Micriμm, Inc.

μC/OS-View

V1.10

User's Manual

www.Micrium.com

1.00 Introduction

µC/OS-View is a combination of a Microsoft Windows application program and code that resides in your target system (i.e. your product). The Windows application connects with your system via an RS-232C serial port as shown in Figure 1-1. The Windows application allows you to 'View' the status of your tasks which are managed by µC/OS-II.

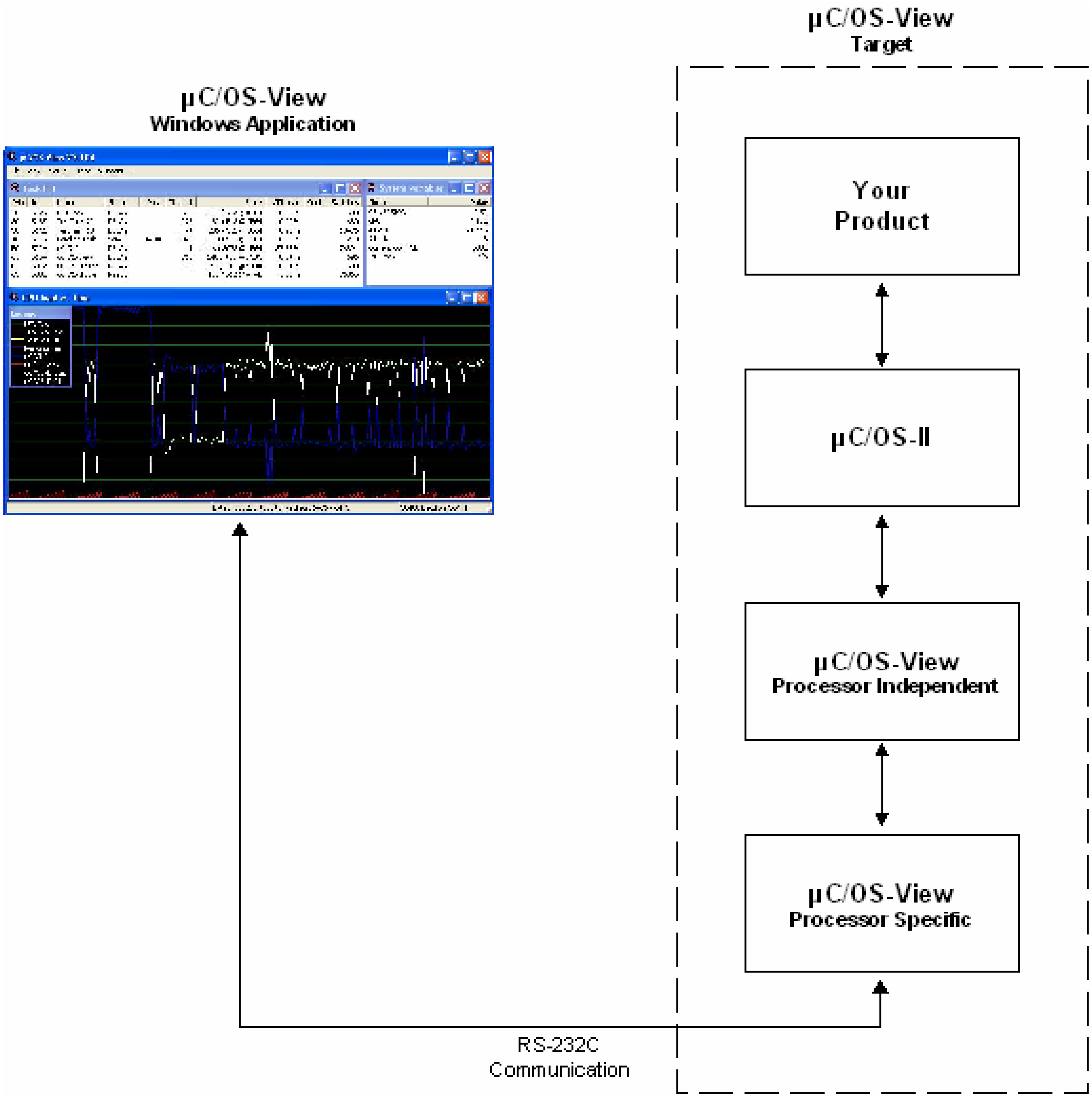


Figure 1-1, PC to target system via an RS-232C serial port.

µC/OS-View allows you to view the following information from a µC/OS-II based product:

- The address of the TCB of each task
- The name of each task
- The status (Ready, delayed, waiting on event) of each task
- The number of ticks remaining for a timeout or if a task is delayed
- The amount of stack space used and left for each task
- The percentage of CPU time each task relative to all the tasks
- The number of times each task has been 'switched-in'
- The execution profile of each task
- More.

µC/OS-View V1.10 also allows you to:

- 'Suspend' the tick interrupt from decrementing delays and timeouts of tasks. However, you can 'step' on tick at a time by pressing the *F8* key from the Windows application. The *F6* key cancels this mode, the *F7* key enables this mode and the *F8* key enables one tick to be processed.
- Pass keystrokes to you application from the 'Terminal' window. In other words, you can now send commands to your product from the Windows application. You determine the command structure.
- Output ASCII strings from the target to the 'Terminal' window. These ASCII strings are target specific and thus you can define those specific to your product.

1.01 Revision History

1.01.01 µC/OS-View V1.10

µC/OS-View (V1.00) has been revised to support new features introduced in µC/OS-II V2.61 (and higher). Specifically:

- **µC/OS-View** now support the stepping feature of the viewer. In other words, pressing the F7 key on the viewer pauses the µC/OS-II ticker. Pressing the F8 key allows one tick to be executed and pressing the F6 key resumes normal operation of the ticker. Note that `OSTimeTickHook()` is still called at the tick rate in case your application has time critical needs.
- **µC/OS-View** no longer needs to use `OSTCBExtPtr`, µC/OS-II's `OS_TCB` extension pointer. This allows you to extend an `OS_TCB` for your own use.
- µC/OS-II V2.61 now allows you to assign a name to a task and thus, this feature is no longer part of **µC/OS-View**.
- There is no need for a **µC/OS-View** task anymore since stack usage statistics are now determined by µC/OS-II's statistic task. This means that **µC/OS-View** doesn't 'eat up' a task and stack space. Also, statistics (variables) allocated by **µC/OS-View** are no longer needed since those have been placed in µC/OS-II's `OS_TCB`.

2.00 µC/OS-View Windows Application

Figure 2-1 shows **µC/OS-View**’s four main display areas which are described next:

1. Task List
2. System Variables
3. Terminal Window
4. CPU load vs Time

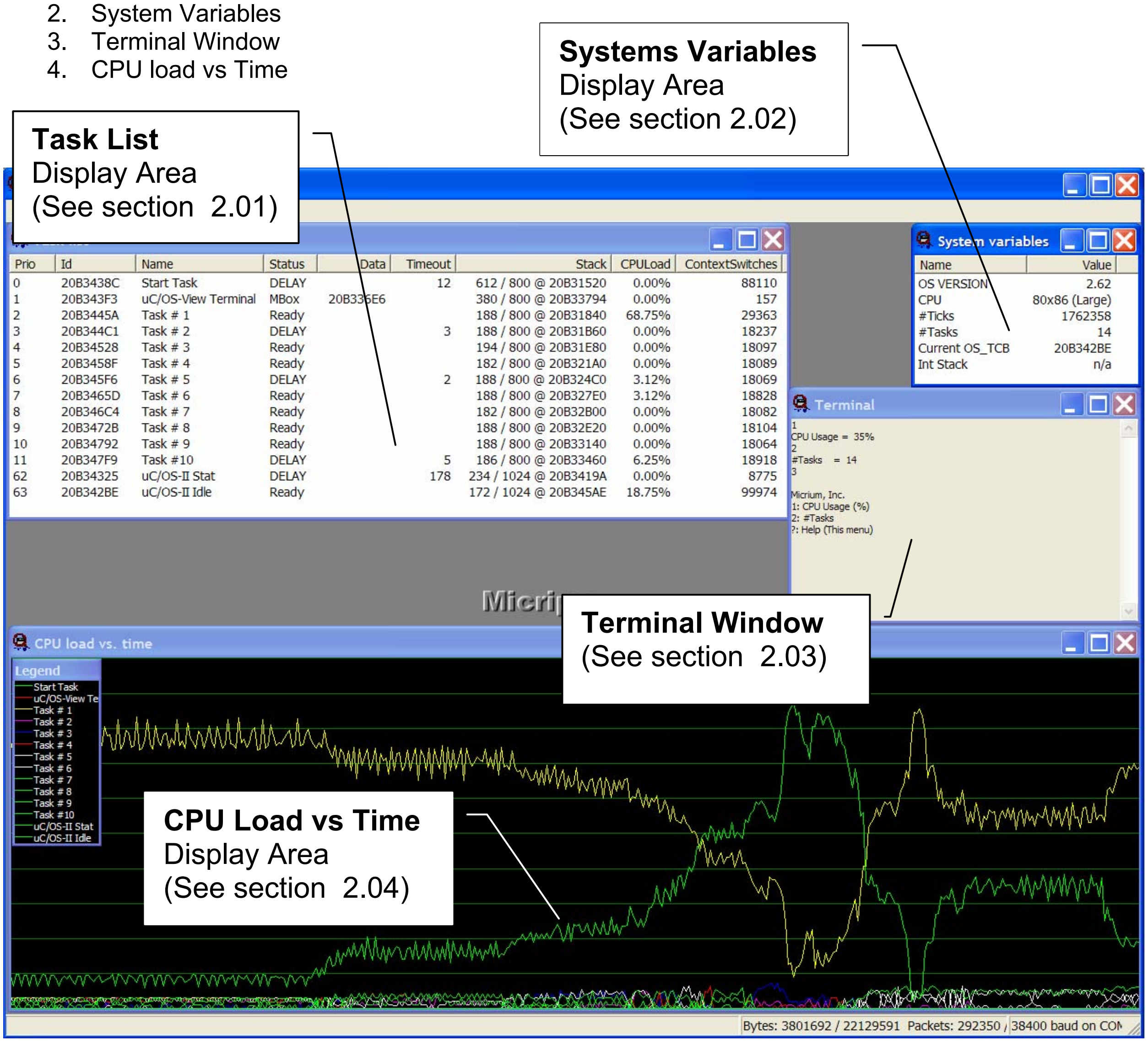


Figure 2-1, Windows application view.

2.01 Task List

The task list shows all the tasks in your application and displays information about those tasks. Specifically:

Prio Indicates the task priority of each task. Note that entries in the task list are always sorted by task priority.

Id Is a unique ID for each task. In fact, this field contains the address of the `OS_TCB` of the task.

Name Is the name of the task. Assigning a name to a task is a new feature that was added to µC/OS-II V2.6x. The name is actually stored in the target system and sent to the Windows application along with the other information about your tasks (see section 3.03 for details on how to set the task names). The number of characters you can use for a task name depends on the µC/OS-II configuration constant `OS_TASK_NAME_SIZE` found in `OS_CFG.H`.

Status This field indicates the status of each task. A task can have the following statuses:

- **Ready** The task is ready to run.
- **DELAY** A task is waiting for time to expire. The Timeout (described later) field indicates how many ticks remains before the task is ready to run.
- **MBOX** The task is waiting on a message mailbox. The Data field shows the address (in Hexadecimal) of the Event Control Block associated with the mailbox. The Timeout field indicates the amount of time (in ticks) that the task is willing to wait for the mailbox to be posted. An empty timeout indicates that the task will wait forever for the mailbox to be posted.
- **Q** The task is waiting on a message queue. The Data field shows the address (in Hexadecimal) of the Event Control Block associated with the queue. The Timeout field indicates the amount of time (in ticks) that the task is willing to wait for the queue to be posted. An empty timeout indicates that the task will wait forever for the queue to be posted.
- **MUTEX** The task is waiting on a mutex. The Data field shows the address (in Hexadecimal) of the Event Control Block associated with the mutex. The Timeout field indicates the amount of time (in ticks) that the task is willing to wait for the mutex to be signaled (i.e. released). An empty timeout indicates that the task will wait forever for the mutex.
- **FLAG** The task is waiting on an event flag group. The Data field shows the address (in Hexadecimal) of the event flag group. The Timeout field indicates the amount of time (in ticks) that the task is willing to wait for the desired flags to be set or cleared. An empty timeout indicates that the task will wait forever for the desired flag(s).

Data See Status.

Timeout See Status.

Stack — This field contains three pieces of information: the amount of stack space used (in bytes), the total stack space available to the task (in bytes) and the base address of the task's stack. If the stack grows downwards (from high to low memory) then this field contains the highest memory location of your stack. If the stack grows upwards (from low to high memory) then this field contains the lowest memory location of your stack. This field is useful to see just how much stack space is left for each task.

CPULoad — This field indicates the amount of CPU time consumed by each task expressed as a percentage. Of course, a higher number indicates that your task consumes a high amount of the CPU's time.

ContextSwitches — Contains the total number of context switches to the task since your application started.

2.02 System Variables

The *Systems Variables* area contains general information about your target. Specifically:

OS_VERSION — This field indicates the version of µC/OS-II running in your target system. This field will not change at run-time.

CPU — Indicates the type of CPU in your target system. This field will not change at run-time.

#Ticks — Contains the value of µC/OS-II's global variable `OSTime` which indicates the number of ticks since power up or, since you last called `OSTimeSet()`.

#Tasks — Contains the total number of tasks in your target application.

Current OS_TCB — Contains the address of the current task's `OS_TCB`.

IntStack — Indicates the base address of the interrupt stack if a separate stack area is reserved for interrupts.

2.03 Terminal Window

The *Terminal Window* area allows the Windows application (i.e. the Viewer) to communicate you're your target. Specifically, the Terminal Window allows you to send keystrokes (that you type from on the Windows PC) to your target system. Your target system can then process these keystrokes and responds back with ASCII strings that are displayed on the Terminal Window. Your application actually determines what to do with the keystrokes. The terminal window is described later.

2.04 CPU load vs time

This area allows you to 'see' the execution profile of your tasks – the percentage CPU usage of tasks as a function of time. This area is interesting since it gives you an idea about where your CPU is spending it's time.

You can setup **µC/OS-View** to display either all your tasks at the same time or select to display up to 5 tasks on the graph. To change this option, simply click on the *Setup* menu item and then click on the *CPU View* tab as shown in Figure 2-2.

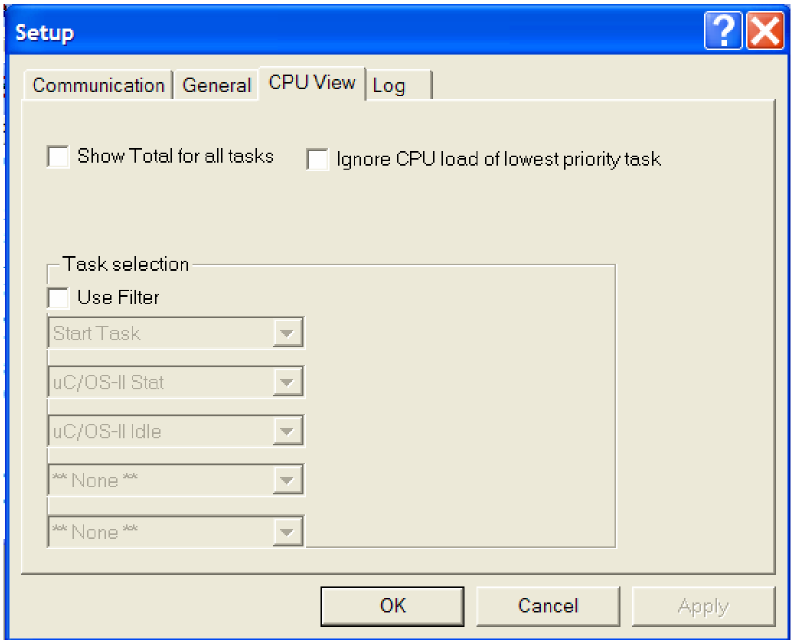


Figure 2-2, CPU View Options

As shown in Figure 2-3, if you click on the *Task Selection*'s *Use Filter* check box, you can actually specify which of up to 5 tasks you can show on the graph. You simply select the tasks that you want to view by name.

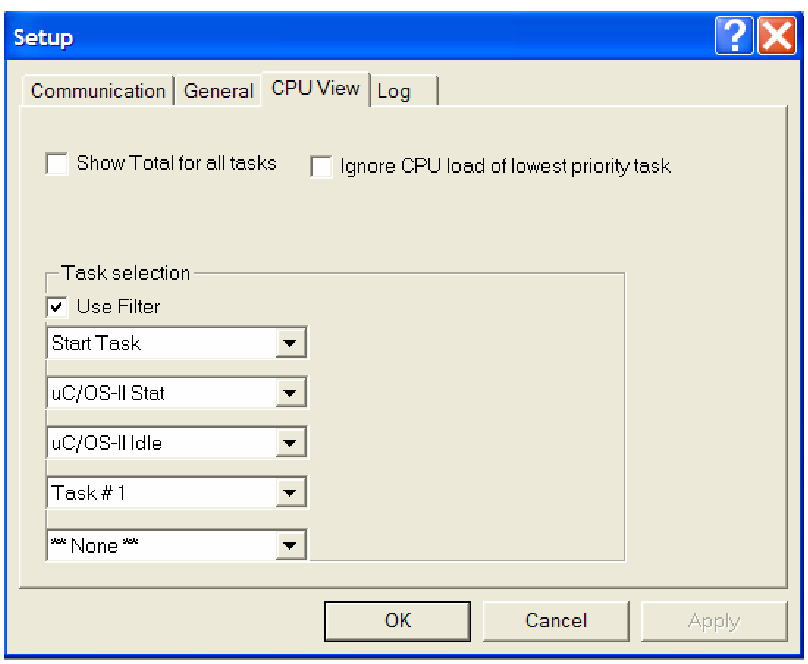


Figure 2-3, Selecting to graph only four tasks.

Figure 2-4 shows the profile of the four tasks selected.

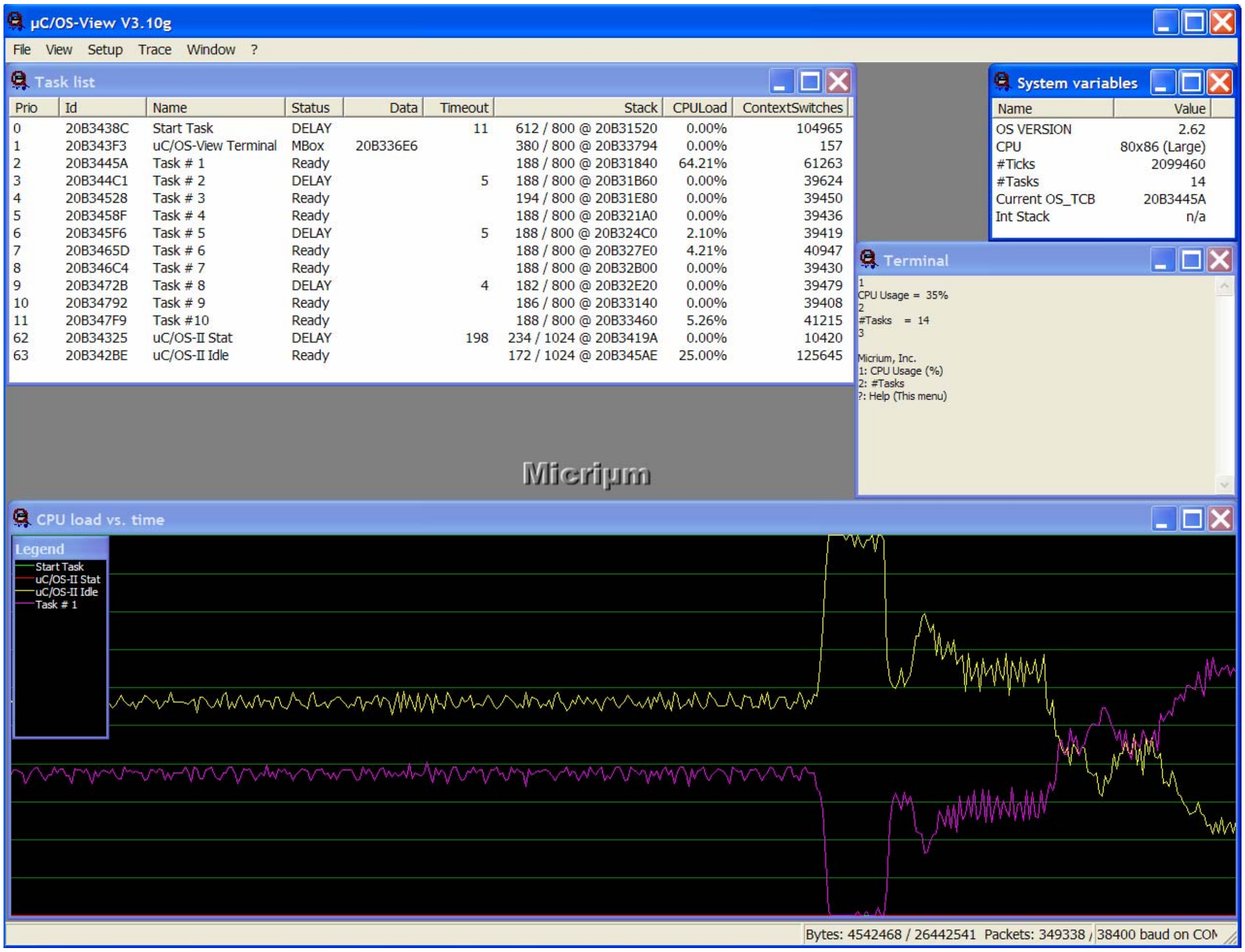


Figure 2-4, Profile of the four tasks selected.

3.00 µC/OS-II Target Resident Software

To run µC/OS-II, your target system needs to have an RS-232C port and include target resident software that communicates with the Windows application. The target resident software is found in five files that starts with the prefix `OS_VIEW`:

File name	Description
`OS_VIEW.C`	Target independent code
`OS_VIEW.H`	Target independent header file
`OS_VIEWc.C`	Target specific C source file (i.e. port) which changes based on the target CPU used.
`OS_VIEWc.H`	Target specific C header file
`OS_VIEWa.ASM`	Target specific assembly language file

Table 3-1, µC/OS-View source files.

You should not have to change the target independent code (i.e. `OS_VIEW.C` and `OS_VIEW.H`).

3.01 Restrictions

When you compile the target resident code with your µC/OS-II application, you need to abide by the following restrictions:

1) You MUST use µC/OS-II V2.61 or higher.
2) You need to change the port files (see `OS_CPU_C.C`) such that:
   - `OSTaskCreateHook()` CALLS `OSView_TaskCreateHook()` and passes it `ptcb`.
   - `OSTaskSwHook()` CALLS `OSView_TaskSwHook()`.
   - `OSTimeTickHook()` CALLS `OSView_TickHook()`.
3) Your `OS_CFG.H` MUST set the following constants as follows:
   - `OS_TASK_CREATE_EXT_EN` to `1` Enable `OSTaskCreateExt()`
   - `OS_TASK_STAT_EN` to `1` Enable the statistic task
   - `OS_TASK_STAT_STK_CHK_EN` to `1` Enable stack checking by the statistic task.
   - `OS_TASK_NAME_SIZE` to `16` (or higher) Enable task names
   - `OS_TASK_PROFILE_EN` to `1` Enable the profiling variables in `OS_TCB`.
4) You MUST create ALL your tasks using `OSTaskCreateExt()` instead of `OSTaskCreate()`.
5) You MUST call `OSTaskNameSet()` AFTER you call `OSTaskCreateExt()` (see below) if you want to see the task names appear in the Windows application. If you don't assign names to your tasks, they will be displayed as "`?`".
6) You need to dedicate an asynchronous serial port (also known as a UART) to **µC/OS-View**.
7) You cannot have more than 63 tasks (including the idle task). In other words, **µC/OS-View** doesn't currently allow you to monitor an application that has the µC/OS-II maximum number of task (i.e. 64 tasks).
8) If you want to use the Terminal Window feature, you need to declare a *Callback* function that will handle characters sent from the Windows application (See section 3.04).
9) You MUST call `OSView_Init()` AFTER calling `OSStatInit()`.

3.02 Initializing µC/OS-View

In order to use µC/OS-View, you need to initialize the target resident software. The function `OSView_Init()` is provided to initialize µC/OS-View and should be called AFTER you called `OSStatInit()` (which you must do).

3.03 Creating tasks

All your µC/OS-II tasks need to be created with `OSTaskCreateExt()` because the target resident software makes use of additional variables only available with `OSTaskCreateExt()`.

L3-1(1) Make sure you use `OSTaskCreateExt()`.

L3-1(2) You should pass a NULL pointer as the `pext` argument if you are not using the TCB extension.

L3-1(3) You must also specify that you want to enable stack checking for the task as well as clear the stack upon task creation.

L3-1(4) You must call `OSTaskNameSet()` and pass this function the task priority as well as a name you would like to give to the task. The name can contain spaces and some punctuation marks. The size (number of characters) of the name MUST be less than `OS_TASK_NAME_SIZE-1`.

```
    INT8U  err;


    OSTaskCreateExt(YourTask,                                                  (1)
                    (void *)0,
                    &YourTaskStk[YOUR_TASK_STK_SIZE - 1],
                    your_task_prio,
                    your_task_prio,
                    &YourTaskStk[0],
                    YOUR_TASK_STK_SIZE,
                    (void *)0,                                                 (2)
                    OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR);                (3)
    OSTaskNameSet(prio, "Your Task Name!", &err);                              (4)
```

Listing 3-1, Creating a task when using µC/OS-View.

3.04 Terminal Window

Support for a 'Terminal Window' has been added in V1.10. The terminal window basically allows you to 'send' characters (i.e. keystrokes) that you type on the Windows application's keyboard to your target. Your target can then 'interpret' these characters and perform actions that **you** determine for your target. You target can also send ASCII strings back to the terminal window to provide you with feedback.

Figure 3-1 shows what happens when you select the Terminal Window and type at the keyboard.

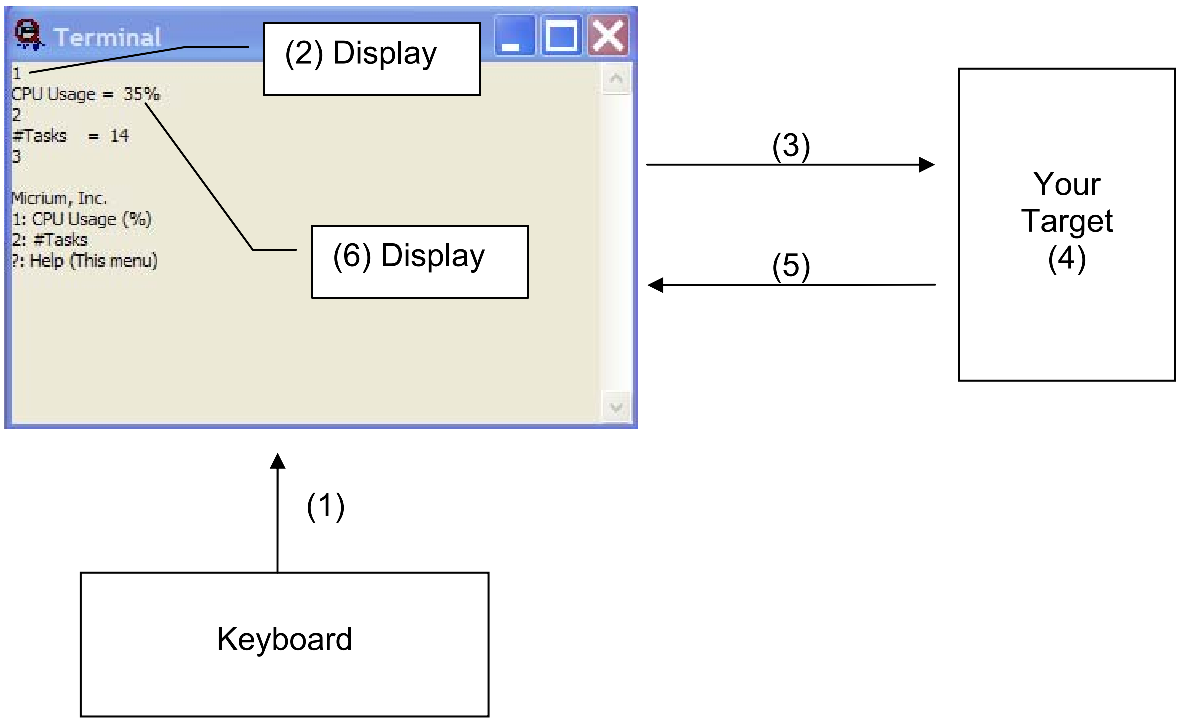


Figure 3-1, Using the Terminal Window.

F3-1(1) You press the character `'1'` at the keyboard.

F3-1(2) µC/OS-View displays the character typed on the terminal window.

F3-1(3) µC/OS-View sends the character typed to your target.

F3-1(4) Your target gets a 'notification' that a character has arrived from µC/OS-View and performs some action that YOU determine. In this case, I decided that receiving a `'1'` meant that you wanted to know what the current CPU usage is.

F3-1(5) The target then formats a reply string and send it to the terminal window.

F3-1(6) µC/OS-View sees that an ASCII string is received from the target and displays it on the terminal window.

Figure 3-1 shows that I typed other characters. Specifically, typing a '2' results in the target replying back with the number of tasks created. Typing a '3' resulted in displaying this simple menu. In fact, the code I have only recognizes either a '1' or a '2' and, any other key displays the menu.

In order for the target to recognize the keys being sent from the terminal window, you need to install a 'callback' function by calling `OSView_TerminalRxSetCallback()`. In other words, when a character is received, the µC/OS-View's target resident code calls the function that you install. The code below shows how that's done:

```
OSView_TerminalRxSetCallback(TestTerminalCallback);
```

and the code for the callback looks as follows:

```
void  TestTerminalCallback (INT8U data)
{
    OSMboxPost(TestTerminalMbox, (char *)data);
}
```

µC/OS-View will call your function and pass it the character received from the terminal window. You need to know that the callback (`TestTerminalCallback()` in the case above) is called by an ISR and thus, you should post the character received to a task (as shown above), and let the task process the received character (as shown in the code below). Of course, the mailbox needs to be created before it's actually used by the callback. Also, YOUR callback function needs to be declared as shown above.

You can create your own commands for your target system. Specifically, you could have code that initiates special tests on your hardware in response to certain keystrokes, you could create commands that changes the operating mode of your target, you could create commands that displays the contents of variables and memory locations and more.

```
static  void  TestTerminalTask (void *pdata)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    char       s[100];
    char       key;
    INT8U      err;


    pdata = pdata;
    while (TRUE) {
        key = (char)OSMboxPend(TestTerminalMbox, 0, &err);
        TestTerminalMsgCtr++;
        sprintf(s, "%05u", TestTerminalMsgCtr);
        PC_DispStr(60, 22, s, DISP_FGND_YELLOW + DISP_BGND_BLUE);
        switch (key) {
            case '1':
                 sprintf(s, "\nCPU Usage = %3u%%\n", OSCPUUsage);
                 OSView_TxStr(s, 1);
                 break;

            case '2':
                 sprintf(s, "\n#Tasks    = %3u\n", OSTaskCtr);
                 OSView_TxStr(s, 1);
                 break;

            default:
                 OSView_TxStr("\n\nMicrium, Inc.",       1);
                 OSView_TxStr("\n1: CPU Usage (%)",      1);
                 OSView_TxStr("\n2: #Tasks",             1);
                 OSView_TxStr("\n?: Help (This menu)\n", 1);
                 break;
        }
    }
}
```

3.05 Porting µC/OS-View

Porting **µC/OS-View** should involve changing or creating only three files: `OS_VIEWc.C`, `OS_VIEWc.H` and `OS_VIEWa.ASM`. The assembly language file generally contains ISRs which, with µC/OS-II, should be written in assembly language as described in the µC/OS-II book.

You can download µC/OS-View ports from the Micriµm web site at www.Micrium.com. **µC/OS-View** ports are found next to µC/OS-II ports.

As a minimum, a port should define the code for 18 functions as listed below. Most of these functions are expected by `OS_VIEW.C` (i.e. the target independent code).

```
OSView_Exit()
OSView_GetCPUName()
OSView_GetIntStkBase()
OSView_GetIntStkSize()
OSView_InitTarget()
OSView_RxIntDis()
OSView_RxIntEn()
OSView_RxISR()
OSView_RxISRHandler()
OSView_RxTxISR()
OSView_RxTxISRHandler()
OSView_TickHook()
OSView_TimeGetCycles()
OSView_Tx1()
OSView_TxIntDis()
OSView_TxIntEn()
OSView_TxISR()
OSView_TxISRHandler()
```

OSView_Exit()

void OSView_Exit(void);

File	Called from
OS_VIEWc.C	Your Application

`OSView_Exit()` is called by your application when you want to stop running **µC/OS-View**. This function is meant to perform some cleanup operations such as disabling Rx or Tx interrupts, releasing interrupt vectors, and so on.

Arguments

None.

Returned Value

None

Notes/Warnings

None

Example

```
void Task (void *pdata)
{
    pdata = pdata;
    while (1) {
        .
        .
        if (Done using uC/OS-View) {
            OSView_Exit();
        }
        .
        .
    }
}


void OSView_Exit (void)
{
    /* Disable Tx and Rx interrupts */
    /* Release interrupt vectors    */
}
```

OSView_GetCPUName()

void OSView_GetCPUName(char *s);

File	Called from
OS_VIEWc.C	OSView_CmdGetSysInfo() (OS_VIEW.C)

OSView_GetCPUName() is called by the processor independent code to obtain the name of the CPU that the viewer is connected to. This function is trivial to write since it only involves copying the name of the processor into a string.

Arguments

s is a pointer to the name of the CPU. The name of the CPU should NOT exceed 29 characters (30 if you include the NUL character).

Returned Value

None

Notes/Warnings

You should not be calling this function from your application because it is called by the processor independent code.

Example

```
void OSView_GetCPUName (char *s)
{
    strcpy(s, "M16C");
}
```

OSView_GetIntStkBase()

INT32U OSView_GetIntStkBase(void);

File	Called from
OS_VIEWc.C	OSView_CmdGetSysInfo() (OS_VIEW.C)

OSView_GetIntStkBase() is called by the processor independent code to obtain the base address of the interrupt stack, if a separate interrupt stack is used. If the processor you are using doesn't have an interrupt stack or, you have not implemented that feature in software then this function should return 0.

Arguments

None

Returned Value

None

Notes/Warnings

You should not be calling this function from your application because it is called by the processor independent code.

Example

```
INT32U OSView_GetIntStkBase (void)
{
    return ((INT32U)&OSIntStkBase);
}
```

OR

```
INT32U OSView_GetIntStkBase (void)
{
    return ((INT32U)0);              /* If there is no interrupt stack */
}
```

OSView_GetIntStkSize()

INT32U OSView_GetIntStkSize(void);

File	Called from
OS_VIEWc.C	OSView_CmdGetSysInfo() (OS_VIEW.C)

OSView_GetIntStkSize() is called by the processor independent code to obtain the size (in number of bytes) of the interrupt stack, if a separate interrupt stack is used. If the processor you are using doesn't have an interrupt stack or, you have not implemented that feature in software then this function should return 0.

Arguments

None

Returned Value

None

Notes/Warnings

You should not be calling this function from your application because it is called by the processor independent code.

Example

```
INT32U OSView_GetIntStkSize (void)
{
    return ((INT32U)&OSIntStkSize);
}
```

OR

```
INT32U OSView_GetIntStkSize (void)
{
    return ((INT32U)0);              /* If there is no interrupt stack */
}
```

OSView_InitTarget()

`void OSView_InitTarget(void);`

File	Called from
OS_VIEWc.C	OSView_Init() (OS_VIEW.C)

`OSView_InitTarget()` is called by the processor independent code `OSView_Init()` to initialize the timer, interrupt vectors and the RS-232C serial port used to interface with the Windows application portion of **µC/OS-View**.

Arguments

None

Returned Value

None

Notes/Warnings

You should not be calling this function from your application because it is called by the processor independent code.

Example

```
void OSView_InitTarget (void)
{
    /* Initialize the 32-bit timer used to measure execution time */
    /* Initialize the RS-232C port to the desired baud rate       */
    /* Setup interrupt vectors                                    */
    /* Enable timer and UART interrupts                           */
}
```

OSView_RxIntDis()

void OSView_RxIntDis(void);

File	Called from
OS_VIEWc.C	The processor specific code

OSView_RxIntDis() is not currently called by the processor independent code but could be in a future release. However, it can be used by the target specific code to disable interrupts from the UART (Universal Asynchronous Receiver Transmitter) receiver. This function must ONLY disable receiver interrupts.

Arguments

None

Returned Value

None

Notes/Warnings

This function should only disable interrupts from the receiver and not affect other interrupt sources. For this, you may need to read the current interrupt disable mask register, alter the bit(s) needed to disable the receiver interrupt and write the new value to the interrupt mask register.

Example
(80x86 port using COM1 on a PC)

```
void OSView_RxIntDis (void)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    INT8U      stat;
    INT8U      mask;


    OS_ENTER_CRITICAL();
    stat = inp(OS_VIEW_8250_BASE + OS_VIEW_8250_IER) & ~BIT0;
    outp(OS_VIEW_8250_BASE + OS_VIEW_8250_IER, stat);
    if (stat == 0x00) {
        mask  = inp(OS_VIEW_8259_MASK_REG);
        mask |= OS_VIEW_8259_COMM_INT_EN;
        outp(OS_VIEW_8259_MASK_REG, mask);
    }
    OS_EXIT_CRITICAL();
}
```

OSView_RxIntEn()

void OSView_RxIntEn(void);

File	Called from
OS_VIEWc.C	The processor specific code

OSView_RxIntEn() is not currently called by the processor independent code but could be in a future release. However, it can be used by the target specific code to enable interrupts from the UART (Universal Asynchronous Receiver Transmitter) receiver. This function must ONLY enable receiver interrupts.

Arguments

None

Returned Value

None

Notes/Warnings

This function should only enable interrupts from the receiver and not affect other interrupt sources. For this, you may need to read the current interrupt disable mask register, alter the bit(s) needed to enable the receiver interrupt and write the new value to the interrupt mask register.

Example
(80x86 port using COM1 on a PC)

```
void OSView_RxIntEn (void)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr;
#endif
    INT8U      stat;
    INT8U      cmd;


    OS_ENTER_CRITICAL();
    stat = inp(OS_VIEW_8250_BASE + OS_VIEW_8250_IER) | BIT0;
    outp(OS_VIEW_8250_BASE + OS_VIEW_8250_IER, stat);
    cmd  = inp(OS_VIEW_8259_MASK_REG) & ~OS_VIEW_8259_COMM_INT_EN;
    outp(OS_VIEW_8259_MASK_REG, cmd);
    OS_EXIT_CRITICAL();
}
```

OSView_RxISR()

void OSView_RxISR(void);

File	Called from
OS_VIEWa.ASM	The UART Rx interrupt

OSView_RxISR() is the interrupt service routine (ISR) that is invoked by the processor hardware when a character is received by the UART (Universal Asynchronous Receiver Transmitter). If the UART issues a combined interrupt for a received and transmitted character then your interrupt should vector to OSView_RxTxISR() instead.

Arguments

None

Returned Value

None

Notes/Warnings

You don't need to write the contents of this function if your UART issues a combined ISR for both Rx and Tx characters. However, you MUST declare the function but leave the contents empty.

Example (Pseudo-code)

```
OSView_RxISR:
    Save ALL CPU registers;
    OSIntNesting++;                     /* Notify uC/OS-II of ISR entry       */
    if (OSIntNesting == 1) {            /* Save SP in TCB if first nested ISR */
        OSTCBCur->OSTCBStkPtr = SP;
    }
    OSView_RxISRHandler();              /* Call the C handler in OS_VIEWc.C   */
    OSIntExit();                        /* Notify uC/OS-II of ISR completion  */
    Restore all the CPU registers;
    Return from Interrupt;
```

OSView_RxISRHandler()

void OSView_RxISRHandler(void);

File	Called from
OS_VIEWc.C	OSView_RxISR() (OS_VIEWa.ASM)

OSView_RxISRHandler() is called by the UART (Universal Asynchronous Receiver Transmitter) ISR (Interrupt Service Routine) that is generated when a character is received. The Rx ISR should be called OSView_RxISR() (see previous page) in OS_VIEWa.ASM. If the UART issues a combined interrupt for a received and transmitted character then the ISR should call OSView_RxTxISRHandler().

OSView_RxISR() should call OSView_RxISRHandler() to process the interrupt from C instead of assembly language.

Arguments

None

Returned Value

None

Notes/Warnings

You don't need to write the contents of this function if your UART issues a combined ISR for both Rx and Tx characters. However, you MUST declare the function but leave the contents empty.

Example

```
void OSView_RxISRHandler (void)
{
    INT8U c;


    c = Read character from UART;
    OSView_RxHandler(c);          /* Pass to processor independent code */
    Clear interrupt;
}
```

OSView_RxTxISR()

void OSView_RxTxISR(void);

File	Called from
OS_VIEWa.ASM	The UART Rx interrupt

OSView_RxTxISR() is the interrupt service routine (ISR) that is invoked by the processor hardware when either a character is received or transmitted by the UART (Universal Asynchronous Receiver Transmitter). If the UART issues a separate interrupt for a received character and another one for a transmitted character then, you should instead use OSView_RxISR() and OSViewTxISR(), respectively.

OSView_RxTxISR() should call OSView_RxTxISRHandler() to process the interrupt from C instead of assembly language.

Arguments

None

Returned Value

None

Notes/Warnings

You don't need to write the contents of this function if your UART issues a separate ISR for for Rx and Tx characters. However, you MUST declare the function but leave the contents empty.

Example (Pseudo-code)

```
OSView_RxTxISR:
    Save ALL CPU registers;
    OSIntNesting++;                 /* Notify uC/OS-II of ISR entry       */
    if (OSIntNesting == 1) {        /* Save SP in TCB if first nested ISR */
        OSTCBCur->OSTCBStkPtr = SP;
    }
    OSView_RxTxISRHandler();        /* Call the C handler in OS_VIEWc.C   */
    OSIntExit();                    /* Notify uC/OS-II of ISR completion  */
    Restore all the CPU registers;
    Return from Interrupt;
```

OSView_RxTxISRHandler()

void OSView_RxTxISRHandler(void);

File	Called from
OS_VIEWc.C	OSView_RxTxISR() (OS_VIEWa.ASM)

OSView_RxTxISRHandler() is called by the UART (Universal Asynchronous Receiver Transmitter) ISR (Interrupt Service Routine) that is generated when either a character is received or a character has been transmitted. The Rx/Tx ISR should be called OSView_RxTxISR() in OS_VIEWa.ASM. If the UART issues a combined interrupt for a received and transmitted character then the ISR should call OSView_RxTxISRHandler().

Arguments

None

Returned Value

None

Notes/Warnings

You don't need to write the contents of this function if your UART issues a combined ISR for both Rx and Tx characters. However, you MUST declare the function but leave the contents empty.

Example

```
void OSView_RxTxISRHandler (void)
{
    INT8U c;


    if (Rx interrupt) {
        c = Read character from UART;
        OSView_RxHandler(c);        /* Pass to processor independent code */
        Clear Rx interrupt;
    }
    if (Tx interrupt) {
        OSView_TxHandler();         /* Call processor independent code    */
        Clear Tx interrupt;
    }
}
```

OSView_TickHook()

void OSView_TickHook(void);

File	Called from
OS_VIEWc.C	OSTimeTickHook() (OS_CPU_C.C)

OSView_TickHook() MUST be called from the µC/OS-II function OSTimeTickHook(). Hopefully, you would have access to the µC/OS-II port files and thus, you should simply add the call to the function there.

Arguments

None

Returned Value

None

Notes/Warnings

None

Example

See OSView_TimeCyclesGet() (next function) for a description of the code presented in this example.

```
void OSTimeTickHook (void)     /* uC/OS-II function is OS_CPU_C.C */
{
    OSView_TickHook();
}


void OSView_TickHook (void)    /* Function defined in OS_VIEWc.C  */
{
    INT16U  cnts16;
    INT16U  delta;


    cnts16             = Read counts from timer chip;
    delta              = cnts16 - OSView_TmrCntsPrev;
    OSView_CyclesCtr += delta;
}
```

OSView_TimeGetCycles()

INT32U OSView_TimeGetCycles(void);

File	Called from
OS_VIEWc.C	Processor independent code (OS_VIEW.C)

OSView_TimeGetCycles() is called by the processor independent code to read the current 'absolute' time which is generally provided by a real-time clock or a timer. Preferably, this clock would have a resolution in the microsecond range, or better. A 32-bit counter is preferable. However, if you can't get this from your hardware, you can obtain sufficient resolution from a 16-bit counter as long as you can keep track of overflows or sample the timer faster than its overflow rate.

OSView_TimeGetCycles() is called whenever a context switch occurs to record when a task completes and when the new task starts executing.

Arguments

None

Returned Value

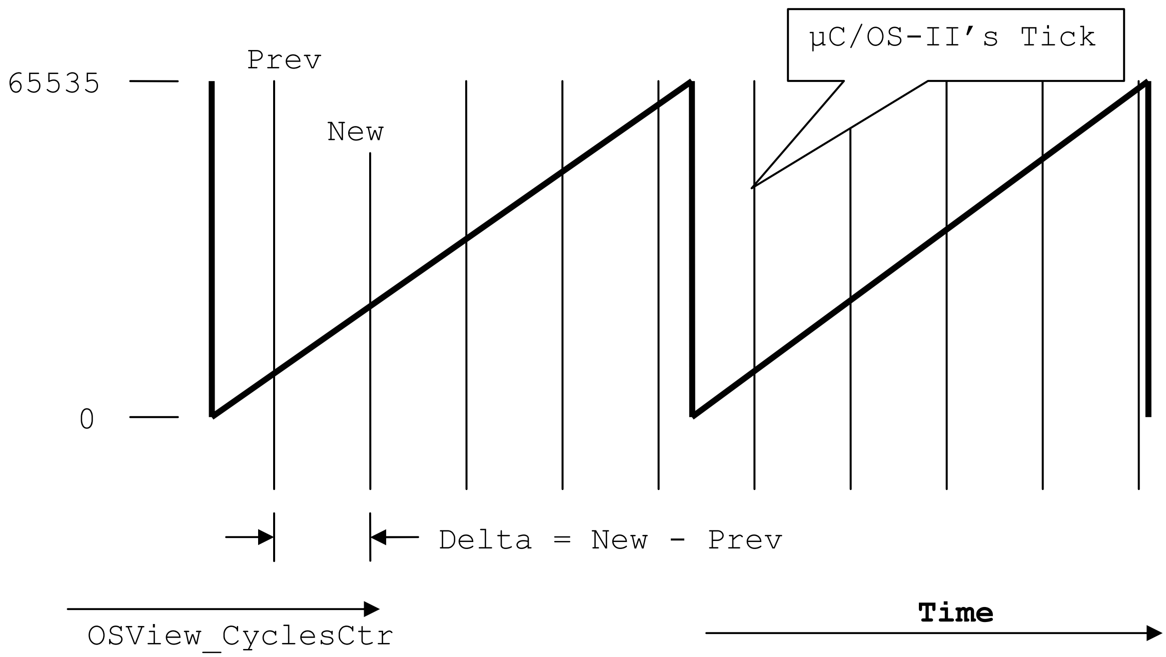
A 32-bit value representing absolute time.

Notes/Warnings

You should not be calling this function from your application because it is called by the processor independent code.

Example (assuming a 16-bit up counter)

The drawing below shows a free running timer that counts up from `0` to `65535` and rolls over to `0`. If µC/OS-II's tick rate is faster than the rollover rate then you could 'sample' this free-running timer by hooking into µC/OS-II's tick ISR as shown in the code below. `OSView_CyclesCtr` simply needs to contain the sum of the `Deltas`. Because we use 'unsigned' arithmetic, a 'small' `New` value minus a 'large' `Prev` value results in the proper delta. Because `OSView_CyclesCtr` is actually updated in `OSView_TickHook()`, `OSView_TimeGetCycles()` simply needs to return the value. You should be able to do something similar with a 16-bit down counter.




```
INT32U OSView_TimeGetCycles (void *pdata)
{
    INT32U  cycles;


    OS_ENTER_CRITICAL();
    cycles = OSView_CyclesCtr;
    OS_EXIT_CRITICAL();
    Return (cycles);
}



void OSView_TickHook (void)
{
    INT16U  cnts16;
    INT16U  delta;


    cnts16           = Read counts from timer chip;
    delta            = cnts16 - OSView_TmrCntsPrev;
    OSView_CyclesCtr += delta;
}
```

OSView_Tx1()

void OSView_Tx1(INT8U data);

File	Called from
OS_VIEWc.C	Processor independent code in OS_VIEW.C

`OSView_Tx1()` is called by the processor independent code to send a single byte to the serial port.

Arguments

`data` is the single character to send to the serial port.

Returned Value

None

Notes/Warnings

You should not be calling this function from your application because it is called by the processor independent code.

Example

```
void OSView_Tx1 (INT8U data)
{
    Write ‘data’ to UART Tx register;
}
```

OSView_TxIntDis()

void OSView_TxIntDis(void);

File	Called from
OS_VIEWc.C	The processor specific and independent code (OS_VIEW.C and OS_VIEWc.C)

OSView_TxIntDis() is called by the processor specific and independent code to disable interrupts from the UART (Universal Asynchronous Receiver Transmitter) transmitter.

Arguments

None

Returned Value

None

Notes/Warnings

This function should only disable interrupts from the transmitter and not affect other interrupt sources. For this, you may need to read the current interrupt disable mask register, alter the bit(s) needed to disable the transmitter interrupt and write the new value to the interrupt mask register.

Example

```
void OSView_TxIntDis (void)
{
    Disable Tx interrupts coming from the UART;
}
```

OSView_TxIntEn()

void OSView_TxIntEn(void);

File	Called from
OS_VIEWc.C	The processor specific and independent code (OS_VIEW.C and OS_VIEWc.C)

OSView_TxIntEn() is called by the processor independent code to enable interrupts from the UART (Universal Asynchronous Receiver Transmitter) receiver.

Arguments

None

Returned Value

None

Notes/Warnings

This function should only enable interrupts from the transmitter and not affect other interrupt sources. For this, you may need to read the current interrupt disable mask register, alter the bit(s) needed to enable the transmitter interrupt and write the new value to the interrupt mask register.

Example

```
void OSView_TxIntDis (void)
{
    Enable Tx interrupts coming from the UART;
}
```

OSView_TxISR()

void OSView_TxISR(void);

File	Called from
OS_VIEWa.ASM	The UART Tx interrupt

OSView_TxISR() is the interrupt service routine (ISR) that is invoked by the processor hardware when a character has been transmitted by the UART (Universal Asynchronous Receiver Transmitter). If the UART issues a combined interrupt for a received and transmitted character then your interrupt should vector to OSView_RxTxISR() instead.

OSView_TxISR() should call OSView_TxISRHandler() to process the interrupt from C instead of assembly language.

Arguments

None

Returned Value

None

Notes/Warnings

You don't need to write the contents of this function if your UART issues a combined ISR for both Rx and Tx characters. However, you MUST declare the function but leave the contents empty.

Example (Pseudo-code)

```
OSView_TxISR:
    Save ALL CPU registers;
    OSIntNesting++;                 /* Notify uC/OS-II of ISR entry       */
    if (OSIntNesting == 1) {        /* Save SP in TCB if first nested ISR */
        OSTCBCur->OSTCBStkPtr = SP;
    }
    OSView_TxISRHandler();          /* Call the C handler in OS_VIEWc.C   */
    OSIntExit();                    /* Notify uC/OS-II of ISR completion  */
    Restore all the CPU registers;
    Return from Interrupt;
```

OSView_TxISRHandler()

void OSView_TxISRHandler(void);

File	Called from
OS_VIEWc.C	OSView_TxISR() (OS_VIEWa.ASM)

OSView_TxISRHandler() is called by the UART (Universal Asynchronous Receiver Transmitter) ISR (Interrupt Service Routine) that is generated when a character has been transmitted. If the UART issues a combined interrupt for a received and transmitted character then the ISR should call OSView_RxTxISRHandler(). OSView_TxISRHandler() needs to call OSView_TxHandler() to have this processor independent function determine whether there is another character to send and send the character if there is.

Arguments

None

Returned Value

None

Notes/Warnings

None

Example

```
void OSView_TxISRHandler (void)
{
    OSView_TxHandler();          /* Call the processor independent handler */
    Clear the Tx interrupt;
}
```

3.05 µC/OS-View Target Configuration

µC/OS-View requires that you set some configuration constants which are generally placed in `OS_VIEWc.H`:

OS_VIEW_BAUDRATE

This constant defines the RS-232C communications baud rate between the Windows application and your target system. The default baud rate is 38400 and you should not change it unless your processor and serial port is unable to support this speed. The serial port is configured for 8 bits, no parity and 1 stop bit.

OS_VIEW_RX_BUF_SIZE

This `#define` determines the size of the buffer used to receive packets from the Windows application. You should not have to change the default value of `20`.

OS_VIEW_TX_BUF_SIZE

This `#define` determines the size of the buffer used to hold reply packets going back to the Windows application. The size you need depends on the number of tasks in your application. Each task requires 4 bytes. However, you should not set this `#define` to a value less than `64`. The maximum is `255` which allows you to display the status of up to `63` tasks.

OS_VIEW_TX_STR_SIZE

This `#define` determines the size of the buffer used to hold reply packets going back to the 'Terminal Window'. The size you need depends on the maximum number of characters you will send to the terminal window using `OSView_TxStr()`. You should not set this `#define` to a value less than `64`. The maximum is `255`.

3.06 ROM and RAM usage

µC/OS-View consumes both code space (i.e. ROM) and data space (i.e. RAM).

Code Space It's difficult to determine the exact amount of code space needed because this is compiler dependent. On an Intel 80x86 (Large Model) compiled using the Borland C/C++ V4.52 compiler, code space is about 4.5 Kbytes.

Data Space Data space usage is determined by the following equation:

```
7 * sizeof(INT8U) +
OS_VIEW_RX_BUF_SIZE * sizeof(INT8U) +
OS_VIEW_TX_BUF_SIZE * sizeof(INT8U) +
OS_VIEW_TX_STR_SIZE * sizeof(INT8U) +
9 * sizeof(INT16U) +
2 * sizeof(void *)
```

A different way of computing this information is shown in table 3-2:

Data Type	Quantity required
INT8U	7 + OS_VIEW_RX_BUF_SIZE + OS_VIEW_TX_BUF_SIZE + OS_VIEW_TX_STR_SIZE
INT16U	9
void *	2

Table 3-2, µC/OS-View data storage requirements.

On an Intel 80x86 (Large Model) compiled using the Borland C/C++ V4.52 compiler, data space is shown in the table below:

Data Type	Quantity required	As Configured	#Bytes
INT8U	7 + OS_VIEW_RX_BUF_SIZE + OS_VIEW_TX_BUF_SIZE + OS_VIEW_TX_STR_SIZE	7 + 20 + 255 + 255	539
INT16U	9	9	18
void *	2	2	8
		Total:	**565**

Table 3-3, µC/OS-View data storage on an 80x86 (Large Model).

References

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

Contacts

Micriµm, Inc.
949 Crestview Circle
Weston, FL 33327
USA
+1 954 217 2036
+1 954 217 2037 (FAX)
e-mail: Jean.Labrosse@Micrium.com
WEB: www.Micrium.com

CMP Books, Inc.
1601 W. 23rd St., Suite 200
Lawrence, KS 66046-9950
USA
+1 785 841 1631
+1 785 841 2624 (FAX)
WEB: http://www.rdbooks.com
e-mail: rdorders@rdbooks.com

JK Microsystems, Inc.
1403 Fifth Street, Suite D
Davis, CA 95616
USA
+1 530 297 6073
+1 530 297 6074 (FAX)
WEB: http://www.JKMicro.com
e-mail: jkmicro@jkmicro.com