OFESKÉ VYSOKÉ UC TECHNICAL ENGINEERING
FACULTY IN PRAGUE

DEPARTMENT OF CONTROL TECHNIQUES

THESIS

Profibus DP Master on PC

2004

Pavel Trnka

Annotation

This work presents a solution to the problem of connecting the Profibus fieldbus to a regular PC without the use of special hardware. The work implements a bus control unit (master) for Profibus DP (Distributed Peripherals), which is the most widespread variant of this standard intended for communication between control units and remote peripherals (sensors, actuators).

Commonly used solutions are expensive because they are built on special expansion cards using their own processors, customer circuits and other circuits with sufficient power to meet the high requirements of Profibus.

This work, on the other hand, allows you to connect Profibus using a standard RS-232 serial port or using simple plug-in PCI cards with UART circuits. Profibus DP Master is created by software and special features of hardware solutions are replaced by the maximum use of commonly available resources in the PC, which was mainly made possible by writing a program as a driver for Windows NT / 2000 / XP operating systems. In addition, the application interface is compatible with Siemens solutions, allowing for easy interchange.

Annotation

Solution for connecting fieldbus Profibus to common PC without use of special hardware will be proposed in this work. The work implements bus control unit (master) for Profibus DP (Distributed Peripherals), which is most widely used variation of this standard. It's designed for communication between control units and distributed peripherals (sensors, actuators).

Commonly used solutions are expensive because they are built on special expansion cards, which are using their own processors, customer's circuits or another circuits with enough power to meet high requirements of Profibus.

On the other side, this work allows for connecting Profibus to standard serial port RS – 232 or to simple PCI expansion card based on UART circuit. Profibus DP Master is created as software implementation and special features of hardware implementations are substituted by maximal use of common means in PC. This was mainly possible by creating Profibus DP Master as system driver for Windows NT / 2000 / XP. Moreover application interface is compatible with solutions from Siemens company, which allows for simple replacement.

AND

Declaration

Thanks

First of all, I would like to thank my thesis supervisor Ing. Petr Smolík for his helpfulness, willingness to consult and the time he devoted to me. Furthermore, Ing. Pavel Píš for his many inspiring advice from his rich practical experience and Ing. Pavel Burget for his help in creating this thesis.

Thanks also go to my family for their constant support and home background throughout the study. Last but not least, many friends whose company has always given me energy for further work.

Content

Chapter 1

Introduction

The Profibus industrial bus [5] is one of the most widely used buses in the industry. Its most commonly used variant is Profibus DP (Distributed Peripherals). It is designed for communication between control units (PLC, industrial computers) and decentralized peripherals, which are remote sensors and actuators. Instead of a multi-wire connection that would connect each peripheral separately with one line, one Profibus connection is used with a linear bus topology that connects everything.

Deployment of the Profibus guarantees fast and reliable data exchange and, in addition, it is possible to share one bus with several control units. On the other hand, if we want to use a regular PC as an active master control station and connect Profibus DP to it, we will encounter the high price of plug-in cards that the Profibus master implements using customer circuits and other hardware solutions.

The motivation for this work was therefore to design the cheapest and easiest to use active station implementation Profibus DP Master for PC, which would be from the application point of view compatible with some practical solution and thus allow its replacement.

The hardware layer of the Profibus is based on the RS-485 standard, so a standard RS-232 serial port with a simple RS 232/485 converter, which does not need an external power supply, is sufficient for the Profibus connection implemented in this work. This is a solution applicable for low bit rates. If we need to achieve high transmission speeds (currently the maximum is 12Mbps), then this work also allows the use of PCI cards with UART (Universal Asynchronous Receiver / Transmitter) circuits, where the maximum speed is determined only by processor power and UART circuit options.

However, the simplicity of the connecting hardware has shifted all the work of the special circuits commonly used for this purpose to software. It must therefore replace the hardware solutions with maximum use of all available resources of a regular PC.

Progressive software work has shown that the only viable implementation path for modern operating systems is to create Profibus DP Master as a driver. And so it was created driver for Windows NT 4.0 and adding Plug and Play support also for Windows 2000 and Windows XP operating systems. The created driver implementsProfibus DP Master for PC up to FDL layer (line layer).

An undervalued part of this work was also the choice of the name of the project by which the word was chosen ProfiM for a certain similarity with the name of the problem and mainly due to the small number of relevant links found by Internet search engines.

1.1 Why a driver?

There were several reasons to create a DP master as a driver. Above all, it was necessary in Windows operating systems to time with periods in the order of tens of microseconds, which is not achievable for common applications. By timing is meant the invocation of service routines after very short time intervals. The only way to achieve this was to take advantage of the interrupt capabilities and hardware features of a standard serial port or UART circuitry on PCI expansion cards. However, this would not be possible from a normal application, as they are not allowed direct access to the operating system by the operating system.

Another reason was the need to control the RS 232/485 converter, which is theoretically simple, but in the software implementation under Windows it was a challenging problem, which was also solved only by creating a driver. Finally, the driver implementation integrated the Profibus interface into the operating system, which brings benefits from the application layer for further use.

1.2 The structure of this work

To read and have a good understanding of this work, it is advisable to have at least partial knowledge of Profibus issues - this is roughly described in Chapter 2, but in some chapters there may be principles and concepts that are not fully explained.

Chapter 2 roughly describes the principles of Profibus, especially its line layer, is the main part of this work. A more detailed description can be found, for example, in [1] or directly in the standard [5].

Chapter 3 shows how to connect Profibus to a PC in terms of physical layer, or how to connect bus using the RS-485 industry standard.

Chapter 4 describes how drivers are created in Windows operating systems. Describing their basic structure, what tools are needed for creation and what are the differences from writing common programs. This chapter is useful in that the information about writing drivers for physical devices summarized in this way is little available. We will meet either with a superficial description or with detailed documentation.

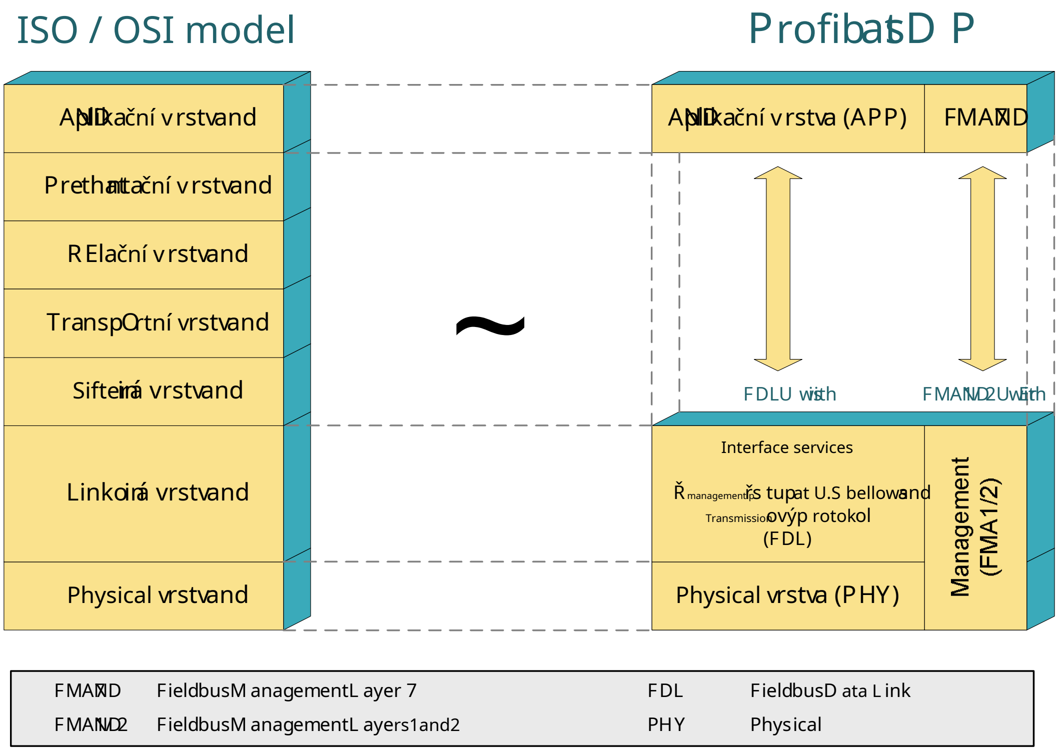
Chapter 5 shows how the Profibus DP Master line layer is created - how it went its development and what problems needed to be solved.

Chapter 6 describes the interface that the application (higher layer DP master or directly FDL application) used to communicate with the FDL layer (driver).

Chapter 2

Profibus DP

Profibus DP (Distributed Peripherals) is the most used variant of the Profibus industrial bus, which is designed mainly for communication between control units and decentralized peripherals. Usually, one communication channel replaces the multi-wire connection of the control unit with sensors and actuators, where the use of Profibus also ensures reliable and fast data exchange.



FMA7ND	Fieldbus Management Layer 7	FDL	Fieldbus Data Link
FMA1/2	Fieldbus Management Layers 1 and 2	PHY	Physical

Figure 2.1: ISO / OSI model and Profibus DP

Profibus DP has a layered architecture built according to the ISO / OSI model (Figure 2.1). It uses physical, line and application from individual layers. The unused four layers are partially included in the others. This work implements the physical and line layer and offers an interface for the application layer.

2.1 Physical layer

The physical layer defines the requirements for the properties of the transmission channel. The Profibus standard [5] offers an extended RS-485 standard as one of the options for the implementation of the physical layer of use in industry. This determines the physical properties of the channel and the data encoding on the bus.

An important property, which is defined by the standard for the physical layer, is the speed of communication. This is specified for Profibus DP in the range of 9.6 kbps to 12 MBps (Table 3.1). The standard defines a maximum permissible deviation of 0.3%, which in practice may make it impossible to achieve certain communication speeds. For example, if the UART has an inappropriate base clock frequency and only rough options for dividing that frequency. However, experience shows that the acceptable speed deviation is up to 1%.

2.2 Line layer

The link layer has the task of controlling medium access control, compiling transmitted frames, decoding incoming frames and providing data exchange and link layer control services to the higher layer.

The link layer consists of the FDL (Fieldbus Data Link) part, which provides its main functions, and the FMA (Fieldbus Management) part, which provides its control. Although not entirely accurate, the line layer is often referred to as the FDL layer.

2.2.1 Bus operation

Profibus allows you to connect up to 127 stations on one bus. The communication channel formed by the bus thus becomes a shared physical resource on which only one station can transmit at a time. The standard therefore specifies precise rules for the efficient and deterministic use of the bus.

Stations are divided into two types: master and slave. Master stations are designed to control bus traffic and initiate communication. In contrast, a slave station can only start transmitting if it has been directly requested to do so by the master station.

In order for it to work on the communication bus, it must have at least one master station on it. However, there can be several master control stations. In order for them to be on the same bus at the same time, they need to take turns in bus control. This is ensured by the fact that the master stations pass on the authorization to send the so-called token. The transmission takes place in order growing addresses, and the master with the highest address passes the token back to the master with the lowest address. By this gradual transfer, the master stations form a logical circle structure.

The structure of the logical circle is determined by the List of Active Stations (LAS), which is created and maintained by each master on the bus. It is a list of all addresses, in which it is stated for each station whether it is an active master station. Each station creates this list by monitoring bus traffic. From it, it also finds out which next station to pass the token (NS - Next Station) and from which station the token will come back (PS - Previous Station). When the master receives the token, it obtains the credential to transmit on the bus. It can send requests to slave stations or communicate with another master.

The token holding time is limited to ensure an upper time limit after which the transmission credentials are returned to each master. It is limited by time T_{TR} (Time To Reach), which is defined in each Profibus configuration and specifies the required maximum token circulation period between all master stations.

If he holds a master token, he is obliged to find out before each transmission whether he still has time to hold it. For this purpose, it measures the time interval from the last time the token was passed to the next master station in the logical token ring (NS). This interval compares with the value of the parameter T_{TR}. If the time interval exceeds this parameter, the token passes to the next master station. This is a special case when there is no time left to accept the token. In this case, the master can process one high priority request.

An essential part of the standard is a description of Profibus's behavior in so-called transient states, which are temporary events on the bus suspending its normal operation, such as: first start of bus communication, connection or disconnection of the master station, token failure, bad frame, etc. will be described in section 2.2.4).

If the bus is not in a transient state, then cyclic data exchange usually takes place between the master stations and its respective slave stations (distributed peripherals). During this data exchange, the master cyclically sends output data to each of its slave stations and at the same time reads the input data. It most often sets and reads digital or analog outputs and inputs. In addition, the use of a special bus command ensures the synchronization of input / output values on all slave stations, ie the simultaneous setting of all outputs and the simultaneous reading of all inputs at one point in time.

The whole data exchange thus resembles a part of the scan cycle in the PLC, not by chance, because PLCs are the most common master stations.

In order to add or remove stations on the bus while running, each master maintains a list of occupancy of individual addresses in its address space (GAP), which is the range of addresses from its address (TS - This Station) to the address of the next master (NS). This list is called a "GAP list" and its addresses are cyclically tested by the master at certain time intervals to determine if there is a master station, a slave station or an unoccupied address.

2.2.2 Frame formats

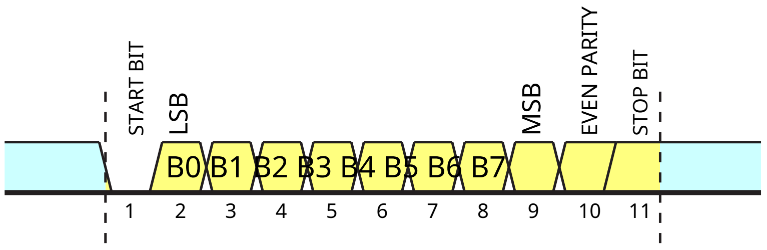


Figure 2.2: Profibus character format

The basic data unit on the Profibus bus is the eleven bit character (Figure 2.2), which consists of one start bit, followed by eight data bits, an even parity bit and one stop bit. Frames (packets) are compiled from these characters, for which there is an important requirement on Profibus that there are no time gaps between the following frame characters. This requirement may seem trivial, but it can also be a problem if the serial channel transmitter used does not have FIFO memory and we do not have the ability to handle incoming interrupts fast enough each time.
A total of four types of frames are used on all buses on Profibus:

1. Framework without data (Figure 2.4)
2. Fixed length data frame (Figure 2.5)
3. Variable-length data frame (Figure 2.6)
4. Transmission Credential Framework - Token (Figure 2.3)

For the meanings of the abbreviations of the names of individual flats in the boxes, see Annex no. 1. Each frame type has a specific request frame structure and a response frame structure that should come to the request. A special answer is the Short Acknowledge framework (Figure 2.4), which can come to any request and usually confirms the execution of the request.

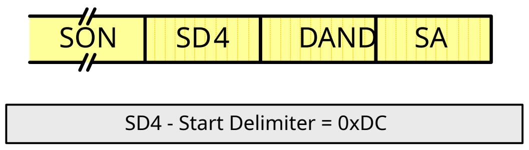


Figure 2.3: Token Frame format

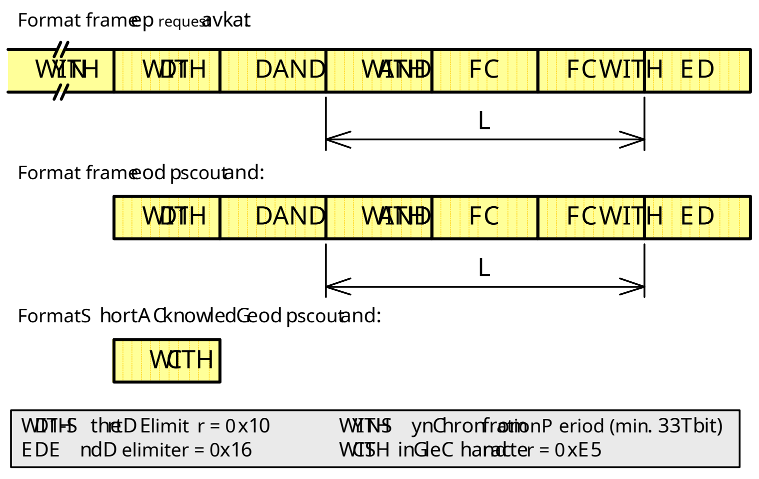

Format framep requesavkat
DAND
FC
FCWITH ED
L
Format frameod pscoutand:
FormatS hortA Cknowled Geod pscoutand:
WDTH S thetD Elimit r = 0x10
EDE ndD elimiter = 0x16
WITHS ynChronfrationP eriod (min. 33Tbit)
WCTSH inGleC haracter = 0xE5

Figure 2.4: Frame format without data

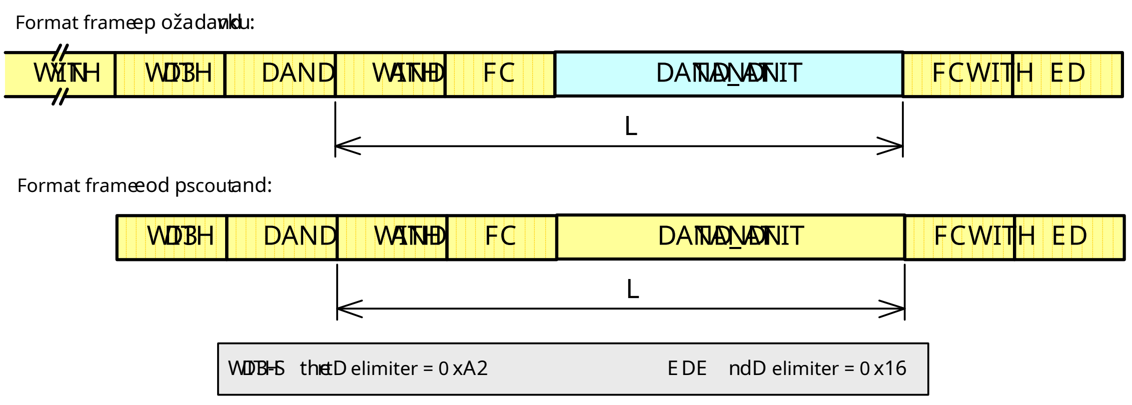


Figure 2.5: Format of a frame with fixed length data

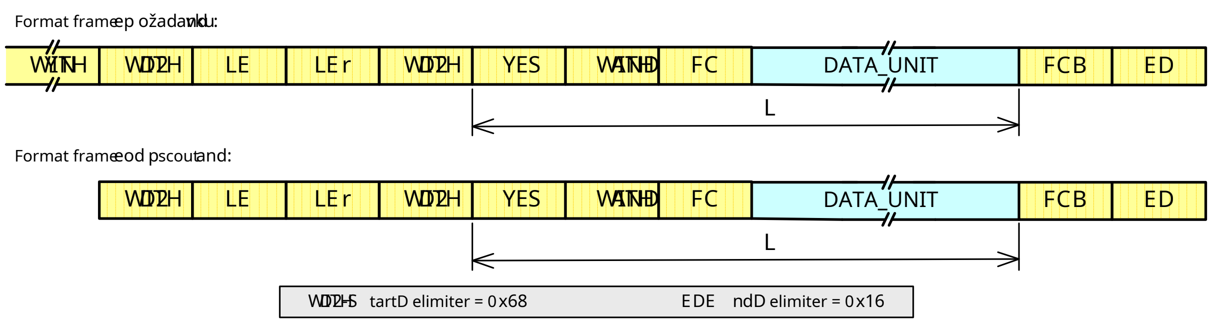


Figure 2.6: Format of a frame with variable length data

2.2.3 Profibus state machine for FDL layer

The FDL function, or line layer, can be well described and also implemented as a state machine with ten states and transitions between them (Figure 2.7). After switching on the power supply or reset, the machine starts in the default state "Offline". In response to external events, lower and higher layer requests, the automaton switches between states. The transitions that the machine goes through during the operation of the station in the multi-master system without exceptional situations are marked in red. Transitions marked in black correspond to events related to bus collisions, token loss, bus failures, and logic circuit changes.

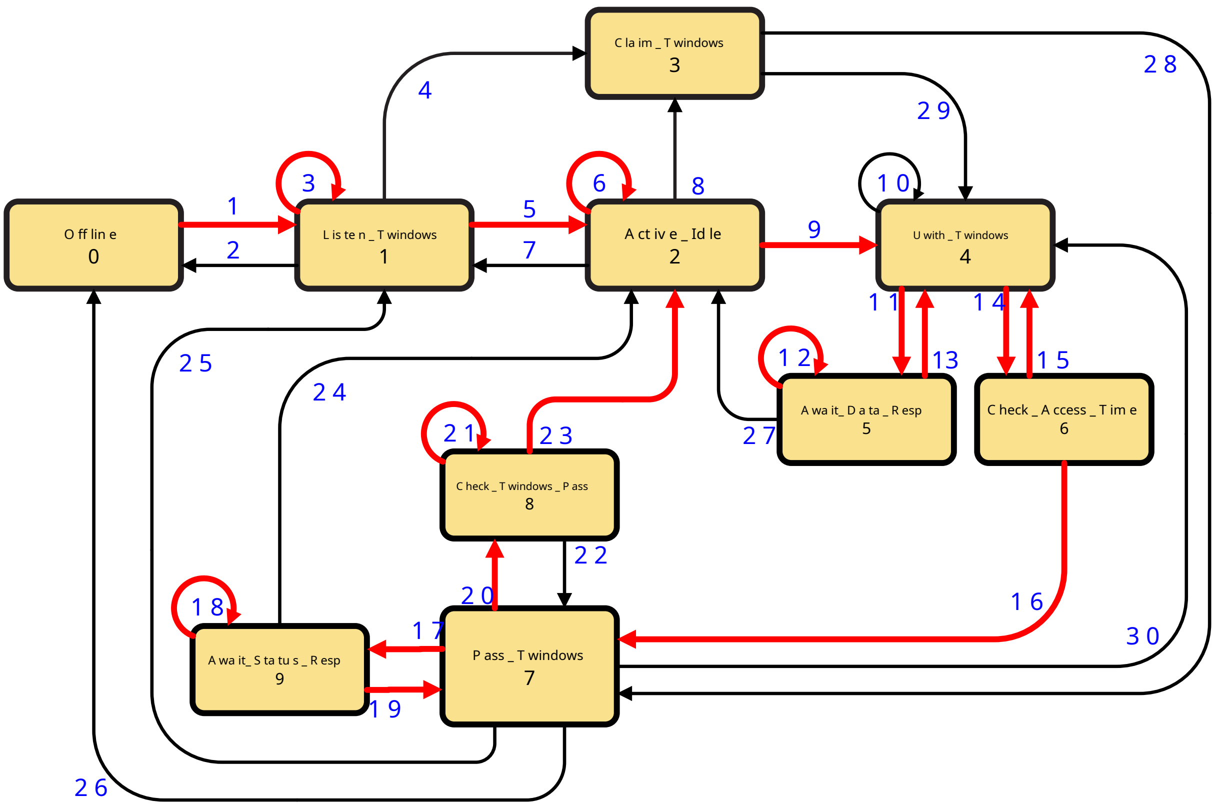


Figure 2.7: FDL layer state machine

The following sections describe the operation of the state machine. The numbers in parentheses correspond to the numbered arrows in Figure 2.7, which represent the transitions between states.

0 Offline

The master enters the "Offline" state as soon as the power is switched on. In this state, the operating parameters are initialized and a self-test is performed. When initializing the parameters, the configuration is read and the FDL layer data structures are initialized. Performing a self-test depends on the specific implementation of the master and usually involves testing the transmitter and receiver circuits. In the bus disconnection state, the internal loop-back is closed and the test data transfer is tested. After successful initialization and self-test, (1) the master enters the "Listen Token" state.

Another reason to go to the "Offline" state can occur if a station with the same address (2) is detected on the bus or if a hardware fault in the bus access (26) is detected - a fault in the transmitting or receiving circuits.

1 Listen Token

When the master is ready for communication, it enters the "Listen Token" state. In this state, it is still passive and only monitors (3) bus traffic to find out which stations are active. It detects this by receiving and analyzing all "Token Frame" frames (Figure 2.3), which are used to pass credentials between active stations in the logical circuit. By retrieving addresses from the Token Frames, the station creates a list of active stations - LAS (List of Active Stations).

If the master manages to eavesdrop on an identical token cycle between active stations when creating a LAS list, it sets its status to "Ready to enter logical token ring" and continues to monitor bus traffic and update the LAS list. From the created list, the LAS master determines its predecessor (PS) and successor (NS) in the logical circuit. It remains in the "Listen Token" state until the "Request FDL Status" that is addressed to it arrives. In response, the master sends its "Ready to enter logical token ring" status and, with the immediate arrival of the credentials, switches to the "Active Idle" status (5). This is how the process is described in the standard, but after the arrival of the credential framework, it is more of a simultaneous transition (5) and (9) to the "Use Token" state.

When entering this state, the FDL starts a time-out timer with a period T_{TT}. If no traffic is detected on the bus until this timer expires, then the FDL assumes that a logic circuit needs to be initialized or reset. For this purpose, the master enters the "Claim Token" state (4).

When eavesdropping on credential frames, a frame whose sender address (SA) is the same as our address (TS) may arrive. If two such frames are received, then probably the station with our address on the bus already exists and is included in the logical circle. The master goes (2) to the "Offline" state and notifies the FMA1 / 2 layer of this error.

2 Active Idle

"Active Idle" is a state in which the master is fully included in the logical circle, but does not currently have a token. It monitors bus traffic and responds to requests addressed to it, or only receives data from the bus that does not require a response.

When the credential frame addressed to us (DA = TS) arrives, (9) the master enters the "Use Token" state. If the master detects that it was skipped when passing credentials in the logical circle (eg the PS station passed the token to the NS station), it goes (7) to the "Listen Token" state. It also enters this state (7) if it detects that a station with a duplicate address is transmitting on the bus (the frame received from outside has SA = TS).

When entering this state, the FDL starts a time-out timer with a period T_{IT}. If no traffic is detected on the bus until this timer expires, then the FDL assumes that the logical ring has broken and needs to be restored. For this purpose, the master enters the "Claim Token" state (8).

3 Claim Token

The master enters this state due to the expiration of the timer for monitoring activity on the bus, either from the "Listen Token" or "Active Idle" state. This means that the credential transfer logic has broken or our station is the only master on the bus. If the logical circle breaks down, we already have the GAPL and LAS lists and it is not necessary to create them again, so we immediately go (29) to the "Use Token" state. In order to avoid concurrence by simultaneously appropriating the token of several masters, it is worth the time-out T_{IT} the TS address is taken into account according to the formula:

$$T_{IT} = 6.T_{SL} + 2.TS.T_{SL},$$

where T_{SL} is the time slot interval. As a result, the master station with the lowest address is the first to try to restore the logical circle.

If the logical circle needs to be initialized, a credential frame with addresses SA = TS and DA = TS is first sent twice on the bus. The master then enters (28) the "Pass Token" state to create a GAPL and designate an NS station to pass credentials.

4 Use Token

A state in which the master has authorization and can therefore initiate a transmission on the bus. When entering this state, the station first determines how much time it has to use the authorization. From the "Token Rotation Timer", it finds out how long the token actually lasted in the logical circle cycle T_{RR} (Real Rotation time). By comparison with the required circulation time T_{TR} (Target Rotation time) is the maximum time to hold the token T_{TH} (Token Holding time) as:

$$T_{TH} = T_{TR} - T_{RR}.$$

If time remains ($T_{TH} > 0$), the master selects a frame to send from the request queue. Preferably, it selects the first request that is in line from the high-priority queue or, if empty, then selects the request from the low-priority queue. After sending the framework for which it is expected the response goes (11) to the "Await Data Resp" waiting state and at the same time the "Slot Timer" is started, which ensures the invocation of a time-out if the interrogated station does not respond.

If, when entering the Use Token state, the time (T_{TH}) exhausted, ie ($T_{TH} <= 0$), the master can still process one high priority request.

If a request from the queue is of a type that does not require a transmission on the bus (eg a request to activate SAP), it is processed, the response is sent to the higher layer and the master starts processing the next request.

If both queues are empty, the master (14) switches to the "Check Access Time" state.

5 Await Data Resp

The state in which we wait for a response after sending the request frame. If the sent frame was of the SDN (Send Data with No Acknowledge) type, we do not wait for a response and immediately return (13) to the "Use Token" state.

While we wait, we can accept these three types of frames:

1. Valid confirmation or response from the station we addressed. We process the response and return (13) to the "Use Token" state.
2. A valid frame, but of an unexpected type (such as a credential frame) or from a different station than we expect. Apparently an error occurred somewhere and so we go (27) to the "Active Idle" state.
3. Invalid frame, ie a frame with an incorrect Start Delimiter, End Delimiter, incorrect checksum or incorrect parity of a character. If this response arrives or the Slot Timer expires, the request is resent. If the station does not respond to repeated requests, the higher layer is notified and the master (13) enters the "Use Token" state.

6 Check Access Time

In this state, the master checks if he still has time to hold the authorization. If there is enough time left ($T_{TH} > 0$), returns (15) to the "Use Token" state. When all the time is exhausted, (16) enters the "Pass Token" state.

7 Pass Token

In this state, the FDL attempts to pass credentials to the next station in the logical circuit (NS). At the same time as sending the credential frame, the master should verify, if the framework is really successful. If the same frame does not come back from the monitor, there is probably an error in the transmission circuits and the FDL goes (26) to the "Offline" state.

If the "GAP Update Time" expires before the credential frame is sent, the oldest entry in the GAP list is restored by the master sending the "Request FDL Status" frame from the corresponding GAP and goes (17) to the "Await Status Response" status. Two responses can come to the request:

1. A response is received from the slave station or no response is received on the request (the station with the given address does not exist on the bus or does not communicate). The entry in the GAP list is updated and after sending the credentials for the NS, the master (20) switches to the "Check Token Pass" status.

2. A response is received from the master station, which is ready to enter the logical circle. The corresponding master is marked as NS (Next Station) and the GAP list is shortened accordingly. At the same time, a credential frame is sent to this master and the FDL goes (20) to the "Check Token Pass" state.

9 Await Status Response

The master enters this state after sending a "Request FDL Status" to find out the status of the station at the given address. When entering this state, an interval timer is started T_{SL}. If a response arrives or the timer expires (the station at the given address is not or does not respond), the FDL goes to the "Pass Token" state, where the result is processed.

Another possibility is that while waiting for a response, a frame of a different type than expected arrives, the FDL then assumes that another station is active and enters (24) the "Active Idle" state.

8 Check Token Pass

The master enters this state to wait for verification of a successful credential transfer. The handover is considered successful if the frame sent by the NS (SA = NS) is received from the bus. After verification, the master (23) switches to the "Active Idle" state. If the NS station fails to pass the credentials, the master (22) returns to the "Pass Token" state so that the next station after the NS in the logical circle is found to pass the credentials.

If the master is the only one on the bus (mono-master system), then immediately without authentication, which does not make sense in this case, it goes through the "Active Idle" status to the "Use Token" status.

2.2.4 Bus event handling

The strength of Profibus is well-developed procedures for dealing with exceptional events on the bus. Most of them result from the described state machine, but we will show some specifically.

Connection of the master station on the bus

After connecting the master station to the bus, the station behaves passively and only monitors the traffic on the bus. It creates a list of LAS from the frames sent over the bus and thus finds out the addresses of other master stations. After the list has been successfully created, the master waits for the "Request FDL Status" request to be received from the master station in whose GAP address space it is located. When this request arrives, it sends a response stating that it is ready. This causes a token to be sent to him immediately, which enters the logical ring and he can start actively communicating.

Token loss

If the token is lost for any reason, bus operation stops. This condition is detected by the master stations and after the time-out expires, the master station with the lowest address appropriates the token and reinicalizes the logic circuit. The length of the time-out is determined by the station address, so the master with the lowest address will try to reinitialize the circuit first.

Disconnection of the master station from the bus

If the master station is suddenly disconnected from the bus, the following two cases can occur:

1. At the time of disconnection, another master station owned the token. When it is the turn of the disconnected station, the previous station (PS) will try to pass the token to it. After a repeated unsuccessful attempt to pass the token to the disconnected station, its address in the LAS list will be marked as passive and at the same time the next station from the logical circuit to which the token will be passed will be determined from this list.

2. If the disconnected master station held the token at the time of disconnection, then the disconnection will result in its loss. This condition is handled as described in the previous paragraph.

The requested station is not responding

If the master sends a request, then it starts a time-out timer with a period with the last sent byte of the frame T_{SL}. If there is no response until it expires, then the master repeats the request. The number of repetitions is determined by the bus parameterretry ctr. If the station does not respond to retries, it is marked as inactive.

Error receiving query response

It is the duty of each station on the bus to keep in mind the response frame to the last request from the bus. If this response is lost or damaged, then the master station sends the request repeatedly. The fact that this is a recurring request is identified by the unchanged FCB bit, which is part of the Frame Control Character (FC). This bit is inverted with each new request. Therefore, if a request with the same FCB as the previous request occurred, the station repeats the last response.

Duplicate addresses on the bus

If a station detects traffic sent from a station with the same address by listening on the bus, then after receiving two such frames, it goes to the "Offline" state and signals a bus error.

Chapter 3

Physical layer implementation

When writing to ProfiM, care was taken to consistently separate hardware-dependent parts of the program from those that may be hardware-independent. This allows the controller to work on different implementations of the Profibus physical layer, or different ways to connect Profibus to a PC. These implementations may vary in complexity, cost, and maximum achievable baud rate.

The simplest connection option with minimal costs is to use a standard PC serial port with a simple RS 232/485 converter without external power supply. This will allow us to fully connect to Profibus for baud rates of 9600bps and 19200bps. Higher speeds on the standard serial port cannot be achieved for the reasons given in section 3.2.

Another option is to use expansion plug-in PCI cards with UART (Universal Asynchronous Receiver / Transmitter) circuitry, which allows you to move the speed limit much further, and with a suitable circuit type and sufficiently powerful processor, the current maximum transmission speed of 12Mbps can be achieved.

3.1 Using the standard serial port

An RS 232/485 converter must be used to connect the Profibus to the PC serial port. Profibus works on the RS-485 standard and the PC serial port on RS-232. The difference between these standards is mainly in the voltage levels they use to represent logical values. The second important difference is also that the RS-485 standard allows the connection of more than two devices on one line. Multiple devices are connected so that they transmit only one at a time and the others only receive, so devices that want to communicate via RS-485 tend to switch the direction of data flow.

A possible diagram of the converter is shown in Figure 3.1. It is a simple converter built on circuit 75176, which does not need an external power supply to operate. It is powered directly from the serial port by setting the DTR output to logic one. This simplifies the use of the converter, but also limits the overall maximum load of the bus driver. This limits the maximum number of connectable devices, the line length and does not allow the use of bus line terminating resistors necessary for higher baud rates.

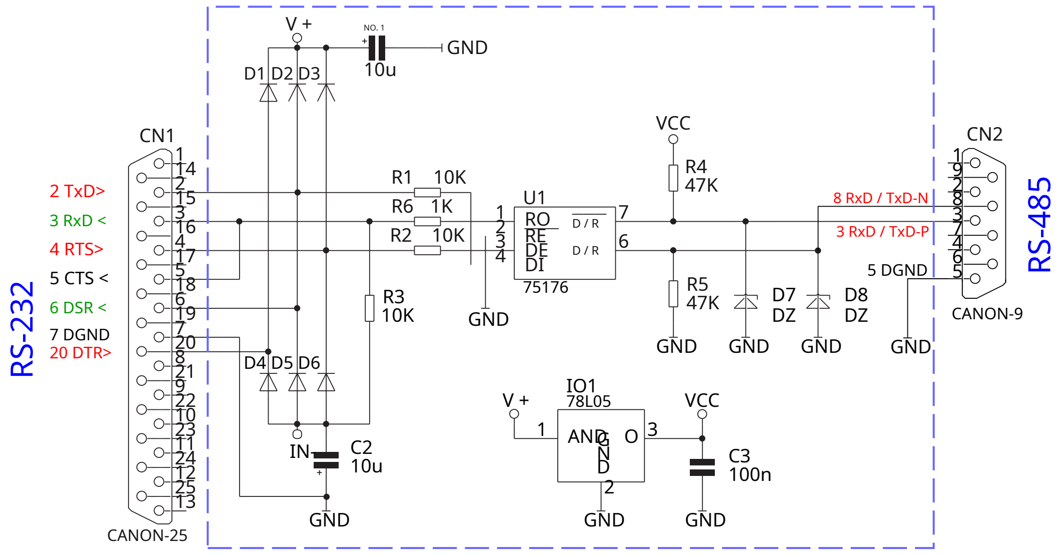


Figure 3.1: Simple RS-232 / RS-485 interface

If we need a converter with better properties, we have no choice but to use a converter with external power supply. In addition to the power line driver, it can also offer galvanic isolation, which is often used in industry due to the usual use of the RS-485 standard.

Switching the direction of data flow is an essential part of the converter. The RTS signal is used for this. For RTS = 0 the line driver outputs are in a high impedance state and the converter is thus switched to receive, for RTS = 1 the bus drivers are activated and the converter can transmit.

At this point, it is worth noting that there are converters with automatic direction switching. The principle of their operation consists in the use of a monostable flip-flop circuit for direction control tOto give. This flip-flop switches to falling edge transmissionandeach start bit n andčtransmitted character. The flip-flop period is set to a character, which for a length of poison o n himmeans that it is necessary to change this for each baud rate. length of periods y oHowever, it can be used for communication on Profibus in our case use them thatEbecause we need to use the converter to provide additional timing, to noEthe man is pt Oreba software ovlad direction for the data flow (will be described in section Next avl stnOti of this apofjeni transfer 5.3)skhardware Je-land converter switched to youír(RTS = 1), then everything that is sent to the input of the converter by the signal and the TxD with intřn dishwasher returns to RxD. In other words, everything from the serial port of the PC through the converter will also be sent immediately to the serial port input. This feature can be used to switch the direction of data flow.

An important part of the converter, which is also used by ProfiMem, is the connection between the TxD output and the DSR input. If we use a converter other than the one shown in Figure 3.1 with ProfiM, it needs to have this jumper and a hardware loop-back for the transmitted data.

A major problem when connecting a bus using the RS-485 standard via the RS 232/485 converter to most UART circuits with RS-232 is the inability to generate an interrupt from a blank output shift register (Figure 3.2). If we want to use the RS 232/485 converter, we need to control the direction of data transfer by software. This means switching the direction of the converter to transmission before each transmission and switching back to reception after sending the last bit of the last character. Switching the direction to transmission is not a problem, this can be done before sending the first byte of transmitted data, but switching back is not so easy, precisely because most standard UART circuits (Table 3.2) do not offer an interrupt from a blankoutput shift register.

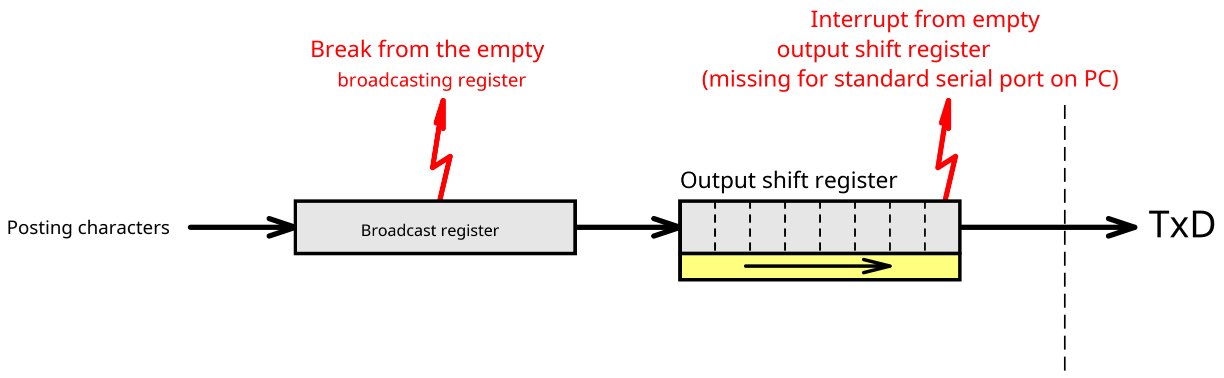


Figure 3.2: UART transmitter interruption

The only interrupt we can get from such circuits during transmission only says that it is empty broadcasting register, which we can refill, however output shift register it is not empty yet and is only finishing ejecting the character, which means that the character has not yet been completely sent from the UART. Therefore, if we switch the direction after the arrival of such an interrupt, an error occurs on the bus, because the direction of transmission is changed in the middle of the transmitted character and so only a part of it is sent on the bus.

The only way for these circuits to detect full character transmission is to read the Link Status Register (LSR), which contains a bit indicating an empty output shift register. However, since we can't generate an interrupt from this bit, it is then necessary to use infinite loops to wait, which is not a good way for programs on event-driven systems such as Windows and a completely unacceptable way if we write a driver and need this.

PC RS-232	...	9600	19200	38400		57600		111520		
Profibus		9600	19200		45450		93750		187500	...

Table 3.1: Comparison of RS-232 baud rates in PC and Profibus (values are in bps)

wait to perform in an interrupt handler (ISR) or DPC routine (see Chapter 4). However, this problem has been solved and will be described in Chapter 5.

3.2 Using expansion plug-in cards

If we compare the permissible transmission speeds offered by the standard RS-232 on a PC with the possible transmission speeds of Profibus, we will find that the tolerance required by Profibus 0.3% will only fit for speeds 9600bps and 19200bps (Table 3.1). In order to achieve higher speeds, we can only use plug-in cards with UART circuits, which have higher frequencies of the basic clock frequency with good possibilities of its division. These not only allow us to achieve the required baud rates (using a suitable basic clock frequency crystal), but also bring new possibilities over the UART circuits standardly used in PCs (Table 3.2), which are compatible with the 16C450 circuit.

Some newer UARTs offer the ability to generate an interrupt even after the output shift register is emptied. If we had this option, it would simplify the program handling of bus access and would also simplify the way the serial port is used for timing. Table 3.2 lists some of the newer UARTs along with the basic characteristics. It can be seen that the circuit that offers such an interrupt is 16C950. In addition, the new circuits offer a large FIFO buffer, which allows you to reduce the CPU load at high bit rates.

Therefore, a card for the PCI-1482 PCI bus from Tedia [10] was chosen as another hardware solution option, which is based on the OX16PCI954 circuit, which is compatible with the 16C950. It is also possible to use any card from the PCI-14xx or PCI-16xx series from this company. The ports on the card must be switched to the "RS-485" mode using the switches and the "High Speed" mode must be set using the jumper. It is also necessary to use a simple reduction (Figure 3.3), as the output connector of the card has a different signal distribution than the Profibus connector.

If we use another card with a UART circuit, then the best one is one that is also built on a circuit compatible with 16C950. If a card with a lower type of circuit 450 to 750 is used, it needs to have at least the possibility of data loop-back and connection between the TxD output and the DSR input (for reasons, see section 5.3). The program operation of such a card would then be almost the same as when using the RS 232/485 converter.

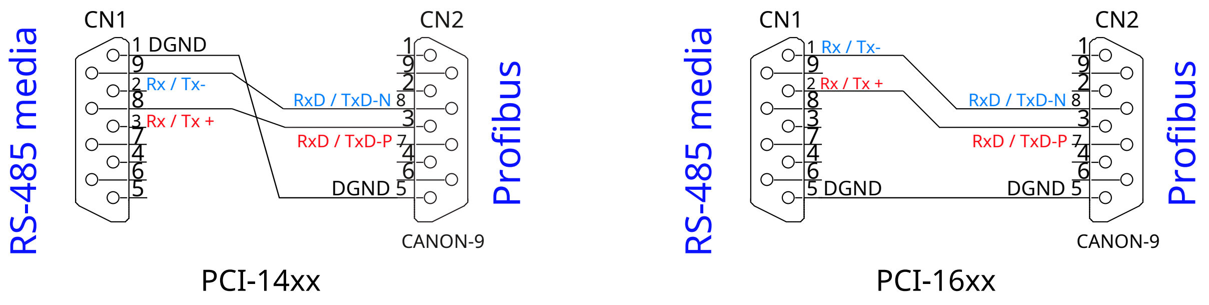


Figure 3.3: Reduction for connecting Tedia cards to Profibus

UART circuit type	FIFO size memoirs	Interrupt from empty shift transmission register
450	1B	No
550	16B	No
650	128B	No
750	128B	No
950	128B	Yes

Table 3.2: Properties of UART circuits

3.2.1 OX16PCI954 circuit

Circuit manufactured by Oxford Semiconductor [9] designed for use on PCI bus expansion plug-in cards. It is a complex circuit that offers other functions in addition to serial channels, such as parallel port, IrDA support, etc. It is important for us that this circuit integrates four high-speed UART channels, each compatible with 16C950 circuit and 128B FIFO memory for transmitter and receiver.

Depending on the speed of the base clock frequency used, the maximum achievable transfer rate can be up to 15Mbps for asynchronous mode. A good feature of this circuit is the possibility of non-integer division of the clock frequency, which better allows to achieve several atypical transmission speeds for one clock crystal at the same time.

Furthermore, this circuit is ready for connection to the PCI bus and fully integrates Plug and Play support with the ability to read the configuration from the EEPROM memory. Last but not least, it allows you to interrupt from an empty transmission shift register.

Chapter 4

Creating drivers for the Windows operating system

Since writing a driver was an essential part of this thesis, we will show in this chapter how to create drivers for physical devices in Windows operating systems. There are other types of drivers, but in this chapter we will focus exclusively on drivers for physical devices. We will describe their basic structure, show what tools are needed for their creation and how writing drivers differs from writing common programs.

If we need to create a driver for the Windows operating system, we will initially encounter a lack of information that would roughly describe the solution to this problem, show simple examples and outline the procedure. The DDK (Driver Development Kit) from Microsoft includes very good documentation, but it is too detailed for the first approach and does not offer an overall view of the issue. On the other hand, with the use of the Internet, we find many instructions and descriptions, which, however, are in the vast majority too superficial. The problem is that there is nothing like the short obligatory program "Hello World" for the introduction to driver creation. Although the creation of drivers is often based on sample source code, which are part of the DDK, even for the simplest usable cases, the number of lines is in the thousands. It is given by that even the simplest driver must meet some basic requirements and must have routines that allow the system to communicate with the driver. This also necessitates a good knowledge of the operating mechanisms of the operating system of which the driver is a part.

If we want to directly access hardware under Windows operating systems, which is our case, or use low-level system services, then we need to be in so-called privileged mode. Windows programs work on two levels. The first is the unprivileged level or User mode (Ring 3). It is designed for common applications that are not hardware dependent and will suffice with services provided via API

(Application Interface) of the operating system. Most operating system applications run at this level and do not allow privileged processor instructions, such as input and output instructions.in, out and instructions halt. Constraints allow the system to control the operation of user programs, their separation and thus increase the stability and protection of the operating system.

The second level is privileged or Kernel mode (Ring 0), in which the kernel of the operating system and its components (Figure 4.1), which includes drivers. Code executed at this level is no longer limited by user mode limits, allowing it direct access to the hardware and control over its course. At the same time, however, this increases the risk of errors. Kernel-level program errors usually lead to a system crash. Therefore, creating a driver requires very consistent and careful writing of the source code. A program loop or deadlock in critical pieces of code, such as interrupt handling, causes the system to stop and, for example, accessing memory that does not belong to the driver will cause the entire system to crash almost certainly.

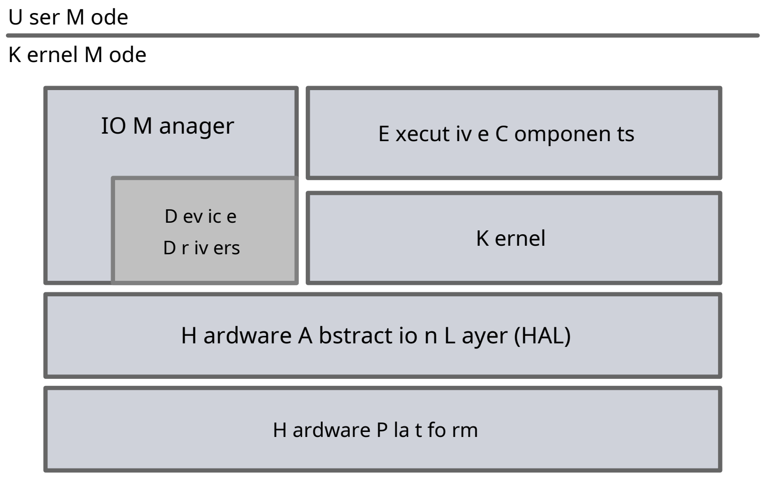


Figure 4.1: Kernel Mode

The use of the C language, which is mainly used to create drivers, also contributes to increasing these risks. His strong point is working with indicators, but thanks to great benevolence in this area, for example, an error in a program may result in the use of an uninitialized pointer referring to a non-program memory. Such a mistake can be easily made in a large project and, moreover, it does not have to show up every time, so it is necessary to write high caution when writing code.

4.1 Tools needed for driver development

Drivers can be developed in any language that uses the C language convention to call functions, but full development support can only be found for the C language (header files, source code samples). It's not even customary to use C ++, but mostly C, to write drivers. It would be possible, but Microsoft recommends using C.

The following group of tools is usually used to write drivers, which is part of "MSDN Professional Subscription" in addition to Visual Studio:

- Windows NT free build
- Windows NT checked build (not required)
- Software Development Kit (SDK)
- Windows NT Device Driver Kit (DDK)
- Visual Studio (VS or VC ++)

What does it mean checked build and free build? We often come across these concepts in the field of drivers. These are two different versions into which a driver or any other program can be compiled.Checked build This means that the program has been compiled with the addition of debugging information and advanced internal error checking, which allows us to develop more easily, but also reduces speed and increases memory requirements. Compared to thatfree build is intended for final translation with full optimization, where debugging information is separated from the code.

The DDK package is intended for creating drivers, it contains header files and static libraries that are necessary for their compilation. It needs an SDK for its work, from which it uses a compiler and other tools.

4.2 System requirements for the driver

When writing a driver, make sure that it meets certain criteria given by the operating system. Their fulfillment allows the system good flexibility and fast response to events. These criteria are:

Driver code switchability and interruptibility. Since the driver code will run under emptive multitasking operating system, must be resilient in case of switching or interruption. This means that temporarily interrupting the code must not cause it to fail. It should be taken into account that switching or interrupts can occur at any time, so it is important to ensure that there are no deadlocks or problems with shared data in such cases. The system provides a wide range of tools such as spinlocks, critical sections, etc.

It should be noted that although an interrupt masks the arrival of other interrupts of the same priority level, its handling can be suspended at any time by the arrival of a higher priority interrupt.

Hardware and software configurability. The driver should not depend on the specific hardware settings. For example, if we write a serial port driver, it should be able to work with all serial ports in the computer (COM1, COM2, etc.).

Packaging of I / O operations. All requests to the driver come in the form of packets, which are called IRP (I / O Request Packet). Each driver must have an interface that allows these packets to be received, processed, and resent in IRP form. The driver should be able to work with several pending requests at once.

4.3 Difference between Legacy and PnP drivers

Legacy driver is a name introduced since Windows 2000 for an old type of driver that does not have PnP (Plug and Play) support. If we want to create a full-fledged driver for Windows 2000 and higher operating systems, we need to meet the requirements of WDM (Windows Driver Model), which mainly include:

- Plug and Play support - PnP support is required to obtain hardware resources and unlimited driver functionality in the PnP operating system.
- Power Management Support - Devices for which the driver is written do not have to support hardware hardware directly, but the driver should at least implement its operating system interface. Otherwise, for example, if there is one driver in the system without Power Management, then the system cannot go into sleep mode.

If the driver does not meet any of these conditions, then from the point of view of Windows 2000, it is a so-called Legacy driver, ie an old type driver that can be used within the framework of Windows NT compatibility. However, the problem may occur if there is a conflict between the PnP and Legacy drivers. For example, if we want to use a Legacy serial port driver, we need to remove the standard PnP serial driver from the system, which is almost impossible. In addition, there may be problems with the PnP manager not connecting the port range needed for the legacy driver, making it easier to add PnP support to the driver.

4.3.1 Plug and Play

PnP-enabled driver requirements:

The PnP driver must not take up hardware resources directly. Instead, the driver builds a list of resources it needs and sends it to the PnP Manager. The resources requested by the driver include: IRQ interrupt request numbers, I / O port ranges, DMA channels, and memory mapped access ranges on the device. After gathering all the resource requests, the PnP manager decides how best to allocate them so that everyone can be satisfied, and which resources will be allocated to which driver. These resources are automatically prevented for the driver by the PnP Manager, of which the driver is informed via the IRP message IRP MN START_DEVICE. Only when this message arrives can the driver start accessing the hardware, and only using the allocated resources.

The PnP driver must work according to the PnP Manager. PnP Manager manages the device in the system as a whole and prompts the controller to operate its individual devices. For example, a PnP driver must not search for its device when it is loaded (DriverEntry), but must instead be designed to call the AddDevice function when it finds a device for the driver. PnP Manager calls the AddDevice function for each device found that the driver is able to operate. The device is operated by the driver until PnP Manager stops its operation.

The PnP driver should be as flexible as possible. If the device can work with different By configuring hardware resources (for example, it may use different IRQ interrupts or different I / O port ranges), the driver should provide all possible combinations of PnP Manager resources to provide the best possible way to meet all resource requirements.

4.4 How the driver works

Now let's try to roughly describe the operation of the PnP driver in Windows.

The driver is started automatically at system startup or manually at the user's initiative. When it starts, it registers the addresses of several important routines, which creates an interface between the operating system and the driver. By calling these routines, the operating system communicates with the driver. It then goes into a waiting state.

According to the information stored in the registers during driver installation, PnP Manager knows which devices it is able to operate and what hardware resources it needs. If such a device is found on the computer, the driver is notified by calling the AddDevice routine.

For each device served, a Device Object is created in the AddDevice routine, which represents this device and thus allows one driver code to operate multiple device. However, the driver does not yet have the hardware resources allocated to the device, so it goes back to waiting.

Access to the hardware is allowed only after the driver receives an IRP MN START DEVICE message from the system. The driver then determines the allocated resources and initializes the hardware. From this point on, it can begin to meet I / O operation requirements.

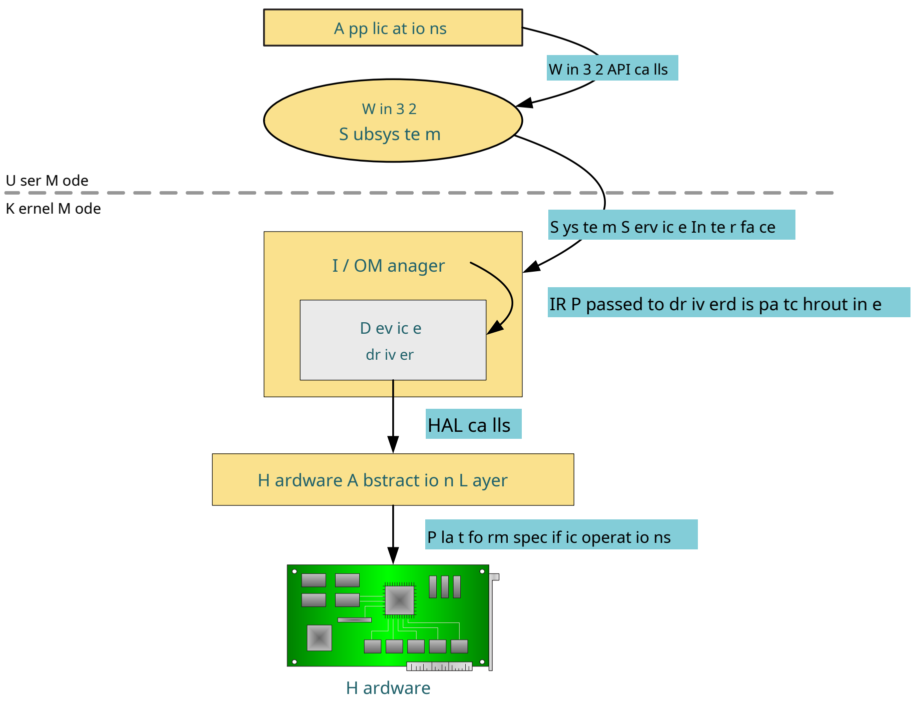

A pp lic at io ns
W in 3 2 API ca lls
W in 3 2
S ubsys te m
U ser M ode
K ernel M ode
S ys te m S erv ic e In te r fa ce
I / OM anager
IR P passed to dr iv erd is pa tc hrout in e
D ev ic e
dr iv er
HAL ca lls
H ardware A bstract io n L ayer
P la t fo rm spec if ic operat io ns
H ardware

Figure 4.2: IRP processing

If an application wants to send a request to a driver, it must first open a file handle to it, using the command CreateFile with a registered name, forming the driver identifier, such as "\\. \ProfiM ". It then sends requests to the driver via this handle. These access the I / O Manager via the API (Figure 4.2). It creates IRP packets, which it sends to the driver. He starts processing requests and holds the IRP for this time. After the request is processed, the IRP is marked as finished and the result is stored in it. The I / O manager then sends the result back to the application. The use of IRP packets simplifies the handling of requests by the driver, as their management is provided by the I / O manager.

I / O operations performed on a physical device are performed through a Hardware Abstraction Layer (HAL), which forms an abstract layer that hides the implementation details of each specific hardware platform.

4.5 Priority levels (IRQL)

A very important feature for all driver routines is the priority level at which they are run. Unlike common programs in user mode, individual driver routines are run at different priority levels (IRQL - Interrupt request level). What level the routine runs at is essential becausedetermines what kernel functions it can call. For example, some memory allocation functions, synchronization functions (KeWaitForSingleObject,. . .)or, for example, registry access functions. The restrictions are approximately so that routines run with a higher priority level do not use functions that require paged memory (reason in section 4.8) or those that are time consuming. This corresponds to the system's requirement to run routines with increased priority as quickly as possible.

The level at which a function can be called thus becomes an integral feature of all kernel functions that the driver can call and must be taken into account each time it is used. This level can be found in the documentation [12].

The level at which the routine runs is determined for each standard driver routine by the system or hardware level (DIRQL) at which the physical device interrupts (Table 4.1).

IRQL	Mask interrupts	Standard routines running at this level
PASSIVE LEVEL	none	DriverEntry, Unload, AddDevice, Dispatch routines and driver-created threads.
DISPATCH LEVEL	Masks interrupts with DISPATCH LEVEL. Over- however, interference from physical devices may occur.	DpcForIsr, CustomTimerDpc, CustomDPC, IoTimer, Cancel, StartIo routines.
DIRQL	All interruption with IRQL≤DIRQL. A higher priority interrupt may occur.	ISR and SyncCritSection

Table 4.1: Priority levels

4.6 Basic structure and routines of Legacy drivers

First, we will describe the structure of the Legacy driver and in the next section we will show its extensions to support PnP. The Legacy driver consists of several basic routines in its simplest form (Figure 4.3). On the one hand, they cooperate with the I / O Manager, through which all communication between the driver and the applications, and on the other hand, the driver accesses the hardware being serviced.

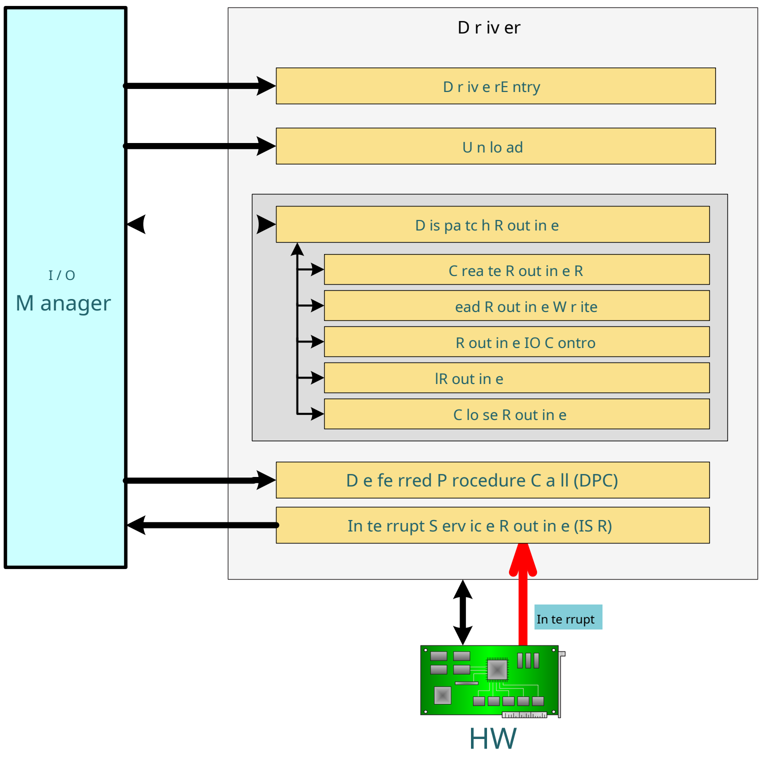


Figure 4.3: Legacy driver structure

4.6.1 DriverEntry

The basic routine of each driver. The I / O Manager calls DriverEntry immediately after the driver is loaded and started. As a parameter, it passes a pointer to the Driver Object by which I / O Manager represents a driver instance. DriverEntry sets the pointers to standard and Dispatch routines representing the entry points of driver handlers that allow the I / O Manager to interact with the driver:

```
DriverObject-> MajorFunction [IRP_MJ_CREATE] = CreateRoutine;
DriverObject-> MajorFunction [IRP_MJ_CLOSE]     = CloseRoutine;
DriverObject-> MajorFunction [IRP_MJ_READ]      = ReadRoutine;
DriverObject-> MajorFunction [IRP_MJ_WRITE]     = WriteRoutine;
DriverObject-> MajorFunction [IRP_MJ_DEVICE_CONTROL] = IOControlRoutine;
DriverObject-> DriverUnload = UnloadDriver;
```

Each handler can have its own function, but it is customary to create one Dispatch routine that handles all requests (the request type is passed as one of the parameters by the I / O Manager).

Furthermore, in DriverEntry, the driver allocates and initializes the necessary memory structures for its operation (see 4.8). A peculiarity is the usual need of the driver to get space in the so-called nonpaged memory, or such memory that is not paged by the system, so it is never stored on disk. This is necessary especially for fast access to memory in high priority states (IRQL), when access to data stored in the part of the memory that was paged would be delayed or not possible at all (eg in the interrupt handler - ISR).

In DriverEntry, the driver usually also finds out its configuration stored in the registers (see 4.9). There are two ways to get there. They can either be stored in the location specified by one of the parameters with which the I / O Manager calls DriverEntry or in the location specified from the Driver Object according to the itemDriverObject→HardwareDatabase.

Unlike PnP drivers, Legacy driver performs device hardware initialization already in DriverEntry. However, they must also obtain hardware resources from the system before the first direct access to the device. This means obtaining I / O ports, memory ranges with mapped device registers, or interrupts through which the device will be accessed directly. In Windows NT 4.0, they are managed through registry entries that are located in \Registry \ Machine \Hardware \ ResourceMap. To obtain them, the driver calls the function IoAssignResources, IoReportResourceUsage and HalAssignSlotResources. If no conflict occurs when requesting resources, driver requests are added to the registry. This mechanism prevents drivers from taking over other drivers' already used resources, causing a conflict on a device that has already been initialized by another driver.

After obtaining the resources, the driver puts its device into its initial state and usually prepares it to generate an interrupt. In order to process an interrupt, it still needs to use a functionIoConnectInterrupt register an interrupt handler (ISR).

4.6.2 Unload

A driver that can be removed at run time must have an Unload routine. It is responsible for releasing all system objects and resources that the driver has taken for its operation. Only then can the driver be removed. Deactivating the driver can cause a fatal error if an inappropriate interference sequence is used. For example, at the beginning of the Unload routine, an object is deleted, then before the interrupt is deactivated, the ISR is called and the deleted object is used in it, which causes a critical system error. It is therefore advisable to follow the procedure below:

1. Disable all interrupt sources on physical devices. Subsequently, disconnect the interrupt handler from them by calling the functionIoDisconnectInterupt.

2. Ensure that no other driver routine continues to use the resources we want to free - for example, by calling IoStopTimer we deactivate the IoTimer driver routine call, etc.

3. Release system objects and resources that were prevented during driver initialization or operation.

4.6.3 Interrupt Service Routine (ISR)

One of the most important routines of a physical device driver. An ISR is a handler that is invoked when a device that the driver operates generates an interrupt request (IRQ). It is called in increased priority mode (IRQL = DIRQL, ie the level specified as a parameter when calling IoConnectInterrupt) and thus masks interrupts of the same and lower priority level. This places high demands on interrupt processing speed. The ISR should run as quickly as possible so that the system does not lose masks. Typically, the ISR only reads the device status along with the most important data, forces the device to stop generating interrupts, and requests a DPC, or Deferred Procedure Call, a function that is later invoked by the kernel and completes the service at a reduced priority level (IRQL = DISPATCH LEVEL). interruption.

The reason for using the DPC routine is also that some functions cannot be called in the high priority mode in which the ISR is running and so they can only be executed in the DPC.

An example is data reception via a serial channel with FIFO memory. If the FIFO memory is full up to a certain limit, the serial channel will generate an interrupt indicating the arrival of data. This triggers an ISR that does not read the data from the FIFO memory directly, but only stores the interrupt type identifier, requests DPC, and quickly returns control. The system then invokes the DPC routine (already running with a lower priority level IRQL = DISPATCH LEVEL), which completes the interrupt handler by reading the received data from the FIFO memory. The DPC is already running at a lower priority level, so it can process the data and possibly pass it to the application layer.

A problem with using DPC as a routine to complete interrupt handling after an ISR has occurred is the long time delay between the ISR call and the DPC. Therefore, in some time-consuming cases, we do not avoid having to perform the entire interrupt handler in the ISR.

4.6.4 Deferred Procedure Call (DPC)

Deferred Procedure Call - The meaning of this feature for use with an interrupt handler (ISR) was described in the previous paragraph. DPCs designed to complete interrupt handler processing are also called DpcForIsr.

4.6.5 IoTimer

If the driver needs a function that would be called at regular intervals, then the system offers IoTimer routines. The smallest time interval with which it can be called is around 10*ms*and is usually used to detect time-out I / O operations on the device or to periodically check the operation of the driver (Watchdog).

When using the standard IoTimer routine, the system guarantees its regular calling at intervals of approximately one second. The function is used to start a periodic call to the IoTimer routineIoInitializeTimer. This function is nothing more than a DPC routine called after an interrupt from the system timer.

4.6.6 Dispatch Routine

If the driver-ready I / O Manager has an IRP packet request, then it calls the registered Dispatch routine. She takes it over and starts processing it. What the request is is determined by the main function code of the IRP packet (IRP MJ xxx). One dispatch routine can be registered for each master function code. Often, however, instead of several separate routines, the driver has only one routine, which the IRP processes according to the main function code.

In general, drivers handle the following requests, which are defined by these main function codes:

IRP MJ CREATE means an application request from user mode to create a file handle, which will represent the communication channel associated with the driver.

IRP MJ_CLOSE indicates that the application is closing communication with the driver.

IRP MJ READ means an I / O request to transfer data from a physical device to a system.

IRP MJ WRITE means an I / O request to transfer data from the system to a physical . In the IRP object, there is a pointer to the buffer that the application provided for the required data.

IRP MJ DEVICE CONTROL indicates a request to the driver to execute a certain operation of the device. This is specified by the IRP auxiliary code MN xxx. Their meanings are either defined by the system or they can be custom codes defined specifically for a given driver and device.

4.7 Driver structure with PnP support

If we want our driver to have PnP support included, then we need to add a few more routines to it (Figure 4.4). In addition, the importance of several routines standard, mainly due to a different way of working with hardware resources.

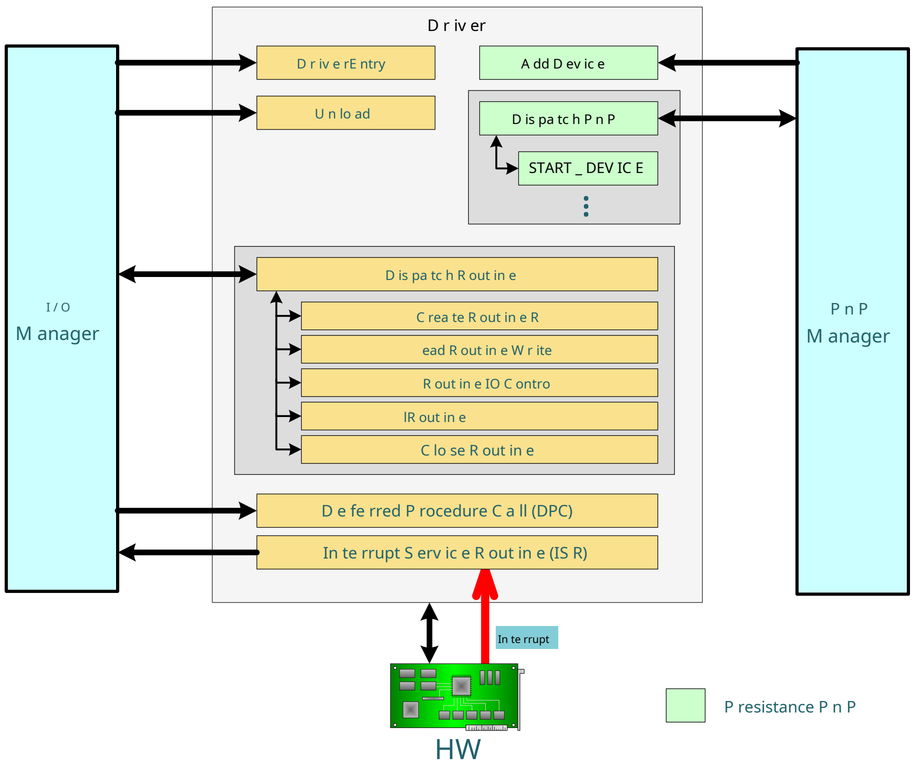


Figure 4.4: PnP driver structure

4.7.1 DriverEntry

If we add PnP support, then the DriverEntry routine is only responsible for initializing the driver, which, in addition to defining entry points for handlers, also means initializing data structures common to all devices supported by the driver. However, the hardware initialization itself moves elsewhere. The driver is informed about the found device by calling the AddDevice routine, but even in that it does not yet access the hardware.

DriverEntry is called only for the first occurrence of a device supported by the driver. For example, if there are two expansion cards on the computer that the driver supports, when the first one is found by the PnP Manager, the driver is loaded, DriverEntry is called, and then AddDevice. When the second card is found, the driver is already initialized, so adding the card starts from AddDevice. For the AddDevice routine, the DriverEntry for the I / O Manager defines a pointer to the entry point:

```
DriverObject-> DriverExtension-> AddDevice = DDAddDevice;
```

and also adds pointers to utility functions for PnP, Power Management and System Control:

```
DriverObj-> MajorFunction [IRP_MJ_PNP] DriverObj-> MajorFunction [IRP_MJ_POWER] = DispatchPnP;
DriverObj-> MajorFunction [IRP_MJ_SYSTEM_CONTROL] = DispatchSystemControl; = DispatchPower;
```

4.7.2 AddDevice

Routine necessary for every PnP-enabled driver. As mentioned, it is called when it finds a device that the driver supports. When invoked, it creates a Device Object that represents this physical, logical, or virtual device. All information about the driver created by the Device Object is stored by the I / O Manager in the appropriate Driver Object, which represents the driver instance. Therefore, AddDevice only prepares before initializing the found device.

4.7.3 DispatchPnP

Dispatch routine receiving IRP packets with the main IRP MJ PNP function code, which are sent to the driver by the PnP Manager. The requirements relate to the state of the PnP device. The most important is the IRP packet with the secondary function code IRP MN START DEVICE. Along with it, the driver gains hardware resources and can start communicating with the device.

4.7.4 Interrupt Service Routine (ISR)

With the addition of PnP support, the return value of the ISR routine becomes more important. In retrospect, it tells the system if the interrupt was for us and if we served it. This is because in PnP systems, it can happen that the interrupt vector is shared by multiple physical devices. The necessary requirement for the ISR routine is then to determine immediately after its invocation whether the interrupt was generated from the device it serves (usually by reading the interrupt identification register - for example for the IIR UART register). If not, it must return FALSE for the system to invoke another interrupt handler. Failure to do so may result in system malfunction.

4.8 Using memory with the driver

Windows uses so-called virtual memory. This allows it to use more RAM than is actually physically installed. It achieves this by dividing the memory into smaller parts - pages that can be stored in secondary memory, usually a hard disk, if needed. This snoozing or paging is common for programs and data in user mode. For programs in kernel mode, some parts of the memory need to be protected from paging.

Specifically, for example, in an interrupt handler, we cannot use any data that is not protected against paging. If this data were being stored on disk and an interruption occurred, it would be necessary to read it back into memory from disk. However, this is time consuming and we cannot afford it because the interrupt handler runs in high priority mode.

In these cases, memory allocated from so-called non-paged pool is used for data. For our case, ie writing a driver, it is customary that most global variables and data are stored in a single structure, which is attached as an "Device Extension" to an object that represents a specific instance of the controlled device (Device Object). "Device Extension" is non-paged and is created during a callIoCreateDevicein the AddDevice routine. This is called when adding a physical device and its parameter is the amount of non-paged memory required for "Device Extension".

The Device Extension is accessible from all driver routines and thus forms the basic area with variables and data. If we need to allocate non-paged memory while the driver is running (creating queues, buffers, etc.) we can use the kernel function ExAllocatePool with parameter PoolType set to NonPagedPool. It will try to allocate, but keep in mind that the "non-paged pool" is a limited system resource and the allocation may fail.

4.9 Configuration location in the registry

In Windows NT 4.0, the information in the registry related to the driver is related to the service that is registered with the driver. We can save the configuration to a branch \Registry \Machine \ System \CurrentControlSet \Services \DriverName. This is also the path that the driver obtains from the system as a routine parameter DriverEntry called when the driver starts the system.

In contrast, for a PnP driver, the configuration belongs to the specific device itself and is stored in the device entry in the registers. For example, the path to them may be for serial port COM2 \Registry \Machine \System \CurrentControlSet \Enum \ACPI \PNP0501 \2 \Device Parameters. As you can see, the location is no longer so obvious and therefore the system function is used to access the driver configuration IoOpenDeviceRegistryKey, which opens the correct registry branch according to the device.

4.10 Debuggers

Powerful debugging tools are included in almost all today's programming packages (eg C ++ Builder). They allow you to monitor the program, catch errors, step through the program, add debug points, etc. A programmer who is used to these tools will have a hard time finding their equivalent when writing drivers, especially if he writes a driver for a physical device and not, for example, a driver for the disk file system.

There are several reasons. First, the drivers run in a special mode as part of the kernel. Errors at this level in the vast majority cause a system crash. In addition, most drivers are tied to a physical device that must be operated in near real time, making it almost impossible to pause a program for debugging purposes. Another problem is caused by the fact that driver routines (interrupt handling, dispatch routines, DPC routines, etc.) can be executed simultaneously and at different priority levels.

The DDK includes a debugger WinDbg from Microsoft, which is very cumbersome and requires two computers connected by a serial line to debug drivers. A better solution is a debuggerSoftICE from Compuware. It allows you to debug drivers on a single computer and its options are wider.

Despite various complex debuggers, the most commonly used way to control the operation of the driver is to print text strings directly from the program using the function DbgPrint (ntddk.h), for example:

```
    DbgPrint ("ProfiM: Allocated Resources Port = 0x% x Irq =% d",
                    DeviceExtension -> Port, DeviceExtension -> Irq);
```

This call causes the string to be written to standard debug output. It can be displayed using a freely distributable programDebug View from Sysinternals, which is on the enclosed CD.

4.11 INF File

An essential part of the driver for Windows 2000 / XP operating systems is the so-called INF file, which is a text file that contains all the necessary information for installing the driver in the system. It lists which files are part of the driver and where to copy them, which devices the driver supports, information for writing settings to the registry, and much more.

The structure of the INF file for the driver is quite complex and complex, so it pays to use the tools for the first version geninf.exe, which is part of the DDK. After generating several dialog boxes with basic information about the driver, it will generate a rough skeleton, but manual editing will still not be avoided.

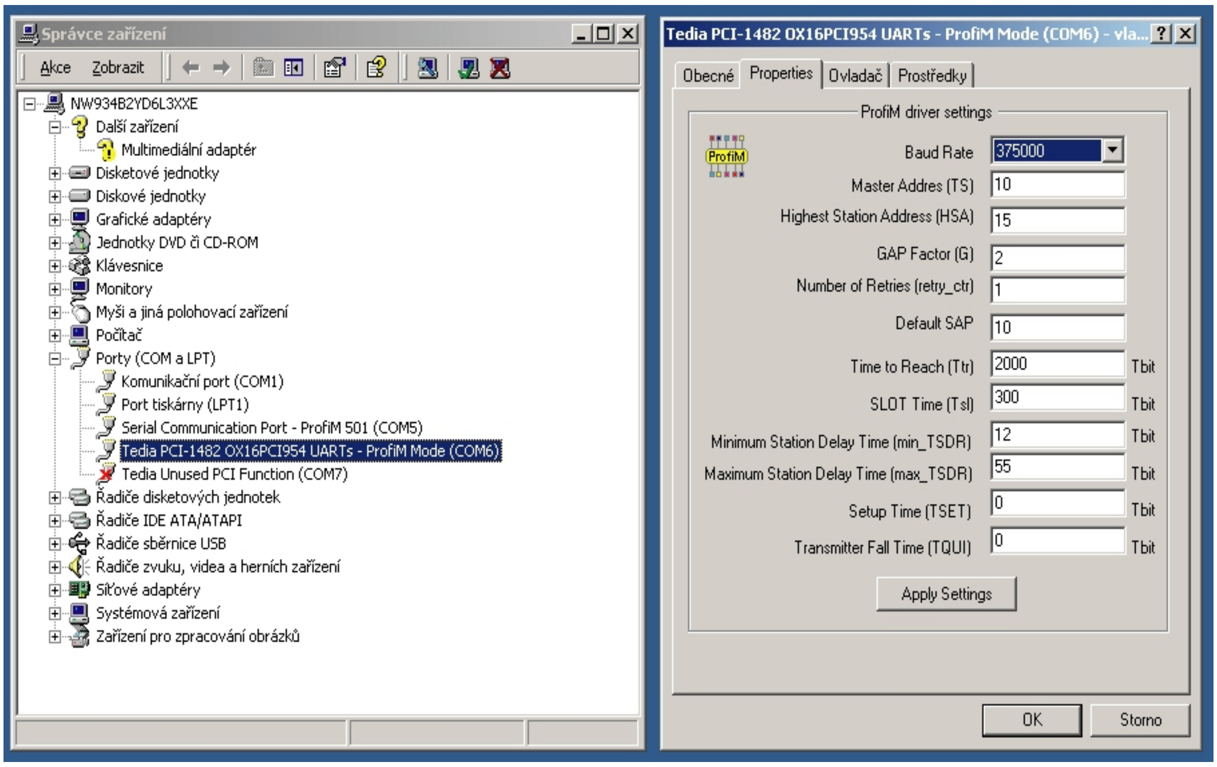
Important information for the operating system that the INF file contains is a list of devices that the driver supports and with which it can work. For example, this section might look like this:

```
[ControlFlags]
ExcludeFromSelect = PCI \ VEN_1760 & DEV_8004
ExcludeFromSelect = PCI \ VEN_1760 & DEV_8005
[Tedia]
% PCI \ VEN_1760 & DEV_8004.DeviceDesc% = PCI_950A,% PCI \PCI \ VEN_1760 & DEV_8004
DEV_8005.DeviceDesc% = NoDrv, PCI \ VEN_1760 & DEV_8005
```

It tells the operating system that the driver supports PCI devices with manufacturer numbers 1760 and device numbers 8004 and 8005. These numbers uniquely identify each PCI bus device and are assigned to hardware manufacturers by the international PCI-SIG.

4.12 Property Page

If we have a driver created, we almost certainly need to set its working parameters in some way. To do this, it is best to create a so-called Property Page. This allows us to add several pages to the device properties window (we can access it via the Device Manager by opening the properties window for a specific device - Figure 4.5).



NW934B2YD6L3XXE
Další zařízení
Multimediální adaptér
Disketové jednotky
Diskové jednotky
Grafické adaptéry
Jednotky DVD či CD-ROM
Klávesnice
Monitory
Myši a jiná polohovací zařízení
Počítač
Porty (COM a LPT)
Komunikační port (COM1)
Port tiskárny (LPT1)
Serial Communication Port - ProfiM 501 (COM5)
Tedia PCI-1482 OX16PCI954 UARTs - ProfiM Mode (COM6)
Tedia Unused PCI Function (COM7)
Řadiče disketových jednotek
Řadiče IDE ATA/ATAPI
Řadiče sběrnice USB
Řadiče zvuku, videa a herních zařízení
Síťové adaptéry
Systémová zařízení
Zařízení pro zpracování obrázků

ProfiM driver settings

Baud Rate	375000	
Master Addres (TS)	10	
Highest Station Address (HSA)	15	
GAP Factor (G)	2	
Number of Retries (retry_ctr)	1	
Default SAP	10	
Time to Reach (Ttr)	2000	Tbit
SLOT Time (Tsl)	300	Tbit
Minimum Station Delay Time (min_TSDR)	12	Tbit
Maximum Station Delay Time (max_TSDR)	55	Tbit
Setup Time (TSET)	0	Tbit
Transmitter Fall Time (TQUI)	0	Tbit

Figure 4.5: Property Page

Another option is to write your own application, which will allow you to set parameters after its launch. However, the property page has several advantages:

- One driver can handle multiple devices - the Property page is automatically used for each instance and its specific parameters.
- When opening the Property page, we get the path to the registers to the saved device parameters from the system.
- It is not necessary to search for an application for setting parameters. The property page is directly accessible from the Device Manager.

The property page is a dynamic-link library that has an operating system-defined access point:

```
BOOL APIENTRY PropertyPageProvider (LPVOID Info,
                                    LPFNADDPROPSHEETPAGE AddFunc,
                                    LPARAM Lparam)
```

If the system needs to display our Property page, it loads the dynamic library and calls this function. Its task is to create a dialog window, which is added as a bookmark to the device properties window, then reads the parameters stored in the registers and starts processing messages from the system, which are extended for the Property page message loop compared to the normal window.

Chapter 5

Line layer implementation - ProfiM

The behavior of the line layer is very well described by the state machine, as shown in Chapter 2. This is also the way in which the line layer can be implemented. It is created as a software state machine with ten basic states (see section 2.2.3), which is controlled by the following events after its start:

- receive request from FLD user interface (interface for higher layers or direct access to FDL)
- receiving a character from the bus (receiving the entire frame)
- timer event (time-out expiration, for example while waiting for a response or monitoring bus activity)

These events cause an appropriate response and transitions between states. Although the principle of the state machine is very simple, it can also implement very complex and complex behavior.

5.1 Program implementation

In the first phase, the DP master was created as a common program for user mode. Standard operating system services were used to access and control the serial port. The first more serious problem, which appeared right at the beginning, was the control of the RS 232/485 converter. The main problem was switching the direction of data flow, which caused constant problems.

Windows seems to offer a solution to control the converter, as described in Chapter 3.1. Using the serviceSetCommState, setting the parameters of the communication device, automatic direction switching can be activated by the RTS signal (by setting the parameterfRtsControl to the value of RTS CONTROL TOGGLE). This should ensure that the direction of the transmission is switched after the data is written to the port and that the reception is switched immediately after the transmission is completed. Little however, the documented fact is that this switching has long delays in switching back to receive.

It is not implemented in hardware (UART circuits up to 16C950 do not offer this option), but instead a function connected to a system timer with a long period is used to control the direction. It is triggered at intervals of approximately 20ms and, if it detects that the transmission has ended, it switches to receive. However, the time delay that can occur between sending the last character of the frame and switching back to receive is too long to use on Profibus. The interrogated station on the bus can then start sending a response to our request while the outputs of the converter are still holding the bus, which will cause a collision.

Manually switching the transmitter's direction of transmission also proved almost impossible for the reasons described in 3.1; Therefore, the DP master has been rewritten to the driver to run in kernel mode. This also made it possible to later add support for PCI plug-in cards with UART circuits as an alternative to connecting the Profibus at high baud rates.

However, with the transition to the driver, it was necessary to rewrite the entire source program from C ++ to C. The first version was built using object-oriented programming, but C does not allow this, so it was necessary to delete objects and return to non-object access.

The driver was first created only for the Windows NT 4.0 operating system, which does not require PnP support for the drivers. It was added later, so it can be used for Windows 2000 and XP operating systems.

5.2 Profibus requirements

To enable work with Profibus, we generally need to ensure the full ability to communicate via the RS-485 bus and, above all, to be able to time with short periods, which is a big problem if we need to achieve such periods in Windows. To give you an idea of the lengths of these periods that we need to achieve, let's take the example of a Profibus running at 1.5Mbps. All time intervals for Profibus, such as time-outs, maximum delays and necessary time delays, are given with the unitt_{bit}. This is defined as the interval given by the transmission length of one bit over the bus, or the inverse of the baud rate $t_{bit} = 1/$(baud rate in bps). For 1.5Mbps speed coming out $t_{bit} = 0,67.s$. From a practical point of view, given the The accuracy of the resolution is about ten times smaller, so for a speed of 1.5Mbps we need to reach a minimum period of about 6.s, which is unattainable from a Windows user mode perspective. There is a minimum achievable period in the order of milliseconds.

The only useful feature that Windows offers for working with short time intervals valy is Perfomance Counter. It allows you to detect high-resolution moments. It is a counter whose time resolution reaches an accuracy of around 1 on most computers.*s*, which is sufficient for our needs.

The reason why ProfiM managed to ensure the generation of such short periods is the effective use of the possibility of interrupts from the serial port with the connected RS 232/485 converter, or interrupts from the expansion card. We can afford this mainly due to the fact that ProfiM is written as a driver with direct access to the hardware.

5.3 Converter control and its use for timing

As already described in section 3.1 for the standard serial port on the PC, we do not have an interrupt from an empty output shift register. We therefore need to help by using the hardware features of the converter (Figure 3.1), such as loop-back returning all data transmitted by the TxD output to the RxD input (when the converter is switched to transmission) and the connection between the TxD output and the DSR input.

The use of the serial port together with the converter, which provides access to RS-485 and at the same time sufficiently fast timing, solves the following modes of operation:

Broadcasting characters

Before starting the transmission of the frame (block of data), switch the converter to transmission (RTS = 1). Thanks to the loop-back in the converter, each sent character will also be received back (Figure 5.1a). With the arrival of each interrupt indicating the reception of a character, we determine whether it is the last character of the frame. With its arrival we can switch the converter back to receive (RTS = 0). Thus, the switch cannot occur at an inopportune moment (in the middle of the last transmitted character), because we do it only after receiving an interrupt indicating the return reception of the last character, when the character had to be sent and not after an interrupt indicating an empty transmission register.

Receiving characters

RTS is at logic zero, which means that the outputs of the bus drivers are in a high impedance state and the serial channel can receive characters from the bus (Figure 5.1b). However, this state is hardly used during ProfiM's work, because if we do not transmit, a time-out always runs, for example a time-out while waiting for a response, which corresponds to the following mode.

Timing with the current receiving characters

The most demanding and also the most important mode. In it, the converter is switched to receive (RTS = 0), so the loop-back loop is interrupted and the converter can receive characters from the bus (Figure 5.1b). At the same time, however, it is required to measure a time interval (usually time-out).

By timing from the serial port, we get a resolution of approximately 11 t_{bit}, which corresponds to the interval required to send one character, which has one start bit, one stop bit and one even parity bit in addition to the eight data bits. Required interval in units t_{bit} so divide by 11 and the resulting value then determines how many characters need to be sent to measure the required interval. Since the loopback loop is broken, we can start sending arbitrary characters to the converter - For our purposes, as we will see later, the sign is broadcast 0x00. These do not reach the external bus and do not interfere with the reception of characters.

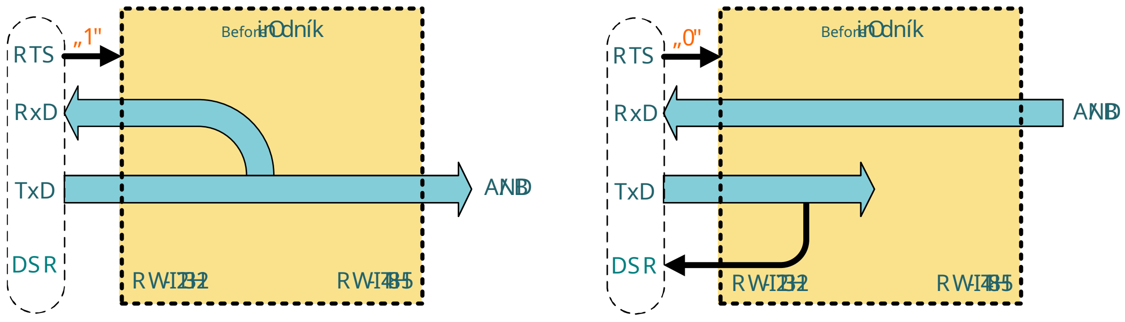
Again, we need to solve the problem of switching the direction of data flow on the converter. In this mode, we do not transmit on the external bus, but after receiving the required data or timeout, we must ensure a correct switch to the transmission that usually follows.

The completion of the transmission of each character is indicated by an interrupt announcing an empty transmission register, thus receiving a signal to transmit the next character. When we get an interrupt announcing the sending of the last character, we have a problem. Although the transmission register is empty, the character is still output from the output shift register. The loopback solution from the character transmission mode is not applicable here because the loop-back is disconnected. Immediate switching of the direction with the arrival of the last character would cause the timing character to get on the bus, which is unacceptable.

To solve this problem, there is a jumper between the TxD output and the DSR input in the converter connection (Figure 3.1). This input allows us to generate a so-called "modem interrupt". We will use it as follows. After receiving the interrupt from the last transmitted character, we enable the interrupt from the modem. This interruption is due to the fact that the transmitted timestamp has a value0x00, arrives when the leading edge of the stop bit is sent to the converter. We can already afford to switch direction to broadcasting. A single stop bit value means the idle value on the RS-485 bus. After switching, we will disable the interrupt from the modem so that the next transmitted character does not cause us an interruption.

Synchronization gap It is used if we need to insert a time delay before the transmitted data frame. For example, for each query frame, the Profibus standard requires a delay of 33 t_{bit}. It works similarly to the previous mode of operation, only it does not expect to receive any characters.

In fact, ProfiM works by constantly sending characters to the converter. This is either data intended for the external RS-485 bus, ie Profibus, or these are timing features that do not pass on the external bus.



(a) broadcastsnízn andt b) prutm ofandů + časoyouand

Figure 5.1: Use of RS 232/485 converter

If we use an expansion card with a UART circuit type 16C450 to 16C750, then the method of access to the Profibus bus and the method of timing are the same. The only difference is that some cards already have an RS-485 output, but the need for software control of the direction remains (due to the possibility of timing).

5.4 Timing with 16C950 circuit

This circuit is much easier to work with, as it is a circuit that provides an interrupt from an empty output shift register. This eliminates all the complex software constructions that tried to circumvent this shortcoming.

On the other hand, the direction of data flow on the RS-485 bus is still program-controlled and the principles used for timing remain. All that is needed is a hardware loop-back and a jumper between the TxD output and the DSR input, which were the hardware resources of the converter that made it possible to replace the interrupt from the empty output shift register.

The use of this circuit is very reliable and the baud rate limit is only a matter of processor performance and the maximum allowable CPU load.

5.5 Principle of ProfiM operation

The block diagram of ProfiM is shown in Figure 5.2. The central block is the FDL layer state machine. All other parts are subject to this.

The bus is accessed via the "Tx / Rx control, timing" block. It forms an interface to the physical layer and therefore its implementation depends on the hardware used, because it directly accesses it and uses its capabilities. In general, the function of this block is to provide access to the bus, especially when it is necessary to use a converter (port with RS-232 standard).

For the state machine, it provides frame transmission, indication of the event of receiving a character from the bus and, above all, provides timing of short intervals, for which it uses the possibilities of the hardware layer and a possible converter (especially the possibility of hardware interrupts). In the version of ProfiM, which is a part of this diploma thesis, this block supports standard RS-232 serial port and PCI expansion cards with OX16PCI954 circuit.

On the other hand, the FDL layer is oriented towards higher layers. It consists of a static libraryfdl rb.lib (the name is chosen to maintain maximum compatibility with the Siemens solution [6]), which is linked to programs requiring communication with the FDL layer. The program first opens the communication channel to the FDL layer and then can start sending its requests (description in chapter 6). Requests are stored in two queues according to priority (high / low).

When our master obtains an initiative on the bus (by obtaining a token), it selects one request from the request queues and starts processing it. Most requests involve bus communication, but some are only local (for example, master configuration or SAP activation). After the request is processed, the result is added to the result queue. In addition to responding or acknowledging a favorably processed request, this may indicate an error or time-out.

The results are selected from the queue by the application via the FDL layer API. An important feature in the communication between the application and the FDL layer API is that from the application's point of view, each open communication channel with the FDL layer has its own queue of requests and results. This means that one FDL layer (one DP master) can be shared by multiple applications at once without any conflict between requirements and results. However, the resources that the DP master offers, such as SAP access points, must be shared.

As we implement a master station, the FDL layer state machine provides not only services for higher layers but also its share in the administration and management of communication on Profibus. These include monitoring bus traffic to detect error conditions, creating and updating a list of active LAS stations and a GAP list, which monitors the occupancy of individual Profibus addresses for the address range belonging to our master station.

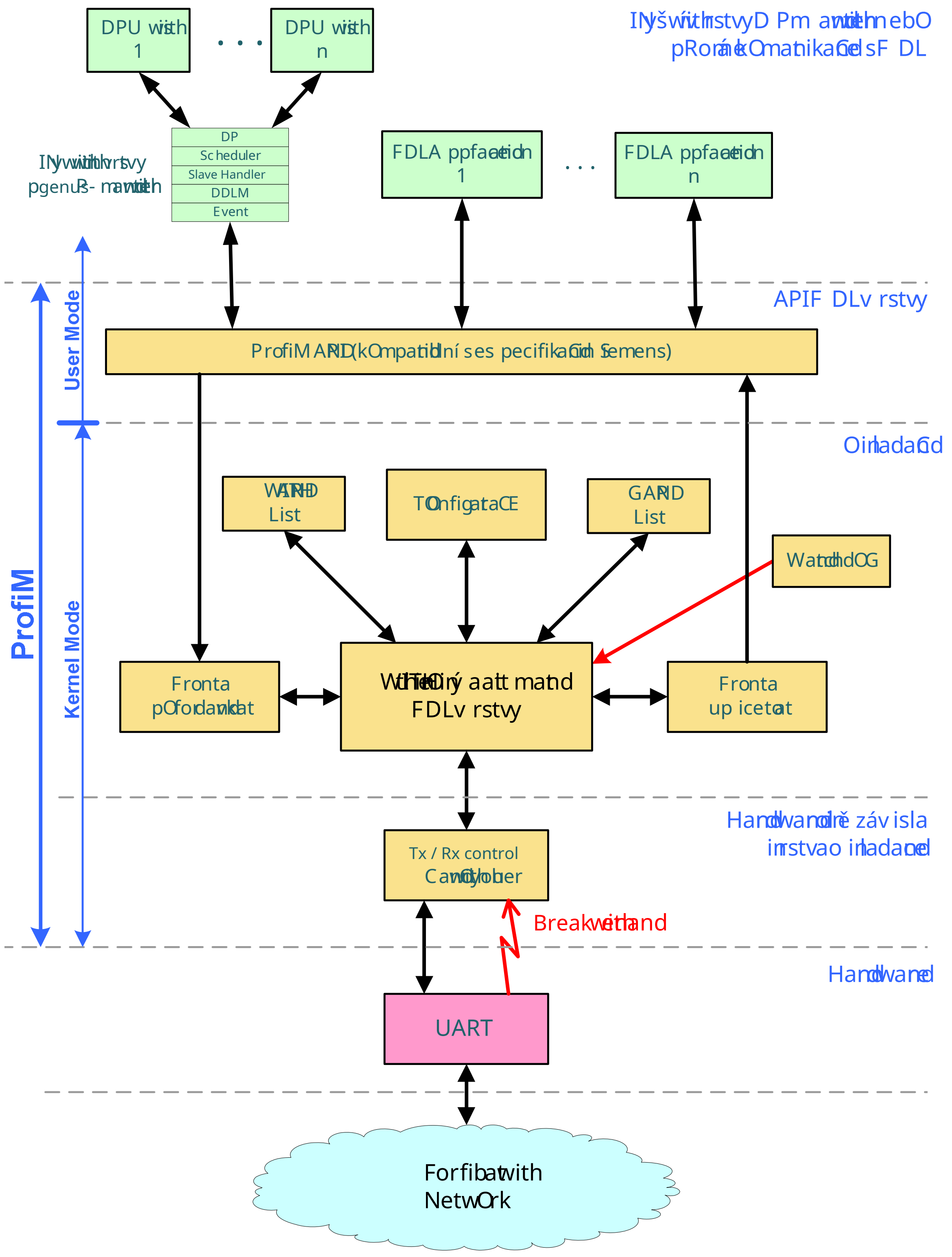


Figure 5.2: ProfiM block diagram

5.6 Watchdog

A watchdog has been added to ProfiM to increase reliability. It monitors the master's activity and resets it if it is not used for a long time.

It is practically implemented as a routine, which is called by an independent system timer at regular intervals of approximately one second. An activity flag (variable) is used to detect master inactivity, which is set at regular intervals during master action. After calling the watchdog routine, this flag is checked and reset immediately. If this flag is not set during the watchdog check, it means an error and a master reset is performed.

The watchdog is mainly used in case of errors in the interrupt system or in case of using the converter, when disconnecting the converter from the port stops the operation of the state machine, the operation of which is directly dependent on the converter (does not apply to plug-in cards with 16C950). The watchdog then restarts the master after reconnecting the converter.

5.7 CPU load

Since ProfiM does not use any special hardware when implementing the Profibus DP master, only the processor bears the entire load. In addition, the specific feature of Profibus is that the bus operation does not stop even in cases where no data is transmitted. Each DP master station must participate in the token logic circle, look for changes in its corresponding GAP address space, and in addition constantly monitor traffic and analyze all frames.

The graph of the processor load measured with ProfiM using different baud rates is shown in Figure 5.3, the average numerical values are summarized in Table 5.2. Processor load values were measured on a computer with an AMD Athlon XP 1600+ processor and a Tedia PCI-1482 serial port expansion card with an OX16PCI954 circuit.

As can be seen, CPU usage is not a bigger problem for lower speeds. Especially when using the standard serial port, where we have only speeds of 9600bps and 19200bps, the increase in load is almost non-existent. However, if we want to work at high transmission speeds, it is necessary to take into account the increased load and possibly adjust the time parameters of the network, which will allow longer time delays, such as T_{SL} and $maxT_{SDR}$.

Baudrate [kbps]	9.6	19.2	45.45	93.75	187.5	375	500	750	1500
CPU load [%]	~1%	~1%	~2%	6%	10%	17%	20%	25%	50%

Table 5.2: CPU load by baud rate

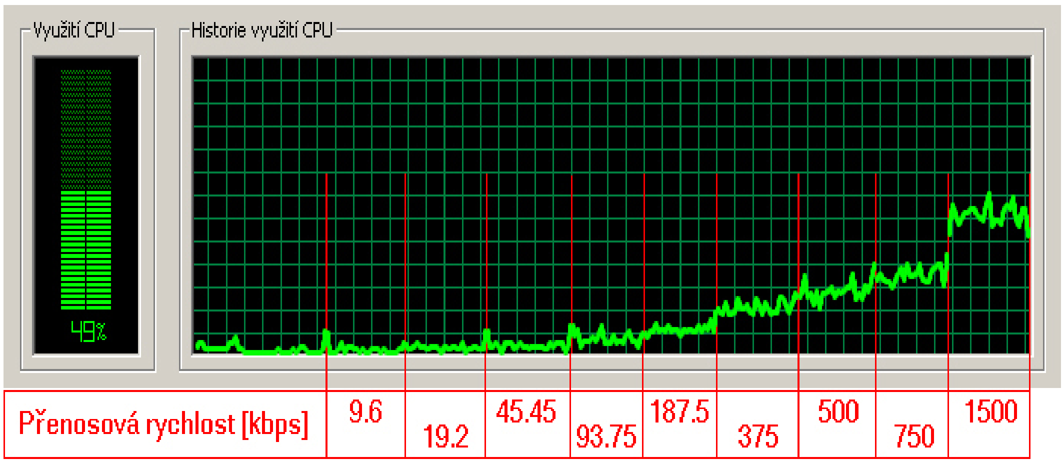


Figure 5.3: CPU load according to baud rate

Chapter 6

FDL layer application interface

The FDL application interface of the ProfiMu layer is created according to the standard defined by Siemens [6]. According to this standard, communication with the FDL layer takes place using a unified data structure called Request Block (RB). This structure is a kind of universal framework that can represent not only any request, but also the response to a request or event flag. The size of the Request Block is approximately 600B and contains many items, only some of which are currently in use.

The communication takes place in such a way that the application (higher layer of the DP master or directly of the FDL application) fills the Request Block structure to pass the Request to the FDL layer. Depending on the type of request, only certain items of this structure are met and others are left unused. Subsequently, the structure is sent to the FDL layer, which performs the given action as required and sends the Request Block back to the application as a response (Confirmation). It is sent both in the case of successful processing of the request and in the event of an error.

The Request Block is also used as an event indication in the FDL layer (Indication). If an event occurs (eg data arriving from another master), the FDL layer sends a Request Block to the application with the appropriate settings according to the type of event.

The application interface consists of five simple functions SCP_xxx for direct communication with the driver using the Request Block structure:

SCP open (char * name)	
Opens the communication channel between the application and the FDL layer.	
Parameters	
on me	specifies the driver name. The first instance of ProfiM is named "\\.\ ProfiMÿ next then „\\.\ProfiM1ÿ, „\\.\ProfiM2ÿ. . .
Return value	
The function returns a handle to the communication channel between the driver and the application. This value is used by all other functions to call the driver. If the handle cannot be created, the function returns a system-defined valueINVALID HANDLE VALUE.	

SCP send (int handle, UWORD length, char * rb)	
Sends an FDL to the Request Block layer representing the request.	
Parameters	
handle	communication channel opened by SCP open
* rb	pointer to the sent Request Block
length	Request Block length
Return value	
SCP SUCCESS	the driver received a Request Block.
SCP ERROR	an error occurred during transmission. The specific error code is returned by the function SCP get wrong.

SCP receive (INT handle, UWORD timeout, UWORD * data only, UWORD length, CHAR * buffer)	
This feature allows data, request results and event indications to be retrieved from the FDL layer. The transmitted data is again in the form of a Request Block and the application can use it according to the parameter settingstimetou wait synchronously or asynchronously.	
Parameters	
handle	communication channel open via SCP open
timeout	determines in milliseconds how long the function will wait for the result. In addition to the number of milliseconds, values can be usedSCP NOWAIT, which causes the data to be returned only if it is already available at the time the function is called and will not be waited for. Another possible value isSCP FOREVER,which causes the function to wait until the result is available returns the
* data only	length of the data written to the buffer
length	buffer size for received data - should be at least the size of a Request Block
* buffer	buffer pointer for returned data
Return value	
SCP SUCCESS	the Read Block was successful.
SCP ERROR	an error occurred while reading. The specific error code is returned by the function SCP get wrong.

SCP close (int handle)	
Closes and releases the communication channel handle to the FDL layer.	
Parameters	
handle	specifies the communication channel to close
Return value	
SCP SUCCESS	the communication channel was successfully closed.
SCP ERROR	error closing communication channel. The specific error code is returned by the functionSCP get wrong.

SCP get errno ()
It is called if any of the previous functions fail and return a value SCP ERROR.
Parameters -
Return value The return value indicates a specific error code. These error codes are described in [6].

6.1 Line layer services

The FDL layer offers the following data exchange services (details [6]):

- SDA
- SDN
- REPLY UPDATE SINGLE
- REPLY UPDATE MULTIPLE

and the following link layer administration and management services:

- FDL READ VALUE
- SAP ACTIVATE
- RSAP ACTIVATE
- SAP DEACTIVATE
- LSAP STATUS
- FDL LIFE LIST CREATE REMOTE
- FDL LIFE LIST CREATE LOCAL
- FDL IDENT
- FDL READ STATISTIC COUNTER
- AWAIT INDICATION
- FDL EVENT
- WITHDRAW INDICATION

for each of these services, the format and meaning of individual items of the Request Block according to [6] are determined. ProfiM adds several additional features to simplify the creation of the correct Request Blocks for these requests. According to the parameters, they create a Request Block corresponding to the request type and send it to the FDL layer. The most used of these functions are the following:

SAP activate (int DriverHandle, BYTE sap nr, BYTE ACCSAP, BYTE ACCSTAT, BYTE SDA R, BYTE SDN R, BYTE SRD R, BYTE priority)
Activates the Service Access Point with number sap no.

Send SRD Hex (int DriverHandle, BYTE adr, UBYTE ssap, UBYTE dsap, unsigned char * data, BYTE priority)
Sends a request to the FDL layer to send a data frame to the address station adr with waiting for a response - Send and Request Data (SRD). Allows you to enter data in hexadecimal format as a string - eg "B8 12 45 ".

Send SRD Bin (int DriverHandle, BYTE addr, BYTE ssap, BYTE dsap, unsigned char * data, int length, BYTE priority)
Sends a Send and Request Data (SRD) with binary data to the FDL layer.

Send SDN Hex (int DriverHandle, BYTE addr, BYTE ssap, BYTE dsap, unsigned char * data, BYTE priority)
Sends a request to the FDL layer to send a data frame to the address station adr no response - Send Data with no Acknowledge (SDN). With data entered in hexadecimal format as a string - e.g. "B8 12 45 ".

Send SDN Bin (int DriverHandle, BYTE addr, BYTE ssap, BYTE dsap, unsigned char * data, int length, BYTE priority)
Sends a Send Data with no Acknowledge (SDN) request to the FDL layer with data in binary form.

Read FDL value (int DriverHandle, BYTE priority)
Allows you to find out the current network parameters.

FDL life list remote (int DriverHandle, BYTE priority)
Creates a list of active stations on the bus.

6.2 Communication with FDL layer

The method of communication with the FDL layer usually depends on the complexity of the program (examples are given in Appendix A). In simple applications, a synchronous approach is usually sufficient. The application sends the request and usingSCP receive with the timeout parameter>0 starts waiting for the result. During this callSCP receive the program is suspended at the call point until the FDL layer returns a result for the application. This method is straightforward, but does not allow you to send multiple requests at once and respond to events in the FDL layer.

Another more complex way is to create a separate thread that only serves to receive Request Blocks from the FDL layer. He will call when he startsSCP receive with parameter timeout =SCP FOREVER and starts waiting for the Request Block. Upon its arrival, the Request Block is passed for processing and the thread calls againSCP receive with parameter timeout =SCP FOREVER. However, the receiving thread needs to distinguish for whom the result is intended (requests can be processed from one application at a time). The Request Block item named is used for thisuser, which is intended for the needs of the application and the FDL layer retains its value when processing the request.

A static library is prepared for the use of ProfiM when writing applications fdl rb.lib and a header file with all the necessary definitions fdl rb.h. The names are chosen in the same way as the original Siemens libraries. To use ProfiM in a project that originally used a solution from Siemens, all you have to do is replace these two original files with ProfiM files, change the name of the device being opened in the functionSCP open from for example the usual "/ CP L2 1: / FLC" to " \\.\ProfiMÿ and thus the replacement should be ready.

Chapter 7

Conclusion

Despite the considerable difficulties that accompanied the development of this thesis, we finally managed to achieve the goal and perhaps even exceed the original expectations. The result is the implementation of the Profibus DP Master on a PC, created up to and including the FDL layer. The work allows to connect Profibus via a serial port with transmission speeds of 9600bps and 19200bps or to achieve higher speeds to use PCI cards with UART circuit (16C950). With the use of an expansion card, it was possible to reach speeds of up to 3Mbps, but this was only limited by the performance of the processor (for configuration of the test set, see 5.7).

In addition, it was possible to link this diploma thesis with another diploma thesis from previous years [2], which implemented the higher layers of the Profibus DP Master, thus creating a fully fledged implementation in all layers up to the application.

It can be assumed that this work could be of interest to companies working in the field, as the opportunity to implement a Profibus DP Master with minimal costs, as this work allows, will certainly be attractive to them. Especially since nothing similar has yet been created for the PC, due to the obvious implementation complexity.

The software implementation was created as a driver for Windows NT, 2000 and XP operating systems, which can also handle up to several Profibus connections on one computer and can be used synchronously or asynchronously by several applications.

Because Windows operating system environments are not designed for real-time applications, due to high baud rates, there is no guarantee that the DP Master made by ProfiM will always adhere to the standard time intervals. On the other hand, this happens exceptionally, and even in the worst cases, the Profibus design guarantees a trouble-free resumption of bus operation. This also results in the impossibility of deploying ProfiM for applications requiring high reliability.

For a good possibility of using the driver by applications, an application interface of the FDL layer was created, which is compatible with the application interface used by Siemens. For programs that use it, this allows you to very easily replace the original hardware solutions for ProfiM. To exchange, simply copy the ProfiMu static library into the project and change the name of the device being opened in the functionSCP open._

In diploma theses from previous years, which dealt with Profibus, it was customary to add bus traffic reports from the Profibus analyzer to the diploma thesis as proof of functionality. However, this seems confusing and not very convincing, and therefore, as an example of the proper operation of the Profibus DP master, a simple demonstration application was created in the engine room on Charles Square, where two models of a belt conveyor and a stepper motor manipulator are controlled using a computer with ProfiM (Appendix B) . All sensors and actuators are connected to two input and output modules, which are connected to Profibus as slave units. In addition, the bus through which communication takes place can be shared with another master, which is a PLC from Siemens.

7.1 What could be improved or added

- Rewrite ProfiM for one of the real-time operating systems, such as RT-Linux, or use some real-time extension for Windows. This could ensure high operational reliability and precise adherence to bus time parameters.
- Add Power Management support to the driver. It is not necessary for proper function, but it should be part of it.
- Further code optimization will reduce CPU usage, which would allow higher transfer rates to be achieved.

Attachment AND

Example of using ProfiM

If we want to use ProfiM in a program that originally used a Siemens-compatible interface to access Profibus, or if we want to create a new program, we add the static library "fdl rb.lib" and the header file "fdl rb.h" to the project. _

A simple example

The following code section shows the easiest way to use ProfiM. It is a single-threaded program that initializes the Siemens ET-200B I / O module and performs one iteration of the data exchange. All waiting for the resultSCP receive they block and stop the program.

```
# include "fdl_rb.h"

int DriverHandle;
int SlaveAddress = 13;

void main ()
{
    unsigned    short       length;
    fdl_rb                  rb;
    BYTE                    Buffer [256];
    BYTE                    out, in1, in2,          in3;

    / * Open communication channel to driver * /DriverHandle =
    SCP_open ("\\\\. \\ ProfiM");

    / * Activation of SAP access points * /SAP_activate
    (                       DEFAULT_SAP,     ALL, ALL,      SERVICE_NOT_ACTIVATED,
                            BOTH_ROLES,     BOTH_ROLES,    high);
    SCP_receive(DriverHandle, 3000, & length,sizeof(rb), (char *)SAP_activate (62,    & rb);
    ALL, ALL, SERVICE_NOT_ACTIVATED,
                           INITIATOR, INITIATOR, high);
    SCP_receive(DriverHandle, 3000, & length,sizeof(rb), (char *) & rb);

    / * Parameterization frame sent * /Send_SRD_Hex
    (DriverHandle, SlaveAddress, 62.61,
                         "B0 08 09 0B 00 0F 00 00 00 00 00", high);
    SCP_receive(DriverHandle, 3000, & length,sizeof(rb), (char *)              & rb);

    / * Configuration frame sent * /
    Send_SRD_Hex(DriverHandle, SlaveAddress, 62.62, "20 12", high);SCP_receive
    (DriverHandle, 3000, & length,sizeof(rb), (char *) & rb);

    / * One iteration of data exchange * /Buffer [0] = out; / * output value * /Send_SRD_Bin
    (DriverHandle, SlaveAddress, DEFAULT_SAP, DEFAULT_SAP,

                         Buffer, 1, high);
    SCP_receive(DriverHandle, 3000, & length,sizeof(rb), (char *) Offset =     & rb);
    rb.user_data_2 [0];
    in1   =  rb.user_data_2 [Offset];          / *  1st byte   entry    * /
    in2   =  rb.user_data_2 [Offset +1];       / *  2nd byte   entry    * /
    in3   =  rb.user_data_2 [Offset +2];       / *  3rd byte   entry    * /

    / * Communication channel closed * /SCP_close
    (DriverHandle);
}
```

A more comprehensive example

When solving more complex communication problems, it is usually not possible to use a linear program structure and usually we do not avoid the use of multiple threads. The basis is a thread that processes all the responses that the FDL layer sends back to the application. This thread works in an infinite loop, which first waits for a response (timeout = SCP FOREVER) and then, according to the interviewer's identifier, each response is processed separately. Eventually, the thread goes back in wait for a response.

The response thread

```
void ReceiverThread ()
{
    unsigned short          length;
    fdl_rb                  rb;

    while (1)
    {
        / * Waiting for the result without time -out * /
        SCP_receive (       DriverHandle, SCP_FOREVER, & length,sizeof(rb),
                            (char *) & rb);

        / * Processed received answers according to the interviewer's identifier * /switch (
                        rb.rb2header.user)
        {
            case   RequestorID_1:
                . . .
            case   RequestorID_2:
                . . .
            case   . . . :
                . . .
        }
    }
}
```

Requests can then be sent from several threads simultaneously. The reason for receiving responses in only one thread is the ability to distinguish to whom the response is addressed. If each requesting thread had its own response, they could be swapped between threads. The answers are tied to the requirements according to the used DriverHandle and function SCP receive returns the first response with the same DriverHandle. Another solution, then, is to open its own handle for each thread. Then there can be no confusion and each thread only gets its answers.

Attachment B

ProfiMu sample application

To demonstrate the functionality of the created Profibus DP Master, a simple demonstration application was created in the engine room on Charles Square (Figure B.1). Using a computer with ProfiMem installed, a simple model of two conveyors and a manipulator with stepper motors, which is placed between them, is controlled. All inputs and outputs of this model are connected to two remote input and output units (Figure B.2):

- WAGO-I / O-SYSTEM 752–323 16DI / 16DO
- WAGO-I / O-SYSTEM 750–301 8DI / 8DO

These units are connected as slave stations to the Profibus bus, which connects them to the control computer. The task of ProfiM as a Profibus DP Master is to initialize and configure both remote I / O units and then start a data exchange, in which it transmits the output images to the units and at the same time reads back the images of their inputs. In addition, it manages traffic on the Profibus and, if necessary, shares the bus with another master control station. The input and output images are provided by the ProfiM application layer to the application, which thus has conditions similar to those in the PLC and ensures model control.

The ProfiMu application interface can be used by several applications at the same time. This is demonstrated by another Siemens ET-200B 24DI / 8DO remote I / O station, which is connected to the same bus as the two previous stations and controlled from another application. However, no model is connected to it, so the control application only sets the outputs in a certain sequence, which can be seen on the indicator LEDs.

This application is very simple, but the task of this work was not to manage a complex model. It is quite sufficient to demonstrate the communication skills of the created Profibus DP Master.

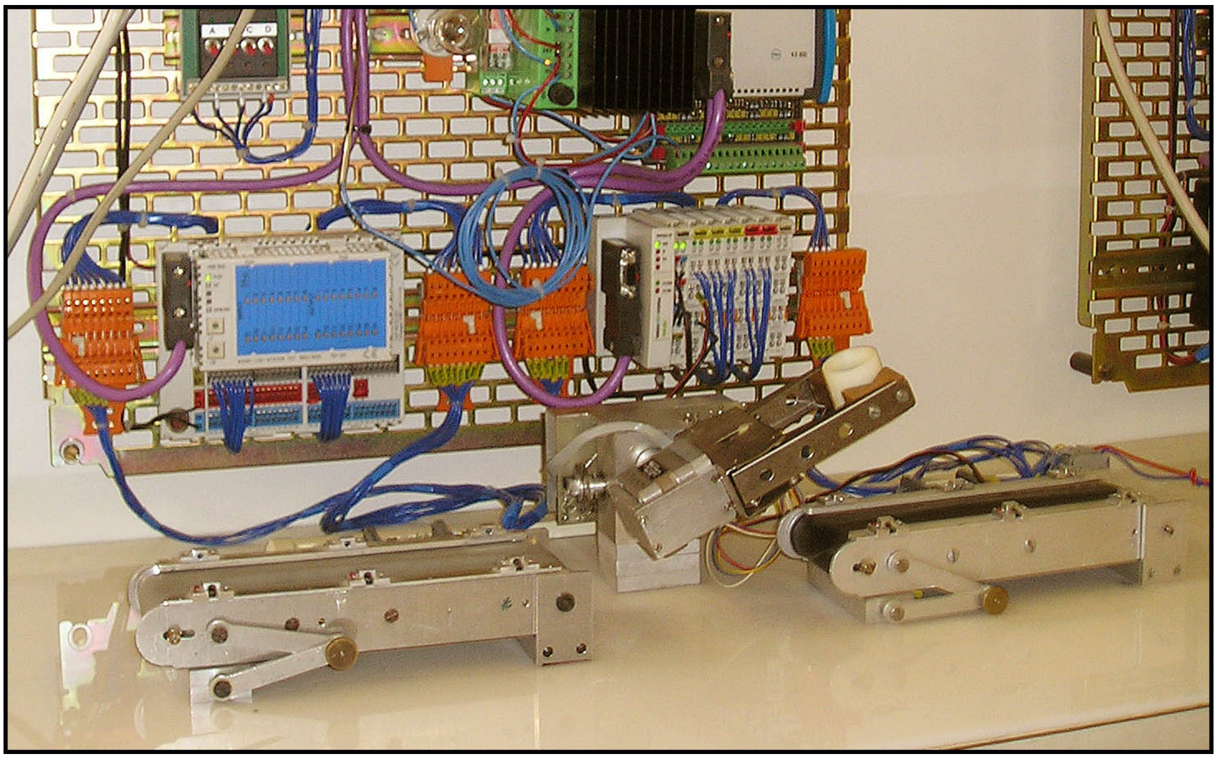


Figure B.1: Model of conveyors with manipulator

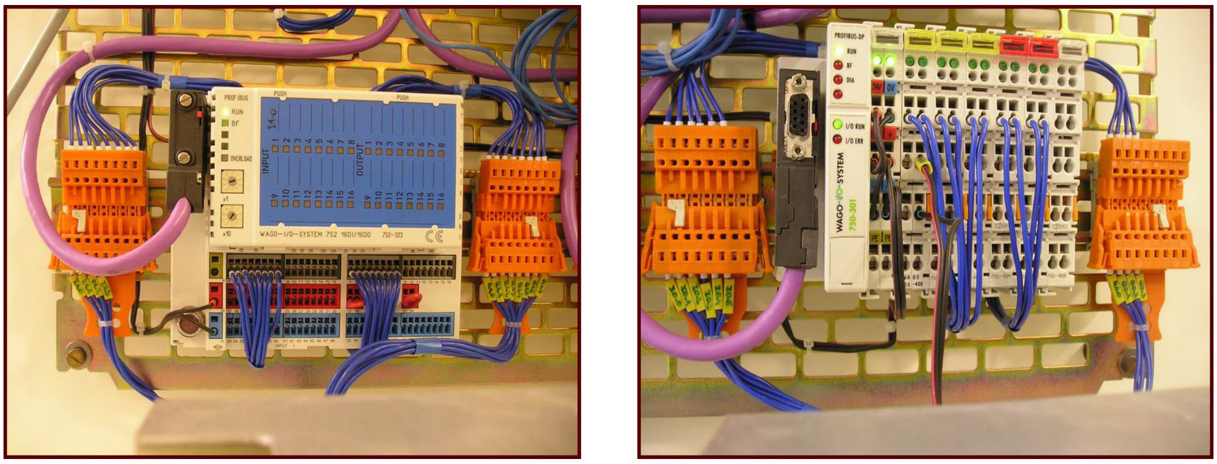


Figure B.2: Remote I / O stations connecting the model to Profibus

Attachment C

List of abbreviations

C.1 Abbreviations for Profibus

CSRD	Cyclic Send and Request Data with reply (FDL Service)
YES	Destination Address of a frame
DAE	Destination Address Extension (s) of a frame conveys DSAP and / or destination Bus ID
DDML	Direc Data Link Mapper
DSAP	Destination Service Access Point a LSAP which identifies the remote FDL User
ED	End Delimiter of a frame
FC	Frame Control (frame type) of a frame
FCS	Frame Check Sequence (checksum) of a frame used to detect corrupted frames
FDL	Fieldbus Data Link layer, OSI layer 2
FMA1 / 2	Fieldbus MAnagement layer 1 and 2
G	GAP update factor the number of token rounds between GAP maintance (update) cycles
GAP	Range of station addresses from This Station (TS) to its successor (NS) in the logical token ring, excluding stations above HSA
GAPL	GAP List containing the status of all stations in this station's GAP Highest
HSA	Stations Address installed (configured) on this Profibus segment List of
LAS	Active Stations
LE	field giving LEngth of frame beyond fixed part field
LEr	that repeats LEngh to increase frame integrity
LSAP	Link Service Access Point identifies one FDL User in a particular station Next
NS	Station (FDL), the station to which this Master will pass the token Open
OSI	System Interconnection
PHY	PHYsical layer, OSI layer 1

PS	Previous Station (FDL), the station which passes the token to this Master station
RSAP	Reply Service Access Point an LSAP at which Request Data may be obtained
SA	Source Address of a frame
SAE	Source Address Extension (s) of a frame conveys SSAP and / or source Bus ID
SAP	Service Access Point, the point of interaction between entities in different protocol layers
SD	Start Delimiter of a frame
SDA	Send Data with Acknowledge FDL Service) Send
SDN	Data with No acknowledge (FDL Service) Send
SRD	and Request Data with reply (FDL Service)
SSAP	Source Service Access Point, an LSAP which identifies the local FDL User which initiates a transaction
SON	SYNchronization bits of a frame (period of IDLE) it guarantees the specified frame integrity and allow for receiver synchronization
t_{BIT}	BIT Time, FDL symbol period of the time to transmit one bit on this Profibus System

C.2 Abbreviations for DDK

DDK	Drivers Development Kit
DIRQL	Device Interrupt ReQuest Level - the IRQL at which a given device interrupts.
DPC	Deferred Procedure Call - a Kernel-defined control object type which represents a procedure that is to be called later. DPC usually finishes time consuming operations initialized by ISR.
GUID	Globally Unique IDentifier
HAL	Hardware Abstraction Layer - a Windows NT / Windows 2000 executive component that provides platform-specific support for the Kernel, I / O Manager, kernel-mode debuggers, and lowest-level device drivers.
ISR	Interrupt Service Routine - a routine whose function is to service a device when it generates an interrupt.
IRP	I / O Request Packet - is the basic I / O Manager structure used to communicate with drivers and to allow drivers to communicate with each other
IRQ	Interrupt ReQuest line - a hardware line over which a peripheral device, bus controller, other processor, or the Kernel signals a request for service to the microprocessor.
IRQL	Interrupt ReQuest Level - the hardware priority level at which a given kernelmode routine runs, thereby masking off interrupts with equivalent and lower IRQL on the processor.
PnP	Plug and Play
WDM	Windows Driver Model
WMI	Windows Management Instrumentation

Attachment D

Contents of the enclosed CD

\src	ProfiMu source codes
\Controller	Driver and installation file source codes (INF file)
\Libraries	Application interface library source codes
\Property Page	Source codes for the Property Page - property settings panel in Device Manager
\Samples used	Simple programs in C ++ Builder showing the use of the ProfiMu application interface
\bin	Resulting files ready to use
\Controller	Compiled driver along with other files ready to install. Divided by operating system into versions for Windows NT 4.0 and Windows 2000 / XP
\Libraries	Static libraries of the ProfiMu application interface
\Documentation	All documentation in electronic form, which was mostly obtained from the Internet about the Profibus, the RS-485 standard, writing drivers, etc.
\HTML	This diploma thesis converted into HMTL format for display on the Internet.
\Latex	The complete "source code" of this document in LANDTEX along with pictures. Good inspiration for those who also want to write their diploma thesis in LANDTEX.

Picture list

List of tables

Literature

[1] RŮŽIČKA, Pavel. Implementation of Profibus FDL layer. Thesis. Prague: Czech Technical University, Faculty of Electrical Engineering, Department of Control Engineering, 2002. 73 p.

[2] SMEJKAL, Radek. Implementation of Profibus DP master. Thesis. Prague: Czech Technical University, Faculty of Electrical Engineering, Department of Control Engineering, 2002. 79 p.

[3] BARTOSIŃKI, Roman. Implementation of USB interface for computer peripherals. Thesis. Prague: Czech Technical University, Faculty of Electrical Engineering, Department of Control Engineering, 2003. 82 p.

[4] ONEY, WALTER. Programming the Microsoft Windows Driver Model. Second Edition. Microsoft Press, 2002. 800 pp. ISBN 07-3561-803-8.

[5] PROFIBUS Specification. Edition 1.0. Karlsruhe: PROFIBUS International, 1997. 924 p. European Standard EN 50 170.

[6] FDL Programming Interface. Release 4. Karlsruhe: Siemens AG, 1995. 126 p.

[7] BURGET, Pavel. Implementation of DP Slave devices. Prague: Czech Technical University, Faculty of Electrical Engineering, Department of Control Engineering, 1999. 5 p.

[8] RS – 232 and RS – 485 Application Note. Octomber 1997 Revision. Ottawa: B&B Electronics Ltd., 1997. 44 p.

[9] OX16PCI954 Data Sheet. Revision 1.3. Oxfordshire: Oxford Semiconductor LTD., 1999. 72 p.
URL: <http://www.oxsemi.com>

[10] PCI-COM communication cards for the PCI bus. Revision 04.2003. Plzen: TEDIA spol. sro, 2003. 28 p.
URL: <http://www.tedia.cz>

[11] VIRIUS, Miroslav. C ++ programming. Prague: Czech Technical University, 2001. 364 pp. ISBN 80-01-01874-1.

[12] Microsoft Developer Network - Device Drivers Development. Microsoft. URL:<http://msdn.microsoft.com>

[13] BOLDIS, Petr. Bibliographic citations of documents according to ČSN ISO 690 and ČSN ISO 690-2 (01 0197): Part 1 Citation: methodology and general rules. Version 3.2. © c 1999-2002, latest updates 3.9. 2002.
URL: <http://www.boldis.cz/citace/citace1.pdf>