Programming the ROBO TX Controller

Part 3: C compiler for robotics programs

"C_Compiler_RoboTXC"
package and documentation

V1.2 dated 4/25/2012

References:

Description	Version	Date
ROBO TX Controller firmware	1.30	3/19/2012
GNU ARM C compiler (GCC)	4.4.1	7/22/2009

MSC Vertriebs GmbH
Design Center Aachen
Pascalstr. 21
52076 Aachen, Germany

Contents

1 Introduction

This document describes the direct programming of the fischertechnik ROBO TX Controller in the form of a locally executed program written in C. This is an alternative to programs that were developed in ROBO Pro and that are executed as ROBO Pro machine code compilers in download mode. Programming in C also offers the possibility of working without ROBO Pro.

The ROBO TX Controller has a 32-bit ARM9 Atmel microprocessor with the designation AT91SAM9260 and an operating frequency of approximately 200 MHz. A suitable development environment for this CPU is the YAGARTO toolchain (see also www.yagarto.de). This includes the GNU ARM C compiler GCC, which can generate the executable code for this CPU, and the Make Tool. These are all available for free. The C compiler and YAGARTO toolchain versions included in this package do not necessarily represent the latest version, but rather a verified version that has been approved by us.

You can load your own programs using the standard PC interfaces of the ROBO TX Controller, USB and Bluetooth, just as you would from ROBO Pro.

There are also a number of C source code demo programs provided that already offer programmers with basic C skills an introduction into direct programming of the ROBO TX Controller. Experts, on the other hand, can further enhance and refine the system from the tools side and from the complexity of the C programs.

1.1 System requirements

C programming of the ROBO TX Controller, insofar as it is described in this document, requires a Microsoft Windows operating system (Windows 2000 or later).

A full-speed USB host port on the PC and a Bluetooth interface (integrated in the PC or via an external adapter, such as a Bluetooth USB stick) are also required.

1.2 Terminology

ROBO TX Controller: the name of the fischertechnik robotics controller itself. It is also abbreviated as **TX-C**.

Firmware: the basic software system that runs on the ROBO TX Controller and functions as the operating system, device driver, logger and boot loader.

Robotics program: the software component that can run as a loadable, executable robotics application on the ROBO TX Controller. An existing executable firmware on the TX-C is of course required.

IMPORTANT NOTE: the C compiler package is always intended for a specific version of the ROBO TX Controller firmware. The supplied examples and source files may not be compatible with older or newer versions of the firmware. Therefore, please make sure that you always use the firmware version on the ROBO TX Controller that is referenced on the cover sheet.

2 Software structure on the ROBO TX Controller

The following describes exactly what the firmware does on the ROBO TX Controller, how it is structured and other useful information on how to program your own robotics programs in C.

2.1 The 4NetOS® operating system

The firmware that runs on the ROBO TX Controller forms the basis of the 4NetOS® multi-tasking, real-time operating system. Different system tasks run in parallel to the following (generally cyclically executed) tasks:

- Reading inputs and setting outputs
- Synchronizing transfer area data with the PC or with the slave boards
- Operating the serial interfaces (incl. USB and Bluetooth)
- Managing the file system
- Reading out push-button switches and displaying the output
- Executing a local robotics program, if present and started, in time division multiplexing operation with other tasks
- Other internal tasks

These different tasks are always active and act like finite state machines, i.e. they respond to events. This could be, for instance, data incoming via one of the communication interfaces or elapsing timer events.

The basic operation of the operating system as it relates to I/O communication is as follows: startup is in "local" mode. As long as the configuration option "Load-after-power-on" is set to "Yes", the system checks if an executable robotics program is configured and present on the flash disk. If it is, and the "Start-after-power-on" configuration option is set to "YES", this program is also immediately executed. If it is not, there will be no I/O communication initially. If I/O commands are received by one of the communication interfaces (USB or Bluetooth), the system will dynamically switch to online mode and the I/O commands will be immediately interpreted and executed. When I/O commands are received while a local robotics program is being executed, the local program will be interrupted. The program will then have to be restarted using a command.

File system and program memory

The ROBO TX Controller has a file system with two "disks": the RAM disk and the flash disk. Even if physically they are only memory chips and not hard disks, the file structures on them are still used in such a way as to justify use of the term "disk". The flash disk represents a non-volatile "disk" that can store files regardless of any on/off cycles, while the RAM disk is a volatile "disk" that "forgets" its contents every time it is shut down.

Data on the flash disk and on the RAM disk can be accessed remotely from the PC, making it possible to transfer files easily from the PC to the ROBO TX Controller. Even firmware can be updated this way.

Robotics programs, whether created using ROBO Pro or the C compiler, are first always stored as files. Due to the presence of a file system, a large number of programs can be stored on the interface. They are usually stored on the flash disk. Only when the programs are to be dynamically loaded and not permanently stored are they stored temporarily as a file on the RAM disk.

To execute programs, the programs first have to be loaded into what is called program memory. This is understood to be a reserved area in the RAM where the contents of

program files are copied in order to be executed from there. The following figure shows the interface memory structure:

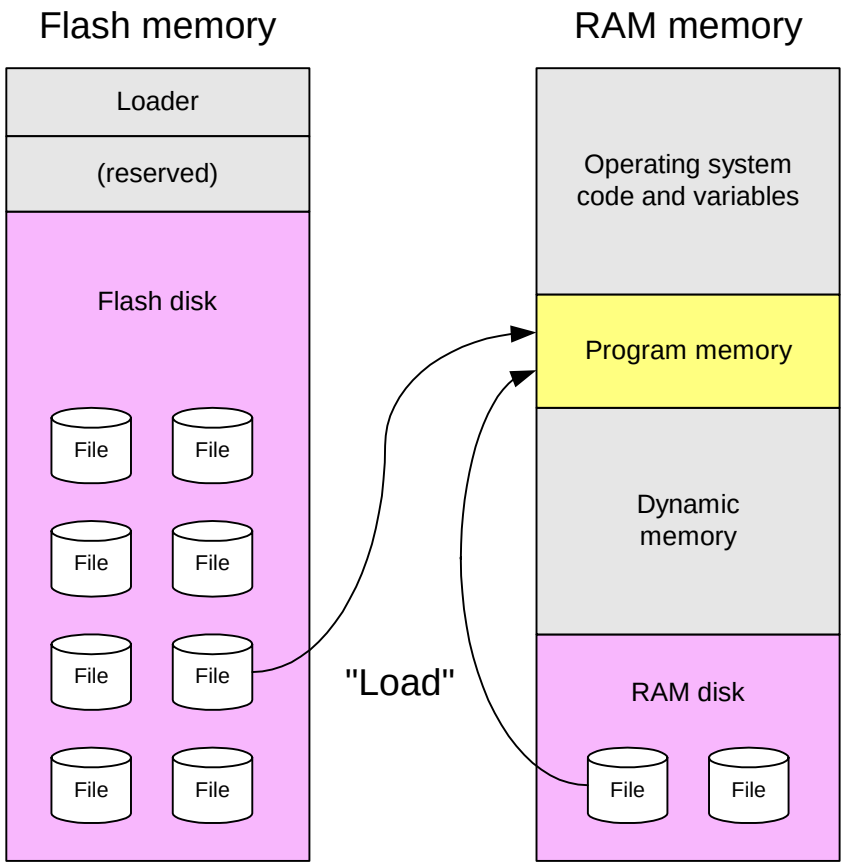


Fig. 1: Memory structure on the ROBO TX Controller

The following rules apply to the program memory:

- Only one program may be in the program memory at any point in time.
- Only the program in the program memory can be started. However, the program may also be present without being executed ("Stopped" status).
- With the "Load-after-power-on" and "Start-after-power-on" configuration options, it is possible to set a configuration that allows a program to be loaded automatically from a program file into the program memory after the interface is switched on and then automatically started there as needed. Alternatively, it can just be loaded and then started only after a push-button switch is depressed.

2.2 The transfer area

The transfer area is a memory area where input and output values are stored temporarily as a process image. This area is compared to the hardware every 10 ms by the ROBO TX Controller firmware after the program is started. In addition, configuration data are also stored in the transfer area. This data must be configured for specific options and parameters, such as the type of input: digital/analog, voltage/resistance measurement or bidirectional for the ultrasonic distance sensor, or the speed value for the motors, etc.

Programmers who have already worked with the "PC_Programming_RoboTXC" package and the ftMscLib.dll library on the PC are already familiar with the principle of the transfer area. The PC library, however, has a large number of encapsulation functions, whereas a C program on the controller usually accesses the transfer area directly. This is a good point at which to refer to the "PC_Programming_RoboTXC" documentation, where you will find more detailed descriptions of the fields and variables of the transfer area.

The transfer area on the controller also differs in another way: in contrast to the PC version, it has a table with function pointers known as the "Hook Table" behind which are hidden the firmware function calls that can be used by the robotics programs. For more information, see Chapter 7, "List of function calls (API)".

2.3 Executing local programs

As already indicated further back in this document, a local robotics program is part of a multi-tasking system that runs on the ROBO TX Controller as firmware. In this respect, the robotics program is executed in 1 ms time slots, of which approximately half (0.5 ms) is available to the program in each time slot. This (and only this) ensures that there is also sufficient computing time available for all other system tasks while still allowing the robotics program to run precisely and in realtime.

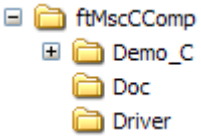
3 Installing the development environment

To explain as simply as possible how C programs are created and loaded for the ROBO TX Controller, we have prepared a simple command line based development environment that is compiled using makefiles C programs and implemented in an executable format. There are a number of convenient, graphically oriented development interfaces that can also be used, only describing them would be going beyond the general framework of this document.

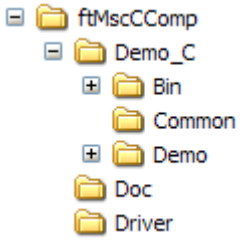
3.1 Unpacking the supplied ZIP files (compiler + tools)

The supplied development environment can be installed in any folder on the PC. In this description, the directory path C:\ftMscCComp will be used for the output. This directory along with all the subdirectories can be moved at any time to any other location without impacting its usability because only relative paths are used in the batch and make files. Please note, however, the name of the last predefined subdirectory.

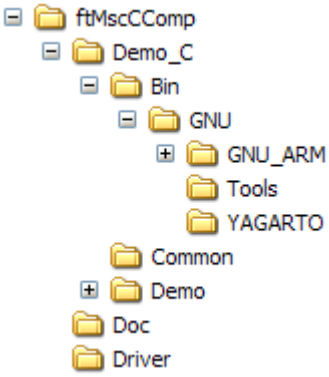
The first step is to unpack the contents of the ZIP file ftMscC**Demo**.zip to the folder C:\ftMscCComp. The folder contents should look like the following:



The "Doc" folder contains this documentation. The "Demo_C" contains the following folder structure:



The "Bin" folder does not currently contain any subfolders. Unpack the contents of ftMscC**Comp**.zip from this folder. The "Gnu" subfolder as well as additional subfolders will then appear under "Bin":



The "YAGARTO" subfolder also contains the setup files used to install the complete YAGARTO toolchain. Since not all components in this toolchain are used for the supplied demo programs, we have extracted the required software components into the subfolders "GNU_ARM" (GNU ARM GCC compiler) and "Tools" (make and rm tools). For the particular examples, the tools from these two folders are called directly and therefore it is not necessary to install the entire YAGARTO toolchain on the PC.

3.2 Installing the USB driver

You can skip this section if you already have a USB connection to the ROBO TX Controller.

The currently running ROBO TX Controller is connected to the PC using a USB cable (if necessary, disconnect and then reconnect the USB cable). Windows then reports the presence of a new USB device and searches for a driver. The installation wizard launches automatically. When prompted for "Use Windows Update", choose "No, not this time". On the next screen, choose "Install from a list or specific location " and point to the supplied file **`\Driver\ROBO_TX_Controller.inf`**. A new COM port device should then appear in the Device Manager with the description "fischertechnik USB ROBO TX Controller". It is recommended that you make note of the COM port number that is automatically assigned to it by Windows (see Device Manager).

3.3 Installing the Bluetooth driver

You can skip this section if you already have a Bluetooth connection to the ROBO TX Controller or if you want to work exclusively with USB.

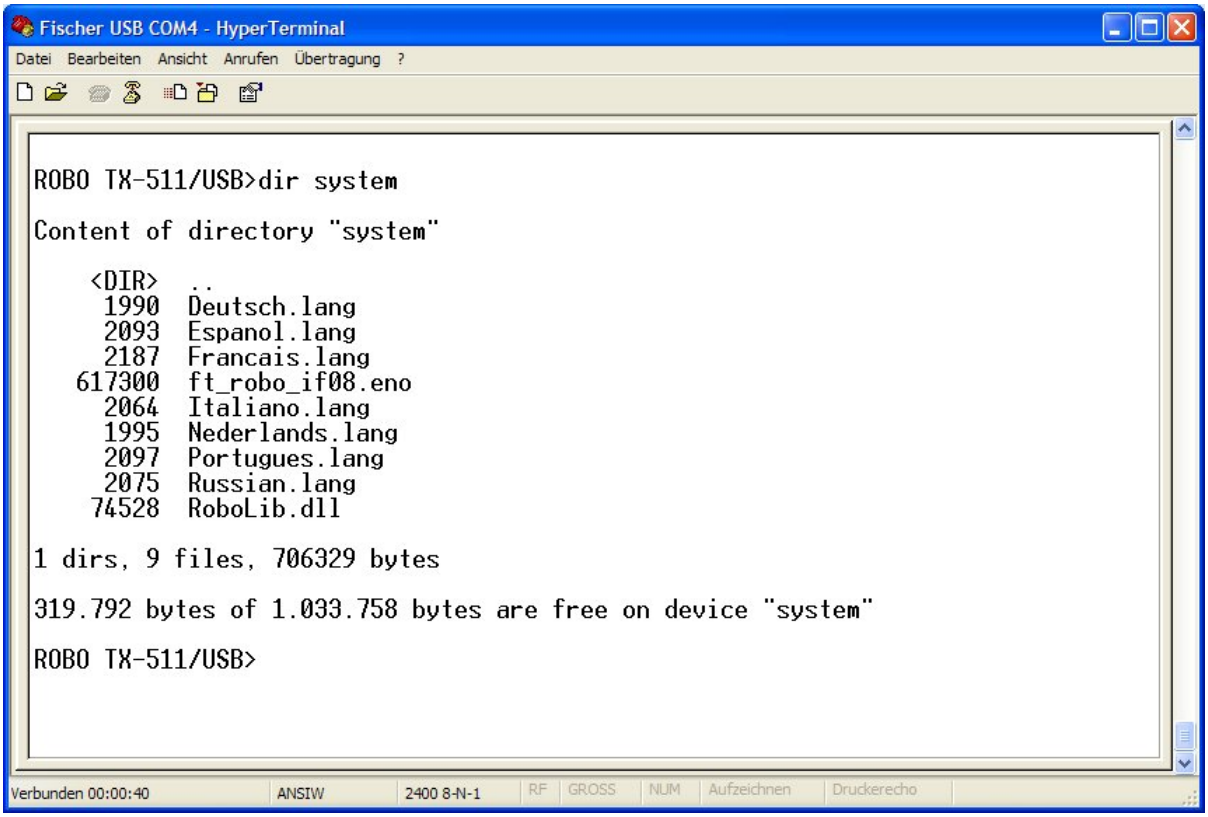
The currently running ROBO TX Controller can be detected by the PC Bluetooth service at any time as long as this property ("Bluetooth device discoverable") is not disabled in the relevant settings. The controller appears in the search window on the PC with its host name, which is shown on the display (e.g. "ROBO TX-511"). A connection is made to the PC by using the main PIN "1234". From this moment on, a fixed COM port is assigned to the ROBO TX Controller and the Bluetooth connection is made automatically from the PC as soon as the relevant COM port is accessed.

The procedures for the installation and PC connection with the TX-C may differ depending on the Bluetooth protocol stack and Bluetooth hardware (integrated or Bluetooth USB stick) used on the PC. In any case, follow the installation instructions of the particular manufacturer.

3.4 Initial USB and Bluetooth connection test

An initial test to check USB or Bluetooth communications with the ROBO TX Controller board can be carried out using a simple terminal program (e.g. HyperTerminal, ZOC or TeraTerm). Select the correct COM port number for the USB or Bluetooth driver and set any baud rate (does not apply to virtual COM ports).

If the connection could be made, you will see in the terminal program the monitor console of the 4NetOS operating system, which is running on the ROBO TX Controller. A command line prompt should now appear each time <ENTER> is pressed. By entering "dir", you can continue to view the contents of the "system", "flash" or "ramdisk" virtual disks:



You can view additional commands by entering "?" or "help". These are also explained in more detail in the separate documentation on PC programming (PC programming package).

3.5 COM port setting for the 4load_ft.exe load tool

Some batch files (with the .BAT extension) are located in the "Bin" folder. The files with the leading underscore character '_" in the name are for internal use in the environment compiled by us. These should therefore not be changed. Only one file, the "set_port.bat" file, may need to be adapted to your needs. This file sets the COM port on which the ROBO TX Controller can be activated for loading programs with our "4load_ft.exe" tool. There is only one line in this batch file:

```
set COM_PORT=
```

If you want to activate the ROBO TX Controller from the PC using the USB port, you can leave this setting empty. The "4load_ft.exe" load tool can detect the right COM port for the USB device automatically by searching for the name "fischertechnik USB ROBO TX Controller" in the list of COM ports (Windows Device Manager).

However, if you want to activate ROBO TX Controller via Bluetooth in order to load programs wirelessly, then you will have to provide the "4load_ft" tool with the correct COM port number by specifying it in the "set_port.bat" batch file because the port number cannot be detected automatically due to the large number of different Bluetooth protocol stacks and Bluetooth interfaces available solely for Windows PCs. If you reach the ROBO TX Controller via the port COM6, for instance, then enter the following in "set_port.bat":

```
set COM_PORT=COM6
```

4 Supplied demo programs written in C

The directory C:\ftMscCComp\Demo_C\Demo\ contains source code examples for robotics programs written in the C programming language.

The examples show simple applications for controlling the ROBO TX Controller:

LightRun – digital outputs (lights) An example of using the individual outputs O1 through O8, such as for controlling the timing of lights (flashing). The program quits automatically after running briefly.
MotorRun – motor Simple motor control on output M1 with different speeds in both directions of rotation. The program quits automatically after running briefly.
StopGo – encoder motor, digital input (push-button switch) and LC display output Demo program for controlling an encoder motor. Counter pulses are read and, if the counter status is 1000, the program quits. The motor is started (=1) or stopped (=0) via digital input I8. Also contains examples for using the message output on the LC display and for using a digital input (push-button switch).
WarningLight – ultrasonic distance sensor and digital output (light) Demo program for controlling a lamp on O8 with distance measurement via an ultrasonic distance sensor on I1. The light flashes at varying speed intervals, depending on the distance measured via the ultrasonic distance sensor. The program is closed by pressing the "Stop" push-button switch on the controller.
MotorEx_2M_Master – synchronizes two encoder motors Demo program for controlling two encoder motors at M1 and M2 (encoder outputs at C1 and C2 respectively) and for synchronized operation of the two motors. Both motors run back and forth cyclically every 200 steps. If the motor is braked while running (e.g. by using mechanical resistance), the other motor adapts its speed accordingly. The program is closed by pressing the "Stop" push-button switch on the controller.
MotorEx_Ext1 – operates encoder motor on extension controller Demo program for controlling an encoder motor at M1 (encoder output at C1 respectively) on an extension module (slave extension 1). The motor runs back and forth cyclically every 200 steps. The master module is the one connected to the PC. The program is closed by pressing the "Stop" push-button switch on the master controller.
StopGoBtButtonPart and **StopGoBtMotorPart** – Bluetooth messaging For a detailed description, see section 5.1 further down.

I2cTemp – I²C temperature sensor DS1631 and LC display output

Demo program for controlling a DS1631 external temperature sensor with the I²C interface (Conrad Electronic part. no. 19 82 98).

After a brief initialization sequence, the current temperature value is read out every second and displayed on the LC display.

The program is closed by pressing the "Stop" push-button switch on the controller.

Note: other Conrad C control series sensors can also be used and connected directly to the ROBO TX Controller. The I²C connection is compatible.

I2cTpa81 – I²C thermopile array infrared sensor TPA81 and LC display output

Demo program for controlling an external TPA81 thermopile array sensor with the I²C interface (available, for instance, from www.roboter-teile.de). This provides non-contact temperature control as well as easy locating of heat-emitting objects thanks to its array structure (8 pixels).

After a brief initialization sequence, the value of the ambient temperature as well as all 8 measuring points (pixels) are output on the LC display. If you move the thermal source in front of the lens, you can see the value with the pixel that has the thermal source in focus rise. The display is updated every second.

The program is closed by pressing the "Stop" push-button switch on the controller.

The created executable robotics programs (BIN files) must be located in the file system of the ROBO TX Controller in order to start them using the push-button switch.

5 Compiling and loading the demo programs

All examples in this package are stored in the "Demo" subfolder of the "Demo_C" folder. Each program there has its own subfolder that is named after the particular program it contains. At the same folder level as the "Demo" folder you will find the "Common" folder, which contains the header files (.H) and code fragments (.C) that can also be used by all programs.

For instance, switch to the directory C:\ftMscCComp\Demo_C\Demo\StopGo\ and you will see the batch files it contains (practically the same for all demos):

clean.bat – a batch file used to delete all generated (compiled) files. It also deletes all temporary files created during compiling and resets the particular directory back to its default condition.

load_flash.bat – a batch file used to load a compiled ROBO program onto the flash disk of the ROBO TX Controller (for permanent storage).

load_flash.bat – a batch file used to load a compiled ROBO program onto the RAM disk of the ROBO TX Controller (for temporary storage).

run.bat – a batch file used to start a robotics program loaded in the program memory of the ROBO TX Controller from the PC. In this case, the "run" command is passed to the 4cmd_ft.exe command tool. A good alternative is to start the program using the relevant push-button switch on the controller.

stop.bat – a batch file used to stop a robotics program already running on the ROBO TX Controller from the PC. In this case, the "stop" command is passed to the 4cmd_ft.exe command tool. A good alternative is to stop the program using the relevant push-button switch on the controller.

make.bat – a batch files used to compile the C source code of the demo program. This batch file calls the Make tool from the YAGARTO toolchain; it also calls the linker which for its part generates an ELF file and then a loadable BIN file. These types of BIN files can be loaded directly onto the ROBO TX Controller and executed there.

StopGo.c – C source file of the "StopGo" demo program.

param.mk – parameters for the Make tool. This includes information on names and, if applicable, the number of C source files and the name of the (output) files to be generated (i.e. file name of the robotics program).

Now run the "make.bat" batch file from the directory C:\ftMscCComp\Demo_C\Demo\StopGo\. You should then see something similar to the following on the screen:

```
arm-elf-gcc -S -mcpu=arm9e -DENDIAN_LITTLE -O3 -gdwarf-2 -fno-dwarf2-cfi-asm -ms
oft-float -fno-builtin -x c -Wall -c -I. -I../../Common -C -E ../../Common/prg_d
isp.c > ../../Common/prg_disp.p
arm-elf-gcc -S -mcpu=arm9e -DENDIAN_LITTLE -O3 -gdwarf-2 -fno-dwarf2-cfi-asm -ms
oft-float -fno-builtin -x c -Wall -c -I. -I../../Common -o ../../Common/prg_disp
.asm ../../Common/prg_disp.c
arm-elf-as -EL -mfloat-abi=soft -almns=../../Common/prg_disp.lst -o ../../Common
/prg_disp.o ../../Common/prg_disp.asm
arm-elf-gcc -S -mcpu=arm9e -DENDIAN_LITTLE -O3 -gdwarf-2 -fno-dwarf2-cfi-asm -ms
oft-float -fno-builtin -x c -Wall -c -I. -I../../Common -C -E StopGo.c > StopGo.
p
arm-elf-gcc -S -mcpu=arm9e -DENDIAN_LITTLE -O3 -gdwarf-2 -fno-dwarf2-cfi-asm -ms
oft-float -fno-builtin -x c -Wall -c -I. -I../../Common -o StopGo.asm StopGo.c
arm-elf-as -EL -mfloat-abi=soft -almns=StopGo.lst -o StopGo.o StopGo.asm
arm-elf-ld --cref --oformat elf32-littlearm --trace --nmagic --architecture=armv
5tej \
        --library-path=../../Bin/GNU/GNU_ARM/lib/gcc/arm-elf \
        --script=../../Common/ld.lcf \
        ../../Common/prg_disp.o StopGo.o \
        --library=gcc \
        -Map StopGo.map \
        -o StopGo.elf
arm-elf-ld: mode armelf
../../Common/prg_disp.o
StopGo.o
arm-elf-objcopy --output-target=binary StopGo.elf StopGo.bin
```

If any errors or warnings occur during the compiling process, they will also be output in the same window. This may occur when using programs you created yourself. The supplied demo programs, on the other hand, compile without error messages.

A variety of new files are generated during compiling. One of them is the "StopGo.bin" file, which is the executable robotics program. Now you can transfer this to the file system of the ROBO TX Controller and run it from there:

1. Switch on the ROBO TX Controller. It should be possible to establish a USB or Bluetooth connection from the PC to the ROBO TX Controller (see above).
2. Run "load_ramdisk.bat" or "load_flash.bat", depending on whether you want to transfer the robotics program to the RAM disk (volatile) or to the flash disk (non-volatile). The "4load_ft.exe" load tool called from the Bin folder via the referenced batch files finds the COM port automatically (USB) or finds it via the setting in "set_port.bat" (Bluetooth, see above). You can also overwrite these presets by explicitly entering a COM port number if you enter the following for COM6 as an example:

   ```
   load_ramdisk COM6
   ```

   The loading process itself (transfer of the robotics program to the ROBO TX Controller file system) appears on the screen as follows:

   ```
   Loading program file ...
   Connecting to target on port COM3...OK.
   Read file "StopGo.bin" (376 bytes).
   Loading.....OK.
   ```

   You can now see the name of the robotics program that has just been loaded in the second line of the status indicator on the ROBO TX Controller display. In this way you can load a large number of programs onto the ROBO TX Controller

controller that differ only in their unique file names. The only limitation is the size of the file system. You can then select the program to start via the File menu and the push-button switch of the ROBO TX Controller.

3. Now start the loaded program by pressing the left "Start" push-button switch. Once started, you can stop the program using the left "Stop" button as well, but only when the word "Stop" is visible next to the button. This is not the case when the robotics program itself outputs information to the display, which covers over the Status screen. In such a case, you can still force the program to quit by pressing both buttons at the same time. Alternatively, you can also stop and start the loaded robotics program from the PC as long as there is still a connection to it. Use the RUN.BAT and STOP.BAT batch files to do this (see above).

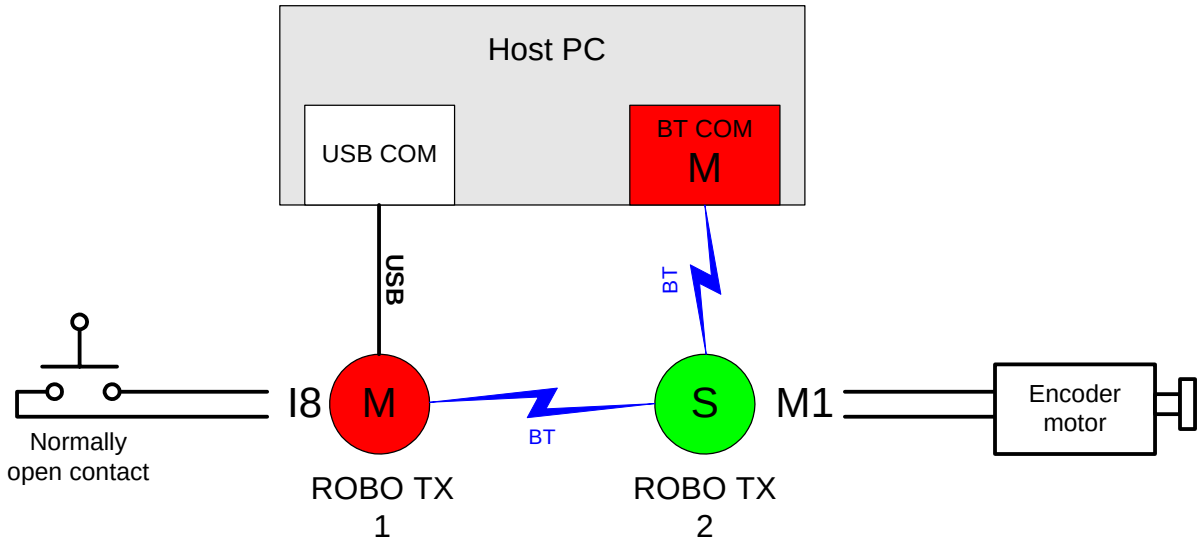
5.1 Example: two controllers and Bluetooth messaging

The following example demonstrates the exchange of Bluetooth messages between two controllers using C programs.

IMPORTANT NOTE: In regards to the "Bluetooth Messaging API", it is recommended that you read the introductory chapter 6 from the "PC_Programming_RoboTXC" package documentation. There you will find detailed information and a large number of illustrations explaining what to be aware of when using Bluetooth communication and the meaning of the terms MASTER (active connection, connect) and SLAVE (passive connection, listen) in this context.

The connections to the host PC illustrated in the following figure serve only to demonstrate how to load the robotics program files onto the controller. This can be done via USB or Bluetooth, even in parallel, as shown in the figure, but—unlike the illustration in the figure—it can be done using only a single connection if, for instance, both controllers were connected to the PC in succession via USB.

For instance, the following setup is required in order to run the supplied demo programs:



The MASTER controller (ROBO TX 1, on the left in the figure), is connected to the PC via the USB port; the SLAVE controller (ROBO TX 2, on the right in the figure) is connected via Bluetooth.

The StopGoBtButtonPart robotics program is loaded onto the MASTER controller. For this you use, for instance, the LOAD_RAMDISK.BAT batch file, as you would also for the other demo programs. The SLAVE controller, on the other hand, must be assigned a COM port number on the PC. This COM port number (e.g. COM111) is entered in the LOAD_RAMDISK_BT.BAT batch file in the directory C:\ftMscCComp\Demo_C\ Demo\StopGoBtMotorPart\. This allows the StopGoBtMotorPart robotics program to be loaded to the SLAVE controller via Bluetooth.

Before starting the programs, the Bluetooth addresses of the controllers must be entered into the BT_addr.c file in the folder `Common\C\` as follows (example only):

```
UCHAR8 bt_address_table[BT_CNT_MAX][BT_ADDR_LEN] =
{
   {0x00, 0x13, 0x7B, 0x53, 0x10, 0xE7}, // Bluetooth address of ROBO TX 1 (MASTER)
   {0x00, 0x13, 0x7B, 0x52, 0xB2, 0x11}, // Bluetooth address of ROBO TX 2 (SLAVE)
};
```

Next, the programs need to be regenerated so that the correct Bluetooth addresses are used in them.

The programs are then started as usual via the push-button switch on the relevant controller:

First the StopGoBtMotorPart program is started on the SLAVE controller and then the StopGoBtMotorPart program is started on the MASTER controller.

If you now press push-button switch I8 on the ROBO TX 1 (MASTER), the StopGoBtButtonPart program sends a Bluetooth message to ROBO TX 2 (SLAVE) and the motor on this controller begins to turn. When you release the button, the motor stops. The StopGoBtMotorPart program on ROBO TX 2 sends the position of the encoder motor back to ROBO TX 1 via a Bluetooth message. If the 1000 position is reached, the StopGoBtButtonPart program stops and the Bluetooth connection between ROBO TX 1 and ROBO TX 2 is terminated.

Various LCD screen outputs provide additional viewable information about the Connect status or indicate that the motor has reached the end position (1000).

6 Writing your own robotics programs

The easiest way to write your own robotics programs is to make a copy of one of the demo program folders. The subfolder and the generated program can be named differently (but this is not necessary).

6.1 Modifying one of the demo programs

Let's assume that you want to create a program called "MyProg", which should simply switch on only one light on the output O1 of the ROBO TX Controller. To do this, we would copy the entire "StopGo" folder to the path "Demo_C\Demo" and rename it to "MyProg". We would then rename the file "StopGo.c" in the "MyProg" folder to "MyProg.c". Next, we would modify the contents of the "param.mk" file as follows:

```
PROJ = MyProg
OBJS = MyProg.o
```

In this case, the output name (the file name of the executable robotics program) must be specified under **PROJ** (which stands for "project"). The ".bin" extension is appended automatically through the Make process. The list of object files to be included (linked) in this project must be specified under OBJS (for "objects"). If there is more than one, the object files must be separated by a space. The object files are created by the C compiler when the C source files are compiled. Before OBJS, simply list all names of the associated C source files that are required to create the robotics program, but just replace the ".c" extensions with ".o". In our example here, the program is very small and consists only of one C source file called "MyProg.c". The list before OBJS also consists of only one entry: "MyProg.o".

Now we want to change the "MyProg.c" source code, so let's take a look at the source code structure.

First you see the include file "ROBO_TX_PRG.h", which should be used by every robotics program. This contains important references and function prototypes, but above all contains an additional include file called "ROBO_TX_FW.h". This in turn contains all definitions of what is called the transfer area. Both include files are in the folder "Demo_C\Common", since they are used by all demo programs. This folder also contains additional files that generally apply to all demo programs:

Makefile – file with settings for the Make tool.

prg_disp.c – source code for the "Program Dispatcher"; it contains the main part of the "PrgDisp" function, which is the entry point for every robotics program. "PrgDisp" is periodically called by the firmware during local program execution (Download mode) in 1 a millisecond cycle. Each time "PrgDisp" is called, a decision is made as to whether the function "PrgInit" is called (only the first time "PrgDisp" is called). Otherwise, the "PrgTic" function is always called and the return code from "PrgTic" is returned to the firmware. Both functions "PrgInit" and "PrgTic" must be present in every robotics program.

ROBO_TX_FW.h – C include file with definitions of transfer area structures.

ROBO_TX_PRG.h – main C include file from every robotics program project. It must always be included by the main part of the robotics program and all additional C source files (if present) as long as they also want to access the transfer area.

ld.lcf – script file for the linker.

prg_bt.c – help functions for the Bluetooth messaging functions. These are also used by the demo programs StopGoBtButtonPart and StopGoBtMotorPart.

prg_bt_addr.c – list of Bluetooth addresses of the participating controllers used for Bluetooth messaging. These are also used by the demo programs StopGoBtButtonPart and StopGoBtMotorPart.

Returning to the structure of the "MyProg.c" file, we can see that the functions "PrgInit" and "PrgTic" are always included. "PrgInit" is an initialization function. It is called only once when the program starts in order to configure the inputs and outputs.

In the following we will make changes to the "PrgInit" function as an example:

```
void PrgInit
(
    TA * p_ta_array,    // pointer to the array of transfer areas
    int ta_count        // number of transfer areas in array (equal to TA_COUNT)
)
{
    TA * p_ta = &p_ta_array[TA_LOCAL];

    // Configure M1 to be used as separate O1 and O2 outputs
    p_ta->config.motor[0] = FALSE;

    // Inform firmware that configuration was changed
    p_ta->state.config_id += 1;

    // Switch off the lamp O1
    p_ta->output.duty[0] = 0;
}
```

Actually, we could have left the function "PrgInit" completely empty in this case, since all initializations that we are making also correspond to the initial state of the corresponding variables when the program starts. The initial state is set by the firmware by setting all configuration and output fields to zero before starting the program.

"PrgTic" is the main function of any robotics program. It is called cyclically by the "Program Dispatcher" every millisecond. The only limitation to the code in the "PrgTic" function is that it should not take too much time to run. It is recommended that the execution time should be under a half a millisecond (500 microseconds). Otherwise there will not be enough CPU time left for the remaining firmware and it will no longer be possible to perform controlled realtime multitasking.

In other words: processes that take a long time (worst case: infinite loops) in the robotics program and, if applicable, in all other local functions that call these processes, should be avoided at all costs. If you cannot avoid extended processes, these should be broken down in a suitable manner into individual substeps so that their processing can be distributed across several call cycles. To establish the execution time, you can use the firmware function "IsRunAllowed" in the "Hook Table" (see also Chapter 7, "List of function calls") to determine if there is still sufficient computing time available. If the function returns TRUE (1), additional code can be executed. Otherwise, you must quit the function "PrgTic" if a return code is returned.

To trace the target of our program (i.e. switching on a light at output O1), we will now modify the function "PrgTic", for instance, as follows:

```
int PrgTic
(
    TA * p_ta_array,    // pointer to the array of transfer areas
    int ta_count        // number of transfer areas in array (equal to TA_COUNT)
)
{
    int rc = 0x7FFF; // return code:
                     // 0x7FFF - program should be further called by the firmware;
                     // 0 - program should be normally stopped by the firmware;
                     // any other value is considered by the firmware as
                     // an error code and the program is stopped.
    TA * p_ta = &p_ta_array[TA_LOCAL];

    // Switch on the lamp O1 with the maximum brightness
    p_ta->output.duty[0] = DUTY_MAX;

    // Demonstration of usage of the IsRunAllowed function
    while (p_ta->hook_table.IsRunAllowed());

    return rc;
}
```

From this point on, compiling and loading of the "MyProg" program takes place as usual (see Chapter 5, "Compiling and loading demo programs"). After loading and starting the program (using the push-button switch), the light at output O1 should switch on.

6.2 Return codes

With each cyclical call, the main function "PrgTic" returns a return code. By default, 0x7FFF is returned, which means that the robotics program is requesting to be called again. If the robotics program runs infinitely or passes a (recognizable) error, "PrgTic" should return the code "0", which prompts the calling firmware to quit the program. This has the same effect as pressing the "Stop" button, only that in this case the decision to quit the program comes from the program.

For error messages that the program wants to report itself, all other (differing from 0x7FFF) return codes would also be suitable for stopping the program. Different types of errors could be distinguished using these return codes. The return codes are output to the LCD screen of the ROBO TX Controller after the robotics program is closed.

6.3 Activating slave controllers via the RS-485 extension

Up to this point we have always assumed that only one ROBO TX Controller is being operated. This is reflected by the fact that the transfer area variable is used in the transfer area array of index TA_LOCAL (value of 0):

```
TA * p_ta = &p_ta_array[TA_LOCAL];
```

On the other hand, if you have a combination of two (or more) ROBO TX Controllers connected via an RS-485 extension cable, where one controller (with which the PC also communicates) is the "Master" and the other is the "Slave", the master continues to be activated using TA_LOCAL and the slave is activated using TA_EXT1, for instance, if it has been configured at Slave ID 1 (see also the fischertechnik ROBO TX Controller handbook). You would program this, for instance, as follows:

```
TA * p_ta = &p_ta_array[TA_LOCAL];
TA * p_ta_ext = &p_ta_array[TA_EXT_1];
```

This makes it possible to control multiple ROBO TX Controllers simultaneously from one robotics program (running on the master board). The slaves are controlled remotely, as it were, by the master program (like the online mode from the PC). In this case it is not necessary (and also not possible) for the robotics programs to run simultaneously on the slave boards.

6.4 Warning

It is possible to create programs using the C compiler that could damage the processor in the ROBO TX Controller! Unlike programs for the fischertechnik ROBO Pro software, the hardware ports of the processor can be accessed directly using your own C program. Since in the case of this latest processor a majority of the internal hardware is configured by software commands, it is conceivable that if incorrect settings are made, the processor may no longer work correctly with the rest of the ROBO TX Controller hardware, and in extreme cases irreparable damage could ensue. Particularly in the case of accessing memory through pointers, the danger involved in being able to access the processor register this way is considerable.

The manufacturer of the ROBO TX Controller must therefore reject claims of warranty in the event of damage to processors caused by errors in C programs.

7 List of function calls (API)

The following is a list of all calls of external operating system functions to which a robotics program has access via the "Hook Table".

7.1 IsRunAllowed

*BOOL32 (*IsRunAllowed) (void)*

This function provides information to the robotics program as to whether there is any computing time remaining in the current execution time slot.

Return:	TRUE (1)	There is still computing time remaining. The program can continue with its execution.
	FALSE (0)	There is no computing time remaining. The program must stop the "PrgTic" function call immediately using a return code.

7.2 GetSystemTime

*UINT32 (*GetSystemTime) (enum TimerUnit unit)*

This function returns the system time since the ROBO TX Controller started in seconds, milliseconds or microseconds, depending on the value of the "unit" parameter. It can serve as an absolute time reference.

Call:	enum TimerUnit unit –	TIMER_UNIT_SECONDS (2) TIMER_UNIT_MILLISECONDS (3) TIMER_UNIT_MICROSECONDS (4)
Return:	UINT32 time	A 32-bit integer value with the number of accumulated time units (depending on the "unit" parameter) since the system started.

7.3 DisplayMsg

*void (*DisplayMsg) (struct ta_s * p_ta, char * p_msg)*

This function allows a robotics program to show a text message (max. 32 characters) on the ROBO TX Controller display.

Call:	struct ta_s * p_ta	Pointer to valid transfer area
	Char * msg	Pointer to text message to be displayed (null-terminated C string). If NULL is set here, any message shown previously is deleted and the Status screen reappears on the display.

7.4 IsDisplayBeingRefreshed

*BOOL32 (*IsDisplayBeingRefreshed) (struct ta_s * p_ta)*

This function provides information to the robotics program about whether additional output can be displayed at present or not yet because, for instance, the previous output is still being written to the display buffer.

Call:	struct ta_s * p_ta	Pointer to valid transfer area
Return:	TRUE (1)	The output can be written to the display.
	FALSE (0)	Nothing more can be written to the display yet.

7.5 Standard C library functions (list of 21 functions)

As in all C programming environments, there are a series of standard C library functions that can be used universally. The available functions are only listed and not described in detail (see also ROBO_TX_FW.H header file) below. This type of description can be found when needed in any relevant C programming manual or even on the Internet, e.g. under the following link (no guarantee is made with regard to the currentness of the link or the accuracy of the contents):

http://www.acm.uiuc.edu/webmonkeys/book/c_guide/

```
INT32     (*sprintf)     (char * s, const char * format, ...);
INT32     (*memcmp)      (const void * s1, const void * s2, UINT32 n);
void     *(*memcpy)      (void * s1, const void * s2, UINT32 n);
void     *(*memmove)     (void * s1, const void * s2, UINT32 n);
void     *(*memset)      (void * s, INT32 c, UINT32 n);
char     *(*strcat)      (char * s1, const char * s2);
char     *(*strncat)     (char * s1, const char * s2, UINT32 n);
char     *(*strchr)      (const char * s, INT32 c);
char     *(*strrchr)     (const char * s, INT32 c);
INT32     (*strcmp)      (const char * s1, const char * s2);
INT32     (*strncmp)     (const char * s1, const char * s2, UINT32 n);
INT32     (*stricmp)     (const char * s1, const char * s2);
INT32     (*strnicmp)    (const char * s1, const char * s2, UINT32 n);
char     *(*strcpy)      (char * s1, const char * s2);
char     *(*strncpy)     (char * s1, const char * s2, UINT32 n);
UINT32    (*strlen)      (const char * s);
char     *(*strstr)      (const char * s1, const char * s2);
char     *(*strtok)      (char * s1, const char * s2);
char     *(*strupr)      (char * s);
char     *(*strlwr)      (char * s);
INT32     (*atoi)        (const char * nptr);
```

7.6 Bluetooth Messaging API

The functions described below are part of the Bluetooth Messaging API. For an introduction to this topic, we recommend reading chapter 6 in the "PC_Programming_RoboTXC" package and chapter 6 in the Windows library documentation (also part of the "PC_Programming_RoboTXC" package), which will provide an explanation for the terms "channel number" (channel index), "on-call duty status" and "receive ready status".

7.6.1 BtConnect

*void (*BtConnect) (UINT32 channel,*
*UCHAR8 * btaddr,*
P_CB_FUNC p_cb_func);

Active establishment of a Bluetooth connection to a Bluetooth remote party uniquely identified by the Bluetooth target address. The result of the attempt to connect is reported asynchronously via the callback function. After a successful connection is made, this connection is managed in the firmware with the specified channel number for further accesses. To make multiple Bluetooth connections active simultaneously, this function can be called multiple times. (each time with its own channel number).

Limitations:

The BTConnect function returns an error via the callback call (**BT_CHANNEL_BUSY**) if on the same channel index an on-call duty status has already been registered via the function BtStartListen(). This prevents multiple connections between two controllers which could occur if a program initiates calls to another controller while simultaneously also attempting to accept calls from the same controller.

Call:

channel	-	channel number (index) under which the connection is managed in all subsequent calls. Determined by the calling application (1 to 8).
*btaddr	-	pointer to the associated Bluetooth address (6 bytes)
p_cb_func	-	callback function reporting the result of the connection (see also section 7.6.9)

Callback function parameter:

*p_data	-	pointer to data structure BT_CB

Data structure:

```
typedef struct bt_cb_s {      //  4 bytes
      UINT16      chanIdx;    //  Channel index
      UINT16      status;     //  Connection result
} BT_CB;
```

7.6.2 BtStartListen

*void (*BtStartListen) (UINT32 channel,*
*UCHAR8 * btaddr,*
P_CB_FUNC p_cb_func);

Passive establishment of a Bluetooth connection by a Bluetooth remote party uniquely identified by the specified Bluetooth source address. In this case, the ROBO TX Controller signals it is ready to receive exactly one Bluetooth connection by the named remote party (activated on-call duty status). Via the callback function, the incoming connection is reported asynchronously, as soon as this status is reached, just as any possible errors are reported. After a connection is made, the connection is managed in the firmware with the specified channel number (channel) for further accesses. To make multiple Bluetooth connections at the same time, this function can be called multiple times. (each time with its own channel number).

Limitations:

The BtStartListen function returns an error via the callback call (**BT_CHANNEL_BUSY**) if on the same channel index there is already an active connection that was made via BtConnect(). This prevents multiple connections between two controllers which could occur if a program initiates calls to another controller while simultaneously also attempting to accept calls from the same controller.

Call:

channel	-	channel number (index) under which the connection is managed in all subsequent calls. Determined by the calling application (1 to 8).
*btaddr	-	pointer to the associated Bluetooth address (6 bytes)
p_cb_func	-	callback function reporting the result of the connection (see also section 7.6.9)

Callback function parameter:

*p_data	-	pointer to data structure BT_CB

Data structure:

```
typedef struct bt_cb_s {      //  4 bytes
      UINT16      chanIdx;    //  Channel index
      UINT16      status;     //  Connection result
} BT_CB;
```

7.6.3 BtStopListen

*void (*BtStopListen) (UINT32 channel,*
P_CB_FUNC p_cb_func);

This function deactivates an on-call duty status that was previously activated by the BtStartListen function on the specified channel number. This means that as of this moment, Bluetooth connections are no longer accepted. However, a Bluetooth connection that already exists on this channel number will not be dropped by this function call, but will instead remain intact. Ending the on-call duty status in this case only has an effect on the period after the existing connection is ended (a connection can no longer be accepted until BtStartListen() has been called again).

Call:

channel - channel number (index 1 through 8).
p_cb_func - callback function reporting the result of the disconnection (see also section 7.6.9)

Callback function parameter:

*p_data - pointer to data structure BT_CB

Data structure:

```
typedef struct bt_cb_s {      //  4 bytes
      UINT16      chanIdx;    //  Channel index
      UINT16      status;     //  Disconnection result
} BT_CB;
```

7.6.4 BtDisconnect

*void (*BtDisconnect) (UINT32 channel,*
P_CB_FUNC p_cb_func);

Disconnection of an active Bluetooth connection referenced by the specified channel number. In this case it does not matter whether the connection was made actively or passively. The result of the attempt to disconnect is reported asynchronously via the callback function.

Call:

channel - channel number (index 1 through 8).
p_cb_func - callback function reporting the result of the disconnection (see also section 7.6.9)

Callback function parameter:

*p_data - pointer to data structure BT_CB

Data structure:

```
typedef struct bt_cb_s {      //  4 bytes
      UINT16      chanIdx;    //  Channel index
      UINT16      status;     //  Disconnection result
} BT_CB;
```

7.6.5 BtSend

*void (*BtSend) (UINT32 channel,*
UINT32 len,
*UCHAR8 *p_msg,*
P_CB_FUNC p_cb_func);

Writes data to an active Bluetooth connection referenced by the channel number. This call is used to pass a pointer to a send data buffer (p_msg) and a length (len). The function reads the specified number of bytes from the send data buffer. After the function call, the send data buffer can be freed up again. The result of the send attempt is reported asynchronously via the callback function.

Call:

channel	- channel number (index 1 through 8).
len	- length of the send data in the send buffer (max. 255 characters)
*p_msg	- pointer to send buffer with the send data (message)
p_cb_func	- callback function reporting the result (see also section 7.6.9)

Callback function parameter:

*p_data	- pointer to data structure BT_CB

Data structure:

```
typedef struct bt_cb_s {      //  4 bytes
      UINT16      chanIdx;    //  Channel index
      UINT16      status;     //  Result of the operation
} BT_CB;
```

7.6.6 BtStartReceive

*void (*BtStartReceive) (UINT32 channel,*
P_RECV_CB_FUNC p_cb_func);

This function is used to display the receive ready status of data (messages) on an active Bluetooth connection referenced by the channel number. When receiving a message, the callback function is called which contains a pointer to the received data and the length of the data. For the receive ready status, this function must be called only once (per channel number). Accordingly, incoming data call the callback function multiple times.

Call:

channel - channel number (index 1 through 8).
p_cb_func - callback function reporting the result (message) (see also section 7.6.9)

Callback function parameter:

*p_data - pointer to data structure BT_RECV_CB

Data structure:

```
typedef struct bt_receive_cb_s
{
    UINT16          chan_idx;         //  Channel index
    UINT16          status;           //  Result of operation
    UINT16          msg_len;          //  Length of received data
    UCHAR8          msg[BT_MSG_LEN];  //  Buffer for received data
} BT_RECV_CB;
```

7.6.7 BtStopReceive

*void (*BtStopReceive) (UINT32 channel,*
P_RECV_CB_FUNC p_cb_func);

Deactivation of the activated receive ready status by the function *BtStartReceive()*. Calling this function prevents the receipt of data on the connection with the specified channel number. However, incoming data are dismissed in the interim by the ROBO TX Controller on any Bluetooth connection that may still exist.

Call:

channel - channel number (index 1 through 8).
p_cb_func - callback function reporting the result (see also section 7.6.9)

Callback function parameter:

*p_data - pointer to data structure BT_RECV_CB

Data structure:

```
typedef struct bt_receive_cb_s
{
    UINT16          chan_idx;         //  Channel index
    UINT16          status;           //  Result of operation
    UINT16          msg_len;          //  (not important here)
    UCHAR8          msg[BT_MSG_LEN];  //  (not important here)
} BT_RECV_CB;
```

7.6.8 BtAddrToStr

*char *(*BtAddrToStr) (UCHAR8 *btaddr,*
*char *str);*

Converts a Bluetooth address in 6-byte format (btaddr) into a readable C string (str) in the format "xx:xx:xx:xx:xx:xx" for suitable output on the LCD screen, for instance.

Call:

*btaddr - pointer to the associated Bluetooth address (6 bytes)
*str - pointer to buffer for result string

Return code:

*str - pointer to result string

7.6.9 List of status codes in the callback functions

List of possible status codes: (`enum CB_BtStatus`), see also ROBO_TX_FW.H header file

Status	Meaning
0	= **BT_SUCCESS** , action successful
1	= **BT_CON_EXIST** , already connected
2	= **BT_CON_SETUP** , connection to this BT address is actively being carried out
3	= **BT_SWITCHED_OFF** , connection failed: Bluetooth is switched off locally as per configuration
4	= **BT_ALL_CHAN_BUSY** , connection failed: Bluetooth channel no longer available locally
5	= **BT_NOT_ROBOTX** , connection failed: incompatible BT device cannot be connected (not a ROBO TX Controller)
6	= **BT_CON_TIMEOUT** , failed: timeout, no device can be reached at this address (timeout)
7	= **BT_CON_INVALID**, there is no active connection with the specified channel number (index).
8	= **BT_CON_RELEASE**, termination of connection to this BT address is already activated and is being carried out
9	**= BT_LISTEN_ACTIVE**, the listen function has already been activated for the specified channel index.
10	**= BT_RECEIVE_ACTIVE**, the receive function has already been activated.
11	**= BT_CON_INDICATION**, signals a connection on the passive side. A Bluetooth connection from the specified Bluetooth address is established.
12	**= BT_DISCON_INDICATION**, signals a passive connection (e.g. triggered by remote party). The Bluetooth connection no longer exists.
13	**= BT_MSG_INDICATION**, signals the receipt of a Bluetooth message from the remote party.
14	**= BT_CHANNEL_BUSY**, the specified channel index is already registered (on-call duty status) or in use (active connection).
15	**= BT_BTADDR_BUSY**, for this Bluetooth address, there is already an on-call duty status or an active connection via a different channel number (index).
16	**= BT_NO_LISTEN_ACTIVE**, on the remote party no listen function was activated; connection is not possible.

7.7 I²C API

The functions described below are part of the I²C API. For an introduction to this topic, we recommend reading chapter 7 in the "PC_Programming_RoboTXC" package documentation.

Important notes:

The I²C device address must be specified with only 7 bits in accordance with the I²C specification (value range: 0 to 127).

Moreover, I²C the device addresses 80 and 84 (0x50 and 0x54) are reserved for an internal EEPROM of the ROBO TX Controller. Access to these addresses is not permitted by the API. The I²C device addresses 81 through 83 and 85 through 87 (0x51 through 0x53 and 0x55 through 0x57) are also reserved memory areas for the same EEPROM, but are not used by the firmware. In this case, bus access conflicts could (but may not necessarily) occur with external I2C devices that use one of these addresses.

7.7.1 I2cRead

void I2cRead (UCHAR8 devaddr,
UINT32 offset,
UCHAR8 flags,
P_I2C_CB_FUNC p_cb_func);

One byte (8-bit) or two bytes (16-bit) is read on the I²C bus at the "devaddr" device address and possibly within the device at the "offset" subaddress. The addressing, data bus width, byte sequence (16-bit values only), behavior in the case of bus errors and access speed are specified using the "flags" parameter. Via the callback function, the result of the read access as well as the read datum is returned asynchronously as soon as this status is reached.

Call: UCHAR8 devaddr - 12C device address

UINT32 offset - if addressing is required within the device, then this internal address is passed here. The value of "flags" specifies the length of the internal address in bits 0..1.

UCHAR8 flags - Access flags used:

Bit 0..1	Addressing	00: none ("Offset" invalid) 01: 8-bit addressing 10: 16-bit addressing, MSB first 11: 16-bit addressing, LSB first
Bit 2..3	Data width	00: - *not permitted* - 01: 8-bit data (1 byte) 10: 16-bit data (2 bytes), MSB first 11: 16-bit data (2 bytes), LSB first
Bit 4	KeepOpen	0: normal access 1: quick access without STOP/START
Bit 5..6	Error Mask = behavior in the case of bus error	00: abort 01: repeat up to 10 times 10: repeat until successful 11: - *not permitted* -
Bit 7	Clock rate	0: standard (100 kHz) 1: fast (400 kHz)

p_cb_func - callback function that reports back the result of the operation asynchronously.

Return: (none)

Callback function return codes: pointer to data structure 12C_CB

Data structure:

```
typedef struct {            // 4 bytes
      UINT16      value;    // Datum read (with 8-Bit, only LSByte is valid)
      UINT16      status;   // Result of I2C bus operation (see 7.7.3)
} I2C_CB;
```

7.7.2 I2cWrite

void I2cWrite (UCHAR8 devaddr,
UINT32 offset,
UINT16 data,
UCHAR8 flags,
P_I2C_CB_FUNC p_cb_func);

One byte (8-bit) or two bytes (16-bit) is written on the I^2C bus at the "devaddr" device address and possibly within the device at the "offset" subaddress. The addressing, data width, byte sequence (16-bit values only), behavior in the case of bus errors and access speed are specified using the "flags" parameter. Via the callback function, the result of the write access as is returned asynchronously as soon as this status is reached.

Call:

UCHAR8 devaddr - 12C device address

UINT32 offset - if addressing is required within the device, then this internal address is passed here. The value of "flags" specifies the length of the internal address in bits 0..1.

UINT16 data - datum (8-bit or 16-bit) to be written

UCHAR8 flags - Access flags used:

Bit 0..1	Addressing	00: none ("Offset" invalid) 01: 8-bit addressing 10: 16-bit addressing, MSB first 11: 16-bit addressing, LSB first
Bit 2..3	Data width	00: - *not permitted* - 01: 8-bit data (1 byte) 10: 16-bit data (2 bytes), MSB first 11: 16-bit data (2 bytes), LSB first
Bit 4	KeepOpen	0: normal access 1: quick access without STOP/START
Bit 5..6	Error Mask = behavior in the case of bus error	00: abort 01: repeat up to 10 times 10: repeat until successful 11: - *not permitted* -
Bit 7	Clock rate	0: standard (100 kHz) 1: fast (400 kHz)

p_cb_func - callback function that reports back the result of the operation asynchronously.

Return: (none)

Callback function return codes: pointer to data structure 12C_CB

Data structure:

```
typedef struct {            // 4 bytes
      UINT16      value;    // Written datum repeated
      UINT16      status;   // Result of I2C bus operation (see 7.7.3)
} I2C_CB;
```

7.7.3 Callback status

Return codes during callback (status)

Status code	Meaning	
0	I2C operation successful	I2C_SUCCESS
1	I2C read error	I2C_READ_ERROR
2	I2C write error	I2C_WRITE_ERROR

8 Updating the ROBO TX Controller firmware

There are several ways to update the ROBO TX Controller firmware:

- Via ROBO Pro (usually offered automatically the first time a connection is made, as long as an update is necessary)
- Using the Repair tool (see the separate package with the relevant description)

9 Document change history

Version	Date	Author	Other comments
1.0	6/22/2010	Alexey Kucherenko Peter Duchemin	- First version - Adapted for firmware V1.18
1.1	1/24/2012	Peter Duchemin	- New functions, including standard C library and Bluetooth Messaging API - Demo program enhancements - Adapted for firmware V1.24
1.2	4/25/2012	Peter Duchemin	- Added I^2C API functions - Demo program enhancements - Adapted for firmware V1.30