Final thesis

Implementation of a Profibus agent for the Proview process control system

by

Ferdinand Hauck

LIU-IDA/LITH-EX-G--09/004--SE

2009-06-03
Final thesis

Implementation of a Profibus agent for the Proview process control system

Performed for SSAB Oxelösund
by

Ferdinand Hauck

LIU-IDA/LITH-EX-G--09/004--SE
2009-06-03

Supervisor: Robert Karlsson
Examiner: Petru Eles
Abstract

Proview is an open source system for process control, developed at SSAB Oxelösund. A wide range of IO systems are already supported by Proview. This thesis documents the implementation of support for the Profibus DP master board Hilscher CIF 50-PB.

Most of the features of the board are supported by the agent. The agent is intended to support up to four boards per system. Adding support for additional types of Profibus DP master boards from Hilscher to the agent should be straightforward.

This thesis covers some technical aspects of the Profibus DP technology and also provides some background of Proview’s IO system. The design decisions behind the Hilscher agent implementation will be discussed. All functions and data structures of the implementation are documented. Testing of the implementation is also included in the thesis.

The thesis may also be used as technical documentation for the agent implementation.

Finally, we look at the strengths and shortcomings of the completed agent implementation.
# Contents

1 Introduction 
   1.1 Background .............................................. 1
   1.2 Introduction to Proview ................................ 1
   1.3 IO systems in Proview ................................... 2
      1.3.1 Agent interface .................................... 3
   1.4 About Profibus ........................................... 3
   1.5 Profibus DP ............................................... 4
   1.6 Generic station description ............................. 4
   1.7 Diagnostics ............................................... 5
   1.8 Profibus DP in Proview .................................. 5
   1.9 Hilscher CIF 50-PB specifications ...................... 5
   1.10 Hilscher API ............................................. 6
   1.11 Requirements ........................................... 6
   1.12 Motivation ............................................... 7

2 Problem analysis ............................................. 8
   2.1 Online configuration ..................................... 8

3 Implementation ............................................... 9
   3.1 Macros ................................................... 10
   3.2 Data structures .......................................... 10
      3.2.1 Local structure .................................... 10
      3.2.2 Known boards table ................................ 11
   3.3 Initialization and termination functions .............. 12
   3.4 Error logging functions ................................. 15
3.5 IO exchange and diagnostics functions .......................... 16
3.6 Proview class ......................................................... 17
3.7 Proview interface ...................................................... 17
3.8 Using the board ...................................................... 17
3.9 Dual port memory allocation ....................................... 18

4 Testing ........................................................................... 20
   4.1 Basic functional test ............................................... 20
   4.2 Interoperability with Profiboard agent ....................... 20
   4.3 Dual port memory allocation code ............................... 21
   4.4 Watchdog .............................................................. 22
   4.5 Diagnostics ........................................................... 22
   4.6 Flashing ............................................................... 23
   4.7 Multiple calls to DevOpenDriver ............................... 23

5 Conclusion and future work .......................................... 24

Bibliography .................................................................. 26

A Terms and abbreviations ............................................. 27
Chapter 1

Introduction

In this chapter we will first provide a background to the thesis work. After that, an overview of the involved systems and components will be presented. Finally, we will look into those systems and components.

1.1 Background

SSAB Oxelö sund is a leading producer of steel plate [1]. The company focuses on the high-end segment, producing quenched and tempered steels. The production line at SSAB Oxelö sund is capable of producing the steel plates from raw materials.

Obviously, such a plant is very complicated and involves many different machines. Moreover, every major piece of machinery requires a control system. Since the products are produced in a line, at least the adjacent control systems normally need to communicate with each other.

To solve this issue, the engineers at SSAB Oxelö sund developed a distributed control system named Proview.

1.2 Introduction to Proview

Proview is an open source system for process control [2]. The system is released under the GNU General Public License version 2. It is developed by SSAB Oxelö sund and is used to control their plant.

One of the central components of the system is the soft PLC, a software realization of a programmable logic controller. The soft PLC is linked to an IO system of choice. Now, the PLC may use the IO system to read values
of sensors and control the process using actuators, which are also attached to the IO system.

Proview’s build system mainly uses standard tools such as GNU Make and GCC. Invoking Proview’s build system can be done by calling make directly, however for development purposes it is recommended to use the shell function pwre [3] instead. This function provides better control of the building process and also helps with other build related tasks.

Proview’s IO system is implemented in the C language. This is also the language used for the Hilscher agent’s implementation.

1.3 IO systems in Proview

The purpose of the IO system is to provide a means for the PLC to read input channels and set control values. These channels may be of different types depending on their use. They may be digital channels (for example to sense whether a button has been pushed), analog channels (for measuring a distance or level) or the number of pulses counted over some time interval (to measure motor speeds for example). Also the actuators may accept different types of data, such as binary or integer values.

The native IO system for Proview is based on Q-bus, which is one of the bus technologies that was formerly used with the DEC PDP-11. This has some impact on other IO systems as these are mapped to Q-bus IO entities. When looking at a local Q-bus IO system, one recognizes that this is a three tier hierarchy consisting of the levels rack, card and channel [4]. At the top of the hierarchy is the rack level, this represents a rack of cards. The card level represents an IO card with the actual IO channels.

For a distributed IO system like Profibus DP, there is one more level to consider, the agent level. The agent level is situated above the rack level.

Profibus uses different terminology to designate its hierarchy levels; these are master, slave, module and channel. The mapping of Proview levels to Profibus entities is shown in table 1.1.

<table>
<thead>
<tr>
<th>Proview</th>
<th>Profibus</th>
</tr>
</thead>
<tbody>
<tr>
<td>Agent</td>
<td>Master</td>
</tr>
<tr>
<td>Rack</td>
<td>Slave</td>
</tr>
<tr>
<td>Card</td>
<td>Module</td>
</tr>
<tr>
<td>Channel</td>
<td>Channel</td>
</tr>
</tbody>
</table>

Proview supports a significant amount of IO-systems such as Profibus, Modbus TCP, MotionControl USB IO and Q-bus.
1.3.1 Agent interface

An agent may interface with Proview by implementing a set of methods realized as C functions. The methods that an agent may implement are listed below:

- **IoAgentInit**: Should contain code to initialize the agent and bring its IO subsystem up.
- **IoAgentClose**: Responsible for cleaning up after the agent and terminating the IO subsystem.
- **IoAgentRead**: Transfers process data from the agent to Proview.
- **IoAgentWrite**: Writes process data from Proview to the agent.
- **IoAgentSwap**: Called during a soft restart.

The methods are registered with the Proview framework using the macros defined in `rt_io_methods.h`. When processed by the C preprocessor, these macros will build a vector that maps method names to function pointers. Figure 1.1 shows the method registration for the Hilscher agent. Note the absence of a swap method; this is not needed due to the Hilscher API’s design.

```c
pwr_dExport pwr_BindIoMethods(Pb_Hilscher) = {
    pwr_BindIoMethod(IoAgentInit),
    pwr_BindIoMethod(IoAgentRead),
    pwr_BindIoMethod(IoAgentWrite),
    pwr_BindIoMethod(IoAgentClose),
    pwr_NullMethod
};
```

Figure 1.1: Method registration in the Hilscher agent

1.4 About Profibus

Profibus is a bus for communication between control units, sensors and actuators within an industrial automation environment. The values are transmitted using digital signalling over a common multidrop bus instead of using separate analog signals. This is known as a fieldbus [5]. Because data from several sources may share the bus, this form of communications is more cost efficient in terms of wiring.

Several different physical layer technologies are supported by Profibus (see [5]); all hardware used for the thesis work runs on RS-485. This is a serial multidrop bus using differential signalling.

A token based protocol is used to control medium access. Only masters are allowed to initiate communication and only when they hold the token. The
token is passed between the masters in a circular fashion. If there is only one master on the bus, it will own the token permanently.

### 1.5 Profibus DP

Profibus DP (Decentralized Periphery) is a variant of Profibus featuring rapid cyclic data transfers. During a cycle, the master receives data from every slave configured by that particular master. This enables the master to acquire small data sets, such as sensor values, at a high refresh rate, typically < 10 ms/cycle according to [5].

Up to 126 Profibus DP stations can be connected to a bus. Different masters may coexist on one bus in which case medium access is handled through token passing. However, a slave may only be configured by one master. As the slaves only respond when addressed, they need not take part in the token passing. A Profibus DP message may contain up to 244 data bytes.

Profibus DP implements layer 1, 2 and 7 of the OSI model as shown in table 1.2.

<table>
<thead>
<tr>
<th>OSI Layer</th>
<th>Profibus Layer</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Application DDLM (Direct data link mapper)</td>
</tr>
<tr>
<td>6</td>
<td>Presentation –</td>
</tr>
<tr>
<td>5</td>
<td>Session –</td>
</tr>
<tr>
<td>4</td>
<td>Transport –</td>
</tr>
<tr>
<td>3</td>
<td>Network –</td>
</tr>
<tr>
<td>2</td>
<td>Data link FDL (Fieldbus data link)</td>
</tr>
<tr>
<td>1</td>
<td>Physical PHY (Physical layer)</td>
</tr>
</tbody>
</table>

### 1.6 Generic station description

Profibus DP slaves are described by GSD-files (generic station description). The content of these files are used by the bus configuration tools in order to determine what the valid configuration options for the slave are. They also contain the actual configuration data that needs to be sent to the slave in order to get it operational. Proview’s Profibus configurator includes a parser for this file format. Profibus configurator is used to configure Profibus DP slaves in Proview. Manufacturers of Profibus DP slaves typically provide GSD-files for their devices.
1.7 Diagnostics

The Profibus DP master maintains a diagnostics image of its slaves. The host may fetch diagnostics data from this buffer or from the slave directly. In this implementation the diagnostics data will be fetched from the master’s buffer.

Diagnostics consist of six bytes of standardized diagnostics data and a number of proprietary diagnostics bytes. The Hilscher CIF 50-PB supports a maximum of 100 bytes proprietary diagnostics data.

The standardized diagnostics data is the most interesting to Proview. Of these, the first two bytes contain information about the status of the slave’s bus communication. From these bytes it is possible to find out, among other things, whether the slave is addressable and properly configured.

1.8 Profibus DP in Proview

The Profibus subsystem of Proview has long supported only one host interface, the Proﬁboard from Softing. Thus, the infrastructure for handling Profibus IO in Proview was already present. The purpose of this thesis work was to add support for the CIF 50-PB by Hilscher to Proview.

1.9 Hilscher CIF 50-PB specifications

The CIF 50-PB from Hilscher has the following specifications [6]:

- Supported number of slaves: 125
- Dual port memory size: 8 KB
- Process image size: 7 KB (within the dual port memory)
- RAM size: 512 KB or 384 KB depending on revision
- Flash size: 512 KB
- Physical layer: RS485, 9.6 kBd – 12 MBd
- Watchdog for supervision of communication with host

The layout of the dual port memory of the Hilscher CIF 50-PB, as provided by [7], is shown in figure 1.2. Most of the address space is consumed by the process image: one section for the input area and another one for the output area. However, one kilobyte is reserved for other tasks. The send and receive mailboxes, used for message based communication between the
board and the host, are mapped in here. This address area is also used by
the protocol parameter, protocol status and system status fields.

Figure 1.2: Dual port memory map of Hilscher CIF 50-PB

1.10 Hilscher API

From a hardware point of view, the Profibus master board communicates
with the host computer via a dual port memory which is mapped into the
host’s address space. However, Hilscher provides a high level API to make
things easier for the software implementer. Most of this API as well as the
required kernel module and some documentation is included in the Linux
driver package which may be obtained from Hilscher. Up to four Profibus
DP master boards per system are supported by the API and Linux kernel
module. The Linux driver package is licensed under the GNU Lesser General
Public License version 2.1.

1.11 Requirements

The objective was to add support for the Hilscher CIF 50-PB Profibus mas-
ter board to the process control system Proview. This meant implementing
a module with the Proview IO interface on one end and an interface to
1.12 Motivation

There are several reasons for supporting a wide range of hardware in a control system, a few of which are listed below.

Adding support for additional hardware would make Proview more attractive, for instance by enabling entities reliant on that hardware to switch to Proview. The dependence on a single hardware manufacturer would be reduced somewhat as the Profibus interface could then be obtained from any of two independent vendors. The development process would also expose Proview’s Profibus framework to additional peer review.
Chapter 2

Problem analysis

The problem of implementing a Proview agent can be considered as three sub-problems. Getting the hardware operational, implementing Proview’s IO interface and finally adding non-essential parts of the module, for example diagnostics. This was the order in which the sub-problems were solved. Certainly it could have been done differently but this order eliminates dependencies during testing.

As a first step towards an implementation of the agent, a stand alone test program was implemented in C. At the point where the test program could initialize the board, read and write the process image and collect diagnostics from the slaves the code was moved into Proview. Because Proview already had a Profibus subsystem with support for another Profibus master, that agent was used as a starting point for the new implementation.

Some changes had to be introduced into the existing Profibus code. Most notably, the process image exchange had to be moved to the agent level; this was done in order to keep all the board specific code at the agent level.

After the Proview agent was operational, diagnostics and flashing were added. First to the test program and then migrated to the agent module.

2.1 Online configuration

The Hilscher CIF 50-PB boards are factory configured for offline configuration. When the board is configured in this way, the bus layout is fixed. To add or remove slaves a new configuration database has to be uploaded using Hilscher’s proprietary configuration tool Sycon. Boards are made online configurable by removing the database from the flash. The Hilscher agent will automatically remove this database if the macro FLASH_WRITE_ENABLE is defined.
Chapter 3

Implementation

This chapter deals with the completed Proview agent implementation. In particular we will go through all its macros, data structures and functions.

Using Proview’s build system is the preferred way of compiling and linking the agent. On the GNU/Linux platform this will invoke gcc and other parts of the GNU toolchain to compile the code.

The source files for the agent code developed for this project resides under /profibus/lib/rt/src/os_linux/ in the Proview source tree. This is also where the source code for the Hilscher agent resides, in the file rt_io_m_pb_hilscher.c.

Figure 3.1: Call graph for rt_io_m_pb_hilscher.c

The call graph in figure 3.1 shows the relationship between the functions of the agent.
Care has been taken to make the code reentrant. In particular, static and global variables have been avoided. Because of these properties, the code should support more than one board, although this has not been tested.

### 3.1 Macros

There are two macros in the agent, which may be modified to control conditional compilation of certain parts of the agent. The macros are documented below.

**SLAVE_DIAG_VERBOSE**

If the macro `SLAVE_DIAG_VERBOSE` is defined, additional diagnostics messages will be written to the Proview log by the function `dpm_print_diag`. As these messages can be issued very frequently, they make the log file hard to read when enabled, risking more important messages to go unnoticed. It is included mostly for the sake of completeness.

**FLASH_WRITE_ENABLE**

The `FLASH_WRITE_ENABLE` controls the conditional compilation which enables or disables the flashing functionality of the board. There is no good reason to disable it, since the agent will not be able to use a board with an offline configuration database. During the initialization the presence of an offline configuration database will be detected. The actual flashing routines will only be invoked if the board is found to have such a database in its flash.

### 3.2 Data structures

We will now go into some of the data structures used by the agent.

#### 3.2.1 Local structure

Each class in the IO system may use a local structure to maintain state for the current instance. Figure 3.2 shows the local structure for the Hilscher agent.
typedef struct {
    unsigned short dev_number;
    unsigned short input_size;
    unsigned short output_size;
    unsigned char cif_msgcnt;
    unsigned short watchdog;
} io_sAgentLocal;

Figure 3.2: Agent’s local structure

The list below explains the purpose of the members of the local structure.

- **dev_number** selects the board to be used by this agent instance.
- **input_size** and **output_size** hold the next offset in the dual port memory to be used by the allocator.
- **cif_msgcnt** is an (optional) message counter.
- **watchdog** is a counter used by the board’s watchdog.

The message counter is incremented for every message sent to the board (this does by definition not include the process image exchange). As the board responds to the message, this counter will be included. When the agent receives messages from the board, the message counter is used to filter out the irrelevant ones. Also during debugging the message counter may prove to be useful as it helps correlate asynchronous messages.

### 3.2.2 Known boards table

In some cases it is necessary to remove the configuration database from the board’s flash. This is performed through the message interface of the board. The start segment of the database needs to be included in the message. In the protocol interface manual [8] the start segments for the supported boards are provided.

In order for the agent’s flashing function to work, this information needs to be included in the agent’s *known_boards* array. The format of *known_boards* array is shown in table 3.1. Here DEVINFO structures (as defined in *cif_user.h* of the Hilscher API) are mapped to start segments and board names. Figure 3.3 shows what the *known_boards* structure looks like in the completed implementation.

<table>
<thead>
<tr>
<th>Data type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsigned char</td>
<td>entry</td>
<td>Used for terminating the list</td>
</tr>
<tr>
<td>DEVINFO</td>
<td>info</td>
<td>DEVINFO structure for this board</td>
</tr>
<tr>
<td>unsigned char</td>
<td>db_startsegment</td>
<td>Start segment of config db</td>
</tr>
<tr>
<td>char[100]</td>
<td>board_name</td>
<td>Name of board</td>
</tr>
</tbody>
</table>
boards known_boards[] = {
    { 1, { 8, 50, 66, { 'C', 'I', 'F' } }, 8, "Hilscher CIF 50-PB" },
    { 0 }
};

Figure 3.3: Example of known_boards entries

Adding support for additional boards

Adding support for another board is easy, if the agent has flashing support enabled: the function dpm_check_board_type will write a template line to Proview’s error log along with instructions on how to add it. The user can look up the starting address in the protocol interface manual [8]. Then insert the line into the known_boards array, with the <db start segment> and <board name> fields changed according to the data found in the protocol interface manual.

3.3 Initialization and termination functions

These functions handle the initialization of the bus. There are also a few functions for bringing the communications down when the IO system is shutting down.

Some of the initialization steps are handled by sending messages to the Profibus master. The master is supposed to answer those messages and the initialization functions will wait for the response. Although this message passing could be handled asynchronously, the current implementation uses busy-waiting instead as this simplifies the code considerably. In most cases one wants to receive the response for the previous step before starting the next one, so there would be little to gain by receiving the messages asynchronously.

IoAgentInit

Initialization starts with Proview calling IoAgentInit. IoAgentInit first allocates heap memory for the local data structure that accompanies every instance of the different levels of the IO system. The local data structure is also initialized here. Next the function calls dpm_init to initialize the Hilscher API. Then dpm_init_master_check_sycon_db is called to set some basic master parameters and check that the board has not already been configured by Hilscher’s configuration tool Sycon. After that, the function iterates over every slave configured by Proview and downloads parameterization data to them by dpm_download_slave_prm. Finally, dpm_download_master_prm is called to parameterize the master and start the cyclic data exchange.
dpm_init

The function `dpm_init` links the agent with the kernel driver and the desired Profibus master board.

dpm_init_master_check_sycon_db

To check that the board is not set for offline configuration by Hilscher’s configuration tool Sycon, the function `dpm_init_master_check_sycon_db` should be used as a wrapper for `dpm_init_master`. A board can only be configured once after a reset and the Sycon configuration will take precedence over any provided online configuration parameters.

After the master has been configured by `dpm_init_master`, the code checks if the board is already in its running state. This process is explained in [9]. If that is the case, the board must have an offline configuration database in its flash. Now, if the macro `FLASH_WRITE_ENABLE` is defined (which is true by default) the flashing routine `dpm_delete_flash_prmdb` will be called to remove the offline configuration database.

dpm_init_master

The function `dpm_init_master` is used to configure certain supervisory settings of the board. This will set up certain low level Profibus master features, for example the watchdog, process image handshake mode and byte order.

dpm_delete_flash_prmdb

If an offline configuration database is found on the board, it may be deleted by a call to `dpm_delete_flash_prmdb`, provided that the board is known by the agent. The reason for this prerequisite is that the flashing routine needs to know the start segment of the configuration database in the flash. This varies between different boards.

When called, the function uses `dpm_check_board_type` in order to try to find the startsegment of the board. If successful, a delete database message will be sent to the board. Then the function will wait until a response is received, and return `DRV_NO_ERROR` if successful.

dpm_check_board_type

The function `dpm_check_board_type` will check whether the current board’s configuration database start segment is known by the agent. This is done by getting the type, model and id of the board and comparing them to
the entries in the structure known_boards. If there is a matching entry, the entry’s db_startsegment field will be written to a pointer provided by the caller of dpm_check_board_type. The function returns DRV_NO_ERROR if successful.

**dpm_download_slave_prm**

Each slave that is to be logically connected to this master needs to be parameterized. This is done by a call to dpm_download_slave_prm for each slave.

The function builds a parameterization message that is sent to the slave using the function dpm_ddlm_download. Most of the parameters are hardcoded but some of them can be changed from Proview. The watchdog is one example of a changeable parameter. The size of the parameterization message depends on the exact settings of the relevant slave. Therefore, the offsets of the various data fields are calculated on the fly. Every slave attached to the Profibus master must be configured by Proview prior to the initialization of the IO framework. Thus, the data fields needed to parameterize the slaves are obtained from Proview. The function dpm_set_add_tab is used to map the IO area of the current slave to an area in the dual port memory.

**dpm_set_add_tab**

The address table is part of the parameterization message sent to every slave. It is generated by the function dpm_set_add_tab. Two counters in the local structure of the agent keep track of the used dual port memory area. One counts the number of inputs and the other one counts the number of outputs. It is important to understand that these address offsets do not directly relate to the dual port memory as a whole. Rather, the input offsets relate to the process image input area while the output offsets relate to the process image output area (see figure 1.2). Therefore, keeping separate counters for inputs and outputs respectively is necessary as the offsets are independent.

**module_cnt_inputs and module_cnt_outputs**

Proview knows the memory area needed to represent the state of the IO area for the slaves. That fact is used by the functions module_cnt_inputs and module_cnt_outputs in order to calculate the IO size for each slave. This is accomplished by looping over every channel and adding the size required to represent the channel to an internal counter which is returned at the end of the respective function.
3.4. ERROR LOGGING FUNCTIONS

**dpm_ddlm_download**

The master and its slaves are parameterized similarly. That is, by sending a DDLM.DOWNLOAD.REQUEST message to the correct address. The purpose of *dpm_ddlm_download* is to provide a higher level interface to send these messages.

**IoAgentClose**

When Proview terminates the IO system, it will invoke *IoAgentClose*. This trivial function calls *dpm_exit* to shut down the Hilscher API and returns the memory used for the local structure.

**dpm_exit**

The function *dpm_exit* will unlink the Hilscher API from the Profibus master board and the kernel driver.

### 3.4 Error logging functions

The purpose of this set of functions is to write error and diagnostics messages in human readable form to the Proview log.

**dpm_ddlm_answer_msg_print_error**

When sending a DDLM download or diagnostics request, the Profibus master board may respond with an error code. Descriptions for these error codes are provided in [8]. If an error message is received, the function *dpm_ddlm_answer_msg_print_error* will write the relevant error description to the Proview log.

**flashing_disabled_warning**

If the Profibus master board is set for offline configuration and the Profibus agent failed to remove the configuration database from the board, then this function may get called. Its only purpose is to write instructions for removing the configuration database to the Proview error log.
dpm_print_diag

The diagnostic error messages will be written to the Proview log in human readable form by this function. These messages can be found in the dpm_user.h header file. Some of the messages will only be written if the macro SLAVE_DIAG_VERBOSE is defined (it is undefined by default). This is to prevent the agent from writing too much to the log.

3.5 IO exchange and diagnostics functions

These functions give Proview access to the process image. Both of the functions will also service the Profibus master’s watchdog. As a consequence of this, the master will be reset if any of these functions are not called within the StallTime period, as configured in Proview. On top of that, the IoAgentRead function is also in charge of acquiring diagnostics from the slaves.

IoAgentRead

First of all, the function services the watchdog. Subsequently it will read the input area of the process image for every configured slave and transfer the data to Proview.

Next, the protocol state will be fetched from the Profibus master. The function dpm_print_diag is called to write the generic errors to the Proview log.

The protocol state also contains a bitmap, showing which slaves have unread diagnostics data. For those slaves, the function dpm_req_slave_diag is called to request that the slave transfers this diagnostics data to the master. The master will then make this data available in the form of a message, that can be fetched asynchronously.

Finally, dpm_update_slave_diag is invoked to fetch any pending diagnostics messages.

dpm_req_slave_diag

When a slave has unread diagnostics data, the function dpm_req_slave_diag is used to send a request for the slave to transfer the diagnostics data to the master. Only the request is sent, the response must be fetched by another function. This behaviour is necessary, in order to maintain constant cycle times.
**3.6. PROVIEW CLASS**

**dpm_update_slave_diag**

The function `dpm_update_slave_diag` will process every message in the receive mailbox. Every message in the mailbox that is not of type `DDLM_Slave_Diag` will be discarded in order to make room for new diagnostics messages. Diagnostics messages are processed and the relevant diagnostics data is transferred to Proview’s data structures.

**IoAgentWrite**

This function services the watchdog and writes new data to the process image output area for every configured slave.

**3.6 Proview class**

For the agent to be configurable from Proview, it must have an associated class. Classes are defined in ASCII text files with extension `.wb_load`. Proview includes an editor that greatly simplifies the editing of class files. The class editor can be invoked from the shell by issuing:

```
wblstart.sh <filename.wb_load>
```

For this to work, the build environment must have been initialized prior to the invocation.

**3.7 Proview interface**

The PLC program that is built by Proview’s development environment uses the IO system through function calls. There are five standard functions that one may implement in an agent: `IoAgentInit`, `IoAgentClose`, `IoAgentRead`, `IoAgentWrite` and `IoAgentSwap`.

To make Proview aware of these functions, they must be exported using the `pwr_BindIoMethods` macro as explained in [4]. The macro itself is defined in `/src/lib/rt/src/rt_io_methods.h` in the Proview source tree.

**3.8 Using the board**

When the board is set to use online configuration it needs to be configured by the application. After the board has been configured correctly the cyclic data transfer will commence. The initialization order provided below was derived from [8] along with a certain degree of experimentation.
1. Call `DevOpenDriver` to link the API to the kernel module
2. Call `DevInitBoard` to initialize the desired board
3. Configure the master by sending a `DPM_PLC_PARAMETER` message using `DevPutTaskParameter`
4. Soft reset the card using `DevReset` with the `WARMSTART` argument
5. Parameterize the slaves by sending them parameter data encapsulated in `DDLM_DOWNLOAD_REQUEST` messages using `DevPutMessage`.
6. Parameterize the master by sending it a `DPM_BUS_DP` message encapsulated in a `DDLM_DOWNLOAD_REQUEST` message to Area-Code 127 using `DevPutMessage`.

After completing these steps the process image may be exchanged using the `DevExchangeIO` function.

### 3.9 Dual port memory allocation

As mentioned earlier, the process image exchange takes place in a dual port memory on the Profibus master board. It is important that the host application and the Profibus master agree on the address layout (i.e. which addresses are used by which slaves) and that there are no addresses with the same data direction that overlap. Signals of different data direction may overlap however, that is, an input is allowed to share address with an output. This is a result of the dual port memory having separate areas for the input process image and output process image respectively, which are addressed independently.

The offsets used for inputs and outputs respectively are specified as part of the slave parameterization process. These offsets have to be calculated manually. The function `dpm_set_add_tab` handles the offset calculation. Figure 3.4 illustrates the allocation process.
3.9. DUAL PORT MEMORY ALLOCATION

The agent’s local structure contains two counters, `input_size` and `output_size`. Both of these are initially set to zero (this is done in the `IoAgentInit` function).

When `dpm_set_add_tab` processes the modules of a slave (each module with an input or output area gets its own entry in the `add_tab`), it calculates the size of the input and output areas in two different steps. The `add_tab` structure as defined in [8] requires that input address entries precede the output address entries. As a result of this the inputs are processed before the outputs. The `output_size` counter is used as address offset, then the size of the output area for the current module is added to the counter so that the next module’s area will follow right after the current one. The inputs are handled in the same way, except that the `input_size` counter is used.
Chapter 4

Testing

A number of test cases were designed to test the agent code. The tests emphasized on parts of the implementation that were considered to be particularly error prone or that would be hard to test during normal operation.

Two Siemens Simatic ET 200M (IM 153-1) were used as slaves in all the tests. These are modular slaves, that support several types of IO modules. Up to eight IO modules may be attached to each slave and different types of modules are allowed to be mixed.

4.1 Basic functional test

Testing of the boards basic functions was done in two steps. At first, the code was implemented as a stand alone test program. This version configured the bus, read from an input module and wrote to an output module. The existing Proview agent for the Softing Profiboard was used to generate the slave parameterization data for the test program.

After the hardware and test program was working as intended, the code was reimplemented as a Proview IO agent. Using this agent a similar set of tests were performed. In this case the input and output channels were read and written using Proview’s runtime environment. When Proview’s view of the IO channels was consistent with their actual states, the test was considered to be passed.

4.2 Interoperability with Profiboard agent

To ensure interoperability with the existing Softing Profiboard agent, the boards were tested in parallel. The hardware setup is shown in figure 4.1.
After the hardware had been setup, Proview was used to read and write data from/to the modules of the boards. After verifying that the two buses were indeed working independently, the test was considered passed.

![Test setup for interoperability testing](image)

**Figure 4.1: Test setup for interoperability testing**

### 4.3 Dual port memory allocation code

The code for locating the modules’ data in the dual port memory is somewhat complicated. As the dual port memory of the Hilscher CIF 50-PB is a scarce resource, the allocation code should pack the IO areas to make use of the available memory in an efficient way. Since Proview already knows the memory size needed to represent the configured channels, the exact size for each Profibus module can be calculated.

In order to verify the memory allocation code, the Profibus equipment was connected as shown in figure 4.2. Then, debug output statements were inserted into the code in order to show which memory areas were claimed by which modules. Finally, Proview’s Runtime navigator was used to verify the operation of the slaves’ channels. The test passed.
4.4 Watchdog

When the agent controls potentially dangerous machinery in the plant, it is important that the outputs are only active as long as the PLC program is in control. If the PLC program terminates, the IO should be cleared. This should also hold if the PLC program would crash.

To assist in solving the above issue the Hilscher CIF 50-PB is equipped with a watchdog. After the watchdog has been initialized it will cause a reset of the Profibus master if its watchdog is not serviced within a specified interval. That will in turn cause the watchdogs in the slaves to expire if setup properly. A slave may perform some action when its watchdog expires. For the slaves used in the testing phase this caused the outputs to be cleared.

Testing consisted of terminating the runtime environment. All output channels were cleared after the watchdog of the master and the slaves expired. Therefore the test passed.

4.5 Diagnostics

Getting the diagnostics to work right was a major challenge. The testing primarily consisted of disconnecting or shutting down slaves and verifying that Proview’s diagnostics output reflected the changes. Removal of slaves, as well as re-adding them later does update the diagnostics data shown in Proview.

Testing was carried out in several rounds, as the initial version had troubles recognizing re-adding of slaves in some circumstances. The problem occurred when the slaves were disconnected from the bus and then re-connected before their watchdog had expired. In order to cover this case, a fix was added to the diagnostics collecting code in the function IoAgentRead. This iterates through all slaves that the system deems to be non-existent.
For every such slave the diagnostics data is checked to see if the slave has become available again. Based on the tests performed, this fix resolved the issue and ensured that the status of the slaves would get updated as they are re-added.

Diagnostics data collected using the Hilscher agent is not completely consistent with that collected by the Softing agent. These differences are not very significant however.

4.6 Flashing

The flashing code was tested by uploading an offline configuration database to the board using the tool Sycon, included on the system software disc that came with the board. In order to verify that the card was no longer accepting online configuration, the agent was first recompiled with the flashing code disabled (FLASH_WRITE_ENABLE undefined). Then the agent was started, as expected, the agent could no longer configure the slaves and failed with an error message. Next, the agent was recompiled with the flashing code enabled. After the agent was restarted, it removed the board’s offline configuration data as it was supposed to. Thus the agent passed the test.

4.7 Multiple calls to DevOpenDriver

If more than one Hilscher Profibus master board is used by the same PLC program, the API function DevOpenDriver will be called during the initialization of each board. To verify that this would not cause problems when multiple boards were used, the code for the Hilscher API was first inspected. A quick look at the code for the function implied that calling it multiple times should cause no problems. As a final check, the test program was modified to do two consecutive calls to the function during initialization. The program still worked and thus the test passed.
Chapter 5

Conclusion and future work

While the current implementation passes the tests listed in chapter 4, there are some additional issues that would be interesting to look at.

The agent was never tested with more than one Hilscher CIF 50-PB board. The reason being that only one such board was available during development and testing. Care was taken while writing the agent to make the functions reentrant in order to allow multiple instances to be run in parallel. In particular, static data members and global variables were avoided in the agent.

Diagnostics data is not entirely consistent with that of the Softing Profi-board agent. The author speculates that this may be caused by different interpretations of the relevant standards document.

Consistency is relevant when a slave has channels that require more than one byte to represent. If a channel is represented by several bytes it is, of course, critical that these bytes came from the same sample. According to [7], the process image is guaranteed to be consistent when using the mode \textit{buffered data transfer, host controlled} to communicate with the process image. This is the mode used by the agent. Verifying that this consistency is actually present would be extremely difficult.

The current slave parameterization code uses a single message to upload the configuration data. This way, up to 240 bytes of slave configuration data can be sent. Note that some of this space is consumed by overhead imposed by various length fields [8]. To get around this, the data set may be uploaded by sending several parameterization messages in a sequence. When sequenced parameterization download is used, up to 1000 bytes of configuration data may be sent. Sequenced downloading was not implemented in the agent. None of the slaves available during development and testing required more
than 240 bytes of configuration data, therefore this could not have been tested appropriately. Adding support for sequenced downloads to the agent should be a straightforward task.

As is the case with the agent supporting Softing’s Profiboard, the address of the Profibus master is hardcoded to address 0. If more than one master is to coexist on the same bus, it would be useful to be able to change this setting from Proview’s Profibus configurator. That would mean adding a master address setting to the agent’s Proview class and changing the assignment of `prm.bFDL_Add` in `dpm_download_master_prm` in the agent’s code (`rt_io_m_pb_hilscher.c`).

Performing a `DevReset` from `IoAgentClose` could be used to make the actual slaves shut down a bit faster when Proview’s IO system is brought down (this would bypass the master’s watchdog).
Bibliography


Appendix A

Terms and abbreviations

This section explains some terms and abbreviations used throughout the document.

Agent

The module operating at the highest level in a distributed Proview IO system.

API

Application programming interface: a set of higher level functions that a programmer may use to conveniently use a library, a piece of hardware or similar.

DDLM

Direct data link mapper: operates at the application layer in the OSI reference model. DDLM maps application layer requests to the data link layer.

DP

Decentralized peripherals: a flavour of Profibus emphasizing on fast cyclic data exchange. That is, quickly transferring small data sets over a digital communications link.

Dual port memory

A random access memory that may be accessed simultaneously from two independent ports.
GSD

Generic station description: an ASCII-file containing configuration data for a slave.

PLC

Programmable logic controller: a computer dedicated to controlling an IO system, typically realized as an embedded system.

PLC program

The PLC program is built from within Proview and provides the functionality of a programmable logic controller using generic computer hardware.

Process image

The process image is a memory area within the dual port memory that reflects the state of every configured channel.

Proview runtime environment

The set of processes making up the runtime platform for Proview. Most notably the PLC program.

Q-bus

Originally the system bus used in certain versions of DEC’s PDP-11. In a native Proview IO system, this bus is used for communications with the various IO cards.

Watchdog

A watchdog is a hardware timer controlling the reset line of a processor, microcontroller or similar device. The application program running on the processor is responsible for servicing the watchdog within certain intervals. If the application program fails to service the watchdog within this time, due to a bug causing the program to crash for example, the watchdog will reset the processor.
Proview is an open source system for process control, developed at SSAB Oxelösund. A wide range of IO systems are already supported by Proview. This thesis documents the implementation of support for the Profibus DP master board Hilscher CIF 50-PB.

Most of the features of the board are supported by the agent. The agent is intended to support up to four boards per system. Adding support for additional types of Profibus DP master boards from Hilscher to the agent should be straightforward.

This thesis covers some technical aspects of the Profibus DP technology and also provides some background of Proview’s IO system. The design decisions behind the Hilscher agent implementation will be discussed. All functions and data structures of the implementation are documented. Testing of the implementation is also included in the thesis.

The thesis may also be used as technical documentation for the agent implementation.

Finally, we look at the strengths and shortcomings of the completed agent implementation.
Upphovsrätt

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under 25 år från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämligen som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

Copyright

The publishers will keep this document online on the Internet – or its possible replacement – for a period of 25 years starting from the date of publication barring exceptional circumstances.

The online availability of the document implies permanent permission for anyone to read, to download, or to print out single copies for his/hers own use and to use it unchanged for non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional upon the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page: http://www.ep.liu.se/.

© Ferdinand Hauck