Device Driver Manual

Linux CIF Device Driver

Hilscher Gesellschaft für Systemautomation mbH
Rheinstraße 78
D-65795 Hattersheim
Germany

Tel. +49 (0) 6190 / 9907 - 0
Fax. +49 (0) 6190 / 9907 - 50

Sales: +49 (0) 6190 / 9907 - 0
Hotline and Support: +49 (0) 6190 / 9907 - 99

Sales Email: sales@hilscher.com
Hotline and Support Email: hotline@hilscher.com

Web: http://www.hilscher.com

Index	Date	Version	Chapter	Comment
1	11.08.00	1.000	all	created
2	26.07.02	2.000	all	rewritten

Although this program has been developed with great care and intensively tested, Hilscher Gesellschaft für Systemautomation mbH cannot guarantee the suitability of this program for any purpose not confirmed by us in writing.

Guarantee claims shall be limited to the right to require rectification. Liability for any damages which may have arisen from the use of this program or its documentation shall be limited to cases of intent.

We reserve the right to modify our products and their specifications at any time in as far as this contributes to technical progress. The version of the manual supplied with the program applies.

1 Introduction

This manual describes driver package, load/unload topics, supported hardware and copyright issue. The application programming interface (**API**) to our communication boards is also explained in detail.

1.1 Linux

Linux is a free operating system developed under the GNU General Public License , the source code for Linux is freely available to everyone.

Linux is a cost-effective, reliable and secure operating system. It is constantly being updated and refined with the latest technologies. Linux gains greater acceptance throughout the computing industry. Our company supports the use of Linux in the field of industrial communication. This driver supports all Hilscher cards with Dual Port Memory Interface.

Where to get Linux? Please, visit www.linux.org home page. There you can find any Linux related information and useful links.

1.2 The Driver Versions

Linux CIF Device Driver V2.000 was developed and tested with Linux Kernel version 2.4.0, 2.4.2.

The driver version 1.003 works under Kernel 2.2.10, 2.2.14, 2.2.16

1.3 Supported Hilscher Cards

Linux CIF Device Driver supports Hilscher CIF-50 PCI and CIF-30/CIF-104 ISA cards. These are Profibus, Interbus, DeviceNet and CANopen cards.

1.4 Data transfer

On the communication boards, we distinguish between two types of data transfer.

- The first one is the message oriented data transfer used by message oriented protocols.
- The second one is the data exchange with process images from I/O based protocols.

1.5 Terms for this Manual

DPM	**D**ual-**P**ort **M**emory is the physical interface to all communication board (DPM is also used for PROFIBUS-DP Master).
CIF	**C**ommunication **I**nter**F**ace
COM	**CO**mmunication **M**odule
HOST	Application on the PC or a similar device
DEVICE	Synonym for communication interfaces or communication modules
RCS	**R**ealtime **C**ommunicating **S**ystem, this is the name of the operating system that runs on the communication boards

2 Getting Started

Overview

- Chapter Communication includes general definitions and describes the fundamentals about data transfers between an application and the communication boards.
- Chapter The Device Driver describes an overview, the installation and configuration of the device
- The important chapter Programming Instructions describes the basic functionality of using the device driver and presents an example.
- All functions of the device driver are explained in chapter The Application Programming Interface.
- Chapter Error Numbers lists a detail description of the error numbers
- Chapter Development Environments informs about used development tools.

3 Communication

3.1 About the User Interface

3.1.1 Message Interface and Process Data Image

There are two ways of data transfer between the HOST and the DEVICE:

- **Message oriented data transfer**

  For telegram oriented protocols like PROFIBUS-FMS the data transfer happens with messages, which will be send or received over two mailboxes in the dual-port memory. There is one mailbox for each direction (Send direction and receive direction). Normally, the data transfer will be controlled by events.

- **Process data image transfer**

  In fieldbus systems, which handle input and output data, like PROFIBUS-DP or InterBus-S, there is a data image of the process data inside the dual-port memory. Input data and output data have their own area and the data transfer normally happens cyclic.

3.1.2 The Protocol Dependent and Independent User Interface

The user interface via the dual-port memory of the communication interface and the communication module has two parts, a protocol dependent, and a protocol independent part.

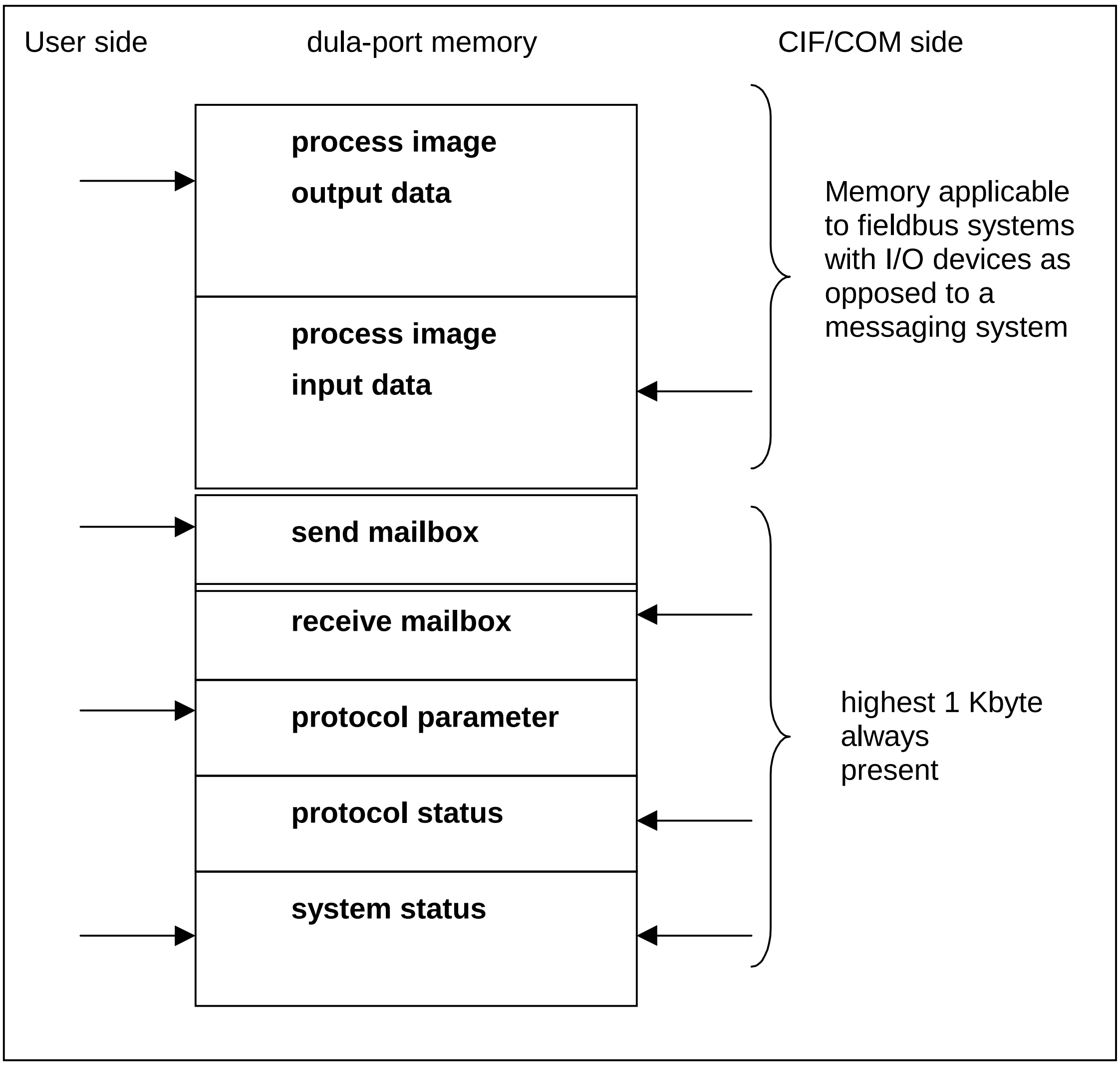
The protocol independent part of the dual-port memory is the main part of the data between HOST and DEVICE.

The particular protocol dependent part are the parameters for initializing the protocol and the message structure for exchanging jobs between the HOST and the DEVICE. These jobs are called messages. The structure of a message has reached a high standard. This means that changing to another protocol is very simple.

The exactly composition of a message is described in the paticular protocol manual. The difference between the various protocols are only the protocol parameters. The data model of the dual-port memory and the mechanism of message exchange are always the same.

3.2 Interface Structure

The interface to the communication board based on a dual-port memory. The following picture shows the various parts of the dual-port memory.



One dual-port memory map for all CIFs/COMs and all protocols with

- Process image for input and output data
- Two mailboxes for message communication
- Parameter area for simple protocols (baudrate, data bits, parity ...)
- Protocol status information (telegram counter, last error, valid slaves...)
- System status (firmware name/version, CIF revision/serial number...)

3.3 Message and Process Data Communication

3.3.1 Message Communication

A message is a unique data structure in which the user transmits or receives commands and data from the CIF or COM.

A message consists of an 8 byte message header, an 8 byte telegram header and up to 247 bytes of user data.

Message Header Used from operating system for transportation of the message. It is defined in this manual and constant for the application.

Telegram Header Defines the action for the protocol task.

User data Send/received data.

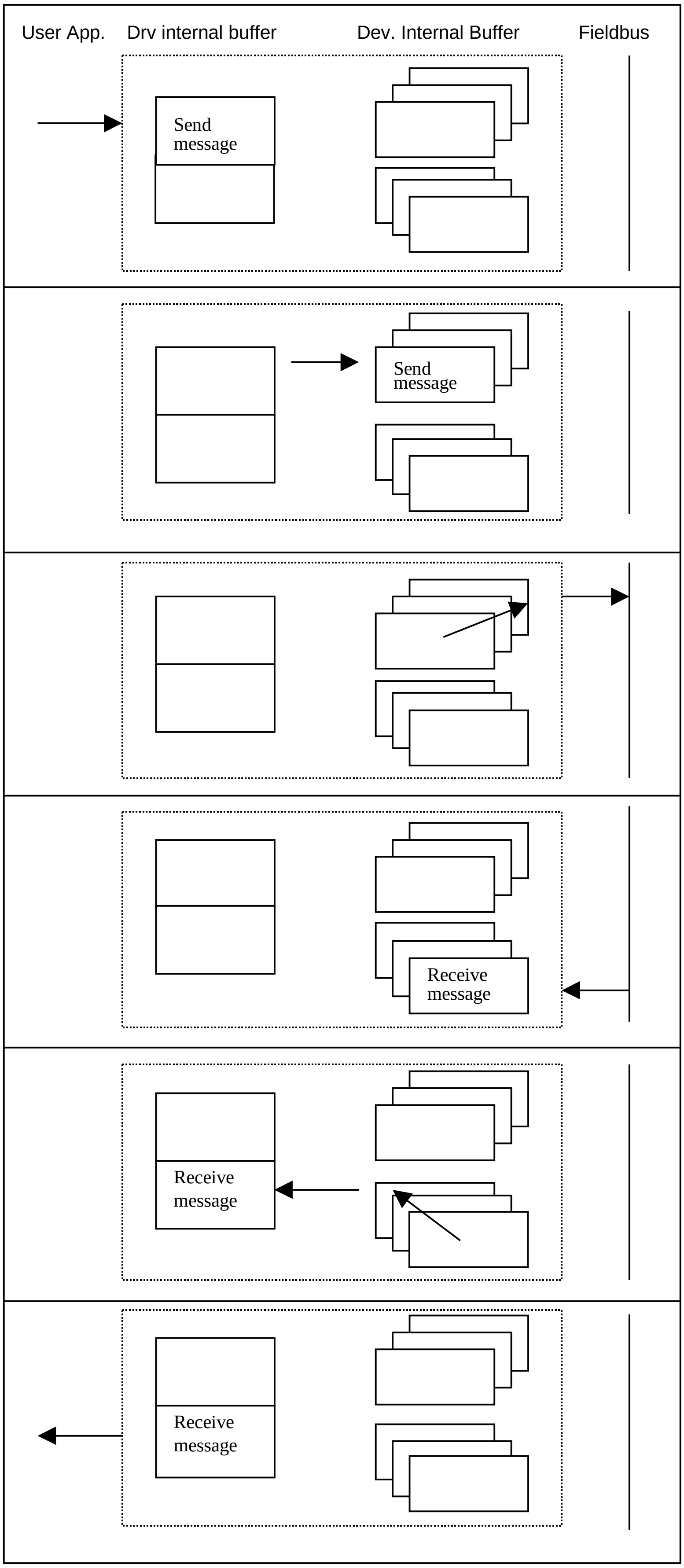
Parameter	Type	Meaning	
Msg.Rx	byte	Number of Receiving Task	Message Header
Msg.Tx	byte	Number of Sending Task	
Msg.Ln	byte	Data length	
Msg.Nr	byte	Number of Message for Identification	
Msg.A	byte	Number of Responses	
Msg.F	byte	Error Code	
Msg.B	byte	Number of Command	
Msg.E	byte	Completion	
Msg.DeviceAdr	byte	Communication Reference	Telegram Header
Msg.DataArea	byte	Data Block	
Msg.DataAdr	word	Object Index	
Msg.DataIdx	byte	Object Subindex	
Msg.DataCnt	byte	Data Quantity	
Msg.DataType	byte	Data Type	
Msg.Fnc	byte	Service	
Msg.D[0-246]	byte	User Data	Telegram User Data

General structure of a message

The meaning of the telegram header is an example for PROFIBUS-FMS. For other protocols the structure is the same but, the parameters change as for example with Modbus Plus, from communication reference to slave address, object index to register address or service to function code.

The driver transfers a message independant from the protocol and works transparent. The message reproduces the telegram.

3.3.1.1 Sending (Putting) and Receiving (Getting) Messages



The user creates the send message and calls `DevPutMessage()` command.

Device Driver copies Msg into internal Msg-Buffer and starts DMA.

The device takes out the message, puts it in an internal queue and signals this action to the HOST.

The queue is handled by the FIFO principle. If the message is on the first position, it will be decoded to generate the send telegram.

If the device receives the acknowledge telegram, it generates a receive message and puts it in the queue.

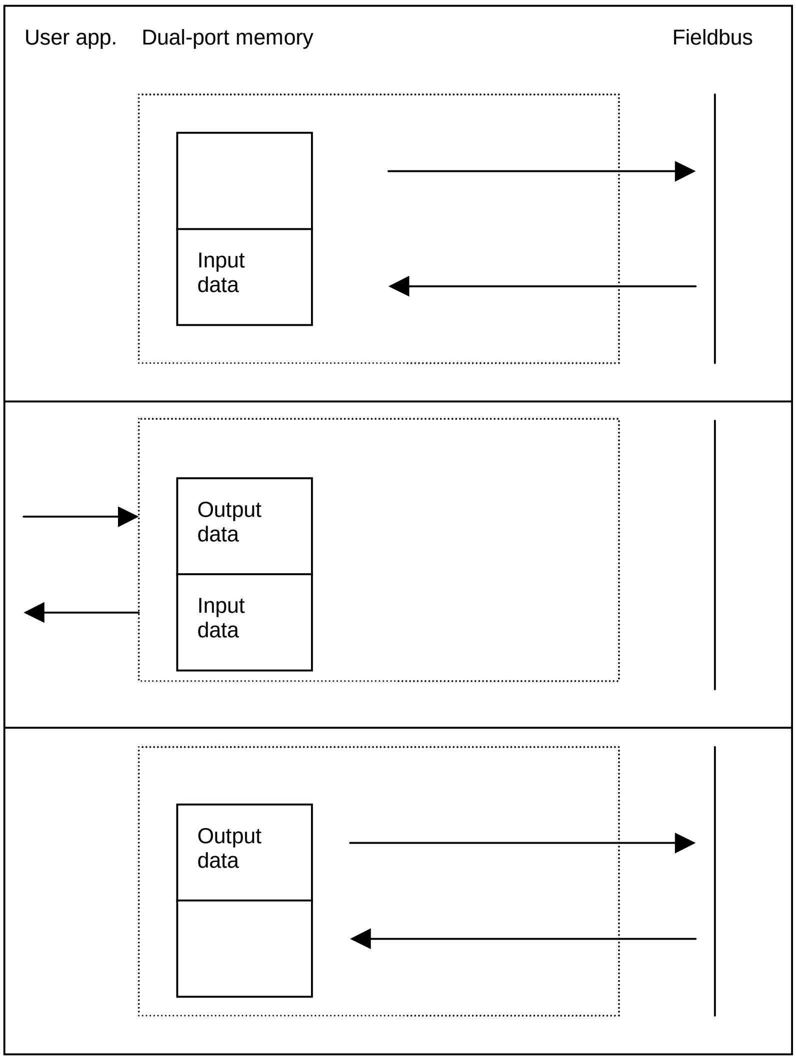
If the message is in the first position and the receive mailbox is empty, the message will be copied in driver internal buffer and the mailbox set valid.

The user takes out the receive message, with the `DevGetMessage()` command, which sets the mailbox state to empty.

3.3.2 I/O Communication with a Process Image

In fieldbus systems with IO devices like PROFIBUS-DP or InterBus-S there is a process image of the IO data available directly in the dual-port memory. The access is the same if the CIF or COM works as master or slave. Depending on the application the user can choose between several handshake modes, or if only byte consistence is required, the user can read and write without any synchronization.

3.3.2.1 Direct Data Transfer, DEVICE Controlled



The CIF starts by itself a data exchange cycle if it is a master, or it receives a data exchange cycle if it is a slave

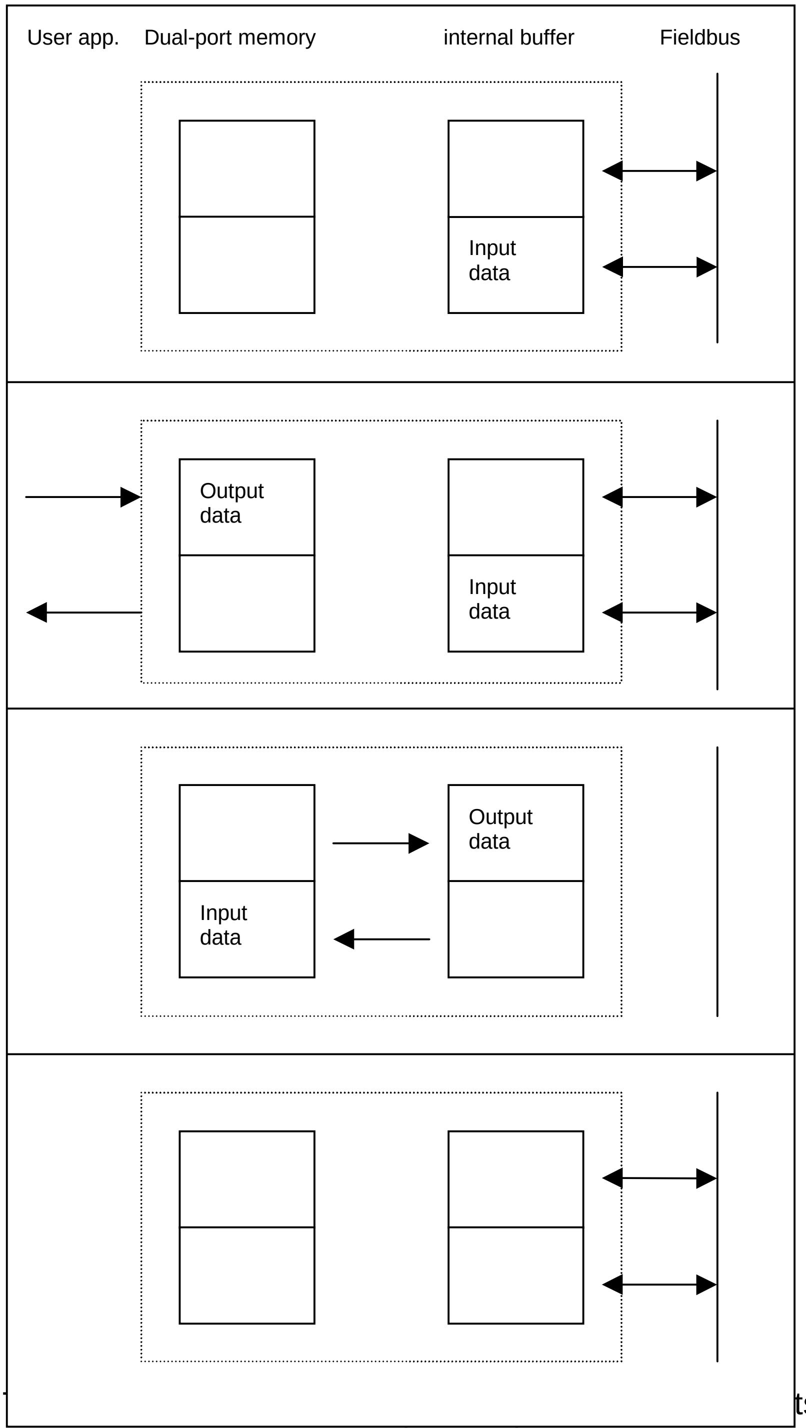
Now the user can read new input data and write the output data in the dual-port memory. This is done by the **DevExchangeIO()** function. .

The CIF/COM starts the next data exchange cycle.

.

Typical application: slave system, which must guarantee that the data from every master cycle must be given to the user program.

3.3.2.2 Buffered Data Transfer, DEVICE Controlled



CIF/COM makes cyclic data exchanges on the bus.

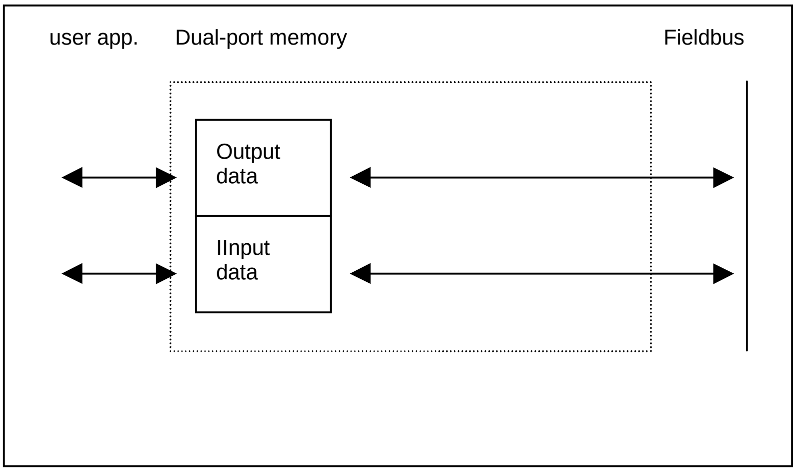
After each data exchange the CIF/COM checks, if the DPM is available.

The user can read out the input data and write the new output data. This is done by the DevEexchangeIO() function.

If there was one data exchange on the bus in the meantime, the CIF/COM exchanges the data between the internal buffer and the dual-port memory.

ts an interrupt with the next data exchange cycle.

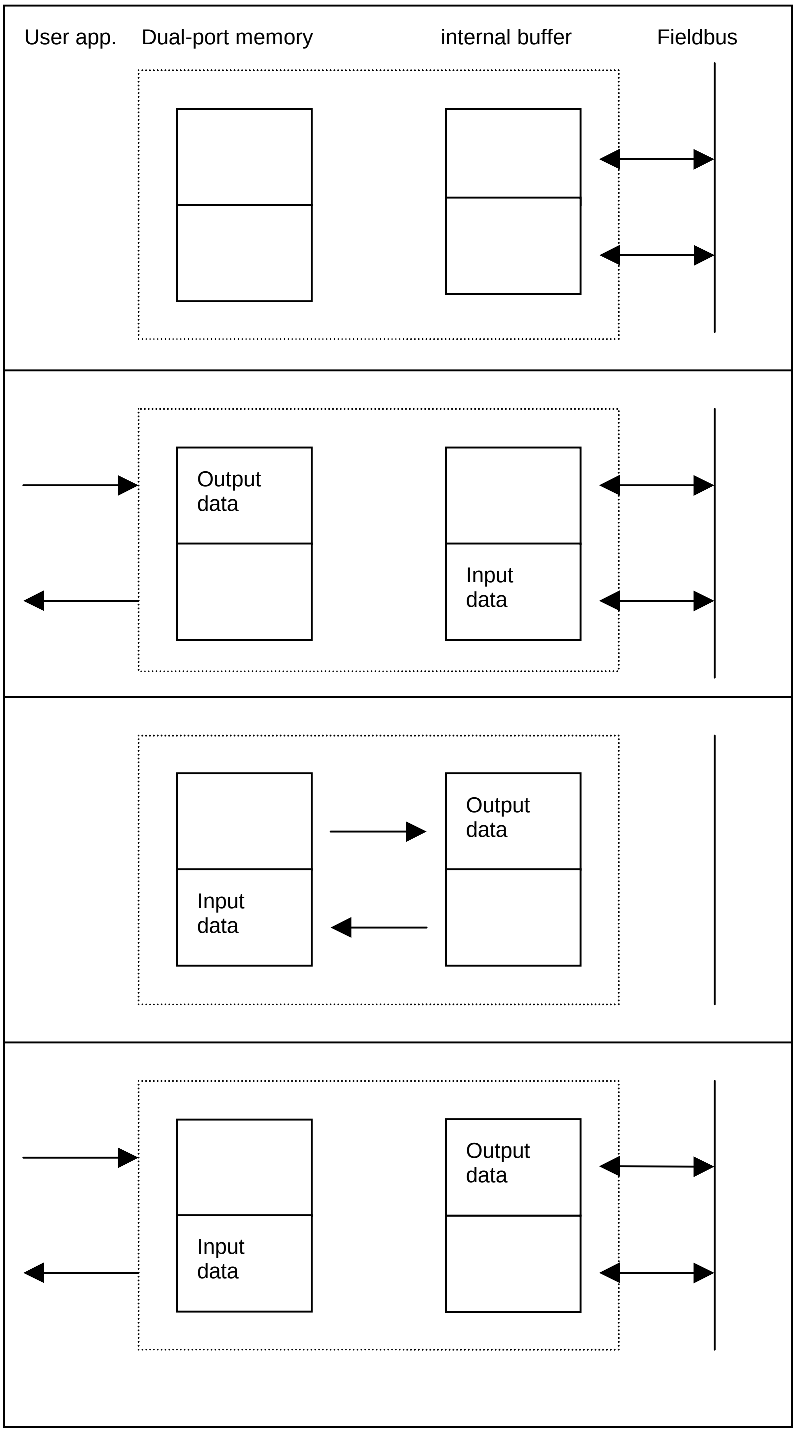
3.3.2.3 Uncontrolled Direct Data Transfer



The user reads and writes the process image, with the DevEexchangeIO() function, at the same time like the CIF/COM.

The CIF/COM does cyclic data exchanges and after every exchange it makes an update of the process image.

3.3.2.4 Buffered Data Transfer, HOST Controlled



Cyclic data exchange between internal buffer and fieldbus.

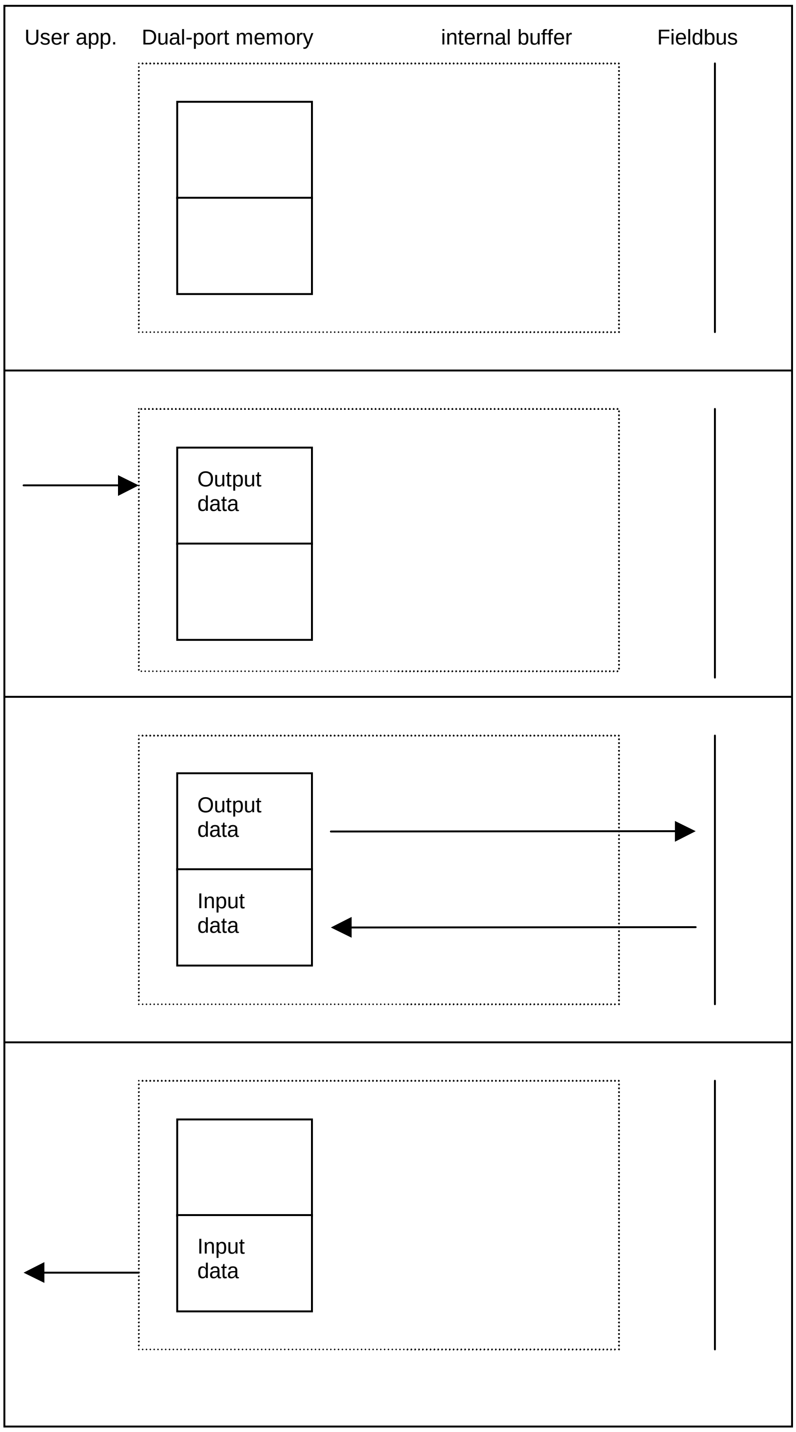
The user reads last input data and writes new output data with the DevEexchangeIO() function. Data exchange continues.

CIF stops data exchange, puts the output data in the internal buffer and the latest input data in the dual-port memory.

User reads input data and writes output data (DevExchangeIO()).

Typical application: easiest handshake in master and slave systems with a guaranteed consistence of the complete process image.

3.3.2.5 Direct Data Transfer, HOST Controlled



No data exchange.

The user writes new output data with the DevEexchangeIO() function.

CIF starts one data exchange with the output data in the DPM and writes the new input data in the dual-port memory.

User reads new input data with the next DevExchangeIO().

Typical application: master system with synchronous IO devices.

The Software Structure on the Communication Boards

The software is based on an extremely modular architecture. The protocol itself is a self-contained module which has no variables in common with any other software module apart from the operating system. It is therefore possible to implement the protocol with the same software module on all our boards, thus ensuring the greatest software quality.

The main parts of the firmware are the real-time operating system and the protocol task(s).

3.3.3 The Real-Time Operating System

The operating system can manage 7 tasks, and is optimized for real-time communications services. It provides the following functions:

- Distribution of computing time among the individual-tasks.
- Task communication.
- Memory management.
- Provision of time functions.
- Diagnostic and general management functions.
- Transmit and receive functions.

The computing time is evenly distributed by the operating system among all tasks ready to run. A task switch, i.e. switch over to the next task, takes place in cycles every millisecond.

If a task has to wait for an external event, e.g. for the receipt of data, it is no longer ready to run and a task switch is performed immediately.

The available computing time and the maximum possible sum baud rate make sure, that a less prior task is not completely blocked by a high priority task. Presumably the data through put is lower in this case.

Communication between the tasks takes place by messages. These are the areas of memory made available by the operating system into which the tasks write data. Transport of messages from one task to another and notification to a task that a message is there is handled by the operating system.

The operating system also manages the memory area for storage of the tasks and their stack. Individual tasks can be deleted or reloaded.

A task can wait for an event and the operating system will restart the task when the event has occurred, the time resolution is 1 millisecond.

The operating system can stop or start individual tasks and pass on certain jobs to them. The tasks thus make available data in the trace buffer which is managed by the operating system.

The operating system communicates with the HOST (PC or a similar device) via the dual-port memory interface. There is access to the individual-operating system functions and to the individual tasks via the communications system.

3.3.4 The Protocol Task

The protocol task is responsible for transmission of the data in accordance with the protocol. The parameters it requires for this are taken from the dual-port memory or from the FLASH-memory.

A transmit job is always initiated with a message. This contains all the data to be transmitted. These are provided with any control characters and checksums required and then output by interrupt or DMA. At the same time, the corresponding monitoring periods are started. When the data has been transfered or an error has occurred, a corresponding acknowledgment is returned to the sender of the message.

Depending on the protocol, receive messages are restored after the transmission. Receiving is done by interrupt or DMA. If a message has been received without error, it is passed on by message to the PC via the dual-port memory interface.

I/O oriented protocol tasks work on the bus independently according to the given protocol specification. The data transfer is not done by a message, but is done by direct reading or writing to the send and receive data in the dual-port memory.

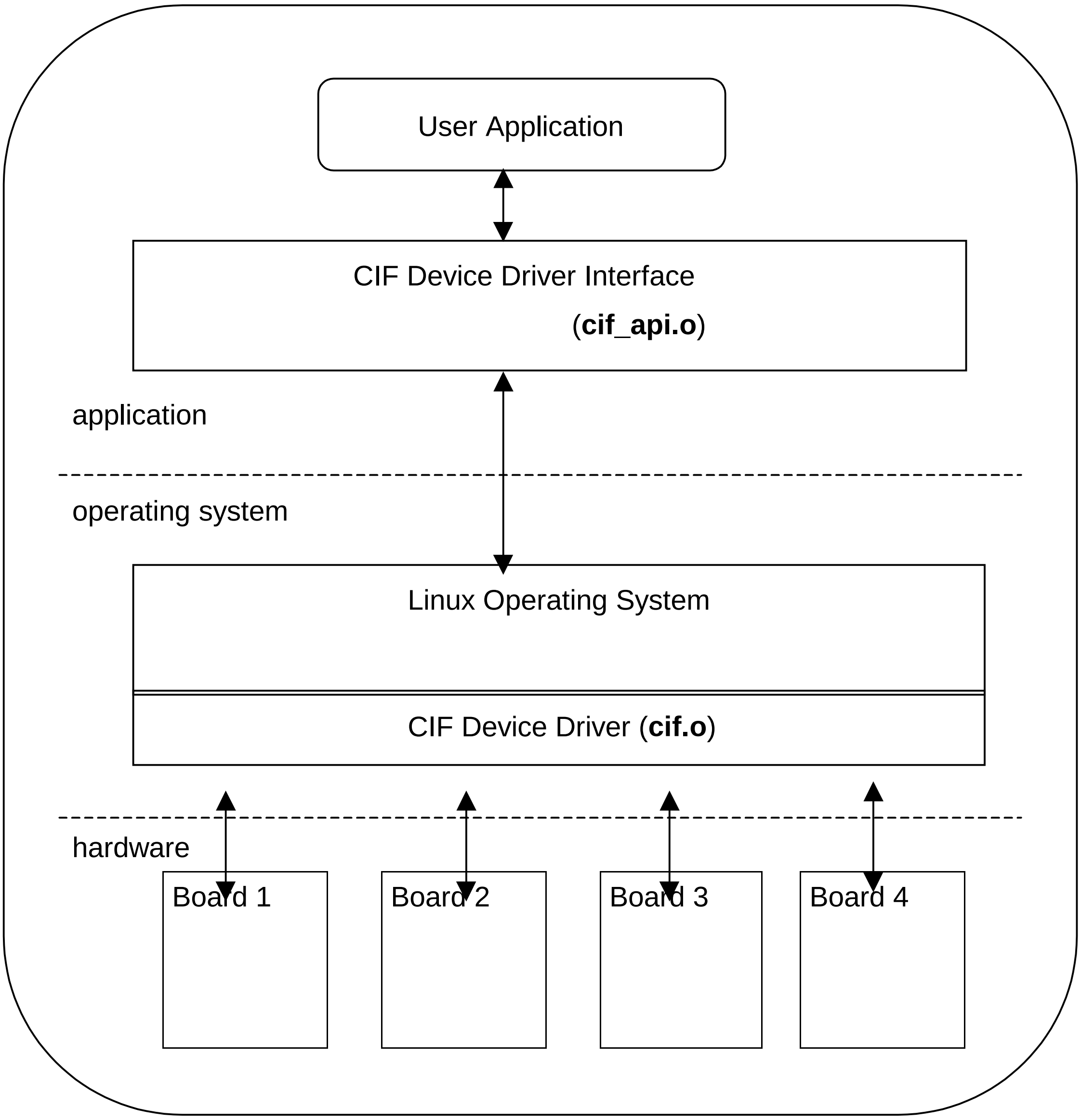
As the protocol task runs independently, a wide variety of protocols can be implemented on the CIF, PC/104 or COM by replacing this task. Different tasks can also be used for the two serial interfaces.

4 The Device Driver

4.1 General

Linux CIF Device Driver was implemented as kernel mode driver and offers the best performance for Hilscher cards on the Linux operating system.

The Driver implements very fast interrupt handler that guarantees optimal utilization of our hardware. It can operate in polling mode too. If there is no mandatory reason to use polling mode, use always interrupt mode. By hardware events interrupts are providing best response time on the event.



Function Overview:

- handles one to four communication boards at once
- Interrupt and polling mode useable for each board (except PCMCIA)

All boards can be run in interrupt or polling mode. If interrupt mode is configured for a board the device driver will install an interrupt service function for this board. The driver will install an own interrupt service function for each interrupt driven board. So the boards can be handled independently.

The difference between interrupt and poll mode is only the handling of application request during timeout situations. If an application has to wait for a function (e.g. `DevReset()`) so in interrupt mode the application will be blocked in the driver and the CPU is free to do other work. After the given timeout or at the end of the command, the application is released and does normal executing.

In poll mode the driver will run a "while loop", waiting until the function has finished or the given timeout is reached. The user can also use the functions without timeout (timeout=0) and run the polling by itself.

It is possible to use independent processes for send message (`DevPutMessage()`), receive message (`DevGetMessage()`) and I/O data transfers (`DevExchangeIO()`). Each process will be blocked in the driver when necessary without blocking the other ones. If threads are used and a function has to wait for a certain operation (timeout paramter unequal 0), the driver blocking mechanism will block each thread which is accesseing the driver. This is by design, because all threads in a process are sharing the same driver handle (hidden in the driver API).

A solution is to use timeout=0 in the driver functions and to check the return values if the function is processed without an error. For the message transfer functions (`DevPutMessage()` and `DevGetMessage()`), `DevGetMBXState()` can be used to check if the function can be e executed. immediatley.

On each board only one receive (`DevGetMessage()`), one send (`DevPutMessage()`) and one IO-Exchange (`DevExchangeIO()`) command can be active at the same time, because there is no command queuing in the driver implemented. So if one command for the specific function is active, all further commands to the same function will be returned with an error. All other driver functions are reentrant and can be called at every time.

Notice **Switching between pooling mode and interrupt mode is supported by the driver setup program (DrvSu)**

4.2 Package Contents

Installation Directory	Subdirectory	Description
. . ./cif-V2.000	**src**	Driver file, driver source files, script files for driver load/unload
	inc	Driver header files
	usr-inc	Header files for API prototypes, protocol dependant header
	api	cif_api.o, API prototypes
	test	Driver test program with source code
	console	Demo console program
	man	Driver manual
	AUTO	Autoloading information

Device Driver files:

cif.o CIF Device Driver file

cif_load, **cif_unload** scripts for loading and unloading of the driver

API files:

cif_api.o Object file of the driver interface

cif_user.h Definition header file for the user interface

Test program:

DrvSu An application for testing and debugging

4.3 Installation of the driver

To install this package on your computer simply extract .tgz file in your installation directory:

```
tar xzfv cifv2000.tgz
```

4.4 Device Driver startup/shutdown

To load and unload a driver - **cif.o** (located in 'src/' subdirectory of the installation directory) please use 'cif_load' and 'cif_unload' scripts. **PCI** cards are autodetected by the driver. If you use **ISA** cards, you have to pass load parameters to the driver.

In order to load/unload the driver at the system start/shutdown you must modify some system scripts. Please consult files located in the ‘AUTO’ subdirectory of the installation directory.

4.4.1 ISA Boards

For the **ISA** boards you have to specify following parameters: DPM-address, DPM-size and IRQ-number. you can have plugged up to four Hilscher communication Boards at a time, so you can pass up to four DPM-addresses, DPM-sizes and IRQ-numbers. The best way to describe this is by showing a few samples of the command line.

```
./cif_load dpm_add=0xCA000 dpm_len=0x2 irq=11
```

if there is only one **ISA** board plugged with appropriate jumper settings.

If you want the board to operate in polling mode simply pass IRQ-number 0:

```
./cif_load dpm_add=0xCA000 dpm_len=0x2 irq=0
./cif_load dpm_add=0xCA000,0xCB000,0xCC000,0xCD000
           dpm_len=0x2,0x8,0x6,0x8
           irq=11,9,12,14
```

if there are four **ISA** boards plugged with appropriate jumper settings

---

Note: On Intel platforms, DPM-addresses for **ISA** boards are in range 640KB-1MB (0xA0000 to 0xFFFFF). Do not forget to tag IRQ-number, you are going to use for **ISA** card, in BIOS as an **ISA** IRQ.

---

5 Programming Instructions

5.1 Include the Interface API in Your Application

For the user API there is only one include file cif_user.h which contains all the necessary information like structure, constant and prototype definitions. A complete function description is given in the chapter 'The Programming Interface'. Link the device API object (cif_api.o) according to your program.

For the support of the various protocols, each protocol has its own header file where all the protocol dependent definition are included (e.g. dpm_user.h for the PROFIBUS-DP Master protocol). Furthermore, there exists an include file rcs_usr.h for the definitions of the operating system of the communication boards.

5.2 Open and Close the driver

Only three functions are needed to get a DEVICE to work:

Open a Driver

- Open the driver

  DevOpenDriver(), checks if a driver is installed

- Initialize your communication board

  DevInitBoard(), check if a specific board is available

- Set the application ready state

  DevSetHostState(HOST_READY), signals the board an application

After these functions your application is able to start with the communication.

Close a Driver

- Clear the application ready state

  DevSetHostState(HOST_NOT_READY), signals the board, no application running

- Close the board link

  DevExitBoard(), unlink from a board

- Close the device driver

  DevCloseDriver(), close a link to the device driver

After calling these functions all resources for the communication API are freed.

5.3 Writing an Application

5.3.1 Determine Device Information

The interface API includes information functions, which gives an application the possibility to determine the installed DEVICEs, the actual driver version and the firmware name and version installed on the device.We suggest to read out these informations and make them accessible to the user. This information can be used by support inquiries to our hotline.

Important information:

- Driver version
- DEVICE type, model and serial number
- Firmware name and version

Read informations about installed devices:

After opening the driver with `DevOpenDriver()`, the function `DevGetBoardInfo()` can be used to read the driver version and the installed devices.

```
void Demo (void)
{
  short      sRet;
  BOARD_INFO tBoardInfo;

  if ( (sRet = DevOpenDriver()) == DRV_NO_ERROR) {
    // Driver successfully opend, read board information
    if ( (sRet = DevGetBoardInfo( &tBoardInfo) != DRV_NO_ERROR) {
      // Function error
      printf( "DevGetBoardInfo      RetWert = %5d \n", sRet );
    } else {
      // Information successfully read, save for further use
      // Check out which boards are available
      for ( usIdx = 0; usIdx < MAX_DEV_BOARDS; usIdx++){
        if ( tBoardInfo.tBoard[usIdx].usAvailable == TRUE) {
          // Board is configured, try to init the board
          sRet = DevInitBoard( tBoardInfo.tBoard[usIdx].usBoardNumber);

          if  ( sRet != DRV_NO_ERROR) {
            // Function error
            printf( "DevInitBoard      RetWert = %5d \n", sRet );
          } else {
            // DEVICE is available and ready............
          }
        }
      }
    }
  }
}
```

Please refer to the function `DevGetBoardInfo()` for a description of the BOARD_INFO structure.

Read informations about a specific DEVICE:

After opening a specific DEVICE with `DevInitBoard()` a lot of informations about a DEVICE can be read by the function `DevGetInfo()`.

```
void Demo (void)
{
  short         sRet;
  BOARD_INFO    tBoardInfo;
  FIRMWARE_INFO tFirmwareInfo;
  VERSION_INFO  tVersionInfo;
  DEVINFO       tDeviceInfo;

  if ( (sRet = DevOpenDriver()) == DRV_NO_ERROR) {
    // Driver successfully opend, read board information
    if ( (sRet =DevGetBoardInfo( &tBoardInfo) != DRV_NO_ERROR) {
      // Function error
      printf( "DevGetBoardInfo      RetWert = %5d \n", sRet );
    } else {
      // Information successfully read, open all existing boards
      for ( usIdx = 0; usIdx < MAX_DEV_BOARDS; usIdx++){
        if ( tBoardInfo.tBoard[usIdx].usAvailable == TRUE) {
          // Board is configured, try to init the board
          sRet = DevInitBoard( tBoardInfo.tBoard[usIdx].usBoardNumber);
          if  ( sRet != DRV_NO_ERROR) {
            // Function error
            printf( "DevInitBoard      RetWert = %5d \n", sRet );
          } else {

            // DEVICE is available and ready............

            // Read DEVICE specific information (VERSION_INFO)
            sRet = DevGetInfo( tBoardInfo.tBoard[usIdx].usBoardNumber,
                               GET_VERSION_INFO,
                               sizeof(tVersionInfo),
                               tVersionInfo);

            // Read DEVICE specific information (DEVICE_INFO)
            sRet = DevGetInfo( tBoardInfo.tBoard[usIdx].usBoardNumber,
                               GET_DEV_INFO,
                               sizeof(tDeviceInfo),
                               tDeviceInfo);

            // Read DEVICE specific information (FIRMWARE_INFO)
            sRet = DevGetInfo( tBoardInfo.tBoard[usIdx].usBoardNumber,
                               GET_FIRMWARE_INFO,
                               sizeof(tFirmwareInfo),
                               tFirmwareInfo);
          }
        }
      } /* end for */
    }
  }
}
```

Please refer to the `DevGetInfo()` function for a description of the different information structures.

5.3.2 Message Based Application

On message based application you have to be aware that a DEVICE can onlyqueue a fix number of messages (normally 20 to 128). Message queuing will be done in send and receive direction. This means, the HOST and the connected protocol will share all available messages. Each request or response from both sides will occupy a message until it is transfered to the other side.If the amount of messages exceeds the given limit, no matter if the HOST or the protocol uses all the messages, the DEVICE is not longer able to create a response for a send or receive request. This will happen until a message is freed by transferring it to the HOST or sending it over by the protocol. This will free a message, which can be used for another data transfer.

So an application should always be able to receive messages to prevent theDEVICE for overrunning by the use of messages.

After opening the device interface and setting the application ready state, the application must be able to process receive messages from the DEVICE.

Example 1:

```
/***************************************************************************
/*  Mainprogram
/***************************************************************************
include "../usr-inc/cif_user.h"
int main( void )
{
  short           sRet;
  MSG_STRUC       tReceiceMessage;
  MSG_STRUC       tSendMessage;

  /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
  /* Open the driver */
  if ( (sRet = DevOpenDriver()) != DRV_NO_ERROR) {
    printf( "DevOpenDriver       RetWert = %5d \n", sRet );

  /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
  /* Initialize board */
  } else if ( (sRet = DevInitBoard (0)) != DRV_NO_ERROR) {
    printf( "DevInitBoard        RetWert = %5d \n", sRet );

  /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
  /* Signal board, application is running */
  } else if ( (sRet = DevSetHostState( 0,
                                       HOST_READY,
                                       0L) != DRV_NO_ERROR) ) {
    printf( "DevSetHostState (HOST_READY) RetWert = %5d \n", sRet );

  } else {

          while ( ...PROGRAM IS RUNNING....) {

        // Application work........
        // Try to read a message
        sRet = DevGetMessage( 0,
                              &tReceiveMessage,
                              100L);   // Wait a maximum of 100 ms

                  if ( sRet == DRV_GET_TIMEOUT ) {
                    // No message available
                    // Try again..............
                  } else if ( sRet != DRV_NO_ERROR ) {
                    // This is a function error
                    // Process error ............
                  } else {
                    // Message available
                    // Process message ..........
                  }

                  // Try to send a message
                  // Create a message like described in the protocol manual
                  sRet = DevPutMessage( 0,
                                        &tSendMessage,
                                        100L);  // Wait a maximum of 100 ms
```

```
                    if ( sRet == DRV_PUT_TIMEOUT) {
                      // Message could not be send
                      // Mailbox full......
                    } else if ( sRet != DRV_NO_ERROR)  ) {
                      // Error during send message
                      // Process message error ........
                    }
                  } /* end while*/

     // Close the application
     /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
     /* Signal board, application is not running */
     if ( (sRet = DevSetHostState(0,
                                  HOST_NOT_READY,
                                  0L)) != DRV_NO_ERROR) {
       printf( "DevSetHostState       RetWert = %5d \n", sRet );
     }

     /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
     /* Free board */
     if ( (sRet = DevExitBoard (0)) != DRV_NO_ERROR) {
       printf( "DevExitBoard        RetWert = %5d \n", sRet );
     }

     /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
     /* Close driver */
     if ( (sRet = DevCloseDriver()) != DRV_NO_ERROR ) {
       printf( "DevCloseDriver      RetWert = %5d \n", sRet );
     }
      }
} /* end main*/
```

DevPutMessage() and DevGetMessage() uses a timeout value to force the driver to wait for the completion of the function, until the given timeout period is passed. This timeout should be used because the device needs also a period of time to get a message or to write a message. This period is normally very short (400 us up to 4 ms) but working in a while loop with timeout equal to zero and try to put a message in such a loop will result in a bad system response.

The given timeout from 100 ms is the maximum time the function will wait for completion. It will return immediately if the function is done.

The application is responsible for the reiteration of messages which could not be send to the DEVICE.

How the device acts after power up or changes of the HOST ready state (e.g. shut down the bus or stop data transmission) is normally configurable by the protocol configuration.

Another way to check if messages can be send or received is the use of the DevGetMBXState() function. This function is used to determine the actual state (DEVICE_MBX_FULL/EMPTY, HOST_MBX_FULL/EMPTY) of the HOST and DEVICE mailbox. This the preferred way for a polling application.

Example 2:

```
/*************************************************************************
/*  Mainprogram
/*************************************************************************
int main( void )
{
  unsigned short  usDevState, usHostState;
  short           sRet;
  MSG_STRUC       tReceiceMessage;
  MSG_STRUC       tSendMessage;


  // ..... see example 1

  // HOST and DEVICE mailbox state
  if ( (sRet = DevGetMBXState( 0,
                               &usDeviceState,
                               &usHostState)) != DEV_NO_ERROR) {
    printf( "DevGetMBXState   RetWert = %5d \n", sRet );
  } else {
    if ( usHostState == HOST_MBX_FULL) {
      // Read device message. message is available
      if ( (sRet = DevGetMessage( 0,
                                  &tReceiveMessage,
                                  0L)) != DRV_NO_ERROR) {
        printf( "DevGetMessage   RetWert = %5d \n", sRet );
      } else {
        // Process message ..........
      }
    }
    if ( usDeviceState == DEVICE_MBX_EMPTY) {
      // Send mailbox is empty
      if ( (sRet = DevPutMessage( 0,
                                  &tSendMessage,
                                  0L)) != DRV_NO_ERROR) {
        printf( "DevPutMessage   RetWert = %5d \n", sRet );
      }
    }
  }

  //....... see example 1
```

In this example, the application must create its own polling cycle an is responsible for freeing the processor for other applications.

5.3.3 Process Data Image Based Application

Applications which working with process data images (IO protocols) are using the DevExchangeIO(), DevExchangeIOErr() or DevExchangeIOEx() function for the data transfer between the HOST and the DEVICE.

ATTENTION: **By using `DevExchangeIO()` it is not possible for master devices to recognize the fault of a specific bus device. Only global errors like whole bus disruptions or communication breaks to all configured device will be indicated by this function.To get specific device fault, the application must read the "TaskState-Field", where device specific datas are located.This must be done after each call to `DevExchangeIO()`.**

Example 1:

```
/************************************************************************
/*  Mainprogram
/************************************************************************
include "../usr-inc/cif_user.h"
int main( void )
{
  short           sRet;
  unsigned char   abIOSendData[512];
  unsigned char   abIOReceiveData[512];


  /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */    /* Open the
driver */
  if ( (sRet = DevOpenDriver()) != DRV_NO_ERROR) {
    printf( "DevOpenDriver       RetWert = %5d \n", sRet );

  /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
  /* Initialize board */
  } else if ( (sRet = DevInitBoard (0)) != DRV_NO_ERROR) {
    printf( "DevInitBoard        RetWert = %5d \n", sRet );

  /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */    /* Signal
board, application is running */
  } else if ( (sRet = DevSetHostState( 0,
                                       HOST_READY,
                                       0L) != DRV_NO_ERROR) ) {
    printf( "DevSetHostState (HOST_READY) RetWert = %5d \n", sRet );


  } else {

          while ( ...PROGRAM IS RUNNING....) {

        // Application work........

        // Insert datas to the send data buffer
        abIOSendData[0] = 11;
        abIOSendData[1] = 22;
        abIOSendData[2] = 33;

        if ( ( sRet = DevExchangeIO( 0,
                                     0,
                                     sizeof(abIOSendData),
                                     &abIOSendData[0],
                                     0,
                                     sizeof(abIOReceiveData),
                                     &abIOReceiveData[0],
                                     100L)) != DRV_NO_ERROR) {
```

```
          // Error during data exchange
            printf( "DevExchangeIO  RetWert = %5d \n", sRet );
        } else {
          // Input data are stored in the abIOReceiveData
              // Check for specific device errors (VERY IMPORTEND)
          if ( (sRet = DevGetTaskState(.......)) != DRV_NO_ERROR) {
                 // Error by reading task state information

              } else {
                 // Check if one of the bus devices are faulty

            // Process input data...........
          }
        }
     } /* end while*/

     // Close the application
     /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */      /*
Signal board, application is not running */
     if ( (sRet = DevSetHostState(0,
                                 HOST_NOT_READY,
                                 0L)) != DRV_NO_ERROR) {
       printf( "DevSetHostState       RetWert = %5d \n", sRet );
     }

     /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
     /* Free board */
     if ( (sRet = DevExitBoard (0)) != DRV_NO_ERROR) {
       printf( "DevExitBoard        RetWert = %5d \n", sRet );
     }

     /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */      /* Close
driver */
     if ( (sRet = DevCloseDriver()) != DRV_NO_ERROR ) {
       printf( "DevCloseDriver      RetWert = %5d \n", sRet );
     }
     }
} /* end main*/
```

This example creates a send and a receive buffer. During the data exchange function call the data from the send buffer (abIOSendBuffer) are written to the DEVICE output process data area and the data from the input process data area are read to the receive buffer (abIOReceiveBuffer). As data buffers, there are fixed data area from 512 bytes for input and 512 bytes for output data used. The real size of the process image can be determine by the DevGetInfo(GET_DEV_INFO) function. This function returns the DPM size of the DEVICE as a multiple of 1024 Bytes (e.g. 2).

process image size = ((bDpmSize * 1024) -1024) /2

From the whole size (2 * 1024 Byte) there must be subtract 1024 Byte, which is the length of the last Kbytes (always reserved for message transfer and protocol independent data). This gives a value of 1024 Bytes, which must be divided by two (the size of the input and output process image is always equal.The synchronization mode for the exchange function (e.g. uncontrolled and so on) will be recognized by the `DevExchangeIO()` function and handled in the right manner.

Read out state information for all connected bus devices when using a master device, to find out if on of the bus devices has a malfunction. This is done by the use of `DevGetTaskState()`. The function must be called after each call to `DevExchangeIO()` to discover problems with particular devices (see also `DevExchangeIOErr()`).

The evaluation of the process data is up to the application. The exchange function only copys a data area (one byte up to the whole data area) from and to the device. Where the data for a particular device is located in the IO process image is defined by the system configuration.

It is also possible to read only one byte from the image. But be aware, depending on the sychronization mode (HOST Controlled, Buffered Data Transfer) , each data exchange by the HOST will result in a complete buffer exchange on the DEVICE. To prevent needless data transfers of unchanged data between the DPM and the internal data buffer of the DEVICE, we suggest to transfer as much data as possible with one `DevExchangeIO()` call to get the best system performence.The DevExchangeIO() function can be used to send and receive process data in one call or in two calls. Where one call writes output data and the other on reads input data. To prevent one of the functions, set the corresponding size parameter equal to zero.

5.4 The Demo Application

We have created demo applications which show the use of the driver.

- If you want to test our driver not in **X-Windows** Environment, there is a simple console demo program incuded in this package.
- For X-Windows system there is CIF Driver Setup and Test Program '**drvSu**' in the package. All of the driver functions are utilized in this application including functions for the message transfer and for reading/writing process images.

  The source code for this application is included, so it can help you understand how to integrate the driver into your application.

5.4.1 C-Example

The sample code demonstrates the initialization and the data transfer for a message and for process image exchange. This source code is available from the driver disk.

```
include "../usr-inc/cif_user.h"

/************************************************************************
/*  Mainprogram
/************************************************************************/
int main( void )
{
  unsigned short   usDevState, usHostState;
  short            sRet;
  MSG_STRUC        tMessage;
  unsigned char    tIOSendData[512];
  unsigned char    tIORecvData[512];

  /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
  /* Open the driver */
  if ( (sRet = DevOpenDriver()) != DRV_NO_ERROR) {
    printf( "DevOpenDriver       RetWert = %5d \n", sRet );

  /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
  /* Initialize board */
  } else if ( (sRet = DevInitBoard (0)) != DRV_NO_ERROR) {
    printf( "DevInitBoard        RetWert = %5d \n", sRet );

  /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
  /* Signal board, application is running */
  } else if ( (sRet = DevSetHostState( 0,           /* DeviceNumber */
                                       HOST_READY, /* Mode */
0L) != DRV_NO_ERROR) ) {
    printf( "DevSetHostState (HOST_READY) RetWert = %5d \n", sRet );


  } else {

     /*==================================================================
     /* Test Message transfer
     /*==================================================================
     /* Build a message */
     tMessage.rx         = 0x01;
     tMessage.tx         = 0x10;
     tMessage.ln         =   12;
     tMessage.nr         =    1;
     tMessage.a          =    0;
     tMessage.f          =    0;
     tMessage.b          =   17;
     tMessage.e          = 0x00;
     tMessage.daten[0]   =    1;
     tMessage.daten[1]   =    2;
     tMessage.daten[2]   =    3;
     tMessage.daten[3]   =    4;

    /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    /* Send a message */
    sRet = DevPutMessage ( 0,
                           (MSG_STRUC *)&tMessage,
5000L );
    printf( "  DevPutMessage       RetWert = %5d \n", sRet );

    /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    /* Receive a message */
    sRet = DevGetMessage ( 0,
                           sizeof(tMessage),
                           (MSG_STRUC *)&tMessage,
                           20000L );
```

```
     printf( "  DevGetMessage       RetWert = %5d \n", sRet );


    /*==================================================================
    /* Test for ExchangeIO
    /*==================================================================

    /* Write test data to Send buffer */
    tIOSendData.abSendData[0] = 0;
    tIOSendData.abSendData[1] = 1;
    tIOSendData.abSendData[2] = 2;
    tIOSendData.abSendData[3] = 3;

    /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    /* Run ExchangeIO */
    sRet = DevExchangeIO ( 0,
                           0,              /* usSendOffset    */
                           4,              /* usSendSize      */
                           &tIOSendData,   /* *pvSendData     */
                           0,              /* usReceiveOffset */
                           4,              /* usReceiveSize   */
                           &tIORecvData,   /* *pvReceiveData  */
                           100L );         /* ulTimeout       */

    printf( "DevExchangeIO RetWert = %5d \n", sRet );

   }
   /*- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
   /* Signal board, application is not running
   if ( (sRet = DevSetHostState( 0,
                                 HOST_NOT_READY,
                                 0L) != DRV_NO_ERROR) ) {
     printf( "DevSetHostState (HOST_NOT_READY) RetWert = %5d \n", sRet );

   }


   /*- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
   /* Close communication */
   sRet = DevExitBoard( 0 );
   printf( "DevExitBoard        RetWert = %5d \n", sRet );

   /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
   /* Close Driver */
   sRet = DevCloseDriver();
   printf( "DevCloseDriver      RetWert = %5d \n", sRet );
   return 0;
 }
```

6 The Application Programming Interface

All definitions for data structures, function prototypes and definitions are located in the user interface header file `cif_user.h`.

Note: Please notice, that the timer resolution on Linux system is 10ms. The use of timeout values lower than the given timer resolution will result in timeout periods between 0 the timer resolution.

6.1 API Functions Overview

Function Group	Function	Description
Installation	DevOpenDriver()	Links an application to the device driver
	DevCloseDriver()	Closes a link to the driver
	DevInitBoard()	Links an application to a board
	DevExitBoard()	Closes a link to a board
Device Control	DevReset()	Resets a board
	DevSetHostState()	Sets/Clears the information bit for HOST is running
	DevTriggerWatchDog()	Serves watchdog function of the board
Message Data Transfer	DevPutMessage()	Transfer a message to the board
	DevGetMessage()	Read a message from a board
	DevGetMBXState()	Read actual mailbox state
	DevGetMBXData()	Read actual mailbox data
IO Data Transfer	DevExchandeIO()	Put/Get IO data to/from a board
	DevExchandeIOEx()	Put/Get IO data to/from a COM module
	DevExchandeIOErr()	Put/Get IO data to/from a board including state information
	DevReadSendData()	Read/Send Rcv/Snd area of the DPM
Protocol,	DevPutTaskParameter()	Writes the parameter for a communication task
Information,	DevGetTaskParameter()	Reads the parameter from a communication task
Configuration	DevGetTaskState()	Read all task states from a board
Device Information	DevGetBoardInfo()	Read global board information
	DevGetBoardInfoEx()	Read board extended information
	DevGetInfo()	Read various information from a board
Other	DevReadWriteDPMData()	Read/Write the DPM directly
System function	DevDownload()	Firmware/Configuration download

6.2 DevOpenDriver()

Description:

If an application wants to communicate with a board, it must call this function first. This function checks if the device driver is available and opens a link to it. Once an link is opened, all other functions can be used. Call DevCloseDriver() to close the link.

```
short DevOpenDrive ();
```

Return value:

Value	Description
DRV_NO_ERROR	0 = no error

6.3 DevCloseDriver()

Description:

Close an open link to the device driver. An application has to call this function before it ends.

```
short DevCloseDriver ();
```

Return value:

Value	Description
DRV_NO_ERROR	0 = no error

6.4 DevGetBoardInfo()

Description:

With DevGetBoardInfo(), the user can read global information of all communication boards the device driver knows. `BOARD_INFO` data structure describes the board information data. This function can be used before opening a specific DEVICE with the DevInitBoard() function.

```
short DevGetBoardInfo (   BOARD_INFO    *pvData);
```

Parameter:

Type	**Parameter**	**Description**
BOARD_INFO *	pvData	Pointer to the user data buffer

Data structure:

```
typedef struct tagBOARD_INFO{
  unsigned char abDriverVersion[16];    // DRV version information
  struct {
    unsigned short  usBoardNumber;      // DRV board number
    unsigned short  usAvailable;        // DRV board is available
    unsigned long   ulPhysicalAddress;  // DRV physical DPM address
    unsigned short  usIrqNumber;        // DRV irq number
  } tBoard [MAX_DEV_BOARDS];
} BOARD_INFO;
```

Type	Parameter	Description
Unsigned short	usBoardNumber	Always 0
Unsigned short	usAvailable	0 = board not available; 1 = board available
Unsigned long	ulPhysicalAddress	Physical memory address
Unsigned short	usIrqNumber	Number of the hardware interrupt

Return value:

Value	Description
DRV_NO_ERROR	0 = no error

6.5 DevGetBoardInfoEx()

Description:

With DevGetBoardInfoEx(), the user can read global information of all communication boards the device driver knows. BOARD_INFOEX data structure which describes the board information data. This function can be used before opening a specific DEVICE with the DevInitBoard() function.

```
short DevGetBoardInfo (   BOARD_INFOEX  *ptBoardInfo);
```

Parameter:

Type	Parameter	Description
BOARD_INFOEX*	ptBoardInfoEx	Pointer to BOARD_INFOEx data structure

Data structure:

```
typedef struct tagBOARD_INFOEx{
  unsigned char abDriverVersion[16];    // DRV version information
  struct {
    unsigned short  usBoardNumber;      // DRV board number
    unsigned short  usAvailable;        // DRV board is available
    unsigned long   ulPhysicalAddress;  // DRV physical DPM address
    unsigned short  usIrqNumber;        // DRV irq number
    DRIVERINFO      tDriverInfo;        // Driver  info structure
    FIRMWAREINFO    tFirmware;          // Driver  info structure
    DEVINFO         tDeviceInfo;        // Device  info structure
    RCSINFO         tRcsInfo;           // RCS     info structure
    VERSIONINFO     tDriverInfo;        // Version info structure
  } tBoard [MAX_DEV_BOARDS];
} BOARD_INFOEX;
```

Type	Parameter	Description
Unsigned short	usBoardNumber	Always 0
Unsigned short	usAvailable	0 = board not available; 1 = board available
Unsigned long	ulPhysicalAddress	Physical memory address
Unsigned short	usIrqNumber	Number of the hardware interrupt
DRIVERINFO	tDriverInfo	See DevGetInfo() description
FIRMWAREINFO	tFirmware	See DevGetInfo() description
DEVINFO	tDeviceInfo	See DevGetInfo() description
RCSINFO	tRcsInfo	See DevGetInfo() description
VERSIONINFO	tDriverInfo	See DevGetInfo() description

Return value:

Value	Description
DRV_NO_ERROR	0 = no error

6.6 DevInitBoard()

Description:

After an application has opened a link to the device driver, it must call `DevInitBoard()` before it can start with the communication. `DevInitBoard()` tells the device driver that an application wants to work with a defined board. The device driver checks, if the board is physical available, if the board works properly and setup up all the internal state flags for the addressed board.

```
short DevInitBoard ( unsigned short   usDevNumber);
```

Parameter:

Type	Parameter	Description
Unsigned short	usDevNumber	Board number (0 . . 3)

Return value:

Value	Description
DRV_NO_ERROR	0 = no error

6.7 DevExitBoard()

Description:

If an application wants to end communication it has to call DevExitBoard(). for each board which has been opened by a previous call to `DevInitBoard().` These function frees all internal driver structures and unlink itself from the communication board.

```
short DevExitBoard ( unsigned short   usDevNumber);
```

Parameter:

Type	Parameter	Description
Unsigned short	usDevNumber	Board number (0 . . 3)

Return value:

Value	Description
DRV_NO_ERROR	0 = no error

6.8 DevPutTaskParameter()

Description:

This function hands over parameter to a task. This is only possible, if the protocol picks up the parameters of the DPM.

The parameters in the DPM will only be taken over from the tasks with the next WARMSTART.

```
short DevPutTaskParameter ( unsigned short  usDevNumber,
                            unsigned short  usNumber,
                            unsigned short  usSize,
                            void            *pvData);
```

Parameter:

Type	Parameter	Description
unsigned short	usDevNumber	Board number (0 . . 3)
unsigned short	usNumber	Number of the parameter area (1 . . 7)
unsigned short	usSize	Size of the parameter area and length of the data to be put
void*	pvData	Pointer to the user task parameters

Please notice, that you have to put the parameters in a structure according to the protocol. The user has to build his own structure definition. The driver do not check the parameters but it checks the length of the parameter structure. If the length of the user data exceed the maximum length, the function call fails with an error. Invalid parameters will be reported by the protocol.

Data structure:

```
typedef struct tagTASKPARAM {
  unsigned char   abTaskParameter[64];
} TASKPARAM;
```

Return value:

Value	Description
DRV_NO_ERROR	0 = no error

6.9 DevGetTaskParameter()

Description:

This function reads the task parameter area from a task.

```
short DevGetTaskParameter ( unsigned short  usDevNumber,
                            unsigned short  usNumber,
                            unsigned short  usSize,
                            void            *pvData);
```

Parameter:

Type	Parameter	Description
unsigned short	UsDevNumber	Board number (0 . . 3)
unsigned short	UsNumber	Task number (1, 2)
unsigned short	UsSize	Size of the user data buffer and length of the data to be read
void *	pvData	Pointer to the user data buffer

Please notice, that you get the parameters in a structure according to the protocol. The user has to build his own structure definition. The driver do not check the parameters but it checks the length of the parameter structure. If the length of the user data exceed the maximum length, the function call fails with an error.

Data structure:

```
typedef struct tagTASKPARAM {
  unsigned char   abTaskParameter[64];
} TASKPARAM;
```

Return value:

Value	Description
DRV_NO_ERROR	0 = no error

6.10 DevReset()

Description:

This function provokes a reset on a communication board. The passed parameter usMode switches a coldstart or a warmstart. The amount of the timeout ulTimeout depends on the used protocol and reset mode. A coldstart needs a longer time then a warmstart because there will be made a complete hardware check by the device operating system. Usually the time for a coldstart will be between 3 and 10 seconds, a warmstart needs between 2 and 8 seconds.

```
short DevReset ( unsigned short usDevNumber,
                 unsigned short usMode,
                 unsigned long  ulTimeout);
```

Parameter:

Type	Parameter	Description
unsigned short	usDevNumber	Board number (0 . . 3)
unsigned short	usMode	2 = COLDSTART, new initializing 3 = WARMSTART, initializing with parameters 4 = BOOTSTART, switches the board into bootstrap loader mode. COM modules use this mode to store user parameters
unsigned long	ulTimeout	Timeout

Return value:

Value	Description
DRV_NO_ERROR	0 = no error

6.11 DevSetHostState()

Description:

The DevSetHostState() function is used, to signal the communication board that a user application is running or not.

The utilization of the host state depends on the used communication protocol. Some of the message based and the I/O based protocols uses this state to signal a requesting station, no user application is running. I/O based protocol, such as InterBus S or PROFIBUS-DP, can use this state to shut down data transmission to other stations. On the most of the protocols, the use of the host state can be configured. A detailed description can be found in the corresponding protocol manual.

```
short DevSetHostState ( unsigned short usDevNumber,
                        unsigned short usMode,
                        unsigned long  ulTimeout);
```

Parameter:

Type	Parameter	Description
unsigned short	usDevNumber	Board number (0 . . 3)
unsigned short	usMode	0 = HOST_NOT_READY; 1 = HOST_READY
unsigned long	ulTimeout	Timeout in milliseconds; 0 = no timeout

The timeout parameter can be used by the user application to change the host state and wait until the communication state of the board has also changed. That means, if the host set HOST_READY and a timeout is configured, then the function returns, if the communication state of the board is ready. Otherwise a timeout occurs and the function returns with an error, which means, the board has not reached communication ready state. If the host set HOST_NOT_READY and a timeout is given, so the function will return, if the communication state of the board reaches not ready. If a timeout occurs, the communication state has not reached not ready and the function will return with an error. If no timeout is given, only the used host state will be written to the communication board. No further check will be done. The timeout period depends on the used bus system and varies between 100 ms up to several seconds.

Return value:

Value	Description
DRV_NO_ERROR	0 = no error

6.12 DevTriggerWatchdog()

Description:

The DevTriggerWatchdog() command can be used to check the device operating system for normal operation. The parameter function determines what action on the boards watchdog should be done (**WATCHDOG_START, WATCHDOG_STOP**). The function reads the **PcWatchDog** cell and write it to the **DevWatchDog** cell of the DPM. With writing a number unequal to zero in the **DevWatchDog** cell of the DPM, the watchdog function of the board is activated. Since the watchdog is activated, the application must trigger the watchdog within the time which is defined in the protocols database. The application must not generate a watchdog counter, because the operating system of the board increments the watchdog counter. This is done by giving an unequal number (1) in the **PcWatchDog**. The trigger function take this number and write it to the **DevWatchDog** cell. If the operating system reads a number unequal to zero from the **DevWatchDog** then it increments the number and write it back to the **PcWatchDog** cell. Every time the function is called, it returns the actual watchdog counter to the application. So, if the application reads the same counter value twice or more after the call to the trigger function, the board failed. To stop the watchdog, the function writes a 0 to the **DevWatchDog** cell. After this the boards operating system stops the watchdog checking.

```
short DevTriggerWatchDog ( unsigned short usDevNumber,
                           unsigned short usFunction,
                           unsigned short *usDevWatchDog);
```

Parameter:

Type	Parameter	Description
Unsigned short	UsDevNumber	Board number (0 . . 3)
Unsigned short	UsFunction	Function of the watchdog 0 = WATCHDOG_STOP 1 = WATCHDOG_START
Unsigned short*	usDevWatchDog	Pointer to a user buffer, where the watchdog counter value can be written to

Return value:

Value	Description
DRV_NO_ERROR	0 = no error

6.13 Message Transfer Functions

Following functions are defined for message transfer:

- `DevGetMBXState()`
- `DevPutMessage()`
- `DevGetMessage()`

6.13.1 DevGetMBXState()

Description:

This function reads the actual state of the host and device mailbox of a communication board.

You can use this function for writing applications to poll the device without waiting for device events.

```
short DevGetMBXState ( unsigned short usDevNumber,
                       unsigned short *pusDevMBXState,
                       unsigned short *pusHostMBXState);
```

Parameter:

Type	Parameter	Description
unsigned short	usDevNumber	Board number (0 . . 3)
unsigned short *	pusDevMBXState	Pointer to user buffer, to hold the device mailbox state 0 = DEVICE_MBX_EMPTY; 1 = DEVICE_MBX_FULL
unsigned long *	pusHostMBXState	Pointer to user buffer, to hold the host mailbox state 0 = HOST_MBX_EMPTY; 1 = HOST_MBX_FULL

Return value:

Value	Description
DRV_NO_ERROR	0 = no error

6.13.2 DevPutMessage()

Description:

This function sends (transfers) a message to the communication board. The function copies the number of data, given in the length entry (msg.ln) of the message structure and the message header.

If no timeout (`ulTimeout` = 0) is used, the function returns immediately. The return code shows if the function was able to write the message to the device or not. If a timeout (`ulTimeout` != 0) is used and the send mailbox of the device is empty, the message is written to the mailbox and the function returns also immediately. If the mailbox is full, the function will wait until the mailbox is free. If this does not happen during the timeout duration, the function returns with an error code. How the timeout is realized depends on the mode the DEVICE is configured. Polling mode will run a loop in the driver while waiting the timeout duration In interrupt mode the calling application will block to free the CPU for other work..

```
short DevPutMessage (     unsigned    short usDevNumber,
                          MSG_STRUC   *ptMessage,
                          unsigned    long  ulTimeout);
```

Parameter:

Type	Parameter	Description
unsigned short	usDevNumber	Board number (0 . . 3)
`MSG_STRUC *`	`ptMessage`	Pointer to the message data
unsigned long	ulTimeout	Timeout in milliseconds; 0 = no timeout

Return value:

Value	Description
DRV_NO_ERROR	0 = no error

The message have to be compatible to the message format and it must be consistent, according to the protocol. The structure of the standard message is located in the users interface header file.

Message structure:

```
#pragma pack(1)
// max. length is 288 Bytes, max. message length is 255 + 8 Bytes
typedef struct tagMSG_STRUC {
  unsigned char   rx;                          // Receiver
  unsigned char   tx;                          // Transmitter
  unsigned char   ln;                          // Length
  unsigned char   nr;                          // Number
  unsigned char   a;                           // Answer
  unsigned char   f;                           // Fault
  unsigned char   b;                           // Command
  unsigned char   e;                           // Extension
  unsigned char   data[ 255];                  // Data
    unsigned char   dummy[25];                 // for compatibility with older
                                                      // versions
} MSG_STRUC;
#pragma pack()
```

Note: Notice, for more information about the message structure refer to the corresponding manual.

6.13.3 DevGetMessage()

Description:

This function reads a message out from a communication board and puts it into the data buffer that is given by the user. The function checks if the message fits in the users data buffer. This is done by comparing the parameter usSize with the length which is given in the message structure. If the message doesn't fit, the function will fail and returns an error.

If no timeout (ulTimeout = 0) is used, the function returns immediately. The return code shows if the function was able to read a message from the device or not.
If a timeout (ulTimeout != 0) is used and a message is available, the function reads the message and returns also immediately. If no message is available, the function will wait until a message is available. If this does not happen during the timeout duration, the function returns with an error code.

How the timeout is realized depends on the mode the DEVICE is configured. Polling mode will run a loop in the driver while waiting the timeout duration
In interrupt mode the calling application will blocked to free the CPU for other work..

```
short DevGetMessage ( unsigned  short usDevNumber,
                      unsigned  short usSize,
                      MSG_STRUC       *ptMessage,
                      unsigned  long  ulTimeout);
```

Parameter:

Type	Parameter	Description
unsigned short	usDevNumber	Board number (0 . . 3)
unsigned short	usSize	Size of the user data buffer (maximum length to be read)
MSG_STRUC *	ptMessage	Pointer to the message data
unsigned long	ulTimeout	Timeout in milliseconds; 0 = no timeout

Notice, the size of the user data buffer has to be large enough to hold all the data of a message. The maximum length of a message can be taken from the message structure in the users interface header file.

Message structure:

```
#pragma pack(1)
typedef struct tagMSG_STRUC {// max. 288 Bytes, max. msg len 255 + 8 Bytes
  unsigned char   rx;           // Receiver
  unsigned char   tx;           // Transmitter
  unsigned char   ln;           // Length
  unsigned char   nr;           // Number
  unsigned char   a;            // Answer
  unsigned char   f;            // Fault
  unsigned char   b;            // Command
  unsigned char   e;            // Extension
  unsigned char   data[ 255];   // Data
  unsigned char   dummy[25];    // for compatibility with older versions
} MSG_STRUC;
#pragma pack()
```

Return value:

Value	description
DRV_NO_ERROR	0 = no error

6.14 DevGetTaskState()

Description:

This function reads one of the task state areas of a DEVICE. The data will be transferred into the user data buffer. The function copies the number of data, given in the parameter `usSize`.

```
short DevGetTaskState ( unsigned short     usDevNumber,
                        unsigned short     usNumber,
                        unsigned short     usSize,
                        void               *pvData);
```

Parameter:

Type	Parameter	Description
Unsigned short	usDevNumber	Board number (0 . . 3)
Unsigned short	usNumber	Number of the state area (1, 2)
Unsigned short	usSize	Size of the user data buffer (maximum length to be read)
Void *	pvData	Pointer to the user data buffer

To handle the data, please use the structures given by the protocols.

Notice, the maximum size of the area given by the user can be taken from the task parameter structure in the users interface header file.

Data structures:

```
typedef struct tagTASKSTATE {
  unsigned char   abTaskState[64];
} TASKSTATE;
```

Return value:

Value	Description
DRV_NO_ERROR	0 = no error

6.15 DevGetInfo()

Description:

This function reads the various information out from a communication board and the driver internal state information for a board. The information that can be read are as followed:

- Driver state information	GET_DRIVER_INFO
- Board version information	GET_VERSION_INFO
- Board firmware information	GET_FIRMWARE_INFO
- Task information area	GET_TASK_INFO
- Board operation system information	GET_RCS_INFO
- Device information area	GET_DEV_INFO
- Device IO information	GET_IO_INFO
- Device IO send data	GET_IO_SEND_DATA

The function copies the number of data, given in the parameter `usSize`. For data structure definitions look up in the user interface header file.

```
short DevGetInfo (  unsigned short    usDevNumber,
                    unsigned short    usInfoArea,
                    unsigned short    usSize,
                    void              *pvData);
```

Parameter:

Type	Parameter	Description
Unsigned short	usDevNumber	Board number (0 . . 3)
Unsigned short	usInfoArea	Defines which area have to be read 1 = GET_DRIVER_INFO 2 = GET_VERSION_INFO 3 = GET_FIRMWARE_INFO 4 = GET_TASK_INFO 5 = GET_RCS_INFO 6 = GET_DEV_INFO 7 = GET_IO_INFO 8 = GET_IO_SEND_DATA
Unsigned short	usSize	Size of the user data buffer and number of bytes to be read
Void *	pvData	Pointer to the user data buffer

Defined data structures:

// GETINFO information definitions

```
#define GET_DRIVER_INFO     1
// Internal driver state information structure
typedef struct tagDRIVERINFO{
  unsigned long ulOpenCnt;              // DevOpen() counter
  unsigned long ulCloseCnt;             // DevClose() counter (not used)
  unsigned long ulReadCnt;              // Number of DevGetMessage() commands
  unsigned long ulWriteCnt;             // Number of DevPutMessage() commands
  unsigned long ulIRQCnt;       // Number of board interrupts
  unsigned char bInitMsgFlag;           // Actual init state
  unsigned char bReadMsgFlag;           // Actual read mailbox state
  unsigned char bWriteMsgFlag;  // Actual write mailbox state
  unsigned char bLastFunction;  // Last driver function
  unsigned char bWriteState;            // Actual write command state
  unsigned char bReadState;             // Actual read command state
  unsigned char bHostFlags;             // Actual host flags
  unsigned char bMyDevFlags;            // Actual device flags
  unsigned char bExIOFlag;              // Actual IO flags
  unsigned long ulExIOCnt;              // DevExchangeIO() counter
} DRIVERINFO;


#define GET_VERSION_INFO    2
// Serial number and OS versions information
typedef struct tagVERSIONINFO {
  unsigned long   ulDate;       // Manufactor date    (BCD coded)
  unsigned long   ulDeviceNo;           // Device number             (BCD coded)
  unsigned long   ulSerialNo;           // Serial number             (BCD coded)
  unsigned long   ulReserved;           // reserved
  unsigned char   abPcOsName0[4];       // Operating system code 0   (ASCII)
  unsigned char   abPcOsName1[4];       // Operating system code 1   (ASCII)
  unsigned char   abPcOsName2[4];       // Operating system code 2   (ASCII)
  unsigned char   abOemIdentifier[4]; // OEM reserved               (ASCII)
} VERSIONINFO;



#define GET_FIRMWARE_INFO   3
// Device firmware information
typedef struct tagFIRMWAREINFO {
  unsigned char   abFirmwareName[16];   // Firmware name             (ASCII)
  unsigned char   abFirmwareVersion[16];       // Firmware version   (ASCII)
} FIRMWAREINFO;



#define GET_TASK_INFO       4
// Device task information
typedef struct tagTASKINFO {
  struct  {
    unsigned char  abTaskName[8];              // Taskname                  (ASCII)
    unsigned short usTaskVersion;              // Task version              (number)
    unsigned char  bTaskCondition;             // Actual task state
    unsigned char  abreserved[5];              // reserved
  } tTaskInfo [7];
} TASKINFO;
```

```
#define GET_RCS_INFO        5
// Device operating system (RCS) information
typedef struct tagRCSINFO {
  unsigned short usRcsVersion;  // Device RCS version        (number)
  unsigned char  bRcsError;             // Operating system errors
  unsigned char  bHostWatchDog; // Host watchdog value
  unsigned char  bDevWatchDog;  // Device watchdog value
  unsigned char  bSegmentCount; // RCS segment free counter
  unsigned char  bDeviceAdress; // RCS device base address
  unsigned char  bDriverType;           // RCS driver type
} RCSINFO;



#define GET_DEV_INFO        6
// Device description
typedef struct tagDEVINFO {
  unsigned char  bDpmSize;              // Device DPM size (2,8..)   (number)
  unsigned char  bDevType;              // Device type                      (number)
  unsigned char  bDevModel;             // Device model                     (number)
  unsigned char  abDevIdentifier[3];// Device identification  (ASCII)
} DEVINFO;


#define GET_IO_INFO         7
// Device exchange IO information
typedef struct tagIOINFO {
  unsigned char   bComBit;              // Actual state of the COM bit (0,1)
  unsigned char   bIOExchangeMode;      // Actual data exchange mode  (0..5)
  unsigned long   ulIOExchangeCnt;      // Exchange IO counter
} IOINFO;
```

Return value:

Value	Description
DRV_NO_ERROR	0 = no error

6.16 Process Data Transfer Functions

Following functions are defined for process data transfer:

-**`DevExchangeIO()`**
Is the standard function for the data transfer of process image datas. Only general bus errors are detected by this function. To get error information about specific devices, the function DevGetTaskState() must be used after each call to `DevExchangeIO()` to read the task information field.

`-DevExchangeIOErr()`
Is an extension of the DevExchangeIO() function. It contains the COMSTATE structure as an parameter, where device specific data will be transferred by each call to the function. No additional call of DevGetTaskState() is required.

`-DevExchangeIOEx()`
`This function is a special function to work with COM modules.`

-**`DevReadSendData()`**
This function can be used to read back the send process image from a device

ATTENTION: **By using `DevExchangeIO()`it is not possible for master devices to recognize the fault of a specific bus device. Only global errors like whole bus disruptions or communication breaks to all configured device will be indicated by this function.**
To get specific device fault, the application must read the "TaskState-Field", where device specific datas are located.

6.16.1 DevExchangeIO()

Description:

The `DevExchangeIO()` function is used, to send I/O data to and receive I/O data from a communication board. This function is able to send and receive I/O data at once. If one of the size parameter is set to zero, no action will be taken for the corresponding function. This means, if `usSendSize` is set to zero, send data will not be written to the board. If `usReceiveSize` is set to zero, receive data will not be read from the board.
The user can wait until a complete action is done, by the use of `ulTimeout`. If an timeout occurs, the function will return with an error. If no timeout is given, the function will return immediately.
The function will automatically recognize the synchronization mode of the process data transfer and handle it in the defined way.

ATTENTION: Only general bus errors are detected by this function.Use DevGetTaskState() after each call to DevExchangeIO() to read the task information field and to check device specific errors.

```
short DevExchangeIO (unsigned short   usDevNumber,
                     unsigned short   usSendOffset,
                     unsigned short   usSendSize,
                     void             *pvSendData,
                     unsigned short   usReceiveOffset,
                     unsigned short   usReceiveSize,
                     void             *pvReceiveData,
                     unsigned long    ulTimeout);
```

Parameter:

Type	Parameter	Description
unsigned short	usDevNumber	Board number (0 . . 3)
unsigned short	usSendOffset	Byte offset in the send IO data area of the communication board
unsigned short	usSendSize	Length of the send IO data
void *	pvSendData	Pointer to the user send data buffer
unsigned short	usReceiveOffset	Byte offset in the receive IO data area of the communication board
unsigned short	usReceiveSize	Length of the receive IO data
void *	pvReceiveData	Pointer to the user receive data buffer
unsigned long	ulTimeout	Timeout in milliseconds; 0 = no timeout

Return value:

Value	description
DRV_NO_ERROR	0 = no error

6.16.2 DevExchangeIOErr()

Description:

`DevExchangeIOErr()` is an extension of the `DevExchangeIO()` function. The handling for sending and receiving I/O data acts in the same way like in the `DevExchangeIO()` function. Furthermore, the function has an additional parameter which holds state information according to the configured bus devices. This information is only available on master DEVICEs.

Normally the DEVICE will set its communication ready bit (COM flag) if at least one of the configured bus devices is connected and running properly. If more modules are configured, the COM flag can not signal an error for a specific device. The COM flag is only able to indicate global failures like whole bus disruptions or communication breaks to all configured devices. In this case the state field information can be used to detect errors of a specific bus device.

Please check, if the DEVICE firmware of the master device supports the several modes of state field handling.

```
short DevExchangeIOErr (  unsigned short    usDevNumber,
                          unsigned short    usSendOffset,
                          unsigned short    usSendSize,
                          void              *pvSendData,
                          unsigned short    usReceiveOffset,
                          unsigned short    usReceiveSize,
                          void              *pvReceiveData,
                          COMSTATE          *ptState,
                          unsigned long     ulTimeout);
```

Parameter:

Type	Parameter	Description
unsigned short	usDevNumber	Board number (0 . . 3)
unsigned short	usSendOffset	Byte offset in the send IO data area of the communication board
unsigned short	usSendSize	Length of the send IO data
void *	pvSendData	Pointer to the user send data buffer
unsigned short	usReceiveOffset	Byte offset in the receive IO data area of the communication board
unsigned short	usReceiveSize	Length of the receive IO data
void *	pvReceiveData	Pointer to the user receive data buffer
COMSTATE *	ptState	Pointer to the user COMSTATE buffer
unsigned long	ulTimeout	Timeout in milliseconds; 0 = no timeout

Return value:

Value	description
DRV_NO_ERROR	0 = no error

COMSTATE structure definition:

```
// Communication state field structure
typedef struct tagCOMSTATE {
  GLD16U      usMode;                 // Actual mode
  GLD16U      usStateFlag;            // State flag
  GLD8U       abState[64];            // State area
} COMSTATE;
```

The COMSTATE structure can be transferred on each function call.

- usMode — Defines the actual configured transfer mode of the state field
  0xFF = Not supported by the firmware
  3 = Cyclic transfer of the state field including the state error flag (usStateFlag)
  4 = Event driven transfer of the state field including the usStateFlag
- usStateFlag — 0 = No entrys in the state field (abState[])
  1 = Entrys in the state available
- abState[64] — Buffer of the actual state field. Refer to the protocol interface manual for a description of the state buffer.

Example:

```
  // Read process image and state field information
  if ( (sRet = DevExchangeIOErr( usBoardNumber,
                                 0,
                                 0,
                                 NULL,
                                 usReadOffset,
                                 usReadSize,
                                 &abIOReadData[0],
                                 &tComState,
                                 100L)) == DRV_NO_ERROR) {
    // Check state field transfer mode
    switch ( tComState.usMode) {
      case STATE_MODE_3:
        // Check state field usStateFlag signals entries
        if ( tComState.usStateFlag != 0) {
          // Show COM errors
        }
        break;
      case STATE_MODE_4:
        // Check state field usStateFlag signals new entries
        if ( tComState.usStateFlag != 0) {
          // Show COM errors
        }
        break;
      default:
        // State mode unknown or not implemented
        // Read the task state field by yourself
        if ( (sRet = DevGetTaskState(....) ) != DRV_NO_ERROR) {
          // Error by reading the task state
        }
        break;
    } /* end switch */
  }
```

6.16.3 DevExchangeIOEx()

Description:

The **DevExchangeIOEx()** function is created for the use with COM mod-ules. It works in the same way like the **DevExchangeIO()** function, except the data transfer mode must be defined by the application.

COM modules are normally not able to signal the actual data transfer modes to the device driver, which means the driver can not decide how to act with the DPM. Therefore the **evExchangeIOEx()** function gets a new parameter which tells the driver how to handle the DPM.

The configuration of the COM modules are done by writing **WARMSTRART** pa-rameters to the board. During configuration, the user defines the IO data transfer mode. The configured mode must be given the **evExchangeIOEx()** function to make sure the driver handles the DPM in the right manner.

```
short DevExchangeIOEx ( unsigned short      usDevNumber,
                        unsigned short      usMode,
                        unsigned short      usSendOffset,
                        unsigned short      usSendSize,
                        void                *pvSendData,
                        unsigned short      usReceiveOffset,
                        unsigned short      usReceiveSize,
                        void                *pvReceiveData,
                        unsigned long       ulTimeout);
```

Parameter:

Type	Parameter	Description
unsigned short	usDevNumber	Board number (0 . . 3)
unsigned short	usMode	Data transfer mode (0 . . 4)
unsigned short	usSendOffset	Byte offset in the send IO data area of the communication board
unsigned short	usSendSize	Length of the send IO data
unsigned char *	pvSendData	Pointer to the user send data buffer
unsigned short	usReceiveOffset	Byte offset in the receive IO data area of the communication board
unsigned short	usReceiveSize	Length of the receive IO data
unsigned char *	pvReceiveData	Pointer to the user receive data buffer
unsigned long	ulTimeout	Timeout in milliseconds; 0 = no timeout

Return value:

Value	description
DRV_NO_ERROR	0 = no error

6.16.4 DevReadSendData()

Description:

The `DevReadSendData()` function is used, to read back send data which are written to send data area with the function `DevExchangeIO()`.
This function can be used by applications to update the user input after the data are successfully written to the communication board.

```
short DevReadSendData (   unsigned short    usDevNumber,
                          unsigned short    usOffset,
                          unsigned short    usSize,
                          void              *pvSendData);
```

Parameter:

Type	Parameter	Description
unsigned short	usDevNumber	Board number (0 . . 3)
unsigned short	usOffset	Byte offset in the send IO data area of the communication board
unsigned short	usSize	Length of the send IO data to be read
`void *`	pvSendData	Pointer to the user send data buffer

Return value:

Value	description
DRV_NO_ERROR	0 = no error

6.16.5 DevReadWriteDPMData()

Description:

The `DevReadSendData()` function is used, to read back send data which are written to send data area with the function `DevExchangeIO()`.
This function can be used by applications to update the user input after the data are successfully written to the communication board.

```
short DevReadSendData (   unsigned short    usDevNumber,
                          unsigned short    usMode,
                          unsigned short    usOffset,
                          unsigned short    usSize,
                          void              *pvData);
```

Parameter:

Type	Parameter	Description
unsigned short	usDevNumber	Board number (0 . . 3)
unsigned short	usMode	1 = PARAMETER_READ 0 = PARAMETER_WRITE
unsigned short	usOffset	Byte offset in DPM of the communication board (0..1022)
unsigned short	usSize	Length of the data to be read/written
void *	pvData	Pointer to the user data buffer

The structure definition RAWDATA can be used as a data buffer definition.

```
// Device raw data structure
typedef struct tagRAWDATA {
  unsigned char   abRawData[1022];             // Definition of the last KByte
} RAWDATA;
```

Return value:

Value	description
DRV_NO_ERROR	0 = no error

6.16.6 DevDownload()

Description:

The `DevDownload()` function can be used to either load a firmware or configuration file to the hardware.

The whole data transfer will be executed in the download function. Therefore, the function loads the file into the memory and transfers it from the memory to the hardware. The transfer function is running in a “loop”, so no other activity during a download is possible.

Firmware files must have a correct file extensions, which is checked in the download function. Configuration files will be checked by the operating system and rejected, if the database name is not known to the firmware.

```
short DevDownload ( unsigned short    usDevNumber,
                    unsigned short    usMode,
                    unsigned char     *pszFileName,
                    DWORD             *pdwBytes);
```

Parameter:

Type	Parameter	Description
unsigned short	usDevNumber	Board number (0 . . 3)
unsigned short	usMode	1 = FIRMWARE_DOWNLOAD 2 = CONFIGURATION_DOWNLOAD
unsigned *char	pszFileName	Pointer to the filename with or without a complete path description. This must be a multibyte string zero terminated.
DWORD *	pdwBytes	Pointer to a dword value which receives the number of bytes transferred to the hardware

Return value:

Value	Description
DRV_NO_ERROR	0 = no error

7 Error Numbers

7.1 List of Error Numbers

The column hint shows if there are additional information. If 'Yes' then see chapter *hints to error numbers*, which is the next chapter.

Value	Parameter	Description	Hint
0	DRV_NO_ERROR	No error	
-1	DRV_BOARD_NOT_INITIALIZED	DRIVER Board not initialized	yes
-2	DRV_INIT_STATE_ERROR	DRIVER Error in internal init state	
-3	DRV_READ_STATE_ERROR	DRIVER Error in internal raed state	
-4	DRV_CMD_ACTIVE	DRIVER Command on this channel is active	
-5	DRV_PARAMETER_UNKNOWN	DRIVER Unknown parameter in function occurred	
-6	DRV_WRONG_DRIVER_VERSION	DRIVER Version is incompatible with API	yes
-7	DRV_PCI_SET_CONFIG_MODE	DRIVER Error during PCI set config mode	
-8	DRV_PCI_READ_DPM_LENGTH	DRIVER Could not read PCI dual port memory length	
-9	DRV_PCI_SET_RUN_MODE	DRIVER Error during PCI set run mode	
-11	DRV_DEV_NOT_READY	DEVICE Not ready (ready flag failed)	yes
-12	DRV_DEV_NOT_RUNNING	DEVICE Not running (running flag failed)	yes
-13	DRV_DEV_WATCHDOG_FAILED	DEVICE Watchdog test failed	
-14	DRV_DEV_OS_VERSION_ERROR	DEVICE Signals wrong OS version	yes
-16	DRV_DEV_MAILBOX_FULL	DEVICE Send mailbox is full	
-17	DRV_DEV_PUT_TIMEOUT	DEVICE PutMessage timeout	yes
-18	DRV_DEV_GET_TIMEOUT	DEVICE GetMessage timeout	yes
-19	DRV_DEV_GET_NO_MESSAGE	DEVICE No message available	
-20	DRV_DEV_RESET_TIMEOUT	DEVICE RESET command timeout	yes
-21	DRV_DEV_NO_COM_FLAG	DEVICE COM-flag not set	yes
-22	DRV_DEV_EXCHANGE_FAILED	DEVICE IO data exchange failed	
-23	DRV_DEV_EXCHANGE_TIMEOUT	DEVICE IO data exchange timeout	yes
-24	DRV_DEV_COM_MODE_UNKNOWN	DEVICE IO data mode unknown	
-25	DRV_DEV_FUNCTION_FAILED	DEVICE Function call failed	
-26	DRV_DEVDPMSIZE_MISMATCH	DEVICE DPM size differs from configuration	
-27	DRV_DEV_STATE_MODE_UNKNOWN	DEVICE State mode unknown	
-30	DRV_USER_OPEN_ERROR	USER Driver not open (device driver not loaded)	yes
-31	DRV_USER_INIT_DRV_ERROR	USER Can’t connect with device	
-32	DRV_USER_NOT_INITIALIZED	USER Board not initialized (DevInitboard() not called)	
-33	DRV_USER_COM_ERR	USER IOCTRL function failed	yes

-34	DRV_USER_DEV_NUMBER_INVALID	USER Parameter DeviceNumber invalid	
-35	DRV_USER_INFO_AREA_INVALID	USER Parameter InfoArea unknown	
-36	DRV_USER_NUMBER_INVALID	USER Parameter Number invalid	
-37	DRV_USER_MODE_INVALID	USER Parameter Mode invalid	
-38	DRV_USER_MSG_BUF_NULL_PTR	USER NULL pointer assignment	
-39	DRV_USER_MSG_BUF_TOO_SHORT	USER Message buffer too short	
-40	DRV_USER_SIZE_INVALID	USER Parameter Size invalid	
-42	DRV_USER_SIZE_ZERO	USER Parameter Size with zero length	
-43	DRV_USER_SIZE_TOO_LONG	USER Parameter Size too long	
-44	DRV_USER_DEV_PTR_NULL	USER Device address is a NULL pointer	
-45	DRV_USER_BUF_PTR_NULL	USER Pointer to buffer is a NULL pointer	
-46	DRV_USER_SENDSIZE_TOO_LONG	USER Parameter SendSize too long	
-47	DRV_USER_RECVSIZE_TOO_LONG	USER Parameter ReceiveSize too long	
-48	DRV_USER_SENDBUF_PTR_NULL	USER Pointer to send buffer is a NULL pointer	
-49	DRV_USER_RECVBUF_PTR_NULL	USER Pointer to receive buffer is a NULL pointer	

-100	DRV_USER_FILE_OPEN_FAILED	USER File not opened	
-101	DRV_USER_FILE_SIZE_ZERO	USER File size zero	
-102	DRV_USER_FILE_NO_MEMORY	USER not enough memory to load file	
-103	DRV_USER_FILE_READ_FAILED	USER File read failed	
-104	DRV_USER_INVALID_FILETYPE	USER File type invalid	
-105	DRV_USER_FILENAME_INVALID	USER File name not valid	
>= 1000	RCS_ERROR	Board operation system errors will be passed with this offset (e.g. error 1234 means RCS error 234). Only if a ready fault occurred during board initialization.	

7.2 Hints to Error Numbers

This chapter contains more information about possible reasons to certain error numbers.

Error: -1
The communication board is not initialized by the driver.
No or wrong configuration found for the given board.
- Check the driver configuration
- Driver function used without calling DevOpenDriver() first

Error: -6
The device driver version does not corresponds to the driver API version

- Make sure to use the same version of the device driver and the driver API

Error: -11
Board is not ready.
This is a general error, the board has a hardware malfunction.

Error: -12
At least one task is not initialized. The board is ready but not all tasks are running.
- No data base is loaded into the device
- Wrong parameter that causes that a task can't initialize. Use ComPro menu *Online-task-version*.

Error: -14
No license code found on the communication board.
- Device has no license for the used operating system or customer software.
- No firmware or no data base on the device loaded.

Error: -17
No message could be send during the timeout period given in the `DevPutMessage()` function.
- Using device interrupts
Wrong or no interrupt selected. Check interrupt on the device and in driver registration. They have to be the same!. Interrupt already used by an other PC component.
- Device internal segment buffer full
`PutMessage()` function not possible, because all segments on the device are in use. This error occurs, when only `PutMessage()` is used but not `GetMessage()`.
- HOST flag not set for the device
No messages are taken by the device. Use DevSetHostState() to signal a board an application is available.

Error: -18
No message received during the timeout period given in the `DevGetMessage()` function.
- Using device interrupts
Wrong or no interrupt selected. Check interrupt on the device and in driver registration. They have to be the same!. Interrupt already used by an other PC component.
- The used protocol on the device needs longer than the timeout period given in the `DevGetMessage()` function

Error: -20
The device needs longer than the timeout period given in the `DevReset()` function
- Using device interrupts
This error occurs when for example interrupt 9 is set in the driver registration but no or a wrong interrupt is jumpered on the device (=device in pollmode).
Interrupt already used by an other PC component.
- The timeout period can differ between fieldbus protocols

Error: -21
The device can not reach communication state.
- Device not connected to the fieldbus
- No station found on the fieldbus
- Wrong configuration on the device

Error: -23
The device needs longer than the timeout period given in the `DevExchangeIO()` function.
- Using device interrupts
Wrong or no interrupt selected. Check interrupt on the device and in driver registration. They have to be the same!. Interrupt already used by an other PC component.

Error: -30
The device driver could not be opened.
- Device driver not installed
- Wrong parameters in the driver configuration
If the driver finds invalid parameters for a communication board and no other boards with valid parameters are available, the driver will not be loaded.

Error: -33
A driver function could not be called. This is an internal error between the device driver and the API.
- Make sure to use a device driver and a API with the same version.
- An incompatible old driver API is used.

8 Development Environments

As we began with the CIF Device Driver code conversion for the Linux, the kernel 2.2.10 was the actual one. With the subsequent kernel development and their distribution we tried to test and/or adjust the code to assure that our driver goes step by step with this evolutionarty kernel development.

Please, consult the Chapter „**The Driver Versions**“ for more information.

The driver represents 32-bit kernel driver and runs in kernel space. It is implemented as a character device driver, the code is written in **C** and compiled with **gcc** compiler.

The Driver Setup and Test program was developed with **GTK+**, version 1.2.8.

9 Copyright