APPLICATION NOTE

HostRdCom - Host to Reader Communication

User & Reference Manual

Revision 1.1
Preliminary

November 2001

Philips
Semiconductors

PHILIPS

CONTENTS

1 GENERAL INFORMATION 4
1.1 Scope 4
1.2 General description MIFARE® 4
1.3 General description for I•CODE 6
2 HOSTRDCOM – LIBRARY DESCRIPTION 8
2.1 Topology of the Library 8
2.2 How to use the Interface 9
2.2.1 Opening the Interface 9
2.2.2 Handling the timeout periods 12
3 REFERENCE MANUAL 13
3.1 Command Object Data Storage 13
3.1.1 Public Methods 13
3.1.2 Function Description 14
3.2 IRDA Communication Interface 18
3.2.1 Public Methods 18
3.2.2 Function Description 18
3.3 IRDA Communication Protocol 19
3.3.1 Public Methods 19
3.3.2 Function Description 19
3.4 ProtocolBase Class 20
3.4.1 Public Methods 20
3.4.2 PROTECTED Types 20
3.4.3 Function Description 20
3.5 ReaderInterface Class 23
3.5.1 Public Methods 23
3.5.2 Protected Attributes 23
3.5.3 Function Description 23
3.6 RS232 Class 26
3.6.1 Public Types 26
3.6.2 Public Methods 26
3.6.3 Function Description 27
3.7 RS232BlockFramingProtocol Class 31
3.7.1 Public Methods 31
3.7.2 Function Description 31
3.7.3 Member Function Documentation 32

3.8 RS232Protocol3964 Class 33
3.8.1 Public Methods 33
3.8.2 Fucntion Description 33
3.9 StrBufferContainer Class 37
3.9.1 Public Methods 37
3.9.2 Function Description 37
3.10 USB Class 40
3.10.1 Public Methods 40
3.10.2 Function Description 40
3.11 USBProtocol Class 43
3.11.1 Public Methods 43
3.11.2 Function Description 43

4 RS232 SERIAL PROTOCOL 45
4.1 Block Framing Protocol 45
4.1.1 Control Character Definition 45
4.2 Character Based Protocol Similar to Serial Protocol 3964 48
4.2.1 Control Character Definition 48

5 USB SERIAL PROTOCOL 52
5.1 Protocol Description 52
5.2 Data Block Formats 53
5.2.1 Description of the Data Block 54

6 REVISION HISTORY 55

1 GENERAL INFORMATION

1.1 Scope

This document describes the functionality of the host to reader communication for the MIFARE® MF RD700 'Pegoda' reader and the I•CODE Evaluation Kit SL EV400. It includes the functional description of the used commands and gives details, how to use or design-in this device from a system and software viewpoint.

The default configuration for the MF RD700 uses the MF RC500 as the contactless reader IC.
In fact, the reader module can be used with all members of the new contactless reader IC family without any additional hardware changes.
The command set defines all commands, which can be used by the different reader ICs as the MF RC530 and the MF RC531. These reader-ICs will be available soon to give the user the possibility to integrate these ICs easy in the Pegoda environment. Consequently not all described commands are available in the standard configuration of the Pegoda reader based on the MF RC500 reader IC.

The default configuration for the SL EV400 uses the SL RC400 as the contactless reader IC.

1.2 General description MIFARE®

The MF RD700 Pegoda reader is ready to be connected to a PC.
Figure 1 shows the basic overview of the MF RD700's software concept. Different levels of the PC libraries can be identified:

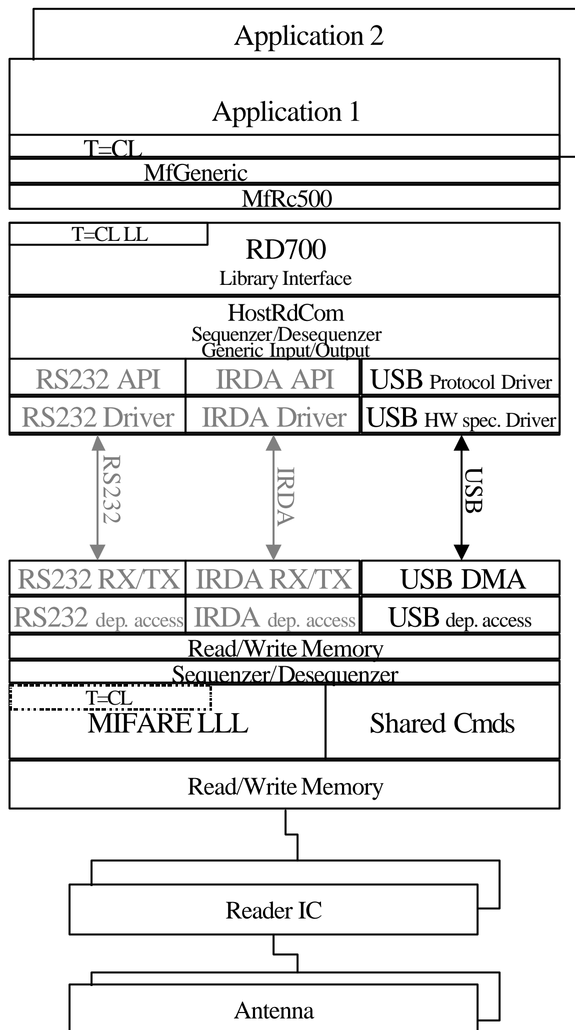
- **Application Level**
  This level is user specific and might be used by the user to implement own applications and test programs. The evaluation kit packages for the MF RC700 provide the *MIFAREWND* program and the source code for the *Rges* program as example for small test programs on application level.

- **RD700 Command Set**
  The complete RD700 command set including all PC relevant commands is described in the application note: *MIFARE® MF RD700 Command Set*.

- **HostRdCom**
  This document describes the communication between the MF RD700 and a host PC.

- The firmware of the MF RD700 covers the functionality of the basic function library of the MF RC 500. This basic function library is described in the Application Note *MIFARE® MF RC500 Basic Function Library.*

The supported operating systems are limited to the Microsoft Windows Platform. Depending on the selected interface connection, Win98, Win2000 or Win NT 4.0 is supported. The content of this document should be precise enough, to give the user the possibility writing own communication libraries for other operating systems.



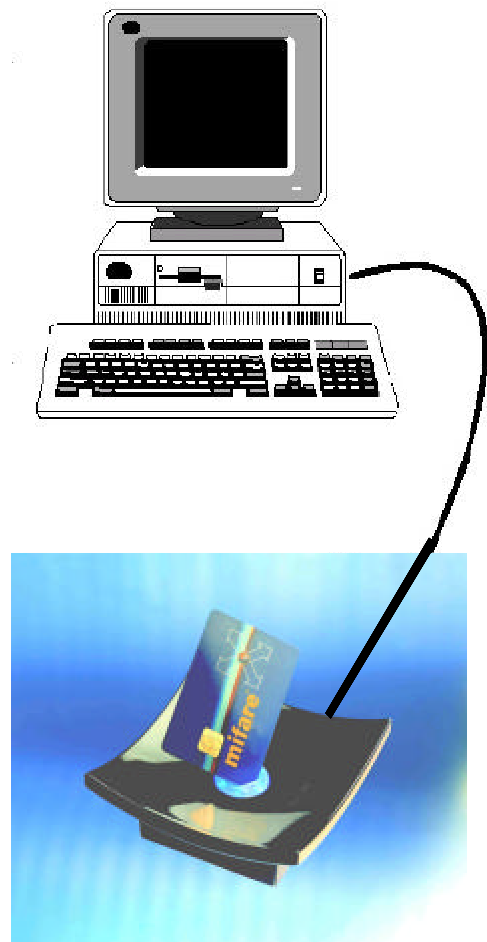


Figure 1. MIFARE® Pegoda General Software Overview

1.3 General description for I•CODE

The **I•**CODE Pegoda read/write device SL EV400 is ready to be connected to a PC.

Figure 2 shows the basic overview of the SL EV400's software concept. Different levels of the PC libraries can be identified:

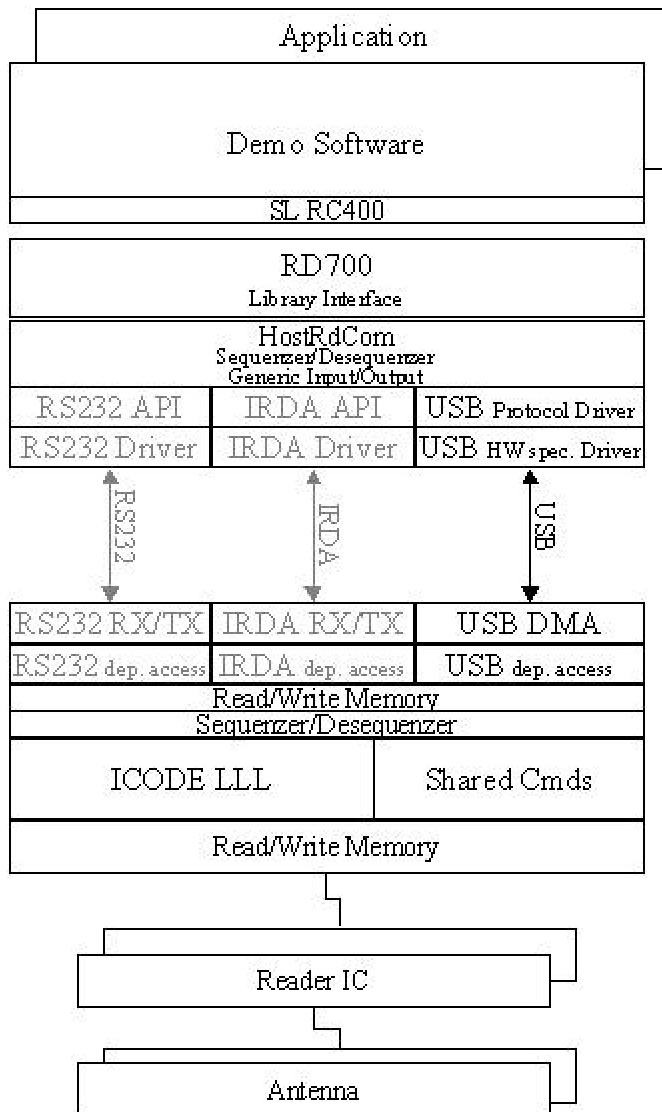
- **Application Level**
  This level is user specific and might be used by the user to implement own applications and test programs. The evaluation kit packages for the SL RC400 provide the *SLEV400Demo* to show the functionality of the SL RC400 with **I•**CODE 1 and **I•**CODE SLI labels.

- **RD700 Command Set**
  The command set for the SL EV400 including all PC relevant commands is described in the application note: *I•CODE SL EV400 Command Set* (not available now).

- **HostRdCom**
  This document describes the communication between the SL EV400 and a host PC.

- The firmware of the SL EV400 covers the functionality of the basic function library of the SL RC400. This basic function library is described in the Application Note *I•CODE SL RC400 Basic Function Library* (not available now).

The supported operating systems are limited to the Microsoft Windows Platform. Depending on the serial communication over the USB only Win98 or Win2000 are supported. The content of this document should be precise enough, to give the user the possibility writing own communication libraries for other operating systems.



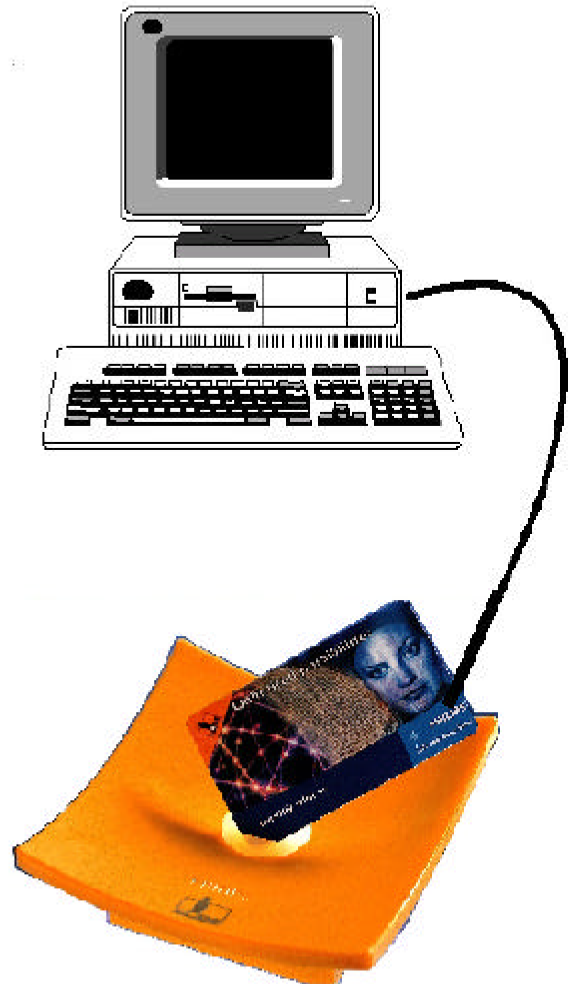


Figure 2, I•CODE Pegoda General Software Overview

2 HOSTRDCOM – LIBRARY DESCRIPTION

This Application Note covers the description of the **HostRdCom** library (**Host** <==> **Read**er **Com**munication) designed for a high-level serial communication meaning serial data exchange. In fact, the serial communication can be established by using USB, RS232 or IrDA for the *MIFARE®* MF RD7000 and USB for the **I•**CODE SL EV400 (RS232 is not supported). All relevant commands are covered in the HostRdCom. The library is programmed in an object-oriented way in C++, in order to provide a possibility to open several communication channels at the same time. This gives the user especially for USB devices the possibility to connect several MF RD 700 or SL EV400 simultaneously.

2.1 Topology of the Library

The functionality of a Reader Interface is a subset of the specific communication channel's functionality. The reader interface is the basis of the specific communication channel.

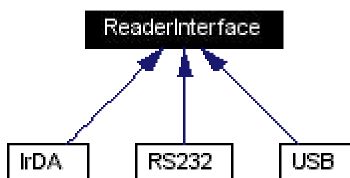


Figure 3. Reader Interface

In the application the user chooses the specific communication protocol and passes the corresponding base object to the library member functions. All included library functions use a smaller functional subset, which is available with every type of channel.

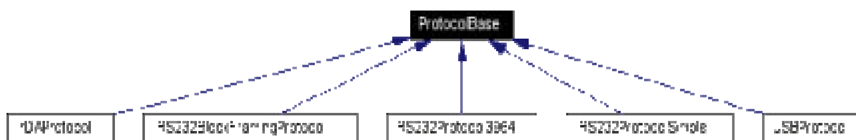


Figure 4. Protocol Base

Each protocol implementation should provide some minimal functionallity. These functional subset is defined in a protocol base class, from which all protocol implementations are derived.

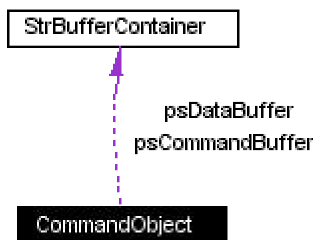


Figure 5. Command Object

In general, a command object is able to generate a 32k bytes data stream between host and reader for both directions. Because of this large memory space reserved on host side, mostly the reader or card side is limiting the maximum communication stream length.

2.2 How to use the Interface

2.2.1 OPENING THE INTERFACE

The following section shows examples for different usage of the protocols. The default configuration of the MF RD700 is the USB interface, but as mentioned also an RS232 interface or an IRDA interface can be established. For the **I•**CODE SL EV400 only the USB interface is available.
The following part covers as an example the RS 232 interface in order to give the user the possibility to establish this interface type if required.
The RS 232 block framing protocol is implemented as described below.

```
#include <Rs232.h>
#include <Rs232BlockFramingProtocol.h>
#include <ProtocolBase.h>
...
signed short          status;
Rs232*                p_Rs232;
ProtocolBase*         p_PB = NULL;
...
if( (p_Rs232 = new RS232()) != NULL )
{
  if( (status = p_Rs232->SetDefaultBaud-rate( 115200 )) == COM_SUCCESS )
  {
    if( (status = p_Rs232->SetComPort( 1 )) == COM_SUCCESS )
    {
      if( (status = p_Rs232->OpenInterface()) == COM_SUCCESS )
      {
         if( (p_PB = new RS232BlockFramingProtocol( *p_Rs232 )) != NULL )
         {
            status = MI_OK;
         }
         else
         {
            status = MI_PROTOCOL_FAILURE;
         }
      }
    }
  }
}
else
{
  status = MI_INTERFACE_FAILURE;
}
...
// successful initialization of the interface and the protocol ?
if( status == MI_OK )
{
  // do the work you want
}
...
...
// all work is done - release the interface and protocol objects
if( p_PB != NULL )
{
  delete p_PB;
  p_PB = NULL;
}
if(p_Rs232 != NULL )
{
  delete p_Rs232;
  p_Rs232 = NULL;
}
...
```

The RS232 communication in the example above uses the COM1 port and a default baud rate of 115 kbaud. Having set the port and the baud rate, the open port command enables the communication and additional parameters e.g. stop bit and parity are set. Additionally, the timeout period for reading and writing are set to their default values.

The communication channel and the selected protocol is now fixed and only a pointer to the protocol is needed for communication with the reader (under the pre-condition that the baud rate and used com-port remain the same). Attention has to be paid to the scope of the objects. The interface object has to have a larger scope than the protocol object, because the interface is passed to the constructor of the selected protocol as parameter. When using the protocol during communication with the reader all the interface operations will be carried out by the protocol using the pointer passed in the constructor.

The use of the defined RS232 protocol is described in the following session.

```
#include <CommandObject.h>
#include <ProtocolBase.h>
#include <CmdExecution.h>

#define RET_STATUS signed short

RET_STATUS
Rd700PcdReadE2(ProtocolBase& prot,
               unsigned short startaddr,
               unsigned char length,
               unsigned char* data)
{
   RET_STATUS          status;
   CommandObject       CO;

   CO.SetCommand(uC_PcdReadE2);
   CO.AddParam(startaddr);
   CO.AddParam(length);

   if ((status = prot.SendCommand(CO)) == MI_OK)
   {
      if ((status = static_cast<RET_STATUS>
                    (CO.GetStatus())) == MI_OK)
      {
         CO.GetParam(data,length);
      }
   }
   return status;
}

...
// somewhere in your program
if ((status = Rd700PcdReadE2(
              p_PB,0,16,data))== MI_OK)
{
   // all the work was done automatically
}
```

Having defined the protocol (and with it the associated interface), it can be used as a 'communication path parameter' when calling functions from the Rd700 library like the function PcdReadE2 in the above example. The function call will look similarly no matter which interface will be actually used. The example also shows that all re-formatting of command parameters as well as the insertion of the command code are done in the respective library function. So using this approach the application programmer can concentrate on the task which shall be performed with the reader in an interface-independent and protocol-independent manner.

Using USB the application code remains exactly the same, only the substitution of the protocol with one of following code fragments has to be done.

```
if ((p_USB = new USB()) != NULL)
{
 if(p_USB->OpenInterface() != COM_SUCCESS)
 {
    status = MI_INTERFACE_FAILURE;
 }
 else
 {
   if ((p_PB = new USBProtocol(*p_USB))
        == NULL)
   {
      status = MI_PROTOCOL_FAILURE;
   }
 }
}
else
{
  status = MI_INTERFACE_FAILURE;
}
```

Using IrDA interface instead of the USB or RS232 interface the following adaptations have to be done.

```
if ((p_IrDA = new IrDA()) != NULL)
{
 if(p_IrDA->OpenInterface() != COM_SUCCESS)
 {
    status = MI_INTERFACE_FAILURE;
 }
 else
 {
   if ((p_PB = new IrDAProtocol(*p_USB))
        == NULL)
   {
      status = MI_PROTOCOL_FAILURE;
   }
 }
}
else
{
  status = MI_INTERFACE_FAILURE;
}
```

The data stream is always constructed correctly according to the used interface and protocol type.

2.2.2 HANDLING THE TIMEOUT PERIODS

Each interface provides at least two timeout periodes. One for sending data to the reader and one for receiving data from the reader. Please pay attention, that the receiving periode also includes the processing time of a command on the reader side, e. g. if the processing time is 500 milliseconds and provided guard time should be another 200 ms, then the minimal receive timeout periode should be 700 ms. The problem in this case is the various processing time of the commands. There are two major possibilities to overcome this problem. Either you select the receive timeout periode large enough to cover also the longest command, or to select individual timeout periodes for every command. Both possibilities are supported by the HostRdCom-library.

If nothing is defined, the library has a default value of 6000 milliseconds (6 seconds) for transmit and receive timeout periode. Using the appropriate member functions, the default value can be overridden by new timeout periodes for sending and receiving separately. Additionally there is the possibility to pass an individual timeout for sending and receiving data with every command exchange with the reader.

The exact time measurement for these timeout values depends on the selected interface, e.g. receiving data using the RS232 interface this periode is measured between two subsequent bytes. That means between sending two subsequent bytes to the reader, the transmit timeout periode must not expire. Between the last byte sent and receiving the first byte, the receive timeout periode must not expire. Between two subsequent bytes received from the reader also the receive timeout periode is relevant.

The values for send and receive timeout can be used for tuning the response in case of a removal of the reader device. If you want to send something to the reader which is not ready - the transmit timeout periode will expire. If the reader is disconnected during processing a command or during sending the response back to the host, the receive timeout periode will expire. As mentioned before, the receive timeout periode is limited mostly by the command processing time of the reader. But in many cases the transmit timeout periode can be kept very short.

3 REFERENCE MANUAL

3.1 Command Object Data Storage

An object of this type is capable to hold all data, which have to be sent to via the serial interface, or which have been received from the serial input stream. The object performs all necessary tasks for converting multi-byte data types into a serial data stream and vice versa (sequencer/de-sequencer).

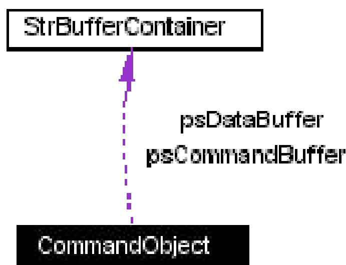


Figure 6. Command Object Data Storage

3.1.1 PUBLIC METHODS

Function name	Function Call
CommandObject	*CommandObject ()* *~CommandObject ()*
SetCommand GetCommand	*void SetCommand (const unsigned char cmd)* *unsigned char GetCommand ()*
SetStatus GetStauts	*void SetStatus (const char status)* *void SetStatus (const short status)* *void SetStatus (const long status)* *long GetStatus ()*
AddParam	*short AddParam (const unsigned char &data)* *short AddParam (const unsigned short &data)* *short AddParam (const unsigned int &data)* *short AddParam (const unsigned long &data)* *short AddParam(const unsigned char *data, const unsigned long &len)*
GetParam	*short GetParam (unsigned char &data)* *short GetParam (unsigned short &data)* *short GetParam (signed short &data)* *short GetParam (unsigned int &data)* *short GetParam (unsigned long &data)* *short GetParam(unsigned char *data, const unsigned long &len)*
ResetBuffers	*void ResetBuffers ()*
GetCommandBuffer	*const unsigned char* GetCommandBuffer ()*
SetDataBuffer GetDataBuffer SetDataByte	*short SetDataBuffer(const unsigned char *buffer, const unsigned long &len)* *const unsigned char* GetDataBuffer ()* *unsigned char SetDataByte(unsigned char &byte)*
GetDataBuffer Get Command Buffer	*unsigned long GetDataBufferLength ()* *unsigned long GetCommandBufferLength ()*

Table 3-1. Command Objects Data Storage

The member functions can be grouped together for different data types.

3.1.2 FUNCTION DESCRIPTION

3.1.2.1 CONSTRUCTOR & DESTRUCTOR

```
CommandObject ()
```

```
~CommandObject ()
```

Construction via copy constructor or assignment operator overloading is disabled.

3.1.2.2 SetCommand, GetCommand

```
void SetCommand (const unsigned char cmd)
unsigned char GetCommand ()
```

Parameters:
cmd command code according to reader firmware codes

Returns: none or command code according to reader firmware codes

Each function call on the reader is uniquely identified by a command code. This code also determines the number and types of parameters in the data stream which follow the command code.

3.1.2.3 SetStatus, GetStatus

```
void SetStatus (const char status)
void SetStatus (const short status)
void SetStatus (const long status)
long GetStatus ()
```

Parameters:
status exit status of the processed command

Returns: none or exit status of the specified command

Status of the processed command
Depending on the protocol type, the return value of a processed command can have a different value range. In former protocols, only one byte was reserved, this implementation also provides two or four bytes to fit into future requirements.

3.1.2.4 AddParam

```
short AddParam (const unsigned char &data)
short AddParam (const unsigned short &data)
short AddParam (const unsigned int &data)
short AddParam (const unsigned long &data)
short AddParam (const unsigned char *data,
                const unsigned long &len)
```

Parameters:
data depending on the data type several functions are declared for appending data at the end of the data stream. If the data type consists of several bytes, the bytes are converted with the least significant byte first.

len for character arrays an additional length information is necessary

Returns:
COM_SUCCESS ok
COM_ERROR memory or system allocation error

Adds one parameter to the protocol buffer.
Depending on the command code, several parameters can be sent to the reader. The order of parameters within the sending buffer depends on calling order of this function. The length of the parameter depends on the data type of the passed parameter.
Several member functions provide a unified interface for passing these parameters.

3.1.2.5 GetParam

```
short GetParam (unsigned char &data)
short GetParam (unsigned short &data)
short GetParam (signed short &data)
short GetParam (unsigned int &data)
short GetParam (unsigned long &data)
short GetParam (unsigned char *data,
                const unsigned long &len)
```

Parameters:
data depending on the data type several functions are declared for extracting data from the beginning of the data stream. If the data type consists of several bytes, the bytes are converted with the least significant byte first.
len for character arrays an additional length information is necessary

Returns:
COM_SUCCESS ok
COM_ERROR too many characters extracted - buffer is empty

Extracts one parameter from the received data buffer
Depending on the command code, several parameters are returned by the reader. The order of parameters within the received buffer depends on the calling order of this function. The length and type of the parameter depends on the data type of the returned function parameter.
Several member functions provide a unified interface for extracting these returned values from the buffer.

3.1.2.6 ResetBuffers

```
void ResetBuffers ()
```

Parameters: none

Returns: nothing

This function initialises all internal buffers and their read/write pointers to an initial state. After calling this function, the command object can be used for a new command.

3.1.2.7 GetCommandBuffer

```
const unsigned char* GetCommandBuffer ()
```

Parameters: none

Returns: character array for outgoing data without framing header

This function returns the prepared buffer for outgoing data without framing header.

Note: This buffer is valid until new data is appended to the buffer.

3.1.2.8 SetDataBuffer

```
short SetDataBuffer
                (const unsigned char *buffer,
                 const unsigned long &len)
```

Parameters:

data character array, which will be appended to the current buffer for incoming data
len length information of the character array

Returns:

COM_SUCCESS ok
COM_ERROR memory allocation error

3.1.2.9 GetDataBuffer

```
const unsigned char* GetDataBuffer ()
```

Parameters: none

Returns: character array for incoming data without framing header

This function returns the prepared buffer for incoming data without framing header.

Note: This buffer is valid until a subsequent character is received.

3.1.2.10 SetDataByte

```
unsigned char SetDataByte (unsigned char &byte)
```

Parameters:

byte character, which will be appended to the current buffer for incoming data

Returns:

COM_SUCCESS ok
COM_ERROR memory allocation error

3.1.2.11 GetDataBufferLength, GetCommandBufferLength

```
unsigned long GetDataBufferLength ()
unsigned long GetCommandBufferLength ()
```

Parameters: none
Returns: length of the sent buffer (*GetCommandbufferLength()*) or received buffer (*GetDataBufferLength()*)

These functions returns the number of bytes of the corresponding memory buffer.

3.2 IRDA Communication Interface

An object of this type is capable to operate an available device with IrDA connectivity. Data can be transmitted or received via the opened interface.

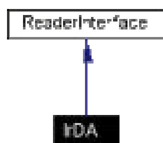


Figure 7. IrDa Communication Interface

3.2.1 PUBLIC METHODS

Function name	Function Call
IrDA	*IrDA ()* *~IrDA ()*
OpenInterface CloseInterface	*short OpenInterface ()* *short CloseInterface ()*
ResetInterface	*short ResetInterface ()*
ClearInternalBuffers	*short ClearInternalBuffers ()*
WriteBytesUnblocked	*short WriteBytesUnblocked* *(unsigned char *ch,* *unsigned long datalen,* *unsigned long &dwBytesWritten,* *unsigned long Timeout)*
ReadBytesUnblocked	*short ReadBytesUnblocked* *(unsigned char *ch,* *unsigned long datalen,* *unsigned long &dwBytesRead, unsigned long Timeout)*

Table 3-2. IrDACommunication Interface

3.2.2 FUNCTION DESCRIPTION

3.2.2.1 CONSTRUCTOR & DESTRUCTOR

Construction via copy constructor or assignment operator overloading is disabled.

```
IrDA ()
```

```
~IrDA ()
```

3.2.2.2 MEMBER FUNCTION DOCUMENTATION

For detailed description of the member functions, please see the description of the corresponding base class member declaration.

3.3 IRDA Communication Protocol

An object of this type is capable to ensure a certain protocol over the IrDA interface channel. This protocol has to correspond with the protocol of the connected device.

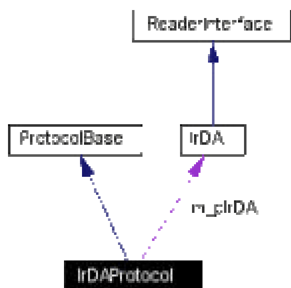


Figure 8. IrDA Communication Protocol

3.3.1 PUBLIC METHODS

Function name	Function Call
IrDAProtocol	*IrDAProtocol (IrDA &pRI)* virtual ~IrDAProtocol ()
ResetProtocol	*virtual short ResetProtocol (void)*

Table 3-3. IrDA Communication Protocol

3.3.2 FUNCTION DESCRIPTION

3.3.2.1 CONSTRUCTOR & DESTRUCTOR

Construction via copy constructor or assignment operator overloading is disabled.
Constructor with corresponding interface as parameter. Please ensure, that the scope of the reader interface (IrDA object) is equal or larger than the scope of the protocol object.

```
IrDAProtocol (IrDA& pRI)
```

```
~IrDAProtocol ()
```

3.3.2.2 MEMBER FUNCTION DOCUMENTATION

For detailed description of the member functions, please see the description of the corresponding base class member declaration.

3.4 ProtocolBase Class

Objects of this type are used to provide a unified protocol interface to the application. The application needs not to be aware of the specific protocol implementation or interface types beeing used.

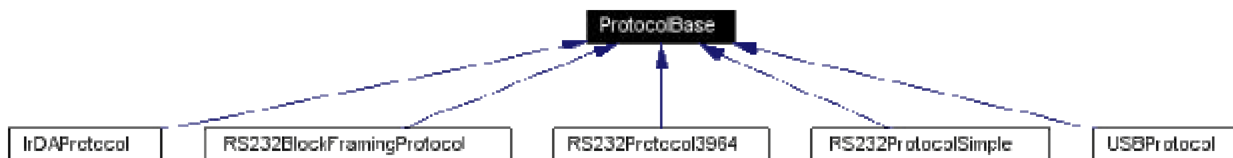


Figure 9. ProtocolBase Class

3.4.1 PUBLIC METHODS

Function name	Function Call
ProtocolBase	*ProtocolBase ()* *~ProtocolBase ()*
SendCommand	*short SendCommand (CommandObject &CmdObject,* *unsigned long RxTimeout=0,* *unsigned long TxTimeout=0)*
ResetProtocol	*virtual short ResetProtocol (void)=0*
SetRXTimeout GetRXTimeout	short SetRxTimeout (unsigned long RxTimeout) unsigned long GetRxTimeout () const short SetTxTimeout (unsigned long TxTimeout) unsigned long GetTxTimeout () const

Table 3-4. ProtocolBase Class

3.4.2 PROTECTED TYPES

enum TIMEOUTS { RX_TIMEOUT = 6000,
TX_TIMEOUT = 6000 }

These values are the default timeout periodes in milliseconds for receiving data from the reader (RX_TIMEOUT) or sending data to the reader (TX_TIMEOUT). These values can be overridden by calling the appropriate member function (*SetRxTimeout()*, *SetTxTimeout()*).

3.4.3 FUNCTION DESCRIPTION

3.4.3.1 CONSTRUCTOR & DESTRUCTOR

Construction via copy constructor or assignment operator overloading is disabled.

```
ProtocolBase ()
```

```
~ProtocolBase ()
```

3.4.3.2 SendCommand

```
virtual short SendCommand
             (CommandObject &CmdObject,
              unsigned long RxTimeout=0,
```

```
unsigned long TxTimeout=0)
```

Parameters:

CmdObject	Data storage for command data to be sent and received.
RxTimeout	receives timeout for frame. This parameter does not need to be provided. If the value is equal to zero, the global timeout period is valid.
TxTimeout	transmits timeout for frame. This parameter does not need to be provided. If the value is equal to zero, the global timeout period is valid.

Returns:

This function is the key member function during communication between host and reader.

From point of view of the caller, this function takes the constructed output data, sends this data to the reader and waits for the reader response, which is returned in the command object. From internal view, this function is the implemenation of the whole protocol state machine.
The send data stream is constructed from the header information, the data passed within the command object and the trailer information. After sending header, user data and trailer to the reader, the function waits for the response.
Timing constraints for sending and receiving can be passed either though the interface object or the additional timout parameters of this function.
After complete receiption of the reader response, the header and trailer data will be evaluated and the user data is passed to the command object, where it is accessable for the user.

The whole protocol processing is embedded in a critical section, which makes the library thread save.

3.4.3.3 ResetProtocol

```
virtual short ResetProtocol (void)=0
```

Parameters: none

Returns: corresponding to the overloaded function implementation

In genral, a protocol is some kind of state machine, which is more or less complex. Calling this function means, that the internal state machine should be initialized to start conditions, which includes, that already received data or data which remains still to send is deleted and any internal variable is set to start condition. For simple protocol implementation the work consists of simply set the send and receive counter to zero.

3.4.3.4 SetRxTimeout, GetRxTimeout

```
virtual short SetRxTimeout (unsigned long RxTimeout)
virtual unsigned long GetRxTimeout () const
virtual short SetTxTimeout (unsigned long TxTimeout)
virtual unsigned long GetTxTimeout () const
```

Parameters:

RxTimeout	timeout periode for receiving one data packet
TxTimeout	timeout periode for transmitting one data packet

Returns: corresponding status

Any Protocol should handle different timeout periods for receiving and transmitting data. The appropriate values are stored in this base class. The unit of the values passed to the function are milliseconds.
In order to provide a complete interface, the corresponding Get-functions are included.

3.5 ReaderInterface Class

Base class for interface properties classes' (e.g. RS232,IrDA, .).

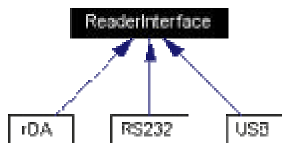


Figure 10.ReaderInterface Class

3.5.1 PUBLIC METHODS

Function name	Function Call
ReaderInterface	*ReaderInterface ()* *virtual ~ReaderInterface ()*
Pure Virtual Member Functions These member functions must be defined in every derived class	
CloseInterface	*virtual short CloseInterface (void)=0*
ResetInterface	*virtual short ResetInterface ()=0*
ClearInternalBuffers	*virtual short ClearInternalBuffers ()=0*
WriteBytesUnblocked	*virtual short WriteBytesUnblocked (unsigned char *data, unsigned long datalen, unsigned long &dwBytesWritten, unsigned long Timeout)=0*
ReadBytesUnblocked	*virtual short ReadBytesUnblocked (unsigned char *ch, unsigned long datalen, unsigned long &dwBytesRead, unsigned long Timeout)=0*

Table 3-5. ReaderInterfaceClass

3.5.2 PROTECTED ATTRIBUTES

OVERLAPPED m_osReader
OVERLAPPED m_osWriter

3.5.3 FUNCTION DESCRIPTION

3.5.3.1 Constructor & Destructor

Construction via copy constructor or assignment operator overloading is disabled.

```
ReaderInterface::ReaderInterface ();
```

```
ReaderInterface::~ReaderInterface () [virtual]
```

3.5.3.2 ClearInternalBuffer

```
short ReaderInterface::ClearInternalBuffers () [pure virtual]
```

Parameters: none

Returns: depending on the current interface type

purges all remaining data and initialise internal structures
Re-implemented in IrDA, RS232, and USB.

3.5.3.3 CloseInterface

```
short ReaderInterface::CloseInterface (void)
[pure virtual]
```

Parameters: none

Returns: depending on the current interface type

Closes the according interface.
Re-implemented in IrDA, RS232, and USB.

3.5.3.4 OpenInterface

```
short ReaderInterface::OpenInterface (void)
[pure virtual]
```

Parameters: none

Returns: depending on the current interface type

Opens the according interface to communicate with the reader.
Re-implemented in IrDA, RS232, and USB.

3.5.3.5 ReadBytesUnblocked

```
short ReaderInterface::ReadBytesUnblocked
            (unsigned char * data,
            unsigned long len,
            unsigned long & dwBytesRead,
            unsigned long Timeout)
[pure virtual]
```

Parameters:

data	byte data stream
datalen	number of bytes to read
dwBytesRead	number of bytes read
Timeout	timeout period for read operations

Returns: depending on the current interface type

Reads a specified number of bytes from the interface. This function returns, if either the bytes are read, or the timeout period expired.
If the Timeout value is zero, a global set timeout period is used.
Re-implemented in IrDA, RS232, and USB.

3.5.3.6 ResetInterface

```
short ReaderInterface::ResetInterface ()
[pure virtual]
```

Parameters: none

Returns: depending on the current interface type

Closes the open interface and re-opens it.
Re-implemented in IrDA, RS232, and USB.

3.5.3.7 WriteBytesUnblocked

```
short ReaderInterface::WriteBytesUnblocked
            (unsigned char * data,
            unsigned long len,
            unsigned long & dwBytesWritten,
            unsigned long Timeout)
[pure virtual]
```

Parameters:

data	byte data stream
datalen	number of bytes to write
dwBytesWritten	number of bytes written
Timeout	timeout period for write operations

Returns: depending on the current interface type

Sends a specified number of bytes to the interface. This function returns, if either the bytes are sent, or the timeout period expired.
If the Timeout value is zero, a global set timeout period is used.
Re-implemented in IrDA, RS232, and USB.

3.6 RS232 Class

The RS232 class defines the RS232 communication interface, which is a derived class from ReaderInterface.
Additional to the timeout periods defined in the base class, RS232 specific parameters are introduced. Using the enlarged interface either the Com port and baud rate can be defined.
The number of bits per byte, stop bit and parity are fixed to 8 bit/byte, 1 stop bit and no parity.
In case of an error, a default baud rate can be specified. Let's assume, that the baud rate is set to 115200 baud and the default baud rate is 9600. In this case the communication speed is 115200 until an error occurred. After the error state is left, the new communication speed is 9600 baud. This feature is used for fault recovery of higher level protocols. If a fixed baud rate is required, only the default baud rate has to be set. In this case the baud rate remains the same in case of an error.

Figure 11. RS232 Class

3.6.1 PUBLIC TYPES

enum RS232_DEFAULT_VALUES
{ COM_PORT = 1,
BAUD_RATE = 57600 }

3.6.2 PUBLIC METHODS

Function name	Function Call
RS232	*RS232 ()* *virtual ~RS232 ()*
OpenInerface	*virtual short OpenInterface ()*
CloseInterface	*virtual short CloseInterface ()*
ResetInterface	*virtual short ResetInterface ()*
ClearInternalBuffers	*virtual short ClearInternalBuffers ()*
SetComPort	*short SetComPort (char ComPort)*
GetComPort	*short GetComPort () const*
SetDefaultBaudRate	*short SetDefaultBaudRate (long BaudDefaultRate)*
GetDefaultBaudRate	*long GetDefaultBaudRate () const*
SetBaudRate	*short SetBaudRate (long BaudRate)*
GetBaudRate	*long GetBaudRate () const*
WriteBytesUnblocked	*short WriteBytesUnblocked* *(unsigned char *ch,* *unsigned long datalen,* *unsigned long &dwBytesWritten,* *unsigned long Timeout*
ReadBytesUnblocked	*short ReadBytesUnblocked* *(unsigned char *ch,* *unsigned long datalen,* *unsigned long &dwBytesRead,* *unsigned long Timeout)*

Table 3-6. RS232 Class

3.6.3 FUNCTION DESCRIPTION

3.6.3.1 Constructor & Destructor

Construction via copy constructor or assignment operator overloading is disabled.

```
RS232::RS232 ()
```

```
RS232::~RS232 () [virtual]
```

3.6.3.2 ClearInternalBuffers

```
short RS232::ClearInternalBuffers () [virtual]
```

Parameters: none

Returns: depending on the current interface type

purges all remaining data and initialise internal structures for reading and writing.
Re-implemented from ReaderInterface.

3.6.3.3 CloseInterface

```
short RS232::CloseInterface (void) [virtual]
```

Parameters: none

Returns: depending on the current interface type

Release Com port and associated buffers.

Re-implemented from ReaderInterface.

3.6.3.4 GetBaudRate

```
long RS232::GetBaudRate () const
```

Parameters: none

Returns: integer value for the currently used baud rate

This function returns an integer value for the currently used baud rate. Only following values are valid:

9600
14400
19200
28800
38400
57600
115200

3.6.3.5 Rs232:GetComPort

```
short RS232::GetComPort () const
```

Parameters: none

Returns: integer value for the currently used COM port

This function returns an integer value for the currently used COM port e. g. 1 for COM1.

3.6.3.6 GetDefaultBaudRate

```
long RS232::GetDefaultBaudRate () const
```

Parameters: none

Returns: similar to *GetBaudRate()*

3.6.3.7 OpenInterface

```
short RS232::OpenInterface (void) [virtual]
```

Parameters: none

Returns: depending on the current interface type

The overloaded functions (e.g. RS232::OpenInterface()) open the according interface to communicate to the reader
Re-implemented from ReaderInterface.

3.6.3.8 ReadBytesUnblocked

```
short RS232::ReadBytesUnblocked
                                (unsigned char * data,
                                 unsigned long datalen,
                                 unsigned long & dwBytesRead,
                                 unsigned long Timeout)
[virtual]
```

Parameters:

data	byte data stream
datalen	number of bytes to read
dwBytesRead	number of bytes read
Timeout	timeout period for read operations

Returns: depending on the current interface type

Read a specified number of bytes from the interface. This function returns, if either the bytes are read, or the timeout period expired.
Re-implemented from ReaderInterface.

3.6.3.9 ResetInterface

```
short RS232::ResetInterface () [virtual]
```

Parameters: none

Returns: depending on the current interface type

ResetInterface - closes the open interface and reopens it
Re-implemented from ReaderInterface.

3.6.3.10 SetBaudRate

```
short RS232::SetBaudRate (long BaudRate)
```

Parameters:
BaudRate communication speed

Returns:
COM_WRONG_VALUE wrong parameter value

Setting a new communication speed different to the default baud rate. The values for *BaudRate* are limited to the following list

9600
14400
19200
28800
38400
57600
115200

3.6.3.11 SetComPort

```
short RS232::SetComPort (char ComPort)
```

Parameters:
ComPort value between 1 and 9 for COM port

Returns:
COM_WRONG_VALUE wrong parameter value

The given number represents the index of the serial port e. g. 1 for COM1.

3.6.3.12 SetDefaultBaudRate

```
short RS232::SetDefaultBaudRate (long BaudRate)
```

Parameters:

BaudRate	communication speed

Returns:
COM_WRONG_VALUE wrong parameter value

The value range is equal to the range in function *SetBaudRate()*.

3.6.3.13 WriteBytesUnblocked

```
short RS232::WriteBytesUnblocked
             (unsigned char * data,
              unsigned long len,
              unsigned long & dwBytesWritten,
              unsigned long Timeout)
[virtual]
```

Parameters:

data	byte data stream
datalen	number of bytes to write
dwBytesWritten	number of bytes written
Timeout	timeout period for write operations

Returns:
COM_NO_INTERFACE_HANDLE
COM_INTERFACE_OPEN
COM_RS232_SEND_DEVICE_ERR
COM_TIMEOUT

Send a specified number of bytes to the interface. This function returns, if either the bytes are sent, or the timeout period expired.
Re-implemented from ReaderInterface.

3.7 RS232BlockFramingProtocol Class

RS232 block based communication protocol.
This class implements a protocol easy to implement and ensures a save communication.
The data to be is framed by a SOF-character, sequence number and two trailing CRC check bytes.
Data integrity is ensured by the sequence number and the two bytes CRC.

SOF character 0xA5 character to signal the start of frame
seqnr 1 byte sequence number, which is increased with every frame sent. The received sequence number from the reader has to match the sequence number sent.
cmd 1 command byte

datalen 2 length bytes with LSB first

data [0 .. datalen] data bytes
crc 16 bit CRC code LSB first

The CRC check bytes are calculated from the whole data stream beginning with SOF and ending with the last data byte.

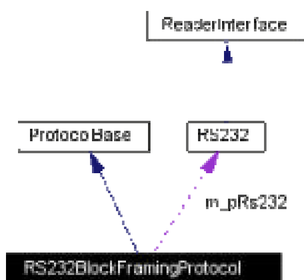


Figure 12. RS232BlockFraming Protocol

3.7.1 PUBLIC METHODS

Function name	Function Call
RS232BlockFramingProtocol	*RS232BlockFramingProtocol (RS232 &pRI)* *virtual ~RS232BlockFramingProtocol ()*
ResetProtocol	*virtual short ResetProtocol (void)*

Table 3-7. Rs232BlockFraming Protocol

3.7.2 FUNCTION DESCRIPTION

3.7.2.1 Constructor & Destructor

Construction via copy constructor or assignment operator overloading is disabled.
Constructor with corresponding interface as parameter. Ensure, that the scope of the reader interface (RS232 object) is equal or larger than the scope of the protocol object.

```
RS232BlockFramingProtocol::
RS232BlockFramingProtocol (RS232 & pRs232)
```

```
RS232BlockFramingProtocol::
~RS232BlockFramingProtocol () [virtual]
```

3.7.3 MEMBER FUNCTION DOCUMENTATION

For detailed description of the member functions, please see the description of the corresponding base class member declaration.

3.8 RS232Protocol3964 Class

RS232 character based communication protocol.
Derived class from ProtocolBase. Defines protocol for RS232 interface.

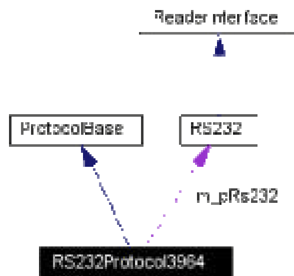


Figure 13. RS232Protocol3964

3.8.1 PUBLIC METHODS

Function name	Function Call
RS232Protocol3964	*RS232Protocol3964 (RS232 &pRI)* *virtual ~RS232Protocol3964 ()*
SetNrRetries	*virtual short SetNrRetries (char NrRetries*
GetNrRetries	*virtual char GetNrRetries () const*
ResetProtocol	*virtual short ResetProtocol (void)*
SetDefaultLengthBytes	*short SetDefaultLengthBytes (char NrLengthBytes)*
GetDefaultLengthBytes	*char GetDefaultLengthBytes () const*
SetNrLengthBytes	*short SetNrLengthBytes (char NrLengthBytes)*
GetNrLengthBytes	*char GetNrLengthBytes () const*
SetCheckSumCalc	*short SetCheckSumCalc (char CheckSum)*
GetCheckSumCalc	*char GetCheckSumCalc () const*

Table 3-8. Rs232Protocol3964

3.8.2 FUCNTION DESCRIPTION

3.8.2.1 Constructor & Destructor

Construction via copy constructor or assignment operator overloading is disabled.
Constructor with corresponding interface as parameter. Please ensure, that the scope of the reader interface (Rs232 object) is equal or larger than the scope of the protocol object.

```
RS232Protocol3964::
RS232Protocol3964 (RS232 & pRs232)
```

```
RS232Protocol3964::~RS232Protocol3964 () [virtual]
```

3.8.2.2 GetCheckSumCalc

```
char RS232Protocol3964::GetCheckSumCalc () const
```

Parameters: none

Returns:
P_CHECKSUM_BCC = 1
P_CHECKSUM_CRC_MSB = 2

P_CHECKSUM_CRC_MSB = 3

This function returns the calculation mode of the check byte.

3.8.2.3 GetDefaultLengthBytes

```
char RS232Protocol3964::GetDefaultLengthBytes() const
```

Parameters: none

Returns: 1 or 2 ... number of length bytes

Returns the number of default length bytes currently defined.

3.8.2.4 GetNrLengthBytes

```
char RS232Protocol3964::GetNrLengthBytes() const
```

Parameters: none

Returns: 1 or 2 ... number of length bytes

Returns the number of length bytes currently defined.

3.8.2.5 GetNrRetries

```
char RS232Protocol3964::GetNrRetries () const
[virtual]
```

Parameters: none

Returns: number of retries before signalling an error

3.8.2.6 ResetProtocol

```
short RS232Protocol3964::ResetProtocol (void)
[virtual]
```

Parameters: none

Returns: depending on the current interface type

Reset protocol to default values.
Having called this function, the initial state will be reached.
Re-implemented from ProtocolBase.

3.8.2.7 SetCheckSumCalc

```
short RS232Protocol3964::SetCheckSumCalc
    (char CheckSum)
```

Parameters:
CheckSum P_CHECKSUM_BCC ... 1

P_CHECKSUM_CRC_MSB . 2
P_CHECKSUM_CRC_LSB .. 3

Returns:
COM_SUCCESS
COM_WRONG_VALUE

Selects different checksum calculation methods.

3.8.2.8 SetDefaultLengthBytes

```
short RS232Protocol3964::SetDefaultLengthBytes
     (char NrLengthBytes)
```

Parameters:
NrLengthBytes number of length bytes

Returns:
COM_SUCCESS
COM_WRONG_VALUE

To enlarge the transmission frames, the length bytes parameter is capable to have 1 or 2 bytes.

3.8.2.9 SetNrLengthBytes

```
short RS232Protocol3964::SetNrLengthBytes
                          (char NrLengthBytes)
```

Parameters:
NrLengthBytes number of length bytes

Returns:
COM_SUCCESS
COM_WRONG_VALUE

To enlarge the transmission frames, the length bytes parameter is capable to have 1 or 2 bytes.

3.8.2.10 SetNrRetries

```
short RS232Protocol3964::SetNrRetries
                          (char NrRetries)
[virtual]
```

Parameters:
NrRetries number of retries, before signalling an error state.

Returns:
COM_SUCCESS
COM_WRONG_VALUE

The number of retries must have a positive value range.

3.9 StrBufferContainer Class

Byte stream class for storage management.
An object of this type manages the storage, which is necessary to sequence the given data. All data conversion is done by the appropriate functions.

3.9.1 PUBLIC METHODS

Function name	Function Call
StrBufferContainer	*StrBufferContainer ()* *virtual ~StrBufferContainer ()*
GetBufferSize	*unsigned long GetBufferSize ()*
IncBufferSize	*short IncBufferSize ()*
Reset	*void Reset ()*
GetWritePosition	*unsigned long GetWritePosition ()*
GetReadPosition	*unsigned long GetReadPosition ()*
GetBuffer	*unsigned char* GetBuffer ()*
Write	*short Write (unsigned char c)* *short Write (unsigned short s)* *short Write (signed short s)* *short Write (unsigned int s)* *short Write (unsigned long s)* *short Write (const unsigned char *a, unsigned long len)*
Read	*short Read (unsigned char *c)* *short Read (unsigned short *s)* *short Read (signed short *s)* *short Read (unsigned int *i)* *short Read (unsigned long *l)* *short Read(unsigned char *a, const unsigned long len)*

Table 3-9 StrBufferContainer

3.9.2 FUNCTION DESCRIPTION

3.9.2.1 Constructor & Destructor

Construction via copy constructor or assignment operator overloading is disabled.

```
StrBufferContainer::StrBufferContainer ()
```

```
StrBufferContainer::~StrBufferContainer () [virtual]
```

3.9.2.2 GetBufferSize

```
unsigned long GetBufferSize ()
```

Parameters: none

Returns: number of bytes allocated from memory

This function returns the current allocated number of bytes. It is not the same as the number of bytes already written in the buffer, which can be read by the function *GetWritePosition()*.

3.9.2.3 IncBufferSize

```
short IncBufferSize ()
```

Parameters: none

Returns:
COM_SUCCESS if successful
COM_ERROR allocation error

This function increments the current allocated memory space by a block size of 512 Bytes.

3.9.2.4 Reset

```
void Reset ()
```

Parameters: none

Returns: none

Reset internal buffers, read and write positions.

3.9.2.5 GetWritePosition, GetReadPosition

```
unsigned long GetWritePosition ()
unsigned long GetReadPosition ()
```

Parameters: none

Returns:
current position of read/write pointer

3.9.2.6 GetBuffer

```
unsigned char* GetBuffer ()
```

Parameters: none
Returns:
character array with written data

The length of the character array can be determined by calling the function *GetWritePosition()*.

3.9.2.7 Write

```
short Write (unsigned char c)
short Write (unsigned short s)
short Write (signed short s)
short Write (unsigned int s)
short Write (unsigned long s)
```

```
short Write (const unsigned char *a,
             unsigned long len)
```

Parameters: s data of different type, which should be appended to the current byte stream.

Returns:
COM_SUCCESS ok
COM_ERROR memory allocation error

3.9.2.8 Read

```
short Read (unsigned char *c)
short Read (unsigned short *s)
short Read (signed short *s)
short Read (unsigned int *i)
short Read (unsigned long *l)
short Read (unsigned char *a,
            const unsigned long len)
```

Parameters: s data of different type, which is read from the byte stream

Returns:
COM_SUCCESS ok
COM_ERROR read position exceeds write position, - no more data in the buffer

3.10 USB Class

USB communication interface.

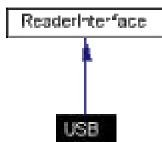


Figure 14. USB Class

3.10.1 PUBLIC METHODS

Function name	Function Call
USB	*USB ()* *virtual ~USB ()*
OpenInterface	*virtual short OpenInterface ()*
CloseInterface	*virtual short CloseInterface ()*
ResetInterface	*virtual short ResetInterface ()*
ClearInternalBuffers	*virtual short ClearInternalBuffers ()*
WriteBytesUnblocked	*short WriteBytesUnblocked* *(unsigned char *data,* *unsigned long datalen,* *unsigned long &dwBytesWritten,* *unsigned long Timeout)*
ReadBytesUnblocked	*short ReadBytesUnblocked* *(unsigned char *data,* *unsigned long datalen,* *unsigned long &dwBytesRead,* *unsigned long Timeout)*

Table 3-10. USB Class

3.10.2 FUNCTION DESCRIPTION

3.10.2.1 Constructor & Destructor

Construction via copy constructor or assignment operator overloading is disabled.

```
USB::USB ()
```

```
USB::~USB () [virtual]
```

3.10.2.2 ClearInternalBuffers

```
short USB::ClearInternalBuffers () [virtual]
```

Parameters: none

Returns:

Purge all remaining data and initialise internal structures.
Re-implemented from ReaderInterface.

3.10.2.3 CloseInterface

```
short USB::CloseInterface (void) [virtual]
```

Parameters: none

Returns:

Re-implemented from ReaderInterface.

3.10.2.4 OpenInterface

```
short USB::OpenInterface (void) [virtual]
```

Parameters: none

Returns:

Open and initialise the interface.
Re-implemented from ReaderInterface.

3.10.2.5 ReadBytesUnblocked

```
short USB::ReadBytesUnblocked
     (unsigned char * data,
     unsigned long len,
     unsigned long & dwBytesRead,
     unsigned long Timeout)
[virtual]
```

Parameters:

data	byte data stream
datalen	number of bytes to read
dwBytesRead	number of bytes read
Timeout	timeout periode for read operations

Returns:
depending on the current interface type

Receive a given number of bytes from the reader.
Re-implemented from ReaderInterface.

3.10.2.6 ResetInterface

```
short USB::ResetInterface () [virtual]
```

Parameters: none

Returns:

Closes the open interface and reopens it.
Re-implemented from ReaderInterface.

3.10.2.7 WriteBytesUnblocked

```
short USB::WriteBytesUnblocked
     (unsigned char * data,
     unsigned long len,
     unsigned long & dwBytesWritten,
     unsigned long Timeout)
[virtual]
```

Parameters:

data	byte data stream
datalen	number of bytes to write
dwBytesWritten	number of bytes written
Timeout	timeout period for write operations

Returns:

depending on the current interface type

Send a given number of bytes to the reader.
Re-implemented from ReaderInterface.

3.11 USBProtocol Class

Defines the USB communication protocol.
USB supports a high-level protocol making additional data integrity checks unnecessary.
Therefore the implemented protocol does not contain any check bytes. Only a sequence number is provided, in order to ensure the correct association between send and receive packet. The sequence number is value between 0 and 255 generated by the host. The reader returns the number in the receive frame. If send and receive sequence number does not match a serious communication error occurred.

Send frame:
seqnr sequence number
cmd command identifier
len 16 bit data length information (least significant byte first)
data *len* number of data bytes

Receive frame:
seqnr sequence number
status status byte of the command
len 16 bit data length information (least significant byte first)
data *len* number of data bytes

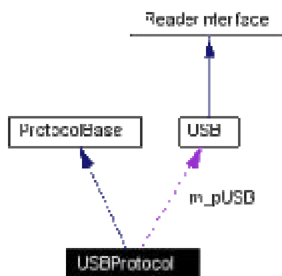


Figure 15. USB Protocol

3.11.1 PUBLIC METHODS

Function name	Function Call
USBProtocol	*USBProtocol (USB &pRI)* *virtual ~USBProtocol ()*
ResetProtocol	*virtual short ResetProtocol (void)*

3.11.2 FUNCTION DESCRIPTION

3.11.2.1 Constructor & Destructor

Construction via copy constructor or assignment operator overloading is disabled.

Constructor with corresponding interface as parameter. Please ensure, that the scope of the reader interface (USB object) is equal or larger than the scope of the protocol object.

```
USBProtocol::USBProtocol (USB & pUSB)
```

```
USBProtocol::~USBProtocol () [virtual]
```

3.11.2.2 ResetProtocol

```
short USBProtocol::ResetProtocol (void) [virtual]
```

Parameters: none

Returns:

Reset protocol to default values.
Re-implemented from ProtocolBase.

4 RS232 SERIAL PROTOCOL

The MF RD700 supports 2 serial protocol types for RS 232 communication. The default used protocol is the block framing protocol. In order to give more flexibility, a character based protocol according to the serial protocol definition 3964 is supported as well.
The implementation for each reader depends on the compiling parameter of the firmware on the microcontroller.
NOTE: The SL EV 400 does not support RS232.

4.1 Block Framing Protocol

The used protocol is a block framing transmission protocol for the link between a control unit (host) and the reader module. Serial communications parameters are:

data bits	8
start bit	1
stop bit	1
parity	none
baudrate	115 kbaud

4.1.1 CONTROL CHARACTER DEFINITION

Description	Char	Value
Start of Frame	SOF	A5 hex

Table 4-1. Start of Frame Definition

This single control character is interpreted as start of frame. Following characters are interpreted as frame or application data.

4.1.1.1 Protocol Description

The host sends a block frame to the reader module, where the check bytes are evaluated. In the case of a correct frame, the corresponding command is executed. Depending on the response, a frame is send back to the host. On the host side, each command consists of a send-frame and a following receive frame.

4.1.1.2 Data Block Formats

Each command frame send from host to the reader module and each response from the reader module to the host has following format.

Host ⇨ Reader Module (Command)

SOF	**TxSeq**	**Command**	**Len** [0]	**Len** [1]	**Par** [0]	...	**Par** [Len-1]	**CRC** [0]	**CRC** [1]
[0]	[1]	[2]	[3]	[4]	[5]	...	[Len+5]	[Len+6]	[Len+7]

Type	**Description**	**No. of bytes**
SOF	start of frame Character	1
TxSeq	Sequence number of the command	1
Command	Command code	1
Len	Number of parameter bytes (low byte first, high byte)	2
Par	Parameter bytes of the command (Len bytes)	len
CRC	16 bit cyclical redundancy check characters (low byte first, high byte)	2

Reader Module ⇨ Host (Response)

SOF	**RxSeq**	**STATUS**	**LEN** [0]	**LEN** [1]	**RESP** [0]	**...**	**RESP** [Len-1]	**CRC** [0]	**CRC** [1]
[0]	[1]	[2]	[3]	[4]	[5]	...	[Len+5]	[Len+6]	[Len+7]

Type	**Description**	**No. of bytes**
SOF	start of frame Character	1
RxSeq	Sequence number of the response	1
Status	Status byte	1
Len	Number of response bytes (low byte first, high byte)	2
Resp	Response bytes	Len
CRC	16 bit cyclic redundancy check characters	2

4.1.1.3 Description of the Data Block

Each frame starts with a SOF character.
The host generates the sequence number TxSeq and sends it within the data block. Having finished the correct command/response exchange the host increases the sequence number before the next command is sent.
The reader module always returns the last received sequence number meaning the TxSeq of a Command is always equal to the RxSeq of the response.
The host-application verifies the equality of the sent and the received sequence numbers after every command/response exchange.
In order to check the data integrity 2 bytes for a CRC check are added to each frame.

The 16-bit cyclic redundancy check character (CRC16) is calculated as described below

Generator Polynomial: $x^{16} + x^{12} + x^5 + 1$ ⇨ CRC_POLYNOM = 8408 hex
Preset Value: ⇨ CRC_PRESET = FFFF hex

A C-based calculation example is added to show how to implement the CRC calculation.

```
unsigned int crc = CRC_PRESET;
for (i = 0; i < cnt; i++)     /* Command:  cnt = Len + 4; */
                       /* Response: cnt = Len + 6; */
{
crc ^= Data[i];
for (j = 0; j < 8; j++)
{
if (crc & 0x0001)
crc = (crc >> 1) ^ CRC_POLYNOM;
else
crc = (crc >> 1);
}
}
/* Command: */
Data[i]   = crc & 0xFF;               /* CRC16 low byte  */
Data[i+1] = crc >> 8;         /* CRC16 high byte */
/* Response: */
if (crc == 0)
{
/* CRC calculation of response OK! */
}
else
{
/* CRC error occurred! */
}
```

The variable ‘crc’ is set to the pre-set value only at the beginning of the preparation of a command / response sequence for transmitting to the reader module / host, respectively.
At the command sequence the CRC value is calculated for all the bytes of the data block (including ***SOF, TxSeq***, ***Command***, ***Len***) and appended to the data stream with least significant byte first.

For the response sequence the CRC value is calculated for all data bytes of any response block (including ***SOF, RxSeq***, ***Status***, ***Len*** and both CRC bytes). The resulting CRC value is zero if no error at transmission occurred.
The host application checks the first received character (should be 0xA5), the received sequence number (should be equal to the latest sent sequence number) and the check bytes.
The period between send- and receive-frame depends on the command type and is therefore not specified.

4.2 Character Based Protocol Similar to Serial Protocol 3964

Additional to the block frame protocol a character based protocol can be implemented as well. The used protocol is a transmission protocol for the link between a control unit (host) and the reader module.
Serial communications parameters are:

data bits	8
start bit	1
stop bit	1
parity	none
baudrate	57.6 kbaud

Note: The default baudrate is 57600 baud. After start-up, the baudrate can be switched in a wide rage (9600 to 115200 baud).

4.2.1 CONTROL CHARACTER DEFINITION

Description	Char	Value
Start of Text	STX	02 hex
End of Text	ETX	03 hex
Data Link Escape	DLE	10 hex
Not Acknowledge	NAK	15 hex

Table 4-2. Control Character Definitions

4.2.1.1 Protocol Description

To start a communication the transmitter and the receiver must be ready. The transmitter starts with STX to establish a data link.
If an NAK or no answer is received the transmitter sends the STX again. If this and the next trial fail again the last STX is transmitted to the receiver. If no valid response (DLE) is returned after a third attempt an error message is generated and the transmitter stops establishing a data link.
Having established the link between transmitter and receiver within a specified period of time (RXTIMEOUT) data can be transmitted.
If the defined maximum character delay (RXTIMEOUT) is exceeded during transmission of the data block the receiver returns to the idle state and waits for another STX to establish a new data link.
If 10 hex appears within a data block it is transmitted twice to distinguish it from the control character DLE.

At the end of transmission of the data block the transmitter transmits DLE and then ETX (DLE is necessary to distinguish a control character from a data byte).
If the receiver detects no error in the transmission (i.e. correct CRC) it answers DLE. If an error is detected the receiver sends NAK, then the transmitter tries to repeat the entire data transmission (maximum RETRIES). If this is not possible it stops sending data and generates an error message.
After each error state, which is reached if the maximum number of retries failed, and after power on, the default communication parameters are restored.

Concatenation Character	NO
Number of length bytes	2
Number of retries	3
Checksum calculation	BCC
Number of checksum bytes	1

4.2.1.2 Command Sequence

Host (Transmitter)		**CRM** (Receiver)	
STX	⇨	Receiver ready?	
	⇦	DLE	DLE: yes!
Data[0]	⇨	Start of data block transmission	
:			
Data[n]	⇨		
DLE	⇨		
ETX	⇨	End of transmission	
	⇦	DLE | NAK	DLE: no error NAK: an error occurred

4.2.1.3 Response Sequence

Host (Receiver)		**CRM** (Transmitter)	
	⇦	STX	Receiver ready?
DLE	⇨		DLE: yes!
	⇦	Data[0]	:
	⇦	Data[n]	
	⇦	DLE	
	⇦	ETX	End of transmission
DLE | NAK	⇨	DLE: no error NAK: an error occurred	

The time in which the receiver has to transmit the control characters upon the transmitter's request is configurable (RXTIMEOUT). This is also the allowed maximum delay time between two characters during the communication.

4.2.1.4 Data Block Formats

The initial state is reached after each power on and if the error state is active. In initial state, the frame has following format.

4.2.1.5 Host ⇨ Reader Module (Command)

TxSeq	**Command**	**Len** [0]	**Len** [1]	**Par** [0]	...	**Par** [Len-1]	**BCC** [0]
Data [0]	Data [1]	Data [2]	Data [3]	Data [4]	...	Data [Len+4]	Data [Len+5]

Type	**Description**	**No. of bytes**
TxSeq	Sequence number of the command	1
Command	Command code	1
Len	Number of parameter bytes (low byte first, high byte	2
Par	Parameter bytes of the command	Len
BCC	8 bit Block Check Character	1

The BCC is calculated for all parameter bytes of any data block. The added control characters DLE and ETX are not included in the BCC calculation. If a DLE (0x10) occurs within the data block, the DLE character is used ONCE for the BCC calculation but transmitted TWICE to the receiver (escape character).

Example:

BCC = par[0] ⊗ par[1] ⊗ ... ⊗ par[len-1]
⊗ EXOR

4.2.1.6 CRM ⇨ Host (Response)

RxSeq	**Status**	**Len** [0]	**Len** [1]	**Resp** [0]	**...**	**Resp** [Len-1]	**BCC**
Data [0]	Data [1]	Data [2]	Data [3]	Data [4]	...	Data [Len+4]	Data [Len+5]

Type	**Description**	**No. of bytes**
RxSeq	Sequence number of the response	1
Status	Status byte (serial communication Host ⇔ CRM)	1
Len	Number of response bytes (low byte first, high byte	2
Resp	Response bytes	Len
BCC	8 bit Block Check Character	1

4.2.1.7 Description of the Data Block

A sequence number (TxSeq) (generated by the host) is sent within the data block. After a correct command/response exchange the host increases the sequence number at the next command.
The reader module always returns the last received sequence numbe that means that the TxSeq of a Command is always equal to the RxSeq of the response.

The host-application may verify the equality of the sent and the received sequence numbers after every command/response exchange, but generally this is not necessary.

Note: To distinguish the data value 10hex from the control character DLE (10hex) each byte with the value **10 hex** within the data block is transmitted **twice**.
Each of these dual transmitted values 10 hex is counted only **once** in *Len* and at the calculation of the BCC.

Example:
The data bytes 10 20 01 00 10 10 hex are transmitted as 10 10 20 01 00 10 10 10 10 hex.
This protocol frame can be changed by software. There are appropriate functions for changing the baudrate (range 9600, 19200, 38400, 57600, 115200), the number for length bytes (range 1 or 2), or activating a concatenation of several frames to one data stream. The concatenation of several frames is intended to be implemented, but is not in the functional scope of the current implementation.
The consistency check is done by a simple BCC calculation. This can be changed by the appropriate function to a 16 bit CRC calculation. Therefore the check bytes are enlarged from 1 to 2 characters.

The 16-bit cyclic redundancy check character (CRC16) is calculated as described in the following:

Generator Polynomial: $x^{16} + x^{12} + x^5 + 1$ ⇨ CRC_POLYNOM = 8408 hex
Preset Value: ⇨ CRC_PRESET = FFFF hex

Calculation Algorithm (C-Example):

```
unsigned int crc = CRC_PRESET;
for (i = 0; i < cnt; i++)     /* Command:  cnt = Len + 4; */
                       /* Response: cnt = Len + 6; */
{
    crc ^= Data[i];
    for (j = 0; j < 8; j++)
    {
        if (crc & 0x0001)
            crc = (crc >> 1) ^ CRC_POLYNOM;
        else
            crc = (crc >> 1);
    }
}
/* Command: */
Data[i]   = crc & 0xFF;              /* CRC16 low byte  */
Data[i+1] = crc >> 8;         /* CRC16 high byte */
/* Response: */
if (crc == 0)
{
    /* CRC calculation of response OK! */
}
else
{
    /* CRC error occurred! */
}
```

The variable ‘crc’ is set to the pre-set value only at the beginning of the preparation of a command / response sequence for transmitting to the reader module / host, respectively.

At the command sequence the CRC value is calculated for the bytes Data[0] ... Data[***Len***+3] of the data block (including ***TxSeq***, ***Command***, ***Len***).
At the response sequence the CRC value is calculated for all data bytes (Data[0] ... Data[***Len***+5]) of any response block (including ***RxSeq***, ***Status***, ***Len*** and both CRC bytes). The resulting CRC value is zero if no error at transmission occurred.
If a 10 hex occurs within the data block, 10 hex is used **once** for the CRC calculation but transmitted **twice** to the receiver.

5 USB SERIAL PROTOCOL

The MF RD 700 supports one single USB protocol. In order to provide a complete description of the USB-communication, some features of the implementation have to be explained.
This chapter presents information about how data is moved across the USB. The information presented is at a level above the signaling and protocol definitions of the system.

The USB provides communication services between a host and attached USB devices. However, the simple view an end user sees of attaching one or more USB devices to a host is in fact a little more complicated to implement. While devices physically attach to the USB in a tiered, star topology, the host communicates with each logical device as if it were directly connected to the root port.

A USB logical device appears to the USB system as a collection of endpoints. Endpoints are grouped into endpoint sets that implement an interface. Interfaces are views to the function of the device. The USB system Software manages the device using the Default Control Pipe. Client software manages an interface using pipe bundles (associated with an endpoint set). Client software requests that data be moved across the USB between a buffer on the host and an endpoint on the USB device.

An endpoint is a uniquely identifiable portion of a USB device that is the terminus of a communication flow between the host and device. Each USB logical device is composed of a collection of independent endpoints. Each logical device has a unique address assigned by the system at device attachment time. Each endpoint on a device is given at design time a unique device-determined identifier called the endpoint number. Each endpoint has a device-determined direction of data flow. The combination of the device address, endpoint number, and direction allows each endpoint to be uniquely referenced. Each endpoint is a simplex connection that supports data flow in one direction: either input (from device to host) or output (from host to device).
An endpoint has characteristics that determine the type of transfer service required between the endpoint and the client software.

Conformity to USB Release	:	1.1	
Device Class	:	vendor specific	
Operating Rate	:	12 MBaud	
Power Consumption	:	300 mA at full Operation	
Number of Interfaces	:	2	
Number of Endpoints	:	4	
Transfer-type	:	Control	for Configuration
		Bulk	for Data

Transfer packet size for data transmission: 64 Bytes

Vendor ID	:	0x074
Product ID	:	0xFF01
Device ID BCD coded	:	0x0001

5.1 Protocol Description

The host sends a frame to the reader module, check bytes are not necessary because of the consistency checks in lower communication layers. The decoded command is executed. Depending on the response, a frame is send back to the host. On the host side, each command consists of a send-frame and a following receive frame.

5.2 Data Block Formats

Each command frame send from host to the reader module and each response from the reader module to the host has following format.

Host ⇨ Reader Module (Command)

TxSeq	**Command**	**Len** [0]	**Len** [1]	**Par** [0]	...	**Par** [Len-1]
[1]	[2]	[3]	[4]	[5]	...	[Len+5]

Type	**Description**	**No. of bytes**
TxSeq	Sequence number of the command	1
Command	Command code	1
Len	Number of parameter bytes (low byte first, high byte)	2
Par	Parameter bytes of the command (Len bytes)	len

Reader Module ⇨ Host (Response)

The response consists of two frames. The first frame is always 2 bytes long and indicates the complete length of the following frame.

TOTAL LENGTH [0]	**TOTAL LENGTH** [1]
[1]	[2]

Type	**Description**	**No. of bytes**
Len	Total number of response bytes (low byte first, high byte)	2

This frame is followed by the data frame which consist of

RxSeq	**STATUS**	**LEN** [0]	**LEN** [1]	**RESP** [0]	**...**	**RESP** [Len-1]
[1]	[2]	[3]	[4]	[5]	...	[Len+5]

Type	**Description**	**No. of bytes**
RxSeq	Sequence number of the response	1
Status	Status byte	1
Len	Number of response bytes (low byte first, high byte)	2
Resp	Response bytes	Len

5.2.1 DESCRIPTION OF THE DATA BLOCK

The host generates the sequence number TxSeq and sends it within the data block. Having finished the correct command/response exchange the host increases the sequence number before the next command is sent.
The reader module always returns the last received sequence number meaning the TxSeq of a Command is always equal to the RxSeq of the response.
The host-application verifies the equality of the sent and the received sequence numbers after every command/response exchange.

6 REVISION HISTORY

REVISION	DATE	CPCN	PAGE	DESCRIPTION
1.1	November 2001		56	second published version
1.0	November 2001	-	56	first published version

Table 6-1: Document Revision History

Definitions

Data sheet status	
Objective specification	This data sheet contains target or goal specifications for product development.
Preliminary specification	This data sheet contains preliminary data; supplementary data may be published later.
Product specification	This data sheet contains final product specifications.
Limiting values	
Limiting values given are in accordance with the Absolute Maximum Rating System (IEC 134). Stress above one or more of the limiting values may cause permanent damage to the device. These are stress ratings only and operation of the device at these or at any other conditions above those given in the Characteristics section of the specification is not implied. Exposure to limiting values for extended periods may affect device reliability.	
Application information	
Where application information is given, it is advisory and does not form part of the specification.	

Life support applications

These products are not designed for use in life support appliances, devices, or systems where malfunction of these products can reasonably be expected to result in personal injury. Philips customers using or selling these products for use in such applications do so on their own risk and agree to fully indemnify Philips for any damages resulting from such improper use or sale.

Philips Semiconductors - a worldwide company

Argentina: see South America
Australia: 34 Waterloo Road, NORTHRYDE, NSW 2113,
Tel. +612 9805 4455, Fax. +612 9805 4466
Austria: Computerstraße 6, A-1101 WIEN, P.O.Box 213,
Tel. +431 60 101, Fax. +431 30 101 1210
Belarus: Hotel Minsk Business Centre, Bld. 3, r.1211, Volodarski Str. 6,
220050 MINSK, Tel. +375172 200 733, Fax. +375172 200 773
Belgium: see The Netherlands
Brazil: see South America
Bulgaria: Philips Bulgaria Ltd., Energoproject, 15th floor,
51 James Bourchier Blvd., 1407 SOFIA
Tel. +3592 689 211, Fax. +3592 689 102
Canada: Philips Semiconductors/Components,
Tel. +1800 234 7381
China/Hong Kong: 501 Hong Kong Industrial Technology Centre,
72 Tat Chee Avenue, Kowloon Tong, HONG KONG,
Tel. +85223 19 7888, Fax. +85223 19 7700
Colombia: see South America
Czech Republic: see Austria
Denmark: Prags Boulevard 80, PB 1919, DK-2300 COPENHAGEN S,
Tel. +4532 88 2636, Fax. +4531 57 1949
Finland: Sinikalliontie 3, FIN-02630 ESPOO,
Tel. +3589 61 5800, Fax. +3589 61 580/xxx
France: 4 Rue du Port-aux-Vins, BP 317, 92156 SURESNES Cedex,
Tel. +331 40 99 6161, Fax. +331 40 99 6427
Germany: Hammerbrookstraße 69, D-20097 HAMBURG,
Tel. +4940 23 53 60, Fax. +4940 23 536 300
Greece: No. 15, 25th March Street, GR 17778 TAVROS/ATHENS,
Tel. +301 4894 339/239, Fax. +301 4814 240
Hungary: see Austria
India: Philips INDIA Ltd., Shivsagar Estate, A Block, Dr. Annie Besant Rd.
Worli, MUMBAI 400018, Tel. +9122 4938 541, Fax. +9122 4938 722
Indonesia: see Singapore
Ireland: Newstead, Clonskeagh, DUBLIN 14,
Tel. +3531 7640 000, Fax. +3531 7640 200
Israel: RAPAC Electronics, 7 Kehilat Saloniki St., TEL AVIV 61180,
Tel. +9723 645 0444, Fax. +9723 649 1007
Italy: Philips Semiconductors, Piazza IV Novembre 3,
20124 MILANO, Tel. +392 6752 2531, Fax. +392 6752 2557
Japan: Philips Bldg. 13-37, Kohnan 2-chome, Minato-ku, TOKYO 108,
Tel. +813 3740 5130,Fax. +813 3740 5077
Korea: Philips House, 260-199, Itaewon-dong, Yonsan-ku, SEOUL,
Tel. +822 709 1412, Fax. +822 709 1415
Malaysia: No. 76 Jalan Universiti, 46200 PETALING JAYA, Selangor,
Tel. +60 3750 5214, Fax. +603 757 4880
Mexico: 5900 Gateway East, Suite 200, EL PASO, Texas 79905,
Tel. +9 5800 234 7381
Middle East: see Italy
Netherlands: Postbus 90050, 5600 PB EINDHOVEN, Bldg. VB,
Tel. +3140 27 82785, Fax +3140 27 88399
New Zealand: 2 Wagener Place, C.P.O. Box 1041, AUCKLAND,
Tel. +649 849 4160, Fax. +649 849 7811
Norway: Box 1, Manglerud 0612, OSLO,
Tel. +4722 74 8000, Fax. +4722 74 8341
Philippines: Philips Semiconductors Philippines Inc.,
106 Valero St. Salcedo Village, P.O.Box 2108 MCC, MAKATI,
Metro MANILA, Tel. +632 816 6380, Fax. +632 817 3474
Poland: Ul. Lukiska 10, PL 04-123 WARSZWA,
Tel. +4822 612 2831, Fax. +4822 612 2327
Portugal: see Spain
Romania: see Italy
Russia: Philips Russia, Ul. Usatcheva 35A, 119048 MOSCOW,
Tel. +7095 247 9145, Fax. +7095 247 9144
Singapore: Lorong 1, Toa Payoh, SINGAPORE 1231,
Tel. +65350 2538, Fax. +65251 6500
Slovakia: see Austria
Slovenia: see Italy
South Africa: S.A. Philips Pty Ltd., 195-215 Main Road Martindale,
2092 JOHANNESBURG, P.O.Box 7430 Johannesburg 2000,
Tel. +2711 470 5911, Fax. +2711 470 5494
South America: Rua do Rocio 220, 5th floor, Suite 51,
04552-903 Sao Paulo, SAO PAULO - SP, Brazil,
Tel. +5511 821 2333, Fax. +5511 829 1849
Spain: Balmes 22, 08007 BARCELONA,
Tel. +343 301 6312, Fax. +343 301 4107
Sweden: Kottbygatan 7, Akalla, S-16485 STOCKHOLM,
Tel. +468 632 2000, Fax. +468 632 2745
Switzerland: Allmendstraße 140, CH-8027 ZÜRICH,
Tel. +411 488 2686, Fax. +411 481 7730
Taiwan: Philips Taiwan Ltd., 2330F, 66,
Chung Hsiao West Road, Sec. 1, P.O.Box 22978,
TAIPEI 100, Tel. +8862 382 4443, Fax. +8862 382 4444
Thailand: Philips Electronics (Thailand) Ltd.,
209/2 Sanpavuth-Bangna Road Prakanong, BANGKOK 10260,
Tel. +662 745 4090, Fax. +662 398 0793
Turkey: Talapasa Cad. No. 5, 80640 GÜLTEPE/ISTANBUL,
Tel. +90212 279 2770, Fax. +90212 282 6707
Ukraine: Philips Ukraine, 4 Patrice Lumumba Str., Building B, Floor 7,
252042 KIEV, Tel. +38044 264 2776, Fax. +38044 268 0461
United Kingdom: Philips Semiconductors Ltd., 276 Bath Road, Hayes,
MIDDLESEX UM3 5BX, Tel. +44181 730 5000, Fax. +44181 754 8421
United States: 811 Argues Avenue, SUNNYVALE, CA94088-3409,
Tel. +1800 234 7381
Uruguay: see South America
Vietnam: see Singapore
Yugoslavia: Philips, Trg N. Pasica 5/v, 11000 BEOGRAD,
Tel. +38111 625 344, Fax. +38111 635 777

Published by:

Philips Semiconductors Gratkorn GmbH, Mikron-Weg 1, A-8101 Gratkorn, Austria Fax: +43 3124 299 - 270

For all other countries apply to: Philips Semiconductors, Marketing & Sales Communications, Internet: http://www.semiconductors.philips.com
Building BE-p, P.O.Box 218, 5600 MD EINDHOVEN, The Netherlands, Fax: +3140 27 24825

© Philips Electronics N.V. 1997 SCB52

All rights are reserved. Reproduction in whole or in part is prohibited without the prior written consent of the copyright owner.

The information presented in this document does not form part of any quotation or contract, is believed to be accurate and reliable and may be changed without any notice. No liability will be accepted by the publisher for any consequence of its use. Publication thereof does not convey nor imply any license under patent- or other industrial or intellectual property rights.

Philips
Semiconductors

PHILIPS