Freescale Semiconductor
Application Note

Document Number: AN3907
Rev. 0, 06/2009

Using MQX Libraries

by: Antonio Ramos, Luis Garabito
Applications Engineering
Freescale Technical Support

1 Introduction

Freescale MQX™ RTOS allows application development for real-time environments. It allows task-oriented application development faster as compared to the market. MQX also contains other core system components like RTCS network stack, MFS file system, and USB host/device drivers. These drivers can be included in a project as software libraries that provide full functionality with zero development time.

The MQX software libraries include:

- RTCS network stack
- Shell interface library
- USB (Host and Device) drivers
- MS-DOS File System Library (MFS)

This application note describes:

- Introduction to libraries
- Basic usage of each library
- How to add libraries to projects

Contents

For more information on the complete features and usage of the libraries, refer to the reference manual for each library.

2 MQX Libraries

2.1 RTCS Network Stack

The RTCS is an embedded Ethernet stack optimized to run on MQX RTOS. The RTCS provides different protocols to support applications like HTTP, Telnet, FTP, DHCP, and others. Figure 1 shows different layers and some of the protocols included in the RTCS.

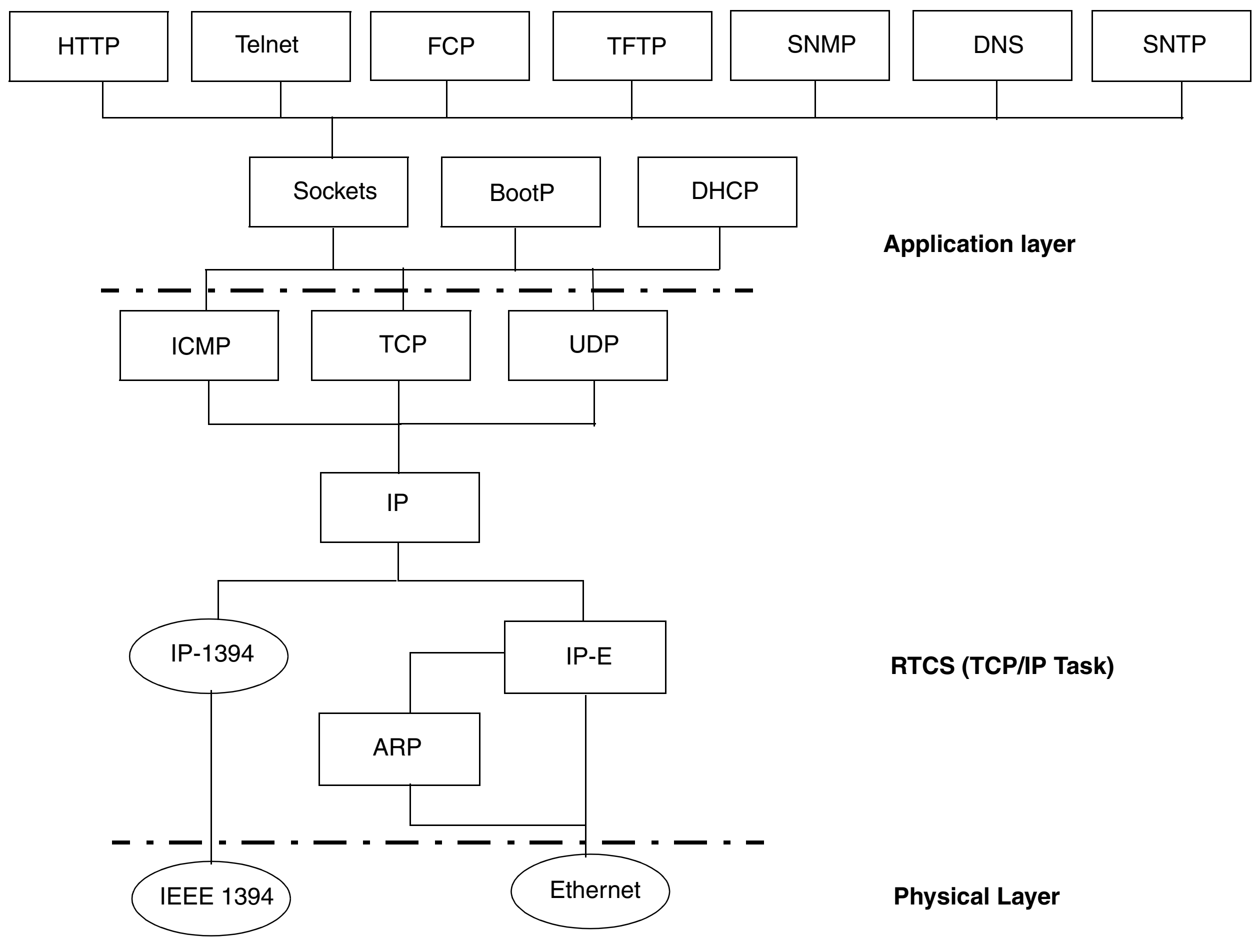

Figure 1. RTCS Layers and Protocols

The RTCS implements the protocols used by the application layer shown in Figure 1. This is called the RTCS TCP/IP task. This task interfaces with the physical layer and is used by the application layer. This application note explains the basic setup of the RTCS. The application note also uses HTTP, DHCP, and socket applications as practical examples on how to use the RTCS.

2.1.1 Setting the RTCS

Figure 2 shows the process required to set the RTCS with the default settings.

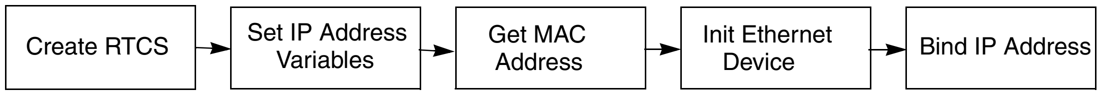

Figure 2. RTCS Setup Process

1. Create the RTCS. This step reserves memory for different RTCS parameters and initializes the TCP/IP task.
2. Set the RTCS IP address to the device after initialization. The IP address is stored using a set of variables from the library.
3. Calculate the MAC address for the device using the variables.
4. Initialize the Ethernet device available in the board with the calculated MAC address.
5. After Ethernet device initialization, bind the device to the IP address and it is ready to communicate.

After this process is completed, it is possible to add applications like Telnet, FTP, HTTP, and others.

2.1.1.1 Creating the RTCS

To create RTCS:

1. Include the RTCS.h and IPCFG.h header files to the project.

```
/* RTCS and HTTP header files */
#include <rtcs.h>
#include  <ipcfg.h>
```

   These files have the prototypes for the library functions that create the RTCS, calculate the MAC address, initialize the Ethernet device, and bind the IP address to the Ethernet device.

2. Use a set of macros to store the IP address parameters. These parameters are stored in the IP address variables.

```
/* Ethernet IP Configuration */
#define ENET_IPADDRIPADDR(169,254,3,3)
#define ENET_IPMASKIPADDR(255,255,0,0)
#define ENET_IPGATEWAY  IPADDR(169,254,0,1)
```

3. To create the RTCS, execute the RTCS_create() function.

```
void initialize_networking(void)
{
    int_32error;
    IPCFG_IP_ADDRESS_DATAip_data;
    HTTPD_STRUCT *server;

    /* runtime RTCS configuration  */
    ENET_Rx_packet_size = 640+64;
    ENET_Tx_packet_size = 256+64;

    /* Create RTCS */
    error = RTCS_create();
```

2.1.1.2 Setting the IP Address Variables

Set the IP address variables from the library to the desired value. The BSP_DEFAULT_ENET_DEVICE macro is defined in the target board header file. For targets with one Ethernet interface, the default device is device number ‘0.’ The following code sets the address, mask, gateway, and server IP addresses.

```
/* Set IP Address Variables */
IPCFG_default_enet_device = BSP_DEFAULT_ENET_DEVICE;
IPCFG_default_ip_address = ENET_IPADDR;
IPCFG_default_ip_mask = ENET_IPMASK;
IPCFG_default_ip_gateway = ENET_IPGATEWAY;
LWDNS_server_ipaddr = ENET_IPGATEWAY;
```

2.1.1.3 Getting MAC Address

Use the ENET_get_mac_address() function to build a valid MAC address for the application. MAC address calculation takes the device number and IP address as parameters, and returns a valid MAC address in the IPCFG_default_enet_address variable.

```
/* Get MAC Address */
ENET_get_mac_address (IPCFG_default_enet_device, IPCFG_default_ip_address,
IPCFG_default_enet_address);
```

2.1.1.4 Initializing Ethernet Device

Use the ipcfg_init_device() function to tell the RTCS task what Ethernet device should be used and to set the calculated MAC address to that device. Once the MAC address is set and the device is initialized, it is possible to bind the device to the IP address.

```
/* Init Ethernet Device */
error = ipcfg_init_device (IPCFG_default_enet_device, IPCFG_default_enet_address);
```

2.1.1.5 Binding IP Address

Bind the IP address to the Ethernet device. Execute the ipcfg_bind_staticip() function with the device number and the ip_data structure as parameters.

```
/* Bind the IP Address Variables to the Ethernet device */
error = ipcfg_bind_staticip (IPCFG_default_enet_device, &ip_data);
}
```

The ip_data object is a local structure that contains the IP address configuration for the device. This is the same information set in the macros. The ip_data structure binds the Ethernet device with the IP address after it is initialized.

```
ip_data.ip = IPCFG_default_ip_address;
ip_data.mask = IPCFG_default_ip_mask;
ip_data.gateway = IPCFG_default_ip_gateway;
```

After the bind process executes, the RTCS is ready to communicate. After the bind process finalizes, it is possible to ping the IP address of the device and to connect it to a network. This is the basic setup of the RTCS.

To learn more about the TCP/IP stack configurable parameters, see *Freescale MQX™ RTCS™ User's Guide* (document MQXRTCSUG).

2.1.2 Hypertext Transport Protocol (HTTP) Server

The RTCS library includes an HTTP server application. To enable the target board as a web server, it is necessary to include the HTTP server application and the web page HTML files to the project.

2.1.2.1 Web Page Files (Trivial File System (TFS))

MQX includes the mktfs.exe application that converts web page files to a source code file. The tool is available in the \Freescale MQX 3.0\tools\ folder. The source file can be included in a project to add html information to an embedded project.

To convert web page files to a source code file:

1. Execute the tool in a command line with the name of the folder where the html files are stored and the name of the output source file. The folder must contains all the images and html files that are to be converted to a source file.

   mktfs.exe <Folder to be converted> <Output source file name>

   for example:

   C:\mktfs.exe web_pages tfs_data_file.c
2. The converted output of the web_pages folder is stored in the tfs_data_file.c file. This file has a TFS format. The HTML information is placed by default in an array named tfs_data.

   ```
   const TFS_DIR_ENTRY tfs_data[]
   ```
3. MQX supports the trivial file system. The _io_tfs_install() function installs the array located in the web page source code with the path, “tfs:”. The tfs_data array is declared as an extern object to include it from the web page source file.

   ```
   #include "tfs.h"

   extern const TFS_DIR_ENTRY tfs_data[];

   /* Instal file system */
   error = _io_tfs_install("tfs:", tfs_data);
   ```

2.1.2.2 Start the HTTP Server

To start the HTTP server:

1. Include the httpd.h file in the project. This file includes the prototypes for the functions that initialize the server parameters and start executing the server.

   ```
   #include "httpd.h"

   const HTTPD_ROOT_DIR_STRUCT root_dir[] = {
       { "", "tfs:" },
       { 0, 0 }
   };
   ```

2. Initialize the server with the trivial file system information. The root directory of the file system is passed as the first parameter. The root_dir structure contains an array of HTTP_ROOT_DIR_STRUCT objects. The elements of the structure are an alias and the path for the file system. This path must match the trivial file system installation path. The second parameter is the index page that is loaded when a client connects to the server. The httpd_server_init() function returns the server information.

   ```
   /* Init Server */
   server = httpd_server_init((HTTPD_ROOT_DIR_STRUCT*)root_dir, "\\cws.html");
   ```

3. Execute the http_server_run() function with the server information as parameter. This function creates a new task that uses the server information to handle any incoming HTTP connection.

   ```
   /* Run Server */
   httpd_server_run(server);
   ```

This completes the RTCS web server application setup. It is possible to add CGI content or to have different parameters for the web server. For more details on the complete features of the HTTP server application, see *Freescale MQX™ RTCS™ User's Guide* (document MQXRTCSUG).

2.1.3 Dynamic Host Control Protocol (DHCP)

DHCP can be enabled for the RTCS. This allows your device to automatically receive a valid IP address when it connects to a network.

To enable DHCP service:

1. Build the library with the User Datagram Protocol (UDP) and gateway options enabled.
2. Open the file:

   C:\Program Files\Freescale\Freescale MQX 3.0\config\m52259demo\user_config.h
3. Define the following macros to 1:

   ```
   #define RTCSCFG_ENABLE_UDP       1
   #define RTCSCFG_ENABLE_GATEWAYS  1
   ```

4. If any changes were done to user_config.h, open the RTCS library project and rebuild the library.
5. Set the MAC address. For this example, the MAC address is set with a fixed number. The ipcfg_init_device() function is used but with a fixed MAC address stored in the address variable.

   ```
   _enet_address  address = { 0x00, 0xcf, 0x52, 0x53, 0x54, 0xcc };

   /* Set MAC Address */
   ipcfg_init_device(IPCFG_default_enet_device, address);
   ```

6. Bind the IP address variables to the Ethernet device. The bind operation for DHCP requires a two step process.
   a) Execute the ipcfg_bind_dhcp() function on the Ethernet device to set the DHCP option to TRUE.
   b) Execute the ipcfg_poll_dhcp() function. This function finishes the binding process. The third parameter is the IP address, mask, and gateway information to be used if the process fails.

   ```
   /* Bind the IP Address Variables to the Ethernet device */
   ```

```
do
{
    _time_delay (200);
    error = ipcfg_bind_dhcp (IPCFG_default_enet_device, TRUE);
    printf ("Error during DHCP bind 1 %08x!\n", error);
} while (error == IPCFG_ERROR_BUSY);

if (error == IPCFG_ERROR_OK)
{
    do
    {
        _time_delay (200);
        error = ipcfg_poll_dhcp (IPCFG_default_enet_device, TRUE, &ip_data);
        printf ("Error during DHCP poll %08x!\n", error);
    }while (error == IPCFG_ERROR_BUSY);
}

if (error != IPCFG_ERROR_OK)
{
    printf ("Error during DHCP bind %08x!\n", error);
}
else
{
    printf ("DHCP bind successful.\n");
}
```

7. To get the IP address bound to the Ethernet device, execute ipcfg_get_ip() function. This function returns the address assigned by the DHCP to the device.

```
ipcfg_get_ip(0, &ip_data);
printf ("IP address = %x.\n",ip_data.ip);
```

2.1.4 Sockets

RTCS sockets are compatible with Unix BSD 4.4 and provide an interface for TCP and UDP protocols. A socket is an abstraction that identifies an endpoint. A socket might have a remote endpoint.

An application uses the following general steps to create and use sockets.

1. Create a new socket by calling socket() function, indicating whether the socket is a datagram socket or a stream socket.
2. Bind the socket to a local address by calling bind() function.
3. If the socket is a stream socket, assign a remote IP address by doing one of the following:
   — Call the connect() function.
   — Call the listen() function followed by the accept() function.
4. Send data with the sendto() function for a datagram socket or with the send() function for a stream socket.
5. Receive data with the recvfrom() function for a datagram socket or with the recv() function for a stream socket.
6. When data transfer is finished, optionally destroy the socket with the shutdown() function.

The process for datagram sockets is illustrated in Figure 3.

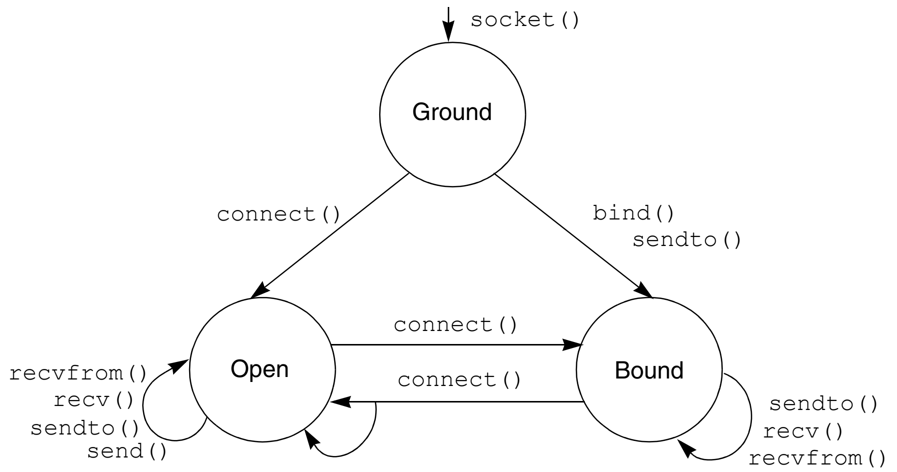

Figure 3. Datagram Sockets (UDP)

The process for stream sockets is illustrated in Figure 4.

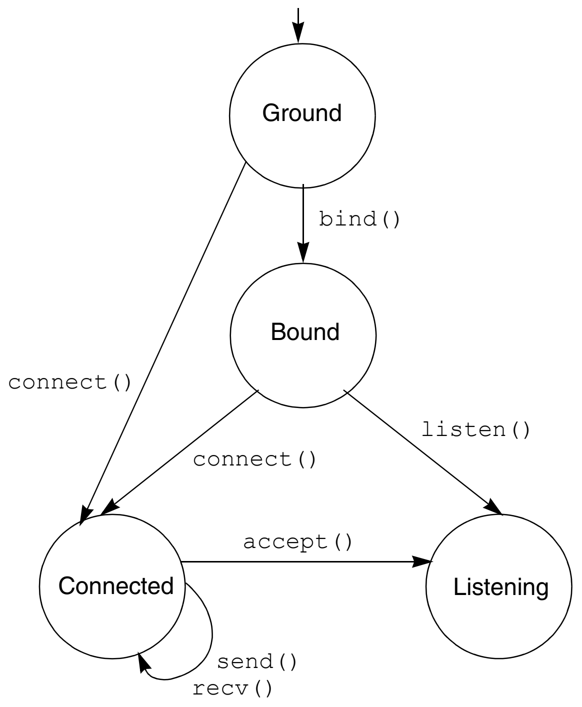

Figure 4. Stream Sockets (TCP)

2.1.5 Client-Server Model

The client-server model is the most common communication model used by the applications such as HTTP, Telnet, FTP, SSH, and others. In a client-server model, the server listens to client requests and waits for new connections. When a client needs to connect to a server, it sends a request. The server acknowledges the request, and if the client is supported by the server, the connection is established.

Communication can then continue between the client and the server. The client-server model is illustrated in Figure 5.

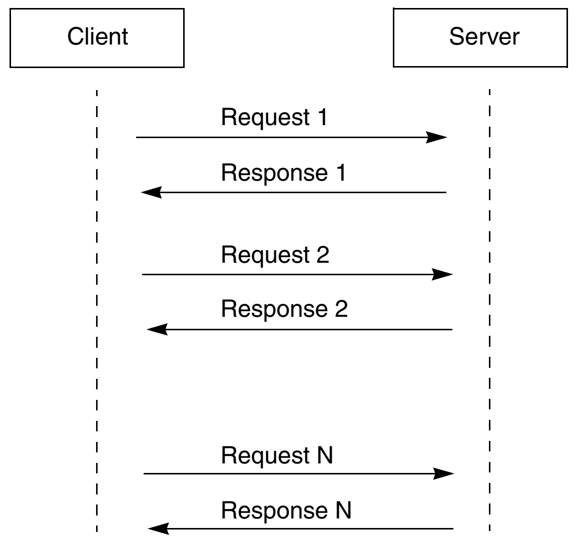

Figure 5. Client-Server Model

A TCP client-server communication can be implemented as in Figure 6.

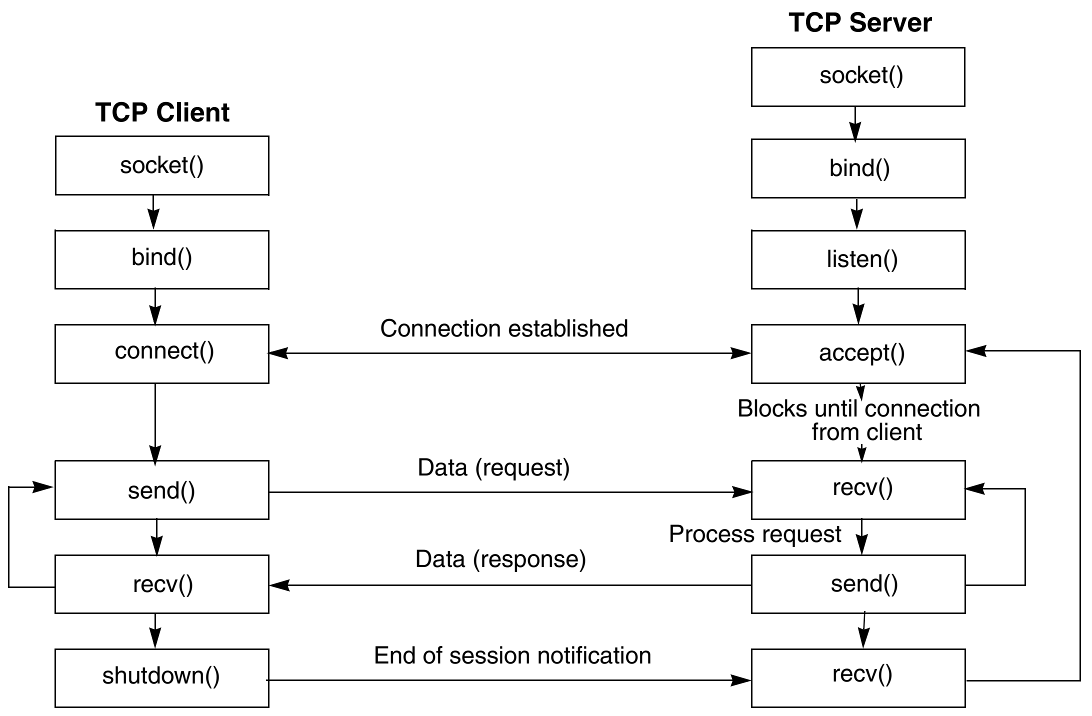

Figure 6. TCP Client-Server Communication Implementation

2.1.5.1 Server Implementation on MQX

For a server implementation, MQX uses a socket structure that is created and filled in this way:

```
sockaddr_in laddr;

laddr.sin_family     = AF_INET; // This field contains the address family, which
                                  //is always AF_INET when TCP or UDP is used.
laddr.sin_port        = CONNECT_PORT;  // port used for the connection.
laddr.sin_addr.s_addr = INADDR_ANY;    // This field contains the Internet address.
                                       // if INADDR_ANY is used then the local host is
                                       //used as the IP address
```

The socket() function creates an unbound socket in a communications domain. This function also returns a handler used in later function calls that operate on sockets.

```
uint_32 listensock;
listensock = socket(AF_INET, SOCK_STREAM, 0); //Create a TCP socket.
```

The bind() function assigns a local socket address to a socket.

```
bind(listensock, &laddr, sizeof(laddr));
```

The listen() function marks a socket as accepting connections.

```
listen(listensock, 0);
```

The accept() function extracts the first connection on the queue of pending connections, creates a new socket with the same socket type protocol and address family as the specified socket, and allocates a new file descriptor for that socket.

```
uint_32 sock;
sock = accept(listensock, &remote_addr, &remote_addr_len);
```

The recv() function receives a message from a socket.

```
uchar cRecvBuff[25]={0};
recv(sock, (void *)cRecvBuff, sizeof(cRecvBuff), 0);
```

The send() function initiates transmission of a message from the specified socket to its peer. The send() function sends a message only when the socket is connected.

```
uchar cSendBuff[25]="Hello I am the Server.\n";
send(sock, (void *)cSendBuff, sizeof(cSendBuff), 0);
```

2.1.5.2 Client Implementation on MQX

For a client implementation, it uses a socket structure that is created and filled in this way:

```
sockaddr_in    addr;

addr.sin_family      = AF_INET;
addr.sin_port        = DESTPORT;    // Destination port where the server is listening
addr.sin_addr.s_addr = DESTIP;     // Server IP address where the clients connects
```

The socket() function creates an unbound socket.

```
uint_32                    sock;
sock = socket(AF_INET, SOCK_STREAM, 0);
```

The bind() function assigns a local socket address to a socket.

```
bind(sock, &addr, sizeof(addr));
```

The connect() function attempts to make a connection on a socket.

```
connect(sock, &addr, sizeof(addr));
```

The recv() function receives a message from a socket.

```
uchar cRecvBuff[25]={0};
recv(sock, (void *)cRecvBuff, sizeof(cRecvBuff), 0);
```

The send() function sends a message only when the socket is connected (including when the peer of a connectionless socket has been set via connect() function).

```
uchar cSendBuff[25]="Hello I am the client.\n";
send(sock, (void *)cSendBuff, sizeof(cSendBuff), 0);
```

2.2 Shell Interface Library

The Shell is a command-line handling code that can be used for terminal input in MQX. The Shell code was moved from the RTCS into a separate library named Shell. The MFS Shell commands are also included in the Shell library. The Shell receives an input string and compares it to a list of commands. For each command, a function is executed. The functionality of the Shell allows the developer to set up a command-line interface.

The Shell library includes the implementation of the most commonly used commands for RTCS and MFS libraries. Custom commands can also be implemented to expand the Shell functionality. The sh_rtcs.h and sh_mfs.h header files included in the directory \Freescale MQX 3.1\shell\source\include\ show a list of the Shell functions included in the Shell library.

To set the Shell interface, add the Shell libraries into the project. Also include the Shell.h header file.

```
#include <shell.h>
```

A list of commands is passed as a parameter to the Shell() function. The list of commands is implemented as an array of SHELL_COMMAND_STRUCT elements. The structure contains a string and the address of a function.

```
int_32 MyFunction(int_32 argc, char_ptr argv[]);

const SHELL_COMMAND_STRUCT Shell_commands[] = {
   { "runmyfunction",    MyFunction },
   { NULL,   NULL }
};

int_32 MyFunction(int_32 argc, char_ptr argv[])
{
printf("Executing MyFunction();\n");
}
```

The Shell_commands array defines one command named runmyfunction that executes the MyFunction() code. This function prints a string to provide visual feedback to the developer when the function executes.

Once the command list is ready, the Shell() function is executed to start the Shell. The Shell uses the ttya: driver as the default interface. Connect a serial port to the board to use the Shell interface.

```
void hello_task(uint_32 initial_data)
{
```

```
            for(;;)
            {
                Shell(Shell_commands, NULL);
                printf("Shell exited, restarting...\n");
            }
        _mqx_exit(0);
    }
```

MFS and RTCS specific functions implemented in the Shell library can be added to the command list. This allows a simple interface to the libraries. The shell.h file has macros to include the header files for the MFS and RTCS functions.

```
#ifdef __rtcs_h__
#include <sh_rtcs.h>
#endif

#ifdef __mfs_h__
#include <sh_mfs.h>
#endif
```

Some of the RTCS and MFS shell commands available in the library are:

- telnet
- telnetd
- ftp
- ftpd
- help
- ipconfig
- ping
- cd
- copy
- del
- dir
- exit
- help
- mkdir

2.2.1 Command Parameters

Commands can receive parameters that are used in their function. These parameters are passed to the function using two parameters.

- argc — An integer variable that gives the number of parameters passed to the function.
- argv — A string array with all the parameters passed to the function.

The parameters are delimited with a space when they are entered with the command. Each element of the array is a different parameter.

The following is a variation of the runmyfunction command. The command runmyfunction was modified to receive two parameters. It calls the function MyFunction(). This function receives the argc and argv arguments.

```
void MyFunction (int_32 argc, char_ptr argv[]);
```

A validation can be done to see if there are arguments from the user.

```
void MyFunction (int_32 argc, char_ptr argv[])
{
  if (argc == 0)
  {
    printf("No arguments received/n/r");
    return;
  }
}
```

The first element of the argv array is the command typed. In the case of the runmyfunction command, argv[0] = "runmyfunction". The rest of the parameters are stored in argv[1] and above.

For parameters that are numbers, it is necessary to read the array to get the parameters. The function sscanf can be used to read argv array.

```
uint_32 firstArg = 0;
uint_32 secondArg = 0;

void MyFunction (int_32 argc, char_ptr argv[])
{

  if (argc == 0)
  {
    printf("No arguments received/n/r");
    return;
  }

  sscanf(argv[1],"%d",& firstArg);
  sscanf(argv[2],"%d",& secondArg);

  printf("My first argument: %d\n", firstArg);
  printf("My second argument: %d\n", secondArg);

  return;
}
```

The string parameters can be read directly from the array.

```
if (strcmp(argv[3],"hello")==0)
{
  printf("Hello Back/n/r");
}
else
{
  printf("Hello There!!/n/r");
}
```

2.3 USB Host and MFS

2.3.1 USB Host

The USB host performs the following functions:

- Provides USB device drivers and applications with a uniform view of the I/O system
- Manages the attachment and detachment of peripherals along with their power requirements dynamically
- Determines which device driver to load for the connected device
- Assigns a unique address to the device for run-time data transfers

The code example configures the USB as a host and configures it to accept mass storage devices. The application starts with the inclusion of the MQX header files.

```
#include <mqx.h>
#include <mqx_arc.h>
#include <hostapi.h>
#include <lwevent.h>
```

- The hostapi.h file contains the function prototypes for the USB host library functions.
- The lwevent.h file contains the prototypes for the lightweight event handling functions.

A single task is needed to set the USB host and to show an example of the MFS library usage. The task list for the example code looks as follows:

```
TASK_TEMPLATE_STRUCT  MQX_template_list[] =
{
   { 1,            USB_task,      2500L,  8L, "USB",     MQX_AUTO_START_TASK},
   { 0,            0,             0,      0,  0,         0}
};
```

A device structure is defined to store the device handler and the interface descriptor handler. This structure is used in the example to handle device events like attach or detach.

```
typedef struct device_struct {
   uint_32                          STATE;
   _usb_device_instance_handle      DEV_HANDLE;
   _usb_interface_descriptor_handle INTF_HANDLE;
   CLASS_CALL_STRUCT                CLASS_INTF;
   boolean                          SUPPORTED;
} DEVICE_STRUCT,  _PTR_ DEVICE_STRUCT_PTR;

volatile DEVICE_STRUCT device = { 0 };
```

The ClassDriverInfoTable object contains the Vendor ID, Product ID, Class, Sub-Class, Protocol, and callback function for each USB device supported by the host. This structure is used to initialize the USB host to accept the list of classes included in the array.

```
/* Table of driver capabilities this application want to use */
static const USB_HOST_DRIVER_INFO ClassDriverInfoTable[] =
{
    /* Vendor ID Product ID Class Sub-Class Protocol Reserved Application call back */

    /* USB 2.0 hard drive */
```

```
        {{0x49,0x0D}, {0x00,0x30}, USB_CLASS_MASS_STORAGE, USB_SUBCLASS_MASS_SCSI,
        USB_PROTOCOL_MASS_BULK, 0, usb_host_mass_device_event},
        {{0x00,0x00}, {0x00,0x00}, 0,0,0,0, NULL}
    };
```

The USB task creates a lightweight semaphore and a lightweight event. These MQX services are used to synchronize the USB task with the USB events.

The procedure for initializing the USB host is as follows:

1. The _usb_driver_install() function installs the USB driver. After it is installed, it can be configured as a USB host.
2. The _usb_host_init() function initializes the USB host hardware and installs an ISR that services all interrupt sources on the USB host hardware.
3. The _usb_host_driver_info_register() function uses the host handler to register the information in the ClassDriverInfoTable object. This function sets the callback function to any event generated by mass storage devices in this application.

```
void USB_task(void)
{
   _usb_host_handle     host_handle;
   pointer              usb_fs_handle = NULL;
   char                 block_device_name[] = "USB:";

   _lwsem_create(&USB_Stick,0);
   _lwevent_create(&USB_Event,0);

   USB_lock();

   _usb_driver_install(0,  (pointer) &_bsp_usb_callback_table);
   _usb_host_init(0, 4, &host_handle);
   _usb_host_driver_info_register(host_handle, (pointer)ClassDriverInfoTable);

   USB_unlock();
```

The USB task then waits for an attach or a detach event. The lightweight event USB_Event puts the task in a not-ready state and allows other task execution. Once an event triggers, two different options can occur.

When an attach event is detected, the device information is used to select the host device interface. The state of the device is changed to interfaced and the file system is installed. File system details are explained in the next section.

For a detach event, the lightweight semaphore is set to wait state with the _lwsem_wait() function. The function that uninstalls the file system is called to close the file system and to be able to accept other devices.

Once the task executes the code in the loop, it clears the USB_Event. This sets the task into a not-ready state until another event is detected. The USB task only executes once for each detected event.

```
for ( ; ; )
   {
      /* Wait for events (Attach or Detach) */
      _lwevent_wait_ticks(&USB_Event,USB_EVENT,FALSE,0);
```

```
            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_handle = (pointer)&device.CLASS_INTF;

               /* Install Partition Manager and File System */
               usb_filesystem_install();
               _lwsem_post(&USB_Stick);

            }
            else if ( device.STATE==USB_DEVICE_DETACHED)
            {
               _lwsem_wait(&USB_Stick);
               /* Uninstall File System */
               usb_filesystem_uninstall();
            }

            /* Clear Event */
            _lwevent_clear(&USB_Event,USB_EVENT);
         }
      }
```

Whenever an event is detected for this class, the callback function is executed by the driver. The usb_host_mass_device_event() function receives a code number for the event causing the callback. The event number is decoded using a switch statement.

If an attach event is detected, the device handler and the interface handler are copied to the local device structure. The lightweight event is set to signal the USB task that an event has occurred.

If a detach event is detected, the device handler and interface handler are set to Null and the USB_Event is set. The callback function is used to receive the interface and device handlers, and to set an event. The rest of the code needed to open the USB device is executed in the infinite loop in the USB task.

```
      void usb_host_mass_device_event
         (
            /* [IN] pointer to device instance */
            _usb_device_instance_handle      dev_handle,

            /* [IN] pointer to interface descriptor */
            _usb_interface_descriptor_handle intf_handle,

            /* [IN] code number for event causing callback */
            uint_32           event_code
         )
      {
         switch (event_code) {
            case USB_CONFIG_EVENT:
            case USB_ATTACH_EVENT:
               if (device.STATE == USB_DEVICE_IDLE ||device.STATE == USB_DEVICE_DETACHED)
               {
                  device.DEV_HANDLE = dev_handle;
                  device.INTF_HANDLE = intf_handle;
                  device.STATE = USB_DEVICE_ATTACHED;
```

```
                    _lwevent_set(&USB_Event,USB_EVENT);
                 }
                 break;
              case USB_INTF_EVENT:
                 device.STATE = USB_DEVICE_INTERFACED;
                 break;
              case USB_DETACH_EVENT:
                 device.DEV_HANDLE = NULL;
                 device.INTF_HANDLE = NULL;
                 device.STATE = USB_DEVICE_DETACHED;
                 _lwevent_set(&USB_Event,USB_EVENT);
                 break;
              default:
                 device.STATE = USB_DEVICE_IDLE;
                 break;
           }
        }
```

The code configures the USB in host mode and enables it to detect mass storage devices. In the next section, the MFS library is used to read the contents of the device and display them in the terminal.

2.3.2 MFS

MFS provides a library of functions that is compatible with the Microsoft MS-DOS file system. The functions let an embedded application access the file system in a manner that is compatible with MS-DOS Interrupt 21 functions. All the functions guarantee mutually exclusive access to the file system.

MFS is a device driver that an application must install over a low-level device driver. MFS uses the lower-level driver to access the hardware device. Low level drivers can be a partition manager, a USB-MFS driver, or a flash disk driver.

To show the usage of the MFS, the USB mass storage driver is used. In the example code, the file system is installed and opened over the USB mass storage device.

MFS functions allow the following:

- Create and remove sub-directories
- Find files
- Create and delete files
- Open and close files
- Get the amount of free space in the file system

In the previous section, the USB interface is used as a host. To show MFS usage, the USB mass storage device is used as the low level driver where the file system is installed.

To use the MFS, follow these steps:

1. Install USB-MFS device driver.
2. Open the USB-MFS device driver.
3. Install the file system driver.
4. Open the file system driver.

Each driver requires the successful installation and setup of the previous driver. Once the process is completed, the MFS library functions can be used to read/write and handle files.

The process to close the USB device requires the inverse process to close and uninstall each of the drivers. To uninstall the USB-MFS device driver, follow these steps:

1. Close the file system driver.
2. Uninstall the file system driver.
3. Close the USB-MFS device driver.
4. Uninstall the USB-MFS device driver.

Include the header files for the MFS library and the USB-MFS routines.

```
#include <mfs.h>
#include <usbmfs.h>
```

The USB_FILESYSTEM_STRUCT holds the handlers and names for the device and the file system. The usb_fs_ptr variable is used to hold the handlers, to install, and to open each of the drivers.

```
typedef struct {
   FILE_PTR    Device_Handler;
   FILE_PTR    FileSystem_Handler;
   char_ptr    Device_Name;
   char_ptr    FileSystem_Name;
} USB_FILESYSTEM_STRUCT, * USB_FILESYSTEM_STRUCT_PTR;

USB_FILESYSTEM_STRUCT  usb_fs_ptr;
```

The USB-MFS device is installed first by calling the _io_usb_mfs_install() function. This function receives the device name "USB:" and a pointer to the interface descriptor handler of the USB device.

```
/* Install a  USB-MFS mass storage device driver. */
   _io_usb_mfs_install("USB:", 0, (pointer)USB_handle);
   usb_fs_ptr.Device_Name = "USB:";
   usb_fs_ptr.Device_Handler = fopen("USB:", (char_ptr) 0);
   printf("\n--->Install and Open USB-MFS \"%s\"",usb_fs_ptr.Device_Name);
```

Once installed, the fopen() function is used to open the driver and receive a handler. The handler is used next to install the file system.

The file system driver is installed with the _io_mfs_install() function. The device handler is used as a parameter for the MFS install function. The name of the file system "c:" is also passed as a parameter.

```
/* install MFS without partition */
   _io_mfs_install(usb_fs_ptr.Device_Handler, "c:", 0);
   usb_fs_ptr.FileSystem_Name  = "c:";
   usb_fs_ptr.FileSystem_Handler = fopen("c:", 0);
   printf("\n--->Install and Open File System \"%s\"",usb_fs_ptr.FileSystem_Name);
```

With the file system installed and opened, it is possible to access the USB stick contents for read and write operations. The code example opens the directory information and prints the files included in the media.

```
/* Read File System Information and display it */
   buffer = _mem_alloc(256);
   path_ptr = "*.*";
   mode_ptr = "m*";
   dir_ptr = _io_mfs_dir_open(usb_fs_ptr.FileSystem_Handler, path_ptr, mode_ptr );
```

```
        if (dir_ptr == NULL)
        {
           printf("File not found.\n");
        }
        else
        {
           while ((len = _io_mfs_dir_read(dir_ptr, buffer, 256))>0)
           {
              printf(buffer);
           }
           _io_mfs_dir_close(dir_ptr);
        }
     }
```

The _io_mfs_dir_open() function receives the file system handler as a parameter. The path that will be displayed and the mode used are also passed as parameters. This function opens a directory and returns a pointer to the contents of the directory. The _io_mfs_dir_read() allows printing of the names of the files using the pointer returned by the _io_mfs_dir_open() function.

This section of code prints the contents of the memory in the terminal. This is an example implementation of the ‘dir’ command available as an MFS shell command.

The directory is closed using the _io_mfs_dir_close() function with the directory pointer as a parameter.

In the case of a detach event or if the low level driver must be closed, the file system has to be closed. It is required to close the file system before closing the device.

The fclose() function with the file system handler as parameter is used to close the driver. After it is closed, it can be uninstalled using the _io_mfs_uninstall() function with the file system name passed as a parameter.

```
printf("\n--->Close and Uninstall File System \"%s\"",usb_fs_ptr.FileSystem_Name);
fclose(usb_fs_ptr.FileSystem_Handler);
_io_mfs_uninstall(usb_fs_ptr.FileSystem_Name);
```

The device is closed in the same way. Execute the fclose() function with the device handler as a parameter and then perform the _io_dev_uninstall() function with the device name as a parameter.

```
printf("\n--->Close and Uninstall Device \"%s\"",usb_fs_ptr.Device_Name);
fclose(usb_fs_ptr.Device_Handler);
_io_dev_uninstall(usb_fs_ptr.Device_Name);
```

3 How to Add Libraries to the Project

Libraries are located in the \lib\ folder in the MQX installation path:

C:\Program Files\Freescale\Freescale MQX 3.1\lib\

The library folder contains folders for each of the supported targets. The target folder includes a folder for each of the available libraries.

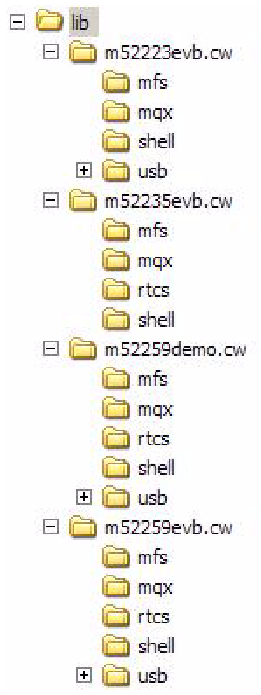

Figure 7. Library Folder

The library files are located within the library folder. Include the desired .a file in your project. Each library is available in debug or release mode. Debug versions of the library files end with _d.

If the project was created with the stationery, the libraries can be selected in the new project window. This adds the library files into the project automatically.

THIS PAGE IS INTENTIONALLY BLANK

Document Number: AN3907
Rev. 0
06/2009