第11章 运行模型：单任务/多任务

µC/GUI 从一开始就被设计成够适应不同的环境类型。它能工作在单任务和多任务应用当中，包括私有的操作系统或任何商业 RTOS，例如 embOS 或 µC/OS。

11.1 支持的运行模型

我们主要区别一下三种不同的执行模型：

单任务系统（超级循环）

所有的程序运行在一个超级循环中。通常情况下，所有软件单元被有秩序地调用。既然没用用到实时内核，软件的实时功能必须要依靠中断来完成。

µC/GUI 多任务系统：一个任务调用 µC/GUI

使用了一个实时内核（RTOS），但是其中只有一个任务调用 µC/GUI 函数。 从图形软件的观点来看，它和单任务系统中的使用是一样的。

µC/GUI 多任务系统：多个任务调用 µC/GUI

使用了一个实时内核（RTOS），其中多个任务调用 µC/GUI 函数。 这样的工作避免了软件造成的线程保护问题，该项工作通过在配置中使能多任务支持及配合内核接口函数来完成。对于普遍的的内核，内核接口函数易于使用。

11.2 单任务系统（超级循环）

描述

所有的程序运行在一个超级循环中。通常情况下，所有软件单元被有秩序地调用。没有使用实时内核，因此软件的实时功能必须要依靠中断来完成。这类系统起初用于更小的系统或如果实时行为并不是很必须的情况下。

超级循环范例（没有使用 µC/GUI）

```
void main(void)
{
    HARDWARE_Init();
/* 初始化软件单元 */
    XXX_Init( ) ;
    YYY_Init( ) ;
/* 超级循环：有秩序地调用所有软件单元 */
```

```
    while(1)
    {
    /* 执行所有的软件单元 */
        XXX_Exec();
        YYY_Exec();
    }
}
```

优点

没有使用实时内核（占用更小的 ROM 空间，只有一个任务，用于堆栈的 RAM 单元也很少），不存在 优先权/同步 问题。

缺点

超级循环程序如果超过一定的大小，可能变得很难维护。实时行为极有限，因为一个软件单元不能被其它的单元所打断（只有通过中断）。这意思是一个软件单元的反应时间依赖于这个系统中所有其它单元的执行时间。

使用 µC/GUI

关于µC/GUI 的使用，没有真的约束。如通常情况一样，GUI_Init()在你使用这个软件前必须要先调用，在这基础上，可以使用任何 API 函数。如果用到视察管理器的回调机制，一个µC/GUI 更新函数必须定期被调用。在从超级循环中调用 GUI_Exec()是一种典型做法。单元化函数例如 GUI_Delay()和 GUI_ExecDialog()，不应在循环中使用，因为它们会妨碍其它软件模块。

使用默认配置，它并不支持多任务系统使用（#define GUI_MT 0）；不需要内核接口函数。

超级循环范例（使用 µC/GUI）：

```
void main(void)
{
    HARDWARE_Init();
/* 初始化软件单元 */
    XXX _Init();
    YYY_Init();
```

```
    GUI_Init();          /* 初始化 μC/GUI * /
/* 超级循环：有秩序地调用所有的软件单元*/
    while(1)
    {
    /* 运行所有的软件单元 */
        XXX_Exec();
        YYY_Exec();
        GUI_Exec();          /* 功能性运行 μC/GUI 如更新窗口*/
    }
}
```

11.3 μC/GUI 多任务系统：一个任务调用 μC/GUI

描述

使用一个实时内核（RTOS）。用户程序被分割成不同的部分，运行在不同的任务中，具有典型不同的优先级别。通常情况下，实时的临界任务（要求某一个反应时间）具有最高级别。一个单个任务调用μC/GUI 函数，用于用户界面。这个任务通常在系统中的任务级别最低，至少是最低的级别之一（一些统计任务或简单的空闲处理的任务级别甚至可能更低）。中断可以用于软件的实时部分，但不是必须的。

优点

系统实时行为极佳。任务的实时行为只受运行在更高级别任务影响。这意思是换一个运行在低级别任务的程序单元的一点也不会影响实时行为。如果用户界面在一个低级别任务中运行，这意味者换到用户界面不会影响实时行为。这种系统使分配不同的软件单元到不同的开发组变得很容易，这能给于彼此工作非常高的独立性。

缺点

你需要一个实时内核（RTOS），这需要花费金钱，耗尽 ROM 和 RAM （用于堆栈）。另外，你要考虑到任务同步和如何从一个任务传输信息到另一个任务。

使用 μC/GUI

如果用到视察管理器的回调机制，一个μC/GUI 更新函数（典型是 GUI_Exec()，GUI_Delay()）必须定期地从调用μC/GUI 的任务中调用。 因为 μC/GUI 只被一个任务调用，

对于μC/GUI 来说，这和单任务系统中使用是一样的。

使用默认配置，它并不支持多任务系统的使用（#define GUI_MT 0）；不需要内核接口函数。你可以使用任何实时内核，包括商业的或私有的。

11.4 μC/GUI 多任务系统：多个任务调用 μC/GUI

描述

使用一个实时内核。用户程序被分割成不同的部分，运行在不同的任务中，具有典型不同的优先级别。通常情况下，实时的临界任务（要求某一个反应时间）具有最高级别。多个任务用于用户界面，调用μC/GUI 函数。这些任务在系统中具有典型的低级别，因此它们不会影响系统的实时行为。

中断可以用于软件的实时部分，但不是必须的。

优点

系统实时行为极佳。任务的实时行为只受运行在更高级别任务影响。这意思是换一个运行在低级别任务的程序单元的一点也不会影响实时行为。如果用户界面在一个低级别任务中运行，这意味者换到用户界面不会影响实时行为。这种系统使分配不同的软件单元到不同的开发组变得很容易，这能给于彼此工作非常高的独立性。

缺点

你需要一个实时内核（RTOS），这需要花费金钱，耗尽 ROM 和 RAM （用于堆栈）。另外，你要考虑到任务同步和如何从一个任务传输信息到另一个任务。

使用 μC/GUI

如果用到视察管理器的回调机制，一个μC/GUI 更新函数（典型是 GUI_Exec()，GUI_Delay()）必须定期地从一个或多个调用μC/GUI 的任务中调用。

默认配置并不支持多任务系统的使用（#define GUI_MT 0），在此不能使用。在配置中，需要启用多任务支持，定义调用μC/GUI 的任务的最大数量(引用自 GUIConf.h)：

```
#define GUI_MT 1              // 启用多任务支持
#define GUI_MAX_TASK 5        // 能调用μC/GUI 的任务的最大数量
```

需要用到内核接口函数，需要与使用的内核相匹配。你可以使用任何实时内核，包括商业的或私有的。在下面的章节，会对宏和函数二者进行论述。

建议

（1）仅仅从一个任务调用μC/GUI 更新函数（即 GUI_Exec()，GUI_Delay()）。这对保持程序结构清晰有帮助。如果你的系统有足够的 RAM，专门使用一个任务（最低级别）更新μC/GUI。该任务将不断地调用 GUI_Exec()，不做其它事情，就象下面例子显示的一样。

（2）保持你的实时任务（决定你的系统行为，关于 I/O，接口，网络等等）与调用μC/GUI 的任务分开。这对保证获得最佳的实时性能有很大帮助。

（3）如果可能的话，你的用户界面只使用一个任务。这利于保持程序结构简洁，易于调试（但是，这不是必需的，在一些系统也可能是不合适的。）

范例

该引用显示专门的 dedicated μC/GUI 更新函数。它从范例 MT_Multitasking 中拿来，这个范例包括在随μC/GUI 发布的范例当中：

```
/*********************************************************************
*                                GUI 背景处理                          *
*********************************************************************/

/*  该任务进行背景处理。主要工作是更新有效窗口，其它的事情，例如测试鼠标或
*   触摸屏输入也可以处理                                               */

void GUI_Task(void)
{
    while(1)
    {
        GUI_Exec();             /* 做背景工作……更新窗口等等 */
        GUI_X_ExecIdle();       /* 剩下暂时不做什么事情……空闲处理 */
    }
}
```

11.5 多任务支持的 GUI 配置宏

下表显示了用于一个多个任务调用μC/GUI 多任务系统的配置宏：

类型	宏	默认值	说明
N	GUI_MAXTASK	4	当多任务支持启用时（如下），定义调用μC/GUI 最大任务数量。
B	GUI_OS	0	激活多任务支持的启用。

GUI_MAXTASK

描述

定义调用μC/GUI 访问显示屏的最大任务的数量。

类型

数值

附加信息

该功能只有在 GUI_OS 激活情况下才相应有效。

GUI_OS

描述

通过激活 GUITask 组件启用多任务支持。

类型

二进制开关

0: 停止，多任务支持禁止（默认值）
1: 激活，多任务支持启用

11.6 内核接口函数 API

一个 RTOS 通常提供一个机制，称为资源旗语。在它的里面，使用一个特定资源的一个

任务在实际使用这个资源之前要声明这个资源。显示屏是一个需要和资源旗语一起被保护的资源的例子。μC/GUI 使用宏 GUI_USE 在访问显示屏之前或使用一个临界内部数据之前调用函数 GUI_Use()。 类似的方法，它在访问显示屏之后或使用一个临界内部数据之后调用函数 GUI_Unuse()。这在模块 GUITask.c 中实现。

GUITask.c 依次使用下表所示的 GUI 内核接口函数。这些函数的前缀为 GUI_X_，因为它们是高层函数（与硬件相关）。它们必须与所使用的实时内核相匹配，以构造μC/GUI 任务（或线程）保护。详细的函数描述在后面，还有范例说明如何与不同的内核匹配。

函数	说明
GUI_X_InitOS()	初始化内核接口模块（建立一个资源“旗语/互斥”）。
GUI_X_GetTaskId()	返回一个当前任务（线程）的唯一的 32 位标识符。
GUI_X_Lock()	锁定 GUI（阻塞资源“旗语/互斥”）。
GUI_X_Unlock()	解锁 GUI（解锁资源“旗语/互斥”）。

GUI_X_InitOS()

描述

建立资源旗语或互斥(体)，最典型是由 GUI_X_Lock()和 GUI_X_Unlock()使用。

函数原型

```
void GUI_X_InitOS(void)
```

GUI_X_GetTaskID()

描述

返回当前任务的唯一 ID。

函数原型

```
U32 GUI_X_GetTaskID(void);
```

返回值

当前任务的 ID 是一个 32 位整数。

附加信息

与实时操作系统一起使用。

返回哪个数值没有关系，只要对于每个使用μC/GUI API 的任务/线程来说，它是唯一的，以及只要对于每个特定的线程来说，该数值总是相同的。

GUI_X_Lock()

描述

锁定 GUI.

函数原型

```
void GUI_X_Lock(void);
```

附加信息

在访问显示屏之前或在使用临界内部数据结构之前，该函数被 GUI 所调用。它从插入相同的临界区处阻塞其它线程，直到调用 GUI_X_Unlock()函数，而该插入使用一个资源“旗语/互斥”完成。当使用一个实时操作系统时，你通常必需增加一个计算资源旗语。

GUI_X_Unlock()

描述

解锁 GUI.

函数原型

```
void GUI_X_Unlock(void);
```

附加信息

这个函数在访问显示屏后或使用一个临界内部数据以后被 GUI 调用。

当使用一个实时操作系统时，你通常必须消耗一个计算资源旗语。

范例

用于μC/OS-II 的内核接口函数

下面范例展示与 uC/OS-II 相匹配的四个函数的（引用自文件 GUI_X_uCOS-II.c）：

```
#include “INCLUDES.H”
static OS_EVENT * DispSem;

U32 GUI_X_GetTaskId(void)
{
    return((U32)(OSTCBCur->OSTCBPrio));
}
void GUI_X_InitOS(void)
{
    DispSem = OSSemCreate(1);
}
void GUI_X_Unlock(void)
{
    OSSemPost(DispSem);
}
void GUI_X_Lock(void)
{
    INT8U err;
    OSSemPend(DispSem, 0, &err);
}
```

Win32 的内核接口函数

以下内容引用自对μC/GUI 的 Win32 仿真。（当使用μC/GUI 仿真时，没有必要加入这些函数，因为它已经放在库里。）

注意：为了清晰起见，清除代码被省略了。

```
/*********************************************************************
*                    μC/GUI 多任务接口，用于 Win32                   *
*********************************************************************/
```

以下部分由 4 个 Win32 用于构造μC/GUI 线程保护的函数组成：

```
static HANDLE hMutex;
void GUI_X_InitOS(void)
{
```

```
    hMutex = CreateMutex(NULL, 0, "μC/GUISim - Mutex");
}
unsigned int GUI_X_GetTaskId(void)
{
    return GetCurrentThreadId();
}
void GUI_X_Lock(void)
{
    WaitForSingleObject(hMutex, INFINITE);
}
void GUI_X_Unlock(void)
{
    ReleaseMutex(hMutex);
}
```