



ADI/DAL for PFC Firmware 11 Patch2 02.08.35(11)

Application Device Interface(ADI)
Device Abstraction Layer(DAL)

© 2018 by WAGO Kontakttechnik GmbH & Co. KG
All rights reserved.

WAGO Kontakttechnik GmbH & Co. KG

Hansastraße 27
D-32423 Minden

Phone: +49 (0) 571/8 87 – 0

E-Mail: info@wago.com

Web: <http://www.wago.com>

Technical Support

Phone: +49 (0) 571/8 87 – 44 555

E-Mail: support@wago.com

Ver.:	Date	Author	Note
1.0.0	12.02.16	u010640	Init: FW8
1.0.1	20.07.17	u010640	Update: FW10
1.1.0	20.08.18	u010640	Behavior change: KBUS “unconfigured” in FW11-Patch 2:

Every conceivable measure has been taken to ensure the accuracy and completeness of this documentation. However, as errors can never be fully excluded, we always appreciate any information or suggestions for improving the documentation.

We wish to point out that the software and hardware terms as well as the trademarks of companies used and/or mentioned in the present manual are generally protected by trademark or patent.

Table of Contents

1	Important Notes	4
1.1	Legal Principles	4
1.1.1	Subject to Changes	4
1.1.2	Copyright	4
1.1.3	Personnel Qualification	4
1.1.4	Intended Use	4
1.2	Scope of Validity	5
1.3	Number Notation	5
1.4	Font Conventions	5
2	Description.....	6
2.1	ADI, DAL and SDI	7
3	Application Device Interface(ADI).....	8
3.1	ADI-Reference	15
4	Stack device interface (SDI).....	21
4.1	SDI Reference	22
5	Device specific functions “CANopen”	25
5.1	Reference “CANopen”	26
6	Device specific functions “KBUS”	32
6.1	Reference KBus	35
7	Device specific functions “PROFIBUS-Slave”	45
7.1	Reference PROFIBUS-Slave	46
8	Attachment A – Licences.....	52

1 Important Notes

To ensure quick installation and start-up of the units, we strongly recommend that the following information and explanations are carefully read and adhered to.

1.1 Legal Principles

1.1.1 Subject to Changes

WAGO Kontakttechnik GmbH & Co. KG reserves the right to provide for any alterations or modifications that serve to increase the efficiency of technical progress. WAGO Kontakttechnik GmbH & Co. KG owns all rights arising from the granting of patents or from the legal protection of utility patents. Third-party products are always mentioned without any reference to patent rights. Thus, the existence of such rights cannot be excluded.

1.1.2 Copyright

This Manual, including all figures and illustrations, is copyright-protected. Any further use of this Manual by third parties that violate pertinent copyright provisions is prohibited. Reproduction, translation, electronic and phototechnical filing/archiving (e.g., photocopying) as well as any amendments require the written consent of WAGO Kontakttechnik GmbH & Co. KG, Minden, Germany. Non-observance will involve the right to assert damage claims.

1.1.3 Personnel Qualification

The use of the product detailed in this document is exclusively geared to specialists having qualifications in PLC programming, electrical specialists or persons instructed by electrical specialists who are also familiar with the valid standards. WAGO Kontakttechnik GmbH & Co. KG declines any liability resulting from improper action and damage to WAGO products and third party products due to non-observance of the information contained in this document.

1.1.4 Intended Use

For each individual application, the components are supplied from the factory with a dedicated hardware and software configuration. Modifications are only admitted within the framework of the possibilities documented in this document. All other changes to the hardware and/or software and the non-conforming use of the components entail the exclusion of liability on part of WAGO Kontakttechnik GmbH & Co. KG.

Please direct any requirements pertaining to a modified and/or new hardware or software configuration directly to WAGO Kontakttechnik GmbH & Co. KG.

1.2 Scope of Validity

This application note is based on the stated hardware and software of the specific manufacturer as well as the associated documentation. This application note is therefore only valid for the described installation.

New hardware and software versions may need to be handled differently.

Please note the detailed description in the specific manuals.

1.3 Number Notation

Table 1: Number Notation

Number code	Example	Note
Decimal	100	Normal notation
Hexadecimal	0x64	C notation
Binary	'100' '0110.0100'	In quotation marks, nibble separated with dots (.)

1.4 Font Conventions

Table 2: Font Conventions

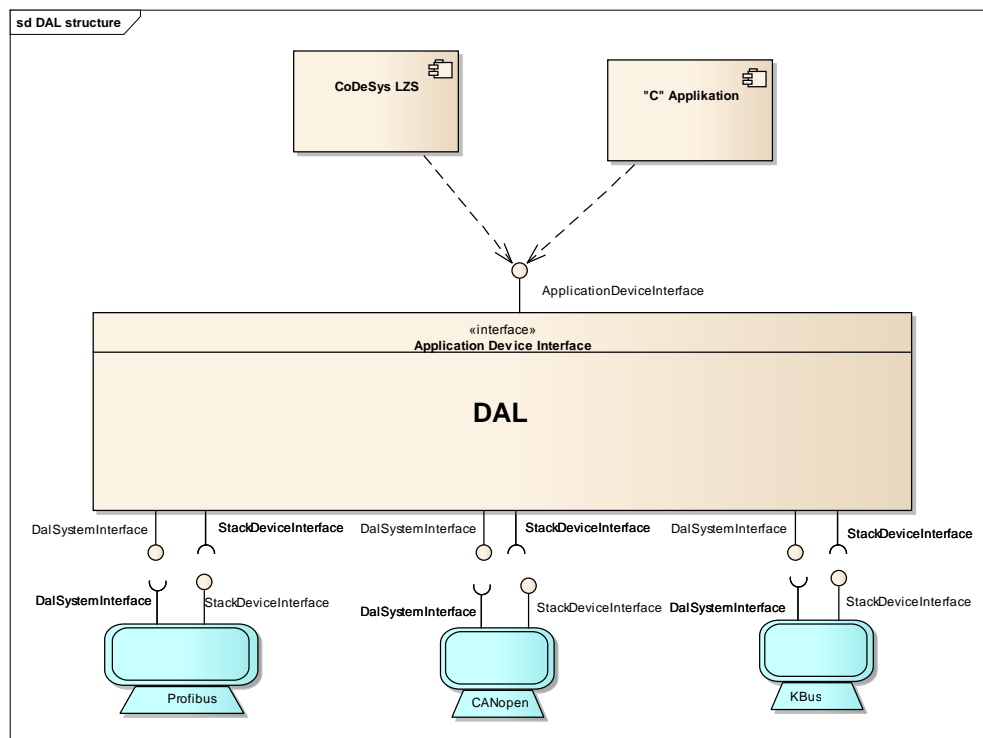
Font type	Indicates
<i>italic</i>	Names of paths and data files are marked in italic-type. e.g.: <i>C:\Programme\WAGO-I/O-CHECK</i>
Menu	Menu items are marked in bold letters. e.g.: Save
>	A greater-than sign between two names means the selection of a menu item from a menu. e.g.: File > New
Input	Designation of input or optional fields are marked in bold letters, e.g.: Start of measurement range
“Value”	Input or selective values are marked in inverted commas. e.g.: Enter the value “4 mA” under Start of measurement range .
[Button]	Pushbuttons in dialog boxes are marked with bold letters in square brackets. e.g.: [Input]
[Key]	Keys are marked with bold letters in square brackets. e.g.: [F5]

2 Description

This application note provides an introduction into WAGO's Device-Abstraction-Layer(DAL) and the usage of the Application-Device-Interface(ADI) within your C/C++ program.

Major targets of the Device-Abstraction-Layer(DAL) and the Application-Device-Interface(ADI) are:

- Offer unified access for different fieldbus devices.
- Offer an open API without copyright or license restrictions.
- Simplify CoDeSys-IO-Driver interface, only required features are ported.



The ADI/DAL currently supports the abstraction and unified access for following standards:

- KBus
- PROFIBUS-DP
- CANopen
- ModbusTCP and ModbusRTU

The ADI Interface implemented as a “single user” API. The API only able to be used by one application. Typical this application are the PLC-Runtime(CoDeSys2 or e!COCKPIT), or your C/C++ application, but not both at the same time! Therefor you have to kill the linux process “plclinux_rt” or “codesys3” on the PLC before running an ADI application like ‘candemo’ or ‘kbusdemo’.

Major tasks of DAL are:

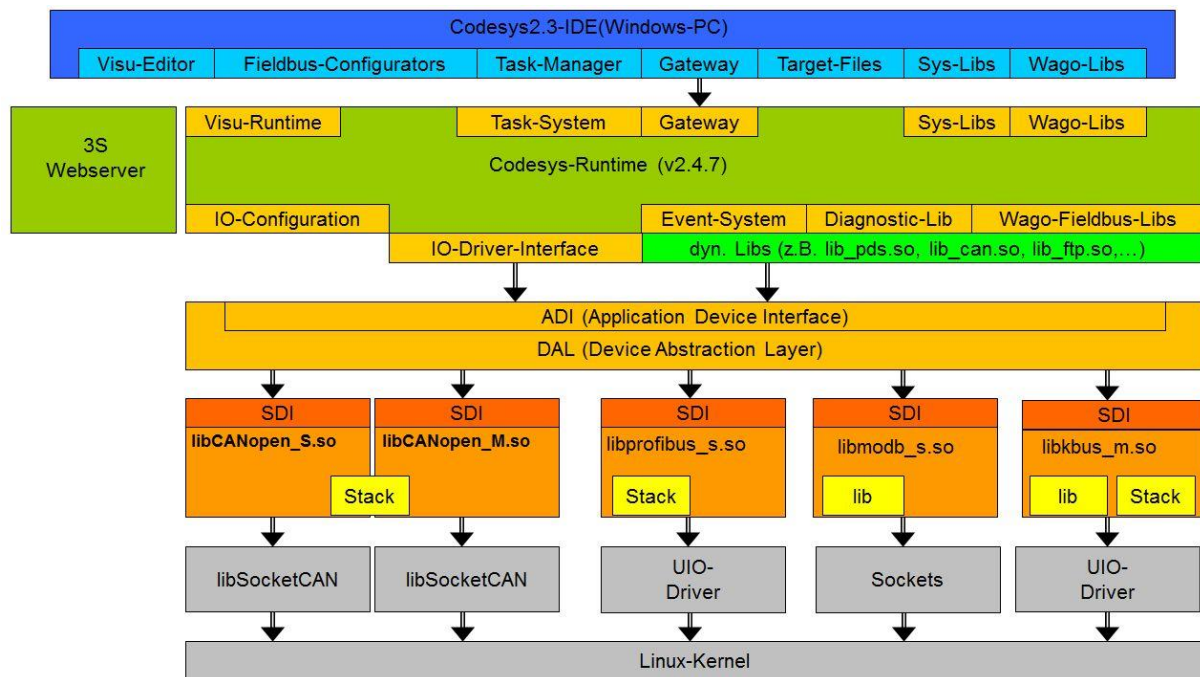
- administration of io driver libraries (list, open, close)
- forwarding function calls to selective io drivers
- broadcasting function to all io drivers, such as “applicationStateChanged”.

2.1 ADI, DAL and SDI

Before ADI/DAL, all devices (e.g. local IO-Modules or remote devices communicating via a fieldbus) communicated directly with the PLC-Runtime, which made it hard up to impossible to use devices within other applications.

The ADI/DAL acts as an administration unit between devices and any application that wants to use devices.

In case a PLC-Runtime (RTS) like CoDeSys2.3 or e!Cockpit is used, the original RTS is extended by an “IO-Driver-Interface” that converts 3S-Software specific requests into ADI/DAL-Requests and vice versa.



A device will never directly use an RTS-Feature, it has to use the ADI/DAL that decorates or transforms the requests.

The ADI defines a mandatory set of functions every device has to provide.

ADI also provides a proxy mechanism to support device specific functionalities.

Device libraries realized as shared objects (.so) use the default search path “/usr/lib/dal”.

During the registration process, each device introduces its implementation to the SDI.

Devices are addressed by a unique “DeviceID”.

3 Application Device Interface (ADI)

The ADI defines a mandatory set of common functions every device has to provide, realized as function table.

Header and Includes:

```
// include file for ADI/DAL
#include <dal/adi_application_interface.h>

typedef struct stApplicationDeviceInterface tApplicationDeviceInterface,
    *IApplication;

// Get the Application interface struct
tApplicationDeviceInterface *adi_GetApplicationInterface(void);

// DAL Application interface defined by following structure
struct stApplicationDeviceInterface
{
    // System functions
    int32_t (*Init)();
    int32_t (*Exit)();
    int32_t (*ScanDevices)();
    int32_t (*GetDeviceList)();
    int32_t (*OpenDevice)();
    int32_t (*CloseDevice)();
    bool (*IsDeviceOpen)();
    int32_t (*GetDeviceName)();
    int32_t (*GetDeviceFlags)();
    int32_t (*GetIoSizes)();
    // Configure devices(Out of scope of this document)
    int32_t (*ConfigureSubdevice)();
    int32_t (*ConfigureDevice)();
    // Process data access
    int32_t (*WriteStart)();
    int32_t (*WriteBit)();
    int32_t (*WriteBool)();
    int32_t (*WriteBytes)();
    int32_t (*WriteEnd)();
    int32_t (*ReadStart)();
    int32_t (*ReadBit)();
    int32_t (*ReadBool)();
    int32_t (*ReadBytes)();
    int32_t (*ReadEnd)();
    // Diagnostic
    int32_t (*DiagnoseGetDeviceState)();
    int32_t (*DiagnoseGetSubdeviceState)();
    int32_t (*GetLastError)();
    // Hooks
    int32_t (*RegisterEventHandler)();
    int32_t (*UnregisterEventHandler)();
    // Devices operating mode
    int32_t (*ApplicationStateChanged)();
    // Proxy for device specific stuff
    int32_t (*CallDeviceSpecificFunction)();
    int32_t (*VCallDeviceSpecificFunction)();
    // Watchdog, when device driven by external hardware/software
    int32_t (*WatchdogSetTime)();
    int32_t (*WatchdogStart)();
    int32_t (*WatchdogStop)();
    int32_t (*WatchdogTrigger)();
};
```

Usage of ADI is limited to one process. It may be the PLC runtime or your application but not both at same time.

Don't forget to free resources after use:

- use `adi->Exit()`, to release occupied resources by `adi->Init()`.
- use `adi->CloseDevice()`, to release occupied resources of `adi->OpenDevice()`.

Using system functions for commissioning ADI/DAL

```
#include <dal/adi_application_interface.h>

int main(int argc, char **argv)
{
    tDeviceInfo deviceList[10];
    size_t nrDevicesFound, i;
    tApplicationDeviceInterface * adi;

    adi = adi_GetApplicationInterface();

    adi->Init();

    adi->ScanDevices();

    adi->GetDeviceList(sizeof(deviceList), deviceList, &nrDevicesFound);

    for(i=0; i<nrDevicesFound; ++i)
        adi->OpenDevice(deviceList[i].DeviceId);

    // Do something...

    for(i=0; i<nrDevicesFound; ++i)
        adi->CloseDevice(deviceList[i].DeviceId);

    adi->Exit();
}
```

ADI functions typically return one of following values:

- DAL_SUCCESS //Successful executed
- DAL_FAILURE //Use GetLastError() for more details
- DAL_NOTUSED //Not implemented

The Function `adi_GetApplicationInterface` returns a pointer to the one and only instance of struct `tApplicationDeviceInterface` (SINGELTON).

The Function `adi->Init` sets up the internal data structure, don't forget call `adi->Exit` when the work is done.

The Function `adi->ScanDevices` parses the name of all device libraries (shared objects) in folder “/usr/lib/dal”.

This search process can be restarted at any time, maybe when a device library was added, updated or deleted.

Deleting a device library makes sense in case you'd like to handle one fieldbus outside of CoDeSys within your own SDI-application.

For questions about the Stack-Device-Interface (SDI) contact support@wago.com.

The Function `adi->GetDeviceList` returns an array of `tDeviceInfo`.

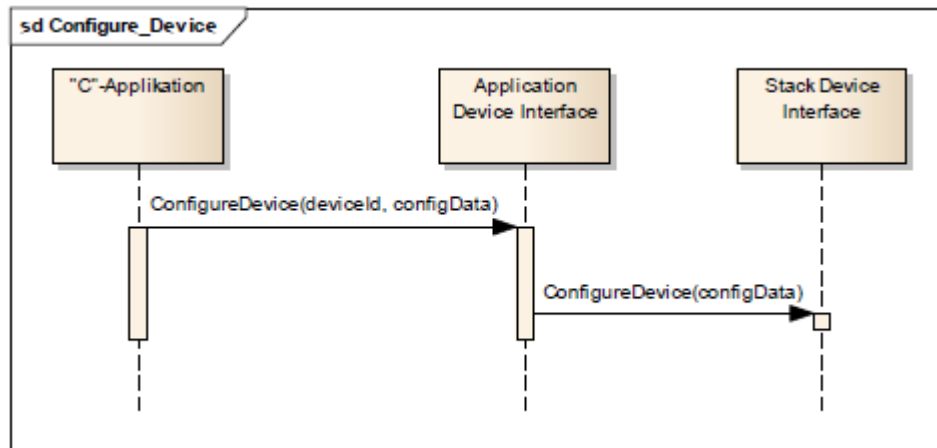
```
typedef struct stDeviceInfo
{
    tDeviceId DeviceId; // Unique "DeviceId", used by all other ADI-functions
    const char* DeviceName; // Name of shared object, without ".so" extension
} tDeviceInfo;
```

Devices are addressed by a unique “DeviceId” assigned by ADI.

Device configuration

Configuration of a device is fieldbus specific, ADI provides just a function to forward binary data to the device.

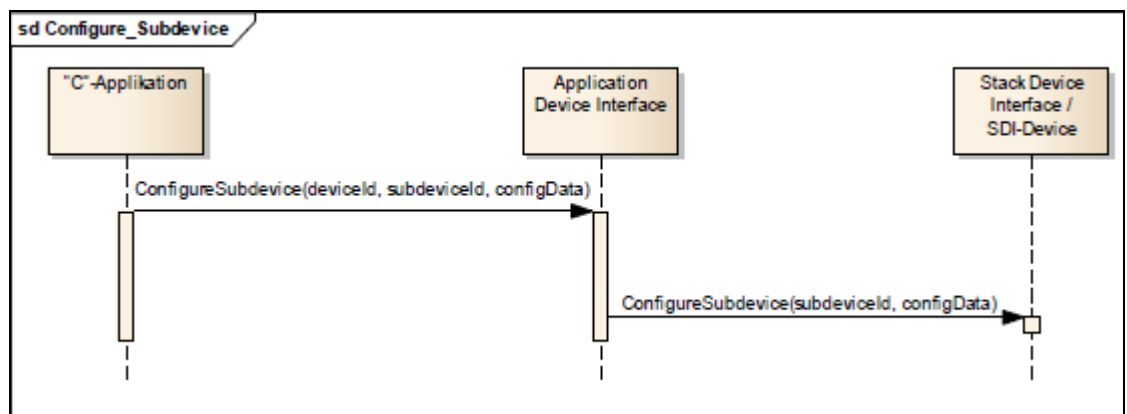
```
adi->ConfigureDevice((tDeviceId) deviceId,
                    (void *)configData);
```



Sub-Device configuration

Configuration of a sub device is fieldbus specific, ADI provides just a function to forward binary data to the sub device.

```
adi->ConfigureSubdevice((tDeviceId) deviceId,
                       (int32_t) subdeviceId,
                       (void *)configData);
```



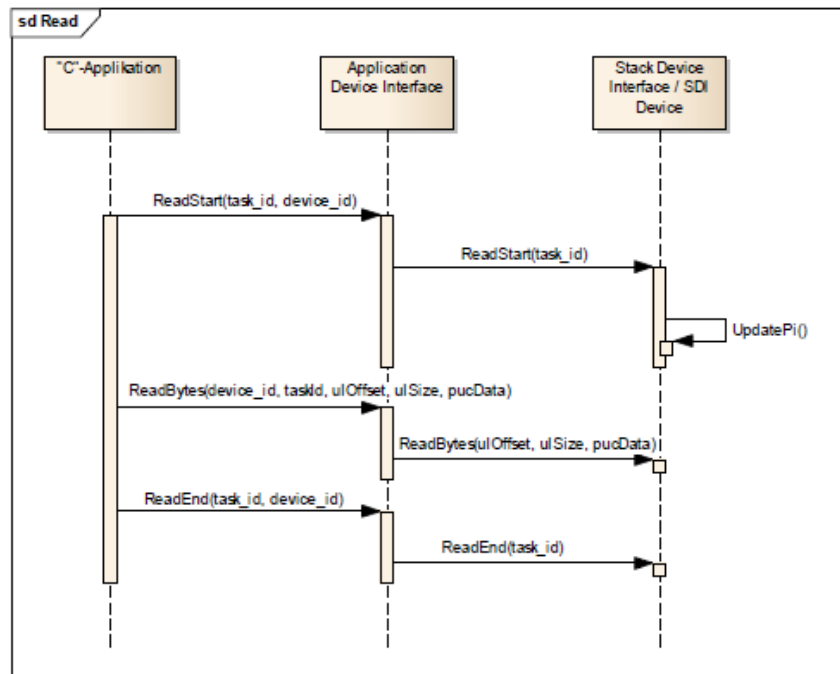
Process data access

Process data images are available for “inputs(PII)” and “outputs(PIO)”. Both process data images provide a linear address room starting with index 0. The size of process images can be queried with function `adi->GetIoSizes`.

```
adi->GetIoSizes((tDeviceId) deviceId,
               (uint32_t*) inputSize,
               (uint32_t*) outputSize);
```

Typical steps are DAL initialization, device registration and device configuration, followed by a change of the “ApplicationState” to enable process data access. Check fieldbus specific sections on additional prerequisites to enable access to process data images.

To keep the input process data image consistent, every read operation has to be encapsulated by `adi->ReadStart` and `adi->ReadEnd` functions.



The number of read operations between function `adi->ReadStart` and `adi->ReadEnd` is not limited.

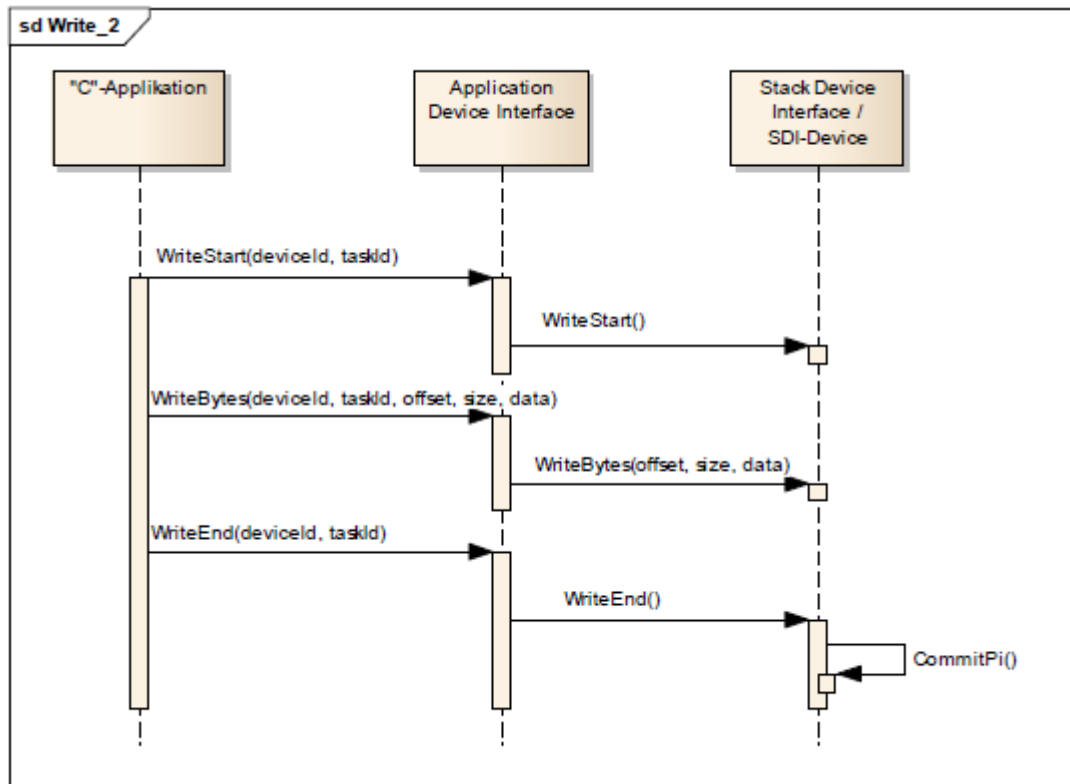
```
adi->ReadStart(deviceId, taskId);
adi->ReadBit(deviceId, taskId, bitOffset, &data);
adi->ReadBool(deviceId, taskId, bitOffset, value);
adi->ReadBytes(deviceId, taskId, offset, size, &data);
adi->ReadEnd(deviceId, taskId);
```

The parameter “taskId” is ignored at the moment, but reserved for further use.

Pseudo code on how the reading of a bit in PII is processed:

```
// Calculate Byte-Offset
const uint32_t byteOffset = bitOffset / 8;
// Assign bitmask who select requested bit
const uint8_t bitMask = 1 << (bitOffset % 8);
// masquerade requeste bit
*buf = PII[byteOffset] & bitMask;
```

To keep the output process data image consistent, every write operation has to be encapsulated by `adi->WriteStart` and `adi->WriteEnd` functions:



The number of write operations between function `adi->WriteStart` and `adi->WriteEnd` is not limited.

```

adi->WriteStart(deviceId, taskId);
adi->WriteBit(deviceId, taskId, bitOffset, &data);
adi->WriteBool(deviceId, taskId, bitOffset, value);
adi->WriteBytes(deviceId, taskId, offset, size, &data);
adi->WriteEnd(deviceId, taskId);
  
```

The parameter “taskId” is ignored at the moment, but reserved for further use.

Pseudo code on how writing of a bit in PIO is processed:

```

// Calculate Byte-Offset
const uint32_t byteOffset = bitOffset / 8;
// Assign bitmask who select requested bit
const uint8_t bitMask = 1 << (bitOffset % 8);
// Create negotiate bitmask
const uint8_t negBitMask = ~bitMask;
// Mascquerade (keep) other bits:
PIO[byteOffset] &= negBitMask;
// Set requested bit
PIO[byteOffset] |= (bitMask & *buf);
  
```

Proxy for device specific stuff

Not all fieldbus specific functionalities can be covered by the DAL-Interface. For situations like this, the ADI provides a proxy function able to call any device specific function by its name together with any number of needed parameters.

```
adi->CallDeviceSpecificFunction((const char *)fnName,
                               (void *)retVal,
                               <additional params>);
```

Each device has to register available ‘function names’. Internally library *libffi* is used to describe and handle functions with unknown signatures at compile time.

Set “application state” or “operating mode”

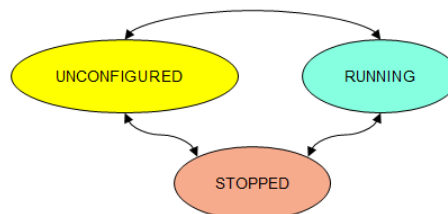
All devices listen for state change events from your application.

```
typedef enum enApplicationState
{
    ApplicationState_BASE = 0,          //!< ApplicationState_BASE
    ApplicationState_Running,          //!< ApplicationState_Running
    ApplicationState_Stopped,          //!< ApplicationState_Stopped
    ApplicationState_Unconfigured,    //!< ApplicationState_Unconfigured
    ApplicationState_TOP,              //!< ApplicationState_TOP
}tApplicationState;
```

Events “fired” with function `adi->ApplicationStateChanged`.

```
adi->ApplicationStateChanged((tApplicationStateChangedEvent) event);
```

The way each device handles a change of state event is fieldbus specific. Check fieldbus specific sections in this document for further information.



Example:

```
tApplicationStateChangedEvent event;
// Preset application state to "Running"
event.State = ApplicationState_Running;
// Raise application state change event
if (adi->ApplicationStateChanged(event) != DAL_SUCCESS)
{
    printf("Set application state to 'Running' failed\n");
}
printf("Set application state to 'Running' \n");
```

Watchdog functions

ADI provides a centralized watchdog feature to inform all devices about the health status of the driving application. This feature becomes important when fieldbuses driven by a separate hardware or independent software stack like PROFIBUS or CANopen are used.

```
adi->WatchdogSetTime((uint32_t) factor,  
                     (uint32_t) timeout_us);  
adi->WatchdogStart(void);  
adi->WatchdogStop(void);  
adi->WatchdogTrigger(void);
```

The PROFIBUS-Slave feature is based on hardware chip “DPC31” from SIEMENS.

The CANopen-feature is realized as independent software stack by WAGO.

If the watchdog is started and (timeout * factor) elapses the fieldbus device changes to the ‘failsave’ state.

3.1 ADI-Reference

```
tApplicationDeviceInterface *adi_GetApplicationInterface(void);
```

```
/**
 * This structure defines the Application interface of the DAL.
 */
struct stApplicationDeviceInterface
{
    /**
     * Opens i.e. initializes the DAL.
     * @return DAL_FAILURE in case another instance is already running.
     *         DAL_SUCCESS on success.
     */
    int32_t (*Init) (void)
                DAL_GNUC_WARN_UNUSED_RESULT;

    /**
     * Main cleanup routine for the DAL.
     * Releases the allocated resources, triggers
     * closing of the devices etc.
     * @note currently this never fails.
     * @return DAL_SUCCESS.
     */
    int32_t (*Exit) (void)
                DAL_GNUC_WARN_UNUSED_RESULT;

    /**
     * ScanDevices() checks all shared objects contained in the device library
     * search path (default path is /usr/lib/dal) and calls their initialisation
     * functions.
     *
     * After a call to ScanDevices() adi->GetDeviceList() returns a list of all
     * found devices along with their error states.
     *
     * @return DAL_SUCCESS if no errors occurred, DAL_FAILURE otherwise
     */
    int32_t (*ScanDevices) (void)
                DAL_GNUC_WARN_UNUSED_RESULT;

    /**
     * @brief Creates a list of the devices managed by the DAL.
     * @param listCapacity The number of devices fitting in the supplied deviceList.
     * @param deviceList This list is filled with up to listCapacity devices.
     * @param deviceNumber The number of devices available.
     * @return DAL_SUCCESS
     */
    int32_t (*GetDeviceList) (size_t listCapacity,
                             tDeviceInfo *deviceList,
                             size_t *deviceNumber)
                DAL_GNUC_WARN_UNUSED_RESULT;

    /**
     * Opens the device.
     * @param deviceId Id of the device.
     * @return Returns 0 on success, -1 otherwise.
     */
    int32_t (*OpenDevice) (tDeviceId deviceId)
                DAL_GNUC_WARN_UNUSED_RESULT;

    /**
     * Closes the device
     * @param deviceId Id of the device.
     * @return Returns 0 on success, -1 otherwise.
     */
    int32_t (*CloseDevice) (tDeviceId deviceId)
                DAL_GNUC_WARN_UNUSED_RESULT;

    /**
     * Checks whether the device is open or not.
     * @param deviceId Id of the device.
     * @return Returns true when device is open, false otherwise.
     */
}
```

```

bool (*IsDeviceOpen) (tDeviceId deviceId)
    DAL_GNUC_WARN_UNUSED_RESULT;

/**
 * Stores the name of the device in the supplied buffer.
 * @param[in] deviceId Id of the device.
 * @param[in] bufferSize Size of the nameBuffer
 * @param[out] nameBuffer Pointer to the buffer to store the name in.
 * @param[out] nameLength The resulting length of the string in nameBuffer.
 * @return Returns DAL_SUCCESS on success, DAL_FAILURE otherwise.
 */
int32_t (*GetDeviceName) (tDeviceId deviceId,
    size_t bufferSize,
    char *nameBuffer,
    size_t *nameLength)
    DAL_GNUC_WARN_UNUSED_RESULT;

/**
 * Returns the devices flags
 * @param deviceId
 * @param flags Pointer to the flags
    #define DRV_CAP_FLAG_IS_PBDEVICE 0x2
    #define DRV_CAP_FLAG_IS_CANDEVICE 0x4
    #define DRV_CAP_FLAG_IS_ASIDEVICE 0x8
    #define DRV_CAP_FLAG_IS_TCPIPDEVICE 0x10
    #define DRV_CAP_FLAG_IS_ARCNETDEVICE 0x20
    #define DRV_CAP_FLAG_IS_IBSDEVICE 0x40
    #define DRV_CAP_FLAG_IS_PBSTANDALONESLAVE 0x82
    #define DRV_CAP_FLAG_IS_COS 0x84
    #define DRV_CAP_FLAG_HAS_RETAINDATA_AREA 0x100
    #define DRV_CAP_FLAG_IS_DEVNETMASTER 0x10000
    #define DRV_CAP_FLAG_IS_DEVNETSLAVE 0x20000
 * @return Returns 0 on success, -1 otherwise.
 */
int32_t (*GetDeviceFlags) (tDeviceId deviceId,
    uint32_t *flags)
    DAL_GNUC_WARN_UNUSED_RESULT;

/**
 * Gets the sizes of the devices input and output.
 * @param deviceId Id of the device.
 * @param inputSize Pointer to where the size of the input will be stored.
 * @param outputSize Pointer to where the size of the output will be stored.
 * @return Returns 0 on success, -1 otherwise.
 */
int32_t (*GetIoSizes) (tDeviceId deviceId,
    uint32_t* inputSize,
    uint32_t* outputSize)
    DAL_GNUC_WARN_UNUSED_RESULT;

/**
 * Configures a sub device.
 * @param deviceId Id of the device.
 * @param subdeviceId Id of the sub device.
 * @param configData The device specific configuration.
 * @return Returns 0 on success, -1 otherwise.
 */
int32_t (*ConfigureSubdevice) (tDeviceId deviceId,
    int32_t subdeviceId,
    void *configData)
    DAL_GNUC_WARN_UNUSED_RESULT;

/**
 * Configures a device.
 * @param deviceId Id of the device.
 * @param configData The device specific configuration.
 * @return Returns 0 on success, -1 otherwise.
 */
int32_t (*ConfigureDevice) (tDeviceId deviceId,
    void *configData)
    DAL_GNUC_WARN_UNUSED_RESULT;

/**
 * Starts the writing of process data.
 * @param deviceId Id of the device.
 * @param taskId The id of the task- (currently not used).

```



```

* @return Returns 0 on success, -1 otherwise.
*/
int32_t (*WriteStart) (tDeviceId deviceId,
                      uint32_t taskId)
                      DAL_GNUC_WARN_UNUSED_RESULT;

/**
* The WriteBit method writes one bit from data to the given offset.
* @param deviceId The device to write the bit to.
* @param taskId The id of the task- (currently not used).

* @param bitOffset The bit offset to write the bit at.
* @param data Pointer from where the bit will be written from.
* @return Returns 0 on success, -1 otherwise.
*/
int32_t (*WriteBit) (tDeviceId deviceId,
                    uint32_t taskId,
                    uint32_t bitOffset,
                    uint8_t* data)
                    DAL_GNUC_WARN_UNUSED_RESULT;

/**
* The WriteBool method writes one bit from data to the given offset.
* @param deviceId The device to write the bit to.
* @param taskId The id of the task- (currently not used).
* @param bitOffset The bit offset to write the bit at.
* @param value The boolean value to be written.
* @return Returns 0 on success, -1 otherwise.
*/
int32_t (*WriteBool) (tDeviceId deviceId,
                     uint32_t taskId,
                     uint32_t bitOffset,
                     bool value)
                     DAL_GNUC_WARN_UNUSED_RESULT;

/**
* Writes size bytes from at the supplied offset to
* the process images of the device.
* @param deviceId Id of the device
* @param taskId The id of the task- (currently not used).
* @param offset Offset where the data will be written at.
* @param size Size of the data to write.
* @param data The data to write.
* @return Returns 0 on success, -1 otherwise
*/
int32_t (*WriteBytes) (tDeviceId deviceId,
                      uint32_t taskId,
                      uint32_t offset,
                      uint32_t size,
                      uint8_t* data)
                      DAL_GNUC_WARN_UNUSED_RESULT;

/**
* Finishes the writing of process data. Usually this is the point
* when the pi is committed to the device.
* @param deviceId The id of the device.
* @param taskId The id of the task- (currently not used).
* @return Returns 0 on success, -1 otherwise
*/
int32_t (*WriteEnd) (tDeviceId deviceId,
                    uint32_t taskId)
                    DAL_GNUC_WARN_UNUSED_RESULT;

/**
* Starts the reading of the pi. Usually this is the point when the pi
* is read out from the device.
* @param deviceId The id of the device.
* @param taskId The id of the task- (currently not used).
* @return Returns 0 on success, -1 otherwise.
*/
int32_t (*ReadStart) (tDeviceId deviceId,
                     uint32_t taskId)
                     DAL_GNUC_WARN_UNUSED_RESULT;

/**
* The ReadBit method reads one bit from the given offset into data.
* @param deviceId The device to read the bit from.

```

```

* @param taskId The id of the task- (currently not used).
* @param bitOffset The bit offset to read the bit at.
* @param data Pointer to where the bit will be read to.
* @return Returns 0 on success, -1 otherwise.
*/
int32_t (*ReadBit) (tDeviceId deviceId,
                   uint32_t taskId,
                   uint32_t bitOffset,
                   uint8_t* data)
    DAL_GNUC_WARN_UNUSED_RESULT;

/**
* The ReadBool method reads one bit from the given offset into value.
* @param deviceId The device to read the bit from.
* @param taskId The id of the task- (currently not used).
* @param bitOffset The bit offset to read the bit at.
* @param data Pointer to where the bit will be read to.
* @return Returns 0 on success, -1 otherwise.
*/
int32_t (*ReadBool) (tDeviceId deviceId,
                    uint32_t taskId,
                    uint32_t bitOffset,
                    bool* value)
    DAL_GNUC_WARN_UNUSED_RESULT;

/**
* Reads size bytes at offset from the devices pi into data.
* @param deviceId Id of the device
* @param taskId The id of the task- (currently not used).
* @param offset Offset at which the data will be read.
* @param size Size of the data to read.
* @param data Pointer to the buffer.
* @return Returns 0 on success, -1 otherwise.
*/
int32_t (*ReadBytes) (tDeviceId deviceId,
                     uint32_t taskId,
                     uint32_t offset,
                     uint32_t size,
                     uint8_t* data)
    DAL_GNUC_WARN_UNUSED_RESULT;

/**
* Finishes the reading of process data.
* @param deviceId The id of the device.
* @param taskId The id of the task- (currently not used).
* @return Returns 0 on success, -1 otherwise.
*/
int32_t (*ReadEnd) (tDeviceId deviceId,
                   uint32_t taskId)
    DAL_GNUC_WARN_UNUSED_RESULT;

/**
* Get the state of the device (corresponds to codesys' bus).
* @param[in] deviceId Id of the device.
* @param[in] bufferSize Size of the supplied buffer.
* @param[out] diagnoseBuffer Buffer where the state is stored in.
* @param[out] busState The state of the bus.
* @return Returns 0 on success, -1 otherwise.
*/
int32_t (*DiagnoseGetDeviceState) (tDeviceId deviceId,
                                   uint32_t bufferSize,
                                   uint8_t *diagnoseBuffer,
                                   tBusState *busState)
    DAL_GNUC_WARN_UNUSED_RESULT;

/**
* Get the state of a subdevice (corresponds to codesys' device).
* @param deviceId Id of the device.
* @param subdeviceId Id of the device to get the state from.
* @param[in] bufferSize Size of the supplied buffer.
* @param[out] diagnoseBuffer Buffer where the state data is stored in.
* @param[out] diagnoseState The state of the diagnose.
* @return Returns 0 on success, -1 otherwise.
*/
int32_t (*DiagnoseGetSubdeviceState) (tDeviceId deviceId,
                                       int32_t subdeviceId,
                                       size_t bufferSize,

```

```

uint8_t* diagnoseBuffer,
tDiagnoseState *diagnoseState)
DAL_GNUC_WARN_UNUSED_RESULT;

/**
 * The GetLastError method gets the last error set by the given device.
 * @param deviceId The device to get the error from.
 * @param error the error.
 * @return Returns the error.
 */
int32_t (*GetLastError) (tDeviceId deviceId,
                        tError *error)
DAL_GNUC_WARN_UNUSED_RESULT;

/**
 * The RegisterEventCallback method registers a callback function,
 * which will get called in case the given device generates an event.
 * @param deviceId The device to register the callback for.
 * @param eventCallback The callback function.
 * @return Returns 0 on success, -1 otherwise.
 * @see URL#generateUrl:fromParams:UnregisterEventCallback
 */
int32_t (*RegisterEventHandler) (tDeviceId deviceId,
                                tEventHandler handler,
                                void *userData)
DAL_GNUC_WARN_UNUSED_RESULT;

/**
 * The UnregisterEventCallback method unregisters a previously
 * registered callback function.
 * @param deviceId The device to unregister the callback for.
 * @param eventCallback The callback to unregister.
 * @return Returns 0 on success, -1 otherwise.
 * @see URL#generateUrl:fromParams:RegisterEventCallback
 */
int32_t (*UnregisterEventHandler) (tDeviceId deviceId,
                                  tEventHandler handler)
DAL_GNUC_WARN_UNUSED_RESULT;

/**
 * Sets the state of the PLC application. Its up to the devices to react accordingly.
 * @param state The state of the application.
 * @return DAL_SUCCESS on success.
 *         DAL_FAILURE when at least one device returned an error.
 */
int32_t (*ApplicationStateChanged) (tApplicationStateChangedEvent event)
DAL_GNUC_WARN_UNUSED_RESULT;

/**
 * @brief Calls a device-specific function implemented by a device library.
 *
 * @param fnName Name of the function to invoke
 * @param retVal Return value
 * @param ... Additional parameters
 * @return DAL_SUCCESS if the function was invoked successfully
 *         DAL_FAILURE on error
 *         DAL_NOTUSED if the function was not found
 */
int32_t (*CallDeviceSpecificFunction) (const char *fnName,
                                       void *retVal,
                                       ...)
DAL_GNUC_WARN_UNUSED_RESULT;

/**
 * @brief Calls a device-specific function implemented by a device library.
 *
 * @param fnName Name of the function to invoke
 * @param retVal Return value
 * @param ... Additional parameters
 * @return DAL_SUCCESS if the function was invoked successfully
 *         DAL_FAILURE on error
 *         DAL_NOTUSED if the function was not found
 */
int32_t (*VCallDeviceSpecificFunction) (const char *fnName,
                                       void *retVal,
                                       va_list args)
DAL_GNUC_WARN_UNUSED_RESULT;

```

```
/**
 * Sets the watchdog time. The time until the watchdog triggers
 * is the multiplication of factor and timeout_us.
 * @param factor The factor specifies how often the timeout_us
 * has to elapse until the watchdog triggers.
 * @param timeout_us The timeout of the watchdog in microseconds.
 * @return Return DAL_SUCCESS on success, DAL_FAILURE otherwise.
 */
int32_t (*WatchdogSetTime)(uint32_t factor,
                           uint32_t timeout_us)
                           DAL_GNUC_WARN_UNUSED_RESULT;

/**
 * Starts the watchdog.
 * @return DAL_SUCCESS on
 */
int32_t (*WatchdogStart)(void)
                           DAL_GNUC_WARN_UNUSED_RESULT;

/**
 * Stops the watchdog.
 * @return Return DAL_SUCCESS on success, DAL_FAILURE otherwise.
 */
int32_t (*WatchdogStop)(void)
                           DAL_GNUC_WARN_UNUSED_RESULT;

/**
 * Triggers the watchdog.
 * @return Return DAL_SUCCESS on success, DAL_FAILURE otherwise.
 */
int32_t (*WatchdogTrigger)(void)
                           DAL_GNUC_WARN_UNUSED_RESULT;
};
```

4 Stack device interface (SDI)

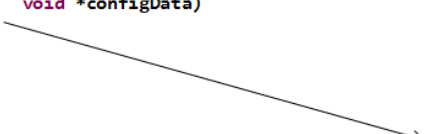
The SDI defines a mandatory set of common functions every device has to provide, realized as function table.

```
/**< The SDI interface. */
struct stStackDeviceInterface
{
    uint32_t Version;
    uint32_t InputByte;
    uint32_t OutputByte;
    tDeviceId DeviceId;
    uint32_t Flags;
    int32_t (*ReadBit)(uint32_t bitOffset,
                      uint8_t *buf);
    int32_t (*ReadBytes)(uint32_t offset,
                        uint32_t size,
                        uint8_t *buf);
    int32_t (*WriteBit)(uint32_t bitOffset,
                      uint8_t *buf);
    int32_t (*WriteBytes)(uint32_t offset,
                        uint32_t size,
                        uint8_t *buf);
    int32_t (*ReadStart)(void);
    int32_t (*ReadEnd)(void);
    int32_t (*WriteStart)(void);
    int32_t (*WriteEnd)(void);
    int32_t (*ConfigureSubdevice)(int32_t subdevice,
                                void *confData);
    int32_t (*ConfigureDevice)(void *confData);
    int32_t (*DiagGetDeviceState)(size_t bufferSize,
                                uint8_t *diagnoseBuffer,
                                tBusState *busState);
    int32_t (*DiagGetSubdeviceState)(void* getState,
                                    uint32_t size,
                                    int32_t subdeviceId,
                                    tDiagnoseState *diagState);

    int32_t (*Close)(void);
    tFnDescr* (*GetDeviceSpecificFnList)(void);
    int32_t (*ApplicationStateChanged)(tApplicationStateChangedEvent event);
    int32_t (*WatchdogSetTime)(uint32_t factor, uint32_t timeout_us);
    int32_t (*WatchdogStart)(void);
    int32_t (*WatchdogStop)(void);
    int32_t (*WatchdogTrigger)(void);
};
```

Most ADI functions can also be found as SDI function without param 'deviceId'.

```
int32_t (*ConfigureSubdevice)(tDeviceId deviceId,
                             int32_t subdeviceId,
                             void *confData)
```



```
int32_t (*ConfigureSubdevice)(int32_t subdevice,
                             void *confData);
```

Every device driver is an SDI instance.

4.1 SDI Reference

```

/**
 * This is the signature of the device open method. The open method has to be named
 * "libraryname underscore open".
 * For example the canopen libs filename is libcopen.so:
 * -> name of the open function: libcopen_open
 * @param sdi This is the interface of the device. The device must fill in its function pointers.
 * @param dsi This is the interface for device to dal communication.
 * @return DAL_SUCCESS on success, DAL_FAILURE otherwise.
 */
typedef int32_t (*device_open) (tStackDeviceInterface* sdi,
                                tDalSystemInterface* dsi);

/**< The SDI interface. */
struct stStackDeviceInterface
{

    /**< The version of the SDI interface.
     * Its up to the device to check it against the \sa DAL_VERSION define
     */
    uint32_t Version;

    /**< Size of the input process image */
    uint32_t InputByte;

    /**< Size of the output process image */
    uint32_t OutputByte;

    /**< The device id is set by the DAL. Don't change.
    .... * The device id is needed for sdi to dal communication and has to be supplied for event generation.
    */
    tDeviceId DeviceId;

    /**< In the flags field the device has to specify its type \sa enum DeviceType. */
    uint32_t Flags;

    /**
     * ReadBit reads one bit at the supplied offset and stores it in buf.
     * The offset is in units of bits.
     * @param[in] bitOffset Bit offset at which to read the bit.
     * @param[out] buf Pointer to where to store the bit.
     * @return DAL_SUCCESS on success, DAL_FAILURE otherwise.
     *
     * @example Lets assume this is the device's pii: { 0x11, 0x44 }
     *      When ReadBit(4, buffer) gets called,
     *      the value in buffer should be 0x10.
     *      ReadBit(0, buffer) => *buffer = 0x01
     *      ReadBit(10, buffer) => *buffer = 0x04
     *      ReadBit(14, buffer) => *buffer = 0x40
     *
     *      @note: Keep in mind that the other bits in buffer
     *      have to be left untouched.
     *
     *      This is what you have to do basically:
     *      1. Get the byte offset.
     *      const uint32_t byteOffset = bitOffset / 8;
     *      2. Get a bit mask which selects the desired bit within the byte
     *      const uint32_t bitMask = 1 << (bitOffset % 8);
     *      3. Mask out the current value of that bit in the buffer.
     *      *buffer &= ~bitMask;
     *      4. Write the bit to the buffer:
     *      *buffer |= pii[byteOffset] & bitMask;
     */
    int32_t (*ReadBit) (uint32_t bitOffset,
                       uint8_t *buf);

    /**
     * Reads size bytes at position offset from the pii into buf.
     * @param[in] offset The offset to start read at.
     * @param[in] size The amount of bytes to read.
     * @param[out] buf
     * @return Returns DAL_SUCCESS on success, DAL_FAILURE otherwise.
     */
    int32_t (*ReadBytes) (uint32_t offset,

```

```

        uint32_t size,
        uint8_t *buf);

/**
 * Writes one bit from buf at bit position offset into the pio.
 * @param[in] bitOffset The offset in units of bits.
 * @param[in] buf Buffer containing the bit to write.
 * @return DAL_SUCCESS on success. DAL_FAILURE otherwise.
 */
int32_t (*WriteBit) (uint32_t bitOffset,
                    uint8_t *buf);

/**
 * Writes size bytes from buf into the pio at position offset.
 * @param offset The offset in bytes.
 * @param size The amount of bytes to write.
 * @param buf Buffer containing the data.
 * @return
 */
int32_t (*WriteBytes) (uint32_t offset,
                     uint32_t size,
                     uint8_t *buf);

/**
 * Starts the reading of process data. ReadStart can be used as
 * indication to update the incoming process data.
 * The device must assure that the pd is NOT updated between
 * ReadStart and ReadEnd.
 * @see ReadEnd
 * @return Returns DAL_SUCCESS on success, DAL_FAILURE otherwise.
 */
int32_t (*ReadStart) (void);

/**
 * Ends the reading of process data.
 * @see ReadStart
 * @return Returns DAL_SUCCESS on success, DAL_FAILURE otherwise.
 */
int32_t (*ReadEnd) (void);

/**
 * Starts the writing of process data.
 * The device has to assure that the pd is NOT transmitted between
 * WriteStart and WriteEnd.
 * @return Returns DAL_SUCCESS on success, DAL_FAILURE otherwise.
 */
int32_t (*WriteStart) (void);

/**
 * Ends the writing of process data.
 * WriteEnd can be used as trigger to transmit the pio over the
 * FB.
 * @return Returns DAL_SUCCESS on success, DAL_FAILURE otherwise.
 */
int32_t (*WriteEnd) (void);

/**
 * This configures a sub device i.e. bus node / slave or what ever.
 * Its only needed by master devices.
 * @param subdevice The address of the sub device.
 * @param confData The configuration data for the sub device.
 * @return Returns DAL_SUCCESS on success, DAL_FAILURE otherwise.
 */
int32_t (*ConfigureSubdevice) (int32_t subdevice,
                              void *confData);

/**
 * This configures a device
 * @param confData The configuration data for the sub device.
 * @return Returns DAL_SUCCESS on success, DAL_FAILURE otherwise.
 */
int32_t (*ConfigureDevice) (void *confData);

/** Get state and diagnostic information of a device
 * @param bufferSize.
 * @param diagnoseBuffer.

```

```

* @param busSate
* @return Returns DAL_SUCCESS on success, DAL_FAILURE otherwise.
*/
int32_t (*DiagGetDeviceState) (size_t bufferSize,
                              uint8_t *diagnoseBuffer,
                              tBusState *busSate);

/**
 * Get state and diagnostic information of a sub device
 * @param getState.
 * @param size.
 * @param subdeviceId
 * @param diagState
 * @return Returns DAL_SUCCESS on success, DAL_FAILURE otherwise.
 */
int32_t (*DiagGetSubdeviceState) (void* getState,
                                  uint32_t size,
                                  int32_t subdeviceId,
                                  tDiagnoseState *diagState);

/**
 * sdi_Close is called to close the device. The device should free any system
 * resources its holding here (threads, memory, mutexes, files, mmmaps, etc..).
 * @return Returns DAL_SUCCESS on success, DAL_FAILURE otherwise.
 */
int32_t (*Close) (void);

/**
 * @brief Call this method to get the list of device-specific functions
 * that are to be exported by the device library.
 *
 * @return array of extRef function descriptions.
 */
tFnDescr* (*GetDeviceSpecificFnList) (void);

/**
 * This function is called when the application changes its status. Its up
 * to the device implementation to react accordingly.
 * @param state The new state of the application.
 * @return Return DAL_SUCCESS on success, DAL_FAILURE otherwise.
 */
int32_t (*ApplicationStateChanged) (tApplicationStateChangedEvent event);

/**
 * Sets the watchdog time. The time until the watchdog triggers
 * is the multiplication of factor and timeout_us.
 * @param factor The factor specifies how often the timeout_us
 * has to elapse until the watchdog triggers.
 * @param timeout_us The timeout of the watchdog in microseconds.
 * @return Return DAL_SUCCESS on success, DAL_FAILURE otherwise.
 */
int32_t (*WatchdogSetTime) (uint32_t factor, uint32_t timeout_us);

/**
 * Starts the watchdog.
 * @return DAL_SUCCESS on
 */
int32_t (*WatchdogStart) (void);

/**
 * Stops the watchdog.
 * @return Return DAL_SUCCESS on success, DAL_FAILURE otherwise.
 */
int32_t (*WatchdogStop) (void);

/**
 * Triggers the watchdog.
 * @return Return DAL_SUCCESS on success, DAL_FAILURE otherwise.
 */
int32_t (*WatchdogTrigger) (void);
};

```


5 Device specific functions "CANopen"

Device or fieldbus specific functions are referenced by their "function_name" and executed via the ADI proxy function "CallDeviceSpecificFunction".

```
(int32_t)lDalReturn = adi->CallDeviceSpecificFunction("<function_name>",
                                                    function_result,
                                                    additional_params);
```

Header and Includes:

```
//-----
// include file for CANopen WAGO ADI
//-----
#include <dal/canopen_types.h>
```

```
//-----
// CANopen device specific function names
//-----
"CANOPEN_CHANGE_STATE"
"CANOPEN_STATUS"
"CANOPEN_GET_NODE_ID"
"CANOPEN_GET_LAST_NODE"
"CANOPEN_GET_NODE_DIAG"
"CANOPEN_GET EMC MSG"
"CANOPEN_GET_NODE EMC MSG"
"CANOPEN_SEND EMC MSG"
"CANOPEN_GET GUARD ERROR"
"CANOPEN_GET_NODE GUARD_ERROR"
"CANOPEN_REGISTER_ID"
"CANOPEN_REGISTER_ALL"
"CANOPEN_GETFRAMECOUNT"
"CANOPEN_GETFRAMECOUNTER"
"CANOPEN_SENDFRAME"
"CANOPEN_GETFRAME"
"CANOPEN_SDO_READ"
"CANOPEN_SDO_WRITE"
"CANOPEN_SDO_READ_ASYNC"
"CANOPEN_SDO_WRITE_ASYNC"
"CANOPEN_SDO_ADD"
"CANOPEN_SDO_READ_DATA"
"CANOPEN_SDO_WRITE_DATA"
"CANOPEN_SENDTIMESTAMP"
"CANOPEN_GETTIMESTAMP"
"CANOPEN_SET_LED"
"CANOPEN_L2OPEN"
"CANOPEN_L2CLOSE"
"CANOPEN_L2RESET"
```

5.1 Reference "CANopen"

Function: CANOPEN_CHANGE_STATE

Set bus to state run / preoperational / resetnode etc

```
adi->CallDeviceSpecificFunction("CANOPEN_CHANGE_STATE",
                                (void*) &result,
                                (uint32_t) nodeID,
                                (uint32_t) state);
```

nodeID = 0 command to all slaves

state = State see CiA 302_2 page 33

0x05 start node
 0x04 stop node
 0x7F pre-operational
 0x06 reset node
 0x08 bootup config

result = DAL_SUCCESS / DAL_FAILURE

Function: CANOPEN_STATUS

Get communication interface status

```
adi->CallDeviceSpecificFunction("CANOPEN_STATUS",
                                (void*) &result,
                                (uint32_t) reset,
                                (uint8_t*) &can_com_state);
```

resetflags = error counter reset flags

result = DAL_SUCCESS

Function: CANOPEN_GET_NODE_ID

Get local node id

```
adi->CallDeviceSpecificFunction("CANOPEN_GET_NODE_ID",
                                (void*) &result);
```

result = node ID

Function: CANOPEN_GET_LAST_NODE

Get number of configured nodes

```
adi->CallDeviceSpecificFunction("CANOPEN_GET_LAST_NODE",
                                (void*) &result);
```

result = number of configured nodes

Function: CANOPEN_GET_NODE_DIAG

get node diagnostic flags of node

```
adi->CallDeviceSpecificFunction("CANOPEN_GET_NODE_DIAG",
                                (void*) &result,
                                (uint32_t) nodeID);
```

result = Diagnostic bitflags, see canopen_types.h

Function: CANOPEN_GET EMC_MSG

Test for new emergency error on any slave

```
adi->CallDeviceSpecificFunction("CANOPEN_GET EMC_MSG",
                                (void*) &result,
```

```
(uint32_t) remove,
(uint8_t*) ErrorMessage);
```

remove = 1 read and remove oldest message from list / 0 read last message (no remove)

can_data = last error message

result = reset 1 > DAL_SUCCESS
 reset 0 > 0 = no message
 slave id of error message

Function: CANOPEN_GET_NODE EMC MSG

Test for new emergency error on slave

```
adi->CallDeviceSpecificFunction("CANOPEN_GET_NODE EMC MSG",
                                (void*)&result,
                                (uint32_t) nodeID,
                                (uint32_t) remove,
                                (uint8_t*) ErrorMessage);
```

remove = 1 read and remove oldest message from list / 0 read last message (no remove)

can_data = last error message

result = 0 no message 1 message present

Function: CANOPEN_SEND EMC MSG

send emc message

```
adi->CallDeviceSpecificFunction("CANOPEN_SEND EMC MSG",
                                uint32_t err code,
                                uint32_t err reg,
                                uint8_t * manufac_err);
```

parameter EMC Error Code (1 Byte)
 Register 1001 (2 Byte)
 uint8_t* Manufacture spec. error field (5 Bytes)

result : OK = DAL_SUCCESS / Error = DAL_FAILURE (1 Error / 2 inhibit time not expired)

Function: CANOPEN_GET_GUARD ERROR

Test for new guard error on any slave

```
adi->CallDeviceSpecificFunction("CANOPEN_GET_GUARD ERROR",
                                (void*)&result);
```

result = 0 no error / slave id of error sender

Function: CANOPEN_GET_NODE_GUARD ERROR

Test for new guard error on slave

```
adi->CallDeviceSpecificFunction("CANOPEN_GET_NODE_GUARD ERROR",
                                (void*)&result,
                                (int32_t) can_slave_num);
```

result = 0 no error / 1 error present

Function: CANOPEN_REGISTER_ID

Set filter for CAN Layer2 frames

```
adi->CallDeviceSpecificFunction("CANOPEN_REGISTER_ID",
                                (void*)&result,
                                (uint32_t) regflag,
                                (uint32_t) MsgID);
```

MsgID = 11Bit/29Bit CAN Message ID
 regflag = 1 register ID / 0 unregister ID
 result = 0 OK / -2 no free index

Function: CANOPEN_REGISTER_ALL

Set filter for CAN Layer2 all frames

```
adi->CallDeviceSpecificFunction("CANOPEN_REGISTER_ALL",
                                (void*)&result,
                                (uint32_t)regflag);
```

regflag = 1 register ID / 0 unregister ID
result = 0 OK / -2 no free index

Function: CANOPEN_GETFRAMECOUNT

Read CAN Layer2 frames numbers

```
adi->CallDeviceSpecificFunction("CANOPEN_GETFRAMECOUNT",
                                (void*)&result,
                                (uint32_t)MsgID);
```

MsgID = 11Bit/29Bit CAN Message ID
result = messages present / -2 no index entry

Function: CANOPEN_GETFRAMECOUNTER

Read CAN Layer2 frames counter

```
adi->CallDeviceSpecificFunction("CANOPEN_GETFRAMECOUNTER",
                                (void*)&result,
                                (uint32_t)MsgID);
```

MsgID = 11Bit/29Bit CAN Message ID
result = messages counter / -2 no index entry

Function: CANOPEN_SENDFRAME

Send CAN Layer2 frame

```
adi->CallDeviceSpecificFunction("CANOPEN_SENDFRAME",
                                (void*)&result,
                                (uint32_t) MsgID,
                                (uint8_t*) can_data,
                                (uint32_t) dlc,
                                (uint32_t) timeout);
```

MsgID = 11Bit/29Bit CAN Message ID
can_data = Message bytes
dlc = dlc byte
timeout = timeout in ms / 0 = dont wait
result = 0 OK -5 OVERFLOW -3 TIMEOUT

Function: CANOPEN_GETFRAME

Get CAN Layer2 frame

```
adi->CallDeviceSpecificFunction("CANOPEN_GETFRAME",
                                (void*)&result,
                                (uint32_t*) MsgID,
                                (uint8_t*) can_data,
                                (uint32_t) timeout);
```

MsgID = 11Bit/29Bit CAN Message ID
can_data = Message bytes
timeout = timeout in ms / 0 = dont wait
result = DLC-byte OK -5 OVERFLOW -3 TIMEOUT

Function: CANOPEN_SDO_READ

read SDO from node and wait until answer or timeout

```
adi->CallDeviceSpecificFunction("CANOPEN_SDO_READ",
                                (void*)&result,
                                (uint32_t) nodeID,
                                (uint32_t) index,
                                (uint32_t) subindex,
                                (uint32_t) size,
                                (uint8_t *) data,
                                (uint32_t) timeout);
```

size = size of sdo (max 1536 byte) / 0 = no size given (max 245 bytes)
 data = SDO data
 index = OD index
 subindex = OD subindex
 timeout = timeout in ms (max. 5000ms) / default waittime (500ms)
 result = Bytes Read / Error Code

Function: CANOPEN_SDO_WRITE

Write SDO to node and wait until answer or timeout

```
adi->CallDeviceSpecificFunction("CANOPEN_SDO_WRITE",
                                (void*)&result,
                                (uint32_t) nodeID,
                                (uint32_t) index,
                                (uint32_t) subindex,
                                (uint32_t) size,
                                (uint8_t *) data,
                                (uint32_t) timeout);
```

size = size of sdo (max 1536 byte)
 data = SDO data
 index = OD index
 subindex = OD subindex
 timeout = timeout in ms (max. 5000ms) / default waittime (500ms)
 result = DAL_SUCCESS / Error Code

Function: CANOPEN_SDO_READ_ASYNC

non blocking read SDO from node

```
adi->CallDeviceSpecificFunction("CANOPEN_SDO_READ_ASYNC",
                                (void*)&result,
                                (uint32_t) nodeID,
                                (uint32_t) index,
                                (uint32_t) subindex,
                                (uint32_t) size,
                                (uint8_t *) data,
                                (uint32_t) timeout,
                                (uint32_t) sdo_channel)
```

size = size of sdo (max 1536 byte) / 0 = no size given (max 245 bytes)
 data = SDO data
 index = OD index
 subindex = OD subindex
 timeout = timeout in ms (max. 5000ms) / default waittime (500ms)
 sdo_channel = channel to use (0..15)
 result = Bytes Read / Error Code

Function: CANOPEN_SDO_WRITE_ASYNC

non blocking Write SDO to node / resets canopen_lasterror before write

```
adi->CallDeviceSpecificFunction("CANOPEN_SDO_WRITE_ASYNC",
                                (void*)&result,
                                (uint32_t) nodeID,
                                (uint32_t) index,
                                (uint32_t) subindex,
                                (uint32_t) size,
                                (uint8_t *) data,
                                (uint32_t) timeout,
                                (uint32_t) sdo_channel);
```

size = size of sdo (max 1536 byte)
 data = SDO data
 index = OD index
 subindex = OD subindex
 timeout = timeout in ms (max. 5000ms) / default waittime (500ms)
 sdo_channel = channel to use (0..15)
 result = DAL_SUCCESS / Error Code

Function: CANOPEN_SDO_ADD

add SDO object

```
adi->CallDeviceSpecificFunction("CANOPEN_SDO_ADD",
                                (void*)&result,
                                (uint32_t) index,
                                (uint32_t) entrys,
                                (uint32_t) datalen,
                                (uint32_t) type,
                                (uint32_t) offset);
```

index = OD index
 entrys = entrys in indexOD subindex
 datalen = size of one entry (max 1536 byte)
 type = SDO type
 offset = offset in process data array

Function: CANOPEN_SDO_READ_DATA

Read local SDO

```
adi->CallDeviceSpecificFunction("CANOPEN_SDO_READ_DATA",
                                (void*)&result,
                                (uint32_t) index,
                                (uint32_t) subindex,
                                (uint32_t) size,
                                (uint8_t *) data);
```

data = SDO data
 size = size of sdo (max 1536 byte)
 index = OD index
 subindex = OD subindex
 timeout = timeout in ms (max. 5000ms) / default waittime (500ms)
 result = Bytes Read / Error Code

Function: CANOPEN_SDO_WRITE_DATA

Write local SDO

```
adi->CallDeviceSpecificFunction("CANOPEN_SDO_WRITE_DATA",
                                (void*)&result,
                                (uint32_t) index,
                                (uint32_t) subindex,
                                (uint32_t) size,
                                (uint8_t *) data);
```

size = size of sdo (max 1536 byte)
 data = SDO data
 index = OD index
 subindex = OD subindex
 result = DAL_SUCCESS / Error Code

Function: CANOPEN_SENDTIMESTAMP

send timestamp

```
adi->CallDeviceSpecificFunction("CANOPEN_SENDTIMESTAMP",
                                (void*) &result,
                                (canopen_device_timestamp*) tminfo);
```

tminfo = struct with time information or null to use system time

Function: CANOPEN_GETTIMESTAMP

send timestamp

```
adi->CallDeviceSpecificFunction("CANOPEN_GETTIMESTAMP",
                                (void*) &result,
                                (canopen_device_timestamp*) tminfo);
```

tminfo = struct with last time information

Function: CANOPEN_SET_LED

set CAN led

```
adi->CallDeviceSpecificFunction("CANOPEN_SET_LED",
                                (void*) &result,
                                (uint32_t) led_mode);
```

0 automatic
 1 configuration running: red 50ms - green 50ms
 2 configuration fault: red 200ms - green 200ms
 3 stopped/init OK: green 200ms - off 800ms
 4 preoperational / no error: green 200ms - off 200ms
 5 preoperational / warning level: 1 * (red 200ms - off 200ms) - 2 * (green 200ms - off 200ms)
 6 preoperational / guard error: 2 * (red 200ms - off 200ms) - 2 * (green 200ms - off 200ms)
 7 preoperational / sync error: 3 * (red 200ms - off 200ms) - 2 * (green 200ms - off 200ms)
 8 operational / no error: green
 9 operational / warning level: 1 * (red 200ms - green 200ms) - green 800ms
 10 operational / guard error: 2 * (red 200ms - green 200ms) - green 800ms
 11 operational / sync error: 3 * (red 200ms - green 200ms) - green 800ms
 12 bus off: red
 13 fatal error: red 200ms - off 200ms

result = DAL_SUCCESS / Error Code

Function: CANOPEN_L2OPEN

open stack in layer2 mode

```
adi->CallDeviceSpecificFunction("CANOPEN_L2OPEN",
                                (void*) &result,
                                (uint32_t) can_baudrate);
```

Function: CANOPEN_L2CLOSE

close stack in layer2 mode

```
adi->CallDeviceSpecificFunction("CANOPEN_L2CLOSE",
                                (void*) &result);
```

Function: CANOPEN_L2RESET

reset stack in layer2 mode

```
adi->CallDeviceSpecificFunction("CANOPEN_L2RESET",
                                (void*) &result);
```

6 Device specific functions “KBUS”

Device or fieldbus specific functions are referenced by their “function_name” and executed via the ADI proxy function “CallDeviceSpecificFunction”.

```
(int32_t)lDalReturn = adi->CallDeviceSpecificFunction("<function_name>",
                                                    function_result,
                                                    additional_params);
```

Header and Includes:

```
//-----
// include files for KBUS WAGO ADI
//-----
#include <libpackbus.h>
```

The following KBUS specific functions are defined:

```
// KBUS control functions
"libpackbus_Push"
// functions to check KBUS configuration
"libpackbus_read_table_9"
"libpackbus_read_conf_reg"
"libpackbus_get_uid"
// functions to manage asynchronous KBUS communication
"libpackbus_async_com_open"
"libpackbus_async_com_close"
// functions for "register communication"
"libpackbus_async_com_startrdreg"
"libpackbus_async_com_pollrdreg"
"libpackbus_async_com_startwrreg"
"libpackbus_async_com_pollwrreg"
"libpackbus_async_com_strdreglst"
"libpackbus_async_com_pordreglst"
"libpackbus_async_com_stwrreglst"
"libpackbus_async_com_powrreglst"
// functions for "parameter channel" communication
"libpackbus_async_com_startrdpar"
"libpackbus_async_com_pollrdpar"
"libpackbus_async_com_startwrpar"
"libpackbus_async_com_pollwrpar"
"libpackbus_async_com_strdparlst"
"libpackbus_async_com_pordparlst"
"libpackbus_async_com_stwrparlst"
"libpackbus_async_com_powrparlst"
// exotic functions
"libpackbus_get_dig_offset"
"libpackbus_set_dig_offset"
```

The Header file “libpackbus.h” additionally provides *#defines* for a better readability:

```
#define LIBPACKBUS_DAL_FUNCTION_PUSH                "libpackbus_Push"
...
#define LIBPACKBUS_DAL_FUNC_ASYNC_COM_START_RD_REG  "libpackbus_async_com_startrdreg"
...
```


KBUS-I/O-Modules:

Every I/O-Module offers some fixed sized process data values and configuration information. Process data size varies from 2 bit up to 24 bytes. Process data values also have a direction, named "input process data" or "output process data".

Also IO-Modules are classified as "digital" or "complex".

Configuration of IO-Modules is stored inside "registers", accessible via "register-communication" or "parameter-channel".

Every WAGO Output-Module monitors elapsed time since the last kbus-cycle.

If the elapsed time exceeds 50ms, Output-Modules shut down their outputs to a failsafe state.

Process images:

With `adi->OpenDevice` the device driver identifies all connected IO-Modules and constructs two process images:

- Process-Image-of-Inputs(PII)
- Process-Image-of-Outputs(PIO)

Constructing the process images is a multistep mechanism.

In the first step only "complex" modules map their fixed sized process data into the process image, according to their position (offset) behind the head station's data.

The second step appends the process data of the "digital" modules according to their position (offset) behind the head station's data in a "packed" manner.

KBUS:

The KBUS acts like a shifting register, and is only able to write the compiled PIO data and read the compiled PII data. Therefor we talk about a "kbus-cycle".

During a kbus-cycle each IO-Module puts (gets) it's process data into (out of) the process images stream. Therefor modification of a single digital output makes no sense.

KBUS device driver:

Provide operating modes "UNCONFIGURED" and "RUNNING".

Process data access only possible in state "RUNNING".

After Power-On the driver always operate in mode "UNCONFIGURED".

In mode "UNCONFIGURED" the KBUS is driven by an internal thread with normal priority that periodically triggers a kbus-cycle for identifying and configuration purpose on connected IO-Module's.

Use `adi->ApplicationStateChanged(event)` to change the operating mode.

In mode "RUNNING" the KBUS has to be driven by your application.

You have to call function "`libpackbus_Push()`" to trigger every single kbus-cycle.

Furthermore you have to call "`libpackbus_Push()`" at least every 50ms.

If the calling interval exceeds 50ms, Output-Modules shut down their outputs to a failsafe state, because every WAGO Output-Module monitors elapsed time since the last kbus-cycle.

Device configuration

For KBUS there is no need to configure something.

```
adi->ConfigureDevice((tDeviceId) deviceId,
                    (void *)configData);
```

The Device driver identifies all connected IO-Modules, constructs the process images and operates automatically in application state "UNCONFIGURED".

Watchdog-functions

ADI provides a centralized watchdog feature, to inform all devices about the health status of the driving application. This feature becomes important if the fieldbus is driven by a separate hardware or independent software stack like PROFIBUS or CANopen.

For KBUS applications controlling the entire fieldbus there is no need to call "WatchdogTrigger()".

The ADI Watchdog functions have nothing to do with "Kbus-TimeOut of Output-Modules".

Making process images more "meaningful"

As pointed out earlier, a process image is nothing else then a linear array of bytes. The place where process data of 3rd channel of 1st digital output pops up depends on all the other plugged IO modules.

For a given configuration ...

```
// 750-8206 Programmable Fieldbus-Controller (PFC200)
// 750-504 4DO 24V DC, 0,5A
// 750-402 4DI 24V DC, 3ms
// 750-467 2AI 0..10V DC, Single-Ended
// 750-556 2AO -10..+10V DC
// 750-469 2AI Thermocouple K, Diag
// 750-652 Serial interface RS232/RS485 (22+2 bytes in processimage)
// 750-600 End-Module
```

... it may be helpful to define suitable structures.

```
struct typInputData
{
    unsigned short chlraw467;    //channel 1 of 750-467: AI 0..10V DC
    unsigned short ch2raw467;    //channel 2 of 750-467: AI 0..10V DC
    unsigned short templraw;     //channel 1 of 750-469: AI Thermocouple K, Diag
    unsigned short temp2raw;     //channel 2 of 750-469: AI Thermocouple K, Diag
    char tty_in[24];             //serial interface module 750-652 input data
    unsigned int ch1_402:1;       //channel 1 of 750-402: DI 24V DC, 3ms
    unsigned int ch2_402:1;       //channel 2 of 750-402: DI 24V DC, 3ms
    unsigned int ch3_402:1;       //channel 3 of 750-402: DI 24V DC, 3ms
    unsigned int ch4_402:1;       //channel 4 of 750-402: DI 24V DC, 3ms
}__attribute__((packed));

struct typOutputData
{
    unsigned short chlraw556;     //channel 1 of 750-556: AO -10..+10V DC
    unsigned short ch2raw556;     //channel 2 of 750-556: AO -10..+10V DC
    char tty_out[24];             //serial interface module 750-652 output data
    unsigned int ch1_504:1;       //channel 1 of 750-504: DO 24V DC, 0,5A
    unsigned int ch2_504:1;       //channel 2 of 750-504: DO 24V DC, 0,5A
    unsigned int ch3_504:1;       //channel 3 of 750-504: DO 24V DC, 0,5A
    unsigned int ch4_504:1;       //channel 4 of 750-504: DO 24V DC, 0,5A
}__attribute__((packed));
```

Afterwards you can address each channel in process image by its name.

6.1 Reference KBus

Function: "libpackbus_Push" - LIBPACKBUS_DAL_FUNCTION_PUSH

Trigger one kbus cycle and evaluates KBus data exchange.
This function executed synchronously, it typical return after 1-3ms.
The needed time depends on number and type of connected IO-Modules.

```
adi->CallDeviceSpecificFunction(LIBPACKBUS_DAL_FUNCTION_PUSH,
                                (int32_t*) &lFuncReturn );
```

IDalReturn: DAL_SUCCESS: Function found and evaluated
DAL_NOTUSED: Indicates a function is not implemented by the device

lFuncReturn: DAL_SUCCESS: OK
DAL_FAILURE: Module isn't initialized, application isn't active, ...

Function: "libpackbus_read_table 9"- LIBPACKBUS_DAL_FUNC_READ_TAB_9

Reads content of table 9.

```
adi->CallDeviceSpecificFunction(LIBPACKBUS_DAL_FUNC_READ_TAB_9,
                                (int32_t*) &lFuncReturn,
                                (uint8_t)   uchRegNo,
                                (uint16_t*) &usValue );
```

IDalReturn: DAL_SUCCESS: Function found and evaluated
DAL_NOTUSED: Indicates a function is not implemented by the device

param[in] uchRegNo register number in table 9 := terminal slot number

param[out] pusValue pointer to target buffer

lFuncReturn: DAL_SUCCESS: OK
-1 = NOT_OK: register number = 0 or terminal not found
-2 = NOT_OK: invalid value pointer (NULL)
2 = NOT_OK: value not available / table not initialized

Function: "libpackbus_read_conf_reg"- LIBPACKBUS_DAL_FUNC_READ_CONF_REG

Reads a copy of terminal register 8, 10, 11, 12, 13, 16, 28, 29, 30, 32, 33, 34 and 35.

```
adi->CallDeviceSpecificFunction(LIBPACKBUS_DAL_FUNC_READ_CONF_REG,
                                (int32_t*) &lFuncReturn,
                                (uint8_t)   uchSlotNo,
                                (uint8_t)   uchRegNo,
                                (uint16_t*) &usValue );
```

IDalReturn: DAL_SUCCESS: Function found and evaluated
DAL_NOTUSED: Indicates a function is not implemented by the device

param[in] uchSlotNo terminal slot number

param[in] uchRegNo terminal register number

param[out] pusValue pointer to target buffer

lFuncReturn: DAL_SUCCESS: OK
-1 = NOT_OK: terminal not found
-2 = NOT_OK: unsupported register number
-3 = NOT_OK: invalid value pointer (NULL)
2 = NOT_OK: value not available / copies not initialized

Function: "libpackbus_get_uid"- LIBPACKBUS_DAL_FUNC_GET_UID

Reads unified identification number of addressed terminal.

```
adi->CallDeviceSpecificFunction(LIBPACKBUS_DAL_FUNC_GET_UID,
                                (int32_t*) &lFuncReturn,
                                (uint8_t)   uchSlotNo,
                                (uint32_t*) &ulUid );
```

IDalReturn: DAL_SUCCESS: Function found and evaluated
 DAL_NOTUSED: Indicates a function is not implemented by the device

param[in] uchSlotNo slot number of the addressed terminal

param[out] pulUid pointer to unified identification number

IFuncReturn: DAL_SUCCESS: OK
 DAL_FAILURE: invalid pointer (NULL), terminal not found, ...

Function: "libpackbus_async_com_open" - LIBPACKBUS_DAL_FUNC_ASYNC_COM_OPEN

Opens a connection to PTP - module therefore it instances a PTP control structure.

```
adi->CallDeviceSpecificFunction(LIBPACKBUS_DAL_FUNC_ASYNC_COM_OPEN,
                                (int32_t*) &lFuncReturn,
                                (void**) &pvHandle );
```

IDalReturn: DAL_SUCCESS: Function found and evaluated
 DAL_NOTUSED: Indicates a function is not implemented by the device

param[out] ppvHandle: pointer to a handle of a channel to PTP - module

IFuncReturn: LPKD_REQ_EVA_STATUS_OK
 LPKD_REQ_EVA_ERROR_VALUE_PTR
 LPKD_REQ_EVA_ERROR_NO_HANDLE_GEN

Function: "libpackbus_async_com_close" - LIBPACKBUS_DAL_FUNC_ASYNC_COM_CLOSE

Closes a connection to PTP - module therefore it checks if the connection is still active.

```
adi->CallDeviceSpecificFunction(LIBPACKBUS_DAL_FUNC_ASYNC_COM_CLOSE,
                                (int32_t*) &lFuncReturn,
                                (void*) &vHandle );
```

IDalReturn: DAL_SUCCESS: Function found and evaluated
 DAL_NOTUSED: Indicates a function is not implemented by the device

param[in] pvHandle: handle of a channel to PTP - module

IFuncReturn: LPKD_REQ_EVA_STATUS_OK
 LPKD_REQ_EVA_ERROR_INV_HANDLE

Function: "libpackbus_async_com_startdreg" - LIBPACKBUS_DAL_FUNC_ASYNC_COM_START_RD_REG

Starts read of addressed register content.

Caution:
 Access interrupts process data exchange several times for several cycles!

```
adi->CallDeviceSpecificFunction(LIBPACKBUS_DAL_FUNC_ASYNC_COM_START_RD_REG,
                                (int32_t*) &lFuncReturn,
                                (void*) &vHandle,
                                (uint8_t) uchSlotNo,
                                (uint8_t) uchChannelNo,
                                (uint8_t) uchRegisterNo );
```

IDalReturn: DAL_SUCCESS: Function found and evaluated
 DAL_NOTUSED: Indicates a function is not implemented by the device

param[in] pvHandle: handle of a channel to PTP - module

param[in] uchSlotNo: slot number of the addressed terminal

param[in] uchChannelNo: channel number in the addressed device

param[in] uchRegisterNo: register number in the addressed channel

IFuncReturn: LPKD_REQ_EVA_STATUS_OK
 LPKD_REQ_EVA_ERROR_INV_HANDLE
 LPKD_REQ_EVA_ERROR_TERM_IDX

Function: "libpackbus_async_com_pollrdreg" - LIBPACKBUS_DAL_FUNC_ASYNC_COM_POLL_RD_REG

Polls the status and data of a started read register request.

```
adi->CallDeviceSpecificFunction(LIBPACKBUS_DAL_FUNC_ASYNC_COM_POLL_RD_REG,
                                (int32_t*) &lFuncReturn,
                                (void*) &vHandle,
                                (uint8_t) uchSlotNo,
                                (uint8_t) uchChannelNo,
                                (uint8_t) uchRegisterNo,
                                (uint16_t*) &usValue );
```

lDalReturn: DAL_SUCCESS: Function found and evaluated

DAL_NOTUSED: Indicates a function is not implemented by the device

param[in] pvHandle: handle of a channel to PTP - module

param[in] uchSlotNo: slot number of the addressed terminal

param[in] uchChannelNo: channel number in the addressed device

param[in] uchRegisterNo: register number in the addressed channel

param[out] pusValue: content of the addressed register

lFuncReturn: LPKD_REQ_EVA_STATUS_OK

LPKD_REQ_EVA_ERROR_INV_HANDLE

LPKD_REQ_EVA_ERROR_VALUE_PTR

LPKD_REQ_EVA_ERROR_CONT_NOT_FIT

Function: "libpackbus_async_com_startwrreg" - LIBPACKBUS_DAL_FUNC_ASYNC_COM_START_WR_REG

Starts write to addressed register.

Caution:

Access interrupts process data exchange several times for several cycles!

Caution:

Terminals do not return negative acknowledge! Write access to a read only register (e.g. SW version) will be accepted but not evaluated.

```
adi->CallDeviceSpecificFunction(LIBPACKBUS_DAL_FUNC_ASYNC_COM_START_WR_REG,
                                (int32_t*) &lFuncReturn,
                                (void*) &vHandle,
                                (uint8_t) uchSlotNo,
                                (uint8_t) uchChannelNo,
                                (uint8_t) uchRegisterNo,
                                (uint16_t) usValue,
                                (uint16_t) usPassWord );
```

lDalReturn: DAL_SUCCESS: Function found and evaluated

DAL_NOTUSED: Indicates a function is not implemented by the device

param[in] pvHandle: handle of a channel to PTP - module

param[in] uchSlotNo: slot number of the addressed terminal

param[in] uchChannelNo: channel number in the addressed device

param[in] uchRegisterNo: register number in the addressed channel

param[in] usValue: data for the addressed register

param[in] usPassWord: password to control write protection

lFuncReturn: LPKD_REQ_EVA_STATUS_OK

LPKD_REQ_EVA_ERROR_INV_HANDLE

LPKD_REQ_EVA_ERROR_TERM_IDX

...

Function: "libpackbus_async_com_pollwreg" - LIBPACKBUS_DAL_FUNC_ASYNC_COM_POLL_WR_REG

Polls the status and data of a started read register request.

Caution:

Terminals do not return negative acknowledge! Write access to a read only register (e.g. SW version) will be accepted but not evaluated.

```
adi->CallDeviceSpecificFunction(LIBPACKBUS_DAL_FUNC_ASYNC_COM_POLL_WR_REG,
                                (int32_t*) &lFuncReturn,
                                (void*) &vHandle,
                                (uint8_t) uchSlotNo,
                                (uint8_t) uchChannelNo,
                                (uint8_t) uchRegisterNo,
                                (uint16_t) usValue );
```

IDalReturn: DAL_SUCCESS: Function found and evaluated

DAL_NOTUSED: Indicates a function is not implemented by the device

param[in] pvHandle: handle of a channel to PTP - module

param[in] uchSlotNo: slot number of the addressed terminal

param[in] uchChannelNo: channel number in the addressed device

param[in] uchRegisterNo: register number in the addressed channel

param[in] usValue: data for the addressed register

lFuncReturn: LPKD_REQ_EVA_STATUS_OK

LPKD_REQ_EVA_ERROR_INV_HANDLE

LPKD_REQ_EVA_ERROR_CONT_NOT_FIT

Function: "libpackbus_async_com_strdreglst" - LIBPACKBUS_DAL_FUNC_ASYNC_COM_START_RD_REG_LIST

Starts read of addressed register list.

Caution:

Access interrupts process data exchange several times for several cycles!

```
adi->CallDeviceSpecificFunction(LIBPACKBUS_DAL_FUNC_ASYNC_COM_START_RD_REG_LIST,
                                (int32_t*) &lFuncReturn,
                                (void*) &vHandle,
                                (uint8_t) uchSlotNo,
                                (uint8_t) uchChannelNo,
                                (uint8_t) uchRegisterNo,
                                (uint8_t) uchRegisterCnt);
```

IDalReturn: DAL_SUCCESS: Function found and evaluated

DAL_NOTUSED: Indicates a function is not implemented by the device

param[in] pvHandle: handle of a channel to PTP - module

param[in] uchSlotNo: slot number of the addressed terminal

param[in] uchChannelNo: channel number in the addressed device

param[in] uchRegisterNo: register number in the addressed channel

param[in] uchRegisterCnt: number of registers to read (1 .. 64)

lFuncReturn: LPKD_REQ_EVA_STATUS_OK

LPKD_REQ_EVA_ERROR_INV_HANDLE

LPKD_REQ_EVA_ERROR_TERM_IDX

...

Function: "libpackbus_async_com_pordreglst" - LIBPACKBUS_DAL_FUNC_ASYNC_COM_POLL_RD_REG_LIST

Polls the status and data of a started read register list request.

```
adi->CallDeviceSpecificFunction(LIBPACKBUS_DAL_FUNC_ASYNC_COM_POLL_RD_REG_LIST,
                                (int32_t*) &lFuncReturn,
                                (void*) &vHandle,
                                (uint8_t) uchSlotNo,
                                (uint8_t) uchChannelNo,
                                (uint8_t) uchRegisterNo,
                                (uint8_t) uchRegisterCnt,
                                (uint16_t*) ausValue );
```

IDalReturn: DAL_SUCCESS: Function found and evaluated

DAL_NOTUSED: Indicates a function is not implemented by the device

param[in] pvHandle: handle of a channel to PTP - module

param[in] uchSlotNo: slot number of the addressed terminal

param[in] uchChannelNo: channel number in the addressed device

param[in] uchRegisterNo: register number in the addressed channel

param[in] uchRegisterCnt: number of registers to read (1 .. 64)

param[out] ausValue: content of the addressed registers

IFuncReturn: LPKD_REQ_EVA_STATUS_OK

LPKD_REQ_EVA_ERROR_INV_HANDLE

LPKD_REQ_EVA_ERROR_VALUE_PTR

LPKD_REQ_EVA_ERROR_CONT_NOT_FIT

Function: "libpackbus_async_com_stwrreglst" - LIBPACKBUS_DAL_FUNC_ASYNC_COM_START_WR_REG_LIST

Starts write to addressed register list.

Caution:

Access interrupts process data exchange several times for several cycles!

Caution:

Terminals do not return negative acknowledge! Write access to a read only register (e.g. SW version) will be accepted but not evaluated.

```
adi->CallDeviceSpecificFunction(LIBPACKBUS_DAL_FUNC_ASYNC_COM_START_WR_REG_LIST,
                                (int32_t*) &lFuncReturn,
                                (void*) &vHandle,
                                (uint8_t) uchSlotNo,
                                (uint8_t) uchChannelNo,
                                (uint8_t) uchRegisterNo,
                                (uint8_t) uchRegisterCnt,
                                (uint16_t*) ausValue,
                                (uint16_t*) usPassWord );
```

IDalReturn: DAL_SUCCESS: Function found and evaluated

DAL_NOTUSED: Indicates a function is not implemented by the device

param[in] pvHandle: handle of a channel to PTP - module

param[in] uchSlotNo: slot number of the addressed terminal

param[in] uchChannelNo: channel number in the addressed device

param[in] uchRegisterNo: register number in the addressed channel

param[in] uchRegisterCnt: number of registers to read (1 .. 64)

param[in] ausValue: data for the addressed registers

param[in] usPassWord: password to control write protection

IFuncReturn: LPKD_REQ_EVA_STATUS_OK

LPKD_REQ_EVA_ERROR_INV_HANDLE

LPKD_REQ_EVA_ERROR_TERM_IDX

...

Function: "libpackbus_async_com_powrreglst" - LIBPACKBUS_DAL_FUNC_ASYNC_COM_POLL_WR_REG_LIST

Polls the status and data of a started read register request.

Caution:

Terminals do not return negative acknowledge! Write access to a read only register (e.g. SW version) will be accepted but not evaluated.

```
adi->CallDeviceSpecificFunction(LIBPACKBUS_DAL_FUNC_ASYNC_COM_POLL_WR_REG_LIST,
                                (int32_t*) &lFuncReturn,
                                (void*) &vHandle,
                                (uint8_t) uchSlotNo,
                                (uint8_t) uchChannelNo,
                                (uint8_t) uchRegisterNo,
                                (uint8_t) uchRegisterCnt );
```

IDalReturn: DAL_SUCCESS: Function found and evaluated

DAL_NOTUSED: Indicates a function is not implemented by the device

param[in] pvHandle: handle of a channel to PTP - module

param[in] uchSlotNo: slot number of the addressed terminal

param[in] uchChannelNo: channel number in the addressed device

param[in] uchRegisterNo: register number in the addressed channel

param[in] uchRegisterCnt: number of registers to read (1 .. 64)

IFuncReturn: LPKD_REQ_EVA_STATUS_OK

LPKD_REQ_EVA_ERROR_INV_HANDLE

LPKD_REQ_EVA_ERROR_CONT_NOT_FIT

Function: "libpackbus_async_com_startrdpar" - LIBPACKBUS_DAL_FUNC_ASYNC_COM_START_RD_PAR

Starts read of addressed parameter content.

Caution:

Access interrupts process data exchange several times for several cycles!

```
adi->CallDeviceSpecificFunction(LIBPACKBUS_DAL_FUNC_ASYNC_COM_START_RD_PAR,
                                (int32_t*) &lFuncReturn,
                                (void*) &vHandle,
                                (uint8_t) uchSlotNo,
                                (uint8_t) uchParameterNo,
                                (uint16_t) usPassWord );
```

IDalReturn: DAL_SUCCESS: Function found and evaluated

DAL_NOTUSED: Indicates a function is not implemented by the device

param[in] pvHandle: handle of a channel to PTP - module

param[in] uchSlotNo: slot number of the addressed terminal

param[in] uchParameterNo: parameter number in the addressed terminal

param[in] usPassWord: password to control write protection

IFuncReturn: LPKD_REQ_EVA_STATUS_OK

LPKD_REQ_EVA_ERROR_INV_HANDLE

LPKD_REQ_EVA_ERROR_TERM_IDX

...

Function: "libpackbus_async_com_pollrdpar" - LIBPACKBUS_DAL_FUNC_ASYNC_COM_POLL_RD_PAR

Polls the status and data of a started read parameter request.

```
adi->CallDeviceSpecificFunction(LIBPACKBUS_DAL_FUNC_ASYNC_COM_POLL_RD_PAR,
                                (int32_t*) &lFuncReturn,
                                (void*) &vHandle,
                                (uint8_t) uchSlotNo,
                                (uint8_t) uchParameterNo,
                                (uint16_t*) &usValue,
                                (uint16_t*) &usStatus );
```

lDalReturn: DAL_SUCCESS: Function found and evaluated

DAL_NOTUSED: Indicates a function is not implemented by the device

param[in] pvHandle: handle of a channel to PTP - module

param[in] uchSlotNo: slot number of the addressed terminal

param[in] uchParameterNo: parameter number in the addressed terminal

param[out] pusValue: content of the addressed parameter

param[out] pusStatus: content of status register 57

lFuncReturn: LPKD_REQ_EVA_STATUS_OK

LPKD_REQ_EVA_ERROR_INV_HANDLE

LPKD_REQ_EVA_ERROR_VALUE_PTR

LPKD_REQ_EVA_ERROR_CONT_NOT_FIT

Function: "libpackbus_async_com_startwrpar" - LIBPACKBUS_DAL_FUNC_ASYNC_COM_START_WR_PAR

Starts write to addressed parameter.

Caution:

Access interrupts process data exchange several times for several cycles!

```
adi->CallDeviceSpecificFunction(LIBPACKBUS_DAL_FUNC_ASYNC_COM_START_WR_PAR,
                                (int32_t*) &lFuncReturn,
                                (void*) &vHandle,
                                (uint8_t) uchSlotNo,
                                (uint8_t) uchParameterNo,
                                (uint16_t) usValue,
                                (uint16_t) usPassWord );
```

lDalReturn: DAL_SUCCESS: Function found and evaluated

DAL_NOTUSED: Indicates a function is not implemented by the device

param[in] pvHandle: handle of a channel to PTP - module

param[in] uchSlotNo: slot number of the addressed terminal

param[in] uchParameterNo: parameter number in the addressed terminal

param[in] usValue: data for the addressed parameter

param[in] usPassWord: password to control write protection

lFuncReturn: LPKD_REQ_EVA_STATUS_OK

LPKD_REQ_EVA_ERROR_INV_HANDLE

LPKD_REQ_EVA_ERROR_TERM_IDX

...

Function: "libpackbus_async_com_pollwrpar" - LIBPACKBUS_DAL_FUNC_ASYNC_COM_POLL_WR_PAR

Polls the status and data of a started write parameter request.

```
adi->CallDeviceSpecificFunction(LIBPACKBUS_DAL_FUNC_ASYNC_COM_POLL_WR_PAR,
                                (int32_t*) &lFuncReturn,
                                (void*) &vHandle,
                                (uint8_t) uchSlotNo,
                                (uint8_t) uchParameterNo,
                                (uint16_t) usValue,
                                (uint16_t*) &usStatus );
```

lFuncReturn: DAL_SUCCESS: Function found and evaluated

DAL_NOTUSED: Indicates a function is not implemented by the device

param[in] pvHandle: handle of a channel to PTP - module

param[in] uchSlotNo: slot number of the addressed terminal

param[in] uchParameterNo: parameter number in the addressed terminal

param[in] usValue: data for the addressed parameter

param[out] pusStatus: content of status register 57

lFuncReturn: LPKD_REQ_EVA_STATUS_OK

LPKD_REQ_EVA_ERROR_INV_HANDLE

LPKD_REQ_EVA_ERROR_CONT_NOT_FIT

Function: "libpackbus_async_com_strdparlst" - LIBPACKBUS_DAL_FUNC_ASYNC_COM_START_RD_PAR_LIST

Starts read of addressed parameter list.

Caution:

Access interrupts process data exchange several times for several cycles!

```
adi->CallDeviceSpecificFunction(LIBPACKBUS_DAL_FUNC_ASYNC_COM_START_RD_PAR_LIST,
                                (int32_t*) &lFuncReturn,
                                (void*) &vHandle,
                                (uint8_t) uchSlotNo,
                                (uint8_t) uchParameterNo,
                                (uint8_t) uchParameterCnt,
                                (uint16_t) usPassWord );
```

lFuncReturn: DAL_SUCCESS: Function found and evaluated

DAL_NOTUSED: Indicates a function is not implemented by the device

param[in] pvHandle: handle of a channel to PTP - module

param[in] uchSlotNo: slot number of the addressed terminal

param[in] uchParameterNo: parameter number in the addressed terminal

param[in] uchParameterCnt: number of parameters to read (1 .. 256)

param[in] usPassWord: password to control write protection

lFuncReturn: LPKD_REQ_EVA_STATUS_OK

LPKD_REQ_EVA_ERROR_INV_HANDLE

LPKD_REQ_EVA_ERROR_TERM_IDX

...

Function: "libpackbus_async_com_pordparlst" - LIBPACKBUS_DAL_FUNC_ASYNC_COM_POLL_RD_PAR_LIST

Polls the status and data of a started read parameter list request.

```
adi->CallDeviceSpecificFunction(LIBPACKBUS_DAL_FUNC_ASYNC_COM_POLL_RD_PAR_LIST,
                                (int32_t*) &lFuncReturn,
                                (void*) &vHandle,
                                (uint8_t) uchSlotNo,
                                (uint8_t) uchChannelNo,
                                (uint8_t) uchParameterNo,
                                (uint8_t) uchParameterCnt,
                                (uint16_t*) ausValue,
                                (uint16_t*) &usStatus );
```

lDalReturn: DAL_SUCCESS: Function found and evaluated

DAL_NOTUSED: Indicates a function is not implemented by the device

param[in] pvHandle: handle of a channel to PTP - module

param[in] uchSlotNo: slot number of the addressed terminal

param[in] uchParameterNo: parameter number in the addressed terminal

param[in] uchParameterCnt: number of parameters to read (1 .. 256)

param[out] ausValue: content of the addressed parameters

param[out] pusStatus: content of status register 57

lFuncReturn: LPKD_REQ_EVA_STATUS_OK

LPKD_REQ_EVA_ERROR_INV_HANDLE

LPKD_REQ_EVA_ERROR_VALUE_PTR

LPKD_REQ_EVA_ERROR_CONT_NOT_FIT

Function: "libpackbus_async_com_stwrparlst" - LIBPACKBUS_DAL_FUNC_ASYNC_COM_START_WR_PAR_LIST

Starts write to addressed parameter list.

Caution:

Access interrupts process data exchange several times for several cycles!

```
adi->CallDeviceSpecificFunction(LIBPACKBUS_DAL_FUNC_ASYNC_COM_START_WR_PAR_LIST,
                                (int32_t*) &lFuncReturn,
                                (void*) &vHandle,
                                (uint8_t) uchSlotNo,
                                (uint8_t) uchParameterNo,
                                (uint8_t) uchParameterCnt,
                                (uint16_t*) ausValue,
                                (uint16_t) usPassWord );
```

lDalReturn: DAL_SUCCESS: Function found and evaluated

DAL_NOTUSED: Indicates a function is not implemented by the device

param[in] pvHandle: handle of a channel to PTP - module

param[in] uchSlotNo: slot number of the addressed terminal

param[in] uchParameterNo: parameter number in the addressed terminal

param[in] uchParameterCnt: number of parameters to read (1 .. 256)

param[in] ausValue: data for the addressed parameters

param[in] usPassWord: password to control write protection

lFuncReturn: LPKD_REQ_EVA_STATUS_OK

LPKD_REQ_EVA_ERROR_INV_HANDLE

LPKD_REQ_EVA_ERROR_TERM_IDX

...

Function: "libpackbus_async_com_pwrparlst" - LIBPACKBUS_DAL_FUNC_ASYNC_COM_POLL_WR_PAR_LIST

Polls the status and data of a started read parameter request.

```
adi->CallDeviceSpecificFunction(LIBPACKBUS_DAL_FUNC_ASYNC_COM_POLL_WR_PAR_LIST,
                                (int32_t*) &lFuncReturn,
                                (void*) &vHandle,
                                (uint8_t) uchSlotNo,
                                (uint8_t) uchParameterNo,
                                (uint8_t) uchParameterCnt,
                                (uint16_t*) &usStatus );
```

lDalReturn: DAL_SUCCESS: Function found and evaluated

DAL_NOTUSED: Indicates a function is not implemented by the device

param[in] pvHandle: handle of a channel to PTP - module

param[in] uchSlotNo: slot number of the addressed terminal

param[in] uchParameterNo: parameter number in the addressed terminal

param[in] uchParameterCnt: number of parameters to read (1 .. 256)

param[out] pusStatus: content of status register 57

lFuncReturn: LPKD_REQ_EVA_STATUS_OK

LPKD_REQ_EVA_ERROR_INV_HANDLE

LPKD_REQ_EVA_ERROR_CONT_NOT_FIT

Function: "libpackbus_get_dig_offset" - LIBPACKBUS_DAL_FUNC_GET_DIGITAL_OFFSET

Reads active - and internal offset value.

```
adi->CallDeviceSpecificFunction(LIBPACKBUS_DAL_FUNC_GET_DIGITAL_OFFSET,
                                (int32_t*) &lFuncReturn,
                                (bool) bOutputNotInput,
                                (uint16_t*) &usActiveOffset,
                                (uint16_t*) &usInternalOffset );
```

lDalReturn: DAL_SUCCESS: Function found and evaluated

DAL_NOTUSED: Indicates a function is not implemented by the device

param[in] bOutputNotInput: TRUE := output offset; FALSE := input offset

param[out] pusActiveOffset: pointer to return active offset value [no of byte]

param[out] pusInternalOffset: pointer to return internal offset value [no of byte]

lFuncReturn: DAL_SUCCESS: OK

1 = NOT_OK: invalid value pointer(s) (NULL)

Function: "libpackbus_set_dig_offset" - LIBPACKBUS_DAL_FUNC_SET_DIGITAL_OFFSET

Set active offset value.

```
adi->CallDeviceSpecificFunction(LIBPACKBUS_DAL_FUNC_SET_DIGITAL_OFFSET,
                                (int32_t*) &lFuncReturn,
                                (bool) bOutputNotInput,
                                (uint16_t) usOffset );
```

lDalReturn: DAL_SUCCESS: Function found and evaluated

DAL_NOTUSED: Indicates a function is not implemented by the device

param[in] bOutputNotInput: TRUE := output offset; FALSE := input offset

param[in] usOffset: offset value [no of byte]; offset := 0 -> use internal offset

lFuncReturn: DAL_SUCCESS: OK

1 = NOT_OK: desired offset value is less then internal offset value

7 Device specific functions "PROFIBUS-Slave"

Device or fieldbus specific functions are referenced by their "function_name" and executed via the ADI proxy function "CallDeviceSpecificFunction".

```
(int32_t)lDalReturn = adi->CallDeviceSpecificFunction("<function_name>",
                                                    function_result,
                                                    additional_params);
```

Header and Includes:

```
//-----
// include files for PROFIBUS-Slave WAGO ADI
//-----
#include "dpsTypes.h"
```

Following PROFIBUS-Slave specific function are defined :

```
//-----
// DPS device specific function names
//-----
"DPS_GET_DEV_STATE"
"DPS_GET_SYS_DIAG"
"DPS_GET_IM_DS"
"DPS_SET_IM_DS"
"DPS_SEND_DIAG"
"DPS_SET_ID_REL_DIAG"
"DPS_SET_MOD_STAT_DIAG"
"DPS_SET_CHAN_REL_DIAG"
"DPS_REM_CHAN_REL_DIAG"
"DPS_SET_STATUS_MSG"
"DPS_REM_STATUS_MSG"
"DPS_SET_ALARM"
"DPS_REM_ALARM"
"DPS_GET_DIAG_ENTRY_STATE"
"DPS_GET_NEXT_DS_JOB"
"DPS_GET_SPEC_DS_JOB"
"DPS_SET_DS_JOB_ACK"
"DPS_GET_DT_DATA"
"DPS_SET_DT_DATA"
```

Example:

```
uint32_t ulStatus;          /* status of the device specific function call */
uint8_t ucDevState = 0;    /* current device state */
...
/* get the device state */
iDalResult = adi->CallDeviceSpecificFunction("DPS GET DEV STATE",
                                             &ulStatus,
                                             &ucDevState);

if (DAL_SUCCESS == iDalResult)
{
    if (DPS_SUCCESS == ulStatus)
    {
        printf ("Device state %d\n", ucDevState);
    }
}
```

7.1 Reference PROFIBUS-Slave

Function: "DPS_GET_DEV_STATE"

This function returns the current device state.

```
adi->CallDeviceSpecificFunction("DPS_GET_DEV_STATE",
                                (int32_t*) &lFuncReturn,
                                (uint8_t*) pucDevState );
///
/// \param pucDevState pointer to the memory where the current device state should be stored.
///
/// \return The function returns \ref DPS_SUCCESS on success. Another value indicates failure.
```

Function: "DPS_GET_SYS_DIAG"

This function returns the currently active WAGO system diagnostic.

```
adi->CallDeviceSpecificFunction("DPS_GET_SYS_DIAG",
                                (int32_t*) &lFuncReturn,
                                (uint8_t*) pucSlot,
                                (uint8_t*) pucCode,
                                (uint8_t*) pucArgument,
                                (uint8_t*) pucExtArgument);
///
/// \param pucSlot pointer to the memory where the slot should be stored
/// \param pucCode pointer to the memory where the code should be stored
/// \param pucArgument pointer to the memory where the argument should be stored
/// \param pucExtArgument pointer to the memory where the extended error should be stored
///
/// \return The function returns \ref DPS_SUCCESS on success. Another value indicates failure.
```

Function: "DPS_GET_IM_DS"

This function returns the I&M dataset specified by the dataset identification.

```
adi->CallDeviceSpecificFunction("DPS_GET_IM_DS",
                                (int32_t*) &lFuncReturn,
                                (uint32_t) ulDsId,
                                (void*) pData);
///
/// \param ulDsId identification of the dataset to be read
/// \param pData destination memory where the dataset data should be copied
///
/// \return The function returns \ref DPS_SUCCESS on success. Another value indicates failure.
```

Function: "DPS_SET_IM_DS"

This function sets the data of the specified I&M data set.
It is only possible to set the data of the I&M datasets 1-4.

```
adi->CallDeviceSpecificFunction("DPS_SET_IM_DS",
                                (int32_t*) &lFuncReturn,
                                (uint32_t) ulDsId,
                                (void*) pData);
///
/// \param ulDsId identification number of the I&M data
/// \param pData pointer to the data to be written
///
/// \return The function returns \ref DPS_SUCCESS on success. Another value indicates failure.
```

Function: "DPS_SEND_DIAG"

This function sends the content of the station diagnostic telegram out of the diagnostic management to the PROFIBUS master in the case that the parameter 'ucReq' is not 0. Additionally the function returns the current diagnostic engine state by the parameter 'penmCurDiagEngState' as well as the diagnostic overflow state.

If the parameter 'ucReq' is 0 only the current diagnostic engine state will be returned.

```
adi->CallDeviceSpecificFunction("DPS_SEND_DIAG",
                                (int32_t*)          &lFuncReturn,
                                (uint8_t)            ucReq,
                                (diagEngState_t*)    penmCurDiagEngState,
                                (uint8_t*)          pucDiagOverflowState);
```

```
/// \param ucReq          unequal 0 performs a sending of the diagnostic telegram
/// \param penmCurDiagEngState return pointer for the current diagnostic engine state
/// \param pucDiagOverflowState return pointer for the diagnostics overflow state
///
/// \return The function returns \ref DPS_SUCCESS on success. Another value indicates failure.
```

Function: "DPS_SET_ID_REL_DIAG"

This function sets the specified module state of the specified slot in the identification related diagnostic object.

```
adi->CallDeviceSpecificFunction("DPS_SET_ID_REL_DIAG",
                                (int32_t*)          &lFuncReturn,
                                (uint32_t)           uSlot,
                                (idRelDiagModState_t) enmIdRelDiagModState);
```

```
/// \param uSlot          slot number
/// \param enmIdRelDiagModState module state
///
/// \return The function returns \ref DPS_SUCCESS on success. Another value indicates failure.
```

Function: "DPS_SET_MOD_STAT_DIAG"

This function sets the specified module state of the specified slot in the module status diagnostic object.

```
adi->CallDeviceSpecificFunction("DPS_SET_MOD_STAT_DIAG",
                                (int32_t*)          &lFuncReturn,
                                (uint32_t)           uSlot,
                                (modStatDiag_t)      enmModuleState);
```

```
///
/// \param uSlot          slot number
/// \param enmModuleState module state
///
/// \return The function returns \ref DPS_SUCCESS on success. Another value indicates failure.
```

Function: "DPS_SET_CHAN_REL_DIAG"

This function generates a channel related diagnostic object with the specified properties in the diagnostic management. With the returned diagnostic entry header it is possible to delete the entry later.

```
adi->CallDeviceSpecificFunction("DPS_SET_CHAN_REL_DIAG",
                                (int32_t*)      &lFuncReturn,
                                (uint32_t)       ulSlot,
                                (uint32_t)       ulChannel,
                                (chanType_t)     enmChannelType,
                                (chanSize_t)     enmChannelSize,
                                (chanDiagErrType_t) enmErrorType,
                                (diagEntrySev_t) enmDiagEntrySev,
                                (void**)          ppDiagEntryHdl);
```

```
/// \param ulSlot          slot number
/// \param ulChannel       channel number
/// \param enmChannelType  channel type
/// \param enmChannelSize  channel size
/// \param enmErrorType    error type
/// \param enmDiagEntrySev diagnostic entry severity
/// \param ppDiagEntryHdl  return pointer for the diagnostic entry header
///
/// \return The function returns \ref DPS_SUCCESS on success. Another value indicates failure.
```

Function: "DPS_REM_CHAN_REL_DIAG"

This function removes the specified channel related diagnostic object out of the diagnostic management. The object is specified by the entry handle.

```
adi->CallDeviceSpecificFunction("DPS_REM_CHAN_REL_DIAG",
                                (int32_t*) &lFuncReturn,
                                (void*)     pDiagEntryHdl);
```

```
/// \param pDiagEntryHdl  handle of the channel related diagnostic object
///
/// \return The function returns \ref DPS_SUCCESS on success. Another value indicates failure.
```

Function: "DPS_SET_STATUS_MSG"

This function generates a status message diagnostic object in the diagnostic management. With the returned diagnostic entry header it is possible to delete the entry later.

```
adi->CallDeviceSpecificFunction("DPS_SET_STATUS_MSG",
                                (int32_t*)      &lFuncReturn,
                                (uint32_t)       ulSlot,
                                (statusMsgType_t) enmStatusMsgType,
                                (statusMsgSpec_t) enmStatusMsgSpec,
                                (void*)          pUsrData,
                                (uint32_t)       ulUsrDataLen,
                                (diagEntrySev_t) enmDiagEntrySev,
                                (void**)          ppDiagEntryHdl);
```

```
/// \param ulSlot          slot number
/// \param enmStatusMsgType status message type
/// \param enmStatusMsgSpec status message specifier
/// \param pUsrData         status message user data
/// \param ulUsrDataLen     status message user data length (max. length is 4 byte)
/// \param enmDiagEntrySev diagnostic entry severity
/// \param ppDiagEntryHdl  return pointer for the diagnostic entry header
///
/// \return The function returns \ref DPS_SUCCESS on success. Another value indicates failure.
```


Function: "DPS_REM_STATUS_MSG"

This function removes the specified status message diagnostic object out of the diagnostic management.
The object is specified by the entry handle.

```
adi->CallDeviceSpecificFunction("DPS_REM_STATUS_MSG",
                                (int32_t*) &lFuncReturn,
                                (void*) pDiagEntryHdl);
```

```
/// \param pDiagEntryHdl   handle of the status message diagnostic object
///
/// \return The function returns \ref DPS_SUCCESS on success. Another value indicates failure.
```

Function: "DPS_SET_ALARM"

This function generates an alarm diagnostic object in the diagnostic management with the specified properties.
With the returned diagnostic entry header it is possible to delete the entry later.

```
adi->CallDeviceSpecificFunction("DPS_SET_ALARM",
                                (int32_t*) &lFuncReturn,
                                (uint32_t) ulSlot,
                                (alarmType_t) enmAlarmType,
                                (alarmSpec_t) enmAlarmSpec,
                                (bool) bAddAck,
                                (void*) pUsrData,
                                (uint32_t) ulUsrDataLen,
                                (void**) ppDiagEntryHdl);
```

```
/// \param ulSlot          slot number
/// \param enmAlarmType    alarm type
/// \param enmAlarmSpec    alarm specifier
/// \param bAddAck         additional acknowledge requested
/// \param pUsrData        pointer to the user data memory
/// \param ulUsrDataLen    status message user data length (max. length is 4 byte)
/// \param ppDiagEntryHdl  return pointer for the diagnostic entry header
///
/// \return The function returns \ref DPS_SUCCESS on success. Another value indicates failure.
```

Function: "DPS_REM_ALARM"

This function removes the specified alarm diagnostic object out of the diagnostic management.
The object is specified by the entry handle.

```
adi->CallDeviceSpecificFunction("DPS_REM_ALARM",
                                (int32_t*) &lFuncReturn,
                                (void*) pDiagEntryHdl);
```

```
/// \param pDiagEntryHdl   handle of the alarm diagnostic object
///
/// \return The function returns \ref DPS_SUCCESS on success. Another value indicates failure.
```

Function: "DPS_GET_DIAG_ENTRY_STATE"

This function returns the state of the specified diagnostic entry. The entry is specified by the diagnostic ID and the diagnostic entry handle.

```
adi->CallDeviceSpecificFunction("DPS_GET_DIAG_ENTRY_STATE",
                                (int32_t*) &lFuncReturn,
                                (void*) pDiagEntryHdl,
                                (entryState_t) penmState);
```

```
/// \param pDiagEntryHdl   handle of diagnostic entry
/// \param penmState        return pointer where the entry state should be stored
///
/// \return The function returns \ref DPS_SUCCESS on success. Another value indicates failure.
```

Function: "DPS_GET_NEXT_DS_JOB"

This function returns the data of the next dataset job generated by a DS_Write access.

```
adi->CallDeviceSpecificFunction("DPS_GET_NEXT_DS_JOB",
                                (int32_t*) &lFuncReturn,
                                (void*) pData,
                                (uint32_t) ulMaxDataLen);
```

```
/// \param pData      return pointer to the memory where the dataset data should be stored
/// \param ulMaxDataLen maximum data length of the destination memory
///
/// \return The function returns \ref DPS_SUCCESS on success. Another value indicates failure.
```

Function: "DPS_GET_SPEC_DS_JOB"

This function returns the data of the specified dataset job generated by a DS_Write access. The dataset is specified by the slot and the index.

```
adi->CallDeviceSpecificFunction("DPS_GET_SPEC_DS_JOB",
                                (int32_t*) &lFuncReturn,
                                (uint32_t) ulSlot,
                                (uint32_t) ulIndex,
                                (void*) pData,
                                (uint32_t) ulMaxDataLen);
```

```
/// \param ulSlot      slot number of the DS_Write telegram
/// \param ulIndex      index number of the DS_Write telegram
/// \param pData      return pointer to the memory where the dataset data should be stored
/// \param ulMaxDataLen maximum data length of the destination memory
///
/// \return The function returns \ref DPS_SUCCESS on success. Another value indicates failure.
```

Function: "DPS_SET_DS_JOB_ACK"

This function sets the acknowledge data of the specified DS_Write job. The job is specified by the job ID.

```
adi->CallDeviceSpecificFunction("DPS_SET_DS_JOB_ACK",
                                (int32_t*) &lFuncReturn,
                                (uint32_t) ulJobId,
                                (void*) pData,
                                (uint32_t) ulDataLen);
```

```
/// \param ulJobId      ID of the DS_Write job
/// \param pData      pointer to the source data memory
/// \param ulDataLen      length of the source data
///
/// \return The function returns \ref DPS_SUCCESS on success. Another value indicates failure.
```

Function: "DPS_GET_DT_DATA"

This function reads the data out of the PROFIBUS data transport buffer input buffer.

```
adi->CallDeviceSpecificFunction("DPS_GET_DT_DATA",
                                (int32_t*) &lFuncReturn,
                                (void*) pData,
                                (uint32_t) ulMaxDataLen);
```

```
/// \param pData      return pointer to the memory where the dataset data should be stored
/// \param ulMaxDataLen maximum data length of the destination memory
///
/// \return The function returns 'true' on success. The return value Another value 'false' indicates failure.
```

Function: "DPS_SET_DT_DATA"

This function writes the data into the PROFIBUS data transport buffer output buffer.

```
adi->CallDeviceSpecificFunction("DPS_SET_DT_DATA",  
                                (int32_t*) &lFuncReturn,  
                                (void*) pData,  
                                (uint32_t) ulDataLen);
```

/// \param pData return pointer to the memory where the dataset data should be stored

/// \param ulDataLen maximum data length of the destination memory

///

/// \return The function returns 'true' on success. The return value Another value 'false' indicates failure.

8 Attachment A – Licenses

```

/* GLIB - Library of useful routines for C programming
 * Copyright (C) 1995-1997 Peter Mattis, Spencer Kimball and Josh MacDonald
 *
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Lesser General Public
 * License as published by the Free Software Foundation; either
 * version 2 of the License, or (at your option) any later version.
 *
 * This library is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
 * Lesser General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public
 * License along with this library; if not, write to the
 * Free Software Foundation, Inc., 59 Temple Place - Suite 330,
 * Boston, MA 02111-1307, USA.
 */

```

libffi - Copyright (c) 1996-2012 Anthony Green, Red Hat, Inc and others.
See source files for details.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED ``AS IS'', WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.

IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

WAGO Kontakttechnik GmbH & Co. KG
Postfach 2880 D-32385 Minden
Hansastraße 27 D-32423 Minden
Phone: +49/5 71/8 87 – 0
Fax: +49/5 71/8 87 – 1 69
E-Mail: info@wago.com
Internet: <http://www.wago.com>

