Micriμm

Empowering Embedded Systems

μC/OS-II

μC/OS-View

and

The Freescale MC9S12NE64

(Using the Freescale DEMO9S12NE64 Evaluation Board)

Application Note

AN-1212

www.Micrium.com

Table Of Contents

1.00 Introduction

This document shows example code for using **μC/OS-II** and **μC/OS-View** on a Freescale MC9S12NE64 processor. To demonstrate the MC9S12NE64, we used a Freescale DEMO9S12NE64 Evaluation Board as shown in Figure 1-1.

We used the Freescale Codewarrior IDE version 4.5 to demonstrate this application. However, other tool-chains could be used.

Figure 1-1, Freescale Explorer 16 Evaluation Board

The application code is downloaded into Flash using a P&E Micro BDM Multilink. When the application is started, the 2 onboard LED's toggle at different rates and **μC/OS-View** is initialized.

1.01 Port Specific Details

This **μC/OS-II** port has been designed to operate using the MC9S12NE64 banked memory model. Paging must not be enabled within the Codewarrior project options since the PPAGE register save and restore functionality has been included in the **μC/OS-II** port.

This example uses the on chip PLL. Once configured, the processor clock is set to 50MHz and the bus clock for 25MHz. The PLL settings may be changed from within `BSP.h`. Refer to the section labeled "Configuring the PLL" for more information.

Additionally, this example assumes the use of ECT TC7 for the **μC/OS-II** Ticker. If TC7 is not available, this you may configure an alternate ECT timer channel by adjusting the macro named "`OS_TICK_OC`" within `BSP.h` accordingly. The file `Vectors.c` will also require modification. See the section labeled "Vectors.c" for more information.

All ISRs must be written as specified in the section labeled "Creating Interrupt Service Routines"

1.02 μC/OS-View

The application code described in this application note allows you to connect a Windows-based PC to your target and display run-time information about your target in a Window as shown in Figure 1-2. This is done via an add-on module called **μC/OS-View**.

Note that you can 'disable' **μC/OS-View** by removing the **μC/OS-View** files from the build and setting `OS_VIEW_MODULE` to `0` in `os_cfg.h`. You would need to do this is you didn't purchase **μC/OS-View** from Micriμm.

μC/OS-View is a combination of a Microsoft Windows application program and code that resides in your target system (in this case, the MC9S12NE64). The Windows application connects with your system via an RS-232C serial port at a default baud rate of 38,400 BPS. This can of course be changed. The Windows application allows you to '**View**' the status of your tasks which are managed by **μC/OS-II**.

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 (up to 253 tasks)
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 also allows you to send commands to your target and allow your target to reply back and display information in a 'terminal window'.

μC/OS-View is currently configured to use SCI0 as the default serial port. This may be changed by adjusting the macro `OS_VIEW_COMM_SEL` within `app_cfg.h` to either `OS_VIEW_SCI_0` or `OS_VIEW_SCI_1` based on the requirements of your application. The interrupt vector table will also require modification to account for the change of SCI port. See the section labeled "Vectors.c" and "Porting to Other MC9S12 Derivatives" for more information.

μC/OS-View is licensed on a per-developer basis. In other words, you are allowed to install **μC/OS-View** on multiple PCs as long as the PC is used by the same developer. If multiple developers are using **μC/OS-View** then each needs to obtain their own copy. Contact Micriμm for pricing information and to obtain the OS-View Windows executable.

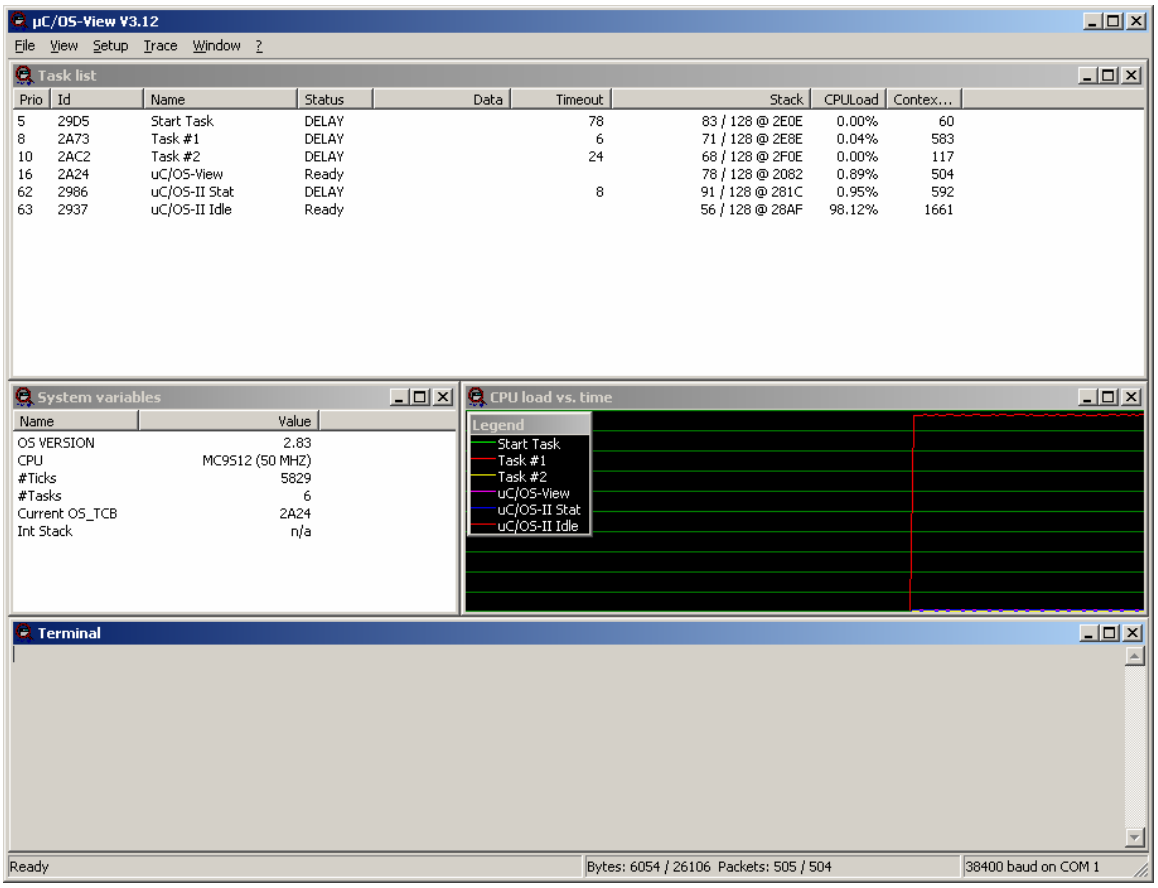


Figure 1-2, μC/OS-View Windows' 'Viewer'

1.03 Directories and Files

The code and documentation of 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 2.82.

\Micrium\Software\uCOS-II\Ports\HCS12\Paged\Metrowerks

This directory contains the standard processor specific files for a **μC/OS-II** port assuming the Freescale Codewarrior IDE. In fact, these files could easily be modified to work with other tool chains. However, you would place the modified files in a different directory. Specifically, this directory contains the following files:

```
os_cpu.h
os_cpu_a.s
os_cpu_c.c
```

μC/OS-View:

\Micrium\Software\uCOSView\Source

This directory contains portable source code for **μC/OS-View.**

```
OS_VIEW.C
OS_VIEW.H
```

\Micrium\Software\uCOSView\Ports\HCS12

This directory contains the **μC/OS-View** processor specific port for the MC9S12NE64.

```
OS_VIEWc.C
OS_VIEWc.H
OS_VIEWa.S
```

`OS_VIEWc.C` contains low level UART initialization, functions for enabling and disabling both Rx and Tx interrupts, and the RxTx_ISR Handler for used for processing both Rx and Tx characters.

`OS_VIEWc.H` contains constants for defining **μC/OS-View** buffer sizes.

`OS_VIEWa.S` contains the low level Interrupt Service Routines for Rx and Tx interrupts. These Interrupt Service Routines handle **μC/OS-II** related functionality, clear the interrupt source, and then call the Rx and Tx ISR Handlers located in `OS_VIEWc.C`.

Application Code:

\Micrium\Software\EvalBoards\Freescale\MC9S12NE64\FreescaleDemo\9S12NE64\Paged\Metrowerks\OS-View

This directory contains the Freescale Codewarrior project file for the AN-1212.

```
OS-View.mcp
```

\Micrium\Software\EvalBoards\Freescale\MC9S12NE64\FreescaleDemo\9S12NE64\Paged\Metrowerks\OS-View\Source

This directory contains the source code for an example running on the DEMO9S12NE64 evaluation board. It assumes the presence of **μC/OS-II**.

This directory contains:

```
app.c
app_cfg.h
includes.h
os_cfg.h
datapage.c
start12.c
```

`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 and `datapage.c` and `start12.c` are provided by Codewarrior but have been placed in the application directory for compatibility purposes.

Please see the section labeled “Porting to Other MC9S12 Derivatives” for more information about `datapage.c` and `start12.c`.

\Micrium\Software\EvalBoards\Freescale\MC9S12NE64\Freescale\Demo 9S12NE64\Paged\Metrowerks\BSP

This directory contains the Board Support Package for the DEMO9S12NE64 evaluation board. While some of the code in this directory may work on other MC9S12 derivatives, routines that are hardware dependent such as `LED_On()`will require modification depending on the hardware design of your EVB.

Please see the section labeled “Board Support Package” and “Porting to Other MC9S12 Derivatives” for more information related to the BSP.

This directory contains:

```
BSP.c
BSP.h
nvm.c
nvm.h
Vectors.c
```

`BSP.c` contains hardware specific source code for LED services, PLL initialization, **μC/OS-II** ticker initialization, and so on.

BSP.h contains macros for configuring the system PLL and **μC/OS-II** time tick ECT channel. Please see the section labeled "Porting to Other MC9S12 Derivatives" for more information related to the BSP.h

nvm.c and nvm.h contain hardware access functions for reading and writing both Flash and EEPROM during run-time. Neither of these files are used within the example after NVM initialization has been completed, however, they have been provided for convenience.

Vectors.c contains the processor interrupt vector table. This array of Interrupt Service Routine addresses must be updated whenever a new interrupt is being configured on the system. Interrupt vectors that are not in use should be plugged with the appropriate Dummy ISR handler provided.

\Micrium\Software\EvalBoards\Freescale\MC9S12NE64\Freescale
\Demo 9S12NE64\Paged\Metrowerks\OS-View\prm

This directory contains the processor linker file. Additionally, interrupt service routine vectors may be specified here as well. The user MUST remove all previously existing vector definitions within this file in favor of those specified in Vectors.c. This file must be changed when porting to a different MC9S12 derivative. See the section labeled "Porting to Other MC9S12 Derivatives below"

1.04 Codewarrior IDE

We used the Freescale Codewarrior IDE version 4.5 to compile and run the MC9S12NE64 example. You can of course use **μC/OS-II** with other tools. Figures 1-3 shows the project source tree with all of the files necessary to build the example.

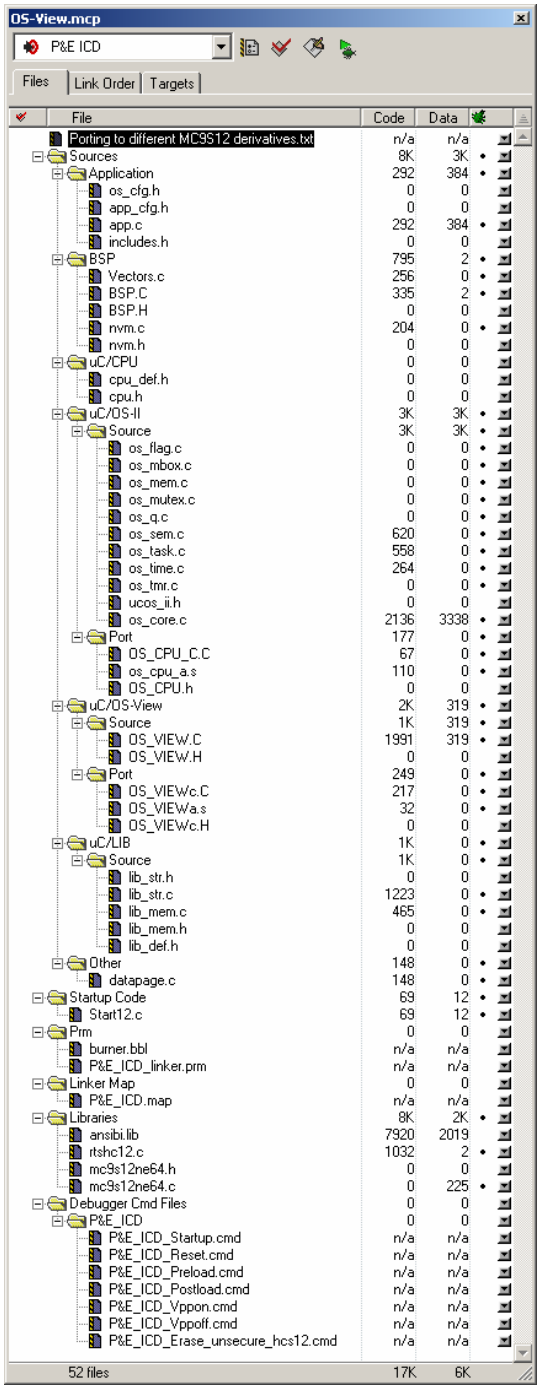


Figure 1-3, Codewarrior Source Tree

2.00 Example Code

As mentioned in the previous section, the test code for this board is found in the following directories and will be briefly described:

\Micrium\Software\EvalBoards\Freescale\MC9S12NE64\Freescale
\Demo 9S12NE64\Paged\Metrowerks\OS-View

It should be noted that processor header files and libraries are not included within the AN-1212 code archive since they are supplied by Freescale via the Codewarrior installation.

2.01 Example Code, app.c

app.c demonstrate some of the capabilities of **μC/OS-II.**

Listing 2-1, main()

```
void  main (void)                                                    (1)
{
    INT8U  err;


    BSP_IntDisAll();                                                 (2)

    OSInit();                                                        (3)

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

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

    OSStart();                                                       (6)
}
```

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

L2-1(2) We start off by calling a BSP function (see `bsp.c`) that will disable all interrupts. We do this to ensure that initialization doesn't get interrupted in case we do a 'warm restart'.

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`). `OS_TaskStat()` is also an internal task in **μC/OS-II**.

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 'Start Task'.

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. `OSStart()` does not return.

Listing 2-2, AppStartTask()

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

    BSP_Init();                                                 (1)

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

#if OS_VIEW_MODULE > 0                                          (3)
    OSView_Init(38400);
    OSView_TerminalRxSetCallback(AppTerminalRx);
    OSView_RxIntEn();
#endif

    AppTaskCreate();                                            (4)

    while (TRUE) {                                              (5)
        OSTimeDlyHMSM(0, 0, 0, 25);                             (6)
    }
}
```

L2-2(1) `BSP_Init()` is called to initialize the Board Support Package – the I/Os, the tick interrupt, and so on. `BSP_Init()` will be discussed in the next section.

L2-2(2) `OSStatInit()` computes how fast the CPU runs when `OS_TASK_STAT_EN` is set to 1 in `OS_CFG.H`.

L2-2(3) **μC/OS-View** has been adapted to this port. Setting `OS_VIEW_MODULE` to 1 in `OS_CFG.H` would enable initialization of this module. A serial cable may be optionally connected between the RS232 port and your PC for use with the **μC/OS-View** Windows application.

L2-2(4) Call a user defined function for creating additional **μC/OS-II** tasks. This function is not required and additional tasks could have been created directly within `AppStartTask()`. In order to make the example more interesting, two additional tasks are created as the result of this function call.

L2-2(5) As with all task managed by **μC/OS-II**, the task body must be in the form of an infinite loop. Tasks managed by **μC/OS-II** must never be allowed to exit. Instead, tasks should be deleted using `OSTaskDel()`when they are no longer desired.

L2-2(6) As **μC/OS-II** tasks must either enter an infinite loop 'waiting' for some event to occur or terminate itself. In this case, we wait for time to expire as the 'event'. This is accomplished by calling `OSTimeDlyHMSM()`.

Listing 2-3, AppTask1 ()

```
static void  AppTask1 (void *p_arg)                                    (1)
{
    (void)p_arg;
    while (TRUE) {
        LED_Toggle(0);                                                 (2)
        OSTimeDlyHMSM(0, 0, 0, 100);                                   (3)
    }
}
```

L2-3(1) The creation of AppTask1 is the result of the function call to `AppTaskCreate()` within the task `AppStartTask.`

L2-3(2) This BSP function allows on board LEDs to be toggled irrespective of their initial state. The function has been designed to take values between 0 and 2 that correspond to each of the onboard LED's. Note: the value 0 may be used to indicates that ALL LEDs be toggled.

L2-3(3) As **μC/OS-II** tasks must either enter an infinite loop 'waiting' for some event to occur or terminate itself. In this case, we wait for time to expire as the 'event'. This is accomplished by calling `OSTimeDlyHMSM()`with a timeout of 100 milliseconds.

Listing 2-4, AppTask2 ()

```
static void  AppTask1 (void *p_arg)                                    (1)
{
    (void)p_arg;
    while (TRUE) {
        LED_Toggle(0);                                                 (2)
        OSTimeDlyHMSM(0, 0, 0, 100);                                   (3)
    }
}
```

L2-4(1) The creation of AppTask1 is the result of the function call to `AppTaskCreate()` within the task `AppStartTask.`

L2-4(2) This BSP function allows on board LEDs to be toggled irrespective of their initial state. The function has been designed to take values between 0 and 2 that correspond to each of the onboard LED's. Note: the value 0 may be used to indicates that ALL LEDs be toggled.

L2-4(3) As **μC/OS-II** tasks must either enter an infinite loop 'waiting' for some event to occur or terminate itself. In this case, we wait for time to expire as the 'event'. This is accomplished by calling `OSTimeDlyHMSM()`with a timeout of 500 milliseconds.

2.02 Example 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
- **μC/OS-View**

2.03 Example 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.

2.04 Example 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** V2.83 or higher but also works with previous versions of **μC/OS-II**.

3.00 Board Support Package (BSP)

BSP stands for Board Support Package and provides functions to encapsulate common I/O access functions in order to make it easier for you to port your application code. In fact, you should be able to create other applications using the DEMO9S12NE64 evaluation board and reuse these functions thus saving you a lot of time.

The BSP performs the following functions:

- Determine the MC9S12NE64s CPU clock and bus frequencies
- Configure the LED I/Os for the DEMO9S12NE64 EVB and MC9S12NE64 CPU
- Configuration and handling of the **μC/OS-II** tick timer
- Configuration and handling of the **μC/OS-View** measurement timer

The BSP for the DEMO9S12NE64 is found in the follow directory.

\Micrium\Software\EvalBoards\Freescale\MC9S12NE64\Freescale
\Demo 9S12NE64\Paged\Metrowerks\BSP

The BSP files are:

```
BSP.c
BSP.h
nvm.c
nvm.h
Vectors.c
```

3.02 Board Support Package, bsp*.*

We will not be discussing every aspect of the BSP but only cover topics that require special attention.

Your application code must call `BSP_Init()` to initialize the BSP. `BSP_Init()` in turn calls other functions when necessary.

Listing 3-1, BSP_Init()

```
void BSP_Init (void)
{
    INT32U  sys_clk_frq;

#if PLL_EN > 0                                                  (1)
    PLL_Init();                                                 (2)
    BSP_SetECT_Prescaler(4);                                    (3)
#endif

    OSTickISR_Init();                                           (4)
    LED_Init();                                                 (5)

    sys_clk_frq  =  BSP_CPU_ClkFreq();                          (6)
    sys_clk_frq /=  1000;                                       (7)
    Flash_Init (sys_clk_frq);                                   (8)
```

```
}
```

L3-1(1) If `PLL_EN` is configured to 1 within `BSP.h`, the processor PLL will be initialized. The conditional compilation for PLL initialization is necessary since the ECT is dependent on system bus frequency. The ECT timer, TCNT, is a 16 bit up counter. The match register used to create the **μC/OS-II** time tick is also a 16 bit value. If the timer operates too quickly, then the number of time ticks necessary to obtain the desired `OS_TICKS_PER_SEC` (see `os_cfg.h`) will overflow during the call to `OSTickISR_Init()`. Therefore, the ECT prescaler must be increased from the default value when the PLL is active.

L3-1(2) This function initializes the on chip PLL. First, the multiplier and divider are configured, then the PLL is enabled, and finally, the system clock is switched from the main oscillator to that of the PLL output.

L3-1(3) Adjust the ECT prescaler if the PLL is enabled to prevent an overflow of `OSTickCnts` during the call to `OSTickISR_Init()`.

L3-1(4) Initialize the selected ECT channel for use with the **μC/OS-II** time tick interrupt. The code for this function is described below.

L3-1(5) Initialize the general purpose I/O pins used for controlling the onboard LEDs.

L3-1(6) Determine the CPU clock frequency in Hz during run time. It is highly recommended that application code make use of this function in order to program system dividers during runtime. This prevents the user from having to change all hard coded divider values should the clock frequency need to be modified at a later time.

Note: The system bus frequency is the CPU clock frequency divided by 2. Some module input clocks use the CPU clock as a reference while others use the bus clock. Be sure to determine the correct clock for the module being initialized and use `BSP_CPU_ClkFreq() / 2` when necessary.

L3-1(7) Convert the CPU clock frequency from Hz to KHz.

L3-1(8) Initialize the Flash memory access dividers.

Note: Flash memory block writing is not utilized within this application. However, the initialization of this module has been provided for convenience.

Listing 3-2, OSTickISR_Init ()

```
static void  OSTickISR_Init (void)
{
    INT32U cpu_frq;
    INT32U bus_frq;
    INT8U  ECT_Prescaler;


    cpu_frq = BSP_CPU_ClkFreq();                                    (1)
    bus_frq = cpu_frq / 2;                                          (2)

    ECT_Prescaler = TSCR2 & 0x07;                                   (3)

    ECT_Prescaler = (1 << ECT_Prescaler);                           (4)

                                                                    (5)
    OSTickCnts    = (INT16U)((bus_frq / (ECT_Prescaler * OS_TICKS_PER_SEC)) - 1);

#if OS_TICK_OC == 4                                                 (6)
    TIOS  |= 0x10;                                                  (7)
    TC4    = TCNT + OSTickCnts;                                     (8)
    TIE |= 0x10;                                                    (9)
#endif

#if OS_TICK_OC == 5
    TIOS  |= 0x20;
    TC5    = TCNT + OSTickCnts;
    TIE |= 0x20;
#endif

#if OS_TICK_OC == 6
    TIOS  |= 0x40;
    TC6    = TCNT + OSTickCnts;
    TIE |= 0x40;
#endif

#if OS_TICK_OC == 7
    TIOS  |= 0x80;
    TC7    = TCNT + OSTickCnts;
    TIE |= 0x80;
#endif

    TSCR1 = 0xC0;                                                  (10)
}
```

L3-2(1) Get the CPU operating frequency in Hz

L3-2(2) Divide the CPU frequency by 2 in order to obtain the bus clock frequency in Hz. This value is used to calculate the correct number of timer increments for the desired OS Tick rate.

L3-2(3) Determine the ECT prescaler value. The ECT prescaler acts as a divider to the input clock of the ECT. The higher the ECT prescaler, the more slowly TCNT, the ECT up counter register, increments. The ECT prescaler is an important piece of information when calculating required timer channel match value.

L3-2(4) Convert the ECT prescaler register value into the decimal equivalent suitable for mathematical calculations. (E.g. Turn the bit pattern ‘01’ into a prescaler value of 2 and so on).

L3-2(5) Compute the number of timer increments necessary to generate the desired μC/OS-II Tick rate.

L3-2(6) If `OS_TICK_OC` in `BSP.H` is defined as 4, then configure TC4 as the **μC/OS-II** tick source.

L3-2(7) Configure the desired timer channel as an output compare.

L3-2(8) Write the match register with the current value of the ECT counter (TCNT) plus the number of ticks until the next desired match.

L3-2(9) Enable output compare interrupts on the desired ECT channel.

L3-2(10) Start the timer.

Listing 3-3, Tmr_TickISR_Handler()

```
void  OSTickISR_Handler (void)
{
#if OS_TICK_OC == 4                                          (1)
    TFLG1 |= 0x10;                                           (2)
    TC4   += OSTickCnts;                                     (3)
#endif

#if OS_TICK_OC == 5
    TFLG1 |= 0x20;
    TC5   += OSTickCnts;
#endif

#if OS_TICK_OC == 6
    TFLG1 |= 0x40;
    TC6   += OSTickCnts;
#endif

#if OS_TICK_OC == 7
    TFLG1 |= 0x80;
    TC7   += OSTickCnts;
#endif

    OSTimeTick();                                            (4)
}
```

This function is called from an assembly interrupt service routine which informs **μC/OS-II** of the interrupt and calls the 'C' code interrupt handler. See `os_cpu_a.s` and the section labeled "Creating Interrupt Service Routines" for more information.

L3-3(1) If `OS_TICK_OC` is configured to 4

L3-3(2) Clear the interrupt source

L3-3(3) Adjust the timer channel match register so that a new time tick will occur after `OSTickCnts` additional counts.

L3-3(4) Call `OSTimeTick()` to inform **μC/OS-II** of the clock tick.

The ECT generates match interrupt when the up-counter value reaches the value stored within the timer channel match register. After an interrupt occurs, the match register is incremented to the next value for which a time tick interrupt is desired. The timer is allowed to free-run and overflow without error when necessary.

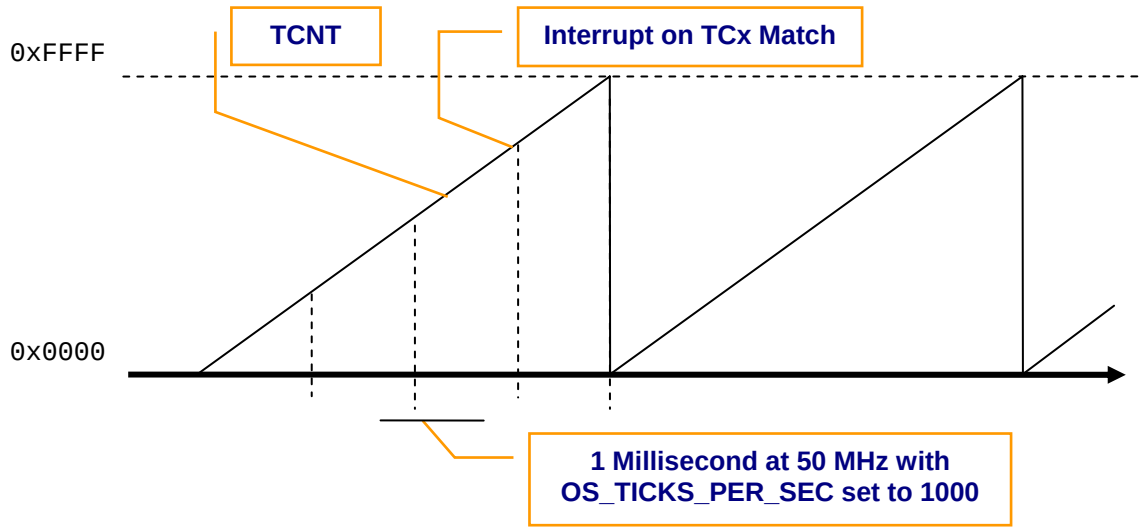


Figure 3-1, OS Tick Timer Operation

When the selected Timer issues an interrupt, the processor vectors to `__T2Interrupt()` or `__T4Interrupt()` depending on which timer is enabled for use with the OS Ticker. These ISR functions both call `Tmr_TickISR_Handler()` as described above in Listing 3-3. Only 1 timer for the OS Ticker may be enabled at a time.

You should note that ALL of your ISRs should be written in assembly where OS related processing may take place before calling an interrupt handler function of the form '`interrupt void MyISR_Handler(void)`' Refer to **AN-1212** for details.

3.03 Configuring the PLL

The PLL is an on chip peripheral capable of boosting the processor clock and bus frequencies higher than the frequency provided by the supplied oscillator across the XTAL pins of the MCU. Before attempting to reconfigure the PLL from BSP.h you should consult your MC9S12 derivative datasheet and understand the MCU's absolute maximum ratings. The absolute maximum ratings must be followed in order to prevent the possibility of damaging the device.

The MC9S12NE64 has a maximum processor clock (SYSCLK) of 50MHz. The bus clock (BUSCLK) is always ½ of the processor clock and must never exceed 25 MHz. The oscillator supplied on the DEMO9S12NE64 has an operating frequency of 25MHz.

Therefore, the highest possible settings for the PLL would be a multiplier of 2, and a divider of 1.

The PLL may be configured by adjusting the following macros within BSP.h:

```
OSCFREQ
PLL_EN
PLL_CLK_MUL
PLL_CLK_DIV
```

and is computed by the following formula:

```
((OSCFREQ * 2) * (PLL_CLK_MUL + 1) / (PLL_CLK_DIV + 1)
```

Where OSCFREQ is the frequency of the oscillator attached to the XTAL pins of the MCU. In the case of the DEM09S12NE64 EVB, OSCFREQ is equal to 25 MHz.

The PLL may be disabled by setting the value of PLL_EN to 0. The example provided is capable of running perfectly with the PLL either enabled or disabled. High performance applications may wish to enable the PLL, while power aware devices such as portable electronics may wish to run the device with the PLL disabled. A lower clock speed requires less operating power. The lowest achievable clock frequency is determined by the value of the input oscillator across the XTAL pins when the PLL is off. The operating frequency of the MC9S12NE64 must never fall below 5MHz. The minimum bus frequency would then be 2.5MHz.

When enabling the PLL on the MC9S12NE64, the highest possible values for PLL_CLK_MUL is 0, while the lowest possible value of PLL_CLK_DIV is 0.

Lets perform the math and see what happens when these values are used.

```
SYSCLK = ((25,000,000 * 2) * (0 + 1) / (0 + 1)
       =   (50MHz * 1) / 1 = 50MHz

BUSCLK =  SYSCLK / 2       = 25MHz
```

These values are within the maximum operating parameters for MC9S12NE64. However, other MC9S12 derivatives may allows for a broader range of multiplies and dividers.

The following macros have been provided and are accessible during run-time should you application need to know the value of SYSCLK or BUSCLK.

```
PLLCLK
BUS_CLK_FREQ
```

However, the BSP function `BSP_CPU_ClkFreq()` also yields the current SYSCLK frequency and is the preferred method for determine the system operating frequency.

Note: `BSP_CPU_ClkFreq()`returns a 32 bit unsigned integer representation of the SYSCLK frequency. Dividing this value by 2 will yield the BUSCLK frequency during run-time. It is recommended that users call this function before programming peripheral clock dividers so that the dividers need not be re-evaluated should the clock frequency be adjusted at a later time.

This method is used when computing the **μC/OS-II** tick number of counts during initialization, and when computing the baud rate for **μC/OS-View**. It is important to note that most, but not all, MC9S12 peripherals use the BUSCLK as a reference clock source. An example of divider initialization based on an unknown operating frequency may be performed as follows:

Listing 3-4, Set_SCI_BaudRate()

```
void  Set_SCI_BaudRate (INT32U baud)
{
    INT32U  baudDiv;                                        (1)


    baudDiv  = BSP_CPU_ClkFreq();                           (2)
    baudDiv /= (2 * baud * 16);                             (3)

    SCI0BDH = baudDiv >> 8;                                 (4)
    SCI0BDL = baudDiv & 0xFF;                               (5)
}
```

L3-4(1) Declare a 32 bit unsigned variable to hold the current SYSCLK frequency.

L3-4(2) Call `BSP_CPU_ClkFreq()` in order to obtain the current SYSCLK frequency.

L3-4(3) Divide the SYSCLK frequency by 2 in order to obtain the BUSCLK frequency. Note: the SCI's reference clock is derived from BUSCLK. Next divide the BUSCLK frequency by 16 to account for the SCI over sampling. The is mentioned in the SCI block documentation under the section for computing the SCI baud rate. Finally, divide by the desired baud rate to achieve the SCI divider that corresponds to the specified baud rate and the current MCU operating frequency. If you look closely, the division by 2, 16, and the desired baud was optimized in order to reduce the amount of truncation. Truncation on smaller dividers such as 6.78 (for 115,200 baud given `BUSCLK = 25MHz`) can be significant. The actual baud rate after truncation would be `(BUSCLK / (2 * 16 * 6)` = 130,208 baud which contains enough error to not work properly. If necessary, take the ceiling of fractional dividers. Its better to operate too slow than too fast.

L3-4(4) Write the high byte of the divider to the baud rate high byte register.

L3-4(5) Write the low byte of the divider to the baud rate low byte register.

3.04 Vectors.c

`Vectors.c` contains the interrupt vector table for the application. The interrupt vector table is necessary so that the processor knows the address of the interrupt service routine to jump to when a specific interrupt occurs. Failure to properly plug the interrupt vector table with the address of a valid handler may cause the application to crash. If a wrong, but valid, interrupt handler address is specified for vector number 'n' and the interrupt occurs, the interrupt source will not be cleared the processor will execute the same interrupt service routine indefinitely.

Care should be taken when working with the interrupt vector table.

For convenience, dummy interrupt service routines have been provided for all 64 vectors. This does not include the reset vector since its value must always be set correctly. When an interrupt vector is not in use, the dummy ISR for that vector should be plugged. In the case of a spurious interrupt, the processor will vector to the dummy ISR and loop indefinitely. Should this occur, you may be able to debug the application and catch the processor in the dummy interrupt service routine thus identifying the source of the spurious interrupt. The correct action may then be taken to correct the application to prevent this type of error in the future.

When plugging the interrupt vector table with a new vector, a 'C' prototype in the form of:

```
extern void near  MyISR(void);
```

Must be provided at the top of the file. The name of the ISR may then be plugged into the correct location of the interrupt vector table.

The following vectors are used by **μC/OS-II** and should not be modified:

Vector 4: SWI. Used to perform the **μC/OS-II** context switch.

Vector 15: Standard Timer Channel 7. This may be adjusted to one of the other Standard Timer Channel vectors if desired. See the section labeled "Porting to Other MC9S12 Derivatives" for more information.

Vector 20: SCI0. The selected communication port for **μC/OS-View**. SCI1 may be used instead of SCI0 if desired. Be sure to adjust `app_cfg.h` accordingly.

3.05 Creating Interrupt Service Routines

All interrupt service routines must contain a short assembly routine. The address of the assembly routine is used to plug the interrupt vector table, while the content is designed to notify **μC/OS-II** of the interrupt and call the user supplied interrupt handler written in either assembly or 'C' code.

The prototype specified at the top of `Vectors.c` (See section 3.04 above) is the 'C' code prototype for the following assembly interrupt service routine. It is this prototype that allows you to plug the interrupt vector table with the name (address) of the ISR from 'C'.

As a reminder, the prototype is written as follows:

```
extern void near  MyISR(void);
```

Of course, the name of the ISR would change each time a new ISR is declared since two ISR's of the same name cannot exist in the system simultaneously.

The format of an interrupt service routine is as follows:

Listing 3-5, MyISR

```
NON_BANKED:       section                                    (1)

PPAGE:            equ $0030                                  (2)

Xdef              MyISR_Handler                              (3)

Xref              OSIntExit                                  (4)
xref              OSIntNesting                               (5)
xref              OSTCBCur                                   (6)
xref              OSView_RxTxISRHandler                      (7)

MyISR:
    ldaa   PPAGE                                             (8)
    psha                                                     (9)

    inc    OSIntNesting                                     (10)

    ldab   OSIntNesting                                     (11)
    cmpb   #$01                                             (12)
    bne    MyISR1                                           (13)

    ldy    OSTCBCur                                         (14)
    sts    0,y                                              (15)

MyISR1:
    call   OSTickISR_Handler                                (16)

    cli                                                     (17)
    call   OSIntExit                                        (18)

    pula                                                    (19)
    staa   PPAGE                                            (20)

    rti                                                    (21)
```

L3-4(1) Force the contents of the assembly file, perhaps named: `myisr_a.s`, into NON_BANKED memory. This is critical since the processor only has a 16 bit address bus. Vectors that are accidentally placed into banked memory will have a 24 bit address (8 bit page number + 16 bit address) and will overflow the slot in the interrupt vector table.

L3-4(2) Define the address of the PPAGE register. This register is memory mapped and located at address 0x30 on the MC9S12NE64 MCU.

L3-4(3) XDEF is a Codewarrior assembly directive for prototyping external functions. This directive is equivalent to 'extern' in 'C' and allows the assembler to find the address of the ISR handler specified below on line item (16). The name being XDEF'd should match the name of your ISR handler whether it be in assembly or 'C' code. This directive is not necessarily portable to other assemblers.

L3-4(4) (4), (5), (6), and (7), are external references to variables defined in 'C'. These variables are referenced from the context of the assembly ISR and must therefore be declared external such that they are visible to the assembler and ISR file. This directive is not necessarily portable to other assemblers.

L3-4(8) Obtain a copy of the PPAGE register. This register must be saved because the **μC/OS-II** is operating under the BANKED memory model.

L3-4(9) Store the PPAGE register on the stack of the task that was interrupted.

L3-4(10) Increment `OSIntNesting`. This notifies **μC/OS-II** that at least one interrupt is in progress and that the scheduler should not schedule any new tasks to run until all nested interrupts have completed (e.g. `OSIntNesting` equals 0).

L3-4(11) Load a copy of `OSIntNesting` from memory into a register so a comparison may be made.

L3-4(12) Check `OSIntNesting` to see if its value is 1. If so, then this is the only interrupt in progress and no nested interrupts are pending completion.

L3-4(13) If interrupt have been nested, skip storing the current tasks stack pointer back into its task control block and jump to `MyISR1`. Note: the name of the ISR and the labels used within it must be changed for each new ISR implemented in the system. For convenience, the number '1' is added to the end of the ISR name in order to create a unique and convenient label to jump to.

L3-4(14) If no interrupts have been nested then the scheduler is free to schedule a new task when the ISR completes. Therefore the address of the current task TCB (Task Control Block) is obtained.

L3-4(15) The stack pointer of the interrupted task is stored within its own TCB should the scheduler perform a context switch at the end of the ISR.

L3-4(16) Call the user defined ISR handler. Generally the ISR handler is defined and prototyped in 'C'.

L3-4(17) Re-enable interrupts. This allows interrupts to nest one another.

L3-4(18) Call `OSIntExit()`. This informs **μC/OS-II** about the end of the interrupt. This is effectively the same as decrementing `OSIntNesting` however, the scheduler is also involved and if a context switch is required, `OSIntExit()` will not return.

L3-4(19) If a context switch is not necessary, obtain the copy of PPAGE saved at the beginning of the ISR

L3-4(20) Restore the PPAGE register.

L3-4(21) Return to the interrupted task.

4.00 Porting to Other MC9S12 Derivatives

Due to the similarities between various MC9S12 derivatives, it is easy to port the sample application from one derivative to another. The following steps must be performed in order to switch MCU derivatives:

Porting to different MC9S12 derivatives:

1) Navigate to C:\Micrium\Software\EvalBoards\Freescale\MC9S12NE64\Freescale Demo 9S12NE64\Paged\Metrowerks\OS-View and open the project file named "OS-View.mcp"
2) Replace the processor header files in the source tree with those from the desired derivative. Adjust includes.h accordingly.
3) Replace the CMD directory contents with the command files from a project built for the derivative of your choice.
4) Obtain a .prm file from a sample project belonging to the derivative of your choice. The .prm file contains linker configuration directives for the desired MCU derivative. Replace existing the linker .prm located in the below directory with the .prm of the new derivative.

   \Micrium\Software\EvalBoards\Freescale\MC9S12NE64\Freescale
   \Demo 9S12NE64\Paged\Metrowerks\OS-View\prm

   Note: Generally Codewarrior defines the Startup vector in the .prm file. All interrupt vector references MUST be removed from the .prm in favor of those already specified in `Vectors.c`.

4) Update BSP.c functions for LED_On(), LED_Off(), LED_Toggle() etc... to match your hardware configuration if required.
6) Adjust the macro `OSCFREQ` within `BSP.h` to account for a different oscillator frequency if applicable.
5) Ensure that the PLL settings within `BSP.h` are suitable for use within the new derivative. If you are unsure, disable the PLL temporarily until the proper settings can be determined.

Caveats:

1) Vectors.c is provided as is and is assumed to be correct for most MC9S12 derivates. Vectors can be added by prototyping the assembly ISR handler as an external at the top of the file, and then plugging the correct array location with the name of the ISR routine. See section 3.04 labeled “Vectors.c” for more information.
2) The example provided assumes the use of ECT TC7 for the OS Ticker. If TC7 is not available, this may be changed by adjusting the macro named "OS_TICK_OC" within BSP.h and by adjusting Vectors.c and placing "OSTickISR" in the desired vector location.
3) Start12.c may need to be replaced with an equivalent file from a project built for the derivative of your choice.

Note: The MC9S12NE64 derivative only has 4 ECT channels starting from number 4 and ending on number 7. Configuration checking within `BSP.h` prevents the setting of `OS_TICK_OC` to any

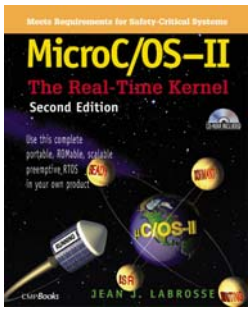
value out side of this range, however, the check may be disabled if you choose to run this example on a MC9S12 derivative with more ECT channels.

Licensing

If you intend to use **μC/OS-II** in a commercial product, remember that you need to contact **Micriμm** to properly license its use in your product. The use of **μC/OS-II** in commercial applications is **NOT-FREE**. Your honesty is greatly appreciated.

References

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



Contacts

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

Freescale Technology Inc.
2355 West Chandler Blvd.
Chandler, Arizona 85224-6199
USA
480-792-7200
WEB: www.Freescale.com