Open Advanced Process Control

Software Development Kit
Developers Manual

Version 5.2

Table Of Contents

1 General........4
- 1.1 Disclaimer........4
- 1.2 Overview........4
- 1.3 Feedback and Contributions........4
- 1.4 Contents of the SDK........4

2 Programming Interfaces........5
- 2.1 Flow plug-in Interface........5
  - 2.1.1 General Usage........5
  - 2.1.2 Plug-In Instances........6
  - 2.1.3 Loading and Saving Configurations........6
  - 2.1.4 Function Description........6
  - 2.1.5 BeamConstruct-specific Functions and Structures........16
- 2.2 HMI plug-in Interface........21
  - 2.2.1 General Usage........21
  - 2.2.2 Plug-In Instances........22
  - 2.2.3 Loading and Saving Configurations........22
  - 2.2.4 Dependencies........22
  - 2.2.5 Function Description........22
- 2.3 Error Codes........27
- 2.4 Binary Data Handling and Structures........29
  - 2.4.1 Using of Binary Data Blocks........29
  - 2.4.2 Binary Data related Structures and Definitions........30
    - 2.4.2.1 Data-Type Dependent Parameter Usage........31
    - 2.4.2.2 Binary Data Structure OAPC_BIN_SUBTYPE_STRUCT_CTRL........36
    - 2.4.2.3 Binary Data Structure OAPC_BIN_SUBTYPE_STRUCT_CTRLEND........37
    - 2.4.2.4 Binary Data Structure OAPC_BIN_SUBTYPE_STRUCT_MOTIONCTRL........38
    - 2.4.2.5 Binary Data Structure OAPC_BIN_SUBTYPE_STRUCT_LASERCTRL........38
    - 2.4.2.6 Binary Data Structure OAPC_BIN_SUBTYPE_STRUCT_BITMAP........39
    - 2.4.2.7 Binary Data Structures OAPC_BIN_SUBTYPE_STRUCT_OUTPUTCTRL, OAPC_BIN_SUBTYPE_STRUCT_WAITINPUTCTRL and OAPC_BIN_SUBTYPE_STRUCT_INPUTCTRL........40
    - 2.4.2.8 Binary Data Structure OAPC_BIN_SUBTYPE_STRUCT_SCANHEADINFO........41
    - 2.4.2.9 Binary Data Structure OAPC_BIN_SUBTYPE_STRUCT_POS_CORR........42
    - 2.4.2.10 Binary Data Structure OAPC_BIN_SUBTYPE_STRUCT_DYNGEOMSTART........43
- 2.5 Configuration XML Structure........43
  - 2.5.1 The Symbol Image for the Flow Editor........44
  - 2.5.2 Dialogue Layout and Parameter Definitions........44
  - 2.5.3 Tab Pane Types........45
  - 2.5.4 Input Fields........45
    - 2.5.4.1 Text Input Fields........46
    - 2.5.4.2 Integer Input Fields........46
    - 2.5.4.3 Floating Point Number Input Fields........47
    - 2.5.4.4 Combo Box Fields........47
    - 2.5.4.5 Colour Chooser Button........48
    - 2.5.4.6 Font Chooser Button........48
    - 2.5.4.7 Check Box........49
    - 2.5.4.8 File Load Selection........49
    - 2.5.4.9 File Save Selection........50
    - 2.5.4.10 Select Directory........50
  - 2.5.5 Param-Panel Fields........51
  - 2.5.6 Help-Panel Fields........51
    - 2.5.6.1 Input Connection Tags........52
    - 2.5.6.2 Output Connection Tags........52
  - 2.5.7 Example........52
- 2.6 Developing own plug-ins........53

3 Localisation........53

3.1 Working Principle 53
3.2 Choosing the Correct Translation File Name 54
3.3 Translating the Applications 54
3.4 Creating own Translations 55

4 ControlRoom Interface 56
4.1 Overview 56
4.2 Data Flow 56
4.3 Example Applications 57
4.4 Interface Implementations 57
4.4.1 Java ControlRoom Interface 57
4.4.1.1 The Interface OAPCListener 58
4.4.1.2 The Class OAPCInterface 58
4.4.1.3 The Class SocketEntry 60
4.4.2 C/C++ and other languages ControlRoom Interface 60

5 Interlock Server Connection 61
5.1 Data Flow 61
5.2 Example Applications 61
5.3 Interface Implementations 62
5.3.1 Java Interlock Server Interface 62
5.3.1.1 The Interface OAPCIServerListener 62
5.3.1.2 The Class OAPCIServer 63
5.3.1.3 The Class IServerData 65
5.3.2 C/C++ and other languages Interlock Server Interface 66
5.3.3 Instruction List Interlock Server Interface 66
5.3.4 LUA Interlock Server Interface 66

6 Shared Library liboapc 67
6.1 TCP/IP Related Functions of liboapc 67
6.2 Serial Interface Functions of liboapc 69
6.3 Utility-Functions of liboapc 72
6.3.1 Ring Buffer Utility Functions of liboapc 74
6.4 Thread-Functions of liboapc 75
6.5 Dynamic Library Functions of liboapc 77
6.6 ControlRoom-Interface-Functions of liboapc 77
6.7 Interlock Server Access Functions of liboapc 80

7 Shared Library liboapcwx 84
7.1 Canvas-Functions of liboapcwx 84
7.2 Unicode-Functions of liboapcwx 85
7.3 Path-Functions of liboapcwx 85

8 Shared Library libsmartfactory 87
8.1 Hermes related functions of libsmartfactory 87
8.2 Smart Interface related functions of libsmartfactory 91

1 General

1.1 Disclaimer

This specification and the related sources are distributed in the hope that they will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. All the information given here and within the source code, the interface description and definitions are subject to change without notice. Errors and omissions excepted.

1.2 Overview

The OpenAPC software components offer programming interfaces that are open and can be used from 3rd parties for implementing their own functionalities. Following these open interfaces are specified so that it is possible to use them conform to the functionality of the main software.

The external plug-ins that can be written using these information are nothing more than a shared library (.DLL on windows, .so on Linux) that provide several predefined functions to the main application. The main application itself will call these exported functions in a predefined manner in order to include them into the program flow. Conform to the OSI layer model the communication is done only between the main application and the plug-in, it is not allowed for a plug-in to interact with the user directly.

Beside of that it is also possible to change the language and texts of the application easily. Here no programming is necessary and no advanced technical skills are required. There is only a small set of rules that have to be fulfilled to localise the application successfully.

1.3 Feedback and Contributions

If you extend the application either by external plug-ins or by translation files – or by whatever else is useful for you – and if you want to share your work with others, you can contribute it to us. If your extensions or modifications look useful to us, if you agree that we will include it into the standard software package, so that everybody is able to use it easily and without the need to install additional files or packages we will be happy to make your work become a part of the OpenAPC software package.

PLEASE NOTE: When you send us your data you fully hand over the copyright to us with no limitations. That means we can use it within our application using our licensing models. You also will not be able to withdraw your data, once you gave us the right to use your data you cannot revoke that. Of course we will show your name or your companies name within the credits when you contribute data to us and will provide the data for free as part of our source distribution (except you don't want to have your sources published but want to provide them to us so that we can distribute your work in binary form – where this is possible).

PLEASE NOTE: Plug-ins are accepted as source code only. Here the same is true like for all other data: by sending your data you give us the non-revocable right to use them freely according to our needs and with no limitations.

1.4 Contents of the SDK

The OpenAPC SDK (Software Development Kit) package contains example data and sources related to the following documentation.

Following folders are contained within the SDK:

- **interface** – here interface implementations and example applications to connect to other applications are contained; within this directory an example project file can be found that can be used to access external OpenAPC Interface implementations, examples of such implementations are provided in Java and C
- **iserver** – this directory contains examples about how the Interlock Server can be accessed from own applications or scripts; within this directory an example project file can be found that can be controlled via the Interlock Server; the related controlling scripts in LUA and IL and a C example is also provided
- **plugins** – this folder contains example Flow and HMI plug-ins as described below; the plug-ins contained there are part of the standard software package; here also the header file "oapc_libio.h" is contained that is required for programming own plug-ins
- **liboapc** – the sources of the shared library "liboapc" and the header file "liboapc.h" that is required when the functions/definitions of this header file have to be used
- **translations** – this folder provides several files that can be used as starting point for own translations

2 Programming Interfaces

2.1 Flow plug-in Interface

The Flow plug-in programming interface can be used to create own shared libraries that act as flow control elements within the software. Such plug-ins can be added and edited within the Flow Editor and can

- implement logic flow functions
- access real hardware
- retrieve data from different sources
- ...and other things more

The programming interface itself consists of a set of function calls that have to be provided by the shared library which implements the plug-in functionality. These functions are called by the software (ControlRoom or BeamConstruct) in order to offer comfortable configuration possibilities to the user and to do their job when a project is executed. Following all these functions and the related constants and defines are described. The related function prototypes and defines can be found within a header file "`oapc_libio.h`" that is part of the OpenAPC SDK.

Such a plug-in that is related to the Flow Editor only has to be placed in sub-folder "flowplugins" of the software packages installation.

2.1.1 General Usage

An Flow-plug-ins shared library is used in two contexts: in configuration mode within the editor and in execution mode within the player or debugger.

In configuration mode only some administrative tasks are done, here mainly a configuration description in XML is fetched from the library, the applications Flow Editor converts it, displays a comfortable configuration dialogue and sends the (modified) data back to the library. Beside of that a possibility is given where the software requests the configuration data from the library in order to store them within a common project file. So the plug-in itself does not have to take care about any configuration files or locations.

In execution mode the software asks the library to initialise at application start-up, sends data to it, fetches new data from it and requests a de-initialisation at the end. Here when the main application sets a value to the plug-in and afterwards fetches the returned value resulting from the preceding operation both calls to the plug-in have to last **less than 50 msec** in order to guarantee a smooth flow of the total application. This fact can be checked using the OpenDebugger, here an error message is printed whenever both calls together need more than this time limit. It is important not to exceed that limit because within the OpenPlayer such plug-ins will be rejected and no longer used once they take more time than allowed.

In both modes functions are used that create and delete a new instance of that plug-in and where the software sends formerly saved configuration data to the library so that it can work using these settings.

2.1.2 Plug-In Instances

Whenever the user decides to use an external plug-in the shared library is loaded in order to be used. When a user wants to use the same plug-in again but with other data, the same already loaded shared library is used. Resulting from that standard working principle of shared libraries every use of it requires a separate instance using its own set of data. This set of data can be defined by the plug-in freely, there are no regulations except the fact that these data need to have a structure, that gives the possibility to reduce them to a void-pointer that can be handled by the main application. So from the main applications point of view these instance data are ignored fully, it only stores this pointer.

The usage of the instance mechanism is quite simple. As very first function of the plug-in the main application calls `oapc_create_instance2()`. Here the plug-in should allocate a memory area where all locally used data can be stored into. This memory area should hold a fixed structure that contains all global variables the plug-in needs to do its job and the pointer to that memory area has to be returned at the end of this function.

Now whenever the main application calls a function of the library, it hands over the pointer to that memory area so that the plug-in is able to access these data and to work according to the context it is used within.

When a plug-in is no longer used, as very last function `oapc_delete_instance()` is called handing over the pointer to the memory area again. Now the plug-in hasn't to do any more than releasing that memory area and all other resources that might be somehow related to this instance of the plug-in.

2.1.3 Loading and Saving Configurations

As stated above a plug-in library does not need to store configuration data for its own. That's something that is done by the main software completely. Here two functions are used by the main application to get the data that have to be saved within a project file and to set the data have been loaded out of an project file: `oapc_get_save_data()` and `oapc_set_load_data()`.

For both one thing is important: all the OpenAPC software components are portable and available for many different platforms. So a project file can be written on one system and will be read and used on a completely different hardware platform and operating system. In case these systems use CPUs with a different byte-order the project files would be incompatible if no further actions are taken.

To avoid that problem all data types with a length greater than 8 bit have to be converted to network byte order before they are handed over to the main software. And vice versa if data are given to the plug-in they have to be converted back to host byte order. That method ensures that the project files are portable and the data are readable on every hardware platform.

Such a conversion can be done e.g. with the standard functions `htons()` and `ntohs()` for 16 bit integer data types and with `htonl()` and `ntohl()` for 32 bit integer data types. For more information please refer to related documentations about “endianness” and “network byte order”. Floating point data types should not be stored, they are platform-dependent and their internal structure may not be portable. Therefore they should be replaced by alternative, whole-numbered data when they have to be stored.

2.1.4 Function Description

Following the functions are described that have to be provided by a shared library in order to act as an flow plug-in correctly. For the exact data types and prototype definitions please use the related header file "`oapc_libio.h`" out of the OpenAPC SDK.

The naming of the functions is done from the main applications point of view. So a function `oapc_get_`.will be called by the main application to get something from the shared library, a function `oapc_set_` will be called to hand over something from the main application to the plug-in.

Some of the functions do not have to be provided by the plug-in in case they are not used, others are mandatory. Within the following description all these functions are mandatory for a plug-in, where no explicit information is given that they can be dropped.

`unsigned long oapc_get_capabilities()`

This function is called by the main application. It returns information about the possibilities and capabilities of the plug-in. Depending on what is sent back from this function the main application knows how to handle this library and what it is useful for. The capabilities of the library are specified by a set of OR-concatenated flags that have to be returned by this function call:

- `OAPC_HAS_INPUTS` – the library has inputs and expects data from the main application
- `OAPC_HAS_OUTPUTS` – the library has outputs and expects that the main application fetches the related data
- `OAPC_HAS_XML_CONFIGURATION` – the plug-in requires custom configuration, the configuration information and default data can be provided by a XML structure; for a description of the allowed XML-tags please refer below
- `OAPC_HAS_LOG_TYPE_DIGI` – this flag causes actions on main application side, when it is set the standard object definition dialogue is extended by a logging configuration panel for digital data; this flag applies to all digital outputs, to HMI plug-ins only and does not have any effect for flow plug-ins. Several flags of type `OAPC_HAS_LOG_TYPE_xxx` can't be combined, a plug-in can use exactly one of them exclusively.
- `OAPC_HAS_LOG_TYPE_INTNUM` – this flag causes actions on main application side, when it is set the standard object definition dialogue is extended by a logging configuration panel for whole-numbered data; this flag applies to all numerical outputs, to HMI plug-ins only and does not have any effect for flow plug-ins. It is not allowed to combine more than one flags of type `OAPC_HAS_LOG_TYPE_xxx`, a plug-in can use exactly one of them exclusively
- `OAPC_HAS_LOG_TYPE_FLOATNUM` – this flag causes actions on main application side, when it is set the standard object definition dialogue is extended by a logging configuration panel for floating-point numerical data automatically. This flag applies to all numerical outputs, to HMI plug-ins only and does not have any effect for flow plug-ins. Several flags of type `OAPC_HAS_LOG_TYPE_xxx` can't be combined, a plug-in can use exactly one of them exclusively.
- `OAPC_HAS_LOG_TYPE_CHAR` – this flag causes actions on main application side, when it is set the standard object definition dialogue is extended by a logging configuration panel for text data. This flag applies to all character outputs, to HMI plug-ins only and does not have any effect for flow plug-ins. Several flags of type `OAPC_HAS_LOG_TYPE_xxx` can't be combined, a plug-in can use exactly one of them exclusively.
- `OAPC_ACCEPTS_PLAIN_CONFIGURATION` – this capability constant corresponds to `OAPC_HAS_XML_CONFIGURATION`, it specifies that the data configured by the user have to be returned in plain format using function `oapc_set_config_data()`;
PLEASE NOTE: at the moment this is the only possibility to get back configuration information, so this capability flag has to be set and the related function call `oapc_set_config_data()` has to be provided
- `OAPC_ACCEPTS_IO_CALLBACK` – this flag corresponds to the outputs of the plug-in that transmit their data to the main application (or to be more exact: where the main application fetches the data from). If this flag is not set and if a plug-in defines outputs, they are polled periodically. The polling time can be specified within the application in this case, here all flow configuration dialogues automatically gets an own input field where the desired time can be entered. When this flag is set, no polling is performed and

the related input fields for the polling cycle time are hidden. In this case the main application hands over a function pointer to the plug-in using the function `oapc_set_io_callback()` (please refer below). That function pointer can be used as callback to inform the main application that something has changed that results in modified output states/values.

- `OAPC_USERPRIVI_DISABLE` and `OAPC_USERPRIVI_HIDE` – can be used for HMI plug-ins only; if at least one of both flags is set the user privilege panel is available within the HMI object property dialogue of this plug-in; there a visibility mode can be chosen that depends on the privileges of a user. There is no capability flag for a enabled element or the special state "ignore", both exist in every case whenever a HMI element can be modified depending on the privileges of a user. When flags `OAPC_USERPRIVI_DISABLE` is set, the related plug-in can be set to disabled (but visible), when `OAPC_USERPRIVI_HIDE` is set, the HMI element can be made invisible
- `OAPC_IS_DEPRECATED` marks an plug-in as outdated, this means the related plug-in is no longer visible within the context menus of Flow Editor or HMI Editor but still accessible so that old project files that use this plug-in continue working. By setting this flag it is avoided that this plug-in is used in new projects again.
- `OAPC_ACCEPTS_WRITE_DATA_MODE` – this flag informs the main application about the plug-ins possibility to write some data (e.g. data for stand-alone mode operations). When this capability flag is set the instance of a plug-in can be created using mode `OAPC_INSTANCE_WRITE_DATA`. In this case the plug-in has not to access its hardware directly but has to write the data to the path specified by parameter `oapc_write_data_path` (please refer to description of function `oapc_set_config_data()`)
- `OAPC_ACCEPTS_SEND_DATA_MODE` – this flag informs the main application about the plug-ins possibility to send some data to it (e.g. data for stand-alone mode operations). When this capability flag is set the instance of a plug-in can be created using mode `OAPC_INSTANCE_SEND_DATA`. In this case the plug-in has not to control its hardware directly but has to send the data to it (e.g. for later usage from device-internal memory). This flag has to be used in cases where such data are anonymous/exclusive and can't be identified by some additional identifier.
- `OAPC_ACCEPTS_SEND_NAMED_DATA_MODE` – this flag informs the main application about the plug-ins possibility to send some data to it (e.g. data for stand-alone mode operations). When this capability flag is set the instance of a plug-in can be created using mode `OAPC_INSTANCE_SEND_NAMED_DATA`. In this case the plug-in has not to control its hardware directly but has to send the data to it (e.g. for later usage from device-internal memory). Comparing to `OAPC_ACCEPTS_SEND_DATA_MODE` this variant has to be used when the data to be sent have to be identified by a (unique) name.

Beside of that there is an additional flag set that behaves slightly different. This flag set contains definitions to which category this plug-in belongs to and within which sub menu of the Flow Editor it is listed. Here <u>exactly one</u> of the following category flags has to be OR-concatenated with the capabilities described above:

- `OAPC_FLOWCAT_CONVERSION` – category "Data Conversion", it has to be set when the plug-in mainly performs some kind of conversion between different data formats or types
- `OAPC_FLOWCAT_LOGIC` – category "Logic Operations", it has to be used for plug-ins that combine data logically
- `OAPC_FLOWCAT_CALC` – category "Mathematical Calculations", assign it to plug-ins that do some calculations
- `OAPC_FLOWCAT_FLOW` – category "Flow Control", use this category flag when a plug-in somehow influences the flow of data
- `OAPC_FLOWCAT_IO` – category "Input/Output Operations", whenever a plug-in accesses external devices and sends data to them or receives data from them this category has to be used. This category does not include plug-ins that control motors or perform any other kind of motion, there the special category flag `OAPC_FLOWCAT_MOTION` has to be used
- `OAPC_FLOWCAT_DATA` – category "Data", to be used whenever any kind of data are handled/stored/managed but without sending them to or receiving them from external devices
- `OAPC_FLOWCAT_MOTION` – category "Motion", to be used whenever a plug-in controls a device which

causes any kind of motion (like motor controllers, XY-tables or robots)

- `OAPC_FLOWCAT_LASER` – category “Laser”, to be used for all kinds of laser-processing related plug ins (like scanner controller plug-ins, laser controllers, plug-ins for accessing laser-related software)

If no category flag is specified or in case an illegal/unknown flag is set the appropriate plug-in will be listed in section “Miscellaneous”.

Parameters: none

Return value: the capability flags, please see above

Remarks: Independent from the capability flags specified here all functions described below have to be provided by the shared library. The ones that are not used for a meaningful operation of the plug-in simply have to return `OAPC_ERROR_NOT_SUPPORTED` instead of any operation.

This function is called in both, configuration and execution mode

`char *oapc_get_name()`

This function is called by the main application in order to get a short, descriptive name that is used within the application to identify this plug-in. The name has to be returned in non-UTF 8 bit Latin-1 ASCII characters that are terminated by a 0. It is recommended to give an English name in every case and to provide the localised name via the application-internal localisation mechanism.

Parameters: none

Return value: the name of the plug-in

Remarks: this function is called in both, configuration and execution mode

`unsigned long oapc_get_input_flags()`

In case the `OAPC_HAS_INPUTS` capability bit is set the main application calls this function in order to get information how many inputs of which type are supported by this plug-in. It is possible to define up to 8 inputs with numbers 0..7. Every input can exist exactly once and can have only one data type. Overlapping definitions of more than one data type for the same input are not allowed and will lead to an undefined behaviour. According to these rules following OR-concatenated flags can be returned:

- `OAPC_DIGI_IO0` .. `OAPC_DIGI_IO7` – definition of digital data types for inputs 0..7
- `OAPC_NUM_IO0` .. `OAPC_NUM_IO7` – definition of numerical data types for inputs 0..7
- `OAPC_CHAR_IO0` .. `OAPC_CHAR_IO7` – definition of character data types for inputs 0..7
- `OAPC_BIN_IO0` .. `OAPC_BIN_IO7` – definition of binary data types for inputs 0..7

Digital data can have exactly two states 0 or 1. Numerical data types are a 32 bit floating point number internally, depending on their purpose they can be used for handling integers too. Char data types are texts or single characters. Binary data types are some kind of BLOBs that consist of a standardised header and the binary payload. Details about how such data have to be created, structured ans handled are given below.

Parameters: none

Return value: a list of OR-concatenated flags that specify which inputs are provided, for a description of the flags please see above

Remarks: this function is called in both, configuration and execution mode

`unsigned long oapc_get_output_flags()`

In case the `OAPC_HAS_OUTPUTS` capability bit is set the main application calls this function in order to get information how many outputs of which type are supported by this plug-in. It is possible to define up to 8 outputs with numbers 0..7. Every output can exist exactly once and can have only one data type. Overlapping definitions of more than one data type for the same output are not allowed and will lead to an undefined behaviour.

Parameters: none

Return value: a list of OR-concatenated flags that specify which outputs are provided, for a description of the flags and the available data types please refer to the description of `oapc_get_input_flags()`

Remarks: this function is called in both, configuration and execution mode

`void* oapc_create_instance()`

This function is deprecated and should not be used any longer. Support for it will be removed in future versions. For a replacement please refer to `oapc_create_instance2()`.

`void* oapc_create_instance2(unsigned long flags)`

Whenever the user selects a plug-in for usage within the application or whenever a plug-in is used by the current project, as very first this function is called. Here the plug-in has to allocate a memory area for an internally used structure to hold all global data that are used by that plug-in.

Parameters: flags – usage flag that informs how the plug-in has to be configured
`OAPC_INSTANCE_OPERATION` – normal operation, the plug-in is used in standard mode
`OAPC_INSTANCE_SIMULATION` – simulated mode; this option is important for plug-ins that use specific hardware, when this option is set, the (possibly not available) hardware has not to be accessed but data handling has to be simulated. That means data that would be received by hardware has to be generated, reactions on input data has to be simulated. This mode is useful when setting up a project without all devices available, here such a project can be tested in some kind of dry run.
`OAPC_INSTANCE_MINIMUM_INIT` – this initialisation mode corresponds to function `oapc_read_pvalue()`; when this flag is set it means the initialisation of the plug-in is done in order to call this function afterwards and to read some special values. Depending on which values are read and how they are used during normal initialisation this information can be used by the plug-in to perform a partial initialisation only, e.g. In order to not to overwrite the value that has to be read by some other, default values.
`OAPC_INSTANCE_WRITE_DATA` – the plug-in is used in an operation mode where it has to write data to disk (e.g .stand-alone data) instead of accessing a device directly. This mode corresponds to capability flag `OAPC_ACCEPTS_WRITE_DATA_MODE`.

Return value: a pointer to the memory area, this pointer is handed over back to the plug-in during every relevant function call where these data may be required in order to perform the plug-ins task using a specific set of parameters.

Remarks: These data are not stored within the project file automatically

`void oapc_delete_instance(void* instanceData)`

When a plug-in is unloaded because it is no longer needed or because the application terminates, this function is called as very last one once for every instance that was created for a plug-in. Here the pointer to the private memory area of the plug-in is handed over again so that this memory area can be released.

Parameters: `instanceData` – the memory to be released

Return value: none

Remarks: none

`char *oapc_get_config_data(void* instanceData)`

If there is the capability flag `OAPC_HAS_XML_CONFIGURATION` returned from `oapc_get_capabilities()` to the main application that specifies custom configuration possibilities for a plug-in, this function has to be provided by the library to send a configuration description to the main application

Parameters: `instanceData` – pointer to the memory area that belongs to this instance of the plug-in

Return value: a description of the custom configuration that is used by the main application in order to create and display a configuration dialogue; at the moment only a XML configuration is supported, for a description of the allowed XML tags please refer to the following sections

Remarks: this function is called in configuration mode only

`void oapc_set_config_data(void* instanceData,char *name,char *value)`

This function (without the "const" modifier) is deprecated and should not be used any longer. It will be removed in future software versions. Please refer to `oapc_set_config_data()` below.

`void oapc_set_config_data(void* instanceData,const char *name,const char *value)`

This function is called by the main application to return the (possibly modified) parameters of the application back to the plug-in.
PLEASE NOTE: at the moment the configuration data can be returned only in plain mode, so this function accepts only pairs of name and value. In case a plug-in handles more than one parameter this function is called repeatedly.

Parameters: `instanceData` – pointer to the memory area that belongs to this instance of the plug-in

`name` – pointer to a char-array where the name of the parameter is stored into according to the names defined within the XML description

`value` – a pointer to a char array where the value is stored that is assigned to the parameter with the given `name`; in case the parameter expects a numerical value a conversion from string to integer or from string to float has to be done by the library

Return value: none, the results are stored within the memory areas handed over as function parameters

Remarks: This function is called in configuration mode only. The value returned in `name` is equal to the parameter names within the XML configuration structure. Beside of that there are predefined names that belong to general parameters set by the main application:

- `oapc_cycletime` – when a plug-in doesn't uses a callback function for submitting changed data the outputs are polled cyclically (capability flag `OAPC_ACCEPTS_IO_CALLBACK` not set). In this case the polling cycle time (in milliseconds) is returned together with this parameter.
- `oapc_write_data_path` – specifies the path data have to be written to, this parameter corresponds to capability flags `OAPC_ACCEPTS_WRITE_DATA_MODE` and instance creation mode `OAPC_INSTANCE_WRITE_DATA`

`char *oapc_get_save_data(void *instanceData,unsigned long *length)`

This function is called by the main application whenever a project file has to be saved: it requests a bunch of data from the plug-in to store them within the project file. The format of the data is not defined, the application handles them as anonymous array of bytes. Therefore a plug-in can organise them as desired.

PLEASE NOTE: to keep project files portable and platform independent all used data types with a length greater than 8 bit have to be converted to network byte order before they are given to the main application. That ensures that they are stored in a portable and endianness-independent way so that they can be loaded successfully also with completely different processor architectures.

Parameters: `instanceData` – pointer to the memory area that belongs to this instance of the plug-in

`length` – a pointer to a variable where the total number of bytes of the configuration data have to be stored into; this length defines the size of the configuration data

Return: an array of bytes with the given `length` that has to be stored within a project file

Remarks: this function is called in configuration mode only

`void oapc_set_loaded_data(void *instanceData,unsigned long length,char *data)`

Whenever a project file is loaded the main application calls this function in order to hand over the custom configuration data out of that project file to the related plug-in. After loading a conversion back from network to host byte order has to be done by the plug-in in order to bring the relevant data types back into the correct format for this platform. In execution mode this function is called before `oapc_init()` is requested by the main application.

Parameters: `instanceData` – pointer to the memory area that belongs to this instance of the plug-in

`length` – the number of bytes the loaded dataset consists of

`data` – a pointer to a byte array with the given `length` where the loaded data are stored into

Return value: none

Remarks: this function is called in both, configuration and execution mode

`unsigned long oapc_init(void *instanceData)`

This function is called after possibly existing configuration data have been handed over and before a library is used by setting inputs / retrieving output data. When this function is called, a plug-in has to perform all operations that are necessary in order to provide its desired functionality.

Parameters: `instanceData` – pointer to the memory area that belongs to this instance of the plug-in

Return value: `OAPC_OK` or an `OAPC_ERROR_`... error code that informs the main application about the initialization state; in case of an error this library will not be used any longer. For a description of available error codes please refer below

Remarks: this function is called in execution mode only

`unsigned long oapc_exit(void *instanceData)`

When the main application is exited, this function is called. It gives a plug-in the possibility to de-initialise and to release used resources.

Parameters: `instanceData` – pointer to the memory area that belongs to this instance of the plug-in

Return value: `OAPC_OK` or an `OAPC_ERROR_`... error code; currently the return value is ignored because an error during releasing resources sound somehow pointless, therefore it is recommended to return `OAPC_OK`

Remarks: This function is called in execution mode only. It should NOT be used to release the memory area of `instanceData`, this has to be done during an additional call to `oapc_delete_instance()`.

`unsigned long oapc_set_digi_value(void *instanceData,unsigned long input,unsigned char value)`

This function is called during operation when the `OAPC_HAS_INPUTS` capability and at least one `OAPC_DIGI_IOx` input flag was set. It is called as soon as there is a data flow within the main application that sets a new digital value for an input.

Parameters: `instanceData` – pointer to the memory area that belongs to this instance of the plug-in

`input` - the number of the input in range 0..7 where a new value has to be set

`value` – the new value 0 or 1 that is set for that input

Return value: `OAPC_OK` if the new value could be set successfully or an error code according to the list below

Remarks: this function is called in execution mode only; when no such input type is provided by a plug-in this function does not need to be implemented

`unsigned long oapc_get_digi_value(void *instanceData,unsigned long output,unsigned char *value)`

This function is called during operation when the `OAPC_HAS_OUTPUTS` capability and at least one

`OAPC_DIGI_IOx` output flag was set. It is called from the main application to get new data that might be available at this output.

Parameters: `instanceData` – pointer to the memory area that belongs to this instance of the plug-in

`output` - the number of the output in range 0..7 where a new value is fetched from by the main application

`value` – pointer to a variable where the new value 0 or 1 has to be stored into

Return value: `OAPC_OK` if the new value could be retrieved successfully, `OAPC_ERROR_NO_DATA_AVAILABLE` if no new data are available or an error code according to the list below

Remarks: A value has to be returned to the main application only in case it has really changed or in case the currently available value is really a new one. The plug-in has to avoid that the same value is given to the main application twice without any reason.

This function is called in execution mode only. When no such output type is provided by a plug-in this function does not need to be implemented.

`unsigned long oapc_set_num_value(void *instanceData,unsigned long input,float value)`

This function is called during operation when the `OAPC_HAS_INPUTS` capability and at least one `OAPC_NUM_IOx` input flag was set. It is called as soon as there is a data flow within the main application that sets a new numerical value for an input.

Parameters: `instanceData` – pointer to the memory area that belongs to this instance of the plug-in

`input` - the number of the input in range 0..7 where a new value has to be set

`value` – the new 32 bit floating point value that is set for that input

Return value: `OAPC_OK` if the new value could be set successfully or an error code according to the list below

Remarks: this function is called in execution mode only; when no such input type is provided by a plug-in this function does not need to be implemented

`unsigned long oapc_get_num_value(void *instanceData,unsigned long output,float *value)`

This function is called during operation when the `OAPC_HAS_OUTPUTS` capability and at least one `OAPC_NUM_IOx` output flag was set. It is called from the main application to get new data that might be available at this output.

Parameters:

`instanceData` – pointer to the memory area that belongs to this instance of the plug-in

`output` - the number of the output in range 0..7 where a new value is fetched from by the main application

`value` – pointer to a variable where the new 32 bit floating point value has to be stored into

Return value: `OAPC_OK` if the new value could be retrieved successfully, `OAPC_ERROR_NO_DATA_AVAILABLE` if no new data are available or an error code according to the list below

Remarks: A value has to be returned to the main application only in case it has really changed or in case the currently available value is really a new one. The plug-in has to avoid that the same value is given to the main application twice without any reason.

This function is called in execution mode only. When no such output type is provided by a plug-in this function does not need to be implemented.

`unsigned long oapc_set_char_value(void *instanceData,unsigned long input,char *value)`

This function is called during operation when the `OAPC_HAS_INPUTS` capability and at least one `OAPC_CHAR_IOx` input flag was set. It is called as soon as there is a data flow within the main application that sets new characters for an input.

Parameters: `instanceData` – pointer to the memory area that belongs to this instance of the plug-in

`input` - the number of the input in range 0..7 where a new value has to be set

`value` – the new zero-terminated C character string that is set for that input

Return value: `OAPC_OK` if the new value could be set successfully or an error code according to the list below

Remarks: this function is called in execution mode only; when no such input type is provided by a plug-in this function does not need to be implemented

`unsigned long oapc_get_char_value(void *instanceData,unsigned long output, unsigned long length,char *value)`

This function is called during operation when the `OAPC_HAS_OUTPUTS` capability and at least one `OAPC_CHAR_IOx` output flag was set. It is called from the main application to get new data that might be available at this output.

Parameters: `instanceData` – pointer to the memory area that belongs to this instance of the plug-in

`output` - the number of the output in range 0..7 where a new value is fetched from by the main application

`length` – the size of the character array where value points to

`value` – pointer to a character array where the new value has to be stored into, the plug-in in no case is allowed to copy more data to that location than the parameter `length` allows, that would lead to a crash of the application

Return value: `OAPC_OK` if the new value could be retrieved successfully, `OAPC_ERROR_NO_DATA_AVAILABLE` if no new data are available or an error code according to the list below

Remarks: A value has to be returned to the main application only in case it has really changed or in case the currently available value is really a new one. The plug-in has to avoid that the same value is given to the main application twice without any reason.

This function is called in execution mode only. When no such output type is provided by a plug-in this function does not need to be implemented.

`unsigned long oapc_set_bin_value(void *instanceData,struct oapc_bin_head input,char *value)`

This function is called during operation when the `OAPC_HAS_INPUTS` capability and at least one `OAPC_BIN_IOx` input flag was set. It is called as soon as there is a data flow within the main application that sets a new binary data block for an input.

Parameters: `instanceData` – pointer to the memory area that belongs to this instance of the plug-in

`input` - the number of the input in range 0..7 where a new value has to be set

`value` – a structure that describes the binary data and is followed by the data immediately, please refer below for a description of this structure.

Return value: `OAPC_OK` if the new value could be set successfully or an error code according to the list below

Remarks: this function is called in execution mode only; when no such input type is provided by a plug-in this function does not need to be implemented.
The binary data are <u>not allowed to be used directly</u>. A plug-in can read them only until the function oapc_set_bin_value() returns. When the data have to be used afterwards or when the data have to be modified by the plug-in, they need to be copied before this function returns or before they are modified.

unsigned long oapc_get_bin_value(void *instanceData, unsigned long output, struct **oapc_bin_head)

This function is called during operation when the `OAPC_HAS_OUTPUTS` capability and at least one `OAPC_BIN_IOx` output flag was set. It is called from the main application to get new data that might be available at this output.

Parameters: `instanceData` – pointer to the memory area that belongs to this instance of the plug-in

`output` - the number of the output in range 0..7 where a new value is fetched from by the main application

`value` – pointer where the pointer to the binary data have to be stored to

Return value: `OAPC_OK` if the new value could be retrieved successfully, `OAPC_ERROR_NO_DATA_AVAILABLE` if no new data are available or an error code according to the list below

Remarks: A value has to be returned to the main application only in case it has really changed or in case the currently available value is really a new one. The plug-in has to avoid that the same value is given to the main application twice without a good reason.

This function is called in execution mode only.
After this function returns the plug-in is no longer allowed to use the returned binary data, it doesn't is allowed to modify them, it is not allowed to return them during an other function call and it is not allowed to release the related memory areas. After returning from this function the main application can use this binary data structure exclusively.

Directly after the call to this function the main application will call the function `oapc_release_bin_data()` to give the plug-in the possibility to release all resources related to these binary data. This call happens immediately, no other functions of the plug-in are executed in between.

When no binary data output type is provided by a plug-in this function does not need to be implemented.

void oapc_release_bin_data(void* instanceData,unsigned long output)

Memory areas that have been allocated by a plug-in have to be released by the same plug-in as soon as they are no longer needed. Therefore the main application calls this function whenever a fetched binary data structure is no longer needed. Here the order is as follows:

1. the main application calls `oapc_get_bin_value()` to get binary data created by a plug-in
2. now the main application is the exclusive owner of these data, the plug-in is no longer allowed to access them in any way
3. directly after calling `oapc_get_bin_value()` the main application calls `oapc_release_bin_data()` to hand over access privileges back to the plug-in, now it is able to re-use the data or to release them, the main application will have copied them at this point (if necessary) so that no collisions happen when the binary data are accessed

The parameters of this function specify which binary data can be released by the application, here the `instanceData` of the related instance are handed over and the `output` number of the binary output where the data have been fetched from.

void oapc_set_io_callback(void *instanceData,lib_oapc_io_callback oapc_io_callback,unsigned long callbackID)

This function is called by the main application only when the capability flag `OAPC_ACCEPTS_IO_CALLBACK` was set. It hands over a pointer to a callback function and a unique callback identifier. Both values have to be stored. Whenever the internal state of the plug-in changes so that its output states/values change, this callback function has to be called together with the ID.

Parameters: `instanceData` – pointer to the private memory area that belongs to this instance of the plug-in

`oapc_io_callback` – pointer to a function that has to be called whenever the output states change; it is not necessary to store this pointer within the private instance data structure of this plug-in, this pointer is the same for all plug-ins and for all instances of them

`callbackID` – a unique identifier for this specific instance of the plug-in that has to be sent back to the main application when the callback function is called; different to the function pointer this value has to be stored within the instance memory area

Return value: none

Remarks: The callback function itself is defined as
`oapc_io_callback(unsigned long outputs,unsigned long callbackID).`
As parameters it expects the `outputs`, a list of OR-concatenated `OAPC_xxx_IO` flags that specify which outputs have changed and therefore need to be accessed using the related `oapc_get_xxx_value()` function call. The second parameter `callbackID` is the unique identifier that was announced by the main application before. Here simply the value has to be sent back to the application that was given together with the call of the function `oapc_set_io_callback()` before.

This function is called in execution mode only.

`unsigned long oapc_get_error_message(void *instanceData, unsigned long length, char *value)`

This function is called immediately after an other function of the same plug-in returned an error code `OAPC_ERROR_CUSTOM` to get the error text from the plug-in that has to be displayed as reaction to the returned error. The new text must not exceed the given `length` and has to be copied into `value`. This function itself can return any of the existing error codes but normally should give back `OAPC_OK`.
The error text should be given in English using plain, Latin-1 ASCII format. Translation to other languages is done within the main application using the integrated translation mechanism as described below.

`unsigned long oapc_read_pvalue(void* instanceData,double param,double *value)`

There exist a special kind of XML definition that can be used to create a panel with freely definable parameters and values. When this panel type <parampanel> is used there exists a possibility within the settings dialogue of a plug-in to read data out of it. Since reading these values may be done by accessing the hardware a plug-in accesses there is a special order of events when the related button is pressed within the plug-ins configuration:

- a new instance is created using the flag `OAPC_INSTANCE_MINIMUM_INIT` to singalise the plug-in not to overwrite values that could be read by the following operation
- the device is initialised by calling `oapc_init()`
- `oapc_read_value()` is called to read the parameter with number given in `param`; the value read out of it has to be stored in pointer to `value` and `OAPC_OK` has to be returned in case of success, otherwise an error code has to be returned
- the device is closed and the instance is destroyed

When the call to this function returns successfully the value given back by the double-pointer is shown within the parameter panel.

2.1.5 BeamConstruct-specific Functions and Structures

BeamConstruct offers the possibility to access motion controllers directly out of the application via suitable motion plug-ins. To find out which of the plug-ins are useable for such a BeamConstruct-internal operation the application checks all available plug-ins and tries to request some specific data from them. As soon as such data are found and in case they fit to BeamConstruct, the related plug-in can be used out of the application.

Retrieval of these special information is done via a function `oapc_get_config_info_data()` that has to be provided by such a motion plug-in and that returns detailed information about the motion controller. In case BeamConstruct does not find such a function within a plug-in or in case the function is available but returns invalid or unknown data, the related plug-in is rejected and not used for BeamConstruct internal operations.

`unsigned long oapc_get_config_info_data(void *instanceData,struct config_info *fillStruct)`

This function is called by BeamConstruct in order to get additional information about a plug-in, about its capabilities and features. The function is called after creation of a new instance so that in parameter `instanceData` valid instance-specific data are handed over to the plug-in. The second parameter `fillStruct` is a pointer to a structure that is held by BeamConstruct and that has to be filled with valid data as described below. In case that operation could be finished successfully the function has to return with `OAPC_OK` or with an error code otherwise.

The structure of type `struct config_info` is defined in header file "oapc_libio.h" and consists of following elements:

```
struct config_info
{
   unsigned short version,length;
   unsigned int   configType;
   union
   {
      struct config_motion_controller  motionController;
      struct config_image_capture      imageCapture;
      struct config_zshifter           zShifter;
      struct config_scanner_controller scannerController;
      struct config_laser_controller   laserController;
      struct config_pcontrol           pControl;
   }
};
```

`version` – specifies the version of the structure, this member in every case has to be set to `OAPC_CONFIG_INFO_VERSION` which is always defined correctly according to the current version of the structure and handled BeamConstruct-internally

`length` – the total length of the structure, to make it easy this member has to be set to `sizeof(struct config_info)` always

`configType` – this value specifies the type of the plug-in and the type of the structure within the following union that has to fit to this plug-in. Currently following types and corresponding structures are supported:

`OAPC_CONFIG_TYPE_MOTIONCONTROLLER` corresponds to `struct config_motion_controller` and specifies motion-controller specific parameters:

```
struct config_motion_controller
{
   unsigned short version,length;
   unsigned char  availableAxes;
   unsigned char  useBinOutput;
   unsigned char  rotationalAxes;
   unsigned char  pad2;
   unsigned int   pad4;
   int            uMinPos[8],uMaxPos[8];
   int            uMaxSpeed[8];
   unsigned int   flags;
};
```

`version` – the version of the structure, this field has to be set to `OAPC_CONFIG_MOTION_CONTROLLER_VERSION` so that it automatically fits to the current version of the structure, different available versions and variants itself are handled BeamConstruct-internal

`length` – the total length of this structure, this member has to be set to `sizeof(struct config_motion_controller)`

`availableAxes` – one motion plug-in can support up to eight separate axes internally. Via this member the plug-in has to specify how much and which axes are available and configured properly so that they can be used by BeamConstruct. This is done via the bits of this structure member, every bit that is set to 1 corresponds to an axis that can be used. Here bit 0 is equal to the first axis, al following bits correspond to the following axes.

`useBinOutput` – this member has NOT to be filled by the plug-in, this is done by BeamConstruct. Here a value 2 or 3 specifies if the motion plug-in will be assigned to OUT2 or OUT3 (according to the motion plug-ins position within BeamConstruct and according to the assigned motion-output of the “BeamConstruct To Control” plug-in within a ControlRoom environment). The plug-in in no case should touch this variable.

`rotationalAxes` – a motion plug-in can drive axes in planar or rotational mode. Via this member the plug-in has to specify how much and which axes are configured for rotational mode. This is done via the bits of this structure member, every bit that is set to 1 corresponds to an axis that is rotational, every bit set to 0 is an axis that operates in planar/linear mode. Here bit number 0 is equal to the first axis, al following bits correspond to the following axes.

`uMinPos[8]` – specifies the minimum position (in unit micrometers) for every available axis that can be handled by the plug-in

`uMaxPos[8]` – specifies the maximum position (in unit micrometers) for every available axis that can be handled by the plug-in

`uMaxSpeed[8]` - specifies the maximum allowed speed (in unit micrometers per second) for every available axis that can be handled by the plug-in

`flags` – this member is currently unused, it is prepared for future use and has to be set to 0

`OAPC_CONFIG_TYPE_IMAGECAPTURE` corresponds to `struct config_image_capture` and specifies parameters related to image-capturing plugins:

```
struct config_image_capture
{
    unsigned short version,length;
    unsigned short frameDelay; // delay in msec between two frames
};
```

`version` – the version of the structure, this field has to be set to `OAPC_CONFIG_IMAGE_CAPTURE_VERSION` so that it automatically fits to the current version of the structure, different available versions and variants itself are handled BeamConstruct-internal

`length` – the total length of this structure, this member has to be set to `sizeof(struct config_image_capture)`

`frameDelay` – the delay that has to be issued after capturing one image (in unit milliseconds), this value influences the frame rate

`OAPC_CONFIG_TYPE_LASERCONTROLLER` corresponds to `struct config_laser_controller` and keeps data about separate plug-ins accessing a laser (e.g. via serial interface or Ethernet but not directly via used scanner controller):

```
struct config_laser_controller
{
   unsigned short version,length;
   unsigned short reserved0,reserved1;
   unsigned int   capabilities;
   unsigned int   reserved2,reserved3,reserved4,reserved5;
};
```

`version` – the version of the structure, this field has to be set to `OAPC_CONFIG_LASER_CONTROLLER_VERSION` so that it automatically fits to the current version of the structure, different available versions and variants itself are handled BeamConstruct-internal

`length` – the total length of this structure, this member has to be set to `sizeof(struct config_laser_controller)`

`capabilities` – this field contains a set of OR-concatenated flags which specify what parameters this controller is able to manage. Depending on these flags the related fields are enabled in BeamConstruct's pen settings dialogue. Here following flags are available:
- `OAPC_LC_HAS_LASERON` – the controller is able to turn the laser on
- `OAPC_LC_HAS_LASEROFF` – the controller is able to turn the laser on
- `OAPC_LC_HAS_FREQ` – the controller can set frequencies at the laser
- `OAPC_LC_HAS_POWER` – the controller can change the power for the laser

Beside of these values this structure currently does not hold any specific parameters. Listed structure members are reserved for future use and have to be set to 0.

`OAPC_CONFIG_TYPE_ZSHIFTER` corresponds to `struct config_zshifter` and keeps data about separate plug-ins accessing a third axis that is controlling the focus or Z-height. This configuration is intended to be used for accessing a Z-axis that is NOT the one that can be handled by scanner controller:

```
struct config_zshifter
{
    unsigned short version,length;
    unsigned short reserved0,reserved1;
    unsigned int   reserved2,reserved3,reserved4,reserved5;
};
```

`version` – the version of the structure, this field has to be set to `OAPC_CONFIG_ZSHIFTER_VERSION` so that it automatically fits to the current version of the structure, different available versions and variants itself are handled BeamConstruct-internal

`length` – the total length of this structure, this member has to be set to `sizeof(struct config_zshifter)`

Beside of these values this structure currently does not hold any specific parameters. Listed structure members are reserved for future use and have to be set to 0.

`OAPC_CONFIG_TYPE_PCONTROL` corresponds to `struct config_pcontrol` and keeps data about separate plug-ins that are able to control additional process parameters in parallel to a marking process. This includes definitions for additional pen parameters that can be given by a user in pen settings dialogue and that are sent to the plug-in during a marking operation whenever the pen changes:

```
struct config_pcontrol
{
   unsigned short version,length;
   unsigned int   flags;
   // *** parameters to be used within pen settings *******************
   char           penPanelName[15];
   unsigned int   paramFlag[OAPC_PCONTROL_MAX_CUST_PARAMS];
   int            paramMin[OAPC_PCONTROL_MAX_CUST_PARAMS],
                  paramMax[OAPC_PCONTROL_MAX_CUST_PARAMS],
                  paramDef[OAPC_PCONTROL_MAX_CUST_PARAMS];
   unsigned int   paramFloatFactor;
   char           paramName[OAPC_PCONTROL_MAX_CUST_PARAMS][40];
   char           paramUnit[OAPC_PCONTROL_MAX_CUST_PARAMS][20];
   unsigned int   dispFlag[OAPC_PCONTROL_MAX_CUST_PARAMS];
   char           dispName[OAPC_PCONTROL_MAX_CUST_PARAMS][40];
   int            dispMin[OAPC_PCONTROL_MAX_CUST_PARAMS],
                  dispMax[OAPC_PCONTROL_MAX_CUST_PARAMS];
   char           dispUnit[OAPC_PCONTROL_MAX_CUST_PARAMS][20];
   unsigned short reserved0,reserved1;
   unsigned int   reserved2,reserved3,reserved4,reserved5;
};
```

`version` – the version of the structure, this field has to be set to `OAPC_CONFIG_PCONTROL_VERSION` so that it automatically fits to the current version of the structure, different available versions and variants itself are handled BeamConstruct-internal

`length` – the total length of this structure, this member has to be set to `sizeof(struct config_pcontrol)`

The following members of this structure are used to display additional parameters in BeamConstructs pen settings:

`penPanelName` – when a process control plug-in is used within BeamConstruct, a new tab-pane is shown in pen settings; here a ASCII-string can be given specifying a name for this panel

`paramFlag` – for each of the `OAPC_PCONTROL_MAX_CUST_PARAMS` custom pen parameters a flag can be specified describing what kind of value is expected; when this is ste to 0, the related parameter value input field is not shown. Here following values can be given:

- `OAPC_CONFIG_PCONTROL_FLAG_INT_TYPE` – the parameter is an integer value
- `OAPC_CONFIG_PCONTROL_FLAG_FLOAT_TYPE` – the parameter is a floating point value

`paramMin` – the minimum value the pen parameter is allowed to have; in case a floating point value is shown in pen settings, `paramFloatFactor` is used to convert from integer to float

`paramMax` – the maximum value the pen parameter is allowed to have; in case a floating point value is shown in pen settings, `paramFloatFactor` is used to convert from integer to float

`paramDef` – the default value the pen parameter is set to; in case a floating point value is shown in pen settings, `paramFloatFactor` is used to convert from integer to float

`paramName` – an ASCII-string that is displayed as the name for the related parameter

`paramUnit` – an ASCII-string that is displayed as the unit for the related parameter

These members are values that are measured by the plug-in and that can be displayed in an own window during marking process:

`dispFlag` – this flag specifies what kind of parameter has to be displayed, it is used to show a suitable symbol; here following values are allowed:

- `OAPC_CONFIG_PCONTROL_FLAG_TEMPERATURE_STYLE` – the value to be displayed is a temperature value, so a thermometer symbol is shown
- `OAPC_CONFIG_PCONTROL_FLAG_PRESSURE_STYLE` – the value to be displayed is a pressure value

- `OAPC_CONFIG_PCONTROL_FLAG_BRIGHTNESS_STYLE` - the value to be displayed is a brightness value

`dispName` – the name of the parameter to be shown in the values window

`dispMin` – minimum value to be shown, reserved for later usage, set it to the theoretical minimum that can be shown here

`dispMax` – maximum value to be shown, reserved for later usage, set it to the theoretical maximum that can be shown here

`dispUnit` – the unit of the parameter to be shown in the values window right beside the value

2.2 HMI plug-in Interface

The HMI plug-in programming interface can be used to create own shared libraries that act as separate user interface element within the software. Such plug-ins can be added and edited within the HMI Editor and can be combined with other elements within the Flow Editor. A HMI plug-in is able to

- visualise states and conditions
- accept direct user input
- implement logic flow functions
- retrieve data from different sources
- ...and other things more

The programming interface itself consists of a set of function calls that include all the functionalities described in section 2.1 Flow plug-in Interface. That subset of functions implements the flow functionality part of the plug-in. Beside of that an additional set of functions exists that have to be provided by the plug-in in order to implement the visual / HMI part functionality of the plug-in.

2.2.1 General Usage

An HMI-plug-in shared library is used in two contexts: in configuration mode within the HMI Editor and the Flow Editor and in execution mode within the player or debugger.

In configuration mode only some administrative tasks are done, here mainly the configuration description in XML format is fetched from the library, the applications HMI Editor or Flow Editor converts it, displays a comfortable configuration dialogue and sends the (modified) parameter values back to the library. Beside of that a possibility is given where the software requests the configuration data from the library in order to store them within a common project file. So the library itself does not to take care about any configuration files or locations.

In execution mode the software asks the library to initialise at application start-up, sends data to it, fetches new data from it and requests a de-initialisation at the end. Here when the main application sets a value to the plug-in and afterwards fetches the returned value resulting from the preceding operation both calls to the plug-in have to last **less than 50 msec** in order to guarantee a smooth flow of the total application. This fact can be checked using the OpenDebugger, here an error message is printed whenever both calls together need more than this time limit. It is important not to exceed that limit because within the OpenPlayer such plug-ins will be rejected and no longer used once they take more time than allowed.

In both modes functions are used where the main software sends formerly saved configuration data to the library so that it can work using these settings and where the main application sends graphical information to the plug-in so that the user interface element can be drawn by the plug-in.

2.2.2 Plug-In Instances

Exactly as described for the Flow plug-in here the same concept of instances and private instance data is used. That means also for a HMI plug-in the functions `oapc_create_instance2()` and `oapc_delete_instance()` have to be provided by the plug-in as described above.

2.2.3 Loading and Saving Configurations

Similar to a Flow plug-in the HMI plug-in does not need to store configuration data for its own. That's also done by the main software in the same way than described above.

Here one additional thing is important: There is no difference between HMI and flow configuration data, so when an application has custom parameters for the HMI and the flow functionality of it, both datasets have to be handled with the same calls of `oapc_get_save_data()` and `oapc_set_load_data()`, means the full set of custom configuration data has to be managed with them.

2.2.4 Dependencies

Different to a Flow plug-in a shared library that implements a HMI plug-in depends on the free wxWidgets tool kit That is the user interface tool kit the main application is developed with. Now when the HMI plug-in wants to access the graphical user interface of the main application and needs to receive user interaction events from the main software it has to use the same software framework, therefore it needs to compile with the wxWidgets headers and use its functions. Linking has to be done against its shared Unicode, non-universal libraries under Windows, for all other operating systems the configuration of the standard WX-system packages has to be used.

For detailed information about the programming with the wxWidgets tool kit please refer to http://www.wxwidgets.org , there different developer resources are available. You also will find help there related to programming with wxWidgets.

Within the function specification below a short description is given whenever a wxWidgets functionality or data type is used, for a more detailed description of them please refer to the official wxWidgets documentation

To avoid that also a Flow plug-in – which does not need to use any wxWidgets data type or functionality – also depends on this tool kit a switch `OAPC_EXT_HMI_EXPORTS` is used. When it is not defined, all HMI- and wxWidgets-related definitions are disabled, when it is defined, they are enabled and the wxWidgets header files have to exist. So you will be able to develop a HMI plug-in only when this definition is done within your programming environment.

2.2.5 Function Description

Following the functions are described that have to be provided by a shared library in order to act as an HMI plug-in correctly. Here only these functions are described that are used for the HMI plug-in exclusively or the ones that have additional meanings and or additional parameters/flags for a HMI plug-in. So for a full description of functions that have to be provided by a HMI shared library please first refer to 2.1.3 Function Description.

For the exact data types and prototype definitions please use the related header file “`oapc_libio.h`” out of the OpenAPC SDK. To enable the HMI definitions within that header file `OAPC_EXT_HMI_EXPORTS` needs to be defined, it encapsulates all wxWidgets-related definitions so that a plain Flow plug-in does not need to depend on them.

`unsigned long oapc_get_capabilities()`

This function is called by the main application. It returns information about the possibilities and capabilities of the plug-in. Depending on what is sent back from this function the main application knows how to handle this library and what it is useful for. The capabilities of the library are specified by a set of OR-concatenated flags that have to be returned by this function call. Beside the capability-flags that are described for flow plug-ins above, for a HMI plug-in following additional flags are valid:

- `OAPC_HAS_STANDARD_FLOW_CONFIGURATION` – when this flag is set, there is no XML configuration requested for the flow configuration, instead of it a standard dialogue is used like it is known e.g. from the Text Field control or the Horizontal/Vertical Gauge display. To use this plain configuration the combination of inputs and outputs has to consist at least of a digital input and output 0 and an optional, numerical or character input and output 7. Beside of that there – optionally – can be a pair of numerical or character inputs and outputs 6 and an – also optional – digital input 1. More inputs or outputs are not allowed when this simple configuration possibility is used. When it is used, no flow image is required and the parameters that have been set by the user within this standard configuration are stored by the main application automatically.
  To check if a specific combination of inputs and outputs is valid to be used within the main application, the following procedure is recommended:
  1. define this capability flag
  2. define your desired combination of input and output flags
  3. do not provide a XML configuration (and resulting from that do not provide a flow image)
  4. add the HMI element provided by this plug-in to the HMI Editor
  5. put the HMI element that was added within the preceding step into the Flow Editor
  If now a symbol with a "?" sign is displayed, the chosen combination of IO's can't be used together with this capability, an own XML configuration structure has to be provided instead
- `OAPC_ACCEPTS_MOUSECLICKS` – the plug-in wants to receive mouse click events within the canvas it is responsible for
- `OAPC_ACCEPTS_MOUSEMOVES` – when there is a mouse movement detected the plug-in needs to be informed about that
- `OAPC_ACCEPTS_MOUSEDRAGS` – this parameter is a mixture out of the preceding two ones, it enables the main application to send mouse event information to the plug-in whenever the mouse is moved with the left mouse button held down within the plug-ins canvas

Beside of that there is an additional flag set that behaves slightly different. This flag set contains definitions to which HMI category this plug-in belongs to and within which sub menu of the HMI Editor it is listed. These flags have to be used exclusively and can't be combined with the OAPC_FLOWCAT_-flags that are reserved for a Flow plug-in: after a HMI plug-in is added only within the HMI Editor and put to the Flow Editor afterwards there can't be a definition for a Flow Editor category. So here exactly one of the following category flags has to be OR-concatenated with the other capabilities:

- `OAPC_HMICAT_CONTROL` – the plug-in implements a HMI element that expects user interaction or input
- `OAPC_HMICAT_DISPLAY` – the user interface element implemented by this plug-in displays some information
- `OAPC_HMICAT_STATIC` – the visual representation implemented by this plug-in is a static one, therefore its name will be listed within the section "Static" of the related menu within the HMI Editor

If no category flag is specified or in case an illegal/unknown flag is set the appropriate plug-in will be listed in section "Miscellaneous".

Parameters: none

Return value: the capability flags, please see above

Remarks: Independent from the capability flags specified here all functions described below have to be provided by the shared library. The ones that are not used for a meaningful operation of the plug-in simply have to return `OAPC_ERROR_NOT_SUPPORTED` instead of any operation.

This function is called in both, configuration and execution mode

`unsigned long oapc_get_no_ui_flags()`

Within the HMI Editor a configuration dialogue can be opened, that consists of a standardised base-set of parameters that can be changed there. This dialogue can't be hidden and the parameters set there are managed by the main application completely. After not all of the parameters that can be set there make sense for a particular HMI element, some of them can be disabled so that the user no longer is able to enter values for these unused or senseless parameters. Which of these parameters have to be disabled is specified by this function. The main application calls it and expects a combination of OR-concatenated flags that specify which parts of the HMI base parameters are not used. When this function returns 0, all parameters are enabled, combinations of the following flags disable parts of it:

- `OAPC_HMI_NO_UI_DISABLED` – the state-checkbox “Disabled” is turned off when this flag is set
- `OAPC_HMI_NO_UI_RO` – this flag should be used when the user interface element does not support a state “read only”, it disables the related option
- `OAPC_HMI_NO_UI_MINMAX` – when this flag is set, the input fields for minimum and maximum values are disabled
- `OAPC_HMI_NO_UI_TEXT` – this flag disables the possibility to set texts for this user element
- `OAPC_HMI_NO_UI_FG` – no foreground colour can be chosen when this flag is set and returned by this function
- `OAPC_HMI_NO_UI_BG` – no background colour can be chosen when this flag is set and returned by this function
- `OAPC_HMI_NO_UI_FONT` – this option disables the possibility to choose a font for the HMI element
- `OAPC_HMI_NO_SIZE` – when this flag is set, the possibility to enter a size is turned of completely; this flag can't be combined with the following one
- `OAPC_HMI_SIZE_FIXED_ASPECT` – in case resizing is allowed but the aspect ratio between width and height has to be kept, this flags should be set; it disables the input field for the height and updates it automatically whenever the width is changed (and vice versa)
- `OAPC_HMI_NO_POS` – when this flag is set there is no possibility to modify the position of the HMI element out of the configuration dialogue
- `OAPC_HMI_NO_UI_LAYOUT` – the UI element can't be managed by the internal, automatic layout

Parameters: none

Remarks: The flags returned here influence the functionality of the elements main configuration panel within the HMI Editor only. Beside of that it is still possible to implement additional, custom configuration possibilities by providing additional panels for that dialogue via an XML structure. For more details please refer to the description of `oapc_get_hmi_config_data()`.

This function is called in configuration mode only.

`void oapc_get_defsize(wxFloat32 *x,wxFloat32 *y)`

This function returns the default size the HMI element is created with when it is added to the HMI Editor for the first time.

Parameters: `x` – pointer to a variable to store the default width into

`y` – pointer to a variable to store the default height into

Remarks: This function is called in configuration mode only.

`void oapc_get_minsize(void *instanceData,wxFloat32 *x,wxFloat32 *y)`

This function returns the minimum size for an HMI element. When the user performs scaling operations the main application will ensure that the size never will fall of the minimum size specified here.

Parameters: `hmiInstance` – pointer to the memory area that is assigned to this instance of the plug-in

`x` – pointer to a variable to store the minimum width into

`y` – pointer to a variable to store the minimum height into

Remarks: This function is called in configuration mode only.

`void oapc_get_maxsize(void *instanceData,wxFloat32 *x,wxFloat32 *y)`

This function returns the maximum size for an HMI element. When the user performs scaling operations the main application will ensure that the size specified here never will be exceeded.

Parameters: `hmiInstance` – pointer to the memory area that is assigned to this instance of the plug-in

`x` – pointer to a variable to store the maximum width into

`y` – pointer to a variable to store the maximum height into

Remarks: This function is called in configuration mode only.

`void oapc_get_numminmax(void *instanceData,wxFloat32 *minValue,wxFloat32 *maxValue)`

Using this function the main application requests the current minimum and maximum values that are used by this HMI plug-in. When a HMI element is added to the HMI editor for the first time this function should return default minimum and maximum values. For all further calls these values should be returned that are set by preceding calls of `oapc_set_numminmax()`.

Parameters: `hmiInstance` – pointer to the memory area that is assigned to this instance of the plug-in

`minValue` – pointer to a variable to store the minimum value into

`maxValue` – pointer to a variable to store the maximum value into

Remarks: This functionality applies to HMI plug-ins only that handle numerical values. Independent from that this function has to be provided by every kind of HMI plug-in. Where no numerical values are used it simply should write 0 into both returning variables.

This function is called in configuration mode only.

`void oapc_set_numminmax(void *instanceData,wxFloat32 minValue,wxFloat32 maxValue)`

When this function is called by the main application the current numerical minimum and maximum values are returned that are configured by the user within the HMI configuration dialogue. The values that are handed over here are stored by the main application, so it is not necessary for the plug-in to put them into the custom data that are used for loading and saving.

Parameters: `hmiInstance` – pointer to the memory area that is assigned to this instance of the plug-in

`minValue` – the new minimum value

`maxValue` – the new maximum value

Remarks: This functionality applies to HMI plug-ins only that handle numerical values. Independent from that this function has to be provided by every kind of HMI plug-in. Where no numerical values are used it simply should ignore the values given here.

This function is called in execution mode and in configuration mode.

`void oapc_get_colours(wxUint32 *background,wxUint32 *foreground)`

This function is executed by the main application exactly once: when the HMI element that is implemented by this plug-in is added to the HMI editor for the very first time. Here the default values for background and foreground colour are returned. After that, both colours are managed and stored by the main application completely, there is no need to take care of them afterwards.

The colours have to be handed over in format 0xBBGGRR where RR stands for the red portion, GG for the green portion and BB of the blue portion of the colour.

Parameters: background – pointer to a variable to store the background colour information into

foreground – pointer to a variable to store the foreground colour information into

Remarks: This function is called once in configuration mode.

char *oapc_get_hmi_config_data(void *instanceData)

When the configuration dialogue for a user interface element of the HMI editor is opened, it consists of one standardised tabbed pane. This default tab pane can be used to configure and modify standard parameters. In case it is necessary to offer extended configuration possibilities some more tab panes can be added to this dialogue. These tab panes can be defined using an XML structure that is returned by this function. For a description of the XML structure and which of its parameters can be used to create what kind of tab pane and input element, please refer below.
Parameters: hmiInstance – pointer to the memory area that is assigned to this instance of the plug-in

Remarks: Within the XML structure returned by this function no flow image has to be defined.

This function is called in configuration mode only.

void oapc_paint(void *instanceData,wxAutoBufferedPaintDC *dc,wxPanel *canvas)

This function is called by the main application whenever the HMI element has to be (re)painted. So the plug-in has to redraw everything that is required to display the HMI element to the user.

Parameters: hmiInstance – pointer to the memory area that is assigned to this instance of the plug-in

dc – the buffered drawing context; this class provides several drawing functionalities in context of the graphics element that is managed by this plug-in. Here the wxWidgets data type wxAutoBufferedPaintDC is used for double-buffering to avoid flickering also when several redrawing operations are done. wxAutoBufferedPaintDC inherits its methods and functionalities from wxDC, so for a detailed description please refer to http://docs.wxwidgets.org/stable/wx_wxdc.html

canvas – the canvas the drawing context belongs to and that is assigned to this plug-in. For every HMI plug-in the main application adds a wxPanel to the main window in order to encapsulate the access of the HMI plug-in. That means, a user can draw freely within that canvas and does not take care about other elements. This value can also be used to evaluate the current size that was specified for the user interface element by the main application. Here the function wxPanel->GetClientSize() has to be called, there is no additional interface function where the main application explicitly informs about resizing. The function oapc_paint() is called by the main application whenever the size of the element has changed.
The wxPanel class is described at http://docs.wxwidgets.org/stable/wx_wxpanel.html

Remarks: This function is called in execution mode and in configuration mode.

void oapc_mouseevent(void *instanceData,wxMouseEvent* event)

When at least one of the OAPC_ACCEPTS_MOUSE... capability flags is set, this function is called by the main application whenever a related mouse event occurs within the canvas this plug-in is responsible for.

Parameters: hmiInstance – pointer to the memory area that is assigned to this instance of the plug-in

event – the mouse event; using the methods of this object the exact type of the mouse event can be evaluated, for a description please refer to http://docs.wxwidgets.org/stable/wx_wxmouseevent.html

This function is called in execution mode only.

2.3 Error Codes

Following the error codes are defined that can be used by a plug-in to inform the main application about illegal states and failed operations. According to the OSI layer model, a shared library from such a low level (like the ControlRoom plug-ins work on) never should interact with the end user directly. Therefore several common error codes are defined that result in meaningful message boxes on application level (when necessary). So it is important to use an error code that really fits to the problem that occurred.

Following error codes can be used:

`OAPC_OK`

No error occurred, an operation could be completed successfully.

`OAPC_ERROR_CONNECTION`

A connection error happened, it was not possible to establish a (communication) connection that is required for an operation. This error code can be returned e.g. during initialization or when handling with input/output data.

`OAPC_ERROR_DEVICE`

A device error occurred, a required device is not available, could not be accessed, could not be opened or similar things. This error code can be returned e.g. during initialization or when handling with input/output data.

`OAPC_ERROR_RESOURCE`

A required resource could not be accessed or allocated. This error code can be returned e.g. during initialization or when handling with input/output data.

`OAPC_ERROR_NO_MEMORY`

The plug-in was not able to allocate memory

`OAPC_ERROR_AUTHENTICATION`

An authentication error occurred because a user name, password, pin, code or anything similar did not match. This error code can be returned e.g. during initialization.

`OAPC_ERROR_NOT_SUPPORTED`

According to the capabilities of a plug-in some of the functions that have to be provided by the shared library can't be supported by the plug-in and never will be called by the application. After these function calls have to be provided independent from the capabilities they simply have to return this “not supported” error.

`OAPC_ERROR_NO_DATA_AVAILABLE`

This error code specifies that there are no new data are available at the requested output.

`OAPC_ERROR_NO_SUCH_IO`

The main application specifies input and output numbers according to the flags that have been set by the plug-in. That means, normally no requests should take place for inputs the plug-in is unable to handle. But to implement a correct range-checking and error handling this error code should be returned to the main application for all these invalid input numbers. As an example: an plug-in specifies the digital inputs 2 and 3 are used. The related function that is called to set these data now will handle data for input 2 and 3 correctly and return with OAKC_OK. For all other cases (input 0, 1, 4, 5, 6 and 7) the data have to be ignored and this error code has to be returned.

OAPC_ERROR_CONVERSION_ERROR

There was an error during conversion of data types. This error code has to be returned in case a given value had to be converted to a different format but did not fit to a format/structure that would be necessary for a successful conversion.

OAPC_ERROR_STILL_IN_PROGRESS

This error can be returned by an input function in case the new data that are handed over by the main application can't be handled at the moment. It tells the application that the new request comes to fast or too early to be processed because the plug-in still tries to manage the data given in a preceding call.

OAPC_ERROR_RECV_DATA

This error has to be returned when the reception of data from an external source has failed.

OAPC_ERROR_SEND_DATA

When the transmission of data to an external target was not possible this error code has to be returned to the main application.

OAPC_ERROR_PROTOCOL

an error with/in a communication protocol of the plug-in occurred, this may happen because of different reasons, e.g. a wrong sequence of commands to the plug-in, an incompatible version or others

OAPC_ERROR_INVALID_INPUT

the input data are invalid, they are either out of the allowed range or – for binary data – the data type is wrong/unknown or in wrong/unknown compression state

OAPC_ERROR_CREATE_FILE_FAILED

the creation of a new file or opening a file for appending data failed; this error code has to be used whenever opening a file for writing failed

OAPC_ERROR_OPEN_FILE_FAILED

opening of an existing file failed, this error code has to be used whenever opening of a file for reading data out of it failed

OAPC_ERROR_WRITE_FILE_FAILED

writing of data into a file failed, this error code has to be used when opening of a file was successful but writing data into it is not possible

OAPC_ERROR_READ_FILE_FAILED

reading data from a file failed, this error code has to be used when opening of a file was successful but reading the (expected) data is not possible

`OAPC_ERROR_CUSTOM`

an error occurred where the plug-in itself has some additional error information. Directly after this error code was returned, the main application calls function `oapc_get_error_message()` of the plug-in to retrieve an error text which has to be displayed. In such a case the plug-in has to store the error information internally to send back the correct error text when this function is called

`OAPC_ERROR_LICENSE`

This error is returned when a function is called that can't be executed due to a missing or wrong license.

`OAPC_ERROR_LIBRARY_MISSING`

Some plug-in require a 3rd party shared library that is not delivered with OpenAPC package. Such a shared library normally should be installed in /usr/lib, /usr/lib64, WINDOWS\SYSTEM32, WINDOWS\SYSWOW64 or similar folders. When such a library could not be found/opened by a plug-in, this error code has to be returned in order to show a informative error message to the user that informs about the missing library and the location it should be copied into.

`OAPC_ERROR_STOPPED`

An operation was interrupted by an (external) stop-event and therefore could not be completed successfully.

`OAPC_ERROR_OUT_OF_RANGE`

A (measured) value is out of its allowed range

`OAPC_ERROR`

A general, not exactly specifiable error occurred. This error code normally should never be used. There is only one conditions where it would be allowed to return this value: when an error is a follow-up of a preceding error that was specified by an other error code more in detail.

2.4 Binary Data Handling and Structures

2.4.1 Using of Binary Data Blocks

Binary data have to be used in a different way than the other data types. Because these data can have a big size every data block exists only once. During the flow of data only references to the same data block are forwarded between the elements of a project.

Resulting from that following rules are important for a plug-in

- it is allowed to access data only during the main application calls the plug-in; in case they have to be used later a copy of the data (including head and appended data block) has to be made
- handed over data are not allowed to be modified by a plug-in; in case that is necessary a copy has to be made before modification
- after a plug-in has returned a binary data block the main application can use this data block exclusively, the plug-in no longer is allowed to access this data block in any way. In case these data are still needed a copy has to be made before the data are sent to the main application. This is very important in case a plug-in uses an own thread that may access such a data block asynchronously. In such cases additional locking mechanisms may be necessary to avoid concurrent accesses to this

data block.

2.4.2 Binary Data related Structures and Definitions

A binary data block consists of a head of type `struct oapc_bin_head` that contains all necessary information about the data and the binary data itself. That payload is appended to the head directly.

The binary data head contains the following members:

```
struct oapc_bin_head
{
   int           version;
   int           sizeHead; // deprecated, do not use any more!
   unsigned char type,subType;
   unsigned char compression;
   unsigned char unit;
   short         unitExponent;
   short         int1;
   int           param1,param2,param3;
   int           sizeData;
   char          data;
};
```

`version` – a version number that describes the version of this structure, it has to be set to `OAPC_BIN_HEAD_CURR_VERSION` when a new data structure is created

`sizeHead` – this element is deprecated, instead of the value given here use `sizeof(struct oapc_bin_head)`

`type` – this element specifies the type of the data, here one of the `OAPC_BIN_TYPE_ttt` constants has to be set. When a new data type has to be introduced because the current ones do not apply, please use the placeholder `OAPC_BIN_TYPE_CUSTOM` during development and contact us to get a unique ID for your application.

`subType` – this element specifies the subtype of the data that belongs to a data type and specifies it more exactly. Here one of the `OAPC_BIN_SUBTYPE_ttt_sss` constants has to be set. When a new data subtype has to be introduced because the current ones do not apply, please use the placeholder `OAPC_BIN_TYPE_ttt_CUSTOM` during development and contact us to get a unique ID for your application. For a list of existing data type constants and their meaning please refer to header file oapc_libio.h.

`compression` – a data block may be compressed, here the compression type is an additional property of the data and encapsulates the real data type described by the `type` and `subType` members. The used compression is specified by the `OAPC_COMPRESS_ccc` constants. In case the data are not compressed this value is set to `OAPC_COMPRESS_NONE`. For a list of existing data subtype constants and their meaning please refer to header file oapc_libio.h.

`unit` – this field is valid only for some special data types and specifies the related measurement unit using one of the constants `OAPC_BIN_UNIT_ttt_uuu`.

`unitExponent` – in case a unit is used for the data type this one specifies an exponent. Positive exponent values are used for exponents greater than 0 (e.g. 3 for “kilo”, 6 for “mega”, 9 for “giga” and so on) negative values define values smaller than 0 (e.g. -1 for “milli”, -6 for “micro”, -9 for “nano”).

`int1` – this value is used application-internal and does not have to be changed; in case a structure is created newly it has to be set to 0

`param1, param2, param3` – user-defined, data-dependent values; these values can be used to specify parameters that belong to the data. For a description how these parameters are used for which data types and subtypes please refer below. In case you want to use these parameters for data types/subtypes where no usage is defined, please contact us so that we can specify this usage and to avoid incompatibilities

`sizeData` – the total size of the data block

`data` – this member is the position of the first byte of the payload. That means the payload and the data head overlap here for one byte

To create a binary data block including a valid binary data head it is recommended to use function `oapc_util_alloc_bin_data()` out of liboapc. Releasing of such a block afterwards can be done by calling `oapc_util_alloc_bin_data()`.

2.4.2.1 Data-Type Dependent Parameter Usage

Following some data types, subtypes, their meaning and the usage of their parameter-members are described. The usage scenario of the types and parameters listed here is mandatory, do not use them in a different way to avoid incompatibilities. When you need additional/missing parameter definitions and/or need to introduce new parameters, please contact us so that we can specify them for you.

Data Type	Data Subtype	Description	param1	param2	param3
`OAPC_BIN_TYPE_IMAGE`	`OAPC_BIN_SUBTYPE_IMAGE_RGB24`	uncompressed raw image data	image width in pixels	image height in pixels	unused, image depth is 24 bit
`OAPC_BIN_TYPE_IMAGE`	`OAPC_BIN_SUBTYPE_IMAGE_GREY8`	uncompressed raw grey-scale image data	image width in pixels	image height in pixels	unused, image depth is 8 bit
`OAPC_BIN_TYPE_IMAGE`	`OAPC_BIN_SUBTYPE_IMAGE_BW1`	uncompressed raw black/white image data or mask bitmap	image width in pixels	image height in pixels	unused, image depth is 1 bit
`OAPC_BIN_TYPE_TEXT`	`OAPC_BIN_SUBTYPE_TEXT_PLAIN`	Contains plain, 7 bit ASCII text data in payload	number of characters in text, for ASCII 7 bit this always should be equal to the data size	unused	unused
`OAPC_BIN_TYPE_STRUCT`	`OAPC_BIN_SUBTYPE_STRUCT_CTRL`	coordinate motion control structure including tool information	unused	unused	unused
`OAPC_BIN_TYPE_STRUCT`	`OAPC_BIN_SUBTYPE_STRUCT_CTRLEND`	marker for end of coordinate motion control data	Unique identifier to specify a point in data stream, 0 means no synchronisation information is available; please also	unused	unused

			refer to `OAPC_BIN_SUBTYPE_STRUCT_SYNC` below		
`OAPC_BIN_TYPE_STRUCT`	`OAPC_BIN_SUBTYPE_STRUCT_MOTIONCTRL`	motion control structure	unused	unused	unused
`OAPC_BIN_TYPE_STRUCT`	`OAPC_BIN_SUBTYPE_STRUCT_WAITTRIG`	marker for stop processing until an external trigger was detected or until distance has passed by	when not equal 0: distance in unit micrometers after what a trigger has to be released automatically	unused	unused
`OAPC_BIN_TYPE_STRUCT`	`OAPC_BIN_SUBTYPE_STRUCT_DELAY`	stop current process for given time	Delay in unit nanoseconds	unused	unused
`OAPC_BIN_TYPE_STRUCT`	`OAPC_BIN_SUBTYPE_STRUCT_LASERCTRL`	Contains special laser control parameters	unused	unused	unused
`OAPC_BIN_TYPE_STRUCT`	`OAPC_BIN_SUBTYPE_STRUCT_OUTPUTCTRL`	output analogue and/or digital data via device	if value greater 0: switch back the outputs to previous values after the given time (in unit nanoseconds) has elapsed; if value smaller than 0: the last delay value has to be increased by this value for the number of times specified with param3	If value greater 0: wait for the given time (in unit nanoseconds) until processing next binary command; if value smaller than 0: the last delay value has to be increased by this value for the number of times specified with param3	loop counter; if value is 0..1 the pulse specified by param1 and param2 has to be issued once, if it is greater 1 these pulses have to be issued for this number of times
`OAPC_BIN_TYPE_STRUCT`	`OAPC_BIN_SUBTYPE_STRUCT_OUTPUTRESP`	Response to a `OAPC_BIN_SUBTYPE_STRUCT_OUTPUTCTRL` structure after it was handled			loop counter result; this value informs the calling application how much of the loops that have been requested could be issued really
`OAPC_BIN_TYPE_STRUCT`	`OAPC_BIN_TYPE_STRUCT_INPUTCTRL`	fetch analogue and/or digital input signals	unused	unused	unused

		from a device			
OAPC_BIN_TYPE_STRUCT	OAPC_BIN_TYPE_STRUCT_BITMAP	additional bitmap parameters that are required in case a bitmap has to be output via a special device	unused	unused	unused
OAPC_BIN_TYPE_STRUCT	OAPC_BIN_TYPE_STRUCT_MARKREADY	Signalises a plug-in to make ready for marking – this is issued typically when the mark dialogue is opened	unused	unused	unused
OAPC_BIN_TYPE_STRUCT	OAPC_BIN_TYPE_STRUCT_ENDMARKREADY	Signalises a plug-in to no longer be ready for marking – this is issued typically when the mark dialogue is closed	unused	unused	unused
OAPC_BIN_TYPE_STRUCT	OAPC_BIN_TYPE_STRUCT_JOBSTART	Signalises a plug-in a marking job is started now – this is issued typically immediately when marking process is started	unused	unused	Unused
OAPC_BIN_TYPE_STRUCT	OAPC_BIN_TYPE_STRUCT_JOBEND	Signalises a plug-in a marking job is finished now – this is issued typically immediately after all marking data have been processed	unused	unused	unused
OAPC_BIN_SUBTYPE_STRUCT	OAPC_BIN_SUBTYPE_STRUCT_SCANHEADNFO	Contains information about the connected scanhead and its current state	unused	unused	unused
OAPC_BIN_SUBTY	OAPC_BIN_SUBT	Stop output of	unused	unused	unused

PE_STRUCT	YPE_STRUCT_STOPOUTPUT	data as fast as possible			
OAPC_BIN_SUBTYPE_STRUCT	OAPC_BIN_SUBTYPE_STRUCT_STARTOUTPUT	Start output of data; this structure can be send from a plug-in to main application	unused	unused	unused
OAPC_BIN_SUBTYPE_STRUCT	OAPC_BIN_SUBTYPE_STRUCT_BRAKEOUTPUT	Stop output of data regularly, comparing to `OAPC_BIN_SUBTYPE_STRUCT_STOPOUTPUT` this command does not expect immediate stop, here data already queued for optput still can be sent	unused	unused	Unused
	OAPC_BIN_SUBTYPE_STRUCT_AXISSTATE	Returns current position and speed of an axis; this structure can be used in cases where more axes are controlled than can be signalled by available outputs	0-based number of the axis the following values are valid for	Current axis position in unit mm	Current axis speed in unit mm/sec
OAPC_BIN_SUBTYPE_STRUCT	OAPC_BIN_SUBTYPE_STRUCT_DYNGEOMSTART	Specifies the beginning of a stream of dynamic data starting with a `struct oapc_bin_struct_dyn_data` and followed by additional data until end is signalised	unused	unused	unused
OAPC_BIN_SUBTYPE_STRUCT	OAPC_BIN_SUBTYPE_STRUCT_DYNGEOMEND	Signalises the end of dynamic data	unused	unused	unused
OAPC_BIN_SUBTYPE_STRUCT	OAPC_BIN_SUBTYPE_STRUCT_RESETTIMER	An plug-in can send this to the main application to reset the process timer; this is useful	unused	unused	Unused

		when an operation does not start immediately after data have been sent but some unpredictable time later			
`OAPC_BIN_SUBTYPE_STRUCT`	`OAPC_BIN_SUBTYPE_STRUCT_SLICESTART`	Signals the beginning of a (new) slice	Thickness of current slice in unit micrometers	unused	unused
`OAPC_BIN_SUBTYPE_STRUCT`	`OAPC_BIN_SUBTYPE_STRUCT_SLICEEND`	Signals the end of current slice	Thickness of current slice in unit micrometers	unused	unused
`OAPC_BIN_SUBTYPE_STRUCT`	`OAPC_BIN_SUBTYPE_STRUCT_FASTOUTPUTCTRL`	output analogue and/or digital data via device; comparing to `OAPC_BIN_SUBTYPE_STRUCT_FASTOUTPUTCTRL` a plug-in has to output these data immediately and as fast as possible – independent from the state of all previously send commands	if value greater 0: switch back the outputs to previous values after the given time (in unit nanoseconds) has elapsed; if value smaller than 0: the last delay value has to be increased by this value for the number of times specified with param3	unused	unused
`OAPC_BIN_SUBTYPE_STRUCT`	`OAPC_BIN_SUBTYPE_STRUCT_STOPAXIS`	Signals a plug-in to stop motion of an axis	number of the axis to be stopped in range 0..7	unused	unused
`OAPC_BIN_SUBTYPE_STRUCT`	`OAPC_BIN_SUBTYPE_STRUCT_HALT`	With this binary structure the main application signals a plug-in to halt or to continue a running operation	1 – halt the current operation 2- contiune a previously halted operation	unused	unused
`OAPC_BIN_SUBTYPE_STRUCT`	`OAPC_BIN_SUBTYPE_STRUCT_POWERDOWN`	Turn off the power at a connected device	unused	unused	Unused
`OAPC_BIN_SUBTYPE_STRUCT`	`OAPC_BIN_SUBTYPE_STRUCT_SY`	Signal a specific synchronsation	Unique identifier to	unused	unused

	NC	point within a data stream via a unique identifier	specify a point in data stream, 0 means no synchronisation information is available; please also refer to OAPC_BIN_SUBTYPE_STRUCT_CTRLEND above		

2.4.2.2 *Binary Data Structure* OAPC_BIN_SUBTYPE_STRUCT_CTRL

The binary data defined by type OAPC_BIN_TYPE_STRUCT and subtype OAPC_BIN_SUBTYPE_STRUCT_CTRL are a formatted structure that contains information about motion and tool data that can be used for material processing:

```
struct oapc_bin_struct_ctrl
{
   unsigned int                 validityFlags;
   unsigned char                on;
   unsigned char                reserved1;
   unsigned short               numCoords;
   int                          power,frequency;
   unsigned int                 offSpeed,onSpeed;
   int                          offDelay,onDelay;
   int                          toolParam[6];
   struct oapc_bin_struct_vec3d coord[1];
};
```

validityFlags – some of the fields within this structure are optional, they may not contain valid data in some cases; using the bits validity-flags-fields the valid ones are specified. Following constants are used here:

- TOOLPARAM_VALIDFLAG_Z – this is the only parameter that may not change between different data packets of the same stream, it signalises if the control data are for 3D operations or if they are flat and the Z coordinate is unused. In case this flag is set/not set for one stream it has to be set/unset for all other data packets of the same stream because plug-ins and devices that accept such data don't have to manage data that toggle between 2D and 3D.
- TOOLPARAM_VALIDFLAG_ON – when this flag is set the value in field on (as described below) is valid and has to be used
- TOOLPARAM_VALIDFLAG_POWER – when this flag is set the field power contains a (new) value
- TOOLPARAM_VALIDFLAG_FREQ – this flags points to a valid field frequency
- TOOLPARAM_VALIDFLAG_OFFSPEED – this flag signals a possibly changed value in field offSpeed
- TOOLPARAM_VALIDFLAG_ONSPEED – when this flag is set the value in field onSpeed may have changed
- TOOLPARAM_VALIDFLAG_OFFDELAY – when this flag is set the field offDelay is valid and may contain a changed value

- TOOLPARAM_VALIDFLAG_ONDELAY – this flag belongs to field onDelay, the related value may have changed when it is set
- TOOLPARAM_VALIDFLAG_PARAM1 – this flags belongs to the optional parameter that is stored in toolParam[0]
- TOOLPARAM_VALIDFLAG_PARAM2 – this flags belongs to the optional parameter that is stored in toolParam[1]
- TOOLPARAM_VALIDFLAG_PARAM3 – this flags belongs to the optional parameter that is stored in toolParam[2]
- TOOLPARAM_VALIDFLAG_VARIABLE_PARAM3 – this flags belongs to the optional parameter that is stored in toolParam[2], different to TOOLPARAM_VALIDFLAG_PARAM3 when this flag is used, the value stored there is variable and depends on some external parameters, both TOOLPARAM_VALIDFLAG_PARAM3 and TOOLPARAM_VALIDFLAG_VARIABLE_PARAM3 can't be used together, they exclude each other
- TOOLPARAM_VALIDFLAG_PARAM4 – this flags belongs to the optional parameter that is stored in toolParam[3]
- TOOLPARAM_VALIDFLAG_PARAM5 – this flags belongs to the optional parameter that is stored in toolParam[4]
- TOOLPARAM_VALIDFLAG_PARAM6 – this flags belongs to the optional parameter that is stored in toolParam[5]
- TOOLPARAM_DESCFLAG_IS_SLICE – this flag does not belong to a parameter of this structure, it is an additional description for the vector data marking them as part of a slice generated out of a 3D mesh

on – this value can have the states 0 or 1 and signals if a tool is on or not

numCoords – this value corresponds to the field coord, it specifies how much coordinate values do really exist in this structure, means the size of numCoords is equal to the real size of the array coord

power – a power value in unit % and with a valid range from 0 to 100

frequency – a frequency in unit Hz

offSpeed – motion speed (in unit micrometers per second) that has to be used when the tool is turned off

onSpeed – motion speed (in unit micrometers per second) that has to be used when the tool is turned on

offDelay – delay (in unit microseconds) when the tool is switched off

onDelay – delay (in unit microseconds) when the tool is switched on

toolParam – up to six additional, optional tool parameters that can be used freely

coord – an array of motion/position coordinates with the length specified by field numCoords, this structure contains separate fields for x, y and the optional z value (in unit micrometers) and is defined as follows:

```
struct oapc_bin_struct_vec3d
{
   int x,y,z;
};
```

2.4.2.3 Binary Data Structure *OAPC_BIN_SUBTYPE_STRUCT_CTRLEND*

The binary data defined by type OAPC_BIN_TYPE_STRUCT and subtype OAPC_BIN_SUBTYPE_STRUCT_CTRLEND does not contain any additional data and belongs to the binary data type OAPC_BIN_SUBTYPE_STRUCT_CTRL. It identifies the end of a binary control data stream and therefore can be used to start processes that require the full set of available control data.

2.4.2.4 Binary Data Structure *OAPC_BIN_SUBTYPE_STRUCT_MOTIONCTRL*

The binary data defined by type `OAPC_BIN_TYPE_STRUCT` and subtype `OAPC_BIN_SUBTYPE_STRUCT_MOTIONCTRL` are a formatted structure that contains plain information about motion for several axes but no synchronous tool data.

```
struct oapc_bin_struct_motionctrl
{
   unsigned char enableAxes;
   unsigned char relativeMovement;
   unsigned char stopAxes;
   unsigned char moveAxesToHome;
   unsigned int  position[8];
   unsigned int  speed[8];
};
```

Using one structure up to eight axes can be supported, currently there are three axes in use at maximum from within BeamConstruct or CNConstruct. The members of this structure define which axis has to be moved in which speed by which distance:

`enableAxes` – this is a bit-pattern that defines which of the axes have to be moved, here every bit that is set corresponds to an axis where movement data are available

`relativeMovement` – this bit-pattern specifies if a movement information for an axis specifies a relative movement or not, when a bit for an axis is set to 0 it means the `position`-parameter specifies an absolute movement, in case it is set to 1 a relative movement is specified

`stopAxes` – this is a bit-pattern that overrides all other parameters except than `enableAxes` and specifies which of the enabled axes has to be stopped immediately

`moveAxesToHome` – this is a bit-pattern that overrides all other parameters except than `enableAxes` and specifies which of the enabled axes have to be moved to its home or reference position; this movement is done using the for homing/referencing that is predefined for the related motion controller

`position` – the position an axis has to be moved to (in case of absolute movements) or the value the position has to be changed by (in case of relative movements), the value is specified in unit micrometers and is valid only for these indices of the array where the bit number of `enableAxis` is set to 1

`speed` – the elements of this array define the motion speed for the axes to be moved in unit micrometers per second, the values given here are valid only for these indices where the corresponding bit number of the structure member `enableAxis` is set to 1

2.4.2.5 Binary Data Structure *OAPC_BIN_SUBTYPE_STRUCT_LASERCTRL*

The binary data defined by type `OAPC_BIN_TYPE_STRUCT` and subtype `OAPC_BIN_SUBTYPE_STRUCT_LASERCTRL` are used to transmit a structure that contains several laser parameters. These parameters are stored within a structure `oapc_bin_struct_laserctrl` and should be checked logically: only these values have to be set that really fit for the currently used laser type. Some of them are no clear laser parameters but are used as a combination of laser and scanner parameters. They reside within this structure because they belong to “laser marking” and therefore to the used lasers in general:

```
struct oapc_bin_struct_laserctrl
```

```
{
   unsigned short version,res;
   unsigned int   uPolyDelayBreakAngle;
   unsigned int   nPulseLength;
   unsigned int   uWobbleAmp,mWobbleFreq;
   unsigned int   uPulseLength;
   unsigned int   nFirstPulse;
   unsigned int   res;
   unsigned int   nStandByPulseWidth;
   unsigned int   standByFreq;
   unsigned int   waveformNum;
   unsigned int   edgeLevel;
   unsigned int   skyTimeLag,skyOnShift,skyPrev,skyPost;
   unsigned int   rampMode;
   unsigned int   uRampStartLen,uRampEndLen;
   unsigned int   mRampStartVal,mTampEndVal;
};
```

`version` – used for upwards compatibility, the current structure uses version number 1 which is increased every time a new member is added or something else is changed

`uPolyDelayBreakAngle` – specifies an angle in unit micro-degrees that is used as threshold to apply a polygon delay

`nPulseLength` – laser pulse length in unit nanoseconds

`uWobbleAmp` and `mWobbleFreq` – specify amplitude (in unit micrometers) and frequency (in Hz/1000) for wobble-modulation during a marking operation, if at least one of both values is set to 0 wobble is disabled

`uPulseLength` – this value is deprecated and will be removed, please use `nPulseLength` instead!

`nFirstPulse` – first pulse time in unit nanoseconds

`nStandByPulseWidth` – stand by pulse length in unit nanoseconds

`standByFreq` – stand by frequency in unit Hz

`waveformNum` – laser waveform number to be used for output

`edgeLevel` – level of an edge as threshold for applying a polygon delay to it

`skyTimeLag`, `skyOnShift`, `skyPrev`, `skyPost` – these are parameters for sky writing mode to get clear and sharp edges and corners in marked geometries

`rampMode` – ramping mode flags, they specify the ramping mode (speed or power) and position (beginning or end of a marking operation) to be used together with the following parameters

`uRampStartLen`, `uRampEndLen`, `mRampStartVal`, `mTampEndVal` – ramping parameters to perform speed and/or power ramping using specific start and end values over a given time

2.4.2.6 Binary Data Structure *`OAPC_BIN_SUBTYPE_STRUCT_BITMAP`*

This structure type flag makes use of a structure `oapc_bin_struct_bitmap` which contains some additional bitmap information. These information can be sent before a image of one of the main types `OAPC_BIN_TYPE_IMAGE` is transmitted to a device:

```
struct oapc_bin_struct_bitmap
{
   int uPosOffsetX,uPosOffsetY,uPosOffsetZ;
   int udPixelSizeX,udPixelSizeY,udPixelSizeZ;
   int flags;
};
```

`uPosOffsetX`, `uPosOffsetY`, `uPosOffsetZ` – offset coordinate information that specify where to position the bitmap at the output device, these fields give the offset in unit micrometers

`udPixelSizeX`, `udPixelSizeY` – the size of every pixel in x and y direction, these values have to be used to scale the image to a size according to the devices output capabilities (unit is micrometers)

`udPixelSizeZ` – reserved for future use (since current bitmaps do not have a thickness it is currently unused)

`flags` – this member contains a set of OR-concatenated flags that describe how the output of the bitmap should be done and give some hints about its structure, here following values are possible:

- `BITMAP_FLAG_MARK_BIDIRECTIONAL` – output of the bitmap should be done bidirectional, means as soon as the end of the first line was reached the second line should be started at the end and processed to its beginning, then the third line should be started from the beginning to the end again and so on
- `BITMAP_FLAG_MARK_FROM_LAST_LINE` – output of the bitmap should be done from top to bottom, means starting from last line and moving up to first line
- `BITMAP_FLAG_MARK_WO_LINE_INCR` - only an output of lines should be done, the vertical dimension (its height) should be ignored and all lines should be output at the same position

2.4.2.7 Binary Data Structures *`OAPC_BIN_SUBTYPE_STRUCT_OUTPUTCTRL`*, *`OAPC_BIN_SUBTYPE_STRUCT_WAITINPUTCTRL`* and *`OAPC_BIN_SUBTYPE_STRUCT_INPUTCTRL`*

Followed by these structure sub-types there is always a structure `oapc_bin_struct_ioctrl` attached. This structure contains information about output ports that have to be set, input states a device has to wait for until operation is continued or input ports values that have been read.

```
struct oapc_bin_struct_ioctrl
{
   unsigned int   enableFlags;
   unsigned char  laserport8[2];
   unsigned char  digital8[2];
   unsigned char  digital8mask[2];
   unsigned short digital16[2];
   unsigned short digital16mask[2];
   unsigned char  analogue8[2];
   unsigned short analogue16_0;
   unsigned short analogue10[6];
   unsigned short analogue12[4];
   unsigned short analogue16_1,analogue16_2;
   unsigned long  digital32;
   unsigned long  digital32mask;
   char           serialData[IOCTRL_SERIAL_DATA_LENGTH+1]
   unsigned char  laserport8mask[2];
};
```

`enableFlags` – these flags specify which of the following members contain valid data and therefore can be used for setting an output port or for reading the state of an input port from. This member can contain following OR-concatenated flags:

- `IOCTRL_LASERPORT_8_1` - the first laserport member of this structure is used for setting full numeric values, this flag can't be combined with `IOCTRL_LASERPORT_8_1_BITS`

- `IOCTRL_LASERPORT_8_2` - the second laserport member of this structure is used for setting full numeric values, this flag can't be combined with `IOCTRL_LASERPORT_8_1_BITS`
- `IOCTRL_ANALOGUE_8_1` - the first 8 bit analogue port member of this structure is used
- `IOCTRL_ANALOGUE_10_1` - the first 10 bit analogue port member of this structure is used
- `IOCTRL_ANALOGUE_10_2` - the second 10 bit analogue port member of this structure is used
- `IOCTRL_ANALOGUE_10_3` - the third 10 bit analogue port member of this structure is used
- `IOCTRL_ANALOGUE_10_4` - the fourth 10 bit analogue port member of this structure is used
- `IOCTRL_ANALOGUE_10_5` - the fifth 10 bit analogue port member of this structure is used
- `IOCTRL_ANALOGUE_10_6` - the sixth 10 bit analogue port member of this structure is used
- `IOCTRL_ANALOGUE_12_1` - the first 12 bit analogue port member of this structure is used
- `IOCTRL_ANALOGUE_12_2` - the second 12 bit analogue port member of this structure is used
- `IOCTRL_ANALOGUE_12_3` - the third 12 bit analogue port member of this structure is used
- `IOCTRL_ANALOGUE_12_4` - the fourth 12 bit analogue port member of this structure is used
- `IOCTRL_DIGITAL_8_1` - the first 8 bit digital port member of this structure is used
- `IOCTRL_DIGITAL_8_2` - the second 8 bit digital port member of this structure is used
- `IOCTRL_DIGITAL_16_1` - the first 16 bit digital port member of this structure is used
- `IOCTRL_DIGITAL_16_2` - the second 16 bit digital port member of this structure is used
- `IOCTRL_DIGITAL_32` – a (first) 32 bit digital port member of this structure is used
- `IOCTRL_ANALOGUE_16_0` (outdated, previous constant name: `IOCTRL_ANALOGUE_16`) – the 16 bit analogue port member `analogue16_0` (outdated, previous variable name: `analogue16`) within this structure is used and contains valid data
- `IOCTRL_ANALOGUE_16_1` – the 16 bit analogue port member `analogue16_1` within this structure is used and contains valid data
- `IOCTRL_ANALOGUE_16_2` – the 16 bit analogue port member `analogue16_2` within this structure is used and contains valid data
- `IOCTRL_SERIAL_DATA` – the `serialData` member contains data with a maximum length of `IOCTRL_SERIAL_DATA_LENGTH` to be transmitted via serial port
- `IOCTRL_LASERPORT_8_1_BITS` - the first laserport member of this structure is used together with the first laserport8mask member for setting and clearing single bits, this flag can't be combined with `IOCTRL_LASERPORT_8_1`
- `IOCTRL_LASERPORT_8_2_BITS` - the second laserport member of this structure is used together with the second laserport8mask member for setting and clearing single bits, this flag can't be combined with `IOCTRL_LASERPORT_8_1`

All other members of this structure can hold the data according to the flags specified here. The valid range of these member depends on the given number of bits. As an example: the analogue 10 bit ports make use of the members `analogue10` which may have a value in range 0..1023.

2.4.2.8 Binary Data Structure *`OAPC_BIN_SUBTYPE_STRUCT_SCANHEADINFO`*

The structure `oapc_bin_struct_scanheadinfo` contains information about a connected scanhead, its axes, galvos and general state:

```
struct oapc_bin_struct_scanheadinfo
{
   unsigned char  ndHead;
   char           res1,res2,res3;
   unsigned int   errorFlags;
   unsigned short galvoTemp[3];
   unsigned short boardTemp[3];
   unsigned int   serialNumber[3],articleNumber[3];
   unsigned int   firmwareVersion[3];
   unsigned int   uAperture;
   unsigned int   pWavelength;
   unsigned int   operatingTime[3];
   char           cardType[128],headType[128];
};
```

`ndHead` – specifies the head number, this has nothing to do with multihead mode but with a possibly existing secondary head option (a mode where both heads mark the same data in parallel)

`errorFlags` – this member specify possible error states via following, or-concatenated flags:
`SCANHEAD_ERRORFLAG_VOLTAGE_X` – voltage error on X-axis
`SCANHEAD_ERRORFLAG_VOLTAGE_Y` – voltage error on Y-axis
`SCANHEAD_ERRORFLAG_VOLTAGE_Z` – voltage error on Z-axis
`SCANHEAD_ERRORFLAG_GALVOTEMP_X` – X galvo temperature error
`SCANHEAD_ERRORFLAG_GALVOTEMP_Y` – Y galvo temperature error
`SCANHEAD_ERRORFLAG_GALVOTEMP_Z` – Z galvo temperature error
`SCANHEAD_ERRORFLAG_POSITION_X` – X positioning error
`SCANHEAD_ERRORFLAG_POSITION_Y` – Y positioning error
`SCANHEAD_ERRORFLAG_POSITION_Z` – Z positioning error

`galvoTemp[3]` – temperature of each possible galvo in unit 1/10 degrees (0=X, 1=y, 2=Z), when there is no temperature information available for one galvo this value has to be set to 0

`boardTemp[3]` – board temperature of each possible galvo in unit 1/10 degrees (0=X, 1=y, 2=Z), when there is no board temperature information available for a galvo this value has to be set to 0

`serialNumber[3]` – serial numbers each possible galvo (0=X, 1=y, 2=Z), when there is no serial number information available for a galvo this value has to be set to 0

`articleNumber[3]` – article numbers each possible galvo (0=X, 1=y, 2=Z), when there is no article number information available for a galvo this value has to be set to 0

`firmwareVersion[3]` – board firmware version number for each possible galvo (0=X, 1=y, 2=Z), when there is no firmware version number available for a galvo this value has to be set to 0

`uAperture` – aperture size in unit micrometers

`pWavelength` – wavelength in unit picometers

`operatingTime[3]` – total operation time each possible galvo (0=X, 1=y, 2=Z), when there is no operating time information available for a galvo this value has to be set to 0

`cardType` - the textual name of a card, set to 0 if no name is available

`headType` - the textual type name of a head, set to 0 if no name is available

2.4.2.9 Binary Data Structure `OAPC_BIN_SUBTYPE_STRUCT_POS_CORR`

The structure `oapc_bin_struct_pos_corr` contains correction information for translation in X, Y and Z direction as well as rotation information for rotations around X, Y and Z axis. This structure can be used to correct and modify existing coordinate systems and vector data:

```
struct oapc_bin_struct_pos_corr
{
   int uPosX,uPosY,uPosZ;        // position offset / linear translation in X, Y
                                 // and Z direction in unit micrometers
   int mXAngle,mYAngle,mZAngle; // rotational correction around X, Y and Z axis
                                 // in milli-degrees
};
```

2.4.2.10 Binary Data Structure `OAPC_BIN_SUBTYPE_STRUCT_DYNGEOMSTART`

The structure oapc_bin_struct_dyn_data contains definitions and handling descriptions for dynamic data that can be created by a device dynamically (e.g. in stand-alone mode):

```
struct oapc_bin_struct_dyn_data
{
   unsigned int   UID;
   char           fmtString[DYN_DATA_MAX_STRING_LENGTH+1];
   unsigned int   type;
   unsigned int   flags,param1,param2,param3;
   unsigned int   uScaleX,uScaleY;
   int            snStartValue;
   int            snResetAt;
   unsigned short snIncrement,snBeatCount,snBeatOffset;
   unsigned char  snNumericBase,snMinDigits;
   int            timeOffset;
   int            res1,res2,res3,res4,res5,res6,res7,res8;
}
```

`UID` – unique identifier, used to modify it from outside e.g. on a stand-alone device

`fmtString` – string that contains the changeable text / the format string with some placeholders for automatically changeable parts of the string according to BeamConstructs format definition

`type` – specifies the type of element to be used (which font or which barcode type)

`flags`, `param1`, `param2`, `param3` – additional parameters belonging to creation of base element, their meaning and usage depends on the type of the element to be created

`uScaleX`, `uScaleY` – scaling factors in X and Y direction

`snStartValue` – value to start serial number counting at

`snResetAt` – value to reset serial number count to start value, does not apply when it is smaller than `snStartValue`

`snIncrement`, `snBeatCount`, `snBeatOffset` – counting definitions increment and beat

`snNumericBase`, `snMinDigits` – numeric base for displayed serial number

`timeOffset` – time value offset (in unit seconds)

`res1`, `res2`, `res3`, `res4`, `res5`, `res6`, `res7`, `res8` – reserved for later usage, have to be set to 0

2.5 Configuration XML Structure

Plug-ins that require additional data from the user can send a XML-structure to the main application. Within that structure a plug-in can define which parameters have to be displayed to the user, which data type has to be used for these parameters, within which valid range the user can enter new values for them and much more.

All these information are taken from the main application to display a nice configuration dialogue.

Following only the special tags are described that have to be used to define such a configuration XML successfully. Independent from that the XML itself need to be well-formatted and complete. For more information about the XML basics please refer to the related specifications.

The XML structure itself needs to be UTF-8 encoded, so it has to start with

```
<?xml version=\"1.0\" encoding=\"UTF-8\" standalone=\"yes\"?>
```

The ControlRoom-specific data have to be encapsulated within a `oapc-config` tag, so everything that is described within the following sections have to be located between a `<oapc-config>` and a `</oapc-config>` tag.

Several additional sub-tags then describe the layout of the configuration dialogue, the elements and their usage. These sub-tags are described in following sections.

2.5.1 The Symbol Image for the Flow Editor

All elements that can be put together and connected by data flows are displayed within the Flow Editor using a special symbol. A plug-in can provide such an image too so that it is displayed within the Flow Editor. That image has to be handed over within the XML structure Base-64-encoded between two tags `<flowimage>` and `</flowimage>`. Following format is necessary:

- size 106 x 50 pixels
- resolution 1..24 bit
- symbols for input and output connectors at defined positions (you can use an base image out of the SDK to create an own one)
- PNG format (binary format)
- Base-64 encoded

The Base-64 encoding can be done with several tools that are available for free or by using the online service at http://www.motobit.com/util/base64-decoder-encoder.asp. This operation creates a string out of the binary data of the image that can be used between the `flowimage`-tags directly.

PLEASE NOTE: the flow image has not to be put into XML structures that define custom configuration possibilities within the HMI editor, it is relevant for flow plug-ins or for the flow-part of an HMI plug-in only!

2.5.2 Dialogue Layout and Parameter Definitions

The parameters that belong to a definition of a configuration dialogue have to be placed between two special tags `<dialogue>` and `</dialogue>`. They define that everything between them belongs to the configuration dialogue that is displayed to a user. On next level a tab pane is defined where the user interface elements and the related values have to be defined. For every separate tab one separate definition of type `<general></general>`, `<dualpanel></dualpanel>`, `<stdpanel></stdpanel>`, `<parampanel></param>` or `<helppanel></helppanel>` has to be set.

Within such a tag that defines a tabbed pane the definitions for the user interface elements, the parameters, its default values, measurement units and default values have to be placed. These tags have to be located between `param`-tags and are described in the following section. So at this point the XML-structure can have the following format:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<oapc-config>
<flowimage>iVBORw0KGgoAAAANSUhEU...</flowimage>
<dialogue>
 <general>
  <param name="..." text="...">

   your parameter definitions here as specified below
  </param>
 </general>
</dialogue>
</oapc-config>
```

The `param`-tag can exist several times here, every appearance of it defines one additional parameter and its special appearance within the user interface. It consists of several additional tags, some of them are required, some of them optional. The following sections will describe which parameter tags will support which tags for what purpose.

2.5.3 Tab Pane Types

There are several types of tab panes are supported:

- The tags `<general></general>` are used to define a new tab pane. All user interface elements specified by the `<param></param>`-tags are located from top to bottom directly below of each other. Within such a general tab pane the name input field and – if valid for this type of plug-in – the cycle time input field is linked into by the main application automatically. If a plug-in does not define a general tab pane it is added by the main application automatically containing at least an input field for the name of the plug-in. For a plug-in not more than one general tab pane can exist.
- When the tag `<dualpanel></dualpanel>` is used, the user interface elements are placed in pairs. That results in a layout like it is known from the element configuration dialogue of the HMI Editor. This layout with pairs of elements is intended to be used for state-dependent configurations that are triggered by the digital input 0. Because of that the two rows of elements get a header automatically. This tag supports the attribute "text" which expects a string value that defines the name of the panel that is shown as the title of the tab-pane.
- The tags `<stdpanel></stdpanel>` define a panel similar to the general panel but there the application does not link any additional input fields into. Beside of that a standard panel can exist more often than only one time. This tag supports the attribute "text" which expects a string value that defines the name of the panel that is shown as the title of the tab-pane.
- The tags `<parampanel></parampanel>` enclose some special definitions of fields that give the possibility to define parameter numbers, parameter values and the possibility to read the current parameters value out of the device by pressing a "Read Value" button that is assigned to these input fields. This tag supports the attribute "text" which expects a string value that defines the name of the panel that is shown as the title of the tab-pane.
- Using the tags `<helppanel></helppanel>` a special tabbed pane can be defined that can't be used for parameter input but for plain texts describing the input and output connectors and their behaviour of a plug-in; for a description of the help-panel-tags please refer below

2.5.4 Input Fields

Following an overview is given about all elements that can be put into a tab pane. Within the XML hierarchy these elements are located within tab pane tags so that these input elements can be added to these panes.

These fields are always encapsulated by a tag `<param></param>` which itself contains two mandatory attributes:

- `name` – the name of the parameter; this name can be chosen freely and is used when the parameters value is returned using function `oapc_set_config_data(`
- `text` – the text that is displayed in front of the input field, here a short name has to be given that tells the user which kind of data have to be entered here; this text can be chosen freely

2.5.4.1 *Text Input Fields*

A text input field is a single line field that expects data of type "string" that can be entered by the user freely. It accepts following tags:

- `<type>string</type>` - the type tag "string" specifies the type of the input element
- `<default> … </default>` - within these tags a default value has to be specified; this value is used as predefined value within the text input field;
  PLEASE NOTE: when a plug-in is called for the first time this tag has to be used to hand over a default value, elsewhere the stored value has to be displayed here so that data changed by the user are used here when the configuration dialogue of this plug-in is called repeatedly
- `<min> … </min>` - this tag requires a number that specifies the minimum number of characters that have to be entered by the user
- `<max> … </max>` - this tag expects a number that is used to limit the users input to that number of characters
- `<unit> … </unit>` - this is an optional tag, if it exists it appends a short text behind the input field to give a measurement unit; here a short text (1..5 characters) can be chosen freely
- `<state>disabled</state>` - this is an optional tag, when it is found the user interface element is disabled so that no changes can be made
- `<enableon> … </enableon>` - this tag can be used to enable/disable the field; between the tags the name of a checkbox has to be given, when it is set, the own element is enabled, when it is unchecked, it is disabled; using this way dependencies can be created where elements are enabled only in case a specific option is turned on via a checkbox
- `<disableon> … </disableon>` - this tag can be used to disable/enable the field; between the tags the name of a checkbox has to be given, when it is set, the own element is disabled, when it is unchecked, it is enabled; using this way dependencies can be created where elements are disabled only in case a specific option is turned on via a checkbox

2.5.4.2 *Integer Input Fields*

A integer input field is a single line field that expects non-floating point numbers that can be entered by the user freely. It accepts following tags:

- `<type>integer</type>` - the type tag "integer" specifies the type of the input element
- `<default> … </default>` - within these tags a default value has to be specified; this value is used as predefined value within this number input field;
  PLEASE NOTE: when a plug-in is called for the first time this tag has to be used to hand over a default value, elsewhere the stored value has to be displayed here so that data changed by the user are displayed here when the configuration dialogue of this plug-in is called repeatedly
- `<accuracy> … </accuracy>` - using this tag a integer value can be given, that specifies the shown accuracy of a floating point number. Here the given value should be bigger than 0 and specifies the number of post decimal positions to be shown in UI for this floating point type. As an example: when a value of 2 is given, a value of "1" will always be shown as "1.00"

- `<min> … </min>` - this tag requires a number that specifies the minimum value the entered number is allowed to have
- `<max> … </max>` - this tag expects a number that is used to specify the maximum number a user is allowed to enter
- `<unit> … </unit>` - this is an optional tag, if it exists it appends a short text behind the input field to give a measurement unit; here a short text (1..5 characters) can be chosen freely
- `<state>disabled</state>` - this is an optional tag, when it is found the user interface element is disabled so that no changes can be made
- `<enableon> … </enableon>` - this tag can be used to enable/disable the field; between the tags the name of a checkbox has to be given, when it is set, the own element is enabled, when it is unchecked, it is disabled; using this way dependencies can be created where elements are enabled only in case a specific option is turned on via a checkbox
- `<disableon> … </disableon>` - this tag can be used to disable/enable the field; between the tags the name of a checkbox has to be given, when it is set, the own element is disabled, when it is unchecked, it is enabled; using this way dependencies can be created where elements are disabled only in case a specific option is turned on via a checkbox

2.5.4.3 Floating Point Number Input Fields

A floating point number input field is a single line field where floating point numbers can be entered by the user freely. Here following tags are accepted:

- `<type>float</type>` - the type tag "float" specifies the type of the input element
- `<default> … </default>` - within these tags a default value has to be specified; this value is used as predefined value within this number input field;
  PLEASE NOTE: when a plug-in is called for the first time this tag has to be used to hand over a default value, elsewhere the stored value has to be displayed here so that data changed by the user are displayed here when the configuration dialogue of this plug-in is called repeatedly
- `<min> … </min>` - this tag requires a number that specifies the minimum value the entered number is allowed to have
- `<max> … </max>` - this tag expects a number that is used to specify the maximum number a user is allowed to enter
- `<unit> … </unit>` - this is an optional tag, if it exists it appends a short text behind the input field to give a measurement unit; here a short text (1..5 characters) can be chosen freely
- `<state>disabled</state>` - this is an optional tag, when it is found the user interface element is disabled so that no changes can be made
- `<enableon> … </enableon>` - this tag can be used to enable/disable the field; between the tags the name of a checkbox has to be given, when it is set, the own element is enabled, when it is unchecked, it is disabled; using this way dependencies can be created where elements are enabled only in case a specific option is turned on via a checkbox

- `<disableon> … </disableon>` - this tag can be used to disable/enable the field; between the tags the name of a checkbox has to be given, when it is set, the own element is disabled, when it is unchecked, it is enabled; using this way dependencies can be created where elements are disabled only in case a specific option is turned on via a checkbox

2.5.4.4 Combo Box Fields

A combo box consists of a list of predefined values where the user can choose from. Here no parameters and values can be used by the user that are not predefined, only one out of a list of possible values can be selected. The value parameters that are accepted and returned when such an element is used, are the 1-based index number of the chosen element, so the order of the data to chose from is important.

Here following tags are accepted:

- `<type>option</type>` - the type tag “option” specifies the type of the input element
- `<value> … </value>` - this tag can exist several times, it specifies the elements in the list to choose from; here every tag defines one entry, the order of appearance of the tags is equal to the order of appearance within the combo box
- `<unit> … </unit>` - this is an optional tag, if it exists it appends a short text behind the input field to give a measurement unit; here a short text (1..5 characters) can be chosen freely
- `<state>disabled</state>` - this is an optional tag, when it is found the user interface element is disabled so that no changes can be made
- `<default> … </default>` - within these tags a default value has to be specified; this value is used to preselect one element of the combo box
- `<enableon> … </enableon>` - this tag can be used to enable/disable the field; between the tags the name of a checkbox has to be given, when it is set, the own element is enabled, when it is unchecked, it is disabled; using this way dependencies can be created where elements are enabled only in case a specific option is turned on via a checkbox
- `<disableon> … </disableon>` - this tag can be used to disable/enable the field; between the tags the name of a checkbox has to be given, when it is set, the own element is disabled, when it is unchecked, it is enabled; using this way dependencies can be created where elements are disabled only in case a specific option is turned on via a checkbox

2.5.4.5 Colour Chooser Button

This type adds a button to the panel that opens a colour chooser dialogue when it is pressed. The button itself uses the chosen colour.

Here following tags are accepted:

- `<type>colourbutton</type>` - the type tag “option” specifies the type of the input element
- `<value> … </value>` - this tag specifies the default colour of the button in format `0xBBGGRR`
- `<enableon> … </enableon>` - this tag can be used to enable/disable the field; between the tags the name of a checkbox has to be given, when it is set, the own element is enabled, when it is unchecked, it is disabled; using this way dependencies can be created where elements are enabled only in case a specific option is turned on via a checkbox
- `<disableon> … </disableon>` - this tag can be used to disable/enable the field; between the tags the name of a checkbox has to be given, when it is set, the own element is disabled, when it is unchecked, it is enabled; using this way dependencies can be created where elements are disabled only in case a specific option is turned on via a checkbox

2.5.4.6 Font Chooser Button

Using this tag a button can be defined for the configuration panel the user can use to choose a font. This tag type is more specific than the other ones: while for a standard element exactly one parameter value is given back to the plug-in a font definition consists of different parameters. Therefore for every font chooser button element several parameters are given back: point size, style, weight and face name. They are given back during separate calls of `oapc_set_config_data()`. Here the name that identifies the parameter consists

of the name defined within the XML-structure via <name></name>tags, an underscore and one of the additional identifiers “pointsize” (32 bit unsigned integer for the size of the font), “style” (32 bit unsigned integer for the character style), “weight” (32 bit unsigned integer for the thickness of the font) and “facename” (string for the name of the font).
Following tags can be used here:

- `<type>fontbutton</type>` - this tag defines the font selector button: a simple button with a bigger size that shows the currently selected font using a fixed text that can't be changed
- `<pointsize>...</pointsize>` - the size of the font in unit points
- `<style>...</style>` - the style of the font, here implementation dependent flags are used that should be set by the main application only; this value should be initialized with 0 when a font definition is set to a default value
- `<weight>...</weight>` - the weight of the font, here implementation dependent values are used that should be set by the main application only; this value should be initialized with 0 when a font definition is set to a default value
- `<face>...</face>` - the name of the font; this name may be platform-dependent, when the specified font doesn't exists on the target platform the main application automatically chooses a similar one
- `<enableon> … </enableon>` - this tag can be used to enable/disable the field; between the tags the name of a checkbox has to be given, when it is set, the own element is enabled, when it is unchecked, it is disabled; using this way dependencies can be created where elements are enabled only in case a specific option is turned on via a checkbox
- `<disableon> … </disableon>` - this tag can be used to disable/enable the field; between the tags the name of a checkbox has to be given, when it is set, the own element is disabled, when it is unchecked, it is enabled; using this way dependencies can be created where elements are disabled only in case a specific option is turned on via a checkbox

2.5.4.7 Check Box

This type adds a check box that supports two selection states “checked” and “unchecked”.

Here following tags are accepted:

- `<type>checkbox</type>` - the type tag “checkbox” specifies the type of the input element
- `<default> … </default>` - this tag specifies the default value of the check box, here the values 0 and 1 are allowed
- `<enableon> … </enableon>` - this tag can be used to enable/disable the field; between the tags the name of a checkbox has to be given, when it is set, the own element is enabled, when it is unchecked, it is disabled; using this way dependencies can be created where elements are enabled only in case a specific option is turned on via a checkbox
- `<disableon> … </disableon>` - this tag can be used to disable/enable the field; between the tags the name of a checkbox has to be given, when it is set, the own element is disabled, when it is unchecked, it is enabled; using this way dependencies can be created where elements are disabled only in case a specific option is turned on via a checkbox

2.5.4.8 File Load Selection

This type adds an input field for a file to load and a button that opens a file open dialogue when it is pressed.

Here following tags are accepted:

- `<type>fileload</type>` - the type tag “fileload” specifies the type of the input element

- `<default> … </default>` - this tag specifies a default file that can be used for loading; here a string is expected that meets the requirements of the underlying platforms file and path name syntax
- `<ffilter> .. </ffilter>` - file filter definitions for the file types that can be loaded; here different file descriptions and extensions can be defined in format "`description|*.extension|..`". Here e.g. "`Text File|*.txt;*.text|All Files|*`" gives the user the possibility to choose between two predefined file types: text files with extension .txt or .txt and all other files independent from their extension
- `<enableon> … </enableon>` - this tag can be used to enable/disable the field; between the tags the name of a checkbox has to be given, when it is set, the own element is enabled, when it is unchecked, it is disabled; using this way dependencies can be created where elements are enabled only in case a specific option is turned on via a checkbox
- `<disableon> … </disableon>` - this tag can be used to disable/enable the field; between the tags the name of a checkbox has to be given, when it is set, the own element is disabled, when it is unchecked, it is enabled; using this way dependencies can be created where elements are disabled only in case a specific option is turned on via a checkbox

2.5.4.9 File Save Selection

This type adds an input field for a file to be saved and a button that opens a file save dialogue when it is pressed.

Here following tags are accepted:

- `<type>filesave</type>` - the type tag "filesave" specifies the type of the input element
- `<default> … </default>` - this tag specifies a default file that can be used for saving; here a string is expected that meets the requirements of the underlying platforms file and path name syntax
- `<ffilter> .. </ffilter>` - file filter definitions for the file types that can be saved; here different file descriptions and extensions can be defined in format "description|*.extension|..". Here e.g. "`Text File|*.txt;*.text|All Files|*`" gives the user the possibility to choose between two predefined file types: text files with extension .txt or .txt and all other files independent from their extension
- `<enableon> … </enableon>` - this tag can be used to enable/disable the field; between the tags the name of a checkbox has to be given, when it is set, the own element is enabled, when it is unchecked, it is disabled; using this way dependencies can be created where elements are enabled only in case a specific option is turned on via a checkbox
- `<disableon> … </disableon>` - this tag can be used to disable/enable the field; between the tags the name of a checkbox has to be given, when it is set, the own element is disabled, when it is unchecked, it is enabled; using this way dependencies can be created where elements are disabled only in case a specific option is turned on via a checkbox

2.5.4.10 Select Directory

This type adds an input field for a path to a directory and a button that opens a directory selection dialogue when it is pressed by the user.

Here following tags are accepted:

- `<type>dirselect</type>` - the type tag "dirselect" specifies the type of the input element
- `<default> … </default>` - this tag specifies a default directory path that can be used for loading; here a string is expected that meets the requirements of the underlying platforms path name syntax

- `<enableon> … </enableon>` - this tag can be used to enable/disable the field; between the tags the name of a checkbox has to be given, when it is set, the own element is enabled, when it is unchecked, it is disabled; using this way dependencies can be created where elements are enabled only in case a specific option is turned on via a checkbox
- `<disableon> … </disableon>` - this tag can be used to disable/enable the field; between the tags the name of a checkbox has to be given, when it is set, the own element is disabled, when it is unchecked, it is enabled; using this way dependencies can be created where elements are disabled only in case a specific option is turned on via a checkbox

2.5.5 Param-Panel Fields

Within a parameter panel only one type of sub-tags is allowed that specifies the combination of parameter number input field, parameter value input field, their allowed ranges and default values and implicitly defines the button to read the value of such a parameter number. Every of these parameter definition sets is encapsulated within a tag that may look like this:

```
<param text="Custom Parameter 1">
   <paramcol name="custparam1" type="integer" default="10" min="0" max="255" />
   <valuecol name="custvalue1" type="integer" default="42" min="-32767"
max="32767" />
</param>
```

The attribute “text” in main tag `<param>` specifies the name of the parameter that is shown in first column of the panel. The enclosed tag `<paramcol>` specifies the input field for the column that shows the parameters number and `<valuecol>` the input field for the column that contains the input field for the value that is assigned to this parameter. The button to read out the current parameters value is placed in fourth column within the panels layout. Both inner tags support the following attributes that specify their behaviour:

- `<name>` – the internal name as it is used to return the value back to the plug-in by using function `oapc_set_config_data()`
- `<type>` – the type of the input field, here “integer” and “float” are allowed
- `<default>` – the default value to be shown within the input field
- `<min>` and `<max>` – the minimum and maximum values the parameter number or parameter value may have, when the user enters a value outside of this range the application limits them automatically.
- `<enableon> … </enableon>` - this tag can be used to enable/disable the field; between the tags the name of a checkbox has to be given, when it is set, the own element is enabled, when it is unchecked, it is disabled; using this way dependencies can be created where elements are enabled only in case a specific option is turned on via a checkbox
- `<disableon> … </disableon>` - this tag can be used to disable/enable the field; between the tags the name of a checkbox has to be given, when it is set, the own element is disabled, when it is unchecked, it is enabled; using this way dependencies can be created where elements are disabled only in case a specific option is turned on via a checkbox

2.5.6 Help-Panel Fields

Following the special tags of a help-panel are described. These special tags are encapsulated using the help-panel definition `<helppanel></helppanel>` and do not offer user interaction. They are used to describe the functionality of input and output connectors of a plug-in.

2.5.6.1 *Input Connection Tags*

Input connectors are defined using the tags `<inx></inx>` where x is a number in range 0..7 and stands for the input number that is described here. A short, descriptive text has to be set between these tags to be displayed within the help-panel. So as an example a tag line

```
<in0>Clock input for data acquisition</in0>
```

would result in a help-panel that contains a line for input “IN0”, its data type and the description “Clock input for data acquisition” for that input.

2.5.6.2 *Output Connection Tags*

Output connectors are defined using the tags `<outx></outx>` where x is a number in range 0..7 and stands for the output number that is described here. A short, descriptive text has to be set between these tags to be displayed within the help-panel. So as an example a tag line

```
<out7>Retrieved text data</out0>
```

would result in a help-panel that contains a line for output “OUT0”, its data type and the description “Retrieved text data” for that output.

2.5.7 Example

Following a short example is given how a valid XML structure can look like. Some more examples can be found within the sources of the different plug-ins that are available within the SDK.

```
<?xml version=\"1.0\" encoding=\"UTF-8\" standalone=\"yes\"?>
<oapc-config>
 <flowimage>iVBORw0KGgoAAAANSUhEUgAAAGoAAAAyBAMAA...</flowimage>
 <dialogue>
  <general>
   <param name=”ip” text=”IP”>
    <type>string</type>
    <default>192.168.1.1</default>
    <min>7</min>
    <max>16</max>
   </param>
   <param name=”port” text=”Port”>
    <type>integer</type>
    <default>1900</default>
    <min>1</min>
    <max>65535</max>
   </param>
   <param name=”mode” text=”Mode”>
    <type>option</type>
    <value>Use mode of transmitter</value>
    <state>disabled</state>
    <default>1</default>
   </param>
  </general>
 </dialogue>
 <helppanel>
  <in0>Transmit text data</in0>
  <out0>Received text data</out0>
 </helppanel>
</oapc-config>
```

This is a very simple example that uses three different kinds of configuration parameters. First of all the head of the XML-structure is given, then the main tag for the ControlRoom configuration is set that encapsulates all that data that are watched by the main software. So if any tags are located outside `<oapc-config>` and `</oapc-config>` they will be ignored by the main application.

Next the image is handed over, here within the example the Base-64-encoded data are not given fully.

Afterwards the definitions of the configuration dialogue and the general tab pane are set. Here three configuration fields are used, one to enter an IP number as a string, one to enter a port number as an integer and one to choose a mode out of a list of modes. The last element consists only of one possibility to choose from, therefore it is disabled.

To specify an IP the user has to enter a minimum of 7 characters and a maximum of 16, otherwise the main software would not accept the entered value. Here as default the IP 192.168.1.1 is given. If the IP is changed by the user and this dialogue is opened for a second time the default-value of the IP of course can't be 192.168.1.1 any longer, now the plug-in would have to send the changed value.

The integer input field for the port number works similar, it defines a default number of 1900 and specifies the valid range 1..65535. If the user would enter a larger or smaller value the main application would not accept this value due to the allowed range specified by `<min></min>` and `<max></max>` tags.

The last field is a special one, here the `<value>`-tag defines one single option to choose from. After there is only one pair of this kind of tag the combo box has no options where somebody really could choose from. Therefore it is disabled and the only available option is set as default value.

Next a help-panel is defined. It contains a description of the input and output connections of this plug-in, here a short description is given for input and output number 0 (the first one).

2.6 Developing own plug-ins

The SDK provides several plug-ins that are distributed within the standard software package. They are provided in source code and can be used as starting point for the development of own plug-ins or for further modifications and for extending their functionality. To start with the programming of external plug-ins it is recommended to use the following examples:

- **libio_parallel_inout** – Flow plug-in that accesses the parallel port
- **libio_clock** – hardware-independent Flow plug-in that provides time information

Both are very simple in its structure and therefore easy to understand. After their working principle is clear the step over to the more complex HMI plug-ins can be made, here example Plug_ins can be found in sub-folder **libio_hmi_***.

3 Localisation

3.1 Working Principle

All OpenAPC software package components use a translation and localisation concept that is very simple and bases on plain text files that can be edited easily. The selection of the correct language file is done by the application automatically, here the default language of the underlying operation system is used. So no user interaction or configuration is required.

3.2 Choosing the Correct Translation File Name

First of all the names of the files that contain the translations have to be chosen. The base file names are

- **openplayer_xx_YY.property, openplayer_xx.property** – for the texts that are used within the OpenPlayer, OpenEditor and OpenDebugger
- **openapc_xx_YY.property** or **openapc_xx.property** – for the texts that are used within OpenEditor and OpenDebugger
- **construct_xx_YY.property** or **construct_xx.property** – for the texts that are used within CNConstruct
- **common_xx_YY.property** or **common_xx.property** – general texts that are used by all applications
- **custom_xx_YY.property** or **custom_xx.property** – an additional translations-file that can be used for own translations (e.g. texts that are defined in user interface elements); this file does not exist by default and therefore will not be overwritten during installation of software updates

Here “xx” and “YY” specify the language: “xx” is the general language (according to ISO 639-1, compare to http://en.wikipedia.org/wiki/List_of_ISO_639-1_codes) and “YY” describes the country code (according to ISO 3166, compare to http://www.iso.org/iso/english_country_names_and_code_elements).

An example: if one of the OpenAPC applications is started on an English system that is located in United Kingdom, it tries to load the translation files **openplayer_en_GB.property** and **openapc_en_GB.property**. If they can't be found, within a next step it tries to find general English language files **openplayer_en.property** and **openapc_en.property**. If that fails too it falls back to the internal default language.

So you are open to use both names with the language code and the country code or only with the language code. In latter case you cover more language variants but you are not able to take care about regional and country-related pecularities.

The files itself have to be deployed within the sub-folder “translations” of the applications installation location. In case a file can be found there at start up, it is used by the applications automatically.

3.3 Translating the Applications

The contents of a language file are quite simple: it is a plain text file in UTF-8 encoding that contains pairs of values that are separated by “ = “ (space, equal-sign, space). On the left side of these separators the original, default texts can be found, they can't be edited and never should be changed. On the right side you can place your translation. So following rules apply to such a language file in order to create a valid translation for a language:

- the file has to be stored with UTF-8 encoding
- the original texts on the left side never have to be changed
- the delimiter between original and translated text has to be a combination “space, equal sign, space” (“ = “)
- translations can't be done using different lines, they have to be placed in one line directly after the original text and the delimiter characters
- you can add a “`\n`” to get a line break wherever the original text makes use of such a line break too (Example: a text “`this is\nthe second line`” would result in an output

  ```
  this is
  the second line
  ```

only when the original text also makes use of the "\n" symbol)

An example to make it more clear:

```
My original text = Mi texto original
This is my\noriginal text = Este es mi\ntexto original
```

Here the original texts stay on the left side, within the second line the special "\n" symbol is used. That means, that also the translation can make use of a line break here. In both cases the text on the left side is specified by the application and has not to be modified. On the right side the translated text is placed (this example uses a language where we think it could be Spanish). So for such an example the language file could have e.g. the name openapc_es.property.

After the original texts are fixed and defined by the contents of the application you have to use an existing translation or the translation file skeleton that comes with the SDK. Copy that file, set the correct name for it and extend its contents by your translation – that's all!

3.4 Creating own Translations

The SDK provides some files openapc_xxx.property, openplayer_xxx.property, construct_xxx.property and common_xxx.property that can be used as starting point for own translations. Here all the used (default) English texts are contained as well as their counterparts in a language that is specified by the "xxx" part of the name. For translating them the localised texts have to be replaced by the ones of the new language and the files have to be renamed according to the description above.

4 ControlRoom Interface

The interface described here can be used to connect ControlRoom to other applications. This connection can be used to remote control these other applications, to remote control the ControlRoom player, to transfer data and information or for other purposes.

The interface described here is designed dynamically on ControlRoom side. So the end user is free to implement an own interface structure to connect to other applications. Nevertheless it is recommended to use the structure specified here for interfacing between applications due to the following reasons:

- the standardized interfacing mechanism is fully tested
- it will be further developed without efforts on your side
- standardised solutions are widely supported
- interface implementations for external applications are provided with the SDK and can be used freely according to the OpenAPC-Dual-License

4.1 Overview

Currently the interface to other applications is done via TCP/IP only. Here ControlRoom player acts as client (using the “Network Client”-plug-in), the application to connect to has to provide a server socket. The communication between both sides is done asynchronously: once ControlRoom has connected to the other application both sides can send data, there is no strict handshake specified. Depending on what has to be done with this interface it could be necessary to implement such a handshake where one side sends a request and the other side answers to it but it is not required generally.

The data that are transferred between both sides are organized in pairs consisting of a command (which is a plain, unique text identifier) and the data (here the same data types are used like within the application: digital, numeric, characters and binary data).

On ControlRoom side these pairs can be generated and decoded using the converters for CMD-Pairs (please refer to the manual, description of flow elements / converters). On side of the interfaced application these pairs are generated and decoded by the interface.

4.2 Data Flow

The data flow between the interfaced applications is as flexible as the whole ControlRoom package: the purpose and sense of the commands and their related data are not predefined, they have to be used depending on what has to be done with the interface.

As an example: an external application has to be remote-controlled. Within the ControlRoom HMI a button is provided that has to control the remote application. Beside of that a static line is provided that can be toggled between red and green background colour depending on its digital IN0. This static line has to act as feedback for the result that is sent from the applications.

The related ControlRoom flow structure now looks as follows: the output of the button is sent to a “Digital to CMD/Value-Pair” flow element. Here the digital value is assigned to a unique command name, e.g. “OAPC_SWITCH_ON”. Afterwards this pair of data (a character and a digital data line) are connected with the “Network Client” that sends them to the external application. This network client plug-in has to operate in mode “Command/Value - transmit only pairs” to support the CMD/Value-pairs.

As soon as the external application receives this command “OAPC_SWITCH_ON” together with a value “1” it reacts on this value and sends a feedback: here a command/value pair is emitted, the application sends a command “APP_SWITCHED_ON” together with a digital “1”.

This pair of data is received at the same “Network Client”. Here the received data are converted using a “CMD/Value-Pair to Digital” converter. Here exactly one of its digital outputs is specified to react on the command “APP_SWITCHED_ON” and to output the related digital signal. This digital output is connected to IN0 of the line – so depending on the reaction of the external application this static line will change its behaviour.

The principle described here can be used to implement complex dependencies with external applications. For such applications one thing is important: the command names that are used together with specific data have to be unique so that the transferred data can be assigned non-ambiguous.

It is recommended to specify command names that start with the name of the application that emits the CMD/value pair. So for an ControlRoom project with the name “myProject” a prefix “CRMYPRJ_” could be used.

4.3 Example Applications

All implementations that are described within the following sections provide an example application that demonstrates their behaviour and usage. This example can be used together with the ControlRoom project file “Example1.apcp”.

The example project contains a toggle button to emit a digital value, a number field and a text field for numerical and character data and a simple button to send an exit command to the remote application. These elements send data to the application, the application responds to them and the response is displayed using a rectangle, and write-protected number and text fields.

The example application itself checks for commands named ECHO_DIGI, ECHO_NUM and ECHO_STR to send back the related data using the commands REPLY_DIGI, REPLY_NUM and REPLY_STR that are handled by the example project. Beside of that the controlled application reacts on an command EXIT when the related digital data is “1”: it terminates itself in this case. So this example demonstrates nothing more than sending data to the application and echoing them back to the project – but that is the functionality that is necessary for remote-controlling or communicating with external applications.

4.4 Interface Implementations

The SDK provides different interface implementations that can be used directly within own applications. Following these implementations are described in detail.

All provided interfaces can be found within the SDK's sub-folder “**interface**”.

4.4.1 Java ControlRoom Interface

The Java implementation of the ControlRoom interface uses the concept of a listener. This listener provides methods that are called whenever data are received. So here an own class has to implement this listener and provide these methods to receive the data easily.

Following classes are provided in package **com.oapc.iserver** to be used by an application:

- **OAPCInterface** – the interface itself that manages the client connections, data reception and transmission from and to them
- **OAPCListener** – the listener interface that has to be implemented in an own class to get informed about data received from ControlRoom OpenPlayer
- **SocketEntry** – an implementation-internal class that is handed over to the application in some cases and that can be used to identify a specific data connection and to transmit data to one connected OpenPlayer exclusively

To use the Java ControlRoom interface following steps are necessary:

- implement an own (inner) class that implements OAPCListener and create an object of this class
- create an object of type OAPCInterface and add the precedent created listener to this one
- react on data transmitted from the OpenPlayer dependent on their purpose
- transmit data to all connected OpenPlayers depending on the purpose of the application

4.4.1.1 The Interface OAPCListener

This interface has to be implemented in an own class and has to provide several methods that are called from the OAPCInterface object where this listener is registered at:

`public void commandReceived(String cmd,String value,SocketEntry socket)`

This method is called whenever a character value was received, `cmd` contains the (unique) command identifier, `value` the transmitted text data and `socket` identifies the connection where these data are received from; here `socket` can be used to submit an answer to this client immediately.

`public void commandReceived(String cmd,float value,SocketEntry socket)`

This method is called whenever a numerical value was received, `cmd` contains the (unique) command identifier, `value` the transmitted number and `socket` identifies the connection where these data are received from; here `socket` can be used to submit an answer to this client immediately.

`public void commandReceived(String cmd,boolean value,SocketEntry socket)`

This method is called whenever a digital value was received, `cmd` contains the (unique) command identifier, `value` the transmitted data (state `true` or `false`) and `socket` identifies the connection where these data are received from; here `socket` can be used to submit an answer to this client immediately.

`public void stateChanged(int state,String message,SocketEntry socket)`

This method informs about changes at the interface. Parameter `message` is a short, descriptive text that specifies these changes in a human-readable way and `socket` is the client connection where that change happened. Please note: depending on the type of state change `socket` can be `null`, e.g. in cases where the client got lost and data can't be transmitted any longer to this client. The first parameter `state` specifies which kind of state change happened, here following constants are used:

- `STATE_NEW_CONNECTION` – a new connection to the interface was established successfully
- `STATE_CONNECTION_CLOSED` – a connection was closed so that it is no longer available for communication
- `STATE_CONNECTION_ERROR` – a connection was interrupted without being closed normally so that it is no longer available for communication
- `STATE_DATA_RECEIVED` – new data have been received from an OpenPlayer, this state is followed by a call to method `commandReceived()`

4.4.1.2 The Class OAPCInterface

This class provides the general communication functionality:

`public OAPCInterface()`

This is the default constructor that creates a server socket using default port 1810 to connect to from an OpenPlayers project.

`public OAPCInterface(InetAddress bindAddr)`

Using this constructor a new object of this type can be created that establishes a server socket that binds to the default port and the address specified by `bindAddr`.

`public OAPCInterface(int port, InetAddress bindAddr)`

Using this constructor a new interface object can be created using a custom `port` number and the specified address `bindAddr` to bind this server socket to. When `bindAddr` is set to null the default address 0.0.0.0 is used.

`public void setAuthentication(String uname,String pwd)`

When the network client plug-in used within the ControlRoom project is configured to use authentication, this method has to be called in order to set the correct user name `uname` and password `pwd`. Whenever a connection is established that uses no or different authentication data, the related connection is rejected and no communication is possible between both sides.

`public void close()`

This method has to be called when the interface connection is no longer required: it closes the connections to all clients and removes the server socket. After this method has been called no more data are received and sending of data is no longer possible.

`public void addOAPCListener(OAPCListener listener)`

Using this method a new `listener` class can be registered at the interface object. This `listener` then will be informed about every data received and about every connection state change.

`public void removeOAPCListener(OAPCListener listener)`

This class removes an already registered `listener` so that it is no longer informed about state changes and data receptions.

`public void sendCommand(String cmd,String value,SocketEntry socket)`

With this method a command/value pair can be transmitted. Here `cmd` is the (unique) command identifier and `value` is the string that is assigned to this command. When `socket` is set to `null` these data are sent to all connected clients, when a valid object of type `SocketEntry` is specified instead of this only the single OpenPlayer connection will receive these data that is assigned to the given `SocketEntry`.

`public void sendCommand(String cmd,float value,SocketEntry socket)`

With this method a command/value pair can be transmitted. Here `cmd` is the (unique) command identifier and `value` is the number that is assigned to this command. When `socket` is set to `null` these data are sent to all connected clients, if a valid object of type `SocketEntry` is specified instead of this only the single OpenPlayer connection will receive these data that is assigned to this `SocketEntry`.

`public void sendCommand(String cmd,boolean value,SocketEntry socket)`

Using this method a command/value pair can be transmitted. Here `cmd` is the (unique) command identifier and `value` is the digital value (`true` or `false`) that is assigned to this command. When `socket` is set to `null` these data are sent to all connected clients, if a valid object of type `SocketEntry` is specified instead of this only the single OpenPlayer connection will receive these data that is assigned to this `SocketEntry`.

4.4.1.3 The Class SocketEntry

This class is used internally only, it doesn't needs to be created by the application. Therefore it does not provide any methods or members that could be used directly.

4.4.2 C/C++ and other languages ControlRoom Interface

The C/C++ interface to ControlRoom does not make use of a special implementation, it uses the `oapc_iface_`-functions of liboapc as described below. This is a general shared library interface. The headerfile “liboapc.h” can be used within C/C++ applications, the shared library interface itself is not limited to this language.

For a more detailed description please refer the section below, an example C application that uses this interface can be found within the SDK in subfolder “**interface**”.

5 Interlock Server Connection

The Interlock Server is a separate, fast application that can be used to reflect the state of the visualisation (which is running in OpenPlayer), to influence its state, to implement automatic sequences, to create interlock dependencies and other things more.

Basically the Interlock Server is a TCP/IP server application that does nothing more than store data that are sent to it and to inform all other connected applications when new data have been added or existing data have been changed.

This section describes how it is possible to connect to this server from outside using own applications. For a detailed description of configuration, set-up and possible usage scenarios of this server please refer to the main application manual.

5.1 Data Flow

The data management and flow that is done by the Interlock Server is quite simple. Dependent on the programming language there are small differences in implementation but the basic usage principle is always the same:

- an application connects to the TCP/IP server socket of the Interlock Server
- now the application is able to send data to the server, these data are stored within the Interlock Server, additionally the server submits these new or changed data to all other connected clients (including the OpenPlayer)
- whenever other applications change data within the Interlock Server, the connected client application is informed automatically, dependent on the name of the data now the application has to decide if it will use these information or not
- as an additional feature a connected application is able to retrieve the data of a special data element using its name or to retrieve all data (these data can be identified by their name too)

So whenever an external application changes a value that is connected to an element within the OpenPlayer, this modification causes a response within it. Thus the visualisation can be manipulated out of such an application. And vice versa the same is true: when a user or the data flow within the OpenPlayer changes a HMI or a flow element that is mapped to the Interlock Server, all connected applications are informed about this modification and therefore can react on such user input.

5.2 Example Applications

There is an example project “**iserver_example1.apcp**” available in SDK within directory “**iserver**”. This is an demonstration project that requires the Interlock Server and uses it: it maps several internal user interface elements to the server.

Additional there are several example scripts available in Java, Lua, IL (Instruction List) and C sources. When these scripts/applications are used, the user input is managed by them and the user elements shown within the OpenPlayer are manipulated by them. That is done via the data mapped in Interlock Server.

The example project consists of a “GO” toggle button and a strip of boxes that can be highlighted each. When the toggle button is pressed the related information is sent to the Interlock Server. There the running example application (or script, depending on which of the implementations is used) watches the related data element. As long as the “GO” button is pressed and the related data element signals HIGH, the application turns on and off the boxes – again via their representation within the Interlock Server – to offer a small light show.

5.3 Interface Implementations

The SDK provides different interface implementations to access the Interlock Server which can be used directly within own applications. Following these implementations are described in detail.

All provided interfaces can be found within the SDKs subfolder “**iserver**” (except the Java interface, the related classes have been put together with the ControlRoom Interface implementation and therefore reside in subfolder “**interface**”).

5.3.1 Java Interlock Server Interface

The Java implementation of an Interlock Server connection uses the concept of a listener. This listener provides methods that are called whenever data are received. So here an own class has to implement this listener and provide these methods to receive the data easily.

Following classes are provided in package **com.oapc.iface** to be used by an application:

- **OAPCIServer** – the interface itself that manages the connection to the server, data reception and transmission from and to them
- **OAPCIServerListener** – the listener interface that has to be implemented in an own class to get informed about data received from the Interlock Server
- **IServerData** – a storage class that is used to store data of different types and that has to be used to transmit values to the server and to decode received data

To use the Java Interlock Server interface following steps are necessary:

- implement an own (inner) class that implements OAPCIServerListener and create an object of this class
- create an object of type OAPCIServer and add the precedent created listener to this one
- react on data transmitted from the Interlock Server via the listener dependent on their purpose
- transmit data to the Interlock Server depending on the purpose of the application

5.3.1.1 The Interface OAPCIServerListener

This interface has to be implemented in an own class and has to provide a method that is called from the OAPCIServer object where this listener is registered at:

`public void dataReceived(String nodeName,int cmd,long ios,IServerData[] values)`

This method is called automatically whenever new data are received from the Interlock Server, it can be used to get data from the server and to react on state changes. Here `nodeName` is the name of the data node that was changes or that is submitted due to a request to get data. The second parameter `cmd` gives additional information why this method was called, following values are possible:

- `OAPC_CMDERR_DOESNT_EXISTS` is given in case data of a node have been requested that is not known to the Interlock Server
- `OAPC_CMD_GET_VALUE` signals that the method is called as a result of a request to get a single value out of the Interlock Servers data space

The parameter `ios` describes which data of which type are submitted with this method call. It consists of up to eight OR-concatenated flags of type `OAPC_xxx_IOy`. Here `xxx` is a place holder for the type which can be one of `DIGI`, `CHAR` or `NUM` and `y` is a number in range 0..8 which specifies which of the eight possible in- or outputs contains these data.
The last parameter `values` is an array with a size of eight elements. Every element is related to one of the eight possible IOs and is either null (when no data are available) or contains an object of type `IServerData` that contains digital, numerical or character data according to the IO-flags given with `cmd`.

5.3.1.2 The Class OAPCIServer

This class contains the whole functionality that is necessary to communicate with the Interlock Server. It extends a Java Socket to connect with the server, to send data to it and to check for data from there and to call all registered listener objects when new data are available.

This class contains several constants as described below.

General constants:

`MAX_NODENAME_LENGTH` – definition for the maximum length of the name of an data element

`MAX_TEXT_LEN` – definition of the maximum length a text (character data) is allowed to have

`MAX_NUM_IOS` – definition of the maximum number of IOs one single data element may have

Error and return codes:

`OAPC_OK` – everything is OK, an operation was performed successfully

`OAPC_ERROR_CONNECTION` – a required connection could not be established

`OAPC_ERROR_NO_DATA_AVAILABLE` – this value is returned when data are requested for an output where no new data are available at this moment

`OAPC_ERROR_RESOURCE` – a required resource could not be created/accessed

Following public methods and constructors can be used in order to interact with the Interlock Server:

`public OAPCIServer()`

Default constructor, a connection is established to a local Interlock Server using the default port number; the constructor throws an IOException in case the connection to the server could not be established

`public OAPCIServer(InetAddress connectAddr)`

Constructor to create a connection using the Interlocks Server default port using the `connectAddr` as IP of the server to connect with. This constructor may cause an IOException in case the connection to the server could not be established.

`public OAPCIServer(int port, InetAddress connectAddr)`

Constructor to create a new connection to an Interlock Server using the port number `port` and the address `connectAddr`. This constructor throws an IOException in case the connection to the server could not be established.

`public void close()`

Closes the connection to the server and releases all resources. After this method was called no data are received from the server and no data can be send to it any more.

`public void addOAPCIServerListener(OAPCIServerListener listener)`

Adds a `listener` object of type OAPCIServerListener to keep track of changes within the server and to receive requested data.

`public void removeOAPCIServerListener(OAPCIServerListener listener)`

Removes a listener out of the list of currently registered listeners. The listener object that has to be removed needs to be handed over in parameter `listener`.

`public boolean connectionValid()`

This is an elementary method, it tells a calling instance if the connection to the server is still valid or not. Only in case this function returns `true` it is possible to send data to the server and to get informed about changes via the listener.

`public int requestData(String nodeName)`

Requests the data of a specific node from the server; the response is not given directly but is sent asynchronously to the connected listener(s). The parameter `nodeName` expects the name of the data node to fetch the data for. This name is given when the listener is called to identify the received data. The method returns `OAPC_OK` in case the request could be submitted successfully or an error code otherwise; in case the operation failed the connection to the server was closed and this object can't be used any longer for communication.

`public int requestAllData()`

Requests the data of all nodes from the server; the response is not given directly but is sent asynchronously to the connected listeners;
PLEASE NOTE: dependent on the amount of data stored within the server this call may cause a heavy load due to the number of transmitted data.
The method returns `OAPC_OK` in case the request could be submitted successfully or an error code otherwise; in case the operation failed the connection to the server was closed and this object can't be used any longer for any communication.

`public int setData(String nodeName,long ios,IServerData[] values)`

Using this method a bunch of up to eight data elements can be sent to the Interlock Server and set for a specific data element. Here `nodeName` specifies the data node where the data have to be set at. This name consists of the unique name of the data node and the direction “in” our “out” in style “/name/dir”.

The parameter `ios` is a bit mask that consists of or-concatenated `OAPC_xxx_IOy` constants and specifies if and which IOs contain which kind of data; here the IO position (`y`) corresponds to the index position of the "`values`"-array.
The last parameter `values` is an array of objects that contains the data to be sent according to the given flags in `ios`. The elements of this array have to consist of objects of type `IServerData`.
This method returns OAPC_OK in case the request could be submitted successfully or an error code otherwise; in case the operation failed the connection to the server was closed and this object can't be used any longer for communication.

`public int setValue(String nodeName,long io,IServerData value)`

Using this method one single set of data can be sent to the Interlock Server and can be set for one

input or one output of a specific node. Here the parameter `nodeName` specifies the data node where the data have to be set at, this name is constructed in style “/name/dir”.
Here the parameter `ios` does not expect several OR-concatenated IO-flags but a single constant of type `OAPC_xxx_IOy` that specifies which IO has to be set using which kind of data. Here the IO position (y) corresponds to the input/output number of the data node. The last parameter `value` keeps an object of type IServerData that holds the data to be sent according to the given `io`.

This method returns OAPC_OK in case the request could be submitted successfully or an error code otherwise; in case the operation failed the connection to the server was closed and this object can't be used any longer for communication.

5.3.1.3 The Class IServerData

This class is used for storing character, digital or numerical data either to transmit them to the Interlock Server or when received from there.

The storage and data identification is done via public member variables, all of them can be accessed directly in order to access the stored values:

`io` – this variable stores constant of type `OAPC_xxx_IOy` which identifies the kind of data and the number of the input it belongs to

`digi` - variable to store digital data, the contents of this variable are valid only in case a constant `OAPC_DIGI_IOy` is stored in `io`

`num` - variable to store numerical data, the contents of this variable are valid only in case a constant `OAPC_NUM_IOy` is stored in `io`

`str` - variable to store character data, the contents of this variable are valid only in case a constant `OAPC_DIGI_IOy` is stored in `io`

`public IServerData()`

Default constructor, creates an object of type `IServerData` and intialises its public members to default values, no data type is set.

`public IServerData(boolean digi)`

Construtor for an object of type `IServerData` that is initialized with an digital value. This constructor does NOT set the io-information that identifies the data type, this public member variable has to be set after construction and before this object is used for data submission.

`public IServerData(double num)`

Construtor for an object of type `IServerData` that is initialized with an numerical value. This constructor does NOT set the io-information that identifies the data type, this public member variable has to be set after construction and before this object is used for data submission.

`public IServerData(String str)`

Construtor for an object of type `IServerData` that is initialized with an character value. This constructor does NOT set the io-information that identifies the data type, this public member variable has to be set after construction and before this object is used for data submission.

5.3.2 C/C++ and other languages Interlock Server Interface

The C/C++ interface to the Interlock Server does not make use of a special implementation, it uses the `oapc_iserver_`-functions of liboapc as described below. This is a general shared library interface. The headerfile liboapc.h can be used within C/C++ applications, the shared library interface itself is not limited to this language.

For a more detailed description please refer the section below, an example C application that uses this interface can be found within the SDK in subfolder “**iserver**”.

5.3.3 Instruction List Interlock Server Interface

The Interlock Server can be accessed from an IL (Instruction List) script. Such a script can be used together with an IL interpreter that comes with the OpenAPC package and is part of the ControlRoom application bundle. After this interface is not a real source implementation but a separate interpreter application the Instruction List commands and parameters and the IL interpreter ilPLC is described within the main manual.

5.3.4 LUA Interlock Server Interface

The Interlock Server can be accessed from a Lua script. Such a script can be used together with an Lua interpreter that comes with the OpenAPC package and belongs to the ControlRoom application bundle. After this interface is not a real source implementation in Lua language but a separate interpreter application the special Lua commands and parameters and the Lua interpreter luaPLC is described within the main manual.

6 Shared Library liboapc

The software components of the OpenAPC package use one general, shared library “liboapc”. This library contains several common functionalities that are ported for all platforms the OpenAPC software package is available for (and some platforms where it currently doesn't runs on officially). This library is used by the main application and by some of the external plug-ins but can be used in other applications too to have platform-independent access to several platform-dependent functionalities.
Following the functions of this library are described so that it can be used by other plug-ins and applications.

The sources of the complete library liboapc are contained within the SDK.

All the functions described below are contained within the general header file “liboapc.h” that is provided by the SDK. Beside of that also the header file “oapc_libio.h” might be necessary. This header file is responsible for all plug-in related definitions but also contains some data that are used by “liboapc.h”.

The external library liboapc is divided into different functional sections that are described below.

6.1 TCP/IP Related Functions of liboapc

The TCP/IP-functionalities of liboapc can be found as source within the SDK in file “liboapcTCPFcts.cpp” and contain several functionalities that are related to data transfers via a TCP/IP connection. All the functions names start with `oapc_tcp_`:

`int oapc_tcp_connect_to(char *address,unsigned short connect_port)`

Using this function a TCP/IP connection to a remote server socket can be established. It requires the `address` in number format “x.x.x.x” or a host name in format “www.domain.tld” and a port number `connect_port` in range 1..65535 to connect with.

When the connection to the remote socket could be established successfully a number >0 is returned, in case of an error -1 is used. The returned positive number is the identifier for the socket connection that was opened, it has to be stored for later usage with data handling functions for sending and receiving.

`void oapc_tcp_set_blocking(int sock,char block)`

This function sets a valid socket that was opened using `oapc_tcp_connect_to()` or `oapc_tcp_listen_on_port()` to blocking or non-blocking mode. When the functions `oapc_tcp_send()` and `oapc_tcp_recv()` have to be used together with their timeout-feature a socket has to be set to non-blocking before.
For the parameter `sock` the socket number is required as returned by the function that created this socket, `block` specifies if this socket has to be set to blocking (=1) or to non-blocking mode (=0).

`int oapc_tcp_send(int sock, const char *msg,int len, int flags,int msecs)`

This function transmits data using a valid socket connection. Here `sock` is the socket that has to be used for data transmission, `msg` is a pointer to the data block that has to be transmitted and `len` specifies the length of this data block in bytes. Using the parameter `flags` some special transmission flags can be given, currently only `MSG_NOSIGNAL` is supported on Linux (this flag is mandatory for Linux in order to avoid application interruptions in case of a data transmission error).
The last parameter `timeout` specifies a time in msec that has to be used for data transmission at maximum. This parameter is valid only when the related socket was set to non-blocking mode before by calling `oapc_tcp_set_blocking()`. In this case the function returns after all data have been sent or latest after

the specified time-out has elapsed. In case the socket is set to blocking, the parameter `timeout` is ignored and the function returns only after all data have been sent.
After finishing the function returns the number of bytes that really have been sent.

`bool oapc_tcp_recv(int sock,char *data, int len,const char *termStr,long timeout)`

With this function data can be received from a valid and opened socket in case some are available. The socket to read the data from is specified with parameter `sock`, `data` points to the memory area where the received data have to be stored into an `len` specifies the maximum size of data that have to be read. In case of ASCII-based data the parameter `termStr` can be used to define an additional condition for the end of the data reception: when a character not equal to NULL is specified here the function returns as soon as this character is received. In this case less data than specified by `len` are received.
The last parameter `timeout` specifies a time in msec that has to be used for data reception at maximum. This parameter is valid only when the related socket was set to non-blocking mode using `oapc_tcp_set_blocking()`. In this case the function returns after all data have been received or after the `termStr`-character was found in the stream of received data or after the specified time-out has elapsed. In case the socket is set to blocking, the parameter `timeout` is ignored and the function returns only after all data have been received or after the `termStr`-character was found.
This function returns `true` in case all data could be received or the `termStr`-character could be found, `false` otherwise.

`void oapc_tcp_closesocket(int sock)`

To finish all data transmissions and to clean up a socket this function has to be called. It shuts down and closes the socket connection specified by parameter `sock`.

`int oapc_tcp_listen_on_port(unsigned short port, char *bindToIP)`

Using this function a server socket can be created that doesn't connects to a remote socket but waits for incoming connections and is able to pick up these connections by calling function `oapc_tcp_accept_connection()`.
Here the parameter `port` specifies the port number the server socket has to listen at. This has to be a number in range 1..65535. The port itself has to be unused, elsewhere this function will fail. On some operating systems the usage of port numbers is restricted to users with special privileges so it is recommended to use a value in range 1025..65535 here. The second parameter `bindToIP` specifies the IP number in format “x.x.x.x” the server socket has to be bound to. In most cases this parameter will be set to “0.0.0.0”.
When the function could create a socket connection successfully it returns a value >0 that is equal to the socket identifier that has to be used for following operations. This socket has to be set to non-blocking mode when the time-out-feature of the data sending/reception functions has to be used.
In case of an error the returned value is <0.

`int oapc_tcp_accept_connection(int sock, unsigned long *remote_ip)`

This function can be used for sockets created with `oapc_tcp_listen_on_port()` only and needs to be called regularly in order to accept new incoming connections. Here the parameter `sock` specifies the server socket that has to be checked for a new incoming connection. In case there is a new connection the remote connections IP of it is stored in `remote_ip`.
When the function returns the socket of the newly accepted connection is returned and should be set to non-blocking mode in case the timeout-feature of the data sending/reception functions has to be used with this new socket. When the server socket is used in blocking mode this function will return only when a new incoming connection was accepted. When the server socket `sock` is used in non-blocking mode the function will return immediately. In this case it returns a value >0 only in case a new socket could be accepted.

6.2 Serial Interface Functions of liboapc

The serial interface functionalities of liboapc can be found as source within the SDK in file "liboapcSerialFcts.cpp" and contain several functionalities that are related to data handling via the serial interface of a computer. Here it doesn't matters if a real serial interface is used or if it is a serial interface provided via the USB-port or via a converter like an USB-to-Serial adapter. All the function names start with `oapc_serial_`. Following the data type of the handler of the serial interface is specified by the place-holder `<handle>`. The data type is different for Windows and for POSIX operating systems, for Windows it is defined as data type `HANDLE` for all other systems it is a plain `int`. For the usage of the functions described here there is no difference, within an application that uses the serial interface with these functions no difference has to be made due to these data types:

`int oapc_serial_port_open(struct serial_params *serialParams,<handle> *fd)`

This function tries to open a serial interface of a native serial port and to configure it using the parameters according to the values specified with the structure `serialParams`. This structure contains the following data that describe the port parameters:

```
struct serial_params
{
   char           port[MAX_TTY_SIZE];
   unsigned short brate,databits,parity,stopbits,flowcontrol;
};
```

Here port is the operating system dependent name of the port that has to be opened (e.g. "`COM1`" or "`/dev/ttyS`" or "`/dev/ser0`"). The following values specify the bitrate, the number of data bits, the parity mode, the number of stop bits and the flow control mechanism that has to be used. Please note: here not the required values are stored but index values that specify them. These index values are defined by the XML configuration structure that has to be used for the set-up of the serial interface. Here the index values returned by the main application can be used directly as index values for the parameters of the structure above. It is recommended to use this standard structure in order to get the correct index values for the different serial interface parameters:

```
<general>
 <param>
  <name>port</name>
  <text>Interface</text>
  <type>string</type>
  <default>%s</default>
  <min>4</min>
  <max>12</max>
 </param>
 <param>
  <name>brate</name>
  <text>Data Rate</text>
  <unit>bps</unit>
  <type>option</type>
  <value>110</value>
  <value>300</value>
  <value>1200</value>
  <value>2400</value>
  <value>4800</value>
  <value>9600</value>
  <value>19200</value>
 <value>38400</value>
  <value>57600</value>
  <value>115200</value>
  <value>230400</value>
  <default>%d</default>
 </param>
 <param>
  <name>databits</name>
  <text>Data Bits</text>
  <type>option</type>
  <value>5</value>
  <value>6</value>
  <value>7</value>
  <value>8</value>
 <default>%d</default>
 </param>
 <param>
  <name>parity</name>
  <text>Parity</text>
  <type>option</type>
  <value>None</value>
  <value>Even</value>
  <value>Odd</value>
  <default>%d</default>
 </param>
 <param>
  <name>stopbits</name>
  <text>Stop Bits</text>
  <type>option</type>
  <value>1</value>
  <value>1.5</value>
  <value>2</value>
  <default>%d</default>
 </param>
 <param>
  <name>flowcontrol</name>
  <text>Flow Control</text>
  <type>option</type>
  <value>None</value>
  <value>Xon / Xoff</value>
  <value>CTS / RTS</value>
  <default>%d</default>
 </param>
 <param>
  <name>uname</name>
  <text>Username</text>
  <type>string</type>
  <default>%s</default>
  <min>0</min>
  <max>%d</max>
 </param>
 <param>
  <name>pwd</name>
  <text>Password</text>
  <type>string</type>
  <default>%s</default>
  <min>0</min>
  <max>%d</max>
 </param>
</general>
```

Beside these parameters the serial interface is set to non-blocking mode automatically.

Please note: it is recommended to use this function only for native serial ports that are not USB-devices. For USB-based serial interfaces where all these settings regarding bitrate, data bits, stop bits, parity anf low control do not matter since they always work with the USB-speed of the device, please use function `oapc_serial_usb_port_open()` instead!

In case the serial port could be opened and configured successfully the function returns OAPC_OK and writes the handle of the opened interface into the variable where `fd` points to. In case opening or configuring failed the function returns a code OAPC_ERROR_xxx.

`int oapc_serial_usb_port_open(const char *portname,<handle> *fd)`

This function tries to open a serial interface of a USB-based device which is not a native serial hardware (in sense of legacy hardware with separate lines for RX and TX). Since such USB-devices always work with the speed of the used USB-port, it is not necessary to specify any further serial interface parameters beside the `portname` which identifies the port to be opened (which is “`COMx`” for Windows or “`/dev/ttyACMx`” oder “`/dev/ttyUSBx`” or similar for Linux, where “x” is a number provided by the operating system and which specifies the port).

In case the USB serial port could be opened, the function returns OAPC_OK and writes the handle of the opened interface into the variable where `fd` points to. In case opening failed the function returns a code OAPC_ERROR_xxx.

`int oapc_serial_recv_data(<handle> fd,char *buffer,int toLoad,long msecs)`

This function is deprecated and should not be used any longer, it will be removed in future software versions. Please use `oapc_serial_recv()` instead!

`int oapc_serial_recv(<handle> fd,char *buffer,int toLoad,char *termStr,long msecs)`

Using this function data can be received from the serial interface. The parameter `fd` specifies the already opened port to read the data from, `buffer` points to the memory area where they have to be written into and `toLoad` specifies the maximum number of bytes that have to be received and written to that buffer. The last parameter `msecs` specifies a maximum time this operation may need for reception of data, when the limit in milliseconds given here is exceeded, the operation is aborted.
This function returns the number of bytes that could be read really. This value may be smaller than the value given in parameter `toLoad` in case the time-out value `msecs` was exceeded.
As a second exit condition the parameter `termStr` can be set to a value not equal to NULL, in this case the function returns as soon as the string that is given here can be found within the received data.

`int oapc_serial_send_data(<handle> fd,const char *msg, int len,int msecs)`

This function is deprecated and should not be used any longer. It will be removed in future software versions. Please use `oapc_serial_send()` instead.

`int oapc_serial_send(<handle> fd,const char *msg, int len,int msecs)`

With this function data can be send to the serial interface. The parameter `fd` specifies the open port to send the data to, `msg` points to a memory area that contains the data to be send and `len` specifies the maximum number of bytes that have to be send out of this buffer. The last parameter `msecs` specifies a maximum time this operation may need for transmission of data, when the limit in milliseconds given here is exceeded, the operation is aborted.
This function returns the number of bytes that could be sent really. This value may be smaller than the value given in parameter `len` in case the time-out value `msecs` was exceeded.

`void oapc_serial_port_close(<handle> *fd)`

After all operations on a serial port have been finished and it is not used any longer, it has to be closed. That has to be done using this function where a pointer to the serial interface's handle has to be handed over. This function closes the serial port and initializes the handle to avoid that this – now invalid – handle is used again.

6.3 Utility-Functions of liboapc

This set of functions provides different things that are used in nearly all kind of applications. These functionalities of liboapc can be found as source within the SDK in file “liboapcUtilFcts.cpp” and their names start with `oapc_util_`:

`double oapc_util_atof_dot(const char *c)`

This function converts a floating point number that is represented by an ASCII-string to a floating point value. So this is similar to functions `atof()` but with one major difference: here conversion is independent from current locale, this function always expects a dot as decimal delimiter. Thus it can be used to convert values out of ASCII file formats that make use of this notation.

`void oapc_util_dbl_to_block(double inValue,struct oapc_num_value_block *outBlock)`

This function converts the architecture-dependent representation of a “double” data type `inValue` to a platform-independent and portable representation that is stored within a structure where `outBlock` points to. The resulting `outBlock` data can be used for transmission over the network or for saving data in a portable way so that these data can be read on completely different platforms.

`double oapc_util_block_to_dbl(struct oapc_num_value_block *inBlock)`

Here the opposite is done than within the preceding function: a platform- and architecture-independent representation of a floating point data type that is stored in `inBlock` is taken and converted to the local format of a “double” data type that is returned by this function.

`bool oapc_util_create_thread(void *(*start_routine)(void*), void *arg)`

This function is deprecated and will be removed in future software versions, please use `oapc_thread_create()` instead

`int oapc_util_thread_sleep(int msecs)`

This function is deprecated and will be removed in future software versions, please use `oapc_thread_sleep()` instead

`bool oapc_util_thread_set_prio(const unsigned char prio)`

This function is deprecated and will be removed in future software versions, please use `oapc_thread_set_prio()` instead

`bool oapc_util_to_unicode(char* str,wchar_t* out,int outSize)`

With this function plain Latin-1 encoded ASCII text can be converted to Unicode. The input text has to be handed over using parameter `str`, the converted data are stored in `out`. The last parameter `outSize` specifies the number of wide characters that fit into the output buffer `out`. The size of a wide character depends on the system where the library is used at depending on the standard definition of `wchar_t`.

`unsigned int oapc_util_colour2gray(unsigned int colour)`

Converts a RGB colour value to its grayscale representation. The colour value has to be given with parameter `colour`, the converted grayscale value is returned by this function.

double oapc_util_atof(const char value)

This function is a replacement for `atof()`, comparing to it it converts a character string `value` into a floating point independent from a locale. Thus this function can be used for any string value independent from its formatting. After OpenAPC applications can be distributed over different systems with probably different locales it is highly recommended to use this function only instead of atof().

void *oapc_util_get_time()

This is a platform independent possibility to get a time information with a resolution better than one second. The maximum available resolution as well as the contained time depends on the operating system. Thus these functions can't be used to evaluate the exact date, here the current time of the day can be extracted or they can be used to evaluate the difference between two time information.
This function returns a time memory area which has to be released either by the comparison function `oapc_util_diff_time()` or by `oapc_util_release_time()`.

void oapc_util_release_time(void *time)

This function releases the memory of the parameter `time` which was allocated by a preceding call of `oapc_util_get_time()`.

double oapc_util_diff_time(void *time1, void *time2)

Using this function the difference between two time values `time1` and a later time value `time2` can be evaluated. The memory areas of these time values are released by this function, no subsequent calls to `oapc_util_release_time()` are necessary. The value returned by this function is the difference between two times in unit second, the maximum accuracy of the returned value is operating system dependent.

double oapc_util_get_timeofday(void *time)

This function extracts the time of the current day out of the given information in parameter `time`. The memory area related to this parameter is not released by this function, here an additional call to `oapc_util_release_time()` is required. The current time is returned by this function (in unit seconds counted from midnight of the current day).

struct oapc_bin_head *oapc_util_alloc_bin_data(unsigned char type,unsigned char subType,unsigned char compression,int sizeData)

Tries to allocate a binary data structure (including the required head) and to initialise it with default values. Here `type` specifies the main structure type using one of the constants `OAPC_BIN_TYPE_xxx`, `subType` the sub structure type specified by one of the constants `OAPC_BIN_SUBTYPE_xxx_yyy` and compression specifies the used compression type using one of the constants `OAPC_COMPRESS_xxx`. The last parameter `sizeData` has to be used to specify the size of the payload data (excluding the length of the binary head). In case of success this function returns a fully initialised structure `oapc_bin_head` that is followed by a memory area large enough to hold `sizeData` bytes of payload data. In case of an error NULL is returned.
Binary structures that have been allocated using this function need to be released by calling `oapc_util_release_bin_data()`.

void oapc_util_release_bin_data(struct oapc_bin_head *bin)

This function has to be used to release binary memory areas that have been allocated using `oapc_util_alloc_bin_data()`.

`int oapc_util_check_maskbit(struct oapc_bin_head *bin,int x,int y)`

This function checks if a mask bit within a black/white mask bitmap is set or not and returns 1 whe nthe mask bit is set and 0 otherwise. This function expects a binary structure `bin` of type `OAPC_BIN_TYPE_IMAGE`, subtype `OAPC_BIN_SUBTYPE_IMAGE_BW1` to be used for checking the mask bit at coordinates specified by `x` and `y`. In case no binary struture is handed over, a binary structure of wrong type/subtype is given, in case the requested coordinates are not located within the bitmap specified by the binary structure or in case of an other error 1 is returned by this function.

6.3.1 Ring Buffer Utility Functions of liboapc

There exists a subset of utility-functions for liboapc that can be used to implement a fast ring buffer. Such a ring buffer can be used to implement FIFO-style data handling where stored values are put into a queue and can be pulled out of that queue in the order as they have been put into it.

This ring buffer is not thread safe, so in case two or more threads are accessing its functions they have to be synchronised via a mutex and the functions `oapc_thread_mutex_lock()` and `oapc_thread_mutex_unlock()`.

The ring buffer functions can be identified via its name, they all start with oapc_util_rb_. To manage a ring buffer a special data storage variable of type `struct oapc_util_rb_data` has to be used, a pointer to this variable has to be handed over at every function call. This variable will be used by the ring buffer functions to store important buffer management information, thus it mustn't be changed in any way.

`int oapc_util_rb_alloc(struct oapc_util_rb_data *buffer,int elements)`

Allocates memory for a new ring buffer and initialises the buffer management information in variable `buffer`. The contents of this variable that have to be handed over as pointer don't need to be initialised before, this is done by this function completely. The second parameter `elements` specifies the capacity of the ring buffer.

This function has to be called as very first. All the other ring buffer functions can be used only in case it returns `OAPC_OK`.

`int oapc_util_rb_release(struct oapc_util_rb_data *buffer)`

This function has to be called whenever a ring buffer is no longer needed, it releases all memory that is required for handling the ring buffer. In case this function is called with a `buffer` value that already has been released, nothing will happen, the function just returns with `OAPC_OK`.

PLEASE NOTE: this function only releases the ring buffer management data, it does not influence any data that may still be stored within the ring buffer! So it is recommended to empty the whole ring buffer and to release the associated memory areas <u>before</u> calling this function.

`int oapc_util_rb_push(struct oapc_util_rb_data *buffer,void *data)`

Appends a new pointer to a memory area to the ring buffer. The variable handed over via `buffer` has to be allocated before by calling `oapc_util_rb_alloc()`, elsewhere the results of this function may be undefined. A pointer to the memory area that has to be added to the buffer needs to be handed over via parameter `data`, it is stored within the ring buffer and can be fetched later via a call to `oapc_util_rb_front()` as soon as it has reached the beginning of the queue.

This function returns `OAPC_OK` in case the new data could be added to the buffer successfully or `OAPC_ERROR_RESOURCE` in case the buffer is full and no more space is available in ring buffer.

`void *oapc_util_rb_front(struct oapc_util_rb_data *buffer)`

This function returns a pointer to a memory area that is positioned at the front of the ring buffer and is available next. This function just returns the next pointer, it does not remove it from the ring buffer. To do that and to get access to the next available element on next call the function `oapc_util_rb_pop()` has to be used separately. In case the buffer is empty and no more data are available the function returns `NULL`.

`int oapc_util_rb_pop(struct oapc_util_rb_data *buffer)`

Removes the current data from the ring buffer that is specified via parameter `buffer`. This function typically is called after fetching the data via `oapc_util_rb_front()` so that the next data out of the ring buffer can be accessed afterwards. This function returns `OAPC_OK` in case the current value could be removed from the buffer successfully or `OAPC_ERROR_RESOURCE` in case the ring buffer is empty and no more data are available for removal.

`bool oapc_util_rb_empty(struct oapc_util_rb_data *buffer)`

Using this function it can be checked if the ring buffer specified by parameter `buffer` is already empty or not. It returns `true` in case no data are stored within the buffer and `false` otherwise.

`bool oapc_util_rb_full(struct oapc_util_rb_data *buffer)`

Using this function it can be checked if the ring buffer specified by parameter `buffer` is full or if there is any space left to add new data. It returns `true` in case it is full and no more data can be stored within the buffer and `false` in case there is some space left for data.

6.4 Thread-Functions of liboapc

The following functions offer a threading-related functionality. This includes management of threads as well as some thread-related functionalities like delays, thread-switches, signalling, mutual exclusions and others more.
Most of these functions exist in very similar implementations on several platforms but the liboapc-functions offer a platform-independent, standardised interface. So when this library is used instead of the platform-dependent ones, porting of liboapc-based applications to an other operating system becomes much easier.

`void *oapc_thread_create(void *(*start_routine)(void*), void *arg)`

This function creates a new thread. Here the parameter `start_routine` points to a function of type "`void *fct(void *arg)`" that is created as new thread. The second parameter `arg` can be NULL or it can point to data that are handed over to the new thread as argument.
This function returns a handle to the newly created thread or NULL in case of an error.

`bool oapc_thread_set_prio(void *handle,const unsigned char prio)`

Changes the priority of the thread specified by parameter `handle`. The second parameter `prio` specifies the new priority of the thread in range -2..2 where -2 is the lowest priority possible, 0 is the default priority of a thread as it would be set after thread creation and 2 is the maximum priority which has the potential to slow down or stall a system completely. The function returns `true` in case the new priority could be set successfully, `false` otherwise.

`void oapc_thread_release(void *handle)`

Releases all resources that are assigned to the thread specified by parameter `handle`. This function does not stop or terminate a thread, is must be called only after a thread has terminated.

`int oapc_thread_sleep(int msecs)`

This function makes the current thread sleep for the given number of `msecs` and typically causes a thread switch so that other threads that are not sleeping at the moment become active. The value for `msecs` has to be a positive number. Its accuracy and minimum resolution is highly platform dependent. Typically a value of 0 causes only a thread switch, values greater than 0 may be used for real delays that can be only as exact as the underlying platform is able to handle them.

`void *oapc_thread_mutex_create(void)`

This function creates a new mutual exclusion instance that can be used together with the following mutex functions.
It returns a handle to the newly created mutex object or NULL in case of an error.

`void oapc_thread_mutex_lock(void* handle)`

Using this function the entry point of a section can be marked. The function returns only in case it has exclusive access to the following code. Means when an other thread has entered that section this function blocks until the other thread makrs this section as left by calling `oapc_thread_mutex_unlock()`.
The parameter expects a handle to a mutex object that was created with a call to `oapc_thread_mutex_create()` before.

`void oapc_thread_mutex_unlock(void* handle)`

Using this function the exit point of a code section can be marked. This function has to be called exactly once after `oapc_thread_mutex_lock()` has been used. It releases the lock to the marked section so that other threads can enter it via the call of `oapc_thread_mutex_lock()` exclusively.

The parameter expects a handle to a mutex object that was created with a call to `oapc_thread_mutex_create()` before.

`void oapc_thread_mutex_releases(void* handle)`

Releases all resources assigned to the mutex that is specified by parameter `handle`. After calling this function no more calls to `oapc_thread_mutex_lock()` or `oapc_thread_mutex_unlock()` are allowed.

`bool oapc_thread_timer_start(void (*start_routine)(void*,int),int time,void *data,int timerID)`

This function starts a timer and executes a given function as soon as the specified time has elapsed. Here the smallest possible resolution of the timer depends on the underlying operating system.
The parameter `start_routine` defines the callback function that has to be executed after the given `time` in milliseconds has elapsed. When the callback routine is executed the specified `data` are handed over to it together with the `timerID` that can be used to identify the timer.

`void *oapc_thread_signal_create()`

This function creates a new handle that can be used for signalling between threads: such a handle can be used to send one signal to an other thread that stops execution until a signal arrives. This is an other inter-thread synchronisation mechanism but different to mutexes its up to the application itself to decide which thread has to wait how long. The signal handler is returned by this function, in case of an error NULL is given back. The returned handler has to be released after it is no longer needed.

`int oapc_thread_signal_send(void *handle)`

A call to this function corresponds to `oapc_thread_signal_wait()`: when this function is called a thread that waits for a signal continues its operation. Both, the signal and the wait function have to use the same signal handler `handle`. In case the operation could be executed successfully `OAPC_OK` is returned or an error code otherwise.

`int oapc_thread_signal_wait(void *handle,int msecs)`

When this function is called execution of the current thread is stopped until a signal is sent by a call to function `oapc_thread_signal_send()`. Once a signal is received from an other thread that uses the same signal handler `handle`, the current thread continues operation. Alternatively a time-out can be specified using parameter `msecs`: when the time specified there has elapsed the function returns also in case no signal was received. To let the function wait for a signal infinitely and without any time-out a value of -1 has to be given here.
Please note: dependent on the operating system this function may return when a signal is already pending. Means when a signal is sent to the given handle before the wait function is called, this pending signal is used to let the wait function return immediately.

`void oapc_thread_signal_release(void *handle)`

After a signal handler `handle` is no longer used this function has to be called to release all signalling-related resources. When this function was called, the signal handler is no longer valid and can't be used any longer.

6.5 Dynamic Library Functions of liboapc

This section describes a bunch of functions that are necessary for managing plug-ins and shared libraries that are loaded during runtime. They offer platform-independent functions for loading an external library and fetching the contained functions via their symbol names.

`void oapc_dlib_load(const char name)`

This function tries to load an external shared library with the given `name` so that it can be used together with the following functions. Here the name depends on the platform, it is case-insensitive on Windows and typically uses the extension ".DLL", it is case-sensitive on Unix- and Linux-systems and uses the extension ".so". The function returns a handle to the opened and loaded library in case it could be loaded successfully or NULL otherwise.

`void *oapc_dlib_get_symbol(void *handle, const char name)`

Using this function a pointer to a function within a loaded library can be retrieved for later use. The first parameter is a handle to the library that was loaded with `oapc_dlib_load()` before, the second one is the name of the symbol the function to be loaded belongs to.
This function returns NULL in case no symbol with the given name could be found or a pointer to the related function otherwise.

`void oapc_dlib_release(void *handle)`

Here all resources related to the loaded library that is specified by the given parameter `handle` are released and the library is unloaded in case no other applications use it. After using this function no more calls to `oapc_dlib_get_symbol()` are allowed.

6.6 ControlRoom-Interface-Functions of liboapc

The `oapc_iface_` functions can be used within external applications to implement and to provide a standard ControlRoom TCP/IP-based interface easily. Once such a interface is set up this application receives data from a connected ControlRoom player and can react on these data depending on their meaning. The whole process of implementing such an interface requires some quite simple function calls only:

1. set a callback where incoming data are announced at by calling `oapc_iface_set_recv_callback()`
2. (optionally) set authentication information using `oapc_iface_set_authentication()`
3. initialise the interface with `oapc_iface_init()`
4. jump into the main loop of the application and react on data announced at the callback function
5. remove the interface by calling `oapc_iface_exit()` before the application is left

The main job is done in step 3, here the callback function becomes active and does the job of the application by reacting on data that are send from the ControlRoom project that connected to this interface.

Following interface functions are provided by the library:

`int oapc_iface_set_recv_callback(lib_oapc_iface_callback oapc_iface_callback)`

This function has to be called before the interface is initialized: here a callback function `lib_oapc_iface_callback` has to be specified where state information and received data are handed over during communicating with connected clients.

This function returns `OAPC_OK` in case the callback function could e registered successfully or an error code otherwise.

The registered callback function is defined as follows:

`typedef void (*lib_oapc_iface_callback)(int type,char *cmd,unsigned char digi,float num,char *str,struct oapc_bin_head *bin,int socket)`

This function is called whenever new data are received from a connected ControlRoom player. It hands over the following data to the application:

`type` – specifies the type of call; depending on this type the following parameters contain some data or not:

- `OAPC_IFACE_TYPE_STATE_NEW_CONNECTION` - a new client has connected to the interface; here the parameter socket is used to identify this new connection
- `OAPC_IFACE_TYPE_STATE_CONNECTION_CLOSED` - a client has closed its connection to the interface
- `OAPC_IFACE_TYPE_STATE_CONNECTION_ERROR` - an error occurred for a client
- `OAPC_IFACE_TYPE_DIGI` - digital data have been received together with the assigned command
- `OAPC_IFACE_TYPE_NUM` - numeric data have been received together with the assigned command
- `OAPC_IFACE_TYPE_CHAR` - character data have been received together with the assigned command
- `OAPC_IFACE_TYPE_BIN` - binary data have been received together with the assigned command

`cmd` - contains the command that is assigned to received data

`digi` - received digital data

`num` - received numeric data

`char` - received character data

`bin` - received binary data

`socket` - the socket where these data have been received at; this value can be used together with `oapc_iface_send_`-functions to give a direct reply to this connection or to send data to all connected ControlRoom applications except this one. Whenever a value >0 is handed over here valid socket data are given that should be stored until a connection leaves with `OAPC_IFACE_TYPE_STATE_CONNECTION_CLOSED` or `OAPC_IFACE_TYPE_STATE_CONNECTION_ERROR` in order to send data to this connection asynchronously.

`int oapc_iface_set_authentication(char *uname,char *pwd)`

The ControlRoom interface implementation may use a user authentication by using a login name and a password. When these authentication parameters are set within the ControlRoom project, the same data have to be specified for the application that implements the interface. That can be done using this function, it accepts a user name `uname` and a password `pwd` as parameter. When a connection is established to this interface afterwards that does not use the correct user name and password, it is rejected automatically.

This function returns `OAPC_OK` in case the authentication data could be set successfully or an error code otherwise.

`int oapc_iface_init(const char *host,unsigned short port)`

This function initialises the ControlRoom interface and starts the related server thread which is used by incoming connections for communication. The given host name `host` and the port number `port` specify where this server has to be accessible at for these incoming connections. When host is set to NULL and port to 0, the default data 0.0.0.0:1810 are used.

This function returns `OAPC_OK` in case the interface could be initialised successfully or an error code otherwise.

`int oapc_iface_send_digi(const char *cmd,unsigned char digi,int socket)`

While incoming data are announced at the registered callback this function can be used to send digital data to a connected ControlRoom application. Here `cmd` is the command that is assigned to the data, `digi` contains the digital value that has to be sent and `socket` is the identifier of the connection where the data have to be sent to.

This function returns `OAPC_OK` in case transmission of the data could be done successfully or an error code otherwise.

`int oapc_iface_send_num(const char *cmd,float num,int socket)`

While incoming data are announced at the previously registered callback, this function can be used to send numeric data to a connected ControlRoom application. Here `cmd` is the command that is assigned to the data, `num` contains the numeric value that has to be sent and `socket` is the identifier of the connection where the data have to be sent to.

This function returns `OAPC_OK` in case transmission of the data could be done successfully or an error code otherwise.

`int oapc_iface_send_char(const char *cmd,char *str,int socket)`

While incoming data are announced at the registered callback, this function can be used to send text data to a connected ControlRoom application. Here `cmd` is the command that is assigned to the data, `str` contains the text value that has to be sent and `socket` is the identifier of the connection where the data have to be sent to.

This function returns `OAPC_OK` in case transmission of the data could be done successfully or an error code otherwise.

int oapc_iface_send_bin(const char *cmd,struct oapc_bin_head *bin,int socket)

While incoming data are announced at the registered callback, this function can be used to send a binary data block to a connected ControlRoom application. Here `cmd` is the command that is assigned to the data, `bin` contains the binary data that have to be sent and `socket` is the identifier of the connection where the data have to be sent to.

This function returns `OAPC_OK` in case transmission of the data could be done successfully or an error code otherwise.

int oapc_iface_exit()

Closes all connections to incoming ControlRoom applications, de-initialises the interface and shuts it down completely. This function returns OAPC_OK in case everything could be closed successfully or an error code otherwise.

6.7 Interlock Server Access Functions of liboapc

The OpenIServer (Open Interlock Server) is an application that works in background and stores state information of all elements used within a currently running ControlRoom project. So depending on the structure of the project the current state of it can be restored automatically as long as the OpenIServer is not stopped. Beside of that it is possible to implement interlocks between user input elements and real hardware and to execute automated sequences by using and modifying these sequences.

The liboapc itself contains the functionality for accessing the Interlock Server out of own applications in order to implement such functionalities. The library contains some functions that don't have to be used externally (and are not described here) and other functions that have to be used by external applications and therefore are specified fully. The functions described below handle connecting to the server, transferring data to it, receiving data from it and perform some decoding/service tasks.

The handled payload data always consist of a member "`ios`" that describes how much and which data blocks will follow, here for IO types `OAPC_DIGI_IOx` a `struct oapc_digi_value_block` is used, for IO types `OAPC_NUM_IOx` a `struct oapc_num_value_block` will follow, for IO types `OAPC_CHAR_IOx` a `struct oapc_char_value_block` will follow and for IO types `OAPC_BIN_IOx` a `struct oapc_bin_head` is used.

If the payload value "`ios`" is equal to NULL no data are attached. The second common parameter "`cmd`" uses the `OAPC_CMD_xxx` constants to inform what has to be done exactly. The values itself are handled using an array of void-pointers. Depending on the "`ios`"-flags these array members point to data structures of type `oapc_xxx_value_block` (where "`xxx`" stands for the data type). Here `oapc_num_value_block` is a bit special, it is a platform-independent representation of a number. Such a structure can be filled using `oapc_util_dbl_to_block()` and it can be converted back to a double value by using function `oapc_util_block_to_dbl()`.

The memory areas the void-array points to are always released by the instance that creates them. So within a callback function the memory areas are valid but they are released immediately after it is left. The same has to be done when an application sends new data using function `oapc_ispace_set_data()`: liboapc will not release these data after sending them to the server, this has to be done by the instance that calls this function.

The complete procedure of using the state data is quite simple and consists of the following steps only:

1. retrieve new connection ressources by calling `oapc_ispace_get_instance()`
2. install the callback function using `oapc_set_recv_callback()`
3. connect to the Interlock Server by calling `oapc_ispace_connect()`
4. communicate with the server/use the state data
5. close the connection to the server with `oapc_ispace_disconnect()` before exiting the

application

Within step 4) data can be set or changed within the data space using function `oapc_ispace_set_data()`. This operation causes the server to immediately inform all other connected clients (including the running OpenPlayer or OpenDebugger) about the modification so that they can react on it in case they are somehow responsible to the modified data node. So if e.g. an application connected to the Interlock Server changes the numeric value of a data field that belongs to a number input field within the HMI of the ControlRoom project, this new numeric value will be displayed immediately within that input field.

Beside of that function an application may also call `oapc_ispace_request_all_data()` to retrieve the list of all data managed within the server or `oapc_ispace_request_data()` to request one specific data node

Following functions are provided by the library:

`void *oapc_ispace_get_instance()`

This function has to be called as very first to obtain instance resources for a new interlock space connection. This resource is required for all following function calls and has to be handed over as value for the parameter "handle". When no more resources are available, NULL is returned instead of a valid pointer, in this case no additional connections to the interlock server are possible. This function is not thread safe!

`int oapc_set_recv_callback(void *handle,lib_oapc_ispace_callback oapc_ispace_callback)`

This function has to be called before a connection to an Interlock Server is established, here a callback function `oapc_ispace_callback` has to be set that receives data sent from the server to the application.
ATTENTION: This function has to be called exactly once <u>before</u> connecting to the server, it is not possible to change the callback function later!
This function returns OAPC_OK when the callback function could be installed successfully or an error code otherwise. The callback function itself is defined as:

`typedef void (*lib_oapc_ispace_callback)(void *handle,char *nodeName,unsigned int cmd,unsigned int ios,void *values[MAX_NUM_IOS])`

When the callback function could be registered successfully and a connection to the Interlock Server was established, this callback function is called every time something is received from the server. Such receptions can occur for different reasons:

- a client changed something within the interlock data space, here the new values are transmitted to all other clients that may be interested in these changed data
- the own application requested a specific data block by calling oapc_ispace_request_data(), here the callback function is called either with the resulting data with no data and an error code when this specific block doesn't exists
- the own application requested a complete update by calling oapc_ispace_request_all_data(), in this case the callback function is called for every data block that exists within the Interlock Server

When this function is called from within the interface following data may be handed over to the application that holds this callback:

`handle` – a communication resource handle as returned by `oapc_ispace_get_handle()` originally, it identifies the connection where the new data arrive at; this parameter would give the possibility to check where data are received at when more interlock server connections are used from within the same application, nevertheless it is highly recommended to use a separate callback for every used connection

`nodeName` – the internal name of the data block the following data belong to; this name consists of a direction information "/in/" or "/out/" for input or output data and the name of the related node as set by an application that accesses the Interlock Server or as specified for the element within the currently running `cmd` – specifies why the function is called, using which command the data have been sent or which reason for an error occured. Here following constants are valid:

- `OAPC_CMD_SET_VALUE` – a new (requested) data block was received
- `OAPC_CMDERR_DOESNT_EXISTS` is given when a requested data block could not be found; in this case the parameter `ios` and all the related values are set to 0

`ios` – the IO flags that specify which fields of the following values-array contain which kind of data; here a OR-concatenated list of `OAPC_xxx_IO`-flags is given that specifies which of the following value array indices contain which kind of data

`values` – the data itself; this is an array of pointer to data blocks containing the data types specified by the parameter `ios`, when no IO exist for a field, the related index is NULL

`int oapc_ispace_connect(void *handle,const char *host,unsigned short port,struct oapc_ispace_auth *auth)`

This function tries to connect to an existing Interlock Server using the host name `host` and the port number `port`. In case one or both of these parameters are set to 0, the default values are used. This function returns `OAPC_OK` when the connection could be established successfully, `OAPC_ERROR_RESOURCE` in case no callback function was defined to receive answers from the server or `OAPC_ERROR_CONNECTION` when the server is not accessible.

`int oapc_ispace_request_data(void *handle,const char*nodeName,struct oapc_ispace_auth *auth)`

This function explicitly requests a specific data block from the server. When the requested data exists, the result will be received liboapc-internal and forwarded to the callback function. It is not necessary to call this function repeatedly in order to get informed about state changes within the Interlock Server, this is done automatically as soon as a client has connected to the server and as soon as a value within that server changes.
This function requires the unique `nodeName` of the requested data block, the same name will be provided within the callback function as soon as the related data are received. When the requested data block could not be found, a specific answer is sent to the callback function, in this case no data are provided but the callbacks "`cmd`" parameter is set to `OAPC_CMDERR_DOESNT_EXISTS`.
The additional parameter `auth` is reserved for future use and has to be set to NULL at the moment.
This function returns `OAPC_OK` when the request could be transmitted successfully or an error code otherwise. In case of an error the socket connection is closed by this function and has to be re-established by the main application in order to continue communication with the Interlock Server.

`int oapc_ispace_request_all_data(void *handle,struct oapc_ispace_auth *auth)`

This function requests all available data blocks from the server. The result will be received internally and forwarded to the callback. The data blocks itself can be identified by the node name that is handed over to the callback function.
PLEASE NOTE: calling this function may result in a longer operation where a bigger amount of data is transferred to the client, that depends on the number of data nodes that are currently handled by the server.
The additional parameter `auth` is reserved for future use and has to be set to NULL at the moment.

`int oapc_ispace_set_data(void *handle,const char *nodeName,unsigned int ios,void *values[MAX_NUM_IOS],struct oapc_ispace_auth *auth)`

This function has to be called by a client application whenever modified data have to be transmitted to the server. When a data block with the given `nodeName` already exists, not all data need to be transmitted but only the modified ones. In such a case the server will change only these fields that are sent and will leave the other ones untouched. That means that no data can be deleted from the server. The required parameters and their meaning are very similar to the ones of the callback function:

`nodeName` - the unique name of the data block to be transmitted

`ios` - a set of OR-concatenated IO-flags that will be sent using the following parameter values, the fields of this value have to be filled with flags of type `OAPC_xxx_IOx` and have to correspond with the number, position and type of data provided in the values array

`values` - an array of pointers to the data which have to be sent to the server

`auth` - reserved for future use and has to be set to NULL at the moment

This function returns `OAPC_OK` when the request could be transmitted successfully or an error code otherwise. In case of an error the socket connection is closed by this function and has to be re-established by the main application. When an error `OAPC_ERROR_NO_DATA_AVAILABLE` is returned the parameters provided to this function have been invalid: in this case at least one IO was specified within the parameter "`ios`" where no data have been set to the corresponding "`values`" index.

`int oapc_ispace_disconnect(void *handle)`

This function closes the connection to the Interlock Server that is identified by the parameter handle and releases all related resources including the callback function and the handle resource. So before the next connection attempt a new handle has to be obtained by calling `oapc_ispace_get_instance()` and the callback has to be installed again.

7 Shared Library liboapcwx

The ControlRoom applications use one general, shared library “liboapcwx”. This library contains several common functionalities that are ported for all platforms the ControlRoom software is available for. Comparing to “liboapc” here these functions are included that somehow depend on the wxWidgets toolkit. They have been put into an own library to keep target system installations as small as possible: wherever plug-ins or other components of the software package are used that do not make use of wxWidgets-functions it is not necessary to install this package just because liboapc needs that dependency. So this library is used by the main application and by only very few and very specific plug-ins. Following the functions of this library are described so that it can be used by other plug-ins and completely other applications too.

The sources of the complete library liboapcwx are contained within the SDK.

All the functions described below are contained within the general header file “liboapc.h” that is provided within the SDK too. Beside of that header file also “oapc_libio.h” might be necessary. This header file is responsible for all plug-in related definitions but also contains some data that are used by “liboapc.h”.

7.1 Canvas-Functions of liboapcwx

This function group is useful for HMI plug-ins, they can be used to access special properties of the canvas object that is handed over from the main application to the plug-in.

`bool oapc_canvas_get_readonly(wxPanel *canvas)`

This function can be used to retrieve the read-only state of the `canvas` that is given by the main application and used by the plug-in to implement the own HMI element. The function returns `true` when the canvas is set to read-only, it returns `false` for a read and writeable state.

`void oapc_canvas_set_readonly(wxPanel *canvas,bool readonly)`

The read-only state of a canvas can be set by this function, the object from the main application has to be handed over with parameter `canvas`, the second parameter `readonly` specifies the state to be set. Is has to be `true` when the canvas has to be set to read only, false otherwise.

`bool oapc_canvas_get_enabled(wxPanel *canvas)`

This function can be used to check if the `canvas` that is given by the main application and used by the plug-in to implement the own HMI element is enabled or not. The function returns `true` when the canvas is set to read-only, it returns `false` for a read and writeable state. Please Note: for compatibility reasons the function `wxPanel::IsEnabled()` has not to be used!

`void oapc_canvas_set_enabled(wxPanel *canvas,bool enable)`

The enabled state of a canvas can be set by this function, the object from the main application has to be handed over with parameter `canvas`, the second parameter `enable` specify the state to be set. Is has to be `true` when the canvas has to be enabled, false otherwise. Please note: for compatibility reasons the function `wxPanel::Enable()` has not to be used!

`void oapc_canvas_release_data(wxPanel *canvas)`

This is an internal function, it is not allowed to use it from within a HMI plug-in!

7.2 Unicode-Functions of liboapcwx

This set of functions can be used to convert between different string formats including plain ASCII strings and several Unicode representations. Conversion is always done between a `char` data type (which may contain strings with more than 8 bit per character) and the `wxString` data type of wxWidgets:

`void oapc_unicode_charToStringUTF16BE(const char *c,int len,wxString *result)`

Converts a UTF-16 string in big-endian byte order to a `wxString`. The input Unicode string is handed over in pointer `c`, `len` specifies the length of that string. The converted string is returned via pointer `result`.

`void oapc_unicode_charToStringUTF8(const char *c,int len,wxString *result)`

Converts a UTF-8 string to a `wxString`. The input Unicode string is handed over in pointer `c`, `len` specifies the length of that string. The converted string is returned via pointer `result`.

`void oapc_unicode_charToStringASCII(const char *c,int len,wxString *result)`

Converts a plain 8 bit ASCII-string of locally used codepage to a `wxString`. The input ASCII string is handed over in pointer `c`, `len` specifies the length of that string. The converted string is returned via pointer `result`.

`void oapc_unicode_stringToCharUTF16BE(wxString s,char *c,int len)`

Converts the `wxString` contained in `s` into a UTF-16 string in big-endian byteorder. The result is written into the memory where `c` points to while the maximum length of `len` bytes is not exceeded.

`void oapc_unicode_stringToCharUTF8(wxString s,char *c,int len)`

Converts the `wxString` contained in `s` into UTF-8 format. The result is written into the memory where `c` points to while the maximum length of `len` bytes is not exceeded.

`void oapc_unicode_stringToCharASCII(wxString s,char *c,int len)`

Converts the `wxString` contained in `s` into plain 8 bit ASCII-format. For this conversion the local codepage is used. The result is written into the memory where `c` points to while the maximum length of `len` bytes is not exceeded.

`void oapc_unicode_utf16BEToASCII(char *utf,int len)`

This function performs an in-place conversion from a UTF-16 formatted string in big-endian byte-order into a plain 8 bit ASCII-string. For conversion the local codepage is used. The input string is expected in memory area where `utf` points to, parameter `len` specifies the length of that memory area (in bytes). After conversion the result can be found in `utf` too using a zero-terminated ASCII-string which will have exactly halve size than specified by `len`.

7.3 Path-Functions of liboapcwx

This function group handles all kinds of file and path name related functionalities and therefore acts on level of disk file system pathes and names.

void oapc_path_split(wxString *path,wxString *dir,wxString *file,wxString extension)

This function can be used to split a full path to its directory part and its file name. Optionally the file name can be checked for an extension and – if it does not own it – the extension is added to it. The full path is handed over via parameter `path`. When the extracted file has to be checked, parameter `extension` needs to be a non-empty string. If it does not own such an extension it is added automatically. The – optionally – extended file and the directory of it are returned via parameters `file` and `dir`.

8 Shared Library libsmartfactory

Some of the software components of the OpenAPC package make use of a library which encapsulates a bunch of Smart Factory / Industry 4.0 related functions.

The sources of the complete library libsmartfactory are contained within the SDK.

All the functions described below are contained within the general header file “libsmartfactory.h” that is provided by the SDK. This library depends on liboapc and liboapcwx, so both must be available in an environment where libsmartfactory shall be used. The error codes used by libsmartfactory are the ones defined in liboapc.h, both make use of the same error handling return codes.

The external library libsmartfactory is divided into different functional sections that are described below.

8.1 Hermes related functions of libsmartfactory

The shared library libsmartfactory implements the industry communication protocol The Hermes Standard as described by the Hermes consortium: https://the-hermes-standard.info.

The related functionalities can be found as source within the SDK in file “libsmartfactoryHermesFcts.cpp” and contain several functionalities that are related to connected machines within an automated production line as defined by the Hermes consortium (for details, the exact specification, usage and handling of this protocol please refer the webpage mentioned above). A typical Hermes session consists of the following sequence:

1. `sf_hermes_create_instance()` - create a new Hermes communication instance and allocate all required resources
2. define callback functions via `sf_hermes_set_prev_state_callback()` and `sf_hermes_set_next_state_callback()` to let the own application be notified whenever the previous or the next machine sends some information the own one has to react on
3. `sf_hermes_open_connections()` - open TCP/IP interfaces which allow the current instance to communicate with the previous and the next machine in the production line
4. repeatedly: communicate with the next machine via commands `sf_hermes_set_next_...()` functions and with the previous machine via `sf_hermes_set_prev_...()` functions according to the own machines production state; in case the own machine is located at the beginning of a production line and there is no previous machine providing any incoming data for every new product `sf_hermes_wx_set_product_identifier()` has to be called
5. `sf_hermes_delete_instance()` - at end of overall operation (e.g. on shutdown of own machine) close all connections and release all resources

The logic and states of the Hermes protocol have to be implemented in point 4), for details please refer to the original specification of The Hermes Standard.

Some of the functions provided by this library exists in two variants. When they start with `sf_hermes_wx_` they expect some wxWidgets-specific data types (typically a wxString). When their name starts with `sf_hermes_` but without the `wx_`-part, all parameters are plain C or C++ data types which do not depend on any other third party frameworks.

Following Hermes-related functions are provided by this library:

```
void *sf_hermes_wx_create_instance(const wxString *machineIdentifier,
     const enum hermes_level level)
```

```
void *sf_hermes_create_instance(const char *machineIdentifier,
     const enum hermes_level level)
```

Creates a new Hermes communication instance, the returned instance resource has to be used with all following `sf_hermes_`-function calls and needs to be released by calling `sf_hermes_delete_instance()` at the end. The parameter `machineIdentifier` has to contain a unique string which identifies the own machine. Via `level` the Hermes compatibility level has to be specified, here `eHERMES_1_0` is the minimum which is supported by all interfaces. For a detailed description which compatibility levels contain which functionalities please refer to the official Hermes specification.

When the returned value is NULL, an error occurred and no new communication instance could be created.

`int sf_hermes_delete_instance(void* instance)`

Removes an existing Hermes instance, closes all related connections and releases all related resources; after this functions has been called the instance handle is invalid and can't be used for any further calls to `sf_hermes_`-functions. Here parameter `instance` is the instance resource which has to be released.

This function returns OAPC_OK or an OAPC_ERROR_-code in case execution failed

```
int sf_hermes_open_connections(void *instance,
       const unsigned short prevMachinePort, const char* prevMachineIP,
       const unsigned short nextMachinePort,
       const char remoteConfigEnabled)
```

Open new Hermes connections according to the given parameters, this includes both, a previous and a next machine (if available). Here `instance` is a pointer to the resources of the current communication instance, `prevMachinePort` is the port number for the client socket connection to the previous machine and `prevMachineIP` an IP in style xxx.xxx.xxx.xxx for the client socket connection to the previous machine. `nextMachinePort` is the port number of the server socket the next machine has to connect with, here the IP is always 0.0.0.0, means all incoming connections are accepted. When parameter `remoteConfigEnabled` is set to 1, the function to change the local configuration is enabled: the related server socket for incoming connection is created and modifications of the local Hermes parameters are accepted.

This function returns OAPC_OK or an OAPC_ERROR_-code in case execution failed

```
int sf_hermes_wx_set_machine_identifier(void* instance,
       const wxString *machineIdentifier)
```

Set an identifier for the local machine which is used in several data structures to uniquely specify this local machine. Here `instance` is a pointer to the resources of the current communication instance, `machineIdentifier` is the unique identification string for the own machine.

This function returns OAPC_OK or an OAPC_ERROR_-code in case execution failed

```
int sf_hermes_wx_set_product_identifier(void* instance,
       const wxString *productIdentifier)
```

Set a new product identifier, this value is used only in case no previous machine exists which already provides product identification data in the provided working piece (aka “board”) structure. When the own machine has a previous one, all values set here are ignored and the product identification data provided by the previous machine are used for all further operations. Here `instance` is a pointer to the resources of the current communication instance, `productIdentifier` is the unique identification string for the current working piece which has to be set newly whenever the product changes.

This function returns OAPC_OK or an OAPC_ERROR_-code in case execution failed

```
int sf_hermes_set_prev_state_callback(void* instance,
      const hermes_prev_state_callback fct, void *data)
```

Specify a callback function which is used whenever the state of the previous machine changes. Here `instance` is a pointer to the resources of the current communication instance, `fct` is a pointer to the callback function and `data` is a pointer to some custom data which are handed over to the callback function whenever it is called. The callback function itself is defined as follows:

```
typedef void(*hermes_prev_state_callback)(const enum hermes_prev_state newState,
const bool error, const void* custData)
```

- `newState` – is the new state of the previous machine and defined in `enum hermes_prev_state`
- `error` – specifies if the previous machine signalled an error with the current state
- `custData` – are the custom data as handed over by parameter data of the callback initialisation function

This function returns OAPC_OK or an OAPC_ERROR_-code in case execution failed

```
int sf_hermes_set_next_state_callback(void* instance,
      const hermes_next_state_callback fct, void *data)
```

Specify a callback function which is used whenever the state of the next machine changes. Here `instance` is a pointer to the resources of the current communication instance, `fct` is a pointer to the callback function and `data` is a pointer to some custom data which are handed over to the callback function whenever it is called. The callback function itself is defined as follows:

```
typedef void(*hermes_next_state_callback)(const enum hermes_next_state newState,
const bool error, const void* custData)
```

- `newState` – is the new state of the next machine and defined in `enum hermes_next_state`
- `error` – specifies if the next machine signalled an error with the current state
- `custData` – are the custom data as handed over by parameter data of the callback initialisation function

This function returns OAPC_OK or an OAPC_ERROR_-code in case execution failed

```
int sf_hermes_wx_set_notification_callback(void* instance,
      const hermes_wx_notification_callback fct, void *data)
```

Specify a callback function which is used whenever a notification arrives at one of the possible connections (previous machine, next machine or configuration socket). Here `instance` is a pointer to the resources of the current communication instance, `fct` is a pointer to the callback function and `data` is a pointer to some custom data which are handed over to the callback function whenever it is called. The callback function itself is defined as follows:

```
typedef void(*hermes_wx_notification_callback)(const long notificationCode,
const long severity, const wxString *description, const wxString *src, const
void* custData)
```

- `notificationCode` – the type-code of the notification according to the Hermes specification
- `severity` – the severity of the notification according to the Hermes specification
- `description` – the notification text as submitted by one of the remote connections
- `src` – a string identifying the source of the notification

- `custData` – are the custom data as handed over by parameter data of the callback initialisation function

This function returns OAPC_OK or an OAPC_ERROR_-code in case execution failed

`int sf_hermes_set_log_callback(void* instance,`
`const hermes_log_callback fct, void *data)`

Specify a callback function which is used whenever an event happens which is interesting for logging purposes. Here `instance` is a pointer to the resources of the current communication instance, `fct` is a pointer to the callback function and `data` is a pointer to some custom data which are handed over to the callback function whenever it is called. The callback function itself is defined as follows:

`typedef void(*hermes_log_callback)(const char *logTxt, const void* custData)`

- togTxt – the logging text as generated by the library and which can be used in own logs
- `custData` – are the custom data as handed over by parameter data of the callback initialisation function

This function returns OAPC_OK or an OAPC_ERROR_-code in case execution failed

`enum hermes_prev_state sf_hermes_get_prev_state(void* instance)`

Retrieve the current state of the previous machine. Here `instance` is a pointer to the resources of the current communication instance.

This function returns OAPC_OK or an OAPC_ERROR_-code in case execution failed

`enum hermes_next_state sf_hermes_get_next_state(void* instance)`

Retrieve the current state of the next machine. Here `instance` is a pointer to the resources of the current communication instance.

This function returns OAPC_OK or an OAPC_ERROR_-code in case execution failed

`int sf_hermes_set_next_BoardAvailable(void* instance)`

Signal the next machine a working piece (aka “board”) is available. Here `instance` is a pointer to the resources of the current communication instance.

This function returns OAPC_OK or an OAPC_ERROR_-code in case execution failed

`int sf_hermes_set_next_RevokeBoardAvailable(void* instance)`

Signal the next machine a working piece (aka “board”) is no longer available. Here `instance` is a pointer to the resources of the current communication instance.

This function returns OAPC_OK or an OAPC_ERROR_-code in case execution failed

`int sf_hermes_set_next_TransportFinished(void* instance)`

Signal the next machine a transport was finished. Here `instance` is a pointer to the resources of the current communication instance.

This function returns OAPC_OK or an OAPC_ERROR_-code in case execution failed

`int sf_hermes_set_prev_StartTransport(void* instance)`

Signal the previous machine to start transport. Here `instance` is a pointer to the resources of the current communication instance.

This function returns OAPC_OK or an OAPC_ERROR_-code in case execution failed

`int sf_hermes_set_prev_StopTransport(void* instance)`

Signal the previous machine to stop the transport. Here `instance` is a pointer to the resources of the current communication instance.

This function returns OAPC_OK or an OAPC_ERROR_-code in case execution failed

`int sf_hermes_set_prev_MachineReady(void* instance)`

Signal the previous machine that the local machine is ready to accept a working piece (aka “board”). Here `instance` is a pointer to the resources of the current communication instance.

This function returns OAPC_OK or an OAPC_ERROR_-code in case execution failed

`int sf_hermes_set_prev_RevokeMachineReady(void* instance)`

Signal the previous machine that the local machine is no longer ready to accept a working piece (aka “board”). Here `instance` is a pointer to the resources of the current communication instance.

This function returns OAPC_OK or an OAPC_ERROR_-code in case execution failed

```
int sf_hermes_wx_get_curr_board_info(void* instance, wxString *BoardId,
      wxString *ProductTypeId)
```

Get information from the working piece (aka “board” according to the Hermes naming conventions) which currently has arrived; this function has to be called **after** `sf_hermes_set_prev_StopTransport()`, elsewhere the returned data are undefined.

In parameter `BoardId` a unique identifier of the current working piece is returned, `ProductTypeId` uniquely identifies the product which is currently processed by the machine. Both data are provided by the previous machine, are transferred to the next machine on further processing and can be retrieved by calling this function.

This function returns OAPC_OK or an OAPC_ERROR_-code in case execution failed

8.2 Smart Interface related functions of libsmartfactory

The shared library libsmartfactory also provides a generic information interface which provides production-related data. These data can be used by external applications to check the current production state, to react on errors which occur on the machine, to implement extended traceability-related functionalities and much more. The provided data are given in XML format as described in manual from https://openapc.com/download/manual.pdf in section “Smart Interface”.

The related functionalities can be found as source within the SDK in file “libsmartfactoryInterfaceFcts.cpp”. A typical usage scenario of the Smart Interface functions looks like this:

1. `sf_if_create_instance()` - create a new Smart Interface communication instance and allocate all required resources
2. `sf_if_open_connections()` - open a TCP/IP socket where external applications can connect with in order to receive data
3. repeatedly: provide production and state information by calling the different sf_if_set_...() and sf_if_wx_set_...() functions dependent on current production state and dependent on data available from the machine
4. `sf_if_delete_instance()` - at end of overall operation (e.g. on shutdown of own machine) close all connections and release all resources

Some of the functions provided by this library exists in two variants. When they start with `sf_if_wx_` they expect some wxWidgets-specific data types (typically a wxString). When their name starts with `sf_if_` but without the `wx_`-part, all parameters are plain C or C++ data types which do not depend on any other third party frameworks.

Following Smart Interface related functions are provided by this library:

`void *sf_if_wx_create_instance(const wxString *machineIdentifier)`

`void *sf_if_create_instance(const char *machineIdentifier)`

Creates a new Smart Interface communication instance, the returned instance resource has to be used with all following `sf_if_`-function calls and needs to be released by calling `sf_if_delete_instance()` at the end. The parameter `machineIdentifier` has to contain a unique string which identifies the own machine.

When the returned value is NULL, an error occurred and no new communication instance could be created.

`int sf_if_delete_instance(void* instance)`

Removes an existing Smart Interface instance, closes all related connections and releases all related resources; after this functions has been called the instance handle is invalid and can't be used for any further calls to `sf_if_`-functions. Here parameter `instance` is the instance resource which has to be released.

This function returns OAPC_OK or an OAPC_ERROR_-code in case execution failed

`int sf_if_open_connections(void* instance)`

Open a new Smart Interface server socket at the local machines IP and using port number 11355. After this function returned with OAPC_OK, external applications are able to connect to the local Smart Interface and to receive state information.

This function returns OAPC_OK or an OAPC_ERROR_-code in case execution failed

`int sf_if_wx_set_machine_identifier(void* instance, const wxString *machineIdentifier)`

Specify a (new) identifier for the local machine. This identifier should be unique for a production line or factory and is provided with every information message in order to identify the source of the message.

This function returns OAPC_OK or an OAPC_ERROR_-code in case execution failed

`int sf_if_wx_set_product_identifier(void* instance, const wxString *productIdentifier)`

Set a new product identifier which specifies to which product the current production data belong to. Here `instance` is a pointer to the resources of the current communication instance, `productIdentifier` is the unique identification string for the current working piece which has to be set newly whenever the product changes.

This function returns OAPC_OK or an OAPC_ERROR_-code in case execution failed

`int sf_if_wx_set_error(void* instance, const wxString *errorText);`

Set an error text which contains detailed information about an error which occurred and is still pending. When parameter `errorText` is a non-empty string, the internal state is set to `eERROR` and this error message is provided to all connected clients via the Smart Interface. When `errorText` is an empty string, the error state is reset to the previous state which was active before the error occurred.

When this command is called, a corresponding state-message is sent to all connected clients.

This function returns OAPC_OK or an OAPC_ERROR_-code in case execution failed

`int sf_if_set_state(void* instance, const enum SmartState state)`

Set a state information which represents the current production state. Here parameter `state` can have one of the following values:

- eIDLE – the machine is currently in idle state and not producing anything
- eREADY – the machine is ready for production but waiting for some (external) event to start a new working/marking cycle
- eMARKING – the machine is currently producing/marking
- eERROR – the machine is in error state and neither producing anything nor able to start a new production cycle
- ePAUSED – the machine is producing/marking, the current cycle is not yet finished but temporarily halted by an (external) event, production will continue after this event is released

When this command is called, the corresponding message is sent to all connected clients.

This function returns OAPC_OK or an OAPC_ERROR_-code in case execution failed

`int sf_if_wx_set_hermes_notification(void* instance, const long notificationCode, const long severity, const wxString *description, const wxString *src)`

This function is intended to be used together with the Hermes interface: it provides a Hermes notification via the Smart Interface so that connected clients also get detailed information from Hermes connections. The parameters given here correspond directly to the ones from a Hermes notification message, so for details please refer to description of callback `hermes_wx_notification_callback` above.

When this command is called, the corresponding message is sent to all connected clients.

This function returns OAPC_OK or an OAPC_ERROR_-code in case execution failed

`int sf_if_wx_set_process_info_data(void* instance, const unsigned int idx, const double value, const wxString *name, const wxString *unit)`

Provides additional process information data. Here up to four different and separated process parameters can be given via parameter `idx` which is allowed to be in range 0..3. Parameter `value` is the (measured) value of the related process parameter, `name` specifies the name of it and `unit` the measurement unit of this value.

When this command is called, the corresponding message is sent to all connected clients.

This function returns OAPC_OK or an OAPC_ERROR_-code in case execution failed

`int sf_if_wx_append_trace_data(void* instance, const wxString *name, const wxString *data)`

Stores trace-data from current production cycle to send them at the end together with a parts-message (please refer to function `sf_if_set_parts()` below for details). Here `name` is the name of the element which has to be traced (and which is the same over all production cycles of the same product) and `data` are some data related to this element (and which may change on every production cycle depending on the capabilities of the element which is traced here).

Different to `sf_if_wx_set_hermes_trace_info()` this function can be called several times for each production cycle and each call will result in one separate trace-entry in related Smart Interface message.

This function returns OAPC_OK or an OAPC_ERROR_-code in case execution failed

`int sf_if_wx_set_hermes_trace_info(void* instance, const wxString *boardID, const wxString *productID)`

Stores production-cycle specific trace-data to send them at the end together with a parts-message (please refer to function `sf_if_set_parts()` below for details). Here `boardID` is a unique identifier for the currently processed working piece and `productID` an identifier for the product. This is a function which is related to the Hermes protocol where only an identifier for the whole product is provided but not for separate elements of this product (please also refer to `sf_if_wx_append_trace_data()`). So this function corresponds to the data which can be retrieved by calling function `sf_hermes_wx_get_curr_board_info()`.

This function returns OAPC_OK or an OAPC_ERROR_-code in case execution failed

`int sf_if_set_parts(void* instance, const unsigned int parts)`

Specifies the number of `parts` which already have been produced.

When this command is called, the corresponding message is sent to all connected clients.

This function returns OAPC_OK or an OAPC_ERROR_-code in case execution failed

`int sf_if_set_slices(void* instance, const unsigned int maxSlices, const unsigned int currSlice)`

This function can be used to provide additional progress information during a production cycle. Typically it is used when several slices are processed, here `maxSlices` specifies the total number of slices to be produced/marked for the current working piece and `currSlice` specifies which of these slices is currently in progress.

When this command is called, the corresponding message is sent to all connected clients.

This function returns OAPC_OK or an OAPC_ERROR_-code in case execution failed

Index

0

0xBBGGRR 26, 48

1

1810 59

A

accuracy 46
addOAPCIServerListener() 64
addOAPCListener() 59
atof() 72f.

B

BITMAP_FLAG_MARK_BIDIRECTIONAL 40
BITMAP_FLAG_MARK_FROM_LAST_LINE 40
BITMAP_FLAG_MARK_WO_LINE_INCR 40
BoardAvailable 90
brate 69

C

CHAR 63
checkbox 49
close() 59, 63
CMD/Value-Pair to Digital 57
colourbutton 48
com.oapc.iface 62
com.oapc.iserver 57
commandReceived() 58
connectionValid() 64
ControlRoom Interface 56

D

databits 69
default 46ff.
dialogue 44f.
DIGI 63
Digital to CMD/Value-Pair 56
dirselect 50
disabled 46ff.
disableon 46ff.
DLL 4
dualpanel 44f.

E

ECHO_DIGI 57
ECHO_NUM 57
ECHO_STR 57
eERROR 93
eIDLE 93
eMARKING 93
enableon 46ff.
ePAUSED 93
eREADY 93
Example1.apcp 57
EXIT 57

F

face....49
facename....49
ffilter....50
fileload....49
filesave....50
float....47
FLOW....8
flow control....5
Flow Editor....5, 21
flowcontrol....69
flowimage....44f.
fontbutton....49

G
general....44f.
GetClientSize()....26

H
handle....69
HANDLE....69
Help-Panel....51
helppanel....44f., 51
Hermes....87
hermes_log_callback....90
hermes_next_state_callback....89
hermes_prev_state_callback....89
hermes_wx_notification_callback....89, 93
HMI Editor....21
htonl()....6
htons()....6

I
IL....5
in....52
Industry 4.0....87
int....69
integer....46
Interlock Server....61
Interlock Server Connection....61
IOCTRL_ANALOGUE_10_1....41
IOCTRL_ANALOGUE_10_2....41
IOCTRL_ANALOGUE_10_3....41
IOCTRL_ANALOGUE_10_4....41
IOCTRL_ANALOGUE_10_5....41
IOCTRL_ANALOGUE_10_6....41
IOCTRL_ANALOGUE_12_1....41
IOCTRL_ANALOGUE_12_2....41
IOCTRL_ANALOGUE_12_3....41
IOCTRL_ANALOGUE_12_4....41
IOCTRL_ANALOGUE_16....41
IOCTRL_ANALOGUE_16_0....41
IOCTRL_ANALOGUE_16_1....41
IOCTRL_ANALOGUE_16_2....41
IOCTRL_ANALOGUE_8_1....41
IOCTRL_DIGITAL_16_1....41
IOCTRL_DIGITAL_16_2....41
IOCTRL_DIGITAL_32....41
IOCTRL_DIGITAL_8_1....41
IOCTRL_DIGITAL_8_2....41
IOCTRL_LASERPORT_8_1....40f.
IOCTRL_LASERPORT_8_1_BITS....40f.

IOCTRL_LASERPORT_8_2........41
IOCTRL_LASERPORT_8_2_BITS........41
IOCTRL_SERIAL_DATA........41
IOCTRL_SERIAL_DATA_LENGTH........41
IServerData........62ff.
IServerData()........65

J
Java........5, 57, 62

L
lib_oapc_iface_callback........78
liboapc........5, 60, 66f., 87
liboapc.h........5, 67, 84
liboapcwx........84, 87
libsmartfactory........87
libsmartfactory.h........87
LUA........5

M
MachineReady........91
max........46f.
MAX_NODENAME_LENGTH........63
MAX_NUM_IOS........63
MAX_TEXT_LEN........63
min........46f.
MSG_NOSIGNAL........67

N
name........46
Network Client........56f.
ntohl()........6
ntohs()........6
NUM........63

O
OAPC_ACCEPTS_IO_CALLBACK........7, 11, 15
OAPC_ACCEPTS_MOUSECLICKS........23
OAPC_ACCEPTS_MOUSEDRAGS........23
OAPC_ACCEPTS_MOUSEMOVES........23
OAPC_ACCEPTS_PLAIN_CONFIGURATION........7
OAPC_ACCEPTS_SEND_DATA_MODE........8
OAPC_ACCEPTS_SEND_NAMED_DATA_MODE........8
OAPC_ACCEPTS_WRITE_DATA_MODE........8, 10f.
OAPC_BIN_HEAD_CURR_VERSION........30
OAPC_BIN_IO0........9
OAPC_BIN_IO7........9
OAPC_BIN_IOx........14f.
oapc_bin_struct_bitmap........39
oapc_bin_struct_ctrl........36
oapc_bin_struct_ioctrl........40
oapc_bin_struct_laserctrl........38
oapc_bin_struct_motionctrl........38
oapc_bin_struct_pos_corr........42
oapc_bin_struct_scanheadinfo........41
oapc_bin_struct_vec3d........36f.
OAPC_BIN_SUBTYPE_IMAGE_BW1........31, 74
OAPC_BIN_SUBTYPE_IMAGE_GREY8........31
OAPC_BIN_SUBTYPE_IMAGE_RGB24........31
OAPC_BIN_SUBTYPE_STRUCT........33ff.
OAPC_BIN_SUBTYPE_STRUCT_AXISSTATE........34

OAPC_BIN_SUBTYPE_STRUCT_BITMAP....39
OAPC_BIN_SUBTYPE_STRUCT_BRAKEOUTPUT....34
OAPC_BIN_SUBTYPE_STRUCT_CTRL....31, 36f.
OAPC_BIN_SUBTYPE_STRUCT_CTRLEND....31, 36f.
OAPC_BIN_SUBTYPE_STRUCT_DELAY....32
OAPC_BIN_SUBTYPE_STRUCT_DYNGEOMEND....34
OAPC_BIN_SUBTYPE_STRUCT_DYNGEOMSTART....34, 43
OAPC_BIN_SUBTYPE_STRUCT_FASTOUTPUTCTRL....35
OAPC_BIN_SUBTYPE_STRUCT_HALT....35
OAPC_BIN_SUBTYPE_STRUCT_INPUTCTRL....40
OAPC_BIN_SUBTYPE_STRUCT_LASERCTRL....32, 38
OAPC_BIN_SUBTYPE_STRUCT_MOTIONCTRL....32, 38
OAPC_BIN_SUBTYPE_STRUCT_OUTPUTCTRL....32, 40
OAPC_BIN_SUBTYPE_STRUCT_OUTPUTRESP....32
OAPC_BIN_SUBTYPE_STRUCT_POS_CORR....42
OAPC_BIN_SUBTYPE_STRUCT_POWERDOWN....35
OAPC_BIN_SUBTYPE_STRUCT_RESETTIMER....34
OAPC_BIN_SUBTYPE_STRUCT_SCANHEADINFO....41
OAPC_BIN_SUBTYPE_STRUCT_SCANHEADNFO....33
OAPC_BIN_SUBTYPE_STRUCT_SLICEEND....35
OAPC_BIN_SUBTYPE_STRUCT_SLICESTART....35
OAPC_BIN_SUBTYPE_STRUCT_STARTOUTPUT....34
OAPC_BIN_SUBTYPE_STRUCT_STOPAXIS....35
OAPC_BIN_SUBTYPE_STRUCT_STOPOUTPUT....33f.
OAPC_BIN_SUBTYPE_STRUCT_SYNC....32, 35
OAPC_BIN_SUBTYPE_STRUCT_WAITINPUTCTRL....40
OAPC_BIN_SUBTYPE_STRUCT_WAITTRIG....32
OAPC_BIN_SUBTYPE_TEXT_PLAIN....31
OAPC_BIN_SUBTYPE_ttt_sss....30
OAPC_BIN_TYPE_CUSTOM....30
OAPC_BIN_TYPE_IMAGE....31, 39, 74
OAPC_BIN_TYPE_STRUCT....31ff., 36ff.
OAPC_BIN_TYPE_STRUCT_BITMAP....33
OAPC_BIN_TYPE_STRUCT_ENDMARKREADY....33
OAPC_BIN_TYPE_STRUCT_INPUTCTRL....32
OAPC_BIN_TYPE_STRUCT_JOBEND....33
OAPC_BIN_TYPE_STRUCT_JOBSTART....33
OAPC_BIN_TYPE_STRUCT_MARKREADY....33
OAPC_BIN_TYPE_TEXT....31
OAPC_BIN_TYPE_ttt....30
OAPC_BIN_TYPE_ttt_CUSTOM....30
OAPC_BIN_UNIT_ttt_uuu....30
oapc_canvas_get_enabled()....84
oapc_canvas_get_readonly()....84
oapc_canvas_release_data()....84
oapc_canvas_set_enabled()....84
oapc_canvas_set_readonly()....84
OAPC_CHAR_IO0....9
OAPC_CHAR_IO7....9
OAPC_CHAR_IOx....14
OAPC_CMD_GET_VALUE....62
OAPC_CMD_SET_VALUE....82
OAPC_CMDERR_DOESNT_EXISTS....62, 82
OAPC_COMPRESS_NONE....30
OAPC_CONFIG_IMAGE_CAPTURE_VERSION....18
OAPC_CONFIG_INFO_VERSION....17
OAPC_CONFIG_LASER_CONTROLLER_VERSION....19
OAPC_CONFIG_MOTION_CONTROLLER_VERSION....18
OAPC_CONFIG_PCONTROL_FLAG_BRIGHTNESS_STYLE....21
OAPC_CONFIG_PCONTROL_FLAG_PRESSURE_STYLE....20
OAPC_CONFIG_PCONTROL_FLAG_TEMPERATURE_STYLE....20

OAPC_CONFIG_PCONTROL_VERSION........20
OAPC_CONFIG_TYPE_IMAGECAPTURE........18
OAPC_CONFIG_TYPE_LASERCONTROLLER........18
OAPC_CONFIG_TYPE_MOTIONCONTROLLER........17
OAPC_CONFIG_TYPE_PCONTROL........19
OAPC_CONFIG_TYPE_ZSHIFTER........19
OAPC_CONFIG_ZSHIFTER_VERSION........19
oapc_create_instance()........10
oapc_create_instance2()........6, 10, 22
oapc_cycletime........11
oapc_delete_instance()........6, 10, 12, 22
OAPC_DIGI_IO0........9
OAPC_DIGI_IO7........9
OAPC_DIGI_IOx........12f.
OAPC_DIGI_IOy........65
oapc_dlib_load()........77
OAPC_ERROR........29
OAPC_ERROR_AUTHENTICATION........27
OAPC_ERROR_CONNECTION........27, 63
OAPC_ERROR_CONVERSION_ERROR........28
OAPC_ERROR_CREATE_FILE_FAILED........28
OAPC_ERROR_CUSTOM........16, 29
OAPC_ERROR_DEVICE........27
OAPC_ERROR_INVALID_INPUT........28
OAPC_ERROR_LIBRARY_MISSING........29
OAPC_ERROR_LICENSE........29
OAPC_ERROR_NO_DATA_AVAILABLE........13ff., 27, 63
OAPC_ERROR_NO_MEMORY........27
OAPC_ERROR_NO_SUCH_IO........27
OAPC_ERROR_NOT_SUPPORTED........27
OAPC_ERROR_OPEN_FILE_FAILED........28
OAPC_ERROR_OUT_OF_RANGE........29
OAPC_ERROR_PROTOCOL........28
OAPC_ERROR_READ_FILE_FAILED........28
OAPC_ERROR_RECV_DATA........28
OAPC_ERROR_RESOURCE........27, 63
OAPC_ERROR_SEND_DATA........28
OAPC_ERROR_STILL_IN_PROGRESS........28
OAPC_ERROR_STOPPED........29
OAPC_ERROR_WRITE_FILE_FAILED........28
OAPC_ERROR_xxx........71
oapc_exit........12
OAPC_EXT_HMI_EXPORTS........22
OAPC_FLOWCAT_CALC........8
OAPC_FLOWCAT_CONVERSION........8
OAPC_FLOWCAT_DATA........8
OAPC_FLOWCAT_FLOW........8
OAPC_FLOWCAT_IO........8
OAPC_FLOWCAT_LASER........9
OAPC_FLOWCAT_LOGIC........8
OAPC_FLOWCAT_MOTION........8
oapc_get_bin_value()........15
oapc_get_capabilities()........7, 10, 23
oapc_get_char_value()........14f.
oapc_get_colours()........25
oapc_get_config_data........10
oapc_get_config_info_data()........16f.
oapc_get_defsize()........24f.
oapc_get_digi_value()........12
oapc_get_error_message()........16, 29
oapc_get_hmi_config_data()........24, 26

oapc_get_input_flags()....9
oapc_get_name()....9
oapc_get_no_ui_flags()....24
oapc_get_num_value()....13
oapc_get_numminmax()....25
oapc_get_output_flags()....9
oapc_get_save_data()....6, 11, 22
OAPC_HAS_INPUTS....7, 9, 12ff.
OAPC_HAS_LOG_TYPE_CHAR....7
OAPC_HAS_LOG_TYPE_DIGI....7
OAPC_HAS_LOG_TYPE_FLOATNUM....7
OAPC_HAS_LOG_TYPE_INTNUM....7
OAPC_HAS_OUTPUTS....7, 9, 12ff.
OAPC_HAS_STANDARD_FLOW_CONFIGURATION....23
OAPC_HAS_XML_CONFIGURATION....7, 10
OAPC_HMI_NO_POS....24
OAPC_HMI_NO_SIZE....24
OAPC_HMI_NO_UI_BG....24
OAPC_HMI_NO_UI_DISABLED....24
OAPC_HMI_NO_UI_FG....24
OAPC_HMI_NO_UI_FONT....24
OAPC_HMI_NO_UI_LAYOUT....24
OAPC_HMI_NO_UI_MINMAX....24
OAPC_HMI_NO_UI_RO....24
OAPC_HMI_NO_UI_TEXT....24
OAPC_HMI_SIZE_FIXED_ASPECT....24
OAPC_HMICAT_CONTROL....23
OAPC_HMICAT_DISPLAY....23
OAPC_HMICAT_STATIC....23
oapc_iface_....60
oapc_iface_exit()....78, 80
oapc_iface_init()....78f.
oapc_iface_send_bin()....80
oapc_iface_send_char()....79
oapc_iface_send_digi()....79
oapc_iface_send_num()....79
oapc_iface_set_authentication()....78f.
oapc_iface_set_recv_callback()....78
OAPC_IFACE_TYPE_BIN....78
OAPC_IFACE_TYPE_CHAR....78
OAPC_IFACE_TYPE_DIGI....78
OAPC_IFACE_TYPE_NUM....78
OAPC_IFACE_TYPE_STATE_CONNECTION_CLOSED....78f.
OAPC_IFACE_TYPE_STATE_CONNECTION_ERROR....78f.
OAPC_IFACE_TYPE_STATE_NEW_CONNECTION....78
oapc_init....12
oapc_init()....16
OAPC_INSTANCE_MINIMUM_INIT....10, 16
OAPC_INSTANCE_OPERATION....10
OAPC_INSTANCE_SEND_DATA....8
OAPC_INSTANCE_SEND_NAMED_DATA....8
OAPC_INSTANCE_SIMULATION....10
OAPC_INSTANCE_WRITE_DATA....8, 11
oapc_io_callback()....16
OAPC_IS_DEPRECATED....8
oapc_iserver_....66
oapc_ispace_connect()....80, 82
oapc_ispace_disconnect()....80, 83
oapc_ispace_get_handle()....81
oapc_ispace_get_instance()....80f., 83
oapc_ispace_request_all_data()....81f.

oapc_ispace_request_data()....81f.
oapc_ispace_set_data()....81f.
OAPC_LC_HAS_FREQ....19
OAPC_LC_HAS_LASEROFF....19
OAPC_LC_HAS_LASERON....19
OAPC_LC_HAS_POWER....19
oapc_libio.h....5, 7, 22, 67, 84
oapc_mouseevent()....26
OAPC_NUM_IO0....9
OAPC_NUM_IO7....9
OAPC_NUM_IOx....13
OAPC_NUM_IOy....65
OAPC_OK....27, 63
oapc_paint()....26
oapc_path_split()....86
OAPC_PCONTROL_MAX_CUST_PARAMS....20
oapc_read_pvalue()....10, 16
oapc_release_bin_data()....15
oapc_serial_port_close()....71
oapc_serial_port_open()....69
oapc_serial_recv_data()....71
oapc_serial_recv()....71
oapc_serial_send_data()....71
oapc_serial_send()....71
oapc_serial_usb_port_open()....71
oapc_set_char_value()....13f.
oapc_set_config_data(....46
oapc_set_config_data()....7f., 11, 48, 51
oapc_set_digi_value()....12
oapc_set_io_callback()....8, 15
oapc_set_load_data()....6, 22
oapc_set_loaded_data()....11
oapc_set_num_value()....13
oapc_set_numminmax()....25
oapc_set_recv_callback()....80f.
oapc_tcp_accept_connection()....68
oapc_tcp_closesocket()....68
oapc_tcp_connect_to()....67
oapc_tcp_listen_on_port()....67f.
oapc_tcp_recv()....67f.
oapc_tcp_send()....67
oapc_tcp_set_blocking()....67f.
oapc_thread_create()....72
oapc_thread_mutex_create()....76
oapc_thread_mutex_lock()....74, 76
oapc_thread_mutex_unlock()....74, 76
oapc_thread_set_prio()....72
oapc_thread_signal_create()....76
oapc_thread_signal_release()....77
oapc_thread_signal_send()....76f.
oapc_thread_signal_wait()....76f.
oapc_thread_sleep()....72
oapc_thread_timer_start()....76
oapc_unicode_charToStringASCII()....85
oapc_unicode_charToStringUTF16BE()....85
oapc_unicode_charToStringUTF8()....85
oapc_unicode_stringToCharASCII()....85
oapc_unicode_stringToCharUTF16BE()....85
oapc_unicode_stringToCharUTF8()....85
oapc_unicode_utf16BEToASCII()....85
OAPC_USERPRIVI_DISABLE....8

OAPC_USERPRIVI_HIDE........8
oapc_util_alloc_bin_data()........31, 73
oapc_util_atof_dot()........72
oapc_util_atof()........73
oapc_util_block_to_dbl()........72, 80
oapc_util_check_maskbit()........74
oapc_util_colour2gray()........72
oapc_util_create_thread()........72, 75ff.
oapc_util_dbl_to_block()........72, 80
oapc_util_diff_time()........73
oapc_util_get_time()........73
oapc_util_get_timeofday()........73
oapc_util_rb_alloc()........74
oapc_util_rb_empty()........75
oapc_util_rb_front()........74f.
oapc_util_rb_full()........75
oapc_util_rb_pop()........75
oapc_util_rb_push()........74
oapc_util_rb_release()........74
oapc_util_release_bin_data()........73
oapc_util_release_time()........73
oapc_util_thread_set_prio()........72, 75
oapc_util_thread_sleep()........72, 76
oapc_util_to_unicode()........72
oapc_write_data_path........8, 11
OAPC_xxx_IOy........63
oapc-config........44f.
OAPCInterface........57f.
OAPCInterface()........59
OAPCIServer........62f.
OAPCIServer()........63
OAPCIServerListener........62
OAPCListener........57f.
option........48
out........52

P
param........45
Param-Panel........51
parampanel........16, 44f.
parity........69
pointsize........49
port........59

R
removeOAPCIServerListener()........64
removeOAPCListener()........59
REPLY_DIGI........57
REPLY_NUM........57
REPLY_STR........57
requestAllData()........64
requestData()........64
RevokeBoardAvailable........90
RevokeMachineReady........91

S
SCANHEAD_ERRORFLAG_GALVOTEMP_X........42
SCANHEAD_ERRORFLAG_GALVOTEMP_Y........42
SCANHEAD_ERRORFLAG_GALVOTEMP_Z........42
SCANHEAD_ERRORFLAG_POSITION_X........42
SCANHEAD_ERRORFLAG_POSITION_Y........42

SCANHEAD_ERRORFLAG_POSITION_Z........42
SCANHEAD_ERRORFLAG_VOLTAGE_X........42
SCANHEAD_ERRORFLAG_VOLTAGE_Y........42
SCANHEAD_ERRORFLAG_VOLTAGE_Z........42
sendCommand()........59
setAuthentication()........59
setData()........64
setValue()........64
sf_hermes_create_instance()........87
sf_hermes_delete_instance()........87f.
sf_hermes_get_prev_state()........90
sf_hermes_open_connections()........87f.
sf_hermes_set_log_callback()........90
sf_hermes_set_next_BoardAvailable()........90
sf_hermes_set_next_RevokeBoardAvailable()........90
sf_hermes_set_next_state_callback()........87, 89
sf_hermes_set_next_TransportFinished()........90
sf_hermes_set_prev_MachineReady()........91
sf_hermes_set_prev_RevokeMachineReady()........91
sf_hermes_set_prev_StartTransport()........91
sf_hermes_set_prev_state_callback()........87, 89
sf_hermes_set_prev_StopTransport()........91
sf_hermes_wx_create_instance()........87
sf_hermes_wx_get_curr_board_info()........91, 94
sf_hermes_wx_set_machine_identifier()........88
sf_hermes_wx_set_notification_callback()........89
sf_hermes_wx_set_product_identifier()........87f.
sf_if_create_instance()........92
sf_if_delete_instance()........92
sf_if_open_connections()........92
sf_if_set_parts()........94
sf_if_set_slices()........94
sf_if_set_state()........93
sf_if_wx_append_trace_data()........94
sf_if_wx_create_instance()........92
sf_if_wx_set_error()........93
sf_if_wx_set_hermes_notification()........93
sf_if_wx_set_hermes_trace_info()........94
sf_if_wx_set_machine_identifier()........92
sf_if_wx_set_product_identifier()........93
sf_if_wx_set_trace_info()........94
shared library........4
Smart Factory........87
Smart Interface........91f.
so........4
SocketEntry........57, 60
StartTransport........91
state........46ff.
STATE_CONNECTION_CLOSED........58
STATE_CONNECTION_ERROR........58
STATE_DATA_RECEIVED........58
STATE_NEW_CONNECTION........58
stateChanged()........58
stdpanel........44f.
stopbits........69
StopTransport........91
string........46
struct config_image_capture........18
struct config_info........17
struct config_laser_controller........18
struct config_motion_controller........17

struct config_pcontrol........19
struct config_zshifter........19
struct oapc_bin_head........30, 80
struct oapc_char_value_block........80
struct oapc_digi_value_block........80
struct oapc_num_value_block........80
style........49

T
text........46
The Hermes Standard........87
TOOLPARAM_DESCFLAG_IS_SLICE........37
TOOLPARAM_VALIDFLAG_FREQ........36
TOOLPARAM_VALIDFLAG_OFFDELAY........36
TOOLPARAM_VALIDFLAG_OFFSPEED........36
TOOLPARAM_VALIDFLAG_ON........36
TOOLPARAM_VALIDFLAG_ONDELAY........37
TOOLPARAM_VALIDFLAG_ONSPEED........36
TOOLPARAM_VALIDFLAG_PARAM1........37
TOOLPARAM_VALIDFLAG_PARAM2........37
TOOLPARAM_VALIDFLAG_PARAM3........37
TOOLPARAM_VALIDFLAG_PARAM4........37
TOOLPARAM_VALIDFLAG_PARAM5........37
TOOLPARAM_VALIDFLAG_PARAM6........37
TOOLPARAM_VALIDFLAG_POWER........36
TOOLPARAM_VALIDFLAG_VARIABLE_PARAM3........37
TOOLPARAM_VALIDFLAG_Z........36
TransportFinished........90
type........46ff.

U
unit........46ff.

V
value........48

W
weight........49
wxPanel::Enable()........84
wxPanel::IsEnabled()........84
wxString........85
wxWidgets........84

........29