Control MC V3.0 MotionController via RS232 An Arduino Library

Summary

This ApplicationNote is intended to show how a connection from an embedded µController to any FAULHABER MC V3.0 via RS232 can be established. An example library has been developed using an Arduino nano[1]. It is meant to be an example how such a communication stack could be implemented on a µController. Details will be different for non-Arduino environments, but these changes are mentioned.

Applies To

Any MotionController out of the MC V3.0 family having an RS232 interface

Description

General Setup

The used setup consists of one Arduino nano every and 1 … 4 MC 5004 P controlled by the code examples. The TTL levels of the Arduinos Rx/Tx ports @ PC4 and PC5 are converted to RS232 levels using a RS232 transceiver based on one of the popular MAX232 – here an old IF232 from elmicro[2].
If multiple MC V3.0 are connected to a single host RS232, the interfaces of the devices are connected in parallel. Rx/Tx must be swapped between the mastr and the slaves.

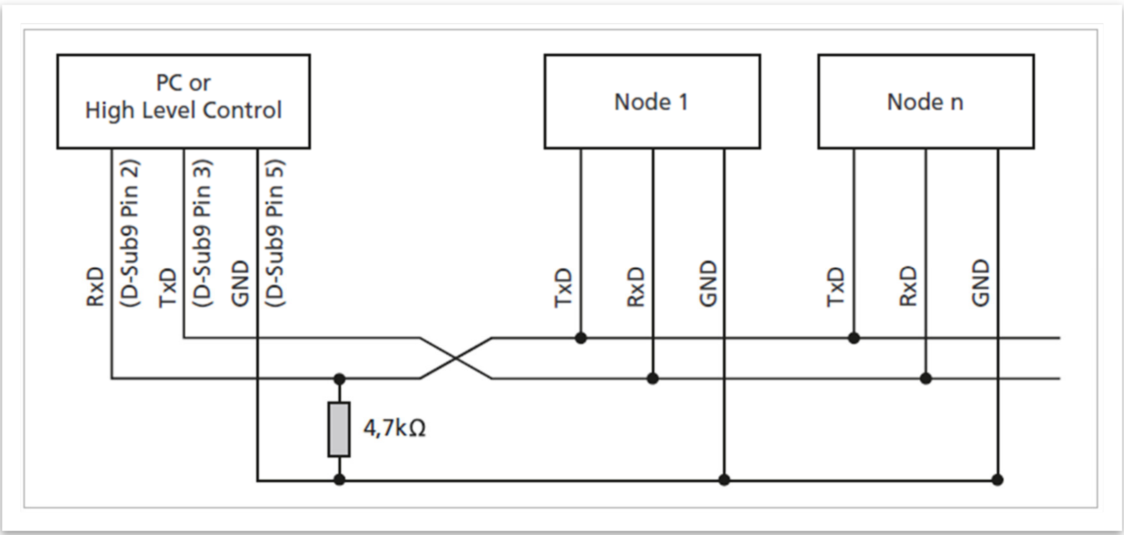


Figure 1 Connection between a RS232 host and 1 ... n slaves

[1] https://www.arduino.cc/en/Guide/NANOEvery
[2] https://elmicro.com/de/ifmodule.html

When a RS232 network is to be used the FAULHABER MCs must be configured for RS232 net-mode to avoid any asynchronous messages. As any MC has to release the TX line again after each transmission, the Tx-line of the drives is tied to GND via a resistor.

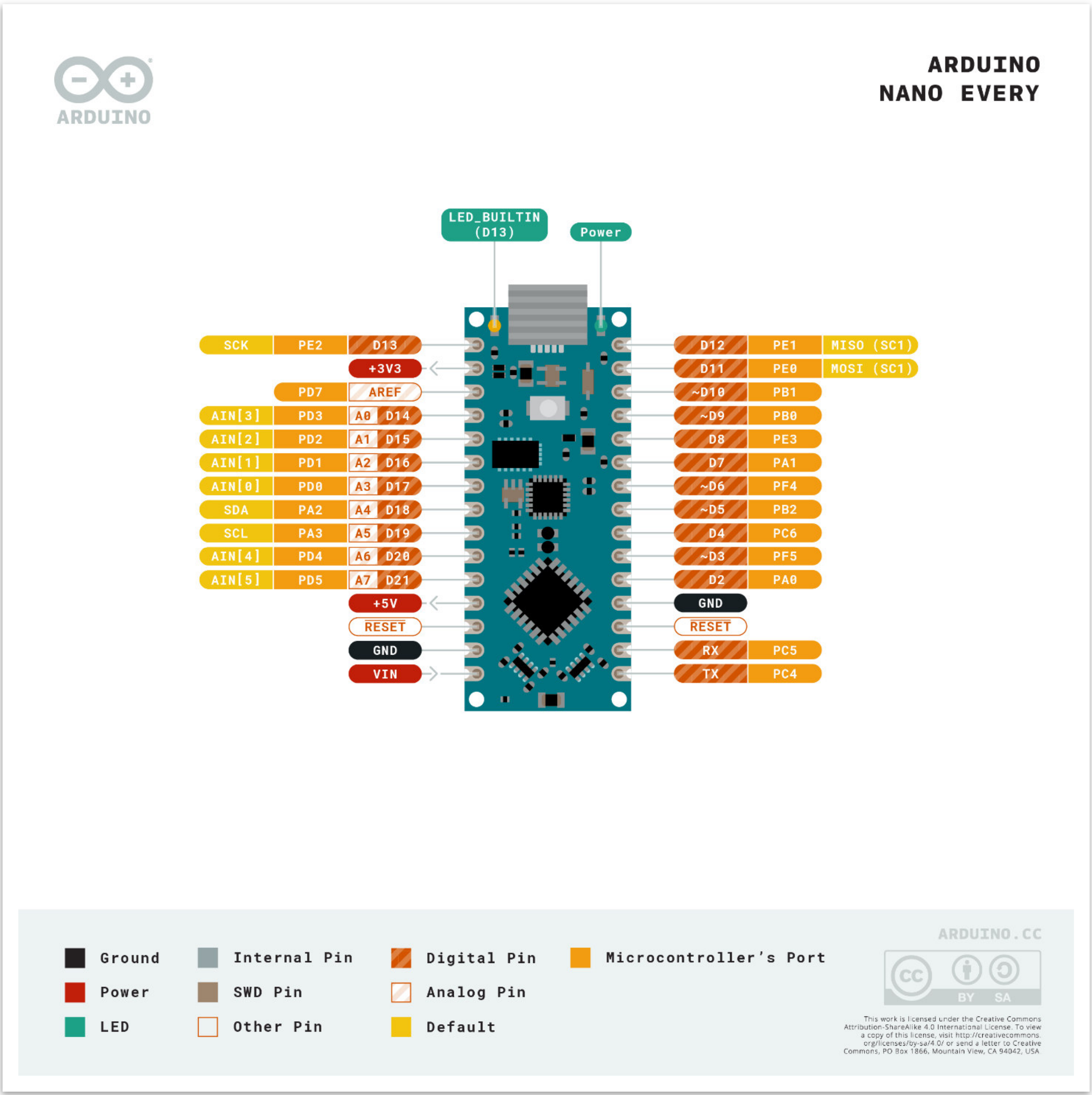


Figure 2 Pinout of an Arduino nano every[3]

[3] https://store.arduino.cc/arduino-nano-every

Arduino Example Environment

The Arduino implementation here uses the built-in Serial library of the Arduino environment. Different from a bare metal µController there is no configuration of whatever peripheral module of the µController based on this approach.

Apart from that the software components and methods could be transferred in a very similar manner to any µController.

The code structure of an Arduino .ino sketch is:

```
//--- defines ---

//--- includes ---

//--- globals ---

void setup() {
  // put your setup code here, to run once:
  pinMode(13,OUTPUT);
  // Debug Port
  Serial.begin(500000);
  …
}

void loop() {
  // put your main code here, to run repeatedly:
  …
}
```

Code 1 Basic structure of an Arduino program

Libraries like the one used here would be included by adding an appropriate #include statement at the top of the file.

Any one-time configuration like opening the interfaces, dynamic configuration of the used software instances and even connecting them would usually happen at the beginning in the setup() method of the framework.

The loop() is then called over and over. There is no guaranteed time to this. If the loop takes longer to be executed the complete execution simply is postponed. So there explicitly is no real-time behavior associated with the loop. The only convention is: it’s restarted again after being executed.

There seem to be some services which are executed by the framework between consecutive calls of loop() though.

An application using the Serial library could implement an event-handler serialEvent() which would be called at the end of each loop to check for newly received characters.

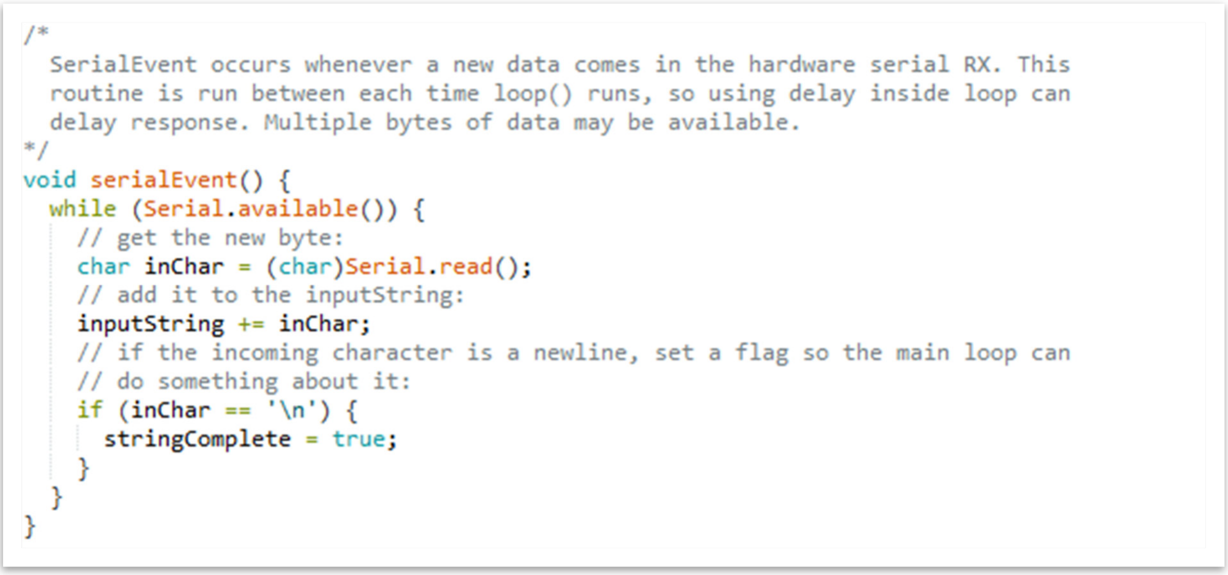

```
/*
  SerialEvent occurs whenever a new data comes in the hardware serial RX. This
  routine is run between each time loop() runs, so using delay inside loop can
  delay response. Multiple bytes of data may be available.
*/
void serialEvent() {
  while (Serial.available()) {
    // get the new byte:
    char inChar = (char)Serial.read();
    // add it to the inputString:
    inputString += inChar;
    // if the incoming character is a newline, set a flag so the main loop can
    // do something about it:
    if (inChar == '\n') {
      stringComplete = true;
    }
  }
}
```

Figure 3 Example code of a serialEvent()[4]

In order to keep the Serial interface updated a short cycle for the loop() seems to be the best idea. The library follows this lead and does not block the loop but implements whatever behavior in a manner which allows for cyclic execution.

A cyclic approach is implemented with the MC V3.0 Arduino RS232 library – loop() can be kept short. But as the serialEvent() would be executed in between the loops anyway - which is kind of implicit polling - the library here actually explicitly uses polling the Serial interface for new characters in each loop based on a check of available characters:

```
while(Serial1.available())
{
  //read the first char
  uint8_t inChar = (uint8_t)Serial1.read();
  …
}
```

Code 2 call to the Serial1 within the MCUart

Table 1 Serial resources used by the Arduino MC V3.0 RS232 library

Resource	usage
Serial	connected to the UART to USB bridge used for development
Serial1[5]	the interface using PC4 and PC5 connected to the drives via level shifter

[4] https://www.arduino.cc/en/Tutorial/BuiltInExamples/SerialEvent

[5] On different boards different Serial ressources are available. It is preferred to use a non shared port to communicate with the MotionController. MCUart.cpp would have to be modified to use whatever different port shall be used then.

Library Overview

The Arduino MC V3.0 RS232 library consists of different components offering different levels of abstraction:

Table 2 components of the Arduino MC V3.0 RS232 library

component	description
MCUart	Open and configure the actual interface. Send messages received from higher layers adding the prefix and suffix characters using `Serial1.write()`. Receive characters using `Serial1.read()` and try to build messages out of it by scanning for a prefix character "S", a message length and a suffix character "E". Notify higher layer services when a complete message has been received.
MsgHandler	Registers at the MCUart. Check the CRC on messages received from the MCUart. Only message having a valid CRC will be distributed to higher layers of the stack. Add the CRC to messages to be sent before handing them over to the MCUart. Implements a semaphore to block multiple Drives to use the interface at the same time.
MCTimer	Abstract timer service for cyclic actions. Not used in the library at the moment.
SDOHandler	Must be registered at the MsgHandler. Creates SDO request messages (read or write a parameter) and handles the response of the drives. Implements the complete SDO service of a single drive. Expedited transfer of the complete data in a single exchange is supported only.
MCNode	Access the status- and controlword of a single drive by creating the read- and write requests. Uses the SDO service via SDOHandler to read the statusword. Uses the controlword service to write to a drives controlword.
MCDrive	Uses its SDOHandler and the MCNode component to implement services on drive level like enabling or disabling, moving to a position or at an intended speed. Implements the main interface for a user of this library.

The relations between the software components are detailed Figure 4. Up to 4 MCDrive components can register at a single MSGHandler component using the default of the library which is also the max recommended size of these small network configurations.

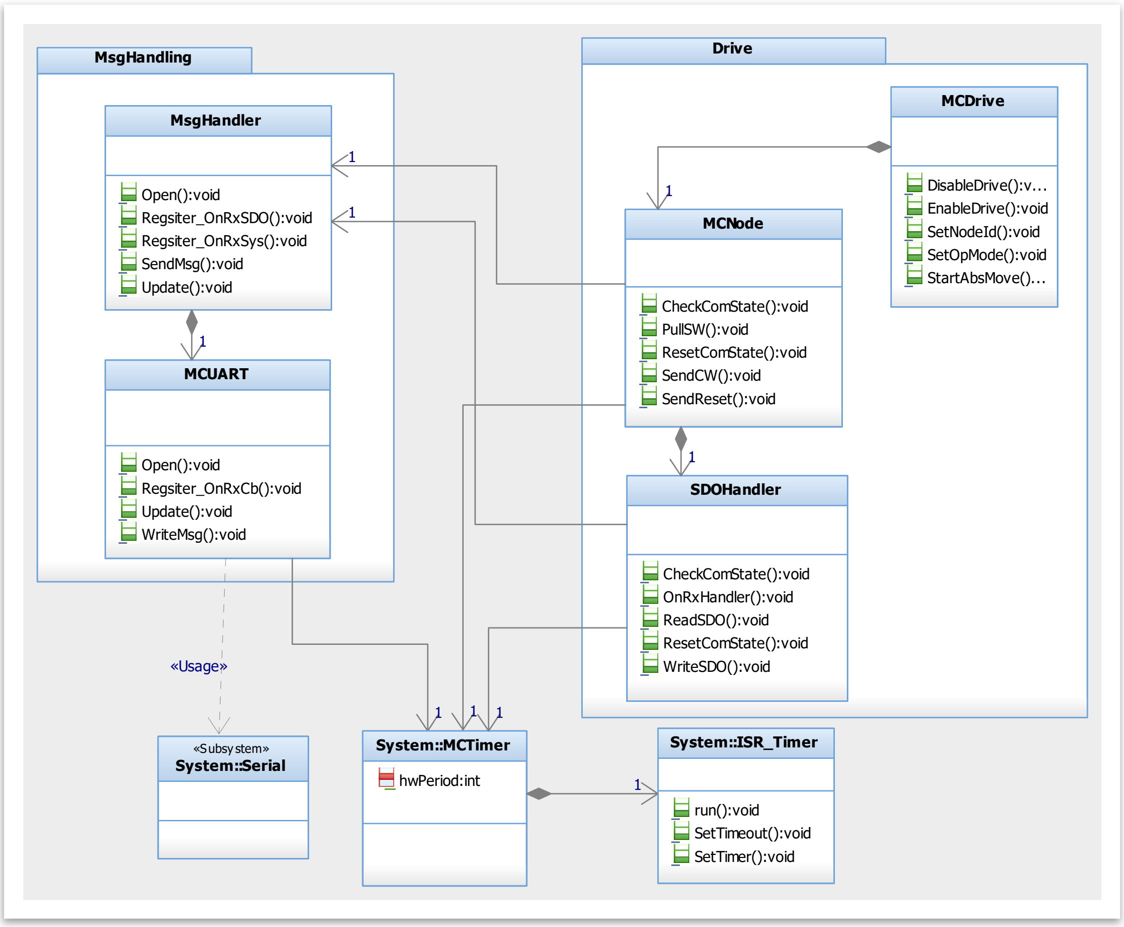


Figure 4 Class diagram of the embedded RS232 library Rev A

Behavior of the Library

There are three major use-cases of the library components.

Create the instances

Even in RS232-network mode the drives share a single RS232 interface. On the host side the library is always using a single instance of the MCUart.cpp which is a part of the MsgHandler.cpp class. MCUart is associated with MsgHander.cpp forming a composition. In any application it is sufficient to include MsgHandler.h and create a single instance of the MsgHandler. No need to care for MCUart.h.
MCDrive and its sub-classes are associated to a single instance of the MsgHandler. To use it, include MCDrive.h and create at least on instance of MCDrive.
Up to 4 instances of the MCDrive can be connected to the MsgHandler. During the setup() the instances must be connected to the MsgHandler explicitly.

```
//--- defines ---

//--- includes ---
#include <MCTimer.h>
#include <MsgHandler.h>
#include <MCDrive.h>
#include <stdint.h>

//--- globals ---

extern MCTimer OsTimer;
MsgHandler MCMsgHandler;
MCDrive Drive_A;

void setup() {
  // Debug Port
  Serial.begin(500000);
  //here we really start
  MCMsgHandler.Open(115200);
  Drive_A.SetNodeId(DriveIdA);
  Drive_A.Connect2MCTimer(&OsTimer);
  Drive_A.Connect2MsgHandler(&MCMsgHandler);
}
```

Code 3 start and initialization using the Arduino MC V3.0 RS232 library

Send a basic request using SDOHandler and MCNode

The MC V3.0 RS232 protocol defines a couple of services which are identified by the command code in the 4th byte of any command frame.

Table 3 Structure of a MC V3.0 RS232 protocol frame

Byte	Name	Meaning
1. Byte	SOF	Character (S) as Start of Frame
2. Byte	User data length	Telegram length without SOF/EOF (packet length)
3. Byte	Node number	Node number of the slave (0 = Broadcast)
4. Byte	Command code	See Tab. 2
5th – Nth byte	Data	Data area (length = packet length – 4)
(N+1). byte	CRC	CRC8 with polynomial 0xD5 over byte 2–N
(N+2). byte	EOF	Character (E) as End of Frame

This library implements read and write access to standard drive parameters via the SDO Read and SDO Write which is handled by the SDOHandler.cpp.
Access to the Controlword as well as to the Statusword and receiving of asynchronous Boot-up and EMCY are implemented via MCNode.cpp.
Each MCNode.cpp includes a single instance of a SDOHandler and a MCNode by means of composition. Therefore they don’t have to be taken care of in any application. Creating an instance of the MCDrive will include the services.

Table 4 List of services implemented in the library

Command code	Name	Function
0x00	`Boot up`	Boot-up message / Reset Node (Receive / Request)
0x01	`SDO Read`	Read the object dictionary entry (Request / Response)
0x02	`SDO Write`	Write an object dictionary entry (Request / Response)
0x03	`SDOError`	SDO error (abort request / error response)
0x04	`Controlword`	Writing the controlword (request / response)
0x05	`Statusword`	Reception of the statusword (receive)
0x06	`Trace Log`	Trace Request for Trace Logger (Request / Response)
0x07	`EMCY`	Reception of an emergency message (receive)

To send a SDO read- or write request the methods SDOHandler::ReadSDO() and SDOHandler::WriteSDO() must be called cyclically. Internally they implement a step-sequence where the real request is sent out only if SDOHandler is in eIdle state and ends up in eDone-state if successful or eError-state if not.
After the request has been sent successfully a timer is started to implement a timeout behavior to either re-send the request if no response is received (eRetry) or finally end up in an error state (eError).
Meanwhile the application code in the loop() can be executed over and over until a response finally arrives or in a multi-axis configuration even until the request was really sent.

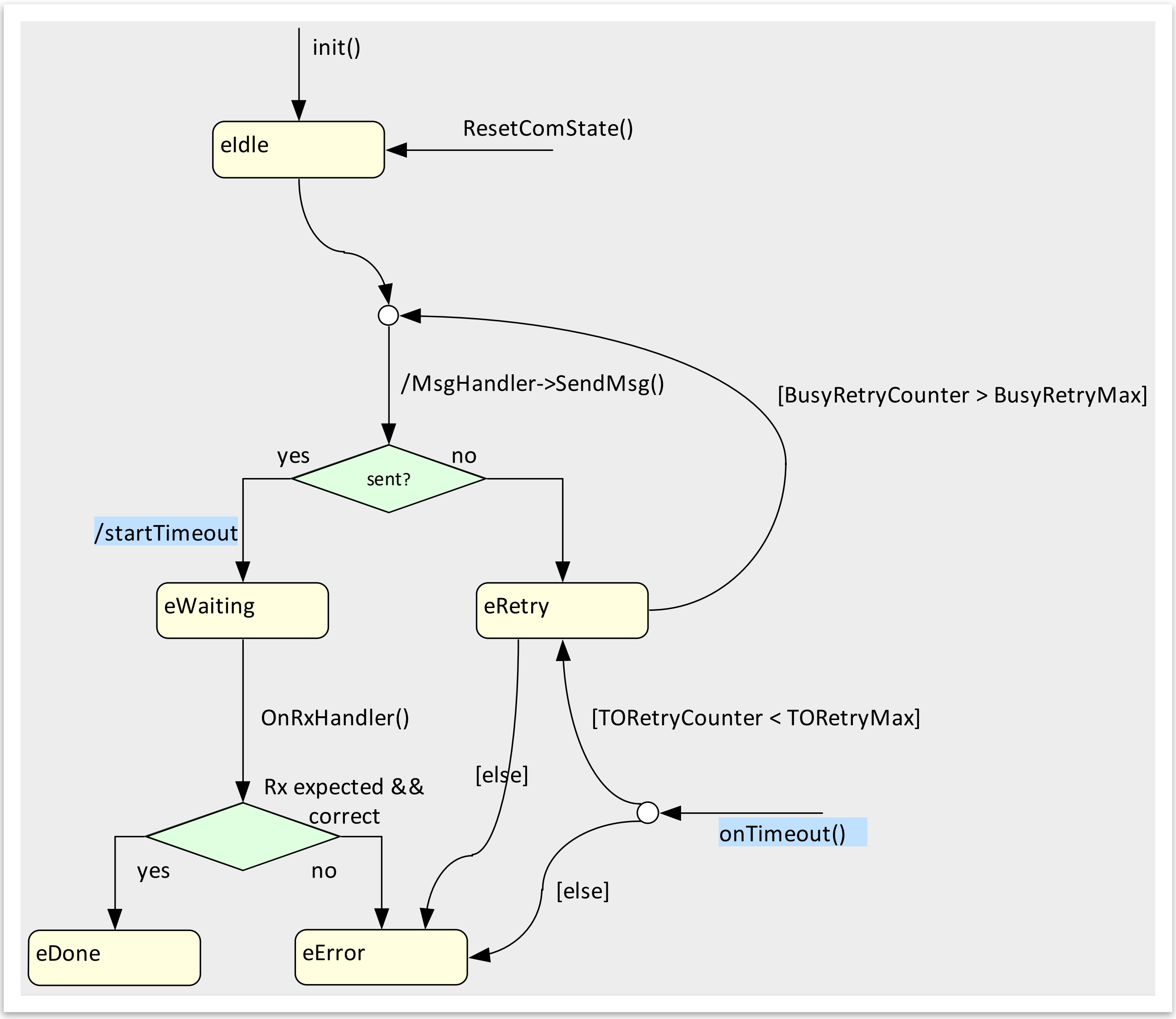


Figure 5 Step-Sequence of SDO access

Using these services has to ensure to cyclically call the SDOHandler::ReadSDO() or SDOHandler::WriteSDO() until they finally end up in the eDone state.
The simplest example is the MCDrive::SetOpMode(int8_t OpMode) where a single parameter – the OpMode – is written to object 0x6060.00 : Modes of operation of the drive.

```
DriveCommStates MCDrive::SetOpMode(int8_t OpMode)
{

  if(SDOAccessState == eDone)
  {
    ThisNode.ResetComState();
    SDOAccessState = eIdle;
    OpModeReported = OpModeRequested;
    RxTxState = eMCDone;

    #if(DEBUG_DRIVE & DEBUG_SetOpMode)
    Serial.println("Drive: OpMode set ");
    #endif
  }
  else
  {
    //skip the first access if not required
    if(OpModeReported == OpMode)
      RxTxState = eMCDone;
    else
    {
      OpModeRequested = OpMode;

      #if(DEBUG_DRIVE & DEBUG_SetOpMode)
      if(SDOAccessState == eIdle)
      {
        Serial.print("Drive: OpMode ");
        Seriel.print(OpMode, DEC);
        Serial.println(" Request");
      }

      #endif
      SDOAccessState = ThisNode.WriteSDO(0x6060, 0x00,
                                             (uint32_t *)&OpModeRequested,1);
    }
    RxTxState = eMCWaiting;
  }

  //always check whether a SDO is stuck final
  return CheckComState();
}
```

Code 4 Usage of the SDOHandler within MCDrive::SetOpMode()

Here the new OpMode is sent only if it’s different that the last one sent to the drive. If it has to be sent, this is done using the WriteSDO() method MCNode of the built-in MCDrive.
The blue lines are used for debugging purpose only. The actual code therefore is compact. To have this work the SDOHandler component should ideally be in eIdle state when MCDrive::SetOpMode(int8_t OpMode) is called the first time. The state of the request can be checked based on the return value of the method which results in eMCDone when the request is finished.

Interact with the MCDrive component

The application code to call whatever method of the MCDrive is similar:

```
void loop() {
    // put your main code here, to run repeatedly:
    DriveCommStates NodeState;
    uint32_t currentMillis = millis();

    Drive_A.SetActTime(currentMillis);
    MCMsgHandler.Update(currentMillis);

    switch(driveStep)
    {
       …
       case 6:
         //set OpMode PV=3
```

```
        if((Drive_A.SetOpMode(3)) == eMCDone)
        {
          driveStep = 7;
          Drive_A.ResetComState();
        }
        break;
      case 7:
      …
   }

   NodeState = Drive_A.CheckComState();
   if((NodeState == eMCError) || (NodeState == eMCTimeout))
   {
      Serial.println("Main: Reset NodeA State");
      Drive_A.ResetComState();
      //should be avoided in the end
      driveStep = 0;
   }
}
```

Code 5 Usage of the MCDrive methods out of the main loop

Within the loop the first calls would be to update the MsgHandler and check the MCDrive for time-outs by setting the current time. The actual update of the behavior is done only by cyclically calling the methods of MCDrive – here MCDrive::SetOpMode().
The key is to call MCDrive::whatever until the response equals eMCDone. Then switch to the next step of the intended behavior.

Methods of MCDrive

Initialization

The methods listed under initialization would usually be used within the on-time setup().

```
void Connect2MsgHandler(MsgHandler *);
```

Is called once within setup() to connect this instance of the MCDrive to the MsgHandler(). The pointer to the instance of the MsgHandler must be given like in Code 3.

```
void Connect2MCTimer(MCTimer *);
```

Connect the MCDrive to the MCTimer. The pointer to the MCTimer must be given. As of now the MCTimer is not used within the MCDrive, it's based on polling as the timer hasn't been stable.
Can be omitted.

```
void SetNodeId(uint8_t);
```

Set the nodeId of the node to be addressed in the range of 1 … 127.

```
void SetTORetryMax(uint8_t);
```

Set a different number of retries due to a time-out. Counting restarts for any single access of either SDO-Handler or MCNode. The default values are 1 retry each.

```
void SetBusyRetryMax(uint8_t);
```

Set a different number of retries due to a busy interface. Counting restarts for any single access of either SDOHandler or MCNode. The default values are 1 retry each.

Cyclic Updates

The methods of cyclic updates are the main common methods used in combination with any behavioral method.

```
void SetActTime(uint32_t);
```

Is to be called cyclically to check for any time-outs. Parameter is the latest value of the millis() call in the loop - see Code 3.

```
DriveCommStates CheckComState();
```

Check the communication status of MCDrive to check whether any fatal error occurred (Code 5). Checking the ComState cyclically can simplify the reaction to any of the fatal errors as they can be dealt with in a single place compared to checking them as a possible response of any call.

```
void ResetComState();
```

Called to switch the communication state of the MCDrive instance back to eMCIdle – similar to the step-sequence in Figure 5.
In Code 5 ResetComState() is used to reset the ComState after successfully ending up in eMCDone.
ResetComState would also be used to restart communication after a fatal error detected on a ComState eMCError or eMCTimeout.

```
bool IsLive();
```

If a single drive is connected only it will send its boot-message, unless it's configured to operate in net-mode. A received boot message of this drive will result in a `true` response here.
Use this as an additional information only as messages can be disturbed in a RS232 system.

```
uint16_t GetLastError();
```

In case off a drive error the drive can send an EMCY message. EMCY messages are asynchronous and can't be sent in a net-mode configuration.
If an EMCY message has been received the received error code can be read out via this call.

```
DriveCommStates SendReset();
```

Not implemented yet.

User Interface to OpModes

The methods to start a move or change parameters of a drive will return a DriveCommState. Internally all these methods again implement a simplified version of the step sequence in Figure 5.
States would be:

DriveCommState	Reached when
eMCIdle	The initial state or after a ResetComState()
eMCWaiting	After a command has been started but not done
eMCBusy	Not used so far
eMCDone	After a command has been completed
eMCError	Reached only if any of the lower layers ends up in an error state
eMCTimeout	Reached only if any of the lower layers ends up in a timeout state

```
DriveCommStates UpdateDriveStatus();
```

Will update the local copy of the Modes of Operation Display value as well as the local copy of the status word.
Could be used in the very beginning of whatever interaction to get an information of which status the drive actually has.
Returns DriveCommState == eMCDone when finished.

```
uint16_t GetSW();
```

Returns the local copy of the statusword. No actual update done here.

```
DriveCommStates EnableDrive();
```

Tries to enable the drive by stepping through the state-machine. When DriveCommState == eMCDone the drive is in operational state.

If the drive fails to enable due to a blocking error like an out of range supply voltage no error will occur, but the call will not reach the eMCDone state. A time-out implemented around the EnableDrive() might be an option then.
EnableDrive() will try to clear the drive from the error state of the drive state-machine if necessary.
Returns DriveCommState == eMCDone when the drive reaches the operation enabled state.

```
DriveCommStates DisableDrive();
```

Disables the drive by sending a disable voltage command. The drive is not stopped actively in such a case. If it must be stopped, use StopDrive().
Returns DriveCommState == eMCDone when the drive reaches the switch on disabled state.

```
DriveCommStates StopDrive();
```

Stops the drive by switching into the stopped state of the drive state-machine. Depending on the configuration of the Quick-Stop behavior the drive either remains in the stopped state or does an automatic transition into the switch on disabled state after reaching 0-speed.
Either way it can be re-enabled by calling EnableDrive().
Returns DriveCommState == eMCDone when the drive ends up in stopped state or in switch on disabled state.

```
DriveCommStates SetOpMode(int8_t);
```

Set the Mode of Operation parameter of the drive. Does not check whether the requested OpMode is a valid one and does not check whether the Modes of Operation Display reflects the requested OpMode.
Returns DriveCommState == eMCDone when the write access to the Mode of Operation parameter is finished.

```
DriveCommStates SetProfile(uint32_t, uint32_t, uint32_t, int16_t);
```

Set a new set of profile parameters. The sequence of the parameters is:

- Profile Acceleration (0x6083)
- Profile Deceleration (0x6084)
- Profile Velocity (0x6081)
- Motion Profile Type (0x6086)

Returns DriveCommState == eMCDone when the write access to all parameters is finished.

```
DriveCommStates StartAbsMove(int32_t, bool);
```

Switch the drive to PP mode, set a new absolute target position and start the move.
This is a move to a defined position within the application.
The first parameter is the absolute target position in user scaling according to the factor group.
Second parameter flags whether the new move has to be started immediately, even when a preceding move is still active (immediate == true) or if it shall start only after the preceding move has been finished.
Returns DriveCommState == eMCDone when the drive acknowledged the command. This usually is: the drive actually started the move. To check whether the last target position has been reached call IsInPos().

```
DriveCommStates StartRelMove(int32_t, bool);
```
Switch the drive to PP mode, set a new relative target position and start the move.
A relative move does not end up at a specific position within the application but will move the given distance starting from either the actual position or the last target position. Which one is going to be the base is configured within the drive using the OpModeOptions parameter 0x233F.
The first parameter is the distance in user scaling according to the factor group.
Second parameter flags whether the new move has to be started immediately, even when a preceding move is still active (immediate == true) or if it shall start only after the preceding move has been finished.
Returns DriveCommState == eMCDone when the drive acknowledged the command. This usually is: the drive actually started the move. To check whether the last target position has been reached call IsInPos().

```
DriveCommStates ConfigureHoming(int8_t);
```
Configures a homing method via 0x6098. Does not start the homing.
In many cases the switch configuration in an application does not change. It's recommended to pre-configure the homing method using the MotionManager and during run-time only start the pre-configured homing.
Returns DriveCommState == eMCDone when the write access to the Homing Method parameter is finished.

```
DriveCommStates StartHoming();
```
Switches the drive to homing mode and starts whatever homing method has been configured. Does not wait for a homing being completed successfully.
Returns DriveCommState == eMCDone when the drive has been switched to homing mode and the start flag in the control word has been given a rising edge.

```
DriveCommStates MoveAtSpeed(int32_t);
```
Switches the drive to PV mode and sets the new target speed as commanded. The target velocity is to be scaled according to the settings of the factor group. For a rotating motor the default is in min^{-1}, for a linear motor the default in mm/s.
Returns DriveCommState == eMCDone when the drive has been switched to PV mode and new target velocity has been successfully written to 0x60FF.

```
DriveCommStates IsInPos();
```
Checks the statusword of the drive cyclically. The actual cycle time of checking the status is configured within the MCDrive.cpp via PullSWCycleTime which defaults to 20ms.
Returns DriveCommState == eMCDone when the drive signals the last position being reached by setting the target reached flag within the status word.
In a sequence of moves which have been sent to the drive without explicitly waiting for a first target being reached, the flag will only be set after the last move has been completed. Details about this behavior can be found in the Drive Functions manual.

```
DriveCommStates IsHomingFinished();
```
Checks the statusword of the drive cyclically. The actual cycle time of checking the status is configured within the MCDrive.cpp via PullSWCycleTime which defaults to 20ms.

Returns DriveCommState == eMCDone when the drive signals the homing sequence being finished success-fully.
As of firmware revision L this final check does not work combined with homing method 37!

Methods for Debugging

The following methods can be utilized to follow the actions below the hood of the MCDrive.

```
CWCommStates GetNodeState();
```

Returns the DriveCommState of the MCDrive instance.

```
SDOCommStates GetSDOState();
```

Returns the SDOCommState of the built-in SDOHandler.

```
CWCommStates GetCWAccess();
```

Returns the CWCommState of the built-in MCNode.

```
uint8_t GetAccessStep();
```

Complex behavior like in SetProfile or StartAbs/RelMove is again implemented using a step-sequence. The actual step the MCDrive is in can be checked using this call.

Step by Step Debugging

Using the Serial Monitor

The debugging capabilities of the original Arduino environment are very basic. While it is reasonable not to use breakpoints in a real-time environment there are of course many cases even in a real-time environment where breakpoints could be used safely.
The more or less only run-time access available is the serial monitor which was a typical approach back when the Arduino environment has been created.
In case of a real-time communication between an Arduino and a MotionController breakpoints will most likely not work. So, we are using the main serial port here for debugging.
In each of the modules there are a lot Serial.print() or Serial.println() statements which can be activated by instrumenting the code.
Instrumentation is done at the head of each module using the same approach. For the MCDrive.cpp it looks like:

```
#define DEBUG_RXMSG        0x0001
#define DEBUG_TO           0x0002
#define DEBUG_ERROR        0x0004
#define DEBUG_UPDATE       0x0010
#define DEBUG_MoveSpeed    0x0020
#define DEBUG_ENABLE       0x0040
#define DEBUG_DISABLE      0x0080
#define DEBUG_STOP         0x0100
#define DEBUG_MOVEPP       0x0200
#define DEBUG_HOME         0x0400
#define DEBUG_RWPARAM      0x0800
#define DEBUG_PULLSW       0x1000

#define DEBUG_DRIVE (DEBUG_TO | DEBUG_ERROR)
```

Code 6 Instrumentation of MCDrive for debugging

Within each method the Serial.print() statements are encapsulated by preprocessor #if #endif statements like:

```
#if(DEBUG_DRIVE & DEBUG_UPDATE)
…
#endif
```

These are the blue lines in Code 4.
The relevant section of the behavior can thus be followed using the serial monitor of the Arduino environment by a fitting configuration of the respective DEBUG_xxx of the modules in question. By default, all modules will report fatal errors only.

Timing

There are a couple of parameters which are defined in the different levels to tailor the access to the RS232. If necessary all of these parameters can be adjusted to the used baud-rate and application requirements in terms of timing.

Module	Parameter and default in ms
MCUart.cpp	`MsgTimeout = 10` After each successfully received character this timeout is restarted. If there is no more character but the frame has not been fully received the reception is reset and the characters are discarded.
MsgHandler.cpp	`MsgHandlerMaxLeaseTime = 20` Any access to the Msghandler will lock the MsgHandler until a response is received. If the response is not received within the MaxLeaseTime the MsgHandler is unlocked automatically.
SDOHandler.cpp	`SDORespTimeOut = 20` Maximum waiting time for a response to either a SDO read request or a write request. After time-out a retry would be started. After too many retries the access would end up in eTimeout state.

MCNode.cpp	`CwRespTimeOut = 20`
	The handshake to a controlword write access is expected in at max this setting. After 50% of the time, the command is re-sent to the drive.
	A first response is the mandatory handshake for any command sent to the drive.
	
	`MaxSWResponseDelay = 50`
	After a new command has been sent to the controlword the drive could react by changing the statusword e.g. when the drive is being enabled or disabled.
	The reception of an updated statusword after an updated controlword can be waited for and if necessary even polled. MCNode::SendCw() does have a parameter for the time-out of the statusword response. If this non-zero the statusword will be polled after the given time if not received asynchronously.

MCDrive.cpp	`MaxSWResponseDelay = 50`
	Is the time used for calls to MCNode::SendCW() if an explicit response is expected.
	
	`PullSWCycleTime = 20`
	Is the time used for cyclic polling the statusword e.g. when checking for a target being reached.

Examples – Using the Library

Single Axis example

Connect2MC_1Node is a simple example which deals with a single instance of MCDrive as in Code 3.
A step-sequence is implemented within loop() where the drive is cycled through different OpModes.

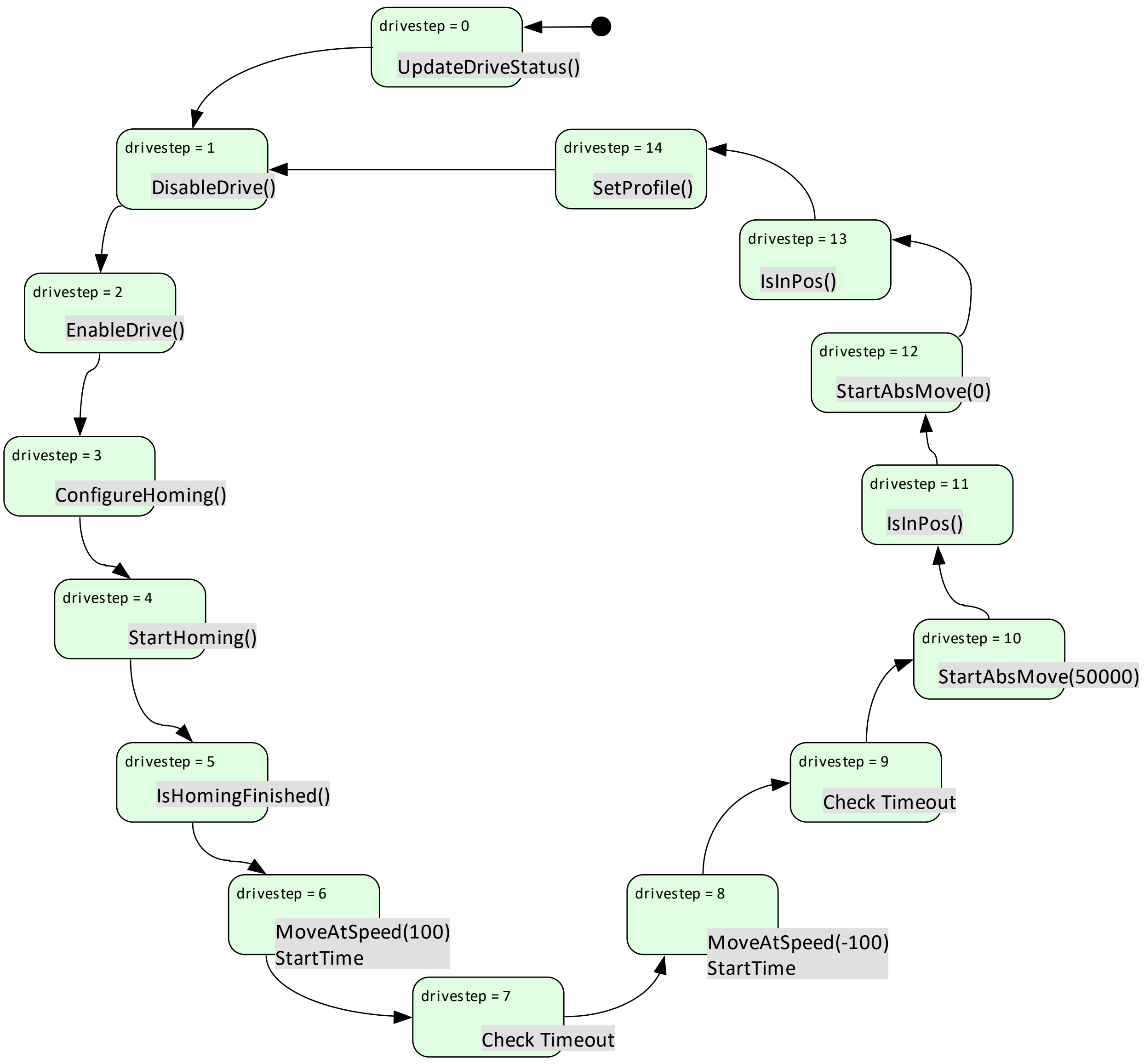


Figure 6 Step-Sequence of the single axis example

What’s the timer being used for?

Both examples – 1Node or 4Nodes use the MCTimer to execute a simple independent behavior – here toggling the built-in LED on pin 13 of the Arduino. Otherwise the timer is not used in the libraries. Using such an approach some kind of parallel processing or time-out for the main application could easily be added.

Rev C - not using a reference to a timer library – pure Arduino feeling

Use of a hardware timer is difficult in an Arduino environment as there meanwhile is a couple of different platforms having different µControllers and therefore different hardware capabilities. Therefore, if you intend to use the MC V3.0 RS232 lib "as is" in an Arduino environment it might be better not to use the version Rev A where a timer is referenced. Within the code archive there is a second version of the libraries which do longer not refer to the timer.

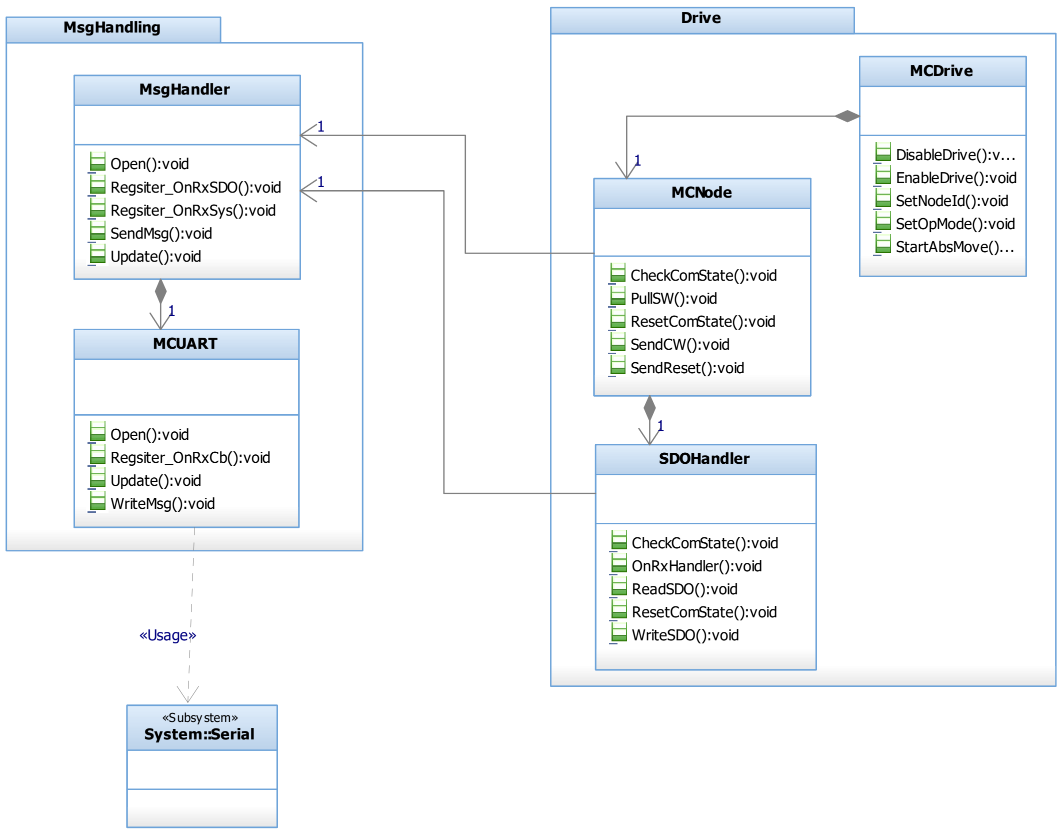


Figure 7 Class diagram of the embedded RS232 library Rev C

Rev C adds a few useful methods to MCDrive.h which are:

```
DriveCommState WriteObject(uint16_t, uint8_t, int32_t, uint8_t);
```

Write a value to an object of the MC V3.0. The parameters are the index, the sub index, the actual value which must casted to an int32_t to be universal and the length of the payload in bytes.
Returns DriveCommState == eMCDone when the parameter has been written successfully.

```
DriveCommState UpdateActValues();
```

Updates to local copy of the drives actual speed and position value.
Returns DriveCommState == eMCDone when the parameter have been updated successfully.

```
int32_t GetActualPosition();
```

Returns the value of the local copy of the drives actual position. UpdateActValues() should be called before the access is used.

```
int32_t GetActualSpeed();
```

Returns the value of the local copy of the drives actual position. UpdateActValues() should be called before the access is used.

```
DriveCommState UpdateMotorTemp();
```

Updates to local copy of the drives estimated motor temperature.
Returns DriveCommState == eMCDone when the parameter has been updated successfully.

```
int32_t GetActualMotorTemp();
```

Returns the value of the local copy of the motor temperature. UpdateMotorTemp() should be called before this access is used.

```
DriveCommSate UpdateDriveErrors();
```

Updates to local copy of the errors reported by the drive in 0x2320.00.
Returns DriveCommState == eMCDone when the parameter has been updated successfully.

```
int32_t GetActualDriveErrors();
```

Returns the value of the local copy of the drive error. UpdateDriveErrors() should be called before this access is used.

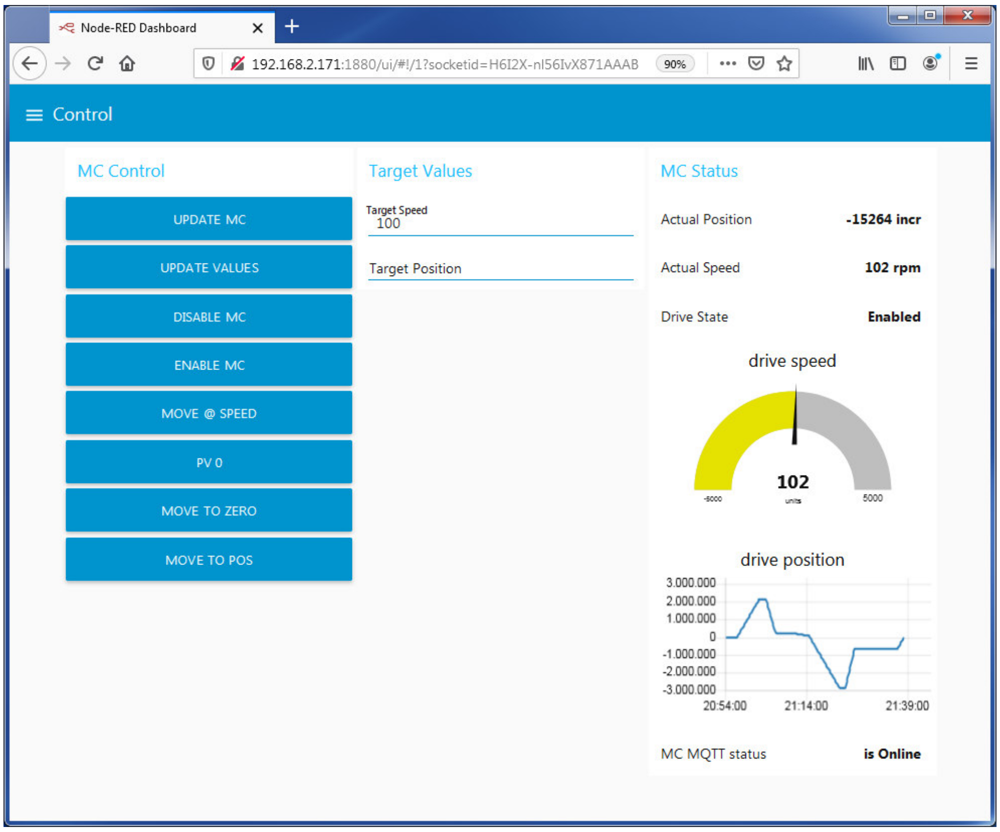
Single Axis example

Connect2MC_1Node is the same simple example as with Rev A but here without any reference to the timer..

4axis example

Connct2MC_4Nodes is doing the very same as the single axis example but here using 4 nodes. The 4 MCs have to be connected to the Arduino according to Figure 1 and have to be preconfigured for the same RS232 baud-rate, different node-Ids and RS232 net-mode enabled.
Within the sketch there are now for instances of the MCDrive. To clean the code a sperate method and code structure is used to organize e.g. different values for the changing profile parameters of the 4 nodes.

Using Rev C on a Nano 33 IoT



Target Speed 100

Target Position

Actual Position	-15264 incr
Actual Speed	102 rpm
Drive State	Enabled
MC MQTT status	is Online

Figure 8 Operating Panel of the MQTT example by NodeRed

Rev C has been successfully tested on an Arduino Nano every which is based on a ATMega4809 as well as on an Arduino Nano 33 IoT which is based on an Arm® Cortex®-M0 32-bit SAMD21.

Using the Nano 33 IoT an example was created which wraps a MQTT server around a MC V3.0.
The original code for the MQTT server was taken out of the examples for the Arduino platform.
The Nano 33 IoT wraps around a SoC Wi-Fi module, the u-blox NINA-W102.
To use it add the WiFiNINA library.
The PubSubLibrary is used to implement the MQTT protocol.
A few samples of the implementation are included here to directly explain the idea. The complete example is included in the software archive.

The main idea is the Arduino registering at the MQTT broker to receive commands for the drive via the topic `"Nano/MC/Ctrl/Control`" and new target values via either `"Nano/MC/Ctrl/TargetPos"` or `"Nano/MC/Ctrl/TargetSpeed"`.
Vice versa it's going to update its latest values via the pubTopicXxx in Code 7.

The callback in Code 8 must be registered at the MQTT client and will be called whenever one of the subscribed topics is received. Here the commands and parameters are extracted from the text-based payloads which are then checked in **updateDriveComm()** in Code 9.

Within **updateDriveComm()** whenever a new command has been received via `"Nano/MC/Ctrl/Control"` and the drive communication is in idle state the latest command is set to be executed next via the switch-case statement.
To actually drive the communication updateDriveComm() has to be called cyclically within the loop() as well as the loop() of the mqtt client - Code 12.

After each command the communication handler returns to the idle state where a next command could be processed.
If an update of actual values was requested these are published directly to the related topics before returning to the idle state.
When in idle state the actual speed and position of the drive are updated cyclically too by automatically switching to the handler of an externally triggered update request.

The actual code for initialization and updating the states is summarized in Code 12.

The frontend then was designed using NodeRed (Figure 9). The blue boxes are taken out of the dashboard library and are visible on the panel in Figure 8. The raspberry colored ones are the ones connecting to the MQTT server either by subscription (left) or publishing (right).

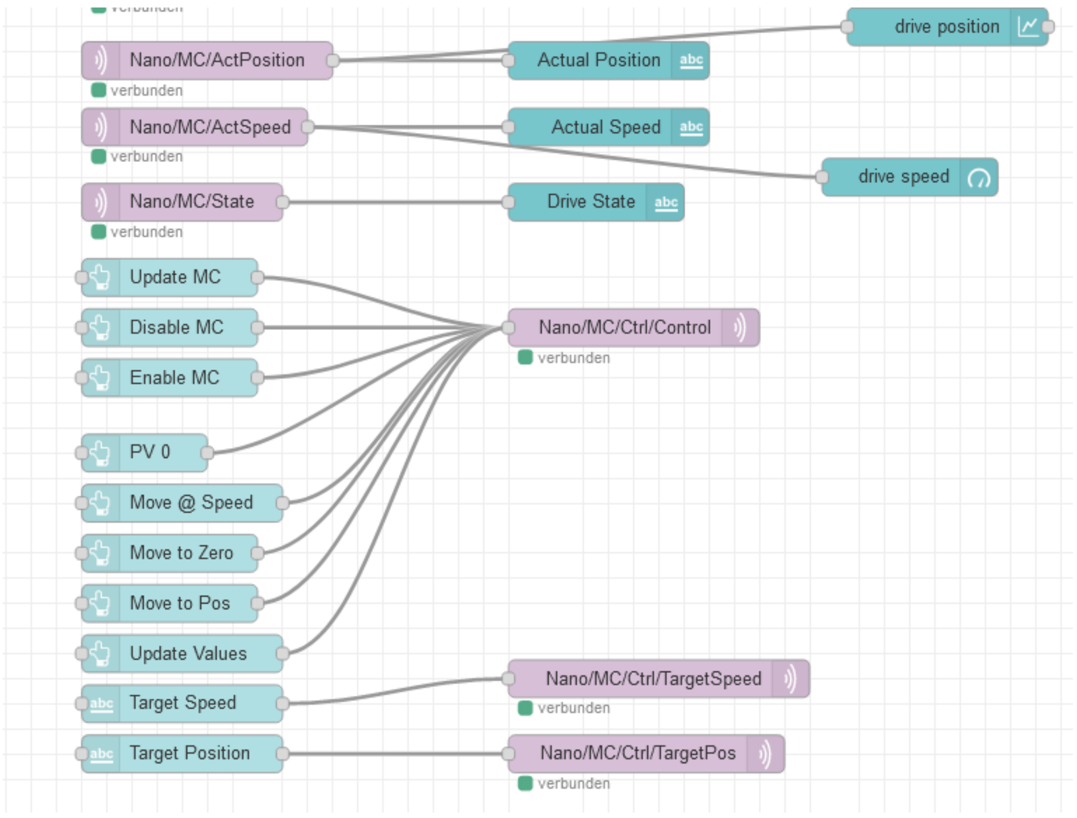


Figure 9 Flow of the monitoring panel

```
//--- defines ---
#define LED_PIN   LED_BUILTIN

//--- includes ---
#include <WiFiNINA.h>
#include <PubSubClient.h>
#include "WiFiAccess.h"
#include <MsgHandler.h>
#include <MCDrive.h>
#include <stdint.h>

//--- globals ---
//--- 4 WiFi / MQTT ----

const char* ssid = networkSSID;
const char* password = networkPASSWORD;

const char* mqttServer = mqttSERVER;
const char* mqttUsername = mqttUSERNAME;
const char* mqttPassword = mqttPASSWORD;

//subscribe to all /MC/* topics
char MCControlSubTopicAll[] = "Nano/MC/Ctrl/+";

//payload[0] will be the requested state
char MCControlSubTopic[] =    "Nano/MC/Ctrl/Control";
//payload[] is a numeric value of speed or position
char MCControlTargetSpeed[] = "Nano/MC/Ctrl/TargetSpeed";
char MCControlTargetPos[] =   "Nano/MC/Ctrl/TargetPos";

//payload will be "EN" or "DI"
char MCStatePubTopic[] =      "Nano/MC/State";
//payload are the current values
char pubTopicDriveError[] =   "Nano/MC/DriveErrors";
char pubTopicPosition[] =     "Nano/MC/ActPosition";
char pubTopicSpeed[] =        "Nano/MC/ActSpeed";
char pubTopicMotorTemp[] =    "Nano/MC/MotorTemp";

WiFiClient wifiClient;
PubSubClient mqttClient(wifiClient);
```

Code 7 Header of the MQTT example

```
void callback(char* topic, byte* payload, unsigned int length)
{
  Serial.print("Message arrived [");
  Serial.print(topic);
  Serial.print("] ");
  for (int i = 0; i < length; i++)
  {
    Serial.print((char)payload[i]);
  }
  Serial.println();

  if(identifyTopic(topic,subTopic))
  {
    // Switch on the LED if 1 was received as first character
    if ((char)payload[0] == '1')
    {
      digitalWrite(LED_PIN, HIGH);
      ledState = 1;
      char payLoad[1];
      itoa(ledState, payLoad, 10);
      mqttClient.publish(pubTopic, payLoad,true);
    }
    else
    {
      digitalWrite(LED_PIN, LOW);
      ledState = 0;
      char payLoad[1];
      itoa(ledState, payLoad, 10);
      mqttClient.publish(pubTopic, payLoad,true);
    }
  }
  if(identifyTopic(topic,MCControlSubTopic))
  {
    requestedDriveStep = (uint16_t)extractValue(payload,length);
  }
  if(identifyTopic(topic,MCControlTargetSpeed))
  {
    TargetSpeed = extractValue(payload,length);
  }
  if(identifyTopic(topic,MCControlTargetPos))
  {
    TargetPos = extractValue(payload,length);
  }
}
```

Code 8 The callback extracting the parameters out of the incoming messages

```
void updateDriveComm()
{
static bool isAutoUpdate;
static uint32_t lastAutoUpdate = 0;
uint32_t currentMillis = millis();
DriveCommStates NodeState = Drive_A.CheckComState();

   Drive_A.SetActTime(currentMillis);
   MCMsgHandler.Update(currentMillis);

   if((NodeState == eMCError) || (NodeState == eMCTimeout))
   {
      //Serial.println("Main: Reset NodeA State");
      Drive_A.ResetComState();
      //should be avoided in the end
      actDriveStep = 0;
   }

   switch(actDriveStep)
   {
      case 0:
        //do nothing - that's the idle state
        if(requestedDriveStep > 0)
        {
          //start handling of the request
          actDriveStep = requestedDriveStep;
          //reset request again
          requestedDriveStep = 0;
          isAutoUpdate = false;
          Serial.print("Start ");
          Serial.println(actDriveStep);
        }
        else
        {
          //no request pending
          //check for timeout of auto-update
          if(lastAutoUpdate + updateRate < currentMillis)
          {
            actDriveStep = 20;
            lastAutoUpdate = currentMillis;
            isAutoUpdate = true;
          }
        }
        break;
```

Code 9 Handling the commands to the drive via the drive library / part A

```
      case 1:
        //get a copy of the drive status
        if((Drive_A.UpdateDriveStatus()) == eMCDone)
        {
          //switch back to idle state
          actDriveStep = 0;
          Drive_A.ResetComState();
          Serial.println("Main: Status updated");
        }
        break;
      case 2:
        //…
```

Code 10 Handling the commands to the drive via the drive library / part B

```
void setup_drive()
{
  MCMsgHandler.Open(115200);
  Drive_A.SetNodeId(DriveIdA);
  Drive_A.Connect2MsgHandler(&MCMsgHandler);
}
```

Code 11 configure the drive libary once

```
void setup()
{
  pinMode(LED_PIN, OUTPUT);
  Serial.begin(500000);
  /*
  while(!Serial)
    ; */
  setup_wifi();
  mqttClient.setServer(mqttServer, 1883);
  mqttClient.setCallback(callback);

  setup_drive();
}

void loop()
{
  if (!mqttClient.connected())
  {
    reconnect();
  }
  mqttClient.loop();

  updateDriveComm();
}
```

Code 12 The setup() and loop() of the example

Customization of the Library

On a bare metal µController the lowest layer – here MCUart.cpp could be a little different.

Any implementing the Uart in such an environment might could have:

UART ISR
An interrupt service routine to handle the Transmit Buffer Empty (TXE) interrupt, the Data Received (RXNE) interrupt and any Error interrupt. This will be static C function not taking any parameters as the ISR is not getting any.

A class which implements the same behavior as the MCUart described here but not being updated by polling but using the TXE to byte by byte send messages until the complete message is sent and the RXNE to actually fetch the data from the UART data register.
TXE will only be active, when there is a message to be sent RXNE will always listen.

The low level ISR will then have to call one of the methods out of the Uart class to actually handle the TX or RX. Code 14 is an example implementation using C and a C-struct to implement OO like handling.

Here `O_Usart2Isr` in Code 13 is a C struct which contains data being used to create an abstract access to the Usarts of the used STM32 device. `O_Usart2Isr.itsC_Usart` then is a pointer to the `struct C_Usart` C struct in Code 15 which is a class like collection of data for a single instance of the `C_Usart`.

```
// Provides the ISR routine needed for UART2
struct O_Usart2Isr_t {
   //
   struct C_Usart itsC_Usart;          /*## link itsC_Usart */
};
```

Code 13 C-data structure O_Usart2Isr to handle the low-level interrupt

`C_Usart.c` then implements methods which are applied to the instance.

The pointer to the actual instance of the `C_Usart` which is associated with the actual peripheral (here USART2) is stored in a static instance `O_Usart2Isr`. Its actual ISR handler `USART2_IRQHandler` (Code 14) – a name defined by the ST environment – uses this pointer to not only call the class-aware handler **`C_Usart_IrqHander()`** (Code 16) but also to pass this pointer.

```
//low level ISR to call the actual handler within the Uart class
void USART2_IRQHandler( void ) {

   // call according C_Usart instance handler
   C_Usart_IrqHandler(&(O_Usart2Isr.itsC_Usart));
}
```

Code 14 Low level ISR to call the handler within Uart.c

```
struct C_Usart {
   // Containes the base address of this USART
   USART_TypeDef* pSTM32Reg;

   // Receive buffer of USART
   Pk_Usart_TSerialMsg rxBuf;
   // Index in receive buffer, where next received character will be 
stored
   uint8_t rxIdx;
   // Number of expected characters for a complete message
   uint8_t rxSize;

   // Transmitt buffer of USART
   Pk_Usart_TSerialMsg txBuf;
   // Index in transmit buffer, from where next character will be trans-
mitted
   volatile uint8_t txIdx;
   // Number of characters, which will be transferred
   volatile uint8_t txSize;

   // Containes the internal node number of USART.
   // Msg will only transfered to C_RS232 if received node number is 
equal to internal node number
   uint8_t nodeID;

   // Holds ObjPtr and FctPtr of message receive callback function
   Pk_HAL_TFunctor msgRcvCb;
   // Holds ObjPtr and FctPtr of messagetransmitted callback function
   Pk_HAL_TFunctor msgTrmCb;

   // threshold for any Rx time-out
   uint8_t thReceiveTo;
   // threshold for receiving a complete message
   uint32_t receiveToValue;

   // handle the actual state of any message to be transmitted
```

```
    volatile C_Usart_TTxState txState;
    // link to the instance of the GpTimer to be used for the time-out
    struct C_GpProgTimer* itsC_GpProgTimer;
};
```

Code 15 Example C-struct to implement a OO like implementation of a class C_Usart.

Main elements within the struct `C_Usart` in Code 15 are the Rx and Tx buffers and their read and write pointers, an element used to track the status of any ongoing transmission and the call-backs to higher layers when either an ongoing transmission is completed or a compete message has been received.

The actual handling of any Usart interrupt is done within **`C_Usart_IrqHandler`** a method within C_Usart.c. It implements a OO like handling within C. Every method therefore needs at least a point to the actual instance of `C_Uart` – the one to be used.
The ISR handler reacts to either:

- A character has been received.
  It's passed over to `RcvdChar2Buffer(me, rcvChar)` which implements more or less the identical treatment like the Update in the MCUart.cpp Arduino lib. Here however, as the implementation does not use polling a real timer resource has to be used to handle the time-outs.
- TX buffer is empty – a next character can be sent
  If there still is a character in the Tx buffer it is passed to the Uart, otherwise the transmission will wait for the last character to actually been sent.
- Last Character has been sent
  The ongoing Tx state if idle again and the next layer of software is informed using the call-back. Higher layer can use this information to pass a next message if there is a buffer of messages to be sent.

```
// the actual class aware ISR handler of C_Usart.c
void C_Usart_IrqHandler(C_Usart* const me) {
    uint8_t rcvChar;
    uint32_t statusReg = me->pSTM32Reg->SR;

    if ((statusReg & USART_SR_RXNE) != 0)
    {
        // data received,
        // read data register, clears error flags at the same time
        rcvChar = me->pSTM32Reg->DR;
        if ((statusReg &
                (USART_SR_PE             // parity error
                    | USART_SR_FE        // framing error
                    | USART_SR_NE)       // noise detected
                ) == 0)
        {
            // no errors detected, accept the received character
            RcvdChar2Buffer(me, rcvChar);
        }
    }

    if (((statusReg & USART_SR_TXE) != 0) && (me->txState == eTxBusy))
    {
        // Transmit data register empty
        if (me->txIdx < me->txSize)
        {
            // TXE/TC is cleared on write to data register
            me->pSTM32Reg->DR = me->txBuf.u8Data[me->txIdx++];
        }
        else if (me->txIdx == me->txSize)
        {
            // all Tx bytes processed
            // disable TXE interrupts
            me->pSTM32Reg->CR1 &= ~USART_CR1_TXEIE;
            // enable Transmission complete IR
            me->pSTM32Reg->CR1 |= USART_CR1_TCIE;
            me->txState = eTxWaiting4TC;
        }
    }

    if (((me->pSTM32Reg->CR1 & USART_CR1_TCIE) != 0)
        && ((statusReg & USART_SR_TC) != 0))
    {
        // Last byte is transmitted to Line by UART
        // HW (Transmit FIFO empty)
        // -> mask TC IR again
        me->pSTM32Reg->CR1 &= ~USART_CR1_TCIE;

        // reset state to indicate no transmit ongoing
        // me->txBusy = false;
        me->txState = eTxIdle;

        // call message transmitted handler
        if (me->msgTrmCb.FctPtr != NULL)
        {
            me->msgTrmCb.FctPtr(me->msgTrmCb.ObjPtr);
        }
    }
}
```

Code 16 the actual handler within C_Usart.c

The last example here is the C_Usart method to be called when a next message is to be transferred. Here the complete message is copied into the Tx-buffer. So, any preceding transmission using this buffer must have been finished and Tx-state has to be idle – something the higher layer will have to check before calling **`C_Usart_WriteMsg()`**. The actual transmission is started by enabling the Tx interrupt which will then detect an empty Tx data register and call the **`C_Usart_IrqHandler()`**.

```
// handler to actually copy messages to be transmitted into the Tx buffer
and start the Tx interrupt
void  C_Usart_WriteMsg(C_Usart*  const  me,  const  Pk_Usart_TSerialMsg*
pSerialMsg) {

   if (me->pSTM32Reg != NULL)
   {

      // copy raw data to txbuf of C_Usart
      memcpy(&me->txBuf.u8Data[0],
              &pSerialMsg->u8Data[0],
              pSerialMsg->Hdr.u8Len + 1);

       // add prefix and suffix chars
      me->txBuf.u8Data[0] = PREFIX;
      me->txBuf.u8Data[pSerialMsg->Hdr.u8Len + 1] = SUFFIX;

      // set number of chars to be transmitted
      // this will be used during the actual transmission
      me->txSize = pSerialMsg->Hdr.u8Len + 2;
      // reset txIdx
      me->txIdx  = 0;

      //now flag the Tx to be busy
      me->txState = eTxBusy;

      // Used interrupts: TXEIE and RXNEIE (always set)
      //  TE  transmits  idle  frame  for  1  bit  time  and  generates  first
interrrupt
      me->pSTM32Reg->CR1 |= (USART_CR1_TXEIE | USART_CR1_TE);
      // interrupt to send out first character
   }
}
```

Code 17 Write-handler of C_Usart.c

The interaction of the different classes and methods in such an interrupt-based message handling scheme is illustrated in Figure 10. Any received character activates the bare-metal interrupt handler of the Usart. It calls the class-aware IRQHandler of the `C_Usart` which adds the received char to its Rx buffer via `RcvdChar2Buffer()`.

If the message was completely received the registered callback – here the `C_MsgHandler_OnRx-Handler()` is called which could store the message and create an event for the next layer. Such an event could be used to finally end the interrupt execution context. If no events available use a flag which C_SDOHandler polls and reads the message to handle it out of the interrupt.

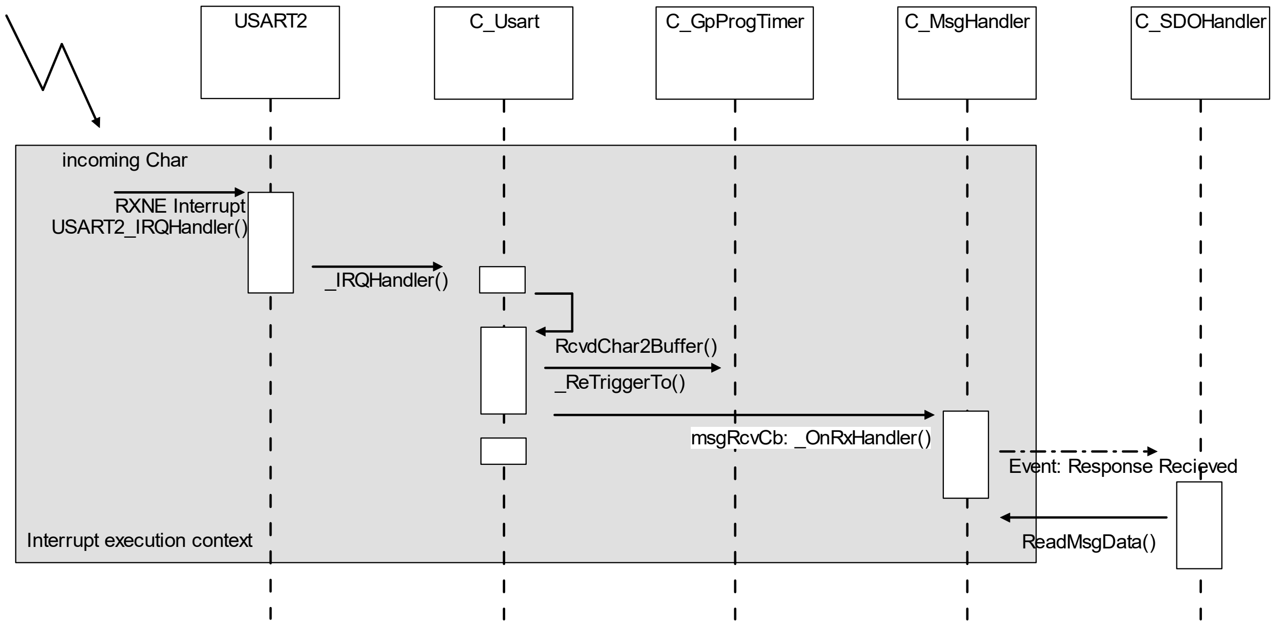


Figure 10 Sequence diagram of an interrupt based message handling

References

Arduino SerialEvent
https://www.arduino.cc/en/Tutorial/BuiltInExam-ples/SerialEvent

FAULHABER manuals
https://www.faulhaber.com/en/support/technical-sup-port/drive-electronics/documentation-for-drive-electronics/

Arduino nano Every
https://store.arduino.cc/arduino-nano-every

Arduino nano 33 IoT
https://store.arduino.cc/arduino-nano-33-iot