The software described in this manual is furnished under a license agreement and may be used only in accordance with the terms of that agreement.

Copyright Notice

Copyright ©2010 Moxa Inc.
All rights reserved.
Reproduction without permission is prohibited.

Trademarks

The MOXA logo is a registered trademark of Moxa Inc.
All other trademarks or registered marks in this manual belong to their respective manufacturers.

Disclaimer

Information in this document is subject to change without notice and does not represent a commitment on the part of Moxa.

Moxa provides this document as is, without warranty of any kind, either expressed or implied, including, but not limited to, its particular purpose. Moxa reserves the right to make improvements and/or changes to this manual, or to the products and/or the programs described in this manual, at any time.

Information provided in this manual is intended to be accurate and reliable. However, Moxa assumes no responsibility for its use, or for any infringements on the rights of third parties that may result from its use.

This product might include unintentional technical or typographical errors. Changes are periodically made to the information herein to correct such errors, and these changes are incorporated into new editions of the publication.

Technical Support Contact Information

www.moxa.com/support

**Moxa Americas**
Toll-free: 1-888-669-2872
Tel: +1-714-528-6777
Fax: +1-714-528-6778

**Moxa Europe**
Tel: +49-89-3 70 03 99-0
Fax: +49-89-3 70 03 99-99

**Moxa China (Shanghai office)**
Toll-free: 800-820-5036
Tel: +86-21-5258-9955
Fax: +86-10-6872-3958

**Moxa Asia-Pacific**
Tel: +886-2-8919-1230
Fax: +886-2-8919-1231
Table of Contents

1. Introduction .................................................................................................................................. 1-1
   Sample Code Architecture ............................................................................................................. 1-1
   Applications Using Moxa Sample Code ......................................................................................... 1-2

2. Installation and Development ....................................................................................................... 2-1
   Installation .................................................................................................................................. 2-1
   Source Tree ................................................................................................................................. 2-2

3. Peripheral I/O Library .................................................................................................................. 3-1
   Buzzer Functions ......................................................................................................................... 3-1
      mbuzzer_open function ........................................................................................................... 3-1
      mbuzzer_close function ......................................................................................................... 3-1
      mbuzzer_beep function ......................................................................................................... 3-2
   LCM Functions ........................................................................................................................... 3-2
      mxlcm_open function .............................................................................................................. 3-2
      mxlcm_close function ............................................................................................................ 3-2
      mxlcm_get_cursor function ................................................................................................. 3-3
      mxlcm_set_cursor function ................................................................................................. 3-3
      mxlcm_control function ....................................................................................................... 3-3
      mxlcm_write function .......................................................................................................... 3-3
      mxlcm_write_screen function ......................................................................................... 3-4
   Keypad Functions ........................................................................................................................ 3-4
      keypad_open function ........................................................................................................... 3-5
      keypad_close function ......................................................................................................... 3-5
      keypad_get_pressed_key function ...................................................................................... 3-5
   RTC Functions ........................................................................................................................... 3-5
      mxrtc_ge function ................................................................................................................ 3-5
      mxrtc_set function ............................................................................................................... 3-6
      mxrtc_set_system_time function ......................................................................................... 3-6
      mxrtc_get_system_time function ......................................................................................... 3-6
   Watchdog Timer Functions ......................................................................................................... 3-6
      mxwdtg_open function ............................................................................................................ 3-6
      mxwdtg_close function ........................................................................................................... 3-7
      mxwdtg_refresh function ...................................................................................................... 3-7
   DIO Functions ............................................................................................................................. 3-7
      mxrdgio_open function ........................................................................................................... 3-7
      mxrdgio_close function ........................................................................................................... 3-7
      mxrdgio_get_input_signal function .................................................................................... 3-8
      mxrdgio_set_output_signal function .................................................................................... 3-8
      mxrdgio_set_output_signal_high function ........................................................................... 3-8
      mxrdgio_set_output_signal_low function ............................................................................. 3-8
   Serial Functions ........................................................................................................................ 3-8
      mxsp_open function ............................................................................................................... 3-8
      mxsp_close function .............................................................................................................. 3-9
      mxsp_set_baudrate function ................................................................................................. 3-9
      mxsp_get_baudrate function ................................................................................................. 3-9
      mxsp_set databits function .................................................................................................... 3-10
      mxsp_get databits function ................................................................................................. 3-10
      mxsp_set_stopbits function .................................................................................................. 3-10
      mxsp_get_stopbits function ................................................................................................. 3-10
      mxsp_set parity function ...................................................................................................... 3-10
      mxsp_get parity function ...................................................................................................... 3-11
      mxsp_set_flow_control function .......................................................................................... 3-11
      mxsp_get_flow_control function .......................................................................................... 3-11
      mxsp_set_interface function ............................................................................................... 3-12
      mxsp_set_current function .................................................................................................... 3-12
      mxsp_set_nonblocking function .......................................................................................... 3-12
      mxsp_read function ............................................................................................................. 3-12
      mxsp_blocking_read function .............................................................................................. 3-13
      mxsp_write function ............................................................................................................. 3-13
   Network Interface Functions ...................................................................................................... 3-13
      mxiface_get_info function .................................................................................................... 3-13
      mxiface_get_info_from_file function ................................................................................... 3-14
      mxiface_get_info static function ......................................................................................... 3-14
      mxiface_get_mac_address function ...................................................................................... 3-14
      mxiface_update_info function .............................................................................................. 3-15
      mxiface_update_interface_file function .............................................................................. 3-15
      mxhost_get_dns function ....................................................................................................... 3-15
4. Peripheral I/O Library Sample Programs ................................................................................................... 4-1
   Keypad Callback Functions .......................................................................................................................... 4-1
      keypad_callback_init function .................................................. 4-1
      keypad_callback_add function .................................................. 4-1
      keypad_callback_dispatch function ........................................... 4-2
      keypad_callback_quit function .................................................. 4-2
   LCM/Keypad Context Functions .................................................................................................................. 4-2
      keypad lcm_menu_init function .................................................. 4-2
      keypad lcm_menu_add function .................................................. 4-3
      keypad lcm_menu_dispatch function ........................................... 4-3
      keypad lcm_menu_quit function .................................................. 4-3
   LCM/Keypad Sample Program .................................................. 4-4
   DIO Sample Program ................................................................................................................................. 4-5

5. Application Library .................................................................................................................................. 5-1
   User-defined Functions ............................................................................................................................... 5-2
      open user-defined function ......................................................... 5-2
      dispatch user-defined function .................................................. 5-2
      close user-defined function ....................................................... 5-3
      timer user-defined function ....................................................... 5-3
      initialization user-defined function ........................................... 5-3
   Data Structures ......................................................................................................................................... 5-3
      DATAPKT data structure ............................................................. 5-3
      USERFUN data structure ............................................................. 5-3
      UARTPRM data structure .............................................................. 5-4
      SRVPRM data structure .............................................................. 5-4
      TCPPRM data structure .............................................................. 5-4
      CLNPTRM data structure ............................................................. 5-5
      UDPPRMP data structure ............................................................. 5-5
   API Functions ............................................................................................................................................ 5-6
      connection_destroy function .................................................... 5-6
      connection_send_data function ................................................ 5-6
      connection_open function ......................................................... 5-6
      connection_dispatch_loop function ........................................... 5-6
      connection_dispatch_quit function .......................................... 5-7
      mxsp_connection_purge function ............................................. 5-7
      mxsp_connection_set_mask function ......................................... 5-7

6. Application Library Sample Programs ................................................................................................... 6-1
   TCP Functions ......................................................................................................................................... 6-1
   UDP Functions .......................................................................................................................................... 6-3
   Serial Functions ....................................................................................................................................... 6-5

7. MODBUS Sample Programs .................................................................................................................. 7-1
   MODBUS Functions .................................................................................................................................. 7-1
   MODBUS TCP to RTU ............................................................................................................................... 7-3
   MODBUS RTU to ASCII ........................................................................................................................... 7-3
Introduction

To help end-users reduce development costs, Moxa provides sample code for a wide range of embedded applications, such as serial-to-Ethernet (S2E), serial-to-serial (S2S), and MODBUS TCP, RTU, and ASCII. The high-level sample code provided by Moxa implements complex data communication concepts with simple functions, making it easier for users to understand how to use the code for their own applications. In addition, low-level sample code that manages direct access to peripheral IOs, such as the LCM, keypad, and digital IO, are also included. With ready access to such a rich assortment of embedded applications, programmers obtain a much greater flexibility than would otherwise be possible.

Moxa’s sample code provides a common C/C++ programming function interface that can be used with all of Moxa’s Linux and Windows embedded computers. The function interface helps developers create portable applications for Moxa’s embedded computers. The purpose of this Programming Guide is to provide the information needed to use the sample code. Here is a brief description of the contents of this guide:

- **Chapter 1**
  Sample code architecture and related applications.

- **Chapter 2**
  Installing the sample code and building procedures.

- **Chapter 3**
  Lower layer peripheral I/O library.

- **Chapter 4**
  Useful sample programs based on the peripheral I/O library.

- **Chapters 5 and 6**
  Higher layer application library and related sample programs.

- **Chapter 7**
  MODBUS functions and related sample programs.

Sample Code Architecture

Moxa’s sample code supports a peripheral I/O library, application library, and related sample programs under Linux, Windows CE, and Windows XPe for Moxa ARM and x86 embedded computers. The sample code uses a layered architecture to build up supported libraries. The peripheral I/O library provides functions to access different kinds of peripherals. The application library provides higher layer functions for networking and serial programming based on corresponding functions in the lower layer peripheral I/O library. Finally, sample programs are provided to illustrate how to use the functions from both libraries.
Applications Using Moxa Sample Code

The first example of an application using Moxa sample code is Moxa Device Manager (MDM), which is used with Moxa’s embedded computers. MDM was created using the application library and peripheral I/O library to implement MDM TCP and UDP requests and response message services.

The next example is to build a MODBUS gateway that supports both MODBUS TCP master and MODBUS RTU/ASCII slave, as well as MODBUS RTU/ASCII master and MODBUS TCP slave. The application library and peripheral I/O library support all required functions for data communication and message verification, and for writing the MODBUS TCP and MODBUS RTU/ASCII applications.

Another example is to build an S2E or S2S gateway for devices with proprietary serial protocol to provide public MODBUS TCP or RTU/ASCII protocol to access the devices. The application library and peripheral I/O library also support all necessary functions for this example.

The final example is to build a configuration program to serve the setup and display of configuration parameters of networking interfaces and application configuration files using LCM and keypad functions in the peripheral I/O library. The example presents another user interface that is suitable for applications that need to be configured on site.
This chapter introduces how to install sample code, the contents of the sample code, and how to create sample code.

### Installation

Sample code can be downloaded from the Rcore Community website. For Linux products, download the code that corresponds to your product model (e.g., download the file `moxalib_UC84XX-LX.tar.gz` for UC-8400 series products). For Windows, one file is available for all models: `moxalib_XPE_CE.zip`.

Next, upload the file to your development machine. For Linux, run the `tar` command to extract the sample code file to your target directory. For Windows, use the appropriate Windows tool to extract sample code zip file to your target folder.

### Source Tree

After installing the sample code for your product, you will have ready access to the files you need to develop your own software.

The files for Linux are located in the top-level directory `moxalib`, and are organized as shown below. There are three prebuilt libraries: `libmxphio.a`, `libmxconn.a`, and `libmodbus.a`, called the peripheral I/O library, connection library, and MODBUS library, respectively.

- `README.txt` -- sample code build procedure
- `makefile` -- top-level makefile
- `rules.make` -- makefile rule
- `inc` -- include directory
  - `mxphio` -- header files for peripheral I/O library functions
  - `mxconn` -- header files for connection library functions
  - `modbus` -- header files for MODBUS library functions
- `lib` -- three prebuilt libraries
- `samples` -- sample program source directory
  - `mxphio` -- sample program source code using peripheral I/O library functions
  - `mxconn` -- sample program source code using connection library functions
  - `modbus` -- sample program source code using MODBUS library functions
- `src` -- source directory
  - `mxphio` -- sources of peripheral I/O library

The files for Windows are located in the top-level directory `moxalib`, and are organized as shown below. The three prebuilt libraries in sample code are `mxphio.lib`, `mxconn.lib`, and `modbus.lib`, called the peripheral I/O library, connection library, and MODBUS library, respectively.

- `README.txt` -- sample code build procedure
- `bin` -- prebuilt sample programs for each product series
- `inc` -- include directory
  - `mxphio` -- header files for peripheral I/O library functions
  - `mxconn` -- header files for connection library functions
modbus -- header files for MODBUS library functions
lib -- three prebuilt libraries for each product series
samples -- sample program source directory
mxphio -- sample program source code using peripheral I/O library functions
mxconn -- sample program source code using connection library functions
modbus -- sample program source code using MODBUS library functions
src -- source directory
mxphio -- sources of peripheral I/O library

If you want to rebuild the peripheral library and sample programs for your Windows product, you must create Visual Studio projects for them. The following example shows one way to organize your project files.

projects -- Windows Visual Studio 2005 project directory
CE -- Visual Studio 2005 project directory for Windows CE
samples
  mxphio
  mxconn
  modbus
src
  mxphio

XPe -- Visual Studio 2005 project directory for Windows XPe
samples
  mxphio
  mxconn
  modbus
src
  mxphio

Build Procedure

To build sample code for Linux, change your working directory to the moxalib directory on your development machine and then type the command make ARCH=arch, where arch is one of the following architecture values.

uc74xx       UC-74xx-LX, DA-660-LX
da66x        DA-66x-LX, UC-74xx-LX Plus
ia24x        IA24x-LX, W3xx-LX, UC-7112-LX Plus, IA26x-LX, W406-LX
uc711x       UC-7101-LX, UC-711x-LX
uc84xx       UC-84xx-LX

To build sample code for Windows, follow the instructions in the previous section to set up your projects. Next, build three libraries first and then build their corresponding sample programs.
Moxa’s sample code uses ten kinds of programming functions for nine peripherals; the sample programs are provided by the peripheral I/O library. The peripherals include buzzer, LCM, keypad, RTC, watchdog timer, digital IO, serial, network interface, and socket. In this chapter, programming functions for each peripheral will be introduced; Sample programs from the sample code will be introduced in the next chapter.

Buzzer Functions

In some cases, such as when the embedded computer detects a high pressure in a vacuum device, your applications may need the attention of the operator. A low volume buzzer is the perfect component for this purpose.

In the following sections we describe how to activate the buzzer and use its programming functions. The header file of all buzzer functions is `mxbuzzer.h` in the Linux directory `inc/mxphio`. For Windows, the header file is located in folder `inc\mxphio`.

**mxbuzzer_open function**

This function opens a handle that operates the buzzer.

```
HANDLE mxbuzzer_open(void);
```

<table>
<thead>
<tr>
<th>Inputs</th>
<th>None.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return Value</td>
<td>When successful, this function returns a positive value (or handle) that represents the buzzer device.</td>
</tr>
</tbody>
</table>

**mxbuzzer_close function**

This function closes the open handle of the buzzer.

```
void mxbuzzer_close(HANDLE fd);
```

<table>
<thead>
<tr>
<th>Inputs</th>
<th><code>fd</code> Specifies the open handle of the buzzer.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return Value</td>
<td>None.</td>
</tr>
</tbody>
</table>

mxbuzzer_beep function

This function regulates the buzzer for Linux.

```c
void mxbuzzer_beep(HANDLE fd, int time);
```

**Inputs**

- `fd` Specifies the open handle of the buzzer.
- `time` Specifies the amount of time (in milliseconds) that the beep lasts.

**Return Value**

None.

This function regulates the buzzer for Windows.

```c
void mxbuzzer_beep(HANDLE fd, int time, int freq);
```

**Inputs**

- `fd` Specifies the open handle of the buzzer.
- `time` Specifies the amount of time (in milliseconds) that the beep lasts.
- `freq` The frequency of the beep.

**Return Value**

None.

**LCM Functions**

Moxa's UC-7400 series and DA-660 series embedded computers have an LCM on the top panel. The LCM on the UC-7420 and UC-7410 embedded computers has an English text display area that holds 8x16 characters, while the LCM on the DA-660 series embedded computers has a display area that holds 2x16 characters.

For each embedded computer with an LCM, an internal service is pre-installed at the software kernel level. Application programs drive the service by opening and writing data in, and controlling a device file. The following sections describe each LCM function and their usage. The header file of LCM functions is `mxlcm.h`, which is located in the `inc/mxphio` directory for Linux, and in the `inc\mxphio` folder for Windows.

**mxlcm_open function**

This function opens a device file associated with the LCM hardware.

```c
HANDLE mxlcm_open(void);
```

**Return Value**

When successful, this function returns a positive file descriptor or handle. A negative value will be returned if the LCM device does not exist.

**mxlcm_close function**

This function closes a device file associated with the LCM hardware.

```c
void mxlcm_close(HANDLE fd);
```

**Inputs**

- `fd` An open file descriptor or handle representing the LCM.
**mxlcm_get_cursor function**

This function gets the position of the cursor on the LCM screen.

```c
int mxlcm_get_cursor(HANDLE fd, int *x, int *y);
```

**Inputs**
- `fd`: An open file descriptor or handle representing the LCM.

**Outputs**
- `x`: Pointer to the horizontal position of the cursor.
- `y`: Pointer to the vertical position of the cursor.

**Return Value**
Returns 0, if successful.

**mxlcm_set_cursor function**

This function moves the cursor to a specified position on the LCM screen.

```c
int mxlcm_set_cursor(HANDLE fd, int x, int y);
```

**Inputs**
- `fd`: An open file descriptor or handle representing the LCM.
- `x`: Horizontal position of the cursor.
- `y`: Vertical position of the cursor.

**Return Value**
Returns 0, if successful.

**mxlcm_control function**

This function controls LCM features such as backlight on/off, cursor blink on/off, etc.

```c
int mxlcm_control(HANDLE fd, int flag);
```

**Inputs**
- `fd`: An open file descriptor or handle representing the LCM.
- `flag`: Indicates the combinational features of the LCM.

**Return Value**
Returns 0, if successful.

**mxlcm_write function**

This function writes a string of English text onto the LCM screen starting from a specified position.

```c
int mxlcm_write(HANDLE fd, int x, int y, char *data, int len);
```

**Inputs**
- `fd`: An open file descriptor or handle representing the LCM.
- `x`: Horizontal position of the text string.
- `y`: Vertical position of the text string.
- `data`: Pointer to the text string.
- `len`: Length of the text string.

**Return Value**
Returns the number of characters written, if successful.
mxlcms_write_screen function

This function writes a full screen text string in English onto the LCM.

```c
int mxlcms_write_screen(HANDLE fd, char text[MAX_LCM_ROWS][MAX_LCM_COLS]);
```

**Inputs**
- `fd`: An open file descriptor or handle representing the LCM.
- `text`: Full screen text string.

**Return Value**
Returns 0, if successful.

**Note:** `MAX_LCM_ROWS` and `MAX_LCM_COLS` are constants that represent the number of rows and the number of columns of the LCM display, respectively.

### Keypad Functions

Moxa’s UC-7400 and DA-660 series embedded computers are equipped with keypad buttons for your touch-and-control applications. The UC-7420 and UC-7410 computers have five keypad buttons and the DA-660 series have four keypad buttons.

![Figure 3-1: UC-7420](image)

![Figure 3-2: DA660](image)

When programming keypad buttons, developers need to recognize that F1 to F5 of the UC-7420 and UC-7410 computers (as shown in Figure 3-1) are represented by key numbers 0 to 4, respectively, and for the DA-660 series computers (Figure 3-2), MENU, up arrow, down arrow, and SEL are represented by key numbers 0, 3, 2, and 1, respectively.

Your programs drive these keypad buttons by opening, writing data in, reading data from, and controlling a device file whose location is hidden by the API function `keypad_open`. The following sections will describe each keypad programming function and its usage. The header file of keypad functions is `mxkeypad.h`, located in the `inc/mxphio` directory for Linux, and in the `inc\mxphio` folder for Windows.
### keypad_open function

This function opens a device file associated with the keypad buttons.

```c
int keypad_open(void);
```

**Return Value**

When successful, this function returns a positive value of a file descriptor or a handle representing the keypad buttons. Otherwise, it returns a negative value if the device file does not exist.

### keypad_close function

This function closes a device file associated with the keypad buttons.

```c
void keypad_close(int fd);
```

**Inputs**

- `fd` An open file descriptor or handle representing the keypad buttons.

**Return Value**

None.

### keypad_get_pressed_key function

This function gets the number corresponding to the keypad button that was pressed (zero to four for the UC-7400 series or zero to three for the DA-660 series).

```c
int keypad_get_pressed_key(int fd);
```

**Inputs**

- `fd` An open file descriptor or handle representing the keypad buttons.

**Return Value**

When successful, this function returns a number (starting from 0) when a keypad button is pressed. Otherwise it returns a negative value if there is an error in the device.

### RTC Functions

RTC (real time clock) is the hardware clock. It is an integrated circuit used by a computer or device to keep track of the current time. The functions in this section can be used to get the current hardware clock time, set the hardware clock time to a specified time, set the hardware clock time to the system time, or set the system time to the hardware clock time. The header file of RTC functions is `mxrtc.h`, located in the `inc/mxphio` directory for Linux, and in the `inc\mxphio` folder for Windows.

### mxrtc_ge function

This function gets the current hardware clock time.

```c
int mxrtc_get(unsigned int *t);
```

**Inputs**

- `t` Pointer to an empty array (at least 6 elements) of time values: year, month, day, hour, minute, and second.

**Return Value**

This function returns 0 if successful. Otherwise, there is an error.
mxrtc_set function

This function sets the hardware clock time to a specified time.

```c
int mxrtc_set(unsigned int *t);
```

Inputs

- `t` Pointer to time values: year, month, day, hour, minute, and second.

Return Value

When successful, this function returns 0. Otherwise, there is an error.

mxrtc_set_system_time function

This function sets the hardware clock time to the system time.

```c
int mxrtc_set_system_time(void);
```

Inputs

None.

Return Value

When successful, this function returns 0. Otherwise, there is an error.

mxrtc_get_system_time function

This function sets the system time to the hardware clock time.

```c
int mxrtc_get_system_time(void);
```

Inputs

None.

Return Value

When successful, this function returns 0. Otherwise, there is an error.

Watchdog Timer Functions

For certain fault conditions, such as when the system hangs or your program cannot complete an operation successfully, you will need the system to return to normal operation. The watchdog timer can be used for this purpose. The following functions allow you to implement a watchdog timer in your program. The header file of watchdog timer functions is `mxwdg.h`, which is in the `inc/mxphio` directory for Linux, and in the `inc\mxphio` folder for Windows.

mxwdg_open function

This function starts a watchdog timer.

```c
int mxwdg_open(unsigned long time);
```

Inputs

- `time` Specifies an exact time period in milliseconds. Note that the watchdog timer needs to be refreshed to avoid reset.

Return Value

When successful, this function returns a positive value representing a file descriptor. Otherwise, it returns a negative value.
**mxwdg_close function**

This function stops the watchdog timer.

```c
void mxwdg_close(int fd);
```

**Inputs**

- `fd` Specifies the file descriptor of the watchdog timer.

**Return Value**

None.

**mxwdg_refresh function**

This function refreshes the watchdog timer, and should be called periodically during timer timeouts.

```c
int mxwdg_refresh(int fd);
```

**Inputs**

- `fd` Specifies the file descriptor of the watchdog timer.

**Return Value**

When successful, this function returns 0. Otherwise, it returns a negative value.

---

**DIO Functions**

Digital input/output channels are featured in some models of Moxa embedded computers, including the UC-7408, UC-8410, IA240, IA260, and W406. These channels can be accessed at run-time for control or monitoring using the functions in the following sections. Digital Output channels can be set to high or low via each port starting from 0. The Digital Input channels can be used to detect the state change of the digital input signal. The header file of digital I/O functions is `mxdgio.h`, which is located in the `inc/mxphio` directory for Linux, and in the `inc\mxphio` folder for Windows.

**mxdgio_open function**

This function opens an access to the DIO device.

```c
HANDLE mxdgio_open(void);
```

**Inputs**

None.

**Return Value**

When successful, this function returns an access to the DIO device. Otherwise, there is an error.

**mxdgio_close function**

This function closes the access to the DIO device.

```c
void mxdgio_close(HANDLE fd);
```

**Inputs**

- `fd` The access to the device.

**Return Value**

None.
mxgdgio_get_input_signal function

This function gets the signal state of a digital input channel.

```
int mxgdgio_get_input_signal(HANDLE fd, int port);
```

Inputs

- **fd**: The access to the device.
- **port**: Port #

Return Value

Returns 1 for a high signal or 0 for a low signal, if successful. Otherwise, it returns a value of -1.

mxgdgio_get_output_signal function

This function gets the signal state of a digital output channel.

```
int mxgdgio_get_output_signal(HANDLE fd, int port);
```

Inputs

- **fd**: The access to the device.
- **port**: Port number

Return Value

Returns 1 for a high signal or 0 for a low signal, if successful. Otherwise, it returns a value of -1.

mxgdgio_set_output_signal_high function

This function sets a high signal to a digital output channel.

```
int mxgdgio_set_output_signal_high(HANDLE fd, int port);
```

Inputs

- **fd**: The access to the device.
- **port**: Port number

Return Value

When successful, this function returns 0. When an error occurs, it returns -1.

mxgdgio_set_output_signal_low function

This function sets a low signal to a digital output.

```
int mxgdgio_set_output_signal_low(HANDLE fd, int port);
```

Inputs

- **fd**: The access to the device.
- **port**: Port #

Return Value

When successful, this function returns 0. When an error occurs, it returns -1.

Serial Functions

The Moxa peripheral I/O library provides a set of functions to set and get serial parameters, including baudrate, data bits, stop bits, parity, interface, and flow control, and to read and write data. The header file of all serial functions is `mserial_port.h`, located in the `inc/mxphio` directory for Linux, and in the `inc\mxphio` folder for Windows.
mxsp_open function

This function opens a serial port.

\[ \text{unsigned int mxsp_open(int port);} \]

**Inputs**
- \( port \) Port number (starting from 1).

**Return Value**
When successful, this function returns the handle of the open port. When an error occurs, it returns a negative value.

mxsp_close function

This function closes a serial port.

\[ \text{int mxsp_close(unsigned int fd);} \]

**Inputs**
- \( fd \) Handle of the open port.

**Return Value**
When successful, this function returns 0. When an error occurs, it returns a negative value.

mxsp_set_baudrate function

This function sets the communication baudrate of a serial port.

\[ \text{int mxsp_set_baudrate (unsigned int fd, int baudrate);} \]

**Inputs**
- \( fd \) Handle of the open port.
- \( baudrate \) Specified baudrate.

**Return Value**
When successful, this function returns 0. If it does not return 0, you can consider it to be unsuccessful.

mxsp_get_baudrate function

This function gets the baudrate of a serial port.

\[ \text{int mxsp_get_baudrate(unsigned int fd);} \]

**Input**
- \( fd \) Handle of the open port.
- \( baudrate \) Specified baudrate.

**Return Value**
When successful, this function returns a positive value. This function returns a negative value when an error occurs.
# mxsp_set.databits function

This function sets the number of data bits for communication through a serial port.

```c
int mxsp_set_databits(unsigned int fd, int bits);
```

**Inputs**
- `fd` Handle of the open port.
- `bits` 5, 6, 7, or 8

**Return Value**
When successful, this function returns 0.

# mxsp_get.databits function

This function gets the number of data bits in for communication through a serial port.

```c
int mxsp_get_databits(unsigned int fd);
```

**Inputs**
- `fd` Handle of the open port.

**Return Value**
When successful, this function returns the number of bits (5, 6, 7, or 8).

# mxsp_set.stopbits function

This function sets the number of stop bits for communication through a serial port.

```c
int mxsp_set_stopbits(unsigned int fd, int bits);
```

**Inputs**
- `fd` Handle of the open port.
- `bits` stop bits; 1, 2, or 3 (for 1.5)

**Return Value**
When successful, this function returns 0.

# mxsp_get.stopbits function

This function gets the number of stop bits for communication through a serial port.

```c
int mxsp_get_stopbits(unsigned int fd);
```

**Inputs**
- `fd` Handle of the open port.

**Return Value**
When successful, this function returns the number of stop bits (1, 2, or 3).

# mxsp_set.parity function

This function sets the parity of a serial port.

```c
int mxsp_set_parity(unsigned int fd, int parity);
```

**Inputs**
- `fd` Handle of the open port.
- `parity` 0: none, 1: odd, 2: even, 3: space, 4: mark, otherwise: none

**Return Value**
When successful, this function returns 0.
### mxsp_get_parity function

This function gets the parity for a serial port.

```c
int mxsp_get_parity(unsigned int fd);
```

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fd</code></td>
<td>Handle of the open port.</td>
</tr>
</tbody>
</table>

**Return Value**

When successful, this function returns the parity of the open port (0, 1, 2, 3, or 4).

### mxsp_set_flow_control function

This function sets the type of flow control for a serial port.

```c
int mxsp_set_flow_control(unsigned int fd, int mode);
```

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fd</code></td>
<td>Handle of the open port.</td>
</tr>
<tr>
<td><code>mode</code></td>
<td>1: software, 2: hardware, otherwise: none</td>
</tr>
</tbody>
</table>

**Return Value**

When successful, this function returns 0.

### mxsp_get_flow_control function

This function gets the type of flow control of a serial port.

```c
int mxsp_get_flow_control(unsigned int fd);
```

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fd</code></td>
<td>Handle of the open port.</td>
</tr>
</tbody>
</table>

**Return Value**

When successful, this function returns the type of flow control being used by that port. The function returns a negative value when an error occurs.

### mxsp_set_interface function

This function sets the type of communication interface for a serial port.

```c
int mxsp_set_interface(unsigned int fd, int mode);
```

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fd</code></td>
<td>Handle of the open port.</td>
</tr>
<tr>
<td><code>mode</code></td>
<td>0: RS232, 1: RS485_2WIRE, 2: RS422, 3: RS485_4WIRE</td>
</tr>
</tbody>
</table>

**Return Value**

When successful, this function returns 0.
**mxsp_get_interface function**

This function gets the type of communication interface of a serial port.

```c
int mxsp_get_interface(unsigned int fd);
```

**Inputs**
- `fd` Handle of the open port.

**Return Value**
When successful, this function returns one of the values from 0 to 3. The function returns a negative value when an error occurs.

**mxsp_purge_buffer function**

This function clears a serial port’s data buffer.

```c
int mxsp_purge_buffer(unsigned int fd, int purge);
```

**Inputs**
- `fd` Handle of the open port.
- `purge` 0: received buffer, 1: transmitting buffer, 2: both

**Return Value**
When successful, this function returns 0.

**mxsp_set_nonblocking function**

For Linux programming, this function sets the open port to operate in non-blocking mode.

```c
void mxsp_set_nonblocking(unsigned int fd);
```

**Inputs**
- `fd` Handle of the open port.

**mxsp_read function**

This function reads data into a buffer from an open serial port in non-blocking mode for Windows. For Linux, if the function `mxsp_set_nonblocking` is called, it reads in non-blocking mode; otherwise it reads in blocking mode.

```c
mxsp_read( unsigned int fd, char *buffer, int size, void *dummy);
```

**Inputs**
- `fd` Handle of the open port.
- `buffer` Pointer to the buffer.
- `size` Maximum size to be read.
- `dummy` Reserved argument.

**Return Value**
If it fails, this function returns a negative value. If no data is read in non-blocking mode, it returns 0; otherwise, it returns the number of bytes read.
**mxsp_blocking_read function**

This function reads data into a buffer from an open serial port in blocking mode (for Windows only).

```
mxsp_blocking_read( unsigned int fd, char *buffer, int size, void *dummy);
```

**Inputs**

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fd</code></td>
<td>Handle of the open port.</td>
</tr>
<tr>
<td><code>buffer</code></td>
<td>Pointer to the buffer.</td>
</tr>
<tr>
<td><code>size</code></td>
<td>Maximum size to be read.</td>
</tr>
<tr>
<td><code>dummy</code></td>
<td>Reserved argument.</td>
</tr>
</tbody>
</table>

**Return Value**

If it fails, this function returns a negative value. Otherwise, it returns the number of bytes read.

**mxsp_write function**

This function writes data from a buffer to an open serial port.

```
mxsp_write(unsigned int fd, char *buffer, int size, void *dummy);
```

**Inputs**

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fd</code></td>
<td>Handle of the open port.</td>
</tr>
<tr>
<td><code>buffer</code></td>
<td>Pointer to the buffer.</td>
</tr>
<tr>
<td><code>size</code></td>
<td>Size of data to be written.</td>
</tr>
</tbody>
</table>

**Return Value**

If it fails, this function returns a negative value. Otherwise, it returns the number of bytes written.

**Network Interface Functions**

Network interface functions provide interfaces for manipulating parameters for an embedded computer’s network interfaces and DNS servers. Network interface parameters include interface name, MAC address, IP address, default gateway address, network address, broadcast address, and a flag for static or dynamic IP. The parameters are included in the definition of the `MXIFACE` data structure.

```c
typedef struct _MXIFACE
{
    char ifname[32];
    unsigned char macaddr[8];
    char ipaddr[16];
    char gateway[16];
    char netmask[16];
    char network[16];
    char broadaddr[16];
    char dhcpsrv[16];
    unsigned int enable_dhcp;
} MXIFACE;
```

The network interface parameters can be obtained from the runtime and interface files for Linux, or from the registry settings for Windows. Windows registry settings take effect at bootup or when the network interface is restarted. Linux parameters can be updated in both the runtime and interface files, but the Windows runtime parameters for the network interface cannot be updated directly. The header file for network interface functions is `netiface.h`, which is located in the `inc/mxphio` directory for Linux, or in the `inc\mxphio` folder for Windows.
**mxiface_get_info function**

This function gets the interface configuration from the runtime.

```c
int mxiface_get_info(char *ifname, MXIFACE *ifaces, int max);
```

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ifname</td>
<td>The name of the interface; NULL for all interfaces.</td>
</tr>
<tr>
<td>max</td>
<td>The maximum number of interfaces.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Outputs</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ifaces</td>
<td>The buffer containing interface information.</td>
</tr>
</tbody>
</table>

**Return Value**
The number of interfaces.

---

**mxiface_get_info_from_file function**

This function gets the interface configuration from the interface file for Linux.

```c
int mxiface_get_info_from_file(char *ifname, MXIFACE *ifaces, int max);
```

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ifname</td>
<td>The name of the interface; NULL for all interfaces.</td>
</tr>
<tr>
<td>max</td>
<td>The maximum number of interfaces.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Outputs</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ifaces</td>
<td>The buffer containing interface information.</td>
</tr>
</tbody>
</table>

**Return Value**
The number of interfaces.

---

**mxiface_get_info_static function**

This function gets the interface configuration from registry settings for Windows CE.

```c
int mxiface_get_info_static(char *ifname, MXIFACE *ifaces, int max);
```

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ifname</td>
<td>The name of the interface; NULL for all interfaces.</td>
</tr>
<tr>
<td>max</td>
<td>The maximum number of interfaces.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Outputs</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ifaces</td>
<td>The buffer containing interface information.</td>
</tr>
</tbody>
</table>

**Return Value**
The number of interfaces.

---

**mxiface_get_mac_address function**

This function gets the MAC address of an interface.

```c
int mxiface_get_mac_address(char *ifname, unsigned char *mac);
```

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ifname</td>
<td>The name of the interface.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Outputs</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mac</td>
<td>MAC address</td>
</tr>
</tbody>
</table>

**Return Value**
When successful, this function returns 0.
mxiface_update_info function

This function updates the runtime configuration of a Linux network interface.

```c
int mxiface_update_info(MXIFACE *iface);
```

<table>
<thead>
<tr>
<th>Inputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>iface</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>When successful, this function returns 0; otherwise, you can consider it to be a failure.</td>
</tr>
</tbody>
</table>

mxiface_update_interface_file function

This function updates the configuration of an interface into the Linux file or Windows registry settings.

```c
int mxiface_update_interface_file(MXIFACE *ifr);
```

<table>
<thead>
<tr>
<th>Inputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>iface</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>When successful, this function returns 0.</td>
</tr>
</tbody>
</table>

mxhost_get_dns function

This function gets the list of DNS servers from a networking interface.

```c
int mxhost_get_dns(char *ifname, char *dns_list, int size);
```

<table>
<thead>
<tr>
<th>Inputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>ifname</td>
</tr>
<tr>
<td>dns_list</td>
</tr>
<tr>
<td>size</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>dns_list</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>When successful, this function returns 0.</td>
</tr>
</tbody>
</table>

mxhost_update_dns function

This function updates the list of DNS servers to a networking interface.

```c
int mxhost_update_dns(char *ifname, char *dns_list);
```

<table>
<thead>
<tr>
<th>Inputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>ifname</td>
</tr>
<tr>
<td>dns_list</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>When successful, this function returns 0.</td>
</tr>
</tbody>
</table>

mxiface_restart function

This function restarts the network interfaces.

```c
int mxiface_restart(void);
```

<table>
<thead>
<tr>
<th>Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>When successful, this function returns 0; otherwise, you can consider it to be a failure.</td>
</tr>
</tbody>
</table>
Socket Functions

Moxa’s embedded computers are equipped with network interfaces that allow your client-server applications to communicate with each other across a computer network. These applications are normally implemented in socket programming with TCP or UDP protocols. Moxa provides a set of socket functions to help you with your software development. The header file of all socket functions is network.h, which is located in the inc/mxphio directory for Linux, or in the inc\mxphio folder for Windows.

net_library_init function

This function initiates the socket library for Windows programming. This function must be called before the user program can use the associated socket functions in the library.

```c
void net_library_init(void);
```

net_library_release function

This function releases the socket library for Windows programming.

```c
void net_library_release(void);
```

tcp_make_client function

This function initiates a TCP client connection to a server program.

```c
int tcp_make_client (const char *host, int *port, unsigned int *ip, int nonblocking);
```

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>host</td>
<td>Pointer to the name of the server.</td>
</tr>
<tr>
<td>port</td>
<td>The listening port of the server.</td>
</tr>
<tr>
<td>ip</td>
<td>The network byte ordered IP address of the server; if &lt;host&gt; is a NULL value, use the ip value.</td>
</tr>
<tr>
<td>nonblocking</td>
<td>Non-zero value for a non-blocking socket.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Outputs</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ip</td>
<td>The ip value of the server.</td>
</tr>
<tr>
<td>port</td>
<td>The local port of the client connection.</td>
</tr>
</tbody>
</table>

Return Value

When successful, this function returns a positive value of a file descriptor representing a communication socket. When an error occurs, it returns a negative value.

tcp_startup_server function

This function starts up a TCP server on a specified listening port.

```c
int tcp_startup_server(unsigned int ip, int port);
```

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ip</td>
<td>The network byte ordered IP address that the server binds to; a value of zero means that it can bind to any interface.</td>
</tr>
<tr>
<td>port</td>
<td>The listening port of the server.</td>
</tr>
</tbody>
</table>

Return Value

When successful, this function returns a positive value of a file descriptor representing a communication socket. When an error occurs, it returns a negative value.
tcp_accept_client function

The server program accepts a TCP client connection and assigns a socket to the connection.

```c
int tcp_accept_client (int fd, int *port, unsigned int *ip, int nonblocking);
```

**Inputs**

- `fd`: The open socket of the server.
- `nonblocking`: Non-zero value for a client socket in non-blocking mode.

**Outputs**

- `port`: The local port of the client socket.
- `ip`: The IP address of the client machine that is making the request for this connection.

**Return Value**

When successful, this function returns a positive value of a file descriptor for the client socket. When an error occurs, it returns a negative value.

tcp_nonblocking_write function

This function sends data to the communication peer on a non-blocking TCP socket.

```c
int tcp_nonblocking_write(int fd, char *buf, int len, void *dummy);
```

**Inputs**

- `fd`: The TCP socket.
- `buf`: Pointer to the data to be sent.
- `len`: The length of the data.
- `dummy`: ignored

**Return Value**

This function returns the number of bytes sent. When an error occurs, it returns a negative value.

tcp_nonblocking_read function

This function reads data from a connected non-blocking TCP socket.

```c
int tcp_nonblocking_read(int fd, char *buf, int size, void *dummy);
```

**Inputs**

- `fd`: The TCP socket.
- `buf`: Pointer to a buffer for incoming data.
- `size`: The size of the buffer.
- `dummy`: ignored

**Return Value**

This function returns the number of bytes received. When an error occurs, it returns a negative value.
tcp_nonblocking_wait function

This function waits for a just connected non-blocking TCP socket to be available for reading or writing. It is used after calling tcp_make_client for a non-blocking TCP client connection.

```
int tcp_nonblocking_wait(int fd, int sec, int write);
```

**Inputs**

- `fd` The TCP socket.
- `sec` The number of seconds to wait.
- `write` Non-zero value to represent a wait for writing, or a zero value to represent a wait for reading.

**Return Value**

When the socket is ready for reading or writing before the time is up, this function returns a positive value. When a connection is not established or an error occurs, this function returns 0.

```
#define tcp_nonblocking_writeable(fd,sec) tcp_nonblocking_wait(fd,sec,1)
#define tcp_nonblocking_readable(fd,sec) tcp_nonblocking_wait(fd,sec,0)
```

udp_make_client function

This function establishes a UDP client connection to a server.

```
int udp_make_client(char *host, int port, struct sockaddr_in *addr_in);
```

**Inputs**

- `host` The name of the server host.
- `port` The listen port of the server.

**Outputs**

- `addr_in` Pointer to a data structure that contains the information of the server.

**Return Value**

When successful, this function returns a positive value of a file descriptor representing the client socket. When an error occurs, it returns a negative value.

udp_startup_server function

This function starts up a UDP server.

```
int udp_startup_server(unsigned int ip, int port, struct sockaddr_in * addr_in);
```

**Inputs**

- `ip` The network byte ordered IP address that the server binds to; a value of zero means that it can bind to any interface.
- `port` The listen port of the server.

**Outputs**

- `addr_in` Pointer to a data structure that contains the information of the server.

**Return Value**

When successful, this function returns a positive value of a file descriptor representing the client socket. When an error occurs, it returns a negative value.
udp_send function

This function sends data to the communication peer on a UDP socket.

```c
int udp_send(int fd, char *buf, int len, struct sockaddr_in *peer);
```

<table>
<thead>
<tr>
<th>Inputs</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fd</code></td>
</tr>
<tr>
<td><code>buf</code></td>
</tr>
<tr>
<td><code>len</code></td>
</tr>
<tr>
<td><code>peer</code></td>
</tr>
</tbody>
</table>

Return Value

This function returns the number of bytes sent. When an error occurs, it returns a negative value.

udp_recv function

This function receives data from a UDP socket.

```c
int udp_recv(int fd, char *buf, int size, struct sockaddr_in *peer);
```

<table>
<thead>
<tr>
<th>Inputs</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fd</code></td>
</tr>
<tr>
<td><code>buf</code></td>
</tr>
<tr>
<td><code>size</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>peer</code></td>
</tr>
</tbody>
</table>

Return Value

This function returns the number of bytes received. When an error occurs, it returns a negative value.

IP Multicast Functions

The IP multicast transmission method is used for efficient one-to-many communications, and is different from unicast and broadcast. Class D addresses from 224.0.0.0 to 239.255.255.255 are designated as multicast addresses. The sender sends a single datagram to the multicast address, and the intermediary routers automatically make copies and send them to all receivers that have registered with the multicast address. IP multicast support is included in most embedded computer models. Developers can easily use IP multicast programming functions in sample code to write their own IP multicast programs. The header file of all IP multicast functions is `mcast.h`, which is located in the `inc/mxphio` directory for Linux, and in the `inc\mxphio` folder for Windows.

mcast_make_client function

This function creates a multicast client (sender) socket.

```c
SOCKET mcast_make_client(int onoff, char *grp, int port, struct sockaddr_in *addr);
```

<table>
<thead>
<tr>
<th>Inputs</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>onoff</code></td>
</tr>
<tr>
<td><code>grp</code></td>
</tr>
<tr>
<td><code>port</code></td>
</tr>
<tr>
<td><code>addr</code></td>
</tr>
</tbody>
</table>

Return Value

When successful, this function returns a socket `fd`. Otherwise, there is an error.
mcast_make_server function

This function creates a multicast server (receiver) socket.

```c
SOCKET mcast_make_server(char *grp, int port, struct sockaddr_in *addr);
```

**Inputs**

- **grp**: The multicast group address.
- **port**: The multicast port number.
- **addr**: Pointer to the multicast socket address structure.

**Return Value**

When successful, this function returns a socket fd. Otherwise, there is an error.

mcast_join function

This function joins a multicast group as a multicast receiver.

```c
int mcast_join(SOCKET fd, char *grp, struct ip_mreq *mreq);
```

**Inputs**

- **fd**: The socket fd.
- **grp**: The multicast group address.
- **mreq**: Pointer to the multicast request structure.

**Return Value**

When successful, this function returns 0. Otherwise, it returns -1 when a failure occurs.

mcast_leave function

This function leaves a multicast group as a multicast receiver.

```c
int mcast_leave(SOCKET fd, struct ip_mreq *mreq);
```

**Inputs**

- **fd**: The socket fd.
- **mreq**: Pointer to the multicast request structure.

**Return Value**

When successful, this function returns 0. Otherwise, it returns -1 when a failure occurs.
In this chapter a set of higher layer functions for LCM/Keypad using LCM/Keypad peripheral I/O functions and related sample programs are introduced. In addition, a sample program defining a set of higher layer functions for DIO using DIO peripheral I/O functions is presented.

Keypad Callback Functions

In the previous chapter we have described some preliminary keypad functions, including `keypad_open` and `keypad_get_pressed_key`. These functions give programmers full access to the keypad buttons from within a program.

In addition to these preliminary functions, we provide programmers with another set of API functions that allow user applications to focus on event handling when keypad buttons are pressed. A callback function is defined to associate with an event. The header file of all keypad callback functions is `keypad_callback.h`, which is located in the `inc/mxphio` directory Linux or the `inc\mxphio` folder for Windows.

**keypad_callback_init function**

This function registers a handle for user-defined callback operations on the keypad buttons. It must be called before any other function is called.

```c
KEYPAD *keypad_callback_init(void);
```

**Return Value**

If successful, it returns a positive value for the handle, or it returns 0 if the system falls short on memory allocation.

**keypad_callback_add function**

This function associates a user-defined function with a keypad button. This function will be called when the keypad button is pressed.

```c
int keypad_callback_add(KEYPAD *hndl, int key, keypad_callback_t func, void *arg);
```

**Inputs**

- `hndl`: The open handle representing a set of callback operations.
- `key`: The key number of the keypad button; note that key numbers start from 0.
- `func`: User-defined function.
- `arg`: The user-defined parameter of the callback function.

**Return Value**

Returns 0 if successful.

**Remarks**

The user-defined callback function has prototype `func(void *arg, int key)` where parameter `<arg>` is the user-defined argument for this function, and parameter `<key>` is the key number of the keypad button.
keypad_callback_dispatch function

This function forces the user program to enter an infinite loop, where the program dispatches the logic flow to
call the associated user-defined function when a keypad button is pressed.

```c
void keypad_callback_dispatch(KEYPAD *hndl);
```

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>hndl</code></td>
<td>The open handle representing a set of callback operations.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td></td>
</tr>
</tbody>
</table>

keypad_callback_quit function

This function forces all keypad operations to stop.

```c
void keypad_callback_quit(KEYPAD *hndl);
```

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>hndl</code></td>
<td>The open handle representing a set of callback operations.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Remarks</th>
<th>Description</th>
</tr>
</thead>
</table>
| If these keypad callback operations are part of a user program, the program can create a child thread to handle
the function keypad_callback_dispatch and leave the control rights to the main thread to stop the dispatch
(or thread) if necessary. |

LCM/Keypad Context Functions

To enhance the usage of the keypad callback functions described in the previous section, a set of context
functions to combine LCM and keypad callback functions are provided in this section. Developers can use these
functions to associate keypad callback function with an LCM display. The header file of all LCM/Keypad context
functions is keypad_lcm.h, which is in the inc/mxphio directory for Linux, or the inc\mxphio folder for Windows.

keypad_lcm_menu_init function

This function initiates a context of operating keypad buttons and LCM display.

```c
LCMKPD *keypad_lcm_menu_init(void);
```

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>None.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Description</th>
</tr>
</thead>
</table>
| When successful, this function returns a positive handle value representing a context of operating keypad
buttons and LCM display. When an error occurs, this function returns 0. |

keypad_lcm_menu_add function

This function adds a set of user-defined functions, each of which handles the data content of the LCM display when a keypad button is pressed.

```c
int keypad_lcm_menu_add(LCMKPD *hndl, lcm_show_t *funs, void *data);
```

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hndl</td>
<td>The context of operations.</td>
</tr>
<tr>
<td>funs</td>
<td>Pointer to an array of user-defined functions. The number of functions is equivalent to the maximum number of keypads in the embedded computers. A user-defined function is called when an associated keypad is pressed. For example, when the first keypad button (&quot;MENU&quot; of a DA-660 series computer or &quot;F1&quot; of a UC-7400 series computer) is pressed, the user-defined function, funs[0] is called. Each user-defined function has the following prototype:</td>
</tr>
<tr>
<td>data</td>
<td>the argument of the functions</td>
</tr>
</tbody>
</table>

Return Value

When successful, this function returns 0. When an error occurs, this function returns a non-zero value.

keypad_lcm_menu_dispatch function

This function detects pressed keypad button events; when a keypad button is pressed, the function will then call an associated user-defined function. When it is called, the program enters an infinite loop. The caller can initiate a thread for this function and call keypad_lcm_menu_quit to quit the dispatch function if necessary.

```c
void keypad_lcm_menu_dispatch(LCMKPD *hndl);
```

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hndl</td>
<td>The context of operations.</td>
</tr>
</tbody>
</table>

Return Value

None.

keypad_lcm_menu_quit function

Call this function to jump out of the infinite dispatch loop.

```c
void keypad_lcm_menu_quit(LCMKPD *hndl);
```

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hndl</td>
<td>The context of operations.</td>
</tr>
</tbody>
</table>

Return Value

None.
LCM/Keypad Sample Program

This LCM/Keypad sample program shows how a user can change the networking IP addresses of the embedded computers by simply pressing the keypad buttons, without needing to log on to the computers. The LCM is used to display the setting changes and the final results. The source code files of the sample program are in the samples/mxphio/keypad_lcm directory for Linux or in the samples\mxphio\keypad_lcm folder for Windows.

We use the DA-660 to illustrate. This embedded computer has 4 keypad buttons: MENU, up arrow, down arrow, and SEL, which are recognized by the computer internally as key 0, 3, 2, and 1, respectively. In the LCM display area, which accommodates 2x16 characters, this example shows an IP address for a LAN port in the format LAN:eth0 in the first row and xxx.xxx.xxx.xxx in the second row.

The user presses the "MENU" button to select a LAN port from two LAN ports of the DA-660 computer. The IP address of the selected LAN port will be displayed on the LCM screen. The down arrow button is used to shift the cursor to each digit of the IP address. Pressing the up arrow button increases a digital value. When finished, press the "SEL" button to activate the IP address shown on the LCM screen.

The main advantage of the LCM/Keypad programming function is that it’s easy to use and the design is based on four user-defined functions with a corresponding keypad button for each one. These functions are ipaddr_get, ipaddr_set, ipaddr_digit_shift, and ipaddr_digit_up. For each LAN port, these functions must be registered once by calling keypad_lcm_menu_add.

```c
int ipaddr_menu_init(LCMKPD *hndl)
{
    int i, max;
    lcm_show_t funs[4];
    MXIFACE gIPIfaces[MAX_IFACES];
    /* get network interfaces */
    memset(gIPIfaces, 0, MAX_IFACES*sizeof(MXIFACE));
    max = mxiface_get_info_from_file(NULL, gIPIfaces, MAX_IFACES);
    if (max < 0)
    {
        printf("fail to get interfaces\n");
        return -1;
    }
    funs[0] = ipaddr_get;
    funs[1] = ipaddr_set;
    funs[3] = ipaddr_digit_up;
    /* based on interfaces, each carries an IPADDR */
    for (i=0; i < max ; i++)
    {
        strcpy(gIPAddrs[i].ifname, gIPIfaces[i].ifname);
        if (keypad_lcm_menu_add(hndl, funs, &gIPAddrs[i]))
            return -1;
    }
    return 0;
}
```

The variable gIPAddr is a global data structure array. Each array item contains a variable ifname for the interface name of the LAN port, a double character array addr representing the 12 digits of an IP address, and two integer values digit_i and digit_j to trace the position of the LCM cursor within the 12 digits.

```c
#define MAX_IFACES 8 /* assume a maximum # of interfaces */
/* about a network interface */
typedef struct _IPADDR
{
    char ifname[16]; /* the name of the interface */
```
unsigned char addr[4][3]; /* ip address in digits */
int digit_i, digit_j; /* position */
} IPADDR;
/* global structures */
static IPADDR gIPAddrs[MAX_IFACES];

In this example, each user-defined function calls a common function, as shown below, to enter data for an IP
address that the LCM displays.

static void ipaddr_flush2lcm(LCMDAT *lcm, IPADDR *ip, char *msg)
{
  sprintf(lcm->text[0], "LAN:%s", ip->ifrname);
  if (msg)
    sprintf(lcm->text[1], "%s", msg);
  else
  {
      int i;
      / * convert IP to a string style shown on LCM */
      for (i=0; i < 4; i++)
       sprintf(lcm->text[1]+4*i, "%d%d%d\",
          ip->addr[i][0],ip->addr[i][1],ip->addr[i][2]);
      lcm->text[1][15] = 0;
  }
  lcm->cursor_y = 1;
  lcm->cursor_x = ip->digit_i*4+ip->digit_j;
}

DIO Sample Program

This DIO sample program shows how users can develop a set of higher layer functions using preliminary DIO
functions from the peripheral I/O library. These functions allow user applications to focus on event handling
when events happen. A callback function is defined by the programmer to associate with an event. The source
code files of the sample program are located in the samples/mxphio/digit_input_change
directory for Linux, or
the samples\mxphio\digit_input_change folder for Windows.

Four higher layer functions, digit_io_timer_init, digit_io_timer_dispatch,
digit_io_timer_add_callback, and digit_io_timer_dispatch_quit, are provided. Four callback functions
in the sample are added for four different events: DGTIO_GET_INPUT_STATE_CHANGE,
DGTIO_GET_INPUT, DGTIO_GET_OUTPUT, and DGTIO_SET_OUTPUT, via the
digit_io_timer_add_callback function.

mngr = digit_io_timer_init();
...
if (digit_io_timer_add_callback(mngr, port, DGTIO_GET_INPUT_STATE_CHANGE, interval,
      input_chg_cb, &port) < 0) { 
  ...
}
if (digit_io_timer_add_callback(mngr, port, DGTIO_GET_INPUT, interval, input_get_cb, &port) <
    0) { 
  ...
}
if (digit_io_timer_add_callback(mngr, port, DGTIO_SET_OUTPUT, interval, output_set_cb, &port)
    < 0) { 
  ...
}
if (digit_io_timer_add_callback(mngr, port, DGTIO_GET_OUTPUT, interval, output_get_cb, &port)
< 0) {
    ...
}
digit_io_timer_dispatch(mngr);
In Moxa’s sample code the higher layer application library contains a connection library developed by Moxa for TCP, UDP, and serial programming, using corresponding functions in the peripheral I/O library. The connection library provides a framework that makes it easier for developers to write socket and serial programs. In this chapter, we introduce user-defined functions, data structures, and API functions of the connection library. In the next chapter, sample programs using the connection library are introduced. To begin with, we define what a connection is.

A connection refers to when data packet transmission is established between a source device and a target device. Currently, networking and serial connections is a major segment of data transmission between an embedded computer and other devices.

The following enumeration in C language differentiates the types of supported connections.

```c
enum {
    CONNECTION_TYPE_NONE,  /* a client connection accepted by a TCP server */
    CONNECTION_TYPE_TCPACCEPT,  /* a connection of a TCP server */
    CONNECTION_TYPE_TCPCLIENT,  /* a TCP client making the connection */
    CONNECTION_TYPE_UDPCLIENT,  /* a UDP client making the connection */
    CONNECTION_TYPE_UDPSERVER,  /* a connection of a UDP server */
    CONNECTION_TYPE_UARTPORT,  /* a connection of a serial port */
};
```

The following set of programming functions consolidates each connection to handle four states: open, packet consumption, packet generation, and close. The functions hide the detailed socket implementation and serial port programming, allowing programmers to focus on implementing user-defined functions for the four connection states. The connection library header file of is `connection.h`, which is located in the `inc/mxconn` directory for Linux, or in the `inc\mxconn` folder for Windows.
User-defined Functions

The five user-defined functions open, dispatch, close, timer, and initialization are described in this section.

**open user-defined function**

This is the prototype of a user-defined function that is called after a connection is created or accepted.

```
typedef void* (*conn_open_t) (MHANDLE conp, void *param, unsigned int *timer_interval);
```

**Inputs**
- `conp`: Specifies the connection.
- `param`: Pointer to a data structure that contains the information, such as the listening port and IP address, of the connection. The information content will vary depending on the type of connection.

**Outputs**
- `timer_interval`: Specifies the period (in milliseconds) of a timer. This timer will call an associated user-defined function periodically.

**Return Value**
Return value that represents a user-defined data area or a parameter that will be used as an argument for other user-defined functions.

**dispatch user-defined function**

This is the prototype of a user-defined function that is called after a connection receives a data packet, or if there is still un-processed data.

```
typedef int (*conn_dispatch_t) (MHANDLE conp, void *private_data, DATAPKT *dpkt);
```

**Inputs**
- `conp`: Specifies the connection.
- `private_data`: Pointer to the user-defined data.
- `dpkt`: Pointer to a data structure that represents the data packet, including the data and the length of the data. Refer to the detailed description of the data structure in the following section.

**Outputs**
- `dpkt`: Specifies the number of bytes in the data packet that have been consumed.

**Return Value**
There are different options for this return value. First, if something is found to be wrong and this connection must be disconnected, then return a negative value. Second, if this function consumes a data frame in the packet and there is data left, then return a positive value. A positive return value forces this user-defined function to be called again until the packet is completely consumed. Lastly, return 0 when the un-processed frame is an incomplete data frame.

**close user-defined function**

This is the prototype of a user-defined function that is called when a connection is disconnected.

```
typedef void (*conn_close_t) (MHANDLE conp, void *private_data);
```

**Inputs**
- `conp`: Specifies the connection.
- `private_data`: Pointer to the user-defined data.

**Return Value**
None.
**timer user-defined function**

This is the prototype of a user-defined function that is periodically called if the value of a timer interval was given.

```
typedef void (*conn_timer_t) (void *private_data);
```

**Inputs**

- `private_data`: Pointer to the user-defined data.

**Return Value**

- None.

**initialization user-defined function**

This is the prototype of a user-defined function that is called by API function `connection_dispatch_loop` at the initialization stage of the user program.

```
typedef int (*app_init_cb) (int argc, char **argv);
```

**Inputs**

- `argc`: Specifies the number of arguments when executing the user program.
- `argv`: Pointer to an array of arguments (command line) of the user program.

**Return Value**

- Return 0 to continue the user program. Otherwise, the user program exits.

---

**Data Structures**

Seven data structures used for socket and serial connections are described in the following sections.

**DATAPKT data structure**

This data structure defines a data packet that is received by a connection.

```
typedef struct _DATAPKT {
    char *packet_data;
    int  packet_size;
    int  packet_consumed;
} DATAPKT;
```

**Members**

- `<packet_data>` Specifies the content of the data packet.
- `<packet_size>` Specifies the length of the data packet.
- `<packet_consumed>` Specifies the number of bytes consumed by a user-defined function.

**USERFUN data structure**

This data structure specifies the user-defined functions of a connection.

```
typedef struct _USERFUN {
    conn_open_t open;
    conn_dispatch_t dispatch;
    conn_timer_t timer;
    conn_close_t close;
} USERFUN;
```
Members
<open> This function is called after the connection is established.
<dispatch> This function is called after the connection receives a data packet.
<timer> This function is periodically called if a timer interval is given at the user-defined function <open>.
<close> This function is called before the connection is disconnected.

UARTPRM data structure

The data structure defines the communication settings of a serial port connection.

typedef struct _UARTPRM
{
    unsigned int port;
    unsigned int baudrate;
    unsigned int parity:3;
    unsigned int data_bits:4;
    unsigned int stop_bits:2;
    unsigned int iface_mode:3;
    unsigned int flow_control:2;
    unsigned int xxx:18; /* unused or reserved */
} UARTPRM;

Members
<port> Specifies the port number (starting from 1) of the serial port.
<baudrate> Specifies the baudrate at which the serial port operates; for example ..., 1200, 2400, ..., 9600, ....
<parity> Specifies the parity to be used. Values of 0 to 4 are possible for no, odd, even, mark, and space parity.
<data_bits> Specifies the number of bits (5, 6, 7, 8) in a byte to be transmitted.
<stop_bits> Specifies the number of bits (1, 2, 3 for 1.5 bits) to be used.
<iface_mode> Specifies the type of serial port interface. Use of 0, 1, 2, or 3 to indicate RS232, RS485 (2-wire), RS422, or RS485 (4-wire), respectively.
<flow_control> Specifies the type of flow control during data communication. Use 0, 1, or 2 to represent none, software (XON/XOFF), hardware (RTS) flow control, respectilly.

SRVRPRM data structure

This data structure defines the communication parameters of a TCP server.

typedef struct _SRVRPRM
{
    unsigned int listen_port:16;
    unsigned int max_clients:16;
    unsigned char ip_range[16];
} SRVRPRM;

Members
<listen_port> Specifies the listening port of the TCP server.
<max_clients> Specifies the maximum number of clients allowed; use 0 for an unlimited number.
<ip_range> Specifies the IP range of clients allowed; use 0 for an unlimited number.

TCPCPRM data structure

This data structure defines the communication parameters of a client connection accepted by a TCP server.

typedef struct _TCPCPRM
{
    unsigned int listen_port:16;
    unsigned int local_port:16;
}
unsigned int ip;
} TCPCPRM;

Members
<listen_port> Specifies the listening port of the TCP server.
<local_port> Specifies the local port assigned to the client socket.
<ip> Specifies the IP address of the server.

**CLNTPRM data structure**

This data structure defines the communication parameters of a client making a TCP connection to a server.

typedef struct _CLNTPRM
{
    char *host;
    unsigned int server_ip;
    unsigned int listen_port:16;
    unsigned int local_port:16;
    unsigned int connection_retrials:8;
    unsigned int connection_timeout:8;
    unsigned int reconnect_interval:8;
    unsigned int xxx:8;
} CLNTPRM;

Members
<host> Specifies the host name of the TCP server.
<server_ip> a converted IP address of <host>.
<listen_port> Specifies the listening port of the TCP server.
<local_port> Specifies the local port assigned to the client socket.
<connection_retrials> Specifies the maximum number of retrials before the connection is established.
<connection_timeout> Specifies the timeout value (in seconds) of wait time for the establishment of the connection in each trial.
<reconnect_interval> Specifies the amount of time (in seconds) that a new trial is initiated after the connection is disconnected.

**UDPXPRM data structure**

This data structure defines the communication parameters of a UDP connection; could be a server connection or client connection.

typedef struct _UDPXPRM
{
    char *host;
    unsigned int ip;
    unsigned int listen_port:16;
    unsigned int local_port:16;
} UDPXPRM;

Members
<host> Specifies the host name of the remote UDP server that a client is trying to connect to. Specify NULL to establish a local UDP server.
<ip> Specifies the converted IP address of the client or the local server.
<listen_port> Specifies the listening port of the remote or the local server.
<local_port> Specifies the local port assigned to the client socket.
API Functions

The seven functions defined in the connection library are described in the following sections.

connection_destroy function

This function destroys a connection.

```
void connection_destroy(MHANDLE con);
```

Inputs

- `con` The connection.

Return Value

None.

connection_send_data function

This function sends a data packet to its peer.

```
void connection_send_data (MHANDLE con, char *buf, int len);
```

Inputs

- `con` The connection.
- `buf` Pointer to the data packet.
- `len` The length of the data packet.

Return Value

None.

connection_open function

This function opens a connection.

```
MHANDLE connection_open (unsigned int type, void *param, USERFUN *funs, void **private_data);
```

Inputs

- `type` Specifies the type of the connection: CONNECTION_TYPE_TCPCLIENT, CONNECTION_TYPE_UDPSERVER, or CONNECTION_TYPE_UARTPORT.
- `param` Pointer to the data structure of the communication parameters defined for the connection.
- `funs` Pointer to the data structure of user-defined functions.
- `private_data` Pointer to the user-defined data.

Return Value

When successful, the function returns a positive pointer representing the connection. When an error occurs, the function returns as NULL.
connection_dispatch_loop function

Your user program calls this routine to enter an infinite loop in which operations of read/write for all types of connections are dispatched.

```c
int connection_dispatch_loop(int argc, char **argv, app_init_cb init);
```

**Inputs**

- `argc` Specifies the number of arguments when executing the user program.
- `argv` Pointer to an array of arguments (command line) of the user program.
- `init` Specifies a user-defined function for initializing the user program. When using this function, you need to start up servers, make connections, etc.

**Return Value**

When successful, returns 0. When an error occurs, a non-zero value is returned.

connection_dispatch_quit function

If you have a thread running, call this function to quit the loop of the main routine, `connection_dispatch_loop`.

```c
void connection_dispatch_quit(void);
```

mxsp_connection_purge function

This function purges the receiving and sending buffers of a serial connection.

```c
void mxsp_connection_purge(MHANDLE *conp);
```

**Inputs**

- `conp` Specifies the serial connection.

**Return Value**

None.

mxsp_connection_set_mask function

This function adds a timer that responds to a change of events (line status and error status) in a serial connection.

```c
int mxsp_connection_set_mask(MHANDLE *hndl, conn_event_t cb, unsigned int mask, unsigned int interval);
```

**Inputs**

- `hndl` Specifies the serial connection.
- `cb` Specifies the user-defined function that is called when events change.
- `mask` Specifies the events to be enabled.
- `interval` Specifies the timer interval.

**Return Value**

When successful, this function returns 0.
The connection library and related sample programs are provided with Moxa’s sample code. The sample programs for the connection library provide programming examples for TCP client and server, UDP client and server, and serial master and slave. This chapter will introduce major parts of sample programs using TCP, UDP, and serial functions from the connection library.

TCP Functions

This is a simple client/server example that utilizes some of API’s advanced functions. The server reverses the contents of each data packet it receives and then sends the result back to the clients.

A. Server Example

The listen port of the server must be specified in a data structure `SRVRPRM` and other members of the structure must be cleaned up (meaning that they must be initialized as zero values). User-defined functions must be specified in a data structure `USERFUN` to be properly called by the user program.

```c
/* This is an initialization function where the user program opens connections */
static int
tcp_server_init (int argc, char **argv)
{
   MHANDLE conp;
   SRVRPRM srvr;
   USERFUN funs;
   memset(&srvr, 0, sizeof(SRVRPRM));
   if (argc > 1)
      srvr.listen_port = atoi(argv[1]);
   else
      srvr.listen_port = SERVER_LISTEN_PORT;
   /* callback functions */
   memset(&funs, 0 , sizeof(USERFUN));
   funs.open = tcp_server_accept_client;
   funs.dispatch = tcp_server_dispatch_client;
   funs.close = tcp_server_close_client;
   /* make a client connection */
   if ((conp=connection_open (CONNECTION_TYPE_TCP SERVER, &srvr, &funs, NULL)) == NULL)
      return -1;
   else
      return 0;
}
```

When dispatching data packets, the member `packet_consumed` of the data structure `DATAPKT` must be specified so that the user program can correctly remove the processed data.

```c
/* this user-defined function is called after the user program receives a data packet. It reverses the contents of the packet and then sends the result back to the client */
static int
```
tcp_server_dispatch_client (MHANDLE conp, void *private_data, DATAPKT *dpkt)
{
    char buffer[1024];
dpkt->packet_consumed = dpkt->packet_size;
memcpy(buffer, dpkt->packet_data, dpkt->packet_size);
/* reverse the received data */
reverse_bytes(buffer, dpkt->packet_size);
/* send data back to the client */
connection_send_data(conp, buffer, dpkt->packet_size);
return 0;
}

B. Client Example

In addition to specifying the host name and the listen port of the server in the data structure CLNTPRM, the initialization function tcp_client_init must define user-defined functions in the data structure USERFUN. After a client connection is made successfully, this function sends the first packet to the server.

/* This is an initialization function where the user program opens connections */
static int
tcp_client_init (int argc, char **argv)
{
    MHANDLE con;
CLNTPRM param;
USERFUN funs;
if (argc < 3)
{
    printf("usage: <progname> <hostname> <listen port>\n");
    return -1;
}
/* setting the parameters of a server */
memset(&param, 0 , sizeof(CLNTPRM));
param.host = argv[1];
param.listen_port = atoi(argv[2]);
/* callback functions */
memset(&funs, 0 , sizeof(USERFUN));
funs.open = tcp_client_open;
funs.dispatch = tcp_client_dispatch;
funs.close = tcp_client_release;
/* make a client connection */
if ((con=connection_open (CONNECTION_TYPE_TCPCLIENT, &param, &funs, NULL)) == NULL)
    return -1;
else
{
    char buf[256];
    /* send 1st packet */
    strcpy(buf, "I am a client 0");
    connection_send_data(con, buf, strlen(buf));
    return 0;
}
}

Again, when dispatching data packets, the member packet_consumed of the data structure DATAPKT must be specified.

/* this user-defined function is called after the user program receives a data packet. It will keep sending packets to the server */
static int
udp_server_init (int argc, char **argv)
{
    MHANDLE con;
    UDPPRM srvr;
    USERFUN funs;
    memset(&srvr, 0, sizeof(SRVRPRM));
    if (argc > 1)
        srvr.listen_port = atoi(argv[1]);
    else
        srvr.listen_port = SERVER_LISTEN_PORT;
    /* callback functions */
    memset(&funs, 0, sizeof(USERFUN));
    funs.open = udp_server_accept_client;
    funs.dispatch = udp_server_dispatch_client;
    funs.close = udp_server_close_client;
    /* make a client connection */
    if ((con=connection_open (CONNECTION_TYPE_UDPSERVER, &srvr, &funs, NULL)) == NULL)
        return -1;
    else
        return 0;
}

When dispatching data packets, the member packet_consumed of the data structure DATAPKT must be specified.

UDP Functions

This is a simple client/server example that utilizes some of API’s advanced functions. The server reverses the contents of each data packet it receives and then sends the result back to the clients.

A. Server Example

The listen port of the server must be specified in a data structure UDPPRM and other members of the structure must be cleaned up (meaning that they must be initialized as zero values). User-defined functions must be specified in a data structure USERFUN to be properly called by the user program.

/* This is an initialization function where the user program opens connections */
static int
udp_server_init (int argc, char **argv)
{
    MHANDLE con;
    UDPPRM srvr;
    USERFUN funs;
    memset(&srvr, 0, sizeof(SRVRPRM));
    if (argc > 1)
        srvr.listen_port = atoi(argv[1]);
    else
        srvr.listen_port = SERVER_LISTEN_PORT;
    /* callback functions */
    memset(&funs, 0, sizeof(USERFUN));
    funs.open = udp_server_accept_client;
    funs.dispatch = udp_server_dispatch_client;
    funs.close = udp_server_close_client;
    /* make a client connection */
    if ((con=connection_open (CONNECTION_TYPE_UDPSERVER, &srvr, &funs, NULL)) == NULL)
        return -1;
    else
        return 0;
}

When dispatching data packets, the member packet_consumed of the data structure DATAPKT must be specified.
/* this user-defined function is called after the user program receives 
a data packet. It reverses the contents of the packet and then sends 
the result back to the client */
static int
udp_server_dispatch_client (MHANDLE conp, void *private_data, DATAPKT *dpkt)
{
    char buffer[1024];
    dpkt->packet_consumed = dpkt->packet_size;
    memcpy(buffer, dpkt->packet_data, dpkt->packet_size);
    /* reverse the received data */
    reverse_bytes(buffer, dpkt->packet_size);
    /* send data back to the client */
    connection_send_data(conp, buffer, dpkt->packet_size);
    return 0;
}

B. Client Example
In addition to specifying the host name and the listen port of the server in the data structure UDPXPRM, the 
initialization function (udp_client_init) must define user-defined functions in the data structure USERFUN. After 
a client connection is made successfully, this function sends the first packet to the server.

/* This is an initialization function where the user program opens connections */
static int
udp_client_init (int argc, char **argv)
{
    MHANDLE con;
    UDPXPRM param;
    USERFUN funs;
    if (argc < 3)
    {
        printf("usage: <progname> <hostname> <listen port>\n");
        return -1;
    }
    /* setting the parameters of a server */
    memset(&param, 0 , sizeof(UDPXPRM));
    param.host = argv[1];
    param.listen_port = atoi(argv[2]);
    /* callback functions */
    memset(&funs, 0 , sizeof(USERFUN));
    funs.open = udp_client_open;
    funs.close = udp_client_close;
    funs.dispatch = udp_client_dispatch;
    /* make a client connection */
    if ((con=connection_open (CONNECTION_TYPE_UDPCLIENT, &param, &funs, NULL)) == NULL)
        return -1;
    else /* send 1st packet */
        connection_send_data(con, "I am a client 0", 15);
    return 0;
}

Again, when dispatching data packets, the member packet_consumed of the data structure DATAPKT must be 
specified.

/* this user-defined function is called after the user program receives 
a data packet. It will keep sending packets to the server */
static int
udp_client_dispatch (MHANDLE conp, void *private_data, DATAPKT *dpkt)
Serial Functions

This is a simple serial communication example that utilizes some of API's advanced functions. The communication involves a master operating on a serial port and a slave operating on another serial port while both are connected by a loop back cable. The slave reverses the contents of each data packet it receives and then sends the result back to the master.

A. Slave Example
You must consider a few things before calling the API function `connection_open`. To begin with, the communication parameters of the serial port must be specified in a data structure `UARTPRM`. Three user-defined functions are also specified in a data structure `USERFUN`.

```c
/* This is an initialization function where the user program opens connections */
static int
serial_slave_init (int argc, char **argv)
{
    MHANDLE conp;
    UARTPRM uart;
    USERFUN funs;
    memset(&uart, 0, sizeof(UARTPRM));
    if (argc < 2)
    {
        printf("usage: <progname> <port>\n");
        return -1;
    }
    uart.port = atoi(argv[1]);
    uart.baudrate = 9600;
    uart.parity = MSP_PARITY_NONE; /* 0~4: none, odd, even, mark, space */
    uart.data_bits = 8; /* 5,6,7,8 */
    uart.stop_bits = 1; /* 1,2,3 (1.5 bits) */
    uart.iface_mode = MSP_RS232_MODE; /* 0~3: RS232, RS485_2WIRE, RS422, RS485_4WIRE */
    uart.flow_control = MSP_FLOWCTRL_HW; /* 0~2: none, software, hardware */
    memset(&funs, 0 , sizeof(USERFUN));
    funs.open = serial_slave_open;
    funs.dispatch = serial_slave_dispatch;
    funs.close = serial_slave_close;
    /* make a client connection */
    if ((conp=connection_open (CONNECTION_TYPE_UARTPORT, &uart, &funs, NULL)) == NULL)
        return -1;
    else
```
When dispatching data packets as below, the member `packet_consumed` of the data structure `DATAPKT` must be specified so that the user program can remove the processed data.

```c
/* this user-defined function is called after the user program receives a data packet. It reverses the contents of the packet and then sends the result back to the client */
static int
serial_slave_dispatch (MHANDLE comp, void *private_data, DATAPKT *dpkt)
{
  char buffer[1024];
  dpkt->packet_consumed = dpkt->packet_size;
  memcpy(buffer, dpkt->packet_data, dpkt->packet_size);
  /* reverse the received data */
  reverse_bytes(buffer, dpkt->packet_size);
  /* send data back to the master */
  connection_send_data(comp, buffer, dpkt->packet_size);
  return 0;
}
```

B. Master Example

The initialization function (`serial_master_init`) is almost the same as the one defined for the slave. The only difference is that the master sends the first packet to the slave after the connection is made successfully.

```c
/* This is an initialization function where the user program opens connections */
static int
serial_master_init (int argc, char **argv)
{
  MHANDLE con;
  UARTPRM uart;
  USERFUN funs;
  memset(&uart, 0, sizeof(UARTPRM));
  if (argc < 2)
  {
    printf("usage: <progname> <port>\n");
    return -1;
  }
  uart.port = atoi(argv[1]);
  uart.baudrate = 9600;
  uart.parity = MSP_PARITY_NONE; /* 0~4: none, odd, even, mark, space */
  uart.data_bits = 8; /* 5,6,7,8 */
  uart.stop_bits = 1; /* 1,2,3 (1.5 bits) */
  uart.iface_mode = MSP_RS232_MODE; /* 0~3: RS232, RS485_2WIRE, RS422, RS485_4WIRE */
  uart.flow_control = MSP_FLOWCTRL_HW; /* 0~2: none, software, hardware */
  /* callback functions */
  memset(&funs, 0 , sizeof(USERFUN));
  funs.open = serial_master_open;
  funs.dispatch = serial_master_dispatch;
  funs.close = serial_master_close;
  /* make a client connection */
  if ((con=connection_open (CONNECTION_TYPE_UARTPORT, &uart, &funs, NULL)) == NULL)
    return -1;
  else
  {
    char buf[256];
  
```
/* send 1st packet */
strcpy(buf, "I am a master 0");
connection_send_data(con, buf, strlen(buf));
return 0;
}

Again, when dispatching data packets as below, the member packet_consumed of the data structure DATAPKT must be specified.

/* this user-defined function is called after the user program receives a data packet. It will keep sending packets to the server */
static int
serial_master_dispatch (MHANDLE comp, void *private_data, DATAPKT *dpkt)
{
    static int count = 1;
    char ch, *buf = (char*) private_data;
    dpkt->packet_consumed = dpkt->packet_size; /* consumed all */
    ch = dpkt->packet_data[dpkt->packet_size];
    dpkt->packet_data[dpkt->packet_size] = 0;
    printf("RECV [%d]: %s\n", dpkt->packet_size, dpkt->packet_data);
    dpkt->packet_data[dpkt->packet_size] = ch;
    #ifdef WIN32
    Sleep(1000);
    #else
    sleep(1);
    #endif
    sprintf(buf, "I am a master %d", count++);
    connection_send_data(comp, buf, strlen(buf));
    return 0;
}
The MODBUS serial communication protocol, which was published in 1979, has become the de facto communication standard for industry. MODBUS has two variants. MODBUS RTU is a compact, binary encoding of the data and MODBUS ASCII is a hexadecimal encoding of the data, represented with readable ASCII characters.

MODBUS TCP is a variant of the MODBUS family and covers the use of MODBUS messaging in an Internet or intranet environment using the TCP/IP protocols.

The MODBUS protocol defines a simple Protocol Data Unit (PDU) independent of the underlying communication layers. The mapping of the MODBUS protocol to serial lines or a network introduces some additional fields to encapsulate PDU to form the Application Data Unit (ADU). For MODBUS TCP, a dedicated MBAP (MODBUS Application Protocol) header is used to identify the MODBUS ADU. The MBAP header, but not including the slave ID field, is defined as an MBTCPHDR structure in the MODBUS programming functions.

For the MODBUS serial line protocol the PDU follows an error check field. The RTU format follows data with a cyclic redundancy check (CRC) checksum, whereas the ASCII format uses a longitudinal redundancy check (LRC) checksum.

In Moxa’s sample code, the MODBUS library and its sample programs are also provided. The MODBUS library provides programming functions for MODBUS TCP, MODBUS RTU, and MODBUS ASCII. The sample programs provide programming examples for MODBUS TCP client and server, MODBUS RTU master and slave, and MODBUS ASCII master and slave. In the following sections all MODBUS functions are introduced and short codes will be presented for message conversion between MODBUS TCP and MODBUS RTU as well as message conversion between MODBUS RTU and MODBUS ASCII.

### MODBUS Functions

The MODBUS library provides two functions for each of MODBUS TCP, MODBUS RTU, and MODBUS ASCII to verify and compose its corresponding messages. In this section these six functions are introduced.

This function parses a MODBUS TCP ADU data packet into a MODBUS TCP structured header and a PDU buffer.

```c
unsigned char* mbtcp_packet_digest(unsigned char *packet, unsigned int *plen, MBTCPHDR *tcp);
```

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>packet</td>
<td>ADU data packet.</td>
</tr>
<tr>
<td>plen</td>
<td>The length of an ADU data packet.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Outputs</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tcp</td>
<td>PDU header, excluding slave ID.</td>
</tr>
<tr>
<td>plen</td>
<td>The # of bytes that have been consumed from the ADU data packet.</td>
</tr>
</tbody>
</table>

**Return Value**

Pointer to the data part of the MODBUS TCP PDU inside the ADU data packet.

This function composes PDU data and a MODBUS TCP header into a MODBUS TCP ADU data packet.
**unsigned int mbtcp_packet_format(MBTCPHDR *tcp, unsigned char *data, unsigned int len, unsigned char *packet);**

**Inputs**
- *tcp* PDU header, excluding slave ID.
- *data* PDU data, including slave ID.
- *len* The length of the PDU data.

**Outputs**
- *packet* ADU data packet.

**Return Value**
The # of bytes of the ADU data packet.

This function digests a MODBUS RTU ADU data packet into a MODBUS RTU PDU buffer.

**unsigned int mbrtu_packet_digest(unsigned char *packet, unsigned int plen);**

**Inputs**
- *packet* ADU data packet.
- *plen* The length of ADU data packet.

**Return Value**
The # of bytes in a packet excluding CRC, 0 meaning incomplete/wrong CRC packet.

This function formats PDU data into a MODBUS RTU ADU packet.

**unsigned int mbrtu_packet_format(unsigned char *data, unsigned int dlen, unsigned char *packet);**

**Inputs**
- *data* PDU data, including slave ID.
- *dlen* The length of the PDU data.

**Outputs**
- *packet* ADU packet.

**Return Value**
The # of bytes of the ADU packet.

This function digests a MODBUS ASCII ADU data packet into a MODBUS ASCII PDU buffer.

**unsigned int mbasc_packet_digest(unsigned char *frame, unsigned int *plen, unsigned char *data);**

**Inputs**
- *frame* ADU data packet.
- *plen* The length of ADU data packet.

**Outputs**
- *data* PDU data, including slave ID.
- *plen* The length of data that is consumed.

**Return Value**
The # of bytes in a packet excluding LRC, 0 meaning incomplete/wrong LRC packet.

This function formats PDU data into a MODBUS ASCII ADU packet.

**unsigned int mbasc_packet_format(unsigned char *data, unsigned int dlen, unsigned char *frame);**

**Inputs**
- *data* PDU data, including slave ID.
- *dlen* The length of PDU data.

**Outputs**
- *frame* ADU frame.

**Return Value**
The length of an ADU frame.
MODBUS TCP to RTU

In addition to transferring messages of the same MODBUS message type between two devices, transferring messages of different MODBUS message types is also required for MODBUS gateways. MODBUS gateways often need to convert between MODBUS TCP messages and MODBUS RTU messages. The following is an example of using the MODBUS library to convert from MODBUS TCP messages to MODBUS RTU messages. Suppose the connection library is used to do the conversion and in the user-defined dispatch function of a TCP connection for MODBUS TCP server, the function `mbtcp_packet_digest` is used to get a MODBUS TCP message from input data as normal MODBUS TCP message handling. Then, the PDU of the MODBUS TCP message is used to compose a MODBUS RTU message using the function `mbrtu_packet_format`. Finally, based on some rules or message content the MODBUS RTU message is sent to the serial port of a determined MODBUS RTU slave using `connection_send_data` function.

```c
MHANDLE uart_hndl; /* connection handle of serial port for MODBUS RTU */
unsigned char *packet; /* PDU, including Unit Identifier */
unsigned int plen = dpkt->packet_size;
MBTCPHDR mb;
unsigned char mbrtu_buffer[261]; /* buffer for a MODBUS RTU message */
int len;
packet = mbtcp_packet_digest((unsigned char *)dpkt->packet_data, &plen, &mb);
if (packet == NULL) /* MODBUS TCP ADU packet not complete */
{
    dpkt->packet_consumed = 0;
    return 0;
}
len = mbrtu_packet_format(packet, mb.length, mbrtu_buffer);
if (len > 0)
{
    connection_send_data(uart_hndl, mbrtu_buffer, len);
}
```

MODBUS RTU to ASCII

MODBUS gateways sometimes need to convert between MODBUS RTU messages and MODBUS ASCII messages. The following is an example of using the MODBUS library to do the conversion from MODBUS RTU messages to MODBUS ASCII messages. Suppose the connection library is used to do the conversion and in the user-defined dispatch function of a serial connection for MODBUS RTU slave, the function `mbrtu_packet_digest` is used to verify a MODBUS RTU message from input data as normal MODBUS RTU message handling. Then, the PDU of the MODBUS RTU message is used to compose a MODBUS ASCII message using the function `mbasc_packet_format`. Finally, based on some rules or message content the MODBUS ASCII message is sent to the serial port of a determined MODBUS ASCII slave using the function `connection_send_data`.

```c
MHANDLE uart_hndl; /* connection handle of serial port for MODBUS ASCII */
unsigned char mbasc_buffer[513]; /* buffer for a MODBUS ASCII message */
int len;

/* determine the length of an incoming MODBUS RTU request message */
len = mbrtu_get_request_len(dpkt);
...
len = mbrtu_packet_digest((unsigned char *)dpkt->packet_data, len);
if (len <= 0)
{
    dpkt->packet_consumed = 0;
}
```
return 0;
}
len = mbasc_packet_format(dpkt->packet_data, len, mbasc_buffer);
if (len > 0)
{
    connection_send_data(uart_hndl, mbasc_buffer, len);
}