LASER INTERFEROMETER GRAVITATIONAL WAVE OBSERVATORY

LIGO Laboratory / LIGO Scientific Collaboration

LIGO- E1200225-v8 Advanced LIGO 5/21/2018

Coding Standard for TwinCAT Slow Controls Software

Daniel Sigg

California Institute of Technology
LIGO Project – MS 18-34
1200 E. California Blvd.
Pasadena, CA 91125
Phone (626) 395-2129
Fax (626) 304-9834
E-mail: info@ligo.caltech.edu

Massachusetts Institute of Technology
LIGO Project – NW22-295
185 Albany St
Cambridge, MA 02139
Phone (617) 253-4824
Fax (617) 253-7014
E-mail: info@ligo.mit.edu

LIGO Hanford Observatory
P.O. Box 159
Richland WA 99352
Phone 509-372-8106
Fax 509-372-8137

LIGO Livingston Observatory
P.O. Box 940
Livingston, LA 70754
Phone 225-686-3100
Fax 225-686-7189

http://www.ligo.caltech.edu/

Table of Contents

1 Introduction

The purpose of this document is to facilitate a single coding standard among the slow controls software written for the TwinCAT system. TwinCAT contains an embedded IEC 61131-3 software PLC which is the main focus here. The document gives guidance how to build a reusable programming structure, how to name objects like variable, structures and function blocks, and how to document a library module.

1.1 Programming Languages

The IEC 61131-3 programming standard supports 5 different languages: structured text (ST), function block diagram (FBD), ladder diagram (LD), instruction list (IL) and sequential function chart (SFC). TwinCAT 3 also supports C/C++ and Matlab/Simulink. For the advanced LIGO slow control systems only structured text shall be used with TwinCAT 2.11. For TwinCAT 3 advanced LIGO also supports C/C++ for integrating already written modules.

Programming language	Description	TwinCAT version
Structured Text	One of the IEC 61131-3 programming languages, Pascal like	2.11 and 3
C/C++	For integrating previously written modules	3

Table 1: Supported languages.

1.2 Project Directories

The project directories on a front-end or development machine are organized in a development area under version control and a target area where the run-times reside.

Items	Path	Owner
TwinCAT	C:\TwinCAT	Beckhoff
Code	C:\SlowControls	Subversion
Target	C:\ SlowControls\Target	Run-time

1.2.1 Target Area

The target area contains the files associated with a specific run-time. The directory structure is organized by target and PLC. The run-time files associated with a specific run-time are copied to the target directory using an installation script. This requires that the code are is a committed revision within subversion.

Items	Path	
Target Area	C:\SlowControls\Target	
Example target system area	C:\SlowControls\Target\H1ECATC1	
Example target PLC 1 code	C:\SlowControls\Target\H1ECATC1\PLC1	
Example target PLC 2 code	C:\SlowControls\Target\H1ECATC1\PLC2	
Example target PLC 3 code	C:\SlowControls\Target\H1ECATC1\PLC3	
Example target PLC 4 code	C:\SlowControls\Target\H1ECATC1\PLC4	
TwinCAT boot files	C:\TwinCAT\Boot	

1.3 Project Archive

All project files are stored in a subversion (SVN) archive on redoubt.ligo-wa.caltech.edu.

Item	Link	Type
Server	redoubt.ligo-wa.caltech.edu	web
Archive	/slowcontrols	web
Full path	https://redoubt.ligo-wa.caltech.edu/svn/slowcontrols/trunk	checkout

Table 2: Subversion archive.

1.3.1 Organization

The slow controls archive contains the folder TwinCAT for storing all files related to TwinCAT. There are currently two sub folders TwinCAT\Library for storing libraries and TwinCAT\Source for the storing project files. Scripts are stored in the TwinCAT\Script\Common folder. The configuration scripts and the system configuration associated with a real-time computer are stored in a target folder within TwinCAT\Script\Configuration. There are up to 4 PLCs allowed in TwinCAT 2.11.

Items	Path	
System documents	SlowControls\Documents	
Network documents	SlowControls\Documents\Network	
TwinCAT files	SlowControls\TwinCAT	
TwinCAT documents	SlowControls\TwinCAT\Documents	
TwinCAT coding standard	SlowControls\TwinCAT\Documents\CodingStandard	
TwinCAT library files	SlowControls\TwinCAT\Library	
Individual TwinCAT library	SlowControls\TwinCAT\Library\CommonModeServo	
…	…	
TwinCAT program source files	SlowControls\TwinCAT\Source	
Current TwinCAT source files	SlowControls\TwinCAT\Source\Current	
Source files for interferometer	SlowControls\TwinCAT\Source\Current\Interferometer	
Corner source files	SlowControls\TwinCAT\Source\Current\Interferometer\Corner	
End station source files	SlowControls\TwinCAT\Source\Current\Interferometer\End	
Import files	SlowControls\TwinCAT\Source\Current\Import	
…	…	
Script files	SlowControls\Scripts	
Common script files	SlowControls\Scripts\Common	
Configuration files	SlowControls\Scripts\Configuration	
Target configuration files	SlowControls\Scripts\Configuration\H1ECATC1	
System configuration files	SlowControls\Scripts\Configuration\H1ECATC1\SYS	
PLC configuration files	SlowControls\Scripts\Configuration\H1ECATC1\PLC	
…	…	
EPICS related files	SlowControls\EPICS	
EPICS utilities	SlowControls\EPICS\Utilities	
…	…	
Modbus related files	SlowControls\Modbus	
Modbus target files	SlowControls\Modbus\Target	
Individual Modbus target	SlowControls\Modbus\Target\H1ModbusC1	
…	…	

Table 3: Organization of the archive.

1.3.3 Version Numbers

The production code is managed by subversion release numbers. The subversion number is part of the run-time code and is archived. When significant changes to a library are made that require supporting both the old and new versions, a new library project has to be created. If the original library was called TimingMasterFanout then new version would be called TimingMasterFanoutV2.

1.4 Cycle Time

An IEC 61131-3 system consists of system task and at least one programmable logic controller (PLC). The system task is responsible for interfacing the hardware and starting the PLC tasks. The field bus of choice in advance LIGO is EtherCAT. The system task transfers data between a shared memory region and hardware at a fixed cycle time. TwinCAT 2.11 supports up to four different update rates. For advanced LIGO the standard update rate is 10 ms. For a limited number of channels a faster update rate of 1 ms is supported.

Task	Description	Rate
Standard	All non time critical software and supervisory tasks	10 ms
Fast	Time critical functions such as RS422 support at 115kbaud	1 ms

Table 4: Supported update rates.

The tasks with the fast update rate are running at a higher priority (lower number).

1.5 Data Tags (Channels)

1.5.1 Input/Output Convention

From the perspective of the TwinCAT program and configuration input channels refer to inputs from the EtherCAT terminals, e.g., analog-to-digital converters and binary inputs, whereas output channels refer to outputs to the EtherCAT terminals, e.g., digital-to-analog converters and binary outputs. The same is true for user inputs which are inputs into TwinCAT and readbacks which are outputs from TwinCAT.

1.5.2 Interface Variables

All external tags (channels) have an initialization record which is periodically updated and is declared PERSISTENT. Upon power failure and loading a new code its value as retained as much as possible. Any initialization that is required, when the PLC is started or when a new version is loaded, needs to be dealt with in software. See the SaveRestore library.

1.6 OPC Interface

We are using the TwinCAT OPC comments denoted by (*~ ... *) to make global variables accessible to the OPC server. The opening bracket annotation needs to be on the same line as the variable. Variable names in TwinCAT are translated one-to-one into OPC tag names, which in turn are translated into EPICS channels using a conversion rule. OPC properties are used to describe additional information such as limits, precision and state names. These OPC properties are translated into corresponding EPICS database fields.

2 Program Organization

The development blocks for the advanced LIGO slow controls software are individual libraries. Each of the basic libraries is tailored to control a single electronics chassis or controller.

A typically library consists of

- one or more type describing the hardware inputs,
- one or more type describing the hardware outputs,
- a type describing the user interface channels or tags (input and output),
- one or more function blocks containing the run-time code, and
- a set of visual templates that can be used for diagnostics.

The main program then consists of a global variable list and a series of function block calls.

2.1 Library

This section gives an example of the structures and the function block defined for the LowNoiseVco library.

2.1.1 Hardware Input Structure

```
TYPE LowNoiseVcoInStruct :
STRUCT
       PowerOk:            BOOL;  (* Voltage monitor readback *)
       TuneMon:            INT;   (* Monitor for the frequency offset  *)
       ReferenceMon:       INT;   (* RF power at the reference input *)
       DividerMon:         INT;   (* RF power at the divider input *)
       OutputMon:          INT;   (* RF power at the output amp *)
       ReferenceTemp:      INT;   (* Temperature of the reference RF detector *)
       DividerTemp:        INT;   (* Temperature of the divider RF detector *)
       OutputTemp:         INT;   (* Temperature of the output RF detector *)
       Excitation:         BOOL;  (* Monitors the excitation input enable *)
       Frequency:          LREAL; (* Measured frequency *)
       FrequencyLive:      BOOL;  (* Keep alive for frequency measurement *)
END_STRUCT
END_TYPE;
```

2.1.3 Hardware Output Structure

```
TYPE LowNoiseVcoOutStruct :
STRUCT
       TuneOfs:            INT;   (* Setpoint for the frequency offset  *)
       ExcitationEn:       BOOL;  (* Enables the excitation input *)
END_STRUCT
END_TYPE;
```

2.1.4 Interface Structure

All elements of an interface structure are getting exported with read and write permission. To prevent output tags from showing an invalid value each output parameter has to overwritten at each cycle. Output parameters in the interface structure should never be read.

```
TYPE LowNoiseVcoStruct :
STRUCT
       (* error handling *)
       Error:              ErrorStruct;  (* Error struct *)
       (* output tags *)
       PowerOk:            BOOL;  (* Voltage monitor readback *)
       TuneMon:            LREAL; (* Monitor for the frequency offset in V *)
       ReferenceMon:       LREAL; (* RF power at the reference input in dBm *)
       DividerMon:         LREAL; (* RF power at the divider  input in dBm *)
       OutputMon:          LREAL; (* RF power after the output amplifier dBm *)
       ReferenceTemp:      LREAL; (* Temperature of the reference RF detector *)
       DividerTemp:        LREAL; (* Temperature of the divider RF detector *)
       OutputTemp:         LREAL; (* Temperature of the output RF detector in C *)
       ExcitationSwitch:   BOOL;  (* Monitor the excitation input enable *)
       Frequency:          LREAL; (* Frequency of the VCO output *)
       FrequncyServoFault: BOOL;  (* Indicates a fault in the frequency servo *)
       (* input tags *)
       TuneOfs:            LREAL; (* Setpoint for the frequency offset in V *)
       ExcitationEn:       BOOL;  (* Enables the excitation input *)
       FrequencySet:       LREAL; (* Setpoint for the VCO frequency output *)
       FrequencyServoEn:   BOOL;  (* Enables the frequency PID *)
END_STRUCT
END_TYPE;
```

2.1.6 Error Handling

Each main function block needs to provide error handling using a single struct defined as the first element of the interface structure: Error of type ErrorStruct. This struct contains a Flag, Code and Msg. If the error flag is set true, it indicates an error condition. The error code is a bit encoded value listing the error conditions with zero indicating no error. The error code number can be used to flag multiple errors by setting corresponding bits. Error conditions are described in the documentation associated with the library.

The error message is a human readable string describing the error condition. It can contain up to 80 characters. If multiple errors are flagged, the error message needs to reflect this. All error messages need to be defined in a global constant of type ErrorMessagesArray.

```
VAR_GLOBAL CONSTANT
       ThermistorStruct_Errors: ErrorMessagesArray := [
              (*  1 *) 'Thermistor resistance is too high',
              (*  2 *) 'Thermistor resistance is too low',
              (*  3 *) 'Thermistor data invalid',
              (*  4 *) 'Thermistor measurement error'];
END_VAR
```

The name of the constant string array has to reflect the name of the structure that contains the error structure with the extension "_Errors" added. In TwinCAT 2.11 this constant has to be linked to a file with the name "ThermistorStruct_Errors.exp" with the option "Export before compile" selected. This will guarantee that the automatic medm screen generator is able to assemble an error list for each structure. For TwinCAT 3.1 no special export option is required.

The function block that is processing the user interface structure needs to call the error handler to flag the error messages. Typically, this works like this:

```
VAR
       ErrorHandler:       ErrorHandlerFB;
END_VAR

// Initialization (before any errors are flagged
ErrorHandler (ErrorMethod := ErrorMethodEnum.Init, Error := Thermistor.Error,
              List := ADR(ThermistorStruct_Errors));
// Flagging an error
IF Thermistor.Resistance > 1000000 THEN
       ErrorHandler (ErrorMethod := ErrorMethodEnum.Report,
       Error := Thermistor.Error, Num := 1);
END_IF;
// Making sure it is getting reported
ErrorHandler (ErrorMethod := ErrorMethodEnum.Commit, Error := Thermistor.Error);
```

A simple library without error conditions just needs to call the error handler with Init and Commit.

If a struct contains other structs that have error handlers themselves, the error handler of the containing struct must check the error flag of the sub structs and report them. For example:

```
TYPE TemperatureStruct :
STRUCT
       (* error handling *)
       Error:              ErrorStruct;  (* Error struct *)
       (* output tags *)
       Thermistor_A:       ThermistorStruct;
       Thermistor_B:       ThermistorStruct;
       Thermistor_C:       ThermistorStruct;
END_STRUCT
END_TYPE;

VAR_GLOBAL CONSTANT
       TemperatureStruct _Errors: ErrorMessagesArray := [
              (*  1 *) 'Thermistor_A',
              (*  2 *) 'Thermistor_B',
              (*  3 *) 'Thermistor_C'];
END_VAR

VAR
       ErrorHandler:       ErrorHandlerFB;
END_VAR

// Initialization (before any errors are flagged
ErrorHandler (ErrorMethod := ErrorMethodEnum.Init, Error := Temperature.Error,
              List := ADR(TemperatureStruct _Errors));
// Flagging the errors of sub structures
IF Temperature.Thermistor_A.Flag THEN
       ErrorHandler (ErrorMethod := ErrorMethodEnum.Report,
       Error := Temperature.Error, Num := 1);
END_IF;
IF Temperature.Thermistor_B.Flag THEN
       ErrorHandler (ErrorMethod := ErrorMethodEnum.Report,
       Error := Temperature.Error, Num := 2);
END_IF;
…
// Making sure it is getting reported
ErrorHandler (ErrorMethod := ErrorMethodEnum.Commit, Error := Temperature.Error);
```

It is important that the error messages for a sub structure have the exact same name as the element it represents. If not the auto generation of the screens will not pick them up.

2.1.7 Function Block

A function block has to declare input and output variables. In the simplest case the input parameter is the hardware input structure, the hardware output structure is the output parameter and the interface structure is the in/out parameter.

```
FUNCTION_BLOCK LowNoiseVcoFB
VAR_INPUT
       LowNoiseVcoIn:      LowNoiseVcoInStruct;       (* Input structure *)
END_VAR
VAR_OUTPUT
       LowNoiseVcoOut:     LowNoiseVcoOutStruct;      (* Output structure *)
END_VAR
VAR_IN_OUT
       LowNoiseVco:        LowNoiseVcoStruct;         (* Interface structure *)
END_VAR
```

2.1.8 Initialization

All function blocks controlling hardware have to support initialization and have to be able to store the current state. This is done by passing a SaveRestoreEnum parameter as well as an additional interface structure that holds the previously stored values.

```
FUNCTION_BLOCK LowNoiseVcoFB
…
VAR_INPUT
       Request:            SaveRestoreEnum;           (* init/save request *)
END_VAR
VAR_IN_OUT
       LowNoiseVcoInit:    LowNoiseVcoStruct;         (* saved interface struct *)
END_VAR
…
```

The additional interface structure should is used to pass the previously saved parameters to the initialization routine. It is also used to store these parameters. The Init parameter will either request no operation, an initialization operation, a save operation, or a transition to a safe operation mode. Typically, only state machines will have to implement the transition to a safe operation mode. Within the function block the initialization code would look like:

```
(* Code *)
CASE Request OF
        (* initialization *)
        Restore:
                LowNoiseVco := LowNoiseVcoInit;
                (* additional initialization steps can be added here *)
        SafeMode:
                (* only for state machines *)
        Save:
                LowNoiseVcoInit := LowNoiseVco;
        Noop:
                (* always ignore *)
END_CASE;
…
```

The LowNoiseVcoInit variable will be stored in a global persistent block. This means its values are preserved between reboots and recompilations as long as the LowNoiseVcoStruct stays the same. Changing the LowNoiseVcoStruct or one of its elements will invalidate its persistent memory and reinitialize with all zeros. However, a change in an unrelated structure should not affect the low noise VCO.

The restore request is basically an initialization request and will be issued once after a reboot or a reload of the program. The save request will be issued at regular intervals, but at a low rate, maybe once a minute. The safe mode request might be issued upon a fatal error or a user request. It would typically affect the entire PLC program and not just one library.

Restoring to the previously saved values is probably the best option in most circumstances, but is unlikely to be appropriate in all cases. For example, a state machine probably needs to start in a well-defined initialization state and not in whatever state it was left in. In these cases additional code needs to be added to the restore request.

2.1.9 Visual Screen Templates

Either one or a set of visual screen templates are associated with a library. The top-level screen template should be a representation of the hardware controlled by the library. It should interface the interface structure, and display all its input and output parameters. Input parameters should be modifiable by the user. Since the library only knows abstract data types, the visual screen template shall deploy placeholder variables to represent actual data. For example, the VCO template screen might reference "vco.OutputMon" in the numeric field describing the output RF power. vco is the placeholder parameter that will be replaced with the actual data of type LowNoiseVcoStruct, when the visual template is embedded into a master screen. In most cases the visual template screens should leave their background transparent, so that it can be set by the master screen.

2.2 Global Variables

Global variables are used to store hardware input structure, the hardware output structure, the associated function blocks and a hierarchical structure representing all interface structure as part of an interferometer structure. The later is outlined in section 3.4 and is used to represent the opc naming tree which in turn is translated into an EPICS name.

Persistent global variables are used to store the initialization structures. Their values are retained between reboots and restarts of the program. They will be reinitialized with zeros, when the structure itself is modified, so.

```
VAR_GLOBAL
       I1:           IfoStruct;    (*~    (OPC : 1 : visible for OPC-Server) *)

       AlsVcoIn      AT %I*:       LowNoiseVcoInStruct;       (* Input *)
       AlsVcoOut     AT %Q*:       LowNoiseVcoOutStruct;      (* Output *)
       AlsVcoFB:                   LowNoiseVcoFB;             (* Function block *)
       …
END_VAR
VAR_GLOBAL PERSISTENT
       AlsVcoInit:                 LowNoiseVcoStruct;         (* Save/restore *)
       …
END_VAR
```

The variable names for the input, output, interface and initialization structures follow the naming of the structure elements within I1 that lead to the VCO. The interferometer tag isn't included in the name to support copy/paste between different interferometers. The actual addresses are wildcards and are configured through the system manager.

2.3 Program

The main program can be a simple series of function block calls. There can be multiple programs to separate subsystems. These programs need to be attached to the standard task, which updates at the 10 ms rate. If a function block requires 1 ms update rate, it needs to be located in separate program that is attached to the fast task.

```
PROGRAM ALS
VAR
       SaveRestore: SaveRestoreFB;(* function block for save/restore *)
       GotoSafe:    BOOL;         (* goto safe mode when transitioning high *)
       Request:     SaveRestoreEnum; (* save/restore request *)
END_VAR

SaveRestore ( SaveInterval := T#1m,
              GotoSafe := GotoSafe,
              Request => Request );

AlsVcoFB (    LowNoiseVcoIn := AlsVcoIn,
              LowNoiseVcoOut => AlsVcoOut,
              LowNoiseVco := I1.Als.VCO,
              Request := Request,
              LowNoiseVcoInit := AlsVcoInit );
…
END_PROGRAM;
```

2.5 Project Files

Project files must not be written site specific. To facilitate this the source files for the interferometer are separated into two subdirectories: corner and end. Each project file must contain at least two imports to determine the subversion number and to specify the target interferometer as well as the appropriate end station. The project name indicates the PLC it is intended for and has the form “PLC1.pro” or similar.

2.5.1 Imports

A normal project files defines two global variable resources which are linked to a file and are imported before compilation: Global_Variables_Version and Global_Variables_IFOVAR.

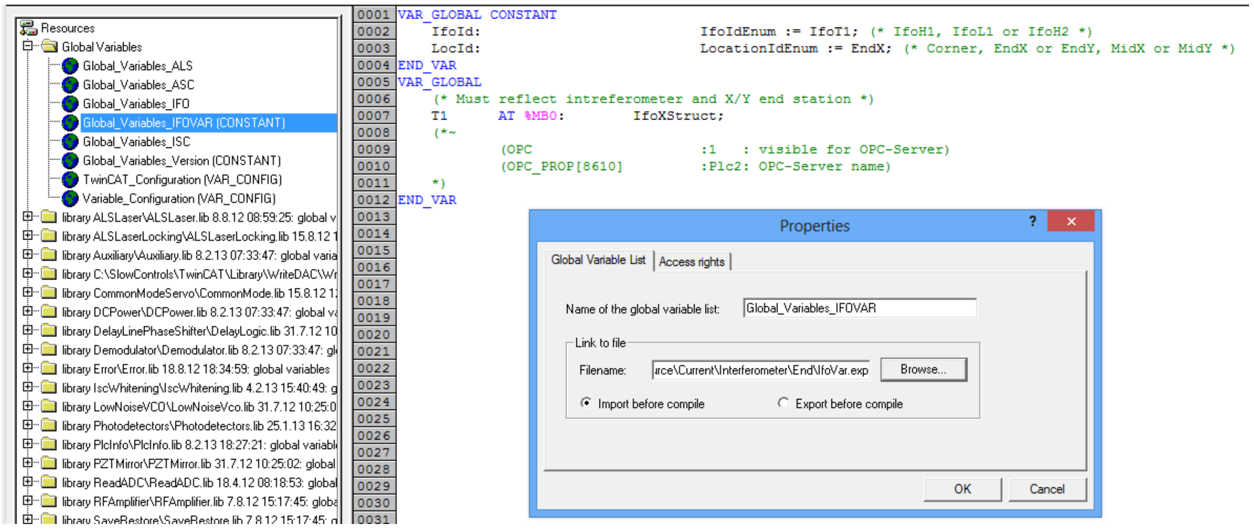
Global_Variables_Version looks like this:

```
VAR_GLOBAL CONSTANT
      SvnRevision:      DINT := 0;
END_VAR
```

Whereas Global_Variables_IFOVAR looks like this

```
VAR_GLOBAL CONSTANT
       IfoId: IfoIdEnum := IfoT1; (* IfoH1, IfoL1 or IfoH2 *)
       LocId: LocationIdEnum := EndX; (* Corner, EndX or EndY, MidX or MidY *)
END_VAR
```

In the source directory the interferometer is set to the test system. The important point is that during installation these two global variable resources will be overwritten with the correct information appropriate for the target. In particular, the installation script guarantees that the current source code revision has no local modifications and sets the SvnRevision number accordingly. Similarly, the interferometer parameters are set to the appropriate configuration for the specific target. These variables are available in the code and can be used in the rare instances where sites or end stations need to be distinguished.



```
Resources
Global Variables
    Global_Variables_ALS
    Global_Variables_ASC
    Global_Variables_IFO
    Global_Variables_IFOVAR (CONSTANT)
    Global_Variables_ISC
    Global_Variables_Version (CONSTANT)
    TwinCAT_Configuration (VAR_CONFIG)
    Variable_Configuration (VAR_CONFIG)
library ALSLaser\ALSLaser.lib 8.8.12 08:59:25: global v
library ALSLaserLocking\ALSLaserLocking.lib 15.8.12 1
library Auxiliary\Auxiliary.lib 8.2.13 07:33:47: global varia
library C:\SlowControls\TwinCAT\Library\WriteDAC\Wr
library CommonModeServo\CommonMode.lib 15.8.12 1:
library DCPower\DCPower.lib 8.2.13 07:33:47: global va
library DelayLinePhaseShifter\DelayLogic.lib 31.7.12 10
library Demodulator\Demodulator.lib 8.2.13 07:33:47: gl
library Error\Error.lib 18.8.12 18:34:59: global variables
library IscWhitening\IscWhitening.lib 4.2.13 15:40:49: g
library LowNoiseVCO\LowNoiseVco.lib 31.7.12 10:25:0
library Photodetectors\Photodetectors.lib 25.1.13 16:32
library PlcInfo\PlcInfo.lib 8.2.13 18:27:21: global variabl
library PZTMirror\PZTMirror.lib 31.7.12 10:25:02: global
library ReadADC\ReadADC.lib 18.4.12 08:18:53: global
library RFAmplifier\RFAmplifier.lib 7.8.12 15:17:45: globa
library SaveRestore\SaveRestore.lib 7.8.12 15:17:45: g
```

```
VAR_GLOBAL CONSTANT
    IfoId:                          IfoIdEnum := IfoT1; (* IfoH1, IfoL1 or IfoH2 *)
    LocId:                          LocationIdEnum := EndX; (* Corner, EndX or EndY, MidX or MidY *)
END_VAR
VAR_GLOBAL
    (* Must reflect intreferometer and X/Y end station *)
    T1      AT %MB0:        IfoXStruct;
    (*~
            (OPC                    :1   : visible for OPC-Server)
            (OPC_PROP[8610]         :Plc2: OPC-Server name)
    *)
END_VAR
```

Name of the global variable list: Global_Variables_IFOVAR

Link to file

Filename: ırce\Current\Interferometer\End\IfoVar.exp

● Import before compile ○ Export before compile

2.5.2 Site and Location Customization

The main program defines

```
VAR_GLOBAL
      Ifo:                    IfoStruct;    (*~
         (OPC              :1   : visible for OPC-Server)
         (OPC_Filter       :Export: Filter string for OPC server)
         (OPC_PROP[8610]  :Plc1: OPC-Server name)
         (OPC_PROP[8620]  :.${IFO}: alias name)
    *)
END_VAR
```

which is getting aliased to an H1 or L1 variable by the TwinCAT-EPICS-gateway. A program, which runs in either the X-end or the Y-end, usually defines a structure element “End”. This is then aliased to “X” or “Y” by the TwinCAT-EPICS-gateway. For example,

```
TYPE PemStruct :
STRUCT
       End:          PemBuildingStruct;    (*~
         (OPC              :1   : visible for OPC-Server)
         (OPC_PROP[8620]  :${END}: alias name)
    *)
END_STRUCT
END_TYPE
```

Dynamically, the variable IfoId and LocId (both enums) are available to distinguish site and location.

2.5.3 Replacement Rules

For the TwinCAT-EPICS-gateway to pick up the correct interferometer name and end station location, it needs to define a set of replacement rule. This is done in the second argument of the tcAlias command, e.g., "IFO=L1, END=Y".

As shown above the IFO and END variables in the PLC programs need to define an alias name that contains the replacement variable name enclosed by "${" and "}". It is possible to define more complicated aliases, such as "${END}1PLC2". The resulting name after replacement then becomes "Y1PLC2".

One place to be careful is a replacement rule for a global variable. TwinCAT2 will prepend a dot, whereas TwinCAT3 will prepend a name and a dot. The TwinCAT-EPICS-gateway will typically remove and letters up and including the first dot. Therefore, if a global variable uses an alias name, it probably needs to start with an explicit dot.

2.6 Installation and Configuration Scripts

The installation scripts are located in the folder SlowControls\Scripts\Common. A separate documentation is available with T1300175.

Typically, a user would run the "install_tc_target" shortcut which starts a GUI that allows the selection of a target with associated PLCs as well as the desired action. Targets are defined in the Scripts\Configuration folder. Each target requires its own folder in there, and each PLC requires its own folder within the target folder. Target specific configuration scripts are specified in the target folder, whereas PLC specific configurations are specified in the PLC folder. Typically, the target configuration script should be sufficient.

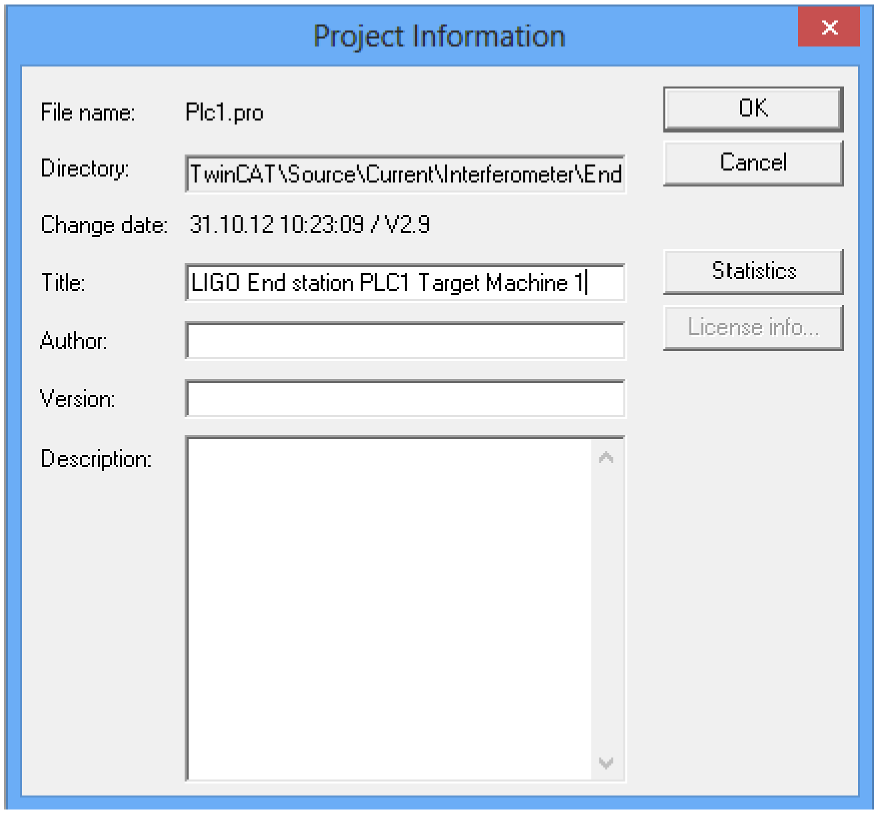
Each target folder also requires a SYS folder which contains the TwinCAT system configuration file (*.tsm). The TwinCAT system configuration in principle is identical between sites and end stations, but there may be slight difference between locations that cannot be easily parameterized. Since software updates are much more frequent than hardware changes, this should not pose a problem.

3 TwinCAT Project Setup

3.1 Setting the Options in the Project

3.1.1 Project Information

Set the title information to the appropriate PLC and station. Since there are typically only source files for corner and end station, there should be no indication of interferometer or which of the two end stations.



File name: Plc1.pro

Directory: TwinCAT\Source\Current\Interferometer\End

Change date: 31.10.12 10:23:09 / V2.9

Title: LIGO End station PLC1 Target Machine 1

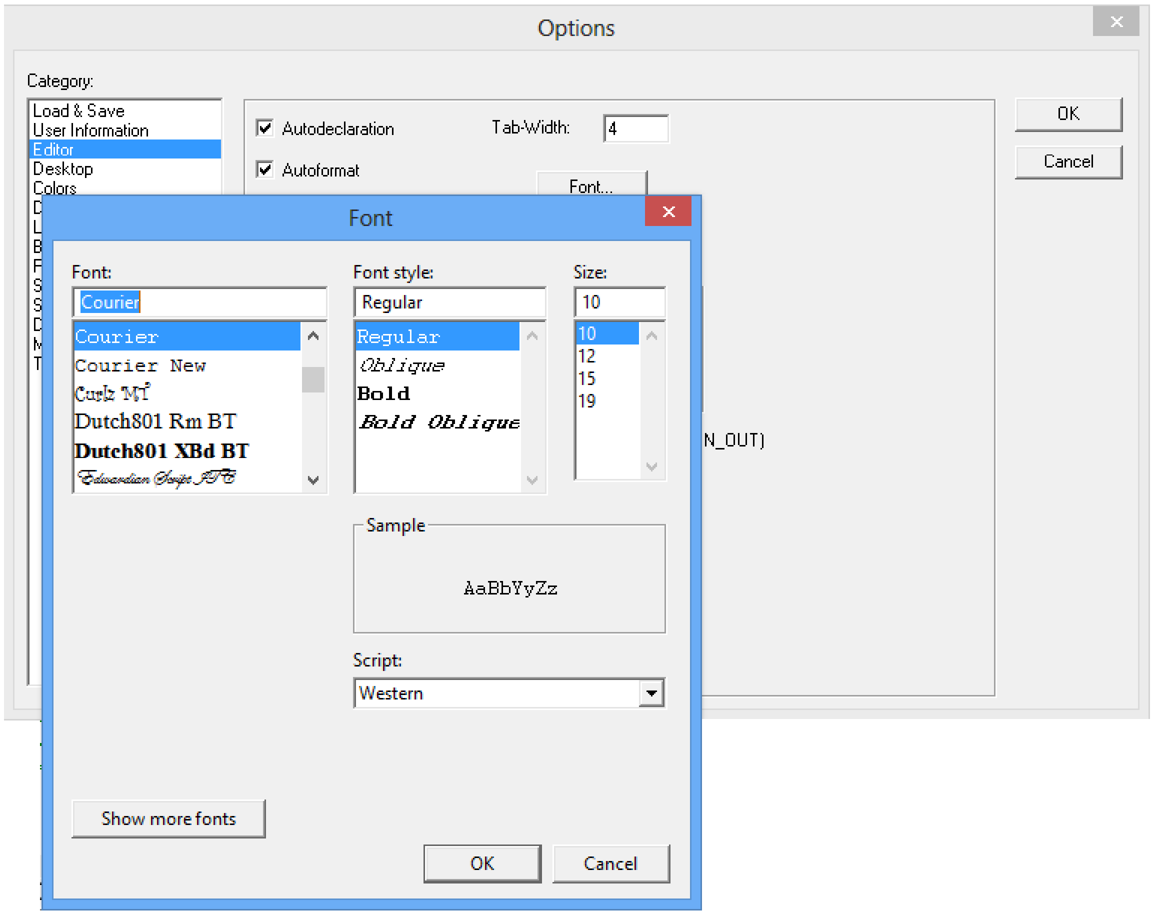
Author:

Version:

Description:

3.1.2 Fonts and Path Options

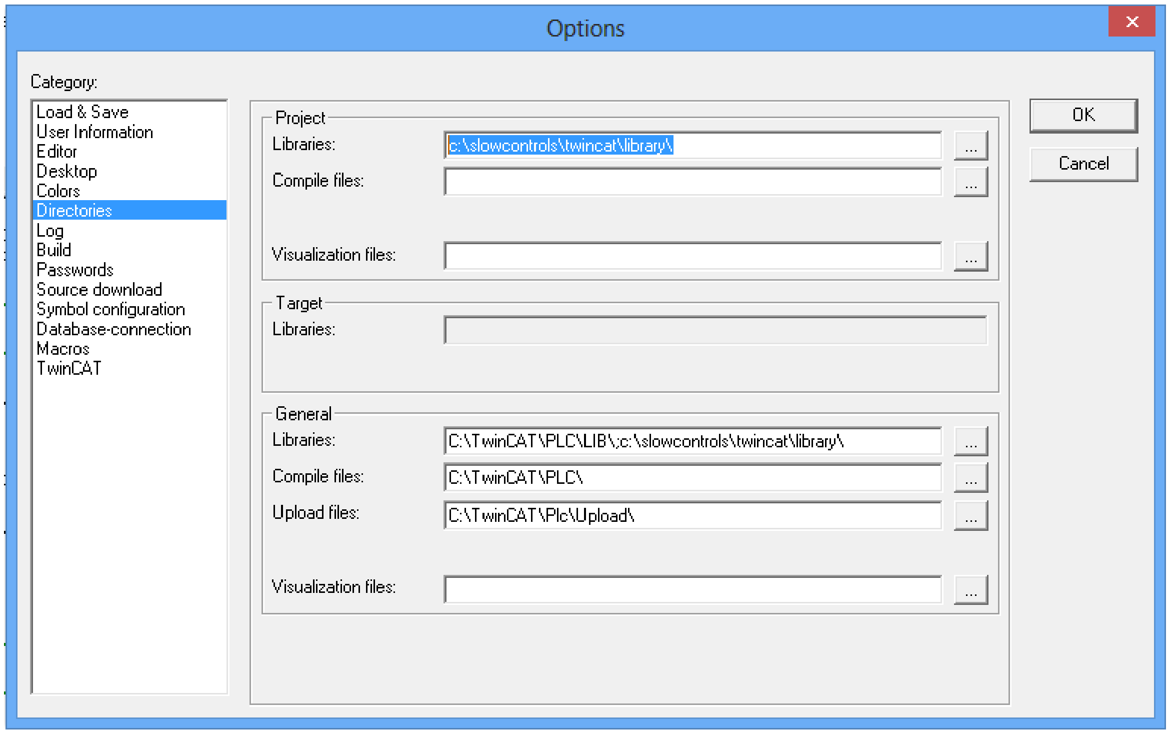
Choose a fixed point font such as “Courier New”. The project library path should include the path to the slow controls libraries.



[✓] Autodeclaration
Tab-Width: 4
[✓] Autoformat

Font
Font: Courier
Courier
Courier New
Curlz MT
Dutch801 Rm BT
Dutch801 XBd BT
Edwardian Script ITC
Font style: Regular
Regular
Oblique
Bold
Bold Oblique
Size: 10
10
12
15
19
Sample
AaBbYyZz
Script: Western

N_OUT)



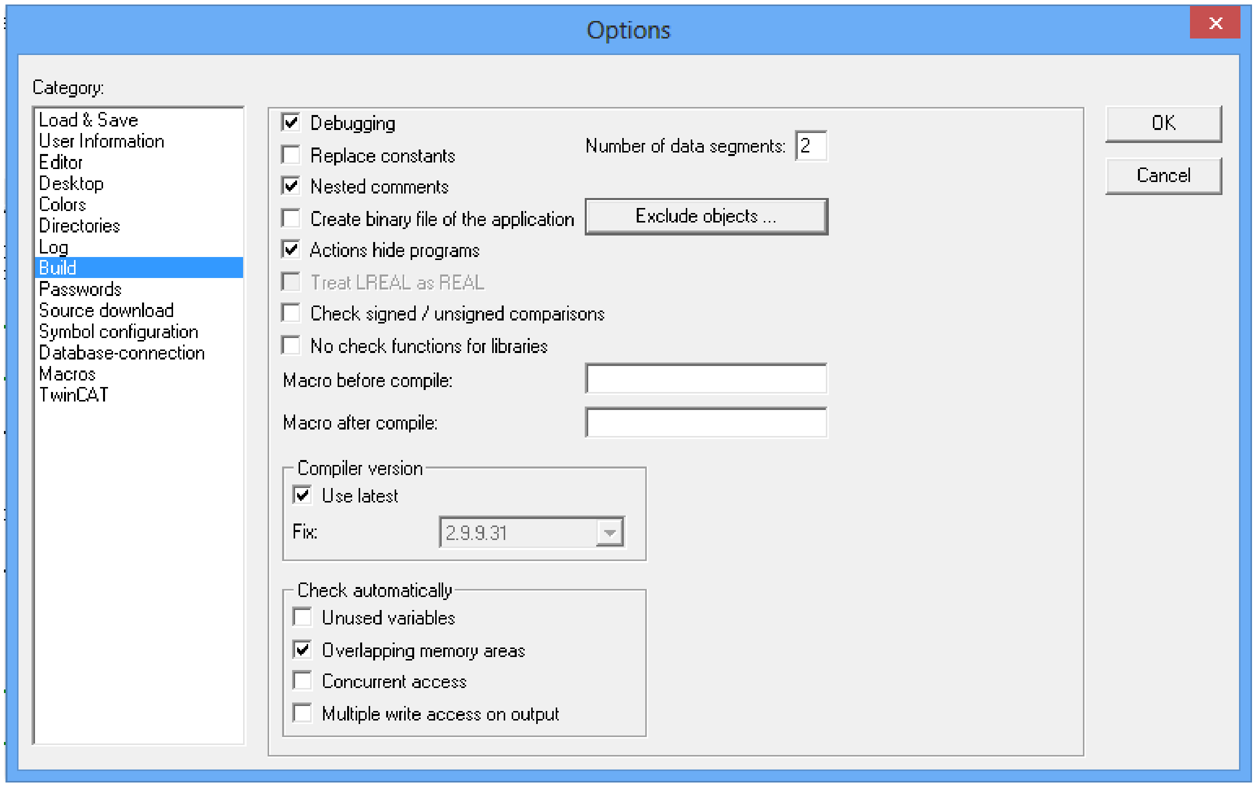
Project
Libraries: c:\slowcontrols\twincat\library\
Compile files:
Visualization files:

Target
Libraries:

General
Libraries: C:\TwinCAT\PLC\LIB\;c:\slowcontrols\twincat\library\
Compile files: C:\TwinCAT\PLC\
Upload files: C:\TwinCAT\Plc\Upload\
Visualization files:

3.1.3 Build Options

Make sure the “check overlapping memory areas” option is selected.



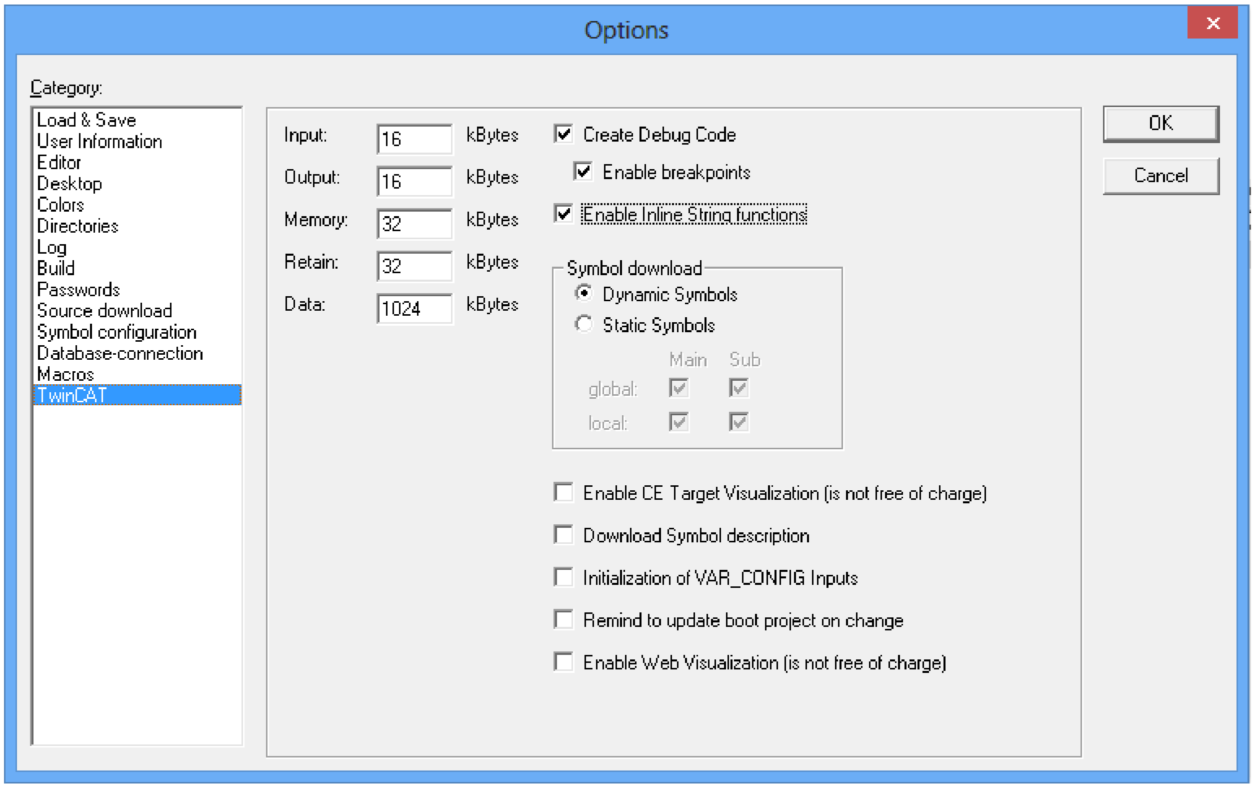
[✓] Debugging
[ ] Replace constants
Number of data segments: 2
[✓] Nested comments
[ ] Create binary file of the application
[✓] Actions hide programs
[ ] Treat LREAL as REAL
[ ] Check signed / unsigned comparisons
[ ] No check functions for libraries
Macro before compile:
Macro after compile:

Compiler version
[✓] Use latest
Fix: 2.9.9.31

Check automatically
[ ] Unused variables
[✓] Overlapping memory areas
[ ] Concurrent access
[ ] Multiple write access on output

3.1.4 TwinCAT Options

Make sure the “Enable inline string functions” option is selected. This will allow us to use the string functions in more than one task. By default they are not multi-thread safe. This option page also allows to change the allocation for the different memory regions. For large project these limits may have to be increased.



Input: 16 kBytes
Output: 16 kBytes
Memory: 32 kBytes
Retain: 32 kBytes
Data: 1024 kBytes

☑ Create Debug Code
☑ Enable breakpoints
☑ Enable Inline String functions

Symbol download
● Dynamic Symbols
○ Static Symbols

	Main	Sub
global:	☑	☑
local:	☑	☑

☐ Enable CE Target Visualization (is not free of charge)
☐ Download Symbol description
☐ Initialization of VAR_CONFIG Inputs
☐ Remind to update boot project on change
☐ Enable Web Visualization (is not free of charge)

3.2 PLC Common Infrastructure

Each PLC project is required to implement the PlcInfo library. This library provides support for reporting errors at the PLC level, it exports the software subversion number and it provides a set of run-time information specific to the PLC.

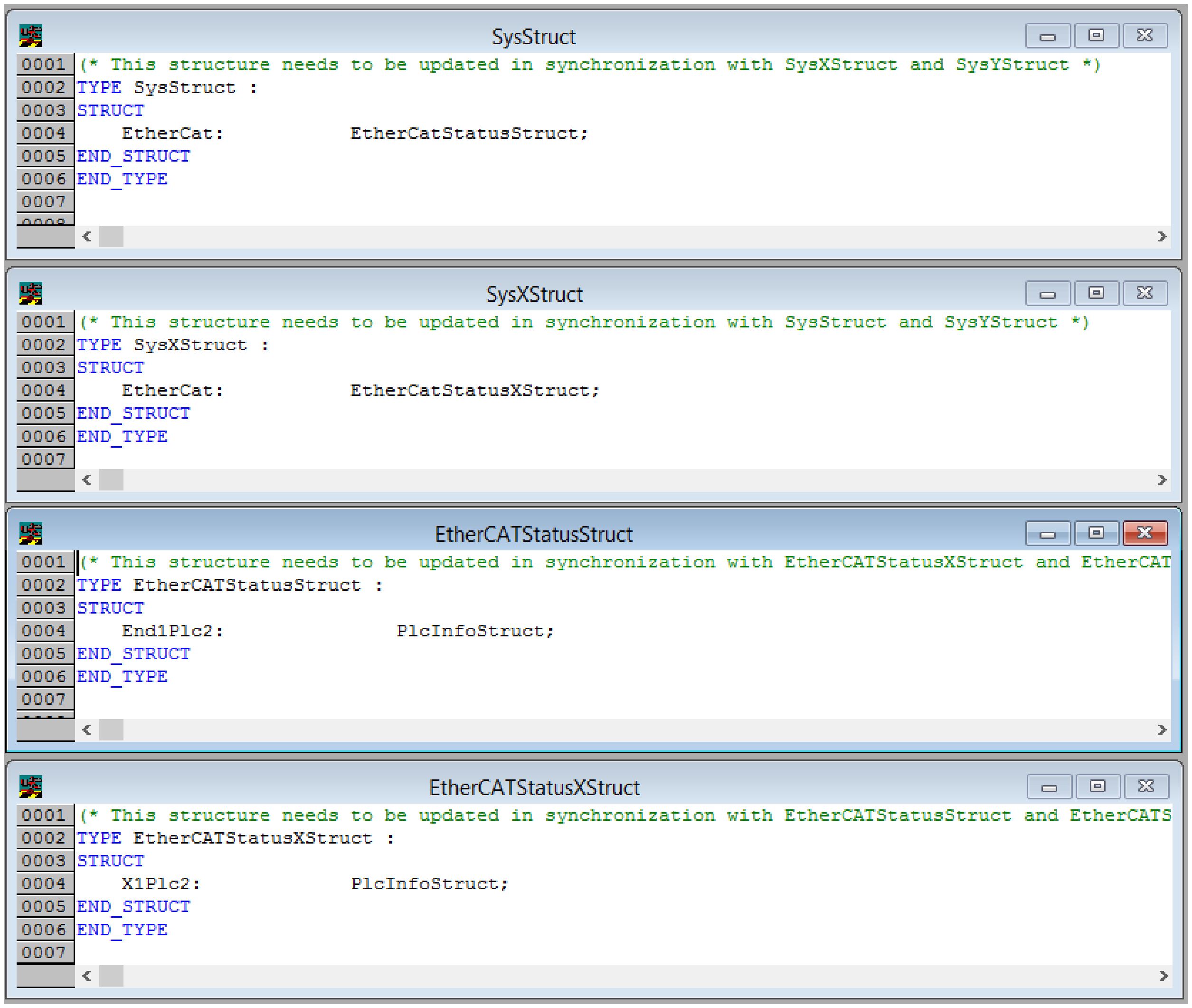
3.2.1 Common libraries

The following libraries need to be imported for every project.

SaveRestore	Support for saving user interface settings to a persistent memory block. It also supports restoring these settings upon restart.
Error	Support for error handling and reporting.
PlcInfo	Support for reporting software version, run-time information and top level errors

3.2.2 Error Handling

Each user interface structure is required to contain an error reporting structure of type ErrorStruct. Error handling is supported with “Error” library. However, this abstraction cannot be propagated to the highest level because a PLC only covers a subset of the available channels. For instance, there are “H1:ALS“ channels in the corner and both end stations. This means that each of the three associated PLCs will have a user structure H1.Als defined. If it would contain an error structure, the names would collide. There is also no error at the H1 level for the same reason. Instead, the error hierarchical error structures stop at the level one below. For instance, corner station PLC would contain an error structure in the H1.Als.C structure, whereas the end stations would contain error structures in the H1.Als.X and H1.Als.Y structures, respectively. Within each PLC code each subsystem has a corresponding top level error reporting structure. These top level local error structures are combined in a PLC error structure which is reported in the PlcInfo structure. The PlcInfo structures are defined in the H1.Sys.EtherCAT structure. Each PLC has exactly one of these structures and the name of the structure contains the target designator. For instance, the user can access H1.Sys.EtherCat.C1Plc2, H1.Sys.EtherCat.X1Plc2 and H1.Sys.EtherCat.Y1Plc2 and the Error element therein to learn about errors occurring with PLC2 of corner, end X and end Y, respectively.



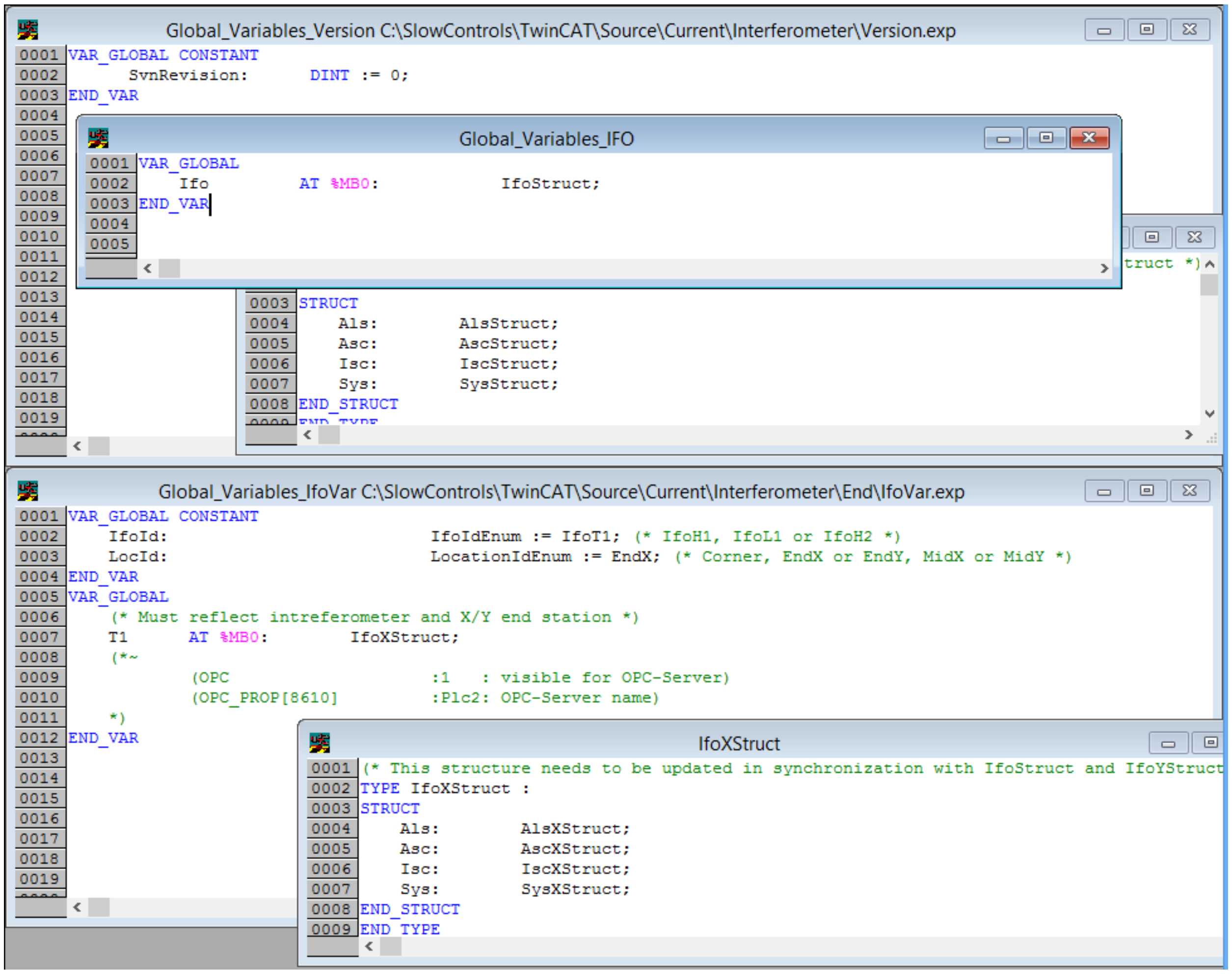
```
(* This structure needs to be updated in synchronization with SysXStruct and SysYStruct *)
TYPE SysStruct :
STRUCT
    EtherCat:           EtherCatStatusStruct;
END_STRUCT
END_TYPE
```

```
(* This structure needs to be updated in synchronization with SysStruct and SysYStruct *)
TYPE SysXStruct :
STRUCT
    EtherCat:           EtherCatStatusXStruct;
END_STRUCT
END_TYPE
```

```
(* This structure needs to be updated in synchronization with EtherCATStatusXStruct and EtherCAT
TYPE EtherCATStatusStruct :
STRUCT
    End1Plc2:               PlcInfoStruct;
END_STRUCT
END_TYPE
```

```
(* This structure needs to be updated in synchronization with EtherCATStatusStruct and EtherCATS
TYPE EtherCATStatusXStruct :
STRUCT
    X1Plc2:             PlcInfoStruct;
END_STRUCT
END_TYPE
```

Since there is only one end station program, a little bit of magic with the overlapping H1 and Ifo variables is required to make it all work.


Global_Variables_Version C:\SlowControls\TwinCAT\Source\Current\Interferometer\Version.exp
Global_Variables_IFO
Global_Variables_IfoVar C:\SlowControls\TwinCAT\Source\Current\Interferometer\End\IfoVar.exp
IfoXStruct

```
VAR_GLOBAL CONSTANT
     SvnRevision:      DINT := 0;
END_VAR
```

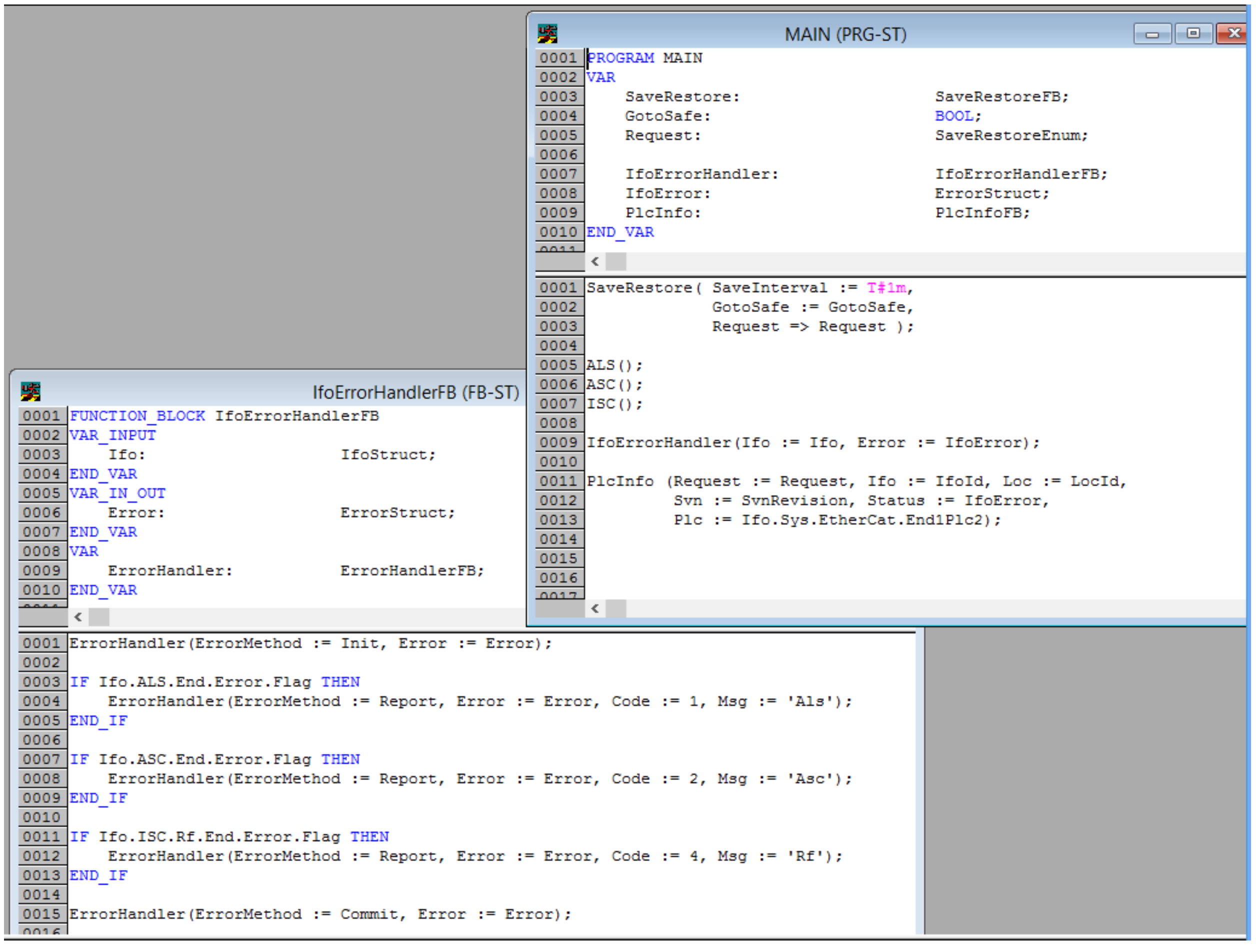
```
VAR_GLOBAL
    Ifo        AT %MB0:           IfoStruct;
END_VAR
```

```
truct *)
STRUCT
    Als:         AlsStruct;
    Asc:         AscStruct;
    Isc:         IscStruct;
    Sys:         SysStruct;
END_STRUCT
```

```
VAR_GLOBAL CONSTANT
    IfoId:                          IfoIdEnum := IfoT1; (* IfoH1, IfoL1 or IfoH2 *)
    LocId:                          LocationIdEnum := EndX; (* Corner, EndX or EndY, MidX or MidY *)
END_VAR
VAR_GLOBAL
    (* Must reflect intreferometer and X/Y end station *)
    T1        AT %MB0:         IfoXStruct;
    (*~
            (OPC                    :1   : visible for OPC-Server)
            (OPC_PROP[8610]         :Plc2: OPC-Server name)
    *)
END_VAR
```

```
(* This structure needs to be updated in synchronization with IfoStruct and IfoYStruct
TYPE IfoXStruct :
STRUCT
    Als:         AlsXStruct;
    Asc:         AscXStruct;
    Isc:         IscXStruct;
    Sys:         SysXStruct;
END_STRUCT
END_TYPE
```

The program unit for the Sys also requires a call to the top level error handler for the PLC which collects the error from each subsystem and combines them into the PLC status information.



```
PROGRAM MAIN
VAR
    SaveRestore:                    SaveRestoreFB;
    GotoSafe:                       BOOL;
    Request:                        SaveRestoreEnum;

    IfoErrorHandler:                IfoErrorHandlerFB;
    IfoError:                       ErrorStruct;
    PlcInfo:                        PlcInfoFB;
END_VAR
```

```
SaveRestore( SaveInterval := T#1m,
             GotoSafe := GotoSafe,
             Request => Request );

ALS();
ASC();
ISC();

IfoErrorHandler(Ifo := Ifo, Error := IfoError);

PlcInfo (Request := Request, Ifo := IfoId, Loc := LocId,
         Svn := SvnRevision, Status := IfoError,
         Plc := Ifo.Sys.EtherCat.End1Plc2);
```

```
FUNCTION_BLOCK IfoErrorHandlerFB
VAR_INPUT
    Ifo:                        IfoStruct;
END_VAR
VAR_IN_OUT
    Error:                      ErrorStruct;
END_VAR
VAR
    ErrorHandler:               ErrorHandlerFB;
END_VAR
```

```
ErrorHandler(ErrorMethod := Init, Error := Error);

IF Ifo.ALS.End.Error.Flag THEN
    ErrorHandler(ErrorMethod := Report, Error := Error, Code := 1, Msg := 'Als');
END_IF

IF Ifo.ASC.End.Error.Flag THEN
    ErrorHandler(ErrorMethod := Report, Error := Error, Code := 2, Msg := 'Asc');
END_IF

IF Ifo.ISC.Rf.End.Error.Flag THEN
    ErrorHandler(ErrorMethod := Report, Error := Error, Code := 4, Msg := 'Rf');
END_IF

ErrorHandler(ErrorMethod := Commit, Error := Error);
```

3.2.3 Software version

The subversion revision is a number describing a snapshot of the software in the archive. The installation scripts for a PLC with make sure the archive is up-to-date before compiling the project. The active subversion number is imported into the code and available to the user as part of the PlcInfo structure.

3.2.5 Run-time information

Let’s take a look at the most important elements of the PlcInfo structure.

IfoId	Interferometer identification, e.g. H1 or L1
LocationId	Location of the computer, i.e., Corner, EndX or EndY
Status	Top level error structure of the PLC
SvnRevision	Subversion revision number
StartTime	Time the PLC was started
CurrentTime	Current system time
Hostname	Computer name of the machine running the PLC
CpuUsage	Percentage of CPU usage
SysLatencyActual	Actual system latency
SysLatencyMax	Maximum system latency

4 Naming Scheme

4.1 Names

Generally, verbose and descriptive names are preferred to short and abbreviated ones. This will make the code more readable and help in maintenance and support. For example, Index is preferred over I and TimingMasterFanout is preferred over Tmfo.

4.1.1 Variable Names

The naming of variables preferably should be unique in all libraries, following the camel case notation: For each variable a meaningful, preferably short, English name should be used, the base name. Always the first letter of a word of the base name is to be written uppercase, the remaining letters lowercase; example: FastGain or InputOffset. Abbreviations are written starting with an uppercase and then all lower case; example: VcoGain or TimingMasterFanout. Pointer variables shall use the suffix **Ptr**, whereas constant variables may use the suffix **Const**.

4.1.2 Type Names

Type names follow the same rule as variable names. A complex type shall incorporate a suffix to denote is derivation: **Enum** for ENUM, **Struct** for STRUCT and **Array** for ARRAY.

Structure members follow the rules of variables.

4.1.3 Function and Method Names

Function and method names follow the same rules as variables but with the suffix **Fun**. Internal helper functions such as conversion routines can also use a lowercase name, so that they look more in line with mathematical notation.

4.1.4 Function Block Names

The names of function blocks follow the same rules as variables but with the suffix **FB**. Interfaces in TwinCAT 3 use the suffix **I**.

4.1.5 Names of Visuals

Visual interfaces have the suffix **Vis**.

4.1.6 Suffix Summary

Element	Description	suffix
Constant	Constant value (optional, may be clear from context)	Const
Pointer	Pointer to a variable	Ptr
ENUM	Enumerated type	Enum
STRUCT	Record type	Struct
ARRAY	Array type	Array
Function	Function or Method declaration	Fun
Function block	Function block declaration	FB
Interface	Abstract function block or interface	I
Visual	Screen interface for diagnostics	Vis

Table 5: Required suffix notation.

4.2 Hardware Channels

Variables that are connected to hardware channels are separated into input variables and output variables. They must be located in the input and output shared memory regions, respectively. A variable describing a list of input channels must have the suffix **In**. The corresponding structure must have the suffix **InStruct**. An output channel list uses the suffix **Out**, whereas the output structure uses **OutStruct**. Channels with different cycle time must be placed into different structures. The above names are for the standard cycle time of 10 ms. Channels that need to be updated at the fast rate need to prepend **Fast** to the above suffixes.

Element	Description	suffix
Input variable	Input variable with standard update rate	In
Output variable	Output variable with standard update rate	Out
Input variable	Input variable with fast update rate	FastIn
Output variable	Output variable with fast update rate	FastOut
Input STRUCT	Input channel structure with standard update rate	InStruct
Output STRUCT	Output channel structure with standard update rate	OutStruct
Input STRUCT	Input channel structure with fast update rate	FastInStruct
Output STRUCT	Output channel structure with fast update rate	FastOutStruct

Table 6: Input and output channel notation.

A code fragment declaring input and output channels in the global variable space:

```
PicoMotorFastIn   AT %IB0100: PicoMotorFastInStruct;
PicoMotorFastOut  AT %QB0200: PicoMotorFastOutStruct;
PicoMotorIn       AT %IB0102: PicoMotorInStruct;
PicoMotorOut      AT %QB0204: PicoMotorOutStruct;
```

4.3 Library Objects

4.3.1 Name Space

Libraries can optionally choose a name space following the variable name notation. This name space is then used to prefix all exported objects. For example: the library TimingMasterFanout has the name space prefix Timing. Within this library TimingSlaveDuoToneStructure, TimingReadSlaveFun and TimingMasterFanoutFB are a valid structure, function and function block, respectively.

Simple libraries that consist of an input structure, an output structure, an interface structure and a function block are not required to choose an explicit name space, but are expected to use the library name as the base for all four objects. Hence, they are defining an implicit name space with the same name as the library name. For example: the library CommonMode may contain the structures CommonModeInStruct, CommonModeOutStruct and CommonModeStruct as well as the function block CommonModeFB.

4.3.2 Folder Names

Program object units (POUs) and data types are organized in folders. These folders are purely organizational and are intended to help grouping items together for easier maintenance. In a library all exported types, functions and function blocks are typically located at the top level. If there are many objects, it may make sense to group them into folders. In any case, internal objects should always be moved into a folder named Internal.

4.4 External Tags

External tags (channels) are organized in a hierarchical structure. Each system defines its own structure. This continues with structures for subsystems that are contained in the system structures.

```
TYPE AlsEndStruct :
STRUCT
      Laser:      ALSLaserStruct;
      VCO:        LowNoiseVcoStruct;
      PZT1:       PZTMirrorStruct;
      PZT2:       PZTMirrorStruct;
      …
END_STRUCT
END_TYPE;

TYPE AlsStruct :
STRUCT
      End:        AlsEndStruct;    (*~
         (OPC            :1   : visible)
         (OPC_PROP[8620] :${END}: alias name)
    *)
END_STRUCT
END_TYPE
…
TYPE IfoStruct:
STRUCT
      Als:        AlsStruct;
      Asc:        AscStruct;
      Lsc:        LscStruct;
      Tcs:        TcsStruct;
END_STRUCT
END_TYPE;

VAR_GLOBAL PERSISTENT
      IFO:        IfoStruct;      (*~
         (OPC            :1   : visible for alias)
         (OPC_Filter     :Export: Filter string for OPC server)
         (OPC_PROP[8610] :Plc1: OPC server name)
         (OPC_PROP[8620] :.${IFO}: alias name)
    *)
END_VAR;
```

This allows for exporting the entire interferometer interface structure at once and it allows for generating tag names automatically while preserving the hierarchical organization.

5 OPC Access and Properties

5.1 OPC Access

The global variable describing the interface structure of the interferometer is made accessible to the OPC server by using the OPC comments. Meaning,

```
IFO:  IfoStruct; (*~
         (OPC           :1   : visible for alias)
…     *)
```

will make the entire IFO variable with all its sub elements visible through the OPC interface. In turn, it can be interfaced to EPICS. Individual tags such as the FastGain of the LaserServo will be available from the OPC server as “H1.Isc.Als.LaserServo.FastGain”. The default EPICS channel name constructed from this tag will then become “H1:Isc-Als_LaserServo_FastGain”. Be aware that IEC 61131-3 names are not case sensitive. The same is true for the corresponding TwinCAT OPC names, whereas EPICS channel names are case sensitive.

5.2 OPC Properties

OPC properties are used to further describe the external tags. These properties are also used to fill in the EPICS database fields. The properties have to be attached to the elements at the end of the hierarchical structure. These are variables with a basic type like INT or LREAL. Due to the program organization most of these variables are defined in libraries through structures. Therefore, the OPC properties are written after the structure elements using the OPC comment structure. For example:

```
TYPE LowNoiseVcoStruct :
STRUCT
       (* output tags *)
       PowerOk:     BOOL;  (*~
                           (OPC_PROP[0005] :1: read-only)
                           (OPC_PROP[0101] :Voltage monitor readback: DESC)
                           (OPC_PROP[0106] :OK: ONAM)
                           (OPC_PROP[0107] :OOR: ZNAM) *)
       TuneMon:     LREAL; (*~
                           (OPC_PROP[0005] :1: read-only)
                           (OPC_PROP[0101] :Frequency offset monitor: DESC)
                           (OPC_PROP[0100] :V: EGU)
                           (OPC_PROP[0103] :-10: LOPR)
                           (OPC_PROP[0102] :+10: HOPR)
                           (OPC_PROP[8500] :3: PREC) *)
       ...
END_STRUCT
END_TYPE;
```

Property ID	Description	Record
5	Access control: 1 – read-only, 3- read/write	all
100	EGU: Engineering units	numeric
101	DESC: Description	all
102	HOPR: High operations value	numeric
103	LOPR: Low operation value	numeric
104	DRVH: Maximum instrument range	numeric
105	DRVL: Minimum instrument range	numeric
106	ONAM: Label for closed (one) state	binary
107	ZNAM: Label for open (zero) state	binary
306	HYST: alarm deadband	numeric
307	HIHI: hihi alarm level	numeric
308	HIGH: high alarm level	numeric
309	LOW: low alarm level	numeric
310	LOLO: lolo alarm level	numeric
8500	PREC: Display precision	numeric
8510 to 8525	ZRST, ONST, ... FFST: Zero string, one string, ... fifteen string	mb binary
8600	EPICS data type (bi, bo, ai, ao, longin, longout, stringin, stringout, mbbi, mbbo, mbbiDirect, and mbboDirect)	all
8601	Input or output: overwrites the default behavior	all
8602	TSE: Time stamp; default is -2	all
8603	PINI: default 1 for input and 0 for output records	all
8604	DTYP: default is opc; can be overwritten with opcRaw	all
8610	Default OPC server name; default is opc	top level
8611	TwinCAT runtime name including ads routing info and port	top level
8620	Alias for structure item or top level symbol	top & items
8700	OSV: one alarm severity	binary
8701	ZSV: zero alarm severity	binary
8702	COSV: change of state alarm severity	(mb) binary
8703	UNSV: unknown state alarm severity	mb binary
8710 to 8725	ZRSV, ONSV, … FFSV: zero, one, … fifteen state alarm severity	mb binary
8727 to 8730	HHSV, HSV, LSV and LLSV	analog
8800 to 8999	FIELD: Any database field can be specified in the comment string; does not perform any checks; use only when truly desperate	don't use

Table 7: Supported OPC properties.

Only a small subset of EPICS database fields are supported. In general, fields associated with conversion and calculations are not supported, since all processing should be done within the PLC program. The supported general properties are listed in the above table.

If a property is specified for a structure, it is used as the default value for all its elements. It can be overwritten by each element below. NO_ALARM, MINOR and MAJOR are the allowed alarm severity values. HIHI and LOLO alarms are assigned major severity, if they are defined; whereas LOW and HIGH alarms are assigned minor severity, if they are defined. Custom fields for are currently not supported.

The alias property supports replacement rules.

5.3 Automatic Type Support

By default all variables that are read-only will be represented by EPICS input records, whereas all variables that have read/write access will be represented by EPICS output records. This behavior can be overwritten, but there should never be a reason to.

The table below shows the default EPICS type selected for the database depending on the TwinCAT datatype.

Type	Description	
longin/longout	SINT, INT, DINT, LINT, USINT, UINT, UDINT, ULINT, BYTE, WORD, DWORD, LWORD	
bi/bo	BOOL	
mbbi/mbbo	Enumerated data type with 16 or fewer labels	
stringin/stringout	STRING	
ai/ao	REAL, LREAL, any other	

Table 8: Automatic type support.

5.4 Enumerated Types

An enumerated type will be converted into a multi-bit binary record, if there are 16 or fewer labels and if all numeric representations are between 0 and 15. There is no conversion possible. The numeric value of the enum type has to be the same as its EPICS representation, i.e., The zero value will be set to 0, etc. The string values of the multi-bit binary record are automatically set to the labels of the enumerated type.

Since enum labels need to be unique in TwinCAT, one usually has to add a prefix to guarantee that there are no name conflicts. This leads to somewhat cumbersome names in EPICS. It is therefore possible to specify the EPICS enum labels specifically. Example:

```
TYPE IfoIdEnum : (IfoH1, IfoL1, IfoH2, IfoT1, IfoI1);
END_TYPE
(*~
       (OPC_PROP[8510] :H1: ZRST)
       (OPC_PROP[8511] :L1: ONST)
       (OPC_PROP[8512] :H2: TWST)
       (OPC_PROP[8513] :T1: THST)
       (OPC_PROP[8514] :I1: FRST)
*)
```

This leads to EPICS labels of the form H1, L1, etc. rather than the default IfoH1, IfoL1, etc.

5.5 Array Variables

Array variables are supported by IEC 61131-3 and can be exported through OPC as well. They will also be accessible through EPICS, but require an extension to the LIGO channel naming convention. For example, if the structure "L1.Io.Wfs1" contains the members:

```
TYPE DemodComplex:
STRUCT
       I:     LREAL;
       Q:     LREAL;
END_STUCT
END_TYPE;

Gain:         ARRAY [1..4] OF LREAL;
Rotation:     ARRAY [1..4,1..4] OF LREAL;
Signal:       ARRAY [1..4] OF DemodComplex;
```

The corresponding OPC and EPICS variables are (with *m* and *n* ranging from 1 to 4):

Type	**OPC name**	**EPICS name**
LREAL	L1.Io.Wfs1.Gain[*m*]	L1:Io-Wfs1_Gain[*m*]
LREAL	L1.Io.Wfs1.Rotation[*m*][*n*]	L1:Io-Wfs1_Rotation[*m*][*n*]
LREAL	L1.Io.Wfs1.Signal[*m*].I	L1:Io-Wfs1_Signal[*m*]_I
LREAL	L1.Io.Wfs1.Signal[*m*].Q	L1:Io-Wfs1_Signal[*m*]_Q

Table 9: Array variables with OPC and EPICS.

Each individual array index will be exported as separate EPICS channel.

5.6 ICS Database Generation

All EPICS fields must be defined in the PLC through OPC properties. TwinCAT will automatically generate an XML file with the extension “.tpy” which can be parsed to generate an EPICS database file. The program EpicsDbGen is available to do this. It is called from the command line as follows:

```
Usage: EpicsDbGen ['options'] -i 'input' -o 'output'
       Generates an EPICS database from a TwinCAT tpy file.
       -ea exports all variables regardless of their opc setting
       -l[l][a|e|b] generate an [extended] [atomic|epics|burt] channel listing
       -r[n|d] [no|dot] conversion rule for EPICS names
       -c[u|l] force upper/lower case for EPICS names
       -nd eliminates leading dot
       -ni replaces array brackets with underscore
       -ns ignores channels of type string
       -sio splits database into input only and input/ouput recrods
       -sn 'num' splits database into files with no more than num records
       -i 'input' input file name (stdin when omitted)
       -o 'output' output database file (stdout when omitted)
```

The input file is the TwinCAT file with the extension “.tpy”. The output file can either be a list of channels or an EPICS database file depending on the arguments. If no input file is specified, input from the standard input is taken. If no output file is specified, the output is written to the standard output.

The argument “-a” specifies that all global variables will be exported. The “-l” argument generates a channel name list rather than a database file. With the “-ll” extension a long list is generated. Normally, the case of the TwinCAT variable is preserved. However, the option “-cu” forces all upper case names, whereas the option “-cl” forces all lower case names. If the option “-ni” is specified, array indices of the form “[n]” are replaced by “_n” when translating to the EPICS channel name.

The option “-sio” can be used to split the database records between input and in/out into separate files. The option “-sn N” can be used to split files, so that they contain no more than N records. Both options can be used individually or combined. If either option is used, an output file has to be specified. For example, if the file name is “PLC.db”, the “-sio” option will generate two files “PLC.in.db” and “PLC.io.db”. With the option “-sn 1000”, we will get “PLC.001.db”, “PLC.002.db”, etc. Each file but the last will contains exactly 1000 records. Both options together will generate files of the form “PLC.in.001.db” and “PLC.io.001.db”.

6 Documentation

A template for documenting a TwinCAT library exists in the DCC, F1200003. It contains the project information, a description of the function blocks as well as detailed listing of the input and output types. Some specialized libraries may require additional information for functions, interfaces or global variables. An example can be found in E1200226.

6.1 Project Information

The following project information is required: title, version, name space, author and a short description.

Field	Description	Mandatory
Title	Name of the library, usually in camel case, e.g., LowNoiseVco	Yes
Version	Library version number, usually 1, 2, etc.	Yes
TwinCAT	Version of TwinCAT for which the library was developed	Yes
Name space	Name space of the library	Yes, if exists
Author	Name of the programmer	Yes
Description	Short description of the purpose of the library	Yes
Error code	Lists the available error codes	Yes

Table 10: Project Information.

6.2 Type Information

Each external type of a library require the following information: name, definition and short description. For a complex type each element should contain a short description as well.

Field	Description	Mandatory
Type name	Name of the type, e.g., LowNoiseVcoStruct	Yes
Definition	Type definition used by the library	Yes
Description	Short description of the purpose of the type	Yes
Elements	For complex types a list of elements	Yes, if exist

Table 11: Type Information.

6.3 Global Variables

Generally, there should be no need for global variables in a library. If they exist, the following information is required: name, type, a possible initialization value and a short description.

Field	Description	Mandatory
Variable name	Name of the global variable	Yes
Type	Type of the global variable	Yes
Initialization	Initialization value of the variable	Yes, if exist
Description	Short description of the purpose of the variable	Yes

Table 12: Global variables.

6.4 Interfaces

In TwinCAT 3 abstract classes are called interfaces. They contain a list of abstract methods. Each interface definition requires name, list of methods and a short description.

Field	Description	Mandatory
Interface name	Name of the type, e.g., LowNoiseVcoStruct	Yes
Methods	List of methods used by the interface	Yes
Arguments	Each method can have a list of arguments	Yes, if exist
Description	Short description of the purpose of the interface	Yes

Table 13: Interfaces.

6.5 Functions

Each function requires the following information: name, return type, list of input parameters, list of output parameters, list of in/out parameters and a short description.

Field	Description	Mandatory
Name	Name of the, e.g., TimingSlaveDuoToneReadFunc	Yes
Return	Return type	Yes
Inputs	List of input parameters	Yes, if exist
Outputs	List of output parameters	Yes, if exist
In/Outs	List of in/out parameters	Yes, if exist
Description	Short description of the purpose of the function or function block	Yes

Table 14: Functions.

6.6 Function Blocks

Each function and function block requires the following information: name, list of input parameters, list of output parameters, list of in/out parameters and a short description. In TwinCAT 3 function block are treated as classes and can extend a base class, inherit from an interface definition and contain methods. If used, the information of all class elements are required.

Field	Description	Mandatory
Name	Name of the function or function block, e.g., LowNoiseVcoFB	Yes
Parent	For classes that extend a parent function block	Yes, if exist
Interfaces	For classes that implement an interface	Yes, if exist
Inputs	List of input parameters	Yes, if exist
Outputs	List of output parameters	Yes, if exist
In/Outs	List of in/out parameters	Yes, if exist
Methods	List of methods used by the function block	Yes, if exist
Description	Short description of the purpose of the function or function block	Yes

Table 15: Function blocks.

6.7 Visuals

Each visual screen element requires the following information: screen snapshot, name, a short description and a list of placeholders. Placeholders are parameters denoted by $paramter_name$ in the visuals that are required to be defined when the visual is embedded. Since the visual of a library usually represents an interface structure, there should be at least one placeholder parameter denoting a variable of this type.

Field	Description	Mandatory
Name	Name of the function or function block, e.g., IscWhiteningVis	Yes
Description	Short description of the purpose of the function or function block	Yes
Placeholder	Parameters used for variable substitution	Yes, if exist

Table 16: Visuals.