4-2 HOW THE EXAMPLE PROJECT WORKS

The code for **main()** is duplicated and shown in Listing 4-1 so that it is more readable.

```
int  main (void)
{
    OS_ERR  err;


    BSP_IntDisAll();                                                        (1)
    OSInit(&err);                                                           (2)
    OSTaskCreate((OS_TCB     *)&AppTaskStartTCB,                            (3)
                 (CPU_CHAR   *)"App Task Start",
                 (OS_TASK_PTR )AppTaskStart,
                 (void       *)0,
                 (OS_PRIO     )APP_TASK_START_PRIO,
                 (CPU_STK    *)&AppTaskStartStk[0],
                 (CPU_STK_SIZE)APP_TASK_START_STK_SIZE / 10,
                 (CPU_STK_SIZE)APP_TASK_START_STK_SIZE,
                 (OS_MSG_QTY  )5,
                 (OS_TICK     )0,
                 (void       *)0,
                 (OS_OPT      )(OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR),
                 (OS_ERR     *)&err);
    OSStart(&err);                                                          (4)
}
```

Listing 4-1 **main() in app.c**

L4-1(1) **main()** starts by calling **BSP_IntDisAll()**. The code for this function is found in **bsp_int.c**. **BSP_IntDisAll()** simply calls **CPU_IntDis()** to disable all interrupts. The reason a bsp.c function is used instead of simply calling **CPU_IntDis()** is that on some processors, it is necessary to disable interrupts from the interrupt controller, which would then be appropriately handled by a bsp.c function. This way, the application code can easily be ported to another processor.

L4-1(2) **OSInit()** is called to initialize µC/OS-III. Normally, you will want to verify that **OSInit()** returns without error by verifying that **err** contains **OS_ERR_NONE** (*i.e.* the value 0). You can do this with the debugger by single stepping through the code (step over) and stop after **OSInit()** returns. Simply 'hover' the mouse over **err** and the current value of **err** will be displayed.

OSInit() creates four internal tasks: the idle task, the tick task, the timer task, and the statistic task. The interrupt handler queue task is not created because in os_cfg.h, OS_CFG_ISR_POST_DEFERRED_EN set to 0.

L4-1(3) OSTaskCreate() is called to create an application task called AppTaskStart(). OSTaskCreate() contains 13 arguments described in Appendix A, *µC/OS-III API Reference Manual* in Part I of this book.

AppTaskStartTCB is the OS_TCB used by the task. This variable is declared in the 'Local Variables' section in app.c, just a few lines above main().

AppTaskStartStk[] is an array of CPU_STKs used to declare the stack for the task. In µC/OS-III, each task requires its own stack space. The size of the stack greatly depends on the application. In this example, the stack size is declared through APP_TASK_START_STK_SIZE, which is defined in app_cfg.h. 128 CPU_STK elements are allocated, which, on the Cortex-M3, corresponds to 512 bytes (each CPU_STK entry is 4 bytes, see cpu.h,) and this should be sufficient stack space for the simple code of AppTaskStart().

APP_TASK_START_PRIO determines the priority of the start task and is defined in app_cfg.h.

L4-1(4) OSStart() is called to start the multitasking process. With the application task, µC/OS-III will be managing five tasks. However, OSStart() will start the highest priority of the tasks created. In our example, the highest priority task is the AppTaskStart() task. OSStart() is not supposed to return. However, it would be wise to still add code to check the returned value.

The code for AppTaskStart() is shown in Listing 4-2.

```
static  void  AppTaskStart (void *p_arg)
{
    CPU_INT32U  cpu_clk_freq;
    CPU_INT32U  cnts;
    OS_ERR      err;


    (void)&p_arg;
    BSP_Init();                                                          (1)
    CPU_Init();                                                          (2)
    cpu_clk_freq = BSP_CPU_ClkFreq();                                    (3)
    cnts         = cpu_clk_freq / (CPU_INT32U)OSCfg_TickRate_Hz;
    OS_CPU_SysTickInit(cnts);
#if OS_CFG_STAT_TASK_EN > 0u
    OSStatTaskCPUUsageInit(&err);                                        (4)
#endif
    CPU_IntDisMeasMaxCurReset();                                         (5)
    BSP_LED_Off(0);                                                      (6)
    while (DEF_TRUE) {                                                   (7)
        BSP_LED_Toggle(0);                                               (8)
        OSTimeDlyHMSM(0, 0, 0, 100,                                      (9)
                      OS_OPT_TIME_HMSM_STRICT,
                      &err);
    }
}
```

Listing 4-2 **AppTaskStart() in app.c**

L4-2(1) **AppTaskStart()** starts by calling **BSP_Init()** (See **bsp.c**) to initialize peripherals used on the µC/Eval-STM32F107. **BSP_Init()** initializes different clock sources on the STM32F107. The crystal that feeds the µC/Eval-STM32F107 runs at 25 MHz and the STM32F107's PLLs (Phase Locked Loops) and dividers are configured such that the CPU operates at 72 MHz.

L4-2(2) **CPU_Init()** is called to initialize the µC/CPU services. **CPU_Init()** initializes internal variables used to measure interrupt disable time, the time stamping mechanism, and a few other services.

L4-2(3) The Cortex-M3's system tick timer is initialized. **BSP_CPU_ClkFreq()** returns the frequency (in Hz) of the CPU, which is 72 MHz for the µC/Eval-STM32F107. This value is used to compute the reload value for the Cortex-M3's system tick timer. The computed value is passed to **OS_CPU_SysTickInit()**, which is part of the µC/OS-III port (**os_cpu_c.c**). However, this file is not provided in source form with the book. Once the system tick is initialized, the STM32F107 will receive interrupts at the rate specified by **OS_CFG_TICK_RATE_HZ** (See **os_cfg_app.c**), which in turn is assigned to **OSCfg_TickRate_Hz** in **os_cfg_app.c**. The first interrupt will occur in **1/OS_CFG_TICK_RATE_HZ** second, since the interrupt will occur only when the system tick timer reaches zero after being initialized with the computed value, **cnts**.

L4-2(4) **OSStatTaskCPUUsageInit()** is called to determine the 'capacity' of the CPU. μC/OS-III will run 'only' its internal tasks for 1/10 of a second and determine the maximum number of time the idle task loops. The number of loops is counted and placed in the variable **OSStatTaskCtr**. This value is saved in **OSStatTaskCtrMax** just before **OSStatTaskCPUUsageInit()** returns. **OSStatTaskCtrMax** is used to determine the CPU usage when other tasks are added. Specifically, as you add tasks to the application OSStatTaskCtr (which is reset every 1/10 of a second) is incremented less by the idle task because other tasks consume CPU cycles. CPU usage is determined by the following equation:

$$OSStatTaskCPU\ Usage_{(\%)} = (100 - \frac{100 \times OSStatTaskCtr}{OSStatTaskCtrMax})$$

The value of **OSStatTaskCPUUsage** can be displayed at run-time by μC/Probe. However, this simple example barely uses any CPU time and most likely CPU usage will be near 0.

Note that as of V3.03.00, **OSStatTaskCPUUsage** has a range of 0 to 10,000 to represent 0.00 to 100.00%. In other words, **OSStatTaskCPUUsage** now has a resolution of 0.01% instead of 1%.

L4-2(5) The μC/CPU module is configured (See **cpu_cfg.h**) to measure the amount of time interrupts are disabled. In fact, there are two measurements, total interrupt disable time assuming all tasks, and per-task interrupt disable time. Each task's **OS_TCB** stores the maximum interrupt disable time when the task runs. These values can be monitored by μC/Probe at run time. **CPU_IntDisMeasMaxCurReset()** initializes this measurement mechanism.

L4-2(6) **BSP_LED_Off()** is called to turn off (by passing 0) all user-accessible LEDs (red, yellow and green next to the STM32F107) on the μC/Eval-STM32F107.

L4-2(7) A typical µC/OS-III task is written as an infinite loop.

L4-2(8) `BSP_LED_Toggle()` is called to toggle all three LEDs (by passing 0) at once. You can change the code and specify 1, 2 or 3 to toggle only the green, yellow and red LED, respectively.

L4-2(9) Finally, a µC/OS-III task needs to call a µC/OS-III function that will cause the task to wait for an event to occur. In this case, the event to occur is the passage of time. `OSTimeDlyHMSM()` specifies that the calling task does not need to do anything else until 100 milliseconds expire. Since the LEDs are toggled, they will blink at a rate of 5 Hz (100 milliseconds on, 100 milliseconds off).

4-3 MONITORING VARIABLES USING µC/PROBE

Click the 'Go' button in the IAR C-Spy debugger in order to resume execution of the code.

Now start µC/Probe by locating the µC/Probe icon on your PC as shown in Figure 4-7. The icon, by the way, represents a 'box' and the 'eye' sees inside the box (which corresponds to your embedded system). In fact, at Micriµm, we like to say, "Think outside the box, but see inside with µC/Probe!"

Figure 4-7 **µC/Probe icon**

Figure 4-8 shows the initial screen when µC/Probe is first started.