Micriµm

Empowering Embedded Systems

µC/OS-II and Microsoft Windows XP

(Using Visual Studio 6.0 or 2003)

Application Note

AN-1032

www.Micrium.com

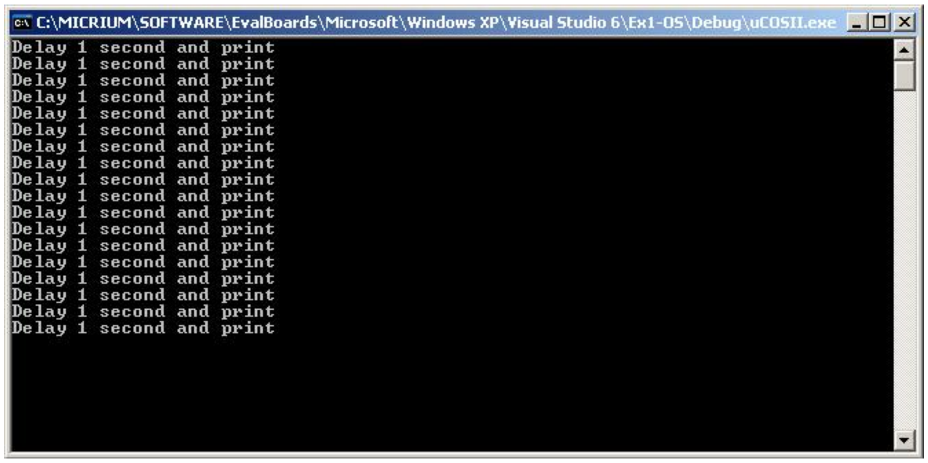
Table Of Contents

1.00 Introduction

This document shows example code for running μC/OS-II within Microsoft Windows XP. The example initializes and starts μC/OS-II so that basic application testing may be performed prior to receiving final operating hardware.

This example uses a μC/OS-II port designed to run on top of Windows XP by means of Win32 system API calls. These calls are responsible for creating several Win32 tasks that drive `OSCtxSw()` and `OSTickISR()` as well as other μC/OS-II components.

The following example has been built and run with both Visual Studio 6.0 and Visual Studio .NET 2003. When executed, it loads a DOS window with μC/OS-II running. The example has one application task that displays "Delay 1 second and print" continuously to the screen.


C:\MICRIUM\SOFTWARE\EvalBoards\Microsoft\Windows XP\Visual Studio 6\Ex1-OS\Debug\uCOSII.exe

```
Delay 1 second and print
Delay 1 second and print
Delay 1 second and print
Delay 1 second and print
Delay 1 second and print
Delay 1 second and print
Delay 1 second and print
Delay 1 second and print
Delay 1 second and print
Delay 1 second and print
Delay 1 second and print
Delay 1 second and print
Delay 1 second and print
Delay 1 second and print
Delay 1 second and print
Delay 1 second and print
Delay 1 second and print
Delay 1 second and print
```

Figure 1-1, μC/OS-II for Windows XP

1.01 Windows XP / μC/OS-II Interface

Figure 1-2 shows the relationship between the user application, **μC/OS-II** and the underlying Windows system.

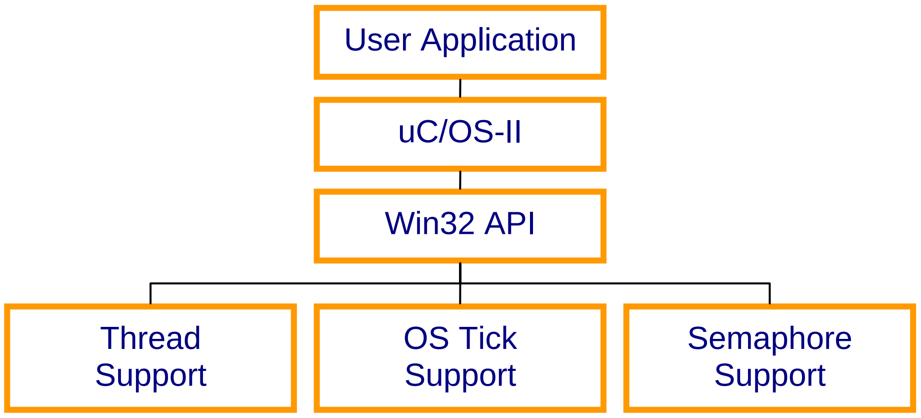


Figure 1-2, Application / μC/OS-II / Windows Relationship

As a result of this hierarchy, μC/OS-II tasks are really Windows threads and their stacks are converted to Windows thread stacks. The system ticker is driven by the high resolution multi-media timer if `WIN_MM_TICK` is defined in os_cpu.h. Otherwise it is driven by sleep(), the system coarse timer. A more realistic real-time effect can be achieved by using the multi-media timer since it has finer granularity (1ms) than the system coarse timer.

Critical sections are implemented using the Win32 API.

Fortunately, the underlying architecture is transparent to the application programmer and all μC/OS-II application code can utilize various features using tradiational documented μC/OS-II function calls.

Since μC/OS-II is an infinite loop by nature, it should be noted that the processor utilization under windows will remain close to 100% while μC/OS-II is running. This is normal operating behavior for infinite loop consol based programs under Windows.

1.02 Directories and Files

The code and documentation for 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 2.80.

`\Micrium\Software\uCOS-II\Ports\Win32`

This directory contains the standard processor specific files for a µC/OS-II port assuming the Visual Studio 6.0 or .NET 2003 toolchain. In fact, these files could easily be modified to work with other toolchains. Specifically, this directory contains the following files:

```
os_cpu.h
os_cpu_c.c
os_trace.c
os_trace_c
```

As mentioned above, you can either compile the code under Visual Studio 6.0 or Visual Studio .NET 2003. Project files are included for both.

Application Code:

`\Micrium\Software\EvalBoards\Microsoft\`
`     Windows XP\Visual Studio 6\Ex1-OS\`

`Or`

`\Micrium\Software\EvalBoards\Microsoft\`
`     Windows XP\Visual Studio 2003\Ex1-OS\`

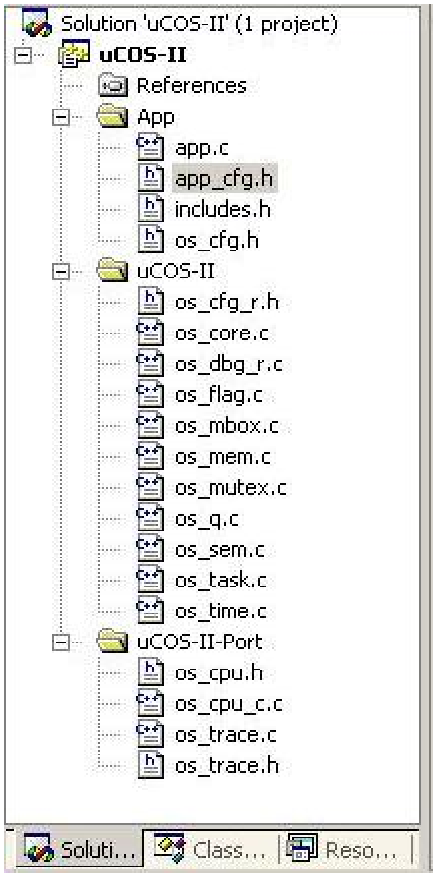
This directory is the directory that contains the source code for an example running on the Windows XP. It assumes the presense of µC/OS-II.
This directory contains:

```
app.c
app_cfg.h
includes.h
os_cfg.h
```

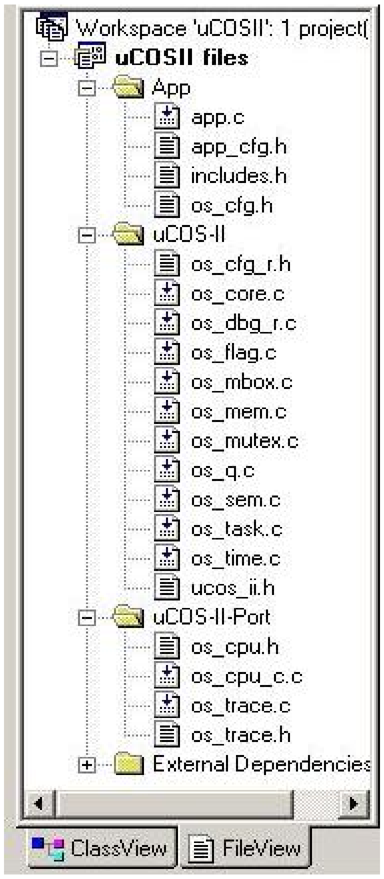
`app.c` contains the test code, `app_cfg.h` contains application specific configuration information such as task priorities and stack sizes. `includes.h` contains a master include file used by the application, `os_cfg.h` is the µC/OS-II configuration file.

1.03 Visual Studio 6.0 / .NET 2003

We used the Visual Studio .NET 2003 (left) and Visual Studio 6.0 (right) to build this example. You can of course use μC/OS-II with other tools. Figure 1-2 shows the project tree for both IDE's.



```
Solution 'uCOS-II' (1 project)
⊟ uCOS-II
    References
  ⊟ App
      app.c
      app_cfg.h
      includes.h
      os_cfg.h
  ⊟ uCOS-II
      os_cfg_r.h
      os_core.c
      os_dbg_r.c
      os_flag.c
      os_mbox.c
      os_mem.c
      os_mutex.c
      os_q.c
      os_sem.c
      os_task.c
      os_time.c
  ⊟ uCOS-II-Port
      os_cpu.h
      os_cpu_c.c
      os_trace.c
      os_trace.h
Soluti... | Class... | Reso...
```



```
Workspace 'uCOSII': 1 project(
⊟ uCOSII files
  ⊟ App
      app.c
      app_cfg.h
      includes.h
      os_cfg.h
  ⊟ uCOS-II
      os_cfg_r.h
      os_core.c
      os_dbg_r.c
      os_flag.c
      os_mbox.c
      os_mem.c
      os_mutex.c
      os_q.c
      os_sem.c
      os_task.c
      os_time.c
      ucos_ii.h
  ⊟ uCOS-II-Port
      os_cpu.h
      os_cpu_c.c
      os_trace.c
      os_trace.h
  ⊞ External Dependencies
ClassView | FileView
```

Figure 1-3, Visual Studio Projects

Note: If you choose to re-create the project, remember to add additional source directory paths in the project/compiler options for the \Micrium\Software\uCOS-II\Source\ and \Micrium\Software\uCOS-II\Ports\Win32\ source and header files.

2.00 Test Code, app.c

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

Listing 2-1, main()

```
void main(int argc, char *argv[])                                    (1)
{
    INT8U  err;

#if 0
    BSP_IntDisAll();                                                 (2)
#endif

    OSInit();                                                        (3)

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

#if OS_TASK_NAME_SIZE > 11
    OSTaskNameSet(APP_TASK_START_PRIO, (INT8U *)"Start Task", &err);  (5)
#endif

#if OS_TASK_NAME_SIZE > 14
    OSTaskNameSet(OS_IDLE_PRIO, (INT8U *)"uC/OS-II Idle", &err);
#endif

#if (OS_TASK_NAME_SIZE > 14) && (OS_TASK_STAT_EN > 0)
    OSTaskNameSet(OS_STAT_PRIO, "uC/OS-II Stat", &err);
#endif

    OSStart();                                                       (6)
}
```

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

L2-1(2) This block is excluded from compilation since the example is not running on an embedded target. If it was, it would start off by calling a BSP function to disable all interrupts. This ensures that initialization would not get interrupted in case a 'warm restart' is performed.

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`). However, in this port, OS_TASK_STAT_EN is left as 0, disabled.

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 as well as the two internal µC/OS-II tasks.

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.

Listing 2-2, AppTaskStart()

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

#if 0
    BSP_Init();                                                     (1)
#endif

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

    while (TRUE)                                                    (3)
    {
        OS_Printf("Delay 1 second and print\n");                    (4)
        OSTimeDlyHMSM(0, 0, 1, 0);                                  (5)
    }
}
```

L2-2(1) `BSP_Init()` is excluded from compilation since this port runs under Windows and does not have additional BSP support at this time. However, if this were running on an embedded target, this would be a good place to initialize BSP.

L2-2(2) `OSStatInit()` is used to initialize µC/OS-II's statistic task. This only occurs if you enable the statistic task by setting `OS_TASK_STAT_EN` to `1` in `OS_CFG.H`. The statistic task measures overall CPU usage (expressed as a percentage) and also, performs stack checking for all the tasks that have been created with `OSTaskCreateExt()` with the stack checking option set. In this port, support for OSStat is disabled and therefore the block to initialize this code is excluded from compilation.

L2-2(3) `All` µC/OS-II tasks body's must contain an infinite loop. However, in the <u>first</u> created task (in this case AppTaskStart), it's common to call OSTaskCreateExt() and create additional application tasks just before entering the infinite loop.

L2-2(4) `OS_Printf()` provides a thread safe version of printf(). In this example, OS_Printf() is called from the only application task created and is used to generate user recognizable output in order to indicate that µC/OS-II is running.

L2-2(5) All µC/OS-II tasks are required to wait for an event or terminate from within their infinite loop at some point during execution In this case, `OSTimeDlyHMSM()` is used to generate a delay of 1 second between each call to OS_Printf().

2.01 Test 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

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

2.02 Test 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.

2.03 Test 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.

References

μC/OS-II, The Real-Time Kernel, 2nd Edition
Jean J. Labrosse
R&D Technical Books, 2002
ISBN 1-57820-103-9

Embedded Systems Building Blocks
Jean J. Labrosse
R&D Technical Books, 2000
ISBN 0-87930-604-1

Contacts

IAR Systems
Century Plaza
1065 E. Hillsdale Blvd
Foster City, CA 94404
USA
+1 650 287 4250
+1 650 287 4253 (FAX)
e-mail: Info@IAR.com
WEB : www.IAR.com

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

Validated Software
Lafayette Business Park
2590 Trailridge Drive East, Suite 102
Lafayette, CO 80026
USA
+1 303 531 5290
+1 720 890 4700 (FAX)
e-mail: Sales@ValidatedSoftware.com
WEB: www.ValidatedSoftware.com