Freescale Semiconductor
Application Note

Document Number: AN4345
Rev. 0, August 2011

Using MQX Communications Drivers to Implement Protocol Converters

by: Wang Hao
Microcontroller Solutions Group

1 Introduction

This application note gives a brief introduction to existing MQX drivers for common communications protocols such as SPI, I^{2}C, USB, UART, and Ethernet, and shows how to use these drivers to make the job of implementing a protocol converter easier.

Many user applications need to use communications modules on an MCU to interface with external devices. Some examples:

- Using I^{2}C or SPI to read sampled data from a sensor
- Storing a data log regularly to a USB stick
- Transferring data via Ethernet
- Displaying real-time sensor values in a web server

Sometimes there is even a requirement that data be relayed from one communications interface to another, such as from I^{2}C to SPI, serial to Ethernet, etc.

There are different ways to implement a protocol converter. The user can do it using only hardware by using the CPLD or FPGA to implement the required conversion logic. Alternatively the implementation can be through software alone, using an MCU equipped with communications modules such as UART, I^{2}C, SPI, etc., and then writing the appropriate software drivers.

Contents

© 2011 Freescale Semiconductor, Inc.

But for end users, reading lengthy reference manuals to find out how to program a module is time-consuming. For that reason, having some sample code to help the user get started will be very helpful. To bridge this need, both RTOS-based and bare-metal solutions are provided: users can either use ready-to-use software drivers included in the MQX software solution, or use Processor Expert (integrated in the CodeWarrior tool suite) to specify the functions needed in a graphic user interface, and let PE help you generate the code.

This application note will focus on how to use MQX communications drivers to speed up your design. The latter approach, using Processor Expert to write communications drivers, will be covered in an application note that will be created later.

2 MQX communications drivers

The MQX software solution includes many communications drivers, located in <MQX installation folder>\mqx\source\io. You can find drivers for every available communications module on devices in the Kinetis, Coldfire, and PowerPC families. Current supported communications drivers in the MQX solution include:

- CAN drivers — support either FlexCAN or MSCAN, depending on the MCU
- I^{2}C drivers
- SPI drivers — support either DSPI, QSPI, or Coldfire V1 SPI module, depending on the MCU
- Serial drivers — support RS-232 and RS-485
- Ethernet drivers — support legacy Ethernet controller on Coldfire devices, as well as new Ethernet controller equipped with IEEE1588 on Kinetis devices
- USB drivers

The first benefit of using I/O drivers under MQX is that you don't need to understand the underlying details of how to program the communications module to send or receive data — you just need to use file I/O APIs such as open, read, write, ioctl, or close to control the device. Refer to either existing sample code in <MQX installation folder>\mqx\examples or in Freescale document MQXIOUG, *Freescale MQX I/O Drivers Users Guide,* for any particulars of a specific driver, such as what kind of IOCTL commands it accepts.

The second benefit is that when you are using file I/O to access the device, your code will be portable from one platform to another. Therefore when you change from Coldfire to Kinetis, the only work needed will be to recompile your code using the MQX libraries in <MQX installation folder>\lib\<specific platform folder>.

As you may notice, some communications modules such as Ethernet and USB actually require a communications stack in order to work. Therefore the access method is different. Table 1 shows a summary of the access methods in MQX, as well as reference documents for each communications module.

Table 1. Different access methods and reference documents for communications modules

Communications module	MQX access method	Reference material (title and ID)
CAN, I^{2}C, SPI, Serial	File I/O	*Freescale MQX I/O Drivers Users Guide* — MQXIOUG
Ethernet	Socket interface	*Freescale MQX RTCS User's Guide* — MQXRTCSUG
USB	USB stack-specific APIs	*Freescale MQX USB Host User's Guide* — MQXUSBHOSTUG

2.1 Common elements of MQX communications drivers

There are some elements that are quite similar among different MQX communications drivers. Here is a list for your reference:

- Driver installation method
  Each device driver has a driver-specific installation function which is called in init_bsp.c under <MQX installation folder>\source\bsp\<board folder>. That function will then call _io_dev_install internally. For example, the following code snippet installs the I^2C device driver and initializes I^2C with information found in the I^2C initialization record.

```
#if BSPCFG_ENABLE_I2C0
   _ki2c_polled_install("i2c0:", &_bsp_i2c0_init);
#endif
```

- Initialization record
  Stores initial settings for the communications driver, such as the channel used, operation mode, baud rate, interrupt level, and transmit and receive buffer sizes. Here is an example for an I^2C initialization record.

```
const KI2C_INIT_STRUCT _bsp_i2c0_init = {
   0,                       /* I2C channel    */
   BSP_I2C0_MODE,           /* I2C mode       */
   BSP_I2C0_ADDRESS,        /* I2C address    */
   BSP_I2C0_BAUD_RATE,      /* I2C baud rate  */
   BSP_I2C0_INT_LEVEL,      /* I2C int level  */
   BSP_I2C0_INT_SUBLEVEL,   /* I2C int sublvl */
   BSP_I2C0_TX_BUFFER_SIZE,/* I2C int tx buf */
   BSP_I2C0_RX_BUFFER_SIZE /* I2C int rx buf */
};
```

- I/O control commands

  Device drivers have many I/O control commands which can be issued with an ioctl call. This allows configuration of the driver operation or the ability to return driver settings and current status. Each communications driver has its own specific I/O control commands — please refer to the MQX I/O driver user guide for details.

2.2 I^2C driver

Accessing an I^2C device with file I/O is quite straightforward. Here is the code snippet for accessing the accelerometer MMA7660 with I^2C interface on the K60 TWR board.

```
void InitializeI2C()
{
  /* Open the I2C driver, and assign a I2C device handler*/
  fd = fopen ("i2c0:", NULL);
  …
  /* Set I2C into Master mode */
  ioctl (fd, IO_IOCTL_I2C_SET_MASTER_MODE, NULL);
}

void write_I2C(int i2c_device_address, uchar reg, uchar value)
{
  uchar data[2];

  data[0]=reg;   //Sensor register
  data[1]=value; //Byte of data to write to register

  /* Set the destination address */
  ioctl (fd, IO_IOCTL_I2C_SET_DESTINATION_ADDRESS, &i2c_device_address);

  /* Write 2 bytes of data: the desired register and then the data */
  fwrite (&data, 1, 2, fd);
  fflush (fd);

  /* Send out stop */
  ioctl (fd, IO_IOCTL_I2C_STOP, NULL);
}

void read_I2C(int i2c_device_address, int sensor, int length)
```

```
{
  int n=length;

  //The starting register for the particular sensor requested
  uchar reg=sensor;

  //Set the I2C destination address
  ioctl (fd, IO_IOCTL_I2C_SET_DESTINATION_ADDRESS, &i2c_device_address);

  //Tell the QE96 which sensor data to get
  fwrite (&reg, 1, 1, fd);

  //Wait for completion
  fflush (fd);

  //Do a repeated start to avoid giving up control
  ioctl (fd, IO_IOCTL_I2C_REPEATED_START, NULL);

  //Set how many bytes to read
  ioctl (fd, IO_IOCTL_I2C_SET_RX_REQUEST, &n);

  //Read n bytes of data and put it into the recv_buffer
  fread (&recv_buffer, 1, n, fd);

  //Wait for completion
  fflush (fd);

  //Send out stop
  ioctl (fd, IO_IOCTL_I2C_STOP, NULL);
}
```

You can see that most control of an I^{2}C device is done by issuing ioctl commands to it. Table 2 shows the ioctl commands used in this example, for full commands supported by the I^{2}C driver. Please refer to *Freescale MQX I/O Drivers Users Guide* for more details.

Table 2. I^{2}C ioctl commands

Command	Description
IO_IOCTL_I2C_SET_MASTER_MODE	Set device to I^{2}C master mode.
IO_IOCTL_I2C_SET_DESTINATION_ADDRESS	Set address of called device — in this case, it’s the address for MMA7660.
IO_IOCTL_I2C_STOP	Generate I^{2}C stop condition.
IO_IOCTL_I2C_REPEATED_START	Initiate I^{2}C repeated start condition.
IO_IOCTL_I2C_SET_RX_REQUEST	Set number of bytes to read before stop.

2.3 SPI driver

The next code snippet is an example of using the SPI MQX driver to access SPI flash on a TWR-MEM card. Implemented SPI instructions include read status, memory read, and memory write. For a full example, please refer to files in <MQX installation folder>\mqx\examples\spi.

```
uint_8 memory_read_status (MQX_FILE_PTR spifd)
{
   uint_32 result;
   uint_8 state = 0xFF;

   send_buffer[0] = SPI_MEMORY_READ_STATUS;
   /* Write instruction */
   result = fwrite (send_buffer, 1, 1, spifd);

   /* Read memory status */
   result = fread (&state, 1, 1, spifd);

   /* Wait till transfer end (and deactivate CS) */
   fflush (spifd);

   return state;
}

void memory_write_byte (MQX_FILE_PTR spifd, uint_32 addr, uchar data)
{
   uint_32 result;
…
   send_buffer[0] = SPI_MEMORY_WRITE_DATA;
   for (result = SPI_MEMORY_ADDRESS_BYTES; result != 0; result--)
   {
      send_buffer[result] = (addr >> ((SPI_MEMORY_ADDRESS_BYTES - result) << 3)) & 0xFF; //
Address
   }
   send_buffer[1 + SPI_MEMORY_ADDRESS_BYTES] = data; //Data

   /* Write instruction, address and byte */
   result = fwrite (send_buffer, 1, 1 + SPI_MEMORY_ADDRESS_BYTES + 1, spifd);

   /* Wait till transfer end (and deactivate CS) */
   fflush (spifd);
}

uint_8 memory_read_byte (MQX_FILE_PTR spifd, uint_32 addr)
{
   uint_32 result;
   uint_8 data = 0;

   send_buffer[0] = SPI_MEMORY_READ_DATA;

   for (result = SPI_MEMORY_ADDRESS_BYTES; result != 0; result--)
   {
      send_buffer[result] = (addr >> ((SPI_MEMORY_ADDRESS_BYTES - result) << 3)) & 0xFF; //
Address
   }

   /* Write instruction and address */
   result = fwrite (send_buffer, 1, 1 + SPI_MEMORY_ADDRESS_BYTES, spifd);

   /* Read data from memory */
   result = fread (&data, 1, 1, spifd);

   /* Wait till transfer end (and deactivate CS) */
   fflush (spifd);

   return data;
}
```

2.4 Serial driver

Serial drivers do not require much explanation. When MQX starts, it is normally initialized with some I/O driver for at least one serial port which will be used as the default I/O channel. Such functions as printf in your code will assume it's sending or receiving data from that channel. You can change the default I/O channel in the board configuration file — for example, for K60 TWR, it's twrk60n512.h in the mqx\source\bsp folder.

```
#define BSP_DEFAULT_IO_CHANNEL       "ttyf:"    /* OSJTAG-COM   polled mode   */
```

For RS-485 examples, refer to the sample code in mqx\examples\rs485.

2.5 Ethernet driver and socket interface

Using an Ethernet interface in MQX is quite different from the approach used by I^{2}C, SPI, or serial drivers. Though there is an Ethernet driver in <MQX installation folder>\mqx\io\enet, normally you will use the high-level socket interface provided by the RTCS stack to transmit and receive data. The next subsections explain the setup sequence required to start using RTCS, as well as how data sent or received through the socket interface binds to the Ethernet interface.

2.5.1 Setting up RTCS

1. Change RTCS creation parameters, such as initial, growing, and maximum value for packet control block, message pool, and socket partition.

```
_RTCSPCB_init = 4;
_RTCSPCB_grow = 2;
_RTCSPCB_max = 20;
_RTCS_msgpool_init = 4;
_RTCS_msgpool_grow = 2;
_RTCS_msgpool_max  = 20;
_RTCS_socket_part_init = 4;
_RTCS_socket_part_grow = 2;
_RTCS_socket_part_max  = 20;
```

2. Create RTCS by calling RTCS_create(). This function will create the TCPIP task which services the request from socket layer and application. It also services incoming packets from the link layer. The RTCS task communicates with the application task through the message queue.

```
uint_32 RTCS_create(void)
{
…
error = RTCS_task_create("TCP/IP", _RTCSTASK_priority, _RTCSTASK_stacksize, TCPIP_task,
NULL);
…
}

void TCPIP_task(pointer  dummy, pointer  creator)
{
…
tcpip_msg = (TCPIP_MESSAGE_PTR)RTCS_msgq_receive(tcpip_qid, timeout, RTCS_data_ptr-
>TCPIP_msg_pool);
if (tcpip_msg) {
         if (NULL != tcpip_msg->COMMAND)
            tcpip_msg->COMMAND(tcpip_msg->DATA);

         RTCS_msg_free(tcpip_msg);
}
…
}
```

3. Initialize device interface

   RTCS is independent of device interfaces. Currently it supports either Ethernet devices or point-to-point devices. Before writing code to send or receive data through the underlying network device, the device needs to be initialized and put in a proper state.
   For an Ethernet device, calling ENET_initialize() is needed to initialize the Ethernet device. It internally calls ENET_initialize_ex() and passes it the ENET_default_params parameter, which is part of the Ethernet initialization record defined in init_enet.c.

```
/* Initialize Ethernet device and return device handle */
ENET_initialize(DEMOCFG_DEFAULT_DEVICE, address, 0, &ehandle);
```

```
uint_32 ENET_initialize(
      uint_32             device,
      _enet_address       address,
      uint_32             flags,
      _enet_handle _PTR_   handle)
{
…
ENET_initialize_ex(&ENET_default_params[device],address,handle);
…
}
```

The following code snippet is the initialization record for the Kinetis Ethernet driver. Here MACNET_IF and phy_ksz8041_IF are initialization structures for the MAC and PHY interface of Kinetis, respectively. Both are part of the Kinetis Ethernet I/O driver.

```
const ENET_IF_STRUCT ENET_0 = {
    &MACNET_IF,-> defined in macnet_init.c
    &phy_ksz8041_IF,-> defined in phy_ksz8041.c
    MACNET_DEVICE_0,
    MACNET_DEVICE_0,
    BSP_ENET0_PHY_ADDR,
    BSP_ENET0_PHY_MII_SPEED
};

const ENET_PARAM_STRUCT ENET_default_params[BSP_ENET_DEVICE_COUNT] = {
    {
        &ENET_0,
        Auto_Negotiate,
        ENET_OPTION_RMII | ENET_OPTION_PTP_MASTER_CLK,
        BSPCFG_TX_RING_LEN,   // # tx ring entries
        BSPCFG_TX_RING_LEN,   // # large tx packets
        ENET_FRAMESIZE,       // tx packet size
        BSPCFG_RX_RING_LEN,   // # rx ring entries
        BSPCFG_RX_RING_LEN,   // # normal rx packets - must be >= rx ring entries
        ENET_FRAMESIZE,       // ENET_FRAMESIZE,   // rx packet size
        BSPCFG_RX_RING_LEN,   // # rx PCBs - should be >= large rx packets.
        0,
        0
    },
};
```

4. Add the device interface to RTCS by calling RTCS_if_add(). Here RTCS_IF_ENET is the structure used for binding the IP address to the underlying Ethernet packet driver.

```
/* passing the Ethernet device handle */
RTCS_if_add(ehandle, RTCS_IF_ENET, &ihandle);

static const RTCS_IF_STRUCT rtcs_enet = {
   IPE_open,
   IPE_close,
   IPE_send_IP
#if RTCSCFG_ENABLE_IGMP && BSPCFG_ENABLE_ENET_MULTICAST
   ,
   IPE_join,
   IPE_leave
#endif
};
const RTCS_IF_STRUCT_PTR RTCS_IF_ENET = (RTCS_IF_STRUCT_PTR)&rtcs_enet;
```

5. Bind IP address to device interface.

```
RTCS_if_bind(ihandle, ENET_IPADDR, ENET_IPMASK);
```

6. Add default gateway with RTCS_gate_add().

2.5.2 Socket interface

A socket is an abstraction which identifies a communications endpoint and includes the type of socket and socket address. There are two types of socket: a datagram socket using the UDP protocol, and a stream socket using the TCP protocol. A socket address is identified by both the port number and IP address.

The steps for using a socket interface are:

1. Create a new socket with socket().
2. Bind the socket to the local address with bind().
3. Assign a remote IP address for the stream socket by calling connect() for a server application, or by calling listen() followed by accept() for a client application.
4. Send data with sendto() for a UDP socket or send() for a TCP socket.
5. Receive data with recvfrom() for a UDP socket or recv() for a TCP socket.
6. When the data transfer is finished, destroy the socket with shutdown().

2.5.3 HTTP server example

To enable a web server in your application, there are three main steps needed. For detailed steps, please refer to the sample code in <MQX installation folder>\demo and to Freescale application note AN3907, “Using MQX Libraries."

1. Install trivial file system for web pages.

   ```
   error = _io_tfs_install("tfs:", tfs_data)
   ```
2. Initialize HTTP server with trivial file system information. Pass root directory as first parameter.

   ```
   const HTTPD_ROOT_DIR_STRUCT root_dir[] = {
       { "", "tfs:" },      //Internal flash with Trivial File System (TFS)
       { "sdcard", "a:" }, //SDCard
       { 0, 0 }
   };

   httpd_server_init((HTTPD_ROOT_DIR_STRUCT*)root_dir, "\\mqx.html");
   ```
3. Execute http_server_run() to handle incoming HTTP connections.

   ```
   httpd_server_run(server);
   ```

Looking into the internals of http_server_init() and http_server_run() will help to understand how socket API is used to realize certain TCP/IP applications. In the HTTP server case, it’s using a TCP socket and listening for HTTP connections. This will determine which socket interface it will use.

Please note http_server_init() will call httpd_init() internally, while http_server_run() will create http_server_task to handle incoming connections.

```
HTTPD_STRUCT* httpd_init(HTTPD_PARAMS_STRUCT *params) {
struct sockaddr_in sin;
HTTPD_STRUCT *server = NULL;
…
/* initial listen socket */
server->sock = socket(AF_INET, SOCK_STREAM, 0)

…
sin.sin_port = server->params->port;//port
sin.sin_addr.s_addr = server->params->address;//IP address
sin.sin_family = AF_INET;

/* bind socket to IP address */
bind(server->sock, &sin, sizeof(sin));

/* listen for client connections */
listen(server->sock, 5);
…
```

```
}

/* handles incoming HTTP connection */
static void httpd_server_task(pointer init_ptr, pointer creator) {
…
while (1) {
        httpd_server_poll(server, 1); //this calls httpd_ses_poll internally
}
…
}

void httpd_ses_poll(HTTPD_STRUCT *server, HTTPD_SESSION_STRUCT *session) {
struct sockaddr_in sin;
…
accept(server->sock, &sin, &len);
…
while (HTTPD_SESSION_VALID == session->valid)
{
    httpd_ses_process(server, session);
}
…
}
```

Here http_ses_process() implements the httpd session state machine. It will use recv() or send() to communicate with the client based on what kind of request it received from the client. Please refer to source files in <MQX installation folder>\rtcs \source\httpd for details.

2.6 USB driver

The MQX USB stack provides USB host and device controller drivers, plus common class drivers such as mass storage device class, HID class, CDC class, etc. Communication with a USB device is allowed with either USB class driver API or low-level host or device controller API. Class driver APIs will call host or device controller APIs internally, so the next subsections will discuss the major steps needed to access a USB device with controller level API. For a specific class driver API, please refer to sample code in <MQX installation folder>\usb.

2.6.1 Setup sequence for USB host

Here is an example of how to set up a mass-storage class USB host to allow connecting to a USB stick.

1. Install USB host controller driver, passing the callback table from the underlying USB host controller. Here _bsp_usb_host_callback_table is defined in either ehci or khci folder in <MQX installation folder>\usb\host\source \host, depending on which USB controller is used.

   ```
   _usb_host_driver_install(0,  (pointer) &_bsp_usb_host_callback_table);
   ```
2. Initialize USB hardware and install USB interrupt handler.

   ```
   _usb_host_init(0, 4, &host_handle);
   ```
3. Register device class with USB host, passing ClassDriverInfoTable as parameter.

   ```
   static const USB_HOST_DRIVER_INFO ClassDriverInfoTable[] =
   {
      /* Vendor ID Product ID Class Sub-Class Protocol Reserved Application call back */
      /* Floppy drive */
      {{0x00,0x00}, {0x00,0x00}, USB_CLASS_MASS_STORAGE, USB_SUBCLASS_MASS_UFI,
   USB_PROTOCOL_MASS_BULK, 0, usb_host_mass_device_event },

      /* USB 2.0 hard drive */
      {{0x00,0x00}, {0x00,0x00}, USB_CLASS_MASS_STORAGE, USB_SUBCLASS_MASS_SCSI,
   USB_PROTOCOL_MASS_BULK, 0, usb_host_mass_device_event},
   ```

```
        /* USB hub */
        {{0x00,0x00}, {0x00,0x00}, USB_CLASS_HUB, USB_SUBCLASS_HUB_NONE,
    USB_PROTOCOL_HUB_LS, 0, usb_host_hub_device_event},

        /* End of list */
        {{0x00,0x00}, {0x00,0x00}, 0,0,0,0, NULL}
    };
    _usb_host_driver_info_register(host_handle, (pointer)ClassDriverInfoTable);
```

4. Register services with _usb_host_register_service().
5. Write device callback functions to handle different event types. Normally only USB_ATTACH_EVENT and USB_DETACH_EVENT need to be handled. This next code example selects the interface after the ATTACH event and installs the USB file system for the USB stick. It then removes the file system after the DETACH event.

```
for(;;){
if(device.STATE == USB_DEVICE_ATTACHED){
_usb_hostdev_select_interface(device.DEV_HANDLE,
               device.INTF_HANDLE, (pointer)&device.CLASS_INTF);
device.STATE = USB_DEVICE_INTERFACED;
usb_fs_handle = usb_filesystem_install( USB_handle, "USB:", "PM_C1:", "c:" );
}
else if ( device.STATE==USB_DEVICE_DETACHED) {
// remove the file system
usb_filesystem_uninstall(usb_fs_handle);
}
}
```

2.6.2 Setup sequence for USB device

1. Initialize USB device controller with _usb_device_init(). Internally _usb_device_init() calls _usb_dev_driver_install() to install the callback table for the underlying USB device controller.

```
_usb_device_init(0, controller_handle, endpionts);
```

2. Register services with _usb_device_register_service(). For example, the following code appears in USB_Class_Init() to register service for a reset event on the USB bus.

```
 _usb_device_register_service(handle,
            USB_SERVICE_BUS_RESET, USB_Reset_Service,(void*)class_object_ptr);
```

3. Respond to IN token from host on non-control endpoints with _usb_device_send_data().
4. Respond to OUT token from host on non-control endpoints with _usb_device_recv_data().

3 Protocol converter

There are a large number of samples in MQX which implement protocol converters, such as the virtual_com example to implement serial-to-USB conversion, and the virtual_nic example to implement serial-to-Ethernet conversion. You can use these examples as a starting point to implement your own protocol converter. Here is the code snippet for virtual_com demo. It mostly uses class driver APIs such as USB_Class_CDC_Init(), USB_Class_CDC_Send_Data(), and USB_Class_CDC_Recv_Data(). These APIs will then call device controller APIs internally.

```
/* virtual_com demo */
void TestApp_Init(void) {
CDC_CONFIG_STRUCT cdc_config;
…
/* register application and notify callback function */
cdc_config.cdc_class_cb.callback = USB_App_Callback;
cdc_config.param_callback.callback = USB_Notif_Callback;
/* set notify endpoint */
cdc_config.cic_send_endpoint = CIC_NOTIF_ENDPOINT;
cdc_config.dic_send_endpoint = DIC_BULK_IN_ENDPOINT;
cdc_config.dic_recv_endpoint = DIC_BULK_OUT_ENDPOINT;
…
/* Initialize the USB interface, passing cdc_config */
g_app_handle = USB_Class_CDC_Init(&cdc_config);

while(TRUE)
{
if((start_app==TRUE) && (start_transactions==TRUE))
{
    if(g_recv_size)
{
//copy data from recv buffer to send buffer
}
if(g_send_size)
{
…
USB_Class_CDC_Send_Data(g_app_handle,DIC_BULK_IN_ENDPOINT,
        g_curr_send_buf, size);
…
}
}
}
}

void USB_Notif_Callback(uint_8 event_type,void* val,pointer arg)
{
…
if((event_type == USB_APP_DATA_RECEIVED)&&
                         (start_transactions == TRUE))
{
BytesToBeCopied = dp_rcv->data_size;
       for(index = 0; index < BytesToBeCopied; index++)
       {
           g_curr_recv_buf[index] = dp_rcv->data_ptr[index];
       }
       …

       /* Schedule buffer for next receive event */
USB_Class_CDC_Recv_Data(handle, DIC_BULK_OUT_ENDPOINT,
                g_curr_recv_buf, DIC_BULK_OUT_ENDP_PACKET_SIZE);
 }
…
}
```

4 Conclusion

The MQX software solution provides a substantial amount of ready-to-use sample code which makes writing code for communications interfaces easier. You don’t need to understand low-level details of the communications modules and can quickly write demos to communicate with external devices, such as sample sensor data, and display it in a web server. There is also sample code for protocol converters, quite easy to change, which will be a good starting point to implement your own converter using MQX.

How to Reach Us:

Home Page:
www.freescale.com

Web Support:
http://www.freescale.com/support

USA/Europe or Locations Not Listed:
Freescale Semiconductor
Technical Information Center, EL516
2100 East Elliot Road
Tempe, Arizona 85284
+1-800-521-6274 or +1-480-768-2130
www.freescale.com/support

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
www.freescale.com/support

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064
Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor China Ltd.
Exchange Building 23F
No. 118 Jianguo Road
Chaoyang District
Beijing 100022
China
+86 10 5879 8000
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
1-800-441-2447 or +1-303-675-2140
Fax: +1-303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

Information in this document is provided solely to enable system and software implementers to use Freescale Semiconductors products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits or integrated circuits based on the information in this document.

Freescale Semiconductor reserves the right to make changes without further notice to any products herein. Freescale Semiconductor makes no warranty, representation, or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale Semiconductor assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any liability, including without limitation consequential or incidental damages. "Typical" parameters that may be provided in Freescale Semiconductor data sheets and/or specifications can and do vary in different applications and actual performance may vary over time. All operating parameters, including "Typicals", must be validated for each customer application by customer's technical experts. Freescale Semiconductor does not convey any license under its patent rights nor the rights of others. Freescale Semiconductor products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which failure of the Freescale Semiconductor product could create a situation where personal injury or death may occur. Should Buyer purchase or use Freescale Semiconductor products for any such unintended or unauthorized application, Buyer shall indemnify Freescale Semiconductor and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claims alleges that Freescale Semiconductor was negligent regarding the design or manufacture of the part.

RoHS-compliant and/or Pb-free versions of Freescale products have the functionality and electrical characteristics as their non-RoHS-complaint and/or non-Pb-free counterparts. For further information, see http://www.freescale.com or contact your Freescale sales representative.

For information on Freescale's Environmental Products program, go to http://www.freescale.com/epp.

Freescale™ and the Freescale logo are trademarks of Freescale Semiconductor, Inc. All other product or service names are the property of their respective owners.

© 2011 Freescale Semiconductor, Inc.

Document Number: AN4345
Rev. 0, August 2011