Micriμm

Empowering Embedded Systems

μC/OS-II
μC/Probe

and the

The Microchip PIC24

(Using the Explorer 16 Evaluation Board)

Application Note

AN-1017

www.Micrium.com

Table Of Contents

1.00 Introduction

This document shows example code for using **μC/OS-II** and **μC/Probe** on a Microchip dsPIC33 processor. To demonstrate the dsPIC33, we used a Microchip Explorer 16 Evaluation Board as shown in Figure 1-1. **μC/LCD** is used to drive the Hitachi complaint LCD controller.

This example uses the **μC/OS-II** port described in AN-1024.

We used the Microchip MPLab IDE and C30 Compiler Tools to demonstrate this application. The software versions used were MPLAB v8.02, C30 v2.02, and ASM v2.02.

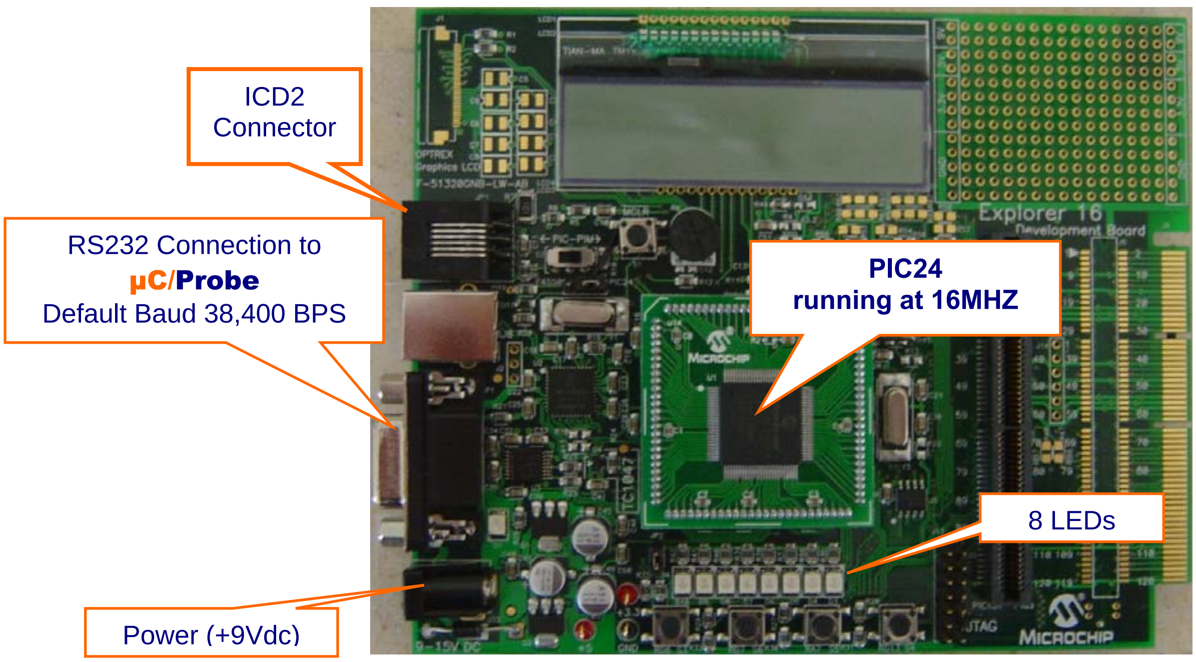


Figure 1-1, Microchip Explorer 16 Evaluation Board

The application code is downloaded into Flash using an ICD2 or Real ICE in 'Programming Mode'. When the application is started, the 8 onboard LED's switch on and off from left to right producing a scrolling like effect. Additionally, the LCD screen is initialized and displays various messages thoughout the run-time of the application.

1.01 Port Specific Details

This **μC/OS-II** port has been designed to operate using the PIC24 Large memory model.

Furthermore, in order to utilize all of the onboard LEDs, it is critical that the optional JTAG port be disabled. This can be done from the configuration bit screen within MPLab, via software during runtime, or via Macro in source code. Each compiler has different requirements for setting the configuration bits. In most cases, the configuration bits are set via macro from the top of `BSP.H`. However, some circumstances require the use of the MPLAB configuration bits as an alternative. Both methods should not be used simulataneously.

The OS port makes use of the on chip PLL. In order to select the XT/HS/EC with PLL oscillator source, an MPLab macro has been used at the top of `BSP.C.` When enabled, the on chip PLL multiplies the primary oscillator by 4. The processor is then clocked at one half of the PLL frequency. In this case, the PIC24 is running at 16 MHz.

The OS port assumes the availability of Timer number 2, a B-type timer for use with the OS Ticker. If your application absolutely requires this timer, then you may optionally select Timer number 4, also a B-type timer, for use with the OS Ticker by changing `BSP_OS_TMR_SEL` in `BSP.H` from 2 to 4.

ISR handlers are defined using predefined compiler specific keyword and names such that MPLab automatically populates the vector table upon loading the executable into flash memory. Every vendors compiler handles vector table population differently. Users should not use IDE or compiler specific keywords in order to create ISRs written entirely in C unless the necessary steps to ensure that the OS is properly informed of the interrupt are taken. This usually involves inline assembly. A better method is to populate the ISR table with the address of a first level ISR handler written in assembly which calls a user specified 'C' code handler function. An example assembly ISR may be copied from the OS tick ISR with minimal changes. All ISRs should be written as specified in the PIC24 OS port application note, see AN-1024.

Finally, **μC/Probe** requires a free running timer for making time measurements. Since the PIC24 does not provide a free-running timer mode, the example described herein cannot share a timer with the **μC/OS-II** Ticker. As a result, a second timer is configured from `BSP.C` with interrupts disabled and a Period register value of `0xFFFF`. This simulates a free-running timer that can be utilized by **μC/Probe** without having any effect on the **μC/OS-II** Tick interrupt source.

1.02 μC/Probe

μC/Probe is a Microsoft Windows program that displays the content of system variables on various user definable graphical elements such as simulated mechanical counters, graphs, on-screen LEDs and so on.

In order for **μC/Probe** to display information about your application, an ELF file, must be generated by the user's compiler. The ELF file contains the names and addresses of all the global symbols referenced within the users embedded application. Only symbols that have been allocated memory, e.g. not allocated on the stack, are able to be monitored by **μC/Probe**. Global and static variables are examples of variables that may be monitored.

The user places components (such as gauges, labels, and charts) into a Data Screen in a **μC/Probe** workspace. Each one of these controls is then assigned to one or more of the variables from the Symbol Browser. The Symbol Browser lists all symbols referenced from within the ELF file. Symbols associated with components placed on an open Data Screen will be updated after the user presses the start button (assuming the user's PC is connected to the target and the target is running).

μC/Probe currently interfaces with a target processor via JTAG, RS-232, UDP and USB. A small section of code resident on the target receives commands from the Windows application and responds to those commands. The commands ask for a certain number of bytes located at a certain address, for example, "Send 16 bytes beginning at 0x0040102C". The Windows application, upon receiving the response, updates the appropriate component(s) on the data screen(s) with the new values.

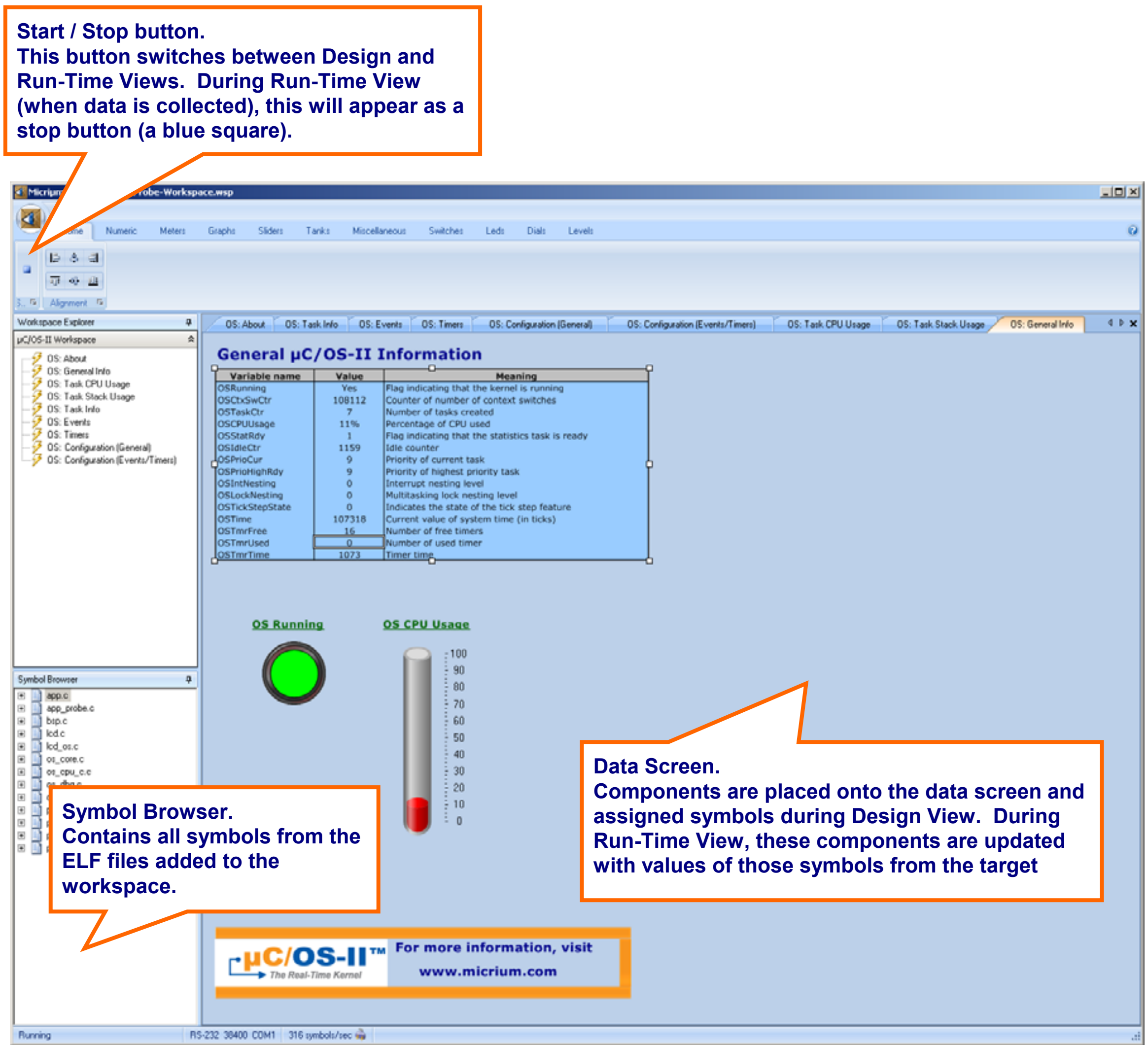


Figure 2-1. μC/Probe Windows Program

To use **μC/Probe** with the example project (or your application), do the following:

1. **Download and Install μC/Probe.** A trial version of **μC/Probe** can be downloaded from the Micriμm website at

   http://www.micrium.com/products/probe/probe.html

2. **Open μC/Probe.** After downloading and installing this program, open the example **μC/Probe** workspace for **μC/OS-II**, named ***OS-Probe.wsp***, which should be located in the AN-1208 Codewarrior project directory.

You may also open one of the sample workspaces that comes with **µC/Probe**. The sample workspaces, located in the **µC/Probe** target directory, contains generic workspaces for **µC/OS-II**, as well as other Micrium software modules.

3. **Connect Target to PC**. Currently, **µC/Probe** can use RS-232 to retrieve information from the target. You should connect a RS-232 cable between your target and computer.

   Load Your ELF File. The example projects included with this application note are already configured to output an ELF file. (If you are using your own project, please refer to Appendix A of the **µC/Probe** user manual for directions for generating an ELF file with your compiler.) Codewarrior generates an ELF file with a .abs extension. This file is located in a directory named BIN within the sample project directory.

   To load this ELF file, right-click on the symbol browser and choose “Add Symbols”. Navigate to the file directory, select the file, and choose “OK”.

4. **Configure the RS-232 Options**. In **µC/Probe**, choose the “Options” menu item on the “Tools” menu. A dialog box as shown in Figure 6-2 (left) should appear. Choose the “RS-232” radio button. Next, select the “RS-232” item in the options tree, and choose the appropriate COM port and baud rate. The baud rate for the projects accompanying this application note is 38,400 baud.
5.
6. **Start Running**. You should now be ready to run **µC/Probe**. Just press the run button to see the variables in the open data screens update.

1.03 Directories and Files

The code and documentation of the port are placed in a directory structure according to "AN-2002, µC/OS-II Directory Structure". Specifically, the files are placed in the following directories:

μC/OS-II:

\Micrium\Software\uCOS-II\Source

This directory contains the processor independent code for **μC/OS-II**. The version used was 2.86.

\ Micrium\Software\uCOS-II\Ports\Microchip\PIC24FJ128\C30

This directory contains the standard processor specific files for a **μC/OS-II** port assuming the Microchip MPLab IDE and C30 Compiler Tools. In fact, these files could easily be modified to work with other tool chains. However, you would place the modified files in a different directory. Specifically, this directory contains the following files:

```
os_cpu.h
os_cpu_a.s
os_cpu_c.c
os_cpu_util.s
os_dbg_c
```

`os_dbg.c` is included to provide additional information to Kernel Aware debuggers like IAR's C-Spy and is not required for this **μC/OS-II** port. However, we recommend building the project with it for reference on future projects.

μC/OS-II ports for the Hi-Tech iccDSPIC and IAR dsPICC compilers are available upon request.

μC/Probe:

\Micrium\Software\uC-Probe\Target\Communication\Generic\OS\uCOS-II

This directory contains the OS dependent interface for the communication layer of **μC/Probe**. If you plan to run **μC/Probe** with a different RTOS, or without any RTOS, the following files would have to be adjusted accordingly:

```
probe_com_os.c
```

**\Micrium\Software\uC-Probe\Target\Communication\Generic
\RS-232\OS\uCOS-II**

This directory contains OS dependent interface code for the RS-232 specific portion of **μC/Probe**, specifically the code necessary to generate an optional Rx packet parse task. If you plan to run **μC/Probe** with a different RTOS, modifications to the files listed below will have to be made. If you are not running an RTOS, the following files may be excluded from the build.

```
probe_rs232_os.c
```

\Micrium\Software\uC-Probe\Target\Communication\Generic\RS-232\Ports\Microchip\PIC24\C30

This directory contains the **μC/Probe** hardware port files for the dsPIC33 processor.

```
probe_rs232c.c
probe_rs232_a.s
```

\Micrium\Software\uC-Probe\Target\Communication\Generic\RS-232\Source

This directory contains target independent source code for the **μC/Probe** RS-232 communication layer. Specifically, this directory contains the following files:

```
probe_rs232.c
probe_rs232.h
```

\Micrium\Software\uC-Probe\Target\Communication\Generic\Source

This directory contains target independent source code for the **μC/Probe** communication layer. Specifically, this directory contains the following files:

```
probe_com.c
probe_com.h
```

\Micrium\Software\uC-Probe\Target\Plugins\uCOS-II

This directory contains the target independent source code for **μC/Probe**. Specifically, this directory contains the following files:

```
os_probe.c
os_probe.h
```

μC/CPU:

\Micrium\Software\uC-CPU

This directory contains processor independent files for **μC/CPU**. **μC/CPU** contains code for entering and existing critical sections, as well as macro definitions for the 'C' programming datatypes used in most Micrium products. Specifically, this directory includes:

```
cpu_def.h
```

\Micrium\Software\uC-CPU\Microchip\PIC24FJ128\C30

This directory contains processor port files for **μC/CPU**. **μC/CPU** contains code for entering and existing critical sections, as well as macro definitions for the 'C' programming datatypes used in most Micrium products.

```
cpu.h
```

μC/LIB:

\Micrium\Software\uC-LIB

This directory contains `lib_def.h`, which provides `#defines` for useful constants (like `DEF_TRUE` and `DEF_DISABLED`) and macros.

\Micrium\Software\uC-LIB\Doc

This directory contains the documentation for **μC/LIB**.

Application Code:

\Micrium\Software\EvalBoards\Microchip\Explorer16\PIC24FJ128 \MPLAB_C30\OS-Probe

This directory is the directory that contains the source code for an example running on the Explorer 16 evaluation board. It assumes the presence of **μC/OS-II**.
This directory contains:

```
app.c
app_cfg.h
app_hooks.c
app_probe.c
includes.h
os_cfg.h
OS-Probe.*
probe_com_cfg.h
```

`app.c` contains the example code, `app_cfg.h` contains application specific configuration information such as task priorities and stack sizes, `app_hooks.c` contains user code for the **μC/OS-II** hooks, while `app_probe.c` contains initialization code for **μC/Probe**.

`includes.h` contains a master include file used by the application, `os_cfg.h` is the **μC/OS-II** configuration file and `probe_com_cfg.h` contains user modifiable configuration constants for **μC/Probe**. Lastly, `OS-Probe.*` are the Microchip MPLab IDE and C30 Compiler Tools project files.

Micrium\Software\EvalBoards\Microchip\Explorer16\PIC24FJ128 \MPLAB_C30\BSP

This directory contains the Board Support Package for the Explorer16 evaluation board and the PIC24 MCU. While some of the code in this directory may work on other PIC24 derivatives, routines that are hardware dependent such as LED_On() will require modification depending on the hardware design of your EVB. This directory contains:

```
bsp.c
bsp.h
bsp_a.s
bsp_lcd.c
p24FJ128GA010.gld
p24FJ128GA010.h
p24FJ128GA010.inc
```

1.04 MPLab IDE

We used the MPLab IDE and C30 Compiler Tools to test the PIC24 example. You can of course use **μC/OS-II** with other tools. Figures 1-3 and 1-4 show the project build options and figure 1-5 shows the MPLab source tree respectively. See section 1.01 for more information about the build options.

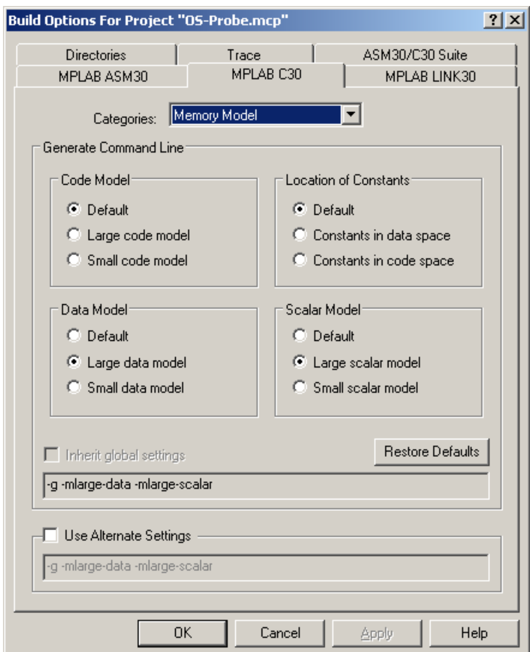


Figure 1-3

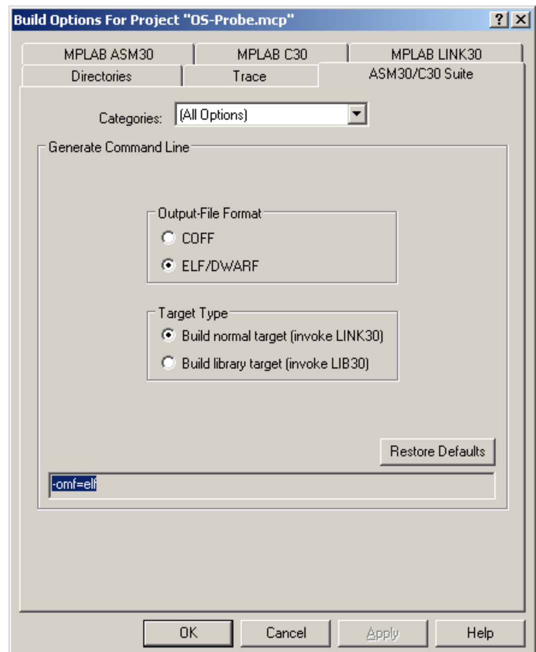


Figure 1-4

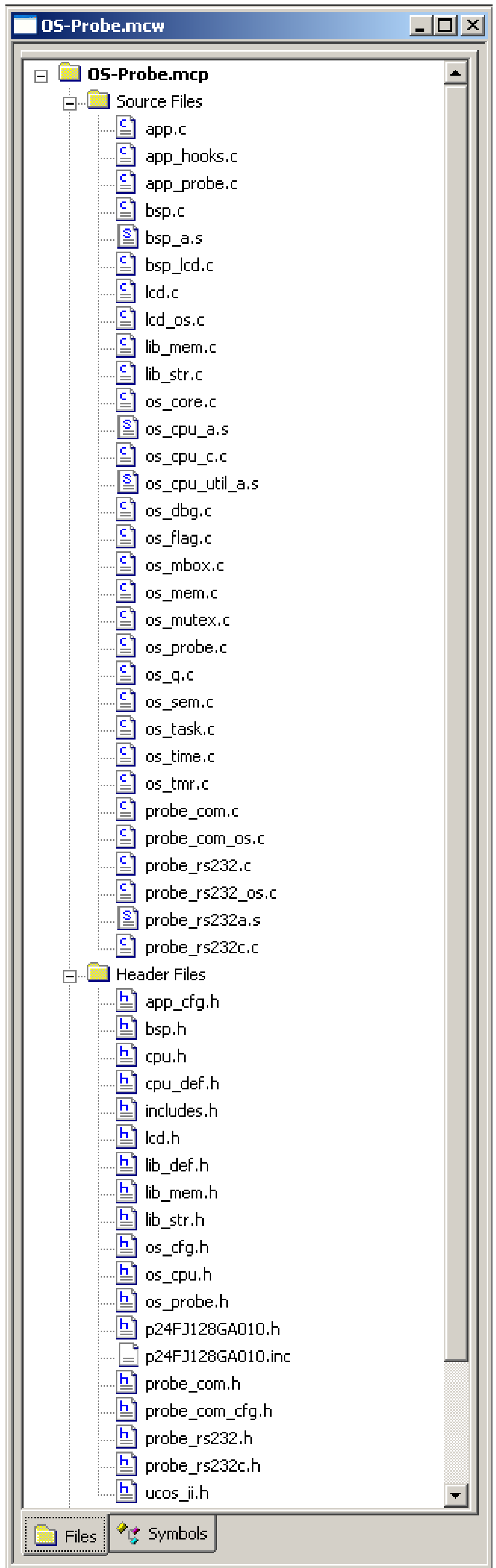


Figure 1-5, MPLab Source Tree

2.00 Example Code

As mentioned in the previous section, the example code for this board is found in the following directories and will be briefly described:

**\Micrium\Software\EvalBoards\Microchip\Explorer16\PIC24FJ128
\MPLAB_C30\OS-Probe**

It should be noted that the file `p24FJ128GA010.gld` (BSP directory) is critical for linking the example code to the correct address ranges suitable for the PIC24. This file should not need to be modified.

2.01 Example Code, app.c

`app.c` demonstrate some of the capabilities of **µC/OS-II.**

Listing 2-1, main()

```
void  main (void)                                                   (1)
{
    INT8U  err;


    BSP_IntDisAll();                                                (2)

    OSInit();                                                       (3)

    OSTaskCreateExt(AppStartTask,                                   (4)
                    (void *)0,
                    (OS_STK *)& AppStartTaskStk[APP_START_TASK_STK_SIZE - 1],
                    APP_START_TASK_PRIO,
                    APP_START_TASK_PRIO,
                    (OS_STK *)&AppStartTaskStk[0],
                    APP_START_TASK_STK_SIZE,
                    (void *)0,
                    OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR);

#if OS_TASK_NAME_SIZE > 11
    OSTaskNameSet(APP_START_TASK_PRIO, "Start Task", &err);         (5)
#endif

    OSStart();                                                      (6)
}
```

L2-1(1) As with most C applications, the code starts in `main()`.

L2-1(2) We start off by calling a BSP function (see `bsp.c`) that will disable all interrupts. We do this to ensure that initialization doesn't get interrupted in case we do a 'warm restart'.

L2-1(3) As will all **µC/OS-II** applications, you need to call `OSInit()` before creating any task or other kernel objects.

L2-1(4) We then create at least one task (in this case we used `OSTaskCreateExt()` to specify additional information about your task to **µC/OS-II**). It turns out that **µC/OS-II** creates one and possibly two tasks in `OSInit()`. As a minimum, **µC/OS-II** creates an idle task (`OS_TaskIdle()` which is internal to **µC/OS-II**) and `OS_TaskStat()` (if you set `OS_TASK_STAT_EN` to `1` in `OS_CFG.H`). `OS_TaskStat()` is also an internal task in **µC/OS-II**.

L2-1(5) As of V2.6x, you can now name **µC/OS-II** tasks (and other kernel objects) and be able to display task names at run-time or, with a debugger. In this case, we name our first task 'Start Task'.

L2-1(6) We finally start **µC/OS-II** by calling `OSStart()`. **µC/OS-II** will then start executing `AppStartTask()` since that's the highest priority task created. `OSStart()` does not return.

Listing 2-2, AppTaskStart()

```
static  void  AppStartTask (void *p_arg)
{
   (void)p_arg;

    BSP_Init();                                                         (1)

#if OS_TASK_STAT_EN > 0
    OSStatInit();                                                       (2)
#endif

#if (uC_PROBE_OS_PLUGIN > 0) || (uC_PROBE_COM_MODULE > 0)
    AppProbeInit();                                                     (3)
#endif

    AppTaskCreate();                                                    (4)

    LED_Off(0);                                                         (5)

    while (DEF_TRUE) {                                                  (6)
        for (j = 0; j < 4; j++) {
            for (i = 1; i <= 8; i++) {
                LED_On(i);
                OSTimeDlyHMSM(0, 0, 0, 20);                             (7)
                LED_Off(i);
                OSTimeDlyHMSM(0, 0, 0, 20);
            }

            for (i = 7; i >= 2; i--) {
                LED_On(i);
                OSTimeDlyHMSM(0, 0, 0, 20);
                LED_Off(i);
                OSTimeDlyHMSM(0, 0, 0, 20);
            }
        }

        for (i = 0; i < 4; i++) {
            LED_On(1);
            LED_On(2);
            LED_On(3);
            LED_On(4);
            OSTimeDlyHMSM(0, 0, 0, 50);
            LED_Off(1);
            LED_Off(2);
            LED_Off(3);
            LED_Off(4);
            OSTimeDlyHMSM(0, 0, 0, 50);
        }
```

```
    }
}
```

L2-2(1) `BSP_Init()` is called to initialize the Board Support Package – the I/Os, the tick interrupt, and so on. `BSP_Init()` will be discussed in the next section.

L2-2(2) `OSStatInit()` computes how fast the CPU runs when `OS_TASK_STAT_EN` is set to 1 in os_cfg.h.

L2-2(3) Initialize **µC/Probe**. Probe communication port settings may be adjusted from within `probe_com_cfg.h` as well as `app_probe.c`.

L2-2(4) `AppTaskCreate()` is a user defined function for creating additional **µC/OS-II** tasks. This function is not required and additional tasks could have been created directly within `AppStartTask()`. This function has been used to create the `AppLCDTask()` which drives the LCD for this example.

L2-2(5) Shut off all onboard LEDs.

L2-2(6) As with all task managed by **µC/OS-II**, the task body must be in the form of an infinite loop. Tasks managed by **µC/OS-II** must never be allowed to exit. Instead, tasks should be deleted using `OSTaskDel()` when they are no longer desired. This task performs the LED illumination and scrolling effect used within this application.

L2-2(7) As **µC/OS-II** tasks must either enter an infinite loop 'waiting' for some event to occur or terminate itself. In this case, we wait for time to expire as the 'event'. This is accomplished by calling `OSTimeDlyHMSM()`.

2.02 Example Code, app_cfg.h

This file is used to configure:

- the **µC/OS-II** task priorities of each of the tasks in your application
- the stack size for each tasks
- **µC/OS-Probe**

The reason this is done here is to make it easier to configure your application from a single file.

2.03 Example Code, includes.h

`includes.h` is a 'master' header file that contains `#include` directives to include other header files. This is done to make the code cleaner to read and easier to maintain.

2.04 Example Code, os_cfg.h

This file is used to configure **µC/OS-II** and defines the maximum number of tasks that your application can have, which services will be enabled (semaphores, mailboxes, queues, etc.), the size of the idle and statistic task and more. In all, there are about `60` or so `#define` that you can set in this file. Each entry is commented and additional information about the purpose of each `#define` can be found in the **µC/OS-II** book. `os_cfg.h` assumes you have **µC/OS-II** V2.80 or higher but also works with previous versions of **µC/OS-II**.

2.05 Example Code, OS-Probe.*

`*.MCP`, `*.MCS`, `*. MCW` are MPLab project files.

3.00 Board Support Package (BSP)

BSP stands for Board Support Package and provides functions to encapsulate common I/O access functions in order to make it easier for you to port your application code. In fact, you should be able to create other applications using the PIC24 and Explorer 16 Evaluation Board and reuse these functions thus saving you a lot of time.

The BSP performs the following functions:

- Determine the PIC24s CPU clock and peripheral frequencies
- Configure the LED I/Os for the Explorer 16 Evaluation Board and PIC24 CPU
- Configuration and handling of the **μC/OS-II** tick timer
- Configuration and handling of the **μC/OS-Probe** measurement timer

The BSP for the Explorer 16 Evaluation Board is found in the follow directory.

```
\Micrium\Software\EvalBoards\Microchip\Explorer16\PIC24FJ128
        \MPLAB_C30\BSP
```

The BSP files are:

```
bsp.c
bsp.h
bsp_a.s
bsp_lcd.c
p24FJ128GA010.gld
p24FJ128GA010.h
p24FJ128GA010.inc
```

3.01 Board Support Package, p24FJ128GA010.gld

`p24FJ128GA010.gld` is a linker command file that allows the example code to be targeted to the PIC24 Explorer 16 Evaluation Board. This file basically indicates where code and data will be located in memory. Example code is assumed to be placed in Flash memory using the Microchip ICD2 in circuit debugger module.

3.02 Board Support Package, bsp*.*

We will not be discussing every aspect of the BSP but only cover topics that require special attention.

Your application code must call `BSP_Init()` to initialize the BSP. `BSP_Init()` in turn calls other functions as needed.

Listing 3-1, MPLab Macros

```
    _CONFIG1(JTAGEN_OFF);                                          (1)
    _CONFIG2(FNOSC_PRIPLL & POSCMOD_XT);                           (2)
```

L3-1(1) This macro disables JTAG capabilities of the PIC24 and thus enables the use of all PORT A I/O pins which are connected to the onboard LEDs.

L3-1(2) This macro selects the primary XT oscillator and PLL as the PIC24 core clock source. The PLL output frequency is then divided by 2 and fed into the core as SYSCLOCK, the core clock frequency. In this case, SYSCLOCK is 16 MHz.

Note: These macros must be declared in the GLOBAL area of `BSP.C`

Listing 3-2, BSP_Init()

```
void  BSP_Init (void)
{
    LED_Init();                                                    (1)
    Tmr_TickInit();                                                (2)
}
```

L3-2(1) We then call `LED_Init()` to initialize the onboard LED I/O Pins as outputs.

L3-2(2) We then call `Tmr_TickInit()`to initialize Timer #2 or Timer #4 depending on the value of `BSP_OS_TMR_SEL` in `BSP.H`. The configured timer is then used to generate interrupts for the **µC/OS-II** clock tick. The code for this function is described below.

Listing 3-3, Tmr_TickInit()

```
static  void  Tmr_TickInit (void)
{
    INT32U  tmr_frq;
    INT16U  cnts;


    tmr_frq    =   BSP_CPU_ClkFrq();                                  (1)
    cnts       =   (tmr_frq / OS_TICKS_PER_SEC) - 1;                  (2)

#if BSP_OS_TMR_SEL == 2                                               (3)
    T2CON      =   0;                                                 (4)
    TMR2       =   0;                                                 (5)
    PR2        =   cnts;                                              (6)
    IPC1      &=  ~T2IP_MASK;                                         (7)
    IPC1      |=  (TIMER_INT_PRIO << 12);                             (8)
    IFS0      &=  ~T2IF;                                              (9)
    IEC0      |=   T2IE;                                             (10)
    T2CON     |=   TON;                                              (11)
#endif

#if BSP_OS_TMR_SEL == 4                                              (12)
    T4CON      =   0;
    TMR4       =   0;
    PR4        =   cnts;
    IPC6      &=  ~T4IP_MASK;
    IPC6      |=  (TIMER_INT_PRIO << 12);
    IFS1      &=  ~T4IF;
    IEC1      |=   T4IE;
    T4CON     |=   TON;
#endif
}
```

L3-3(1) Get the CPU operating frequency in Hz in order to calculate the correct number of timer increments for the desired OS Tick rate.

L3-3(2) Compute the number of timer increments necessary to generate the desired OS Tick rate.

L3-3(3) If `BSP_OS_TMR_SEL` in `BSP.H` is defined as 2, then configure timer #2.

L3-3(4) Set default settings for the timer. Stop the timer, set the timer to derive its clock from Fcy (PLL Output), configure the timer with a prescaler of 1, and set the timer to operate in 16 bit mode.

L3-3(5) Reset the timer count to 0.

L3-3(6) Set the compare value that the counter will increment to before generating a match interrupt. This was computed in L3-3(2).

L3-3(7) Clear the interrupt priority bits so that the correct priority may be written in L3-3(9).

L3-3(8) Set the timer interrupt priority to a default value of 4.

L3-3(9) Clear all pending timer interrupts.

L3-3(10) Enable timer interrupts.

L3-3(11) Start the timer.

L3-3(12) Perform the same procedure for timer #4 instead of timer #2 If `BSP_OS_TMR_SEL` in `BSP.H` is defined as 4.

It should be noted that the timer automatically resets to 0 after a match interrupt is generated. See figure 3-1 for more details.

Listing 3-4, Tmr_TickISR_Handler()

```
void  Tmr_TickISR_Handler (void)
{
#if  BSP_OS_TMR_SEL == 2
    IFS0 &= ~T2IF;                                                    (1)
#endif

#if  BSP_OS_TMR_SEL == 4
    IFS1 &= ~T41F;
#endif

    OSTimeTick();                                                     (2)
}
```

This function is called from an assembly interrupt service routine which properly informs **μC/OS-II** of the interrupt condition, calls the interrupt handler, and clears the interrupt source. See AN-1024 and the file description `BSP_A.S` for more details.

L3-4(1) Clear the interrupt source.

L3-4(2) `OSTimeTick()` is called to handle the **μC/OS-II** clock tick.

The on chip timer generates a time tick by letting the up-counter run from `0x0000` to the computed match value configured in `Tmr_TickInit()`. After an interrupt is generated, the counter resets to zero and continues counting until the next match occurs.

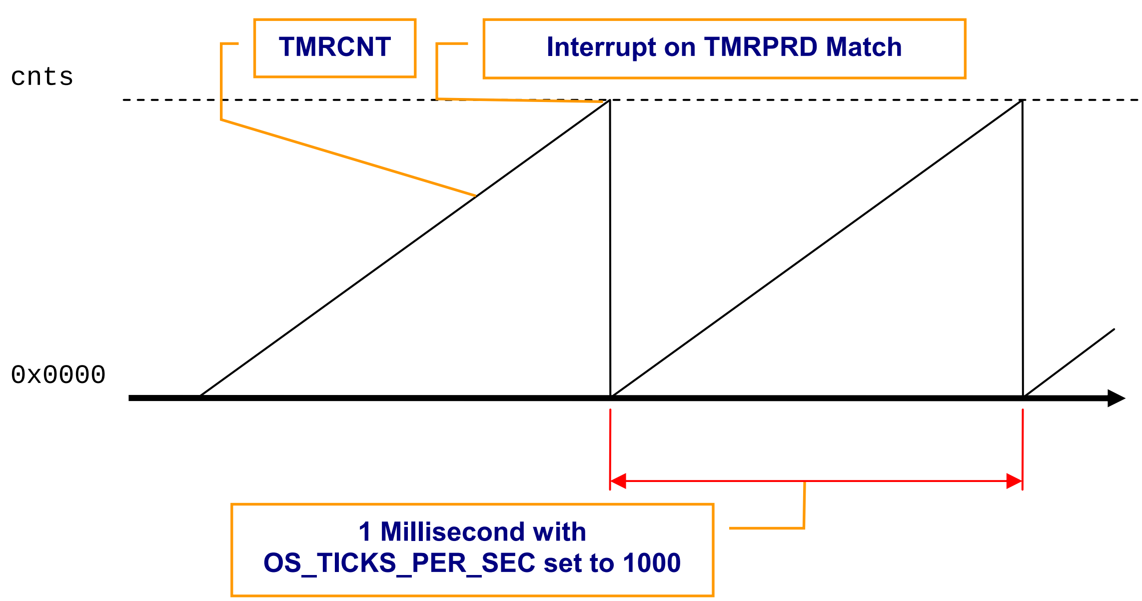


Figure 3-1, OS Tick Timer Operation

When the selected Timer issues an interrupt, the processor vectors to `__T2Interrupt()` or `__T4Interrupt()` depending on which timer is enabled for use with the OS Ticker. These ISR functions both call `Tmr_TickISR_Handler()` as described above in Listing 3-3. Only 1 timer for the OS Ticker may be enabled at a time.

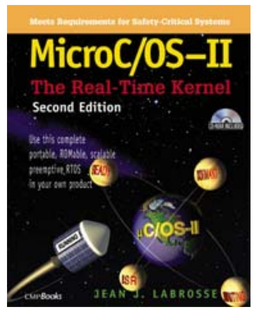
You should note that ALL of your ISRs should be written in assembly where OS related processing may take place before calling an interrupt handler function of the form '`interrupt void MyISR_Handler(void)`' Refer to **AN-1024** for details.

Licensing

If you intend to use **µC/OS-II** in a commercial product, remember that you need to contact **Micriµm** to properly license its use in your product. The use of **µC/OS-II** in commercial applications is **NOT-FREE**. Your honesty is greatly appreciated.

References

MicroC/OS-II, The Real-Time Kernel, 2nd Edition
Jean J. Labrosse
CMP Technical Books, 2002
ISBN 1-5782-0103-9



Contacts

CMP Books, Inc.
6600 Silacci Way
Gilroy, CA 95020 USA
Phone Orders: 1-800-500-6875
or 1-408-848-3854
Fax Orders: 1-408-848-5784
e-mail: rushorders@cmpbooks.com
WEB: http://www.cmpbooks.com

Micriµm
949 Crestview Circle
Weston, FL 33327
USA
954-217-2036
954-217-2037 (FAX)
e-mail: Jean.Labrosse@Micrium.com
WEB: www.Micrium.com

Microchip Technology Inc.
2355 West Chandler Blvd.
Chandler, Arizona 85224-6199
USA
480-792-7200
WEB: www.MicroChip.com