Micriμm

Empowering Embedded Systems

μC/OS-II

μC/TCP-IP

and

The NXP LPC2378 CPU

(Using the Keil MCB2300 EVB)

Application Note

AN-9078

www.Micrium.com

Table Of Contents

1.00 Introduction

This document shows example code for using **μC/OS-II** and **μC/TCP-IP** on a NXP LPC2378 (ARM7) processor, demonstrated on an Keil MCB2300 EVB as shown in Figure 1-1. The example is based off of Micrium AN-1014, the **μC/OS-II** port for ARM processors, and can be run in either ARM or Thumb mode.

We ported **μC/OS-View** to this board (see Section 1.01). If you purchased **μC/OS-View** from Micriμm, you can enable it by adding the **μC/OS-View** files to the build and setting the `OS_VIEW_MODULE` variable defined in `os_cfg.h` to 1.

We used the IAR's Embedded Workbench (EWARM) to demonstrate the examples, but other tool chains can be used.

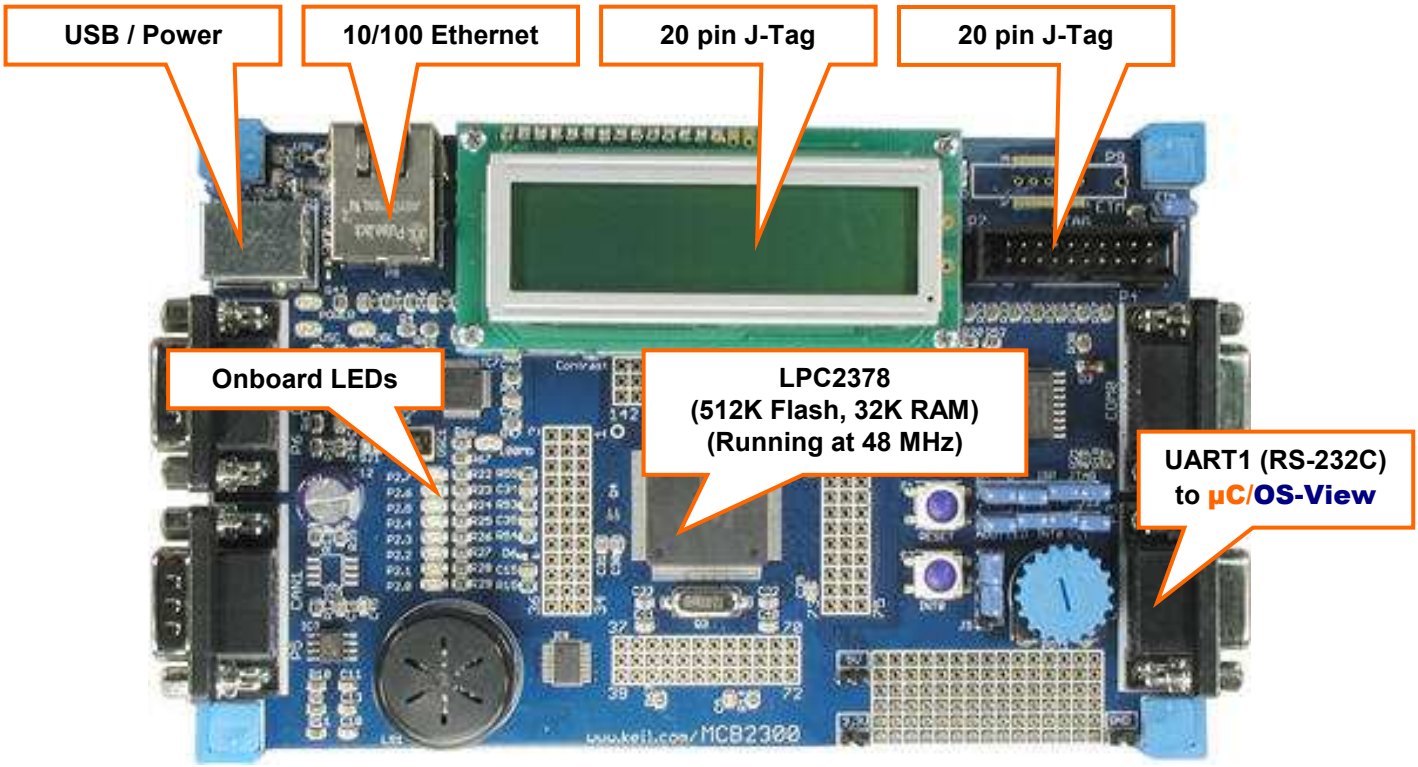

Figure 1-1, Keil MPC2300 EVB

1.01 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 2.83.

\Micrium\Software\uCOS-II\Ports\ARM\Generic\IAR

This directory contains the standard processor-specific files for the generic **μC/OS-II** ARM port assuming the IAR tool chain. These files could easily be modified to work with other tool chains (i.e. compiler/assembler/linker/locator/debugger); however, you would place the modified files in a different directory. Specifically, this directory contains the following files:

- **os_cpu.h**
- **os_cpu_a.asm**
- **os_cpu_c.c**
- **os_dcc.c**
- **os_dbg.c**

`os_dbg.c` is included to provide additional information to Kernel Aware debuggers like IAR's C-Spy.

With this port, you can use **μC/OS-II** in either ARM or Thumb mode. Thumb mode, which drastically reduces the size of the code, was used in this example, but compiler settings may be switched to generate ARM-mode code without needing to change either the port or the application code. The ARM/Thumb port is fully described in application note **AN-1014** which is available from the Micrium web site.

μC/OS-View:

\Micrium\Software\uCOSView\Source

This directory contains the processor independent code for **μC/OS-View**. The version used was 1.33. This directory contains the following files:

- **os_view.c**
- **os_view.h**

\Micrium\Software\uCOSView\Ports\ARM7\LPC2378\IAR

This directory contains the LPC2378 specific port for **μC/OS-View**:

- **os_viewc.c**
- **os_viewc.h**

Application Code:

\Micrium\Software\EvalBoards\NXP\MCB2300\IAR\OS-View-LCD-TCPIP

This directory contains the source code for the example application, composed of the following files:

- **app.c** contains the test code for the example application including the functions calls that start **μC/OS-II**, register tasks with the operating system, and toggle the onboard LEDs. **μC/OS-View** and **μC/TCP-IP** are also initialized from within this file. **app_cfg.h** is a configuration file specifying stack sizes and priorities for all tasks and `#defines` for important global application constants.
- **includes.h** is a master include file used by the application.
- **os_cfg.h** is the **μC/OS-II** configuration file.
- **net_conf.h** contains **μC/TCP-IP** configuration parameters.
- **OS-View-LCD-TCPIP.*** are the IAR Embedded Workbench project files.

\Micrium\Software\EvalBoards\NXP\LPC2378\IAR\BSP

This directory contains the Board Support Package for the Keil MCB2300 EVB:

- **bsp.c** contains the board support package which initializes critical processor functions (e.g., the PLLs) and provides support for peripherals such as the LED on the board. **bsp.h** contains prototypes for functions that may be called by the user.
- **net_bsp.c** and **net_bsp.h** contain low level hardware access routines which make up part of the LPC2378 EMAC network driver.
- **LPC2378_Flash.xcl** and **LPC2378_Ram.xcl** are IAR linker files which contain information about the placement of data and code segments in the processor's memory map. The data, code, and execution stacks are all mapped to Flash and RAM, respectively.
- **LPC2378_Ram.mac** contains instructions that are executed prior to loading code onto the processor. In this case, the lower 64 bytes of RAM are remapped onto the interrupt vector table at `0x00000000`.
- **cstartup.s79**

\Micrium\Software\uC-CPU\ARM\IAR

This directory contains processor-specific code intended to be used with the IAR compiler for ARM processors.

- **cpu_def.h,** which is located directly in `\Micrium\Software\uC-CPU`, declares `#define` constants for CPU alignment, endianness, and other generic declarations.
- **cpu.h** defines the Micriμm portable data types for 8, 16, and 32-bit signed and unsigned numbers (such as `CPU_INT16U`, which is a 16-bit unsigned type). These allow code to be independent of processor and compiler word size definitions.
- **cpu_a.s** contains generic assembly code for ARM7 or ARM9 processors which is used to enable and disable interrupts within the operating system. This code is called from C with `OS_ENTER_CRITICAL()` and `OS_EXIT_CRITICAL()`.

\Micrium\Software\EvalBoards\NXP\LPC2378\Doc

This directory is the directory that contains the documentation for the Keil MCB2300 evaluation board test code.

1.03 IAR Embedded Workbench

We used the IAR Embedded Workbench (EW) V4.40a to test the example. Of course, **μC/OS-II** can be used with other tools. Figure 1-3 shows the project configuration tree.

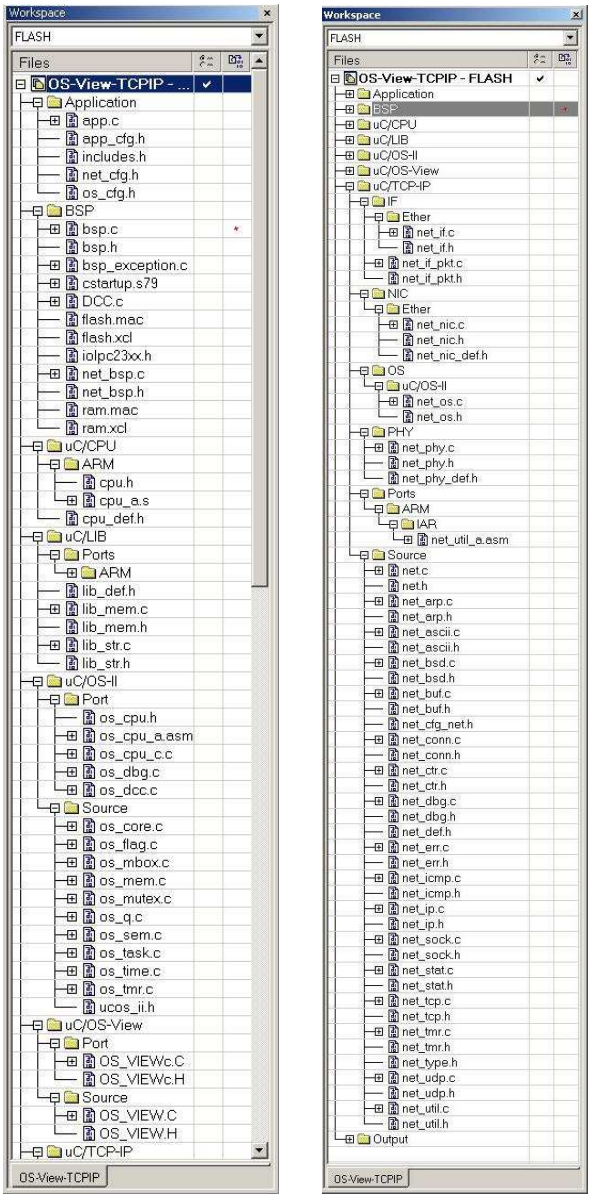

Figure 1-3, IAR EW Project Configuration

The test code works either in ARM or Thumb mode. In fact, if you switch between ARM and Thumb Processor Mode in the settings dialog box (see Figure 1-4) and rebuild the project, your code should run just as well. By selecting 'Thumb' and choosing to generate 'Interwork' code, you can mix ARM and Thumb code in your application.

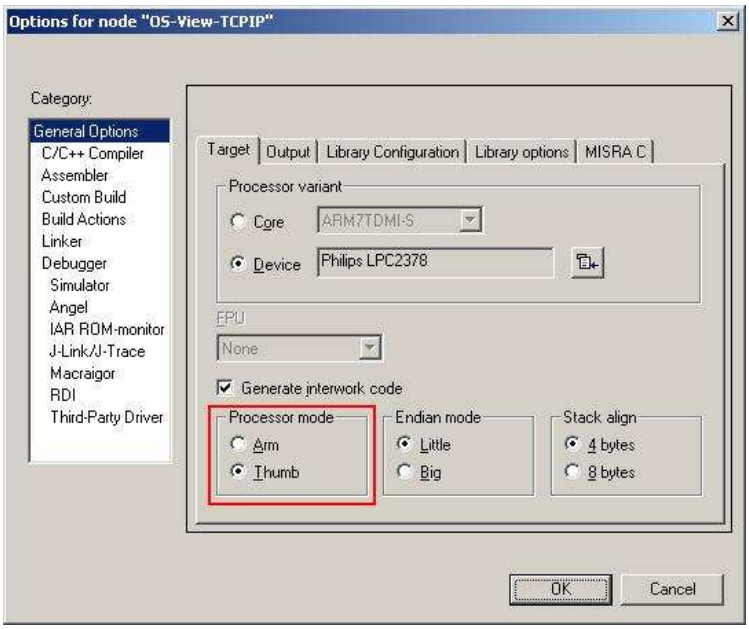

Figure 1-4, IAR EWARM Options

The IAR Embedded Workbench works with Micrium's **µC/OS-II** Kernel Awareness Plug-In which allows you to examine **µC/OS-II** kernel objects in tabular format when running the IAR C-Spy debugger.

Figure 1-5 shows all the tasks created in the example. For each task, you can see where the current stack pointer is pointing, how much stack space is being used, and other properties. The task names (which you may assign) are also listed.

The Kernel Awareness Plug-In provides a number of other useful information about **µC/OS-II** (semaphore list, mailbox list, queue list, etc.).

Task List

Name	Ref	Prio	State	Dly	Waiting On	Msg	Ctx Sw	Stk Ptr	Max%	Cur%	Max	Cur	Size	Starts @	Ends @
Start Task	3	1	Dly	1			16015	400006C0	16%	8%	192	104	1200	40000728	40000278
uC/OS-II Tmr	2	6	Sem	0	OS-TmrSig		1944	400071D8	25%	21%	132	108	512	40007244	40007044
Net IF Rx Task	5	8	Sem	0	Net IF Rx Queue		5062	40003F84	52%	11%	540	120	1024	40003FFC	40003BFC
Net Timer Task	4	10	Dly	7			1923	40003B9C	13%	8%	156	96	1200	40003BFC	4000374C
uC/OS-II Stat	1	62	Dly	5			1962	400060C8	28%	18%	148	96	512	40006128	40005F28
uC/OS-II Idle	0	63	Ready	0			21192	400062F0	19%	13%	100	68	512	40006334	40006134

Figure 1-5, µC/OS-II Kernel Awareness in C-Spy, Task List

2.00 Example Code

The application code is downloaded into Flash using a J-Link J-Tag emulator (though other emulators can be used). When the application is started, the eight onboard LEDs scroll from side to side and blink rapidly. If an Ethernet cable is plugged in, and your network is configured for the 192.168.0.x IP address range, then you may ping the LPLC2378 EVB by typing "ping 192.168.0.60" from the command prompt without the quotation marks. You may change the target IP address by adapting the function `AppInit_TCPIP()` within `app.c` accordingly. Micrium offers **μC/DHCPc** as an add-on module if required. Figure 2-1 demonstrates the use of 'ping' on a Microsoft Windows based computer.

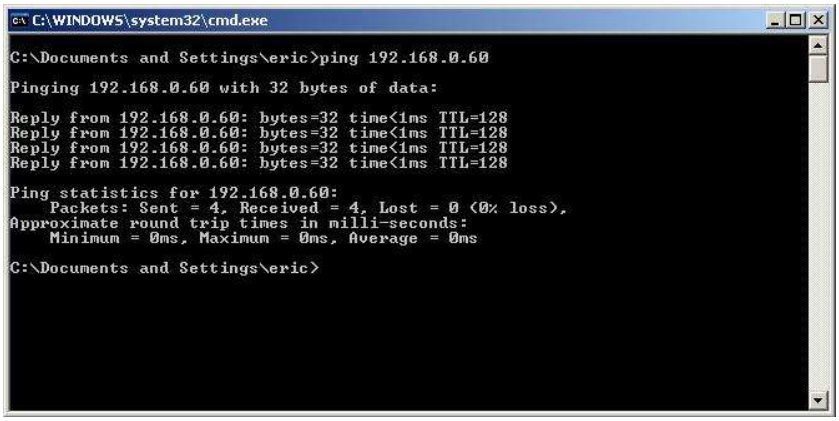

Figure 2-1, Pinging the Target Device

2.01 Example Code, `app.c`

A limited set of the LPC2138 capabilities are exhibited by the application code in `app.c`. For example, only one user task is created which is used for blinking the onboard LEDs and initializing **μC/TCP-IP**. However a statistics task, idle task, timer task, and network task are created by the operating system and TCP-IP stack module.

As with most C programs, we assume that the compiler startup code brings the CPU to execute `main()`. That being said, if you design an embedded application running out of Flash, we expect that you will properly initialize the CPU (clocks, power management, memory management, chip selects, etc.) and have your code call `main()`.

Listing 2-1, main()

```
void  main (void)                                                   (1)
{
    CPU_INT08U  err;


    BSP_IntDisAll();                                                (2)

    OSInit();                                                       (3)

    OSTaskCreateExt(AppTask_Start,                                  (4)
                    (void *)0,
                    (OS_STK *)&AppTask_StartStk[APP_TASK_START_STK_SIZE - 1],
                    APP_TASK_START_PRIO,
                    APP_TASK_START_PRIO,
                    (OS_STK *)&AppTask_StartStk[0],
                    APP_TASK_START_STK_SIZE,
                    (void *)0,
                    OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR);

#if OS_TASK_NAME_SIZE > 13                                          (5)
    OSTaskNameSet(APP_TASK_START_PRIO, "Start Task", &err);
#endif


    OSStart();                                                      (6)
}
```

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

L2-1(2) All interrupts are disabled to make sure we will not get interrupted until the application is fully initialized.

L2-1(3) As with all **μC/OS-II** applications, you need to call `OSInit()` before creating any task or any other kernel objects.

L2-1(4) We then create at least one task (in this case we use `OSTaskCreateExt()` to obtain additional information about your task). **μC/OS-II** creates either one or two internal tasks in `OSInit()`. **μC/OS-II** always creates an idle task, `OS_TaskIdle()`, and will create a statistics task, `OS_TaskStat()`, if you set `OS_TASK_STAT_EN` to `1` in `OS_CFG`.

L2-1(5) As of V2.6x, you can now name **μC/OS-II** tasks (and other kernel objects) and 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. Because C-Spy can work with the Kernel Awareness Plug-In available from Micrium, task names can be displayed during debugging.

L2-1(6) Finally, **μC/OS-II** is started by calling `OSStart()`. **μC/OS-II** will then begin executing `AppStartTask()` since that is the highest priority task created (both `OS_TaskStat()` and `OS_TaskIdle()` have lower priorities).

Listing 2-2, `AppTaskStart()`

```
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
    OSView_Init(38400);                                                  (3)
    OSView_TerminalRxSetCallback(AppTerminalRx);                         (4)
    OSView_RxIntEn();                                                    (5)
#endif

#ifdef  uC_TCPIP_MODULE
    AppInit_TCPIP();                                                     (6)
#endif

    LED_Off(0);                                                          (7)

    AppTaskCreate();                                                     (8)

    while (DEF_TRUE) {
        OSTimeDlyHMSM(0, 0, 0, 100);                                     (9)
    }
}
```

L2-2(1) `BSP_Init()` is called to initialize the Board Support Package—the I/Os, the tick interrupt, etc. (See section 3.0 for details.)

L2-2(2) `OSStatInit()` is used to initialize **μC/OS-II**'s statistics task. This only occurs if you enable the statistics task by setting `OS_TASK_STAT_EN` to 1 in `OS_CFG.H`. The statistics 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.

L2-2(3) `OSView_Init()` is called to initialize the **μC/OS-View** module. Here we need to specify the baud rate of the RS-232C port connecting the **μC/OS-View** 'viewer'. If you did not purchase **μC/OS-View** and 'enable' it (as covered in Section 1.01), this function will not be called.

L2-2(4) `OSView_TerminalRxSetCallback()` allows you to specify the name of a function that will be called by **μC/OS-View** when characters are typed on the 'Terminal Window' of the **μC/OS-View** viewer.

L2-2(5) `OSView_RxIntEn()` simply enables receive interrupts from the UART used for **μC/OS-View**.

L2-2(6) If **μC/TCP-IP** is present, then a call to `AppInit_TCPIP()` is made in order to initialize the TCP-IP stack. The constant `uC_TCPIP_MODULE` is defined at the top of `bsp.h` and can be toggled between `DEF_ENABLED` and `DEF_DISABLED` for debugging purposes.

L2-2(7) This BSP function turns off all the LEDs. The MCB2300 has a total of 8 onboard LEDs which are enabled and disabled in sequence during runtime.

L2-2(8) We then create additional application tasks by calling `AppTaskCreate()`. However, in this example, no additional tasks are necessary. You can of course create and delete tasks from anywhere in your code, however, for organization and convenience we have included an application hook to do so.

L2-2(9) Any task managed by **μC/OS-II** must either enter an infinite loop 'waiting' for some event to occur or terminate itself. We decided to use the startup task to drive the onboard LEDs. This task calls the `OSTimeDlyHMSM()` function in order to satisfy the above requirement.

Listing 2-3, AppInit_TCPIP()

```
static void AppInit_TCPIP (void)
{

#if EMAC_CFG_MAC_ADDR_SEL == EMAC_CFG_MAC_ADDR_SEL_CFG              (1)
    NetIF_MAC_Addr [0] = 0x00;
    NetIF_MAC_Addr [1] = 0x50;
    NetIF_MAC_Addr [2] = 0xC2;
    NetIF_MAC_Addr [3] = 0x25;
    NetIF_MAC_Addr [4] = 0x60;
    NetIF_MAC_Addr [5] = 0x01;
#endif

    err = Net_Init();                                               (2)

    ip      = NetASCII_Str_to_IP("192.168.0.60",  &err);            (3)
    msk     = NetASCII_Str_to_IP("255.255.255.0", &err);
    gateway = NetASCII_Str_to_IP("192.168.0.1",   &err);

    err     = NetIP_CfgAddrThisHost(ip, msk);
    err     = NetIP_CfgAddrDfltGateway(gateway);
}
```

L2-3(1) If `EMAC_CFG_MAC_ADDR_SEL` is defined as `EMAC_CFG_MAC_ADDR_SEL_CFG`, that is to say, the MAC address is user defined in software, then this is where the user specifies the device MAC address. If an EEPROM is to be used for setting the MAC address, then software must read the MAC address from the EEPROM and fill the contents of the `NetIF_MAC_Addr[]` array before calling `Net_Init()`.

L2-3(2) `Net_Init()` is called to initialize **μC/TCP-IP** stack.

L2-3(3) The user should specify an IP, Netmask, and Gateway address for **μC/TCP-IP**. After converting the dotted decimal notation to 32 bit values, a call to both `NetIP_CfgAddrThisHost()` and `NetIP_CfgAddrDfltGateway()` is made in order to configure **μC/TCP-IP** to use the specified addresses.

2.02 Example Code, `os_cfg.h`

This file is used to configure **μC/OS-II**. Among the approximately 60 `#define`s in this file are included variables defining the maximum number of tasks that your application can have, which services will be enabled (semaphores, mailboxes, queues, etc.), and the size of the idle and statistic task. Each entry is commented and additional information about the purpose of each `#define` can be found in *μC/OS-II, the Real-Time Kernel* by Jean Labrosse. `os_cfg.h` assumes you have **μC/OS-II** V2.83 or higher.

3.00 Board Support Package (BSP)

The Board Support Package (BSP) provides functions to encapsulate common I/O access functions and make porting your application code easier. Essentially, these files are the interface between the application and the Keil MCB2300 EVB. Though one file, `bsp.c`, contains some functions which are intended to be called directly by the user (all of which are prototyped in `bsp.h`), the other files serve the compiler (as with `cstartup.s79`).

The BSP includes functions to

- Set and determine the LPC2378 CPU clock frequency (set to 48MHz).
- Configure the I/Os for the LPC2378 Evaluation Board.
- Provide hardware access functions for **μC/LCD**.
- Read the status of the onboard INT0 push button.
- Handle IRQ and FIQ ISRs.
- Sets up **μC/OS-View** timer functions (if **μC/OS-View** is enabled).
- Hande **μC/OS-II**'s tick timer.
- Sets up the VIC (Vectored Interrupt Controller).
- Configure the external PHY address, and LPC2378 EMAC descriptor list.

3.01 IAR-Specific BSP Files

The BSP includes two files intended specifically for use with IAR tools: `flash.xcl`, and `cstartup.s79`. These serve to define the memory map, ARM exception stack sizes, and initialize the processor prior to loading or executing code. If the example application is to be used with other tool chains, the services provided by these files must be replicated as appropriate.

Before the processor memories can be programmed, the compiler must know where code and data should be placed. To accomplish this, IAR requires a linker command file, such as `flash.xcl`, that provides directives to accomplish this. In the former, all code, data, and stack and heap segments are placed in the 32kB internal RAM between `0x40000040` and `0x40007FFF`. The first 64 bytes of RAM are reserved for the exception vector table.

In `cstartup.s79` is code which will be executed prior to calling `main()`. One important inclusion is the specification of the exception vector table (as required for ARM cores) and the setup of various exception stacks. After executing, this function branches to the IAR-specific ?`main` function, in which the processor is further readied for entering application code.

3.02 BSP, bsp.c and bsp.h

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

Please take special care to notice the macro named `BSP_DEBUG` at the top of `bsp.c`. During normal operation this macro should be defined to 0, when debugging via the JTAG interface, it should be set to 1. Setting the macro to 0 will cause the OS Tick timer to free run and thus provide

accurate statistics measurements for **μC/OS-View**. However, debugging your application with the macro set to 0 is not possible since the LPC2378 does not disable the internal timers while in debug mode. This causes **μC/OS-II** to miss the next tick interrupt. In order to recover from the missed interrupt, the timer must wrap all the way around to the previous match value. This can take up to several minutes depending on your operating frequency. It is therefore best to define `BSP_DEBUG` to 1 when debugging, and 0 when releasing final code. A side effect to setting this macro to 1 is that the task CPU usage counters in **μC/OS-View** will report the wrong values since the timer resets to 0 after each match interrupt.

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

Listing 3-1, `BSP_Init()`

```
void  BSP_Init (void)
{
    BSP_PLL_Init();                                          (1)
    BSP_MAM_Init();                                          (2)
    BSP_IO_Init();                                           (3)
    VIC_Init();                                              (4)
    LED_Init();                                              (5)
    Tmr_TickInit();                                          (6)
}
```

L3-1(1) The PLL is setup. See Listing 3-2 for details.

L3-1(2) The MAM (Memory Acceleration Module) is setup. The MAM uses a bank of Flash memory to accelerate the performance when the processor is running code from Flash.

L3-1(3) The board I/O is initialized.

L3-1(4) `VIC_Init()` places 'dummy' vectors in the interrupt controller, allowing easier capture of uninitialized interrupt vectors.

L3-1(5) The LED services are initialized. After this function call, your application can call `LED_On()`, `LED_Off()`, or `LED_Toggle()` to turn on, turn off, or toggle, the onboard LEDs.

L3-1(6) Timer #0, which will generate interrupts for the **μC/OS-II** clock tick, is initialized by `Tmr_TickInit()`. See Listing 3-3 for details.

Listing 3-2, `BSP_PLL_Init()`

```
static  void  BSP_PLL_Init (void)
{
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr = 0;
#endif
    CPU_INT32U  m;
    CPU_INT32U  n;
    CPU_INT32U  cClkDiv;
    CPU_INT32U  usbClkDiv;


    m           =       11;                                     (1)
    n           =        0;
    cClkDiv     =        5;
    usbClkDiv   =        5;

    if ((PLLSTAT & (1 << 25)) > 0) {                            (2)
        CPU_CRITICAL_ENTER();
        PLLCON  &= ~(1 << 1);
        PLLFEED  =    0xAA;
        PLLFEED  =    0x55;
        CPU_CRITICAL_EXIT();
    }

    CPU_CRITICAL_ENTER();                                       (3)
    PLLCON     &= ~(1 << 0);
    PLLFEED     =     0xAA;
    PLLFEED     =     0x55;
    CPU_CRITICAL_EXIT();

    SCS        &= ~(1 << 4);                                    (4)
    SCS        |=  (1 << 5);                                    (5)

    while ((SCS &  (1 << 6)) == 0) {                            (6)
        ;
    }

    CLKSRCSEL   =  (1 << 0);                                    (7)

    CPU_CRITICAL_ENTER();                                       (8)
    PLLCFG      =  (m << 0) | (n << 16);
    PLLFEED     =     0xAA;
    PLLFEED     =     0x55;
    CPU_CRITICAL_EXIT();

    CPU_CRITICAL_ENTER();                                       (9)
    PLLCON     |=  (1 << 0);
    PLLFEED     =     0xAA;
    PLLFEED     =     0x55;
    CPU_CRITICAL_EXIT();

    CCLKCFG     =   cClkDiv;                                   (10)
    USBCLKCFG   =   usbClkDiv;                                 (11)

    while ((PLLSTAT & (1 << 26)) == 0) {                       (12)
        ;
    }

    CPU_CRITICAL_ENTER();                                      (13)
    PLLCON     |=  (1 << 1);
    PLLFEED     =     0xAA;
    PLLFEED     =     0x55;
    CPU_CRITICAL_EXIT();

    while ((PLLSTAT & (1 << 25)) == 0) {                       (14)
        ;
    }
}
```

L3-2(1) The PLL is setup with a multiplier (M) = 12 and divider (N) = 1, while the CPU clock and USB clock dividers = 6 respectively. The PLL input frequency (Fin) is defined in `bsp.h` as 12MHz. The PLL output frequency, (Fcco), is calculated as Fcco = 2 * Fin * M / N = (2 * 12 * 12 / 1) = 288MHz. This value is then divided by the CPU clock divider to form the CPU clock frequency. Therefore, the CPU clock frequency = 288MHz / 6 = 48MHz. The same holds true for the USB clock frequency which is created by dividing Fcco by the USB clock divider which = 288MHz / 6 = 48MHz.

Note: For engineering samples, the value of Fcco must never exceed 288MHz.

L3-2(2) If the PLL is already connected, disconnect the PLL before changing settings.

L3-2(3) Ensure that the PLL is disabled before changing settings.

L3-2(4) Inform the processor that the Main oscillator is between 1 and 20 MHz.

L3-2(5) Enable the Main oscillator.

L3-2(6) Wait for the Main oscillator to become ready for use.

L3-2(7) Switch to the Main oscillator. PLL Fin = 12MHz.

L3-2(8) Update the PLL block with the desired values for M and N, followed by a PLL feed sequence.

L3-2(9) Enable the PLL, followed by a PLL feed sequence.

L3-2(10) Configure the CPU clock divider.

L3-2(11) Configure the USB clock divider.

L3-2(12) Wait for the PLL to lock.

L3-2(13) Connect the PLL, followed by a PLL feed sequence.

L3-2(14) Wait for the PLL to become connected.

Listing 3-3, `Tmr_TickInit()`

```
void  Tmr_TickInit (void)
{
    CPU_INT32U  cClkFrq;
    CPU_INT32U  pClkFrq;


    VICIntSelect       &= ~(1 << VIC_TIMER0);                      (1)
    VICVectAddr4        =  (CPU_INT32U)Tmr_TickISR_Handler;
    VICIntEnable        =  (1 << VIC_TIMER0);

    cClkFrq             =   BSP_CPU_ClkFreq();                     (2)
    PCLKSEL0           &= ~(3 << 2);                               (3)
    pClkFrq             =   cClkFrq / 4;                           (4)

    Tmr_ReloadCnts      =   pClkFrq / OS_TICKS_PER_SEC;            (5)

    T0TCR               =  (1 << 1);                               (6)
    T0TCR              &= ~(1 << 1);                               (7)
```

```
#if BSP_DEBUG == 0                                                  (8)
    T0MR0               =   T0TC + Tmr_ReloadCnts;
    T0MCR               =   1;
#else                                                               (9)
    T0MR0               =   Tmr_ReloadCnts;
    T0MCR               =   3;
#endif
    T0CCR               =   0;                                     (10)
    T0EMR               =   0;                                     (11)
    T0TCR               =   1;                                     (12)
}
```

L3-3(1) This code sets up the interrupt controller to vector to `Tmr_TickISR_Handler()` (see `BSP.C`) when Timer #0 issues an interrupt. Timer #0 is designed to use VIC vector #4.

L3-3(2) We determine the peripheral clock frequency by calling `BSP_CPU_ClkFreq()`. The value returned is in Hertz.

L3-3(3) The timer peripheral clock divider is configured to 4. This means that the timer operates at a frequency of CPU Clock / 4;

L3-3(4) Calculate the peripheral clock frequency for timer 4, knowing the divider was previously set to 4.

L3-3(5) Determine the number of timer increments necessary in order to sustain `OS_TICKS_PER_SEC`.

L3-3(6) Reset and clear the timer counter register.

L3-3(7) Release the reset bit.

L3-3(8) When not in BSP debug mode, configure the timer to free-run

L3-3(9) When in BSP debug mode, configure the timer to reset to 0 after a successful match.

L3-3(10) Capture is disabled.

L3-3(11) No external match enabled.

L3-3(12) Enable the timer.

We can setup the timer in one of two ways, see L3-3(8) and L3-3(9):

1. As shown in Figure 3-1, `TC` free runs from `0x00000000` to `0xFFFFFFFF`. An interrupt is generated upon compare of Timer #0's `TC` register and the match register `MR0`, and the match register is reloaded for the next `TC` match. If we needed to use a timer for both **μC/OS-View** and **μC/OS-II**'s tick interrupt, we could use this method. However, this setup has one major drawback: if the processor were stopped for debugging purposes, timer interrupts would not occur until the `TC` once again matches the value of the match register. In other words, under worst case conditions, it could take several minutes for tick interrupts to resume.

2. In Figure 3-2, the `TC` is reset upon compare with the match register. The tick interrupt is generated by a timer configured in this manner.

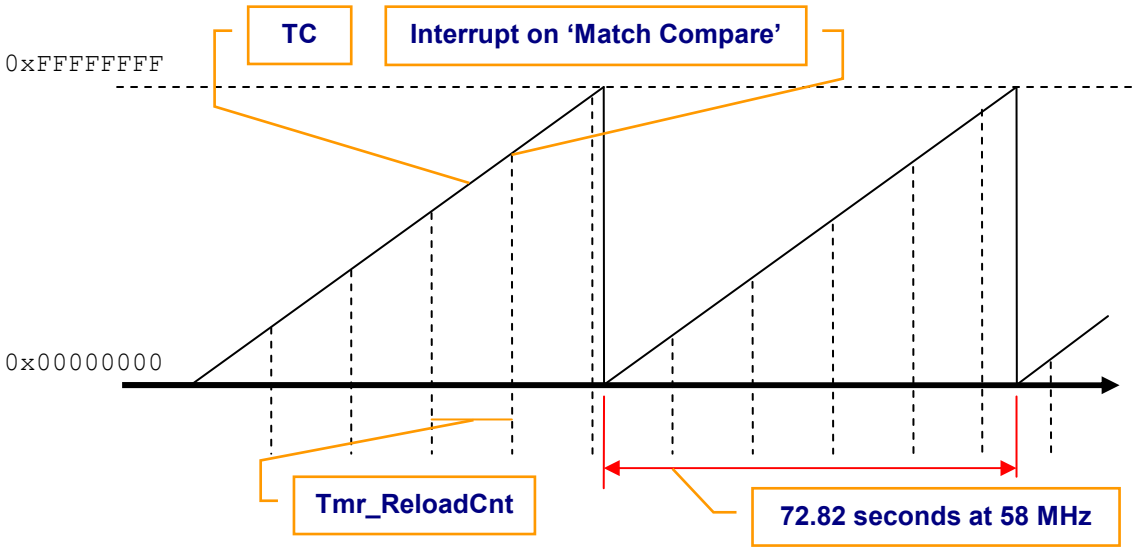

Figure 3-1, TC Free runs; Reload Match Register upon Compare

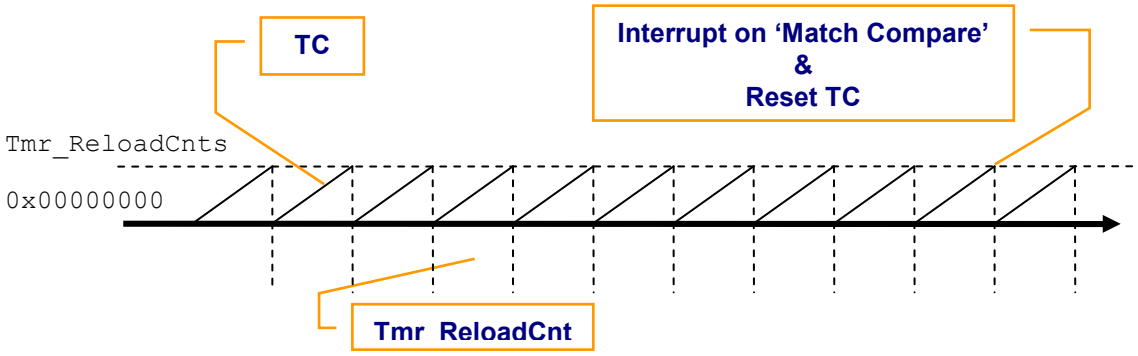

Figure 3-2, TC Free runs; Reload Match Register upon Compare

When Timer #0 issues an interrupt, the processor vectors to `ARM_CPU_ExceptIRQHndlr()` which then calls `OS_CPU_ExceptHndlr()` (see `bsp_exception.c`).

`OS_CPU_ExceptHndlr()` reads the VIC to obtain the address of the interrupting device and then calls this function. In our case, this is `Tmr_TickISR_Handler()` as shown in Listing 3-4.

Listing 3-4, Tmr_TickISR_Handler()

```
void  Tmr_TickISR_Handler (void)
{
    T0IR        = 0xFF;                             (1)

#if BSP_DEBUG == 0
    T0MR0      += Tmr_ReloadCnts;                   (2)
#endif

    OSTimeTick();                                   (3)
}
```

L3-4(1) This code clears the interrupt source (the Timer #0 interrupt).

L3-4(2) If `BSP_DEBUG == 0`, then update the match register to the next match value while the timer continues to count toward this value. Otherwise, if `BSP_DEBUG == 1`, the timer will reset to 0 and the existing match value will remain.

L3-4(3) `OSTimeTick()` is called to handle the **μC/OS-II** clock tick.

3.03 BSP, Interrupts

Application ISRs should be initialized as follows:

1) Write VICIntSelect and configure the local interrupt source for either IRQ or FIQ mode

2) Write corresponding vector address register with the address of the ISR handler function, ex: VICVectAddr4 = (CPU_INT32U)(MyISRHandler), where MyISRHandler is the name of the ISR handler function. In this case, vector 4, the timer interrupt vector is patched. Vector numbers run from 0 to 31. Consult the documentation for a list of vector numbers and their associated interrupt sources.

3) Write the VICIntEnable register to enable VIC interrupts for the desired interrupt source

4) Enable the local interrupt source

5) When an interrupt occurs, clear the local interrupt source within the ISR handler. The BSP code will handle clearing the VIC interrupt by means of writing 0x00 to the VICAddress register when the ISR handler returns.

You should note that ALL of your ISRs should be written as '`void MyISR(void)`' functions as shown. Refer to AN-1014 for details.

Listing 3-5 `VIC_Init()`

```
void  VIC_Init (void)
{
    VICIntEnClear =  0xFFFFFFFF;                                    (1)
    VICAddress    =  0;                                             (2)
    VICProtection =  0;                                             (3)

    VICVectAddr0  = (CPU_INT32U)VIC_DummyWDT;                       (4)
    VICVectAddr1  = (CPU_INT32U)VIC_DummySW;
    …
    VICVectAddr31  = (CPU_INT32U)VIC_DummyI2S ;
}
```

L3-5(1) Clear any pending interrupts at the VIC level.

L3-5(2) Acknowledge any pending interrupts to reset the VIC priority hardware.

L3-5(3) Disable VIC protection. Allow access in all ARM processor modes.

L3-5(4) Initialize all VIC vectors to a dummy ISR handler until modified by user software. Uninitialized spurious interrupts will be trapped in `VIC_Dummy()` with variable `VIC_SpuriousInt` containing the interrupt vector number of the source interrupt.

Listing 3-6 `OS_CPU_ExceptHndlr()`

```
void  OS_CPU_ExceptHndlr (CPU_DATA ID) (void)
{
    PFNCT  pfnct;

                                                                    (1)
    if ((ID == OS_CPU_ARM_EXCEPT_IRQ) || (ID == OS_CPU_ARM_EXCEPT_FIQ)) {
        pfnct = (PFNCT)VICAddress;                                  (2)
        if (pfnct != (PFNCT)0) {                                    (3)
          (*pfnct)();                                               (4)

            VICAddress  =  0;                                       (5)
        }
    }
}
```

L3-6(1) Check if the interrupt is due to an IRQ or FIQ exception.

L3-6(2) Read the active interrupt source vector number from the VIC.

L3-6(3) Ensure that the function pointer is not NULL.

L3-6(4) Call the user ISR handler function.

L3-6(5) Acknowledge the VIC interrupt and update the VIC priority hardware.

4.00 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 4-1. This is done via an add-on module called **μC/OS-View**.

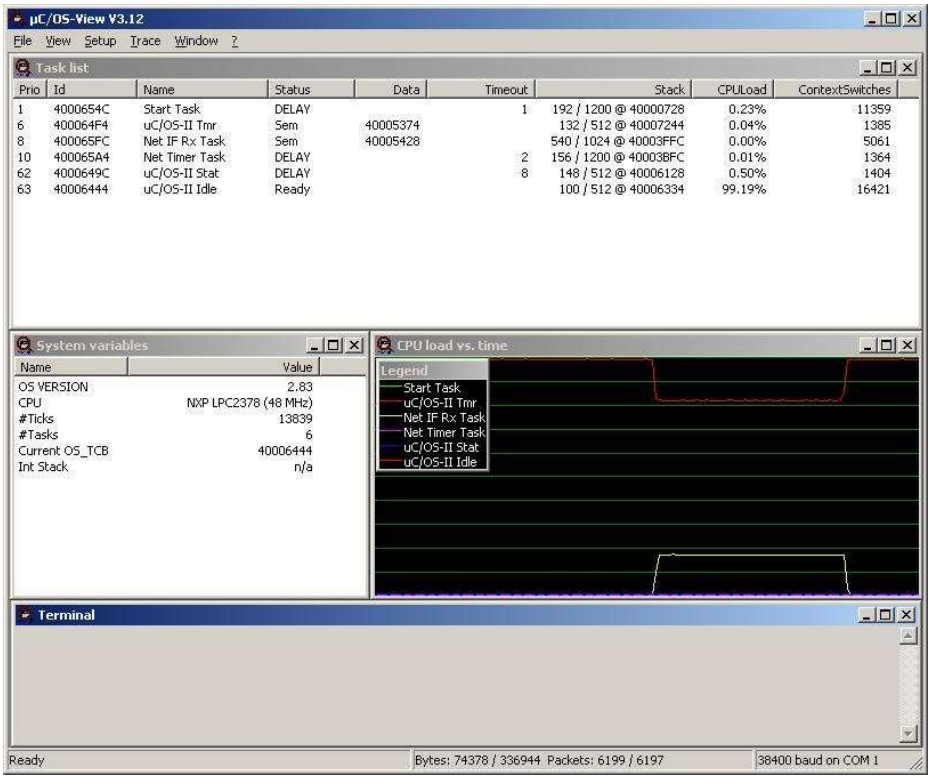

Figure 4-1, μC/OS-View Windows' 'Viewer'

If you purchased **μC/OS-View** from Micriμm, you can 'enable' it by adding the **μC/OS-View** files to the build and setting the `OS_VIEW_MODULE` variable defined in `os_cfg.h` to 1.

μC/OS-View is a combination of a Microsoft Windows application program and code that resides in your target system (in this case, the LPC2378 Evaluation Board). The Windows application connects with your system via an RS-232C serial port (we used UART1 of the LPC2378). 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 (up to 253 tasks);
- The status (e.g., 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'; and
- The execution profile of each task.

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 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.

5.00 Board Support Package, net_bsp.c

The source code located within `net_bsp.c` is mainly responsible for configuring the hardware pins connecting the LPC2378 and the on board National DP83848 PHY. However, as you will see, other small utility functions are provided as well.

Listing 5-1, NetBSP_Phy_HW_Init()

```
void NetBSP_Phy_HW_Init (void)
{
#if EMAC_CFG_RMII                                                    (1)
    PINSEL2             =   0x50151105;
    PINSEL3             =   0x00000005;
#else
    PINSEL2             =   0x55555555;
    PINSEL3             =   0x00000005;
#endif
}
```

L5-1(1) If `EMAC_CFG_RMII` in `net_bsp_h` is defined greater than 1, then the PHY will operate in RMII mode, otherwise if defined to 0, additional I/O pins will be configured for MII mode.

Note: The LPC2378 only supports RMII mode. However, this configuration option has been included in order to provide support for future LPC23xx derivatives that support MII. Users should configure `EMAC_CFG_RMII` to 1 at all times while using the LPC2378.

Listing 5-2, NetBSP_DlyMs()

```
void  NetBSP_NIC_PhyRdWrDly (CPU_INT32U ms)
{
    OSTimeDlyHMSM(0, 0, 0, ms);                                    (1)
}
```

`NetBSP_DlyMs()` is called by `NetNIC_PhyRegRd()`, `NetNIC_PhyRegWr()`, `EMAC_Init()`, `NetNIC_PhyInit()`, and `NetNIC_PhyAutoNeg()` every time a PHY register needs to be read or written. This function creates a delay of `ms` such that there is enough time for the register reads and writes to complete. The calling function uses this as way of determining whether a read or write failed due to a timeout. This is a user specified function which must be implemented. In this case, we use the built in `OSTimeDlyHMSM()` function of **μC/OS-II**, however, a delay created by any means is acceptable.

Listing 5-3, Time Stamp Functions

```
NET_TS                NetUtil_TS_Get            (void);
void                  NetTCP_InitTxSeqNbr       (void);
NET_TCP_TX_RTT_TS_MS  NetTCP_TxRTT_GetTS        (void);
NET_TCP_TX_RTT_TS_MS  NetTCP_TxConnRTT_GetTS_ms (void);
```

The above functions are used for initializing **μC/TCP-IP** sequence numbers and time stamps. They are also used for getting time stamp values that are used within various **μC/TCP-IP** services. These functions are user defined and must be implemented in `net_bsp.c`. For a full explanation of the above functions, please see the **μC/TCP-IP** manual.

5.01 Board Support Package, net_bsp.h

The purpose of `net_bsp.h` is to provide hardware API function prototypes for use with **μC/TCP-IP** and the associated network interface drivers. In addition to function prototypes, it is not uncommon for `net_bsp.h` to contain configuration parameters necessary for proper setup of the integrated EMAC and the attached PHY. For the LPC2378, configuration options for the MAC address, PHY operating mode (RMII versus MII), PHY address, and EMAC descriptor setup are accessible from within this file. Listing 5-1 describes one correct configuration of these constants, however, many application specific variations exist.

Listing 5-4, Net_BSP Configuration Constants

```
#define  EMAC_CFG_MAC_ADDR_SEL          EMAC_CFG_MAC_ADDR_SEL_CFG     (1)
#define  EMAC_CFG_RMII                  1                             (2)
#define  PHY_ADDR                     0x01                            (3)

#define  EMAC_RX_BUF_SIZE              256                            (4)
#define  EMAC_NUM_RX_DESC               36                            (5)
#define  EMAC_NUM_TX_DESC                4                            (6)
```

L5-4(1) Since the LPC2378 EMAC does not support the automatic loading of the Ethernet MAC address from an external EEPROM, the constant `EMAC_CFG_MAC_ADDR_SEL` must always be configured to `EMAC_CFG_MAC_ADDR_SEL_CFG`. If an EEPROM is to be used for setting the MAC address, then software must read the MAC address from the EEPROM and fill the contents of the `NetIF_MAC_Addr[]` array before calling `Net_Init()` within `app.c`.

L5-4(2) The constant `EMAC_CFG_RMII` is used to configure the MII operating mode for the attached PHY. Note: The LPC2378 does not support MII and this constant must be configured to 1.

L5-4(3) The constant `PHY_ADDR` is used to set the PHY bus address. Consult your PHY documentation and hardware schematics for the correct setting. Since the MCB2300 EVB grounds all but bit 0 of the PHY address pins, the address latched by the National DP83848 PHY after reset is 0x01.

L5-4(4) The constant `EMAC_RX_BUF_SIZE` is used to determine the size of the receive buffers used by the EMAC DMA for the reception of Ethernet frames. This constant may take on any value between 64 and 1536 bytes. Too small or too large of a buffer size may negatively impact performance. Therefore the recommended buffer size is 256 bytes since the constant `NET_BUF_CFG_DATA_SIZE_SMALL` located within `net_cfg.h` has been configured to 256 bytes. Please see the description for `EMAC_NUM_RX_DESC` and the **μC/TCP-IP** manuals explanation of buffer sizes before deciding on a final value for this constant.

L5-4(5) The constant `EMAC_NUM_RX_DESC` is used to determine the number of receive descriptors used by the EMAC while receiving Ethernet frames. Ideally, the greater number of descriptors, the better. However, each descriptor has a corresponding receive buffer of size `EMAC_RX_BUF_SIZE` associated with it. The LPC2378 dedicates 16KB of internal RAM for use with the integrated EMAC DMA functionality. Therefore, all declared receive buffers AND descriptors must fit within the dedicated memory space. Each descriptor, associated status words, and buffers take the following amount of space:

```
(EMAC_NUM_RX_DESC * EMAC_RX_BUF_SIZE)       +
(EMAC_NUM_RX_DESC * 8 bytes per descriptor) +
(EMAC_NUM_RX_DESC * 8 bytes per status)
```

Please keep in mind that there must be space for the transmit descriptors, transmit status words, and transmit buffers within the dedicated EMAC RAM as well.

L5-4(6) The constant `EMAC_NUM_TX_DESC` is used to determine the number of transmit descriptors used by the EMAC and device firmware while transmitting Ethernet frames. More transmit descriptors do not necessarily mean better performance since the EMAC reads frames from memory much faster than the device firmware can produce them. Therefore, a value of 4 is recommended. The amount of RAM consumed by the transmit descriptors, status words and buffers is calculated as follows:

```
(EMAC_NUM_TX_DESC * 1536 bytes per frame)   +
(EMAC_NUM_TX_DESC * 8 bytes per descriptor) +
(EMAC_NUM_TX_DESC * 4 bytes per status)
```

All transmit descriptors have an associated 1536 byte (non configurable) buffer size. The reason for this is because **μC/TCP-IP** allocates its own internal buffers for storing frame data before the driver is called upon to transmit the frame. Therefore, the driver must be prepared to accept a 1536 byte buffer to be transmitted.

Please keep in mind that there must be space for the receive descriptors, receive status words, and receive buffers within the dedicated EMAC RAM as well.

In order to facilitate the configuration of the above constants, configuration checking within `net_nic.h` prevents improper configuration of the above constants should the allocated resources overflow their dedicated memory space.

6.00 EMAC Notes

1) Since the LPC2378 can only DMA Ethernet frames to and from the dedicated 16KB EMAC RAM, **μC/TCP-IP** is unable to utilize DMA functionality for transmission and receive DMA transfers are not currently supported. Therefore all EMAC transactions and receptions on the LPC2378 require a full frame copy to and from the EMAC dedicated memory space into the **μC/TCP-IP** buffers.

2) Limited support for the National DP83848 PHY has been provided with this example. Future functionality may support the use of PHY interrupts in order to detect Ethernet link state changes during run-time. Currently, the EMAC driver, `net_nic.c` assumes that the cable is plugged in. However, if the user application needs to learn of the current link state, a call to `NetNIC_ConnStatusGet()` may be performed. This function returns 0 when the link is down, otherwise 10, or 100 depending on the current link speed.

Licensing

μC/OS-II is provided in source form for **FREE** evaluation, for educational use or for peaceful research. If you plan on using **μC/OS-II** in a commercial product you need to contact Micriμm to properly license its use in your product. We provide **ALL** the source code with this application note for your convenience and to help you experience **μC/OS-II**. The fact that the source is provided does **NOT** mean that you can use it without paying a licensing fee. Please help us continue to provide the Embedded community with the finest software available. Your honesty is greatly appreciated.

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.** 1601 W. 23rd St., Suite 200 Lawrence, KS 66046-9950 USA +1 785 841 1631 +1 785 841 2624 (FAX) e-mail: rushorders@cmpbooks.com WEB: http://www.cmpbooks.com
Micriμm 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	**NXP** 1110 Ringwood Court San Jose, CA 95131 USA +1 408 474 8142 WEB: www.nxp.com