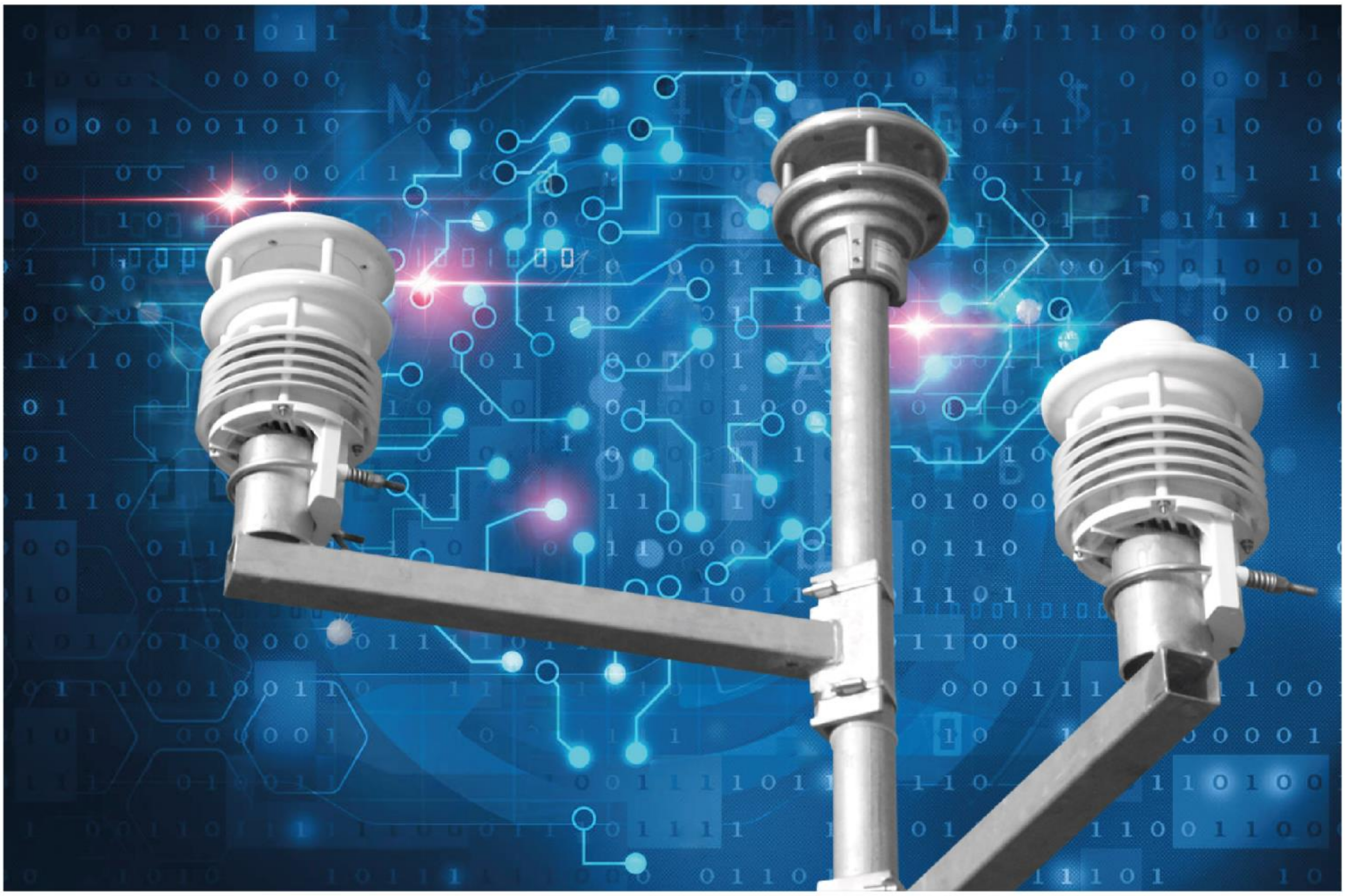

UMB Software Library

Instruction Manual

Content

Change History

Version	Datum	Änderungen
V0.1	12.03.2021	Initial Version
V0.2	22.03.2021	Screenshots adjusted Explanations to UMB Specification
V0.3	19.04.2021	64-bit versions of the library
V0.4	06.05.2021	64-bit version for ARM
V0.5	20.05.2021	Table of supported UMB commands

1 The UMB Protocol

The UMB protocol is an open binary protocol specified by the Lufft company for the configuration and data retrieval of measuring devices.

The current version of the specification can be found in the download area of the homepage www.Lufft.de. The document contains all information on the frame structure and timing as well as a detailed description of all commands.

2 The UMB Library

The library is written in the C language and is available for Windows and Linux.

It does not use dynamic memory allocation.

The commands of the UMB protocol listed in Table 1 are implemented in the library.

<cmd>	Description	Library V0.4
20h	Hardware and software version	
21h	Read out EEPROM	●
22h	Program EEPROM	●
23h	Online data request	
24h	Offline data request	
25h	Reset / default	●
26h	Status request	●
27h	Set time / date	
28h	Read out time / date	
29h	Test command	
2Ah	Monitor	
2Bh	Protocol change	
2Ch	Last fault message	
2Dh	Device information	●
2Eh	Reset with delay	
2Fh	Multi-channel online data request	●
30h	Set new device ID permanently (verc 1.0)	
30h	Set new device ID temporarily (verc 1.1)	
36h	UMB-Tunnel	
37h	Transfer Firmware	●
38h	Transfer Binary Data	
40h – 7Fh	Reserved for device-specific commands (see device description)	
80h – 8Fh	Reserved for development	
F0h	Program EEPROM with PIN	

Table 1 Commands of the UMB protocol, which are implemented by the library

A large number of device properties can be queried with the command 'Device information' (2Dh). So far, the sub-commands specified in Table 2 are supported.

<info>	Description	Library V0.4
10h	Device identification	●
11h	Device description	
12h	Hardware and software version	
13h	Extended version info	
14h	EEPROM size	
15h	No. of channels available	●
16h	Numbers of the channels	●
17h	Read number of device specific version information slots	
18h	Read device specific version information	

<info>	Description	Library V0.4
20h	Meas. variable of channel	
21h	Meas. range of channel	
22h	Meas. unit of channel	
23h	Data type of channel	
24h	Meas. value type	
30h	Complete channel info	●
40h	Number of IP interfaces	
41h	IP Information	

Table 2 Sub-commands of the 'Device information' command, which are supported by the library

3 Scope of Delivery

The folder "**lufft**" contains all files that are required to use the UMB library:

- Software libraries for Windows and Linux / Linux on ARM

	windows	linux	Linux / ARM
64 bit	UmbControllerLib.lib	libUmbController.a	libUmbControllerArm_64.a
32 bit	UmbControllerLib_32.lib	libUmbController_32.a	libUmbControllerArm_32.a

- The header files to use the library:
  UmbControllerLib.h: Interface of the library
  Umb_Types.h: General type definitions

In the "**src**" folder you will find files with examples for connecting the library to your own system:

- **UmbCtrlTest.cpp**: Test program to illustrate how it works
- **ComWin.c/.h**: Example implementation for connection under Windows
- **ComLinux.c/.h**: Example implementation for connection under Linux

The "**win**" folder contains non-Lufft files that are used in the test program or in the example implementations under Windows. The terms of use specified in the respective source files must be observed here.

The "**examples**" folder contains an example for installing the library on a RaspberryPi. Further examples are planned.

4 Commissioning

To use the UMB library, the two header files Umb_Types.h and UmbControllerLib.h must be copied into your own project.

Dependent on the system in use (Windows, Linux, Linux on ARM) the respective library is required, see also chapter 3.

The installation instructions for a RaspberryPi can be read separately in the README.txt file in the /examples/RaspberryPi directory.

5 Usage

The current functional scope of the library can be found in the interface file UmbControllerLib.h.

5.1 System Connection

The serial interface is controlled via function pointers that are defined in the `UMB_CTRL_COM_FUNCTION_T` structure, see Figure 1.

```
//! callback functions for communication
typedef struct
{
    void* pUserHandle;

    Std_ReturnType(*pfnInit)    (void* pUserHandle);
    Std_ReturnType(*pfnDeinit)  (void* pUserHandle);
    Std_ReturnType(*pfnUse)     (void* pUserHandle);
    Std_ReturnType(*pfnUnuse)   (void* pUserHandle);

    Std_ReturnType(*pfnTx)      (void* pUserHandle, const uint32 length, const uint8* const pBytes);

    Std_ReturnType(*pfnRxAvail) (void* pUserHandle, uint32* const pAvail);
    Std_ReturnType(*pfnRx)      (void* pUserHandle, const sint32 timeoutMs, const uint32 maxLen,
                                 uint32* const pLength, uint8* const pBytes);
    Std_ReturnType(*pfnRxClearBuf)(void* pUserHandle);

} UMB_CTRL_COM_FUNCTION_T;
```

Figure 1 Structure with function pointers for controlling the serial interface

The function pointers (*pfnInit) and (*pfnDeinit) are optional and e.g. can be used to open or close the serial interface. However, if this is already done elsewhere, the two function pointers can also be set to NULL.

All other function pointers are mandatory and must be implemented.

The function pointers (*pfnUse) und (*pfnUnuse) are intended for the protection of variables or code segments by semaphores. In the current example implentations these functions do not include active code.

The handle *pUserHandle can be used to pass user-specific data on to the callback functions. In the example implementations comWin.cpp and ComLinux.cpp, all data that are required during operation are summarized in a structure `COM_HANDLE_T`. *pUserHandle points to the address of such a data record, which means that this data is then available in the callback functions. Figure 2 shows the initialization of a *pUserHandle, Figure 3 the subsequent application.

```
UMB_CTRL_COM_FUNCTION_T* pComFunction = (UMB_CTRL_COM_FUNCTION_T*)malloc(sizeof(UMB_CTRL_COM_FUNCTION_T));

if (pComFunction)
{
    pComFunction->pUserHandle = malloc(sizeof(COM_HANDLE_T));

    if (pComFunction->pUserHandle)
    {
        COM_HANDLE_T* pComHandle = (COM_HANDLE_T*)pComFunction->pUserHandle;

        pComHandle->config = *pConfig;
        memset(&pComHandle->port, 0, sizeof(pComHandle->port));
    }
```

Figure 2 Initialization of a *pUserHandle

```
static Std_ReturnType ComInit(void* pUserHandle)
{
    COM_HANDLE_T* pComHandle = (COM_HANDLE_T*)pUserHandle;

    try
    {
        int number = std::strtoul(pComHandle->config.serialIf, NULL, 10);
        pComHandle->port.Open(number, pComHandle->config.baudrate, CSerialPort::NoParity, 8,
                              CSerialPort::OneStopBit, CSerialPort::NoFlowControl);
    }
    catch (CSerialException& e)
    {
        printf("Unexpected CSerialPort exception, Error:%u\n", e.m_dwError);
        return E_NOT_OK;
    }

    return E_OK;
}
```

Figure 3 Usage of a *pUserHandle

The modules ComLinux.cpp/.h and ComWin.cpp/.h show examples of how the assignment of these function pointers can be implemented:

The control of the serial interface is implemented directly in ComLinux, whereas ComWin uses third-party software (`SerialPort.h`) for which only the wrapper functions compatible with the UMB library are provided, see also Figure 4.

```c
static Std_ReturnType ComTx(void* pUserHandle, const uint32 length, const uint8* const bytes)
{
    COM_HANDLE_T* pComHandle = (COM_HANDLE_T*)pUserHandle;

    if (write(pComHandle->m_fdTTY, bytes, length) > 0)
    {
        return E_OK;
    }
    return E_NOT_OK;
}


static Std_ReturnType ComRx(void* pUserHandle, const sint32 timeoutMs, const uint32 maxLen,
    uint32* const pLength, uint8* const pBytes)
{
    COM_HANDLE_T* pComHandle = (COM_HANDLE_T*)pUserHandle;
    int retval;
    fd_set set;
    struct timeval timeout;

    if((pComHandle->m_fdTTY < 0) || (pLength == nullptr) || (pBytes == nullptr))
    {
        return E_NOT_OK;
    }
    FD_ZERO(&set);
    FD_SET(pComHandle->m_fdTTY, &set);
    timeout.tv_sec = timeoutMs / 1000;
    timeout.tv_usec = (timeoutMs % 1000) * 1000;
    retval = select(pComHandle->m_fdTTY + 1, &set, NULL, NULL, &timeout);
    if(retval > 0)
    {
        retval = read(pComHandle->m_fdTTY, pBytes, maxLen);
        if(retval > 0)
        {
            *pLength = retval;
            return E_OK;
        }
    }
    return E_NOT_OK;
}
```

```c
static Std_ReturnType ComTx(void* pUserHandle, const uint32 length, const uint8* const bytes)
{
    COM_HANDLE_T* pComHandle = (COM_HANDLE_T*)pUserHandle;

    pComHandle->port.Write(bytes, length);

    return E_OK;
}




static Std_ReturnType ComRx(void* pUserHandle, const sint32 timeoutMs, const uint32 maxLen,
    uint32* const pLength, uint8* const pBytes)
{
    COM_HANDLE_T* pComHandle = (COM_HANDLE_T*)pUserHandle;
    COMMTIMEOUTS timeouts;

    pComHandle->port.GetTimeouts(timeouts);
    timeouts.ReadIntervalTimeout = MAXDWORD;
    timeouts.ReadTotalTimeoutMultiplier = MAXDWORD;
    timeouts.ReadTotalTimeoutConstant = timeoutMs;
    pComHandle->port.SetTimeouts(timeouts);

    *pLength = pComHandle->port.Read(pBytes, maxLen);

    return E_OK;
}
```

Figure 4 Implementation examples for controlling the serial interface:
left: Example for Linux, manual implementation
right: Example for Windows, usage of already existing implementation

5.2 Initialization

The initialization of the UMB library comprises 3 points:

- Allocation of the function pointers to control the serial interface
  For the sake of clarity, it is best to assign the required function pointers in a separate function defined by the user, see section 5.1.

- Provision of the handle
  The UMB library does not use dynamic memory allocation. Therefore, the user must provide the memory for the library instances used.
  This handle is required when calling all other functions of the UMB library.

- Calling the initialization function of the library
  The handle and the variable that contains the function pointers, must be given to the initialization function UmbCtrl_Init().

Figure 5 shows an example of the initialization sequence, Figure 6 a query of the device name and the device status.

```
int main(int argc, char* argv[])
{
    UMB_CTRL_STATUS_T status;
    UMB_CTRL_T *pUmbCtrl;

    // UMB lib version
    UMB_CTRL_VERSION_T version = UmbCtrl_GetVersion();
    printf("UMB Lib Version: major=%d, minor=%d\n", version.major, version.minor);

    // Initialization
    // TODO: Adjust to used serial interface
    char serialIf[] = { "1" };
    COM_CONFIG_T comConfig;
    UMB_CTRL_COM_FUNCTION_T * pUmbCtrlComFunction;

    // TODO: Adjust to used baudrate
    comConfig.baudrate = 19200;
    comConfig.serialIf = serialIf;
    pUmbCtrlComFunction = ComFunctionInit(&comConfig);

    pUmbCtrl = malloc(UmbCtrl_GetHandleSize());
    status = UmbCtrl_Init(pUmbCtrl, pUmbCtrlComFunction, 0);
```

Figure 5 Initialization of the UMB library

```
// Further processing
UMB_ADDRESS_T umbAddress;
// TODO: Adjust to used class id / device id
umbAddress.deviceId = 0x01; // device id: 1
umbAddress.classId = 0x70;  // class id: 7 = weather station

uint8 name[41] = { 0 };
status = UmbCtrl_GetDevName(pUmbCtrl, umbAddress, name);
if (status.global == UMB_CTRL_STATUS_OK)
{
    printf("Device name: %s\n", name);
}
else
{
    printf("ERROR [request device name]: lib=0x%0X dev=0x%0X\n",
        status.detail.library, status.detail.device);
}

ERROR_STATUS_T deviceStatus;
status = UmbCtrl_GetDevStatus(pUmbCtrl, umbAddress, &deviceStatus);
if (status.global == UMB_CTRL_STATUS_OK)
{
    printf("Device status: %d\n", deviceStatus);
}
else
{
    printf("ERROR [request device status]: lib=0x%0X dev=0x%0X\n",
        status.detail.library, status.detail.device);
}
```

Figure 6 Query of device name and device status

5.3 Test Programm

The test program in UmbCtrlTest.cpp shows an example of how to use the UMB Controller library. Before using the test program, all places marked with 'TODO' in the `main()` program must be adapted to your own test system. These are

- Preprocessor definition _USE_NCURSES, in order to be able to use the graphical progress display for the update function under Linux (for more details see below)
  ```
  #define _USE_NCURSES
  ```

- Used serial interface, e. g.
  ```
  char serialIf[] = { "3" };
  ```
  Note:
  Under Linux, the entire path of the serial interface must be specified here, e.g.
  ```
  char serialIf[] = { "/dev/tty03" };
  ```

- Baud rate of the serial interface, e. g.
  ```
  comConfig.baudrate = 19200;
  ```

- UMB address of the UMB device to be used for communication, e.g.
  ```
  umbAddress.deviceId = 0x01; // device id: 1
  umbAddress.classId = 0x70;  // class id: 7 = weather station
  ```

- Path and name of the firmware file, e.g.
  ```
  char fileName[] = { "C:\\Projekte\\UmbController\\WS100_update.bin" };
  ```

The functions that have been commented out (see Figure 7) are best transferred into the test program individually and as required in order to become familiar with the respective functionality.

```
    //writeMemory(pUmbCtrl, umbAddress);
    //getChannelInfo(pUmbCtrl, umbAddress);
    //getChannelData(pUmbCtrl, umbAddress);
    //firmwareUpdate(pUmbCtrl, umbAddress);

    // De-Initialization
    UmbCtrl_Deinit(pUmbCtrl);
}
```

Figure 7 Example functions for using the UMB library

About the preprocessor definition_USE_NCURSES

The example implementation `firmwareUpdate()` uses a graphical representation of the update progress, which requires the ncurses package under Linux. This must be installed manually e. g. on a RaspberryPi, since it is not preinstalled via raspbian-stretch-lite.

Instructions for this can be found in the README.txt file in the /examples/RaspberryPi directory.

If this progress display is to be used, the preprocessor definition _USE_NCURSES must be set after the ncurses package is installed. If, on the other hand, this instruction is commented out, a simple progress display is used instead of the graphical one, which does not require any further packages.

6 Notes on Firmware Update

Older UMB devices such as WSx00, Ventus, Anacon etc. use an update file in .mot format. These cannot be transferred to a device via the UMB protocol, but only via Hexload.

Therefore, for the new generation of UMB devices such as MARWIS, WS1000, WS100, SHM31 and others the .bin file format was defined, which also enables a firmware update via UMB.

- ➔ Firmware updates via the UMB protocol are only possible for UMB devices whose update file is in .bin format

Lufft
an OTT HydroMet brand

OTT
HydroMet