4.2 实例项目是如何工作的

main()函数的代码如清单 4 - 1 所示。

```
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)
}
```

代码清单 4 - 1　app.c,main()

L4 - 1(1)　main()函数中首先调用 BSP_IntDisAll()。此函数的代码在 bsp
int.c 中实现。BSP_IntDisAll()再调用 CPU_IntDis()关闭所有中
断,不直接调用 CPU_IntDis()关中断的原因是在某些处理器上,
须通过中断控制器禁用中断。在 bsp.c 中关中断,应用代码可以
容易地移植到另一款处理器。

L4 - 1(2)　OSInit()用于初始化 μC/OS - III。通常情况下,你需要验证 OSInit
是否正确,可以通过检查 err 的值是否为 OS_ERR_NONE(即
为 0)。你可以单步(step over)调试代码,执行到 OSInit()返回后

止。将鼠标停在 err 上，它将会显示变量的值。

OSInit()创建了 4 个内部任务：空闲任务，时钟节拍任务，定时器任务和统计任务。由于 os_cfg. h 中，OS_CFG_ISR_POST_DEFERRED_EN 设定为 0，所以没有创建中断处理队列任务。

L4 - 1(3)　调用 OSTaskCreate()创建一个应用任务 AppTaskStart()。OSTaskCreate()包含 13 个参数，关于参数的描述可参考《嵌入式实时操作系统 μC/OS - III》一书中的附录 A，μC/OS - III API 参考手册部分。

AppTaskStartTCB 是任务控制块 OS_TCB。此变量在 app. c 中 local variables 部分声明。

AppTaskStartStk[]是 CPU_STKs 类型的数组，用来声明任务的堆栈。在 μC/OS - III 中，每个任务都需要单独的堆栈空间。堆栈大小在很大程度上取决于应用程序。在本例中，堆栈大小通过 app_cfg. h 中 APP_TASK_START_STK_SIZE 定义，分配了 128 个 CPU_STK 类型的元素，在 Cortex - M3 中，相当于 512 个字节(每个 CPU_STK 项是 4 个字节，见 cpu. h)，对简单的 AppTaskStart()应用，堆栈空间足够。APP_TASK_START_PRIO 决定启动任务的优先级，在 app_cfg. h 中定义。

L4 - 1(4)　OSStart()用来启动多任务调度。加上应用程序任务，μC/OS - III 将管理 5 个任务。OSStart()将启动创建的最高优先级任务。在本例中，优先级最高的任务是 AppTaskStart()任务。OSStart()不会返回，所以添加代码检查其返回值是明智的。

AppTaskStart()的代码如代码清单 4 - 2 所示。

```
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);
#endif
    CPU_IntDisMeasMaxCurReset();
    BSP_LED_Off(0);
    while (DEF_TRUE) {
        BSP_LED_Toggle(0);
        OSTimeDlyHMSM(0, 0, 0, 100,
                      OS_OPT_TIME_HMSM_STRICT,
                      &err);
```

(4) (5) (6) (7) (8) (9)

代码清单 4-2　app.c,AppTaskStart()

L4-2(1)　AppTaskStart()首先调用 BSP_Init()(见 bsp.c)来初始化 μC/Eval-STM32F107 的外设。BSP_Init()初始化 STM32F107 的时钟源。μC/Eval-STM32F107 晶振运行在 25 MHz,通过配置 STM32F107 的 PLLs(锁相环)和分频器,使得 CPU 工作在 72 MHz。

L4-2(2)　调用 CPU_Init()初始化 μC/CPU 的服务。CPU_Init()初始化测量中断禁止时间,时间戳,以及其他一些服务的内部变量。

L4-2(3)　初始化 Cortex-M3 的系统节拍定时器。BSP_CPU_ClkFreq()返回 CPU 的频率(Hz),μC/Eval-STM32F107 是 72 MHz。该值用来计算 Cortex-M3 系统时钟节拍定时器的重载值。计算结果传递给 μC/OS-III 移植代码(OS_CPU_C.C)中的 OS_CPU_SysTickInit()。一旦系统时钟初始化完成,STM32F107 将以 OS_CFG_TICK_RATE_HZ(见 os_cfg_app.c)指定的频率接收中断,在 os_cfg_app.c 中,OS_CFG_TICK_RATE_HZ 被映射到 OSCfg_TickRate_Hz。因为系统时钟节拍以计算值 cnts 初始化,并运行到 0 时才产生中断,所以第一次中断发生在 1/OS_CFG_TICK_RATE_HZ 秒。

L4-2(4)　调用 OSStatTaskCPUUsageInit()来确定 CPU 的“能力”。μC/OS-III 将运行其内部任务 1/10 s,确定空闲任务执行次数的最大值。执行次数将存放在变量 OSStatTaskCtr 中。该值在 OSStatTaskCPUUsageInit()返回之前保存到变量 OSStatTaskCtrMax 中。

当添加了其他任务时，可通过 OSStatTaskCtrMax 计算 CPU 的使用率。具体来说，当你在应用中添加任务时，空闲任务中 OSStatTaskCtr(每 1/10 s 复位)递增减缓，因为其他任务消耗了 CPU 的周期。CPU 使用率是由下列公式确定：

$$\text{OSStatTaskCPUUsage}(\%) = \left(100 - \frac{100 \times \text{OSStatTaskCtr}}{\text{OSStatTaskMax}}\right)$$

OSStatTaskCPUUsage 的值可以在运行时通过 μC/Probe 显示。然而，这个简单的例子几乎没有使用任何 CPU 时间，CPU 使用率接近 0。

注意，μC/OS - III V3.03.00 版中，OSStatTaskCPUUsage 范围从 0 到 10 000，表示 0.00 到 100.00%。换句话说，OSStatTaskCPUUsage 现在的分辨率为 0.01%，而不是 1%。

L4 - 2(5)　配置 μC/CPU 模块(见 cpu_cfg.h)来测量中断禁止时间。事实上，有两个值需要测量，所有任务的中断禁止时间和每个任务的中断禁止时间。当任务运行时，每个任务的任务控制块 OS_TCB 存储最大中断禁止时间。这些值可以在运行时通过 μC/Probe 监测。CPU_IntDisMeasMaxCurReset()初始化这个测量机制。

L4 - 2(6)　调用 BSP_LED_Off()(参数为 0)来关闭 μC/Eval - STM32F107 上所有用户可访问的 LED(靠近 STM32F107 的红色，黄色和绿色 LED)。

L4 - 2(7)　一个典型的 μC/OS - III 任务是一个无限循环。

L4 - 2(8)　调用 BSP_LED_Toggle()触发 3 个 LED(参数为 0)。可以更改代码，通过指定参数为 1，2 或 3 来分别触发绿色，黄色或红色 LED。

L4 - 2(9)　最后，任务调用等待某个事件发生的 μC/OS - III 函数。在本例中，等待的事件是时间。OSTimeDlyHMSM()指定任务延时 100 ms。由于触发了 LED，它们将以 5 Hz 的频率闪烁(间隔 100 ms)。

4.3　使用 μC/Probe 观测变量

单击 IAR C - SPY 调试器的 Go 按钮，继续执行代码。

双击 PC 上的 μC/Probe 图标，如图 4 - 7 所示，启动 μC/Probe。顺便提一

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.