
Integrating Microchip Libraries with a Real-Time Operating System

*Author: Darren Wenn
Microchip Technology Inc.*

INTRODUCTION

As customer applications evolve into ever more complicated and feature rich designs, there is a requirement to rationalize the development process and streamline the device's software. Microchip provides a number of middleware stacks and libraries that are designed to aid customers in the creation of these advanced designs. A Real-Time Operating System (RTOS) offers an application developer a number of aids that allow a complex design to be completed in a timely fashion, permit easy integration of existing components and allow for simpler code re-use in the future.

With the demand for increased functionality and ever decreasing development times, an RTOS provides a good method of organizing and scheduling the interaction of the various libraries now being used in these advanced applications. However, the question that frequently arises is, how best can one's software be organized to take advantage of the RTOS services. Equally important is what modifications might be required to existing libraries or stacks in order that they will operate in the context of an RTOS along with other libraries.

This application note examines the reasons for migrating to a RTOS-based platform. It then discusses the various changes that may be required to one's software in order to use an RTOS. When discussing this topic, it is easier to do this in the context of a real world application, such as a home utility metering, as an example. The demonstration shows how a complex application can be built using Commercial Off-The-Shelf (COTS) hardware and software components. By using an RTOS, the workload involved in integrating multiple libraries has been significantly reduced.

BACKGROUND

Microchip provides several robust and free libraries that include:

- TCP/IP Ethernet Stack
- Zigbee[®] Protocol Stack
- MiWi[™] (and MiWi Peer-to-Peer (P2P)) Networking Protocol Stack
- USB Host and Client Stacks
- IrDA[®] Stack
- Microchip Graphics Library
- Microchip mTouch[™] Sensing Solution Library
- Memory Disk Drive (MDD)
- Audio Library

It can be observed that with such a large variety of libraries, there are innumerable ways in which an application might combine them. Therefore, it is not surprising that a general method for interconnecting them does not exist. Furthermore, since the authors of the various libraries cannot be sure that the libraries will be compiled to execute in a particular run-time environment, they have generally been written to assume that no underlying operating system exists; instead, the libraries will execute in a cooperative multitasking environment. To explain how this architecture is used, consider the basic code loop of the TCP/IP Stack V4.51 found in the file, `TCPIP_DemoApp/MainDemo.c`.

Note: Before continuing with this discussion, it is worthwhile to be clear on terminology. While the term, 'software library', can apply to any combination of code modules placed together, stack is more generally used in the context of a communications application library. Since not all of the Microchip libraries are stacks, but the integration issues are common to stacks and libraries, we will use the terms interchangeably within this document.

EXAMPLE 1: MAIN LOOP FROM TCP/IP STACK

```
// Main application entry point.
int main(void)
{
    ...
    //Initialize application
    //specific hardware
    InitializeBoard();
    ...
    //Initialize Stack and
    //application related NV
    //variables into AppConfig.
    InitAppConfig();

    //Initialize core stack layers
    StackInit();

    //Begin the co-operative
    //multitasking loop.
    while(1)
    {
        ...
        //perform basic stack functions
        StackTask();
        //perform higher level tasks
        StackApplications();

        //Process application
        //specific tasks here.
        ProcessIO();
        ApplicationTask1();
    }
}
```

The sample code in Example 1 shows the 'C' function, main, which begins by initializing the hardware on the board and then continuing to initialize the various elements of the TCP/IP Stack. Finally, it enters a while(1) loop, which performs a round-robin sequence of operations. During each pass around the main loop, the StackTask and StackApplications functions are called to ensure that the TCP/IP Stack keeps operating and services any network requests, such as a PING or web page GET.

Following the mandatory function calls to the TCP/IP Stack, users provide application-specific function calls, such as ProcessIO and ApplicationTask1. Since it is necessary to keep the stack working by periodically calling the stack functions, any user code must be written so as not to block while waiting to complete any operation. For existing code, this will often mean that it will have to be rewritten in a non-blocking fashion perhaps introducing unwanted time dependencies and extra code in the form of switch statements.

Most of the Microchip software libraries are provided with comprehensive demo applications written specifically to demonstrate a full range of features, and are thus, inherently quite large. For example, in the Graphics Object Layer Demo application, the MainDemo.c file contains over 4000 lines of code. This application is complete and showcases a full range of the graphics library capabilities, but it may not be readily apparent how users should integrate their code. Note that while simple examples make code transition and integration easier, it is inevitable that any TCP/IP or QVGA-based application will always be complex, and hence, the examples will be lengthy.

When users want to integrate their application with a COTS component, such as a Microchip library, it may be possible to redesign their code to work alongside the program structure provided by the library. Equally, for relatively simpler libraries, it may be possible to modify the library itself to fit with either the existing or planned program architecture. However, the situation becomes significantly more complicated when the application uses more than one COTS component, or the program architecture does not conveniently fit into the super-loop programming paradigm.

In these cases, a RTOS can assist with the programming effort by allowing a designer to break down the application into convenient isolated functional modules or tasks that perform parts of the application. Later, we will demonstrate how it is possible to move the software libraries into their own tasks and modify them so that they can continue to execute as if they were the only piece of software on the microcontroller. The users' new or existing code base can then be placed into its task, and RTOS services, such as "Semaphores" and "Queues", can be used to communicate between the various tasks.

Embedded RTOS

For applications based upon larger non-embedded platforms, it is possible to assume the existence of a suitable run-time environment. For instance, for devices based upon an embedded PC, it is not uncommon for the various software libraries to be written assuming that they will have access to a well understood kernel, such as Linux or Windows® CE. Such a kernel provides a rich set of features that can simplify the task of integrating multiple libraries.

However, the overhead associated with running such resource intensive kernels is not always appropriate when a cost-effective application is being designed to run on an embedded 16 or 32-bit processor core, such as the PIC24, dsPIC® DSC or PIC32 families. For the more deeply embedded applications that are commonly designed to run on Microchip microcontrollers, a more compact RTOS is required.

There are several good third party products currently available (see Table 1). For the purpose of this application note, it was decided to design the demo application using FreeRTOS. This is a popular product with a large existing user base readily able to provide support and advice to any author. Particularly for this application, it can be used and distributed without any fees or royalties associated with it so long as the standard General Public License (GNU) is complied with. Rather conveniently, this exists in a modified form in this distribution so that general publication of the product-specific code is not required (something that often deters designers from using GPL code in their own applications).

Note that while FreeRTOS has been employed, no unique functions have been used; hence, the demo application is easy to port to another host operating system.

This application note is not intended to be an authoritative article on how an RTOS works and how code should best be written to work with it. While the basic concepts of an RTOS are simple, the details can be complicated and lengthy.

Refer to the “**References**” section for more details on how an RTOS works and the vendor web sites for the specifics of each RTOS.

Note: Users should be familiar with basic RTOS concepts, such as Tasks, and the difference between cooperative and pre-emptive multitasking. We have provided some additional topics that have relevance to our application.

TABLE 1: THIRD PARTY PRODUCTS

| Vendor/RTOS | Product | MPLAB® IDE Plug-in | Microchip 16/32 Ports | Modules/Support |
|---------------|----------------------------|--------------------|-----------------------|--------------------|
| AVIX | AVIX-RT 2.2.1 | Yes | 16/32 | — |
| CMX Systems | CMX 5.30 | Yes | 8/16/32 | TCP/IP |
| Express Logic | ThreadX G5.1.5.0 | Yes | 16/32 | — |
| FreeRTOS | FreeRTOS v5.1.1 | Yes | 16/32 | — |
| Micrium | μC/OS-II v2.84 | Yes | 8/16/32 | TCP/IP |
| Pumpkin | Salvo 4 Pro and Salvo 4 LE | No | 8/16/32 | — |
| RoweBots | DSPNanoUnison | No | 1632 | DSP, TCP/IP, POSIX |
| Segger | embOS V3.52 | Yes | 16/32 | GUI, TCP/IP |

Blocking Functions

The majority of Microchip libraries and stacks are written to use non-blocking function calls for a TCP/IP Stack-based application. For example, assume that the user application is waiting for an incoming packet of data, which may arrive at any time because of the structure of the Internet. Programmers must allow for repeated function calls to the lower levels of the TCP/IP Stack to process the received packets, so users cannot simply wait, or block, for the packet to be received.

To simplify the coding problem, the libraries are typically written using state machines that can be called many times and only advance the state when the event has occurred. Any user application should be written in a similar manner. Blocking function calls must not be used as they will prevent program execution from continuing until the time elapse or an event occurs.

On the contrary, for an RTOS-based application, blocking function calls are desirable. Since any RTOS function call may cause a change of task, the code that needs to wait for a fixed period can simply use the provided delay routines and other unrelated tasks will automatically be scheduled to run. Furthermore, when the RTOS is operated in a pre-emptive mode, blocking code or time-consuming sequences will automatically be interrupted and the other tasks will be given the opportunity to run if they have the required priority. Blocking function calls can be viewed as opportunities when other tasks can be allowed to run and their introduction into an application is desirable.

Priority Inversion, Mutexes and Semaphores

A problem that often occurs when writing an application and breaking it up into appropriate tasks is priority inversion.

Consider the example of a serial-based output channel. Serial output devices are inherently slow, and since they will spend most of the time waiting for the UART to complete a transaction, it makes sense to run the serial task at a low priority. If we were to send something to be printed via the serial routine from a very high-priority task, but the routine was not yet ready, then the high-priority task would be made to wait. Depending on our system, we may need to make the low-priority serial task active in order to complete so that the high-priority task can then complete its printing function call. In this case, we have made the high-priority task wait on a low-priority one and the low-priority task has been raised to the same level as the high-priority one. This is the so-called priority inversion problem and it is best avoided by careful program design. In the demo application, a message queue was used to send the printed strings so that the high-priority task no longer has to wait for the lower priority one.

Another problem that can occur is when multiple tasks wish to access the same or related hardware peripheral.

Consider several LEDs all connected to the same peripheral port, for example, PORTA. If we were to start modifying the bits of PORTA, and in the middle of the sequence, a pre-emption occurs and another task runs that also switches an LED, then a conflict may occur and PORTA may be incorrectly set; this is the well-known read-modify-write problem.

The typical method to solve this problem is to create an atomic sequence or critical section. In other words, we somehow form a sequence of code that cannot be interrupted, such that only one task can change PORTA at a time.

The common way around the priority inversion and the non-atomic access is to create some form of mutual exclusion or mutex. An RTOS will provide constructs for the creation of mutexes so that the code sequence is protected until it has completed. Depending upon the RTOS, the mutex object may explicitly exist, or it may be created via the use of a semaphore, and in the case of FreeRTOS, there is a great deal of similarity between the two. At program start-up, a semaphore or binary flag is created, and when a task wishes to access the peripheral or function, it must be obtained by the calling task. Once it has been obtained, the peripheral can be modified or the slow functions can be called, but any other task that wants access must wait or block until the semaphore has been released.

Since the demo application uses FreeRTOS, all of the function calls are related to that kernel. Though, as we have stated, to allow the application to be easily ported, no functions that are unique to that operating system have been used. **Appendix A: "Function Calls"** lists the functions that have been used along with their parameters.

| |
|--|
| Note: To overcome the above mentioned problem, a serial output task has been considered later in this application note. |
|--|

LIBRARY AND APPLICATION MIGRATION

In the previous sections we have discussed a number of RTOS features that will aid us when porting the libraries into our new application. However, these are only tools and one of the more fundamental questions is how to modify the software.

There are several possible techniques for performing the integration; these are broken down into three options:

- First, the entire library can be rewritten to take advantage of the intrinsic multitasking provided by the RTOS. This would entail the identification of all parallel and sequential operations, and the creation of multiple tasks to handle independent sections of the code. A good example would be the TCP/IP Stack, where the main modules could be broken down with separate tasks for TCP/IP, HTTP Server and ICMP. This could also simplify some of the complex state machines present in the code with a separate task being dynamically created for each socket created.

This technique would ultimately allow for the highest performance but with a number of drawbacks. A deep understanding of the library must be obtained in order to perform the rewrite. This is, in turn, likely to introduce errors and the resulting package will need extensive testing and possible recertification. In the likely event of new versions of the library becoming available, much of the porting work would have to be performed again.

- Second, time-critical parts of the code can be identified and replaced with calls to the RTOS kernel. This might require the modification of small parts of the code separating short, high-priority code from the majority of the slow blocking code. Time delays can be replaced with API calls performing the same actions, but it may also be possible to remove the delays completely by the introduction of a blocking event, such as a 'wait for semaphore' or 'queue'. Data to be sent in and out of the library can then be sent via queues.
- Third, the unmodified code can be placed inside a task wrapper. This will require the minimum amount of modifications to the library. However, an issue arises with the resulting task and how frequently it performs its processing. If the task is given a low priority by the RTOS, then it may never complete its functions or may take too long. Equally, if given a high priority, the library may consume all of the processor time, stopping other tasks from executing. To prevent this situation, a blocking call or RTOS delay function must be introduced into the task loop so that the lower priority tasks are given the chance to execute. The delay will impair the cycle time of the affected library and so must be kept to a minimum while not degrading the performance of other software modules.

When migrating libraries into a combined application, it is desirable to keep changes in the COTS components to a minimum. If we can keep the changes to a minimum, it is then a relatively simple matter to take advantage of new library versions when they are released by just copying them over existing, unmodified versions. We have adopted the following technique for structuring the project and workspace:

- Place the COTS libraries within the main project directory with a path, such as
`C:\MainProject\Microchip\TCPIP Stack.`
- Source files specific to the project are kept within the `C:\MainProject\src` directory and any files from the libraries that need to be modified are also copied into this directory.
- Include within the MPLAB IDE project, the required files from the original library along with the modified local copies and our program source files.

This handles the project source, but many of the libraries also have associated header files, which are included using relative paths.

To overcome this, the directories that are searched by the MPLAB IDE, when including a file, have been set up so that the local include path is searched first, followed by the library default paths. This, then, provides the modified local header files, and allows the local source files to correctly locate and include library-specific header files without any errors.

The reader should examine the project path dialog in the MPLAB IDE to see the path priority settings.

DEMO APPLICATION

As mentioned in previous sections, it is difficult to develop an application that demonstrates certain capabilities without it becoming overly complex and large; as in the case of the Graphics Library demonstration application. For the purposes of this demonstration, it was desirable to target an application that would need the use of several existing Microchip libraries and would benefit from the use of an RTOS.

One area of technology that is currently attracting a lot of attention is energy metering. This is being driven by user demand and also by government legislation that is mandating the introduction of smart energy meters into people's homes. It is hoped that with the addition of meters that provide accurate and up to date information, customers will more actively regulate the amount of energy that they use.

The demand for energy meters is being driven by users' demand and also by government legislation mandating the introduction of smart energy meters into home utility sectors. With the introduction of meters that provide accurate and up-to-date information, the amount of energy used can be actively regulated.

For our example, a smart meter was designed that demonstrated some of the capabilities that would be found in one of the new generation of meters. The actual requirements may vary, but for this design, the following capabilities were chosen:

- Information to be presented on a QVGA display with touch screen.
- Separate itemized accounts for electricity and gas, detailing the number of 'units' used along with the total cost.
- Remote connection to the smart meter, via the Internet, allowing current readings to be viewed and permit forced disconnection of the energy supply. This would simulate users disconnecting the gas supply because it had accidentally been left ON in their absence.
- Remote setting of the energy per unit costs, simulating an energy provider charging the rates.
- Actual readings from the electricity and gas meters provided by an RF link from a remote sensor device.

The last requirement is often needed in residential installations where the energy supply enters the building at some remote location and must be measured at that point. However, for the convenience of the users, the display must be located within the living space of the property, for example, the kitchen.

Demonstrator Hardware

The following components are used to develop this demonstration application:

- All of the components should be available off-the-shelf.
- Explorer 16 (DM240001) Development Board and PIC24F Plug-in Module (PIM) that is supplied with the board installed in the processor socket.

To demonstrate the capability of easily scaling the application, the software was designed to also run on the PIC32 processor using the separately available PIC32 PIM (MA320001).

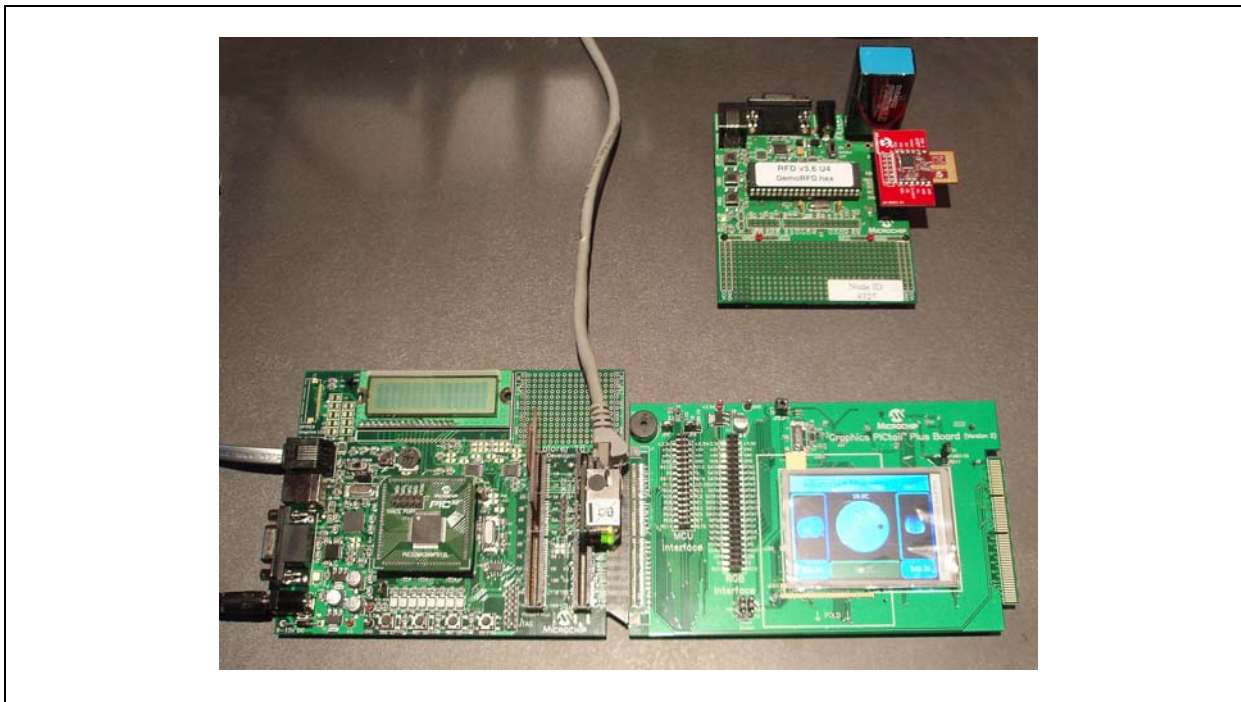
- The Graphics PICtail™ Plus Daughter Board (AC164127) to provide the QVGA graphics display.
- The hardware connection to the Internet was made with the Ethernet PICtail™ Plus Daughter Board (AC164123).
- For the wireless connection to the remote board, the MRF24J40MA PICtail Plus 2.4 GHz RF Card (AC164134) provides the radio link.
- The remote end is built using the standard demonstration application on a PICDEM™ Z MRF24J40MA 2.4 GHz Demo Kit (DM163027-5).

The source code for this PIC18F-based device is provided in the `\PIC18 MiWi Meter` directory.

For hardware-specific reasons, the MRF24J40 is located in the upper part of the PICtail Plus slot on the Explorer 16, and the Ethernet PICtail Plus is placed into the lower part of the second slot. If the second connector is not populated, then it is possible to separately obtain the connector and solder it onto the board (part number CON0197).

The baseline project compiles and executes on the PIC24FJ128GA010 supplied with the Explorer 16; consequently, some small concessions to the user interface are made because of the 128k Flash limit on that part. On the contrary, the PIC32MX360F512L has more memory and increased processor capacity; as a result, the PIC32 project features enhanced QVGA buttons with bit-mapped images. This kind of trade-off is typical of the decisions that are made when designing such a product. The PIC32 PIM (MA320001) is available separately and can be simply installed in the Explorer 16. The final hardware configuration can be seen in Figure 1.

The demo application makes extensive use of the Microchip Graphics Library for user interaction. In this case, the physical connection between the processor and the display is via the Parallel Master Port (PMP) peripheral. The MPLAB IDE projects and workspace have been provided for the PIC24F and PIC32F processors; however, the software could also easily be run on a PIC24H or dsPIC device with a suitable PMP peripheral.

FIGURE 1: DEMONSTRATOR HARDWARE

Note that, with a few exceptions, the program source files are common to all of the projects regardless of the processor family, thus making demonstration an easy migration between 16 and 32-bit processors. Because of the use of an RTOS, the different software tasks are effectively isolated. As a result, even if the MRF24J40 PICtail Plus is missing, the demonstration can still be built and tested. This is useful if the user wants to test the application but does not want to purchase all of the hardware.

IMPLEMENTATION

This section discusses the specific changes made to each of the stacks and how they have been modified to create the demo application. Note that the modifications have been done in such a way that the changes can be readily recreated when new libraries are released. It would be possible to use the modified code in another application by taking the individual task files and libraries that they use. The mechanisms for inter-task communication can then be modified or recreated to suit the particular target application. Table B-1 lists all of the RTOS related objects used in the demonstration application.

For each major task in the code, its operation, along with how any change to the associated COTS components were made, is described. For some components, there could be interactions between the various elements and other unrelated tasks due to their use of common library calls or functions within the Microchip stacks. An example of this is with the TCP/IP and MiWi Networking Protocol Stacks, which require a TICK timer routine to manage the delays associated with sending data and also to provide time stamps to messages. These common functions and the changes to those functions are explained where most appropriate.

One of the primary objectives in this application is to minimize the amount of modifications to the COTS components used in the project. To this end, few changes have been made to the core functionality of each library and only the highest levels have been modified to allow their use in a RTOS environment. Because of this, the basic documentation on the operation of each library is still applicable and the reader is directed to the relevant library application note to obtain an understanding of how they work.

FreeRTOS Modifications

The original FreeRTOS installation allocates Timer1 as the default RTOS timer. This is set to periodically interrupt the processor and allow the RTOS to perform a pre-emptive context switch to another task if required. Timer1 is also used by default by many of the Microchip libraries and because of this, it is easier to initially re-assign the RTOS TICK timer to Timer5. Another reason for this change is that the MiWi P2P networking protocol makes extensive use of Timer1, and often manipulates the interrupt flag and enable bits directly, so it is simpler to modify the RTOS source.

This section lists the five timers and the way they are used by the various software elements.

- TMR1 – Used exclusively by the MiWi P2P Networking Protocol Stack.
- TMR2 and TMR3 – Form a 32-bit timer used primarily for the TICK functions. TMR2 is also used by the MiWi Networking Protocol Stack to generate packet sequence numbers.
- TMR4 – Used to sequence the ADC sampling by the touch screen routines.
- TMR5 – FreeRTOS periodic TICK timer.

The licensing conditions of the FreeRTOS are that any changes made to the real-time kernel, along with the kernel source, must be made available to everyone. However, the modified GPL means that any project code that merely calls RTOS functions does not need to be released. This is convenient if we want to produce a commercial product and do not want to reveal the source code of our product.

The following components included in the demo application are discussed in the following sections.

UART

Many of the Microchip libraries assume that a UART is available for the output of diagnostic information or for configuration. It is possible to configure the TCP/IP Stack parameters using the serial port and a terminal window. Equally, when designing any system, it is useful to have some form of simple diagnostic output so that the operation can be confirmed, and this is particularly true for systems that use an RTOS. To support this, a simple UART handling task is created, which could accept messages for output via a queue, and then print each character in the output string to the UART peripheral.

The serial output functions can be found in the `taskUART.c` file. The UART code performs the following:

1. An initialization function is used, which creates both the queues and the UART task itself.
2. The task blocks waiting on data to be received on the transmit queue.
3. Once a message is received, it is placed one character at a time into the UART peripheral transmit buffer.

The task has been given a low priority as it is only outputting diagnostic information to a serial peripheral, which is inherently slow.

4. Incoming data is handled by an Interrupt Service Routine (ISR).
5. Each received character is placed onto a receive queue and tasks that require the incoming data can retrieve it.

This mechanism could be improved upon; however, it is sufficient at present as no tasks in the current application use incoming serial data.
6. A generic `UARTprint` function is provided, which takes a character array as an argument.
7. It then constructs a message of the correct type and places the data on the transmit queue.

A problem might arise with the `UARTprint` function being called by multiple tasks simultaneously. Consider a sample scenario where several low-priority tasks call the function and place a number of messages on the transmit queue. Since the UART, itself, is slow (running at only 19200 Baud) this may take some time to print a string. If another high-priority task then attempts to print some data, it may find the transmit queue full. We could block until space is available on the queue; however, the result is that a high-priority task (the sender) is waiting for a low-priority task (the UART) to complete.

This forms a priority inversion, which is undesirable in any application.

In the case of the metering application, only diagnostic information is output, so we have adopted the simpler, if less rigorous, technique of allowing a zero wait time when attempting to write to the queue. So, if the queue is full when the high-priority task attempts to write data, it will fail and simply not print the message. It is generally not advisable to have high-priority tasks waiting on lower priority ones, and even though RTOS constructs, such as mutexes can help ease the problem, it is still a design issue. As a result, it is often difficult to output diagnostic information from tasks of different priorities within an application. In the case of operating systems like Linux, specialized routines, such as `printk` (print from kernel), are provided to help log data during kernel operations. Their use can slow down the system and so should be minimized.

Meter

A `meterTask` is written to control the data related to the main task of metering.

At initialization, the meter readings are zeroed and the starting values for the unit costs are set up. There are two types of event that can affect or access the meter related data:

- Updates can be received from the MiWi Networking Protocol Stack that cause the total number of units used to be incremented.
- Changes to the meter billing rates and control actions can be received from both the Graphical User Interface (GUI) and the TCP/IP connection, and these other tasks may also require updating with the changing meter values.

For both types of updates, the meter task defines that update messages are only received via the `hMETERQueue`. While values may be read from the `gMeter` object, at any time, the incoming queue is the only method by which changes to the data are allowed. This allows the task to block waiting for updates, and hence, consumes no processor cycles when nothing is changing.

The second type of update requires that the totals be modified, taking into account the current unit cost. However, to complicate matters, there are various tasks that access this data using different methods. We use the term, asynchronous tasks, as those tasks that access the data from the meter task when they require it.

For example, the HTTP Server, which in response to a web page update request (HTTP GET), reads the current values from the `gMeter` global data structure, formats it into hypertext and returns it to the user via a web browser. These asynchronous tasks require no further action to be taken by the meter task since they will obtain the new values when they are required. Conversely, there are tasks that need to be informed when the `gMeter` data structure has been updated, and these are termed synchronous tasks. In any generic application, this style of interaction is desirable because it allows tasks that are dependent upon the data to block until a change occurs, and hence, not consume any processor cycles directly polling for changes.

Because of the need for these updates, changes to the energy values are sent via queues to the synchronous tasks; in our application, this is limited to the QVGA display task.

Since the meter data is stored in a global data structure, called `gMeter`, which can be accessed from multiple tasks, we must also ensure the consistency of data contained within it. The access is regulated by a semaphore (called `METERSemaphore`), and only when a task owns the semaphore, should it read or modify the meter data.

An advantage with synchronous events is that since the meter task is responsible for sending the update message, it can also propagate the new value along with the message, which simplifies some of the mutual exclusion problems inherent when using a semaphore. Note that while the meter task is the only part of the program that updates the metering data, it also obtains the semaphore before modifying the values. This is done to ensure that any change occurs as a single atomic sequence and the coherency of the data in `gMeter` is ensured.

MiWi P2P Networking Protocol

The Microchip MiWi P2P Networking Protocol Stack is designed for the interconnection of small, embedded devices in a simple network structure. This application note uses the first public release, V 0.1, of the stack. Since this is a simple stack, all of the main functions are contained within one file called, `p2p.c`. Further examinations of this file revealed that there were numerous calls to `printf` for diagnostic output purposes. Unfortunately, this makes a large amount of the stack incompatible with our multitasking application design.

For ease of development, a local copy of the `p2p.c` file is created in the project source tree, which can then be modified without affecting the original file.

As previously noted, the console output functions could cause hardware access issues. It would have been possible to redirect this output to the new UART output task. However, for simplicity, the existing functions are removed. A conventional P2P application keeps the stack operating by periodically calling the `P2PTasks` function (which is indirectly called by the `ReceivedPacket` function). This returns `TRUE` if the data has been received, and also ensures that packets are correctly sent and Acknowledgements processed.

To keep the modifications to the MiWi Networking Protocol Stack to a minimum, the majority of the code is left unaltered and blocking functions are directly introduced into the stack itself. The MiWi networking protocol task, itself, is responsible for initializing the stack, and connecting to the PIC18F transmitting node, after which it enters a `while` loop, which performs the main processing. The task is given a low-priority level (just above that of the Idle task) but to prevent it from starving out other low-priority tasks and the Idle task itself, it is made to execute only periodically using the `vTaskDelay` function, which halts its execution for 20 ms. This kind of approach is justified in the case of this stack, since the data is sent at a very low rate, and so a 16 or 32-bit processor can easily keep up with the message updates when executed every 20 ms.

The received messages are of two types:

- Electricity Unit Updates
- Gas Unit Updates

The start of each message contains either an E or G character to distinguish the type and the remaining data contains the ASCII meter reading values. In a real application, other data would also be sent, such as temperature, and any practical application would probably use some form of security or encryption. The received data is then passed to the meter task using the `hMETERQueue`.

One area, which can cause conflict between the various stacks, is in their use of common functions. While, in an ideal situation, there would be no interaction between the various modules, programmers often write code that replicates functions found in other programs but with slightly different behavior. Structured programming techniques and code re-use can mitigate these overlaps but they still occur. In the case of the MiWi Networking Protocol Stack, the `SymbolTime.c` file provides support for obtaining a high-resolution, 24-bit timer value. Within the code, this data is expressed using the custom TICK data type.

However, this same data type name is used within the TCP/IP Stack and many of the functions for accessing its elements have the same names in the MiWi Networking Protocol and TCP/IP Stacks. This would not normally pose a problem; however, in the TCP/IP Stack, the TICK data type is 32 bits in length and so poses a conflict. This is overcome by creating a `Tick.c` file that replicates the functionality of the TCP/IP version and provides a MiWi networking protocol compatible access function to the enlarged timer (`MiWiTickGet`). Since the functions within the file are identically named to the TCP/IP ones, no changes are required in that stack and only minor changes are required to the MiWi P2P Networking Protocol Stack to use the newly provided function, and these alterations are all isolated within the single `P2P.c` file.

QVGA and Touch Screen Interface

The Microchip Graphical Software Library allows software engineers and designers to rapidly develop customized user interfaces on small embedded micro-controllers. The library consists of a hierarchical software structure that allows programmers to pick the most suitable functionality for their design. At the most basic level, it provides a driver interface to many commercially available LCD controllers, or LCD panels, with integrated controllers (so called intelligent glass). Above this driver layer, a drawing primitives API allows simple shapes, such as points, lines and circles, to be drawn, which displays fonts and bitmaps. Finally, above this layer comes the Graphics Object Layer (GOL), which contains functions for drawing and managing entities, such as buttons, sliders, text boxes, dials and so on. For user input, the library provides routines and drivers for handling both keyboards (or buttons) and resistive touch screens.

When a program uses the higher level items, such as buttons, they are created dynamically using the standard `malloc/free` calls and adding them to an active display list. The GOL is then responsible for calling drawing primitives for each of the widgets that are currently in the display list, forcing them to update and reflect their current state. When a button press is received, or as in our case, a press of the touch screen occurs, the graphics library passes the message around all of the objects in the active display list, allowing each of them to respond if the press occurred in their screen area. Example 2 provides the Graphics Object Layer loop.

EXAMPLE 2: GRAPHICS OBJECT LAYER MAIN LOOP

```
...
...
while(1)
{
    // Draw GOL objects
    If (GOLDraw())
    {
        // Get message from touch screen
        TouchGetMsg(&msg);
        // Process message
        GOLMsg(&msg);
        ...
    }
}
```

In the demonstration project provided with the Graphics Library (called Graphics Object Layer Demo), the `MainDemo.c` file contains a `while(1)` loop; this is depicted in Figure 1. When `GOLDDraw` is called, the library iterates through all of the objects in the active display list giving them the opportunity to update and redraw themselves if required. The touch screen is then checked for any activity and this information is passed on to the message handling functions for the objects in the display list. It can be seen that there are no delays in this processing algorithm, and the screen redraw, touch screen and message processing functions, will be called as rapidly as the processor will allow.

When any form of user interaction is required, it is desirable to offer fast cycle times so that the user interface feels responsive. This polling mechanism is suitable for a dedicated demonstration where this is the only application that is executing. However, for a multi-tasking system, it has the disadvantage of consuming excessive processor cycles repeatedly drawing and analyzing the touch screen, when in fact, in most typical applications, the display will actually be dormant and no user interaction is required. Generalizing this further, it can be seen that once the screen has been drawn, it is unlikely that any further updates will be required unless either the touch screen is pressed or a meter update occurs, hence, mandating a redraw.

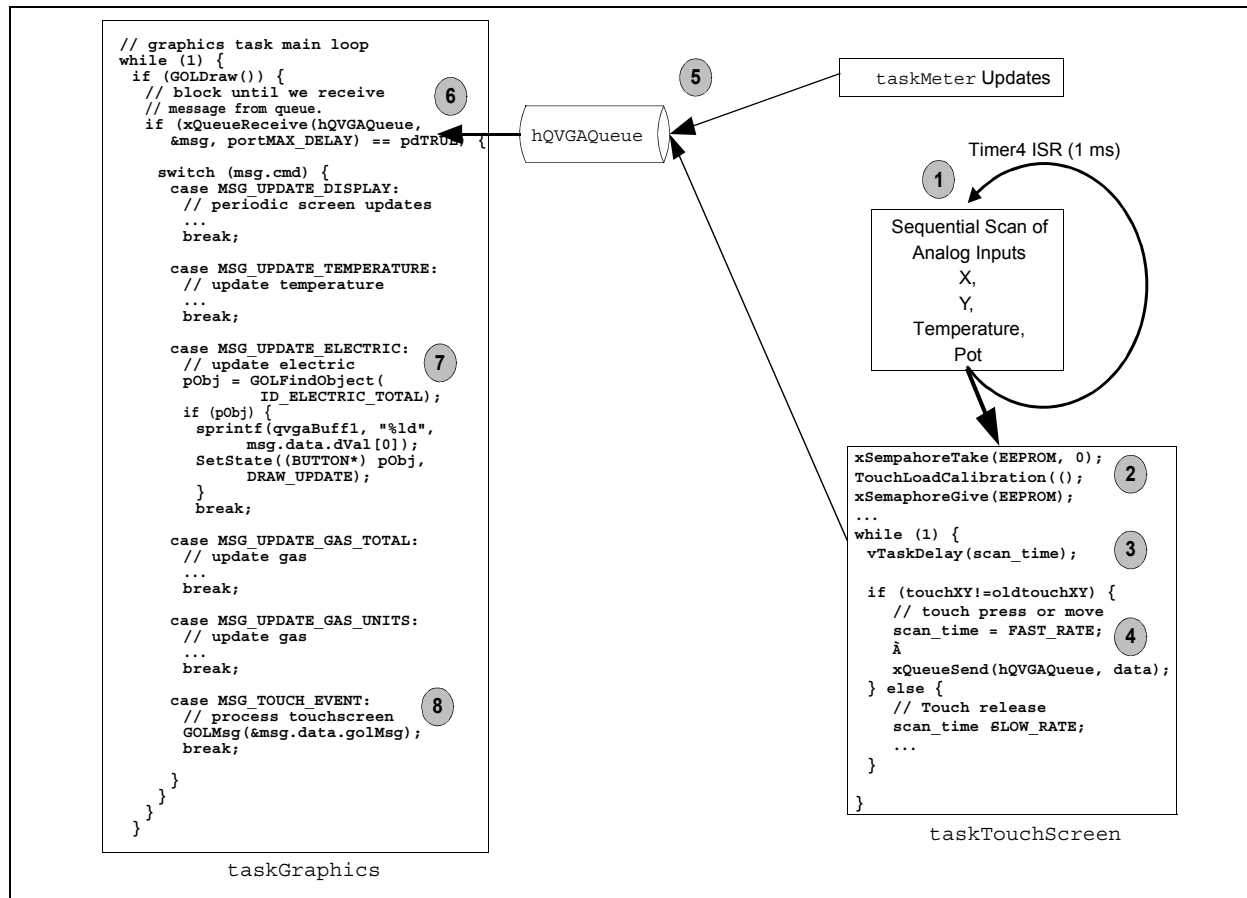
For this demo application, the functionality is separate from the GOL library and the touch screen. By making the touch screen operate in a separate task it can be simplified and made to operate at a high priority, allowing for good responsiveness. At the same time, the graphics task can be made lower priority as it only updates the displayed items when it has passed a touch screen press or a data update. This means that updates cause the display to redraw immediately but at all other times, it blocks, hence consuming no processor cycles which give the illusion of high speed.

It is worthwhile examining in some detail how our new improved 'broken apart' graphics library now works. The preceding paragraphs have explained the major changes to this library, but it is useful to see these changes in context with a diagram.

Some of the techniques used are often found in RTOS-based programs and a programmer can use them in their own applications. Figure 2 provides the pseudocode for the graphics and touch screen tasks. It contains two main blocks showing the basic program flow. The related code can be found in the two files: `taskTouchScreen.c` and `taskGraphics.c`. Here we consider the various numbered stages:

1. The Timer4 Interrupt Service Routine (ISR) is triggered every 1 ms. The touch screen requires that a bias voltage be applied to the X or Y axis while the reading is being taken. To help this sequencing, the ISR implements a state machine that iterates through the various analog channels and performs the required readings. Once a reading has been taken, it is placed into volatile variables for access by the `taskTouchScreen` function.
2. At the start of `taskTouchScreen`, check if the screen needs calibrating. As the EEPROM on the Explorer 16 board is accessed by the touch screen during initialization and the TCP/IP Stack when displaying web pages, a simple semaphore is used to control access. The touch screen task obtains the `EEPROMSemaphore` and reads the calibration data. If it detects that calibration is required, then it also obtains control of the QVGA display by obtaining `QVGASemaphore` and directly writing to the screen using drawing primitives. While the TCP/IP and QVGA tasks also access the EEPROM and QVGA display, because the `taskTouchScreen` is higher priority, it will always be able to obtain the semaphores and carry out its calibration at program start-up.
3. The main touch screen routine operates in a `while(1)` loop. At the start of the loop, the task blocks for a programmable amount of time. Initially, this block time is set to 100 ms. Once the block time has elapsed, the current value of the touch screen ADC conversions are compared to the previous versions. Only when these do not match does the task perform any processing.
4. Once a touch is detected, the task switches to a fast scanning rate, where the main loop block time is reduced to 10 ms. The inputs are analyzed, and if the screen has been pressed, a `MSG_TOUCH_EVENT` message will be sent to `hQVGAQueue`. The type of event depends on the current action and could be either a pressed, released or a move event.
5. All of the data from `taskTouchScreen` is sent to the display using `hQVGAQueue`. This queue is used by other tasks, such as `taskMeter`, to send updated meter readings asynchronously to the display. This mechanism of sending data on queues, but prefixing the data with a command type message (such as `MSG_TOUCH_EVENT` or `MSG_UPDATE_GAS`), is a very common mechanism within a RTOS application and allows efficient use of the message passing mechanisms. Depending on the RTOS, it may be possible to wait for data to be received simultaneously on multiple queues; however, if this facility does not exist, then the command and data method used here works efficiently.

FIGURE 2: RTOS VERSION OF GRAPHICS AND TOUCH SCREEN



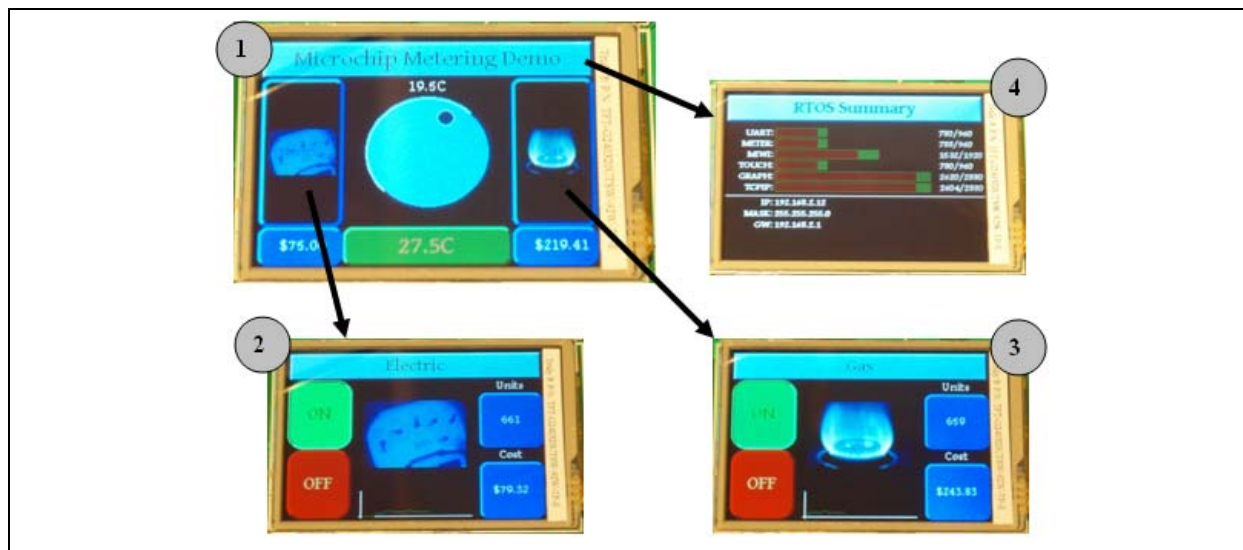
6. In the `taskGraphics` routine, the main drawing is again performed in a `while(1)` loop. Once the drawing is completed, the task blocks waiting for data to arrive on its `hQVGAQueue`. As a `portMAX_DELAY` parameter is used, the task will effectively Sleep until new data arrives or the screen is pressed.
7. After the arrival of the message, the type is checked and various actions occur based upon the command. The statement shown next to the label is for the arrival of a `MSG_UPDATE_ELECTRIC` message, which is used to indicate that a new electricity unit reading has arrived. The control associated with `ID_ELECTRIC_TOTAL` is obtained, and provided it exists (we may not currently be displaying a screen with this control on it), a new value to display is created and the button is marked as needing an update.
8. Finally, at the end of the main loop, the general touch screen handling is performed when `MSG_TOUCH_EVENT` arrives. Here, the message is translated by the standard `GOLMsg` function, which will call the control handling functions later in the same file.

The graphics library is used to display a range of screens in the home metering application, as depicted in Figure 3. The summary of the functionality is:

- Figure 3 provides a summary of the meter information along with a dial for room temperature set-point and a real-time temperature display. On the left of the screen is a display of the current electricity cost, and the button will take the user to screen 2. On the right is the gas usage summary and the button will reveal screen 3. Finally, the upper title bar can be pressed to display screen 4.
- The electricity usage screen displays the total number of units consumed along with the total cost based upon the current unit cost. On/Off buttons disable the logging and a graph at the bottom shows a history. A press of the title bar will return to the main screen.
- The gas screen operates in a similar way to the electricity usage screen.
- Finally, the RTOS summary screen displays information on the main tasks in the application. For each task a bar graph shows the maximum stack size allocated to that task along with the current position of the stack high water mark; a text equivalent is also provided. This information could be used to optimize the amount of stack allocated to each task. At the bottom of the screen the current DNS settings for the TCP/IP Stack are shown.

Figure 3 depicts the Microchip Metering Demo, Electric, Gas and RTOS summary graphics.

FIGURE 3: GRAPHIC SCREENS



TCP/IP

The demon application has been built upon version 4.51 of the Microchip TCP/IP Stack. The stack is designed to offer a range of Internet protocols and services that complement the embedded silicon solutions found in the ENC28J60 Ethernet transceiver and newer products. The TCP/IP Stack consists of many software modules that can be selectively compiled into the end user application depending upon the program requirements. The provided protocols include, ARP, IP, ICMP, UDP, TCP, DHCP, SNMP, SMTP, SNT, HTTP, FTP, TFTP and application support for a web and telnet server. With such a range of supported protocols and application layers, the TCP/IP Stack is naturally quite large and complex. Just like the graphics library, modifications to the stack are involved and it is best if it can be treated as an integral whole.

The standard method for using the stack with one's own application is to modify the `MainDemo.c` file, and integrate a project-specific `ProcessIO` handler along with the appropriate handlers for any web-based events. For more information, refer to the library help files and also AN833, "The Microchip TCP/IP Stack" and AN1120, "Ethernet Theory of Operation". These are the demo metering application specifications.

Considering our specification for the metering application, we needed to implement a simple HTTP server capable of displaying web pages from either the on-board EEPROM on the Explorer 16 or with web pages stored directly in program memory on the PIC32 Starter Kit. To make network operation as simple as possible, the stack was configured to use Dynamic Host Configuration Protocol (DHCP), and it was assumed that the device would always be used in a network with a suitable DHCP server.

Based upon these requirements the following configuration options were selected in the `TCPIPconfig.h` file:

- `STACK_USE_IP_GLEANING` – Obtains an IP address by pinging for an unused address.
- `STACK_USE_UART` – Allows diagnostic printing via the UART.
- `STACK_USE_ICMP_SERVER` – Responds to ping requests.
- `STACK_USE_ICMP_CLIENT` – Sends ping requests.
- `STACK_USE_HTTP2_SERVER` – New web server.
- `STACK_USE_DHCP_CLIENT` – Automatically configures the IP address.
- `STACK_USE_DNS` – Resolves addresses via the domain name server.
- `STACK_USE_NBNS` – NetBIOS names server support.
- `STACK_USE_MPFS2` – New style file system for web pages.
- `STACK_USE_EEPROM` – Support for MPFS files in EEPROM.

To observe the changes to the TCP/IP Stack, the reader should examine the original `TCPIP_MainDemo.c` file and compare it with the new `taskTCPIP.c` file shown in Example 3. The use of an RTOS has resulted in a main network processing task (`taskTCPIP`) that has equivalent functionality to the original sample without requiring time consuming rewriting for the RTOS-based system. At the start of the task, a semaphore controlling access to the EEPROM is obtained. This is not only necessary, since the EEPROM is used by the HTTP server to store web pages and network addresses, but is also used by the graphics library to store screen calibration information.

- Once the graphics library has completed initialization, it will release the semaphore and the TCP/IP Stack can then continue.
- The task then performs initialization of the filing system, application configuration structures and the core stack layers.

EXAMPLE 3: `taskTCPIP.c` MAIN TASK

```
void taskTCPIP(void* pvParameter)
{
    //obtain the semaphore
    //to access the SPI EEPROM
    xSemaphoreTake(EEPROMSemaphore,
        portMAX_DELAY);

    //Initialize the MPFS2
    MPFSInit();
    //initialize the configuration
    InitAppConfig();
    //Initialize the stack
    StackInit();
    while (1)
    {
        vTaskDelay(50 / portTICK_RATE_MS);
        //perform stack tasks
        StackTask();

        //call the stack
        //related applications
        StackApplications();
    }
}
```

- Following initialization, the main task loop is entered.

Here, we have decided to implement a simple blocking scheme by using a `vTaskDelay` function call of 50 ms. In order to enhance throughput, it would have been more beneficial to separate the functionality of the TCP/IP Stack into various sub-tasks for each block pending upon new data arriving from the various TCP/IP modules. However, this is quite a challenging modification to make for this library given its complex nature and the various interactions between the protocols.

- It was decided to minimize the changes to the library by implementing the top level task with a simple timed blocking call, and given the nature of the web pages and the network data rates, this provides for adequate data rates in the meter application.
- Following the delay are calls to the `StackTask` function, which ensures operation of the lower protocol layers and the `StackApplications` function, which in turn, runs the high-level applications, such as the HTTP Server.

To prevent hardware conflicts, we must ensure that the EEPROM and Ethernet PICtail Plus are not accessed at the same time; otherwise, their serial data out lines could interfere. To prevent this, replacement files, `ENC28J60.c` and `SPIEEPROM.c`, are created in the source (`src`) directory. These two files replicate the behavior of the standard files provided with the TCP/IP Stack, except that they have been modified to include an additional semaphore (`SPI2Semaphore`) to control access to SPI Channel 2. With the use of the current release of the TCP/IP Stack in this application, it is unlikely that simultaneous accesses would occur. However, the use of a semaphore provides an additional level of security and a degree of future proofing to the program. These replacement files further demonstrate the useful technique of keeping the main stack and libraries intact, and just modifying a few individual files.

The standard TCP/IP demo application comes with a set of web pages that demonstrate various methods of interacting with an embedded HTTP server. These pages require more application-specific modifications for the metering application. Since all of the web pages must fit inside the 32k x 8 EEPROM on the Explorer 16 board (or inside the program Flash of the PIC32 Starter Kit-based version), they must be compact. At the top level of the source tree, a Microsoft® Visual Studio® development system-based project, `WebPages.sln`, has been provided which contains all of the web related material. Either this tool, or a simple text editor, can be used to customize the pages if required.

These pages demonstrate a range of features permitting remote users to view a summary of the application. Real-time updates on the actual energy used are provided using AJAX techniques that asynchronously query the data from the embedded device. A password protected page allows the unit costs for gas and electricity to be set remotely in a similar manner to a real world system.

Once the pages are created, they must be converted into a format suitable for use in the application. This is done with the `MPFS2.exe` utility supplied with the TCP/IP Stack. The `WebPages` directory is selected as a source and is converted into a file suitable for download into the EEPROM. When the application is built for the PIC32 Starter Kit running on the I/O Expansion Board (DM320002), no EEPROM is present so the pages must be stored in the internal program Flash.

As part of the MPFS conversion sequence, several support files are automatically generated. In the event that a change of platform results in any of these files changing, then the code will need recompilation. For reference purposes, original copies for the Explorer 16 and I/O Expansion Board systems are provided in the `src\Explorer16` and `Starter Kit Web Files` directory.

Another interesting feature is a provision of a dynamic monitoring page, customized for display on a hand-held PDA. This has its screen layout modified in order to correctly display on a 240 x 320 pixel PDA using Internet Explorer® Mobile. The page can be accessed using the `\pda.htm` URL.

One final area in the TCP/IP Stack that required modification was its accesses to the ADC. While in normal operation, it performs no access to the analog data; at initialization, it takes control of the ADC in order to generate a pseudo random number used for packet sequencing. Because of the task priorities that were assigned at design time, this initialization happens after the touch screen task has been started and the result is that proper ADC interrupts are disabled. It would have been possible to rectify this with either a different random number generator or modifications to the algorithm, but for our purposes, a constant value has been used as the seed.

General Utility Tasks (TickHook)

On a typical embedded system, there may be many operations that take only a few cycles to execute and which have some time dependency associated with them. For example, the debouncing of a key switch. Here a change in the state of a switch needs to be recorded and checked again, perhaps 20 ms later to ensure that any bounce has been removed, and the switch has actually been pressed. This could also be performed within the RTOS task that monitors the switch or a separate task could be created that is solely responsible for monitoring key presses. Unfortunately, this consumes significant memory in creating a new task for a simple operation.

An alternative approach is to add the required code into the RTOS `TickHook` routine. This function is normally called every time a RTOS pre-emptive TICK occurs (in the demo application, this is at 200 Hz). In `MainDemo.c`, a `vApplicationTickHook` function has been added that is executed every 5 ms. The events are counted and after 2 seconds have elapsed, a message is sent to the `hMETERQueue` and also to the `hQVGAQueue`, allowing the tasks to update the displayed temperature, and perform general periodic housekeeping functions.

It should be noted that the `TickHook` function is called as part of the pre-emption ISR. Because of this, any RTOS related calls must be suitable for use within an ISR, and therefore, the `xQueueSendToBackFromISR` functions are used.

GUIDELINES

The main part of this application note is focused on the modifications required in order to allow a number of Microchip stacks to be used together in a RTOS application. We have deliberately kept these changes as simple as possible, since this allows for the shortest development time, and will permit a simple upgrade as new versions of the libraries are released. It is difficult to arrive at a fixed set of rules for modifying software that will work in all future cases, but here are a number of items to consider when modifying any library for a new application.

- Consider all low-level hardware resources, I/O pins and peripherals. If they could be accessed by more than one task at the same time, then the resource should be protected by a mutex using a semaphore or critical section. If they are accessed by more than one task, but only sequentially, then task priorities may be sufficient to protect accesses.
- Having advised the use of critical sections, you should, however, minimize their use as they will typically cause interrupts to be disabled for their duration, possibly preventing execution of other tasks.
- Analyze the libraries and your own application code for delay loops or calls to delay functions. These delay functions should be replaced with calls to the RTOS provided delay functions, as this will allow other tasks to execute if required.
- Check for state machines with delay states, where the code stops at a particular stage waiting for an event to occur. If possible, replace potentially long waits with calls that block waiting for a semaphore or queue event. This may require the creation of another task that can signal the event so it could incur a memory or performance penalty.
- Any libraries that are dependent upon slow external events can potentially have separate tasks created to handle these asynchronous events.
- Application code, and indeed a number of the libraries, will often have common functions. It is fairly obvious that these functions should be replaced with local alternatives that reduce code size. However, when performing these changes, be aware that they may introduce mutual exclusion and atomic access problems if called by multiple tasks.
- Rather than introducing a mutex to prevent multiple tasks from accessing a shared resource, use a 'gatekeeper' task to regulate access. Multiple tasks can then send messages to the gatekeeper requesting operations to be performed without introducing a mutex. These messages can be sent using queues, rather than global memory blocks, which will simplify future code changes if required.
- Allow COTS components and libraries to be installed in their default locations. However, any modifications that are required should be performed upon local copies of the files placed in the project source directory. This ensures the integrity of the original library installation, and if done correctly, the project relative paths will ensure the include files are located easily.
- When upgrading the libraries, they can be installed to their default locations as normal, avoiding overwriting the modified local copies. But it is necessary to ensure that changes to the library files are correctly reflected in the local copies and this can be achieved via a 'diff' utility, which will identify the differences.

CONCLUSION

This application note describes how to use a Real-Time Operating System (RTOS) designed for embedded microcontrollers to simplify the task of integrating multiple software libraries into one application. The demo application uses FreeRTOS; any one of the alternatives listed in Table 1 could be used instead.

Microchip provides source code for its application libraries which allows them to be more easily modified and integrated into a multi-library RTOS-based application. This would not be possible if they were provided in binary form only.

The demo application demonstrates how a feature rich product can be generated using COTS components. If the code is examined in detail, it can be seen that the number of program lines that implement the smart meter functionality is small compared to the size of the library code. This has resulted in an application that could be written in a much shorter amount of time than if all of the code had to be written from scratch. It also demonstrates that complex products can be built with small 16 and 32-bit embedded microcontrollers in a more cost-effective manner than with complicated microprocessor-based solutions that require many megabytes of RAM in order to run systems, such as Linux.

The modifications to the Microchip MiWi networking protocol, graphics and TCP/IP libraries could be simply copied into a new application with minimal changes. Some guidelines have been provided that would allow the programmer to replicate these modifications in their own project if required.

REFERENCES

- *“dsPIC30F/33F Programmer’s Reference Manual”* (DS70157)
- *“16-Bit Language Tools Libraries”* (DS51456)
- *“MPLAB® C32 C Compiler User’s Guide”* (DS51686)
- AN833, *“The Microchip TCP/IP Stack”* (DS00833)
- AN1120, *“Ethernet Theory of Operation”* (DS01120)
- AN1136, *“How to Use Widgets in Microchip Graphics Library”* (DS01136)
- AN1066, *“MiWi™ Wireless Networking Protocol Stack”* (DS01066)
- AN1204, *“Microchip MiWi™ P2P Wireless Protocol”* (DS01204)
- FreeRTOS – <http://www.freertos.org>

APPENDIX A: FUNCTION CALLS

EXAMPLE A-1: FreeRTOS.org FUNCTION

```
xTaskCreate(tskFunc, sName, stkSize, param, priority, &handle);
    tskFunc - pointer to the task function
    sName - task name string
    stkSize - size of the stack in words
    param - a parameter to be passed to the task
    priority - priority of the task
    handle - pointer to the created task handle

xSemaphoreTake(semaphore, timeout);
    semaphore - handle of a semaphore
    timeout - delay timeout (ticks), portMAX_DELAY waits forever

xSemaphoreGive(semaphore);
    semaphore - handle of a semaphore

xQueueSend(hQueue, &msg, timeout);
    hQueue - handle to the queue
    msg - pointer to data to place on the queue (size varies)
    timeout - delay timeout (ticks), portMAX_DELAY wait forever

xQueueSendToBackFromISR(hQueue, &msg, &flag);
    hQueue - handle of the queue
    msg - pointer to data to be placed on queue
    flag - indicates if another task was woken as a result of the send

vTaskDelay(ticks);
    ticks - delay specified number of ticks

portENTER_CRITICAL();
    start a critical section, forming an atomic action, disables interrupts
portEXIT_CRITICAL();
    leave a critical section
```

APPENDIX B: RTOS RESOURCES

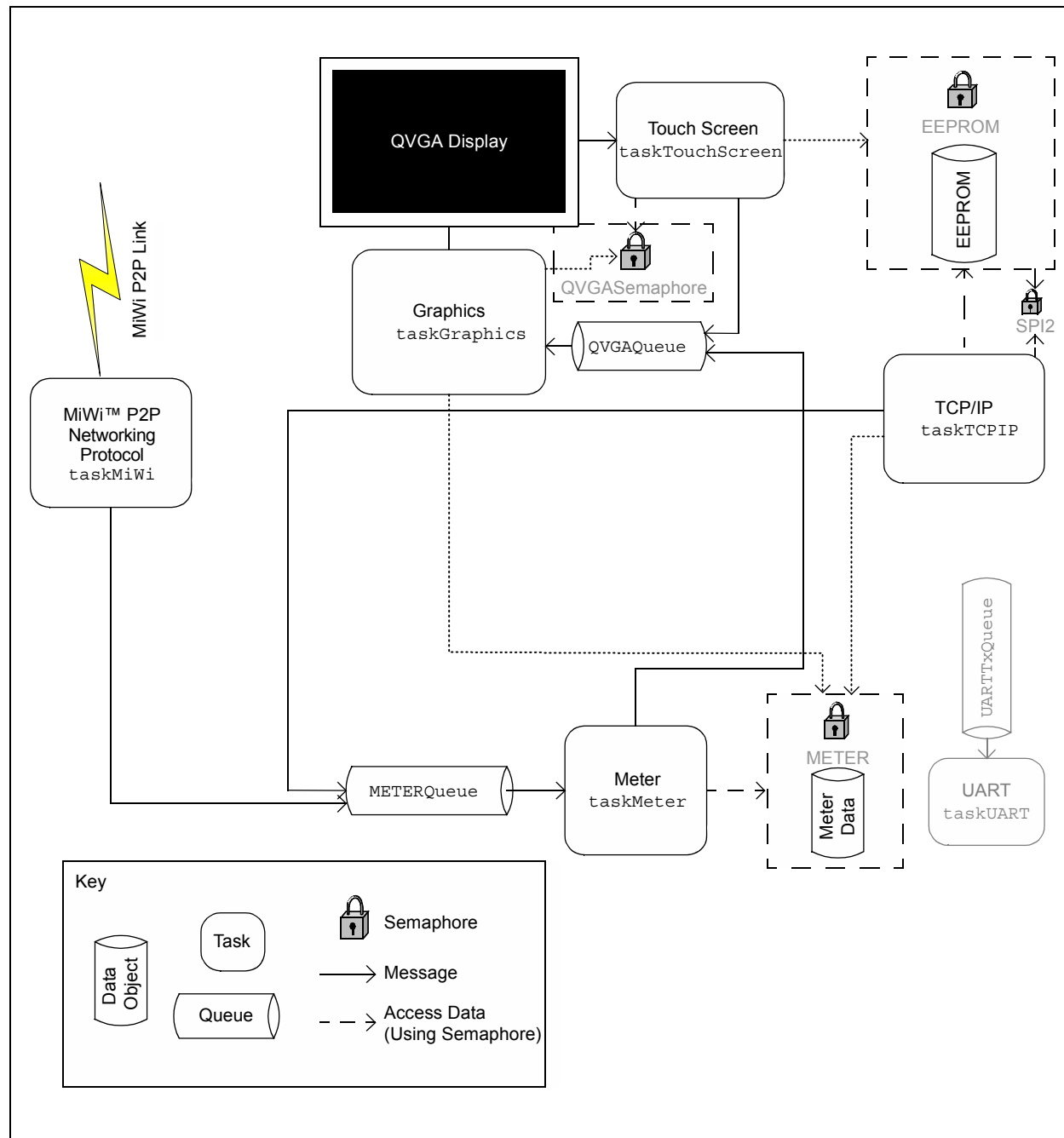
Table B-1 lists the available RTOS resources.

TABLE B-1: DEMONSTRATOR RTOS RESOURCES

| Resource | Object Name | Inputs | Outputs | Function |
|------------------|-----------------|--------------------------|------------------------------------|---|
| Task | taskMeter | METERQueue | Updates to QVGAQueue | Control meter updates |
| Task | taskMiWi | | Updates Meter via METERQueue | Receive RF updates from PIC18F node |
| Task | taskUART | UARTTxQueue | | Displays diagnostic information via UART |
| Task | taskGraphics | QVGAQueue | Accesses gMeter Directly | All screen related output and input; direct access to gMeter object |
| Task | taskTouchScreen | ADC Readings | Touch Events via QVGAQueue | Handle touch screen calibration and processing |
| Task | taskTCPIP | Accesses gMeter Directly | Updates Meter via METERQueue | All TCP/IP and Web-related services |
| Queue | METERQueue | | | Meter updates of temperature, units and unit costs |
| Queue | QVGAQueue | | | Meter updates and touch screen actions |
| Queue | UARTTxQueue | | | Data to be transmitted |
| Queue | UARTRxQueue | | | Unused |
| Semaphore | QVGASemaphore | | | Regulate access to PMP and screen |
| Semaphore | METERSemaphore | | | Regulate access to global gMeter data structure |
| Semaphore | EEPROMSemaphore | | | Regulate access to EEPROM |
| Semaphore | SPI2Semaphore | | | Regulate access to SPI channel, used by EEPROM functions and TCP/IP functions |
| Critical Section | LEDUtils.c | | Isolates Modifications to LED Port | Accesses to port on PIC24F are protected by critical section |
| Critical Section | Tick.c | | Isolates Timer Calculations | Ensures 32-bit timer overflows are atomic actions |

B.1 Resource Interdependency

FIGURE B-1: DIAGRAM OF RESOURCE INTERDEPENDENCY



Note the following details of the code protection feature on Microchip devices:

- Microchip products meet the specification contained in their particular Microchip Data Sheet.
- Microchip believes that its family of products is one of the most secure families of its kind on the market today, when used in the intended manner and under normal conditions.
- There are dishonest and possibly illegal methods used to breach the code protection feature. All of these methods, to our knowledge, require using the Microchip products in a manner outside the operating specifications contained in Microchip's Data Sheets. Most likely, the person doing so is engaged in theft of intellectual property.
- Microchip is willing to work with the customer who is concerned about the integrity of their code.
- Neither Microchip nor any other semiconductor manufacturer can guarantee the security of their code. Code protection does not mean that we are guaranteeing the product as "unbreakable."

Code protection is constantly evolving. We at Microchip are committed to continuously improving the code protection features of our products. Attempts to break Microchip's code protection feature may be a violation of the Digital Millennium Copyright Act. If such acts allow unauthorized access to your software or other copyrighted work, you may have a right to sue for relief under that Act.

Information contained in this publication regarding device applications and the like is provided only for your convenience and may be superseded by updates. It is your responsibility to ensure that your application meets with your specifications. MICROCHIP MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND WHETHER EXPRESS OR IMPLIED, WRITTEN OR ORAL, STATUTORY OR OTHERWISE, RELATED TO THE INFORMATION, INCLUDING BUT NOT LIMITED TO ITS CONDITION, QUALITY, PERFORMANCE, MERCHANTABILITY OR FITNESS FOR PURPOSE. Microchip disclaims all liability arising from this information and its use. Use of Microchip devices in life support and/or safety applications is entirely at the buyer's risk, and the buyer agrees to defend, indemnify and hold harmless Microchip from any and all damages, claims, suits, or expenses resulting from such use. No licenses are conveyed, implicitly or otherwise, under any Microchip intellectual property rights.

Trademarks

The Microchip name and logo, the Microchip logo, dsPIC, KEELOQ, KEELOQ logo, MPLAB, PIC, PICmicro, PICSTART, rPIC and UNI/O are registered trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.


FilterLab, Hampshire, HI-TECH C, Linear Active Thermistor, MXDEV, MXLAB, SEEVAL and The Embedded Control Solutions Company are registered trademarks of Microchip Technology Incorporated in the U.S.A.

Analog-for-the-Digital Age, Application Maestro, CodeGuard, dsPICDEM, dsPICDEM.net, dsPICworks, dsSPEAK, ECAN, ECONOMONITOR, FanSense, HI-TIDE, In-Circuit Serial Programming, ICSP, Mindi, MiWi, MPASM, MPLAB Certified logo, MPLIB, MPLINK, mTouch, Octopus, Omniscient Code Generation, PICC, PICC-18, PICDEM, PICDEM.net, PICkit, PICtail, PIC³² logo, REAL ICE, rLAB, Select Mode, Total Endurance, TSHARC, UniWinDriver, WiperLock and ZENA are trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.

SQTP is a service mark of Microchip Technology Incorporated in the U.S.A.

All other trademarks mentioned herein are property of their respective companies.

© 2009, Microchip Technology Incorporated, Printed in the U.S.A., All Rights Reserved.

 Printed on recycled paper.

QUALITY MANAGEMENT SYSTEM
CERTIFIED BY DNV
== ISO/TS 16949:2002 ==

Microchip received ISO/TS-16949:2002 certification for its worldwide headquarters, design and wafer fabrication facilities in Chandler and Tempe, Arizona; Gresham, Oregon and design centers in California and India. The Company's quality system processes and procedures are for its PIC® MCUs and dsPIC® DSCs, KEELOQ® code hopping devices, Serial EEPROMs, microperipherals, nonvolatile memory and analog products. In addition, Microchip's quality system for the design and manufacture of development systems is ISO 9001:2000 certified.



WORLDWIDE SALES AND SERVICE

AMERICAS

Corporate Office

2355 West Chandler Blvd.
Chandler, AZ 85224-6199
Tel: 480-792-7200
Fax: 480-792-7277
Technical Support:
<http://support.microchip.com>
Web Address:
www.microchip.com

Atlanta

Duluth, GA
Tel: 678-957-9614
Fax: 678-957-1455

Boston

Westborough, MA
Tel: 774-760-0087
Fax: 774-760-0088

Chicago

Itasca, IL
Tel: 630-285-0071
Fax: 630-285-0075

Cleveland

Independence, OH
Tel: 216-447-0464
Fax: 216-447-0643

Dallas

Addison, TX
Tel: 972-818-7423
Fax: 972-818-2924

Detroit

Farmington Hills, MI
Tel: 248-538-2250
Fax: 248-538-2260

Kokomo

Kokomo, IN
Tel: 765-864-8360
Fax: 765-864-8387

Los Angeles

Mission Viejo, CA
Tel: 949-462-9523
Fax: 949-462-9608

Santa Clara

Santa Clara, CA
Tel: 408-961-6444
Fax: 408-961-6445

Toronto

Mississauga, Ontario,
Canada
Tel: 905-673-0699
Fax: 905-673-6509

ASIA/PACIFIC

Asia Pacific Office

Suites 3707-14, 37th Floor
Tower 6, The Gateway
Harbour City, Kowloon
Hong Kong
Tel: 852-2401-1200
Fax: 852-2401-3431

Australia - Sydney

Tel: 61-2-9868-6733
Fax: 61-2-9868-6755

China - Beijing

Tel: 86-10-8528-2100
Fax: 86-10-8528-2104

China - Chengdu

Tel: 86-28-8665-5511
Fax: 86-28-8665-7889

China - Hong Kong SAR

Tel: 852-2401-1200
Fax: 852-2401-3431

China - Nanjing

Tel: 86-25-8473-2460
Fax: 86-25-8473-2470

China - Qingdao

Tel: 86-532-8502-7355
Fax: 86-532-8502-7205

China - Shanghai

Tel: 86-21-5407-5533
Fax: 86-21-5407-5066

China - Shenyang

Tel: 86-24-2334-2829
Fax: 86-24-2334-2393

China - Shenzhen

Tel: 86-755-8203-2660
Fax: 86-755-8203-1760

China - Wuhan

Tel: 86-27-5980-5300
Fax: 86-27-5980-5118

China - Xiamen

Tel: 86-592-2388138
Fax: 86-592-2388130

China - Xian

Tel: 86-29-8833-7252
Fax: 86-29-8833-7256

China - Zhuhai

Tel: 86-756-3210040
Fax: 86-756-3210049

ASIA/PACIFIC

India - Bangalore

Tel: 91-80-3090-4444
Fax: 91-80-3090-4080

India - New Delhi

Tel: 91-11-4160-8631
Fax: 91-11-4160-8632

India - Pune

Tel: 91-20-2566-1512
Fax: 91-20-2566-1513

Japan - Yokohama

Tel: 81-45-471- 6166
Fax: 81-45-471-6122

Korea - Daegu

Tel: 82-53-744-4301
Fax: 82-53-744-4302

Korea - Seoul

Tel: 82-2-554-7200
Fax: 82-2-558-5932 or
82-2-558-5934

Malaysia - Kuala Lumpur

Tel: 60-3-6201-9857
Fax: 60-3-6201-9859

Malaysia - Penang

Tel: 60-4-227-8870
Fax: 60-4-227-4068

Philippines - Manila

Tel: 63-2-634-9065
Fax: 63-2-634-9069

Singapore

Tel: 65-6334-8870
Fax: 65-6334-8850

Taiwan - Hsin Chu

Tel: 886-3-6578-300
Fax: 886-3-6578-370

Taiwan - Kaohsiung

Tel: 886-7-536-4818
Fax: 886-7-536-4803

Taiwan - Taipei

Tel: 886-2-2500-6610
Fax: 886-2-2508-0102

Thailand - Bangkok

Tel: 66-2-694-1351
Fax: 66-2-694-1350

EUROPE

Austria - Wels

Tel: 43-7242-2244-39
Fax: 43-7242-2244-393

Denmark - Copenhagen

Tel: 45-4450-2828
Fax: 45-4485-2829

France - Paris

Tel: 33-1-69-53-63-20
Fax: 33-1-69-30-90-79

Germany - Munich

Tel: 49-89-627-144-0
Fax: 49-89-627-144-44

Italy - Milan

Tel: 39-0331-742611
Fax: 39-0331-466781

Netherlands - Druenen

Tel: 31-416-690399
Fax: 31-416-690340

Spain - Madrid

Tel: 34-91-708-08-90
Fax: 34-91-708-08-91

UK - Wokingham

Tel: 44-118-921-5869
Fax: 44-118-921-5820

03/26/09