# Table of Contents

## Chapter 1  Introduction and Installation

- Field Applications ................................................................. 1-1
- MPC Software Architecture ................................................... 1-2
- Installing MPC on a Linux Computer ....................................... 1-3
- Installing MPC on a WinCE Computer ..................................... 1-3

## Chapter 2  Configuration

- Configuration File—config.mpc ................................................ 2-2
- Example: Configuring a MODBUS TCP/RTU Gateway ................. 2-2
- Example: Configuring a MODBUS TCP/ASCII Gateway ............... 2-3
- Example: Configuring a Meter-Reading Channel on a MODBUS TCP/RTU Gateway ........ 2-4
- Using Moxa Device Manager to Update the Configuration ............ 2-5
- Starting a New Configuration ................................................... 2-6
- Establishing a Communication Channel ..................................... 2-7
- Configuring a Driver ................................................................. 2-8
- Configuring a TCP Server ....................................................... 2-9
- Configuring a TCP Client ......................................................... 2-10
- Configuring a Serial Port ......................................................... 2-11
- Deleting a Channel or Driver .................................................... 2-12
- Saving the Current Configuration ............................................. 2-12
- Opening the Current Configuration .......................................... 2-13
- Loading a Sample Configuration .............................................. 2-13
- Example: Configuring a MODBUS TCP/RTU Gateway ................. 2-13
- Example: Configuring a MODBUS TCP/ASCII Gateway ............... 2-14
- Example: Configuring a Meter-Reading Channel on a MODBUS TCP/RTU Gateway ........ 2-14

## Chapter 3  MPC Driver Programming

- Structure DRVRPKT ................................................................. 3-1
- Driver Functions ........................................................................ 3-2
- Building a Linux MPC Driver ................................................... 3-5
- Building a WinCE MPC Driver .................................................. 3-5
Chapter 1

Introduction and Installation

Moxa Protocol Converter (referred to here as “MPC”) is a software program that runs on Moxa’s ready-to-run embedded computers. MPC is an engine that loads custom programs called drivers. The drivers audit bi-directional data streams between a network client and a serial device, or between two serial devices.

The MPC engine implements a streamline architecture consisting of channels and drivers. A channel is a logical path connecting two physical communication ports (serial or network). A sequence of user-programmed drivers works in the middle to convert data from one protocol to another or handle data processing tasks. The engine receives a data stream at one communication port, assembles each data packet in the stream, and then passes the packets to the drivers for processing. Each driver performs data formatting or conversion on the packet, as required. The resulting packet is then passed to the next driver for further conversion. This process continues until the final packet reaches the physical port at the other end of the channel.

One of the key benefits provided by MPC is that both serial and networking communication are built into the engine. Without any additional programming on the part of the user, the engine alone can perform transparent data transmission between any two communication ports (i.e., the data channel mentioned above).

Another key benefit of MPC is that port-to-port communication is driver programmable. The engine supports multiple drivers in a channel, which simplifies and modularizes the design of an application that needs layers of data processing. In addition, most systems used in the field are created by making minor changes to existing systems. With the MPC engine, these changes can be achieved by either modifying a driver, or adding a driver to a channel.

Field Applications

Many industrial applications rely on a central system that monitors and controls a large number of devices distributed around the site. In this case, control commands are handled by Remote Terminal Units (RTUs) or Programmable Logic Controllers (PLCs), although many of the control commands are issued by a centralized host housed in an operations center. Communication between the different types of controllers can be carried out through a serial cable or over a LAN (local area network).

Based on current practice, it is quite likely that embedded front-end computers will be installed at field sites to provide a kind of store-and-forward mechanism. The embedded computers get raw data, such as meter readings and device status reports, from the RTUs or PLCs, after which the data is compiled and/or formatted as needed. The resulting data is transmitted to the host in the operations room, and the operator generates control commands that are sent to the RTUs and PLCs.
The following figure indicates the general relationship between the host level, gateway level, and device level.

Moxa’s embedded computers act at the gateway level to acquire information from serial devices and then report to the operators in the control room. The gateway can be used to send control signals to devices through the serial links, and provide access by multiple clients over the local network. The gateway can also be used to convert bi-directional data streams between serial devices.

The gateways are used to allow serial and networking devices to communicate with each other. Since a poorly designed system could require engineers to implement the same task over and over again, there is a lot of motivation to find ways to streamline operations by reducing, or eliminating altogether, the number of duplicate tasks that must be performed. MPC comes to the rescue in this regard by consolidating common device communication tasks into a software engine, leaving the task of developing drivers for various types of data conversion to programmers who are experts in using the data formats and transmission protocols supported by the devices.

**MPC Software Architecture**

The MPC engine manages several communication channels, each of which has two communication ports that convert data streams. The ports can be either serial or network ports, and data received at one port will eventually be forwarded to the other port.

The following figure shows an MPC channel linking two communication ports: port A (LAN) and port B (serial). The channel processes data packets in a pair of data streams: an A2B-stream and a B2A-stream. As shown, the receiving port of a data stream is also the sending port of another stream. Drivers (represented by dotted boxes in the figure) are arranged sequentially in the channel, and work to format the data packets.
Let’s look at the A2B-stream part of the streamline to illustrate how the process works. After detecting the arrival of a data packet at port A, the channel passes it to the first driver for data processing. After formatting the packet, the driver forwards the resulting data to the second driver. The process proceeds in this way until the packet arrives at port B.

Each driver is comprised of at least two user-defined logic functions, with each of the functions commanding data conversion for one side of the data stream. For example, in a MODBUS/TCP driver, one function extracts and formats a MODBUS/TCP packet into a raw packet for the A2B-stream, and another function extracts and formats a raw packet into a MODBUS/TCP packet for the B2A-stream.

The MPC engine also allows two communication ports of the channel to be of the same type. One example is a networking channel in which the receiving port is the same as the sending port. Incoming data packets are sent back to the sender either with no formatting, or after being formatted. Another example is a serial channel that has two different communication ports. Data from a serial port will be relayed to another serial port, again either with no formatting, or after being formatted.

**Installing MPC on a Linux Computer**

MPC is a user-space program called mpc. To install the program, first create a working directory in which the program will be stored. For example, you could create a working directory called /home/mpc. Next, use an ftp client to copy the program to this directory. Before executing the program, be sure to change the file mode so that the program will be executable in your system. Use the following command to modify the file mode.

```
> chmod +x mpc
```

**Installing MPC on a WinCE Computer**

MPC is a user-space program called mpc.exe. To install the program, first create a working directory. For example, you could create a working directory called /NORFlash/mpc. Next, use an ftp client to copy the program to this directory.
After you install the software package on your development workstation, you will find several new directories, as shown below, under the MPC directory.

```
- drvlh
  - mbascii
  - mbbtc
  - mbtcp
  - meter
  - rfc2217
- drvlib_projects
  - mbascii
  - mbbtc
  - mbtcp
  - meter
  - rfc2217
- release
  - config_samples
  - drivers
```

Several driver examples are provided as a reference for programmers. The source code of the drivers can be found in the “drvlh” directory. Also included in this directory is a “Makefile” file, which can be used to compile all of the drivers in a Linux development workstation. To compile WinCE drivers, use the eVC++ project files that are stored in the “drvlib_projects” directory.

The “release” directory contains the MPC programs `mpc.exe` and `mpcdbg.exe`. Sample configuration files are stored in the “config_samples” sub-directory. In addition, executable drivers are stored in the “drivers” sub-directory.

MPC is a user-space program. To install the program on the embedded computer, first create a working directory. Next, use an ftp client to copy all the files and directories under the “release” directory to this working directory.

The general format of the command used to invoke the MPC engine is:

```
mpc [-d config_path]
```

Use the “-d” option to specify the working directory, in which “config_path” is a full path. If you run the program without specifying a configuration path, the MPC engine will assume that the configuration file `config.mpc` is in the default directory, “/home/mpc” for Linux computers and “/NORFlash/mpc” for WinCE computers.
Configuration File—config.mpc

The configuration file is written in XML format, and consists of entries in text lines. The text is NOT case sensitive. The following simple configuration example illustrates some of the features of the protocol converter.

```xml
<MPC_DOC_ROOT>
  <XPORTS>
    <UART>
      <MPC_PORT port="1" baud_rate="115200" data_bits="8" parity="None" stop_bits="1" interface="RS232" flow_control="None"/>
    </UART>
    <MPC_PORT port="8" baud_rate="115200" data_bits="8" parity="None" stop_bits="1" interface="RS232" flow_control="None"/>
  </UART>
  <TCP_SERVER>
    <MPC_PORT port="5001" max_connections="8" accepted_ips="" rt_timeout="200"/>
    <MPC_PORT port="6001" max_connections="10" accepted_ips="" broadcast="yes"/>
  </TCP_SERVER>
  </XPORTS>
  <XCHANNELS>
    <MPC_CHANNEL name="CH1">
      <PORT_A type="TCP_SERVER" port="5001"/>
      <PORT_B type="UART" port="1"/>
      <DRIVER name="D1" dll="/home/mpc/mbtcp.dll" prefix="mbtcp"/>
    </MPC_CHANNEL>
    <MPC_CHANNEL name="CH8">
      <PORT_A type="TCP_SERVER" port="6001"/>
      <PORT_B type="UART" port="8"/>
    </MPC_CHANNEL>
  </XCHANNELS>
</MPC_DOC_ROOT>
```

The entire file is enclosed by a document root, `<MPC_DOC_ROOT></MPC_DOC_ROOT>`. Two major sections are enclosed by element pairs `<XPORTS></XPORTS>` and `<XCHANNELS></XCHANNELS>`, respectively. The first pair defines communication ports of different types. The second pair defines channels that are associated with the communication ports.

Example: Configuring a MODBUS TCP/RTU Gateway

Modbus is a standard serial communication protocol used in field devices, such as programmable logic controllers (PLCs) and remote terminal units (RTUs). Modbus is most commonly used to connect a host computer to industrial electronic devices in SCADA (Supervisory Control And Data Acquisition) systems.

We provide the source code required to enable an MPC channel as a Modbus TCP/RTU/ASCII gateway. Source code is available for shared objects in Linux, or dynamic-link libraries in WinCE. You can search the software package we provide to locate the source code you need. If the source code is not appropriate for your application, you can always modify the code and then recompile it.

Here is an example:

```xml
<MPC_DOC_ROOT>
  <XPORTS>
    <UART>
      <MPC_PORT port="1" baud_rate="115200" interface="RS485" timeout="50"/>
    </UART>
  </UART>
  <TCP_SERVER>
    <MPC_PORT port="502" rt_timeout="200"/>
  </TCP_SERVER>
</MPC_DOC_ROOT>
```
The above example sets up a MODBUS TCP/RTU/RS485 channel between TCP server port 502 and serial port #1. The driver `mbtcp.dll` converts each MODBUS/TCP packet into raw data, and then the driver `mbrtu.dll` assembles the data into a MODBUS/RTU packet. In the reverse direction, the driver `mbrtu.dll` converts each MODBUS/RTU packet into raw data and then the driver `mbtcp.dll` assembles the data into a MODBUS/TCP packet.

**Example: Configuring a MODBUS TCP/ASCII Gateway**

```xml
<MPD_DOC_ROOT>
  <XPORTS>
    <UART>
      <MPC_PORT port="2" baud_rate="115200" interface="RS485" timeout="50" /> 
    </UART>
    <TCP_SERVER>
      <MPC_PORT port="5001" rt_timeout="200" /> 
    </TCP_SERVER>
  </XPORTS>
</MPD_DOC_ROOT>
```
The above example sets up a MODBUS TCP/ASCII/RS485 channel between TCP server port 5001 and serial port #2. The driver `mbtcp.dll` converts each MODBUS/TCP packet into raw data and then the driver `mbascii.dll` assembles the data into a MODBUS/ASCII packet. In the reverse direction, the driver `mbascii.dll` converts each MODBUS/ASCII packet into raw data and then the driver `mbtcp.dll` assembles the data into a MODBUS/TCP packet.

Example: Configuring a Meter-Reading Channel on a MODBUS TCP/RTU Gateway

Most applications need timers to trigger periodic requests to field devices to get readings and report to the host center. For example, a meter application needs to query device readings every 5 seconds. One can implement a timer function, `meter_exec_timer`, inside the driver `meter.dll` and set attribute `timer_interval` to 5000 milliseconds.

```xml
<MPC_DOC_ROOT>
  <XPORTS>
    <UART>
      <MPC_PORT port="1" baud_rate="115200" interface="RS485" timeout="50"/>
    </UART>
    <TCP_SERVER>
      <MPC_PORT port="502" rt_timeout="200"/>
    </TCP_SERVER>
  </XPORTS>
  <XCHANNELS>
    <MPC_CHANNEL name="Modbus_TCP_RTU">
      <PORT_A type="TCP_SERVER" port="502"/>
      <PORT_B type="UART" port="1"/>
      <DRIVER name="MBTCP" dll="/home/mpc/mbtcp.dll" prefix="mbtcp"/>
      <DRIVER name="METER" dll="/home/mpc/meter.dll" prefix="meter" timer_interval="5000"/>
    </MPC_CHANNEL>
  </XCHANNELS>
</MPC_DOC_ROOT>
```
Using Moxa Device Manager to Update the Configuration

Instead of editing the XML configuration file manually, users can use a windows-based graphic interface to configure communication channels in MPC. The following section shows how to configure MPC using Moxa Device Manager.

Moxa Device Manager (short for MDM) is an easy-to-use remote management tool for managing Moxa’s ready-to-run embedded computers on a local area network (LAN). For detailed information about this tool, please refer to the MDM user’s guide. Follow the steps below to activate MDM:

1. Log on to the embedded computer and start the MDMAgent program.
   #MDMAgent &
2. Start the MDM program on your own computer.
3. Start the MPC program on the embedded computer.

To begin configuring channels and drivers, locate “Group of MPC” in the left panel of the Device Manager window.

IP addresses of embedded computers currently running MPC will be listed under “Group of MPC.” Double click on the IP address of the device you would like to configure to invoke “Moxa Protocol Converter.” Channels currently running in the embedded computer will be listed on the main page.

The main page consists of a toolbar and a channel display. On the toolbar, there are several buttons that allow users to create, save, and load configuration files. On the channel display, the serial ports (the DB9 icon) are on the right side and Ethernet ports (the RJ45 icon) are on the left side.
The number of serial port displayed is identical with that on the embedded computer. The Ethernet ports are given by default. The number of Ethernets port is the same as the number of serial ports on the embedded computer.

Starting a New Configuration

You can add, delete, or edit a channel in the current configuration or create a new configuration from scratch. To start a new configuration, click on the “New Configuration” button on the toolbar. A dialog window pops up to confirm that a new configuration will be created. All current channel settings will be erased if you press “Yes.”

You will see a clear channel display page, with each icon showing a serial port or Ethernet port with a “disabled” icon nearby.
Establishing a Communication Channel

A communication channel can be established between two ports. The data stream in a communication channel is bidirectional. The channel can start from either a serial port or an Ethernet port. The following steps show an example of building a serial to Ethernet (TCP server) communication channel.

**Step 1:** Click on a port icon to set one end of a channel. You may start from either a serial port or an Ethernet port. If you click on a serial port, you will see an active arrow pointing to the other end of the channel. Right click on the active node and select “Property” to configure the serial port settings.

**Step 2:** After configuring the serial port configuration, click “OK.” The Ethernet port can be set up in a similar manner. Once you confirm the Ethernet port settings, the communication channel configuration will be complete.
MPC Configuration and Programming Guide

Configuration

Step 3: Type a name for the channel in the “Configuring a Channel” window and then click “OK.”

### Configuring a Driver

To add a driver to a channel, right click on the communication channel and click on “Add driver.” A configuration window pops up. Select the driver located in the mpc driver directory and input the driver prefix string. The prefix is the special string defined in the driver function. If the driver is a timer, you can provide the timer interval based on the direction the timer triggers data flow. Ignore the timer interval setting if the driver is not a timer.

#### Table of Settings and Descriptions

<table>
<thead>
<tr>
<th>Setting</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Path</strong></td>
<td>Specify the path of the driver object or the dynamic library.</td>
</tr>
<tr>
<td><strong>Prefix</strong></td>
<td>The <em>prefix</em> is a short string used as a prefix in the name of the driver function that the MPC engine calls. For example, the prefix mbtcp could be used in the names of functions implemented by the driver. Examples of possible function names are mbtcp_open_A2B, mbtcp_close_A2B, mbtcp_exec_A2B, and mbtcp_exec_B2A.</td>
</tr>
<tr>
<td><strong>Timer interval(A2B)</strong></td>
<td>The <em>timer_interval_A2B</em> attribute, which is measured in milliseconds, should be defined if you implement a timer function (e.g., meter_timer_A2B) that the engine calls periodically. This timer is particularly useful when an application needs to issue a request to a serial port periodically. However, please note that every timer uses CPU resources, so do not use a timer unless you really need it.</td>
</tr>
<tr>
<td><strong>Timer interval(B2A)</strong></td>
<td>The <em>timer_interval_B2A</em> attribute is similar to the <em>timer_interval_A2B</em> attribute. It should be defined if you implement a timer function (e.g., meter_timer_B2A) that the engine calls periodically.</td>
</tr>
</tbody>
</table>
Configuring a TCP Server

A TCP server listens to connection requests from remote TCP clients. This section describes the settings for a TCP server. Right click on the Ethernet icon and select “Property.”

### Configuring a TCP/IP Port

<table>
<thead>
<tr>
<th>Setting</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP/IP port number</td>
<td>N/A</td>
<td>Specifies the listening port of the server. This attribute must be defined.</td>
</tr>
<tr>
<td>RT timeout</td>
<td>2000</td>
<td>The MPC engine does not split threads, but it tends to serve concurrent clients fairly. For this reason, the engine will not block transmissions while waiting for a response after sending a request to a slow serial device that is a shared resource. If the engine receives requests to the same serial device from more than one client, the requests are placed in a queue. While serving one request, the other requests in the queue are blocked temporarily. In some cases, a serial device may not respond to a request. In this case, blocked requests waiting for service will not be served. To avoid this kind of deadlock, the engine triggers a timer (measured in milliseconds) specified by <code>rt_timeout</code> for each request. If the timer expires before the engine gets a response, the blocked requests will be unblocked. The choice of a proper value for this timer depends on the number of concurrent clients, the loading of the computer, the baud rate of</td>
</tr>
</tbody>
</table>
MPC Configuration and Programming Guide

Configuration

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max connections</td>
<td>4</td>
<td>Limits the number of concurrent clients that the server will accept. The default value (4) will be used if this attribute is left open.</td>
</tr>
<tr>
<td>Broadcast</td>
<td>No</td>
<td>Setting this attribute to yes will allow clients connected to the same TCP server to share a response, regardless of which client issued the request.</td>
</tr>
<tr>
<td>Accepted IPs</td>
<td>NA</td>
<td>This option is used to grant access right to privileged clients identified by their IP addresses. If this attribute is not defined, then any client will be granted access. Setting the attribute to 172.16.2.0, for example, will cause the MPC engine to accept clients with IP addresses ranging from 172.16.2.0 to 172.16.2.255. For multiple settings of this type, the IP addresses should be separated by spaces.</td>
</tr>
</tbody>
</table>

**Configuring a TCP Client**

Right click on the Ethernet icon and select “Property.” A window for configuring the TCP/IP port will pop up. Choose TCP client.

![Configuring a TCP/IP Port](image)
### Attribute Default Description

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>IP address</strong></td>
<td>N/A</td>
<td>The host IP address.</td>
</tr>
<tr>
<td><strong>Port number</strong></td>
<td>N/A</td>
<td>The port number that host listens.</td>
</tr>
<tr>
<td><strong>RT timeout</strong></td>
<td>2000</td>
<td>Works exactly the same as the one defined for a TCP server.</td>
</tr>
<tr>
<td><strong>Reconnect period</strong></td>
<td>10</td>
<td>The MPC engine will initiate periodical attempts to reconnect the TCP/IP client connection that gets closed. If the connection fails, the connection attempt will be repeated for a user-defined number of times and interval. Setting <code>reconnect_period</code> specifies the frequency of these attempts (in seconds).</td>
</tr>
<tr>
<td><strong>Reconnect times</strong></td>
<td></td>
<td>Specify the maximal number of connection attempts.</td>
</tr>
</tbody>
</table>

#### Configuring a Serial Port

Right click on the Ethernet icon and select “Property.” A window for configuring the serial port will pop up.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Interface</strong></td>
<td>RS232</td>
<td>The interface attribute can be set to RS322, RS422, RS485-2WIRE, or RS485-4WIRE.</td>
</tr>
<tr>
<td><strong>Baud rate</strong></td>
<td>9600</td>
<td>The baud_rate attribute can be set to any value from 50 to 921600.</td>
</tr>
<tr>
<td><strong>Data bits</strong></td>
<td>8</td>
<td>The data_bits attribute must be set to 5, 6, 7, or 8.</td>
</tr>
<tr>
<td><strong>Parity</strong></td>
<td>None</td>
<td>The parity attribute should be set to either None, Odd, Even, Mark, or Space.</td>
</tr>
<tr>
<td><strong>Stop bits</strong></td>
<td>1</td>
<td>The stop_bit attribute must be set to 1 or 2.</td>
</tr>
<tr>
<td><strong>Flow control</strong></td>
<td>Hardware</td>
<td>To avoid overflow on the serial port during data communication, enable flow control through the port by setting the <code>flow_control</code> attribute to Hardware or Software. We</td>
</tr>
</tbody>
</table>
MPC Configuration and Programming Guide

Configuration

| Timeout | 100 | Data received from a serial port may be part of a data frame that will be processed by an application. To insure that the application processes the correct data, you can assign a value (measured in milliseconds) to the timeout attribute. After receiving the first piece of data in a frame from the serial port, a timer keeps track of the time (in milliseconds). Data received from the serial port before the timeout value is reached will be appended to the data frame. Once the timer reaches the timeout value, the data frame will be sent to the appropriate channel. |

Deleting a Channel or Driver

To delete a channel, right click on the edge of the channel and select “Delete Channel.” You will see the selected channel in blue. Click “Yes” to confirm the deletion. If you delete a driver, right click on the driver you want to delete. The selected driver shows the “D” icon and select “Delete Driver.” After you confirm, the driver will be deleted.

Saving the Current Configuration

The current configuration is the configuration in the config.mpc file on the embedded computer. After you configure all communication channels, be sure to save them in the current configuration. Otherwise, MPC won’t recognize the newly created channels. Press the “Save Current Configuration” button on the toolbar to write the new communication channels to the current configuration file.
Opening the Current Configuration

If you want to discard all of your changes and reload the original configuration, press the “Open current configuration” button on the toolbar. After you confirm that you want to discard all changes, the original configuration will be shown on display panel.

Loading a Sample Configuration

MPC comes with sample configurations that can be used as template for channel configurations. You can modify a sample configuration to fit your own communication architecture. To load a sample configuration, select a configuration file from the drop-down menu on the toolbar. After you confirm the selection, all channels on the display panel will be erased. You can modify the sample configuration, and then save it as a current configuration if you want the configuration to take effect.

Example: Configuring a MODBUS TCP/RTU Gateway

Modbus is a standard serial communication protocol used in field devices, such as programmable logic controllers (PLCs) and remote terminal units (RTUs). Modbus is most commonly used to connect a host computer to industrial electronic devices in SCADA (Supervisory Control And Data Acquisition) systems.

We provide the source code required to enable an MPC channel as a Modbus TCP/RTU/ASCII gateway. Source code is available for shared objects in Linux, or dynamic-link libraries in WinCE. You can search the software package we provide to locate the source code you need. If the source code is not appropriate for your application, you can always modify the code and then recompile it.
The above example sets up a MODBUS TCP/RTU/RS485 channel between TCP server port 502 and serial port #1. The driver mbtcp.dll converts each MODBUS/TCP packet into raw data, and then the driver mbrtu.dll assembles the data into a MODBUS/RTU packet. In the reverse direction, the driver mbrtu.dll converts each MODBUS/RTU packet into raw data and then the driver mbtcp.dll assembles the data into a MODBUS/TCP packet.

**Example: Configuring a MODBUS TCP/ASCII Gateway**

The above example sets up a MODBUS TCP/ASCII/RS485 channel between TCP server port 5001 and serial port #2. The driver mbtcp.dll converts each MODBUS/TCP packet into raw data and then the driver mbascii.dll assembles the data into a MODBUS/ASCII packet. In the reverse direction, the driver mbascii.dll converts each MODBUS/ASCII packet into raw data and then the driver mbtcp.dll assembles the data into a MODBUS/TCP packet.

**Example: Configuring a Meter-Reading Channel on a MODBUS TCP/RTU Gateway**

Most applications need timers to trigger periodic requests to field devices to get readings and report to the host center. For example, a meter application needs to query device readings every 5 seconds. One can implement a timer function, meter_timer_A2B, inside the driver meter.dll and set attribute timer_interval_A2B to 5000 milliseconds.

This driver meter.dll is inserted between driver mbtcp.dll and driver mbrtu.dll.
Chapter 3
MPC Driver Programming

We assume that you have installed either a Linux tool chain or Microsoft eVC++ 4.0 on your development workstation. Linux drivers are in the format of executable shared objects, and WinCE drivers are in the format of dynamic-link libraries. Note that the MPC engine does not support computers that use the µcLinux operating system. This is because µcLinux does not support shared objects.

The MPC engine is written in C, and for this reason, the drivers you create must also be written in C. After compiling the drivers, the resulting driver files will be in the form of shared objects or dynamic link libraries.

Note that your drivers use the same memory context as the MPC engine. For this reason, it is your responsibility to use a “clean design” for your drivers. If your driver design is flawed, then code violations of your drivers could cause a core dump of the MPC engine.

There is no need to use binary APIs when programming your drivers, but developers need to follow the correct function definitions for a given x_driver.h file. Developers can make a copy of this file, and then save it in a location accessible by your program.

After you install the software package on your development workstation, locate the “drvlib” directory. In addition to containing the file x_driver.h, there are sub-directories used to store sample code for drivers.

Structure DRVRPKT

The structure DRVRPKT is an I/O interface between the MPC engine, the caller, and a driver function. Before calling this driver function, the engine prepares a data structure that describes the content and the size of a data packet. The driver function processes the data packet, and provides the content/size of the resulting data in the same data structure before ending the call.

```c
typedef struct _DRVRPKT
{
    unsigned char *packet_data;
    unsigned int packet_size;
    unsigned int packet_consumed;
    void *private_data;
    void *connection;
    void *portA, *portB;
} DRVRPKT;
```
Members

packet_data
Specifies the memory area for storing the data packet that is received by the engine and passed to a driver function. Before handing process control back to the engine, the driver function must assign another malloc or static area that stores the resulting data packet.

packet_size
Specifies the size of the data packet that packet_data initially points to. Before ending its call, a driver function must specify it to be the size of the resulting data. Setting the value to zero means it will stop forwarding the data packet to another driver.

packet_consumed
A driver function must specify the number of bytes that are consumed in the original packet. With this information, the engine is informed to remove the data to be processed. This value is initially equal to the number of bytes of the original data packet.

private_data
A driver function must specify the number of bytes that are consumed in the original packet. With this information, the engine is informed to remove the data to be processed. This value is initially equal to the number of bytes of the original data packet.

connection
Specifies the handle of an open connection associated with the communication port which data packets are received from and sent to.

portA and portB
Specify two pointers to the two end ports of the underlying channel.

Driver Functions

A driver is composed of driver functions prefixed by a string (xxx, for example). The driver has the option to implement each of the following functions.

```c
int xxx_driver_init(void);
void xxx_driver_release(void);
int xxx_open_A2B(DRVRPKT *pktX, unsigned int *timer_interval);
int xxx_open_B2A(DRVRPKT *pktX, unsigned int *timer_interval);
void xxx_close_A2B(DRVRPKT *pktX);
void xxx_close_B2A(DRVRPKT *pktX);
int xxx_exec_A2B(DRVRPKT *pktX);
int xxx_exec_B2A(DRVRPKT *pktX);
int xxx_timer_A2B(DRVRPKT *pktX, unsigned int *timer_interval);
int xxx_timer_B2A(DRVRPKT *pktX, unsigned int *timer_interval);
```

The function xxx_driver_init is called when the associated channel is established. In this function, you can execute global parameter initialization, memory allocation, and etc. On the other hand, the function xxx_driver_release is called when the program exits and the associated channel must be released.

The function xxx_open_A2B/xxx_open_B2A is called when an open connection is set up at one end (port A/B) of the channel.

```c
int xxx_open_A2B(DRVRPKT *pktX, unsigned int *timer_interval);
inxxx_open_B2A(DRVRPKT *pktX, unsigned int *timer_interval);
```
Parameters

pktX

It is a pointer to a structure DRVPKT. Its member connection specifies the handle of the open connection. The function normally assigns the address of an allocated memory space to its member private_data. The memory will be referenced when the following functions are executed. A NULL value can be assigned if the reference is of no importance.

timer_interval

If the function specifies a value (in milliseconds) to this parameter and the driver has defined the timer function xxx_timer_A2B/xxx_timer_B2A, the engine would periodically call the timer function.

Return Values

If the function has allocated memory successfully, it returns MPC_DRV_OK. Otherwise, it returns MPC_DRV_FAIL, and the MPC engine closes the connection accordingly.

Example

int rfc2217_open_A2B(DRVPKT *pktB, unsigned int *timer_interval)
{
  pktB->private_data = calloc(1, 256);
  if (pktB->private_data)
    return MPC_DRV_OK;
  else
    return MPC_DRV_FAIL;
}

If the driver needs to respond the established connection with data, it should store the data on the allocated memory, assign packet_data and packet_size of pktX, and return the function with the symbolic constant MPC_DRV_BACKWARD.

The function xxx_close_A2B/xxx_close_B2A is implemented when the open connection is broken at one end (port A/B) of the channel. Normally, this function frees up the allocated memory pointed to by the member private_data of a structure DRVPKT.

void xxx_close_A2B(DRVPKT *pktX);
void xxx_close_B2A(DRVPKT *pktX);

Example

void rfc2217_close_A2B(DRVPKT *pktB)
{
  if (pktB->private_data) free(pktB->private_data);
}

The function xxx_exec_A2B is called each time a data packet is passed from port A to this driver. In contrast, the function xxx_exec_B2A handles data packets in the reverse direction. Use the following formats to implement these functions:

int xxx_exec_A2B(DRVPKT *pktX);
int xxx_exec_B2A(DRVPKT *pktX);
Parameters

pktX

It is a pointer to a structure DRVRPKT. Before the function is called, the engine prepares the structure body DRVRPKT pointed to by the argument pktX. Member private_data points to the memory space allocated in function xxx_open_A2B or xxx_open_B2A. Member packet_data points to the data packet, and member packet_size is the length of the data packet. In addition, member packet_consumed has the same value as member packet_size. This implies that the engine assumes that the function consumes the whole packet.

After processing the data, the function is responsible for pointing member packet_data to the memory area (normally member private_data) that stores the resulting data, and specify the length of it in member packet_size. If the function decides not to forward data to the next driver, it sets packet_size to zero. Furthermore, this function must specify member packet_consumed to be the length of the data being consumed, and informs the engine to remove the processed data.

Return Values

This function returns one of the following symbolic constants:

- MPC_DRV_FORWARD – indicates that the driver has processed data successfully and asks the engine to keep forwarding the resulting data packet.
- MPC_DRV_BACKWARD – indicates that the driver has processed data successfully and asks the engine to pass the resulting data packet in the reverse direction.
- MPC_DRV_FAIL – indicates that the driver has stopped forwarding the data packet.

Each driver has the option to implement two timer functions, xxx_timer_A2B and xxx_timer_B2A. They are called by the engine periodically.

Parameters

pktX

It is a pointer to a structure DRVRPKT. Member private_data points to the memory space allocated in function xxx_open_A2B or xxx_open_B2A. The function uses member packet_data and packet_size of pktX to forward data.

timer_interval

This parameter specifies the time interval (in milliseconds) between each call. Before ending the call, the function can change the timer interval of the timer.

Remarks

The main role of a timer function is normally to generate a data query, and asks the engine to relay the data to the next driver or a target port.

Example

```c
int mbrtu_timer_A2B(DRVRPKT *pkt, unsigned int *timer_in)
{
    unsigned char *buffer = (unsigned char *) pkt->private_data;
    buffer[0] = 0x01;
    buffer[1] = 0x03; /* read data */
    buffer[2] = 0x00; /* read start address HI */
    return MPC_DRV_FORWARD;
}
```
buffer[3] = 0x00; /* read start address LO */
buffer[4] = 0x00; /* # of elements HI */
buffer[5] = 0x01; /* # of elements LO */

pkt->packet_size = 6;
pkt->packet_data = buffer;

return MPC_DRV_FORWARD;
}

Building a Linux MPC Driver

Before building a Linux driver, you need to create a Makefile. For example, the following Makefile would be used with the UC-7420-LX.

CC=mxscaleb-gcc
STRIP=mxscaleb-strip -s
AR=mxscaleb-ar rcs

EXEC=dxxx.dll

CFLAGS=-Wall -fPIC
LINK=$(CC) -shared -Wl,-soname,$(EXEC) -o $@
LIBS=
# add objects
OBJS= dxxx.o
DEPS=

all: $(EXEC)
$(EXEC): $(OBJS) $(DEPS)
$(LINK) $(OBJS) $(LIBS)
clean:
rm -f *.o $(EXEC)

Compile the shared object, dxxx.dll, for the driver.
> make

Building a WinCE MPC Driver

The following steps show you how to generate a project, and program a driver with Microsoft eVC++ 4.0.

1. Open the eVC++ 4.0 program. On the menu bar, click File and New… to create a new project.
2. In the pop-up window, select WCE Dynamic-Link Library. Enter a project name and then click OK.

3. Select An empty Windows CE DLL project, and then click Finish.
4. Click OK to complete the project creation.

5. On the eVC++ 4.0 menu bar, click **Project → Add to Project → Files** to add source files to the project. Remember to add an empty file, `dxxx.def`. 
6. Edit the source files.

7. Edit the file `dxxx.def`. Behind the keyword `LIBRARY` in the first line, specify the name of the dll, `dxxx` for example. Add the keyword `EXPORTS` in the second line, and then add the function names of the driver, as shown below.

8. Press the F7 key to compile the driver.