Best practice RS232 communication

Summary

How does asynchronous communication using a RS232 interface works in general and how to implement such a communication in a generc application controller environment.

Applies To

FAUHLHABER MotionControlSystems and MotionControllers out of Product Family V3.0 having a RS 232 interface

How to establish asynchronous communication

When establishing communication between two devices via an RS232 interface there are a few key-aspects of this communication which have to be dealt with.
First of all – here we are considering the situation of the central device – might it be a PC or whatever embedded controller. The task is to command the behavior of a single or multiple drives – here FAULHABER MotionControllers.
In this application note we will refer to the PC or high-level control by it being the application controller whereas the drives $node_1$ … $node_n$ will be referred as being MotionContollers.

Using FAULHABER MotionControllers there are two different configurations.

1:1 Connection

Here a single application controller is connected to a single MotionController via a dedicated RS232 interface. This is the standard usage of a RS232 interface. As there is a RxD and a separate TxD signal the application controller as well as the MotionController could send messages at any time without risking messages being in conflict – full duplex communication.

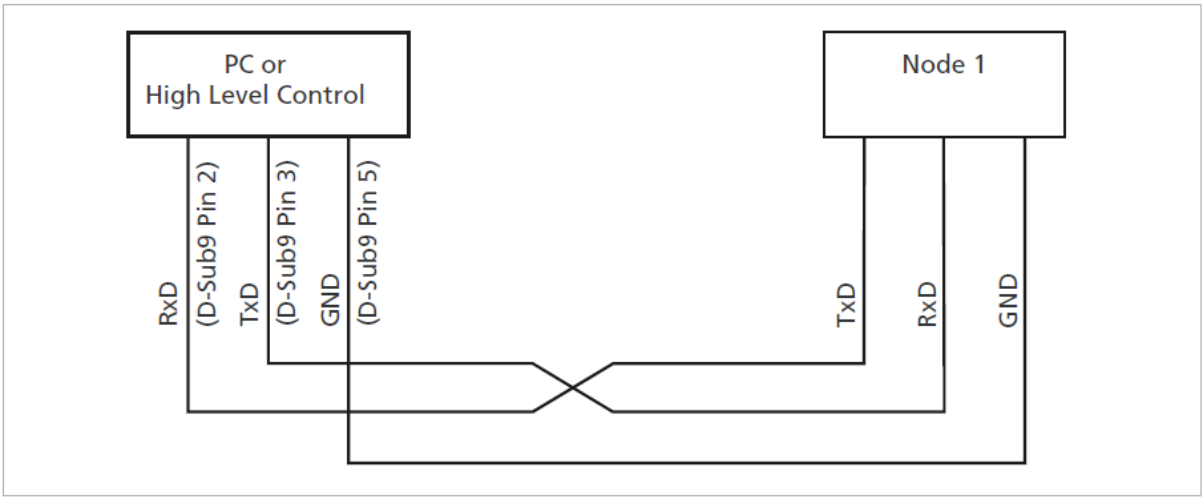

Figure 1 classic direct communication setup using RS232

Based on the RS232 protocol defined for FAULHABER MotionControllers most of the communication uses a request response pattern. The application controller sends a parameter read- or write request to the MotionController and the MotionController then reacts with a response message. Otherwise, nothing happens.
The pattern is typically used in Client – Server communication where a client sends the request and the server responds. This is a strictly synchronous communication. Talk only when asked!

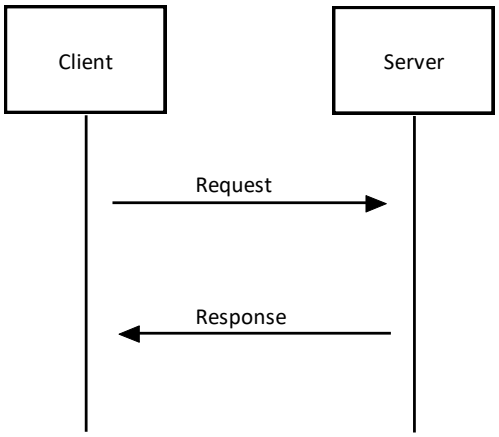

Figure 2 Request - Response communication pattern

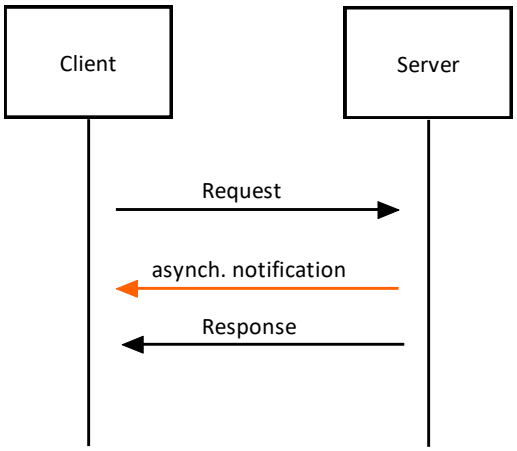

Figure 3 enhanced Request - Response using asynchronous notifications too

Additionally, there can be messages sent by the MotionController without having received a request. These are so called asynchronous messages. The typical asynchronous messages are the boot message of a MotionController after power-up plus notifications of changes in the drive status word which would occur during enabling or disabling the drive or when a profile-based move is started or the target position is reached.
Using these asynchronous notifications can be useful to not have to poll the status of the drive continuously but receiving asynchronous messages adds some complexity to the receiving task, so for the first tests it might be advisable to disable asynchronous messages initially.

Configuration whether asynchronous notifications are to be sent by the MotionController is configured in the Communication settings – object 0x2400.04.

1: n Connection – “net”-mode

Here a single application controller is connected to several MotionControllers using a single RS232 interface.
Such a configuration extends the original usage of the RS232 which was meant to be a 1:1 connection. Most FAULHABER MotionContollers can be configured to support a so-called net-mode where the MotionController enables its TxD driver only when sending a response. Otherwise, the TxD driver is disabled and the Tx line of the motion controllers would be floating which is why the pull-down resistor in Figure 4 is required.

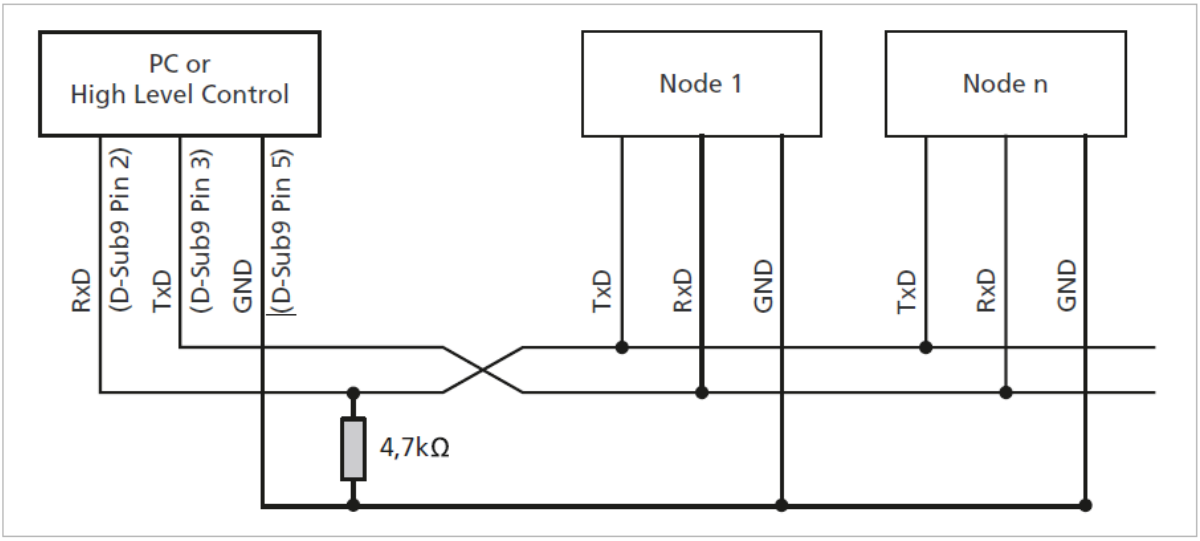

Figure 4 "Net"-mode RS232 configuration with n FAULHABER MotionControllers being connected to the single RS232 in parallel

If several MotionContollers have to share a single RS232 interface, they have to have different node numbers and their RS232 option using “net”-mode has to be configured in advance using 0x2400.05.

In a net-configuration all asynchronous notifications (Figure 3) are disabled. MotionControllers are then allowed to send a response only in reaction to a request – s strict request – response pattern.

In such a “net” configuration it is then of course the responsibility of the application controller to wait for an answer of any request before initiation a next one as the MotionControllers hooked too the RS232 have no means of arbitration.

Configuring the RS232 interface

The RS232 UART interface of the application controller must be configured to match the configuration of the MotionController:

- Configure the same baud rate as with the MotionController
- RS232 frame settings are 8-N-1
- No support for Xon/Xoff protocoll

Sending commands or requests

Sending a command from the application controller is usually straight forward:

a) Identify the object (parameter) to be accessed (read or write)
b) Allocate a buffer for a command frame and fill the payload (object + data)
c) Add the command type
d) Add the node-id of the node to be requested
e) Optionally calculate and add the CRC
f) Add the frame delimiters 'S' as the very first character and 'E' as the last one
g) Send the request using whatever send command the application controller provides

For details about the request frames check either the RS232 Communication manual of the FAULHABER MC V3.0 product series of refer to the appendix of this AppNote.
Using an Arduino as an example the steps could be like:

```
void setup(){
Serial.begin(112500);
}


void loop(){
uint8_t rxRequest[9];
uint8_t txRequest[13];

  // … fill a rx-request – steps a) … f)
  Serial.write(rxRequest,9);

  // … alternatively in case of a write request – steps a) … f)
  Serial.write(txRequest,13);

  // …
}
```

In fact, using a quick implementation not caring for any responses could be a first test whether the command sequence actually works and the FAULHABER MotionController can be enabled, disabled or commanded to move a certain distance.

Receiving responses

Receiving and processing responses is a little more work and depends on the capabilities of the communication library provided by the application controller.

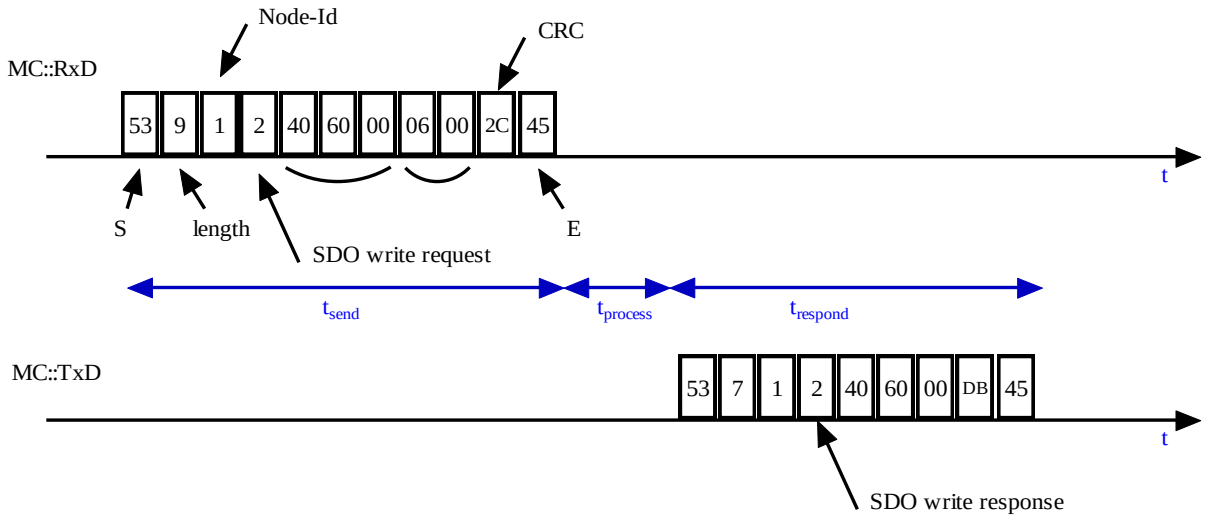

Figure 5 Request - Response Rx and Tx lines

Main challenges are:

a) Sending a request and waiting for a response takes some time (see Figure 5) – should it block the application controller?
b) Propper detection of a complete response

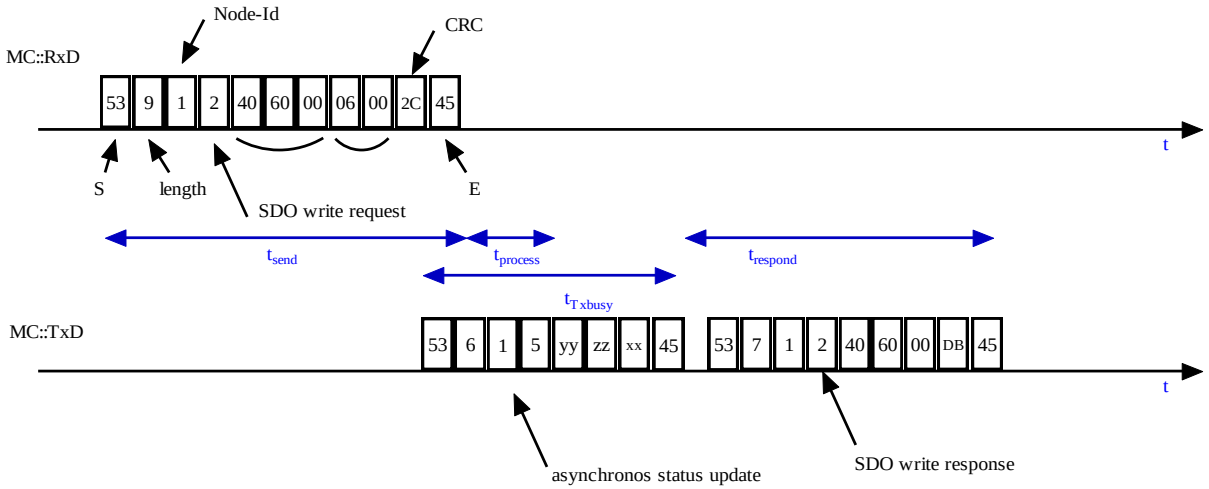

Figure 6 Request - Response with additional asynchronous notification

The situation gets more challenging when asynchronous notifications are used and have to be received too because you can no longer assume receiving bytes only after a request but rather have to check for received characters independent from any sent requests cyclically.
Basically, the steps to receive a response are similar to sending a request:

a) Read whatever character has been received at the RS232 and copy it into a local response buffer
b) When a complete response has been received:
    a. optionally check for the correct node-id

   b. optionally check its CRC
c) Extract the affected object and its value
d) Process the response

First challenge is to detect the response frame delimiters and decide whether a complete and correct response has been received. This can be implemented in a lowest level of message handling software – here I'll call it the receiver.
Techniques could be:

a) After initialization or after not having received a message for some given time put the receiver into a "wait for SOF mode"
b) Cyclically and continuously check for received characters
c) Whenever a first character is received check whether it is a 'S'. Only then start copying the received characters into a response buffer – put the receiver into a "listening" mode.
d) As soon as a second byte has been received extract the payload length from the second character of the received response and keep copying the received characters into a response buffer
e) Only when the last character of the frame according to its length is a 'E' proceed with further processing the received frame – step b) in the list above to process the response.
   Otherwise reset the receiver to "wait for SOF mode" without processing the received data.
f) In case of longer idle times, reset the receiver into the "wait for SOF mode"

These cyclic checks could be placed in an **UART::Update()** method to be called cyclically without blocking other activities. Processing of a frame after step e) can then be done in a higher software level whenever the UART flags a prospect of a received frame. An example of how such a cyclic update could be implemented is given in the appendix.

Software Layers

When dealing with communication in software the different tasks are typically implemented in different layers of a communication software stack.
For the RS232 communication used to interface with a FAULHABER MotionController two levels can be identified.

Driver Layer

The very basic level is the driver level where methods can be provided to **open()** and **configure()** the RS232 interface of the application controller. Additionally this would be the level to implement an actual **SendMessage()** routine using whatever library support for UART based communication is available.
The **SendMessage()** of the driver layer can also add the frame delimiters for transmitted requests.

The checks of incoming bytes of data can either be implemented based on a cyclically called non-blocking **Update()** method or of course alternatively based on reception via an Rx interrupt. As interrupt processing tends to be target specific we use the **Update()** method here.
The **UART::Update()** has to implement the checks as described above and should notify the message level whenever a complete frame has been received. It doesn't matter whether that's done by implementing call-back mechanisms or by a simple flag / return value of the **Update()**. Using a cyclically called **Update()** simplifies the mentioned time-outs as these can simply be implemented as thresholds of auto-incremented cycle counter.

Message Layer

The message layer would typically add administrative information to the message frame like the command code, the node-id of the addressed MotionController and the CRC of the payload.
When a message is to be sent and the administrative information has been added the complete frame can be sent directly using the **UART::SendMessage()**.
Whenever the driver layer signals a complete frame being received the message layer would check the frame for consistency. This is what the CRC in the frame can be used for. For a first test the check of the CRC can be ignored.
If multiple MotionControllers are to be commanded sharing a single RS232 interface the message layer should also implement semaphoric mechanisms where a next request to whatever MotionController can only be sent, when the last one has either been confirmed, answered or timed-out.

Application Layer

Finally on the application layer the **Open()** methods of the message layer will be called in the beginning to access the MotionController. Mainly the application layer however will implement whatever step-sequence or application logic to actually command movements. The message layer should then provide a method to fill object read or write requests with object index and sub-index or to extract received responses.
It could be a convenient idea to more or less have a proxy of the drive status word of the MotionController which gets updated e.g., when a status-word change notification is received or whenever the contents of the status word has been requested and was received. Proxy here means a copy of

the last received value which can be accessed by the application layer without having to request an update every time the status word is used.
It might even be useful to implement something like a cyclic update of key-values of the drive like the drive status word, an actual speed, op-mode and position and have them available for the application.

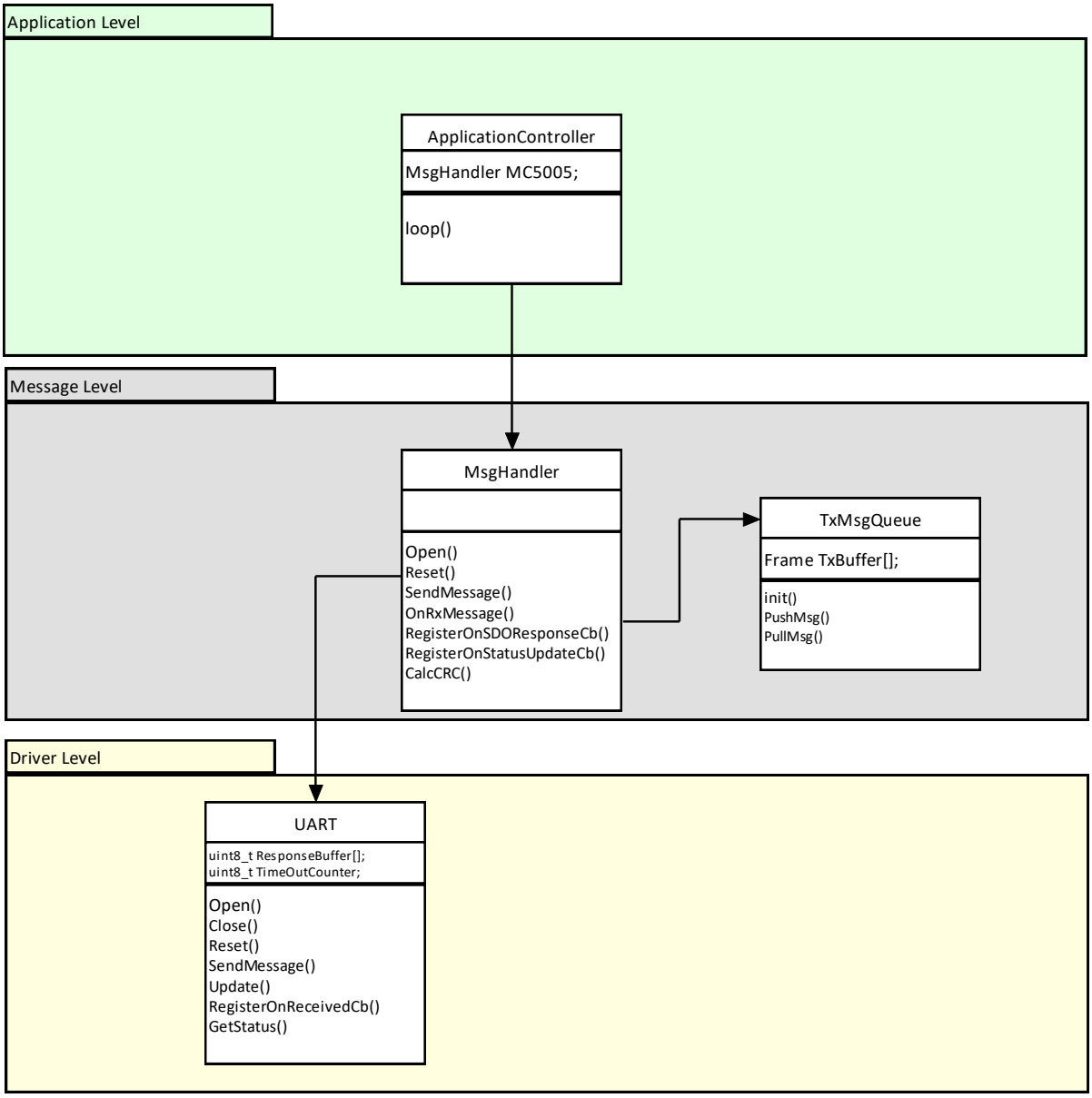

Figure 7 Software layers associated with the RS232 communication stack of the application controller

Steps to set up RS232 communication

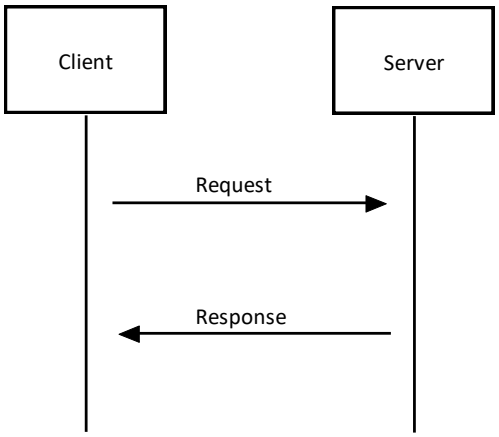

Figure 8 Request - Response communication pattern

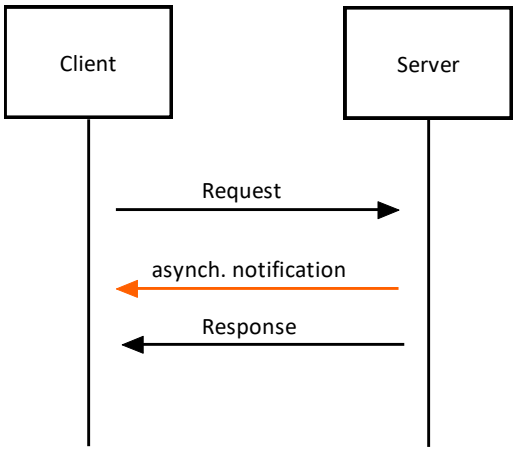

Figure 9 enhanced Request - Response using asynchronous notifications too

To get the concepts working a step wise approach is likely best. Therefore, we will start simple and add complexity depending on the requirements on your application.

Step 1 – no handling of responses

In this first step we do only send commands to the MotionController and assume they are executed. This is an approach used for simple test-scripts or even interactively direct from the terminal of the MotionManger.

a) To keep it simple disable CRC checks for the RS232 for now and to start with disabled EMCY and asynchronous messages:

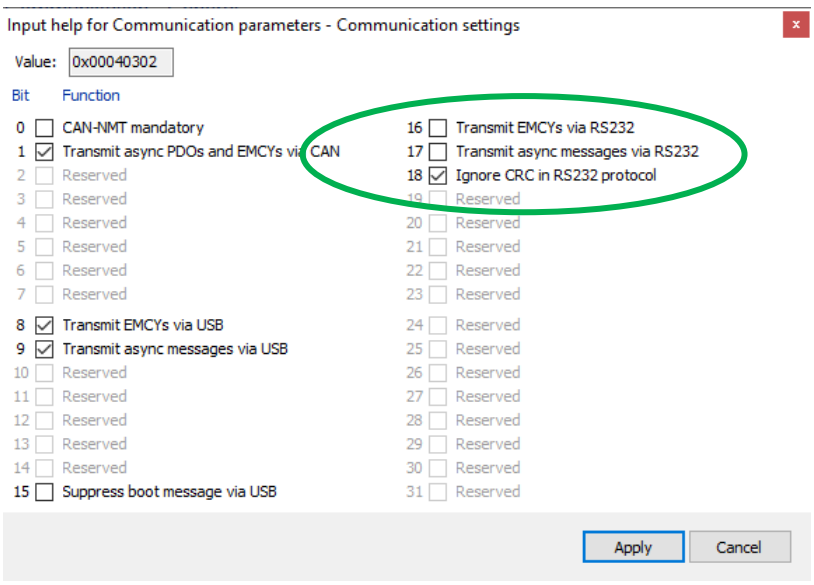

Figure 10 MotionManager 6.9::Drive functions::Communication::general::Communication Settings

b) Next build a **SendMessage()** method to write commands. Parameters would be the object identified by a 16 bit index and an 8 bit sub-index plus the value to be written. In a C implementation a specific parameter for the data-length of the value to be sent will be necessary. In C++ 3 implementations of **SendMessage()** having either a uint8_t, uint16_t or uint32_t payload would be best. Signature could then be:
void **MsgHandler::SendMessage (**uint16_t index, uint8_t sub, uin8_t value**)**;
void **MsgHandler::SendMessage (**uint16_t index, uint8_t sub, uin16_t value**)**;
void **MsgHandler::SendMessage (**uint16_t index, uint8_t sub, uin32_t value**)**;

The **SendMessage()** would then acquire a buffer for the command frame, fill the command type, the node to be addressed, the index, sub-index and value and hand it to the lower level Uart::SendMessage() which simply takes the complete buffer as an argument.

c) Start sending commands and check the response:

```
//force the drive to be disabled
SendMessage(0x6040,0x00,0x0000);
//switch to "ready to switch on"
SendMessage(0x6040,0x00,0x0006);
//switch to "operation enbled"
SendMessage(0x6040,0x00,0x000F);
//select PV mode
SendMessage(0x6060,0x00,0x03);
//set a target speed
SendMessage(0x60FF,0x00,(uint32_t)100);
//wait some time
delay(1000);
//force the drive to be disabled
SendMessage(0x6040,0x00,0x0000);
```

After you managed to get the drive operating by simply sending a sequence of commands add some security by adding CRC calculation to your **SendMessage()** method. The code is given in the appendix.
Please note the CRC is calculated over the contents of the command frame starting with byte 1 in Table 1 to the last byte of the actual value.
Now enable CRC checking in the MotionController and test your implementation using the same test-sequence again.

Step 2 – polling for data

Next step is to poll actual values like the actual contents of the drive status-word or an actual position. This requires caring for the received responses. Now would be the moment to implement the **Uart::Update()** and call it cyclically. Again, an example is given in the appendix. The example code uses a callback from a next software layer to be called whenever a valid raw frame has been received. Alternatively using a common RxBuffer and a "received" flag returned by the **Uart::Update()** will also do the job.
To not complicate this step it might be a good idea to still have all asynchronous notifications disabled in a first step.

Two patterns could be used to implement such a read-request – one is using a blocking call to the message level (Figure 11), the other one avoids blocking the application level by cyclically only checking for a response (Figure 12).

A blocking ReadObject()

In an architecture where the Application Level can wait for the MotionController answering a **MsgHandler::ReadObject()** call the MsgHandler can implement the complete behavior.
The **ReadObject()** could – as the **SendMessage()** method above – take the object identification as call parameters. The direct return value could be a boolean indicating whether the read was successful or failed.
After a successful read the actual read value can either be requested explicitly by an additional call to the MsgHandler or the received value is part of the **ReadObject()** call signature.
Within the MsgHandler at least the **Uart::Update()** would have to called cyclically to poll the interface until the complete response has been received. This could be implemented in a loop which ends when either the complete frame has been received or the call timed-out.
Only after the driver level indicates having received the complete frame the MsgHandler would read the frame, test it for consistency and then return to the application level.
In the same way a **Msghandler::WriteObject()** can implement a write access to the MotionController which explicitly checks the success by waiting for a response.
The application layer code itself can then be pretty simple. A list of subsequent **MsgHandler::WriteObject()** and **MsgHandler::ReadObject()** calls combined with some control logic can then do the job.

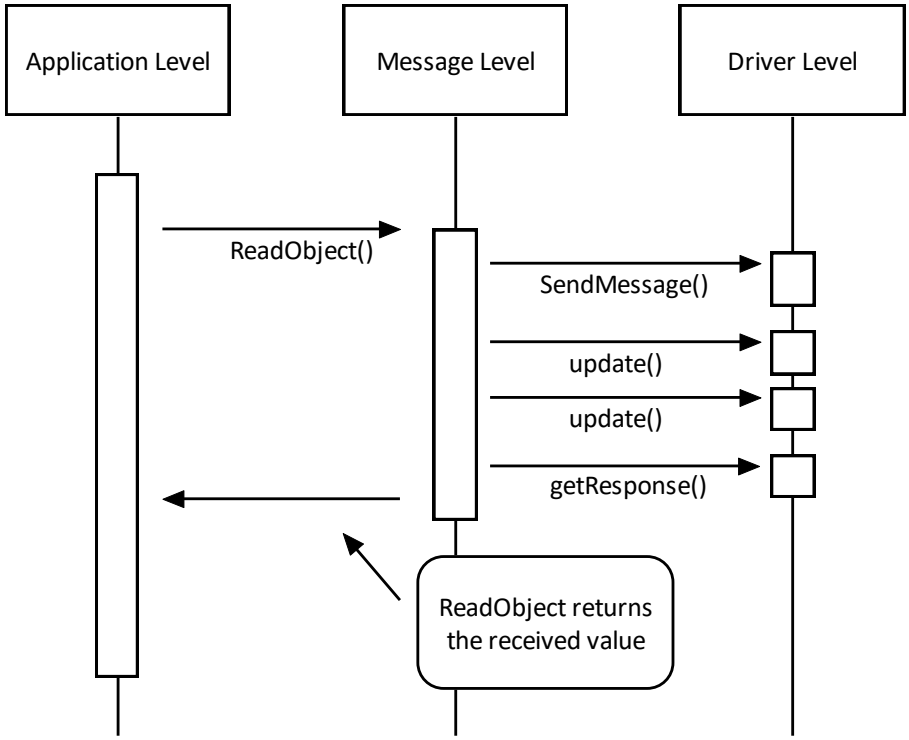

Figure 11 call sequence of reading a parameter or actual value in a blocking call

A non-blocking ReadObject()

Waiting for an answer in the blocking approach takes some time, at least compared to the typical acceptable delays in an embedded application controller. Non-blocking calls avoid being stuck in such a case.

In such a non-blocking implementation where the application level can't afford to actively wait for the response being received and processed the application level software would start the interaction by a non-blocking **MsgHandler::ReadObject()** call and then cyclically call the **MsgHandler::Update()** to check the RS232 for received characters, accumulate the response frame and finally process it. The **MsgHandler::Update()** could as a first step itself call the **Uart::Update()** which then would flag to the caller whether a frame can be processed or is still busy.
Such an application layer implementation could be embedded into some patters where an endless main loop is responsible for different tasks like updates of higher level GUI or communication too.
The simple sequence of **Msghandler::WriteObject()** and **MsgHandler::ReadObject()** calls has then to be embedded in a step-sequence where a next command is sent only when the result of the preceding call has been received. Such an approach is not uncommon for asynchronous reception of answers in any RS232 based communication. It is necessary as typical loop times are shorter than any RS232 based asynchronous communication.

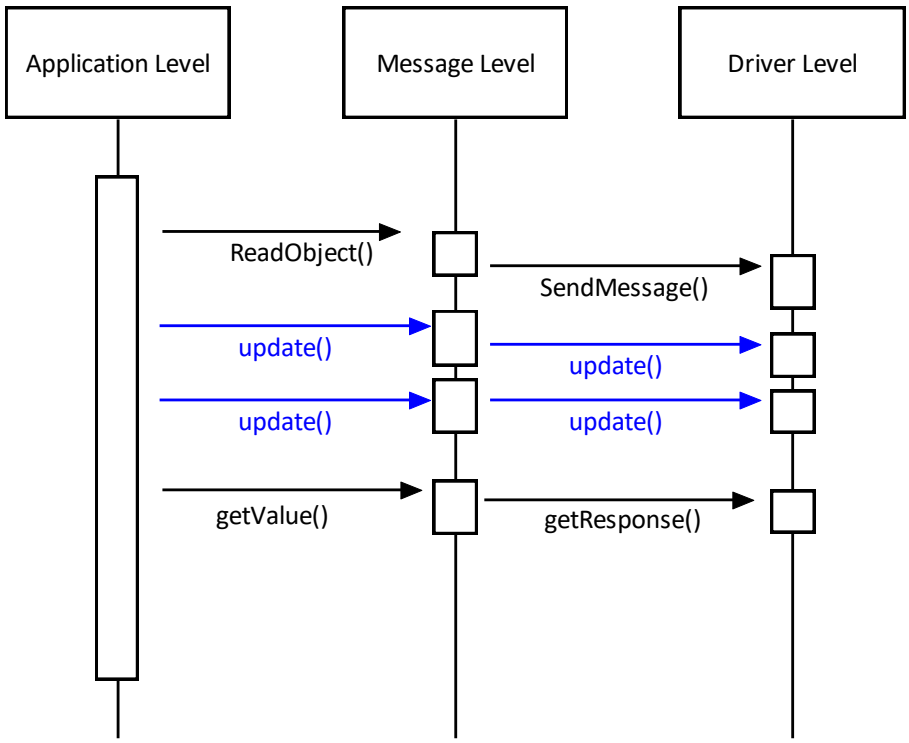

Figure 12 call sequence of reading a parameter or actual value in a non-blocking call

Dealing with time-out

In both cases the expected answer can time-out which means there could have been a disturbance on the communication which damaged a frame. During the cyclic **Update()** calls a counter can be increased or an internal time-source checked to terminate any of these calls when a time-out threshold is reached. It is the responsibility of the application layer whether such a non-successful call has then to be repeated or whatever error handling has to be implemented.

Step 3 – using notifications (optional)

So far, we disabled asynchronous messages like EMCY messages or automatic updates of the status word. Such an approach is pretty robust but lacks efficiency.
To reduce the communication load accepting status or error notifications being sent can be convenient.
Messages can then be received at any time. Therefore the **MsgHandler::Update()** should be called cyclically independent from waiting for a **ReadObject()** or **WriteObject()** response. The implementation then is similar to the one using the non-blocking call, but it then could be convenient to hold a copy of the latest received error in the MsgHandler for the application to get an update even without sending a request. Same for the drive status-word 0x6041.00.
Unfortunately, in an environment having some electrical disturbance relying on asynchronous updates of the status-word alone is not robust enough – these notifications can be disturbed too. Therefore, the time-out mechanism can be a two-fold. Where in a first step we rely on responses to changes triggered by the control-word should be received directly by notification and only if this takes too long explicitly send a request to the status-word.

Appendix

Command Frame

All messages defined for the FAULHABER MotionController V3.0 RS232 communication are built as given in Table 1. The content of the Command code determines how the Data part of the frame is to be filled or interpreted.

Table 1 – General structure of a command frame

Byte	Name	Meaning
0	SOF	Character ‚S' – hex 0x53
1	User data lenght	Without delimiters
2	Node-id	
3	Command code	See additional tables: 0x00: boot message 0x01: SDO read request/response 0x02: SDO write request/response 0x03: SDO abort request of error response 0x05: status word notification 0x07: EMCY notification
4 … n	Data	Depending on the type of request/response/notification
n+1	CRC	
n+2	EOF	Character ‚E' - hex 0x45

Parameter read request / response (SDO)

Any read access to a readable parameter uses the SDO read request and receives either the response as an acknowledge or an SDO error (see RS232 manual).
Contents of the Data field of Table 1 here is then the object to be read identified by its index and sub-index.
The response repeats index and sub-index and adds the value to the frame.

Table 2 SDO read parameter request

Byte	Content	Description
1	7	User data length 7 bytes
2	Node number	Node number
3	0x01	SDORead command
4	Index LB	Index of the LB object entry
5	Index HB	Index of the HB object entry
6	Subindex	Subindex of the object entry
7	CRC	Checksum

Table 3 SDO read parameter response

Byte	Content	Description
1	Length	User data length > 7 bytes
2	Node number	Node number
3	0x01	SDORead command
4	Index LB	Index of the LB object entry
5	Index HB	Index of the HB object entry
6	Subindex	Subindex of the object entry
7–N	Value	Current value of the specified object entry
(N+1)	CRC	Checksum

Parameter write request / response (SDO)

Any write access to a writable parameter uses the SDO write request and receives either the response as an acknowledge or an SDO error (see RS232 manual).
Contents of the Data field of Table 1 here is then the object to be read identified by its index and sub-index + the value to be written to the identified object.
The response repeats index and sub-index but does not carry the data anymore.

Table 4 SDO write parameter request

Byte	Content	Description
1	Length	User data length > 7 bytes
2	Node number	Node number
3	0x02	SDOWrite command
4	Index LB	Index of the LB object entry
5	Index HB	Index of the HB object entry
6	Subindex	Subindex of the object entry
7–N	Value	New value for the specified object entry
(N+1)	CRC	Checksum

Table 5 SDO write parameter response

Byte	Content	Description
1	7	User data length 7 bytes
2	Node number	Node number
3	0x02	SDOWrite command
4	Index LB	Index of the LB object entry
5	Index HB	Index of the HB object entry
6	Subindex	Subindex of the object entry
7	CRC	Checksum

Notifications

Boot message

The boot message is sent by the MotionContoller after power-up when it is not configured to operate in net-mode and asynchronous messages are not disabled. The device name is ascii coded.

Table 6

Byte	Content	Description
1	Length	User data length > 4 bytes
2	Node number	Node number
3	0x00	BootUp command
4-N	Device Name	Device name as boot-up message
(N+1)	CRC	Checksum

Status word change notification

The status-word changed notification is sent out by the MotionController when not in net-mode and asynchronous messages are not disabled whenever the contents of the drive status word at 0x6041.00 changes.

Table 7

Byte	Content	Description
1	6	User data length 6 bytes
2	Node number	Node number
3	0x05	Statusword command
4	Statusword LB	Current value of the statusword in accordance with CiA402
5	Statusword HB	Current value of the statusword in accordance with CiA402
6	CRC	Checksum

Code examples

Uart::Update()

```
void MCUart::Update(uint32_t actTime)
{
  //default is to fetch received bytes and store them in the
  //message buffer – unless we don't
  bool store = true;

  if(state == eUartOperating)
  {
    while(Serial1.available())
    {
      //read the first char
      uint8_t inChar = (uint8_t)Serial1.read();
      //now add it to the buffer if applicable
      if(rxIdx < UART_MAX_MSG_SIZE)
      {
        if(rxIdx == 0)
        {
          rxSize = UART_MIN_MSG_SIZE;
          if(inChar ==  MsgPrefix)
          {
            To_Threshold = actTime + MsgTimeout;
            isTimerActive = true;
          }
          else
            store = false;
        }
        else if (rxIdx == 1)
        {
          rxSize = inChar + 2;
        }

        //now store or not store the char
        if(store)
        {
          To_Threshold = actTime + MsgTimeout;

          RxMsg.u8Data[rxIdx++] = inChar;
          //check for finished
          if(rxIdx == rxSize)
          {
            //all characters received
            rxIdx = 0;

            isTimerActive = false;

            if(inChar == MsgSuffix)
            {

            if(OnRxCb.callback != NULL)
              OnRxCb.callback(OnRxCb.op,(void *)&RxMsg);
```

```
            }
          }
        }
      }
      else
      {
        //overflow
        rxIdx = 0;
        rxSize = 0;
      }
    }

    if((isTimerActive) && (To_Threshold < actTime))
    {
      OnTimeOut();
      isTimerActive = false;
      //once again set a time-out which has to elapse before new
      //messages are to be handled
      To_Threshold = actTime + MsgTimeout;
      state = eUartTimeout;
    }
  }
  else
  {
    //we are in TO state
    if(To_Threshold < actTime)
    {
      //elapsed
      state = eUartOperating;
    }
  }
}
```

MsgHandler::CalcCRC()

```
uint8_t MsgHandler::CalcCRC(const uint8_t *buffer,int len)
{
  uint8_t calcCRC = 0xFF;
  for(uint8_t i = 0;i<len;i++)
  {
    calcCRC = calcCRC ^((uint8_t *)buffer)[i];
    for(uint8_t j = 0;j < 8;j++)
    {
      if(calcCRC & 0x01)
        calcCRC = (calcCRC >> 1) ^ 0xd5;
      else
        calcCRC = (calcCRC >> 1);
    }
  }
return calcCRC;
```

Rechtliche Hinweise

Urheberrechte. Alle Rechte vorbehalten. Ohne vorherige ausdrückliche schriftliche Zustimmung der Dr. Fritz Faulhaber & Co. KG darf diese Application Note oder Teile dieser unabhängig von dem Zweck insbesondere nicht vervielfältigt, reproduziert, gespeichert (z.B. in einem Informationssystem) oder be- oder verarbeitet werden.
Gewerbliche Schutzrechte. Mit der Veröffentlichung, Übergabe/Übersendung oder sonstigen Zur-Verfügung-Stellung dieser Application Note werden weder ausdrücklich noch konkludent Rechte an gewerblichen Schutzrechten, übertragen noch Nutzungsrechte oder sonstige Rechte an diesen eingeräumt. Dies gilt insbesondere für gewerbliche Schutzrechte, die mittelbar oder unmittelbar den beschriebenen Anwendungen und/oder Funktionen dieser Application Note zugrunde liegen oder mit diesen in Zusammenhang stehen.
Kein Vertragsbestandteil; Unverbindlichkeit der Application Note. Die Application Note ist nicht Vertragsbestandteil von Verträgen, die die Dr. Fritz Faulhaber GmbH & Co. KG abschließt, und der Inhalt der Application Note stellt auch keine Beschaffenheitsangabe für Vertragsprodukte dar, soweit in den jeweiligen Verträgen nicht ausdrücklich etwas anderes vereinbart ist. Die Application Note beschreibt unverbindlich ein mögliches Anwendungsbeispiel. Die Dr. Fritz Faulhaber GmbH & Co. KG übernimmt insbesondere keine Gewährleistung oder Garantie dafür und steht auch insbesondere nicht dafür ein, dass die in der Application Note illustrierten Abläufe und Funktionen stets wie beschrieben aus- und durchgeführt werden können und dass die in der Application Note beschriebenen Abläufe und Funktionen in anderen Zusammenhängen und Umgebungen ohne zusätzliche Tests oder Modifikationen mit demselben Ergebnis umgesetzt werden können. Der Kunde und ein sonstiger Anwender müssen sich jeweils im Einzelfall vor Vertragsabschluss informieren, ob die Abläufe und Funktionen in ihrem Bereich anwendbar und umsetzbar sind.
Keine Haftung. Die Dr. Fritz Faulhaber GmbH & Co. KG weist darauf hin, dass aufgrund der Unverbindlichkeit der Application Note keine Haftung für Schäden übernommen wird, die auf die Application Note und deren Anwendung durch den Kunden oder sonstigen Anwender zurückgehen. Insbesondere können aus dieser Application Note und deren Anwendung keine Ansprüche aufgrund von Verletzungen von Schutzrechten Dritter, aufgrund von Mängeln oder sonstigen Problemen gegenüber der Dr. Fritz Faulhaber GmbH & Co. KG hergeleitet werden.
Änderungen der Application Note. Änderungen der Application Note sind vorbehalten. Die jeweils aktuelle Version dieser Application Note erhalten Sie von Dr. Fritz Faulhaber GmbH & Co. KG unter der Telefonnummer +49 7031 638 688 oder per Mail von mcsupport@faulhaber.de.

Legal notices

Copyrights. All rights reserved. This Application Note and parts thereof may in particular not be copied, reproduced, saved (e.g. in an information system), altered or processed in any way irrespective of the purpose without the express prior written consent of Dr. Fritz Faulhaber & Co. KG.
Industrial property rights. In publishing, handing over/dispatching or otherwise making available this Application Note Dr. Fritz Faulhaber & Co. KG does not expressly or implicitly grant any rights in industrial property rights nor does it transfer rights of use or other rights in such industrial property rights. This applies in particular to industrial property rights on which the applications and/or functions of this Application Note are directly or indirectly based or with which they are connected.
No part of contract; non-binding character of the Application Note. The Application Note is not a constituent part of contracts concluded by Dr. Fritz Faulhaber & Co. KG and the content of the Application Note does not constitute any contractual quality statement for products, unless expressly set out otherwise in the respective contracts. The Application Note is a non-binding description of a possible application. In particular Dr. Fritz Faulhaber & Co. KG does not warrant or guarantee and also makes no representation that the processes and functions illustrated in the Application Note can always be executed and implemented as

described and that they can be used in other contexts and environments with the same result without additional tests or modifications. The customer and any user must inform themselves in each case before concluding a contract concerning a product whether the processes and functions are applicable and can be implemented in their scope and environment.

No liability. Owing to the non-binding character of the Application Note Dr. Fritz Faulhaber & Co. KG will not accept any liability for losses arising from its application by customers and other users. In particular, this Application Note and its use cannot give rise to any claims based on infringements of industrial property rights of third parties, due to defects or other problems as against Dr. Fritz Faulhaber GmbH & Co. KG.

Amendments to the Application Note. Dr. Fritz Faulhaber & Co. KG reserves the right to amend Application Notes. The current version of this Application Note may be obtained from Dr. Fritz Faulhaber & Co. KG by calling +49 7031 638 688 or sending an e-mail to mcsupport@faulhaber.de.