

Control MC V3.0 MotionController via RS232

An Arduino Library

Summary

This ApplicationNote is intended to show how a connection from an embedded μ Controller to any FAULHABER MC V3.0 via RS232 can be established. An example library has been developed and tested using different Arduino boards. The revision provided here is revision D of the library.

The Arduino implementation is meant to be an example how such a communication stack could be implemented on a μ Controller. 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

Environment

Rev D of the library has been successfully tested with or without a MQTT connection on:

- an Arduino Nano every
based on a ATmega4809 + W5500 SPI to Ethernet controller
- an Arduino Nano 33 IoT
based on an Arm[®] Cortex[®]-M0 SAMD21 + u-blox NINA-W102
- an Arduino Nano RP2040 Connect
based on the dual Arm[®] Cortex[®]-M0+ RP2040 + u-blox NINA-W102
- an Arduino R4 wifi
based on an Renesas Arm[®] Cortex[®]-M4 RA4M1 + ESP 32 wifi

General Setup

The used setup has one Arduino (nano every, nano IoT, nano RP2040 connect or R4 wifi) and 1 ... 4 MC V 3.0 controlled by the code examples.

The CMOS logic levels of the Arduinos serial1 Rx/Tx ports are converted to RS232 levels using a RS232 transceiver.

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 central device and the drive nodes.

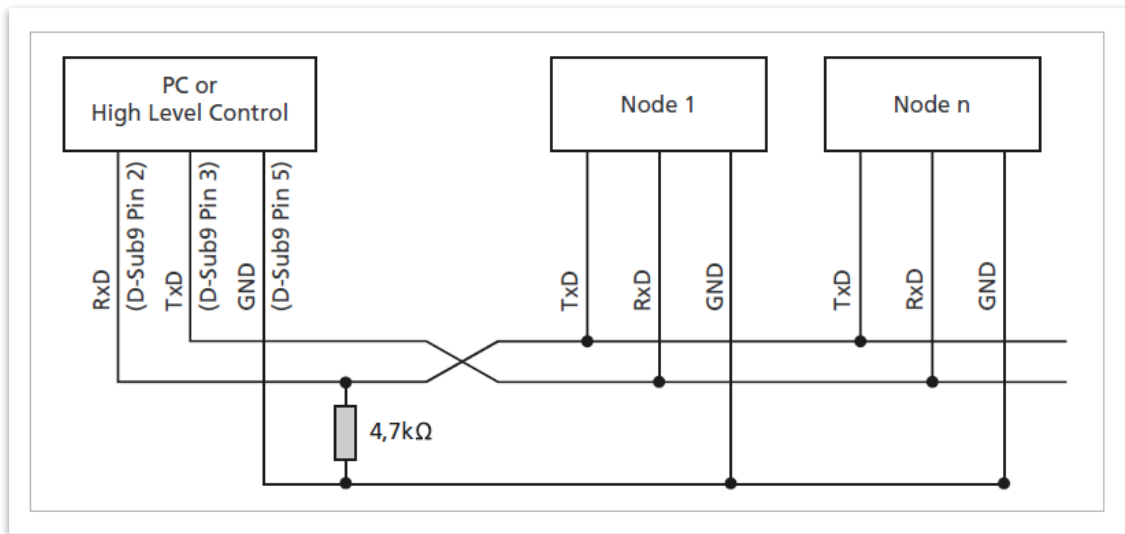


Figure 1 Connection between a RS232 host and 1 ... n drive nodes



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.

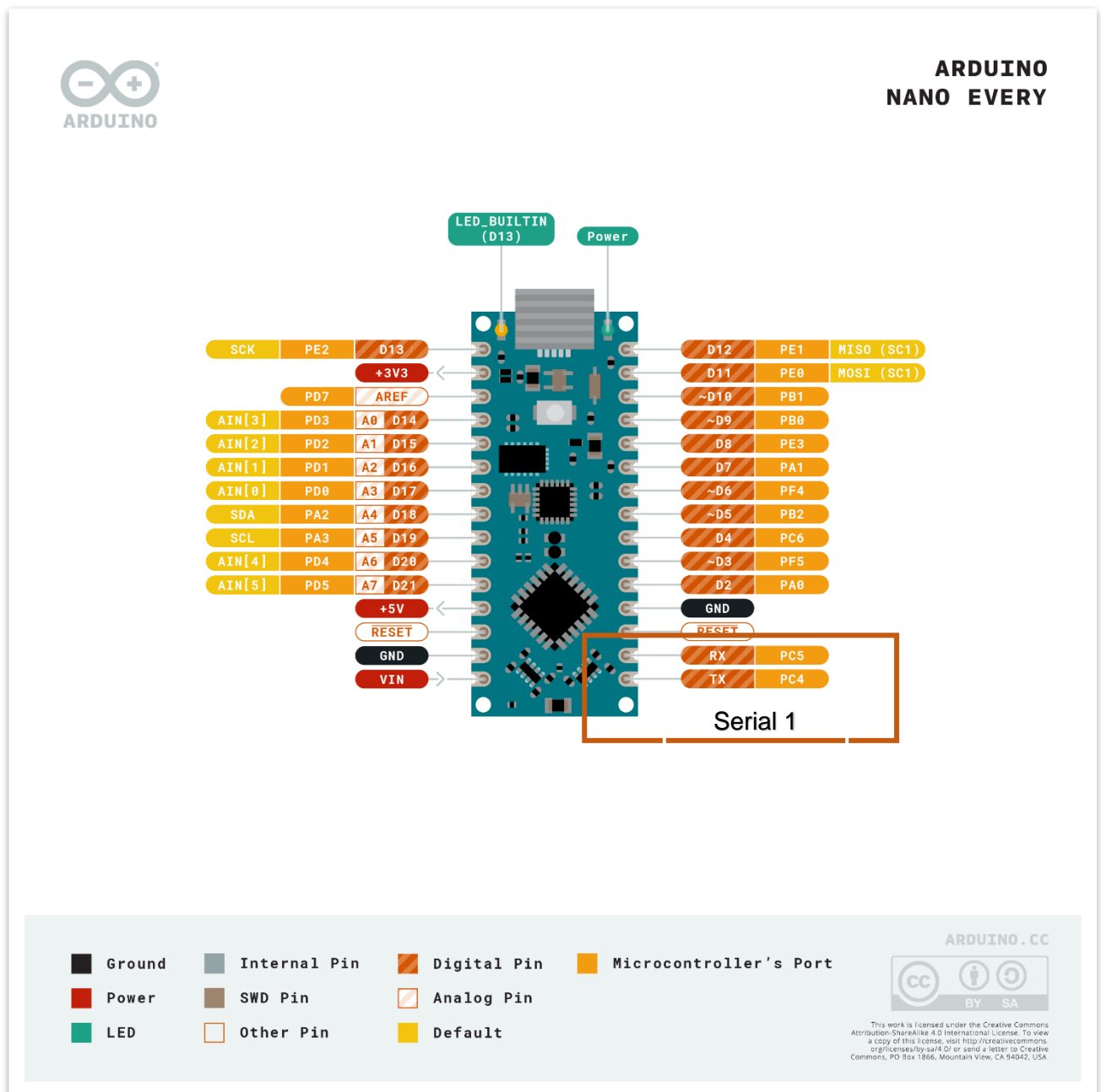


Figure 2 Pinout of an Arduino nano every¹ - one of the reference systems here

¹ <https://store.arduino.cc/arduino-nano-every>

Arduino Example Environment

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

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

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(115200);  
    ...  
}  
  
void loop() {  
    // put your main code here, to run repeatedly:  
    ...  
}
```

Code 1 Basic structure of an Arduino program

Libraries like the one used here are referenced 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.

Within the MC V3.0 Arduino RS232 library there are no blocking calls, all methods check their status and return immediately. Loop() can therefore be kept short which is useful as there is no interrupt based handling of receive and transmit of messages implemented beyond the capabilities of the Serial library.

MCUart.update() uses polling of the Serial1 instead – see Code 2.

```
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	Connection to the development PC
Serial1²	Used in MCUart.cpp to communicate with the MCs

² On different boards different Serial resources 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 <code>Serial1.write()</code> . Receive characters using <code>Serial1.read()</code> 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 of 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.
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	Instances a MCNode and uses its SDOHandler 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.

Table 3 Test components used for better readability

component	description
MCTestCycle	Implements the test-cycle of Figure 5 for demo purposes using the library in Table 2.
MCMQTTBroker	Provides a wrapper around the PubSub library for better readability
MCRemoteControlled	Implements the services of a single instance of a node to connect to a MC V3.0 via the library in Table 2 and registered at a MQTT broker.

The relations between the software components are detailed in Figure 3. 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.

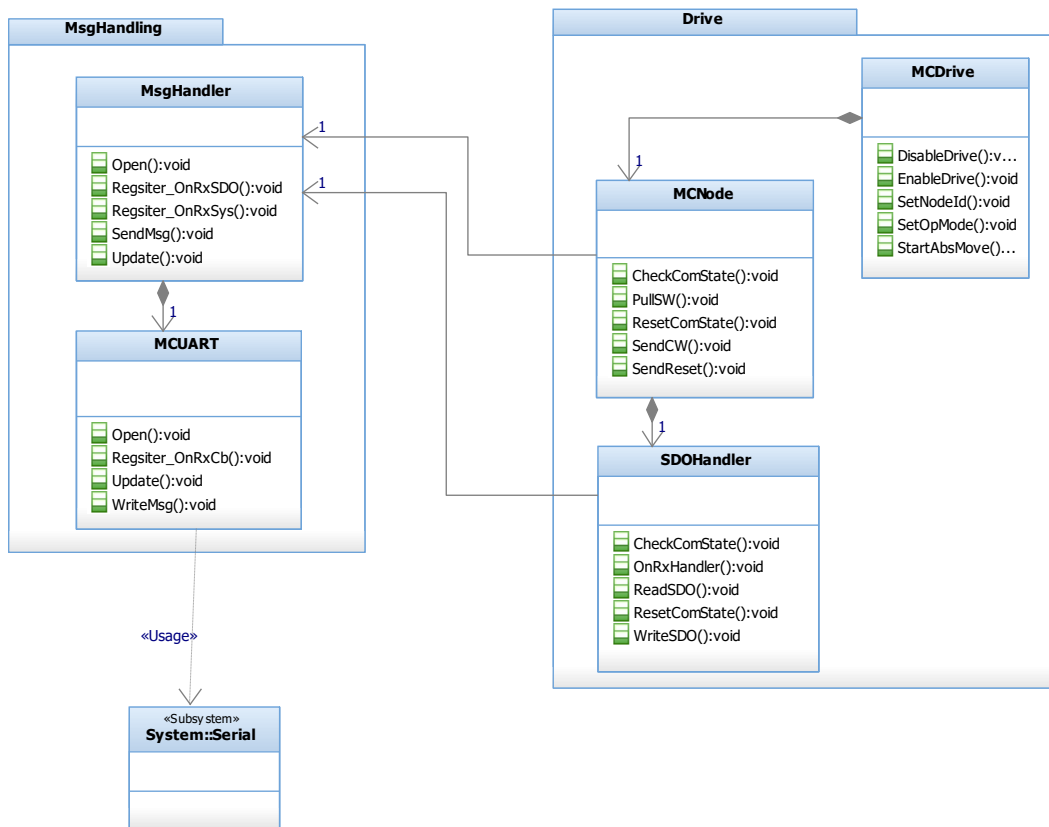


Figure 3 Class diagram of the embedded RS232 library Rev D

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 MsgHandler.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 <MsgHandler.h>
#include <MCDrive.h>
#include <stdint.h>

//--- globals ---

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.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 4 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 messages is implemented within MCNode.cpp using the additional services defined in Table 5.

Each MCDrive.cpp includes a single instance of a MCNode and its SDOHandler by means of composition (Figure 1). Therefore they don't have to be explicitly taken care of in any application. Creating an instance of the MCDrive will include the service – see Code 3.

Table 5 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 Parameter read- or write request the methods SDOHandler::ReadSDO() and SDOHandler::WriteSDO() must be called cyclically until they end up in the eSDODone state.

Internally they implement a step-sequence where the real request is sent out only if SDOHandler is in eSDOIdle state and ends up in eSDODone-state if successful or eSDOError-state if not (Figure 4).

After the request has been sent successfully the time when it has been sent is stored and used to check for any time-out condition. In case of the drive node not reacting to a request the SDO service will automatically enter a eSDORetry state and try again. Ultimately it will end up an error state (eSDOError) after a configurable number of retries.

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 (see Code 5).

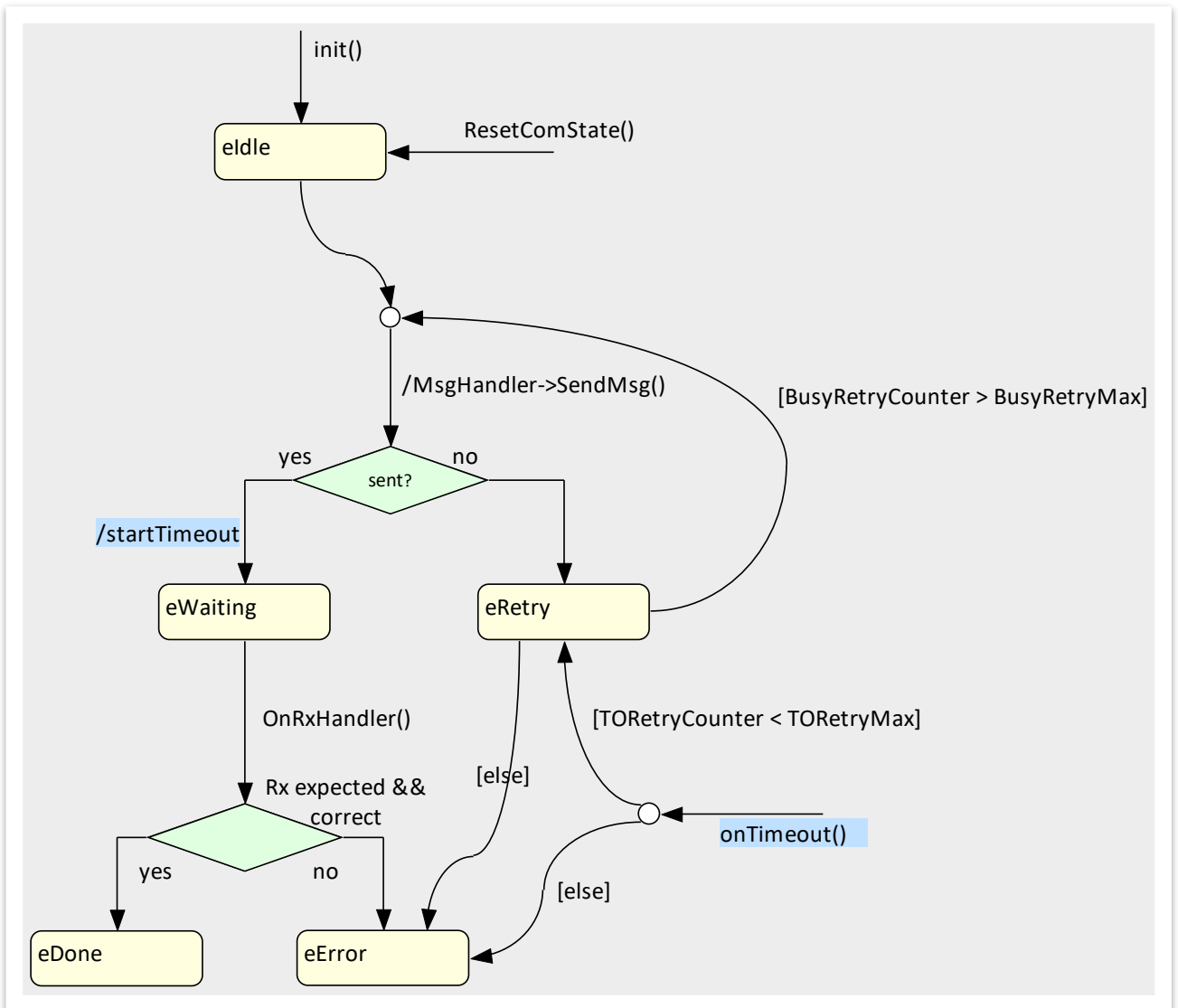


Figure 4 Step-Sequence of SDO access

Using these services has to cyclically call the `SDOHandler::ReadSDO()` or `SDOHandler::WriteSDO()` until they finally end up in the `eMCDone` 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)
{
    OpModeRequested = OpMode;

    if((MCDriveRxTxState =
        WriteObject(0x6060, 0x00, (uint8_t)OpModeRequested)) == eMCDone)
    {
        OpModeReported = OpModeRequested;
    }

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

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

The `MCDrive::SetOpMode()` method uses one of the additional generic methods `MCDrive::WriteObject()` available for either `uint8_t`, `uint16_t` or `uint32_t` parameters. `MCDrive::ReadObject()` is available for the same parameter sizes.

To have this work the `SDOHandler` component should ideally be in `eSDOIdle` state when `MCDrive::SetOpMode(int8_t OpMode)` is called the first time which it is, when the previous access either finished successfully or has been reset from an `eSDOError` state. The state of the component and 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;
            }
            break;
        case 7:
            ...
    }

    NodeState = Drive_A.CheckComState();
    if((NodeState == eMCErrror) || (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 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. A busy interface can occur when multiple instances of MCDrive interact with a network of drive nodes.

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 5.

```
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 4.

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 eMCErrror or eMCTimeout.

```
bool IsLive();
```

If a single drive is connected only and not configured for net-mode operation it will send its boot-message after power-up. 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 of 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.

Drive parameter access

Drive parameter read and write access is provided on the MCDrive level by following calls:

```
DriveCommStates ReadObject(uint16_t, uint8_t, uint8_t *);  
DriveCommStates ReadObject(uint16_t, uint8_t, uint16_t *);  
DriveCommStates ReadObject(uint16_t, uint8_t, uint32_t *);
```

Is to be called cyclically to check for any time-outs. Parameters are

- the object index of the parameter to be read,
- the sub-index and
- a pointer to the variable which shall hold the received value. Signed integers have to be casted to unsigned ones.

Returns DriveCommState == **eMCDone** when finished.

```
DriveCommStates WriteObject(uint16_t, uint8_t, uint8_t);  
DriveCommStates WriteObject(uint16_t, uint8_t, uint16_t);  
DriveCommStates WriteObject(uint16_t, uint8_t, uint32_t);
```

Is to be called cyclically to check for any time-outs. Parameters are

- the object index of the parameter to be read,
- the sub-index and
- the value of the parameter to be written. Signed integers have to be casted to unsigned ones.

Returns DriveCommState == **eMCDone** when finished.

```
DriveCommStates DownloadParameterList(MCDriveParameter *, uint8_t);
```

Can be used to simplify the download of a couple of parameters by collecting them in vector. Uses:

```
typedef struct MCDriveParameter {  
    uint16_t index;  
    uint8_t subIndex;  
    uint32_t value;  
    uint8_t length;  
} MCDriveParameter;
```

Plus the number of parameters to be downloaded. Length refers to the size of the parameter to be accessed.

```
DriveCommStates UploadParameterList(MCDriveParameter *, uint8_t);
```

Can be used to simplify the download of a couple of parameters by collecting them in vector. Uses the same definition for the MCDriveParameter vector.

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 4.

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.

```
int8_t GetOpMode();
```

Returns the local copy of the OpMode requested. 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 (0x6083.00)
- Profile Deceleration (0x6084.00)
- Profile Velocity (0x6081.00)
- Motion Profile Type (0x6086.00)

Could also be handled via the `MCDrive::DownloadParameterList()`.

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 DoHoming();
```

Switches the drive to homing mode and starts whatever homing method has been configured.

Returns DriveCommState == eMCDone when the homing was successful. Internally uses the MCDrive::IsHomingFinished() method.

Returns DriveCommState == eMCDone when the Homing was successful.

```
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 successfully.

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 under the hood of the MCDrive.

```
CWCommStates GetNodeState();
```

Returns the DriveCommState of the MCDrive instance and updates it by checking the SDOState.

```
SDOCommStates GetSDOState();
```

Returns the SDOCommState of the built-in SDOHandler.

```
CWCommStates GetCWAccess();
```

Returns the CWCommState of the built-in MCNode without updating it from the SDOState.

```
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
```

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	<p><code>MaxMsgTime = 3 (ms)</code> only reasonable when 115200 Baud ist used <code>MsgTimeout = MaxMsgTime</code> 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.</p>
MsgHandler.cpp	<p><code>MsgHandlerMaxLeaseTime = 2 x MaxMsgTime + 2</code> 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.</p>
SDOHandler.cpp	<p><code>SDORespTimeOut = 4 x MaxMsgTime</code> Maximum waiting time for a response to either a SDO read request or a write request. After time-out a retry is started. After too many retries the access would end up in eTimeout state.</p>
MCNode.cpp	<p><code>CwRespTimeOut = 20</code> 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.</p> <p><code>MaxSWResponseDelay = 50</code> 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.</p>
MCDrive.cpp	<p><code>MaxSWResponseDelay = 50</code> Is the time used for calls to MCNode::SendCW() if an explicit response is expected.</p> <p><code>PullSWCycleTime = 20</code> Is the time used for cyclic polling the statusword e.g. when checking for a target being reached.</p>

Examples – Using the Library

MCTestCycle

A test-cycle show-casing many of the implemented methods has been encapsulated into a class MCTestCycle.

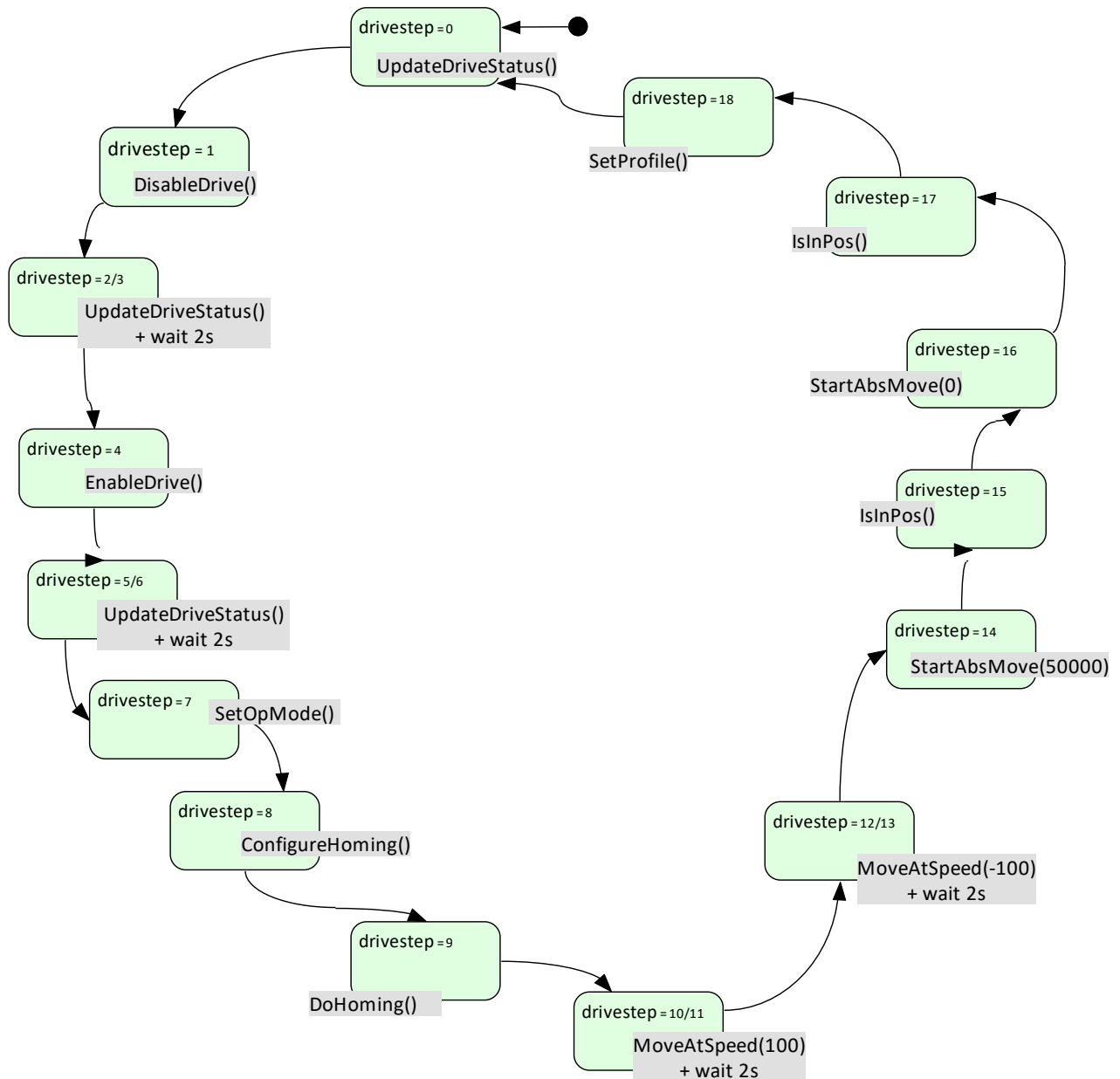


Figure 5 Step-Sequence of the single axis example

A few simple methods allow for a quick test.

```
MCTestCycle MCTestCycle(uint8_t, char *, int8_t);
```

Initializes an instance of the MCTestCyle. Parameters are the node-id to be used, a readable name and the default homing method. Can be used in the global segment to instantiate the 1 ... 4 supported drive nodes.

Example:

```
const int16_t DriveIdA = 1;
char DriveA_Name[] = "AHall";
int8_t HomingA = 33;
const int16_t DriveIdB = 2;
char DriveB_Name[] = "IE3";
int8_t HomingB = 33;
const int16_t DriveIdC = 3;
char DriveC_Name[] = "IER3";
int8_t HomingC = 37;
const int16_t DriveIdD = 4;
char DriveD_Name[] = "AES-L";
int8_t HomingD = 33;

MsgHandler MCMsgHandler;
MCTestCycle DriveA(DriveIdA, DriveA_Name, HomingA);
MCTestCycle DriveB(DriveIdB, DriveB_Name, HomingB);
MCTestCycle DriveC(DriveIdC, DriveC_Name, HomingC);
MCTestCycle DriveD(DriveIdD, DriveD_Name, HomingD);
```

Code 7 Globals of the test sketch w/o MQTT connection

```
void ConnectToMessageHandler (MsgHandler *);
```

Usually called in the `setup()` after the `MsgHandler` has been initialized.

Example:

```
setup()
{
    //start the Msg-Handler
    MCMsgHandler.Open(115200);

    //connect drives to Msg-Handler
    DriveA.ConnectToMsgHandler(&MCMsgHandler);
    DriveB.ConnectToMsgHandler(&MCMsgHandler);
    DriveC.ConnectToMsgHandler(&MCMsgHandler);
    DriveD.ConnectToMsgHandler(&MCMsgHandler);
}
```

```
uint8_t DoCycle(uint32_t);
```

The actual step function for the test-cycle. A non-blocking call. Parameter is the actual time in milliseconds. Returns the `stepCycle` – see Figure 5 Step-Sequence of the single axis example Figure 5.

Example:

```
loop()
{
    uint32_t currentMillis = millis();

    MCMsgHandler.Update(currentMillis);
    DriveA.DoCycle(currentMillis);
    DriveB.DoCycle(currentMillis);
    DriveC.DoCycle(currentMillis);
    DriveD.DoCycle(currentMillis);
}
```

```
void ResetComState();
```

Checks the communication state of the `MCDrive` and resets it when in error state.

Example

```
DriveA.ResetComState();
DriveB.ResetComState();
DriveC.ResetComState();
DriveD.ResetComState();
}
```


Using Rev D connected to MQTT

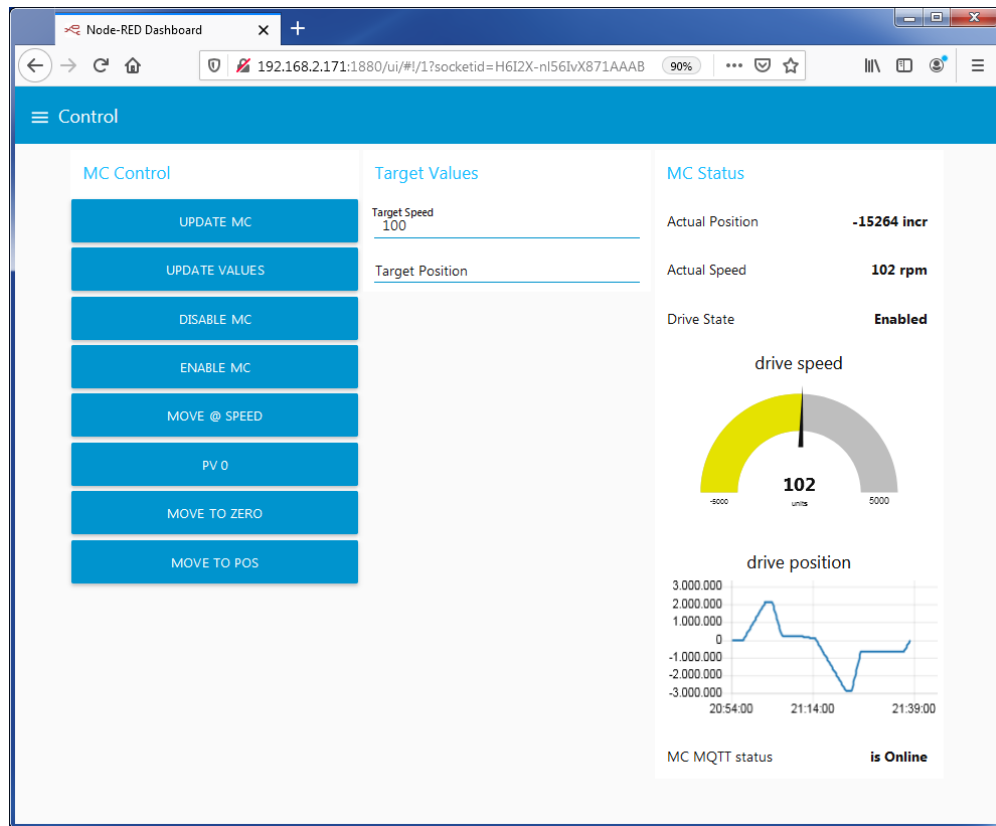


Figure 6 Operating Panel of the MQTT example by NodeRed

An additional 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.

Depending on the wifi module or ethernet module in use different libraries have to be included.

```
//#define MyBoardType WiFiTypeNone
//#define MyBoardType WiFiTypeNINA
#define MyBoardType WiFiTypeS3

#include <stdint.h>

#if (MyBoardType == WiFiTypeS3)
#include <WiFiS3.h>
#elif (MyBoardType == WiFiTypeNINA)
#include <WiFiNINA.h>
#elif (MyBoardType == WiFiTypeNone)
#include <SPI.h>
#include <Ethernet.h>
#endif
```

The PubSub-Library is used to implement the MQTT protocol.

MCMQTTBroker

A first module MCMQTTBroker.cpp encapsulates the access to the broker provided via the PubSub library and allows for several instances of a MCRemoteControlled class to register at the MQTT broker. Its interface is not meant to be an example of software engineering but tries to result in a simple main sketch for the demo system.

MQTT exchanges data based on messages which are identified by a string – the topic and a payload of the message which could here be a parameter value.

In this demo-implementation the topic name is constructed by:

ClientName/DriveName/SubTopic

Where

- ClientName identifies the Arduino at the broker,
- DriveName is a name which has to be assigned to the drive node(coded in the sketch) to be able to address the different drives – see Code 7
- SubTopic is a name for a parameter published or subscribed to which is right now hard coded in the example class – MCRemoteControlled.cpp

In the cases of a DriveName as in Figure 7 a topic could be:

MC4AxisDemo/IE3/TPos for the target position or
MC4AxisDemo/IE3/Command for the action to be executed

During the initialization (Figure 7) the topic name strings are constructed and stored in the instances of MCRemoteControlled.

During the setup() the ip name of the MQTT broker and a first global receive topic are registered with the local instance of the PubSub client. Additionally a name for the Arduino board to be used as a part of the topics is defined which helps to identify the board and the drive nodes in the message tree of a broker.

After this basic information is configured the drive nodes can already register their receive topics at the client library.

The connection to the broker is established within the loop only via the Reconnect call which also subscribes to all the topics out of the setup() phase.

Within the loop the connection is re-checked in each cycle. For each instance of the MCRemoteControlled the update method is called which checks the latest received commands and starts the movements using a local instance of MCDrive.

Cyclically some of the actual values are probed from the MotionControllers and published at the MQTT broker.

The main structures maintained here is the data of the broker to connect to and a list of registered topics the implementation will handle.

The API of the module is:

```
void MCMQTT_Init(MQTTBrokerData *);
```

A call to preset some internal variables.

```
void MCMQTT_SetClientName(MQTTBrokerData *, const char *);
```

Preset a name for the Arduino within the name space of topics at the MQTT broker.

```
void CMQTT_RegisterBroker(MQTTBrokerData *, PubSubClient *, const char *);
```

Takes a pointer to a structure where the broker data is to be stored, a pointer to the instance of the PubSubClient and the IP name of the broker and registers the broker name with the PubSubClient without actually connecting to it.

```
void MCMQTT_RegisterSubTopic(MQTTBrokerData *, MQTTSubTopic *);
```

Adds a topic to be subscribed to in the list of topics maintained globally in the MCMQTTBroker module. MQTTSubTopic is a pointer to a structure identifying a topic to be received. It contains its fully qualified name – the topic at the broker and provides a call-back to be called on reception of the topic. These call-backs are implemented as methods of MCRemoteControlled as an example.

```
void MCMQTT_Reconnect(MQTTBrokerData *);
```

Called in the main loop to check the connection and if necessary establishes or re-establishes a connection to the MQTT broker. Will register all the topics to be listened to that have been registered by MCMQTT_RegisterSubTopic() for all nodes.

```
void MCMQTT_Publish(MQTTBrokerData *, char*, char*, bool);
```

Publishes a topic value pair at the broker which is identified by the pointer to the MQTTBrokerData structure.

Topics to be published do not require a preceeding initialization. Having its name and the payload – typically numeric – is sufficient.

```
void MCMQTT_Update(MQTTBrokerData *);
```

Called in loop() after the connection to the broker has either been checked or been re established to update the MQTT stack.

Globals

```
char MQTTClientName[] = "MC4AxisDemo";

MQTTBrokerData myMQTTData;
const int16_t DriveIDB = 2;
char DriveB_Name[] = "IE3";
int8_t HomingB = 33;

MCRremoteControlled RemoteDriveA(&myMQTTData, DriveIDB, DriveB_Name, HomingB, MQTTClientName);
```

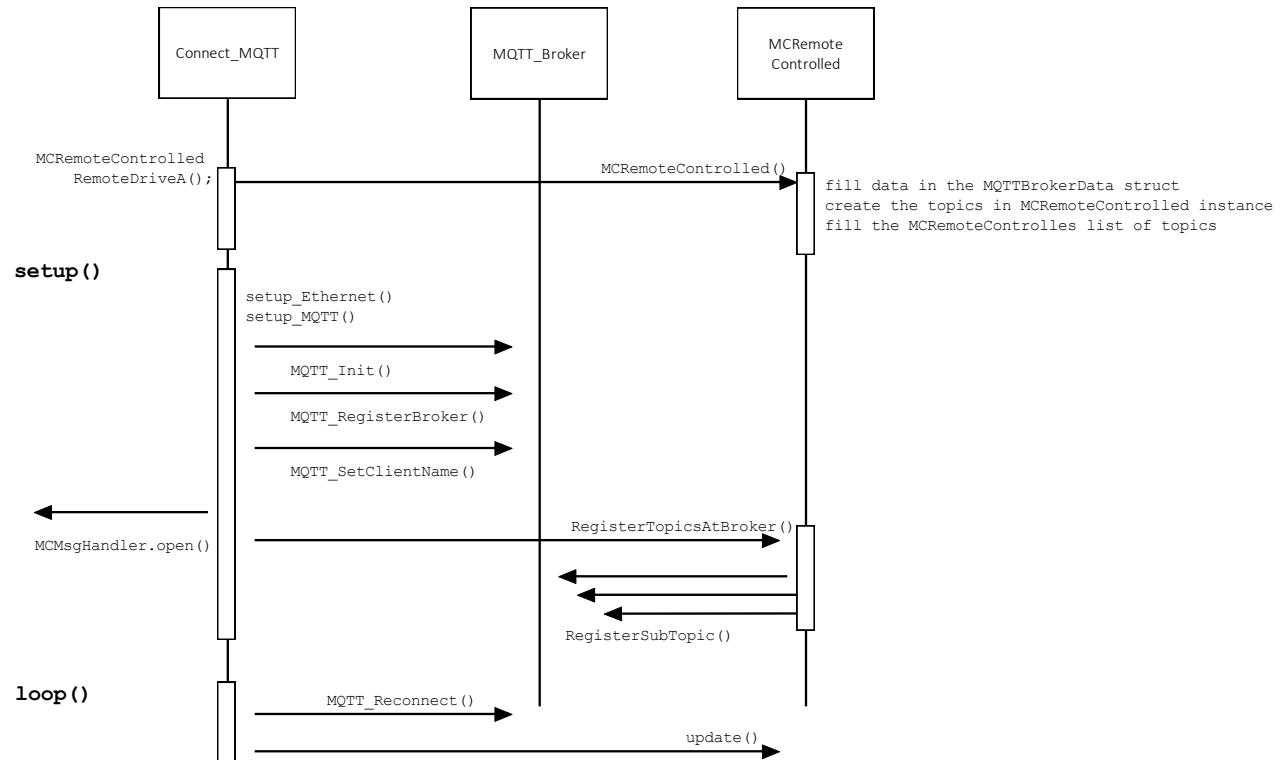


Figure 7 Initialization of the MQTT access

MCRremoteControlled

The behavior of the remotely controlled drive nodes is once again implemented in a class – **MCRremoteControlled**. As a starter a static list of topics to be published is created covering:

- A command for what action to be taken
- A target speed for speed controlled operation
- A target position for position controlled operation
- A few names of parameters of general interest which are used for publishing them at the broker

The main idea is for each remote controlled drive to register at the MQTT broker and to receive commands for the drive via the topic "Command" sub topic. New target values are received either via the "TPos" or the "TSpeed" sub topic.

Vice versa it's going to update its latest values via different predefined topics.

Within **MCRremoteControlled.Update()** whenever a new command has been received via the "Command" sub topic and the drive communication is in idle state the latest command is set to be executed next via the switch-case statement (see Code 8).

.Update() has to be called cyclically within the loop() as well as the loop() of the mqtt client via `MCMQTT_Update()`.

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 frontend then was designed using NodeRed (Figure 8). The blue boxes are taken out of the dashboard library and are visible on the panel in Figure 6. The raspberry colored ones are the ones connecting to the MQTT server either by subscription (left) or publishing (right).

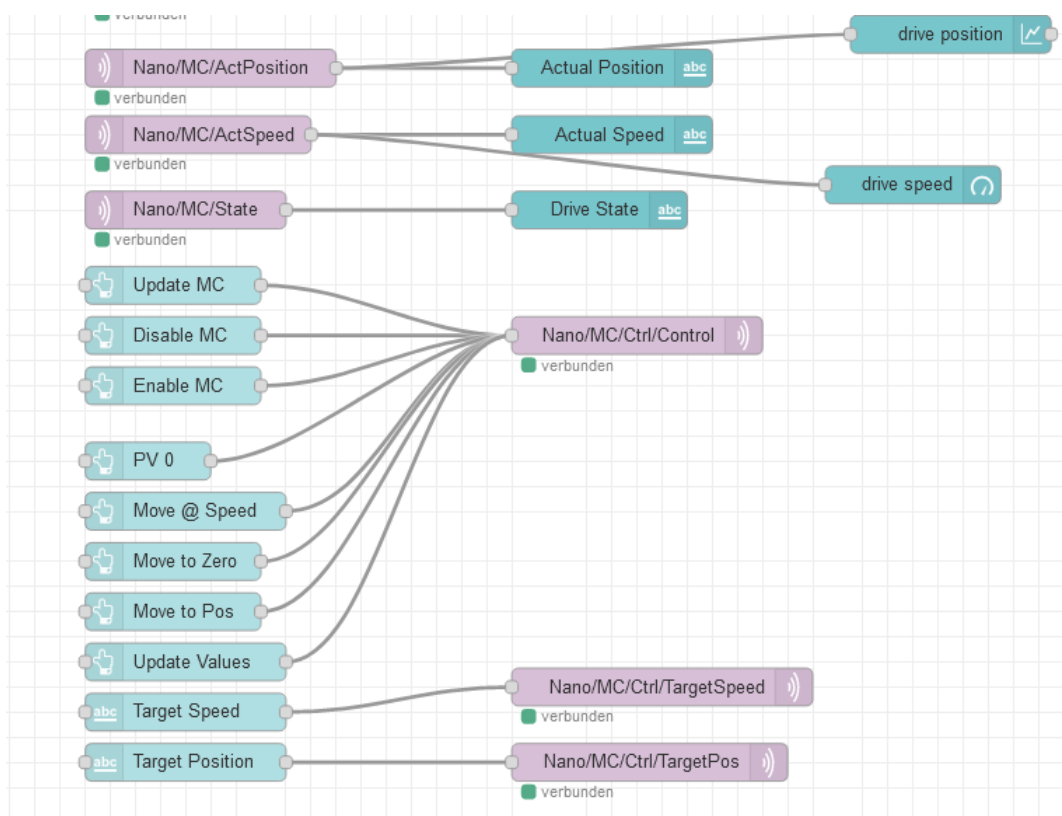


Figure 8 Flow of the monitoring panel

```
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 8 Handling the commands to the drive via the drive library

Customization of the Library

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

Any implementation of 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 10 is an example implementation using C and a C-struct to implement OO like handling.

Here `O_Usart2Isr` in Code 9 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 11 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 9 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 10) – a name defined by the ST environment – uses this pointer to not only call the class-aware handler `C_Usart_IrqHandler()` (Code 12) 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 10 Low level ISR to call the handler within Uart.c

```
struct C_Usart {
    // Contains 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;

    // Contains 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 11 Example C-struct to implement a OO like implementation of a class C_Usart.

Main elements within the struct C_Usart in Code 11 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 complete 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 pointer to the actual instance of C_Usart – 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 callback. 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 12 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
interrupt
        me->pSTM32Reg->CR1 |= (USART_CR1_TXEIE | USART_CR1_TE);
        // interrupt to send out first character
    }
}
```

Code 13 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 9. 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.

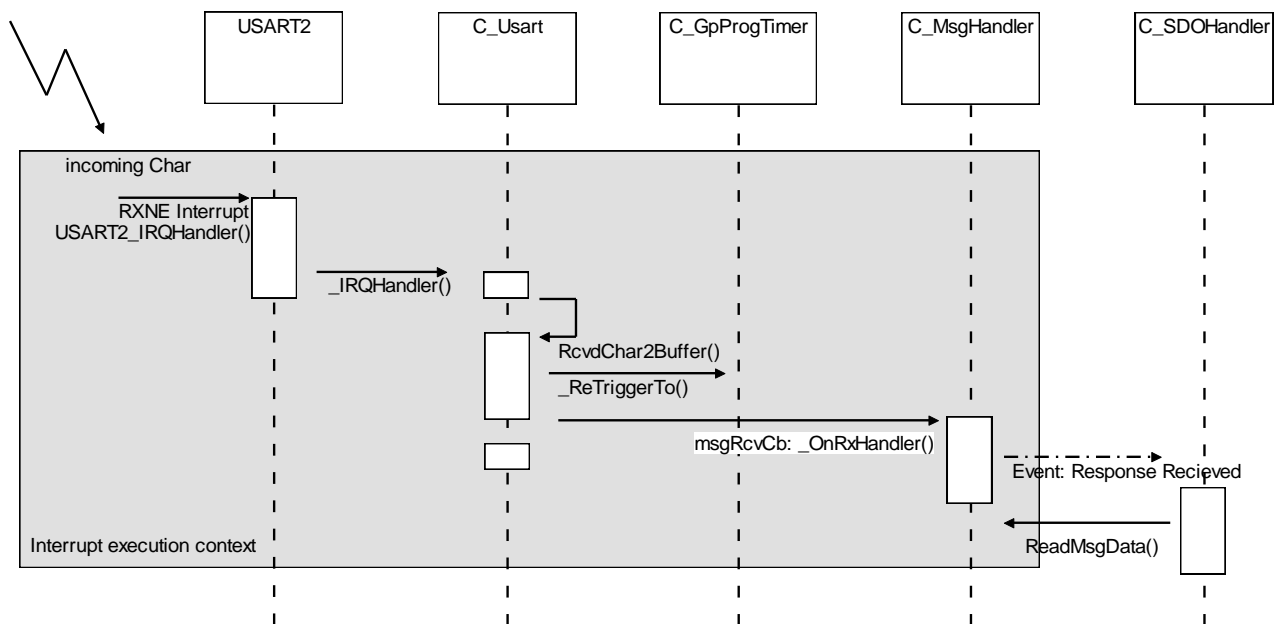


Figure 9 Sequence diagram of an interrupt based message handling

References

Arduino nano Every

<https://store.arduino.cc/arduino-nano-every>



Arduino nano 33 IoT

<https://store.arduino.cc/arduino-nano-33-iot>



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.