Freescale MQX RTOS Example Guide

web_hvac example

This document explains the web_hvac example, what to expect from the example and a brief introduction to the API.

The example

This application represents the residential HVAC controller system requirements which are as follows:

Control

- Control of 3 outputs: Fan on/off, Heating on/off, A/C on/off
- Thermostat Input
- Serial interface to set the desired temperature and to monitor the status of the thermostat and outputs

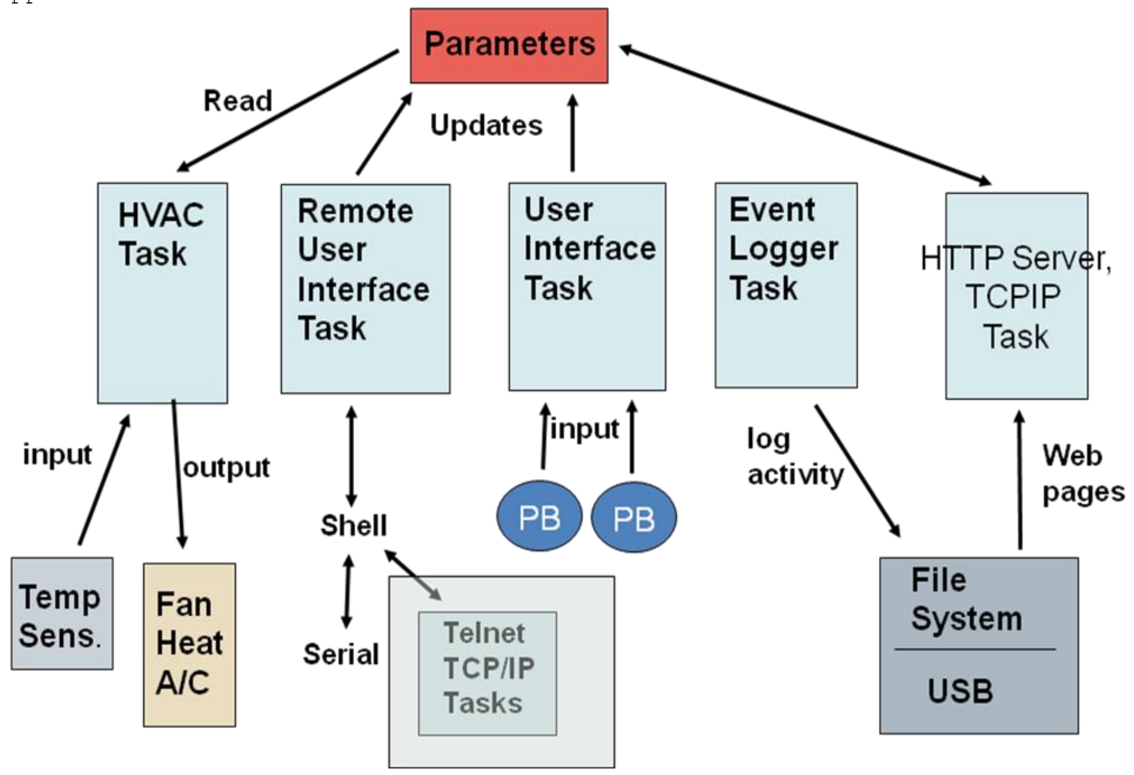
Data logging:

- Log ambient temperature and output status on a periodic basis
- Store log information on a USB Memory Stick

Ethernet:

- Provide Ethernet connectivity to facilitate monitoring and control of the device via a Telnet connection
- Support transfer of the logging information over Ethernet with FTP
- Use web pages to display status and receive setting commands

The next figure shows in detail all the parts that interacts with this demo application.



Running the example

The explanation on how to run the example is described in the MQX RTOS—Lab Tutorials document. This document can be found in the following link: http://www.freescale.com/files/soft_dev_tools/doc/support_info/MQXTUTORIALLAB.pdf?fpsp=1

Explaining the example

The Web HVAC demo application implements 6 main tasks in the MQX OS. The objective of the application is to show the user an example of the resources that MQX provides as a software platform and to show the basic interface with Ethernet and USB peripherals.

The HVAC Task simulates the behavior of a Real HVAC system that reacts to temperature changes based on the temperature and mode inputs from the system. The user interacts with the demo through the serial interface with a Shell input, through the push buttons in the hardware, and with a USB Stick that contains information with File System format.

The task template list is a list of tasks (TASK_TEMPLATE_STRUCT). It defines an initial set of tasks that can be created on the processor.
At initialization, MQX creates one instance of each task whose template defines it as an auto start task. As well, while an application is running, it can create a task present on the task template or a new one dynamically.

Tasks are declared in the MQX_template_list array as next:

```
       --web_hvac_m52259demo.mcp--Tasks.c--
const TASK_TEMPLATE_STRUCT MQX_template_list[] =
{
    /* Task Index,   Function,         Stack,  Priority, Name,      Attributes,
Param,  Time Slice */
    { HVAC_TASK,    HVAC_Task,         1400,    9,      "HVAC",
MQX_AUTO_START_TASK,    0,      0            },
#if DEMOCFG_ENABLE_SWITCH_TASK
    { SWITCH_TASK,  Switch_Task,        800,   10,      "Switch",
MQX_AUTO_START_TASK,    0,      0            },
#endif
#if DEMOCFG_ENABLE_SERIAL_SHELL
    { SHELL_TASK,   Shell_Task,        2900,   12,      "Shell",
MQX_AUTO_START_TASK,    0,      0            },
#endif
#if DEMOCFG_ENABLE_AUTO_LOGGING
    { LOGGING_TASK, Logging_task,      2500,   11,      "Logging",
0,    0,      0            },
#endif
#if DEMOCFG_ENABLE_USB_FILESYSTEM
    { USB_TASK,     USB_task,         2200L,   8L,      "USB",
MQX_AUTO_START_TASK,    0,      0            },
#endif
    { ALIVE_TASK,   HeartBeat_Task,    1500,   10,      "HeartBeat",
0,    0,      0            },
    {0}
```

Some tasks in the list are auto start tasks, so at initialization MQX creates an instance of each task. The task template entry defines the task priority, the function code entry point for the task, and the stack size.

HVAC task

This task initializes the RTCS, the Input/Output driver, and implements the HVAC simulation state machine. The HVAC Demo implementation represents the user application and shows how to use different MQX resources.

The initial part of the HVAC task installs unexpected ISRs.

Install the MQX-provided unexpected ISR, _int_unexpected_isr(), for all interrupts that do not have an application-installed ISR. The installed ISR writes the cause of the unexpected interrupt to the standard I/O stream.

```
void HVAC_Task(uint_32)
{
   HVAC_Mode_t mode;
   uint_32 counter = HVAC_LOG_CYCLE_IN_CONTROL_CYCLES;

   _int_install_unexpected_isr();
```

The MQX uses kernel log to analyze how the application operates and uses resources. Kernel log is not enabled by default in the demo.

```
#if DEMOCFG_ENABLE_KLOG && MQX_KERNEL_LOGGING && defined(DEMOCFG_KLOG_ADDR) &&
defined(DEMOCFG_KLOG_SIZE)

   /* create kernel log */
   _klog_create_at(DEMOCFG_KLOG_SIZE, 0,(pointer)DEMOCFG_KLOG_ADDR);

   /* Enable kernel logging */
   _klog_control(KLOG_ENABLED | KLOG_CONTEXT_ENABLED |
      KLOG_INTERRUPTS_ENABLED|
      KLOG_FUNCTIONS_ENABLED|KLOG_RTCS_FUNCTIONS, TRUE);

   _klog_log_function(HVAC_Task);
#endif
```

The HVAC task initializes the RTCS, the parameters of the HVAC application and loads the Input/Output driver. RTCS and Input/Output initialization is explained with detail in the RTCS section of this document. The HVAC parameter initialization function sets the initial values for the temperature, temperature scale, fan mode, and HVAC mode variables.

```
#if DEMOCFG_ENABLE_RTCS
       HVAC_initialize_networking();
#endif

       //Initialize operatiing parameters to default values
       HVAC_InitializeParameters();

       //Configure and reset outputs
       HVAC_InitializeIO();
```

The ALIVE_TASK is only a monitor that helps us to see if the system is up and running.

```
       _task_create(0, ALIVE_TASK, 0);

#if DEMOCFG_ENABLE_AUTO_LOGGING
    LogInit();
    _time_delay (2000);
    Log("HVAC Started\n");
#endif
```

The HVAC main loop executes the HVAC system simulation. Based on the mode input and the measured temperature it controls the fan. The HVAC data is stored in the HVAC_State structure.
This loop uses functions from the Input/Output driver and from the HVAC_Util.c source file. The HVAC_Util.c contains a group of functions that control all the variables required for the HVAC simulation.

```
   while (TRUE) {
      // Read current temperature
      HVAC_State.ActualTemperature = HVAC_GetAmbientTemperature();

      // Examine current parameters and set state accordingly
      HVAC_State.HVACState = HVAC_Off;
      HVAC_State.FanOn = FALSE;

      mode = HVAC_GetHVACMode();

      if (mode == HVAC_Cool || mode == HVAC_Auto)
      {
         if (HVAC_State.ActualTemperature > 
(HVAC_Params.DesiredTemperature+HVAC_TEMP_TOLERANCE))
         {
            HVAC_State.HVACState = HVAC_Cool;
            HVAC_State.FanOn = TRUE;
         }
      }

      if (mode == HVAC_Heat || mode == HVAC_Auto)
      {
         if (HVAC_State.ActualTemperature < (HVAC_Params.DesiredTemperature-
HVAC_TEMP_TOLERANCE))
         {
            HVAC_State.HVACState = HVAC_Heat;
            HVAC_State.FanOn = TRUE;
         }
      }

      if (HVAC_GetFanMode() == Fan_On) {
         HVAC_State.FanOn = TRUE;
      }

      // Set outputs to reflect new state 
      HVAC_SetOutput(HVAC_FAN_OUTPUT, HVAC_State.FanOn);
      HVAC_SetOutput(HVAC_HEAT_OUTPUT,  HVAC_State.HVACState == HVAC_Heat);
      HVAC_SetOutput(HVAC_COOL_OUTPUT,  HVAC_State.HVACState == HVAC_Cool);

      // Log Current state
      if (++counter >= HVAC_LOG_CYCLE_IN_CONTROL_CYCLES)
      {
         counter = 0;
         HVAC_LogCurrentState();

      }

      // Wait for a change in parameters, or a new cycle
      if (HVAC_WaitParameters(HVAC_CONTROL_CYCLE_IN_TICKS))
      {
        counter = HVAC_LOG_CYCLE_IN_CONTROL_CYCLES;
      }

#if DEMOCFG_ENABLE_RTCS
      ipcfg_task_poll ();
#endif
```

```
    }
}
```

The ipcfg_task_poll(); function is part of the Ethernet link/bind monitoring task. This function checks for all available Ethernet devices. Configuration for each device is checked and the link and bind status are tested.

The application interfaces with the HVAC variables by using the public variables listed in HVAC_public.h. These functions are implemented in HVAC_Util.c and HVAC_IO.c.

RTCS Initialization

```
void HVAC_initialize_networking(void)
{
    int_32                              error;
       IPCFG_IP_ADDRESS_DATA      ip_data;
#if DEMOCFG_USE_POOLS && defined(DEMOCFG_RTCS_POOL_ADDR) &&
defined(DEMOCFG_RTCS_POOL_SIZE)
    /* use external RAM for RTCS buffers */
    _RTCS_mem_pool = _mem_create_pool((pointer)DEMOCFG_RTCS_POOL_ADDR,
DEMOCFG_RTCS_POOL_SIZE);
#endif
```

Global variables from the RTCS library are initialized to configure the amount of Packet Control Block and the size of the message pool to be allocated when the RTCS is created. These values are set by default by the library and don’t require an initialization from the user.

```
       --web_hvac_m52259demo.mcp--RTCS.c--
   /* runtime RTCS configuration */
   _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;
   FTPd_buffer_size = 536;
```

With the parameters set the RTCS is created with the RTCS_create(); function. The function allocates resources that RTCS needs and creates TCP/IP Task. For more details on how the RTCS_create function works please look at the RTCS source code located in \Freescale MQX 3.x\rtcs\.

```
    error = RTCS_create();
```

The RTCS is configured by setting the MAC address, IP address, IP mask, and Gateway. These parameters are entered into the IPCFG global variables created in the library. The ip_data structure is a local object used in the bind process.

```
       IPCFG_default_enet_device = BSP_DEFAULT_ENET_DEVICE;
       IPCFG_default_ip_address = ENET_IPADDR;
       IPCFG_default_ip_mask = ENET_IPMASK;
       IPCFG_default_ip_gateway = 0;
```

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

These variables are initialized with macros defined in the HVAC.h file. There is only one Ethernet device in the target so the BSP_DEFAULT_ENET_DEVICE is set to 0. The ENET_IPADDR and ENET_IPMASK macros can be changed to modify the IP address of the device.

The ip_data structure holds the same configuration values. This structure is created as a requirement of the ipcfg_bind_staticip(); function. This structure will be passed as a parameter to execute the bind operation.

A combination of the Ethernet Device and the IP address is used to generate the MAC address. The value of IPCFG_default_enet_address is set inside the function. The ipcfg_init_device(); function uses this value to set the MAC address of the device.

The ipcfg_init_device(); function uses the ip_data structure to set the IP address of the device and to perform the bind operation.

```
    ENET_get_mac_address (IPCFG_default_enet_device, IPCFG_default_ip_address,
IPCFG_default_enet_address);
    error = ipcfg_init_device (IPCFG_default_enet_device,
IPCFG_default_enet_address);
    error = ipcfg_bind_staticip (IPCFG_default_enet_device, &ip_data);
```

The web server is implemented as an HTTPD_STRUCT. The http server requires a root directory and an index page. Web page contents in the Demo are stored in the WEB folder. These files were created using the mktfs.exe tool located in the \Freescale MQX 3.x\tools\ directory.

The external symbol tfs_data holds the web page information as an array. This array is insalled as a Trivial File System with the _io_tfs_install(); function. This allows the RTCS to access the web page data stored in the arrays in the “tfs:” partition.

If no error occurs the server initializes with the specified root_dir and with the “\\mqx.html” file as the index page. Before the server runs the server is configured with the CGI information. The cgi_lnk_tbl contains a list of the different CGI services available in the web page.

The fn_lnk_tbl contains an event that notifies the client when a USB event occurs. For the demo this changes the layout of the web page when a USB stick is connected or disconnected.

```
#if DEMOCFG_ENABLE_WEBSERVER
       {
              HTTPD_STRUCT *server;
              extern const HTTPD_CGI_LINK_STRUCT cgi_lnk_tbl[];
              extern const HTTPD_FN_LINK_STRUCT fn_lnk_tbl[];
              extern const TFS_DIR_ENTRY tfs_data[];

              if ((error = _io_tfs_install("tfs:", tfs_data))) {
                  printf("\ninstall returned: %08x\n", error);
              }

              server = httpd_server_init((HTTPD_ROOT_DIR_STRUCT*)root_dir,
"\\mqx.html");
              HTTPD_SET_PARAM_CGI_TBL(server,
(HTTPD_CGI_LINK_STRUCT*)cgi_lnk_tbl);
              HTTPD_SET_PARAM_FN_TBL(server, (HTTPD_FN_LINK_STRUCT*)fn_lnk_tbl);

              httpd_server_run(server);
       }
```

```
#endif
```

The call to function httpd_server_run() initialize the server and opens a socket at port 80 to listen for new connections and establish a HTTP connection.

The last part of the function initializes other servers if available. The FTPd_init() provides FTP server feature to the Freescale MQX.

```
#if DEMOCFG_ENABLE_FTP_SERVER
   FTPd_init("FTP_server", 7, 3000 );
#endif
```

The TELNETSRV_init(); function initializes the telnet shell.

```
const RTCS_TASK Telnetd_shell_template = {"Telnet_shell", 8, 2000,
Telnetd_shell_fn, NULL};

#if DEMOCFG_ENABLE_TELNET_SERVER
   TELNETSRV_init("Telnet_server", 7, 2000, (RTCS_TASK_PTR)
&Telnetd_shell_template );
#endif

}
```

This is the list of available Telnet commands that are passed to the new Shell task.

```
const SHELL_COMMAND_STRUCT Telnet_commands[] = {
   { "exit",      Shell_exit },
   { "fan",       Shell_fan },
   { "help",      Shell_help },
   { "hvac",      Shell_hvac },
   { "info",      Shell_info },
#if DEMOCFG_ENABLE_USB_FILESYSTEM
   { "log",       Shell_log },
#endif

#if DEMOCFG_ENABLE_RTCS
#if RTCSCFG_ENABLE_ICMP
   { "ping",      Shell_ping },
#endif
#endif

   { "scale",     Shell_scale },
   { "temp",      Shell_temp },
   { "?",         Shell_command_list },

   { NULL,        NULL }
};
```

HVAC I/O Interface
The inputs and outputs of the system are defined using macros. The macros for LED1 through LED4, SWITCH1 and SWITCH2 define the pins used as the interface of the application with the user.

Two local arrays are created: one groups the outputs and the other groups the inputs. These local arrays are the input to the fopen() function. The fopen() function returns a handler that is assigned to a global variable. The ioctl(); function performs Input/Output operations through the rest of the application.

```
boolean HVAC_InitializeIO(void)
{
      const uint_32 output_set[] = {
              LED_1 | GPIO_PIN_STATUS_0,
              LED_2 | GPIO_PIN_STATUS_0,
              LED_3 | GPIO_PIN_STATUS_0,
              LED_4 | GPIO_PIN_STATUS_0,
              GPIO_LIST_END
       };

       const uint_32 input_set[] = {
              TEMP_PLUS,
              TEMP_MINUS,
#if defined(FAN_ON_OFF)
              FAN_ON_OFF,
#endif
              GPIO_LIST_END
       };

       /* Open and set port TC as output to drive LEDs (LED10 - LED13) */
       output_port = fopen("gpio:write", (char_ptr) &output_set);

       /* Open and set port DD as input to read value from switches */
       input_port = fopen("gpio:read", (char_ptr) &input_set);

       if (output_port)
              ioctl(output_port, GPIO_IOCTL_WRITE_LOG0, NULL);

       return (input_port!=NULL) && (output_port!=NULL);
}
```

The fopen function returns a value of NULL when it fails. Port variables input_port and output_port are tested to check if there was an error when setting up the system's Input/Output. The function returns a value of 1 when both the input and output were set correctly.

The output_port file pointer passes as a parameter to the ioctl(); function to change the state of the LEDs in the application. The HVAC_SetOutput(); function receives the Output and the state that should be set to that output.

The desired state is compared to the actual state of the output. The states of the output ports are stored in the HVAC_OutputState global array. When the state requested is different to the actual state the ioctl(); function sets the output to the new state.

```
void HVAC_SetOutput(HVAC_Output_t signal,boolean state)
{
   static const uint_32 led1[] = {
       LED_1,
       GPIO_LIST_END
   };
   static const uint_32 led2[] = {
       LED_2,
       GPIO_LIST_END
   };
   static const uint_32 led3[] = {
       LED_3,
       GPIO_LIST_END
   };
   static const uint_32 led4[] = {
       LED_4,
       GPIO_LIST_END
   };

   if (HVAC_OutputState[signal] != state) {
      HVAC_OutputState[signal] = state;
      if (output_port) {
         switch (signal) {
             case HVAC_FAN_OUTPUT:
                ioctl(output_port, (state) ? GPIO_IOCTL_WRITE_LOG1 :
GPIO_IOCTL_WRITE_LOG0, (pointer) &led1);
                break;
             case HVAC_HEAT_OUTPUT:
                ioctl(output_port, (state) ? GPIO_IOCTL_WRITE_LOG1 :
GPIO_IOCTL_WRITE_LOG0, (pointer) &led2);
                break;
             case HVAC_COOL_OUTPUT:
                ioctl(output_port, (state) ? GPIO_IOCTL_WRITE_LOG1 :
GPIO_IOCTL_WRITE_LOG0, (pointer) &led3);
                break;
             case HVAC_ALIVE_OUTPUT:
                ioctl(output_port, (state) ? GPIO_IOCTL_WRITE_LOG1 :
GPIO_IOCTL_WRITE_LOG0, (pointer) &led4);
                break;
         }
      }
   }
}
```

The ioctl function takes care of any I/O control in the system. The first parameter in the ioctl(); function is a FILE_PTR which can either be the handle of a specific file, or the handle of a device driver. It varies depending on which command is used. The third parameter is a uint_32_ptr. Depending upon the I/O control command it might be a char_ptr, a pointer to a structure, or even just a NULL pointer.

For this case the ioctl function is called with the output port handler as the file pointer and the command is selected depending on the state parameter received by the function. The third parameter is the value that is modified in the function and it contains the address of the LED to be changed.

Read accesses are also performed with the ioctl(); function. A local variable is created to hold the status of the inputs. The ioctl(); function receives the input_port handle and and the GPIO_IOCTL_READ macro to perform a read operation. The third parameter is the local structure that receives the values of the input. After the execution of the ioctl(); function the local structure named data holds the values of the inputs. The HVAC_GetInput(); returns a boolean value. A switch statement converts the data structure into a boolean value.

```
boolean HVAC_GetInput(HVAC_Input_t signal)
{
   boolean  value=FALSE;
   static uint_32 data[] = {
             TEMP_PLUS,
             TEMP_MINUS,
#if defined(FAN_ON_OFF)
             FAN_ON_OFF,
#endif
             GPIO_LIST_END
       };

   if (input_port) {
      ioctl(input_port, GPIO_IOCTL_READ,  &data);
   }

   switch (signal) {
      case HVAC_TEMP_UP_INPUT:
         value = (data[0] & GPIO_PIN_STATUS)==0;
         break;

      case HVAC_TEMP_DOWN_INPUT:
         value = (data[1] & GPIO_PIN_STATUS)==0;
         break;

#if defined(FAN_ON_OFF)
      case HVAC_FAN_ON_INPUT:
         value = (data[2] & GPIO_PIN_STATUS)==0;
         break;
#endif
   }

   return value;
}
```

The HVAC_ReadAmbienTemperature simulates temperature change across time in the Demo. The _time_get(); function returns the amount of milliseconds since MQX Started. Using this RTOS service the function updates temperature every second. Depending on the state of the output temperature is increased or decreased by 1.

```
void HVAC_ReadAmbientTemperature(void)
{
   uint_32     desired_temp = HVAC_Params.DesiredTemperature;
   TIME_STRUCT time;

   _time_get(&time);
   if (time.SECONDS>=(LastUpdate.SECONDS+HVAC_TEMP_UPDATE_RATE)) {
      LastUpdate=time;
      if (HVAC_GetOutput(HVAC_HEAT_OUTPUT)) {
         AmbientTemperature += HVAC_TEMP_UPD_DELTA;
      } else if (HVAC_GetOutput(HVAC_COOL_OUTPUT)) {
         AmbientTemperature -= HVAC_TEMP_UPD_DELTA;
      }
   }
}
```

Shell task
The implementation of all shell commands has been moved out from the RTCS library into a separate “shell” library.

This task uses the shell library to set up the available commands in the HVAC demo. The Shell library provides a serial interface where the user can interact with the HVAC demo features.

```
void Shell_Task(uint_32 temp)
{
   /* Run the shell on the serial port */
   for(;;)  {
      Shell(Shell_commands, NULL);
      printf("Shell exited, restarting...\n");
   }
}
```

The shell library source code is available as a reference. To understand the execution details of the Shell function review the source code for the library located in:
\Freescale MQX 3.x\shell\build\codewarrior\

The Shell function takes an array of commands and a pointer to a file as parameters. The Shell_commands array specifies a list of commands and relates each command to a function. When a command is entered into the Shell input the corresponding function is executed.

```
typedef struct shell_command_struct  {
   char_ptr  COMMAND;
   int_32      (*SHELL_FUNC)(int_32 argc, char_ptr argv[]);
} SHELL_COMMAND_STRUCT, _PTR_ SHELL_COMMAND_PTR;
```

Each shell command includes the string that executes a command and the function executed when the command is typed.

```
const SHELL_COMMAND_STRUCT Shell_commands[] = {
#if DEMOCFG_ENABLE_USB_FILESYSTEM
   { "cd",        Shell_cd },,
   { "copy",      Shell_copy },
   { "del",       Shell_del },
   { "dir",       Shell_dir },
   { "log",       Shell_log },
   { "mkdir",     Shell_mkdir },
   { "pwd",       Shell_pwd },
   { "read",      Shell_read },
   { "ren",       Shell_rename },
   { "rmdir",     Shell_rmdir },
   { "type",      Shell_type },
   { "write",     Shell_write },
   { "scale",     Shell_scale },
   { "temp",      Shell_temp },
#endif
   { "exit",      Shell_exit },
   { "fan",       Shell_fan },
   { "help",      Shell_help },
   { "hvac",      Shell_hvac },
   { "info",      Shell_info },

#if DEMOCFG_ENABLE_RTCS
   { "netstat",   Shell_netstat },
   { "ipconfig",  Shell_ipconfig },
#if RTCSCFG_ENABLE_ICMP
   { "ping",      Shell_ping },
#endif
#endif
   { "?",         Shell_command_list },
   { NULL,        NULL }
};
```

Some of the functions executed using the Shell are provided by the MQX RTOS. For example, functions that are related to the file system are implemented within the MFS library. HVAC specific functions are implemented within HVAC_Shell_Commands.c source file.

```
extern int_32 Shell_fan(int_32 argc, char_ptr argv[] );
extern int_32 Shell_hvac(int_32 argc, char_ptr argv[] );
extern int_32 Shell_scale(int_32 argc, char_ptr argv[] );
extern int_32 Shell_temp(int_32 argc, char_ptr argv[] );
extern int_32 Shell_info(int_32 argc, char_ptr argv[] );
extern int_32 Shell_log(int_32 argc, char_ptr argv[] );
```

Functions implemented in HVAC_Shell_Commands.c are listed in the header file. These functions use the terminal to display the user how to use the Shell commands. Every function validates the input of the Shell. When commands are entered correctly a specific HVAC command is executed.

As an example, the Shell_fan function:

```
int_32  Shell_fan(int_32 argc, char_ptr argv[] )
{
   boolean           print_usage, shorthelp = FALSE;
   int_32            return_code = SHELL_EXIT_SUCCESS;
   FAN_Mode_t        fan;

   print_usage = Shell_check_help_request(argc, argv, &shorthelp );

   if (!print_usage)  {
      if (argc > 2) {
         printf("Error, invalid number of parameters\n");
         return_code = SHELL_EXIT_ERROR;
         print_usage=TRUE;
      } else {
         if (argc == 2) {
            if (strcmp(argv[1],"on")==0) {
               HVAC_SetFanMode(Fan_On);
            } else if (strcmp(argv[1],"off")==0) {
               HVAC_SetFanMode(Fan_Automatic);
            } else {
               printf("Invalid fan mode specified\n");
            }
         }

         fan  = HVAC_GetFanMode();
         printf("Fan mode is %s\n", fan == Fan_Automatic ? "Automatic" : "On");
      }
   }

   if (print_usage)  {
      if (shorthelp)  {
         printf("%s [<mode>]\n", argv[0]);
      } else  {
         printf("Usage: %s [<mode>]\n", argv[0]);
         printf("   <mode> = on or off (off = automatic mode)\n");
      }
   }
   return return_code;
}
```

The mode specified to the "fan" command in the shell input is compared to "on" and "off" strings. When the string received through the shell command is "on" the function HVAC_SetFanMode(Fan_On); is executed. When the string received through the shell command is "off" the function HVAC_SetFanMode(Fan_Automatic); is executed.
After the fan mode is set the function HVAC_GetFanMode(); reads and displays the fan mode. The usage of the function is printed as a short help message for the user if needed.

Other functions within HVAC_Shell_Commands.c execute different HVAC functionalities but the implementation is similar to the example. This file implements each custom shell command and executes the code required by the command.

Shell implementation is now provided as a single library that interfaces with MQX. The MQX related commands as well as RTCS and MFS commands can be executed from the shell. Other custom Shell implementations can be created by the user to customize the available commands and execute application specific operations.

USB task

The USB Task creates a semaphore and an event related to the USB resource. The Event indicates an USB event to the rest of the application code. For the case of the demo events are attach and detach of a USB memory stick. The semaphore notifies the availability of a valid USB stick connected to the Demo. The semaphore is enabled after the USB stick is detected as a valid USB device and after the file system installs correctly.

The ClassDriverInfoTable array contains the class information supported in the application. This array also relates the Vendor and Product ID to a specific USB class and sub-class. Callback functions for each class is also included as a part of the elements of the array. In this case any event related to the USB 2.0 hard drive executes the usb_host_mass_device_event(); function.

```
/* 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 */
   /* 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 */
   {{0x49,0x0D}, {0x00,0x30}, 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}
};
```

The usb_host_mass_device_event(); function executes when a device is attached or detached. This function tests the switch_code number that caused the callback. In the case of an attach event the structure device is filled with the USB_DEVICE_ATTACHED code and the USB_Event event is set. The USB_Event is created in the USB_Task(); function. Detach events are similar to attach events. In the case of a detach event the device structure is filled with the USB_DEVICE_DETACHED code and the USB_event is set.

```
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
   )
{
   INTERFACE_DESCRIPTOR_PTR   intf_ptr =
      (INTERFACE_DESCRIPTOR_PTR)intf_handle;

   switch (event_code) {
      case USB_CONFIG_EVENT:
         /* Drop through into attach, same processing */
      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;
            device.SUPPORTED = TRUE;
            _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 USB_task function creates a light weight semaphore named USB_Stick();. This semaphore is set by the task when a USB Stick is connected and it is available for read write operations. Light weight semaphores are created with the _lwsem_create(); function. The first parameter of the function receives the address of a semaphore. The second parameter receives the initial semaphore counter.

The Reference Manual describes the following functions to control semaphores:

_lwsem_create
_lwsem_destroy
_lwsem_poll
_lwsem_post
_lwsem_test
_lwsem_wait

An event is created to poll the USB device status within the USB_Task(); function. The event is created with the _lwevent_create(); function. The first parameter is the address of the event to be created. The second parameter receives flags to set event options. These functions are available to handle events:

_lwevent_clear
_lwevent_create
_lwevent_destroy
_lwevent_set
_lwevent_test
_lwevent_wait

```
void USB_task(uint_32 param)
{
   _usb_host_handle     host_handle;
   USB_STATUS           error;
   pointer              usb_fs_handle = NULL;

#if DEMOCFG_USE_POOLS && defined(DEMOCFG_MFS_POOL_ADDR) &&
defined(DEMOCFG_MFS_POOL_SIZE)
```

```
   _MFS_pool_id = _mem_create_pool((pointer)DEMOCFG_MFS_POOL_ADDR,
DEMOCFG_MFS_POOL_SIZE);
#endif
```

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

The USB_lock macro disables interrupts to proceed with driver installation. The task installs the USB driver with the default BSP callback table.

…

```
   USB_lock();
   _int_install_unexpected_isr();
   _usb_driver_install(0,  (pointer) &_bsp_usb_callback_table);
```

The first step required to act as a host is to initialize the stack in host mode. This allows the stack to install a host interrupt handler and initialize the necessary memory required to run the stack.

The host is now initialized and the driver is installed. The next step is to register driver information so that the specific host is configured with the information in the ClassDriverInfoTable array. The _usb_host_driver_info_register links the classes specified in the array with the callback function that each class executes on events.

```
   error = _usb_host_init(0, 4, &host_handle);
   if (error == USB_OK) {
      error = _usb_host_driver_info_register(host_handle,
(pointer)ClassDriverInfoTable);
      if (error == USB_OK) {
         error = _usb_host_register_service(host_handle,
USB_SERVICE_HOST_RESUME,NULL);
      }
   }
```

The USB_unlock(); macro enables interrupts.

```
   USB_unlock();
```

Once initialization and configuration finishes the task loop executes. The _lwevent_wait_ticks(); function waits forever until the USB_Event is set. The event is only set when attach or detach occurs. When an event occurs the device.STATE condition variable is tested.

```
      for ( ; ; ) {
         // Wait for insertion or removal event
         _lwevent_wait_ticks(&USB_Event,USB_EVENT,FALSE,0);
```

```
         if ( device.STATE== USB_DEVICE_ATTACHED) {
```

```
            if (device.SUPPORTED)  {
```

On the detection of an event the device variable information is used to select the USB interface. The usb_hostdev_select_interface(); function caused the stack to allocate memory and do the necessary preparation to start communicating with this device.

If the device installed correctly the task can install the file system for the USB Stick. File system installation is explained in the next section.

```
               error = _usb_hostdev_select_interface(device.DEV_HANDLE,
               device.INTF_HANDLE, (pointer)&device.CLASS_INTF);
               if(error == USB_OK) {
                  device.STATE = USB_DEVICE_INTERFACED;

                  USB_handle = (pointer)&device.CLASS_INTF;

                  // Install the file system
                  usb_fs_handle = usb_filesystem_install( USB_handle, "USB:",
"PM_C1:", "c:" );
```

After correct file system installation the USB_Stick semaphore is posted to indicate the other tasks that there is a USB Mass storage device available as a resource. For the case of a detach event the file system is uninstalled and the _lwsem_wait(); function disables the semaphore.

```
                     // signal the application
                     _lwsem_post(&USB_Stick);

               }
            } else {
                device.STATE = USB_DEVICE_INTERFACED;
            }
         } else if ( device.STATE==USB_DEVICE_DETACHED) {
            _lwsem_wait(&USB_Stick);
            // remove the file system
           usb_filesystem_uninstall(usb_fs_handle);
         }

         // clear the event
         _lwevent_clear(&USB_Event,USB_EVENT);
      }
   }
}
```

MFS
The partition manager device driver is designed to be installed under the MFS device driver. It lets MFS work independently of the multiple partitions on a disk. It also enforces mutually exclusive access to the disk, which means that two concurrent write operations from two different MFS devices cannot conflict. The partition manager device driver can remove partitions, as well as create new ones.
The partition manager device driver creates multiple primary partitions. It does not support extended partitions.

The function initializes MFS and allocates memory for all of the internal MFS data structures. It also reads some required drive information from the disk on which it is installed. MFS supports FAT12, FAT16, and FAT32 file systems. If the disk has a different file system or if it is unformatted, you can use MFS to format it to one of the supported file systems.

The usb_filesystem_install(); function receives the USB handler, the block device name, the partition manager name, and the file system name. Some local variables are used to execute each step of the file system installation process.

The usb_fs_ptr value is returned after the execution of the file system install process. The first step of the process allocates zeroed memory with the required size of a USB file system structure.

```
pointer usb_filesystem_install(
   pointer     usb_handle,
   char_ptr    block_device_name,
   char_ptr    partition_manager_name,
   char_ptr    file_system_name )
```

```
{
   uint_32                    partition_number;
   uchar_ptr                  dev_info;
   int_32                     error_code;
   uint_32                    mfs_status;
   USB_FILESYSTEM_STRUCT_PTR  usb_fs_ptr;


   usb_fs_ptr = _mem_alloc_system_zero(sizeof(USB_FILESYSTEM_STRUCT));
   if (usb_fs_ptr==NULL) {
      return NULL;
   }
```

The USB device is installed with the _io_usb_mfs_install(); function with the device name and the USB handle as parameters. After installation the DEV_NAME of the usb_fs_ptr variable is set to “USB:”.

```
   _io_usb_mfs_install(block_device_name, 0, (pointer)usb_handle);
   usb_fs_ptr->DEV_NAME = block_device_name;
```

A 500 milliseconds delay is generated using the _time_delay(); function. Next, the USB device is open as a mass storage device. Function fopen(); opens the USB device and the resulting handle is assigned to the DEV_FD_PTR element of the usb_fs_ptr structure. If the fopen operation failed an error message is displayed.

```
   /* Open the USB mass storage  device */
   _time_delay(500);
   usb_fs_ptr->DEV_FD_PTR = fopen(block_device_name, (char_ptr) 0);

   if (usb_fs_ptr->DEV_FD_PTR == NULL) {
      printf("\nUnable to open USB disk");
      usb_filesystem_uninstall(usb_fs_ptr);
      return NULL;
   }
```

The _io_ioctl(); function accesses the mass storage device and set it to Block Mode. When access to the device is available the vendor information, the product ID, and the Product Revision are read and printed in the console.

```
   _io_ioctl(usb_fs_ptr->DEV_FD_PTR, IO_IOCTL_SET_BLOCK_MODE, NULL);

   /* get the vendor information and display it */

printf("\n*******************************************************************
***");
   _io_ioctl(usb_fs_ptr->DEV_FD_PTR, IO_IOCTL_GET_VENDOR_INFO, &dev_info);
   printf("\nVendor Information:     %-1.8s Mass Storage Device",dev_info);
   _io_ioctl(usb_fs_ptr->DEV_FD_PTR, IO_IOCTL_GET_PRODUCT_ID, &dev_info);
   printf("\nProduct Identification: %-1.16s",dev_info);
   _io_ioctl(usb_fs_ptr->DEV_FD_PTR, IO_IOCTL_GET_PRODUCT_REV, &dev_info);
   printf("\nProduct Revision Level: %-1.4s",dev_info);

printf("\n*******************************************************************
***");
```

The partition manager device driver is installed and opened like other devices. It must also be closed and uninstalled when an application no longer needs it.

Partition Manager is installed with the _io_part_mgr_install(); function. The device number and partition manager name are passed as parameters to the function. If an error results of the partition manager installation a message is displayed in the console. On successful partition manager installation the PM_NAME element of the usb_fs_ptr structure is set to “PM_C1:”.

```
/* Try Installing a the partition manager */
   error_code = _io_part_mgr_install(usb_fs_ptr->DEV_FD_PTR,
partition_manager_name, 0);
   if (error_code != MFS_NO_ERROR) {
      printf("\nError while initializing (%s)",
MFS_Error_text((uint_32)error_code));
      usb_filesystem_uninstall(usb_fs_ptr);
      return NULL;
   }
   usb_fs_ptr->PM_NAME = partition_manager_name;
```

Fopen opens the partition manager and the resulting file pointer is assigned to the PM_FD_PTR element of the usb_fs_ptr structure. In the case of an error a message is displayed in the console and the file system is uninstalled.

A partition is opened with the fopen(); function using the handle for the partition manager.

```
   usb_fs_ptr->PM_FD_PTR = fopen(partition_manager_name, NULL);
   if (usb_fs_ptr->PM_FD_PTR == NULL) {
      error_code = ferror(usb_fs_ptr->PM_FD_PTR);
      printf("\nError while opening partition (%s)",
MFS_Error_text((uint_32)error_code));
      usb_filesystem_uninstall(usb_fs_ptr);
      return NULL;
   }

   printf("\n--->USB Mass storage device opened");

   partition_number = 1;
   error_code = _io_ioctl(usb_fs_ptr->PM_FD_PTR, IO_IOCTL_VAL_PART,
&partition_number);
```

The partition_number parameter of the _io_ioctl(); function is passed by reference. This value is modified inside the function. If an error code is returned by _io_ioctl(); the partition manager handler is closed with the fclose(); function. The partition manager uninstalls with the _io_part_mgr_uninstall(); function.

In this case the MFS is installed without partition with the _io_mfs_install(); function.

MFS is installed with the device handler pointer, a file system name and a default partition value of 0.

```
   if (error_code == PMGR_INVALID_PARTITION) {
      printf("\n--->No partition available on this device");

      /* uninitialize */
      fclose(usb_fs_ptr->PM_FD_PTR);
      usb_fs_ptr->PM_FD_PTR = NULL;

      _io_part_mgr_uninstall(usb_fs_ptr->PM_NAME);
      usb_fs_ptr->PM_NAME = NULL;

      /* install MFS without partition */
      mfs_status = _io_mfs_install(usb_fs_ptr->DEV_FD_PTR, file_system_name,
0);
   } else {
```

If the partition number is valid the MFS installs with the same handler and file system name but using the partition number as a parameter.

```
      printf("\n--->Partition Manager installed");
       /* Install MFS on the partition #1 */
      mfs_status = _io_mfs_install(usb_fs_ptr->PM_FD_PTR, file_system_name,
partition_number);

   }
```

After file system installation the status of the MFS is tested. The FS_NAME element of the usb_fs_ptr structure is set to “c:”.

```
   if (mfs_status != MFS_NO_ERROR) {
      printf("\nError initializing MFS (%s)",
MFS_Error_text((uint_32)mfs_status));
      /* uninitialize and exit */
      usb_filesystem_uninstall(usb_fs_ptr);
      return NULL;
   }
   printf("\n--->File System installed");

   usb_fs_ptr->FS_NAME  = file_system_name;
```

The fopen(); function takes the file system name as parameter. If no error occurs the file system is ready to be used by the application and the usb_fs_ptr structure is returned.

```
   usb_fs_ptr->FS_FD_PTR = fopen(file_system_name, 0);
   if (usb_fs_ptr->FS_FD_PTR==NULL) {
      usb_filesystem_uninstall(usb_fs_ptr);
      return NULL;
   }

   printf("\n--->File System opened");
   return (pointer) usb_fs_ptr;
}
```

WEB Folder
MQX includes the MKTFS.exe application that converts web page files into a source code file to be used in MQX. The tool is available in the \Freescale MQX 3.0\tools\ folder.

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

The tool is executed in the demo using a batch file. The converted output of the web_pages folder is stored in the tfs_data.c file. The file has TFS format. Information is accessed by the application through the tfs_data array.

```
const TFS_DIR_ENTRY tfs_data[]
```

IO Driver in MQX
MQX defines the complete set of ports as macros. These macros are defined in a special format that is interpreted by the fopen(); function that routes the selected pin to the register where it is addressed in memory. The following Macros are available to define any available GPIO in the device:

```
      --web_hvac_m52259evb.mcp--io_gpio_mcf5225x.h--
#define GPIO_PORT_TE ((0x0000 << 3) | GPIO_PIN_VALID)
#define GPIO_PORT_TF ((0x0001 << 3) | GPIO_PIN_VALID)
#define GPIO_PORT_TG ((0x0002 << 3) | GPIO_PIN_VALID)
```

```
#define GPIO_PORT_TH ((0x0003 << 3) | GPIO_PIN_VALID)
#define GPIO_PORT_TI ((0x0004 << 3) | GPIO_PIN_VALID)
#define GPIO_PORT_TJ ((0x0006 << 3) | GPIO_PIN_VALID)
#define GPIO_PORT_NQ ((0x0008 << 3) | GPIO_PIN_VALID)
#define GPIO_PORT_AN ((0x000A << 3) | GPIO_PIN_VALID)
#define GPIO_PORT_AS ((0x000B << 3) | GPIO_PIN_VALID)
#define GPIO_PORT_QS ((0x000C << 3) | GPIO_PIN_VALID)
#define GPIO_PORT_TA ((0x000E << 3) | GPIO_PIN_VALID)
#define GPIO_PORT_TC ((0x000F << 3) | GPIO_PIN_VALID)
#define GPIO_PORT_UA ((0x0011 << 3) | GPIO_PIN_VALID)
#define GPIO_PORT_UB ((0x0012 << 3) | GPIO_PIN_VALID)
#define GPIO_PORT_UC ((0x0013 << 3) | GPIO_PIN_VALID)
#define GPIO_PORT_DD ((0x0014 << 3) | GPIO_PIN_VALID)
```

```
      --web_hvac_m52259evb.mcp--io_gpio.h--
#define GPIO_PIN0   (0)
#define GPIO_PIN1   (1)
#define GPIO_PIN2   (2)
#define GPIO_PIN3   (3)
#define GPIO_PIN4   (4)
#define GPIO_PIN5   (5)
#define GPIO_PIN6   (6)
#define GPIO_PIN7   (7)
#define GPIO_PIN(x) (x)
```

A combination of the port and the pin number defines a Macro for the specific interface. The MCF52259DEMO has 4 LEDs in port TC.

```
      --web_hvac_m52259evb.mcp--HVAC_IO.c--
#define LED_1 (GPIO_PORT_TC | GPIO_PIN0)
#define LED_2 (GPIO_PORT_TC | GPIO_PIN1)
#define LED_3 (GPIO_PORT_TC | GPIO_PIN2)
#define LED_4 (GPIO_PORT_TC | GPIO_PIN3)
```

The GPIO driver is installed automatically by the BSP. The system is ready to perform fopen and ioctl operations since the functions are already installed as the “gpio:” device.

To use the I/O driver two arrays must be declared. One of the array declares the inputs and the other the outputs. Use the port declaration macros to list the I/O in the arrays.

```
      --web_hvac_m52259evb.mcp--HVAC_IO.c--
const uint_32 output_set[] = {
      LED_1 | GPIO_PIN_STATUS_0,
      LED_2 | GPIO_PIN_STATUS_0,
      LED_3 | GPIO_PIN_STATUS_0,
      LED_4 | GPIO_PIN_STATUS_0,
      GPIO_LIST_END
};

const uint_32 input_set[] = {
      SWITCH_1,
      SWITCH_2,
      GPIO_LIST_END
};
```

The fopen(); function can be used to open a group of pins for input or output. The first parameter passed to the function is the name of the device to be opened. For the case of the I/O Driver the name can be “gpio:write” or “gpio:read”. The “gpio:write” device name sets the array of pins to be outputs, “gpio:read” sets them to be inputs. The output_set array is passed as a parameter with the “gpio:write” device. This returns a handle to access the outputs using the ioctl(); function.

```
output_port = fopen("gpio:write", (char_ptr) &output_set);
```

```
input_port = fopen("gpio:read", (char_ptr) &input_set);
```

The ioctl(); function is used to read and write to the I/O handles. The first parameter of the ioctl(); function is the handle of the group of I/O in the control command returned by the fopen();. The second parameter could be any of the following macros:

```
#define GPIO_IOCTL_WRITE_LOG1       /* Set pins on output port */
#define GPIO_IOCTL_WRITE_LOG0       /* Clear (set to 0) pins on output port */
#define GPIO_IOCTL_READ             /* Read data from input port */
#define GPIO_IOCTL_WRITE            /* Write data to output port */
```

Write commands should only be executed on I/O handles declared as outputs and read command should only be used on I/O handles declared as inputs.

To set an output with the ioctl(); function enter the address of the pin to be set or cleared as the third parameter.

```
ioctl(output_port, (state) ? GPIO_IOCTL_WRITE_LOG1 : GPIO_IOCTL_WRITE_LOG0,
(pointer) &led2);
```

To read an input with the ioctl(); function enter the address of an array of values where the state of the input pins is returned. This is because the array variable is passes as reference and not as value.

```
ioctl(input_port, GPIO_IOCTL_READ, &data);
```