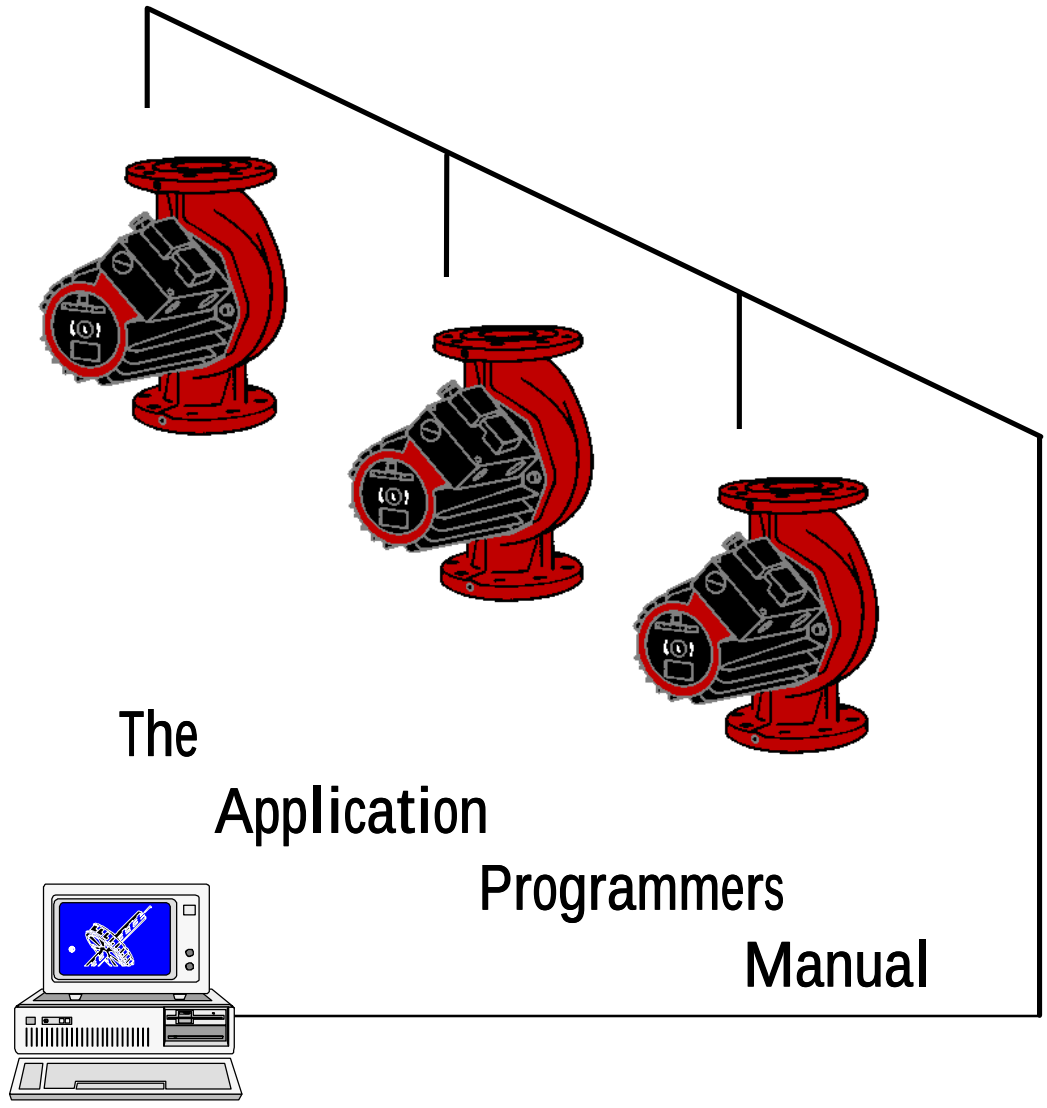
GENIbus

Grundfos Electronics
Network Intercommunications
Bus

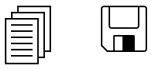


The Application Programmers Manual

Henrik Amdisen
April 1997

3. Edition
Revised and updated
Covers *GENIpro* v2.0

Support



We supply, free of charge:

- **Documentation:**
  - *GENIbus*: The Application Programmers Manual.
  - *GENIbus*: Functional Profiles Supplement.
  - Controlling UPE Pumps via the *GENIbus* Interface.
  - Controlling the 3 phase MGE Motor via the *GENIbus* Interface.
  - Controlling UPS Pumps via the *GENIbus* Interface.
  - Product Information (FB-073): The Grundfos Gateway G100.
- ***GENItools*:** A collection of general and application specific PC based software tools and device simulators. Most of them requires *GENIbus* PC Adapter (RS485/RS232).
  The delivery of some of these tools might be restricted.
- ***GENIpro*:**
  - A Dialogue Driver capable of *GENIpro* Dialogues is available for implementing Windows 16 bit applications (geni_dia.dll) and Windows 32 bit applications (geni_d32.dll) controlling a *GENIbus* system of slaves (like single pump UPE's).
  - A complete *GENIpro Windows 95 Master Device Driver* (geni_man.dll) with Master Management is available for implementing Windows 95 applications controlling all *Genibus* systems, multi master as well as single master.

For help solving *GENIbus* problems please contact:

GRUNDFOS A/S
Fieldbus Group, Dept. 5055
8850 Bjerringbro
DENMARK

	Henrik Amdisen	**Niels Mogensen**
Phone: (+45) 86681400	Direct: (+45) 86684444.5562	Direct: (+45) 86684444.5465
Fax: (+45) 86680438	E-mail: hamdisen @ grundfos.com	E-mail: nmogensen @ grundfos.com

This manual is mainly intended for developing engineers implementing embedded software for units with a *GENIbus* or *GENIlink* interface or for Automation System Companies making a gateway or integrating *GENIbus* into a Management System.

Part I of the manual can be read with benefit by any engineer with interest in the subject. Pay special attention to Chapter 1.3: *Interfacing To GENIbus.*

Part II of the manual and App. A: *The API Reference* is only useful for those software developers implementing a *GENIbus* interface by using the official *GENIpro* 2.x source code release. This source code is for internal Grundfos use only.

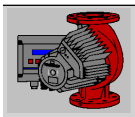
App. C: *GENIpro Telegram Specification* and App. E: *Technical Specifications Summery* can be used as a complete *GENIpro* short form reference.

The manual can be freely distributed.

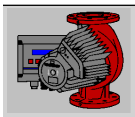
CONTENTS

– PART I. The *GENIbus* System –

– PART II. Using *GENIpro* –



– PART III. Appendices –



GRUNDFOS

GENIbus Protocol Specification

May 2000

Contents		
1.	**Introduction**	
2.	**Technical Data Summery**	
3.	**Telegram Specification**	
	3.1 Telegram Format	
	3.2 APDU Specification	
	3.3 CRC Generation	
	3.4 Connection Request Mechanism	
	3.5 Telegram Examples	
4.	**Scaling of Values**	
5.	**References and GENItools**	

P R E L I M I N A R Y E D I T I O N

1. Introduction

GENIbus, the **G**rundfos **E**lectronics **N**etwork **I**ntercommunications **bus** is a fieldbus developed by Grundfos with the purpose of fulfilling the need for data transfer in all typical Grundfos motor/pump applications. In the field of *Building Management*, control of *Water Purifying Plants*, *Water Works* and *Industry applications* etc. Grundfos devices with GENIbus can be wired together in networks and integrated in automation systems. The major employment's are:

1) Set point control
2) Close loop control of slow systems (sampling rate < 10Hz)
3) Monitoring and data logging
4) Configuration
5) Faultfinding

GENIbus is based on the RS485 hardware standard and operates at a baud rate of 9600 bit/s. This relatively slow communication speed makes it possible to communicate up to 1200 m without the use of termination resistors. On the other side, the slow speed makes GENIbus unsuitable for applications that requires fast control loops e.g. servo applications.

The GENIbus protocol is based on master/slave communication and can handle multi-master networks (not described in this document) if needed. However, a standard GENIbus device from Grundfos, like an E-pump or a CU-control unit, acts as a slave. It will not interfere with the bus control and it will only send a reply when it receives a request from a master device. This means that a GENIbus network will normally have only one master which could be the central management system (SCADA), a local controller like a PLC or a gateway to another type of network. A total of 32 devices can be connected.

Like most other fieldbusses, GENIbus supports the mechanisms for single-casting (single-addressing), multicasting (group addressing) and broadcasting (global addressing). A unique feature for GENIbus is the *Connection Request*, which makes it possible to recognize all connected units on a network without having to poll through all possible addresses.

A summary of GENIbus technical data is given in chapter 2. This gives an overview of the functionality, the performance and the limitations.

Chapter 3 is a detailed specification of the GENIbus telegram format. It describes how data is organized and how operations on data take place. The mechanism in cyclic redundancy checking (CRC) is explained and a straight forward guiding in the CRC implementation is given. The chapter ends with some illustrative telegram examples.

Chapter 4 explains how GENIbus values which represent physical units are scaled and shows some examples.

Finally chapter 5 gives an overview of available GENIbus tools and functional profiles. A Dynamic Link Library and an OPC server has been developed to ease the development of GENIbus applications under Windows. A GENIbus functional profile document is associated with each GENIbus device. The functional profile specifies all data that can be exchanged and how to use it. It is indispensable when making software to operate GENIbus devices.

2. Technical Data Summary

Physical Layer (hardware)	
Topology	Bus
Transmitter	EIA RS485, half duplex
Coding	NRZ (none return to zero)
Data format	Start bit (=0), 8 data bits with least significant bit first, stop bit (=1)
Baud rate	9600 bits/s
Distance	Daisy chain: 1200m Multidrop: 500m Twisted pair cable with shield is recommended
No. of bus units	Max. 32
Data Link Layer (timing, verification)	
Inter byte delay	<=1.2ms
Inter telegram delay	>=3ms
Reply delay	[3ms; 50ms]
Cyclic redundancy checking	16 bit CCITT, polynomial is 0x1021, Start Delimiter excluded. Initialised to 0xFFFF, CRC value bit inverted after calculation. High order byte transmitted first.
Medium access	Master/Slave
Physical address range	Master address range: [0; 231] Slave address range: [32; 231] *) Connection request address: 254 Broadcast address: 255
Presentation Layer (data processing)	
Data orientation	byte

*) The physical address range [32; 95] corresponds to the infra red remote controller R100 number range [1; 64]

Table 1: *Short form technical specification for GENIbus.*

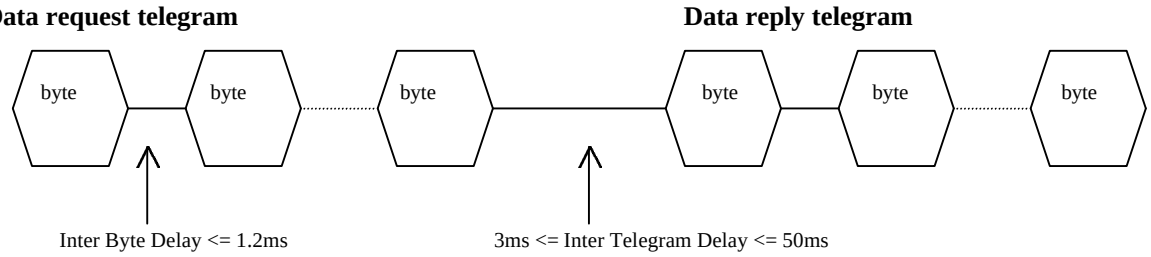


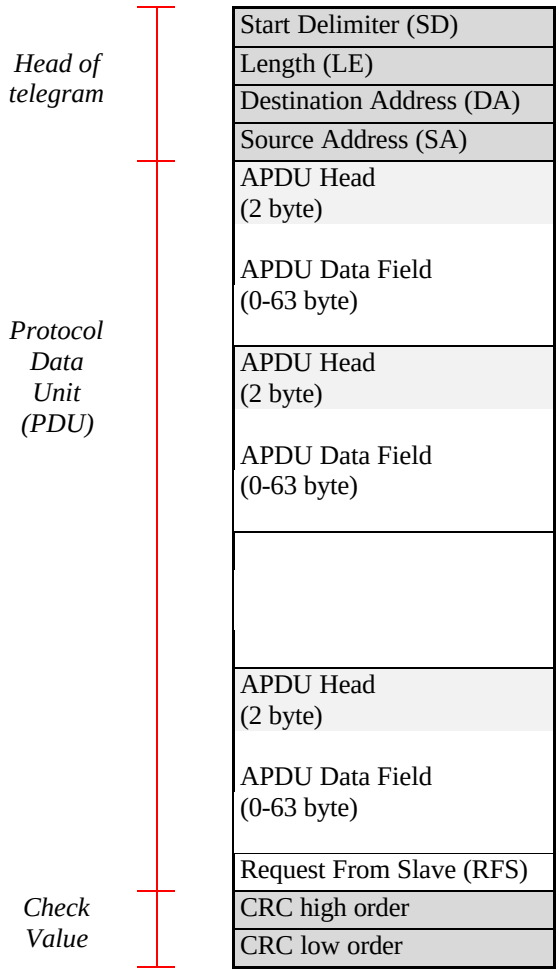
Figure 1: *Illustration of the timing requirements for a Communication Session. These requirements implies the following:*

- *The bytes in a telegram must be send consecutively with an Inter Byte Delay less than 1.2 ms*
- *The Data Reply from a GENIbus unit will always be delayed at least 3ms and maximum 50 ms. Using a Reply Timeout of approximately 60ms in a master is suitable.*
- *A master must leave the bus idle for at least 3 ms after the reception of a reply telegram before the next request is transmitted. This triggers the idle detection circuit in all GENIbus units.*

When using a Data Message telegram (which is not very common) the bus must be left idle by the master for a time period corresponding to the maximum Reply Delay (=50ms) before the next Communication Session must be initiated.

3. Telegram Specification

3.1 Telegram Format



SD=27H: Data Request
SD=26H: Data Message (no reply)
SD=24H: Data Reply

Length: Number of following bytes excluding the Check Value

Destination Address: Address of the receiving unit (destination unit)

Source Address: Address of the transmitting unit (source unit)

PDU: Protocol Data Unit, the data field of the telegram

APDU: Application Program Data Unit. This reflects the organization of application data in Classes and the data access in Operations.

RFS: Request From Slave. This byte is optional. It is used for transfer of bus control in multi-master networks

CRC: A 16 bit Cyclic Redundancy Check Value for validation of a correct data transfer.

***Figure 2**: Format of GENIbus telegram. Each horizontal field is a byte unless otherwise stated. A PDU can consist of zero, one or many APDU's*

3.2 APDU Specification

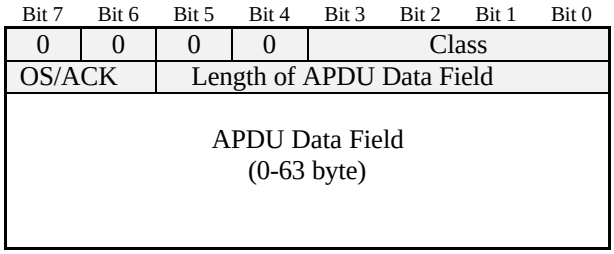


***Figure 3**: Format of Application Program Data Unit*

Class: Specifies which Data Class the APDU belongs to. The APDU Data Field will be interpreted accordingly. Table 2 shows a survey of the Data Classes together with the possible Operations that can be performed on them.

OS/ACK: OS: *Operation Specifier* (for Data Request and Data Message)
- 00: GET, to read the value of Data Items
- 10: SET, to write the value of Data Items
- 11: INFO, to read the scaling information of Data Items, an Info Data structure (fig. 4) will be returned.

ACK: *Acknowledge Code* (for Data Reply)
- 00: Everything is OK
- 01: Data Class unknown, reply APDU data field will be empty
- 10: Data Item ID unknown, reply APDU data field contains first unknown ID
- 11: Operation illegal or Data Class write buffer is full, APDU data field will be empty

	Operation		
Data Class	**GET**	**SET**	**INFO**
1. -			
2. Measured Data	X		X
3. Commands		X	X
4. Configuration Parameters	X	X	X
5. Reference Values	X	X	X
6. -			
7. ASCII-strings	X		

Table 2: *The Data Classes and the possible Operations*

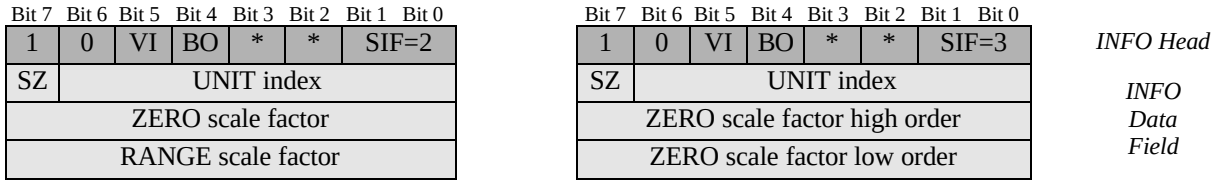


Figure 4: *Info Data structure, the reply to the INFO Operation. Requesting INFO from data items which are not scaled will only return the INFO Head (1 byte). Scaled data items result in a reply with an INFO Data Field as well. 8/16 bit data items use the left Data Field format (standard). The format to the right is for 24/32 bit data items (extended precision). See details in chapter 4.*

VI: *Value Interpretation:*
- 0: Only values from 0-254 are legal. 255 means "data not available"
- 1: All values 0-255 are legal values

BO: *Byte Order:*
- 0: High order byte, this is default for all values that are only 8 bit
- 1: Low order byte to a 16 bit, 24 bit or 32 bit value

SIF: *Scale Information Format:*
- 00: Scale information not available (no UNIT, ZERO or RANGE in reply)
- 01: Bit wise interpreted value (no UNIT, ZERO or RANGE in reply)
- 10: Scaled 8/16 bit value (UNIT, ZERO and RANGE in reply)
- 11: Extended precision, scaled 24/32 bit value (UNIT and ZERO hi/lo in reply)

SZ: *Sign of ZERO:*
- 0: Positive
- 1: Negative

UNIT index: A 7 bit index to the GENIbus Unit Table (Chapter 4)

ZERO/RANGE scale factors: For conversion between the physical representation and the computer representation of a value. See details in chapter 4.

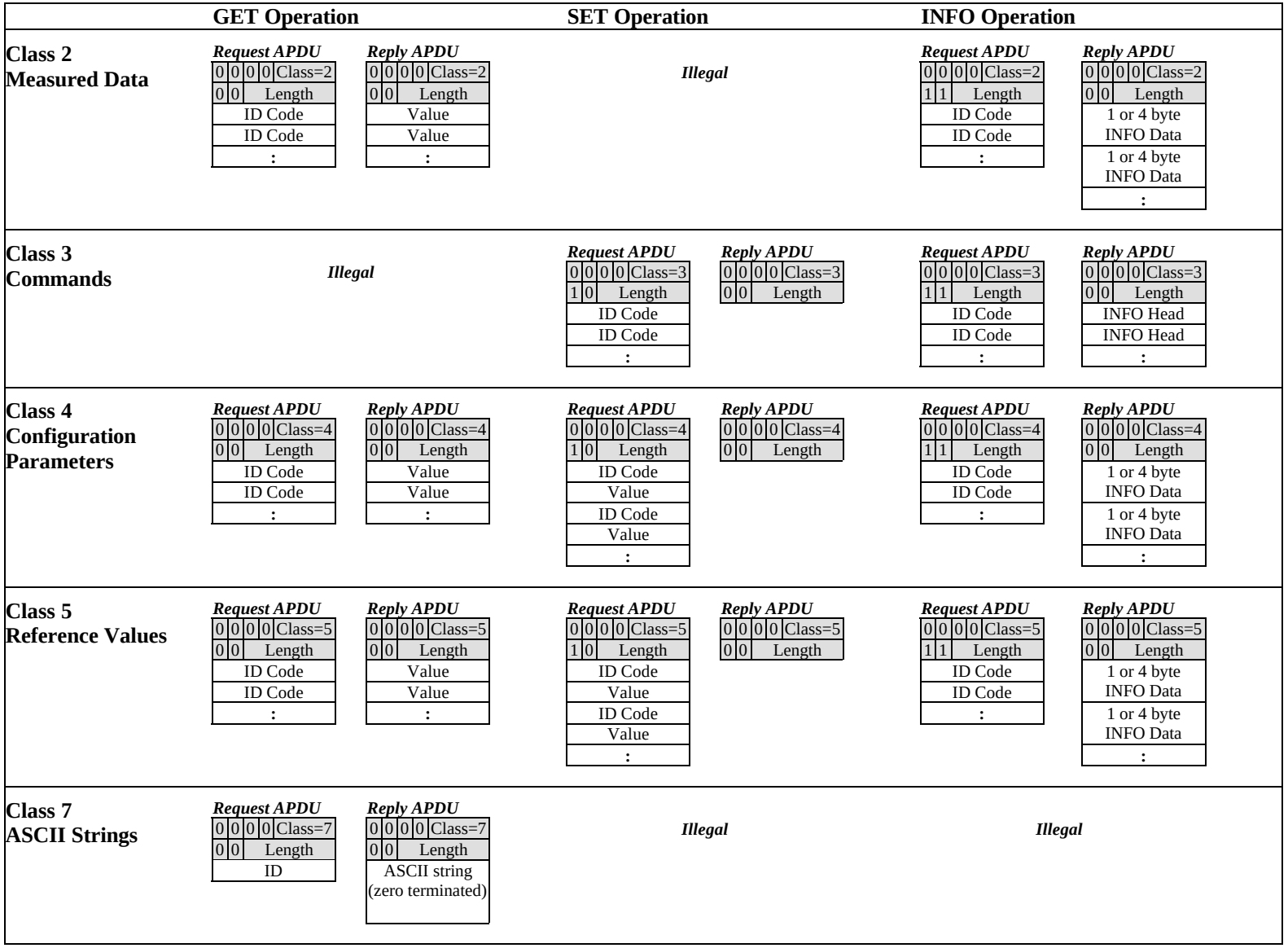


***Figure 5**: Survey of possible APDU's for the different Data Classes. Complete telegram examples can be found in chapter 3.5*

3.3 CRC Generation

The figure below shows a hardware equivalent of the CRC generation. It can be a great help in trying to understand the mechanism. The dark shaded 16 bit register is called the *CRC-Accumulator*. When the whole telegram has been shifted into the machine the Accumulator will hold the CCITT version of the CRC-value. CCITT specifies an initialization of the CRC-Accumulator with all zeros. The Accumulator must be initialized with all 1's and the bits inverted just before transmitting to make the CRC resistant to leading erroneous zeros and to merged telegrams. A function that implements this behavior in software is written below:

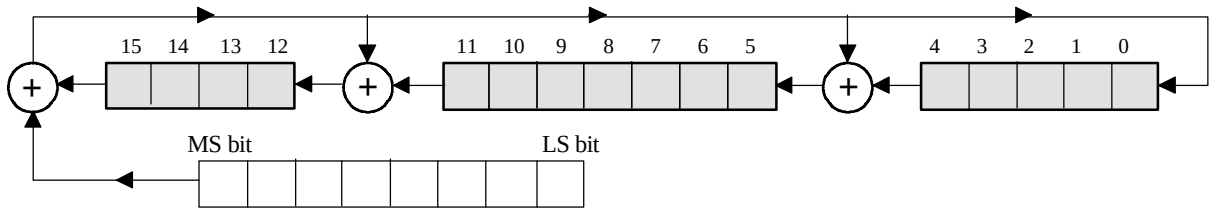


***Figure 6**: A hardware equivalent CRC generator*

```
ushort crchware(ushort data, ushort accum)
 {
  uchar i;
  data <<=8;
  for (i=8; i>0; i--)              /* Do for each bit:        */
    {
    if ((data ^ accum) & 0x8000)   /* IF a 1 in feedback path */
      accum = (accum <<1)^genpoly; /*    feedback interaction */
    else                           /* ELSE                    */
      accum <<= 1;                 /*    transparent shift    */
    data <<=1;                     /* Make next bit ready     */
    }
 return accum;
 }
```

Each byte in the telegram which is to take part in the cyclic redundancy check is passed to the function one by one along with the value of the Accumulator. When the last byte of the telegram has been passed the return value will be the CRC-value. Data has been declared as a 16 bit value due to its presence in a 16 bit expression.

By studying the CRC circuit we can see, that when applying a new byte to the CRC circuit the feedback path will not be influenced by the existing low order byte of the Accumulator. Only the high order Accumulator byte interacts with the data bits. We refer to the result of this XOR'ing as the *combining value*. This leads to the observation that the new Accumulator is equal to the CRC of the combining value XOR'ed with the unchanged half of the Accumulator. This relationship can be expressed in C.

```
comb_val = (accum >>8) ^ data;
tmp = crchware(comb_val,0);
accum = tmp ^ (accum <<8);
```

Since there are only 256 possible combining values, it would be a good idea to calculate their CRC's in advance and store them in a table, `crctab[256]`, thereby saving a great deal of run time computer power. The following piece of code uses `crchware` to generate the lookup table.

```
void main(void)
   {
   unsigned short i,j;
   printf("\nunsigned short crctab[256] = \n    {");
   for (j=0; j<=31; j++)
        {
        printf("\n    ");
        for (i=8*j; i<8*j+8; i++) printf("%6u,  ", crchware(i,0));
        }
   printf("\n    }");
```

By redirecting the output to a file you have the table ready to paste into the protocol source code. The CRC calculation now takes this form.

```
comb_val = (accum >>8) ^ data;
tmp = crctab[comb_val];
accum = tmp ^ (accum <<8);
```

Combining this into a more compact form leads to the final CRC function. The Accumulator has been removed from the arguments and made a global variable, to avoid the overhead of passing it to the function for each byte to process.

Final GENIbus CRC algorithm

```
ushort accum;
void crc_update(uchar data)
      {
      accum = (accum <<8)^ crctab[(accum >>8) ^ data];
      }
```

When a telegram is transmitted the CRC-Accumulator is initialized to 'all ones' and each byte, except the *Start Delimiter*, is processed through the `crc_update` function before being send to the Drivers. Finally the CRC-Accumulator is inverted and its two bytes are appended to the telegram with high order byte first. These two bytes are what we define as the *CRC-Value*.

The receiver performs a similar procedure. Initializes the CRC-Accumulator to 'all ones'. Then, each byte received, except the *Start Delimiter*, is processed through the `crc_update` function. When the CRC-Value bytes arrive they are inverted and then also processed through `crc_update`. If the CRC-Accumulator hereafter is equal to zero the received telegram is considered as sound.

3.4 Connection Request Mechanism

The GENIbus protocol offers an effective mechanism for a master device to recognize all units connected to the bus. The master can use a Connection Request Telegram. This telegram is characterized by the usage of the destination address DA=254 (0xFE). The Connection Reply which results is characterized by

- only generated if the unit has not been requested (polled) by using its unit address within the last 20 seconds
- random reply delay [3ms; 43ms] to minimize the probability of more units replying simultaneously

When a network is powered on, and the master has no previous knowledge about which units are connected, it can use Connection Requests (instead of polling through all possible addresses) to recognize the units. When all units has been recognized (no more replies to connection requests), the master can use Connection Requests occasionally as a simple means to detect if new units are connected (or units that has been disconnected or switched off are being reconnected or switched on).

Connection Request

Start Delimiter	0x27
Length	0x0E
Destination Address	0xFE
Source Address	0x01
Class 0: Protocol Data	0x00
OS=0 (GET), Length=2	0x02
`df_buf_len` = ID 2	0x02
`unit_bus_mode` = ID 3	0x03
Class 4: Configuration Parameters	0x04
OS=0 (GET), Length=2	0x02
`unit_addr` = ID 46	0x2E
`group_addr` = ID 47	0x2F
Class 2: Measured Data	0x02
OS=0 (GET), Length=2	0x02
`unit_family` = ID 148	0x94
`unit_type` = ID 149	0x95
CRC high	0xA2
CRC low	0xAA

Connection Reply

Start Delimiter	0x24
Length	0x0E
Destination Address	0x01
Source Address	0x20
Class 0: Protocol Data	0x00
OS=0 (GET), Length=2	0x02
Value example of `df_buf_len`	0x46
Value example of `unit_bus_mode`	0x0E
Class 4: Configuration Parameters	0x04
Ack=0, Length=2	0x02
Value example of `unit_addr`	0x20
Value example of `group_addr`	0xF7
Class 2: Measured Data	0x02
Ack=0, Length=2	0x02
Value example of `unit_family`	0x03
Value example of `unit_type`	0x01
CRC high	0x00
CRC low	0x04

Figure 7: *Connection Request/Reply example. In this example the master has a Unit Address of 1 and the slave is a CU3 unit with unit address 32 (0x20), corresponding to No. 1 given with the infra red remote controller R100.*
The Class 0 Data Items need not to be considered.

3.5 Telegram Examples

Both examples assume communication with a Control Unit CU3 with unit address 0x20 (No. 1 given with R100). Compare the examples with the general telegram format in figure 2 and the APDU survey in figure 5.

Data Request

Start Delimiter	0x27
Length	0x07
Destination Address	0x20
Source Address	0x01
Class 2: Measured Data	0x02
OS=3 (INFO), Length=3	0xC3
`i_rst` = ID 2	0x02
`t_mo` = ID 16	0x10
`p_hi` = ID 26	0x1A
CRC high	0x90
CRC low	0x1C

Data Reply

Start Delimiter	0x24
Length	0x10
Destination Address	0x01
Source Address	0x20
Class 2: Measured Data	0x02
Ack=0, Length=12	0x0C
Value example of `i_rst` INFO head	0x82
Value example of `i_rst` UNIT Index	0x3E
Value example of `i_rst` ZERO	0x00
Value example of `i_rst` RANGE	0x39
Value example of `t_mo` INFO head	0x82
Value example of `t_mo` UNIT Index	0x15
Value example of `t_mo` ZERO	0x00
Value example of `t_mo` RANGE	0x64
Value example of `p_hi` INFO head	0x82
Value example of `p_hi` UNIT Index	0x09
Value example of `p_hi` ZERO	0x00
Value example of `p_hi` RANGE	0xFA
CRC high	0x91
CRC low	0x0A

Figure 8: *Request of scaling information of 3 data items by using the INFO operation. Notice that data items with scaling information returns INFO head, UNIT index, ZERO and RANGE. The values in the reply are examples, your reply might be different.*

Data Request

Start Delimiter	0x27
Length	0x0F
Destination Address	0x20
Source Address	0x01
Class 2: Measured Data	0x02
OS=0 (GET), Length=4	0x04
`i_rst` = ID 2	0x02
`t_mo` = ID 16	0x10
`p_hi` = ID 26	0x1A
`p_lo` = ID 27	0x1B
Class 4: Configuration Parameters	0x04
OS=0 (GET), Length=2	0x02
`t_mo_stop` = ID 4	0x04
`i_rst_max_stop` = ID 5	0x05
Class 3: Commands	0x03
OS=2 (SET), Length=1	0x81
`START` = ID 6	0x06
CRC high	0x80
CRC low	0x2A

Data Reply

Start Delimiter	0x24
Length	0x0E
Destination Address	0x01
Source Address	0x20
Class 2: Measured Data	0x02
Ack=0, Length=4	0x04
Value example of `i_rst`	0x7A
Value example of `t_mo`	0x42
Value example of `p_hi`	0x39
Value example of `p_lo`	0x80
Class 4: Configuration Parameters	0x04
Ack=0, Length=2	0x02
Value example of `t_mo_stop`	0xB5
Value example of `I_rst_max_stop`	0xC8
Class 3: Commands	0x03
Ack=0, Length=0	0x00
CRC high	0xF2
CRC low	0xD7

Figure 9: *The example shows how a reading of 4 data items from class 2 (`p_hi` and `p_lo` combines to a 16 bit value), and 2 data items from class 4 and writing of a command (class 3) can be done in one telegram. The values in the reply are examples, your reply might be different.*

4. Scaling of values

The purpose of scaling is to map the value of a variable *x*, representing some physical entity, into its computer representation *X*. The *GENIbus* value representation is based on 8 bit quantities. This means, that scaling maps an interval of real numbers with any chosen length and from any chosen decade linearly into an 8 bit integer representation:

$$x \in [a;b] \quad \rightarrow \quad X \in [0;254] \qquad ;(a,b) \in \Re$$

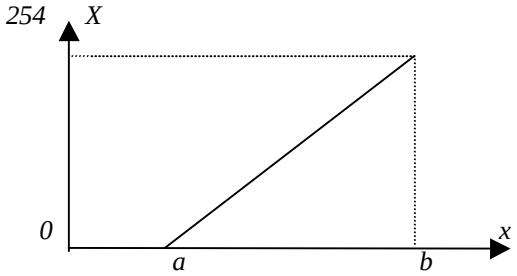


Figure 10: *Mapping real numbers into a byte value.*

This mapping is shown graphically in the figure above. The value *255* is reserved for indication of "data not available". What is needed now is a mathematical expression for the mapping and its inverse. The use of the symbols in the figure and the straight line relationship give us this equation to start with:

$$(1) \qquad X(x) = -254 \cdot \frac{a}{b-a} + 254 \cdot \frac{1}{b-a} x = \frac{254}{b-a}(-a+x)$$

a is the *"zero"* for the interval to be mapped, and *(b-a)* is its *"range"*. Both occur directly in the equation. Contained in this *"zero"* and *"range"*, however is the physical unit with some prefix factor to take the decade into consideration. To isolate this, a multiplier called *UNIT* is introduced in the following way:

$$\begin{array}{lll} b\text{-}a & = \text{"range"} & = RANGE \times UNIT \\ a & = \text{"zero"} & = ZERO \times UNIT \end{array}$$

Substituting this in (1) gives the final *conversion formulas*:

$$(2a) \qquad X = \frac{254}{RANGE \cdot UNIT}(-ZERO \cdot UNIT + x)$$

$$(2b) \qquad x = \left(ZERO + X \frac{RANGE}{254} \right) \cdot UNIT$$

ZERO and *RANGE* can be represented in 8 bit, consequently they are suitable for *GENIbus*. *UNIT* is an index to a standard table of defined physical units with a prefix factor. This table, from now on called *The Unit Table*, can cover a substantially amount of different scaling's although it is small.
UNIT is chosen to be in 8 bit (not surprisingly). Bit 7 is reserved for one problem, that was left over: the sign of *ZERO*. Left are 7 bits giving 128 possible entries in The Unit Table. The job of converting a scaled *GENIbus* data item value into its real number representation with physical units, now means to read (with INFO Operation) *ZERO, RANGE* and *UNIT* for that data item, and then process the value through formula (2b).

The scaling formulas (2a-b) are only valid for 8 bit values, but it is only natural to extend them to count for 16 bit values as well. Understanding X_{16} as a 16 bit computer representation leads directly to the 16 bit version of the conversion formulas:

$$(3a) \qquad X_{16} = \frac{254 \cdot 256}{RANGE \cdot UNIT}(-ZERO \cdot UNIT + x)$$

$$(3b) \qquad x = \left(ZERO + X_{16} \frac{RANGE}{254 \cdot 256} \right) \cdot UNIT = \left(ZERO + X_{hi} \frac{RANGE}{254} + X_{lo} \frac{RANGE}{254 \cdot 256} \right) \cdot UNIT$$

Because *GENIpro* handles single byte *Data Items*, X_{hi} and X_{lo} must have one ID code each. Per definition, the scaling information *ZERO*, *RANGE* and *UNIT* is connected with the high order byte, H_i. The low order byte X_L has no scaling information because this comes implicit from (3a-b). Using the *INFO* operation on a low order byte will however return an *INFO head*, where the *BO* bit is set to indicate that this is a low order byte (See figure 4). The functional profile for the device in question specifies which data items are split in a high/low pair.

The Unit Table								
Index	**Physical entity**	**Unit with prefix**	**Index**	**Physical entity**	**Unit with prefix**	**Index**	**Physical entity**	**Unit with prefix**
1	Electrical current	0.1 A	**20**	Temperature	0.1 °C	**30**	Percentage	1 %
42	Electrical current	0.2 A	**21**	Temperature	1 °C	**31**	Energy	1 kWh
62	Electrical current	0.5 A	**57**	Temperature	1 °F	**32**	Energy	10 kWh
2	Electrical current	5 A	**22**	Flow	0.1 m^3/h	**33**	Energy	100 kWh
3	Voltage	0.1 V	**23**	Flow	1 m^3/h	**40**	Energy	512 kWh
4	Voltage	1 V	**69**	Flow	0.1 ml/h	**46**	Energy	1 MWh
5	Voltage	5 V	**41**	Flow	5 m^3/h	**47**	Energy	10 MWh
6	Elec. resistance	1 Ω	**73**	Flow	0.5 l/h	**48**	Energy	100 MWh
43	Elec. resistance	10 kΩ	**52**	Flow	1 l/s	**34**	Ang. velocity	R rad/s
7	Power (active)	1 W	**63**	Flow	0.1 l/s	**39**	Time	1024 h
8	Power (active)	10 W	**53**	Flow	1 m^3/s	**72**	Time	1024 min
9	Power (active)	100 W	**54**	Flow	1 gpm	**35**	Time	1 h
44	Power (active)	1 kW	**58**	Flow	10 gpm	**36**	Time	2 min
45	Power (active)	10 kW	**24**	Head	0.1 m	**37**	Time	1 s
10	Power (apparent)	1 VA	**25**	Head	1 m	**49**	Ang. degrees	1 °
11	Power (apparent)	10 VA	**26**	Head	10 m	**50**	Gain	1
12	Power (apparent)	100 VA	**56**	Head	1 ft	**71**	Volume	1 nl
13	Power (reactive)	1 VAr	**59**	Head	10 ft	**70**	Volume	0.1 ml
14	Power (reactive)	10 Var	**51**	Pressure	0.001 bar	**64**	Volume	0.1 m^3
15	Power (reactive)	100 VAr	**27**	Pressure	0.01 bar	**67**	Volume	256 m^3
16	Frequency	1 Hz	**28**	Pressure	0.1 bar	**65**	Volume	1000 m^3
38	Frequency	2 Hz	**29**	Pressure	1 bar	**66**	Energy pr vol.	10 kWh/m^3
17	Frequency	2.5 Hz	**61**	Pressure	1 kPa	**74**	Energy pr vol.	1 Wh/m^3
18	Rot. velocity	12 rpm	**55**	Pressure	1 psi	**68**	Area	1 m^2
19	Rot. velocity	100 rpm	**60**	Pressure	10 psi			

Table 3: *The Unit Table*

Scaling example for 8 bit data item

Assume that a GET request for the data item `t_m` (Motor temperature, Class 2, ID 29) in a UPE pump (or E-pump) returns the value 163 and an INFO request for the same data item returns UNIT=21, RANGE=90, ZERO=10, then

$$T_m = \left(ZERO + t\_m \frac{RANGE}{254} \right) \cdot UNIT = \left(10 + 163 \cdot \frac{90}{254} \right) \cdot 1^oC = 68^oC$$

Scaling example for 16 bit data item

Assume that a GET request for the data items `p_hi` and `p_lo` which constitute a 16 bit high/low data item pair (Power consumption, Class 2, ID 26/27) in a CU3 control unit returns the values 16/214 and an INFO request for `p_hi` returns UNIT=44, RANGE=120, ZERO=0, then

$$P = \left(ZERO + p\_hi \frac{RANGE}{254} + p\_lo \frac{RANGE}{254 \cdot 256} \right) \cdot UNIT = \left(0 + 16 \cdot \frac{120}{254} + 214 \cdot \frac{120}{254 \cdot 256} \right) \cdot 1kW = 7.95kW$$

It can generally be assumed that no GENIbus device changes the scaling of the data items dynamically. This mean, that once ZERO, RANGE and UNIT is know for all the data items to use (by using INFO requests when starting) it is from now on only necessary to request the data item values and process the reply through the scaling formula with the value of the scaling parameters inserted.

Extended precision

To be able to deal with data that spans a range of several decades and which at the same time must preserve the same resolution in the high decade as in the low decade, the GENIbus protocol supports a data format with extended precision using 16 bit, 24 bit or 32 bit (see also figure 4 in chapter 3.2). This format differs from the standard scaling format in two ways: it does not include a RANGE specifier, and the ZERO specifier is in 16 bit. The conversion formulas for 16 bit, 24 bit and 32 bit respectively are shown below:

(4.a) $$X_{16} = \frac{x}{UNIT} - ZERO_{16}$$

(4.b) $$x = (ZERO_{16} + X_{16}) \cdot UNIT$$

(4.a) $$X_{24} = \frac{x}{UNIT} - 256 \cdot ZERO_{16}$$

(4.b) $$x = (256 \cdot ZERO_{16} + X_{24}) \cdot UNIT$$

(5.a) $$X_{32} = \frac{x}{UNIT} - 256^2 \cdot ZERO_{16}$$

(5.b) $$x = (256^2 \cdot ZERO_{16} + X_{32}) \cdot UNIT$$

Note that X_{16} consist of 2 data items: $X_{16} = X_{hi} \cdot 256 + X_{lo}$

that X_{24} consist of 3 data items: $X_{24} = X_{hi} \cdot 256^2 + X_{lo1} \cdot 256 + X_{lo2}$

that X_{32} consist of 4 data items: $X_{32} = X_{hi} \cdot 256^3 + X_{lo1} \cdot 256^2 + X_{lo2} \cdot 256 + X_{lo3}$

and that $ZERO_{16} = 256 \cdot ZERO_{hi} + ZERO_{lo}$

From the INFO head it can be seen if a data item is scaled according to the extended precision specification or not. However it will also be noticed explicitly in the function profile of the product in question.

Extended precision scaling example for 16 bit data item

Extended precision with 16 bit is typically used for data items which need a higher precision than 8 bit and where the increments (1 bit) should have a nice value (e.g. 1W instead of 1.18W). Assume that a GET request for the data items `water_level_hi` and `water_level_lo` which constitute a 16 bit data item pair (Class 2, ID 201/202) in an APE sewage pump, returns the values 18, 12, and an INFO request for `water_level_hi` returns UNIT=179 (negative ZERO and UNIT index = 51), $ZERO_{hi}$=3, $ZERO_{lo}$=245, then

$$L_{water} = (ZERO_{16} + water\_level_{16}) \cdot UNIT$$

Substituting:

$$water\_level_{16} = water\_level\_hi \cdot 256 + water\_level\_lo$$

$$ZERO_{16} = 256 \cdot ZERO_{hi} + ZERO_{lo}$$

$$UNIT\,51 = 1\,mbar$$

results in:

$$L_{water} = (-(256 \cdot 3 + 245) + 18 \cdot 256 + 12) \cdot 1\,mbar = -1013\,mbar + 4620\,mbar = 3607\,mbar$$

Extended precision scaling example for 24 bit data item

Extended precision with 24 bit is typically used for counters which count events or time. Assume that a GET request for the data items `power_on_time_hi, power_on_time_lo1` and `power_on_time_lo2` which constitute a 24 bit data item trio (Class 2, ID 192/193/194) in a UPE pump returns the values 7, 108, 32, and an INFO request for `power_on_time_hi` returns UNIT=36, $ZERO_{hi}$=0, $ZERO_{lo}$=0, then

$$t_{power\_on} = (256^2 \cdot ZERO_{16} + power\_on\_time_{24}) \cdot UNIT$$

Substituting:

$power\_on\_time_{24} = power\_on\_time\_hi \cdot 256^2 + power\_on\_time\_lo1 \cdot 256 + power\_on\_time\_lo2$

$ZERO_{16} = 256 \cdot ZERO_{hi} + ZERO_{lo}$

$UNIT\,36 = 2\,\text{min}$

results in:

$T_{power\_on} = (256^2 \cdot (256 \cdot 0 + 0) + 7 \cdot 256^2 + 108 \cdot 256 + 32) \cdot 2\,\text{min} = 972864\,\text{min} = 675\,d\ \ 14\,h\ \ 24\,\text{min}$

Extended precision scaling example for 32 bit data item

Extended precision with 32 bit is typically used for physical values (measured or calculated). Assume that a GET request for the data items `dosing_flow_hi, dosing_flow_lo1 dosing_flow_lo2` and `dosing_flow_lo3` which constitute a 32 bit data item quartet (Actual dosing flow, Class 2, ID 39/40/41/42) in a DME dosing pump returns the values 23, 216, 42, 214 and an INFO request for `dosing_flow_hi` returns UNIT=69, $ZERO_{hi}$=0, $ZERO_{lo}$=0, then

$Q_{dosing} = \left(256^2 \cdot ZERO_{16} + dosing\_flow_{32}\right) \cdot UNIT$

Substituting:

$dosing\_flow_{32} = dosing\_flow\_hi \cdot 256^3 + dosing\_flow\_lo1 \cdot 256^2 + dosing\_flow\_lo2 \cdot 256 + dosing\_flow\_lo3$

$ZERO_{16} = 256 \cdot ZERO_{hi} + ZERO_{lo}$

$UNIT\,69 = 0.1\,ml/h$

results in:

$Q_{dosing} = (256^2 \cdot (256 \cdot 0 + 0) + 23 \cdot 256^3 + 216 \cdot 256^2 + 42 \cdot 256 + 214) \cdot 0.1 ml/h = 40.0043 \cdot 10^6\ ml/h$

5. References and GENItools

Tools for developing of GENIbus applications	
GENIman.dll	A Dynamic Link Library (DLL file) for accessing GENIbus units from Windows applications. A user manual is included
GENIman.opc	An OPC driver for accessing GENIbus units from Windows applications. A user manual is included
Functional Profiles for GENIbus devices	
upe.pdf	Operating the UPE pump via GENIbus or G100
ups.pdf	Operating the UPS pump via GENIbus or G100
mge.pdf	Operating the 3 phase MGE motor via GENIbus or G100
cu3.pdf	Operating the CU3 control unit via GENIbus or G100
cu300.pdf	Operating the CU300 control unit via GENIbus or G100
dme.pdf	Operating the DME dosing pump via GENIbus or G100

1. Introduction to the *GENI* Concept

1.1 Overview

Grundfos **E**lectronics **N**etwork **I**ntercommunication is a concept for remote communication between distinct units in distributed systems. The *GENI* concept consists of a *fieldbus* called ***GENIbus*** and an *infra red* communications link called ***GENIlink***. Besides, different software tools for service, development and debugging of devices with *GENIbus* interface has been developed and named ***GENItools***. Some of the tools can be used for configuration of units and includes bus control and bus diagnostics facilities.

GENIlink and *GENIbus* employ the same communications protocol ***GENIpro***. Only the low level parts: the hardware and accompanying *Software Drivers* are different. This is illustrated in the figure below.

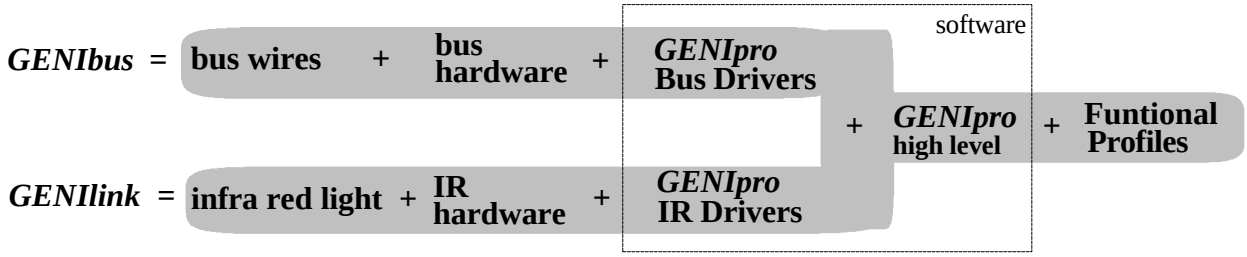


Figure 1.1: *The relationship between GENIbus, GENIlink and GENIpro.*

GENIpro can be understood as a two channel fieldbus protocol (fig 1.2). Priority is given to the IR channel in case the external devices should try to make use of both channels at the same time.

The bus channel as well as the IR channel are what in communication terminology is known as *half duplex*. Communication can take place in both directions, but it must be in one direction at a time.

Figure 1.2:*GENIpro, a dual channel half duplex protocol*

GENIpro works like an intelligent *front end* to an *Application Program* offering the *servic* necessary for communication through a standardized interface. *GENIpro* is the means for the Application Program to send data to other devices and to receive data from them. Data is transferred in a specified package called a *Telegram* or a *Data Frame*.

In addition to being a remedy for *data transfer*, *GENIpro* also has means to *control* a *GENIbus* System where several devices are connected and dependent on the exchange of *Telegrams*. This implies, among other things, automatic *Bus Configuration* with the appointment of a *Bus Master* and automatic registration of devices being connected and disconnected.

GENIpro deals with the problems of organizing data in Telegrams and transmitting them without errors. The protocol itself cannot ensure that any device on the bus knows what to do with data it receives, or that data it transmits actually contains the information it is supposed to. For the devices to be able to understand each other, they must "speak the same language". This necessitates not only a common way for data transfer, but just as important, a common way of interpreting data. This is what the ***Functional Profiles*** are for.

The Functional Profiles define a uniform data handling and interpretation. When they accompany or supplement the protocol, devices on a *GENIbus* System present a standardized information and operational interface to the bus. Details of how specific functions are actually implemented, how data is actually generated or acquired from internal or external sources, what kind of mechanical or electrical construction is used as actuator, etc. - it is all hidden behind the Functional Profiles. In this way, what is known as *interoperability* between devices can be assured. It is important to notice, that Functional Profiles are specifications, and that these normally differ between devices from different application areas. Functional Profiles can be found in the separate manual ***GENIbus*** - Functional Profiles Supplement.

Devices provided with *GENIlink* can be operated by the graphical menu based *R100* terminal. The use of *R100* is mainly for the configuration of devices when a system is installed the first time or if changes in parameters like setpoint or control mode is to be made later. Simple graphics and texts in a display helps the operator to complete the configuration and give a visual feedback to acknowledge the transmission or to inform him, if anything went wrong. The introduction of *GENIlink* makes the conventional configuration via hardware switches, jumpers, etc. obsolete. A device can be operated by *R100* while connected to a working *GENIbus.*

The purpose of *GENIbus* is to fulfil the need for data transfer in all typical GRUNDFOS motor/pump applications in the field of *Building Management* and the control of *Water Purifying Plants* and *Water Works* etc. This can be concretized into four major employments:

1) ***Open and closed loop control***
   - Transmitting of setpoints to embedded controllers for the operation of local control loops (distributed control)
   - Requesting feedback signal from a device and transmission of a control signal to the device using it as an actuator (control loop closed via the bus, central control).

2) ***Monitoring***
   - Acquiring data from the device for supervising its operational conditions and the variables in the plant it controls.
   - Acquiring data from the device for logging.

3) ***Configuration***
   - Factory configuration
   - Installation configuration of device parameters like mode of operation, alarm limits etc. via a *GENIbus* master device like a PC Service Tool, or a *GENIlink* device like R100.

4) ***Test***
   - Factory testing when devices are being manufactured.
   - Servicing and doing service testing in the field.
   - Network diagnosis.

Many of the *GENItools*, which are all PC based, can be connected to a working *GENIbus* System. Some has all master facilities for controlling the bus. Most of them has a menu based user interface and are suited for configuration of devices and performing tests, logging and debugging.

A question that very often comes up is whether to use a company specific bus standard like *GENIbus* in-stead of one of the open standards on the market like *Profibus, LON, EIB* etc. In any case there are advantages and disadvantages. In general the advantage is the freedom to attain a solution which is technically and economically at its optimum. For Grundfos this can be concretized to the following list:

Advantages:
1) Freedom to choose hardware specification and components (can be application optimized).
2) High degree of hardware independence by implementing the whole protocol in software.
3) Freedom to define protocol functionality to exactly match the needs of the applications.
4) Optimize Functional Profiles taking special product features into consideration.
5) The special feature of two channels in one protocol.
6) Easy to extent functionality if required by future demands.

Disadvantages:
1) All development has to be done from the beginning, this also includes protocol tools.
2) Problems are really your own, there is no standard organization to support and help you.
3) The connection to any market standard network requires the development of a special solution (once developed these can however be reused).
4) The general approach to communication system design and implementation is to a certain extend lost, when a protocol is optimized to a specific application.

1.2 Manual Contents

System Integrators or others working with the problem of interfacing to *GENIbus*, should give special attention to chapter 1.3 which follows. Here a short description of the different ways to implement an interface, and what it takes to do it, is listed.

Chapter 1 ends with a description of two *GENIbus* application examples. Better than reading through a lot of technical specifications (which this manual is also filled with), these examples give a good feeling of what kind of extended and improved functionality of pump systems can be achieved, when units can communicate. Chapter 2 is a thorough description of this **functionality** seen from a network operation and communications protocol point of view.

If this description seems complicated, it is important to understand that it should only give the reader a feeling of what happens on the bus and introduce the terminology. By using the official *GENIpro v2.0* software release (C-source code or Windows Device Driver) all handling of abstract concepts like Bus Modes and Bus Control is taken care of "behind the curtains". The only thing which the application programmer must relate to is an Application Program Interface (API) based on function calls - we will return to that in chapters 4 and 6.

GENIpro is implemented as a 3 layer communications protocol which is common practice for fieldbus protocols. The layers corresponds to equivalent layers in the ISO specification for Open Systems Interconnection. This **architecture** of the protocol is described in chapter 3. As mentioned already data and control is transferred between *GENIbus* units in the form of Telegrams. These are physically perceivable as a serial stream of bits on the bus wires. The Telegrams comply to the *GENIpro* ***Telegram Specification*** (Appendix C) also described in chapter 3.

Chapter 4 is about the functionality described in Chapt. 2 but seen from the inside- or software point of view. It must be studied by the developer of an Application Program for a Master Unit Type. This could be a gateway, a Management System or a functional device utilising master facilities. Many of the function calls in the **API** is explained with examples together with the mechanisms in *GENIpro* which they activate, deactivate and control (a survey of all API functions can be seen in appendix A).

The **configuration** of *GENIpro* to suit the application is described in Chapt. 5. Also the interfacing to a target dependent Bus Driver and the service requirements from an operating system is briefly discussed (a detailed documentation can be found in Appendix D.2 and D.4 respectively).

Appendix E is the whole *GENIpro* Technical Specification in a short compact three pages format. This might be helpful as a look up reference. Appendix F is an explanation to all special *GENIpro* terms and some general communication terms as well used in this manual. Also abbreviations is explained here.

1.3 Interfacing to *GENIbus*

The communications protocol *GENIpro* defines a standardized data and control interface to Grundfos motor and pump units. When these units are wired together in an twisted pair bus topology the protocol becomes the basis for the versatile Grundfos fieldbus system - *GENIbus* - with new possibilities for distinct devices to work together, share data and centralize monitoring and control.

GENIbus is based on the EIA RS485 standard for data transmission using differential voltage signals. This standard is not documented in this manual but can be found in other literature, f. ex. /4/. The *GENIpro* Physical Layer Specification includes Bus Idle Detection (BIDIrq) and Bus Busy Detection (BBD). Using the *GENIpro* software release which means developing a **full functional implementation** requires these signals to be available from the hardware (see figure D.2.2 in appendix D). Implementations based on the official *GENIpro* software C-source code release will be guaranteed a full functional compatibility with all *GENIbus* units - Master Unit Types as well as Slave Unit Types.

Devices operating as *GENIbus* Master Unit Types	- Pump Management Unit, PMU - Pump Communication Unit, PCU - Pump Functional Unit, PFU - Grundfos Gateway, G100 - Submersible pump control unit, CU3 (if paired with SM100 module) - UPE twin pump
Devices operating as *GENIbus* Slave Unit Types	- UPE single pumps - Pumps with MGE motor, E-pumps - Submersible pump control unit, CU3 (not paired with SM100 module - Sensor Module, SM100

Table 1.1: *Grundfos GENIbus Devices April 1997.*

However, establishing a **simple dialogue** with a *GENIbus* Slave Unit Type (like a UPE single pump, see also table 1.1) can be done in a very simple way as well, because accessing a *GENIbus* system of all slaves requires no handling of Bus Control. All it takes is the correct hardware, the correct Telegram format and a few timing rules (see table 1.2). Proceeding in this way - and being aware of the limitations - makes *GENIbus* communication easy. The application programmer can write the complete protocol software himself and design a poll cycle mechanism of his own. This makes him free to use whatever operating system, target controller, programming language, development environment and presentation system he likes.

The requirements for the two different "levels" for an implementation of a *GENIbus* interface is summarized in table 1.2

Necessary to interface to *GENIbus*	Where to read about it	Full Functional Implementation		Dialogue Implementation	
		PC with Windows	Any target	PC with Windows	Any target
Bus Unit Tables	Chapter 5.1 Chapter 5.5	All handled by ***GENIpro Master Device Driver***	Bus Unit Tables written by application programmer according to Functional Profile for the unit	All handled by ***GENIpro Dialogue Driver***	Not needed
Operating System	Chapter 5.1 Appendix D.3		Must fulfill *GENIpro* specification for Operating System Services (use fx. RTOS78K)		Free design and implementation Using *GENIpro* Telegram format Reduced Timing demands: - Inter Byte Delay <= 1.2ms - Inter Data Frame Delay >= 3ms - Reply Timeout >= 50ms No Bus Control demands except that bus must be left idle (=1) between Telegrams.
Application Program Interface (API)	Chapter 4 Chapter 6 Appendix A		*GENIpro* C-source code files (*GENIpro* Master Management, Presentation Layer and Logical Link Control)		
Handling of - Telegram Format - Timing Specification - Bus Control	 Chapt. 3 / App. C Chapt. 2 / App. E Chapt. 2 and 4				
GENIpro Bus Drivers	Appendix D.2		*GENIpro* C-source code target Bus Driver		
GENIpro Physical Layer (hardware)	Appendix D.2	- UART (9600baud, Start bit=0, Stop bit=1) - RS484 Transceiver circuit - BIDIrq/BBD Signals A Grundfos *GENIbus* PC-adapter (No. 625395) can be used	- UART (9600baud, Start bit=0, Stop bit=1) - RS485 Transceiver circuit - BIDIrq/BBD Signals	- UART (9600baud, Start bit=0, Stop bit=1) - RS484 Transceiver circuit A Grundfos *GENIbus* PC-adapter (No. 625395) or a standard RS232/RS485 adapter can be used	- UART (9600baud, Start bit=0, Stop bit=1) - RS485 Transceiver circuit
Limitations		None! Is interoperable with all *GENIpro* based Slave Unit Types and Master Unit Types	None! Is interoperable with all *GENIpro* based Slave Unit Types and Master Unit Types	- Cannot co-operate with *GENIbus* Master Unit Types. - All other units on the bus must be Slave Unit Types.	- Cannot co-operate with *GENIbus* Master Unit Types. - All other units on the bus must be Slave Unit Types.
Usage		- Advanced Tools or bus units with networking facilities - Management of any *GENIbus* system	- Advanced Tools or bus units with networking facilities - Management of any *GENIbus* system	- Service- and Configuration Tools - Management of systems with only slaves	- Service- and Configuration Tools - Management of systems with only slaves

Table 1.2: *Implementation of a GENIbus Interface for gateways, control units, management systems or the like.*

1.4 Application Example: Pump Management System for Heating

The figure below illustrates two *GENIbus* systems with a number of "actuator" devices connected. These are in this example all GRUNDFOS circulator pumps with electronic speed control (UPE pumps), but other types of devices could be connected as well. We can think of both examples as a part of a *Building Management System*. The *GENIbus* network works as a subsystem where more or less of the control and supervision can be taken care of autonomously.

The example to the left shows the Grundfos Pump Management System 2000. A *Pump Management Unit (PMU)* is a device for operating and controlling UPE pumps using *GENIbus*. A user can, via a menu based interface, configure the PMU and thereby select which pumps to control and how. Typical options are *clock program* (energy saving) and *zone control* (operation of pumps in parallel). What functionality a Pump Management Unit has is a matter of its *Application Program*, and of no concern to *GENIpro*. The PMU interface to the user has an integrated keypad and a display. A PMU controlling a number of UPE pumps constitute in itself a simple version of a Management System.

The *Pump Communication Unit (PCU)* is used as a bridge to non protocol based control. Analogue inputs are used for setpoint control, digital (on/off) inputs for START/STOP commands and digital outputs for status reporting (Alarm/Operation/Local). The PCU can use the control inputs directly to operate pumps or - if a PMU is present as well - let the PMU use them for zone control.

The *PC-master*, shown in the example, should illustrate, that a PC can be connected to *GENIbus* when it has *GENIpro* installed and is equipped with the bus interface hardware (RS232/RS485 Adapter). This PC-master would typically be some kind of Service Tool and, if needed, it could take over the control of the bus while servicing takes place.

The example at the right shows the use of the *Grundfos Gateway G100* for connection to another fieldbus which could be the main bus in a Building Management System. G100 supports communication between *GENIbus* units and several other protocol based communication systems among which are modem, radio and Profibus. Because *GENIpro* allows the shift of bus control from one master device to another, the control of the bus can be taken over by the gateway when data exchange and control from the main bus is needed. This means that all the *GENIbus* units can be virtually operated and supervised from a central BMS.

The use of a *GENIpro* based PC Tool is possible in this configuration too. Either the tool takes over the bus control or it operates on the bus without interfering the operation of the gateway.

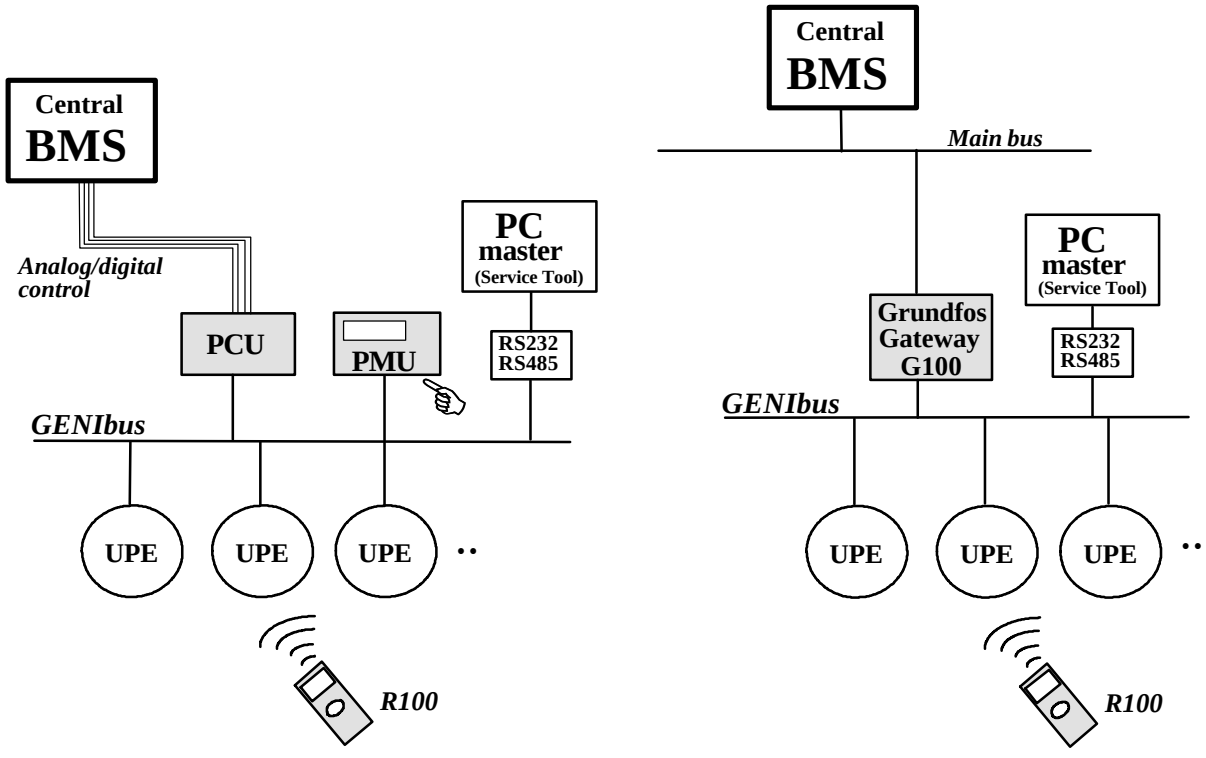


Figure 1.4: *Examples of GENIbus systems.*

1.5 Application Example: Water Works Control

This example illustrates two of the facilities supported by a *GENIbus* system: the ability to define *Subnetworks* and to dynamically shift the control of the bus from one master device to another.

The CU3 is a control unit for submersible pumps. Among other things, it supervises the line currents, line voltages, power consumption and temperature in the submersible motor. The CU3 protects the submersible motor against overload. Alarm limits and operation mode can be programmed via push buttons or via R100. CU3's can be wired to *GENIbus* for central supervision and control.

Each CU3 can operate with an optional sensor module (SM100) which is also wired to the bus. By use of R100 each SM100/CU3 couple is logically *bound* together - *GENIpro* now handles them as a Subnetwork. The CU3's request data from the SM100 they are bound to. To avoid the bus being jammed by units trying to communicate simultaneously, all this data traffic is controlled by a *Bus Master*. *GENIpro* will automatically appoint one of the CU3's to play this role. From the outside it cannot be seen which one it is - and it doesn't really matter to the operator of the system. The CU3 with Bus Master Status takes care that all the other CU3's are allocated time to use the bus in an organized manner. The mechanism in *GENIpro* making this type of operation possible is the ability to have *Submasters*.

A management device, f. ex. implemented on a PC with *GENIpro* included, could be connected to the bus for centralising control and supervision and to give the operator a highly detailed graphical interface. To be able to control all the CU3's and to request all the information from them, the management device has to be Bus Master. Because *GENIpro* supports the shifting of Bus Master Status between units, the management device can automatically be brought to take control without the operator even knowing from which CU3 to take it.

If instead this PC based management device was the Grundfos Gateway G100, the control and supervision of the water works could be done remotely via a telephone modem.

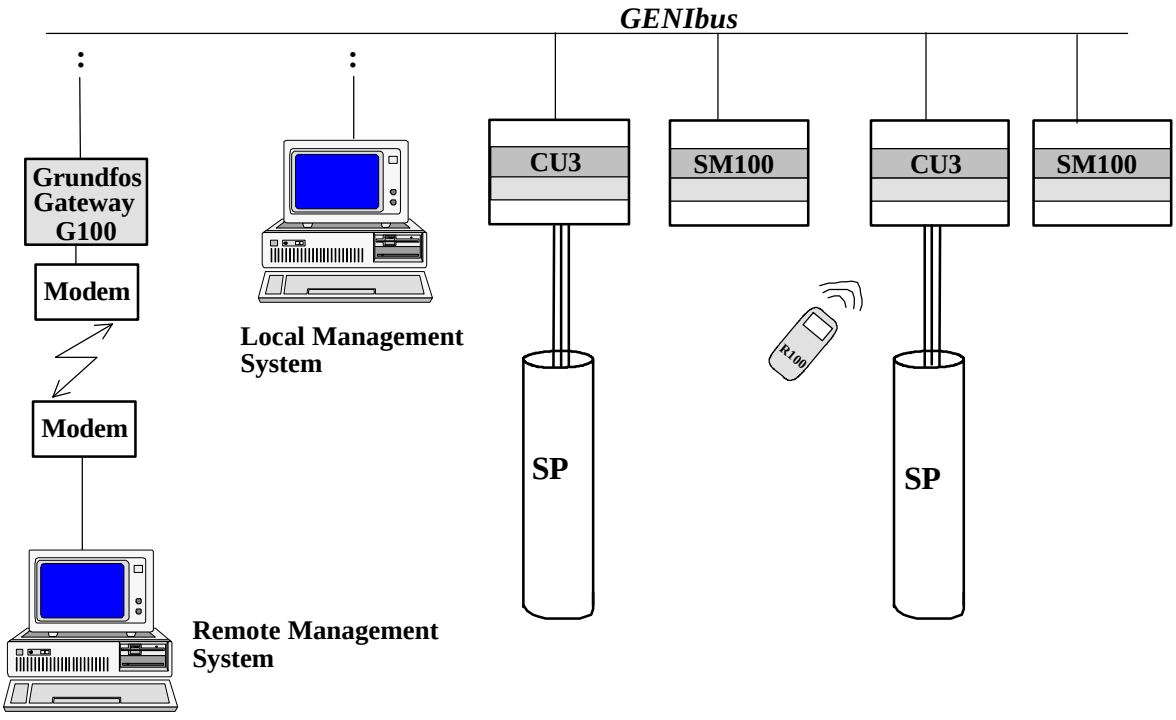


Figure 1.5: *Water works control with GENIbus.*

2. Bus Topology and Functionality

2.1 Topology

Figure 2.1 below shows what could be an installed Central Network Management System with several different units connected and communicating via *GENIbus.* Any such units physically connectable to the bus wires and which apply to the *GENIpro Specification* (summarized in App. E) will generally be termed *GENIbus Units*. These units could be different kinds of pumps, motor and the like, but to the bus, the main thing of interest is which one of two *Bus Unit Types*, a given unit belongs to, and which one of four possible *Bus Modes* it operates in.

The *Bus Unit Types* and their respective modes are shown in figure 2.2. *Master Unit Types* and *Slave Unit Types* only have one mode in common, the *Slave Mode.* In this mode the two types will show an identical behaviour. How a *GENIbus Unit* can use the bus and how it cooperates with other *GENIbus Units*, its permissions and bus characteristics, depends highly on its *Type* and its *Mode*.

Figure 2.1 also shows an *open Subnetwork*, a *closed Subnetwork* and a *Group.* The description of what's behind these concepts will be put off while we concentrate on the more superior functions of the bus.

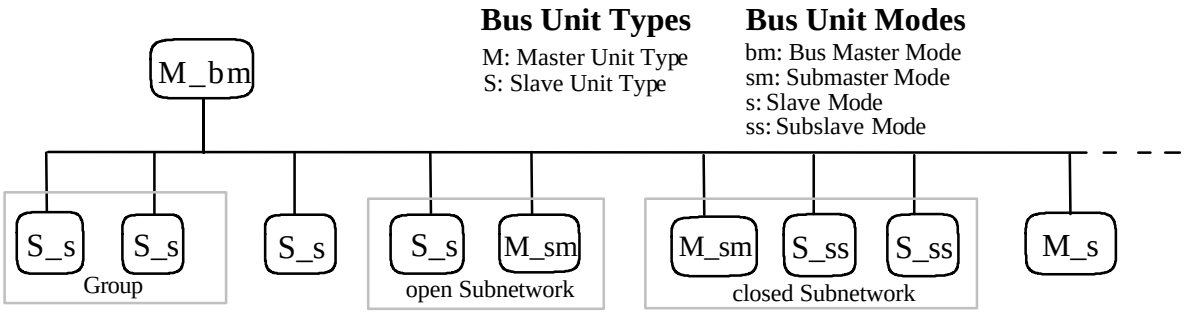


Figure 2.1: *Bus topology*

Basically *GENIpro* is a *Master-Slave Protocol*: the master requests, and the slave replies. The master is always the *Initiator* of a *Communication Session.* Although the physical topology of the network is a *bus*, with all units connected directly to the same pair of wires, we can see, that the logical topology is a *star*. All data exchange between *GENIbus Units* must take place via a central point, the master. At any instant in time there can only be one bus unit (or none) which has the control over the bus, all other bus members must be obedient slaves. This master we name the ***Bus Master***. We say it has *Bus Master Status* or operates in *Bus Master Mode.* It has the full control of when a given bus unit is allowed to communicate and for how long. This is called *Bus Allocation*, and the mechanism is seen to be *central*. There can be other units of the Master Unit Type connected to the bus, but they operate either in Slave Mode or in Submaster Mode. However they carry the potential of becoming a Bus Master. More about that later.

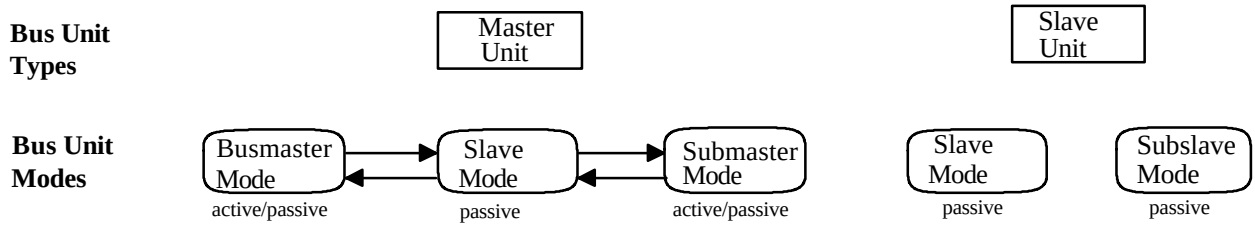


Figure 2.2: *The different Bus Unit Types and Bus Unit Modes*

The rules in the protocol deciding which unit is to be the master is in network terminology called the *Arbitration Mechanism.* A distinction between the Arbitration Modes *Bus Configuration* and *Bus Operation* is made. In the *Bus Configuration Mode* the arbitration is a matter of *appointing a Bus Master*. The Arbitration Mechanism is *decentral*, meaning that all units of the Master Unit Type participate. They are all in their Slave Mode. The mechanism is based

on the assumption, that all these units have a different and unambiguous *address*. Because the process of *Bus Master Appointment* during *Bus Configuration* uses the *Unit Address* as the basis for a timeout time, it is predictable which unit will be appointed to Bus Master - it will always be the one with the lowest address. Meanwhile, Bus Master Status can also be handed over between bus units like a *token* when the bus is operating. Again, the default Arbitration Mechanism gives the highest priority to the Master Unit Type with the lowest address but this can be controlled from the Application Program.

In the *Bus Operation Mode* the Arbitration Mechanism is *central.* After having recognized all bus members (how this is done is described later), the Bus Master normally starts requesting data from them and at the same time it controls (by having to acknowledge) any mode shift another Master Unit Type might wish to carry out.

Because a Master Unit Type is not necessarily confined to operate in one specific Bus Unit Mode once the bus has been configured, we talk about a *dynamic* mode concept, which is indicated by the arrows in figure 2.2. Contrary to this the Bus Unit Mode for a Slave Unit Type never changes. It operates either in Slave Mode or in *Subslave Mode*, which is the result of a manual setup. In other words: a Slave Unit Type works with a *static* mode concept.

In its basic configuration the *GENIbus* has one Master Unit Type operating as the Bus Master and controlling several Slave Unit Types operating in their Slave Mode (a Pump Management System 2000 with one PMU-master and a number of UPE-pumps is an example). To fulfil the requirements in this simple configuration, no *dynamic* mode changing is needed when the bus is operating. The described situation is the basis for all *GENIbus* applications, and in many cases this is as far as it goes in complexity. Still, by utilising the dynamic mode concept *GENIpro* can offer some extra features, to extend the functionality of a *GENIbus System* to meet special demands, or to optimize the bus usage in certain applications. The Waterworks example from chapter 1.3 is such a case. With reference to figure 2.1 we shall now go deeper into the details.

The term ***Submaster*** will be used as a short name for a *GENIbus Unit* that regularly or permanently uses the *Submaster Mode* for the purpose of controlling a *Subnetwork.* A Submaster is essentially a Master Unit Type, that can get permission from the Bus Master to use the bus and operate like a master for a defined short time slot. The Bus Master is *passive* while this takes place, and the Submaster becomes *active* - meaning the only *Initiator of Communication Sessions.* Referring to the network in figure 2.1, we can conclude, that out of the three Master Unit Types shown, that are not operating in Slave Mode, only one can be active. Although we can not predict which one by just looking at the system, we know for sure, that the Bus Master is the one controlling the situation.

A Submaster will restrict Communication Sessions to its own preconfigured *Sublist Members.* Sublist Members are always Slave Unit Types. We talk about a *closed Subnetwork* if these members are in Subslave Mode, otherwise we call it an *open Subnetwork* (both are shown grey framed in figure 2.1).

To the Bus Master the closed Subnetwork will appear as if it was only one unit, or to put it in other words, the Bus Master sees only the Submaster, treats it as a slave, and is unaware of the logical underlying Subnetwork. Subnetworks offers the possibility of utilising the same bus for applications like twin pump control, bus connected sensor feedback etc. Any application where distributed control depends on data being transferred between two or more units might benefit from using the logical subnetwork instead of physically wiring an extra network.

To service its subnetwork, the Submaster needs to be *active* with more or less regular intervals. It has to request the Bus Master for a time slot each time. This request it tails to a reply when it is addressed. The mechanism is called *Request From Slave (RFS).* Requesting *Submaster Status* is only one example of what *RFS* can be used for. The Submaster has to remain *passive* until the Bus Master acknowledges the request. The Allocation Mechanism ensures that this is done before a Communication Session with another unit is initiated.

A unit belonging to a *closed* Subnetwork is called a ***Subslave*** as a short name for a Slave Unit Type in its preconfigured Subslave Mode. The only special about a Subslave is, that it does not respond to a *Connection Request* (described later). This means it can not be automatically recognized by the Bus Master.

An *open* Subnetwork is somehow similar to the *closed* Subnetwork, but some important differences are worth to notice. The *members* are operating in their Slave Mode, and can thus be addressed by the Bus Master as well as by the Submaster. One sensefull utilisation of this would be to have the Submaster taking care of control actions and the Bus Master taking care of system data acquisition and supervision. On an *open* Subnetwork is it possible to have more than one Submaster (all members could for that sake be Submasters). The *GENIpro* Allocation Mechanism guarantees that only one is *active* at a time.

Figure 2.1 also shows the concept of a ***Group***. Here no logical Subnetwork exists - all units are addressed directly from the Bus Master. However, they are not treated like individuals, but like a logical unit - they share the same

address. By having them share this *Group Address*, they can be operated synchronously, and moreover, a lot of time consuming data transmission is avoided, which raises the efficiency substantially for a *GENIbus* with many slaves. Group addressing can not be used for data requesting. It would lead to data collision on the bus if several slaves tried to reply simultaneously. Besides having a Group Address, any slave always has a *Unit Address* as well. So, to request data from a Group the group members must be addressed individually.

In table 2.1 and 2.2 the mode characteristics for a Master Unit Type and a Slave Unit Type are summarized.

Master Unit Type

Mode	Characteristics	
Bus Master	Entering:	- Can be Bus Master as a result of a Bus Configuration - Can be appointed to Bus Master by the current Bus Master (*BM_ack*).
	Actions:	- Performs an Auto Poll Cycle which can be controlled by the Application Program. - RFS-acknowledge (*BM_ack, SM_ack)* done automatically. - Performs Connection Requests. - Updates the *Network List.* - Is passive when a Submaster is active.
	Leaving:	- Goes into Slave Mode when RFS-acknowledge to be Bus Master (*BM_ack*) is sent to another unit. (NB! recaptures the bus control if this other unit fails to be active within $t_{bus\_act\_to}$).
Submaster	Entering:	- When the Application Program wants to carry out a Communication Session, *GENIpro* uses the RFS-option to request Submaster Status (*SM_rfs*) and will receive an *SM_ack* from the Bus Master.
	Actions:	- Performs an Application Program controlled Poll Cycle of a preconfigured Subnetwork. - Ignores any RFS-options received from slaves.
	Leaving:	- Returns to Slave Mode when the Submaster Timeout event occurs
Slave	Entering:	- Is in Slave Mode after power up. - Whenever Bus Master Mode or Submaster Mode is left Slave Mode will be entered.
	Actions:	- Records itself as Unpolled if not addressed for a period of $t_{max\_unpolled}$. Will then be sensitive to Connection Requests.
	Leaving:	- By entering Submaster Mode (reception of *SM_ack*) which is used by *GENIpro* to carry out a Communication Session on request from the Application Program. - Can become Bus Master as a result of a Bus Configuration. - Can be appointed to Bus Master by the present Bus Master (reception of *BM_ack*).

Table 2.1: *Summarizing the dynamic mode characteristics for the Master Unit Type.*

Slave Unit Type

Mode	Characteristics
Slave	- Records itself as Unpolled if not addressed for a period of $t_{max\_unpolled}$. Will then be sensitive to Connection Requests.
Subslave	- Configured to Subslave Mode by the Configuration Parameter `geni_setup` or by some hardware setting. - Never sensitive to Connection Request.

Table 2.2: *Summarizing the static mode characteristics for the Slave Unit Type.*

2.2 Addresses

The address range for *GENIpro* is from 0 to 255. These are allocated according to Table 2.3. Due to the reservation of certain addresses for special purpose, and to the existence of both a *Group Address* and a *Unit Address* the maximum number of logical connected bus units is lower than 256. When used to its full extent, the capacity of logic addresses will be:

229 for Master Unit Types
200 for Slave Unit Types

and each of these units can belong to one of 16 possible groups.

The *Connection Address* is the "door" to *GENIbus* - it is used for *Connection Requests*. Via this address new units, that were previously unknown to the Bus Master, can be recognized and added to the *Network List* and the *Poll List*. The Application Program can via function calls remove units from the Poll List as well as add units. As seen in table 2.4, Subslaves don't have the Connection Address, and will not be recognizable by the Bus Master. This means that *GENIpro* will never install Subslaves automatically in the Bus Master's Network List (They can however be deliberately installed in the Network List by the Application Program). The Subslave should appear in the *Sublist* of a Submaster (this is described later).

The *Global Address* can be used for the special purpose of addressing all bus units at the same time. Closing down a whole system simultaneously could be one reason. Because any *GENIbus Unit* can always be addressed via the Global Address it can be used for communication between a master and a slave in a situation where they are the only units on the bus (or when using the *GENIlink* channel).

Global Addressing and Group Addressing are both situations where a reply Telegram must often be avoided. *GENIpro* has the option of omitting a reply. Such a Telegram is called a *Message*. The active master can select whether it wants the slave to send a reply back or not. This information is contained in the *Start Delimiter* of the master Telegram. For Application Program initiated Communication Sessions it is the responsibility of the Application Program to choose the correct option - reply or not - in any situation.

Possible addresses	Range	Comments
Reserved	0	
Unit Addresses	1-29	Only allowed to be used by *Master Unit Types*
Reserved	30-31	
Unit Addresses	32-231	Can be used by all bus units: *Master Unit Types* and *Slave Unit Types* 231 (0xE7) used as factory default Unit Address
Group Addresses	232-247	For logical grouping of units. 247 (0xF7) used as factory default Group Address
Special Addresses	248-253	Reserved for special purpose 253 (0xFD) used for *GENIlink* remote controller R100 252 (0xFC) used for *GENIlink* remote controller R50
Connection Address	254	For recognizing of new Bus Units (Connection Request)
Global/Broadcast Address	255	For transmission of information common to all connected Bus Units

Table 2.3: *The address allocation scheme*

Bus Unit Type	Addresses
Master	A Unit Address in range [0;29] or [32; 231] A Group Address in range [232; 247] The Connection Address [254] The Global Address [255]
Slave	A Unit Address in range [32; 231] A Group Address in range [232; 247] The Connection Addr. [254] (not in Subslave Mode) The Global Address [255]

Table 2.4: *Addresses for a GENIbus Unit*

2.3 Bus Configuration

Bus Configuration means the state where the bus hasn't got any *Bus Master*, and tries to figure out which of the connected units should be appointed. This situation arises when the installation is powered on or when the Bus Master on an operating bus is suddenly switched off. *GENIpro* does this configuration automatically. We already know that the Master Unit Type with the lowest Unit Address will be appointed to Bus Master. The following is a description of how this mechanism works.

Figure 2.3 shows an example of a *GENIbus* under Bus Configuration. Some manual setup of the bus units might have to be done prior to the configuration done by *GENIpro*. In the present example M3_sm will need to know the existence of its Subslave S3_ss, because automatic configuration is not possible on Subnetworks.

Figure 2.3: *An example of a GENIbus System under Bus Configuration. The lines indicate logical connections.*

With no Bus Master the bus is *idling*, because only the Bus Master can initiate a Communication Session or send a RFS-ack to a slave in order to give it Submaster or Busmaster Status. All Master Unit Types have the potential of being appointed to Bus Master, but initially they are in Slave Mode. The Bus Configuration in the example now goes on according to a specific procedure. Notice that this is handled automatically by *GENIpro* and takes place in all of the units. It is a *decentral* or *distributed* mechanism, where units via their physical connection to the bus, come to an agreement about who to give control of the bus. The *Bus Configuration Procedure* goes on as follows. Figure 2.5 shows a simple flow chart:

1) When the time with no bus activity exceeds $t_{bus\_act\_to}$ the internal event BUS ACTIVITY TIMEOUT will be generated.
2) A master appointment timer with a timeout period of 4 times the Unit Address in ms will be initiated. When it times out, the internal event MASTER APPOINTMENT TIMEOUT will be generated.
   If, at any time during this timeout, the unit receives the Start Delimiter of a Telegram the procedure will be stopped at this very moment, because a Telegram on the bus means that another Bus Master has been appointed.
3) When the timeout event happens, a *Bus Busy Check* is performed to see if anyone is transmitting. If the bus is not busy a Message called a *Bus Capture Telegram* is broadcasted to tell all other bus members that this unit has been appointed to Bus Master.

Because the Unit Address is used as the basis for the master appointment timeout, the master unit with the lowest address will always get the Bus Master Status as a result of a Bus Configuration. The Bus Master Status can be shifted between the Master Unit Types on the bus controlled by their Application Programs. This type of control is called *Master Management* but this description is put off until chapter 4.

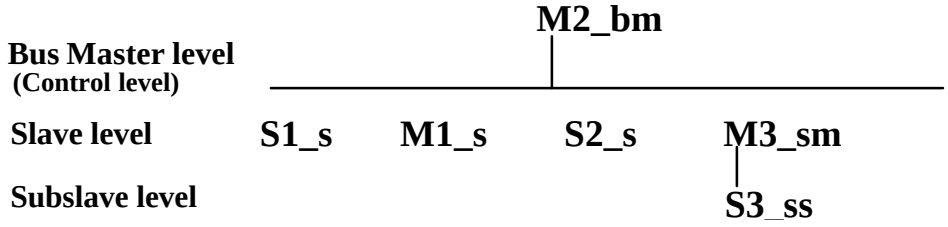


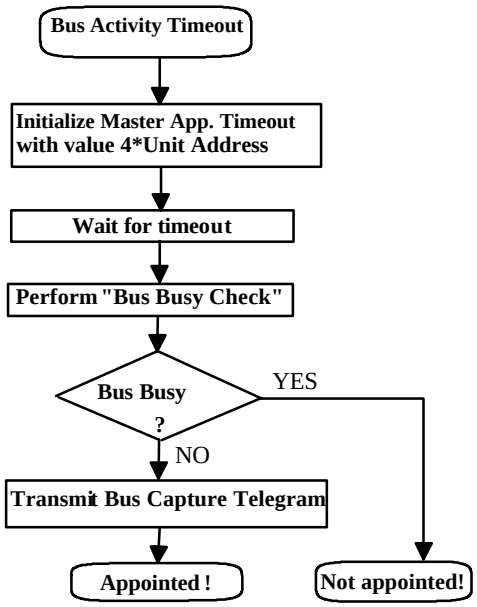
Figure 2.4: *The GENIbus example after Bus Configuration, assuming M2 has the lowest Unit Address*

Let us for the time being assume that M2 has the lowest address on the bus and due to this it became the Bus Master. The situation in the example will be as shown in figure 2.4. M2 is logically at the control level, but no other unit is yet known on the Slave level.

The Application Program in the Bus Master (M2) can communicate with any legal Unit Address. The control of the bus is however still taken care of by *GENIpro*. In th "background" *GENIpro* runs an *Auto Poll Cycle* which takes care of recognizing units on the bus and - when recognized - polling them regularly.

After being recognized with *Connection Request* a bus unit is added to the Network List. This list is a very important remedy for the protocol to keep account of connected units. Moreover, the Application Program can use it for Application Program initiated communication - known as *Direc Addressing* (See chapter 4).

Any bus unit in Slave Mode will reply to a Connection Request, if it hasn't been requested for data from the bus within the time $t_{max\_unpolled}$. It will delay its reply the randon time period $t_{conn\_rep\_to}$ and abstain from replying if it detects any bus activity in this period.



Figur 2.5: *Bus Master Appointment*

Figure 2.6 shows the reply to the first Connection Request, assuming $t_{max\_unpolled}$ has passed and that S2 was the first unit to time out. The Bus Master has *recognized* S2 and will add it to the Network List and the Poll List.

Having replied to a Connection Request will not make a bus unit record itself as Polled. It will still consider itself as Unpolled and try replying to any new Connection Request until it receives an ordinary poll (a Data Request) which asks it for data. To make a bus unit Polled, the Auto Poll Cycle sends a Data Request to a bus unit as soon as it is recognized. This is shown in figure 2.6 as well.

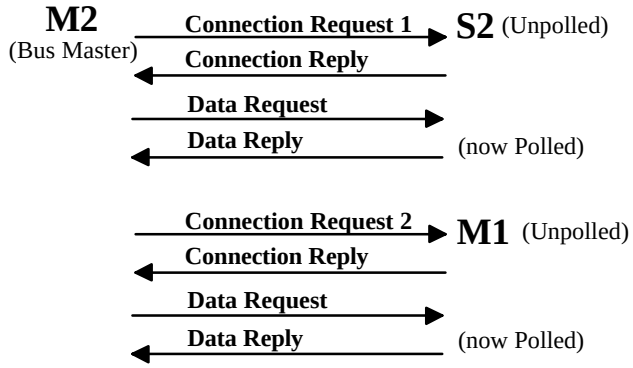


Figure 2.6: *Replies to the Connection Requests and polling of recognized units*

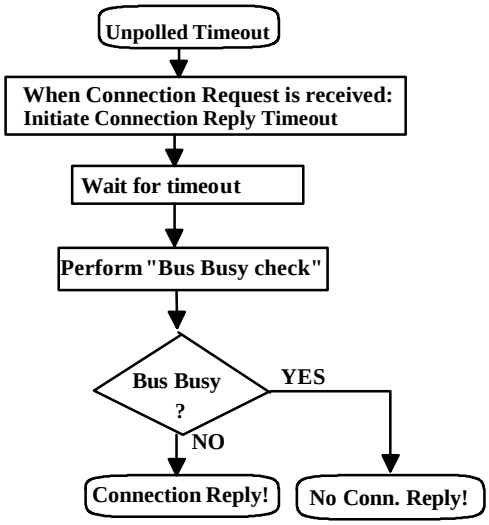


Figure 2.7: *Generation of Connection Reply*

For each succeeding Connection Request all the bus units except Subslaves will be recognized one by one in random order and added to the Network List. We see in figure 2.6 that after having recognized S2 the Bus Master M2 recognises M1 and so on. We end up with the situation shown in figure 2.8.

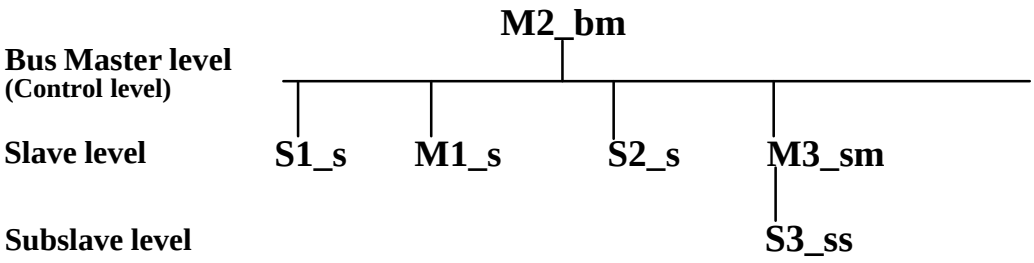


Figure 2.8**:** *The GENIbus example when all bus members have been recognized.*

A Master Unit Type M4, assumed to have a higher priority than M2 is now connected to the bus. In *GENIpro* priority is related to the Unit Address both during Bus Configuration and during Bus Operation. The lower the address the higher the priority. Priority controls which unit ends up being the Bus Master. After having been recognized by M2, M4 is polled by M2 (see fig. 2.9). It compares its own Unit Address with that of the Bus Master M2. Concludes that M2 has a lower priority and tails an *RFS to be Bus Master* (BM_rfs) to the Data Reply. *GENIpro* acknowledges any RFS immediately (BM_ack) so M4 is appointed as new Bus Master. M4 will then recognise all bus members one by one and the situation will eventually be as shown in figure 2.10.

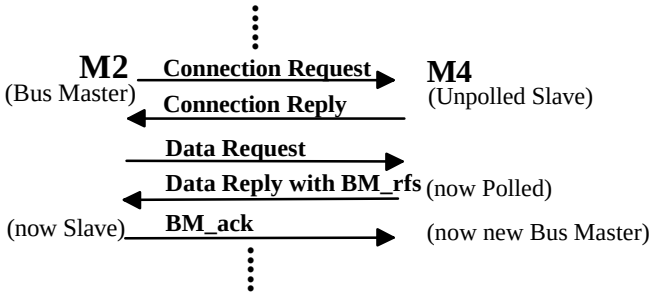


Figure 2.9*: Higher Priority Master Unit Type M4 gets Bus Master Status.*

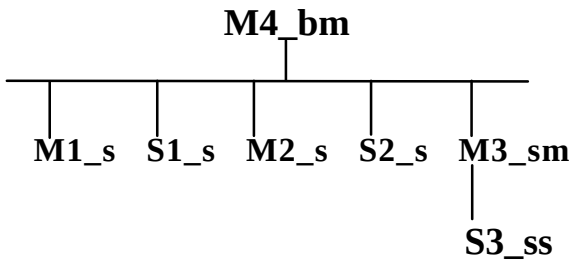


Figure 2.10*: The high priority M4 has Bus Master status*

With any new appointed Bus Master, e.g. a laptop service tool or a Gateway, it is easy to re-establish the previous bus configuration. Presupposing the old Bus Master was the only master connected to the bus, the laptop can just be disconnected and the Gateway can stop all communication actions. The result will be a Bus Configuration where the old Bus Master regains its status as the *active* master.

An alternative is to appoint the old master *explicitly* and avoid a Bus Configuration. If a Master Unit Type receives an RFS-acknowledge to be Bus Master (BM_ack) it will accept it, as if it had actually requested this appointment. The mechanism is similar to *token passing* on token based networks. Any acknowledging that the passing went well is not involved, nor is retransmission. A lost token will automatically make the former Bus Master immediately recapture the Bus Master Status when the first Bus Activity Timeout occurs.

2.4 Timing and Time Consumption

On a network with a *bus topology* all units can directly receive all Telegrams transmitted. A given unit is of course interested in receiving only those Telegrams, that are actually addressed to it. Spending time storing and treating bytes for which it has no use, is a waste of computer power. A *GENIbus Unit* is able to discard Telegrams, without having to receive more than a fraction of them because it fulfils the following:

1) *Destination Address* is placed as a part of the *Telegram Head.*
2) It is possible to detect the beginning and the end of a Telegram without having to process it.

In the *GENIbus* Telegram format (see App. C) the Destination Address information is placed as the 3'rd byte. As soon as this byte is received, the unit can decide if the receiver should be switched off for the remaining bytes or not. This fulfils the first condition.

The second condition seems more involved, but in effect it means detection of some "separator" between Telegrams. We define this "separator" to be an *idling line* and call the sensing mechanism for ***Bus Idle Detection***. This solves the problem. Any *GENIbus Unit* must be equipped with this facility - its included in the *GENIpro Physical Layer Specification.* The timing in *GENIpro* ensures that enough time, called the *Inter Data Frame Delay*, is given between Telegrams for the Bus Idle Detection mechanism to work. Details in Bus Idle Detection can be found in App. D.2.

Besides serving as the separating event between Telegrams, Bus Idle Detection also plays the role of *synchronization* on the bus. Each time the bus idles the units can syncronize their timing, which is important for the *Bus Activity Timeout*, and internal states in *GENIpro* can get reset to assure recovery from a possible "hang-up" condition. The first legal *Start Delimiter* character received after a Bus Idle Detection is interpreted as the beginning of a Telegram.

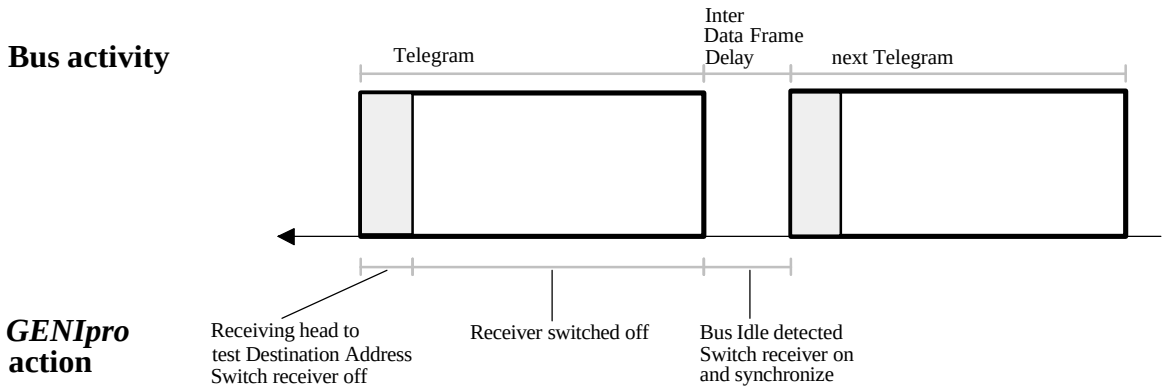
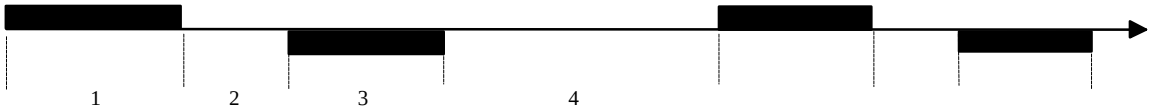


***Figure 2.9**: The Illustration of Bus Idle Detection used for Telegram synchronization and used for computer load relief by receiver switch off.*

GENIpro is based on polling. As it has been described the bus should not be left idle, as this would lead to a Bus Configuration. It is the responsibility of the *Bus Master* that the maximum allowed idle time, $t_{bus\_act\_to}$ is not exceeded.

Any member being addressed consumes time having to receive the Telegram transmitted to it byte for byte. It is the responsibility of the application programmer of a *GENIbus Unit*, that application tasks have enough time to execute when *GENIpro* has taken its share - even in a worst case situation. What this means more precisely is of course target dependent. So, it is out of concern to the Bus Master whether a fast polling of a slave or polling with long Telegrams cuts deeply into the time resources of this slave. A unit applying to the *GENIpro Specification* has enough capacity to deal with the worst case situation and to deal with it reliably all the time.

In figure 2.10 a typical time sequence of the bus activity is shown. What time consumption this means to a particular unit, depends on its computational power and its available peripheral devices (UART). A relative load of 5-20% of the time resources during reception and transmission is likely for 8 bit computers with clock speeds around 5-10MHz. The *GENIlink* channel is expected to be a 2 time bigger time consumer during reception and transmission than the *GENIbus* channel.



1: Bus Master initiates a *Data Request Session* by transmitting a request to a slave
2: Bus idles while the slave treats the request and prepares a reply. This time period must be at least equal to the Inter Data Frame Delay.
3: Slave transmits the reply. The reply must be started before the maximum specified Reply Timeout Time is exceeded
4: Bus idles while the Bus Master treats the reply and prepares the next request. This time period must be at least equal to the Inter Data Frame Delay.

***Figure 2.10:** Example of bus loading and bus idling. The Inter Data Frame Delay is small compared to the other time periods and is not shown.*

2.5 Poll Cycle Control and the Network List

GENIpro is the means to make several units - sometimes functionally quite different, operate together as a *distributed system.* The Bus Master makes it work. It administrates the control laws on the system level, monitors that any unit fulfils its purpose, collects data and presents them to an operator or route them to a higher level network.

GENIpro uses the *Network List* to record bus members and their bus characteristics. The Bus Master can recognize members automatically with the Connection Request, *GENIpro* adds them to the Network List. A Submaster has no Network List because it is not used as the basis for a Poll List. Instead the Application Program in a Submaster uses a *Sublist* for this purpose.

All access to *GENIpro* from the Application Program must take place through a special part of the protocol called the *Application Program Interface (API)*. The *Services* in the API defines precisely what the Application Program is allowed to do. In figure 2.11 the principles of polling are shown for a Bus Master and for a Submaster. Below the characteristics are summarized, together with illustrative examples.

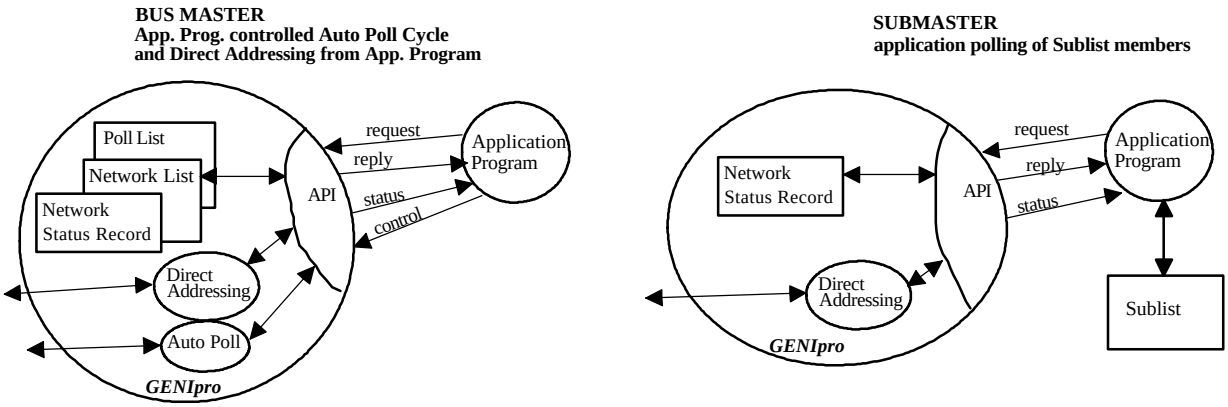


Figure 2.11: *Bus Master polling and Submaster polling*

Bus Master doing auto polling:
* Polling done automatically from the Poll List but under control of the Application Program.
* The Application Program can modify the *Network List* and the *Poll List*.
* Used by a "master management device" to operate a system of more slaves (Like the PMS2000 System)
* The Application Program can use Direct Addressing to any unit

Submaster:
* Polling done from the Application Program as Direct Addressing .
* Units to be polled are recorded in the *Sublist*.
* The *Sublist* is updated by the system operator (via the Application Program). It is a static list reflecting an *open* or *closed Subnetwork* as planned by the system designer.
* A typical application is twin pump control or bus connected sensor feedback (Se example in Chapt 1.3).

The Network List plays an important role for the network control. All *GENIbus Units* working as Bus Master maintain this list. Each entry is a *Bus Unit Record* containing information about a certain bus unit. This information is its main bus characteristics like *Bus Unit Type* and *Unit Address.* Also the recording of communication error rate for that particular unit and information to identify what Functional Profile it adheres to, are stored in its Bus Unit Record.

The updating of the Network List is the responsibility of *GENIpro* and it is done whenever a Connection Reply arrives. A reply with an error will increment the error counter for that particular Bus Unit and increment an error counter residing in the *Network Status Record* counting total no of Communication Session errors. The *Application Program* can read these error counters via the API Services, or reset them, whenever it likes.

Updating the *Network List* also means adding and removing members. A member is automatically removed if its error rate gets higher than a predefined value, and a member is added when *recognized* as a new bus member. All changes in membership are reported in the Network Status Record giving the Application Program an easy way to see what happens on the bus. Figure 2.12 illustrates the updating of the Network List.

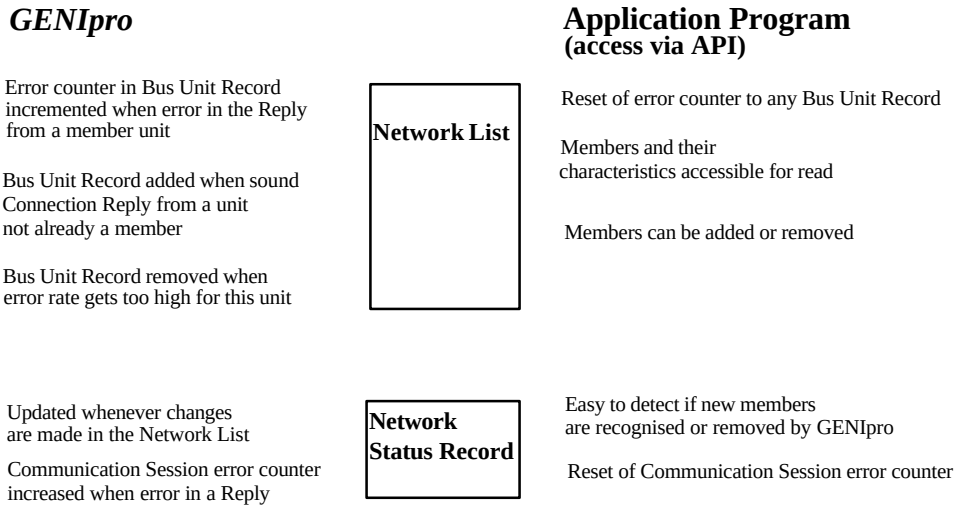


Fig. 2.12: *Access to the Network List and the Network Status Record*

The normal way to *recognize* new members, as mentioned already, is via Connection Request. This is indeed the most obvious and excellent way to do it - but not the only way. The Application Program can send a Data Request to a unit (an address), that does not already exist as a member of the Network List. If a sound Data Reply comes back, the Application Program can deliberately (via API) add this unit to the list. If an error existed in the reply, e. g. CRC-error, the unit should not be added to the Network List - even though a reply was actually received. It should be a general rule, which goes for Connection Replies as well, that "getting listed" requires a *sound* reply.

When a new Bus Master is appointed, whatever remains of an old Network List is cleared. The building of the list must start from scratch. This leads to a small time delay - typically a few seconds - before all units are recognized and listed.

How good the Arbitration Mechanism ever may be a *Bus Master Conflict* may arise. If somehow two or more units end up having the Bus Master Status the bus system will be inoperational - a chaos of colliding Telegrams will result. *GENIpro* is robust to a fault condition like that. A Bus Master detecting a Bus Busy condition repeatedly when it tries to transmit, recognizes this as a Bus Master Conflict and changes to Slave Mode. Such a behaviour is an example of an Exception Handling and in a few seconds this will lead to a recovery of the bus system.

2.6 Communication Sessions

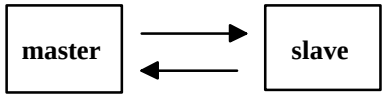


Figure 2.13: *master/slave couple*

In the following we shall briefly discuss what is meant by a *Communication Session.* For that purpose we look at an isolated master/slave couple as shown in figure 2.13. We divide Communication Sessions into:

- *Data Sessions*, with the purpose of transferring data
- *Control Sessions*, with the purpose of transferring control

A Data Session is always initiated by the *active master* (the *Initiator*) by transmission of a Telegram. All Communication Sessions are disjoint in time. A foregoing Data Session or Control Session has to be completed before the next can be initiated.

A Telegram in a Data Session can only be one of three kinds shown in table 2.5. The slave regards a Data Session as initiated if it receives a Data Request without any errors, and will behave accordingly. Because no *Data Reply* is given to a *Message* a slave can only transmit two kinds of Telegrams which are also shown in table 2.5.

Master (Initiator) transmits		**Slave transmits**	
Start Delimiter	**Telegram**	**Start Delimiter**	**Telegram**
27H	Connection Request	24H	Connection Reply
27H	Data Request	24H	Data Reply
26H	Message	-	-

Table 2.5: *Overview of the different kinds of Telegrams used for Data Sessions. Request, Message and Reply are distinguished only by the Start Delimiter of the Telegram.*

Data Sessions for an active master (Initiator)	
Session initiation event	**Session completion event**
Transmission of a Connection Request	• Reception of sound or erroneous Connection Reply • Reply Timeout
Transmission of a Data Request	• Reception of sound Data Reply • Reception of erroneous Reply after attempted retransmission • Reply Timeout after attempted retransmission
Transmission of a Message	• When the Message has been transmitted and the Reply Timeout time has elapsed

Table 2.6: *Data Sessions are always bounded by two events: a session initiation event and a session completion event.*

Data Sessions for a Slave	
Session initiation event	**Session completion event**
Reception of a sound Connection Request when *Unpolled*	• Reception of valid *Start Delimiter* during Connection Reply Timeout • Detection of "Bus Busy" after Connection Reply Timeout • When the Connection Reply has been transmitted
Reception of a sound Data Request	• Detection of "Bus Busy" when trying to reply • When the Data Reply has been transmitted
Reception of a sound Message	• When the Message has been processed

Table 2.7: *Data Sessions are always bounded by two events: a session initiation event and a session completion event.*

When a Telegram is received with transmission errors it is discarded. So, if it was a *Data Request* it will not be replied. This is a general principle in *GENIpro*. To deal with this situation, and the situation where an addressed slave does not exist, the master must regard the Telegram as "lost" when a Data Reply is not received within a given time interval called the *Reply Timeout Time*, t_{reply_to}. *GENIpro* automatically tries one *retransmission* when this happens.

If a Data Reply was received but with a transmission error, retransmission would also be attempted. The master regards the Data Session as terminated, if the Data Reply to the retransmitted Data Request fails to appear or is received with transmission errors. *Connection Requests* are not retransmitted nor are *Messages.*

Control Sessions for an active master

Session Name	Session initiation event	Session completion event
Bus Configuration (Master Appointment)	• Bus Activity Timeout	• Reception of a SD • Bus Busy Detection after Master Appointment Timeout • Sending of Bus Capture Message after being appointed to Bus Master
Bus Master Acknowledge	• Reception of RFS-Request to be Bus Master (BM_rfs)	• Sending RFS-acknowledge to be Bus Master (BM-ack)
Submaster Acknowledge	• Reception of RFS-Request to be Submaster (SM_rfs)	• Sending RFS-acknowledge to be Submaster (SM_ack)

Table 2.8: *Control Sessions are always bounded by two events: a session initiation event and a session completion event*

3. Introduction to *GENIpro*

3.1 The Protocol Layer Model

Network protocols are usually organized as a series of *layers* each one built upon its predecessor. In all networks the purpose of each layer is to offer certain services to the higher layers, shielding those layers from the details of how the offered services are actually implemented.

Layer *n* on one bus unit carries on a conversation with the corresponding layer on another bus unit. In reality, no data are directly transferred between corresponding layers on different bus units. Instead, each layer passes data and control information to the layer immediately below it, until the lowest layer is reached. Below layer 1 is the *physical medium* through which actual communication occur.

Between each pair of adjacent layers, and between the protocol and its environment, there is an *interface*. The interface defines which primitive operations and services the lower layer offers to the upper one. When designing a protocol, one of the most important considerations is to define a clean interface, minimizing the amount of information that must be passed between layers. Each layer should perform a specific collection of well-understood and related functions, breaking the whole protocol up in simpler parts easier to understand, implement and test. Once this *architecture* is founded, the implementation of one layer can be completely replaced by a different implementation, because all that is required of the new implementation is that it offers exactly the same set of services to its upstairs neighbour as the old implementation did.

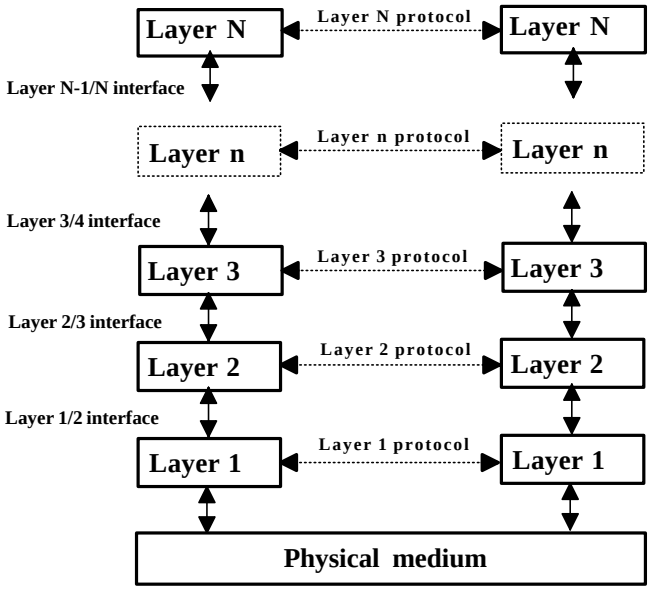


Figure 3.1: *An N layer protocol model. Virtual communication is shown by dotted lines, physical communication by solid lines.*

The International Standards Organization (ISO) has proposed a general model for the interconnection of open network systems, which is referred to as the *OSI Reference Model* (open systems interconnection). This is a seven layer model with a defined name and functionality for each layer. These names are shown in Figure 3.2 on the next page. We will not go into any further description of the OSI reference model. Readers with this interest must refer to /2/ in the literature reference, App. G.

However, it is common practice to include only those layers in a protocol design, that actually contains required functionality, leaving the rest - not empty - but out. Field buses normally use a 3 layer model resulting from a reduction of the general 7 layer OSI Model. This model is adopted as the basis for *GENIpro*.

The 3 layer model is shown in figure 3.3. Each layer corresponds to one or more equivalent layers in the *OSI Reference Model*, also the names are adopted. In the following the most important content of the different layers will be described. A more detailed description of the implementation can be found in App. D.

OSI Reference Model		Field Bus Model
Layer no	Name	
7	Application Layer	Application Oriented Layer
6	Presentation Layer	
5	Session Layer	
4	Transport Layer	
3	Network Layer	
2	Data Link Layer	Data Link Layer
1	Physical Layer	Physical Layer

***Figure 3.2**: Layers in the OSI Reference Model and the reduction to a general 3 layer Field Bus Model. In GENIpro the name Presentation Layer has been chosen for the highest fieldbus layer, because the data processing taking place here reflects the OSI specification for a Presentation layer very well.*

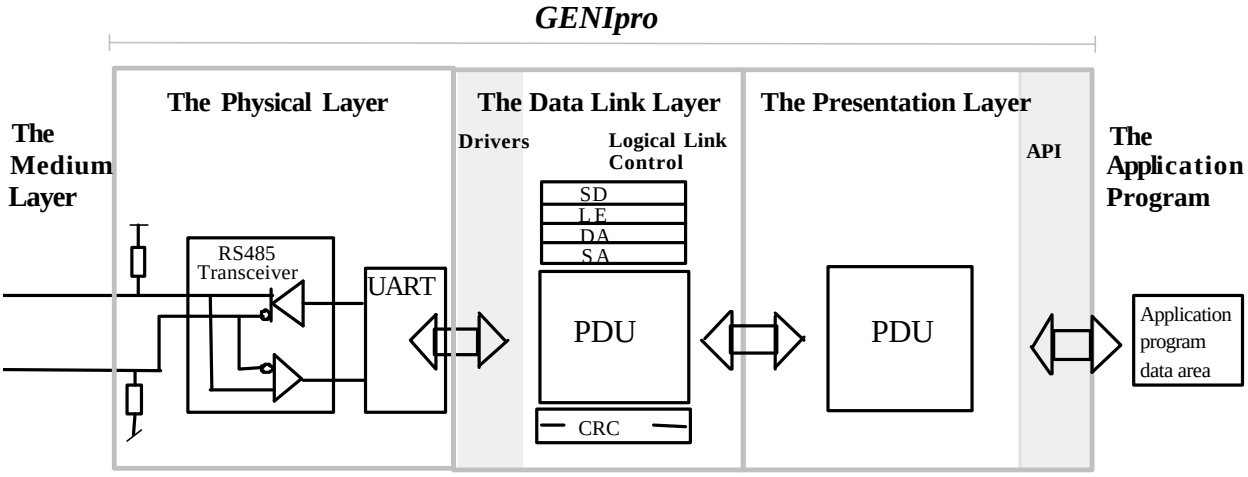


***Figure 3.3**: GENIpro three layer model. The Protocol Data Unit (PDU) is the information carrying central part of a Telegram. The Framing consist of a Head and a CRC-value (described in Chap 3.2).*

Layer 0: The Medium Layer

This is the physical medium used for the data transmission. In this case it is the electrical wiring with recommended protection circuits if used (Appendix D.2). The wires could be a twisted pair, or a shielded pair. The addition of *repeaters* can increase the transmission distance to several km. The Medium Layer is not a part of the protocol.

Layer 1: The Physical Layer

This is the *hardware* used for the data transmission. In *GENIpro* the hardware standard EIA RS485 is adopted as the physical layer. Data is transmitted one byte at a time with least significant bit first via a ***UART***. One ***start bit*** (=0) and one ***stop bit*** (=1) is used giving a total *character length* of 10 bits. An idle line is high. An interrupt driven transmission can prevent delays between characters (max. allowed delay is 15 bits). *The data coding* by the ***RS485 Transceiver*** is ***NRZ (None Return to Zero)*** meaning that a "1" is coded as a positive voltage on the line and a "0" is coded as a negative voltage. Voltage- and current characteristics according to *EIA RS485*. Hardware for Bus Idle Detection and Bus Busy Detection is also a part of the physical layer, but not shown in the figure.

Layer 2: The Data Link Layer

The Data Link Layer (DLL) and upper layers are all implemented in *software*. The boundary to the *Physical Layer* is transmission/reception of distinct bytes to/from the *UART*. A constant "data flow" can be maintained by using the *UART interrupt*. All activities having a strong connection to real time are placed in this layer. DLL is further divided into

Drivers taking care of the hardware dependencies and the ***Logical Link Control (LLC)*** taking care of *buffer handling, framing, CRC checking, Telegram synchronizing* (Idle detection and Inter Data Frame Delay) and the control of all timing (Reply Timeout, Connection Reply Timeout, Bus Activity Timeout, Master Appointment Timeout, Submaster Timeout, Unpolled Timeout, Auto Poll Clocking).

By analogy with the drivers isolating hardware dependencies from the upper layers, the LLC isolate all *real time dependency* from the upper layers. Every activity in LLC is either related to timing or is an interrupt driven data processing. The only exception being the initiation of a transmission (which comes from the Presentation Layer). Data processing in LLC has to be kept to a minimum to minimize the time spent servicing interrupts.

Figure 3.3 shows how data is being passed through the layers. Having received a complete Telegram from the *Receiver Driver* and stripped it for framing information, the Logical Link Control checks the CRC-value for errors. If no errors are detected the PDU part will be handed over to the Presentation Layer for further processing, while the whole Data Link Layer remains waiting, and the network is *idle*. If an error is detected the whole Telegram will be discarded and nothing else happens. In an *Initiator* this layer will take care of retransmission if no answer from a slave is received within the *Reply Timeout Time*, or if a received reply contains errors. When data goes in the other direction LLC receives a PDU from the Presentation Layer together with a channel No. and a destination address. It will add the Telegram Head and CRC-information and activate the *Transmitter Driver* which byte for byte transmits the telegram.

Layer 6: The Presentation Layer

The Presentation Layer will, when handed over a PDU, start processing it, fetching the requested data from - and delivering data to the Application Program data area. Information about where data is located is contained in so called *Data Item Pointer Tables.*

The Application Program can interact with the Presentation Layer through the ***Application Program Interface*** *(API).* In a Master Unit Type services from this interface will be used by the Application Program to initiate the transmission of *Direct Telegrams*, to empty the *Reply Buffer* where Data Replies are delivered and other services to be able to operate on the *Network List*, the *Poll List* and in a certain extend to control the *Auto Poll Cycle*. In a *Slave Unit Type* however, the Application Program will never directly make any use of the protocol (except enabling it), in a sense it is not even aware of its operations. Reception of a *Data Request*, delivering and fetching of data through the API, generation of a *Data Reply* and the transmission of this reply - it all takes place automatically (or in the background so to speak).

All the data processing involved in the interpretation of a received Telegram, the delivering of data, the fetching of data to use in the reply and the controlling of *Bus Mode* (Bus Master, Submaster and Slave) is done in the Presentation Layer. None of these activities are very real time critical. Therefore *GENIpro* perform them under Operation System control.

The Application Program

While bordering on to the transmission medium on the low layer end, *GENIpro* has the Application Program as its border at the high layer end. The application programmer configures via *Compile Time Options GENIpro* to work according to his needs in the application concerned - this is primarily a question of selecting between Slave Unit Type or Master Unit Type.

The *run time control* is done via the services in the ***API***. The Application Program in a Master Unit Type uses the API for the control of units on the bus - contrary to a Slave Unit Type which doesn't use the API for anything but activating and deactivating the communication channels and data class access. The use of the API is described in details in Chapt. 4 and Chapt. 6. Appendix A contains an API reference.

3.2 Data Organization

The communication between bus units can basically be split into *Bus Control* (use of Control Sessions) and *Data Transfer* (use of Data Sessions). Bus Control means the information necessary to make the *Arbitration Mechanism* work. This is already described in Chap 2.3 and 2.4. What remains to be described is what kind of data is communicated and how it is organized.

Everything is communicated in *Telegrams* (*Data Frames*) following a specified syntax: a *Head*, a *Body* and a *CRC-value* (Fig. 3.3 and App. C.1 gives you the best picture of this). The body is called a *Protocol Data Unit (PDU)*, but could as well be given the name Presentation layer Data Unit, because it is processed in this layer. The Head is used solely by the Logical Link Control (LLC) to identify the type of Telegram (Request/Message/Reply coded in the Start Delimiter), the Length, the Destination Address and the Source Address. The CRC-value completing the Telegram makes the LLC able to see if any transmission errors have occurred. The Head and the CRC-value constitute the *Framing* of the telegram. This Framing is stripped of by LLC for incoming Telegrams and added by LLC for Telegrams going out.

The PDU consist of a number of smaller units called an *Application Program Data Unit (APDU)*. While the concept of a PDU is known to both the Presentation Layer and the Data Link Layer, the concept of an APDU is not known by the Data Link Layer. APDU's are a way of further organizing data to a form that complies with how the Application Program - or to be precise the application programmer - finds it natural to organize data. Indeed APDU's are known to the Application Program as the building blocks that makes up the Telegram. Any other details in Data Frame building and handling need not to be known outside the protocol.

To explain what makes up the APDU it is necessary to take a look at how the communication is seen from an Application Program point of view.

The Application Program receives and transmits/delivers data in the form of *Data Items*. A Data Item is the *GENIpro* name for the smallest quantity of information to be handled, f. ex. the measured value of a motor temperature or the command to stop a pump. Data Items are organised in *Classes*.

A Class reflects common characteristics for the Data Items that belongs to it. Each Data Item in a given Class has a name (an identifier) and an *Identification Code (ID-code)*. In this way a Data Item is uniquely defined by its Class number and its ID-code. Figure 3.4 below shows an example with 3 different Classes. App. C gives a complete specification of all the Classes.

A collection of *Operations* to perform on Data Items exist:

***GET*:** to read the value of a Data Item
***SET*:** to write the value of a Data Item
***INFO*:** to get information about the scaling of a Data Item

Each Class defines which of these Operations it is possible to perform on its Data Items.

Class 2: Measured Data	
Operations: GET, INFO	
ID-code	Identifier
26	i_dc
27	v_dc
28	t_e
29	t_m
34	p
37	h
39	q

Class 3: Commands	
Operations: SET, INFO	
ID-code	Identifier
1	RESET
2	RESET_ALARM
3	FACT_BOOT
4	USER_BOOT
5	STOP
6	START
7	REMOTE

Class 5: Reference Values	
Operations: GET, SET, INFO	
ID-code	Identifier
1	ref_rem
2	ref_ir
3	sys_fb_rem
10	sec
11	min
12	hour
13	day

Figure 3.4: *Example of some Classes, some belonging Data Items and the possible Operations*

Now, how does the concept of an *APDU* relate to *Classes*, *Operations* and *Data Items*?

An APDU generally contains a Class specification, a specification of what Operation to perform and a specification of the Data Items to perform it on. For the SET Operation which involves writing a value, this value must also be specified. A Telegram can contain an arbitrary number of APDU's as long as the maximum length of the whole Telegram is not exceeded.

It is allowed to have a telegram without any APDU's, this would lead to an empty PDU - a Telegram made up of framing only. It is also allowed to have empty APDU's. An empty APDU only contains a head which specifies Class and Operation but no body with Data Items.

GENIpro is basically a *byte oriented protocol*. Data is handled in the form of bytes and all ID-Codes and Data Items in the Classes mentioned so far must have values representable as integers from 0 to 255. However, 16 bit data can be handled as exceptions by using separate ID-Codes for high and low order byte (specified in App. C.2).

The organizing of data in Classes makes it possible to define new Classes where the handling of other data formats can be introduced. The *ASCII String Class* (Class no 7) is an example of this. The Data Items in this Class is not byte values but character strings with arbitrary length (App. C.10).

We end this chapter with an example. A detailed and comprehensive specification of the APDU for each of the Classes can be found in Appendix C.

Example of a Telegram

A Master Unit Type wants to request a *GENIbus Slave* for the following:

1) Reading (GET) some Measured Data (Class 2): `p`, `h`, `q` (ID 34, 37, 39)
2) Reading scaling information (INFO) for the same Measured Data.
3) Sending (SET) some Commands (Class 3): REMOTE, START (ID 7, 6)
4) Reading (GET) the Reference Value (Class 5) `ref_rem` (ID 1)
5) Writing (SET) the Reference Value (Class 5) `ref_rem` (ID 1)

Telegram Head

Start Delimiter=0x27 (Data Request)
Length=23 (no of bytes following, excluding CRC)
Destination Address (in this case the slaves address)
Source Address (in this case the masters address)

Data Request PDU

	Class=2
Operation=GET	Length=3
34	
37	
39	
	Class=2
Operation=INFO	Length=3
34	
37	
39	
	Class=3
Operation=SET	Length=2
7	
6	
	Class=5
Operation=GET	Length=1
1	
	Class=5
Operation=SET	Length=2
1	
"value"	

CRC

CRC high order
CRC low order

Telegram Head

Start Delimiter=0x24 (Data Reply)
Length=28 (no of bytes following, excluding CRC)
Destination Address (in this case the masters address)
Source Address (in this case the slaves address)

Data Reply PDU

	Class=2
acknowledge=OK	Length=3
"value of ID 34"	
"value of ID 37"	
"value of ID 39"	
	Class=2
acknowledge=OK	Length=12
"INFO data to ID 34" (1.byte)	
"INFO data to ID 34" (2.byte)	
"INFO data to ID 34" (3.byte)	
"INFO data to ID 34" (4.byte)	
"INFO data to ID 37" (1.byte)	
"INFO data to ID 37" (2.byte)	
"INFO data to ID 37" (3.byte)	
"INFO data to ID 37" (4.byte)	
"INFO data to ID 39" (1.byte)	
"INFO data to ID 39" (2.byte)	
"INFO data to ID 39" (3.byte)	
"INFO data to ID 39" (4.byte)	
	Class=3
acknowledge=OK	Length=0
	Class=5
acknowledge=OK	Length=1
"value of ID 1"	
	Class=5
acknowledge=OK	Length=0

CRC

CRC high order
CRC low order

Figure 3.5: *Example of a Data Request and the corresponding Data Reply. APDU Heads shown shaded. Each line represent a byte value. More detailed Telegram examples can be found in Appendix C.15.*

In figure 3.5 is shown what the Data Request PDU and the Data Reply PDU look like. Some details are worth noticing:

1) Generally there isn't a 1 to 1 byte relationship between Data Request and Data Reply because the INFO operation will return 4 bytes for each ID code if scaling information exists, and the SET operation never returns anything but the APDU Head.
2) APDU Heads always take up 2 bytes. To an APDU in a *Request* there always corresponds an APDU in the Data Reply.
3) The Data Reply is not "self contained", meaning that the Data Reque*st* is necessary to process it.
4) Trying to access Data Items or Classes that does not exist or using Operations not allowed for the Class in question will lead to an error code instead of the acknowledge code "OK" (App. C.13).

3.3 Functional Profiles

For functional devices connected via a network to be interoperable, they must comply with the same standard on

1) Communications medium: For "physical" connection between units
2) Communications protocol: For safe transport of data and control
3) Functional Profiles: For agreement on what data is available and how to use it

The medium for *GENIbus* is a pair of wires connected in a bus topology. The communications protocol is *GENIpro* consisting of software using a RS485 based physical layer.

Now, what implements the Functional Profiles in a *GENIbus* unit?

The Data Items are the *GENIpro* means to concretize Functional Profiles. The static data interface for read (GET, INFO) and partly for write (SET) is implemented by the so called *Bus Unit Tables* written by the application programmer. They declare the Data Item to ID Code relationships and allocate memory for each item (described in details in chapter 5.5). Using the Data Items - and generating them - according to the specification for each Data Item is the responsibility of the Application Program.

Below is a very simple specification of a *GENIbus* Functional Profile for a UPE pump.

It is important to understand, that apart from being specified in the form of Data Items, a functional profile has nothing to do with *GENIpro*. This is the reason why these specifications is placed in their own manual: ***GENIbus*** - Functional Profiles Supplement which is updated each time new profiles are specified.

Action to carry out	Operation	Data Item Name	Class	Data Item ID code
Starting the pump	SET	START	Command, Class 3	6
Stopping the pump	SET	STOP	Command, Class 3	5
Selecting Constant Pressure Mode	SET	CONST_PRESS	Command, Class 3	24
Selecting Const. Frequency Mode	SET	CONST_FREQ	Command, Class 3	22
Adjusting the setpoint	SET	`ref_rem`*)	Reference Values, Class 5	1
Reading the actual pump head	GET	`h`	Measured Data, Class 2	37
Reading the actual pump flow	GET	`q`	Measured Data, Class 2	39
Reading the head scaling	INFO	`h`	Measured Data, Class 2	37
Reading the Operation Mode	GET	`act_mode1`*)	Measured Data, Class 2	81

*) Further specification of these is necessary to use them correct

Table 3.1: *Example of a simple Functional Profile for a UPE pump*

4. Master Management

4.1 General Description

The handling of Bus Master Mode, Submaster Mode and the Arbitration Mechanisms needed for bus control during operation are taken care of by a part of the Presentation Layer called the Master Management module. The purpose of the MM-module is to relieve the application programmer from having to worry about bus control. *GENIpro* ensures that different Master Unit Types developed by different people are always *interoperational*. The MM-module makes it possible for several masters to operate on the same bus. While the Data Link Layer controls the arbitration during Bus Configuration the MM-module controls the arbitration when the bus is operating.

The MM-module makes the bus control invisible to the Application Program. It takes care that the bus is maintained as required by the *GENIpro* specification. All units are regularly polled, new units and disconnected units are recognized. The handling of the necessary shifts in Bus Mode between Slave, Submaster and Bus Master to maintain the bus control and to fulfil what is required from the Application Program is done automatically.

To keep account of the units on the bus, the MM-module maintains a *Network List*. Each bus unit recognized by *GENIpro* is added to the Network List with its own entry called a *Bus Unit Record*. This record contains what *GENIpro* needs of information to service the bus unit concerned. The API offers functions to the Application Program for accessing the Network List.

When a new unit is recognized it is automatically added not only to the Network List but also to the *Poll List*. Units on the Poll List are automatically polled regularly. This so called *Auto Poll Cycle* works "in the background" without the Application Program having to do anything but processing the Data Replies received. The Data Requests used for the Auto Poll Cycle are declared as variables by the Application Program and can thus be changed dynamically during run time. Via API functions the Application Program can control the Auto Poll Cycle, starting and stopping it for example. Also adding or removing units from the Poll List can be done from the API.

The Application Program can do a *Direct Addressing* of any unit or group on the bus. This possibility is necessary to be able to send commands to a specific unit or to update Reference Values and Configuration Parameters with new values. In other words addressing a unit directly with the aim of controlling it. Direct Telegrams can be constants as well as variables. A mixing of variable and constant Telegrams are possible as well, which gives the application programmer the full flexibility to design according to his special needs and memory resources. Powerful function calls exist to make multisession dialogues like the transmission of a Telegram to all members of the Poll List.

The *Network Status Record* together with the *Session Event Register* always contain the result of the last completed Communication Session whether the result of a session came from the Auto Poll Cycle or from a Direct Addressing. These can be accessed by the Application Program and used for direction of received data into storage locations defined by the application programmer.

In the following we shall go into details about the parts of the Master Management which is important for the application programmer to understand. The reader might find it instructive to look at figure D.1.2 in Appendix D while reading. The complete description of return codes and arguments to the functions mentioned and used in the examples can be found in appendix A, The API Reference.

4.2 Auto Poll Cycle

The application programmer working with the design of the more simple Control Master can skip to chapter 4.3.

The Auto Poll Cycle is a facility which the MM-module offers to the application programmer to take care of all Data Requests in the background. When designing a Master Unit Type to operate as a Management Master (see later), let the Auto Poll Cycle handle as much of the data requesting as possible - preferably all of it - to simplify the interactions between *GENIpro* and the Application Program. What is to be requested from which units and how often can be planned prior to compile time or can be changed during run time.

The mechanism in auto polling uses a *Device Type* information to select which Telegrams to use for polling a specific bus unit. F. ex. UPE-pumps are polled with those Telegrams that belong to their Device Type. CU3 control units have a different Device Type and as a consequence they are polled with other Telegrams. This is defined in the so called *Auto Poll Device Table* (`device_tab`) from where *GENIpro* fetches the numbers of the Telegrams to use for auto polling (see figure 4.1). The first row in this two dimensional table consist of the numbers of the telegrams for polling Device Type No. 0. The second row consist of the numbers of the Telegrams for Device Type No. 1 and so on. 255 different Device

Types (rows) can exist, but only three different Telegrams (columns) for each. Numbers in the Device Table reference Telegrams in the *Auto Poll Telegram Table* as shown in the figure. To illustrate how it works three telegram numbers have been filled in for Device Type 2.

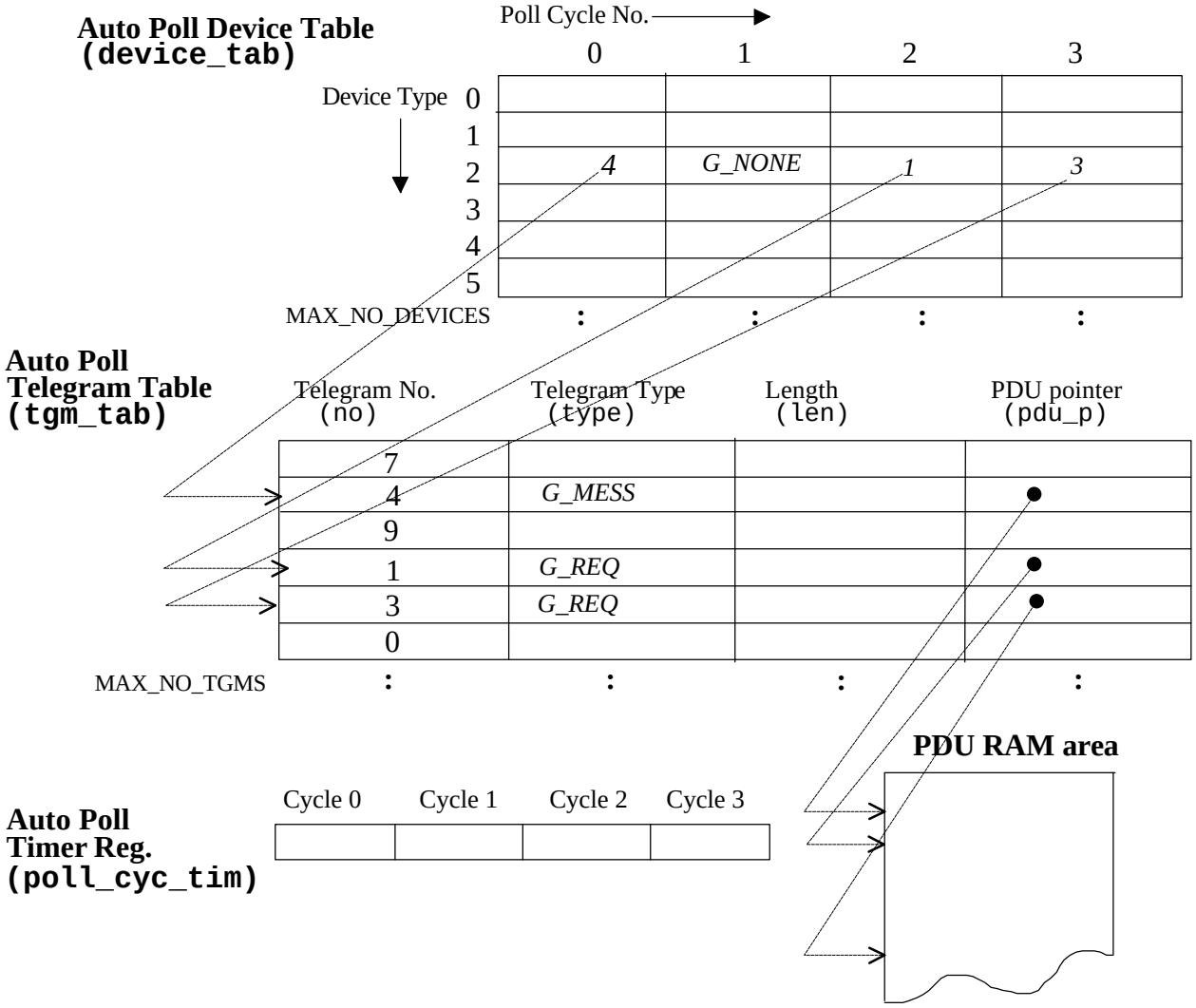


***Figure** 4.1: The Auto Poll mechanism uses the Auto Poll Device Table and the Auto Poll Telegram Table to reference the Telegrams used for auto polling. Both tables are defined by GENIpro and are accessible via the API. The figure shows the relation between them and how the Telegram data field (the PDU's) are fetched from RAM. The Auto Poll Timer Register define the time period for each cycle.*

Each row (entry) in the Telegram Table define a Telegram, which has its body (the PDU) placed in a RAM area allocated by the Application Program with the function call `AllocPduRAM`. Telegram No. 1 and 3 in the example, are both of type `G_REQ` (=request), while Telegram No. 4 is of type `G_MESS` (=message). Only these two types are possible. As seen, the numbering of the Telegrams in the Telegram Table is independent of the table index and is defined by the Application Program.

The four Telegrams for a specific Device Type in the Device Table are used to poll all bus units of this Device Type, but the four Telegrams are not send with the same frequency. The application programmer specifies cycle times in multiples of 0.2s for each via the function call `SetPollCycTime`. An example is shown below:

```
err_code=SetPollCycTime(1, 20, 100, 10000);
```

These values are written to the *Auto Poll Timer Register*. This means that the Poll Cycle No. 0 Telegrams is sent with a period of 1*0.2s = 0.2s. Poll Cycle No. 1 Telegrams is sent with a period of 20*0.2s = 4s and so on. In this way Data Items

which are changing frequently can be requested frequently. Those not changing value very often can be requested very seldom.
Specific Poll Cycles can be disabled by giving the poll cycle times the value G_OFF. The following line of code disables Poll Cycle 0, 1 and 3, but leaves cycle 2 active with a cycle time of 10*0.2s:

```
err_code=SetPollCycTime(G_OFF, G_OFF, 10, G_OFF);
```

When Poll Cycle 0 is disabled like this, the Prompt Telegram (explained later) will automatically be used with a cycle period of 4s. This is a special behaviour related only to Poll Cycle 0 which means that it can not be switched off completely. Another speciality for Poll Cycle 0 is, that a cycle time higher than 4s cannot be specified (it will be rounded down if specified higher). In other words, *GENIpro* uses Poll Cycle 0 as a means to guarantee, that any member of the Poll List is polled at least within 4s. This mechanism frees the application programmer from having to worry about fulfilling special minimum poll time requirements from specific units (like Submasters which relies on frequent SM_ack's).

In the following the necessary sequence of actions from the Application Program for preparing the Auto Poll Cycle will be explained. First, memory for physically storing the Telegrams must be allocated. The Telegrams are stored as *PDU*'s in a RAM area defined by the Application Program but made accessible to *GENIpro* via a pointer:

```
uchar   pdu_ram[1024];                  /* 1 K memory for PDU's             */
err_code = AllocPduRAM(1024, *pdu_ram); /* Give GENIpro a size and a pointer */
```

Next, the Telegrams for auto polling have to be defined. *GENIpro* declares a specific type `PDU_TGM` to use for this. The example below will illustrate how to define two Telegrams and make them accessible to *GENIpro*.

```
/* Declarations */
PDU_TGM   pdu_tgm;
uchar pdu_1[] = {......};
uchar pdu_2[] = {......};

/* Code */
pdu_tgm.no   = 1;          /* Information about Tgm. 1 for the Telegram Table */
pdu_tgm.type = G_REQ;           /* This is a Data Request */
pdu_tgm.len  = sizeof(pdu_1);   /* Length of Telegram data field (PDU)      */
pdu_tgm.ptr  = pdu_1;           /* Pointer to this PDU                      */

err_code = StoreTgm(&pdu_tgm);  /* Add the Telegram to the Telegram Table   */

pdu_tgm.no   = 2;          /* Information about Tgm. 2 for the Telegram Table */
pdu_tgm.type = G_REQ;
pdu_tgm.len  = sizeof(pdu2);
pdu_tgm.ptr  = pdu_2;

err_code = StoreTgm(&pdu_tgm);
```

Each Telegram now has a reference in the Telegram Table and its PDU is stored in PDU RAM. The Telegram Table will have the contents shown below.

No.	Type	Length	PDU pointer
1	G_REQ	sizeof (pdu_1)	pdu_1
2	G_REQ	sizeof (pdu_2)	pdu_2

Storing a Telegram with a number already existing will overwrite the old Telegram. Before polling can begin, the Auto Poll Device Table has to be completed. It is automatically initialised with G_NONE, meaning that the application programmer has only to fill out those entries which he wants to use. Say we want UPE devices to be polled with Tgm. No. 1 in poll cycle No. 1 and CU3's with Tgm. No. 2 in poll cycle 0:

```
#define        UPE_DEV_NO  1   /* Device No. then equals the unit family code */
#define        CU3_DEV_NO  3   /* Device No. then equals the unit family code */

err_code=EditDevTab(UPE_DEV_NO, G_NONE, 1, G_NONE, G_NONE);
err_code=EditDevTab(CU3_DEV_NO, 2, G_NONE, G_NONE, G_NONE);
```

GENIpro assigns the Telegram Numbers to the UPE device entry and the CU3 device entry in the Device Poll Table, which will look like this after the function call:

	No. of Poll Cycle 0 Telegram's	No. of Poll Cycle 1 Telegram's	No. of Poll Cycle 2 Telegram's	No. of Poll Cycle 3 Telegram's
Device No. 0 Tgm's	G_NONE	G_NONE	G_NONE	G_NONE
Device No. 1 Tgm's	G_NONE	1	G_NONE	G_NONE
Device No. 2 Tgm's	G_NONE	G_NONE	G_NONE	G_NONE
Device No. 3 Tgm's	2	G_NONE	G_NONE	G_NONE
Device No. 4 Tgm's	G_NONE	G_NONE	G_NONE	G_NONE

Attempts to write a Telegram number to the Auto Poll Device Table which has not previously been stored in the Telegram Table will fail and lead to the return of the error code G_NOT_FOUND.

Finally we will set the Poll Cycle Timers. If Poll Cycle 0 is to take place every second, Poll Cycle 1 every 10 seconds and Poll Cycle 2 and 3 should not take place at all, this is how to proceed:

```
err_code=SetPollCycTime(5, 50, G_NONE, G_NONE); /* Times in multiple of 0.2s */
```

It is possible to specify poll cycle times which are so short that they cannot be fulfilled if the No. of units on the bus becomes high. The time period of the poll cycle will then automatically be prolonged accordingly. An example of that is shown below. The time period for Poll Cycle 0 is set to 1s. It can be seen that this can be fulfilled as long as there are less than 4 units on the bus.

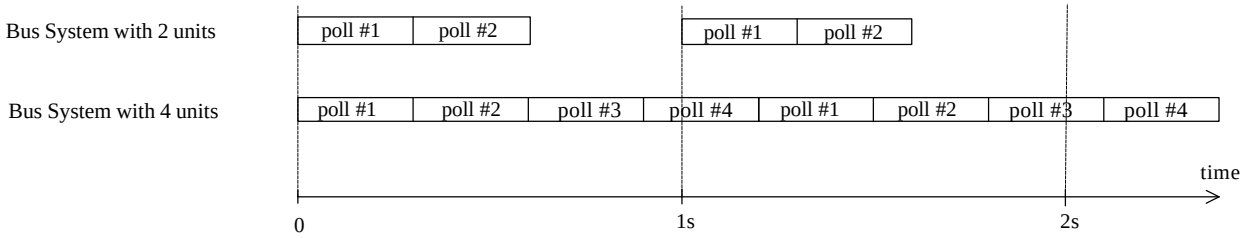


Figure 4.2: *Automatic prolonging of a Poll Cycle time period when it is set too short compared with the No. of units on the bus.*

When *GENIpro* recognize a new unit the value of `unit_family` will automatically be used as the Device Type. (see also Chapt 4.4). This is the reason why it is a good idea to make the Device No. match the Unit Family code if this is known in advance. In the example this means that when a UPE or CU3 is recognized, the Auto Poll Cycle will immediately start polling them with the correct Telegrams without the Application Program having to interfere.

Upon initialization of *GENIpro* the Auto Poll Cycle is enabled and will automatically start if the unit becomes Bus Master. The Application Program can stop the auto polling with the function call:

```
err_code=AutoPollCtr(G_STOP_AUTO_POLL);
```

In this case the Bus Control is no longer taken care of automatically. All Communication Sessions must be handled explicitly from the Application Program in the form of *Direct Telegrams* (se Chapt. 4.3). Incompatibility with other masters is thus very likely to result. So, generally avoid stopping the Auto Poll Cycle for anything but very exceptional cases. To restart auto polling use:

```
err_code=AutoPollCtr(G_START_AUTO_POLL);
```

The Device Poll Table can be reset, meaning filled with values G_OFF, by the usage of the function call

```
ResetDevTab();
```

The Telegram Table can also be reset with a function call:

```
ResetTgmTab();
```

Because Telegram Numbers in the Auto Poll Device Table makes no sense with an empty Telegram Table, this function call also resets the Auto Poll Device Table.

Apart from polling units in the Poll List, the Auto Poll Cycle also recognizes new units on the bus by the use of the Connection Request/Reply Mechanism. It reports the event `G_NEW_UNIT` in the Session Event Register (`ses_event`) when this happens. In case a recognized unit has an unknown Device Type (see Chapt 4.4) the reported event will be `G_UNKNOWN_DEV`. In both cases this is a message to the Application Program that a new unit has been added to the Network List and to the Poll List and will now be serviced by the Auto Poll Cycle. The characteristics of this new unit can be requested with the API function `ReadNetwStatus`.

The Auto Poll Cycle also supervises the error rate for all units. In this way units that are no longer connected to the bus and units having a malfunctioning bus communication are removed from the Network List. When a unit is removed like that, the event `G_UNIT_REMOVED` is reported.

4.3 Direct Telegrams

While the Auto Poll Cycle typically takes care of transferring data from the different bus units to the Bus Master, the Direct Addressing is intended to transfer data the other way - from the Bus Master to the bus unit with the purpose of control. Direct Telegrams must in the general case be modifiable during run time. Normally the Data Items to transfer are the same, only their values change. This can be utilised to simplify the handling of Direct Addressing.

Below is an example of a Direct Telegram transferring two Commands and one Reference Value to a UPE pump.

```
uchar upe_ctr_pdu[]=
    {
    0x03,                /* Class No. 3 = Commands                */
    0x82,                /* Operation=SET, APDU length=2          */
    0,                   /* Placeholder for Command No. 1         */
    0,                   /* Placeholder for Command No. 2         */
    0x05,                /* Class No. 5 = Reference Values        */
    0x82,                /* Operation=SET, APDU length=2          */
    1,                   /* ID code for Remote Reference (ref_rem) */
    0,                   /* Placeholder for ref_rem value         */
    }
#define UPE_cmd1         upe_ctr_pdu[2]
#define UPE_cmd2         upe_ctr_pdu[3]
#define UPE_setpoint     upe_ctr_pdu[7]
```

Once `upe_ctr_pdu` is declared and initialized the Application Program needs only to execute the following code to take control over the pump with address 0x20, start it and give it a setpoint:

```
UPE_cmd1         = REMOTE;
UPE_cmd2         = START;
UPE_setpoint     = <setpoint>;
err_code=SendTgm(0x20, G_REQ, sizeof(upe_ctr_pdu), (void*)upe_ctr_pdu);
```

In this example the Telegram is send as a *variable request* (argument `G_REQ` specifies that the Telegram should be sent as a request) - meaning that a Data Request changeable during run time is used. To support those applications, where Direct Addressing is only a matter of a very few different Telegrams, which could be completely defined at compile time, the option of using *constant* requests or messages is available. This is possible because the pointer to the PDU is a *void pointer*.

If for example the only thing that needs to be done directly is starting and stopping a pump, the PDU's for doing this might look like this:

```
const uchar stop_pdu[] =
      {
      0x03,  /* Class=3              */
      0x81,  /* SET + APDU length=1*/
      0x05   /* Stop command ID     */
      }
```

```
const uchar start_pdu[] =
      {
      0x03,  /* Class=3              */
      0x81,  /* SET + APDU length=1*/
      0x06   /* Start command ID    */
      }
```

Now, the stopping of the pump with address 0x20 without the need for a reply is as simple as this:

```
err_code = SendTgm(0x20,  G_MESS, sizeof(stop_pdu), (void*)stop_pdu);
```

4.4 The Network List

The application programmer working with the design of the more simple Control Master can skip to chapter 4.5.

The Network List is an array of *Bus Unit Records*, one record for each bus unit recognized. It is used by a Bus Master to keep account of units connected to the bus, and contains enough information for the Bus Master to handle the bus control and the data transfer:

```
typedef struct
   {
   uchar unit_addr;     /* The Unit Address                                  */
   uchar unit_descr;    /* Group Address and Bus Unit Type                   */
                        /* bit 3-0: group_addr - 232                         */
                        /* bit 4:   CTO_BUS_UNIT_TYPE                        */
                        /*          0: Slave Unit Type                       */
                        /*          1: Master Unit Type                      */
                        /* bit 5:   CTO_MM                                   */
                        /*          0: Control Master                        */
                        /*          1: Management Master                     */
   ushort unit_err_rate;/* Error rate for this bus unit. When this value     */
                        /* exceeds an upper limit, the unit is               */
                        /* automatically removed from the Network List if    */
                        /* this facility is enabled                          */
   uchar poll_list_ctr; /* Controls the Poll List                            */
                        /* bit 0: 0: The unit is not on the Poll List        */
                        /*        1: The unit is on the Poll List            */
   uchar unit_family;   /* Unit family code                                  */
   uchar unit_type;     /* Unit type code                                    */
   uchar dev_type;      /* Device type code                                  */
   } BUS_UNIT_RECORD;

BUS_UNIT_RECORD  network_list[32]; /*A Network List with records for 32 units*/
```

The Poll List is not a real list, but only a flag (`poll_list_ctr` bit 0) in the Bus Unit Record. It has the value "true" for those units in the Network List which are members.
There are two ways a bus unit can become a member of the Network List. First of all by replying to a Connection Request. The Connection Reply contains all information necessary to make a Bus Unit Record for it, and it automatically becomes a member of both the Network List and the Poll List.
Next, the Application Program can explicitly add a unit to the Network List by the function call

```
err_code = EditNetwList(addr; G_ADD);
```

In this case it is not automatically added to the Poll List as well. To obtain this, execute the function call

```
err_code = EditPollList(G_ADDR; addr; G_ADD);
```

The Bus Unit Record for a unit added in this way is incomplete. To solve this problem *GENIpro* polls the unit with the *Bus Unit Record Telegram* once before its first regular turn in the Poll Cycle. The BUR-Telegram is identical with a Connection Request except that it addresses a Unit Address instead of the Connection Address. The reply to the BUR-Telegram is completely identical with the Connection Reply and returns the values of

`unit_bus_mode, unit_family, unit_type,` and `group_addr`

The Device Type (`dev_type`) is always set equal to `unit_family` by *GENIpro* when a Connection Reply is received. The Application Program can change this to a user defined Device Type if desired:

```
err_code = SetDevType(addr; dev_type);
```

Recognizing a new unit will make *GENIpro* write the event G_NEW_UNIT in the Session Event Register. If a recognized unit has a `unit_family` with a value higher than the No. of entries in the Device Poll Table (MAX_DEV_NO) *GENIpro* assigns the unit to Device Type 0 and writes the event G_UNKNOWN_DEV in the Session Event register instead.

More about Device Types are explained in chapter 4.6.

4.5 Processing a Reply

Each time *GENIpro* completes a Communication Session a data structure called the Network Status Record is updated. It has the following definition:

```
typedef struct
   {
   uchar unit_addr;          /*[0;231]: Unit Address for the unit from which */
                             /*          the latest reply came                */
   uchar unit_descr;         /* Group Address and Bus Unit Type               */
                             /* bit 3-0: group_addr - 232                     */
                             /* bit 4:   CTO_BUS_UNIT_TYPE                    */
                             /*          0: Slave Unit Type                   */
                             /*          1: Master Unit Type                  */
                             /* bit 5:   CTO_MM                               */
                             /*          0: Control Master                    */
                             /*          1: Management Master                 */
   uchar unit_family;        /* Unit family code                              */
   uchar unit_type;          /* Unit type code                                */
   uchar dev_type;           /*[1;254]: Device Type                          */
   uchar pdu_tgm_no;         /*[0;254]: Telegram NO. used for polling in     */
                             /*         the Auto Poll Cycle.                 */
                             /*G_DIR_REP: Received reply is for a Direct     */
                             /*           Telegram                           */
   uchar no_of_units;        /*No. of units in the Network List              */
   uchar bus_err_rate;       /*Total error rate on the bus                   */
   } NETW_STATUS;
```

The structure is accessed via the API-function call `ReadNetwStatus` which reads the content into a user defined variable with the same type:

```
NETW_STATUS     status;
err_code=ReadNetwStatus(&status);
```

The Session Event Register `ses_event` is also updated when a Communication Session completes. It contains an event describing the result of the session. This is at the same time information to the Application Program about what to do, to process a possible Data Reply. We will look at two simple examples:

Example 1. : Handling of Twin Pump Control

A UPE twin pump (TP) is a Control Master application (see Chapt. 4.7). It constitutes a small Subnetwork where the twin pump master is a Submaster and the twin pump slave is a Subslave. The TP-master Application Program should concentrate its bus activities on controlling its TP-slave, with one important exception. Let us assume a TP which becomes Bus Master has to send the Command "LOCAL" to all bus units which are not Subslaves. Such a requirement is dictated by the application so the Application Program must handle it by use of the API. This is what the example is about. The easiest way to implement this behaviour is to send the LOCAL command to each unit on the bus as soon as it is recognized by *GENIpro* (replies to a Connection Request). This will at the same time automatically exclude Subslaves. Here is the code:

```
NETW_STATUS     status;               /*For user copy of Network Status Record*/
local_pdu[]={0x03, 0x03,0x81, 0x08};  /*LOCAL command PDU                      */

if (ses_event)
   {
   err_code=ReadNetwStatus(&status);
   switch (ses_event)
        {
        ...
        ...
        case G_NEW_UNIT: err_code=SendTgm(status.unit_addr, G_MESS, G_CONST,
                                          sizeof(local_pdu), (void*)&local_pdu);
             break;
        ...
        }
   ses_event=G_NO_EVENT;
   }
```

Example 2.: Pump Management Unit (PMU)

A PMU for controlling pumps with *GENIbus* interface is a Management Master application. Data requesting is done as much as possible with the Auto Poll Cycle and control is exerted with Direct Telegrams. In both cases Data Replies will be delivered in `data_reply_buf`, and `ses_event` will contain the event `G_NEW_TGM`.

The example code below shows a very simple structure to process Data Replies. It illustrates the use of the Network Status Record to determine the type of the reply and from which unit it came. What to do with the reply to a Direct Telegram is of course highly dependent of the application and the content of the request. The reply to an Auto Poll Cycle Telegram on the other hand will generally have to be processed by just copying the data content into a Device Object for the bus unit in question. This is what the Application Program function `Store_obj` is supposed to do. It uses the Unit Address of the replying unit and the cycle No. of the Auto Poll Cycle as arguments to get information to tell in which Device Object and how to store the received data.

```
/* Declarations */
NETW_STATUS     status;                    /* For user copy of Network         */

/* Code */
if (ses_event)
   {
   err_code=ReadNetwStatus(*status);     /* Get copy of Network Status        */
   switch (ses_event)
        {
        ...
        ...
        case G_DIR_REPLY:         /*The Reply is for last call to SendTgm      */
                {
                ...
                }
             break;
        case G_AUTO_REPLY:     /*The Reply is for an Auto Poll Cycle Telegram */
                {
                Store_obj(status.unit_addr, status.pdu_tgm_no);
                }
             break;
        ...
        }
   ses_event=G_NO_EVENT;
   }
```

In case the application program hasn't stored a copy of all Telegrams stored in the Telegram Table, a pointer to the PDU belonging to a specific telegram number `(pdu_tgm_no)` can be requested with the following function call:

```
err_code = GetPDUptr(pdu_tgm_no, &pdu_ptr);
```

4.6 Device Types and Device Objects

As mentioned earlier the value of `dev_type` in the Bus Unit Record controls which Telegrams from the Telegram Pointer Table are used by the Auto Poll Cycle to poll this unit. As it has already been described, the application programmer designs these Telegrams himself to supply the data necessary for the Device Type concerned. Also the design of the Device Types (file `dev_typ.h`) is the responsibility of the application programmer. Example files are however delivered together with the *GENIpro* source code files. These example files will in most cases only need small modifications to suit an application.

Below are two simple examples of Device Types.

```
typedef struct
       {
       uchar unit_addr;
       uchar group_addr;
       uchar unit_family;
       uchar unit_type;
       uchar p;
       uchar h;
       uchar q;
       uchar p_range;
       uchar h_range;
       uchar q_range;
       } UPE_DEVICE;
```

```
typedef struct
       {
       uchar unit_addr;
       uchar group_addr;
       uchar unit_family;
       uchar unit_type;
       uchar v_rst;
       uchar i_rst;
       uchar p_hi;
       uchar p_lo;
       uchar v_range;
       uchar i_range;
       uchar p_range;
       } CU3_DEVICE;
```

Device Types are used for the declaration of Device Objects. A Pump Management Unit capable of handling 8 UPE pumps would use the following line of code to declare a Device Object for each of them:

```
UPE_DEVICE      upe[8];
```

Now, when a Data Reply is received from a bus unit, the Application Program must move the content from the Data Reply Buffer into the Device Object representing the unit. This is illustrated in the figure below.

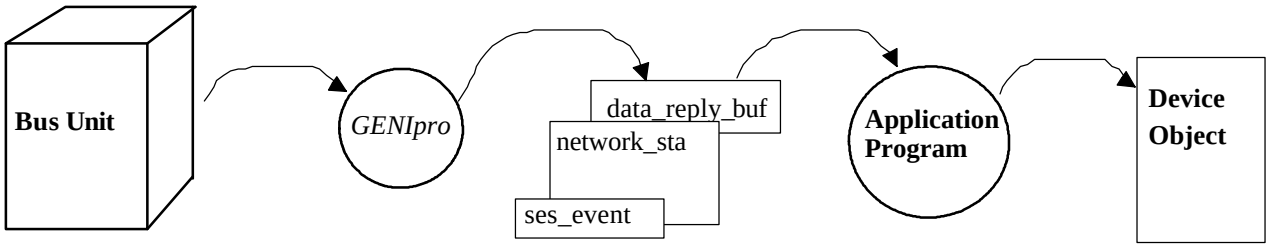


***Figure** 4.2: A Device Object as a data model of a real bus unit. The Application Program must move data from the Data Reply Buffer into the Device Object in question. `network_sta` and `ses_event` are used to inform from which unit the reply came and what Telegram have been used for polling. must be requested via the API function ReadNetwStatus.*

4.7 Control Master and Management Master

The full implementation of the MM module is called a *Management Master*. This requires the following Compile Time Option which gives access to the full functionality and API described in the foregoing sections:

```
#define   CTO_MM     Man_Mast
```

A Management Master is intended to work in Bus Master Mode and be the one in charge of the bus. The typical appliance is therefore devices designed for control and supervision of several units on a whole bus system.

The Master Management module can be implemented in a reduced version with the Compile Time Option

```
#define   CTO_MM     Ctr_Mast
```

This reduced implementation is called a *Control Master*. A great deal of code and memory saving is thus possible. However, the most important argument for a Control Master is, that its reduced functionality and API exactly covers the requirements for those appliances working as a Submaster. Still it fulfils the *GENIpro* specification for Master Unit Types.

Any Master Unit Type can be appointed to Bus Master either by the Bus Configuration or directly by a resigning Bus Master. Besides, it always requests to be Bus Master if it has a lower address than the current Bus Master. This last rule implies, that on a *GENIbus System* the Master Unit Type with the lowest address will always end up being the Bus Master (a Management Master in Monitor Mode is an exception) .

The table below summerizes the Arbitration Mechanisms and the differences between the two Master Unit Types.

	Control Master	Management Master
Application examples	CU3, UPE-twinpumps The typical appliance to operate in Submaster Mode	PMU, Modem Master, Gateway Master The typical appliance to operate in Bus Master Mode
Reduced functionality	1. Auto Poll Cycle only with BUR-Telegram (no poll_tab.c to fill out) 2. No API Control of Auto Poll Cycle 3. No Poll List Functions 4. No Network List Functions 5. Reduced Bus Unit Record in Network List	
Attaining Bus Master Status	1. Appointment by Bus Configuration 2. Direct appointment 3. Requests to be Bus Master if Unit Address is lower than the address of the current Bus Master	1. Appointment by Bus Configuration 2. Direct appointment 3. Requests to be Bus Master if Unit Address is lower than the address of the current Bus Master
Acknowledge of Submaster Request	always immediately	always immediately
Acknowledge of Bus Master Request	always immediately	always immediately
Requests to be Submaster	if the Application Program makes use of Direct Addressing (servicing a Subnet).	Never

***Table** 4.1: Comparisons between Management Master and Control Master. All fundamental mechanisms are handled automatically by GENIpro. This assures interoperability between masters.*

4.8 Bus Master Priority Mechanism

The application programmer working with the design of the more simple Control Master can skip the rest of chapter 4.

As already mentioned and briefly discussed in chapter 2.3 the Bus Master priority is connected to the Unit Address: *a lower Unit Address means a higher priority*.

The Normal (default) way *GENIpro* manages the Bus Master Status is:

1) A Master Unit Type being polled by a Bus Master having a lower Unit Address will to its reply append an RFS option requesting to be Bus Master (BM_rfs).

2) A BM_rfs is always acknowledged (BM_ack) irrespective of address values

This is shown in table 4.2. The default priority mechanism can be changed. With the function call:

```
err_code = Mm_Ctr(G_HIGH_PRIO);
```

the unit is brought into a high Bus Master Priority State. In this state it will use BM_rfs to request Bus Master Status from any unit polling it. Having attained the Bus Master Status it will never give it away again (refuse BM_ack).

Leaving the normal Bus Master Priority State should only be used as an exception in very special situations. The reentering of the normal Bus Master Priority State is done with:

```
err_code = Mm_Ctr(G_NORM_PRIO);
```

	If the requested unit has a lower Unit Address than the requesting unit	Always	Never
BM_rfs	NORM_PRIO	HIGH_PRIO	
BM_ack		NORM_PRIO	HIGH_PRIO

***Table 4.2**: Bus Master Management according to Bus Master priority*

4.9 Operating as a Monitor (Not implemented in *GENIpro* v2.0)

A Management Master can be switched into *Monitor Mode* by use of the API-function call:

```
err_code = Mm_Ctr(G_MONITOR);
```

This can be done irrespective of which Bus Unit Mode the Management Master is currently operating in: Slave Mode or Bus Master Mode.

When switched into Monitor Mode the normal Arbitration Mechanisms in *GENIpro* are suspended. The unit is turned into permanently passiveness: it can not be appointed to Bus Master neither by the Bus Configuration nor directly by another unit. As seen from the bus it is no longer present - it does not reply when addressed. Internally the behaviour is also special.

GENIpro will copy all sound Telegrams on the bus to API-buffers, to enable the Application Program to monitor all bus activity. Data Requests and Data Messages will be placed in `data_request_buf`, and Data Replies will as usual be placed in `data_reply_buf`. The format is a PDU with a length specifier.

The event `NEW_TGM` is written to the `ses_event` register (and the `Com_ses_end` semaphore is updated if used) each time a Data Reply or a Data Message arrives. The Application Program can test for the `NEW_TGM` event and a request/reply pair or a single message will then be ready for processing.

To be able to use Monitor Mode the application programmer must write code for this processing. This is much more involved than processing Data Replies as a normal Management Master because the number of different Data Requests which can occur is not limited. This means that general functions processing each type of reply APDU from its belonging request APDU is needed.

5. *GENIpro* in an Application

5.1 The *GENIpro* Interfaces

GENIpro interacts with its environment through 3 interfaces:

- The Application Program Interface (API)
- The Operating System Interface (OSI)
- The Driver Interface (DI)

and via a direct access to:

- Bus Unit Tables
- Data Items (via pointers in the Bus Unit Tables).

This is illustrated with figure 5.1.

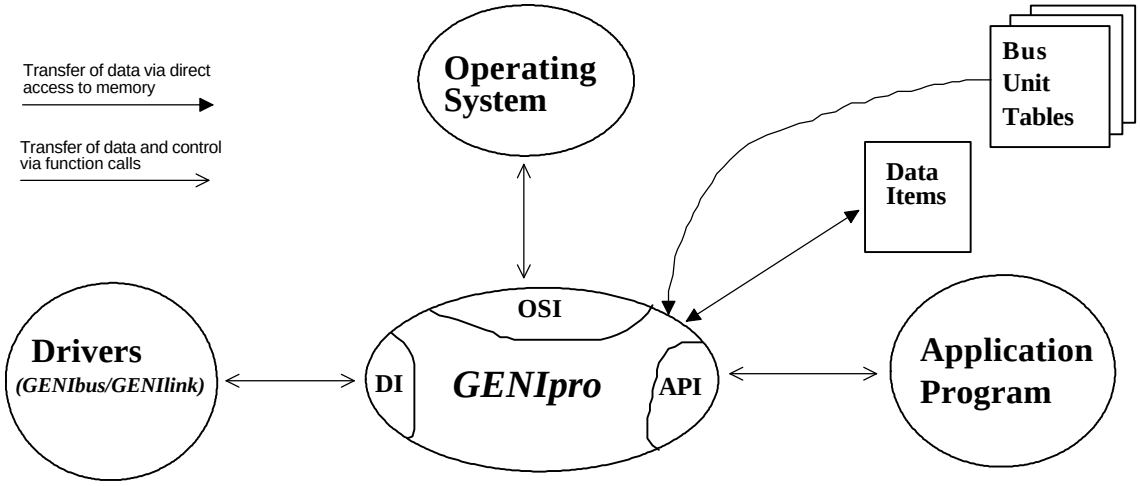


Figure 5.1:*The GENIpro interactions with the environement.*

The Application Program Interface (**API**) consists of the exchange of data, events and commands via function calls, API Receive Buffers, and shared variables. This interface can be thought of as a collection of services offered by *GENIpro* to the Application Program. *GENIpro*, therefore, does not impose special implementation dependent requirements on the Application Program for the use of the API. This use can be considered as target and real time independent except for the requirement that the Application Program in a Management Master must process one Data Reply before a new one can arrive - not to lose data. A whole chapter (Chapt. 6) is devoted to an exemplified description of how to use the API services, and Appendix A gives a short form reference.

With the Operating System Interface (**OSI**) the situation is the opposite. *GENIpro* demands certain timing services to be available via Operating System calls and the occurrence of certain realtime timing events for internal synchronization. By using the Grundfos RTOS (RTOS78K for µPD780xx /µPD783xx and PC_RTOS for PC's) these services are ready for use and chapter 5.2 describes how to configure RTOS. If another Operating System is used, it must still fulfil the same OSI demands. These are documented in Appendix D.4, where different design solutions are described as well.

The Driver Interface (**DI**) transfers data and commands from *GENIpro* to the Drivers and data and events from the Drivers to *GENIpro*. This is exclusively done via function calls. The setup of a Driver module is done in the belonging header file. Drivers exist for PC, µPD780xx and µPD783xx. Specifications for the implementation of Drivers for other targets can be found in Appendix D.3.

The **Bus Unit Tables** contain pointers to the protocol **Data Items** which are specified according to the Functional Profiles for the unit. These tables are necessary for *GENIpro* to be able to perform the data operations SET, GET and INFO. This access is direct, without the Application Program having to do anything or even being aware of it taking place. Bus Unit Tables are made prior to compilation. Chapter 5.5 describes the procedure.

5.2 Configuring *RTOS* to Work With *GENIpro*

The data handling resulting from the processing of a complete *PDU*, when it has been received and buffered, is not so time critical as the treatment of single bytes during communication. For that reason it is done under *Operating System* control, which not only makes the implementation hardware independent but also removes the possibility of realtime conflicts.

This means, that although the *GENIpro* specification is quite general and completely independent of target system and programming language, the implementation of *GENIpro*, as delivered in source files, rely on special services from an Operating System. *GENIpro* is not inextricably bound with the Grundfos operating system ***RTOS*** *(PC_RTOS* for PC applications and *RTOS78K* for the NEC µPD780xx or µPD783xx computer families) although it is nicely designed for it. In other words using RTOS will make life a lot easier for the application programmer (refer to App. D.4 if another Operating System is used).

Notice that the application programmer is not completely free in setting up the RTOS when using *GENIpro*. Study the restrictions carefully as well as the restrictions to the Application Program described in the following chapter and take them into consideration when designing your complete system.

This description tells what setup in the **`rtos_cnf.h`** file is needed to make *GENIpro* work with *RTOS*. The setup of parameters having no influence on *GENIpro* is not described (refer to the RTOS78K Manual).

1) **`T_inc`** is chosen to have a value of 1ms or less.

2) **`T_tick`** is chosen in the interval [10ms; 40ms]

3) *GENIpro* uses `soft_timer0` for some of its timekeeping in the *Data Link Layer*
   a) define `soft_timer0` to be counting in 16 bits and to increment in steps of *T_inc:*

```
#define soft_timer0   T16
```

   b) Attach the *LLC timer interrupt routine* to the timeout of `soft_timer0` and declare it external:

```
extern void llc_tim_irq(void);
#define  Soft_timer0_irq    llc_tim_irq();
```

4) a) *GENIpro* contains two Operating System controlled tasks. Define a number for the two tasks in *GENIpro*. The numbers can be chosen arbitrarily but the names of the tasks are reserved and must be written as shown here:

```
#define Geni_task     0
#define Llc_tim_task  1
```

   b) Make the two tasks known to the operating system (the numbering must be consistent with 4.a above):

```
#define Task0         geni_task
#define Task1         llc_tim_task
```

   These tasks have no initializing functions. Initializing of *GENIpro* is done via *API Services*, giving the Application Program a much better control of the protocol (see Chapt. 6).

5) Define the semaphore for *GENIpro* to use when signalling the end of a Communication Session (the number can be chosen at will, but the name is fixed):

```
#define Com_ses_end   Bit0
```

6) Some target computer Bus Drivers (f.ex. the one for ***mPD783xx***) give you the option of using Bus Idle Detection to be based on polling (this is not recommended but might be your only choise if an external interrupt pin is not available). The RTOS user interrupt rutine `User_irq` is a software interrupt routine executed frequently enough to accomplish this. Add to the configuration file:

```
extern  void bus_idle_poll(void);
#define User_irq   bus_idle_poll()
```

7) Using the *GENIlink* channel will normally imply that the RTOS timer interrupt routine has to be interruptable itself:

```
#define Interruptable_rtos       ON
```

5.3 Restrictions to the Application Program

As described in chapter 3 *GENIpro* is driven by interrupts. Each time a byte is received or transmitted on the *GENIbus* channel, or a bit pair is received or transmitted on the *GENIlink* channel, a software *Driver* is invoked by interrupt. The Drivers use the services of the *Logical Link Control* to deliver data to the *Receive Buffer* or to fetch data from the *Transmit Buffer*.

These communication driven interrupts must be able to coexist with the Application Program without resource and timing conflicts. How this is handled depends mostly of what functionality the hardware architecture offers. It is a matter of writing efficient Drivers which can use this functionality to its full extent, thereby reducing the unavoidable restrictions it puts on the Application Program. But some restrictions remain.

The *GENIbus* Drivers are always *interruptable* to allow for real time critical interrupt routines in the Application Program. On the other hand, the *GENIlink* Drivers are normally *none interruptable* because they are themself extremely real time critical. This is the reality which to comply with.

The following description is made with special reference to an integration on a ***mPD780xx*** or ***mPD783xx*** target computer, but the principles are of course general.

Restrictions to *Application Interrupt Routines* (*AIR*'s)

1) All *AIR*'s must be interruptable by the *GENIlink* interrupt driven Drivers if this channel is used. The code line

   ```
   enable_interrupt();
   ```

   must be inserted in the top of all *AIR*'s.

2) All *AIR*'s must normally have "low interrupt priority". If some *AIR*'s are to be protected against interruption by other *AIR*'s, dynamically changing of interrupt priority can be used on targets having this facility. Add the following code lines to the top of the *AIR* to be protected.:

   ```
   "code to set interrupt priority high";
   enable_interrupt();
   ```

   This, at the same time, prevents the *GENIbus* interrupt from breaking in, whereas the *GENIlink* interrupt with its top priority is not excluded.
   The interrupt priority is for ***mPD780xx*** or ***mPD783xx*** automatically restored to its original value (low) when the *AIR* returns.

3) Due to its ability to interrupt the Bus Driver interrupt routines, *AIR's* must leave enough execution time left for the Bus Driver to be able to process bytes arriving at a frequency of one pr millisecond. For a typical 8 bit controller like ***mPD780xx*** with a clock frequency of 10MHz, this means an AIR execution time well below 0.7ms. If more *AIR* events can occur simultaneously (within 1ms) this goes for the sum of them.

4) Any *AIR* must be resistant to an interrupt latency of approximately 10µs resulting from certain critical regions and system calls in the operating system where uninterruptability is deliberately secured. When a *GENIlink* Driver interrupt routine is active, this delay can be in the order of 50 - 200µs, because the *GENIlink* Drivers are uninterruptable.

5) Any *AIR* must be resistant to IR Driver interrupts stealing 50 - 200µs with a frequency of approximately 2kHz, when the *GENIlink* channel is enabled and active.

6) Any *AIR* must, when the interrupt priority is not set to high (item No. 2 above), be resistant to Bus Driver interrupts stealing 100 - 200µs with a frequency of approximately 1kHz, when the *GENIbus* channel is enabled and active.

Restrictions to Application Tasks

1) To fulfil the *GENIpro* specification of a maximum reply time, no application task may have an execution time exceeding 40 ms.

5.4 Configuring *GENIpro* to the Application Program

Configuration of *GENIpro* is done in the **geni_cnf.h** file by specifying the value of *definements*. In this chapter a short explanation is given to them all. Besides the configuration described here a target dependent configuration must be done in the Driver Header File for the selected driver modules.

Compile Time Options (CTO's)

The CTO's control the code generated during compilation.

Now choose which of the *GENIpro* communications channels to use. They can be enabled or disabled independently:

```
#define   CTO_BUS      Enable | Disable
#define   CTO_IR       Enable | Disable
```

When a channel is disabled via the CTO the code for its *Drivers*, will not be included in the compilation. Therefore the channel can never be activated - neither by an external event (a telegram arriving) nor by any internal event (API Service call). It is safe to let its hardware resources be used for something else.

Next, the application programmer must specify whether his *Bus Unit* is of the *Master Unit Type* or the *Slave Unit Type*

```
#define  CTO_BUS_UNIT_TYPE    Master | Slave
```

A Master Unit Type has an extended set of API functions at its disposal (see Chapt 5), a Slave Unit Type only a minimum set of API functions but with a significant code saving. Choosing a Master Unit Type implies the use of a Master Management module. Two different masters can be build: a *Control Master* (module **ctr_mast.c** is added to the project) or a *Management Master* (module **man_mast.c** is added to the project). Besides the following CTO specifies to the rest of the *GENIpro* modules which Master Management module has been chosen:

```
#define  CTO_MM       Ctr_mast | Man_mast
```

Management Master is for the application which typically works as Bus Master while Control Master is for the application which typically works as a Submaster. Chapter 5 goes deeply into these details.

By default, *GENIpro* delivers commands in the `cmd_buf[]` while Reference Values and Configuration Parameters are written directly to memory. A CTO controls whether the SET operation for the Classes 4 and 5 should work the default way or they should be *customized*:

```
#define  CTO_BUF_OPT      Enable | Disable
```

If *Disable* is selected, the default way will be used. If *Enable* is selected the way specified in the *Customizing Definement* `Buf_opt_ctr` (see below) will be used.

Optimizing and Target Compiler Options

The CRC generation in *GENIpro* can be optimized for high speed or for low memory consumption. Optimizing for memory will save approximately 500 bytes of ROM but will double the CRC execution time (typically from 40*m*s to 80*m*s for a *mPD780xx*)

```
#define    CRC_OPTIMIZE      speed | memory
```

If your target compiler supports the use of short address variables to obtain faster instructions, specify the Short Address Modifier here. Otherwise leave the definement empty. *GENIpro* takes up 4 short address locations if selected.

```
#define    SH_ADDR                          //Short Address Modifier
```

You must specify how your compiler store integers. Either they are stored with the low order byte in the lowest memory address (LO_HI) or they are stored with the high order byte in the lowest memory address (HI_LO).

```
#define    INT_STORAGE       LO_HI | HI_LO
```

Customizing Definements

The first of these specifies the buffer options for Class 4 and 5, the two *API Receive Buffers* which are optional. This definement is only active if `CTO_BUF_OPT` has been enabled:

```
#define  Buf_opt_ctr      "special syntax"
```

Three different ways for the SET operation to deliver data can be chosen for each of the two classes:

1) data written directly to the memory location where the Application Program has declared it: **`C#_da_mem`**
2) ID code and data written to buffer: **`C#_id_da_buf`**
3) data written to memory and the address of this memory location written to buffer: **`C#_da_mem_ad_buf`**

The # symbol denotes the Class number. If f. ex. Class 4 should have option 2) and Class 5 should have option 3) we would write this definement: `#define Buf_opt_ctr C4_id_da_buf + C5_da_mem_ad_buf`

Data Items delivered via API Receive Buffers are not verified against the Data Item Pointer Tables whether they exist or not. It is the responsibility of the Application Program to verify Data Items from a SET operation for those Classes where an API Receive Buffer is used. How the Application Program reads data from the buffers is described in Chapt. 6.

The application programmer must also decide how much memory to allocate for the API-buffers in use. We shall return to that question in a later section.

Internally, *GENIpro* uses a buffer for storing received Telegrams (the *Receive Buffer*) and a buffer for storing Telegrams for transmission (the *Transmit Buffer*). They are called *Data Frame Buffers* and the application programmer must decide how much memory to allocate for them. The larger buffers, the longer Telegrams can be handled. The *GENIbus Specification* prescribes a minimum of 70 bytes to be able to handle APDU's to their full length. Notice, that the specified number of bytes will be allocated twice, once for each of the buffers:

```
#define  DF_buf_len       [70; 256]      //length of Data Frame Buffers in bytes
```

GENIpro gives the application programmer the possibility of having a piece of code executed whenever a sound telegram is received from one of the channels.

```
#define  Bus_user_fct     "code or function call"   //User function for bus
#define  IR_user_fct      "code or function call"   //User function for IR
```

If this code uses any Application Program identifier, which normally is the case, this identifier must be declared external. It could be a function call. In this case, remember to add an external declaration of the function. A function consuming a few ms is allowed to be executed directly. Another way to have code executed is to start a task via one of the RTOS system calls _START(Task_name) or _SIGNAL(Sem_name). Notice, `Bus_user_fct` is only executed if the unit is in Slave Mode.

Bus Unit Table and Buffer Specifications

The Bus Unit Table specifications tell *GENIpro* the length of all the *Data Item Pointer Tables* all the *Tables* and the two *Common Tables* (see Chapt. 5.5). If the actual length of the corresponding table (which is also written by the application programmer) does not match the specified length, the compiler will generate an error message. Data Item Pointer Tables belonging to Classes which the application won't be using, do not have to exist. Give them the length zero and *GENIpro* won't search for them:

```
#define    HIGH_MEAS_ID      [0; 255]    //length of meas_tab,class 2 in bytes
#define    HIGH_CMD_ID       [0; 255]    //length of cmd_tab,class 3 in bytes
#define    HIGH_CONF_ID      [0; 255]    //length of conf_tab,class 4 in bytes
#define    HIGH_REF_ID       [0; 255]    //length of ref_tab,class 5 in bytes
#define    HIGH_TEST_ID      [0; 255]    //length of test_tab,class 6 in bytes
#define    HIGH_ASCII_ID     [0; 255]    //length of ascii_tab,class 7 in bytes
#define    COM_INFO_LEN      [1; 64]     //length of common_info_tab in bytes
#define    COM_PTR_LEN       [1; 64]     //length of common_ptr_tab in bytes
```

The API Receive Buffer specifications tell *GENIpro* the length of the buffers used to deliver data when a SET operation is performed. There is a separate buffer for Commands (Class 3), Configuration Parameters (Class 4) and Reference Values (Class 5). Memory will always be allocated for the command buffer, because commands are always delivered to buffer, whereas the two last buffers should only be allocated memory, if the application programmer has chosen to have data to these classes written to buffer and not directly to memory (se definement `Buf_opt_ctr`). To suppress the allocation of memory to buffers not used, specify a length of zero. When buffers are used, the minimum length to specify is 6 as prescribed by the *GENIbus Specification*. Two extra bytes to be used for buffer indices are automatically allocated.

```
#define    CMD_BUF_LEN       [6; 254]    //length of cmd_buf,class 3 in bytes
#define    CONF_BUF_LEN      [0,6; 254]  //length of conf_buf,class 4 in bytes
#define    REF_BUF_LEN       [0,6; 254]  //length of ref_buf,class 5 in bytes
```

Master Management Options

The length of the Reply Buffer used by *GENIpro* to deliver Data Replies to the Application Program must be specified in any of the Master Unit Types. This buffer must have a minimum length of 20 bytes and a maximum equal to the specified length of the Data Frame Buffers:

```
#define    Reply_buf_len    [20; DF_buf_len]  //Reply buffer length in bytes
```

When a Control Master operates as Bus Master it will take care that all Members of the bus will be polled regularly. The application programmmer can specify how frequently this must take place:

```
#define    AUTO_POLL_PERIOD [0; 255]    //In counts of 200ms, ctr_mast only
```

The two remaining *GENIpro* configurations apply only to Management Masters.

For *GENIpro* to allocate enough RAM for the Device Poll Table and for the Telegram Table, the necessary size of these tables must be known at compile time. The application programmer specifies how many different Device Types and how many different Telegrams, he wants to be handled.

```
#define    MAX_NO_DEVICES   [0; 255]    //Max No. of Device Types, man_mast only
#define    MAX_NO_TGMS      [0; 255]    //Max No. of Telegrams, man_mast only
```

5.5 Creating the Bus Unit Tables

The *Bus Unit Tables* consist of the *Data Item Pointer Tables*, the *Data Item INFO Tables* and two so called *Common Tables* (common for all classes). As seen from *GENIpro*, these tables constitute the major part of the data interface to the the Application Program. We can think of them as the static implementation of the Functional Profile for the unit. They are written in a separate module by the application programmer. Included with the *GENIpro* source code files is a file named **`bu_tab.c`**. It is an example of how to write the Bus Unit Tables, and forms a good basis from where to start. It is intended to let this example file and figure 5.2 explain for themselves how the tables are to be understood. Some might even find it educating to look at figure D.1.1 and D.1.2 in appendix D to get a complete overview of *GENIpro*. However, some important things we need to mention explicitly:

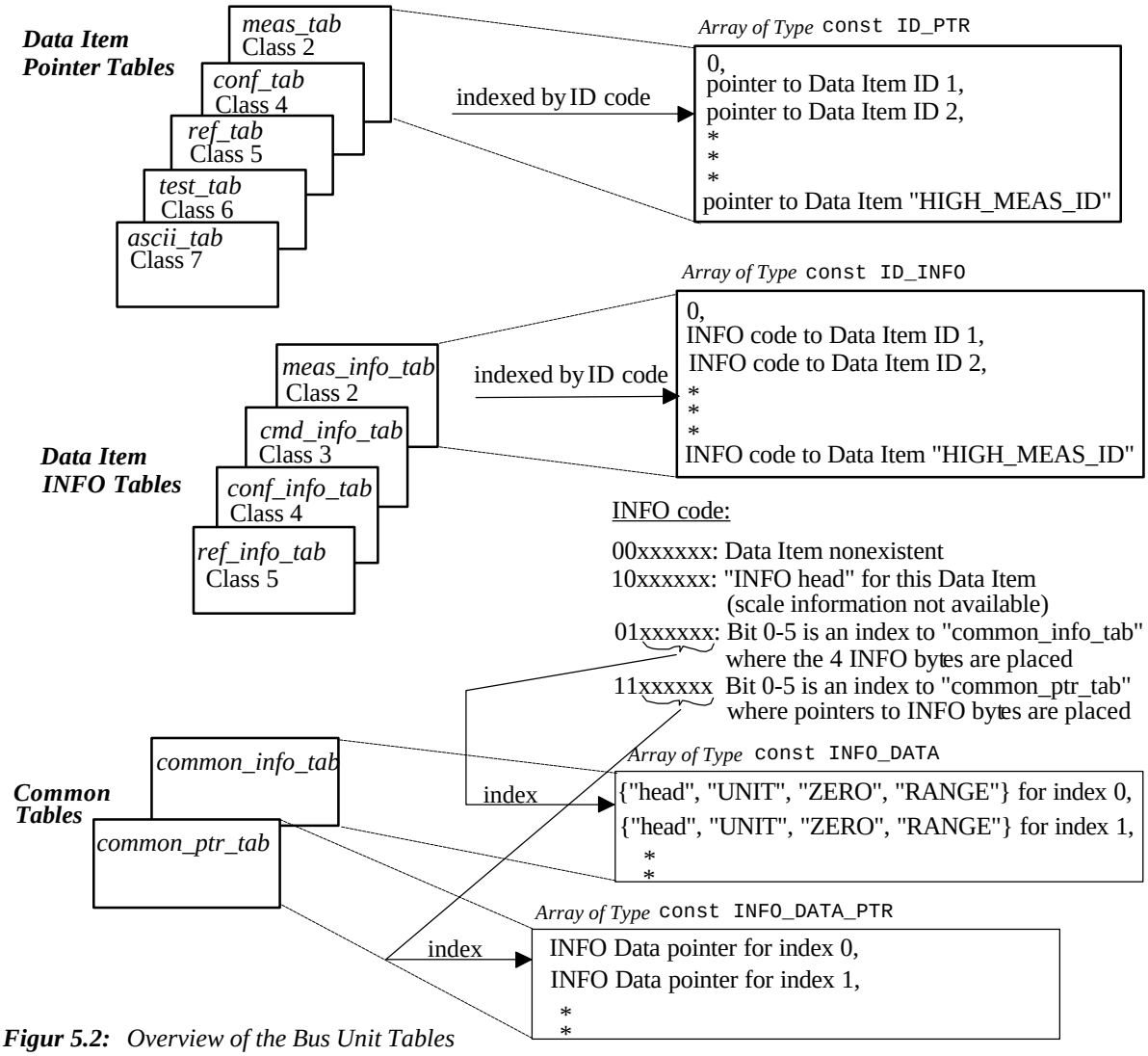
1) Data Item Pointer Tables and Data Item INFO Tables not used do not have to be declared. In this case, remember to define their length to zero in the `geni_cnf.h` file (see Chapt. 5.6).

2) Placing a 0 in a Data Item Pointer Table, means the return of an *Error APDU* if this Data Item is requested (see App C.13). All other Data Items in the same APDU will be lost. Placing a pointer to the value 255 (`&na`) instead will return this value when the Data Item is requested and no errors will be generated. The *Bus Master* will interpret this as "data not available".
   Notice that placing `"&na"` in a Data Item Pointer Table does not prevent a SET operation from writing to the Data Item in question. In this case, the location `"&na"` containing 255 will be written - for that reason, always declare `"na"` as constant (On emulators it might be necessary to have `"&na"` located in RAM when trying out SET operations to location `"&na"` to avoid GUARD BREAK).
   On targets using FLASH memory, be sure that writing to ROM does not cause any accidental erasure of FLASH content. Otherwise the use of `&na` in the Bus Unit Tables must be avoided or memory must be accessed via API Buffers.
   A master cannot see, from the reply to a SET operation, which Data Items were written to real locations and which were attempted to be written to `"&na"`. A following GET operation is needed.

3) Placing a 0 in a Data Item INFO Table, means the return of an Error APDU if INFO to this Data Item is requested. All other INFO values in the same APDU will be lost. Placing the value b10xxxxxx instead will return this value when the Data Item is requested and no errors will be generated. The Bus Master will interpret this as "scale information not available".

4) All the Bus Unit Tables use special type specifiers which the `bu_tab.c` module gets access to via inclusion of the `geni.h` file. The `ID_PTR` type is a pointer to unsigned char. `ID_INFO` type is unsigned char. `INFO_DATA` type is a structure of 4 unsigned chars (`info_head, unit, zero, range`). `INFO_DATA_PTR` type is a pointer to an `INFO_DATA` type.

5) The following Data Items must exist in all *Genibus Units*:
   Class 2: ID 3, `slave_mrk` ID 148, `unit_family` ID 149, `unit_type` ID 150, `unit_version`
   Class 4: ID 46, `unit_addr` ID 47, `group_addr` ID 81, `geni_setup`



Figur 5.2: *Overview of the Bus Unit Tables*

5.6 Configuration Parameters used by *GENIpro*

GENIpro makes use of three Configuration Parameters with reserved names :

46	**`unit_addr`:**	This is the address of the unit (See chapt. 2.2)
47	**`group_addr`:**	This is the group address of the unit (See Chapt. 2.2)
81	**`geni_setup`:**	Configurable *GENIpro* Options

These identifiers are all declared internally in the *GENIpro* modules, but the Application Program has access to them, (via inclusion of **`geni.h`**). This access is necessary for insertion of their pointers in the *Data Item Pointer Tables* and also for the EEPROM Driver part of the Application Program which also via pointers takes care of their EEPROM storage.

Via the Configuration Parameter `geni_setup`, the bit interpretation of which is shown below, certain *GENIpro* options can be modified. Placing these options in a Configuration Parameter makes it possible to do this modification from an external Master Unit (f. ex.. setting up a system where the system functionality requires a specific Bus Unit Mode or Bus Unit Type of the units on the bus).

A modified option is brought into action, as soon as the API function call `GeniReset` is executed from the Application Program - normally this happens after a hardware reset. The application programmer must be aware, that the value of `geni_setup` is vital for a *GENIbus* Unit to work as expected. The Data Item should get a proper value as a result of a

factory or user boot (self configuration). As an example, a standard bus unit operating as slave should give `geni_setup` the value *0x02*.

`geni_setup:` *(ID 81, Class 4), GENIpro* options modifiable via parameter configuration

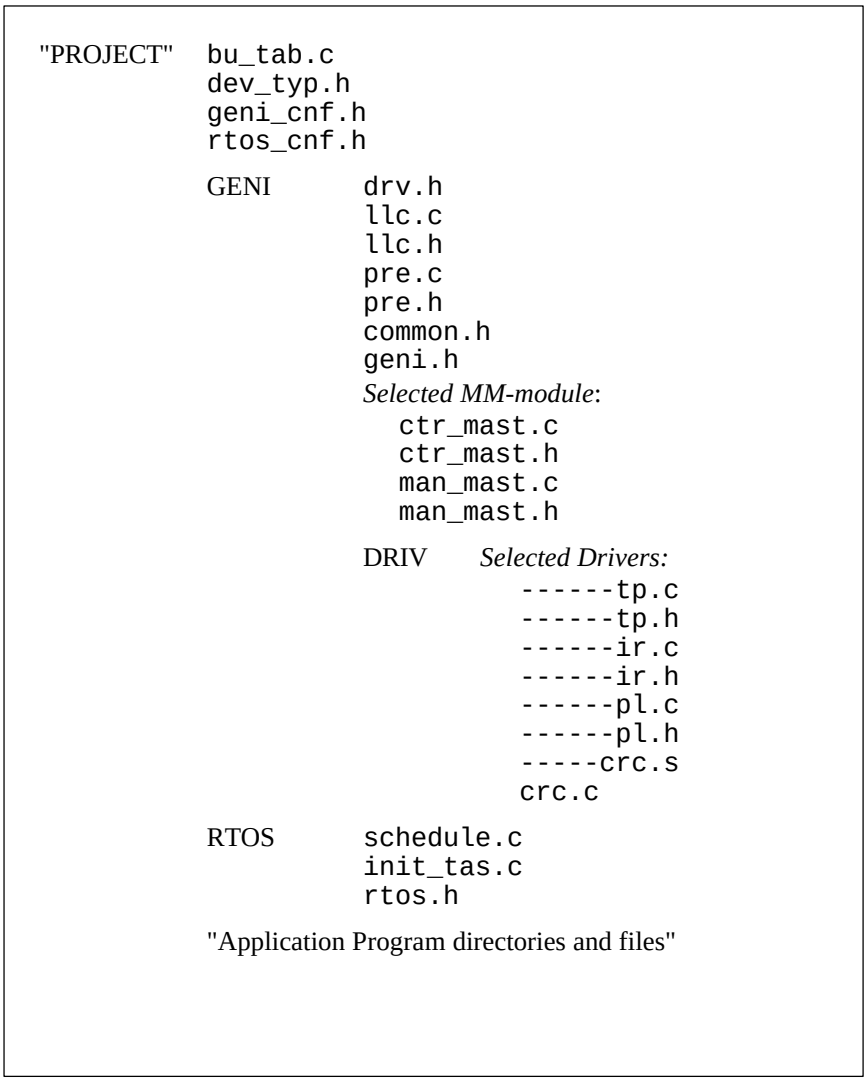
Bit no	Description	
0	0: Slave Unit Type	1: Master Unit Type
1	0: No Connection Reply (Subslave)	1: Connection Reply when Unpolled
2	0: Disabling of *GENIlink* Channel	1: Enabling of *GENIlink* Channel
3	0: Disabling of *GENIbus* Channel	1: Enabling of *GENIbus* Channel

These settings will take action when the API function `GeniReset` is called. The enabling and disabling of the channels can be overruled by the API functions `GeniEnable` and `GeniDisable`.

Notice! The configuration of Bus Unit Type to be a master is of course not possible if the Compile Time Option `CTO_BUS_UNIT_TYPE` has been set to `Slave_unit`. In the same way a channel which is excluded by a compile time option cannot be enabled by the configuration of a bit in `geni_setup` or the use of an API function call.

5.7 Software Modules

Below is an overview of all the modules that constitute the *GENIpro* implementation. A description is found in table 5.1.



***Figur 5.3**: Modules in the GENIpro implementation and a Proposed file organization*

The splitting into modules reflects the three layer model shown in Figure 3.3. The **`geni.txt`** file delivered together with the *GENIpro* source code describes how the files must be organized to comply with the include path's used. This is compatible with the file organization demanded by *RTOS*. To compile an Application Program with *GENIpro*, include **`geni.h`** in those modules using the API Services. Add Drivers for the specific target and add one of the Master Management modules if building a Master Unit This can be excluded when developing a Slave Unit Type.

For very time critical applications the CRC checking can be implemented in an assembler module. An example file `7805xcrc.s` for µPD7805x is included. It must be written with the same interface as the target independent C-code module `crc.c`. which it substitutes.

Module	File	Included Files
Target Dependent Drivers		
GENIbus Driver	`------tp.c`	`------tp.h, drv.h, llc.h`
GENIbus Driver Header File With section for user configuration of Driver	`------tp.h`	
GENIlink Driver	`------ir.c`	`------ir.h, drv.h, llc.h`
GENIlink Driver Header File With section for user configuration of Driver	`------ir.h`	
GENIbus Power Line Driver	`------pl.c`	`------pl.h, drv.h, llc.h`
GENIbus Power Line Driver Header File With section for user configuration of Driver	`------pl.h`	
Target assembler CRC Generator	`-----crc.s`	
Logical Link Control (LLC)		
GENIbus/GENIlink Driver Interface File	`drv.h`	
LLC functions	`llc.c`	`rtos.h, geni_cnf.h, drv.h, llc.h, pre.h`
LLC Header File	`llc.h`	`drv.h`
CRC Generator	`crc.c`	`geni_cnf.h`
Presentation Layer (PL)		
PL basic functions	`pre.c`	`rtos.h, geni_cnf.h, llc.h, pre.h`
PL basic Header File	`pre.h`	
MM-module for Control Master	`ctr_mast.c`	`rtos.h, geni_cnf.h, llc.h, pre.h, ctr_mast.h`
Control Master Header File	`ctr_mast.h`	
MM-module for Management Master	`man_mast.c`	`rtos.h, geni_cnf.h, llc.h, pre.h, ctr_mast.h`
Management Master Header File	`man_mast.h`	
Common *GENIpro* Header File	`common.h`	
API Header File for user inclusion	`geni.h`	`geni_cnf.h, pre.h`
Protocol Configuration		
GENIpro Configuration File	`geni_cnf.h`[+)]	`common.h`
Application Program Interface (API)		
Bus Unit Tables	`bu_tab.c`[*)]	`geni.h`, "app. header files"
Device Type definitions	`dev_typ.h`[*)]	
Application Program		
"application program files"		"target header file", `rtos.h`, `geni.h`, "app. header files"

[+)] To be filled out by the application programmer

[*)] These files are to be made by the application programmer, the names are optional. Example files are delivered together with the protocol files.

Table 5.1: *Overview of GENIpro Software Modules. The first characters in the name of the Driver modules is a type code for the target controller, and the last 2 characters is for the medium: RS485 Twisted Pair (tp), Infra Red light (ir), and Power Line (pl). F. ex. the GENIbus RS485 Driver for mPD7805x is named* `7805xtp.c`*.. GENIpro is completely implemented in C. The CRC generator* `crc.c` *belonging to LLC can be substituted by the assembler implementation* `-----crc.s` *belonging to the Drivers to reduce the time consumptation if necessary.*

5.8 Using the API

Appendix D.1 presents an overview of the API for a Slave Unit Type and a Master Unit Type. You might find it instructive to look at the figures D.1.1 and D.1.2 when reading about the Application Program use of the API.

Appendix A is a look up reference for all the functions and data structures in the API. The data structures accessible via API function calls are also described.

5.8.1 Operating as a Slave Unit Type

A Slave Unit Type has an API solely existing of a few Control Functions, the API Receive Buffers and the `ses_event` register as seen in figure D.1.1. The application programmer only need to write code to take care of the following:

1. Enabling and disabling the communication channels

Before the bus unit can receive anything *GENIpro* must be initialized and one of the channels, or both of them, must be enabled. The following peace of code does the initialization and enables both the *GENIbus* and the *GENIlink* channel:

```
GeniReset();        /* Initializes GENIpro.                          */
GeniEnable(BOTH);   /* Task's, timers and interrupts from both       */
                    /* channels in GENIpro are active.               */
```

A channel which is enabled can be disabled again. For example:

```
GeniDisable(BUS);   /* Bus channel disabled                          */
```

2. Changing the Class Access

The Class Access can be changed if necessary to something else than the default access (table A.1) via a Control Function. For example:

```
ClassAccess(TEST_APDU, SET_ACC, Enable);    /* Enable Test Class for writing.*/
ClassAccess(CONF_APDU, SET_ACC, Disable);   /* Disable Configuration         */
                                            /* Parameter Class for reading.  */
```

3. Emptying the API Receive Buffers

It is the responsibility of the Application Program to initialize the Receive Buffers in use. The Command Buffer is mandatory. Here is how to initialize it:

```
cmd_buf[WR_INDX] = BUF_START;
cmd_buf[RD_INDX] = BUF_START;
```

If a buffer is used for the reception of Configuration Parameters and Reference Values, these buffers must be initialized in the same way.

It is also the responsibility of the Application Program to cyclically emptying these buffers and to store or process the received data accordingly. If this is not done frequently enough, loss of data will occure. Example code for Receive Buffer handling can be seen in Appendix A.1.

4. Using the Session Event Register (optional)

It can be used to test if *GENIpro* has delivered data either in a Receive Buffer or in Application Memory. See the example in Appendix A.1.

5.8.2 Operating as a Control Master

A Control Master is a *GENIpro* implementation specially designed to applications utilising Submaster operation, or for the implementation of relatively simple master applications where all polling of date is done explicitly from the Application Program. It is importent to know, that the Control Master will be the Bus Master in case it has the lowest address on the bus. Handling of the duties which this implies (like Auto Polling) is automatically handled by *GENIpro* and no special actions is required from the Application Program.

Everything written about the API usage for a Slave Unit Type counts for a Control Master as well. Added to this are some extra features (middle block in figure A.1.2) which will be described in the following.

1. Sending a Telegram to a specific bus unit

This is done with the function `SendTgm(..)` which is described in Chapter 4.3. The content of the Telegram, its type and what unit to address can all be changed dynamically. The call will initiate a Communication Session completely handled by *GENIpro*. The Communication Session will always end with *GENIpro* writing an event in the registers `ses_event_tgm` and `ses_event_ctr` . You can look in Appendix A.1 to get a complete list of all possible session events.

2. Processing the Reply

The event G_DIR_REPLY means that a sound Data Reply has arrived, and is placed in the `data_reply_buf` ready for processing (read about `data_reply_buf` in Appendix A.1). If the session completion event G_DIR_ERR arrives, the session has not been carried out succesfully.
When a Communication Session has been initiated, it is the responsibility of the Application program to cyclically poll `ses_event_tgm` and `ses_event_ctr` for useful events and to 'consume' them. An alternative to polling the Session Event Registers is to use the `Com_ses_end` Semaphore which is signalled by *GENIpro* each time a Communication Session is completed. Use it for example to start a task for processing the Data Reply.
The Network Status Register `network_sta` is updated each time a Communication Session ends. It can be read via the function `ReadNetwStatus(..)`. Chapter 4.5 gives a description of what this special feature can be used for.

5.8.3 Operating as a Management Master

When in Slave Mode, the Managment Master works as a normal bus slave. This means that what is described in chapter 5.8.1 counts for Managment Master as well. The processing of Replies and the transmission of Direct Telegrams are as described for the Control Master in chapter 5.8.2. What comes extra for a Managment Master is the full control of the Auto Poll Cycle

1. Control of the Auto Poll Cycle

As a minimum, the Application Program must execute the following four function calls.

```
AllocPduRAM
StoreTgm
EditDevTab
SetPollCycTime
```

Chapter 5.2 is a thorough description of the use of these functions. Appendix A.3 is a complete reference of all possible function calls.

2. Designing a Managment Master

These are the major steps which the application programmer must go through to build a Managment Master application.

1) From the Functional Profiles, design a data model - known as a Device Type - of those units which must be controlled or monitored by the Bus Master (file *dev_typ.h*).
2) Declare storage locations - Device Objects - for as many units as needed.
3) Next, design the Telegrams which should be used for polling by the Auto Poll Cycle (*poll_tab.c*). These Telegrams must supply the data contained in the Device Type in question.
4) Plan the data update rate and design the Auto Poll Cycle timing accordingly. Don't update data in the Device Objects more often than necessary.
5) Write application code to process the Data Replies. This code must empty the Data Reply Buffer and store data in the correct locations in the Device Objects.
6) Define the control which the Bus Master must exercise on the Bus Units. Design the Direct Telegrams for these actions.

A. The API Reference

A.1 Variables

API Receive Buffers

The API Receive buffers are `cmd_buf[]`, `conf_buf[]` and `ref_buf[]`. The data send to the unit as a result of SET Operations when the unit is in Slave Mode is delivered in these buffers (Configuration Parameters and Reference Values can however also be delivered directly to application memory. This is a matter of a compile time option). The buffers all have a read and a write index placed as the first two bytes in the buffer itself, see the figures below. *GENIpro* delivers data to the buffers in a cell format according to the compile time option `CTO_BUF_OPT.` The write index is updated accordingly: it always points to the next free position in the buffer. The Application Program can test the indices to see if a buffer is empty or not. It is the responsibility of the Application Program to update the read index each time a cell is read and to reset both indices when the buffer has been emptied. Here are two examples:

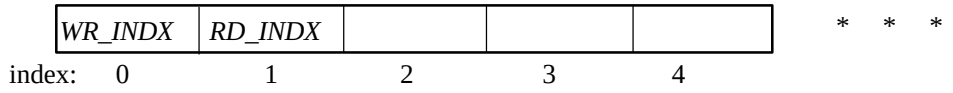


Figure A.1: *Command buffer structure*

```
while (cmd_buf[WR_INDX] != cmd_buf[RD_INDX])
      cmd_interprete(cmd_buf[cmd_buf[RD_INDX]++]);
cmd_buf[WR_INDX] = BUF_START;
cmd_buf[RD_INDX] = BUF_START;
```

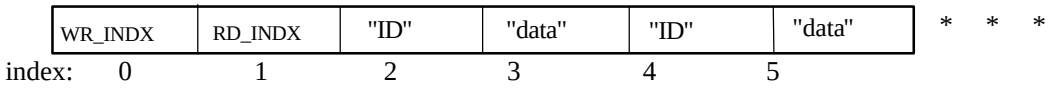


Figure A.2: *Example of Configuration Parameter buffer structure with* ***`CTO_BUF_OPT = C4_ID_DA_BUF`***

```
if (conf_buf[WR_INDX] != conf_buf[RD_INDX])
   {
   conf_id = conf_buf[conf_buf[RD_INDX]++];
   conf_data = conf_buf[conf_buf[RD_INDX]++];
   _SIGNAL(Conf_sem);                /* Signal to a configuration task that*/
   }                                 /* there are configuration data      */
else
   {
   conf_buf[WR_INDX] = BUF_START;
   conf_buf[RD_INDX] = BUF_START;
   }
```

This piece of code administrates the configuration buffer, and signals via a semaphore to another task (which must be waiting for this semaphore) that there are configuration data to process. The data is transferred via global variables (the only way for data transfer between tasks in RTOS78K).

`data_reply_buf`

Is a buffer containing the Data Reply PDU from a Data Request Session when the unit operates as Bus Master or Submaster. The first byte in the buffer specifies the PDU length. It is the responsibility of the Application Program to empty this buffer.

ses_event_tgm, ses_event_ctr

The Application Program can read the result of the last completed Communication Session in these variables called the Session Event Registers .
When operating as a Bus Master or Submaster `ses_event_tgm, ses_event_ctr` can take the following values:

`ses_event_tgm`: *Direct Communication Session completion events:*

`G_DIR_ERR`: Session ended. No Data Reply from addressed bus unit
`G_DIR_REPLY`: Session ended with a succesful Data Reply

Auto Poll Communication Session completion events:

`G_BUS_ERR`: Detection of TRANS_FAILED, CRC_ERR or REC_BREAK
`G_UNIT_ERR`: No Data Reply to Auto Poll request.
`G_AUTO_REPLY`: Data Reply to Auto Poll request (excluded in a Control Master).
`G_UNIT_REMOVED`: Unit removed from Network List due to high error rate.
`G_NEW_UNIT`: Connection Request Session ended with Connection Reply.
`G_UNKNOWN_DEV_TYPE`: A new unit with an unknown Device Type is recognized (there is no entry for it in the Device Poll Table (excl.in a Control Master).

Other events:

`G_NO_EVENT`: Nothing to report

`ses_event_ctr`: *Bus Control events:*

`G_SUB_MASTER`: *GENIpro* now operating in Submaster Mode
`G_BUS_MASTER`: *GENIpro* now operating in Bus Master Mode
`G_BUS_SLAVE`: *GENIpro* now operating in Slave Mode

Other events:

`G_NO_EVENT`: Nothing to report

The Application Program must clear the event after usage by writing the value `G_NO_EVENT` into the Session Event Registers

When operating as a Slave `ses_even_ctr` is not used, and `ses_event_tgm` can take the following simultaneous values:

Values: `G_COMMANDS`, One or more commands has been written into `cmd_buf[]`.
`G_CONF_DATA`, One or more configuration Parameters have been written into `conf_buf[]` or application memory.
`G_REF_VAL`, One or more Reference Values have been written into `ref_buf[]` or application memory.

Because they can occure simultaneous the events have been declared as a mask and must be used like this:

```
if (ses_event_tgm & G_COMMANDS) .....
else if (ses_event_tgm & G_CONF_DATA) .....
else if (ses_event_tgm & G_REF_VAL) .....
```

The Application Program must clear `ses_event_tgm` after usage by writing the value `G_NO_EVENT` into it**.**

Com_ses_end

Each time a Communication Session ends the semaphore `Com_ses_end` will be signalled. This semaphore can be used by the Application Program to start a task, which takes care of all the *GENIpro* interface handling. A simple example illustrates this:

```
void com_task(void)
   {
   "process the received PDU in Reply Buffer"
   _WAIT(0, Com_ses_end);
   }
```

A.2 Data Types

BUS_UNIT_RECORD

```
uchar unit_addr;        /* The Unit Address                                */
uchar unit_descr;       /* Group Address and Bus Unit Type                 */
                        /* bit 3-0: group_addr - 232                       */
                        /* bit 4:   CTO_BUS_UNIT_TYPE                      */
                        /*          0: Slave Unit Type                     */
                        /*          1: Master Unit Type                    */
                        /* bit 5:   CTO_MM                                 */
                        /*          0: Control Master                      */
                        /*          1: Management Master                   */
ushort unit_err_rate;   /* Error rate for this bus unit. When this value   */
                        /* exceeds an upper limit, the unit is             */
                        /* automatically removed from the Network List if  */
                        /* this facility is enabled                        */
uchar poll_list_ctr;    /* Controls the Poll List                          */
                        /* bit 0: 0: The unit is not on the Poll List      */
                        /*        1: The unit is on the Poll List          */
uchar unit_family;      /* Unit family code                                */
uchar unit_type;        /* Unit type code                                  */
uchar dev_type;         /* Device type code                                */
```

PDU_TGM

```
uchar no:               /* Telegram number                                 */
uchar type:             /* Selects the type of Telegram                    */
                        /* G_REQ: Data Request Telegram                    */
                        /* G_MESS: Data Message Telegram                   */
uchar len:              /* Length of Telegram                              */
uchar *pdu_ptr:         /* Pointer to a PDU declared as an array of uchar  */
```

NETW_STATUS

```
uchar unit_addr;        /* [0;231]: Unit Address for the unit from which   */
                        /*          the latest reply came                  */
uchar unit_descr;       /* Group Address and Bus Unit Type                 */
                        /* bit 3-0: group_addr - 232                       */
                        /* bit 4:   CTO_BUS_UNIT_TYPE                      */
                        /*          0: Slave Unit Type                     */
                        /*          1: Master Unit Type                    */
                        /* bit 5:   CTO_MM                                 */
                        /*          0: Control Master                      */
                        /*          1: Management Master                   */
uchar unit_family;      /* Unit family code                                */
uchar unit_type;        /* Unit type code                                  */
uchar dev_type;         /* [1;254]: Device Type                            */
uchar pdu_tgm_no;       /* [0;254]: Telegram No. used for polling in       */
                        /*          the Auto Poll Cycle.                   */
                        /* G_DIR_REP: Received reply is for a Direct       */
                        /*            Telegram                             */
uchar no_of_units;      /* No. of units in the Network List                */
ushort bus_err_rate;    /* Total error rate on the bus                     */
```

A.3 Functions

void GeniReset(void)

Resets *GENIpro's* internal modes and status variables. This function must have been called once before any other function in the API is called. A change of the *GENIpro* setup via the configuration parameter `geni_setup` (Chapt. 5.6) requires a call of *GENI_reset* to be effectuated.

void GeniEnable(uchar ch)

A call to this function enables one or both of the *GENIpro* communication channels. The internal timing control is active and the hardware interrupts used by the enabled channels are active.

Arguments: `ch`: Channel to enable
`BUS | IR | BOTH`

void GeniDisable(uchar ch)

A call to this function disables one or both of the *GENIpro* communication channels. The internal timing control and the hardware interrupts used by the disabled channels are deactivated.

Arguments: `ch`: Channel to disable
`BUS | IR | BOTH`

void ConReplyEnable(void)

A call to this function enables the Connection Reply Mechanism. Default is enabled.

void ConReplyDisable(void)

A call to this function disables the Connection Reply Mechanism. The unit will then appear on the bus as a Subslave. Default is enabled.

void ClassAcc(uchar cl, uchar opr, uchar en_dis)

The access to the *Operations* SET and GET in specific *Classes* can be enabled and disabled. The INFO Operation is always enabled. *GENIpro* initialises all Class Access to the default status shown in the table below.

Arguments: `cl`: Class
`MESS_APDU | CMD_APDU | CONF_APDU | REF_APDU | TEST_APDU | ASCII_APDU`
`opr`: Operation
`SET_ACC | GET_ACC`
`en_dis`: Enable or disable
`Enable | Disable`

Class	GET	SET
2. Measured Data	Enabled	-
3. Commands	-	Enabled
4. Configuration Parameters	Enabled	Enabled
5. Reference Values	Enabled	Enabled
6. Test Data	Disabled	Disabled
7. ASCII Strings	Enabled	-
8. Dump Memory	-	-

Table A.1: *Default status of the Class Access.*

uchar SendTgm(uchar da, uchar tgm_type, uchar pdu_len, (void *) pdu_ptr)

Used to start a transmission of a Direct Telegram to the unit(s) specified by the `da` argument.

Arguments: `da` : Destignation Address.
[0; 247]: Transmits to units with this specified Unit or Group Address.
`G_GLOBAL`: Broadcasts the Telegram
`tgm_type`: Selects the type of Telegram
`G_REQ`: Data Request Telegram
`G_MESS`: Data Message Telegram
`pdu_len`: PDU specifier
[2;253]: Value of the PDU length, `ptr` is a pointer to this PDU
`pdu_ptr`: A void pointer to a PDU declared as an array of `uchar` or `const uchar`.
Returns: `err_code`: `G_OK`
`G_MASTER_DISABLED`
`G_SUBMASTER_BUSY`

uchar ReadNetwStatus(NETW_STATUS *sta_ptr)

Used to read from the Network Status Record.

Arguments: `*sta_ptr`: Pointer to a structure of type `NETW_STATUS` where the Network Status Record is delivered.
Returns: `err_code`: `G_OK`

void ResetDevTab()

Used to clear the Device Poll Table

uchar EditDevTab(uchar dev, uchar tgm_no_0, uchar tgm_no_1, uchar tgm_no_2, uchar tgm_no_3)

Used to change or insert a specific entry in the Auto Poll Device Table. The function simply overwrites the existing Telegram numbers with the values given as arguments in the function call. These Telegram numbers are indices referring to Telegrams in the Telegram Table. The used Telegram numbers must exist in the Telegram Table.

Arguments: `dev`: *[0; 254]*: Device number (row number in Device Table),
`tgm_no_0`: *[0; 254]*: New value of the Telegram No. for Poll Cycle No. 0
`G_NONE`: No polling of this Device in Poll Cycle No. 0
`tgm_no_1`: *[0; 254]*: New value of the Telegram No. for Poll Cycle No. 1
`G_NONE`: No polling of this Device in Poll Cycle No. 1
`tgm_no_2`: *[0; 254]*: New value of the Telegram No. for Poll Cycle No. 2
`G_NONE`: No polling of this Device in Poll Cycle No. 2
`tgm_no_3`: *[0; 254]*: New value of the Telegram No. for Poll Cycle No. 3
`G_NONE`: No polling of this Device in Poll Cycle No. 3

Returns: `err_code`: `G_OK`
`G_NOT_FOUND`

uchar StoreTgm(PDU_TGM *ptr)

Used to store a Telegram in the Telegram Table and in PDU RAM. A call to function `Alloc_PduRAM` must have taken place prior to the use of this function.

Arguments: `*ptr:` Pointer to a structure of type PDU_TGM defining the Telegram
`no:` Telegram number
`type`: Selects the type of Telegram
`G_REQ`: Data Request Telegram
`G_MESS`: Data Message Telegram
`len:` Length of Telegram
`*pdu_ptr`: Pointer to a PDU declared as an array of `uchar`.

Returns: `err_code`: `G_OK`
`G_NO_TGM_SPACE`
`G_NOT_FOUND`

`void ResetTgmTab()`

Used to clear the Telegram Table. This also cleares the complete PDU RAM.

`uchar AllocPduRAM(ushort size, uchar *ram_ptr)`

Used to allocate RAM for the storing of PDU's. These PDU's are referenced via the Telegram Table and used by the Auto Poll Cycle. This function must have been called prior to any call of `StoreTgm`

Arguments: `size`: Number of bytes to allocate
`*ram_ptr`: Pointer to RAM location
Returns: `err_code`: `G_OK`
`G_ARG_ERR`: Error in arguments

`uchar GetPDUptr(uchar tgm_no, uchar **pdu_ptr)`

Used to request the PDU pointer from the Telegram Table

Arguments: `tgm_no`: Telegram number.
`**pdu_ptr`: Pointer to location where to return pointer to belonging PDU residing in PDU RAM.
Returns: `err_code`: `G_OK`
`G_TGM_NOT_FOUND`

`uchar SetPollCycTime(ushort cyc_tim_0, ushort cyc_tim_1,`

Used to set the the Auto Poll Cycle Times

Arguments: `cyc_tim_1`: Poll Cycle Time No. 0
[1;65535] poll cycle time No. 0 in increments of 200 ms
`G_OFF` disables poll cycle No. 0
`cyc_tim_1`: Poll Cycle Time No. 1
[1;65535] poll cycle time No. 1 in increments of 200 ms
`G_OFF` disables poll cycle No. 1
`cyc_tim_2` Poll Cycle Clock No. 2
[1;65535] poll cycle time No. 2 in increments of 200 ms
`G_OFF` disables poll cycle No. 2
`cyc_tim_3` Poll Cycle Clock No. 3
[1;65535] poll cycle time No. 3 in increments of 200 ms
`G_OFF` disables poll cycle No. 3
Returns: `err_code`: `G_OK`

uchar AutoPollCtr(uchar cmd)

Used to control the Auto Poll Cycle. After a Master Appointment will the Auto Poll Cycle automatically be started.

Arguments: cmd: Auto Poll Cycle control command.
- G_START_AUTO_POLL: Resets and starts the Auto Poll Cycle (default)
- G_STOP_AUTO_POLL: Stops the Auto Poll Cycle. The Connection Request/Reply mechanism will still be operating in the background.
- G_SMALL_SWEEP: Polls all members of the Poll List with the Poll Cycle 0 Telegram once
- G_LARGE_SWEEP: Polls all members of the Poll List with the Poll Cycle 0, 1, 2 and 3 Telegram once.
- G_CON_REQ_ENABEL: Connection Requests in the Auto Poll Cycle is enabled (default)
- G_CON_REQ_DISABEL: Connection Requests in the Auto Poll Cycle is disabled

Returns: err_code: G_OK
G_NOT_BUS_MASTER

uchar MmCtr(uchar cmd)

Used to control the behaviour of Master Management

Arguments: cmd: Master Management control command
- G_HIGH_PRIO: Overrules the address priority mechanism in attempt to be Bus Master (not implemented)
- G_NORM_PRIO: Follows the address priority mechanism in attempt to be Bus Master (not implemented)
- G_MAST_ENABLE: Enables the acceptance of Bus Master Appointment and the use of Direct Telegrams (default)
- G_MAST_DISABLE: Forces GENIpro into Slave Mode and prohibits the reentering of Bus Master Mode by rejecting any direct (BM_ack) or indirect appointment to Bus Master. Disables the use of Direct Telegrams.
  If the unit was in Bus Master Mode prior to the function call a Bus Configuration will result.
- G_MONITOR: Sets the unit in Monitor Mode (not implemented)
- G_MONITOR_END: Brings the unit out of Monitor Mode (not implemented)

Returns: err_code: G_OK

uchar EditPollList(uchar sel, uchar no, uchar rem_add)

Used to add members to the Poll List or to remove members from the Poll List. Members to add must exist in the Network List, otherwise an error code is returned.

Arguments: sel: Selects a member group to operate on
- G_ALL: All units in the Network List
- G_DEV_TYPE: All units with the Device Type number specified by the no argument.
- G_ADDR: The unit with the address specified by the no argument.

no: Number of the member(s) to operate on
- <*number*>: Number to select specific unit(s). Can be a Device Type or a Unit Address. This is interpreted according to the sel argument.

rem_add: Operation to perform on selected units.
- G_REM: Removes the specified unit(s) from the Poll List.
- G_ADD: Adds the specified unit(s) to the Poll List.

Returns: err_code: G_OK

ResetPollList()

Resets the Poll List. All members are removed.

uchar EditNetwList(uchar addr, uchar rem_add)

Removes a unit from the Network List or adds a unit to the Network List.

Arguments: `addr` [0;231], specifies the Unit Address
`rem_add`: `G_REM`: Remove the unit
`G_ADD`: Add the unit
Returns: `err_code`: `G_OK`

uchar ReadNetwList(uchar index, BUS_UNIT_RECORD *bur_ptr)

Used to read from the Network List.

Arguments: `index`: [0; 255], index to Bus Unit Record in Network List
`*bur_ptr`: Pointer to a structure of type `BUS_UNIT_RECORD` where the Bus Unit Record for the requested unit is delivered.
Returns: `err_code` `G_OK`
`G_NOT_FOUND`

ResetNetwList()

Resets the Network List. All members are removed.

uchar SetDeviceType(uchar addr, uchar dev_type)

When *GENIpro* first recognize a new unit it is automatically given a Device Type equal to `unit_family`. This function can set the Device Type in the Network List to a user defined type.

Arguments: `addr`: [0;231], specifies the Unit Address
`dev_type`: Specifies the Device Type
Returns: `err_code`: `G_OK`
`G_NOT_FOUND`

InsertUnit(uchar addr)

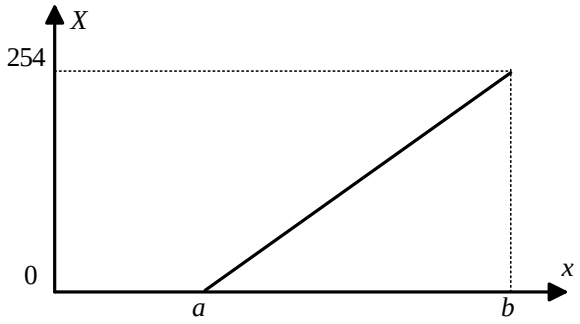
The Application Program can use this function to insert a unit in the Network List. This will result in the event `G_NEW_UNIT` beeing written to the `ses_event_tgm` register. *GENIpro* will by itself request data from the unit to complete the Bus Unit Record.

Arguments: `addr`: [0;231], specifies the Unit Address

B. Scaling of Values

The purpose of scaling is to map the value of a variable *x*, representing some physical entity, into its computer representation *X*. The *GENIbus* value representation is based on 8 bit quantities. This means, that scaling maps an interval of real numbers with any chosen length and from any chosen decade linearly into an 8 bit integer representation:

$$x \in [a;b] \;\rightarrow\; X \in [0;254] \qquad ;a,b \in \Re$$



Figur B.1: *Mapping real numbers into a byte value.*

This mapping is shown graphically in the figure above. The value *255* is reserved for indication of "data not available". What is needed now is a mathematical expression for the mapping and its inverse. The use of the symbols in the figure and the straight line relationship give us this equation to start with:

$$\text{(B.1)} \qquad X(x) = -254\tfrac{a}{b-a} + 254\tfrac{1}{b-a}x = \tfrac{254}{b-a}(-a+x)$$

a is the *"zero"* for the interval to be mapped, and *(b-a)* is its *"range"*. Both occure directly in the equation. Contained in this *"zero"* and *"range"*, however is the physical unit with some prefix factor to take the decade into consideration. To isolate this, a multiplier called *UNIT* is introduced in the following way:

$$\begin{aligned} b\text{-}a &= \text{"range"} = RANGE \cdot UNIT \\ a &= \text{"zero"} = ZERO \cdot UNIT \end{aligned}$$

Substituting this in (B.1) gives the final *conversion formulas*:

$$\text{(B.2a)} \qquad X = \frac{254}{RANGE \bullet UNIT}(-ZERO \bullet UNIT + x)$$

$$\text{(B.2b)} \qquad x = \left(ZERO + X\frac{RANGE}{254}\right) \bullet UNIT$$

ZERO and *RANGE* can be represented in 8 bit, consequently they are suitable for *GENIbus*. But what about *UNIT*? The only way to handle *UNIT*, is to consider it as an index to a standard table of defined physical units with a prefix factor. This table, from now on called *The Unit Table*, can cover a substantially amount of different scalings although it is small. *UNIT* is chosen to be in 8 bit (not surprisingly). Bit 7 is reserved for one problem, that was left over: the sign of *ZERO*. Left are 7 bits giving 128 possible entries in The Unit Table. The job of value scaling to *GENIbus*, now means for the application programmer to determine *ZERO, RANGE* and *UNIT* for each of the entities he wants represented. An example will enlighten the procedure:

i_dc is to be scaled in the interval *[a;b]=[2A;10A]*.

b-a = 10A - 2A = 8A = RANGE·UNIT

A natural choise here would be *UNIT = 1A*. With this value currents up to a range of *255A* could be represented. This seems like overkill for the Grundfos product family. Chosing *UNIT=0.1A* instead, reduces the range with a factor 10 but at the same time increases the accuracy with which limits *a* and *b* can be specified correspondingly. This is a better solution. We now find:

UNIT = *0.1A* (or actually an index to this value in *The Unit Table*)
RANGE = *8A/UNIT* = *80*
ZERO = *2A/UNIT* = *20*

Shown in Table B.1 are examples of the variety of intervals representable by just changing *ZERO* and *RANGE* keeping the same *UNIT*. Table B.2 is The Unit Table with alle the defined standard units.

[a;b]	*RANGE*	*ZERO*
[2A; 10A]	*80*	*20*
[0A; 5A]	*50*	*0*
[0.5A; 3.1A]	*26*	*5*
[0A; 25A]	*250*	*0*
[25A; 50A]	*250*	*250*
[0.1A; 0.2A]	*1*	*1*

***Table B.1*:**
Example showing the variety of scalings that can be covered by just varying RANGE and ZERO keeping UNIT constantly equal to 0.1A

The Scalling formulas (B.2a-b) are only valid for 8 bit values, but it is only natural to extend them to count for 16 bit values as well. Understanding X_{16} as a 16 bit computer representation leads directly to the extended version of the scaling formulas:

(B.3a) $$X_{16} = \frac{254 \bullet 256}{RANGE \bullet UNIT}(-ZERO \bullet UNIT + x)$$

(B.3b) $$x = \left(ZERO + X_{16}\frac{RANGE}{254 \bullet 256}\right) \bullet UNIT = \left(ZERO + X_{hi}\frac{RANGE}{254} + X_{lo}\frac{RANGE}{254 \bullet 256}\right) \bullet UNIT$$

Because *GENIpro* handles single byte *Data Items*, X_{hi} and X_{lo} must have one ID code each. Per definition, the scaling information *ZERO*, *RANGE* and *UNIT* is connected with the high order byte, X_{hi}. The low order byte X_{lo} has no scaling information because this comes implicit from (B.3a-b). Using the *INFO* operation on a low order byte will however return an *INFO head*, where the *BO* bit must be set to indicate that this is a low order byte (See App. C.2).

When using extended precision (32 bit), a RANGE specifier is no longer needed due to the very high dynamics, that can be contained in 32 bits.

(B.4a) $$X_{32} = \frac{x}{UNIT} - 256^2 \bullet ZERO_{16}$$

(B.4b) $$x = (256^2 \bullet ZERO_{16} + X_{32}) \bullet UNIT$$

Note thatt X_{32} consists of 4 data items:

$$X_{32} = X_{hi} * 256^3 + X_{lo1} * 256^2 + X_{lo2} * 256 + X_{lo}$$

As usual the scaling information (requested with INFO) is connected to the high order data item X_{hi}.

The Unit Table					
Index	**Physical entity**	**Unit with prefix**	**Index**	**Physical entity**	**Unit with prefix**
1	Electrical current	0.1 A	24	Head	0.1 m
42	Electrical current	0.2 A	25	Head	1 m
62	Electrical current	0.5 A	26	Head	10 m
2	Electrical current	5 A	56	Head	1 ft
3	Voltage	0.1 V	59	Head	10 ft
4	Voltage	1 V	51	Pressure	0.001 bar
5	Voltage	5 V	27	Pressure	0.01 bar
6	Electrical resistance	1 Ω	28	Pressure	0.1 bar
43	Electrical resistance	10 kΩ	29	Pressure	1 bar
7	Power (active)	1 W	61	Pressure	1 kPa
8	Power (active)	10 W	55	Pressure	1 psi
9	Power (active)	100 W	60	Pressure	10 psi
44	Power (active)	1 kW	30	Percentage	1 %
45	Power (active)	10 kW	31	Energy	1 kWh
10	Power (apparent)	1 VA	32	Energy	10 kWh
11	Power (apparent)	10 VA	33	Energy	100 kWh
12	Power (apparent)	100 VA	40	Energy	512 kWh
13	Power (reactive)	1 VAr	46	Energy	1 MWh
14	Power (reactive)	10 VAr	47	Energy	10 MWh
15	Power (reactive)	100 VAr	48	Energy	100 MWh
16	Frequency	1 Hz	34	Angular velocity	2 rad/s
38	Frequency	2 Hz	39	Time	1024h
17	Frequency	2.5 Hz	35	Time	1 h
18	Rotational velocity	12 rpm	36	Time	2 min
19	Rotational velocity	100 rpm	37	Time	1 s
20	Temperature	0.1 ^{0}C	49	Angular degrees	1º
21	Temperature	1 ^{0}C	50	Gain	1
57	Temperature	1 ^{0}F	70	Volume	0.1 ml
69	Flow	0.1 ml/h	71	Volume	1 nl
22	Flow	0.1 m^3/h	64	Volume	0.1 m^3
23	Flow	1 m^3/h	65	Volume	1000 m^3
41	Flow	5 m^3/h	67	Volume	256 m^3
52	Flow	1 l/s	66	Energy per volume	10 kWh/m^3
63	Flow	0.1 l/s	68	Area	1m^2
53	Flow	1 m^3/s			
54	Flow	1 gpm			
58	Flow	10 gpm			

Table B.2: *The Unit Table. UNIT=255 means unit not available.*

C. *GENIpro* Telegram Specification

C.1 Specification of the Framing

GENIpro can handle APDU's up to a length of 65 bytes, and PDU's up to a length of 253 bytes. This gives the following theoretical maximum values:

Maximum length of Telegram = 259 bytes
Maximum length of PDU = 253 bytes

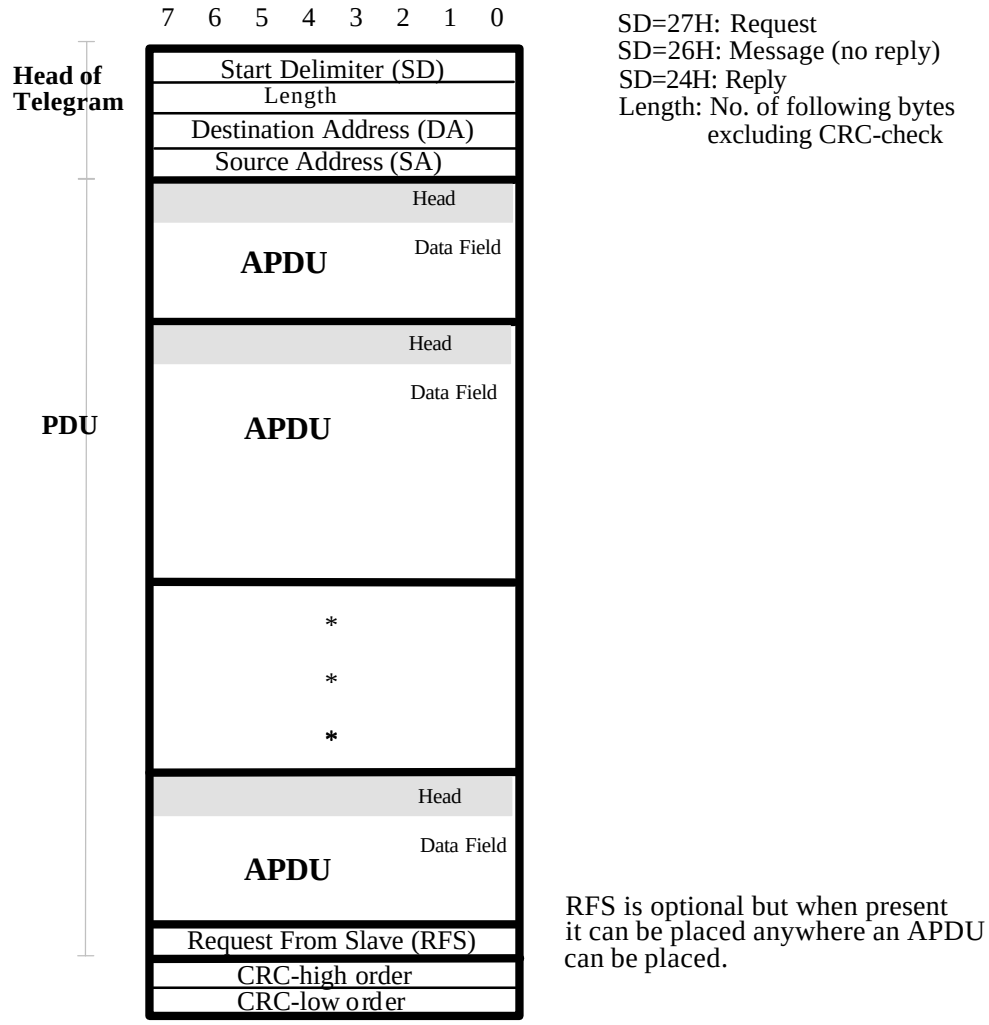


Figure C.1: *The GENIpro Telegram format*

In general, Telegrams with a length utilising the specification to its full extent cannot be used. The maximum allowed length of the Telegram depends on the size of the Receive- and the Transmit Buffer in the units communicating. Receive- and Transmit Buffers have the same length with the symbolic name `df_buf_len` (Data Frame buffer length). When it is taken into account, that the *CRC Valu*e is not buffered, the following equations can be deduced from the figure below and serve as a guide to calculate the maximum allowed size of a Telegram to a specific GENIbus Unit:

length of Telegram <= `df_buf_len` + 2
length of PDU <= `df_buf_len` - 4

It is required for any GENIbus unit, that `df_buf_len` has a minimum value of 70 bytes. Some units might have larger buffers. Request `df_buf_len` (Class 0, ID 2) to see the exact value.

C.2 Specification of the APDU

All *APDU*'s consist of a *Head* and a *Data Field* as shown in the figure below. The maximum length of an APDU can of course never exceed the allowed length of a PDU, and never exceed the theoretical maximum length.

max APDU length <= `df_buf_len` - 4 AND max APDU length <= 65

The different fields in the Head are explained in the following.

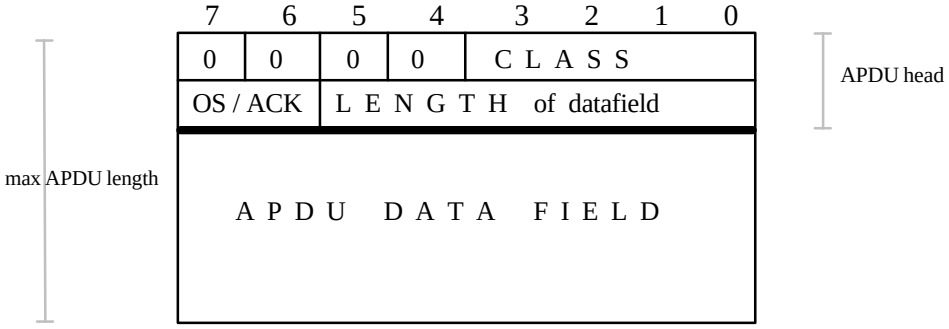


Figure C.2: *APDU Format*

CLASS: Specifies which *Class* the *APDU* belongs to. The *APDU Data Field* will be treated accordingly.

OS/ACK: OS: *Operation Specifier* (for Data Request and Data Message):
00: GET, to read the value of Data Items
10: SET, to write the value of Data Items
11: INFO, to read the scaling information of Data Items (see fig C.3 next page)

ACK: *Acknowledge* (for Data Reply): 00: Everything ok
01: *Class* unknown
10: *Data Item ID-code* unknown
11: *Operation* illegal

The table below shows the *Classes* and the three different *Operations* that can be used from a *Bus Master*.

	Operation		
Data Class	**GET**	**SET**	**INFO**
0. Protocol Data	X		
1. Bus Data	X		
2. Measured Data	X		X
3. Commands		X	X
4. Configuration Parameters	X	X	X
5. Reference Values	X	X	X
6. Test Data	X	X	
7. ASCII-strings	X		
8. Memory Blocks	X	X	
9. Embedded PDU's	X		
10. Data Objects	X	X	
11. 16 bit Measured Data	X		X
12. 16 bit Configuration Parameters	X	X	X
13. 16 bit Reference Values	X	X	X

Table C.1: *All the Data Classes and the allowed Operations*

INFO is an *operation* which can be used to request scaling information about *Data Items.* If the Data Item is scaled the following 4 byte *Info Data* will be returned. Otherwise only the *INFO Head* will be returned.

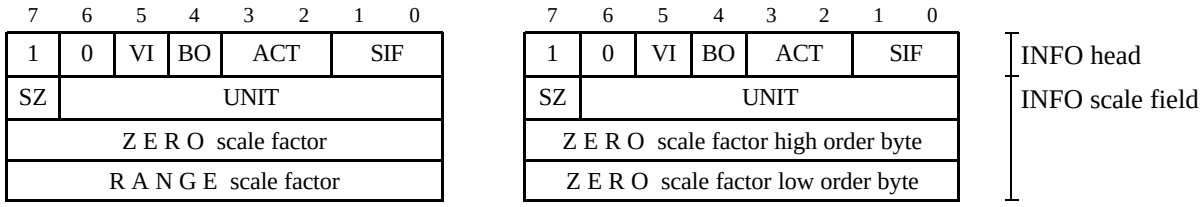


Figure C.3: *Format of the Info Data. Normal 8/16 bit scaling to the left. Extended precision (32 bit) to the right*

Bit 6 & 7: APDU's in a Data Reply will always have the bit values shown. Internally however, these bit positions are used as an *INFO code* to direct *GENIpro* where to find the *INFO Scale Field*. This mechanism is strongly related to the writing of *Data Item INFO Tables* and for that reason relevant for an application programmer (See Chapt 5.5).

VI: Value Interpretation:
0: Use only values from 0-254, 255 means "data not available"
1: All values 0-255 are valid values. This is always the case for Data Items which are a low order part of a 16/32 bit value (See App. B).

BO: Byte Order
0: High order byte, this is default for all values that are only 8 bit
1: Low order byte to a 16/32 bit value

ACT: Access Control Type: (this only counts for the SET operation on Configuration Parameters and Commands)
00: *User* type, meaning SET is possible without any *Access Key*
01: *Service* type, meaning SET is only possible with *Factory* or *Service Access Key*
Notice! This access control is only specified, not implemented!
10: *Factory* type, meaning SET is only possible with *Factory Access Key*
Notice! This access control is only specified, not implemented!
11: Unchangable type. Configuration Parameter can never be written, Command can never be executed.

SIF: Scale Information Format:
00: Scale information not available (no UNIT, ZERO or RANGE)
01: Bitwise interpreted value (no UNIT, ZERO or RANGE)
10: Scaled value (UNIT, ZERO and RANGE exist)
11: Extended precision (UNIT and ZERO hi/lo exist)
If SIF=00 or 01 the 3 last bytes in the Info Data will not be returned.

SZ: Sign of ZERO: 0: positive
1: negative

UNIT: A 7 bit index to the *Unit Table*. This tells the physical unit of the Data Item (see App B).

ZERO scale factor, RANGE scale factor: For convertion between the physical representation and the computer representation of a value.

Definition: x = [ZERO + X*(RANGE/full scale)] * UNIT

x = physical representation
X = computer representation
full scale = 254 or 255 (according to VI bit)
(see App. B for a complete description of scaling, and how to handle 16/32 bit values).

C.3 APDU's For Class 0, Protocol Data

Class Specification Data Items characterising the setup and the present status of *GENIpro* for the specific unit

GET

Request from master

0	0	0	0	Class = 0
0	0	Length		
ID				
ID				
ID				
ID				
:				

Reply from slave

0	0	0	0	Class = 0
0	0	Length		
Value				
Value				
Value				
Value				
:				

SET Illegal Operation

INFO Illegal Operation

Application comments

C.4 APDU's For Class 1, Bus Data

Class Specification

Data Items characterising the *GENIbus Network*. This information can only be extracted from a *Master Unit Type* that has been active as a *Bus Master*.
Each ID code generally represents a data structure. The format is not yet specified.

GET

Request from master

0	0	0	0	Class = 1
0	0	Length		
ID				
ID				
ID				
ID				
:				

Reply from slave

0	0	0	0	Class = 1
0	0	Length		
Data structure				
Data structure				
Data structure				
Data structure				
:				

SET

Illegal Operation

INFO

Illegal Operation

Application comments

C.5 APDU's For Class 2, Measured Data

Class Specification

Data Items generated in the unit. They can be read (GET) but not written (SET) from a master. Contains information about the unit and the application (system) in which it takes part. Can generally be values or bitwise interpreted data,

Categorizing:
- ID-codes 1-20: Application Specific Data Items
- ID-codes 21-99: General Data Items, related to a Functional Profile
- ID-codes 100-140: Application Specific Data Items
- ID-codes 141-255: General Data Items, related to a Functional Profile

GET

Request from master

0	0	0	0	Class = 2
0	0	Length		
ID				
ID				
ID				
ID				
:				

Reply from slave

0	0	0	0	Class = 2
0	0	Length		
Value				
Value				
Value				
Value				
:				

SET

Illegal Operation

INFO

Request from master

0	0	0	0	Class = 2
1	1	Length		
ID				
:				

Reply from slave

0	0	0	0	Class = 2
0	0	Length		
1 or 4 byte INFO Data				
:				

Application comments

C.6 APDU's For Class 3, Commands

Class Specification

A command is a directive from a master to a slave to do something uniquely connected to the command ID and described by its name. A command takes no parameters and for that reason cannot carry any values to the slave.

Type labels:
Complementary Mode Commands (CMo): Mode shift commands that appear in complementary pairs, these are by far the most common,
Mode Commands (Mod): Mode shift commands that select one between several possible modes
Enable/Disable Commands (ED): Can enable/disable certain options
Action Commands (Act): Directs the slave to do a certain action or series of actions,
Setting Commands (Set): Changes parameters that influences the way of operation,

GET

Illegal Operation

SET

Request from master

0	0	0	0	Class = 3
1	0	Length		
ID				
ID				

:

Reply from slave

0	0	0	0	Class = 3
0	0	Length		

INFO

Request from master

0	0	0	0	Class = 3
1	1	Length		
ID				

:

Reply from slave

0	0	0	0	Class = 3
0	0	Length		
INFO Head				

:

Application comments

Not all possible commands are valid in all possible modes and some might have preference to others. It is a matter of the application programmer to define such rules for the unit in question, and a matter of the *Application Program* to handle these rules,When a command is delivered to the Application Program via the command buffer, bit 7 is used for signaling the source channel for the command (0 for *GENIlink*, 1 for *GENIbus*), This means that only 127 different commands are possible.

The total number of allowed Commands in a *PDU* depends on the size of the command buffer `cmd_buf[]` in the addressed unit. This figure is called **`CMD_BUF_LEN`** and is therefore specific for the unit in question.

GENIbus Unit

Must be able to buffer at least 6 commands

C.7 APDU's for Class 4, Configuration Parameters

Class Specification

These Data Items are typically used once under production or once during installation. The Data Items are stored in the slave in none volatile memory (e.g. EEPROM, ROM)

Categorizing:
- ID-codes 1-20: Application Specific Data Items
- ID-codes 21-99: General Data Items, related to a Functional Profile
- ID-codes 100-140: Application Specific Data Items
- ID-codes 141-255: General Data Items, related to a Functional Profile

GET

Request from master

0	0	0	0	Class = 4
0	0	Length		
ID				
ID				

:

Reply from slave

0	0	0	0	Class = 4
0	0	Length		
Value				
Value				

:

SET

Request from master

0	0	0	0	Class = 4
1	0	Length		
ID				
Value				

:

Reply from slave

0	0	0	0	Class = 4
0	0	Length		

INFO

Request from master

0	0	0	0	Class = 4
1	1	Length		
ID				

:

Reply from slave

0	0	0	0	Class = 4
0	0	Length		
1 or 4 byte INFO Data				

:

Application comments

GENIpro always delivers Configuration Parameters in RAM, either in an *API Buffer* or directly in Application Program RAM. It is the responsibility of the *Application Program* to perform the actual EEPROM programming. Via a *Compile Time Option (CTO)* the application programmer can choose to have Configuration Parameters delivered by *GENIpro* in the buffer `conf_buf[]` instead o directly to a location in RAM. The size of this buffer is device dependent so the maximum number of Configuration Parameters, which can be SET via a single Telegram is slave specific. This buffer length has the symbolic name **`CONF_BUF_LEN`.** The *GENIpro Specification* requires.

The delay involved in EEPROM programming might in some implementations dictate a minimum time pr byte for the Application Program to empty the configuration buffer, depending on how much checking is done, and how the programming cycle is timed. If programming is carried out using shadow RAM no delay exists.

NOTICE! The Application Program may use EEPROM as well for backing up of other RAM data, that need to be preserved during power down. The handling of this can be done in common with the Configuration Parameters. The Application Program can f. ex. use a *Shadow RAM* area where both Configuration Parameters from *GENIpro* as well as application variables (status, mode) which need to be preserved are placed. The Application Program task taking care of this need not see any difference, but the application programmer should be aware of the conceptual difference.

GENIbus Unit

Must be able to buffer at least 6 Configuration Parameters if write via buffer is selected

C.8 APDU's for Class 5, Reference Values

Class Specification Values written to a slave from a master, These values are in some way used to operate the slave, like a reference to a control loop.

Categorizing: ID-codes 1-39: General Data Items, related to a Functional Profile
ID-codes 40-79: Application Specific Data Items

GET

Request from master

0	0	0	0	Class = 5
0	0	Length		
ID				
ID				
ID				
ID				
:				

Reply from slave

0	0	0	0	Class = 5
0	0	Length		
Value				
Value				
Value				
Value				
:				

SET

Request from master

0	0	0	0	Class = 5
1	0	Length		
ID				
Value				
ID				
Value				
:				

Reply from slave

0	0	0	0	Class = 5
0	0	Length		

INFO

Request from master

0	0	0	0	Class = 5
1	1	Length		
ID				
ID				
:				

Reply from slave

0	0	0	0	Class = 5
0	0	Length		
1 or 4 byte INFO Data				
1 or 4 byte INFO Data				
:				

Application comments Via a *Compile Time Option* the application programmer can choose to have Reference Values delivered by *GENIpro* in the buffer `ref_buf[]` instead of directly to a location in RAM. The length of this buffer limits the maximum number of Reference Values, that can be SET via a single Telegram. This length can differ between device types. This buffer length has the symbolic name **`REF_BUF_LEN`**..

GENIbus Unit Must be able to buffer at least 6 Reference Values if write via buffer is selected

C.9 APDU's For Class 6, Test Data

Class Specification Exclusively for factory tests and other special purposes.

GET

Request from master

0	0	0	0	Class = 6
0	0	Length		
ID				
ID				
ID				
ID				
:				

Reply from slave

0	0	0	0	Class = 6
0	0	Length		
Value				
Value				
Value				
Value				
:				

SET

Request from master

0	0	0	0	Class = 6
1	0	Length		
ID				
Value				
:				

Reply from slave

0	0	0	0	Class = 6
0	0	Length		

INFO Illegal Operation

Application comments It is the responsibility of the Application Program, to ensure that this Class is not enabled until the TEST command (Class 3, ID 20) is received. Disabling can be implemented as a timeout, by power up reset or by the USE command (Class 3, ID 19).

C.10 APDU's For Class 7, ASCII Strings

Class Specification

Character strings (max 63 bytes) represented in ASCII-code.
Only one ASCII String can be read in one APDU.

Categorizing: ID-codes 1-19: ASCII-Strings contained in one ID code
ID-codes 20-29: ASCII-Strings divided among two succeding ID codes

GET

Request from master

0	0	0	0	Class = 7
0	0	Length		
ID				

Reply from slave

0	0	0	0	Class = 7
0	0	Length		
ASCII String				

SET

Illegal Operation

INFO

Illegal Operation

Application comments

ASCII Strings are placed in memory with a trailing '\0' character (hex-value 0) to indicate "end of string". The '\0' character will be loaded into the APDU when the string is read if the string length is less than the maximum APDU length. Strings with a length that exceeds the maximum APDU length must be divided between more succeding ID codes. Without knowing the actual length of a string, a Bus Master can determine if the string is continued in the next ID code, by just testing the last character in the returned string.

The specification of the ASCII String Class reserves special ID-code areas to strings divided among more ID codes.

C.11 APDU's For Class 8, Memory Blocks

Class Specification

Used to transfer data which is not organized in Data Items but is a simple dump of the contents of a specified memory block (e.g. RAM, ROM, EEPROM).
Only one memory block transfer per telegram is allowed. More blocks are ignored.

GET

Request from master

0	0	0	0	Class = 8
0	0	Length		
Memory Specifier				
Dump address high				
Dump address low				
Dump length				

Reply from slave

0	0	0	0	Class = 8
0	0	Length		
Dump memory content				

SET

Request from master

0	0	0	0	Class = 8
1	0	Length		
Memory Specifier				
Dump address high				
Dump address low				
Dump length				
Dump memory content				

Reply from slave

0	0	0	0	Class = 8
0	0	Length		

INFO

Illegal Operation

Application comments

API
The Application Program must provide a function call for both reading (GET) and writing (SET) of all memory areas which should be supported.

TRUNCATION
If the specified "dump length" in a GET Operation is higher than the maximum APDU length the "dump memory content" will be truncated without any notice. The Bus Master must compare the returned amount of bytes with the requested amount to detect that a truncation has occurred.

REPLY WITH DATA NOT READY
This Data Class is allowed to reply to GENIbus that "Data is not ready" by returning an APDU with acknowledge code OK and a length of zero. The master must request the pending reply by using Class 8, GET, LENGTH=0 until data is ready. If the slave in the meantime is asked for anything else, the pending reply is aborted. A pending reply can be requested any number of times. Once it is ready the master will get it every time (with the original content)

C.12 APDU's For Class 9, Embedded PDU's

Class Specification

Used to transfer data which is not organised in Data Items but is an embedded PDU for routing. Used to directly access a sub network by embedding the PDU for the sub network in a GENIbus telegram. An Obvious application is the transfer of data between a GENIbus master and "sub processors" (like extension modules and "back end" computers) in a GENIbus slave.
Only one embedded PDU per GENIbus telegram is allowed. More embedded PDU's are ignored.

GET

Master request APDU

0	0	0	0	Class = 9
0	0	Length		
Routing Address (RA)				
Channel Specifier (CS)				
Embedded Request PDU				

Slave reply APDU when routing is busy

0	0	0	0	Class = 9
0	0	Length = 1		
Routing Status = 2				

Slave reply APDU when routing is finish

0	0	0	0	Class = 9
0	0	Length		
Routing Adress (RA)				
Channel Specifier (CS)				
Embedded Reply PDU				

SET

Illegal Operation

INFO

Illegal Operation

Application comments

The routing of the embedded Request PDU from GENIbus to the sub network is done automatically, so is the routing of the Reply PDU back to GENIbus. By the usage of a Routing Address and a Channel Specifier, embedded PDU's can be directed from GENIbus to any network member on any GENIpro channel:
0: GENIlink, 1: GENIbus, 2: GENIcom, 3: GENIpower, 4: GENI-RS232, 5: GENImodem

ROUTING STATUS
As long as the routing has not been succesfully fulfilled the routing reply will have a length of 1 and consist of nothing but the Routing Status. The Routing Status is a code telling why routing is not fulfilled (e.g. routing busy, routing failed, etc.)

REPLY TO CLASS 9 REQUESTS
This Data Class will, when requested the first time after request of any other class always reply to GENIbus that "Routing is ready" by returning an APDU with acknowledge code OK and a length of 1 and a Routing Status of 0. When an embedded request PDU is send in a data request telegram (see example next page) the reply to GENIbus will contain the Routing Status "Routing is initiated". The result of the routing (the routing reply) is requested, by simply using a GET APDU with length=0. The Routing Status in the reply will be "Routing busy" until the routing session has been completed. The unit can in the meantime be asked for any other class. This will not disturb the routing session. When the routing session has completed the reply to a class 9 request will be the routig reply.

ROUTED TELEGRAM FORMAT
When the PDU is routed it is supplied with a standard GENIpro telegram head (SD, LE, DA, SA). The Routing Address (RA) is used to generate the Destination Address (DA) for the routing. A check value (An 8 bit aritmetic sum for GENIcom, otherwise CRC-16) is added at the end.

ROUTING STATUS
0: Routing ready, 1: Routing initiated, 2: Routing busy, 3:not used, 4: Routing failed

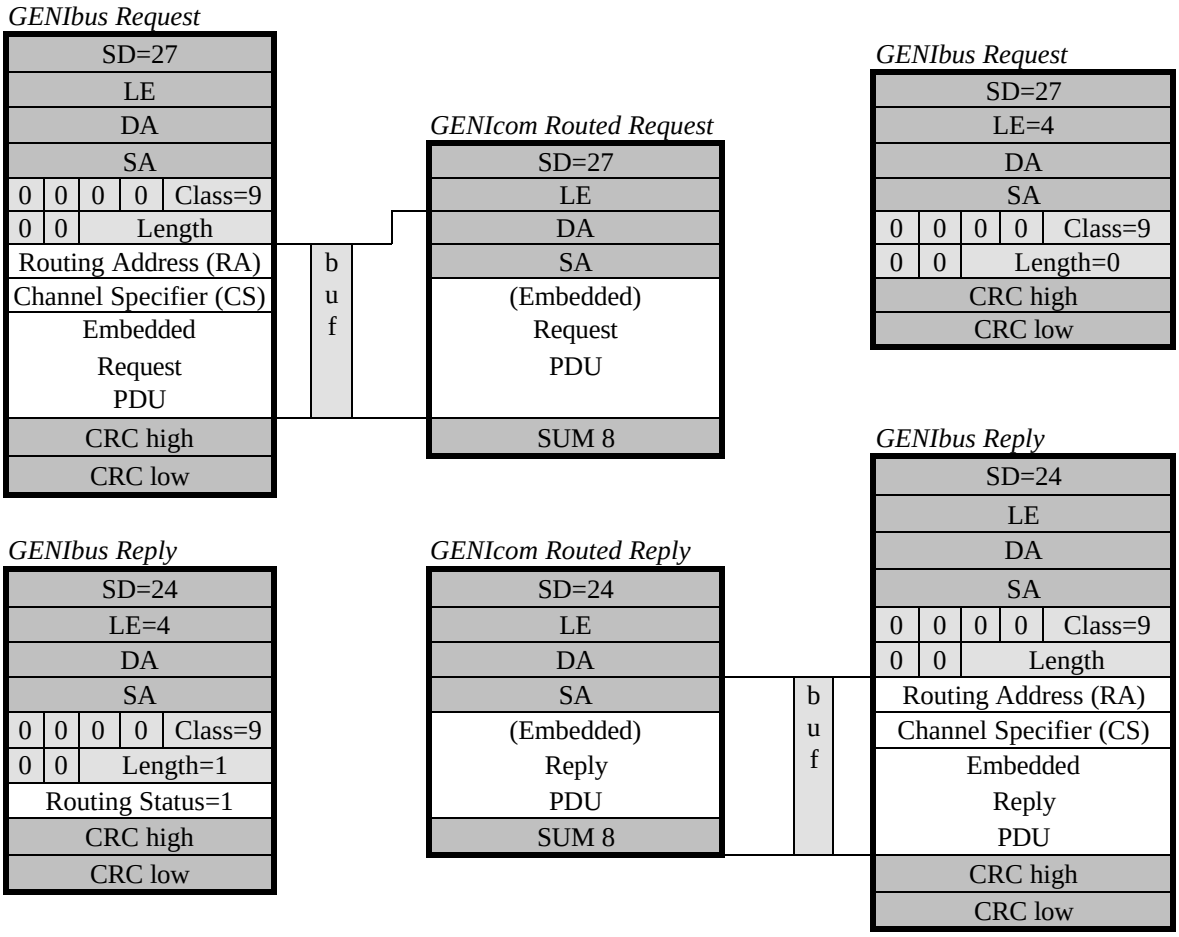
,

The sessions involved in a telegram routing

1.
GENIbus communication session for transfer of Embedded PDU Request

2.
GENIcom communication session for PDU routing

3.
GENIbus communication session for transfer of Embedded PFU Reply



1.
A GENIbus Data Request transfers an Embedded Request PDU. The GENIbus Class 9 Data Reply APDU will always signal "Routing initiated" by using an APDU length of 1 and a Routing Status of 1. Data Reply APDU's to other APDU's in the request telegram (not shown here) will have their normal contents. GENIpro delivers the Embedded Request PDU and the Routing Address in a buffer. The routing from one GENIpro channel to another is handle by GENIpro itself and is hidden for the Application Program. CS selects the GENIpro channel (se previous page) and RA is used as Destination Address.

2.
The GENIcom channel is used here for the further transport of the embedded PDU - now as a regular non-embedded PDU. This further transport could take place on any GENIpro channel (e.g. GENIpower channel). GENIpro delivers the reply PDU in a buffer. Here it will wait until requested from GENIbus.

3.
To request a pending routed reply a Class 9 APDU with length zero is used. If the routed reply is not ready a reply APDU with length=1and Routing Status=2 (Routing busy) will result. If everything has turned out succesfully, then the shown GENIbus reply will result. If the routing due to some reason could not be fulfilled the reply APDU will only contain the Routing Status with Routing Status=4 (Routing failed).

C.13 APDU's For Class 10, Data Objects

Class Specification

Used to transfer data which is organised in objects and referred to by Data Items. The data in the object is not of any particular type or length, but can have any form only limited by the maximum APDU length of 63 bytes.
Only one object transfer per GENIbus telegram is allowed. More objects are ignored.

GET

Request from master

0	0	0	0	Class = 10
0	0	Length		
ID				

Reply from slave

0	0	0	0	Class = 10
0	0	Length		
Data Object				

SET

Request from master

0	0	0	0	Class = 10
1	0	Length		
ID				
Data Object				

Reply from slave

0	0	0	0	Class = 10
0	0	Length		

INFO

Illegal Operation

Application comments

API
The Application Program must provide a function call for both reading (GET) and writing (SET) for each supported Data Item.

REPLY WITH DATA NOT READY
This Data Class is allowed to reply to GENIbus that "Data is not ready" by returning an APDU with acknowledge code OK and a length of zero. The master must request the pending reply by using Class 10, GET, LENGTH=0 until data is ready. If the slave in the meantime is asked for anything else, the pending reply is aborted. A pending reply can be requested any number of times. Once it is ready the master will get it every time (with the original content)

C.14 APDU's For Class 11, 16 bit Measured Data

Class Specification: Data Items generated in the unit. They can be read (GET) but not written (SET) from a master. Contains information about the unit and the application (system) in which it takes part, Can generally be values or bitwise interpreted data.

Categorizing:
- ID-codes 1-20: Application Specific Data Items
- ID-codes 21-99: General Data Items, related to a Functional Profile
- ID-codes 100-140: Application Specific Data Items
- ID-codes 141-255: General Data Items, related to a Functional Profile

GET:

Request from master

0	0	0	0	Class = 11
0	0	Length		
ID				
ID				
ID				
ID				
:				

Reply from slave

0	0	0	0	Class = 11
0	0	Length		
Value				
Value				
Value				
Value				
:				

SET: Illegal Operation

INFO:

Request from master

0	0	0	0	Class = 11
1	1	Length		
ID				
:				

Reply from slave

0	0	0	0	Class = 11
0	0	Length		
1 or 4 byte INFO Data				
:				

Application comments:

C.15 APDU's for Class 12, 16 bit Configuration Parameters

Class Specification

Data Items stored in the slave in none volatile memory (e.g. EEPROM, ROM. *GENIpro* always delivers Configuration Parameters in RAM, either in an *API Buffer* or directly in Application Program RAM (defined by *CTO's*). It is the responsibility of the *Application Program* to perform the actual EEPROM programming.

Categorizing: ID-codes 1-20: Application Specific Data Items
ID-codes 21-99: General Data Items, related to a Functional Profile
ID-codes 100-140: Application Specific Data Items
ID-codes 141-255: General Data Items, related to a Functional Profile

GET

Request from master

0	0	0	0	Class = 12
0	0	Length		
ID				

:

Reply from slave

0	0	0	0	Class = 12
0	0	Length		
Value				

:

SET

Request from master

0	0	0	0	Class = 12
1	0	Length		
ID				
Value				

:

Reply from slave

0	0	0	0	Class = 12
0	0	Length		

INFO

Request from master

0	0	0	0	Class = 12
1	1	Length		
ID				

:

Reply from slave

0	0	0	0	Class = 12
0	0	Length		
1 or 4 byte INFO Data				

:

Application Comments

Via a *Compile Time Option* the application programmer can choose to have Configuration Parameters delivered by *GENIpro* in the buffer `conf16_buf[]` instead of directly to a location in RAM. The size of this buffer is device dependent so the maximum number of Configuration Parameters, which can be SET via a single Telegram is slave specific. This buffer length has the symbolic name **`CONF_BUF16_LEN`.** The delay involved in EEPROM programming might in some implementations dictate a minimum time pr byte for the Application Program to empty the configuration buffer, depending on how much checking is done, and how the programming cycle is timed. If programming is carried out using shadow RAM no delay exists.

NOTICE! The Application Program may use EEPROM as well for backing up of other RAM data, that need to be preserved during power down. The handling of this can be done in common with the Configuration Parameters. The Application Program can f. ex. use a *Shadow RAM* area where both Configuration Parameters from *GENIpro* as well as application variables (status, mode) which need to be preserved are placed. The Application Program task taking care of this need not see any difference, but the application programmer should be aware of the conceptual difference.

GENIbus Unit

A GENIbus Unit must be able to buffer at least 6 16 bit Configuration Parameters if write via buffer is selected.

C.16 APDU's for Class 13, 16 bit Reference Values

Class Specification Values written to a slave from a master, These values are in some way used to control the slave, like a reference to a control loop.
Categorizing: ID-codes 1-39: General Data Items, related to a Functional Profile
ID-codes 40-39: Application Specific Data Items

GET

Request from master

0	0	0	0	Class = 13
0	0	Length		
ID				
ID				
ID				
:				

Reply from slave

0	0	0	0	Class = 13
0	0	Length		
Value				
Value				
Value				
:				

SET

Request from master

0	0	0	0	Class = 13
1	0	Length		
ID				
Value				
ID				
Value				
:				

Reply from slave

0	0	0	0	Class = 13
0	0	Length		

INFO

Request from master

0	0	0	0	Class = 13
1	1	Length		
ID				
ID				
:				

Reply from slave

0	0	0	0	Class = 13
0	0	Length		
1 or 4 byte INFO Data				
1 or 4 byte INFO Data				
:				

Application comments Via a *Compile Time Option* the application programmer can choose to have Reference Values delivered by *GENIpro* in the buffer `ref_buf16[]` instead of directly to a location in RAM. The length of this buffer limits the maximum number of Reference Values, that can be SET via a single Telegram. This length can differ between device types. It has the symbolic name **`REF_BUF16_LEN`**..

GENIbus Unit Must be able to buffer at least 6 16bits Reference Values if write via buffer is selected.

C.17 Error Reporting in a Data Reply APDU

The *GENIpro Data Link Layer* checks a received *Data Request* for Checksum Error, Length Error and Receive Break Error. These errors are termed *Framing Errors.* Any error found in the Data Link Layer implies that the whole Telegram is discarded and nothing is led through to the *Presentation Layer*. When a *PDU* is handed over to the Presentation Layer it is regarded as free of transmission errors. In other words it can be treated as identical with what the transmitting unit actually send.

Error handling in the Presentation Layer is therefore a matter of sorting out - on the APDU level - requests that cannot be fulfilled or requests that are in conflict with the *GENIpro Specification*, and reporting these errors in the *Data Reply*. Notice, that any Data Request making it through to the Presentation Layer, whatever the number of requests it might contain which cannot be fulfilled, will result in the generation of a Data Reply.

Errors are treated on an APDU basis, meaning that errors in one APDU will in no way influence the reply to sound APDU's. When some unfulfilable request is detected in an APDU the normal reply to this APDU will be substituted with an *Error Reply APDU* containing information about the error. Sound requests in a *GET* or *INFO APDU* will be lost if just one "Unknown Data Item" exists. Data to sound ID-codes in a *SET APDU* will however be delivered down to the point where the first "unknown Data Item" is found. This erroneous ID code will be returned and the following ID codes discarded.

The three basic kinds of errors are contained in the *ACK-field* in the head of the Error Reply APDU. This Head is identical with the standard APDU Head shown at page C.2. Each error kind has its own format for the Data Field, as shown in the figure below (the *ACK-field* is shown shaded). A table of possible *causes* for the three kinds of errors is also shown below.

Unknown Class

0	*	0	0	Class
0	1	Length=0		

Unknown Data Item

0	*	0	0	Class
1	0	Length=1		
ID for 1'st unknown Data Item				

Illegal Operation

0	*	0	0	Class
1	1	Length=1		
ID for 1'st inaccesible Data Item				

Error kind	Operations	Possible causes
Unknown Class	SET, GET, INFO	• The Class in question does not exist
Unknown Data Item	SET, GET	• No entry to this ID code exists in the Data Item Pointer Table for the Class in question. • Memory Specifier unknown (Class 8) • Unknown channel (Class 9)
	INFO	• No entry to this ID code exists in the Data Item INFO Table for the Class in question.
Illegal Operation	SET	• Buffer full for the Class in question. • SET access not enabled by the Application Program for the Class in question. • Access key not valid for write protected Conf. Param.
	GET	• GET access not enabled by the Application Program for the Class in question.

C.18 The RFS Option

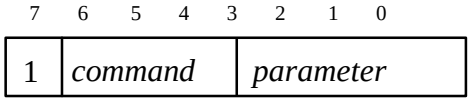
The *RFS-option* is a strong means to give a slave a chance to send a single byte message to the Bus Master "wrapped into" an ordinary Data Reply. Two situations takes advantage of this option.

- A Master Unit Type currently in Slave Mode can request to be a *Submaster* (SM_rfs) for a short duration of time specified in the *RFS-option*. Having received this request, the Bus Master always sends an RFS-acknowledge(SM_ack) back to the slave. This acknowledge is a copy of the request, and is send alone in a Message telegram. RFS-acknowledges can not be queued in the slave nor can pending RFS be stored in the Bus Master. Both have to be acted uppon (consumed) when received. When *GENIpro* finds it necessary get Submaster Status (to fulfil requests from the Application Program) an RFS-option will be appended to to all Data Replies it transmits until it eventually receives an RFS-acknowledge.
  From the moment where the *Bus Master* sends the RFS acknowledge, it becomes *passive* for the requested time.
- A Master Unit Type can request to be a Bus Master (BM_rfs). This goes on the same way as described in 1) except that control is not returned to the previous Bus Master.

The Application Program does not have to worry about sending an RFS Option and receiving an RFS-Acknowledge. This mechanism is totally handled by *GENIpro* and hidden behind a simple function call in the API. Lets assume that the Application Program in a bus unit operating as a Submaster wants to perform a poll cycle at its logical Subnetwork. This is what happens:

- The Application Program calls the `SendTgm` API-procedure with address information and a pointer to the Telegram to send. This is all the application programmer has to worry about. The rest of this description takes place "behind the curtains".
- This function will take care that *GENIpro* is "loaded" with the RFS-option (which contains a request to be a Submaster).
- The next time *GENIpro* makes a Data Reply to a Bus Master, it will add the RFS-option.
- The Bus Master receives the RFS-option, interprets it, sends an RFS Acknowledge back to the unit and remains passive for the time `t_subm`.
- The unit receives the RFS-acknowledge and has status as a network Submaster.
  GENIpro can now do the actual transmission of the Telegram, receive the reply and deliver it to the Application Program. This will all be completed within the time `t_subm`.

The *RFS-option* always has a "1" in the bit 7 position to distinguish it from the APDU Head that always has a "0" in this position. 7 bits are left for *RFS-information* as shown in the figure below. These are split between a command field (3 bits) and a parameter field (4 bits). Only one command has been defined yet.



RFS-command

0: transfer of bus control requested
parameter: *t_subm* = *[1; 15]*, means request to be Submaster.
Time specified in steps of 50ms.
t_subm = *0*, means request to be a *Bus Master*.

C.19 Cyclic Redundancy Checking, CRC

This description follows Ref /1/ pages 69-79 and pages 770-786.

C.19.1 Theory

Redundancy Checking means the addition of bits to a Telegram that somehow represents the contents and transmitting these bits along with the Telegram itself. *Cyclic* steams from the way these bits, normally referred to as the *CRC-value*, are generated.

The method used in *GENIpro* is a 16 bit CRC using the standard *Generator Polynomial* from the International Telecommunications Union, CCITT*). Further a CRC-value initialization to all 1's and a post CRC inversion which makes the checking procedure identical with the popular IBM protocol SDLC+).

The basic for all CRC-value generation is a long division in which the Telegram is treated as if it was one single large binary number:

(1) Telegram/Divisor = Quotient + Remainder

The Remainder in this calculation reveals extremely good properties when it comes to representing a compact "fingerprint" of the Telegram. This will be more clear later. By selection of the Divisor we can control the size of the Remainder - it will by definition always be at least one less than the divisor.

When implemented the division is not performed as an arithmetic division but as a *Modulo-2 Division*. Although it gives another result for the Remainder it has the same properties and - which is the main reason, it is simple to implement in hardware or software. The only procedural difference is that the intermediate results in the long division are obtained by *Exclusive-OR* instead of subtraction.

The former formula then leads to the expression below where some of the names have been substituted with their equivalent CRC-term:

(2) Telegram $/^{M2}$ Generator Polynomial = Quotient + CRC-value

Definition:
A 16 bit CRC-value is the Remainder obtained from a 16 bit Modula-2 Division of the Telegram with a Generator Polynomial in which the Telegram, has been extended with 16 zeros.

The extension with zeros is necessary to force the last 16 bits to participate in the division. Otherwise there would be a simple one to one relationship between these bits and bits in the CRC value. The idea of calling the divisor a polynomial comes from the mathematics theory which usually describes CRC generation in the language of polynomials. There is nothing more to it than expressing the binary equivalent of the value as coefficients to a polynomial in X where the exponent for each term is derived from the ordinal rank of the bits with value 1. The CCITT Generator Polynomial used in *GENIpro* will serve as an illustrating example:

CCITT Generator Polynomial: 0x1021 = 0001.0000.0010.0001 ~ $X^{16} + X^{12} + X^5 + 1$

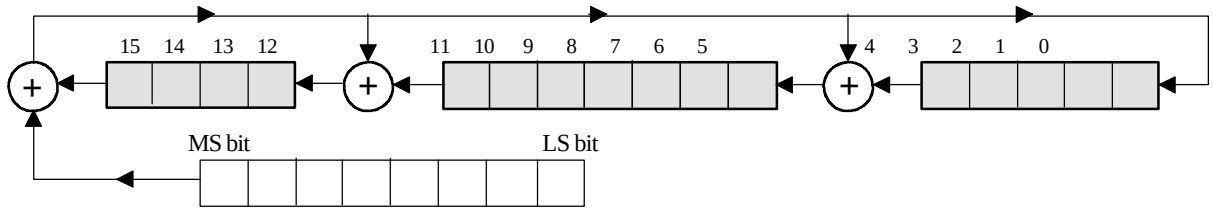
Notice! When specifying the polynomial form the term X^{16} is included to guarantee a 16 bit remainder.
Other polynomials can be used giving different error detection capability. The statistics for this one is indeed very good.

Single bit errors:	100%
Double bit errors: 100%	
Odd numbered errors:	100%
Burst errors shorter than 16 bits:	100%
All other burst errors:	>99.997%

*) Comité Consultatif International de Télégraphique et Téléphonique
+) Synchronous Data Link Control, this protocol was the basis for the ISO HDLC protocol and later the CCITT X25 network Interface Standard

C.19.2 Implementation

The figure below shows a hardware equivalent of the CRC generation. It can be a great help in trying to understand the mechanism. The construction of the circuit makes it unnecessary to extend the Telegram with 16 zeros. This is so to say already taken into account. The dark shaded 16 bit register is called the *CRC-Accumulator*, it holds the Remainder of each stage in the Modulo-2 Division. When the whole Telegram has been shifted into the machine the Accumulator will hold the CCITT version of the CRC-value. CCITT specifies an initialization of the CRC-Accumulator with all zeros. In *GENIpro* we use the SDLC approach initializing with all 1's and inverting the CRC-accumulator before transmitting the CRC-value as mentioned earlier. This makes the CRC resistant to leading erroneous zeros and to merged Telegrams, which wouldn't otherwise be the case.



Telegram bytes processed one by one

Our aim is to write a function that implements this behaviour in software. A first attempt could be:

```
ushort crchware(ushort data, ushort accum)
     {
     uchar i;
     data <<=8;
     for (i=8; i>0; i--)                    /* Do for each bit:          */
         {
         if ((data ^ accum) & 0x8000)       /* IF a 1 in feedback path   */
            accum = (accum << 1) ^ genpoly; /*    feedback interaction   */
         else                               /* ELSE                      */
            accum <<= 1;                    /*    transparent shift      */
         data <<=1;                         /* Make next bit ready       */
        }
     return accum;
     }
```

Each byte in the Telegram which is to take part in the cyclic redundancy check is passed to the function one by one along with the value of the Accumulator. When the last byte has been passed the return value will be the CRC-value for the Telegram. Data has been declared as a 16 bit value (instead of uchar) due to its presence in a 16 bit expression.

It can be seen now, by studying the CRC circuit or the function above, that when applying a new byte to the CRC circuit the feedback path will not be influenced by the existing low order byte of the Accumulator. Only the high order Accumulator byte interacts with the data bits. We refer to the result of this XOR'ing as the *combining value*. This leads to the observation that the new Accumulator is equal to the CRC of the combining value XOR'ed with the unchanged half of the Accumulator. This relationship can be expressed in C.

```
comb_val = (accum >>8) ^ data;
tmp = crchware(comb_val,0);
accum = tmp ^ (accum <<8);
```

Since there are only 256 possible combining values, it would be a good idea to calculate their CRC's in advance and store them in a table, `crctab[256]`, thereby saving a great deal of run time computer power. The following piece of code uses `crchware` to generate the lookup table.

```
void main(void)
   {
   unsigned short i,j;
   printf("\nunsigned short crctab[256] = \n    {");
   for (j=0; j<=31; j++)
        {
        printf("\n    ");
        for (i=8*j; i<8*j+8; i++) printf("%6u,  ", crchware(i,0));
```

```
        }
    printf("\n    }");
```

By redirecting the output to a file you have the table ready to paste into the protocol source code. The CRC calculation now takes this form.

```
comb_val = (accum >>8) ^ data;
tmp = crctab[comb_val];
accum = tmp ^ (accum <<8);
```

Combining this into a more compact form leads to the final CRC function. The Accumulator has been removed from the arguments and made a global variable, to avoid the overhead of passing it to the function for each byte to process.

```
ushort accum;
void crc_update(uchar data)
   {
   accum = (accum <<8)^ crctab[(accum >>8) ^ data];
   }
```

When a Telegram is transmitted the CRC-Accumulator is initialized to 'all ones' and each byte, except the *Start Delimiter*, is processed through the `crc_update` function before being send to the Drivers. Finally the CRC-Accumulator is inverted and its two bytes are appended to the Telegram with high order byte first. These two bytes are what we define as the *CRC-Value*.

The receiver performs a similar procedure. Initialises the CRC-Accumulator to 'all ones'. Then, each byte received, except the *Start Delimiter*, is processed through the `crc_update` function. When the CRC-Value bytes arrive they are inverted and then also processed through `crc_update`. If the CRC-Accumulator hereafter is equal to zero the received Telegram is considered as sound.

C.20 GENIcom Specification

Description	GENIcom is primarily specified to be used as a communications standard between different parts of a Grundfos device (e.g. front end to back end, mother board to add on board, etc.)	
Telegram Format	Header:	GENIpro Header (figur C.1)
	Data Field:	GENIpro PDU (figur C.1)
	Check Field:	8 bit aritmetic sum excluding Start Delimiter
Hardware	- Not specified -	
Topology	- Not specified -	
Transmission distance	- Not specified -	
No. of bus units	- Not specified -	
Timing	Baudrate:	- Not specified -
	Inter Byte Delay:	Max 1.2ms @ 9600bits/s; Changed inversly proportional with baudrate
	Inter Data Frame Delay:	Min 3ms @ 9600bits/s; Changed inversly proportional with baudrate
	Reply Delay:	[3ms; 50ms] @ 9600bits/s; Changed inversly proportional with baudrate
Medium Access	Master/slave; only one master	
Bus Allocation	Dedicated master	
Adress field	0:	reserved
	[1; 254]:	unit address
	255:	broadcast address

GENIcom unit	Telegram buffer	>= 35 bytes
	Command buffer	>= 3 bytes
	Configuration Parameter buffer	>= 3 bytes
	Reference Value buffer	>= 3 bytes
	16 bit Configuration Parameter buffer	>= 6 bytes
	16 bit Reference Value buffer	>= 6 bytes

C.21 Telegram Examples

All examples are Data Requests from a master assumed to have Unit Address 0x04 to a UPE pump, assumed to have Unit Address 0x21.

Data Request

Start Delimiter	0x27
Length	0x0B
Destination Address	0x21
Source Address	0x04

Class 2: Measured Data	0x02
OS=0 (GET), Length=7	0x07
`f_act` = ID 32	0x20
`p` = ID 34	0x22
`h` = ID 37	0x25
`q` = ID 39	0x27
`act_mode1` = ID 81	0x51
`energy_cons_hi` = ID 152	0x98
`energy_cons_lo` = ID 153	0x99

CRC high	0xCC
CRC low	0x5A

Data Reply

Start Delimiter	0x24
Length	0x0B
Destination Address	0x04
Source Address	0x21

Class 2: Measured Data	0x02
Ack.=0 (OK), Length=7	0x07
value example of `f_act`	0xA4
value example of `p`	0x84
value example of `h`	0xC1
value example of `q`	0x39
value example of `act_mode1`	0x00
value example of `energy_cons_hi`	0x12
value example of `energy_cons_lo`	0x06

CRC high	0xF6
CRC low	0xA9

Data Request

Start Delimiter	0x27
Length	0x0B
Destination Address	0x21
Source Address	0x04

Class 3: Commands	0x03
OS=2 (SET), Length=3	0x83
`REMOTE` = ID 7	0x07
`START` = ID 6	0x06
`CONST_PRESS` = ID 24	0x18
Class 5: Reference Values	0x05
OS=2 (SET), Length=2	0x82
`ref_rem` = ID 1	0x01
value example	0xA8

CRC high	0xDE
CRC low	0xF9

Data Reply

Start Delimiter	0x24
Length	0x06
Destination Address	0x04
Source Address	0x21

Class 3: Commands	0x03
Ack.=0 (OK), Length=0	0x00
Class 5: Reference Values	0x05
Ack.=0 (OK), Length=0	0x00

CRC high	0x6F
CRC low	0x79

Data Request

Start Delimiter	0x27
Length	0x0B
Destination Address	0x21
Source Address	0x04

Class 2: Measured Data	0x02
OS=3 (INFO), Length=7	0xC7
`f_act` = ID 32	0x20
`p` = ID 34	0x22
`h` = ID 37	0x25
`q` = ID 39	0x27
`act_mode1` = ID 81	0x51
`energy_cons_hi` = ID 152	0x98
`energy_cons_lo` = ID 153	0x99

CRC high	0xC7
CRC low	0x0B

Data Reply

Start Delimiter	0x24
Length	0x1A
Destination Address	0x04
Source Address	0x21

Class 2: Measured Data	0x02
Ack.=0 (OK), Length=22	0x16
`f_act` INFO Head	0x82
`f_act` UNIT	0x84
`f_act` ZERO	0xC1
`f_act` RANGE	0x39
`p` INFO Head	0x82
`p` UNIT	0x09
`p` ZERO	0x00
`p` RANGE	0x01
`h` INFO Head	0x82
`h` UNIT	0x25
`h` ZERO	0x00
`h` RANGE	0x06
`q` INFO Head	0x82
`q` UNIT	0x23
`q` ZERO	0x00
`q` RANGE	0x05
`act_mode1` INFO Head	0x81
`energy_cons_hi` INFO Head	0x82
`energy_cons_hi` UNIT	0x40
`energy_cons_hi` ZERO	0x00
`energy_cons_hi` RANGE	0x01
`energy_cons_lo` INFO Head	0xB0

CRC high	0xA8
CRC low	0xA9

D.1 Overview of Software Design

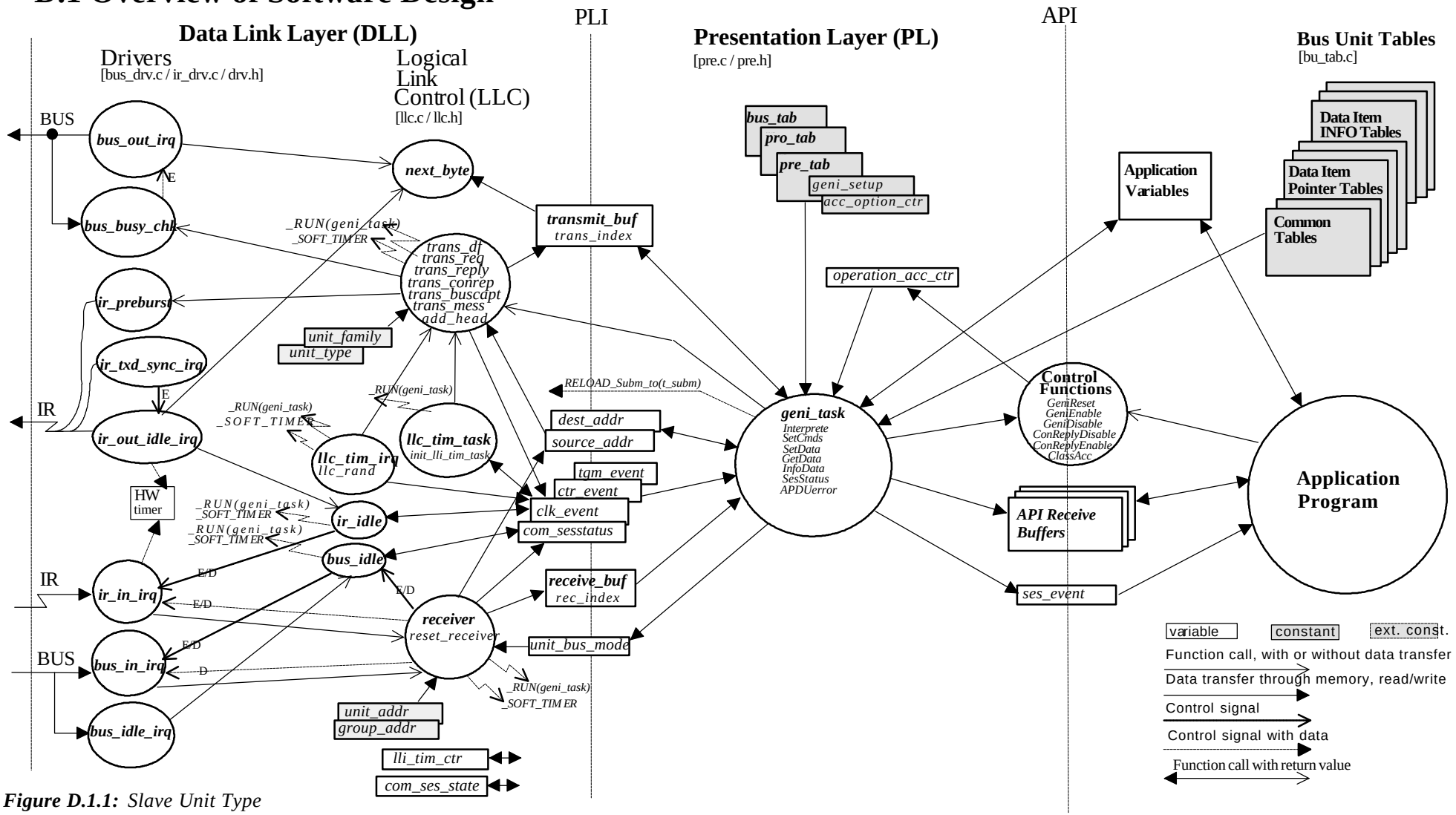


Figure D.1.1: *Slave Unit Type*

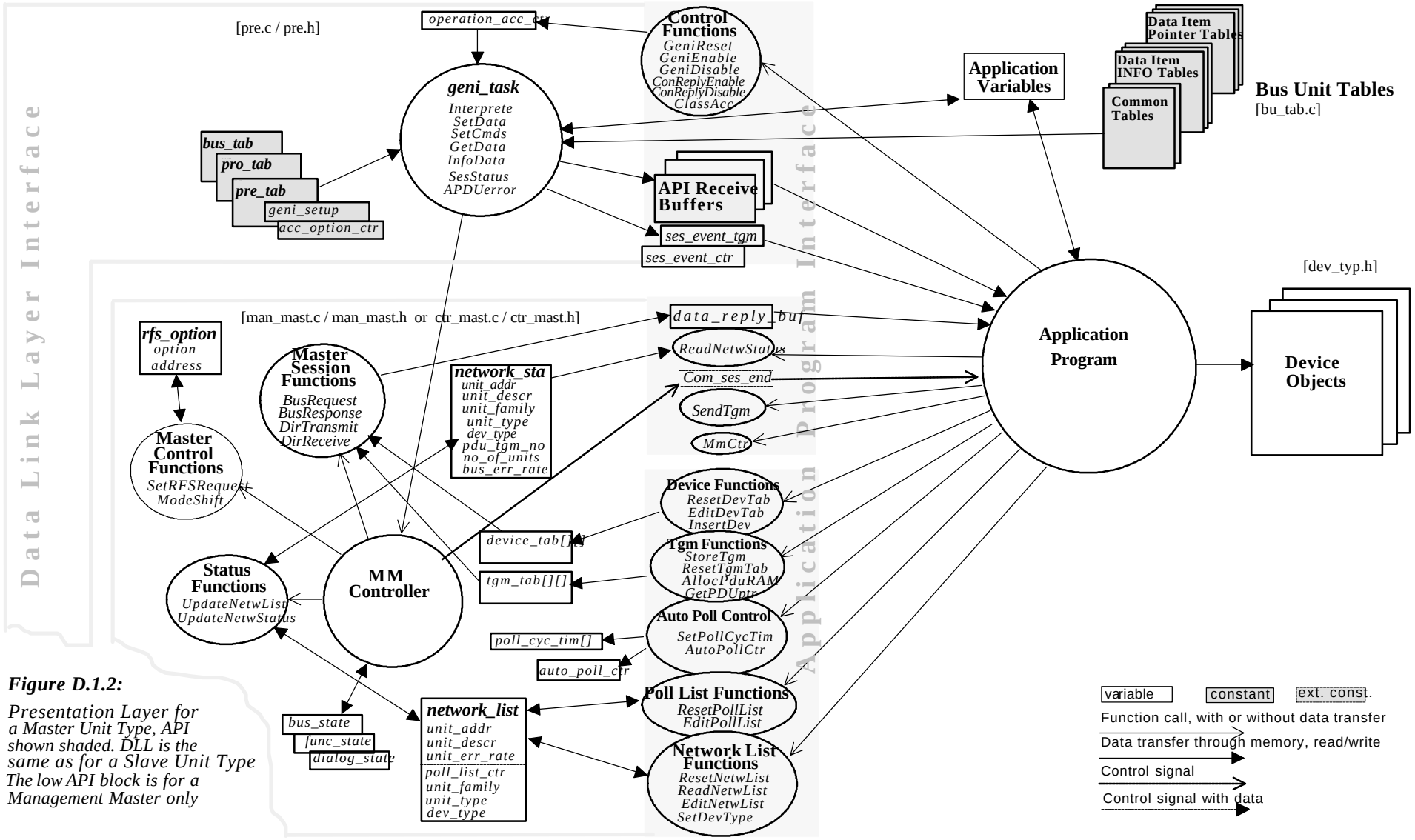


Figure D.1.2:

Presentation Layer for a Master Unit Type, API shown shaded. DLL is the same as for a Slave Unit Type The low API block is for a Management Master only

D.2 The Physical Layer

D.2.1 RS485 Bus Communication, the *GENIbus* Channel

An application example for RS485 hardware for a *GENIbus Unit* is shown in figure D.2.2. Designing of the Bus Idle Detection circuit means selection of proper values for R_1, R_2 and C to fulfil the *GENIpro* Specification. The following calculations are illustrated by figure D.2.1. . The basics from where to start is the Kirchoff current law used on the capacitor C:

(D.2.1) $d/dt\ (v_C)\ C = i_1 - i_2 - i_{bid}$

When the communication line idles, C will be charging. For v_C to ever reach the Idle Detection level $v_{bid,hi}$, the equation:

$$v_{RxD,hi} + v_d > v_{bid,hi}$$

must be fulfilled. This means i_2 is zero. The leakage current i_{bid} can be neglected when

$$i_{bid} << 5V/R_1$$

Using these assumptions in *(D.2.1)* gives the charging equation to solve:

$$d/dt\ (v_C)\ C = (5V - v_C)/R_1$$
$$d/dt\ (v_C) = -v_C/R_1C + 5V/R_1C$$

Solving this as a first order differential equation using the condition $v_C(0) = v_{C,min} = v_d + v_{RxD,lo}$ will lead to:

(D.2.2) $v_C\ (t) = 5V + (v_d + v_{RxD,lo} - 5V)\ exp(-t/R_1C)$

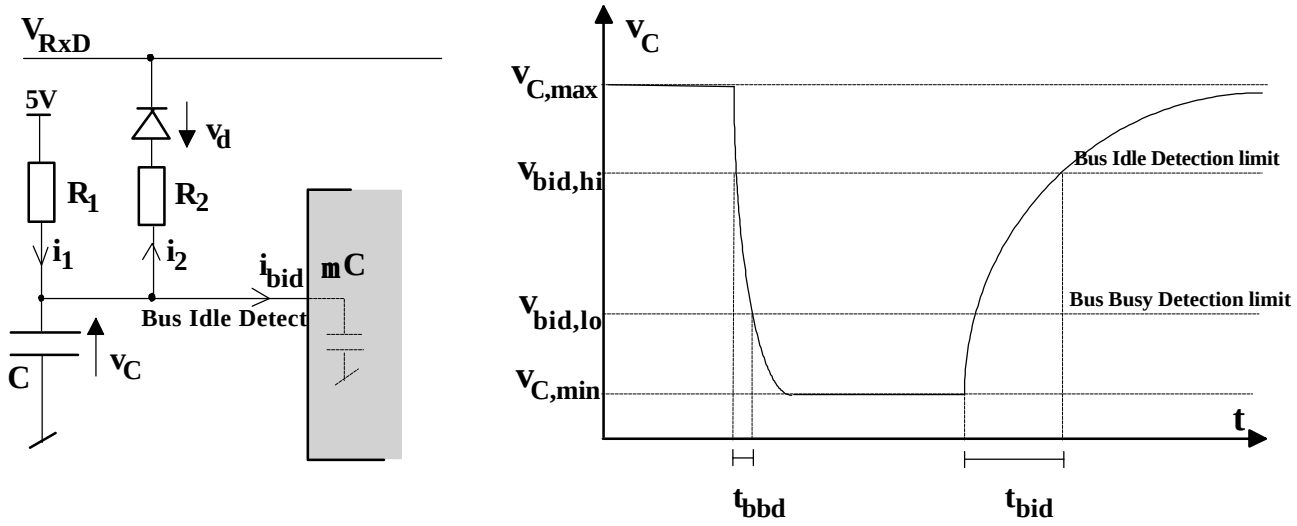


Figure D.2.1: *The Bus Idle Detection circuit and the time behaviour of the Bus Idle Detection signal. The signal is connected to an input port, which must be digitally readable from the GENIbus Drivers to recognize a Bus Busy event. The input port must as well be able to work as a positive edge triggered interrupt to generate a Bus Idle Detect event to the GENIbus Drivers when the bus idles. If a port with this dual functionality is not available, the two events can be detected on one port each instead.*

Apparently v_C reaches 5V. In practice however, due to the fact that i_{bid} is not equal to zero, v_C will saturate at a value $v_{C,max}<5V$. We will neglect this and use *(D.2.2)* as a good approximation to find a formula for t_{bid}:

$$v_{bid,hi} = v_C(t_{bid})$$
$$v_{bid,hi} = 5V + (v_d + v_{RxD,lo} - 5V)\ exp(-t_{bid}/R_1C)$$

(D.2.3)

Examining this result will reveal, that the logarithmic expression is normally close to 1. t_{bid} can then, not surprising, be considered as equal to the time constant built by R_1C:

(D.2.4) $t_{bid} =\sim R_1 C$

The Bus Idle Detection Time should equal the transmission time for 2 data bytes. Taking the *Data Format* 10 bits/byte into consideration this means: 2 •10bit • 1/(9600bit/s) = 2.1 ms.

This gives enough security not to detect idle for a data byte with "all ones" and at the same time only introducing a very short delay between Telegrams before idle is detected (the start bit (=0) will guarantee the necessary discharge of v_C between bytes). A deviation of 0.6 ms can be tolerated:

(D.2.5) $t_{bid} = 2.1\ ms +/- 0.6$ (design goal)

Selecting f. ex. **R_1=100kW** and **C=22nF** will give t_{bid}= 2.2ms.

When calculating v_C during discharge the starting point is again equation *(D.2.1).* As before we make some assumptions to ease the calculations. The charging contribution i_1 through R_1 can be neglected if

$$R_1 >> R_2$$

The Bus Idle Detection leakage current i_{bid} can be neglected if

$$i_{bid} << 5V/R_2$$

Using these assumptions and substituting the expression for i_2:

$$d/dt\ (v_C)\ \ C = -(v_C - v_d - v_{RxD,lo}) / R_2$$
$$d/dt\ (v_C) = -v_C/R_2C + (v_d+v_{RxD,lo})/R_2C$$

Solving this as a first order differential equation using the condition: $v_C(0) = v_{C,max}$ will lead to:

(D.2.6) $v_C\ (t) = v_d + v_{RxD,lo} + (v_{C,max} - v_d - v_{RxD,lo})\ exp(-t/R_2C)$

When the discharge has finished, the capacitor C holds the voltage

$$v_{C,min} = v_d + v_{RxD,lo}$$

Which must be below the Idle Detection low threshold level $v_{bid,lo}$, which is normally not a problem.
The discharge must be fast to have a short Bus Busy Detect Delay Time t_{bbd}. If this value is chosen to be considerably less than the duration of 1 bit, the occurrence of a start bit (=0) can be regarded as sufficient to discharge C completely down to $v_{C,min}$. With this in mind the following value has been chosen:

(D.2.7) $t_{bbd} < 35ms +/- 15ms$ (design goal)

Using (D.2.6) to solve the equation $v_C(t_{bbd}) = v_{bid,lo}$ yields:

(D.2.6) $v_{bid,lo} = v_d + v_{RxD,lo} + (v_{C,max} - v_d - v_{RxD,lo})\ exp(-t_{bbd}/R_2C)$

Now substituting typical values:

$$v_{C,min} = v_d + v_{RxD,lo} = 0.5$$
$$v_{C,max} = 4.5V$$
$$v_{bid,lo} = 1.5V$$

leads to:

$$1.5V = 0.5V + 4V\ exp(-t_{bbd}/R_2C)$$

(D.2.7)

Selecting f. ex. **R_2=1kW** and using the value of **C=22nF** from before will give t_{bbd}= 30.5*ms*

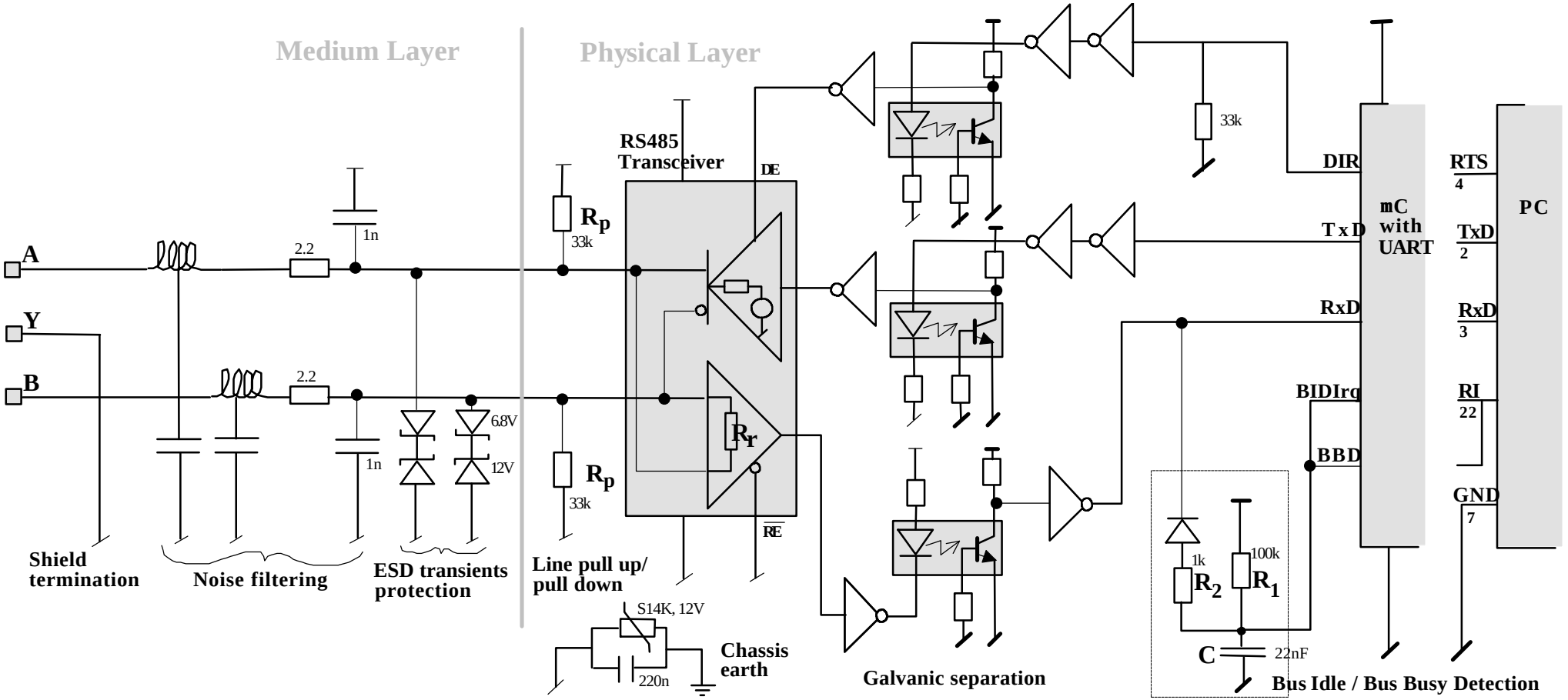


Figure D.2.2: *An application example of the GENIbus hardware with galvanic separation. The galvanic separation is necessary when the micro controller is galvanically connected to the mains. The protocol specification of a RS485 based Physical Layer normally ends with the RS485 Transceiver. The additional circuitry relates to transmission line conditions, and is therefore to be considered as a part of the Medium Layer. An individual circuit design can be chosen depending on the selected cable type and environmental conditions. The shown circuit can serve as the basis for design calculations of component values.*
When calculating the Unit Load equivalent, notice that the effective DC-impedance equals $R_r||(2R_p)$.

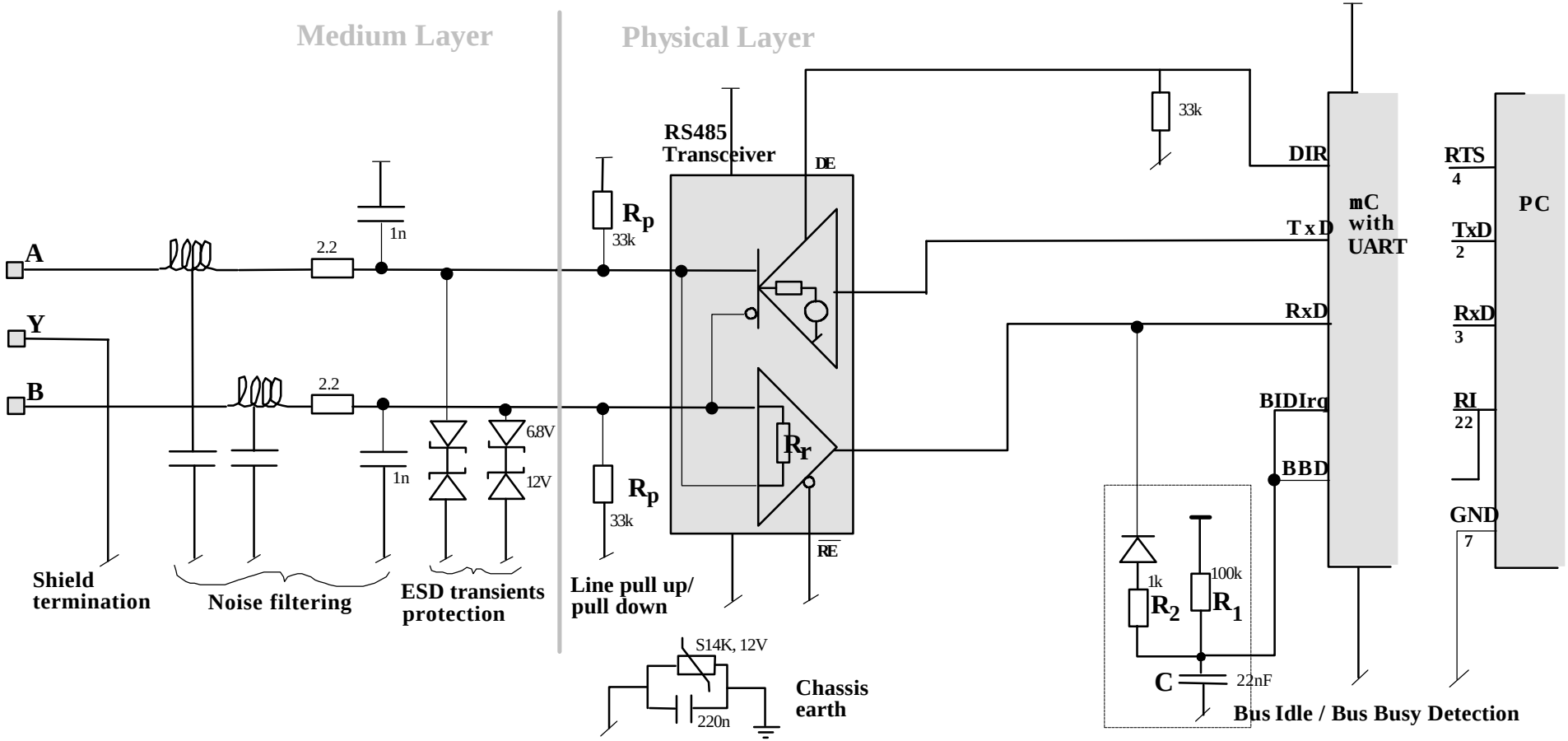


Figure D.2.3: *An application example of the GENIbus hardware without galvanic separation between micro controller and RS485 transceiver. This is the normal circuit when the micro controller is galvanically isolated from the mains.*
The protocol specification of a RS485 based Physical Layer normally ends with the RS485 Transceiver. The additional circuitry relates to transmission line conditions, and is therefore to be considered as a part of the Medium Layer. An individual circuit design can be chosen depending on the selected cable type and environmental conditions. The shown circuit can serve as the basis for design calculations of component values.
When calculating the Unit Load equivalent, notice that the effective DC-impedance equals $R_r||(2R_p)$.

D.2.2 Implementing Target Specific *GENIbus* Drivers

GENIbus Driver Function	Description
`void bus_drv_init(void)`	Initializes the bus channel hardware. Setting UART to 9600baud, 1 start bit (=0), 1 stop bit (=1), no parity
`void txd_driver_off(void)`	Switching the Transceiver Driver off line (DIR signal)
`uchar bus_busy_chk(void)`	Checks for "Bus Busy". If bus is available TxD Irq is enabled (this must start transmission). Return 0 if bus is available else return 1.
`uchar random_no(void)`	Returns a pseudo random No. Preferrably the reading of an 8 bit free running counter
`uchar inter_df_delay(void)`	Returns the value (in measure of T_{inc}) which the driver demands to be used for the Inter Data Frame Delay
`void disable_rxd_irq(void)`	Disables the reception of bytes from the bus channel by disabling the UART "receive register full" interrupt
`void enable_rxd_irq(void)`	Enables the reception of bytes from the bus channel by enabling the UART "receive register full" interrupt
`void disable_txd_irq(void)`	Disables the transmission of bytes on the bus channel by disabling the UART "transmit register empty" interrupt.
`void enable_txd_irq(void)`	Enables the transmission of bytes on the bus channel by enabling the UART "transmit register empty" interrupt.
`void disable_bus_idle_irq(void)`	Disables the "Bus Idle Detection" interrupt
`void enable_bus_idle_irq(void)`	Enables the "Bus Idle Detection" interrupt
`void disable_all_irq(void)`	Disables the whole interrupt system. No reaction to any interrupt events
`void enable_all_irq(void)`	Enables the whole interrupt system. Interrupt events are being processed
`void bus_out_irq(void)`	UART "Tansmit register empty" interrupt function
`void bus_in_irq(void)`	UART "Receive register full" interrupt function
`void bus_idle_irq(void)`	Bus Idle Detection Interrupt function

Table D.2.1: *List of Bus Driver Interface Functions*

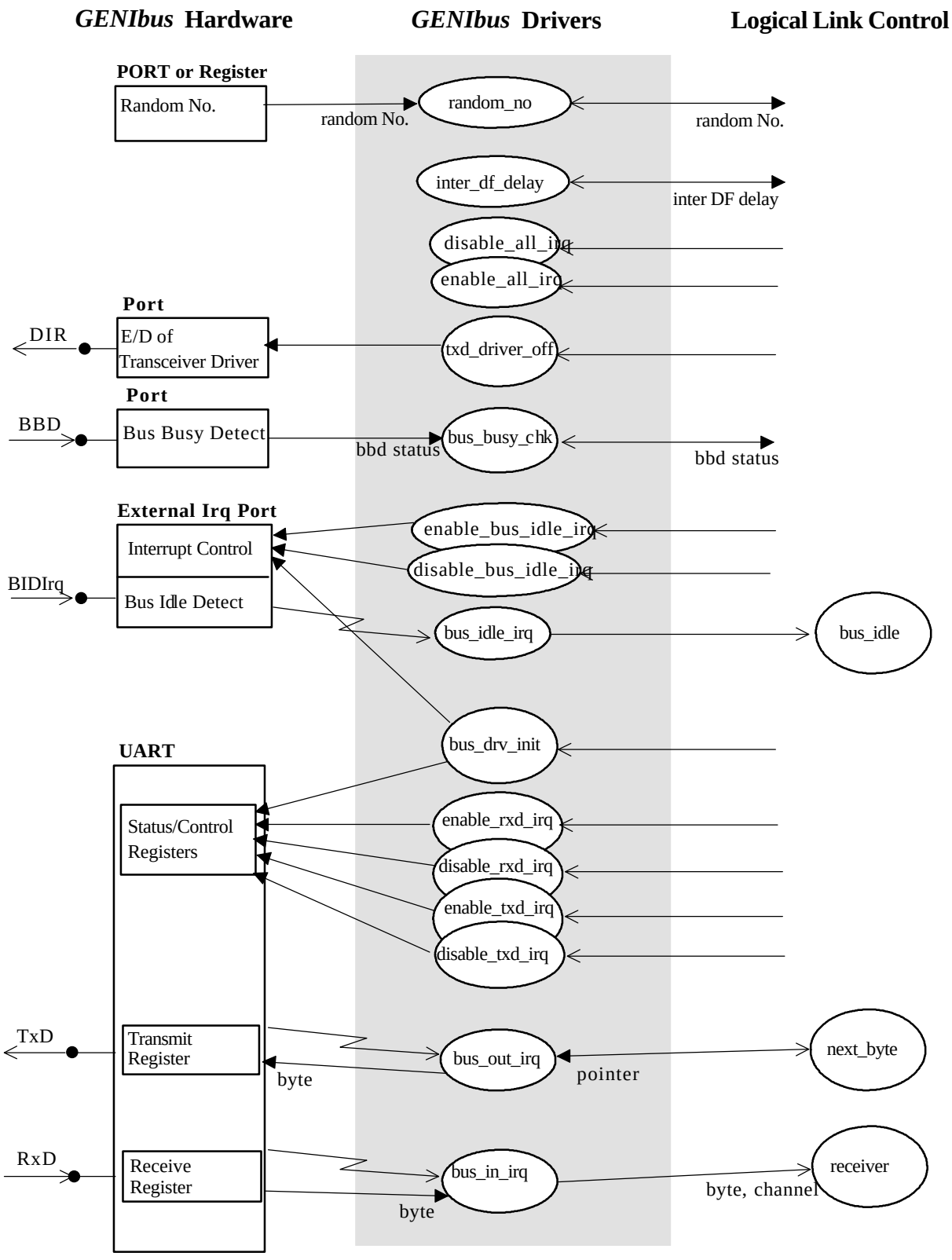


***Figure D.2.4**: Data flow diagram of The GENIbus Driver Interface to hardware and Logical Link Control*

D.2.3 Infra Red Remote Communication, the *GENIlink* Channel

Bits are coded in pairs according to the pulse width modulation scheme shown below. A pulse ("mark") is defined as six cycles of the pulse carrier frequency 38.4kHz. A "space" is defined to have the same length in time as a "mark" but containing no pulse carrier (Frequency Shift Keying). The "mark" to "space" duty cycle must normally be kept low. The scheme below guarantees it to be maximum 33%.

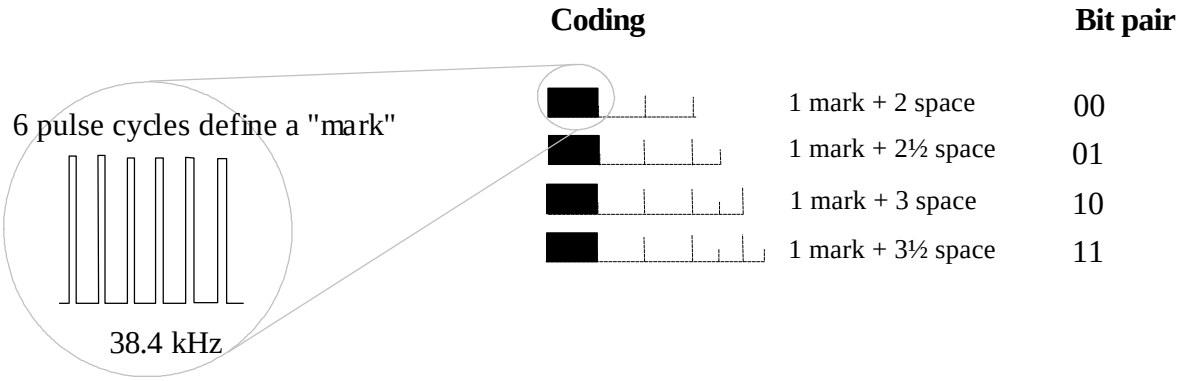


Figure D.2.5: *Bit coding in bit pairs using a mark/space modulation scheme.*

A start sequence with a preburst - equal to 14 pulse carrier cycles - and a start delay time t_{sd} = 4 marks (625µs) is generated whenever a transmit session is started. This is used for Automatic Gain Control (AGC) in the receiver. The example below shows how the two bytes 0x27, 0x45, considered to be the beginning of a Telegram, are coded.

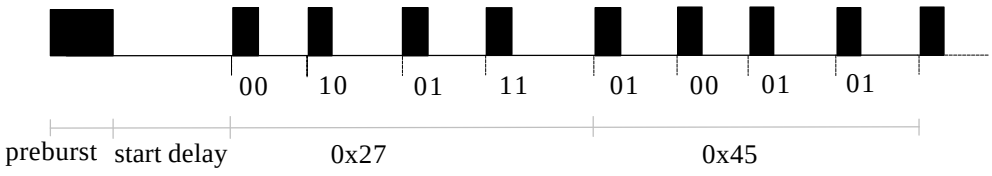


Figure D.2.6*: Example showing how the two first bytes in a Telegram are coded.*

The average baud rate can be calculated if each bit pair are considered to occur with the same likelihood:

(D.2.8) = *1707 baud (bit pairs pr second)*

The transfer of four bit pairs is equivalent to the transfer of 10 bits over *GENIbus*. When comparing with the *GENIbus* baud rate a fair measure has to take this into account:

(D.2.9)

A way of implementing the described behaviour is shown in figure D.2.6 on the next page. A serial synchronous channel is used for the generation of "marks". A capacitor converts the clock signal to short pulses, and the data channel is used to clamp the clock pulses. By transmitting the byte 11000000 the duration of a "mark" will be truncated to 6 clock cycles. The received "marks" generate a negative going edge which triggers the negative edge interrupt. A timer is used to calculate the time between "marks" and thus decode the received bit pairs.

The interrupt latency must be so low, that one bit pair can not be confused with another. The time difference between, say bit pair '00' and '01' is only the half of a "mark". So not to mistake '01' for '00' means:

interrupt latency < ½•½•t_{mark}=1/4 • 6 • 1/38.4kHz

(D.2.10) interrupt latency < 39.1µs

If the 38.4kHz is not realized exactly this extra deviation means that the interrupt latency must be even smaller.

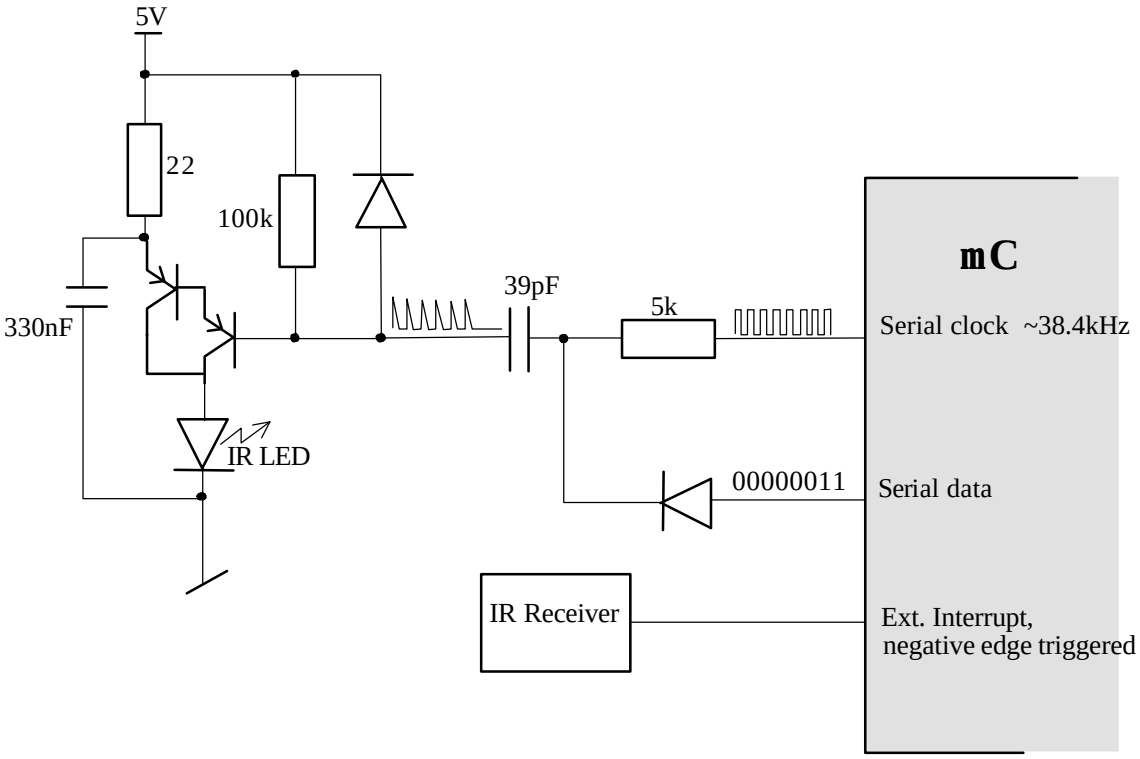


***Figure D.2.7**: Application example of GENIlink hardware. A serial synchronous channel on the micro controller is used for the generation of 38.4kHz 6 pulse marks.*

D.2.4 Implementing Target Specific *GENIlink* Drivers

GENIlink **Driver Function**	**Description**
`void ir_drv_init(void)`	Initialize the *GENIlink* channel hardware
`void ir_preburst(void)`	Generates the IR preburst and starts the transmission on the *GENIlink* channel
`void disable_ir_irq(void)`	Disables the reception of byte from the *GENIlink* channel
`void enable_ir_irq(void)`	Enables the reception of byte from the *GENIlink* channel
`void disable_ir_idle_irq(void)`	Disable the *GENIlink* Idle Detection Interrupt
`void disable_all_irq(void)`	Disables the whole interrupt system. No reaction to any interrupt events This function must be excluded if CTO_BUS is enabled
`void enable_all_irq(void)`	Enables the whole interrupt system. Interrupt events are being processed This function must be excluded if CTO_BUS is enabled
`uchar inter_df_delay(void)`	Returns the value (in measure of T_{inc}) which the driver demands to be used for the Inter Data Frame Delay. This function must be excluded if CTO_BUS is enabled
`void ir_out_idle_irq(void)`	*GENIlink* Transmit ready and *GENIlink* Idle Detection interrupt function
`void ir_in_irq(void)`	Received *GENIlink* byte ready interrupt function

Table D.2.2: *List of GENIlink Driver Interface Functions.*

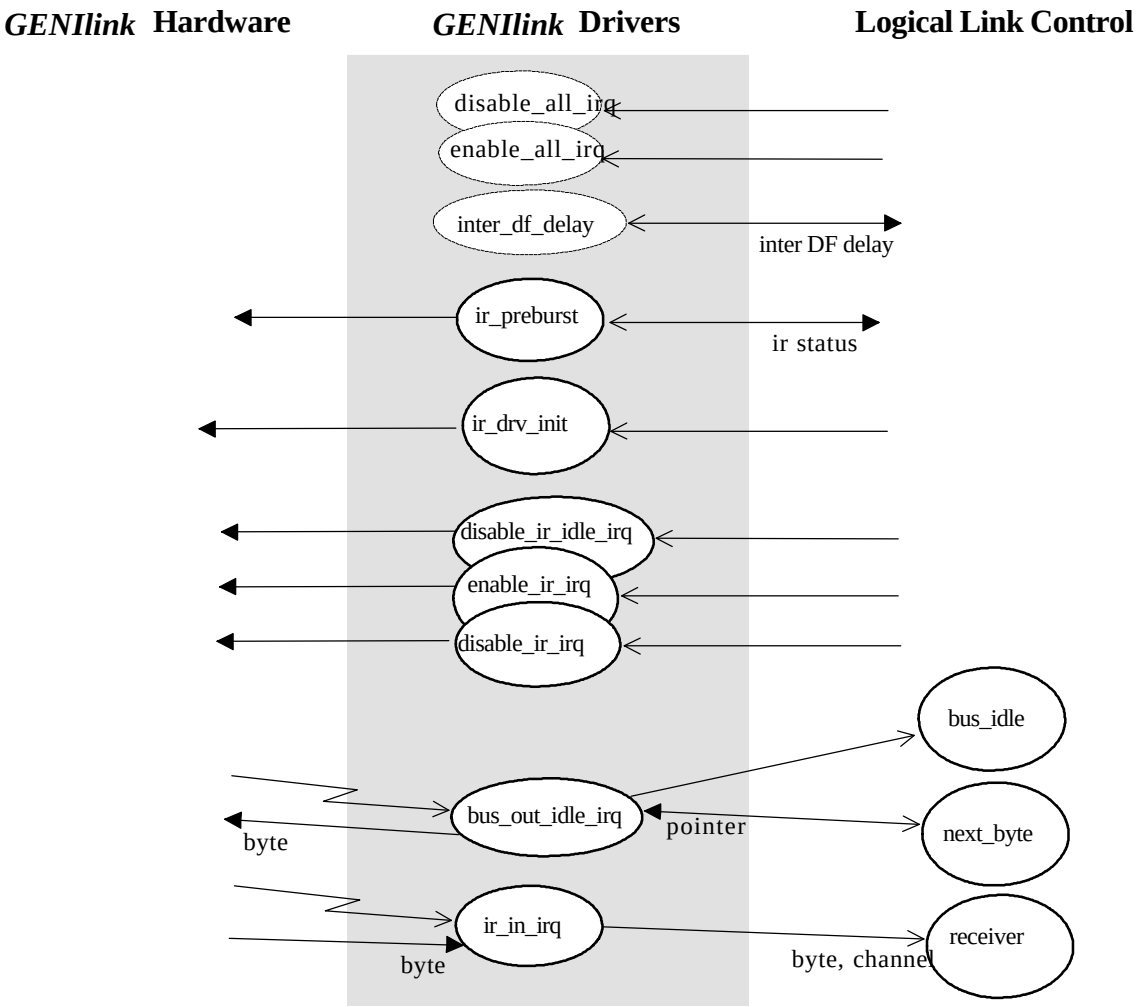


Figure D.2.7*: Data flow diagram of The GENIlink Driver Interface for hardware and Logical Link Control. Functions shown hatched must be excluded if Compile Time Option* `CTO_BUS=Enable`*. No hardware blocks are shown because the necessary functionality can be implemented in many different ways depending on the peripherals of the used target computer.*

D.4 The General Operating System Interface

E. *GENIpro* Technical Specifications Summary

GENIpro in general:

Type:	Dual Channel Fieldbus protocol	
Layer Model:	Physical Layer, Data Link Layer, Presentation Layer - according to the OSI reference model	
Data Orientation:	8 bit	
Data Organization:	In Data Classes with dedicated properties	
Address Allocation:	1-29:	29 Master Unit Types
	32-231:	200 Master Unit Types or Slave Unit Types
	232-247:	16 groups
	254:	Connection Address
	255:	Global Address

Physical Layer:

***GENIbus channel*:**	**Topology:**	Bus (multidrop)
	Hardware standard:	EIA RS485, half duplex. Bus Busy Detection and Bus Idle Detection (see D.2.1).
	Coding:	NRZ (none return to zero)
	Data Format:	Start bit (=0), 8 data bits with least significant bit first, stop bit (=1)
	Baudrate:	9600 +/- 4%
	Transmission Dist.:	Max 1200m
	No of bus units:	Max 32 according to EIA RS485
***GENIlink channel*:**	**Hardware standard:**	Infra Red light, λ= 950nm. Half duplex
	Coding:	PWM coding of bit pairs, Pulse Carrier: 38.4kHz +/- 1kHz
	Data Format:	4 data bit pairs (see D.2.3). 1 preample for each Telegram consisting of preburst (14 carrier cycles) and start delay (625µs). Used for automatic gain control (AGC) and synchronizing of receiver and transmitter.
	Transmission Rate:	Baudrate equivalent to 10 bit format: avarage: 4267 bit/s
	Transmission Dist.:	Not specified

Data Link Layer (DLL):

Bus Configuration: Automatic appointment of the Master Unit Type with the lowest Unit Address to be Bus Master.
Bus Master Automatically recognizes bus members.

Retransmission: Yes, 1 time

Presentation Layer (PL):

Bus Allocation: Dynamic, controlled by Bus Master.
Submaster gets allocated time when it wants to service its Subnetwork.

Medium Access: Hybrid: Master/slave
- Delegated token: Slave -> RFS Option requesting Submaster Status, SM_rfs
  Bus Master -> RFS-acknowledge to be Submaster, SM_ack
- Token passing: Slave -> RFS Option requesting Bus Master Status, BM_rfs
  Bus Master -> RFS-acknowledge to be Bus Master, BM_ack

Bus Unit Types: Master Unit Type: can operate in Slave Mode, Submaster Mode and Bus Master Mode
Slave Unit Type: can operate in Slave Mode and Subslave Mode

Bus Arbitration under Bus Operation: Central (controlled by current Bus Master). Default priority mechanism gives Bus Master Token to The Master Unit Type with the lowest Unit Address.

Minimum Buffer Requirements:

Minimum length of Data Frame Buffer for a *GENIbus Unit*:	70 byte (PDU up to 66 bytes possible)
Maximum No. of SET values alway allowed in a Telegram to a *GENIbus Unit*:	6 bytes

Telegram Specifications:

Format: **Head** (4 bytes): *Start Delimiter* (SD): 0x27: Data Request, 0x26: Message, 0x24: Data Reply
Length (LEN): No. of following bytes excluding CRC-value
Destination Address (DA)
Source Address (SA)

Data Field (=PDU): 0-253 bytes organized in Aplication Program Data Units (APDU's, see C.2) and one optional Request From Slave byte (RFS, see C.13).

APDU: *Class Specifier:*
0: Protocol Data
1: Bus Data
2: Measured Data
3: Commands
4: Configuration Parameters
5: Reference Values
6: Test Data
7: ASCII Strings
8: Dump Memory

Operation Specifier, Opr (in request):
0: GET
1: -
2: SET
3: INFO

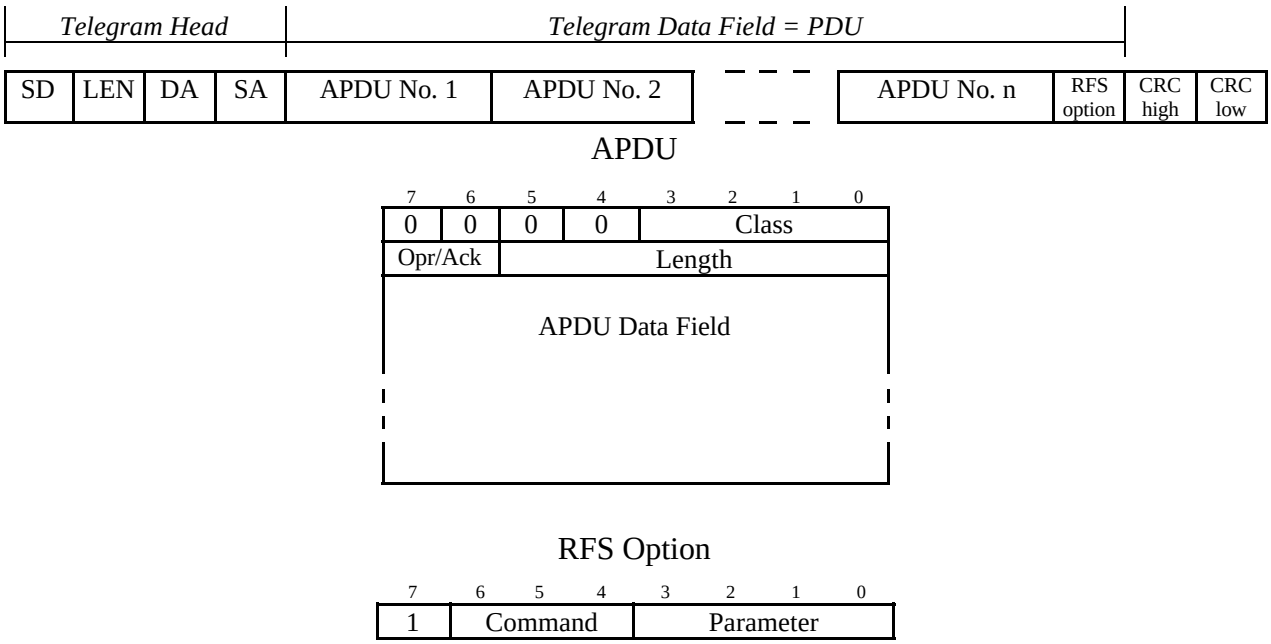
Acknowledge Code, Ack (in reply):
0: OK
1: Class illegal
2: Data Item illegal
3: Operation illegal

APDU Length Specifier: [0;63]

RFS: *RFS Command:* 0: Request Submaster/Bus Master Status
RFS Parameter: 0: Request Bus Master Status, BM_rfs
[1;15]: No. of requested 50ms Submaster time slots, SM_rfs

CRC-value: 16 bit CCITT. polynomium 0x1021. SD excluded. Initialised to 0xFFFF. CRC bits inverted. High order byte transmitted first (See C.14).

Max Values: Total Telegram: 257 bytes
PDU: 253 bytes
APDU: 63 bytes



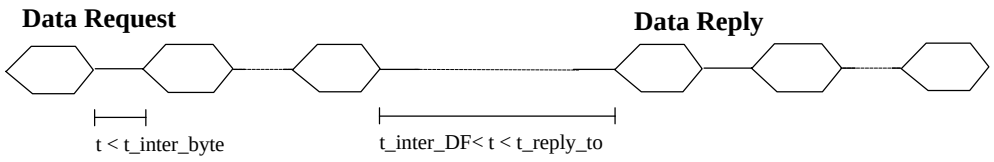
Figur E.1: *Visualizing the format of a Telegram, an APDU and the RFS option.*

	Data Request/Message		Data Reply	
GET	**Class: 0,1,2,4,5,6,7**		**Class: 0,1,2,4,5,6**	**Class: 7**
	ID ID ID :		value value value :	ASCII string ASCII string ASCII string :
SET	**Class: 3**	**Class: 4,5**	**Class: 3,4,5**	
	ID ID ID :	ID value ID value :	APDU Data Field empty	
INFO	**Class: 2,3,4,5**		**Class 2,4,5**	**Class: 3**
	ID ID ID :		1 or 4 INFO bytes 1 or 4 INFO bytes 1 or 4 INFO bytes :	1 INFO byte 1 INFO byte 1 INFO byte :

Figure E.2: *Specification of APDU Data Field. For details see Appendix C.*

Timing Specifications			
Symbol	**Description**	**Value**	**Implemented**
t_{bid}	*Bus Idle Detection Time* Time for the Idle Detection Mechanism to detect an idling bus	min 1.5 ms max 2.7 ms	Phy. Layer for Bus
t_{bbd}	*Bus Busy Detection Time* Time passing before a busy bus can be detected following a Bus Idle	min 20µs max 50µs	Phy. Layer for Bus
t_{inter_byte}	*Inter Byte Delay* Time slot between two consecutive bytes in a Telegram	max 1.2 ms	LLC for Bus
t_{inter_DF}	*Inter Data Frame Delay* Time slot between any two Telegrams	min 3 ms	LLC for Bus (and IR)
t_{reply_to}	*Reply Timeout* Time delay for a slave to start replying to a Data Request	max 50ms	LLC for Bus and IR
$t_{bus\_act\_to}$	*Bus Activity Timeout* Time the bus can idle before *GENIpro* effects a *Bus Configuration*	2.5s	LLC for Bus
$t_{max\_unpolled}$	*Unpolled Timeout* Time a slave accepts not to receive a Data Request before it records itself as Unpolled (becomes sensitive to a *Connection Request*)	20s	LLC for Bus
$t_{conn\_rep\_to}$	*Connection Reply Timeout* Random delay of Connection Reply in an Unpolled slave	random: 3-43ms	LLC for Bus
$t_{master\_app\_to}$	*Master Appointment Timeout* Random delay in Master Appointment Procedure	Unit Addr.*4ms	LLC for Bus
t_{subm}	*Submaster Time Slot* Submaster time slot as requested/acknowledged by the RFS-option	[1-15] * 50ms	LLC for Bus

Table E.1: *Specification of all timing related to a GENIbus Unit*



Figur E.2: *Visualizing of the timing demands in a GENIbus Communication Session*

F. Explanation to *GENIpro* Terms

F.1 Concepts

Access Control	The *Application Program* for a unit in *Slave Mode* can control *GENIpro's* access to the *API Receive Buffers* when data are delivered and it can control *GENIpro's* access to Application Memory when data are being fetched. The Classes 2-6 can independently be enabled/disabled for the SET and GET *Operation.*
API Receive Buffers	Common name for the three buffers `cmd_buf[]`, `conf_buf[]` and `ref_buf[]` where *GENIpro* delivers data to the *Application Program.*
Application Program	Common name for that part of the software, which performs the application specific functions, and interacts with *GENIpro* through the *API*.
Application Program Interface	API. Name for the boundary between the *Application Program* and *GENIpro.* Consists of common data structures and of functions through which the Application Program can access certain internal *GENIpro* data structures in a protected fashion and start Communication Sessions. This is called API Services.
Application Program Data Unit	APDU. Basic format for data transfer in *GENIpro.* Contains a *Class* No., an *Operation* and the *ID-codes* to perform the Operation on.
Auto Poll Cycle	A *Master Unit Type* performs a poll cycle of those bus units which are members of the *Poll List.* This is done automatically by *GENIpro* in the background. This Auto Poll Cycle and the *Telegrams* used are under *Application Program* control for a *Management Master*. For a *Control Master* the only *Telegram* used is the *BUR-Telegram* and the cycle can not be controlled from the *Application Program.*
Auto Poll Tables	These tables completely defines the Auto Poll Cycle in a Management Master. They consist of the Auto Poll Telegram Table, the Auto Poll Device Table and the Auto Poll Timer Register.
Buffer Options	These are *CTO*'s in file `geni_cnf.h` specifying in what way data in *Classes* 3-5 should be delivered (SET operation) from *GENIpro* to the *Application Program* for a unit in *Slave Mode.*
Bus	A network topology where all devices can exchange data directly with each other (=multidrop)
Bus Activity Timeout	The excess of the max allowed time, $t_{bus\_act\_to}$ which the bus must be left idle. This leads to a *Bus Configuration.*
Bus Arbitration Mechanism	The mechanism deciding which unit is to be *Bus Master*. In Configuration Mode the arbitration mechanism is decentral. All *Master Unit Types* follows a *Master Appointment* procedure until a *Bus Master* is found. The unit with the highest priority (=lowest address) will be appointed. In Operation Mode the Arbitration Mechanism is central - it is controlled by the current Bus Master. The default priority during bus operation also gives *Bus Master Status* to the *Master Unit Type* with the lowest *Unit Address*.
Bus Capture Message	A special *Message Telegram* transmitted by a *Master Unit Type* in *Slave Mode* at the moment during *Bus Configuration* where *GENIpro* appoints it to be *Bus Master*.
Bus Configuration	The process of *Master Arbitration* following a *Bus Activity Timeout*.
Bus Idle Detection	The physical bus idles when no transmission is taking place. Bus Idle Detection is used by *GENIbus* Units to detect the end of a *Telegram*. Receivers in all units on the bus will be synchronized.
Bus-IRQ-Receiver-Driver	`bus_irq_in()`. Interrupt driven software driver for controlling the peripheral hardware devices (UART, RS485 line drivers) driving the physical bus when receiving. Receives single bytes and deliver them to the *LLC* which controls the *Receive Buffer*. Also called a MAC Receive Driver.
Bus-IRQ-Transmitter-Driver	`bus-irq_out()`. Interrupt driven software driver for controlling the peripheral hardware devices (UART, RS485 line drivers) driving the physical bus when transmitting. Requests single bytes from the *Transmit Buffer* in the *LLC* and transmits them to the bus. Also called a MAC Transmit Driver.

Bus Master Acknowledge	The acknowledging of a Request to be a Bus Master. This comes in the form of an *RFS-acknowledge* (BM_rfs) from the present *Bus Master*. A *Control Session*.
Bus Master Mode (=Bus Master)	A *GENIbus* unit operating in this mode has the control of the bus. Only one unit connected to the bus can, at a given moment, operate in Bus Master Mode.
Bus Mode	The *GENIbus* distinguishes between four modes for connected units: *Bus Master Mode*, *Slave Mode*, *Submaster Mode* and *Subslave Mode*.
Bus Slave Mode (=Bus Slave)	A *GENIbus* unit operating in this mode behaves as an obedient slave, it only communicates when requested by the *Bus Master*.
Bus Unit Record	An entry In the *Network List*. Contains information about a specific bus unit.
Bus Unit Record Telegram	BUR-Telegram. A special *Telegram* requesting exactly those *Data Items* contained in a *Bus Unit Record*: `group_addr`, `unit_bus_mode`, `unit_family` and `unit_type`. The BUR-Telegram is used for *Connection Request*.
Bus Unit Tables	Common name for the tables which in a given bus unit constitute the data content of its *Functional Profile*. These tables are a part of the data interface between *GENIpro* and the *Application Program*. Built by the application programmer.
Bus Unit Type	A *Master Unit Type* or a *Slave Unit Type*. Selection is done via the *Compile Time Option* `CTO_BUS_UNIT_TYPE` or via Configuration Parameter *geni_setup*.
Class	Data Items to transfer via *GENIpro* are organized in different Classes, each defining what kind of *Operations* can be carried out (SET, GET, INFO) and what format the data has. Some examples of classes are Measured Data and Commands.
Common Tables	Tables containing the scaling information. Used by INFO *Operation*. Belongs to the *Bus Unit Tables* and are built by the application programmer.
Communication Session	The process of transferring data or control in the form of a *Telegram* between a master/slave couple. Consisting of a Transmit Session and a Receive Session. Communication Sessions are generally divided into *Data Sessions* and *Control Sessions*.
Control Master	A *Master Unit Type* with limited master functionallity and limited API. Used for typical *Submaster* applications. A *Control Master* results from a *GENIpro* compilation with the *Compile Time Option*: `#define CTO_MM Ctr_Mast`
Control Session	The process of transferring bus control between a master/slave couple.
Compile Time Options	CTO. Used by the application programmer to select or deselect parts of *GENIpro* code during compilation. As an example: whether a given *GENIbus Unit* is a *Master Unit Type* or a *Slave Unit Type* is a matter of the chosen Compile Time Option. CTO's are primarily a way to minimize ROM-usage by only including the functionality required by the application.
Connection Reply	The reply from a Slave to a *Connection Request*
Connection Reply Timeout	The *Connection Reply* is delayed the time $t_{conn\_rep\_to}$ generated by a random time generator. The elapse of this time is the Connection Reply Timeout.
Connection Request	A special request from the *Bus Master* addressing the *Connection Address*. All *unpolled* units will start a *Connection Reply Timeout*. The first unit experiencing the timeout will respond with a Connection Reply. The Connection Reply/Request mechanism is used by the Bus Master to recognize and identify units connected to the bus.
Connection Address	Special address reserved for the *Connection Request/Reply* mechanism.
Cyclic Redundancy Check	CRC. Special check sum generated from the *PDU* and Telegram Head and added as the two last bytes in the *Telegram* when transmitted. Used by the receiving unit to validate the transferred Telegram.
Data Frame	Synonymous to *Telegram*.
Data Item	Name for the smallest quantity of information to be handled. Each Data Item has a name (identifier) and an *ID-Code*. They are organised in *Classes*.
Data Item INFO Tables	Tables built by the application programmer containing the INFO data (reply to INFO-operation) for the relevant *ID-codes* in the *Classes* Measured Values, Commands, Configuration Parameters and Reference Values. Belongs to the *Bus Unit Tables*.

Data Item Pointer Tables	Tables built by the application programmer containing a pointer for each *Data Item (ID code)* to the location of the value of the Data Item. Only certain *Classes* have related Data Item Pointer Tables: Measured Values (`meas_tab[]`), Configuration Parameters (`conf_tab[]`), Reference Values (`ref_tab[]`), Test Data (`test_tab[]`) and ASCII Strings (`ascii_tab[]`). The Data Item Pointer Tables belong to the *Bus Unit Tables.*
Data Link Layer	DLL. The *GENIpro* Boundary to the physical world.The layer is further divided into the *Medium Access Layer (MAC) and the Logical Link Control (LLC)*. The MAC contains the software Drivers and the timing mechanism to maintain a constant stream of bytes between the hardware and the LLC when a Receive/Transmit Session is going on. The LLC is responsible for *CRC-checking*, Retransmission and the observation of all timing rules (controls all timeouts on bus activities). It feeds the MAC Transmit Driver with bytes from the Transmit Buffer during transmission, and when receiving it writes bytes delivered from the MAC Receive Driver into the Receive Buffer.
Data Request/Reply	Name of the *Telegrams* in a *Data Session* involving the normal transfer of data between a *Bus Master/Submaster* and a Slave. As a parallel to the Data Request/Reply Telegram exists the Connection Request/Reply Telegram which serves the purpose of recognizing units on the network.
Data Reply Buffer	The data buffer in a *Master Unit Type* where *GENIpro* delivers Data Replies to the *Application Program*.
Data Session	The process of transferring data in the form of a *Telegram* between a master/slave couple. For the *Bus Master* the Data Session start is defined to begin at the moment where control of the data transfer is given to *GENIpro* from the *Application Program*, and the end is defined to be the moment where control is returned to the *Application Program*. For a Slave, the Data Session start is the moment of arrival of the first byte in a Data Request, and the Session End is the moment of transmitting the last byte in the Data Reply.
Device Type	A classification of bus units used by *GENIpro* in a *Management Master* to select which Telegrams from the *Telegram Table* to use for polling of a specific unit. The Device Type is recorded in the *Network List* for each member. It can be modified by the Application Program. The default Device Type used by *GENIpro* equals the Unit Family specifier. The file [`dev_type.h`] coming with the *GENIpro* source code contains Device Type definitions for *GENIbus Units*. The application programmer can change these or define his own.
Device Object	The instance of a certain *Device Type*. The *Application Program* declares the number of objects needed
Direct Addressing	The transmission of a Telegram to a bus address directly specified by the Application Program. This is done via a function call to the API.
Direct Telegram	Designation for a Telegram sent via *Direct Addressing*.
Functional Profile	The specification of *Data Items* and their interpretation for *GENIbus Units.*
Fieldbus	Bus communication for connection of sensors, actuators and controllers at the field level of the automation hierarchy.
Global Address	An address shared among all bus members. Can be used by a *Bus Master* to send broadcast messages. Also called Broadcast Address.
Group	*GENIbus* Units with a common Group Address (*group_addr*) are said to belong to the same group and can be addressed as such by a *Bus Master*.
Head	The Head of a *Telegram* is the first 4 bytes: Start Delimiter (SD), Length Specifier, Destination Address (DA) and Source Address (SA). The Head of an *APDU* is the first two bytes. The first one specifying *Class* the second one specifying *Operation* (SET, GET, INFO) and APDU Length (max 63 bytes).
Initiator	The *active* master. This can be the *Bus Master* or a *Submaster*.
ID-Code	Each *Data Item* has an ID-Code which together with its *Class No.* identifies it uniquely. *GENIpro* accesses all data via their ID-Codes.

Inter Data Frame Delay	*GENIpro* inserts this delay time between *Telegrams* to make the *Idle Detection* mechanism work
IR-IRQ-Transmitter-Driver	`ir_out_idle_irq()`. Software Driver controlling the transmission of single bytes via the Infra Red link (*GENIlink*). Uses a hardware timer and is thereby interrupt driven. Bytes are delivered from the *Transmit Buffer* via the LLC. Also called a MAC Transmit Driver.
IR-IRQ-Receiver-Driver	`ir_in_irq()`. Software Driver controlling the reception of single bytes via the Infra Red link (*GENIlink*). Uses a hardware timer and is thereby interrupt driven. Delivers bytes to the Receive Buffer via the LLC. Also called a MAC Receive Driver.
Logical Link Control	LLC. See *Data Link Layer*
Management Master	A full *GENIpro Master Unit Type* implementation. This gives the *Application Program* access to all API variables and functions. Used for typical *Bus Master* applications. A *Management Master* results from a *GENIpro* compilation with the *Compile Time Option*: `#define CTO_MM Man_Mast`
Master Appointment	The process of being appointed to *Bus Master*. This could be as a result of a *Bus Configuration*, or it could be an explicit appointment by receiving an *RFS-acknowledge* to be Bus Master (BM_ack) from the present Bus Master.
Master Appointment Timeout	The *Bus Configuration* resulting from *Bus Activity Timeout* starts a Master Appointment Timeout with the value $t_{master\_app\_to}$ in all connected *Master Unit Types*. This is a value generated from the *Unit Address* and it will therefore be different in different units. The first one to time out (lowest address) will be appointed by *GENIpro* to be *Bus Master* and will automatically transmit the *Bus Capture Message*.
Master Management	The part of the *Presentation Layer* used for handling *Submaster* and *Bus Master Mode*. This is placed in a separate module [`man_mast.c` and `ctr_mast.c`], which must be linked with the rest of *GENIpro* when building *Master Unit Types*.
Master Unit Type	A unit where the control of *GENIpro* is handed over to the *Application Program* if the unit gets appointed to *Bus Master*. A Master Unit Type operating in its *Slave Mode* is said to be a *passive* master. When it operates in its *Bus Master mode* it is said to be *active* as long as it doesn't send an *RFS-acknowledge* to a *Submaster*. If this happens the Submaster will become *active* (until Submaster Timeout occur) and the Bus Master will become a *passive* Bus Master in this period.
Medium Access	MAC. Here used as a synonymous to Drivers. See *Data Link Layer*.
Message	A *Telegram* always generated by a *Bus Master/Submaster* and which doesn't require a reply. This is normal when using *Global* or *Group* Addressing.
Network List	A data structure where the *Bus Master* records information about all bus connected units that can be automatically recognized with *Connection Request* (this is all units except *Subslaves*). An entry in the Network List is called a *Bus Unit Record*. The list is dynamic, it is updated by *GENIpro* whenever a new unit is connected or an old one is disconnected. The Application Program can via API functions operate on the Network List as well.
Network Status Record	Contains Information about changes in the *Network List* and the last completed *Communication Session*.
Operation	Operations to perform on *Data Items* via *GENIbus/GENIlink*: SET, GET and INFO.
Presentation Layer	Name for the *GENIpro* layer interacting directly with the *Application Program*. Layer no. 6 in the ISO reference model for Open Systems Interconnection (OSI).
Poll	Often used as a synonymous for a Data Request. Being polled is a slaves only opportunity to bring information to its master and to request (using the *RFS-option*) special grantings like *Submaster Status*.
Poll Delay Time	Time between the end of one *Communication Session* and the initiation of the next (Inter Session Delay).
Poll List	Used by the *Bus Master* as a list of which units from the *Network List* to include in the *Poll Cycle*. Units can be added and removed by the *Application Program*.

Prompt Telegram	Empty Telegram used for nothing but "prompting" a unit, to give it a chance to reply with an *RFS Option* and to inform the *Application Program* that the contact with a Bus Master is OK.
Protocol Data Unit	PDU. The information carrying part of a *Telegram*. Consist of *APDU*'s and an *RFS-option* (optional).
Receive Break	An error event recognized in the *Data Link Layer*. This event is said to occur if an idling bus is detected (*Bus Idle* Interrupt) while in the middle of receiving a *Telegram*. The Telegram will be discarded and the Receive Break Error will be recorded.
Receiver Driver	See *Bus-IRQ-Receiver-Driver* or *IR-IRQ-Receiver-Driver*
Receive Buffer	Storage location where a Telegram (Data Frame) is buffered during reception and while it is being processed. Has the size `df_buf_len`.
Reply	A Reply is always generated by a Slave as an answer to a *Request*. An answer to a *Data Request* is a *Data Reply*, and the answer to a *Connection Request* is a *Connection Reply*.
Reply Timeout	When the *Data Link Layer* in an *Initiator* sends the last byte in a *Request*, it starts a *Reply Timeout*. If the first byte (SD) in the *Reply* has not arrived within the *Reply Timeout Time* t_{reply_to}, a *Retransmission* will be initiated. If Retransmission has already been attempted a Reply Timeout Error will be recorded.
Retransmission	The *Data Link Layer* in an *Initiator* automatically retransmits a *Reque*st in case of Reply Timeout Error, *Receive Break* Error or *CRC*-Error.
Request	A Request always comes from a *Bus Master* or from a *Submaster*, and requires a *Repl*y from the addressed unit. A Request can be a *Data Request* resulting in a *Data Reply* or a *Connection Request* resulting in a *Connection Reply*.
RFS-option	A special byte which a *Master Unit Type* in *Slave Mode* can tail to a *Data Reply* specifying the request for a certain granting like *Bus Master Status* or *Submaster Status* (SM_rfs, BM_rfs).
RFS-acknowledge	The granting by the *Bus Master* to an *RFS-option* from a *Master Unit Typ*e in *Slave Mode*. Can be a *Submaster Acknowledge* or a *Bus Master Acknowledge* (SM_ack, BM_ack)
RS485	A two wire hardware standard using differential voltage signals for serial digital data transmission. A cheap and very noise immune standard suitable for long distance networking.
Session	See *Communication Session.*
Session Event Register	The *Application Program* can read the result of the last completed *Communication Session* in this register. It must consume the event and take the appropriate action.
Slave Mode	A *GENIbus* Unit operating in *Slave Mod*e behaves like a slave: Its only bus activity is replying to requests from a *Bus Master* or *Submaster*. It can request a change of mode to Bus Master or Submaster if it is a *Master Unit Type* by using the *RFS-option* when polled. During the process of *Bus Configuration* all connected units are in their Slave Mode.
Slave Unit Type	A unit never capable of entering *Bus Master Mod*e or *Submaster Mode*
Start Delimiter	SD. First byte in a *Telegram*. This byte codes for the type of Telegram: *Data Request*, *Data Reply* or *Message.*
Sublist	A Sublist only exists in a *Submaster* and is a datastructure containing the addresses of bus units to be polled. It is a result of a user configuration reflecting the physical system design of the *Subnetwork* it represents. It is never changed automatically by *GENIpro*.
Submaster (=Submaster Mode)	A unit in *Slave Mode* can by the use of the *RFS-option* request Submaster Status if it is a *Master Unit Type*. In this mode it gets the full bus control and can operate similar to a *Bus Master*. It should however only address units listed in its *Sublist*. It is only allowed to remain in this mode for the time t_{subm_to} (which is a variable taking values from 50ms to 750ms) specified by itself, when requesting the Submaster Status. When the time has passed *GENIpro* will generate a Submaster Timeout forcing it back into Slave Mode and the former Bus Master will take over the bus control.

Submaster Acknowledge	The acknowledging of a Request to be a Submaster. This comes in the form of an *RFS-acknowledge* (SM_ack) from the *Bus Master*. Submaster Acknowledge is a *Control Session.*
Submaster Timeout	The elapse of t_{subm_to}, see *Submaster*.
Subnetwork	Slaves listed in a *Submaster's Sublist* are said to constitute a Subnetwork. Each of these slaves can be operated from the Submaster. If these slaves are configured as *Subslaves* they will not be automatically recognaizable (answer to *Connection Request*) by a *Bus Master.* The Subnetwork is then said to be *closed*, otherwise it is said to be *open*.
Subslave (=Subslave Mode)	A configured mode similar to Slave Mode but with the difference that *Connection Requests* are never replied.
Telegram	Name for the total block of data passing through the *Physical Layer* during a Transmit- or Receive Session . Consist of Telegram *Head* plus *PDU* plus *CRC value*. Synonymous to *Data Frame*.
Transmitter Driver	See *Bus-IRQ-Transmitter-Driver* or *IR-IRQ-Transmitter-Driver.*
Transmit Buffer	Storage location where a Telegram (Data Frame) is buffered during transmission and while it is being built. A Data Request is kept here until the Data Reply is received, because the request is needed to interpret the reply. Has the size `df_buf_len`.
Unpolled State	A unit in *Slave Mode* records itself as *unpolled* if it is not addressed within the time interval $t_{max\_unpolled}$. Only when in this state it will reply to *Connection Requests*.
Unit Address	The private address of a *GENIbus* Unit, must be unique
Unpolled Timeout	The elapse of $t_{max\_unpolled}$, see *Unpolled State*.

F.2 Abbreviations

ACT	Access Control Type
AGC	Automatic Gain Control
AP	Application Program
API	Application Program Interface
APDU	Application Program Data Unit
ACK	Acknowledge
BUR	Bus Unit Record
BM	Bus Master
BM_ack	Bus Master Acknowledge by use of RFS Option
BM_rfs	Bus Master Request by use of RFS Option
BO	Byte Order
CRC	Cyclic Redundancy Check
CTO	Compile Time Options
DA	Destination Address
DI	Driver Interface
DLL	Data Link Layer
FSK	Frequency Shift Keying
GENI	Grundfos Electronics Network Intercommunication
ID	Identification code
IR	Infra Red
IRQ	Interrupt Request
ISO	International Standard Organization
LLC	Logical Link Control
MAC	Medium Access
MM	Master Management
OS	Operation Specifier
OSI	Open System Interconnection
OSI	Operating System Interface
PDU	Protocol Data Unit
PL	Presentation Layer

PLI	Presentation Layer Interface (to DLL)
RFS	Request From Slave
SA	Source Address
SD	Start Delimiter
SIF	Scale Information Format
SM	Submaster
SM_ack	Submaster Acknowledge by use of RFS Option
SM_rfs	Submaster Request by use of RFS Option
SS	Subslave
SZ	Sign of Zero
UART	Universal Asynchronous Receiver & Transmitter
VI	Value Interpreter

G. References

/1/	Joe Campbell	C Programmers Guide to Serial Communications, SAMS PUBLISHING, USA 1993, ISBN 0-672-30386-1
/2/	Andrew S. Tanenbaum	Computer Networks, Prentice-Hall, USA 1988, ISBN 0-13-166836-6
/3/	Christoph Stoppok & Dr. Helmut Sturm	Comparative Study of Field Buses for Sensor and Actuator Systems that are Available or in Development, VDI/VDE-Technologiezentrum Informationstechnik GmbH, Berlin 1990, English translation
/4/	Members of the Technical Staff, Linear Products Division, Texas Instruments	Advanced Linear European Seminars. Handbook, 1990
/5/	Dipl.-Phys. Ing. Andreas Papenheim	The M-Bus: A Documentation, Version 4.1, June 9, 1994
/6/		Information om standarder for datatransmission, Bind 1 og 2 Dansk Standardiseringsråd 1990, ISBN 87-7310-095-1
/7/	Klaus Henning Sørensen	RTOS 78K Users Manual, Real Time Operating System for NECK0, June 19. 1995