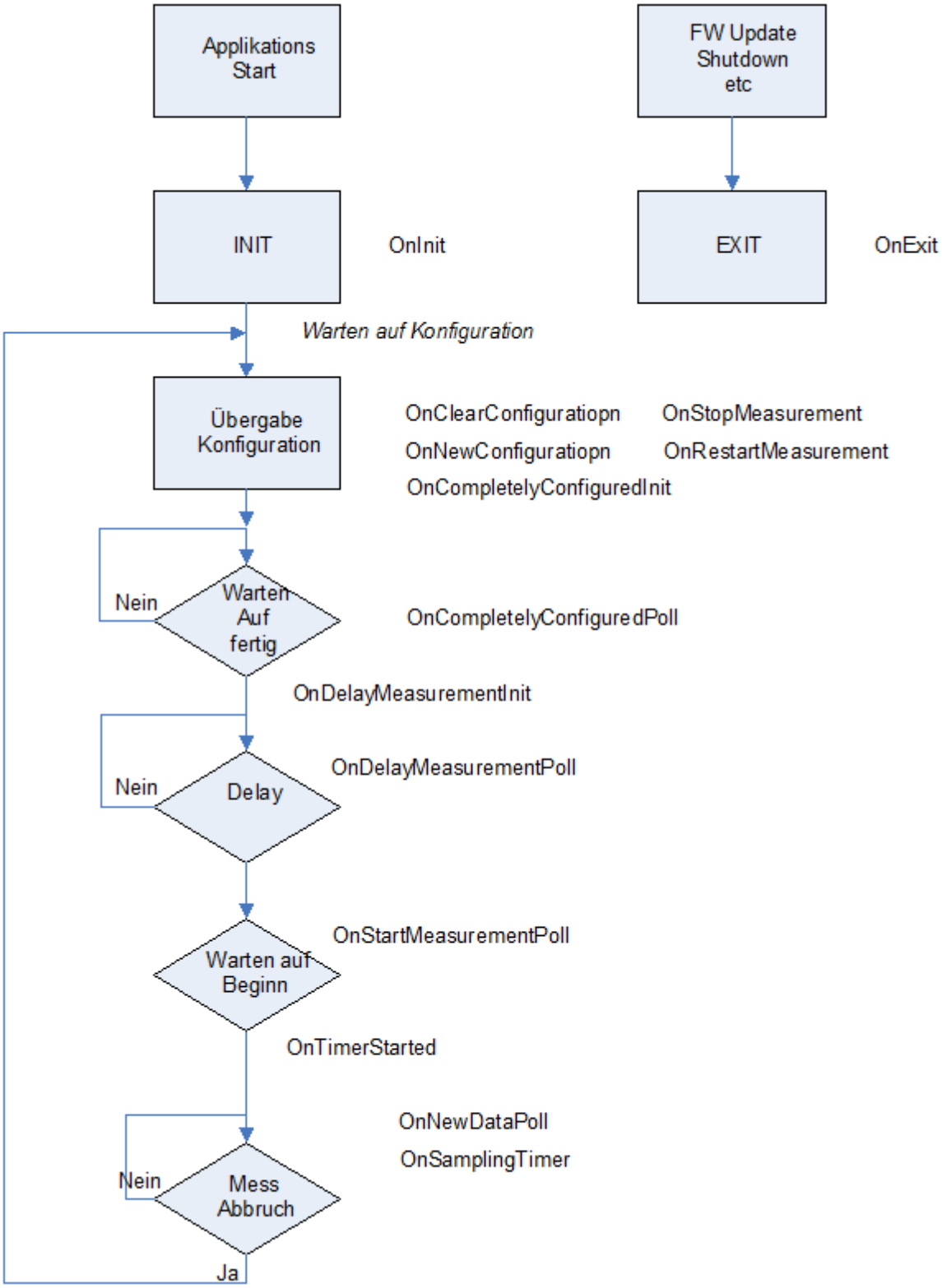
Rückgabewerte

```
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;
```

Hinweis

Die applikationsspezifischen Fehlercodes werden in den allgemeinen imc Fehlerlisten abgebildet. *APP_MOD_ERROR_BASE* entspricht -5432 und *APP_MOD_ERROR_LAST* bei –5447 (jeweils ohne Minuszeichen dargestellt). Mittels der Methode **SignalOnlineError** (siehe unten) kann auch zur Messzeit ein Fehler an den Anwender gemeldet werden.

7.5 Ablaufdiagramm einer Applikation



7.5.1 Methoden der CAppMod Klasse

```
IMC_STRING GetApplicationName();
```
GetApplicationName wird aufgerufen, um den Namen des Applikations-Moduls zu erhalten. Diese Methode ist durch den Applikanten zu implementieren.

```
APP_PIN_MAP * GetPinMap();
```
GetMap ist eine Methode, die durch CappMod definiert und implementiert ist. GetMap ist eine Dienstfunktion für den Applikanten und liefert einen Zeiger auf die eingelesene PIN Konfiguration. Es stehen die üblichen Methoden einer STL MAP zur Verfügung. Als Schlüssel dient der Name eines "PINs", der in der PIN Deklaration verwendet worden ist.

```
APP_DATAIO_MAP * GetDataIOMap();
```
GetDataIOMap ist eine Methode, die durch CappMod definiert und implementiert ist. GetDataIOMap ist eine Dienstfunktion für den Entwickler, die einen Zeiger auf die eingelesene DataIO Parameter liefert. Es stehen die üblichen Methoden einer STL MAP zur Verfügung. Als Schlüssel dient der Name des "DataIO"-Parameters, der in den Modulparameter-Deklaration verwendet worden ist (z.B.: "DataIO NAME4").

```
Node * GetModulParam(IMC_STRING & sGroupName, IMC_STRING & sParamName);
```
GetModulParam erlaubt den Zugriff auf die ModulParameter Datenbasis.

GetModulParam ist eine Dienstfunktion für den Applikanten.

```
IMC_BOOL      GetRealtimeModeNeeded( void );
```
GetRealtimeModeNeeded wird durch das Framework aufgerufen, um zu ermitteln, ob das Betriebssystem in den entsprechenden Modulen geladen werden muss.

```
eAppModError OnInit(void) = 0;
eAppModError OnExit(void) = 0;
```
OnInit wird aufgerufen, wenn das Applikations-Modul komplett instanziiert worden ist. Dies geschieht einmal bei Modulstart. In den Beispielen wird hier eine Referenz auf die PIN MAP angefordert.

OnExit wird aufgerufen, wenn das Applikations-Modul beendet wird. Sie wird maximal einmal aufgerufen.

Diese Methoden dienen dazu Funktionen zu Beginn oder zum Ende der Laufzeit auszuführen. Damit kann z.B. eine Nachricht an ein angeschlossenes Gerät zu senden.

Beide Methoden sind durch den Applikanten zu implementieren.

```
eAppModError OnNewConfiguration(void);
```
OnNewConfiguration wird durch das Framework aufgerufen, wenn eine neue Modulkonfiguration empfangen und verarbeitet worden ist. Das Applikations-Modul kann nun seinerseits die Konfiguration auslesen und nach Bedarf speichern oder verarbeiten. Welche Formen hier gewählt werden, ist nicht festgelegt. Um die Deallokation der in der PIN MAP enthaltenen Ressourcen kümmert sich das Framework.

```
eAppModError OnClearConfiguration(void);
```
OnClearConfiguration wird durch das Framework gerufen, um anzuzeigen, dass die aktuelle Konfiguration nicht mehr gültig ist. Etwaige Verweise auf Ressourcen in der PIN MAP müssen verworfen werden. Diese Methode kann und wird mehrfach aufgerufen. Die Applikation muss hier selbst dafür sorgen, dass sie Ressourcen nicht doppelt zurückgibt (z.B. so: free(pszMeldung);pszMeldung = NULL;).

void OnCompletelyConfiguredInit();

OnCompletlyConfiguredInit wird durch das Framework aufgerufen, wenn alle Konfigurationen geladen und verarbeitet worden sind. Somit befindet sich die gesamte Konfiguration im System, welche vom Modul gelesen werden kann. Dadurch, dass nach der Modulkonfiguration noch die Kanalkonfiguration an das Modul gegeben wird, können eventuell noch Arbeiten nötig werden.

Diese Methode ist nur zu implementieren, wenn für die Endverarbeitung der gesamten Konfigurationen Vorbereitungen getroffen werden müssen.

Die Konfigurationsobjekte kommen in folgender Reihenfolge:

1. IDMAN Konfiguration (füllt den idmanresourcepool)
2. ModulKonfiguration (Pinkonfiguration und Modulparameter, sowie deren Startwerte, sowie das Modulexecutable selbst.
3. Kanalkonfiguration

Im Anschluss an 3. wird das Methodenpaar aufgerufen.

eAppModError **OnCompletelyConfiguredPoll**();

OnCompletlyConfiguredPoll wird durch das Framework solange aufgerufen, wie APP_CONTINUE_POLL zurückgegeben wird. Diese Methode muss regelmäßig wieder zurückkehren, damit das Framework ggf. das Timemout der Systemsoftware zurücksetzen kann. Die Verarbeitung der Kanalkonfiguration wird dem Grundsystem erst bestätigt, wenn APP_SUCCESS oder ein Fehler gemeldet wird.

void OnDelayMeasurementInit();

OnDelayMeasurementInit wird durch das Framework aufgerufen, um dem Applikations-Modul anzukündigen, dass noch Arbeiten erledigt werden können, die für die Messbereitschaft notwendig sind. Hier sind dazu erforderliche Initialisierungen vorzunehmen.

eAppModError **OnDelayMeasurementPoll**();

OnDelayMeasurementPoll wird aufgerufen, damit das Applikations-Modul seine zur Messbereitschaft nötigen letzten Arbeiten ausführt. Die Bestätigung der Messvorbereitung wird erst gesendet, wenn APP_SUCCESS oder ein Fehler gemeldet wird.

Dieses Methodenpaar ist nur bei Bedarf zu implementieren.

eAppModError **OnStartMeasurementPoll**();

OnDelayMeasurementPoll wird aufgerufen, während auf den eigentlichen Messbeginn gewartet wird. Hier können z.B. von einem angeschlossenen Gerät Messdaten empfangen werden (z.B. wenn ein Gerät mit Verbindungsaufbau gleich anfängt Daten zu senden).

Anmerkung: Eventuelle Initialisierungen müssen mit OnDelayMeasurementInit erfolgen.

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

OnNewDataPoll wird durch das Framework während der Messung mindestens einmal pro Abtastintervall aufgerufen. Hier werden regelmäßige Arbeiten ausgeführt. Allerdings darf diese Methode nie länger als die kürzeste Abtastzeit benötigen. Diese Methode ist durch den Applikanten zu implementieren.

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

OnSamplingTimer wird durch das Framework aufgerufen, wenn ein Zeitintervall erreicht worden ist. Das Zeitintervall wurde mittels AddSamplingTimer angefordert. Hierbei wird das Objekt (pObject) zur Identifizierung angegeben.

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

OnStopTriggerDetected und OnStartTriggerDetected werden vom Framework aufgerufen, wenn ein Stopp- bzw. Start-Trigger erkannt worden ist. Das Applikations-Modul kann eine Aktion ausführen, z.B. ein Signal an ein angeschlossenes Gerät zu senden.

Da ein Kanal einen Trigger sowohl starten als auch stoppen kann, muss für jeden Kanal geprüft werden, ob der Stopptrigger mit Beginn oder Ende des Triggers ausgelöst wird. Solange ein Kanal von verschiedenen Triggern gestartet und gestoppt wird, gilt die ansteigende Flanke als Auslöser. Wird ein Kanal von einem Trigger gestartet und gestoppt, so gilt die fallende Flanke als Auslöser des Stopp-Triggers (die steigende hat dann bereits den Starttrigger ausgelöst).

```
void        OnStopMeasurement(const PAI_TIMERTICK &);
```
OnStopMeasurerment wird durch das Framework aufgerufen, wenn der Anwender des Systems während der Messung den Stoppknopf betätigt hat.

```
void        OnRestartMeasurement(const PAI_TIMERTICK &);
```
OnRestartMeasurement wird durch das Framework aufgerufen, wenn der Anwender des Systems nach dem betätigen des Stoppknopfes den Startknopf betätigt.

```
eAppModError OnSampleGenerated(FBI_CHANNEL &, const PAI_TIMERTICK &);
```
Mit OnSampleGenerated wird die Anwendung informiert, dass bei einem Kanal ein Samplewert generiert wird. Es wird das betroffene Kanalobjekt und der Zeitstempel übermittelt.

```
eAppModError OnSyncCreate(FBI_CHANNEL &, const PAI_TIMERTICK &);
```
Mit OnSyncCreate wird die Anwendung informiert, dass bei einem Kanal nicht rechtzeitig Messdaten durch das Applikations-Modul geschrieben und damit ein Ersatzwert geschrieben worden ist. Es wird das betroffene Kanalobjekt und der Zeitstempel übermittelt.

Mit diesen beiden Methoden können ggf. Maßnahmen ergriffen werden, um dem weiteren Messdatenausfall entgegen zu wirken.

```
eAppModError OnStampedChannelCheck(FBI_CHANNEL &, const PAI_TIMERTICK &);
```
Mit OStampedChannelCheck wird die Anwendung informiert, dass bei einem zeitgestempelten Kanal nicht rechtzeitig Messdaten durch die Anwendung geschrieben worden sind. Es werden das betroffene Kanalobjekt und der Zeitstempel übermittelt.

```
eAppModError SignalChangeOfModuleParam();
eAppModError SignalChangeOfModuleParam(Node &);
eAppModError SignalChangeOfModuleParam(IMC_VECTOR<Node *> &);
```
Diese Methoden werden durch das Framework aufgerufen, wenn der Anwender die Modulparameter verändert hat. Hier können Einzelupdates, Listen als auch beide zugleich gemeldet werden.

```
void AddSamplingTimer(const PAI_TIMERTICK& SamplingTimeInTicks, void*
pObject);
```
Diese Methode ist in der CappMod implementiert und dient dazu im Framework einen "Timer" zu registrieren. Während der Messung wird dann die Methode OnSamplingTimer aufgerufen. pObject dient ausschließlich zur Identifizierung auf seitens des Moduls, welches diesen Wert vergibt.

Hinweis

Diese Timereinstellung gilt für die ganze Messdauer. Eine Abmeldung ist nicht möglich. Es geschieht automatisch mit der folgenden Messvorbereitung.

Wird ein einmal angemeldeter SamplingTimer nicht mehr benötigt, kann er einfach mittels dem pObject Zeiger erkannt und in dem Falle OnSamplingTimer ohne eine weitere Verarbeitung wieder verlassen werden.

```
void RequestExtendedTimeout();
```
Diese Methode erlaubt dem Applikations-Modul den Timeout zu verlängern. Dies kann z.B. beim Abarbeiten der Konfiguration in OnNewConfiguration benutzt werden, wenn komplexe Operationen mit einem angeschlossenen Gerät nötig sind. Der Einsatz dieser Funktion sollte allerdings nach Möglichkeit vermieden werden.

```
void SignalOnlineError(eAppModError Error);
```
Hiermit kann die Applikation melden, wenn während der Messung ein Fehler aufgetreten ist. Es ist dem Applikationsentwickler überlassen, ob die Anwendung weiter Daten schreibt. Das Framework selbst wird die betreffenden Methoden der Applikation weiter aufrufen.

7.6 Pin MAP

Hier werden Objekte der Klasse CAppPin über ihrem Namen abgelegt. Die CAppPin Klasse ist die Basisklasse für die "Pins".

Die wichtigsten Methoden dieser Klassen:

Lesen und Schreiben von Werten von und an einen 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 );
```

Hinweis

Derzeit werden diese Methoden ausschließlich durch PVVar PINs unterstützt. Display-Variablen kennen davon **readFloat** und **writeFloat**.

Die folgenden Funktionen dienen dem Schreiben in Kanäle. Der Zeitstempel ist nur bei Kanälen mit zeitgestempelten Werten von Bedeutung. Bei allen wird ein Dummy gebraucht. Der Wert darin wird dabei dann ignoriert.

Schreiben in Kanäle

```
/* 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);
```

Methoden zum schreiben mehrerer Abtastwerte in einem Aufruf:

```
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);
```

Hinweis

Zur Kennzeichnung des "ungültigen" Aufrufes liefern obige Methoden false; Ist ein Wert erfolgreiche gelesen oder geschrieben worden wird true gemeldet.

1. Zum Schreiben mehrerer Abtastwerte wird wie folgt vorgegangen: Instanz der jeweiligen CAppModSampleXXX und Größe anlegen (new CAppModSampleXXX(N);)
2. Abtastwerte mittels
   putSamples(IMC_XXX * pSamples, int at, int n);
   putSample(IMC_XXX Sample, int at);
   in die Instanz schreiben.
3. Mittels der jeweiligen Schreibmethode am Pin diese in den Kanal(FIFO) schreiben. (PinEntry.writeXXX(Tick,CAppModSampleXXXInstanz);)

Folgende Methoden stehen für das Lesen aus KanalFIFOs zur Verfügung:

```
// 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);
```

Liefert über eine Referenz einen Pointer auf die jeweilige Ressource.

1. bei einer ProcessVektorVariable den Zeiger auf die Instanz des PinKonfigObjektes (das PinKonfigObjektes erlaubt den Zugriff auf Konfigurationsdaten der betreffenden ProcessVektorVariable).
2. bei einem Kanal wird ein Zeiger auf die Instanz des PinKonfigObjektes ermittelt (das PinKonfigObjektes erlaubt den Zugriff auf Konfigurationsdaten des betreffenden Kanals).

GetName();

Liefert den Namen des PINS.

ConnectType();

Liefert den PinTyp;

GetType()

Liefert die Richtung (SINK oder SOURCE).

GetValueType()

Liefert den Datentyp (INT16, UINT16, etc).

Mit dem **PinObjekt** aus der **PinMap** können direkt mit den oben genannten Methoden die Daten in eine imcResource geschrieben werden. Zum Schreiben an Kanalressourcen sind die Methoden zu verwenden, die einen Zeitstempel mit übergeben bekommen. Dieser ist jedoch nur bei zeitgestempelten Kanälen von Bedeutung. Bei allen anderen Kanalbetriebsmodi wird der übergebene Wert ignoriert.

Wichtig ist, dass bei Prozessvariablen und Kanälen nur die Methode verwendet wird, die dem Datentyp entspricht. Hier ist es wichtig, dass beim Lesen einer PVVariable, (die von einem Online FAMOS-Programm geschrieben wird), die Werte als *TIFLOAT* gelesen werden. Die betreffende Methode rechnet den Wert in einen *IEEE Float* um, mit dem die Applikation dann arbeiten kann.

Beim Lesen aus FIFOS (Kanäle) gilt es zu beachten, dass kontinuierlich die Methode ***getFIFOReaderState*** an dem betreffenden Kanalobjekt aufgerufen werden muss. Nur so ist sichergestellt, dass ein FIFO vom System korrekt behandelt wird. Dies gilt für alle KanalPins, die mittels Verweis auf eine existierende Ressource angemeldet sind.

Für das Lesen von zeitgestempelten Kanälen gilt derzeit zusätzlich, dass nur ein Abtastwert pro Aufruf ausgelesen werden kann.

Die Methoden zum Lesen aus dem FiFo liefern folgende Ergebniswerte

```
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*** bedeutet, dass die angeforderte Anzahl von Abtastwerten noch nicht im FIFO zu finden ist. Beispiel: Es wird versucht 100 Abtastwerte aus dem FIFO zu lesen (Instanz der Klasse(n) CappModSample... (100)). *LESS_READ* meldet, dass weniger als 100 Abtastwerte in das Objekt geschrieben worden sind.
- ***NOT_READ*** wird gemeldet, wenn die Methode *read*... mit *exact=IMC_TRUE* aufgerufen worden ist. In dem Fall waren weniger als 100 Samples im FIFO und es wurden keine Abtastwerte gelesen. NOT_READ wird in anderen Situationen gemeldet, wenn keine Abtastwerte in das Objekt übertragen worden sind. Wurde das Objekt komplett befüllt, wird *NOERROR* gemeldet.
- ***FULL*** wird gemeldet, wenn beim Lesen mit *exact=IMC_TRUE FIFO_FULL* gemeldet wird. Es werden keine weiteren Daten mehr in den FIFO Puffer geschrieben werden. Die Restdaten im FIFO müssen mit *exact=IMC_FALSE* abgerufen werden. Nur so ist sichergestellt, dass das zum Kanal gehörende FIFO vollständig geleert wird und der Status im Gerät korrekt gesetzt wird.
- ***NOMEM*** wird gemeldet, wenn beim Lesen nicht genügend Speicher zur Verfügung steht.
- ***OVERFLOW*** zeigt an, dass das FIFO übergelaufen ist; es wurden mehr Daten in das FIFO geschrieben, als innerhalb der Pufferdauer abgeholt worden sind. Das FIFO muss dann mittels reSync am Kanalobjekt synchronisiert werden. reSync meldet dabei die Anzahl der verlorenen Abtastwerte. Im Falle eines zeitgestempelten Kanales wird nur die Anzahl der FIFOWorte gemeldet.

GetFIFOReaderState liefert mittels *eAppModReaderState* folgende Meldungen:

- ***APPMOD_FIFO_ERROR*** : Ein Fehlerzustand wird gemeldet.
- ***APPMOD_FIFO_IDLE:*** Das FIFO ist im IDLE Zustand und es können keine Abtastwerte gelesen werden. In diesem Zustand DARF auch nicht versucht werden Abtastwerte aus dem FIFO abzuholen.
- ***APPMOD_FIFO_OVERRUN:*** Zustand des FIFOs nach einem Überlauf. Dieser Zustand wird erst verlassen, wenn reSync() aufgerufen wird.
- ***APPMOD_FIFO_DONE***: Beim Ende der Datenaufzeichnung (MessStopp, StopTrigger) ist der Zustand kurzzeitig im DONE Zustand. Das heißt, alle Daten wurden aus dem FIFO abgeholt. Es wird gewartet, dass das FIFO wieder in den IDLE Modus wechselt.
- ***APPMOD_FIFO_TRIGGERED***: Das FIFO ist im Zustand, in dem Daten gelesen werden können.

7.7 DataIO MAP

Die **DataIO Map** (*GetDataIOMap()*) verhält sich wie die **PinMap**. Ebenso wie die PinMap ist diese eine STLMAP. Im Unterschied enthält diese Instanzen der Klasse CappModDataIO. Abruf entsprechend mit "find".

Die Abfrage der ***DataIOMap*** nach einem DataIO Eintrag ("DataIO Name3") liefert eine Referenz (Pointer) auf eine Instance der Klasse CappModDataIO; (Hinweis: die DataIOMap ist eine STL Map).

Die CappModDataIO stellt dem Anwender folgende Methoden zur Verfügung:

```
void GetResource(PCComm &, ParamVect * &);
```
Liefert einen Zeiger auf die Instanz auf das dazugehörige CComm Objekt sowie einen Vektor mit Stringpaaren.

```
void GetResource(PCComm &);
```
Liefert einen Zeiger auf die Instanz auf das dazugehörige CComm Objekt.

```
const IMC_STRING GetDataIOName();
```
Liefert den Namen des DataIO Kanals.

```
const IMC_STRING GetDataIOType();
```
Liefert den Typ eines DataIO Kanals ("UDP", "TCP, etc).

```
const IMC_STRING GetDataIOConnect();
```
Liefert den einen String, der die Verbindung des DataIO Kanals beschreibt.

```
const IMC_STRING GetDataIOComment();
```
Liefert den Kommentar eines DataIO Kanals.

7.7.1 DataIO Klasse (CCom)

Um eine einheitliche Bedienung der Datenein-/ausgabe zu schaffen gib es die Ccomm Klasse. Diese versteht sich als eine Factory-Klasse. Zum einen registrieren sich Kommunikationsobjekte mit ihr. Anhand des Namens, mit dem sie verbunden werden, erkennt die CComm den Typ und legt ein passendes Objekt an. Derzeit stehen vier Typen zur Verfügung:

Dateien, Serielle Schnittstellen (COMPORT), UDP Socket, TCP Socket.

Alle Methoden liefern ein Ergebnis zurück. Die möglichen Ergebniswerte beschreibt die Enumeration eCommError. Es werden folgende Fehlercodes abgebildet:

```
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 Beschreibung der Methoden der CCom Klasse

```
eCommError open();
```

Öffnet die Datei/Socket/COMPort. Die nötigen Parameter werden beim Konstruieren das Objektes mit angegeben. Diese müssen für PINs aufgerufen werden, wenn der Kanal über die immediate Einstellung noch nicht geöffnet worden ist.

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

Es wird mit der gleichen Objektinstanz die Datei/Socket/etc. neu geöffnet (der alte Kanal wird vorher geschlossen); Eine Änderung des Kommunikationstyps ist dabei möglich.

```
eCommError close(void);
```

Schließt einen Kommunikationskanal.

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

Beide Methoden lesen Daten aus dem Kommunikationskanal ohne zu warten. Wenn der Kanal die angeforderte Menge an Daten nicht liefern kann, wird COMM_LESS_DATA gemeldet. Es wird nie auf Daten gewartet. In den übergebenen IMC_STRING wird der Absenderhost vermerkt (nur bei 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 );
```

Zwei Methoden zum Lesen unter dem Einsatz eines Timeouts. Es wird für die vorgegebene Zeit auf Daten gewartet. Werden nicht hinreichend viele Daten innerhalb des Timeouts empfangen, so wird COMM_READ_TIMEOUT zurückgemeldet. Für den IMC_STRING Parameter gilt das, was bei den anderen "read" Methoden bereits genannte.

```
eCommError BuffLevel();
```
Stoßen die Methoden zum Lesen aus einem Kommunikationskanal auf zu wenige Daten oder werden mit einem timeout (in us) beendet, so kann mit *BuffLevel* erfragt werden, wie viele Daten bereits gelesen werden konnten.

```
eCommError write(IMC_UINT32 wsize, void * , IMC_UINT32 & rWritten);
eCommError write(IMC_UINT32 wsize, void * , IMC_STRING & , IMC_UINT32 & rWritten);
```
Methoden zum Schreiben von Daten in einen Kommunikationskanal.

```
eCommError seek(IMC_UINT32 type, IMC_INT32 offset);
```
Steuert die angegebene Position in einer Datei an. Auf ComPorts und Sockets ist diese Methode nicht anwendbar.

```
eCommError flush(void);
```
Leert die Zwischenpuffer des Datenkanals.

```
CHECK_RESULT check(void);
```
Prüft, ob gelesen oder geschrieben werden kann. Es kann so auf ein Blockieren der Lese /Schreib-Operationen verzichtet werden. Blockieren ist auch nicht zulässig, da sonst das Applikations-Modul-Framework nicht mehr richtig arbeiten kann.

`CHECK_RESULT` ist eine Datenstruktur über die der aktuelle Zustand des Kommunikationskanals gemeldet wird:

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

```
eCommError SetChannelPar(const sParamsValue &);
```
Setzt Parameter des Kommunikationskanals, wenn diese nach dem Anlegen des Objektes nötig sind (z.B. setzen eines ARP Eintrages im System). `sParamsValue` ist ICM_STRING-Paar, das jeweils den Parameter als IMC_STRING sowie den Wert als IMC_STRING.

Für die COMports bestehen folgende Parameter:

- bitrate (50-230400 in den üblichen Schritten. Ab der Firmware Version (imc DEVICES) 2.8R6 auch 14400 und 28800)
- bytesize (5,6,7,8)
- stopbits(1,2)
- parity("none", "even", "odd")
- flowcontrol ("none", "crtscts", "xonxoff")

Für UDP Sockets besteht folgender Parameter:

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

7.8 Kontrollblock

Für den gesonderten Zugriff auf Hardwareressourcen des Moduls besteht eine Kontrollblockklasse, auf welche die Applikation zugreifen kann.

Derzeit stehen folgende Funktionalitäten zur Verfügung:

- Abfrage der FPGA Version
- Abfrage der FPGA Variante
- Abfrage der Taktbasis für den Ausgabetaktgenerator
- Einstellen des Ausgangstaktes
- Aktivieren und Deaktivieren der Ausgabesignaleinheit
- Steuerung von sieben Ausgabeleitungen

Hierzu die zu überladenden Methoden:

```
IMC_BOOL getFPGAVersion(IMC_UINT16 & Version);
```
Abfrage der Version.

```
IMC_BOOL getFPGAVariant(IMC_UINT16 & Variant);
```
Abfrage der Variante.

```
IMC_BOOL getClockBase(IMC_UINT32 & ClockBase);
```
Abfrage des Basistaktes der Taktausgabeeinheit.

```
IMC_BOOL setCLOCKRate(IMC_UINT32 ClockRate);
```
Einstellen des Ausgabetaktes Gerätesynchron

```
IMC_BOOL setAPPSIGBit(IMC_UINT8 BitNo, IMC_BOOL bOn);
```
Setzen einer Ausgangsleitung (Setzen folgt mit der steigenden Taktflanke des eingestellten Taktes).

```
IMC_BOOL enableAPPSIG();
```
Aktivieren der Signalausgabeeinheit.

```
IMC_BOOL disableAPPSIG();
```
Deaktivieren der Signalausgabeeinheit.

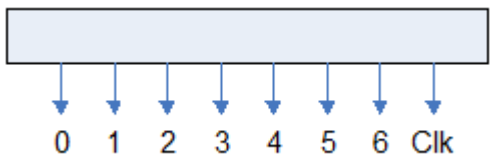
```
IMC_BOOL selectRS485(IMC_UINT8 PortNo, IMC_BOOL brs485enable);
```
Aktivieren des RS485 Modus der seriellen Schnittstelle. Dies steht nur zur Verfügung, wenn ein Applikations-Modul mit RS485 Ausstattung eingesetzt wird.

```
IMC_BOOL enableRS485Sender(IMC_UINT8 PortNo, bEnable);
```
Aktivieren der RS485 Sendeleitung.

Hinweis

Soll der RS485 nur gelesen werden (RS422), dann kann auf das Aktivieren des RS485 Moduls verzichtet werden. Dieser wird nur für das Senden benötigt. Weiterhin stehen derzeit nur die RS232 Bitraten zur Verfügung.

7.8.1 Bedienung der Ausgabeeinheit



Soll die Ausgabeeinheit verwendet werden, muss die Einheit mittels enableAPPSIG aktiviert werden. Erst dann wird das Taktsignal ausgegeben und die Steuersignale nach draußen gelegt. Die Signale sollten im Voraus so eingestellt werden, dass sie ihre Ruheposition einnehmen; entsprechende Pullups/Pulldowns, die an den Leitungen angeschlossen sind, müssen beachtet werden.

7.9 SSI-Controller

Der SSI-Controller steht derzeit nur auf der PBUS_MPC_1 Hardware zur Verfügung. Sollten Sie ein anderes Modul eingebaut bekommen haben und benötigen den SSI-Controller so setzen sie sich mit uns in Verbindung. Es stehen zwei SSI-Controller zur Verfügung (Angabe für SSIPort: 1 oder 2).

Sollte die Funktionalität nicht verfügbar sein, dann wird ein Fehler zurückgegeben.

7.9.1 Bedienung des SSI-Controllers

Der SSI-Controller wird über den folgenden Satz an Methoden des Kontrollblocks bedient. Bei erfolgreicher Verarbeitung wird **APP_SUCCESS** zurückgegeben.

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

Hiermit wird der SSI-Controller an die Leitungstreiber geschaltet. Bitte den weiter unten stehenden Hinweis beachten.

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

Hiermit wird der SSI-Controller von den Leitungstreibern getrennt. Dies sollte immer getan werden, wenn der SSI-Controller nicht mehr verwendet werden soll.

```
eAppModError configSSIController(IMC_UINT8 SSIPort,
      IMC_UINT32 Abtastrate, IMC_UINT16 SizeOfSample,
      IMC_UINT16 ValidBitsOfSample, IMC_UINT32 Schnittstellentakt,
      IMC_SSI_CLOCKTYPE SSIMode);
```

Hiermit wird der SSI-Controller konfiguriert.

Mit SSIPort wird der SSI-Controller ausgewählt.

Mit Abtastrate wird die Abtastrate, mit der ein angeschlossener Sensor abgefragt wird, eingestellt. Hier gilt zu beachten, dass diese immer für beide SSI-Controller gilt und auch immer die zuletzt eingestellte Abtastrate verwendet wird.

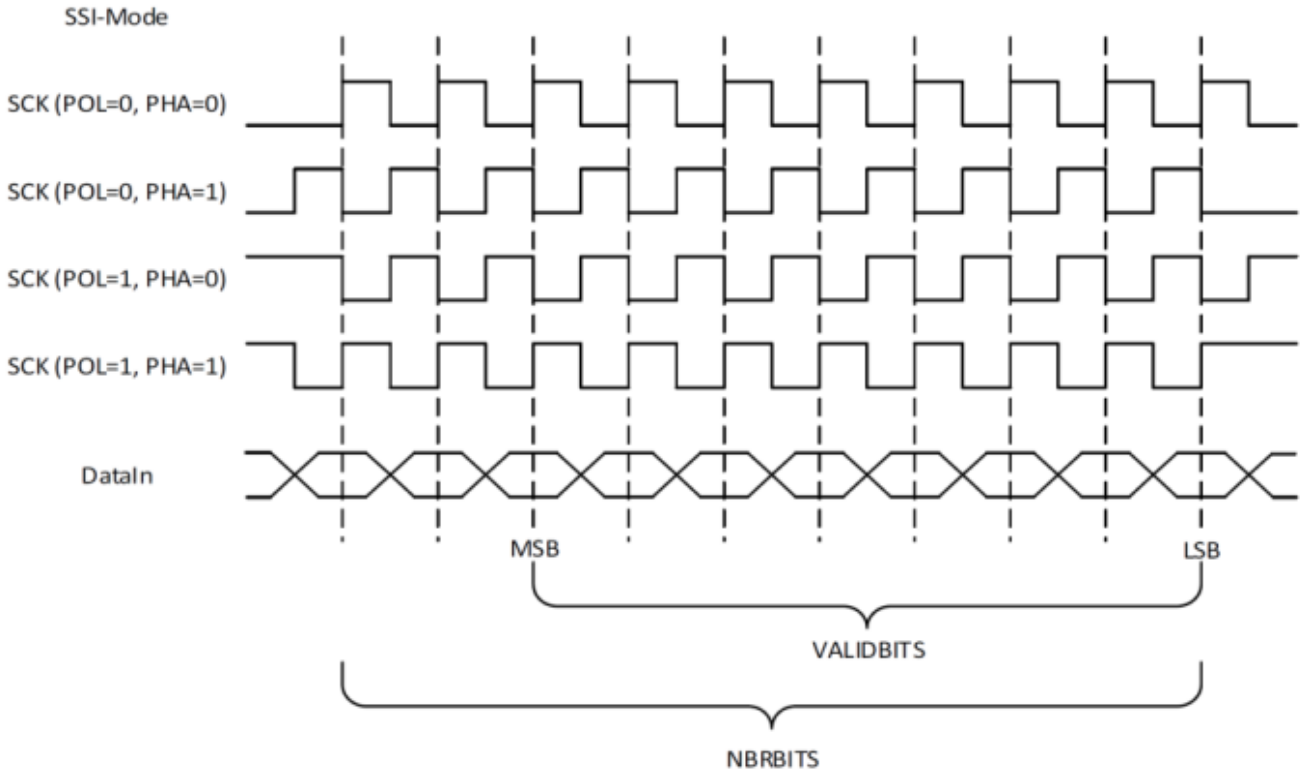
SizeOfSample gibt die Anzahl an Bits an, die mit einem Abtastvorgang von dem angeschlossenen Sensor abgeholt werden müssen.

ValidBitsOfSample gibt die Anzahl an Bits an, die in einem Sample gültig sind. Diese Angabe kann nur kleiner oder gleich als SizeOfSample sein.

Mit Schnittstellentakt wird eingestellt, mit welchem Bittakt der Sensor abgefragt wird.

```
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 gibt an, wann der Sensor die Daten auf die Datenleitung gibt. Siehe hierzu folgende Tabelle:




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

Hiermit wird der SSI-Controller angewiesen Samples mit dem Schnittstellentakt von einem Sensor abzufragen. Dies wird im allgemeinen beim Messstart einmal gemacht. Hierbei gilt es zu beachten, dass kein Sample zum Zeitpunkt T0 entsteht. Diese Methode kann auch mit SSIPort auf 0 aufgerufen werden, in diesem Fall werden dann beide SSI-Controller entsprechend angewiesen.

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

Hiermit wird der SSI-Controller angewiesen das Abfragen des Sensors einzustellen. Diese Methode kehrt erst zurück, wenn ein eventuell laufender Sampleabruf abgeschlossen ist.

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

Abfruf der im SSI-Cotroller angelaufenen Abtastwerte.

SSIPort nennt den SSI-Controller, der abgefragt wird.

MaxSamples gibt an, wieviel Platz in dem übergebenen SampleArray ist.

pSampleArray übergibt ein Feld in den Abtastwerte geschrieben werden sollen.

NumberOfSamples wird mit der tatsächlich gelesenen Anzahl an Abtastwerten gefüllt.

Wird ein FIFO-Überlauf festgestellt, dann wird das Lesen des FIFOs abgebrochen und ein APP_MOD_HWFIFO_OVERRUN gemeldet. In dem Fall steht in NumberOfSamples die bis dahin gelesenen Abtastwerte.

7.9.2 Aufbau 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 und TimeBlock sind BCD-Kodiert

Beispiel

DateBlock(Jahr,Monat): 0x1404,
TimeBlock(Tag,Stunde,Minute,Sekunde): 0x28112213)

Die Applikation muss selbst die "ungültigen" Bits eines Samples ausmaskieren; der SSI-Controller übernimmt dies nicht.

7.9.3 Hinweis zur Initialisierung

Aufgrund der Anschaltung des SSI-Controllers muss folgende Abfolge in der Initialisierung eingehalten werden. Bitte auch beachten, dass ein teil der RS485 Methoden verwendet werden müssen.

```
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);
```

Hinweis

- Wichtig ist, dass selectSSIControler als letztes aufgerufen wird.
- Soll die im Ausgangstreiber vorhandene Terminatorfunktion verwendet werden, so ist die enableRS485Terminator Methode wie weiter oben beschrieben aufzurufen.

7.10 Modulparameter

Modulparameter der Applikation sind grundsätzlich in Parametergruppen organisiert. Um ein Modulparameter vom Framework zu lesen, wird zuerst eine Referenz auf den generischen Datentyp "Node" über die Framework-Methode.

"GetModulParam(IMC_STRING& sGroupName, IMC_STRING& sParamName)" und objParameter& objParamKey = *pNode) hergestellt. Anschließend kann der Wert des Modulparameters gelesen werden. Durch den Zuweisungsoperator wird eine entsprechende cast-Operation ausgeführt (z.B. kann kein passender cast Operator zugeordnet werden -> es wird eine Exception ausgelöst (bad cast)).

Siehe Beispiel unten für einen Modulparameter innerhalb der Parametergruppe "IENA", mit dem Namen "Key" und dem Datentyp "IMC_UINT16".

Beispiel Modulparameter

```
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 …
}
```

Sonderfall Vektor und Matrix Parameter: Beispiel einer Matrix bestehend aus String-Werten

Beispiel Matrix

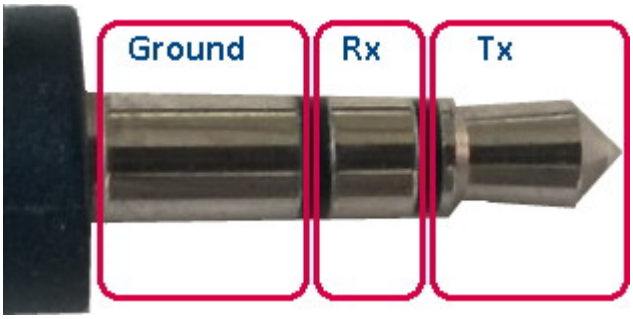
```
IMC_STRING sGroup("IENA");
IMC_STRING sParam("szMatrixTest ");
Node* pNode = GetModulParam(sGroup, sParam);
objParameter& ObjParam_szMatrixTest = pNode;
unsigned int rows = ObjParam_szMatrixTest.getDimArrayFirst();
unsigned int cols = ObjParam_szMatrixTest.getDimArraySec();

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

8 Debuganschluss

Für die Ausgabe von Debugmeldungen einer Applikation ist eine kleine 3.5mm Klinkenbuchse auf der Frontblende des Applikations-Moduls angebracht. Diese wird mittels einer passenden Leitung mit einem COM Port des PC's verbunden (es kann auch ein USB-Serial Adapter verwendet werden).

Mittels einem Terminalemulator (z.B. Putty), welcher auf 115200bps, 8N1, Flusskontrolle abgeschaltet (weder RTS/CTS noch XON/XOFF) eingestellt ist, können die Ausgaben dann betrachtet werden.

Sollte eine solche Leitung nicht zur Verfügung stehen, lässt sie sich gemäß folgenden Belegungsbildes anfertigen:



Anschluss zur DSUB-9 Buchse
Pin 5 -> Ground
Pin 2 -> Tx
Pin 3 -> Tx

9 Tutorium

Dieses Tutorium zeigt die Vorgehensweise bei der Erstellung einer Applikation mit dem Applikations-Modul.

Anhand eines Templates wird eine neue Applikation erstellt. Im Beispiel erfolgt die Umrechnung von Celsius zu Kelvin.

9.1 Installation

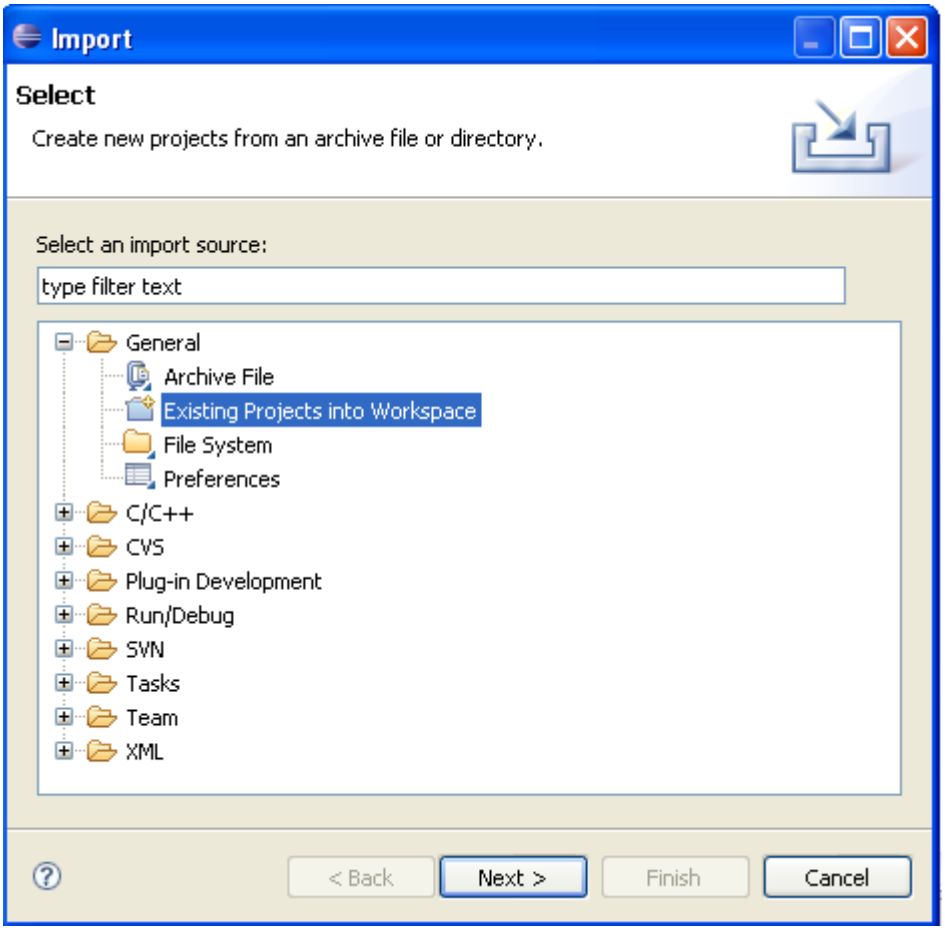
Installieren Sie die Entwicklungsumgebung entsprechend der Anleitung 9.

Hinweis

Es muss ein Installationspfad verwendet werden, der keine Leerzeichen enthält.

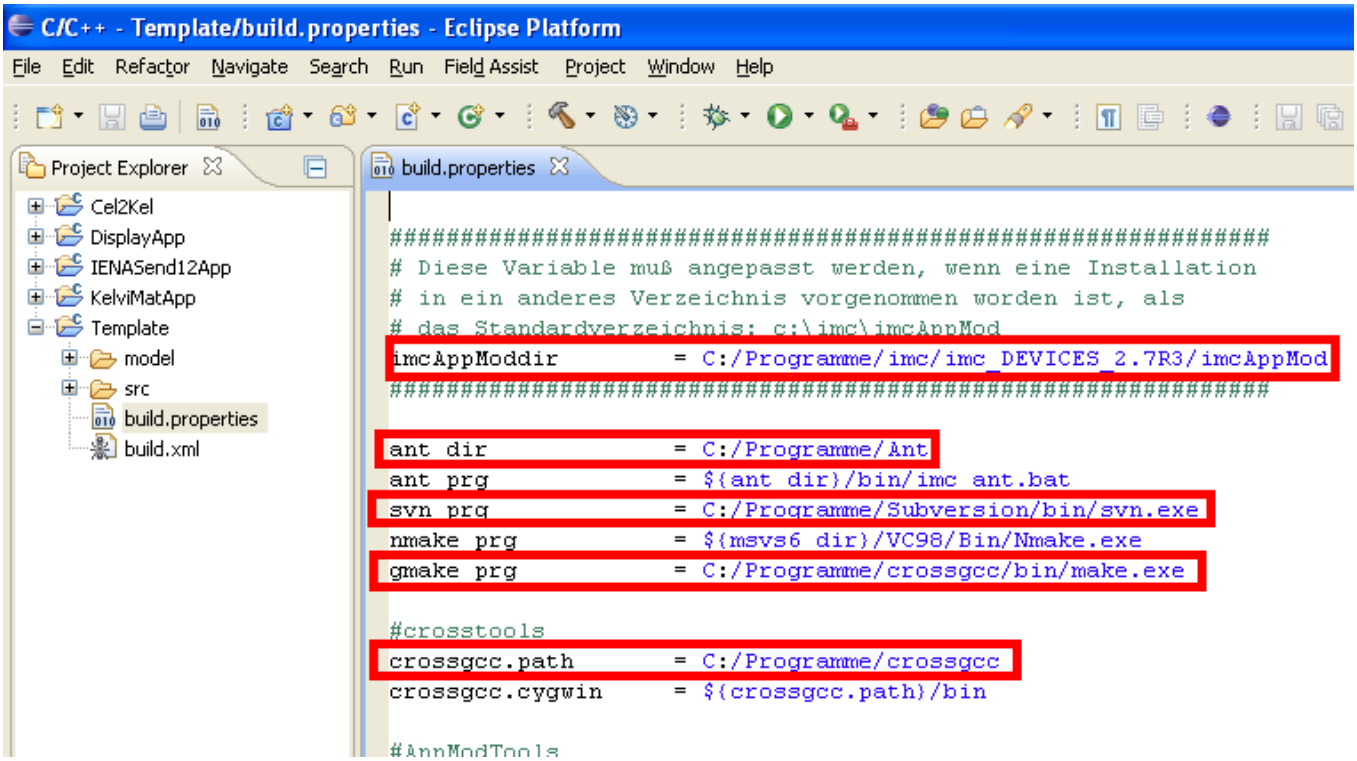
9.2 Öffnen von Projekten in Eclipse

- Kopieren Sie die **Beispielmodelle** in ihr Eclipse Workspace
- Das Beispielmodell finden Sie unter dem Verzeichnis wo Sie die "*imcAppModDevSetup* 48" installiert haben
- Starten Sie **Eclipse**
- Deaktivieren Sie den "*Build Automatically*". Siehe auch hier 17.
- Fügen Sie die Beispiel Projekte ihrem Eclipse Workspace hinzu
- Verwenden Sie dazu die Import Funktion -> "Existing Projects into Workspace" und wählen ihren Workspace aus

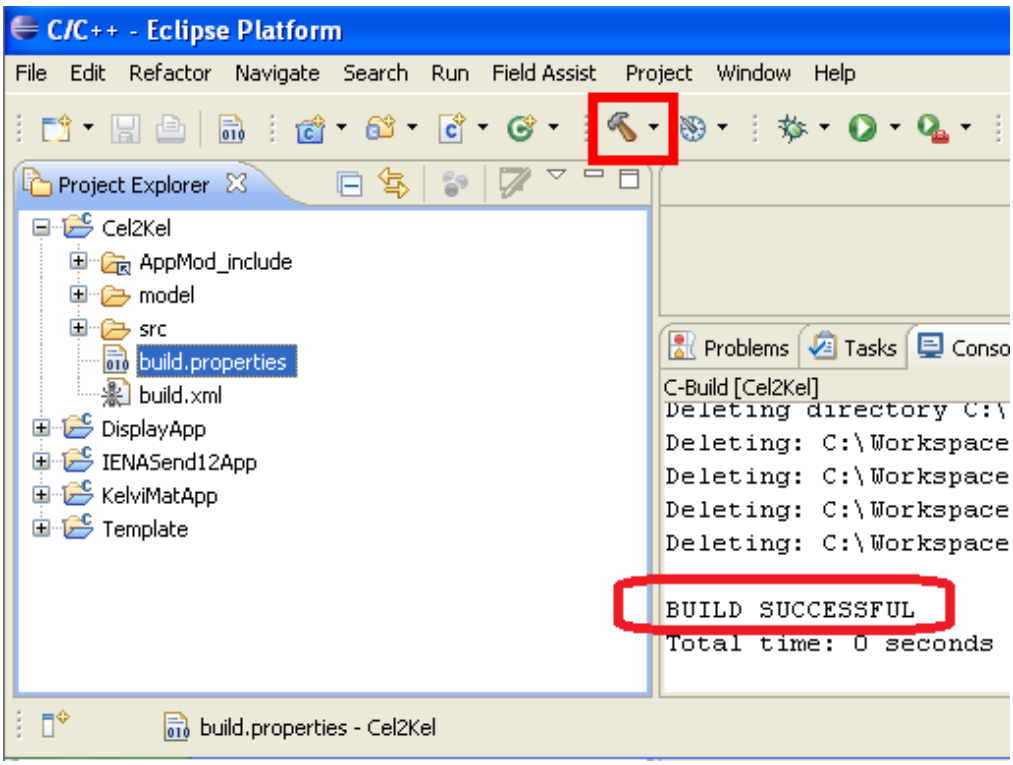


9.3 Pfade des Template anpassen

- Öffnen Sie unter *Template* die Datei *"build.properties"*
- Passen Sie die Pfade der rot markierten Objekte entsprechend ihrer Installation an:

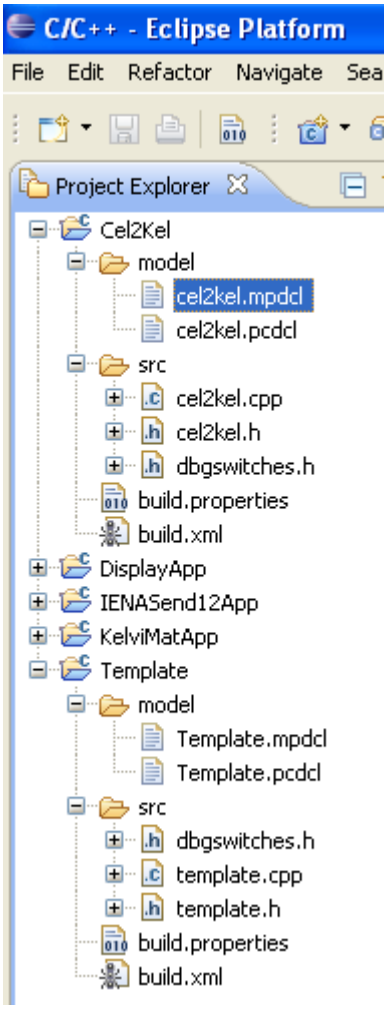


Nachdem alle Pfade korrekt gesetzt sind, können Sie das Template bauen. Klicken Sie hierzu auf das Hammersymbol:



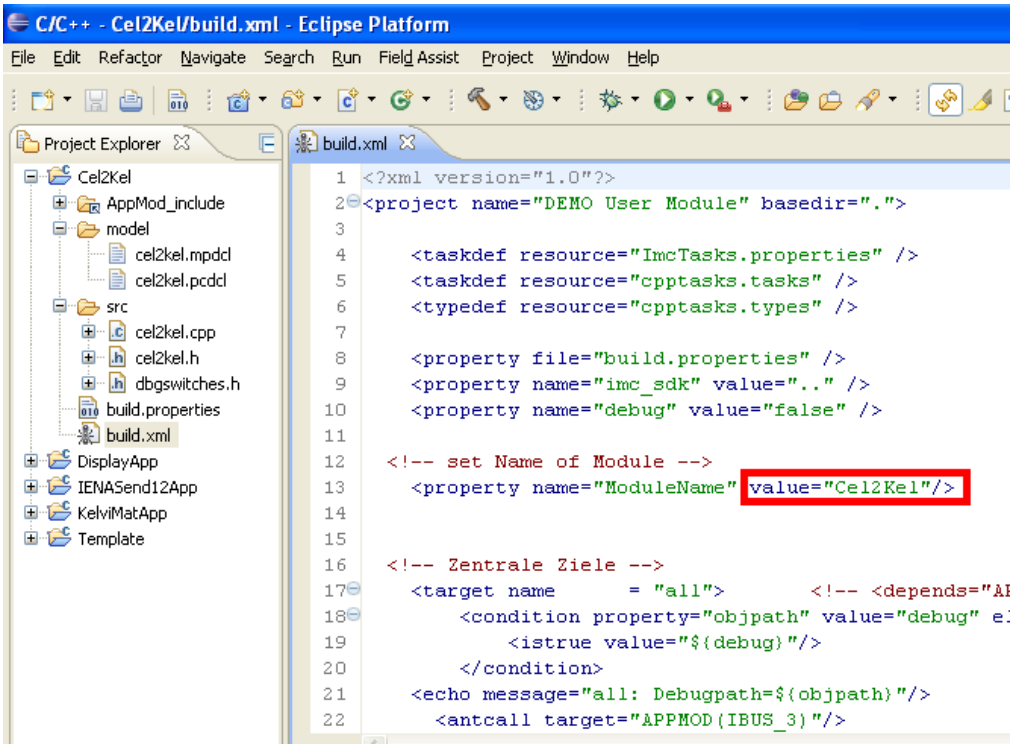
- Wenn der Bau erfolgreich war, erscheint in der Eclipse Konsole die Nachricht "BUILD SUCCESFUL"

9.4 Umbenennen des Template für unser Projekt

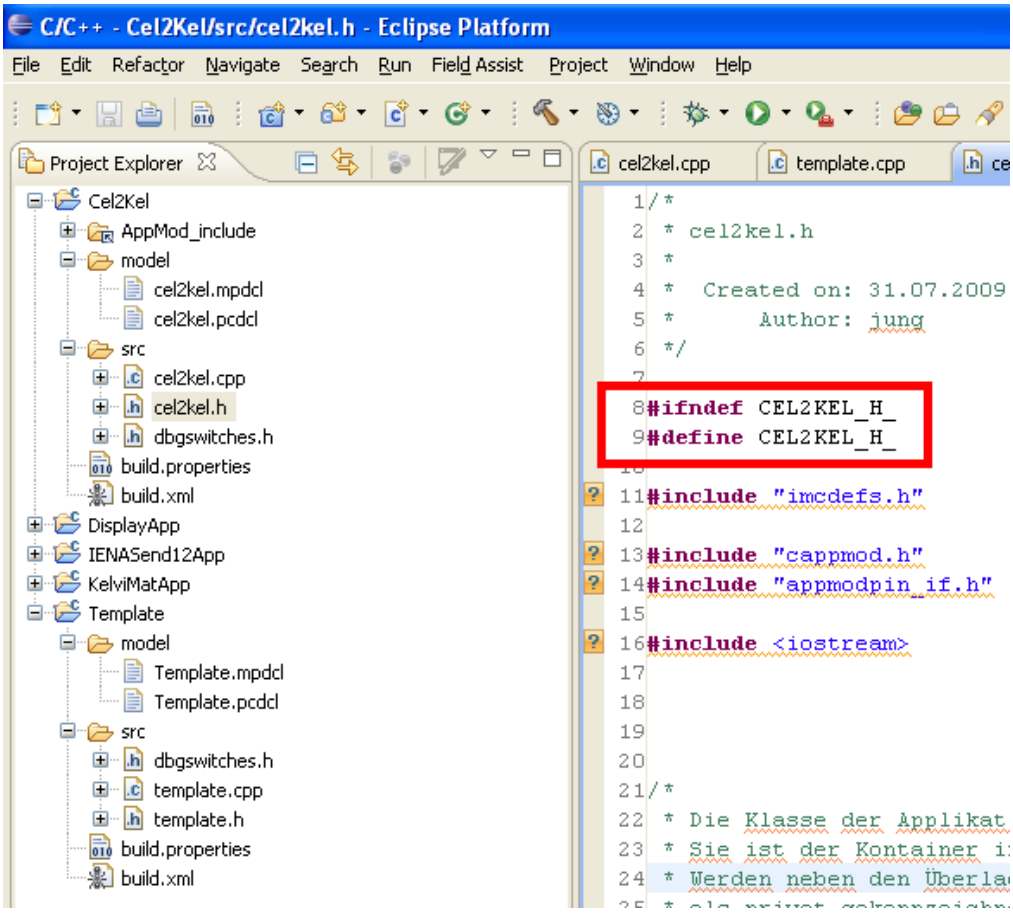


- Kopieren Sie das "*Template*" Projekt in Eclipse und benennen Sie es in "*Cel2Kel*" um.
- Nennen Sie folgende Projektdateien im Projekt *Cel2Kel* um:
  - *Template.mpdcl -> cel2kel.mpdcl*
  - *Template.pcdcl -> cel2kel.pcdcl*
  - *Template.ccpl -> cel2kel.ccp*
  - *Template.h -> cel2kel.h*

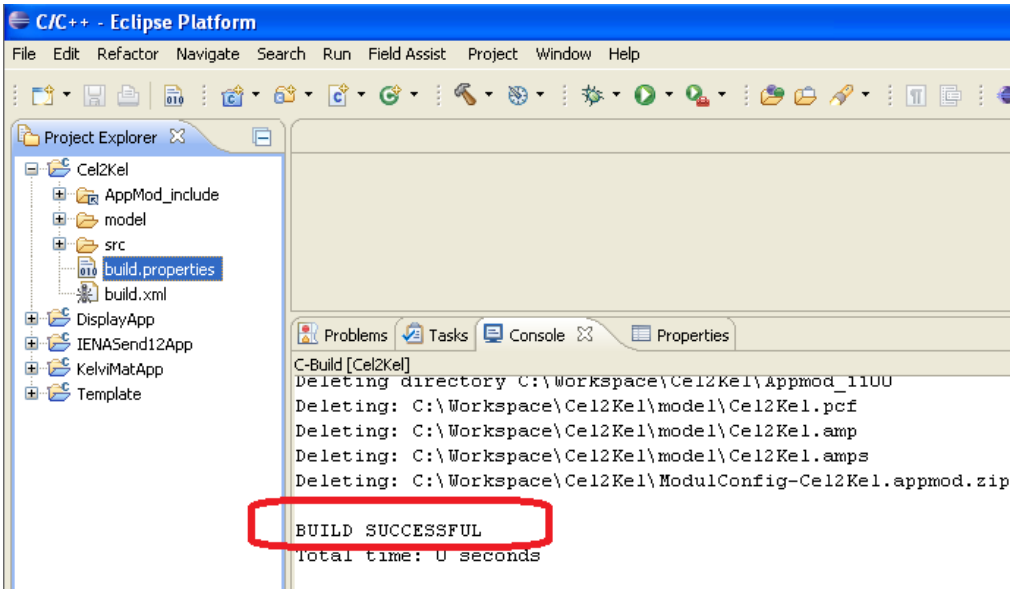
- Ändern Sie in der Datei *build.xml* den rot markierten Eintrag in *Cel2Kel*



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

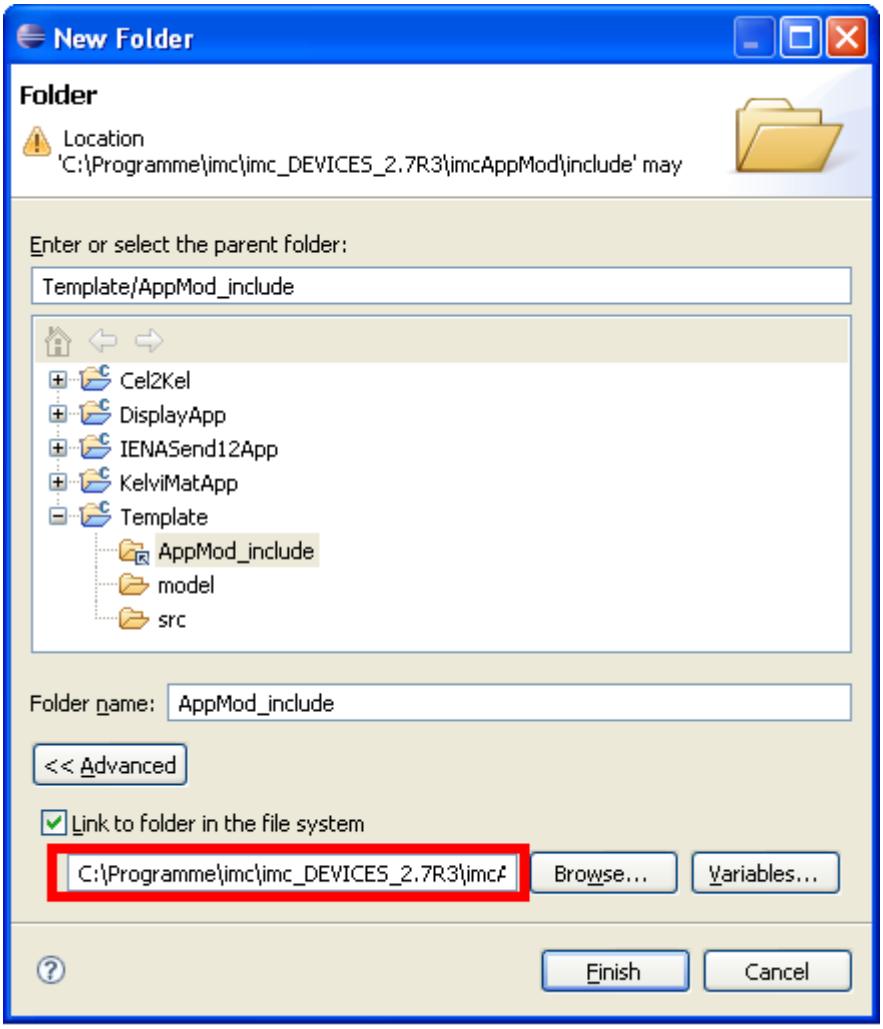


- Wenn die Umbenennung erfolgreich war, können Sie das Projekt kompilieren. Klicken Sie hierfür auf das Hammersymbol.
- Wenn der Bau erfolgreich war, erscheint in der Eclipse Konsole die Nachricht "BUILD SUCCESFUL"



9.5 Realisierung unseres Projektes

- Fügen Sie dem Projekt einen neuen "*Link*" Ordner hinzu. Die Einstellungen entnehmen Sie der Abbildung:



- **Schritt 1: Funktionen hinzufügen**
  - Rot markiert: Wählen Sie den Installationsordner von "imcAppModDevSetup" aus und zwar inklusive"...\imcAppMod\include".
  - Durch diese Verknüpfung haben Sie Zugriff auf fertige Funktionen und Definitionen von Variablen.

- **Schritt 2: Ein-/Ausgänge definieren (cel2kel.pcdcl)**
  - In der Datei **cel2kel.pcdcl** finden Sie zwei definierte Kanäle.

```
# 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
```

- **Schritt 3: Tunable Parameter (cel2kel.mpdcl)**
  - In der Datei **cel2kel.mpdcl** wird ein Tunable Parameter erzeugt.

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

- **Schritt 4: Definition von Zeigern und Hilfsvariablen (cel2kel.h)**
  - In der Datei **cel2kel.h** wurden im Bereich *Lokale Variablen* Änderungen vorgenommen.

```
#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_ */
```

- **Schritt 5: Funktionalität hinzufügen (cel2kel.cpp)**
  - Übernehmen Sie bitte die folgenden Änderungen

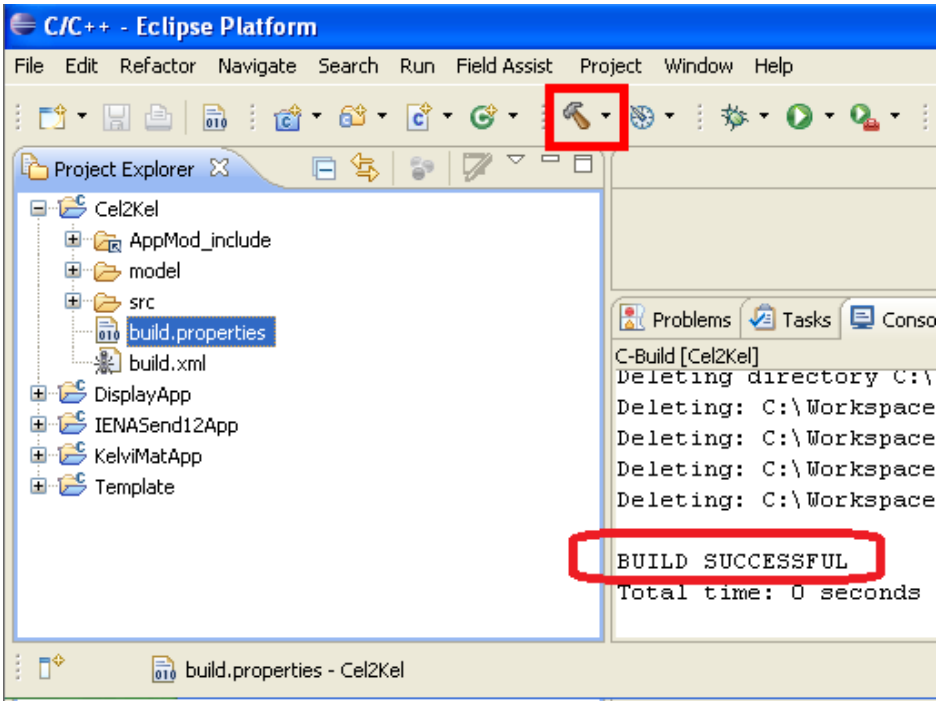
```
#include "ccommtypes.h"
#include "ccommcom.h"
#include "cel2kel.h"
#include <iostream>

/*
 * 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.
     * Entscheidend 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
    }
}
```

- **Schritt 6: Kompilieren**:
  - Nachdem Sie alle Änderungen im Projekt vorgenommen haben, kompilieren Sie das Projekt mit einem Klick auf das Hammersymbol.



Hinweis

Bei einem Wechsel der zugrunde liegenden Standard Software imc STUDIO ist ein erneutes Kompilieren des Projekts erforderlich. Dies ist zwingend notwendig, ab eine Änderung der Revisionsnummer, z.B. imc STUDIO 5.0R1 auf 5.0R3

9.6 Projekt erweitern

Bisher wurde nur der Tunable Parameter definiert. Im nächsten Schritt könnte eine Offsetsteuerung ergänzt werden.

10 Technische Daten - imc APPMOD

Eingebetteter Prozessor		
Parameter	**Wert**	**Bemerkungen**
Eingebetteter Prozessor	Freescale Power PC MPC5200B Core CLK 384 MHz	
RAM	64 MB 48 MB	Gesamtspeicher für die Applikation verfügbar
Flash	16 MB	nur für das Betriebssystem
Betriebssystem	Linux	

Allgemein		
Parameter	**Wert**	**Bemerkungen**
Schnittstellen	1x Ethernet Interface und 1x Serielles Interface 3,5 mm Klinke	Konkrete Applikationen können jeweils genau eine der beiden Schnittstellen verwenden. Eine gleichzeitige Benutzung beider Schnittstellen erfordert zwei Module im System. Service-Schnittstelle (RS232, 115 kBaud, Tx, Rx, GND) Konsole für Entwicklung, Debugging
Modul-Breite	benötigt 1 Steckplatz	fest verbaut, ab Werk
Modularität	Bestell-Option	
Max. Ausbau	3 8 1 2 3 5	in Summe in einer CRFX Basis Einheit in Summe in einem CRC System in Summe in einem BUSFX-4 System in Summe in einem BUSFX-6 System in Summe in einem BUSFX-8 System in Summe in einem BUSFX-12 System

Ethernet Interface		
Parameter	**Wert**	**Bemerkungen**
Anschlüsse / Knoten	1	
Anschluss-Stecker	1x RJ45	
Topologie	Bus	
Übertragungsprotokoll	TCP / IP	IEEE Norm 802.3
Übertragungsmedium	Ethernet	
Datenflußrichtung	senden und empfangen	
Baudrate	100 MBit 10 MBit	100BaseT (Halb- und Vollduplex) 10BaseT (Halb- und Vollduplex) Autosensing
Isolationsfestigkeit	60 V	gegen Systemmasse (CHASSIS)

Serielles Interface		
Parameter	**Wert**	**Bemerkungen**
Anschlüsse / Knoten	1	
Anschluss-Stecker	1x DSUB-9	
Baudrate	300, 1200, 2400, 4800, 9600, *14400*, 19200, *28800*, 38400, 57600, 115200, 230400	Sonder-Bitraten: 14400 und 28800
Isolation Isolationsfestigkeit	galvanisch isoliert 60 V	gegen Systemmasse (CHASSIS) nominale Arbeitsspannung
Betriebs-Modi	RS 232 RS 485 / RS 422	flexibel konfigurierbar: Multi-Protocol Transceiver
RS232 Modus		
Parameter	**Wert**	**Bemerkungen**
Topologie	Punkt zu Punkt	
Übertragungsprotokoll	RS232	
Signalart	Tx, Rx, GND CTS, RTS	Basis Signale Handshake, Fluss-Steuerung
Datenflußrichtung	senden und empfangen	
Byteformat	7 oder 8 Databits, 1 oder 2 Stoppbits, none/odd/even parity	
Flußkontrolle	XON/XOFF, RTS/CTS	
RS485 / RS422 Modus		
Topologie	Bus	
Übertragungsprotokoll	RS485	kompatibel mit RS422
Betriebsmodus	Halb- und Vollduplex	per Software schaltbar
Signalart	2x Tx, 2x Rx, GND	Basis Signale, differentiell
Datenflußrichtung	senden und empfangen	
Terminierung	120 Ω	per Software schaltbar

11 Anschlusstechnik

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

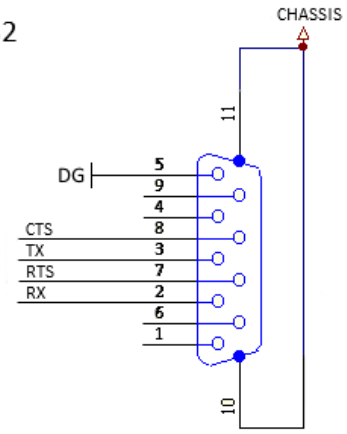
RS 422 / RS 485 Full-Duplex

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

RS 485 Half-Duplex

Signal	PIN
+D	3
-D	7

RS232



Service-Schnittstelle Klinke 3,5mm

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

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

Index