knobloch electronic

fischertechnik Robo Interface

FtLib Library Specification

Version FtLib: 1.70a
Status as of: 5/5/2008

Knobloch GmbH
Weedgasse 14
D-55234 Erbes-Büdesheim

entwicklung@knobloch-gmbh.de
www.knobloch-gmbh.de

1 Table of contents

2 Robo Interface Program

Up to three programs per download can be stored in the Robo interface. Program 1 and program 2 are permanently stored in FLASH memory, a third program can be stored in RAM. RAM memory is erased when a flash program is started and also when a power failure to the interface occurs.

The selection of the active program takes place via the program pushbutton. Should this be active longer than 0.5 seconds, the desired program can be selected. Both program LEDs for program 1 / 2 illuminate in succession if a program is saved in the respective memory cells. If a program is stored in RAM, this will be indicated by the illuminated LEDs. If the memory cells are empty, the respective program cannot be executed.

In order to start or stop the indicated program, the program pushbutton must be briefly activated (<500ms). The selection, starting, and stopping of programs may also occur via the interfaces (USB, serial, radio).

3 PC Interfaces

3.1 General

Interface selection on the Robo interface occurs by pressing buttons. After it is turned on, “AutoScan” mode becomes active. The USB, serial interface, and the radio module (if available) will be checked to see if data exists. This state is identifiable by the illumination of the interface light-emitting diodes.

As soon as an interface sends data, the other interfaces are disabled. The active interface blinks in order to indicate data communication. If no data passes through the active interface for more than 300ms, AutoScan mode is activated again.

By pressing the “Port” pushbutton, the next operating mode is selected based upon the following table.

1. AutoScan USB – Serial - Radio [1)]
2. AutoScan USB - Serial
3. USB
4. Serial
5. IR Direct

1)

This operating mode is only activated if the radio module is installed.

If the Port pushbutton is activated for more than 3 seconds, the interface goes into “Intelligent Interface Online Mode.” The serial interface then operates with the parameters 9600,n,8,1. The operating mode is displayed by the fast-blinking “COM” light-emitting diode. In this operating mode, the interface performs like an Intelligent Interface in online mode. However, no programs can be downloaded. By depressing the Port pushbutton, AutoScan operating mode is once again activated.

3.2 Multiple USB Interfaces on One Computer

In order to operate multiple interfaces on the USB bus, every interface must initially be assigned its own serial number. By default, all interfaces are provided with the same serial number in order to avoid problems during the exchange of interfaces. The Windows operating system distinguishes identical devices on the basis of their serial numbers. For “every” serial number, the corresponding driver is then installed. Administrator privileges are required for this under Windows 2000 / XP.

Thus, all ROBO interfaces and ROBO I/O extensions are provided with the identical serial number by default. There are no problems as long as only one interface is utilized on one computer. The computer differentiates the products through their name (ROBO interface, ROBO I/O extension, and Robo RF DataLink) and by their respective serial numbers. Thus, a Robo interface and a Robo I/O extension can be operated simultaneously on one computer without changing the serial number since different products are involved.

But if multiple identical products (e.g., Robo interfaces) are to be operated on one computer via USB, the serial number of the supplementary interface to be connected to the computer must first be changed so that it can be differentiated by the computer.

Note: no serial number needs to be changed with a connection via the serial interface on the computer.

Thus, the interface has stored two serial numbers. Whether the default serial number “1” or the device serial number “2” programmed by the manufacturer becomes active during activation of the device can be set via the software.

Changing the serial number can be accomplished via the software FtDiag.exe, RoboPro, or the function GetFtDeviceSetting(), or SetFtDeviceSetting() of the Ftlib.

In order to change the serial number, only one product may be connected to the USB, otherwise the computer cannot distinguish it. In FtDiag.exe, after “SCAN USB” and clicking on the button “USB Device,” invoke the Properties / Setup menu. In this template one can set the desired serial number that becomes active after the next start.

Warning: if the serial number is changed, the Windows driver may have to be reinstalled during the activation of the interface. However, administrator privileges are required for this under Windows 2000 / XP. If one does not have this, one cannot install the driver and thus could not access the interface via USB. In this case one can depress the PROG pushbutton during interface activation until the lamp test (“lighting console”) ends during activation. The interface then utilizes the serial number “1” and is recognized again by the installed driver (e.g., in order to change the serial number to permitted values).
Note that the serial numbers imprinted on the products are in hexadecimal format!

Even though USB “theoretically” allows up to 127 devices, practice has shown that under Windows XP (with SP1) running a 3 GHz Pentium 4, only about 4-5 products can be reliably operated in parallel (i.e., the update time of the TransferArea amounts to a maximum of 10ms); under Windows 98, it is up to 10 devices. This is due to the fact that the internal Windows drivers for the motherboard hardware under XP are not optimized for “real time applications.” Thus, it makes more sense to connect IO extensions to the Robo Interfaces instead of every product individually to USB.

4 Library for Windows (FtLib)

The library supports the fischertechnik USB products (serial and USB interfaces) as well as the Intelligent Interface on the serial interface.
The library is available in the following versions for Microsoft Visual C++ Studio 6.0:

FtLib_Static_LIBCMT_Release.lib = Multithreaded Static (Linker option: /MT)
Project settings on C/C++ tab under “Code Generation – Runtime Library:” Multithreaded

FtLib_Static_LIBCMTD_Debug.lib = Multithreaded Static Debug (Linker option: /MT d)
Project settings on C/C++ tab under “Code Generation – Runtime Library:” Multithreaded Debug

FtLib_Static_MSVCRT_Release.lib = Multithreaded DLL (Linker option: /MD)
Project settings on C/C++ tab under “Code Generation – Runtime Library:” Multithreaded DLL

FtLib_Static_MSVCRTD_Debug.lib = Multithreaded DLL Debug (Linker option: /MD d)
Project settings on C/C++ tab under “Code Generation – Runtime Library:” Multithreaded DLL Debug

The following functions are also included in the dynamic library FtLib.DLL.

4.1 Device Handling Function

4.1.1 DWORD GetLibVersion (void)

USB: yes
COM: yes
This routine returns the version number of the current library.

Call: ---
Return: Version number 0 0 x y (4 bytes, xy = version X.Y)

4.1.2 DWORD InitFtLib(void)

USB: yes
COM: yes
In order to be able to use the data control for the interface, variables must be initialized and memory regions are required. This occurs with this routine. It must be called as the first routine. We recommend making the call directly in the initialization routine of the application (e.g., OnInitDiaglog() with MFC applications). The function has multithreading capability so that it can also be invoked at a later time.

Before the conclusion of the application, the counterpart to this routine, “CloseFtLib(void),” must be called in order to free reserved memory.

Call:	---	
Return:	Error code	
	FTLIB_ERR_SUCCESS	no error with Init
	otherwise an error code	(see FtLib.h)
e.g.:	FTLIB_ERR_LIB_IS_INITIALIZED	Library is already initialized
	FTLIB_ERR_USB_NOT_SUPPORTED_FROM_OS	USB is not supported by the operating system (Windows-95 and Windows NT)

4.1.3 DWORD CloseFtLib (void)

USB: yes
COM: yes
This routine should be called at the conclusion of the application. It frees all memory regions and deletes handles that are still open.

Call:	---	
Return:	Error code	
	FTLIB_ERR_SUCCESS	no error
	otherwise an error code	(see FtLib.h)

4.1.4 DWORD IsFtLibInit (void)

USB: yes
COM: yes
This routine returns information as to whether or not InitFtLib() has already been executed.

Call:	---	
Return:	FTLIB_ERR_LIB_IS_INITIALIZED	region already allocated
	FTLIB_ERR_LIB_IS_NOT_INITIALIZED	region not yet allocated

4.1.5 DWORD InitFtUsbDeviceList (void)

USB: yes
COM: no
This routine creates a list of currently connected devices and saves device-specific data. The devices will not be opened. The variable “Device Number” (for GetNumFtUsbDevice()) is also set. The first device in the generated list is addressed with index “0.”

For every RF data link module found on the USB connection, a device type of "FT_ROBO_RF_DATA_LINK" is saved. This type can be queried with GetFtDeviceTyp (). Additionally, with an RF data link module a search will be made over radio for Robo interfaces on the set frequency. For every detected Robo interface, an additional entry will be created directly in the connection as type "FT_ROBO_IF_OVER_RF.“ Through this procedure, all eight of the possible interfaces can be activated via radio. However, only one module can be opened at the same time. Please note that for every detected RF data link module, this function requires a few seconds in order to find the potential Robo interfaces. Program execution can thereby be delayed for several seconds.
If a transfer thread is subsequently started upon insertion of the RF data link module, then FtLib will start the transfer thread for the first detected Robo interface.

Important: The generated list is not sorted. After every call to this function, the detected devices may be arranged in another sequence (particularly if new devices are connected in the meantime). Robo interfaces accessible via this module are solely found with a connection to an RF data link module.

Should there be devices missing during this function call, the function will terminate with an error.

Call:	---	
Return:	Error code	
	FTLIB_ERR_SUCCESS	no error
	otherwise an error code	(see FtLib.h)
e.g.:	FTLIB_ERR_SOME_DEVICES_ARE_OPEN	

4.1.6 UNIT GetNumFtUsbDevice (void)

USB: yes
COM: no
This routine returns the number of connected fischertechnik USB devices. This number is set by invoking InitFtUsbDeviceList().

Call:	---	
Return:	UINT	Number of detected devices

4.1.7 FT_HANDLE GetFtUsbDeviceHandle (UCHAR ucDevNr)

USB: yes
COM: no
This routine returns a handle to the desired USB device in the table generated by InitFtUsbDeviceList(). The first device is addressed with index “0.” Important: The list generated by InitFtUsbDeviceList is not sorted. After every call to this function, the detected devices may be arranged in another sequence. Access to the device or its data occurs via this handle.

Call:	UCHAR ucDevNr	Index in the USB device table
Return:	FT_HANDLE	Handle of the USB device With errors, NULL is returned

4.1.8 FT_HANDLE GetFtUsbDeviceHandleSerialNr (DWORD dwSN, DWORD dwTyp)

USB: yes
COM: no
This routine returns a handle to the USB device if it exists in the list generated by InitFtUsbDeviceList() through the device specified by the serial number. Since a serial number can occur multiple times for various devices, the variable “dwTyp” specifies the device class. The parameter type “FT_AUTO_TYPE” passes the handle of the first detected device with the desired serial number.

Call:	DWORD dwSN	Serial number of the USB device
	DWORD dwTyp	Type of the device to be opened
	FT_AUTO_TYPE	Returns the first device with this SN
	FT_ROBO_IF_USB	Robo interface at the USB port
	FT_ROBO_IO_EXTENSION	Robo I/O extension
	FT_ROBO_RF_DATA_LINK	Robo RF data link
Return:	FT_HANDLE	Handle of the USB device In the event of an error, NULL is returned

4.1.9 DWORD OpenFtUsbDevice (FT_HANDLE hFt)

USB: yes
COM: no
This routine opens various channels to the connected device according to DeviceTyp. Note: the thread that opens the device must also disconnect it!

Call:	FT_HANDLE hFt	Handle of the USB device
Return:	Error code	
	FTLIB_ERR_SUCCESS	no error
	otherwise an error code	(see FtLib.h)

4.1.10 FT_HANDLE OpenFtCommDevice(DWORD dwPort, DWORD dwTyp, DWORD dwZyklus, DWORD *pdwError)

USB: no
COM: yes

This routine opens a connection for the specified COM channel (COM1…COM4) and returns a handle to it. Depending upon the specified device type, the interface will automatically be set to 9600 or 38400 baud.

The value in the variable “dwZyklus” defines query behavior and is interface-specific:

FT_INTELLIGENT_IF:
The value specifies after how many normal “input queries” both of the analog inputs EX and EY are prompted at the Intelligent Interface. Default is “FT_ANALOG_CYCLE.”

FT_ROBO_IF_COM
With slow computing, it is possible that the interfaces can no longer be queried in the required time during a large data arrival. For this reason, one can specify via this parameter if certain rarely used values get passed.
0 = This corresponds to the default query (for slower computers). The analog values A1 and AV of I/O extensions 1..3 will not be queried.
>0 = The analog values A1 and AV of I/O extensions 1..3 will be queried.
Note:
This functionality is integrated only as of Robo interface firmware 1.45.0.3 and FtLib version 0.41.

In the event of an error during connection establishment, the error code will be saved in a DWORD variable to which the pointer “pdwError” will point (NULL=saving not desired).

The type indication is important to the effect that the correct query codes are worked with in the communication thread. The parameter “FT_INTELLIGENT_IF_SLAVE” causes the extension module to be handled like a SLAVE-1 with the communication thread.

The connection is disconnected again with the function “CloseAllFtDevices()” and “CloseFtDevice().”

Call:	DWORD	dwPort	PORT_COM1…PORT_COM4
	DWORD	dwTyp	Type of the connected interface
		FT_INTELLIGENT_IF	Intelligent Interface (9600 bps)
		FT_INTELLIGENT_IF_SLAVE	Intelligent Interface with extension module (9600)
		FT_ROBO_IF_IIM	Robo interface in Intelligent If mode (9600)
		FT_ROBO_IF_COM	Robo interface on the COM port (38400)
	DWORD	dwZyklus	(only with Intelligent Interface, e.g.: FT_ANALOG_CYCLE)
	DWORD	*pdwError	Pointer to an error variable
			Constant values: (see FtLib.h)

Return: Handle to the device (NULL = error).
*pdwError then contains an error code (see FtLib.h)

4.1.11 DWORD SetFtDeviceCommMode (FT_HANDLE hFt, DWORD dwMode, DWORD dwParameter, USHORT *puiValue)

USB: yes
COM: no

This routine sets the operating mode of the serial interface in the interface. After activation, the interface will be in normal operation. In this operating mode, the interface can be operated in online mode.
In “message mode,” messages from one interface can be sent to another via the serial interface. An X cable (crossed sending and receiving path) is required for this.

The operating mode that is set remains in place until another operating mode is set with this function. By pressing the “Port” pushbutton on the interface, the IF_COM_ONLINE operation mode is set again if the interface is in AutoScan mode.

The currently set state can be queried via “IF_COM_PARAMETER.” The result will be saved in the variables pointed to by “puiValue” if, during the call, the value of “puiValue” is not NULL.

Mode =	IF_COM_ONLINE:	Set default mode The interface will be set to the default mode (parameters: 38400,n,8,1).
Mode =	IF_COM_MESSAGE:	Message notification via serial interface
Mode =	IF_COM_PARAMETER	Select operating mode *puiValue (low byte) = set mode

Call:	FT_HANDLE hFt	Handle of the USB device
	DWORD dwMode	Operating mode of the serial interface:
	IF_COM_ONLINE	0x01: Online mode (38400,n,8,1)
	IF_COM_MESSAGE	0x03: Message mode (38400,n,8,1)
	IF_COM_PARAMETER	0x05: do not change anything, only query current status and save in *puiValue
	USHORT puiValue	Pointer to a USHORT variable (NULL, if not desired) Here, the result is saved with low byte = current operating mode (see dwMode)

Return:	Error code	
	FTLIB_ERR_SUCCESS	no error
	otherwise an error code	(see FtLib.h)

4.1.12 DWORD CloseAllFtDevices ()

USB: yes
COM: yes
This routine disconnects all connections for all possible devices.

Call:	---	
Return:	Error code	
	FTLIB_ERR_SUCCESS	no error
	otherwise an error code	(see FtLib.h)

4.1.13 DWORD CloseFtDevice (FT_HANDLE hFt)

USB: yes
COM: yes
This routine disconnects all connections for the specified device. If the communication thread is still running, it will be stopped.
Note: the thread that opens the device must also disconnect it!

Call:	FT_HANDLE hFt	device handle
Return:	Error code	
	FTLIB_ERR_SUCCESS	no error
	otherwise an error code	(see FtLib.h)

4.1.14 DWORD GetFtDeviceTyp (FT_HANDLE hFt)

USB: yes
COM: yes
This routine returns the type of the connected device for the specified device handle.
Note: This data is read by InitFtUsbDeviceList() from the device and saved, or it is set by OpenFtCommDevice().

Call:	FT_HANDLE hFt	device handle
Return:	DWORD device type	
	NO_FT_DEVICE	no device is connected / unknown
	FT_INTELLIGENT_IF	Intelligent Interface (serial)
	FT_INTELLIGENT_IF_SLAVE	Intelligent Interface with extension (serial)
	FT_ROBO_IF_IIM	FT-Robo interface in Intelligent Interface mode (serial)
	FT_ROBO_IF_USB	Robo interface at the USB port
	FT_ROBO_IF_COM	Robo interface at the COM port
		otherwise error code (if value > FT_MAX_TYP_NUMBER, see FtLib.h)

4.1.15 LPCSTR GetFtSerialNrStrg (FT_HANDLE hFt)

USB: yes
COM: yes
This routine returns a pointer to the string with the current serial number in ASCII format with which the device is registered in the computer.

Note: This data is read from the device by InitFtUsbDeviceList() or OpenFtCommDevice() and saved. If a product is not supported by any serial numbers (e.g., Intelligent Interface), a serial number of “1” is written to the string.

Call:	FT_HANDLE hFt	device handle
Return:	LPCSTR	Pointer to string if pointer = NULL, then error (Area opened, correct DevNr?)

4.1.16 DWORD GetFtSerialNr (FT_HANDLE hFt)

USB: yes
COM: yes
This routine returns the current serial number as a long integer (DWORD) with which the device is registered in the computer. “0” is returned with an error since this serial number does not exist.

Note: This data is read from the device by InitFtUsbDeviceList() or OpenFtCommDevice() and saved. If a product is not supported by any serial numbers (e.g., Intelligent Interface), a serial number of “1” is set.

Call:	FT_HANDLE hFt	device handle
Return:	DWORD	Serial number of the device if “0” then error (Area opened, correct DevNr?)

4.1.17 LPCSTR GetFtFirmwareStrg (FT_HANDLE hFt)

USB: yes
COM: yes
This routine returns a pointer to the string with the current interface firmware in ASCII format.
Note: This data is read from the device by InitFtUsbDeviceList() or OpenFtCommDevice() and saved.

Call:	FT_HANDLE hFt	device handle
Return:	LPCSTR	Pointer to string if pointer = NULL, then error (Area opened, correct DevNr?)

4.1.18 DWORD GetFtFirmware (FT_HANDLE hFt)

USB: yes
COM: yes
This routine returns the version number of the firmware of the connected device as an integer (DWORD). “0” is returned with an error since this firmware version does not exist.
Note: This data is read from the device by InitFtUsbDeviceList() or OpenFtCommDevice() and saved.

Call:	FT_HANDLE hFt	device handle
Return:	DWORD	Firmware version of the device in format 3.2.1.0 (3=high byte, 0=low byte) if value = “0” then error (Area opened, correct DevNr?)

4.1.19 LPCSTR GetFtManufacturerStrg (FT_HANDLE hFt)

USB: yes
COM: yes
This routine returns a pointer to the manufacturer string in ASCII format.
Note: This data is read from the device by InitFtUsbDeviceList() or OpenFtCommDevice() and saved.

Call:	FT_HANDLE hFt	device handle
Return:	LPCSTR	Pointer to string if pointer = NULL, then error (Area opened, correct DevNr?)

4.1.20 LPCSTR GetFtShortNameStrg (UCHAR ucDevNr)

USB: yes
COM: yes
This routine returns a pointer to the short name string of the device in ASCII format.
Note: This data is read from the device by InitFtUsbDeviceList() or OpenFtCommDevice() and saved.

Call:	FT_HANDLE hFt	device handle
Return:	LPCSTR	Pointer to string if pointer = NULL, then error (Area opened, correct DevNr?)

4.1.21 LPCSTR GetFtLongNameStrg (FT_HANDLE hFt)

USB: yes
COM: yes
This routine returns a pointer to the long name string of the device in ASCII format.
Note: This data is read from the device by InitFtUsbDeviceList() or OpenFtCommDevice() and saved.

Call:	FT_HANDLE hFt	device handle
Return:	LPCSTR	Pointer to string if pointer = NULL, then error (Area opened, correct DevNr?)

4.1.22 LPCSTR GetFtLibErrorString (DWORD dwErrorCode, DWORD dwTyp)

USB: yes
COM: yes
This routine returns a pointer to a string that describes the error code assigned in dwErrorCode. During the call, it can be specified in dwTyp whether the error constant is delivered as a string or as an error description in English.

Call:			
	DWORD	dwErrorCode	Error code
	DWORD	dwTyp	Type of returned string 0 = constant as a string 1 = error text in English
Return:	LPCSTR		Pointer to string

4.1.23 DWORD GetFtDeviceSetting (FT_HANDLE hFt, FT_SETTING *pSet)

USB: yes
COM: yes
This routine writes the current interface data of the selected device in the struct addressed by the pointer pSet. The FT_SETTING struct is defined in the header file ftlib.h (only the variables identified with RW can be changed with the function SetFtDeviceSetting()).

Call:	FT_HANDLE hFt	device handle
	FT_SETTING*	pointer to an FT_SETTING struct
Return:	Error code	
	FTLIB_ERR_SUCCESS	no error
	otherwise an error code	(see FtLib.h)
	current error codes:	
	FTLIB_ERR_INVALID_PARAM	pointer *pSet is NULL

4.1.24 DWORD SetFtDeviceSetting (FT_HANDLE hFt, FT_SETTING *pSet)

USB: yes
COM: yes

This routine writes the interface data into the desired device from the struct addressed by the pointer pSet. The FT_SETTING struct is defined in the header file ftlib.h (only the variables identified with RW can be changed with this function).

Call:	FT_HANDLE hFt	device handle
	FT_SETTING*	pointer to an FT_SETTING struct If the passed value is NULL, no struct will be used. Functions not utilized within the struct must be deactivated with a NULL pointer.
Return:	Error code	
	FTLIB_ERR_SUCCESS	no error
	otherwise an error code	(see FtLib.h)
	current error codes:	
	FTLIB_ERR_INVALID_PARAM	pointer *pSet is NULL

4.1.25 DWORD SetFtDistanceSensorMode (FT_HANDLE hFt, DWORD dwMode, DWORD dwTol1, DWORD dwTol2, DWORD dwLevel1, DWORD dwLevel2, DWORD dwRepeat1, DWORD dwRepeat2)

USB: yes
COM: yes

This routine initializes the D1/D2 input on the interface to the connection on the fischertechnik gap sensors or for the measurement of voltage ranges of 0-10 volts.

These parameters can also be set with the function SetDeviceSetting().

Import note:
Since the operating mode of the D1 / D2 inputs can be set by means of software, we recommend that no voltage be supplied “directly” to these connections in order to avoid damage to the interface during software errors. Since the inputs are highly resistive, a resistance of approximately 200 Ohm – 470 Ohm should be directly connected to the D1 / D2 socket (series connection). We recommend to connect the voltage range to be measured “behind” it.

Call:		
	FT_HANDLE hFt	device handle
	DWORD dwMode	Operating mode of the connections: IF_DS_INPUT_VOLTAGE = Inputs measure voltage range IF_DS_INPUT_ULTRASONIC = Inputs for ft Ultrasonic sensors

The following parameters are dependent upon the set operating mode.
For dwMode = IF_DS_INPUT_DISTANCE:

DWORD dwTol1	Range of tolerance D1 (recommended: 20)
DWORD dwTol2	Range of tolerance D2 (recommended: 20)
DWORD dwLevel1	Threshold D1
DWORD dwLevel2	Threshold D2
DWORD dwRepeat1	Repetition value D1 (recommended: 3)
DWORD dwRepeat2	Repetition value D2 (recommended: 3)

The gap sensor works with infrared light and can thus be disturbed by outside influences (e.g., IR hand transmitters). In order to eliminate these disturbances, it can be determined through specification of the repetition values how often the “identical” value has to be measured in order to be recognized as valid. Since the measurements can vary slightly from one to the next one, there exists a range of tolerance. As soon as a new measurement begins, you can change the following measurements within this “window” without it resulting in a restart of the measurements. The threshold designates a level for the analysis as “digital” gap sensors. Below the threshold a logical “0” is reported; above it, a “1.” The determined states of the firmware in the locations “base+0x0C” (digital) and “base+0x1C / base+0x1E” (analog) are saved in the transfer area.

Return:	
Error code	
FTLIB_ERR_SUCCESS	no error
otherwise an error code	(see FtLib.h)

4.2 Function for Online Communication

The querying and routing of the outputs occurs through a “transfer area” (communication storage region). This region is synchronized with the interface every 10ms after the start of the communication thread (by means of StartFtTransferArea). It does not matter if the interface is connected to the PC via USB, serially, or by radio. It also does not matter if it involves an intelligent interface, a Robo interface, or an IO extension. The library recognizes the product in question via the device handle and updates the existing inputs / outputs of the respective interface type at the latest every 10ms. In the event that multiple interfaces are connected to the computer, a separate thread for every connected product can be started. The design of the transfer area can be found in this chapter’s extension.

4.2.1 DWORD StartFtTransferArea (FT_HANDLE hFt, NOTIFICATION_EVENTS* sNEvent)

USB: yes
COM: yes

This routine starts a communication thread that continually synchronizes the data from the transfer area with the device. This functionality is required for online operation.

The device must be opened beforehand via the functions OpenFtUsbDevice() or OpenFtCommDevice(). Depending upon the serial device type (indicated by OpenFtCommDevice() during opening of the interface), only the values supported by the interface are queried.

During the call, a pointer to a NOTIFICATION_EVENTS struct can be passed to the function (NULL if non-existent). Pointers or handles for a callback routine and event and message handles can be passed in this struct. When the thread receives new data from the device (every 10ms at the latest), messages are always sent or the callback routine is always called if their pointer values or handle values are not NULL. The invoked callback routines may not call any FtLib functions or else a deadlock may occur.

Call:	FT_HANDLE hFt	device handle
	NOTIFICATION_EVENTS*	Pointer to a notification struct If the passed value is NULL, no struct will be used. Functions not utilized within the struct have to be deactivated with a NULL pointer.
Return:	Error code	
	FTLIB_ERR_SUCCESS	no error
	otherwise an error code	(see FtLib.h)

4.2.2 DWORD StartFtTransferAreaWithCommunication (FT_HANDLE hFt, NOTIFICATION_EVENTS* sNEvent)

USB: yes
COM: yes

This routine starts a communications thread that continually synchronizes the data from the transfer area with the device. This functionality is required for online operation. In addition, messages are sent to the interface and received messages are retrieved. The FtLib contains a message buffer that buffers messages received via SendFtMessage(). StartFtTransferAreaWithCommunication() ensures that these messages are sent to the interface within the online query.

The device must be opened beforehand via the functions OpenFtUsbDevice() or OpenFtCommDevice(). Depending upon the serial device type (indicated by OpenFtCommDevice() during opening of the interface), only the values supported by the interface are queried. If messages are to be sent over the serial interface, it must be set to the operating mode IF_COM_MESSAGE beforehand via the function SetFtDeviceCommMode().

During the call, a pointer to a NOTIFICATION_EVENTS struct can be passed to the function (NULL if non-existent). Pointers or handles for a callback routine and event and message handles can be passed in this struct. When the thread receives new data from the device (every 10ms at the latest), messages are always sent or the callback routine is always called if their pointer values or handle values do not equal NULL. The invoked callback routines may not call any FtLib functions since otherwise a deadlock may occur.

If a message is received by the interface during the online query, the address of the callback function saved in the variable “CallbackMessage” is retrieved and a pointer to the message is passed. It is then the responsibility of the programmer to buffer this message for the main program as quickly as possible in order to not delay the execution time of the online query.

Call:	FT_HANDLE hFt	device handle
	NOTIFICATION_EVENTS*	pointer to a notification struct If the passed value is NULL, no struct will be used. Functions not utilized within the struct have to be deactivated through a NULL pointer.
Return:	Error code	
	FTLIB_ERR_SUCCESS	no error
	otherwise an error code	(see FtLib.h)

4.2.3 DWORD StopFtTransferArea (FT_HANDLE hFt)

USB: yes
COM: yes
This routine stops the communications thread of the desired device.

Call:	FT_HANDLE hFt	device handle
Return:	Error code	
	FTLIB_ERR_SUCCESS	no error
	otherwise an error code	(see FtLib.h)

4.2.4 FT_TRANSFER_AREA* GetFtTransferAreaAddress (FT_HANDLE hFt)

USB: yes
COM: yes
This routine returns a pointer to the address of the transfer area.

Call:	FT_HANDLE hFt	device handle
Return:	FT_TRANSFER_AREA*	Pointer to struct if NULL, then error

4.2.5 DWORD IsFtTransferActiv (FT_HANDLE hFt)

USB: yes
COM: yes
This routine checks if a communication thread is active in the specified device.

Call:	FT_HANDLE hFt	device handle
Return:		
FTLIB_ERR_THREAD_IS_RUNNING		if a thread is running
FTLIB_ERR_THREAD_NOT_RUNNING		if no thread is running An error code can also be returned if necessary. (see FtLib.h)

4.2.6 DWORD ResetFtTransfer (FT_HANDLE hFt)

USB: yes
COM: yes
This routine deletes all outputs on the specified device as well as on all connected I/O extensions (or Intelligent Interfaces and extension modules) provided that no communication thread is running.

Call:	FT_HANDLE hFt	device handle
Return:	Error code	
	FTLIB_ERR_SUCCESS	no error
	otherwise an error code	(see FtLib.h)

4.3 Function for Message Processing

Multiple Robo interfaces can exchange messages with one another via the RS232 interface and the radio interface. The messages consist strictly of a 16-bit message ID and a 16-bit value.

The communication takes place on radio channels (frequencies). A maximum of eight radio interfaces can exchange messages on the same radio channel. Every module thereby has its own “ID” number (RoboPro: “call signal”) that can be set within the range 1..8. The PC module always has ID number “0.” Additionally, messages can also be sent to the serial port of the interface and be received by it.

Every radio channel is divided into 256 logical subchannels that are used to structure the communication. A message is thereby always distributed among all participants (“broadcasting”). Every message consists of five data bytes:

SubId	Number of the subchannel (1 byte)
Message	Message data (4 bytes)
	B1:B0: Message ID (low word)
	B3:B2: Message (high word)

An acknowledgement is not sent to the “sender” as to whether or not the message was received.

Communication is controlled by the PC module that serves as a message router.

In order to be able to use message processing (message system) in online mode, the interface query must be started with StartFtTransferAreaWithCommunication(). The system is designed so that approximately every 10ms two messages are transmitted to the interface and received messages are retrieved.

4.3.1 Serial Messages

In online mode, messages can also be sent and received via the serial interface. For this purpose, the interface operating mode in the interface must be changed with the function “SetFtDeviceCommMode().” The changed operating mode is then recognizable by the continuous illumination of the COM light-emitting diode on the interface. By pressing the port pushbutton, the operating mode is restored; naturally, this can also occur via the software. So it is possible to exchange messages between two Robo interfaces via the serial interface. The connection cable required for this (“X cable” with crossed sending and receiving paths) can be ordered via the fischertechnik Component Part Service.

4.3.2 FtLib Functions

FtLib offers the function “SendFtMessage ()” for data transfer. For sending, a physical channel can be selected via a hardware ID. In the event that a second interface or radio interface are not available, one can send the message “to oneself.” One can send the message to the serial interface or let the message router broadcast it over radio (RF) to other modules.

4.3.3 Message Reception

As soon as the interface receives a message, it invokes a callback routine and passes it a pointer to the received message. The address of the callback routine must be specified at the start of the transfer. We recommend that this callback routine copy the message into its own buffer and that it be processed by the main program at a later time in order not to “prolong” the thread routine for too long.

4.3.4 DWORD SendFtMessage (FT_HANDLE hFt, BYTE bHwId, BYTE bSubId, DWORD dwMessage, DWORD dwWaitTime DWORD)

USB: yes
COM: yes

This routine writes a message to the internal buffer. A physical channel is selected via the parameter bHwId; via the parameter bSubId, this channel can be partitioned into logical subchannels. The actual message consists of a 16-bit message ID in the low word and a 16-bit message value in the high word of the variable dwMessage.
Normally this function writes the delivered message to the internal message buffer and immediately returns. However, in the event that the internal buffer is full, a waiting time in “ms” can be specified with the parameter dwWaitTime. If the message cannot be written to the buffer in the specified time, this function will be terminated with an error message. If dwWaitTime=0, there is no wait time. If the buffer is full, this function returns immediately.

The message system requires approximately 5ms for the dissemination of a message. Since this function could be called a significant number of times, particularly within loops, the number of messages to be sent can be optimized via the parameter “dwOption.” This can prevent the same message from being sent multiple times.

If messages are also to be sent to the serial port of the interface (bHwId = MSG_HWID_SER), the operating mode IF_COM_MESSAGE must be activated with the function “SetFtDeviceCommMode()” before the start of the transfer thread.

Call:	FT_HANDLE hFt	device handle
	BYTE bHwId	Hardware ID
	MSG_HWID_SELF (0x00):	copied directly into its own receive buffer
	MSG_HWID_SER (0x01):	Sent via interface RS232
	MSG_HWID_RF (0x02):	Via radio only to other modules
	MSG_HWID_RF_SELF (0x03):	Via radio also to itself
	BYTE bSubId	Logical channels, IDs 0…219 are allowed. The values 220 to 255 are reserved for internal tasks.
	DWORD dwMessage	Low word: 16-bit message ID High word: 16-bit message
	DWORD dwWaitTime	In the event that the internal buffer is full, this parameter can specify (in ms) how long the wait time is until the function returns.
	DWORD dwOption	Send options
	MSG_SEND_NORMAL (0):	The message is written directly to the sending buffer.
	MSG_SEND_OTHER_THAN_LAST (1):	The message will not be sent if an identical message (bHwId, bSubId, dwMessage) exists at the end of the buffer. If the buffer is empty or if another message exists at the end of the buffer, the message will be sent.
	MSG_SEND_IF_NOT_PRESENT (2): (bHwId, bSubId, dwMessage)	The message will not be sent if an identical message exists anywhere in the buffer. If the buffer is empty, the message will be sent.

Return:	Error code	
	FTLIB_ERR_SUCCESS	no error
	otherwise an error code	(see FtLib.h)
	current error codes:	
	FTLIB_ERR_MSG_BUFFER_FULL_TIMEOUT	Buffer is full, message could not be sent in the specified time

4.3.5 DWORD ClearFtMessageBuffer (FT_HANDLE hFt)

USB: yes
COM: yes
This routine deletes the existing messages for the specified device that still exist in the internal message buffer.

Call:	FT_HANDLE hFt	device handle
Return:	Error code	
	FTLIB_ERR_SUCCESS	no error
	otherwise an error code	(see FtLib.h)

4.4 Function for Data Downloading / Program Control

4.4.1 DWORD GetFtMemoryLayout (FT_HANDLE hFt, BYTE * pbArray, DWORD dwSize)

USB: yes
COM: yes
This routine writes the current memory layout of the connected device in the passed array. For the readout, no communication thread can be running.

Prior to a program download, the user program must be linked to these addresses.

Note: We reserve the right to change the memory layout with future firmware versions if necessary.

Call:	FT_HANDLE hFt	device handle
	BYTE * pbArray	Pointer to a byte array
	DWORD dwSize	Size of the memory region in bytes
Return:	Error code	
	FTLIB_ERR_SUCCESS	no error
	otherwise an error code	(see FtLib.h)
	current error codes:	
	FTLIB_ERR_INVALID_PARAM	Pointer *pbArray is NULL

Memory layout design

FLASH-1 Region Start – end address (20-bit: a bb cc – d ee ff)

Byte 1:	[0]	cc	(the start address)
Byte 2:	[1]	bb	(the start address)
Byte 3:	[2]	0a	(the start address)
Byte 4:	[3]	ff	(the end address)
Byte 5:	[4]	ee	(the end address)
Byte 6:	[5]	0d	(the end address)

FLASH-2 Region Start – end address (20-bit: a bb cc – d ee ff)

Byte 7:	[6]	cc	(the start address)
Byte 8:	[7]	bb	(the start address)
Byte 9:	[8]	0a	(the start address)
Byte 10:	[9]	ff	(the end address)
Byte 11:	[10]	ee	(the end address)
Byte 12:	[11]	0d	(the end address)

RAM Region Start – end address (20-bit: a bb cc – d ee ff)
(usable for programs and variables, region is completely available to the program after the start of a flash program)

Byte 13:	[12]	cc	(the start address)
Byte 14:	[13]	bb	(the start address)
Byte 15:	[14]	0a	(the start address)
Byte 16:	[15]	ff	(the end address)
Byte 17:	[16]	ee	(the end address)
Byte 18:	[17]	0d	(the end address)

Internal RAM1 Start – end address (20-bit: a bb cc – d ee ff)
(usable for variables and stacks)

Byte 19:	[18]	cc	(the start address)
Byte 20:	[19]	bb	(the start address)
Byte 21:	[20]	0a	(the start address)
Byte 22:	[21]	ff	(the end address)
Byte 23:	[22]	ee	(the end address)
Byte 24:	[23]	0d	(the end address)

Internal RAM2 Start – end address (20-bit: a bb cc – d ee ff)
(usable for variables and stacks, bitwise addressing possible)

Byte 25:	[24]	cc	(the start address)
Byte 26:	[25]	bb	(the start address)
Byte 27:	[26]	0a	(the start address)
Byte 28:	[27]	ff	(the end address)
Byte 29:	[28]	ee	(the end address)
Byte 30:	[29]	0d	(the end address)

4.4.2 DWORD DownloadFtProgram (FT_HANDLE hFt, DWORD dwMemBlock, BYTE* pbArray, DWORD dwSize, DWORD dwParameter, BYTE *pbName, DWORD dwNameLen)

USB: yes
COM: yes
This routine writes the passed program to the desired memory region (MemBlock) of the interface. The program lasts a few seconds, depending on memory size.
It can be specified in the variable dwParameter whether or not the program should be automatically started during activation. However, this only works with programs for the “Flash1” memory region. With all other memory regions the specification will be ignored.

During the call, a pointer can be passed to a string and its length (in number of bytes; thus, “\0” at the end of the “string” is not required). This string will be saved as the program name in the same memory block. Since a check of the string for “\0” does not occur, binary numbers can also be saved. In the event that the pointer “pbName” is “NULL” during invocation or if dwNameLen has a value of “0,” no name will be saved. The function “GetFtProgramName()” then returns a string that contains a space character (0x20) in the first byte and that contains “0” for the remaining characters.

Import note:
If you use this function, you must know “what” you are doing! When the program is started, no more security controls occur! Thus, it is possible to generate hardware damage through reprogramming the hardware register in the processor!!! The manufacturer undertakes no guarantee services if an interface defect occurs through incorrect programs!

If the program is provided with the label “Autostart,” the Autostart can be skipped by pressing the “Prog” pushbutton during activation. The pushbutton must be pressed at the latest during the LED test. As soon as the LEDs of the USB / COM interfaces display normal operation, the pushbutton may be released.

Note: Erasing the flash memory can take up to 15 seconds for every block! This must be taken into consideration with timeout monitoring. In addition, the time for the transfer and storage of the data in the interface must be added.
Serial default values:
1 k data transfer in the flash < 1 second
128k data transfer in the flash circa 50 seconds

USB default values:
1 k data transfer in the flash < 1 second
128k data transfer in the flash circa 8 seconds

Call: DWORD DownloadFtProgram (FT_HANDLE hFt, DWORD dwMemBlock,
BYTE* pbArray, DWORD dwSize,
DWORD dwParameter, BYTE *pbName,
DWORD dwNameLen)

FT_HANDLE hFt		device handle
	DWORD dwMemBlock	Number of the memory region 0 = Flash1 1 = Flash2 2 = RAM
	BYTE * pbArray	Pointer to a byte array
	DWORD dwSize	Number of the byte to be programmed
	DWORD dwParameter	Parameter: 0 = normal 1 = Autostart of the program
	BYTE * pbName	Pointer to a byte array with the program name (max. 80 characters) "NULL" = do not store name
	DWORD dwNameLen	Length of the program name in bytes 0 = do not store name
Return:	Error code	
	FTLIB_ERR_SUCCESS	no error
	otherwise an error code	(see FtLib.h)
	current error codes:	
	FTLIB_ERR_SUCCESS	Program saved
	FTLIB_INFO_PROGRAM_0_IS_RUNNING	Flash-1 program is running
	FTLIB_INFO_PROGRAM_1_IS_RUNNING	Flash-2 program is running
	FTLIB_INFO_PROGRAM_2_IS_RUNNING	RAM program is running

4.4.3 DWORD StartFtProgram (FT_HANDLE hFt, DWORD dwMemBlock)

USB: yes
COM: yes
This routine starts the desired program (0..2).

Call:	FT_HANDLE hFt	device handle
	DWORD dwMemBlock	Number of the memory region whose program is to be started
Return:	Error code	
	FTLIB_ERR_SUCCESS	no error
	otherwise an error code	(see FtLib.h)
	current error codes:	
	FTLIB_ERR_SUCCESS	Program started
	FTLIB_INFO_PROGRAM_0_IS_RUNNING	Flash-1 program is running
	FTLIB_INFO_PROGRAM_1_IS_RUNNING	Flash-2 program is running
	FTLIB_INFO_PROGRAM_2_IS_RUNNING	RAM program is running
	FTLIB_ERR_IF_NO_PROGRAM	No program saved

4.4.4 DWORD StopFtProgram (FT_HANDLE hFt)

USB: yes
COM: yes
This routine stops the currently running program.

Call:	FT_HANDLE hFt	device handle
Return:	Error code	
	FTLIB_ERR_SUCCESS	no error
	otherwise an error code	(see FtLib.h)
	current error codes:	
	FTLIB_ERR_SUCCESS	Program was terminated
	FTLIB_ERR_IF_NO_PROGRAM_IS_RUNNING	

4.4.5 DWORD DeleteFtProgram (FT_HANDLE hFt, DWORD dwMemBlock)

USB: yes
COM: yes
This routine deletes the specified program.
Note: Erasing the flash memory can take up to 15 seconds for every block! This must be taken into consideration with timeout monitoring.
No check occurs as to whether the block contains a program. It will be deleted in either case!

Call:	FT_HANDLE hFt	device handle
	DWORD dwMemBlock	Number of the memory region whose program is to be deleted
Return:	Error code	
	FTLIB_ERR_SUCCESS	no error
	otherwise an error code	(see FtLib.h)
	current error codes:	
	FTLIB_ERR_SUCCESS	Program was deleted

4.4.6 DWORD SetFtProgramActiv (FT_HANDLE hFt, DWORD dwMemBlock)

USB: yes
COM: yes
This routine activates the specified program. Thus, the program LEDs are activated without starting the program. It is also possible to deactivate the program LEDs with this.

Call:	FT_HANDLE hFt	device handle
	DWORD dwMemBlock	Number of the memory region that is to be “activated.” The LEDs are turned off with “-1.”
Return:	Error code	
	FTLIB_ERR_SUCCESS	no error
	otherwise an error code	(see FtLib.h)
	current error codes:	
	FTLIB_ERR_SUCCESS	Program started
	FTLIB_INFO_PROGRAM_0_IS_RUNNING	Flash-1 program is running
	FTLIB_INFO_PROGRAM_1_IS_RUNNING	Flash-2 program is running
	FTLIB_INFO_PROGRAM_2_IS_RUNNING	RAM program is running
	FTLIB_ERR_IF_NO_PROGRAM	No program saved
	FTLIB_ERR_DOWNLOAD_CRC	CRC error
	FTLIB_ERR_POWER_TOO_LOW	Voltage on IF too low

4.4.7 DWORD GetFtProgramName (FT_HANDLE hFt, DWORD dwMemBlock, DWORD dwSize, LPVOID pName)

USB: yes
COM: yes
This routine reads the name of the specified program. A pointer to a correspondingly large data block into which the character string will be copied must be passed with the call.

Tip:
With this function it is also possible to check if a program is saved in a specified memory region. For the save, it does not matter whether or not a name was given. If no program is saved, the error code FTLIB_ERR_IF_NO_PROGRAM is returned.

Call:	FT_HANDLE hFt	device handle
	DWORD dwMemBlock	Number of the memory region
	DWORD dwSize	Size of the memory (at least 80 characters)
	LPCSTR pName*	Pointer to the memory region for Prog.Name
Return:	Error code	
	FTLIB_ERR_SUCCESS	no error
	otherwise an error code	(see FtLib.h)
	current error codes:	
	FTLIB_ERR_SUCCESS	Program started
	FTLIB_INFO_PROGRAM_0_IS_RUNNING	Flash-1 program is running
	FTLIB_INFO_PROGRAM_1_IS_RUNNING	Flash-2 program is running
	FTLIB_INFO_PROGRAM_2_IS_RUNNING	RAM program is running
	FTLIB_ERR_IF_NO_PROGRAM	No program saved

4.4.8 DWORD GetFtMemoryData (FT_HANDLE hFt, BYTE * pbArray, DWORD dwSize, DWORD dwAddress)

USB: yes
COM: yes

This routine reads 64 data bytes from the interface memory beginning with the address specified in dwAddress and writes the data to the array to which pbArray points. The array must be at least 64 bytes in size (dwSize). Only the memory regions from 0x00400 to 0xDFFFF can be queried. For memory regions outside of this region, “00” is returned.

Note: This program also works with a running program (“active” mode).
The reading of data via the serial interface is only possible if no communication thread is running. Please bear in mind that the communication region in the interface beginning with address 0x400 is only used for running programs (active mode) in the interface. In online mode (communication thread), this region is deactivated.

Call:	FT_HANDLE hFt	device handle
	BYTE * pbArray	Pointer to a byte array
	DWORD dwSize	Size of the byte array
	DWORD dwAddress	Address in the interface

Return:	Error code	
	FTLIB_ERR_SUCCESS	no error
	otherwise an error code	(see FtLib.h)
	current error codes:	
	FTLIB_ERR_SUCCESS	Data has been read
	FTLIB_ERR_THREAD_IS_RUNNING	Only COM: communication thread is running
	FTLIB_ERR_INVALID_PARAM	Pointer pbArray is NULL

4.4.9 DWORD WriteFtMemoryData (FT_HANDLE hFt, DWORD dwData, DWORD dwAddress)

USB: yes
COM: yes

This routine writes a data byte in the memory of the interface beginning with the address specified in dwAddress. One can only write to the memory regions from 0x00400 to 0x7FFFF. After the writing, a verify is executed.

Note: This program also works with a running program (“active” mode).
The writing of data via the serial interface is only possible if no communication thread is running. Please bear in mind that the communication region in the interface beginning with address 0x400 is only used for running programs (active mode) in the interface. In online mode (communications thread), this region is deactivated.

Call:	FT_HANDLE hFt	device handle
	DWORD dwData	Data byte to be written to
	DWORD dwAddress	Address in the interface
Return:	Error code	
	FTLIB_ERR_SUCCESS	no error
	otherwise an error code	(see FtLib.h)

5 Sequence of USB Functionality for Online Communication

With application start:
InitFtLib();

Determine which devices are connected (create list):
InitFtUsbDeviceList();

Determine the number of active devices:
GetNumFtUsbDevice();

Get a handle to a desired device:
GetFtUsbDeviceHandle(UCHAR ucDevNr) ;
ucDevNr = 0 … GetNumFtUsbDevice()

Determine “which” device (type of the device) is connected:
GetFtDeviceTyp(FT_HANDLE hFt);

Open the data connection to the device:
OpenFtUsbDevice(FT_HANDLE hFt);

Begin the communication (device <-> transfer area):
StartFtTransferArea(FT_HANDLE hFt, NOTIFICATION_EVENTS* sNEvent);

Query the pointer to the transfer area:
GetFtTransferAreaAddress(FT_HANDLE hFt);

.
(your own program)
.

Stop the communication:
StopFtTransferArea(FT_HANDLE hFt);

Close the data connection to the device:
CloseFtDevice (FT_HANDLE hFt) or CloseAllFtUsbDevices()

With application termination:
CloseFtLib();

6 Transfer Area

The querying and routing of the outputs occurs through a “transfer area” (communication storage region). This region is synchronized with the interface every 10ms after the start of the communication thread (by means of StartFtTransferArea). It does not matter if the interface is connected to the PC via USB, serially, or by radio. It also does not matter if it involves an intelligent interface, a Robo interface, or an IO extension. The library recognizes the product in question via the device handle and updates the existing inputs / outputs of the respective interface type at the latest every 10ms. In the event that multiple interfaces are connected to the computer, a separate thread for every connected product can be started. The design of the transfer area can be found in this chapter’s extension.

Note:
With an Intelligent Interface, the data of the extension module is saved to memory locations of extension module (expansion module) “1.”

6.1 Memory Region Design

After the transfer thread has been started by means of StartFtTransferArea(), the base address for the memory region can be queried with the function GetFtTransferAreaAddress().

6.1.1 Memory Layout of the Communication Region

Digital inputs of the base module

```
              | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
Base+0x00: | E8 | E7 | E6 | E5 | E4 | E3 | E2 | E1 |
```

Digital inputs of expansion modules 1-3

```
Base+0x01: | E16| E15| E14| E13| E12| E11| E10| E9 |
Base+0x02: | E24| E23| E22| E21| E20| E19| E18| E17|
Base+0x03: | E32| E31| E30| E29| E28| E27| E26| E25|
```

Reserved (8 bytes)

```
Base+0x04..0x0B
```

Reserved (1 byte)

```
Base+0x0C
```

Reserved (1 byte)

```
Base+0x0D
```

Special inputs from IR sender (motor 1-3 left/right + "1)))" and "2)))")

```
Base+0x0E: | 0 | 0 | 0 | C | T | T | T | T
           C = Code:  0 = Code 1 activated

                              1 = Code 2 activated

             TTTT = pushbutton number 0..11

Pushbutton arrangement on the sender:
    1        8
   2     7
  3          10
  4      9
  5        11
  6

Pushbutton  1 = M3R

Pushbutton  2 = M3L

Pushbutton  3 = Rate M1

Pushbutton  4 = Rate M2

Pushbutton  5 = Rate M3

Pushbutton  6 = Code 2

Pushbutton  7 = M1BW

Pushbutton  8 = M1FW

Pushbutton  9 = M2L

Pushbutton 10 = M2R

Pushbutton 11 = Code 1
```

Reserved (1 byte)

```
Base+0x0F
```

Analog inputs of the base module (4x 16 bit, value range 0..1023, L:H format)

```
Base+0x10..0x11: AX (Master Module)

Base+0x12..0x13: AY (Master Module)

Base+0x14..0x15: A1 (Master Module)

Base+0x16..0x17: A2 (Master Module)

Base+0x18..0x19: AZ (Master Module, from SLAVE Module BUS)

Base+0x1A..0x1B: AV (Master Module supply voltage)

                      in 10mV increments (* 0.01 = Volt)

Base+0x1C..0x1D: D1 (gap sensor 1)

Base+0x1E..0x1F: D2 (gap sensor 2)
```

Analog inputs of extension modules 1-3

Base+0x20..0x21: AX (extension 1 module)

Base+0x22..0x23: AX (extension 2 module)

Base+0x24..0x25: AX (extension 3 module)

Reserved (4 bytes)

Base+0x26..0x27: DS1

Base+0x28..0x29: DS2

Reserved (2 bytes)

Base+0x2A..0x2B

Reserved (4 bytes)

Base+0x2C..0x2F

16-bit timer 1..6

Base+0x30..0x31: Timer 1 ms increment

Base+0x32..0x33: Timer 10 ms increment

Base+0x34..0x35: Timer 100 ms increment

Base+0x36..0x37: Timer 1s increment

Base+0x38..0x39: Timer 10s increment

Base+0x3A..0x3B: Timer 1 min increment

Reserved (4 bytes)

Base+0x3C..0x3F

Outputs of the base module (polarity)

```
              | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |

Base+0x40: |M4B |M4A |M3B |M3A |M2B |M2A |M1B |M1A |
(do not forget: set bits to Base+0xE1!)
```

Outputs of expansion modules 1-3 (polarity)

```
Base+0x41: |M8B |M8A |M7B |M7A |M6B |M6A |M5B |M5A |

Base+0x42: |M12B|M12A|M11B|M11A|M10B|M10A|M9B |M9A |

Base+0x43: |M16B|M16A|M15B|M15A|M14B|M14A|M13B|M13A|
```

Reserved (4 bytes)

Base+0x44..0x47

Outputs of the base module (activate energy-saving mode)

```
              | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |

Base+0x48: | 0  | 0  | 0  | 0  | M4 | M3 | M2 | M1 |

     INIT  | 0  | 0  | 0  | 0  | 1  | 1  | 1  | 1  |

           1 = energy-saving mode activated
           meaning: if both outputs of a motor are at "0,"
           the output in the interface will be "turned off"
           so that no current flows.
```

Outputs of expansion modules 1-3 (deactivate energy-saving mode)

```
Base+0x49: | 0  | 0  | 0  | 0  | M8 | M7 | M6 | M5 |

Base+0x4A: | 0  | 0  | 0  | 0  | M12| M11| M10| M9 |

Base+0x4B: | 0  | 0  | 0  | 0  | M16| M15| M14| M13|

     INIT  | 0  | 0  | 0  | 0  | 1  | 1  | 1  | 1  |

                1 = energy-saving mode activated
```

Reserved (4 bytes)

Base+0x4C..0x4F

Outputs of the base module (PWM values, value range 0…7)

Base+0x50: M1A

Base+0x51: M1B

Base+0x52: M2A

Base+0x53: M2B

Base+0x54: M3A

Base+0x55: M3B

Base+0x56: M4A

Base+0x57: M4B

Outputs of the IO-Ext1 module (PWM values, value range 0…7)

Base+0x58: M5A

Base+0x59: M5B

Base+0x5A: M6A

Base+0x5B: M6B

Base+0x5C: M7A

Base+0x5D: M7B

Base+0x5E: M8A

Base+0x5F: M8B

Outputs of the IO-Ext2 module (PWM values, value range 0…7)

Base+0x60: M9A

Base+0x61: M9B

Base+0x62: M10A

Base+0x63: M10B

Base+0x64: M11A

Base+0x65: M11B

Base+0x66: M12A

Base+0x67: M12B

Outputs of the IO-Ext3 module (PWM values, value range 0…7)

Base+0x68: M13A

Base+0x69: M13B

Base+0x6A: M14A

Base+0x6B: M14B

Base+0x6C: M15A

Base+0x6D: M15B

Base+0x6E: M16A

Base+0x6F: M16B

Reserved (32 bytes)

Base+0x70..0x8F

Analog inputs of I/O extension 1..3 (update time 20ms)
(these are located on the right 10-pin pin strip, pin 10, series resistance of 220..470 Ohm recommended)

Base+0x90..0x91: A1 (I/O extension 1 module)

Base+0x92..0x93: A1 (I/O extension 2 module)

Base+0x94..0x95: A1 (I/O extension 3 module)

Base+0x96..0x97: AV (I/O extension 1 module supply voltage)
in 10mV increments (* 0.01 = volt)

Base+0x98..0x99: AV (I/O extension 2 module supply voltage)
in 10mV increments (* 0.01 = volt)

Base+0x9A..0x9B: AV (I/O extension 3 module supply voltage)
in 10mV increments (* 0.01 = volt)

Reserved (4 bytes)

Base+0x9C..0x9F

Resistance values of analog inputs AX / AY

Base+0xA0..0xA1: AX (interface) resistor value (0..5662 Ohm)

Base+0xA2..0xA3: AY (interface) resistor value (0..5662 Ohm)

Base+0xA4..0xA5: AX (I/O extension 1 module) resistor value (0..5662)

Base+0xA6..0xA7: AX (I/O extension 2 module) resistor value (0..5662)

Base+0xA8..0xA9: AX (I/O extension 3 module) resistor value (0..5662)

Reserved (54 bytes)

Base+0xAA..0xDF

Reserved (1 byte)

Base+0xE0

Options (RW)

```
Base+0xE1: |    |    |    |    |    |    | UA2| UA1|

     INIT  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

               UA1 = 1: Update outputs / PWM every 10ms
               UA2 = 1: Update outputs / PWM one time
Note: Only if UA1 or UA2 are at "1" will the
         preset motor states on the outputs be connected
         (only with download programs).
```

Windows only:

```
Base+0xE2  Windows Communication Thread <-> Windows application
           1 = thread is running, 0=thread is not running
           (byte is used only within the Windows transfer area!)
```

Interface only: Reserve

```
Base+0xE2  byte not used in the interface
```

Reserved

```
Base+0xE3..0x0xE5
```

Number of connected I/O extension modules on the bus (S2..S0 = 0..3)

```
Base+0xE6  |    |    |    |    |    | S2 | S1 | S0 |
```

Reserved (1 byte)

```
Base+0xE7
```

Reserved (1 byte)

```
Base+0xE8
```

Reserved

```
Base+0XE9..0x0xEF
```

Change byte digital inputs

```
Base+0xF0: (1=inputs to the master or IO-Ext1..3 have changed)
            Windows: use InterlockedExchange() function with 0 for
            reading and writing!
```

Change byte analog inputs

Master: AX, AY, A1, A2, AV, AZ, D1, D2 or I/O extension 1..3 AX, AY, AV have changed

```
Base+0xF1: (1=analog inputs to the master or IO-Ext1..3 have changed)
            Windows: use InterlockedExchange() function with 0 for
            reading and writing!
```

Change byte Ir inputs

```
Base+0xF2: (1=change to the IR sensor input)
            Windows: use InterlockedExchange() function with 0 for
            reading and writing!
```

Reserve

```
Base+0XF3..0x0xFF
```

Here follow the (digital) inputs again per input stored in a 16-bit variable. If the input is associated with “+,” this is assessed as “1.” An input that is open or connected with “ground” is assessed as “0.”

```
Base+0x100..0x101  input  1 (Master Module)
Base+0x102..0x103  input  2 (Master Module)
Base+0x104..0x105  input  3 (Master Module)
Base+0x106..0x107  input  4 (Master Module)
Base+0x108..0x109  input  5 (Master Module)
Base+0x10A..0x10B  input  6 (Master Module)
Base+0x10C..0x10D  input  7 (Master Module)
Base+0x10E..0x10F  input  8 (Master Module)
Base+0x110..0x111  input  9 (Slave1 Module)
Base+0x112..0x113  input  10 (Slave1 Module)
Base+0x114..0x115  input  11 (Slave1 Module)
Base+0x116..0x117  input  12 (Slave1 Module)
Base+0x118..0x119  input  13 (Slave1 Module)
Base+0x11A..0x11B  input  14 (Slave1 Module)
Base+0x11C..0x11D  input  15 (Slave1 Module)
Base+0x11E..0x11F  input  16 (Slave1 Module)
Base+0x120..0x121  input  17 (Slave2 Module)
Base+0x122..0x123  input  18 (Slave2 Module)
Base+0x124..0x125  input  19 (Slave2 Module)
Base+0x126..0x127  input  20 (Slave2 Module)
Base+0x128..0x129  input  21 (Slave2 Module)
Base+0x12A..0x12B  input  22 (Slave2 Module)
Base+0x12C..0x12D  input  23 (Slave2 Module)
Base+0x12E..0x12F  input  24 (Slave2 Module)
Base+0x130..0x131  input  25 (Slave3 Module)
Base+0x132..0x133  input  26 (Slave3 Module)
Base+0x134..0x135  input  27 (Slave3 Module)
Base+0x136..0x137  input  28 (Slave3 Module)
Base+0x138..0x139  input  29 (Slave3 Module)
Base+0x13A..0x13B  input  30 (Slave3 Module)
Base+0x13C..0x13D  input  31 (Slave3 Module)
Base+0x13E..0x13F  input  32 (Slave3 Module)
Base+0x140..0x141  gap sensor D1 (Master Module)
Base+0x142..0x143  gap sensor D2 (Master Module)
```

Reserved (12 bytes)

```
Base+0x140..0x14F
```

The IR pushbuttons are presented as individual inputs in a 16-bit variable per pushbutton (1=pushbutton pressed). A pressed pushbutton is indicated with a “1” both in the “undecoded” region as well as in the “decoded” region.

```
Base+0x150..0x151  IR pushbutton  1 (M3R)
Base+0x152..0x153  IR pushbutton  2 (M3L)
Base+0x154..0x155  IR pushbutton  3 (M1)
Base+0x156..0x157  IR pushbutton  4 (M2)
Base+0x158..0x159  IR pushbutton  5 (M3)
Base+0x15A..0x15B  IR pushbutton  6 (code2)
Base+0x15C..0x15D  IR pushbutton  7 (M1BW)
Base+0x15E..0x15F  IR pushbutton  8 (M1FW)
Base+0x160..0x161  IR pushbutton  9 (M2LE)
Base+0x162..0x163  IR pushbutton  10 (M2RI)
Base+0x164..0x165  IR pushbutton  11 (code1)
```

Reserved (10 bytes)

```
Base+0x166..0x16F
```

```
Base+0x170..0x171  IR pushbutton  1 (M3R)    code1
Base+0x172..0x173  IR pushbutton  2 (M3L)    code1
Base+0x174..0x175  IR pushbutton  3 (M1)    code1
Base+0x176..0x177  IR pushbutton  4 (M2)    code1
Base+0x178..0x179  IR pushbutton  5 (M3)    code1
Base+0x17A..0x17B  reserved
Base+0x17C..0x17D  IR pushbutton  7 (M1BW)   code1
Base+0x17E..0x17F  IR pushbutton  8 (M1FW)   code1
Base+0x180..0x181  IR pushbutton  9 (M2LE)   code1
Base+0x182..0x183  IR pushbutton  10 (M2RI)   code1
Base+0x184..0x185  reserved
```

Reserved (10 bytes)

```
Base+0x166..0x18F
```

```
Base+0x190..0x191  IR pushbutton  1 (M3R)    code2
Base+0x192..0x193  IR pushbutton  2 (M3L)    code2
Base+0x194..0x195  IR pushbutton  3 (M1)    code2
Base+0x196..0x197  IR pushbutton  4 (M2)    code2
Base+0x198..0x199  IR pushbutton  5 (M3)    code2
Base+0x19A..0x19B  reserved
Base+0x19C..0x19D  IR pushbutton  7 (M1BW)   code2
Base+0x19E..0x19F  IR pushbutton  8 (M1FW)   code2
Base+0x1A0..0x1A1  IR pushbutton  9 (M2LE)   code2
Base+0x1A2..0x1A3  IR pushbutton  10 (M2RI)   code2
Base+0x1A4..0x1A5  reserved
```

Reserved (10 bytes)

```
Base+0x1A6..0x1AF
```

RF status (only RF data link USB module)

```
Base+0x1B0..0x1B1:    0 = RF connection in order
                      1 = (0x1B4..0x1B5) > 25, bad connection
Base+0x1B2..0x1B3:    Reception quality, with <40 bad connection
                      (8-bit value)
Base+0x1B4..0x1B5:    Error count with RF online mode; is
                      increased with faulty connection and set
                      to "0" if the connection is in order.
```

Reserved

```
Base+0x1B6..0x1FF
```

6.1.2 Digital Inputs E1-E32

The bits for the digital inputs are set to “0” with an open input and set to “1” with an input connected with “+.” Unavailable inputs (missing expansion modules) are set to “0.” Additionally, all 32 inputs are again stored in a 16-bit variable per input (“1”=input active) starting with base+0x100.

6.1.3 Special Inputs

The 11 pushbuttons of the IR radio control are special inputs. The number on the pressed pushbutton on the IR sender as well as the information as to whether code “1” or code “2” is activated is stored in location base+0x0E. Additionally, all pushbuttons are again stored in 16-bit variables (the same as for the digital inputs).

6.1.4 Analog Inputs

The analog inputs are stored as 16-bit values with a value range of 0…1023.

6.1.5 16-bit Timer

The six 16-bit timers with increments of 1ms, 10ms, 100ms, 1s, 10s, and 60s are used for specific timeout variables. No fixed relationship exists between the individual timer values, i.e., the 10ms value is not 10 times the 1ms value, for example.

6.1.6 Outputs

The outputs are controlled via a polarity bit, an energy-saving bit, and a PWM value byte. The PWM value and the polarity bit are indicated per individual output. The energy-saving bit is indicated per output pair. If the polarity bit is “0,” the output is set to ground; if the polarity bit is “1,” the output is set to servicing (9V). If the energy-saving bit is “1,” an output pair is connected with high resistance after a delay (1 sec.) if both accompanying polarity bits are set to “0.” If the energy-saving bit is set to “0,” the associated output pair is not connected with high resistance. The pulse width of the output is set at 8 levels via the PWM byte, which has a value range of 0…7 (e.g., in 12.5% steps between 12.5% and 100%).

Note for download programs:
The output settings are copied to a separate data area by the firmware every 10ms if the bit “UpdateAusgaenge” (UA1) is set to BASE+0xE1. If UA2 is set instead of UA1, then the outputs are written only once with the next 10ms interrupt. After the outputs are set, the UA2 bit is deleted.

After the initialization of the interface (activation), all energy-saving bits are set to “1.” This functionality is activated by default.

6.1.7 Operating Mode, Installed Enhancements

Starting with base+0x0E, information can again be found about the operating state as well as the type of the installed module (number of I/O extensions, radio module, etc.).

7 Revision

Version 0.56: - Complete revision
- New: SendFtMessage()
ClearFtMessageBuffer()
StartDtTransferAreaWithCommunication()
SetFtDeviceCommMode()

Version 0.58: - Revision of InitFtUsbDeviceList()

Version 0.60: - Renamed GetAnzFtUsbDevice() to GetNumFtUsbDevice()
- Renamed ClearFtMessagePuffer() to ClearFtMessageBuffer()

Version 1.61a: - Renamed FtLib from 0.60 to 1.61a
- Support for Interface-Firmware 01.66.00.03

Version 1.70a: - Revision of SetFtDistanceSensorMode()
- Support for Interface-Firmware 01.75.00.04