

Micrium

Empowering Embedded Systems

μC/OS-II

μC/Probe

and the

NXP LPC2378 Processor

(Using the IAR LPC2378-SK Evaluation Board)

Application Note

AN-1077

www.Micrium.com

About Micrium

Micrium provides high-quality embedded software components in the industry by way of engineer-friendly source code, unsurpassed documentation, and customer support. The company's world-renowned real-time operating system, the Micrium **μC/OS-II**, features the highest-quality source code available for today's embedded market. Micrium delivers to the embedded marketplace a full portfolio of embedded software components that complement **μC/OS-II**. A TCP/IP stack, USB stack, CAN stack, File System (FS), Graphical User Interface (GUI), as well as many other high quality embedded components. Micrium's products consistently shorten time-to-market throughout all product development cycles. For additional information on Micrium, please visit www.micrium.com.

About μC/OS-II

μC/OS-II is a preemptive, real-time, multitasking kernel. **μC/OS-II** has been ported to over 45 different CPU architectures.

μC/OS-II is small yet provides all the services you'd expect from an RTOS: task management, time and timer management, semaphore and mutex, message mailboxes and queues, event flags and much more. You will find that **μC/OS-II** delivers on all your expectations and you will be pleased by its ease of use.

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 Micrium 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.

About **μC/Probe**

μC/Probe is a Windows application that allows a user to display the value (at run-time) of virtually any variable or memory location on a connected embedded target. The user simply populates **μC/Probe**'s graphical environment with gauges, tables, graphs, and other components, and associates each of these with a variable or memory location. Once the application is loaded onto the target, the user can begin **μC/Probe**'s data collection, which will update the screen with variable values fetched from the target.

μC/Probe retrieves the values of global variables from a connected embedded target and displays the values in a engineer-friendly format. The supported data-types are: booleans, integers, floats and ASCII strings.

μC/Probe can have any number of 'data screens' where these variables are displayed. This allows to logically group different 'views' into a product.

A 30-day trial version of **μC/Probe** is available on the Micrium website:

<http://www.micrium.com/products/probe/probe.html>

Manual Version

If you find any errors in this document, please inform us and we will make the appropriate corrections for future releases.

Version	Date	By	Description
V.1.00	2007/03/13	BAN	Initial version.
V.1.01	2007/08/24	BAN	Updated with µC/Probe.

Software Versions

This document may or may not have been downloaded as part of an executable file, *Micrium-NXP-uCOS-II-LPC2378-SK.exe*, containing the code and projects described here. If so, then the versions of the Micrium software modules in the table below would be included. In either case, the software port described in this document uses the module versions in the table below

Module	Version	Comment
µC/OS-II	V2.85	ARM port v1.82
µC/Probe	V1.30	
µC/LCD	V3.00	

Document Conventions

Numbers and Number Bases

- Hexadecimal numbers are preceded by the “0x” prefix and displayed in a monospaced font. Example: 0xFF886633.
- Binary numbers are followed by the suffix “b”; for longer numbers, groups of four digits are separated with a space. These are also displayed in a monospaced font. Example: 0101 1010 0011 1100b.
- Other numbers in the document are decimal. These are displayed in the proportional font prevailing where the number is used.

Typographical Conventions

- Hexadecimal and binary numbers are displayed in a monospaced font.
- Code excerpts, variable names, and function names are displayed in a monospaced font. Functions names are always followed by empty parentheses (e.g., OS_Start()). Array names are always followed by empty square brackets (e.g., BSP_Vector_Array[]).
- File and directory names are always displayed in an italicized serif font. Example: */Micrium/Software/uCOS-II/Source/*.
- A bold style may be layered on any of the preceding conventions—or in ordinary text—to more strongly emphasize a particular detail.
- Any other text is displayed in a sans-serif font.

Table of Contents

1.	Introduction	7
2.	Getting Started	9
2.01	Setting up the Hardware	9
2.02	Opening and Viewing the Project	10
2.03	Using the IAR Projects	11
2.03.01	Project Options	11
2.03.02	µC/OS-II Kernel Awareness	13
2.04	Example Application	15
2.04.01	Application Information	15
2.04.02	Additional Application Information	17
3.	Directories and Files	19
4.	Application Code	22
4.01	<i>app.c</i>	22
4.02	<i>os_cfg.h</i>	25
5.	Board Support Package (BSP)	27
5.01	IAR EWARM v4.4x-Specific BSP Files	27
5.02	IAR EWARM v5.1x-Specific BSP Files	27
5.03	RVMDK-Specific BSP Files	27
5.04	BSP, <i>bsp.c</i> and <i>bsp.h</i>	28
5.05	Processor Initialization Function	29
5.06	LCD Driver, <i>glcd.c</i> and <i>glcd.h</i>	31
6.	µC/Probe	32
	Licensing	35
	References	35
	Contacts	35

1. Introduction

This document, *AN-1077*, explains example code for using **μC/OS-II** and **μC/Probe** with the NXP LPC2378 (ARM7TDMI-S) processor on the IAR LPC2378-SK Evaluation Board, as shown in Figure 1-1. The LPC2378 includes a 512-kB flash and 32-kB SRAM and operates at clock speeds as high as 72-MHz. Peripherals for several communications busses are provided on-chip, including UARTs, I²C, I²S, SPI, SSP, CAN, USB and Ethernet. A SD/MMC card interface, a 10-bit A/D converter, a 10-bit D/A converter, four 32-bit general-purpose timers and up to 104 GPIOs round out the features on the chip.

The IAR LPC2378-SK provides two user push buttons, a joystick, a potentiometer, a 132- x 132-pixel LCD, a microphone input and a headphone output. Two RS-232 ports, one USB device port, several CAN interfaces and one Ethernet port provide for external communication. The board as tested incorporates an on-board J-Link, interfaced using a standard USB cable without need for the external debugger hardware; however, another version of the board is also available with a standard 20-pin JTAG connector.

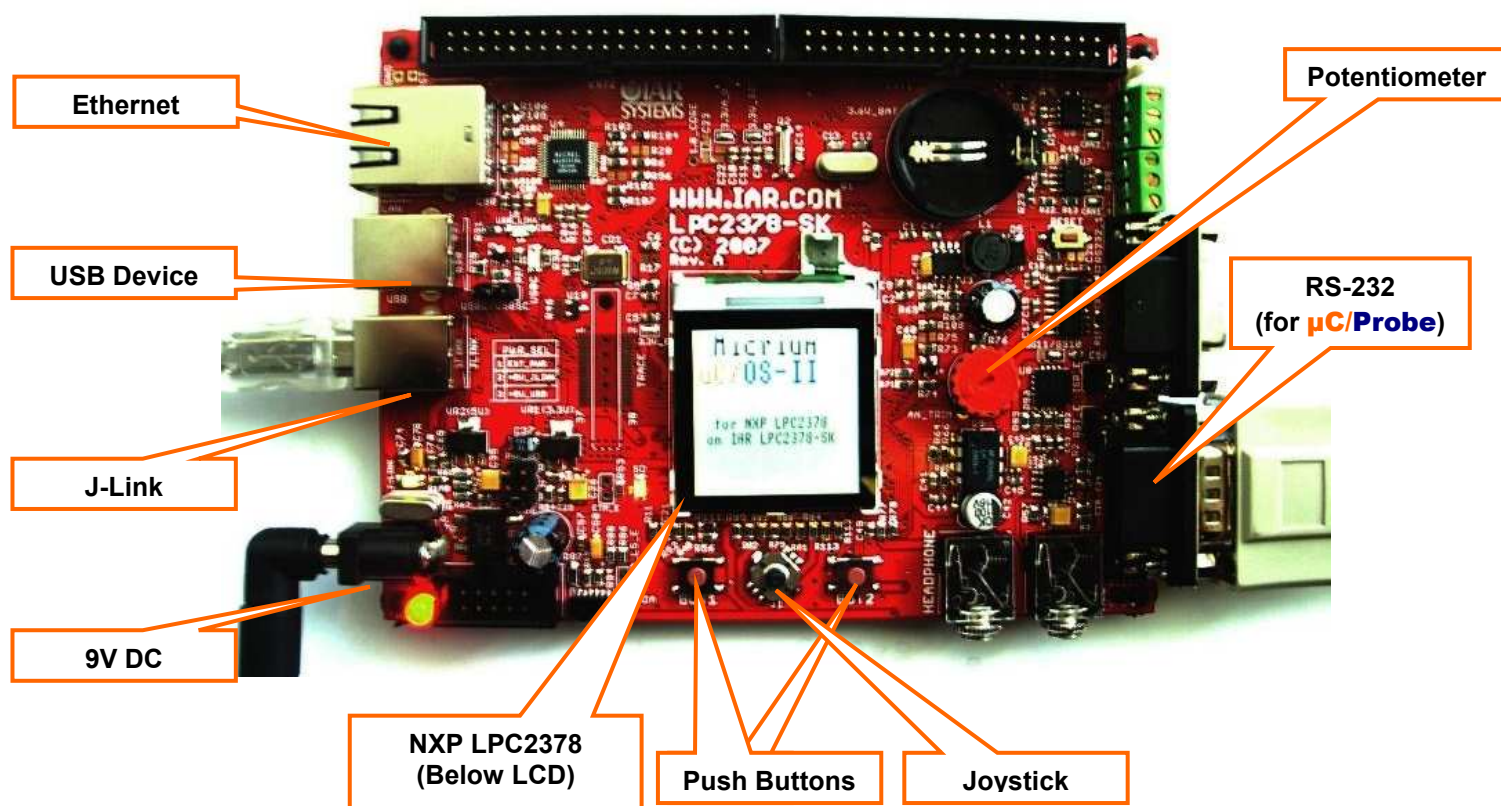


Figure 1-1. IAR LPC2378-SK Evaluation Board

If this appnote was downloaded in a packaged executable zip file, then it should have been found in the directory */Micrium/Appnotes/AN1xxx-RTOS/AN1077-uCOS-II-NXP-LPC2378-SK* and the code files referred to herein are located in the directory structure displayed in Section 2.02; these files are described in Section 3.

The executable zip also includes example workspaces for **μC/Probe**. **μC/Probe** is a Windows program which retrieves the value of variables from a connected embedded target and displays the values in an engineer-friendly format. It interfaces with the LPC2378 via RS-232C. For more information, including instructions for downloading a trial version of the program, please refer to Section 6.

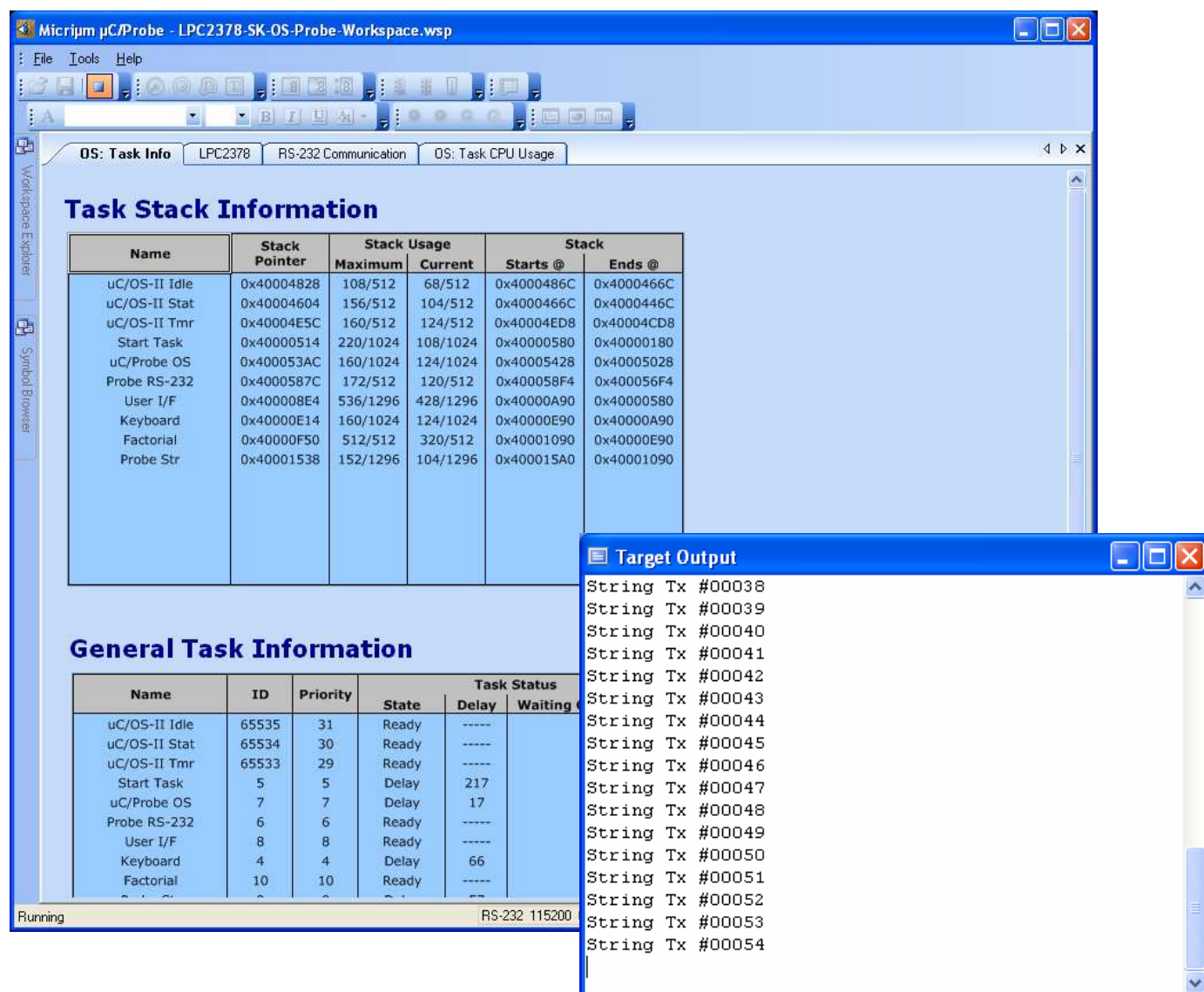


Figure 1-3. **μC/Probe** (with Target Output Window)

2. Getting Started

The following sections step through the prerequisites for using the demonstration application described in this document, *AN-1077*. First, the setup of the hardware will be outlined. Second, the use and setup of the IAR Embedded Workbench project will be described. Thirdly, the steps to build the projects and load the application onto the board through the JTAG will be described. Lastly, instructions will be provided for using the example application.

2.01 Setting up the Hardware

The power selection jumper, as shown in Figure 1-2, should be set for the desired power source. The board can be powered either through the USB, the J-Link, or an external 9 V DC source. The board we tested integrated the J-Link hardware; consequently, the processor was programmed and debugged through this port. Another version of the board is available that includes a standard 20-pin JTAG port to which an external debugger would be connected (such as a J-Link).

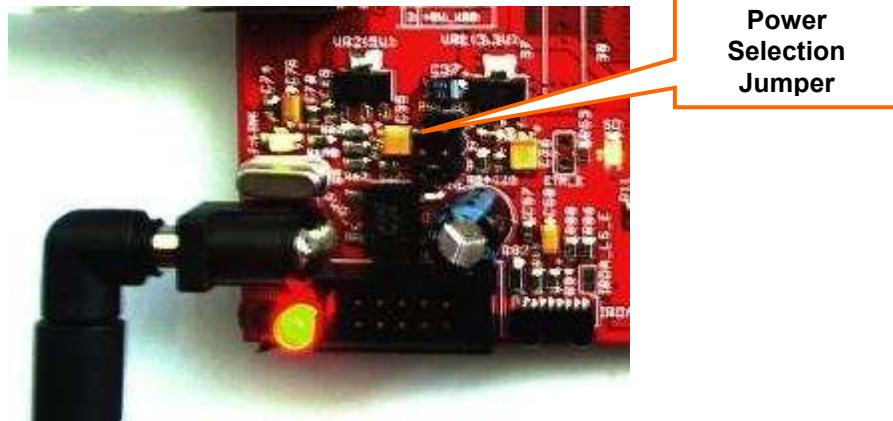


Figure 2-1. Power Selection Jumper

To use μ C/Probe with the LPC2378, download and install the trial version of the program from the Micrium website as discussed in Section 6. After programming your target with one of the included example projects, connect a RS-232 cable between your PC and the evaluation board, configure the RS-232 options (also covered in Section 6), and start running the program. The open data screens should update, as shown in Figure 1-2. The LPC2378 example application is configured to use UART0, the RS-232C connector labeled "RS-232 for μ C/Probe" in Figure 1-1.

2.02 Opening and Viewing the Project

If this file were downloaded as part of an executable zip file (which should have been named *Micrium-NXP-uCOS-II-LPC2378-SK.exe*), then the code files referred to herein are located in the directory structure shown in Figure 2-3.

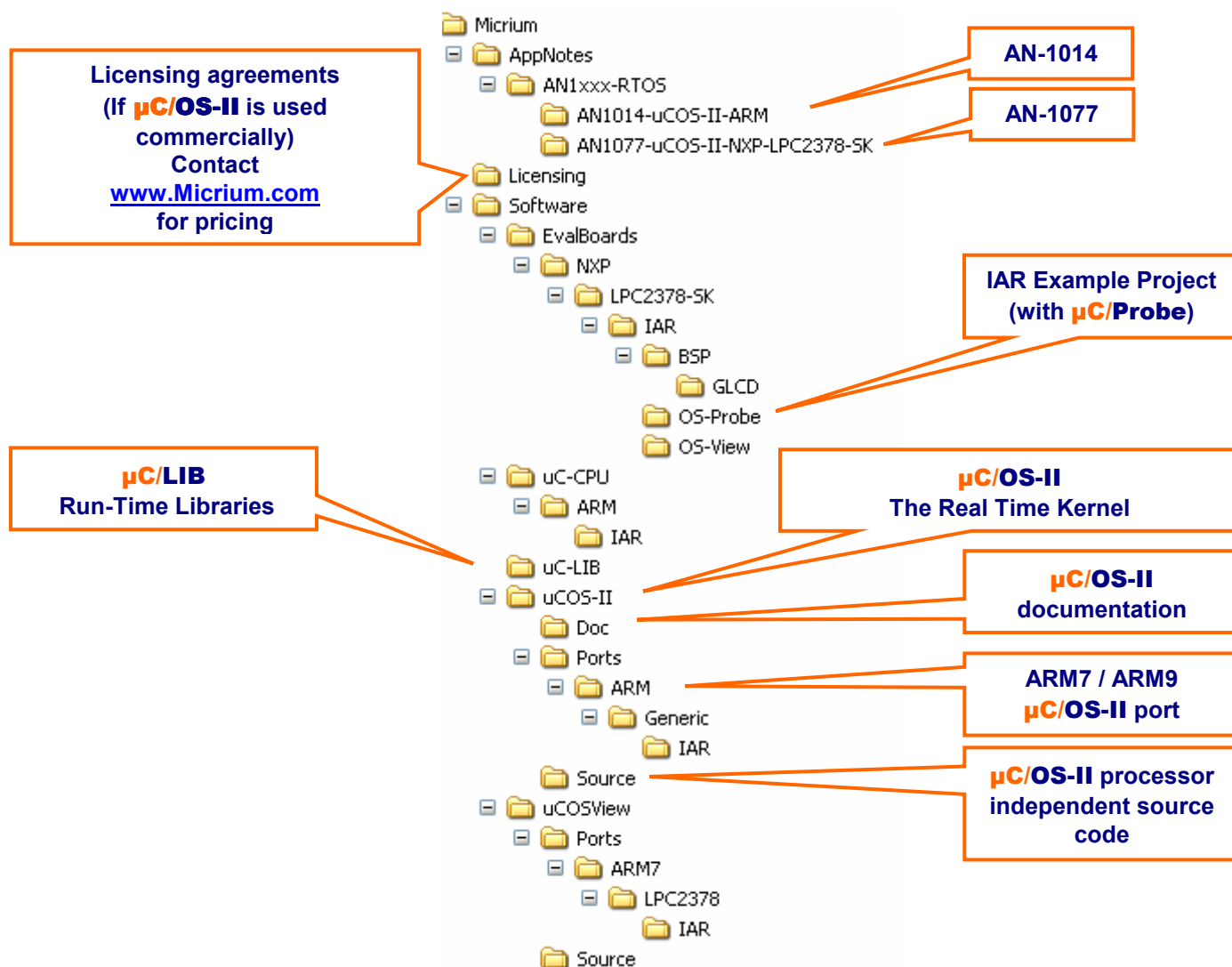


Figure 2-2. Directory Structure

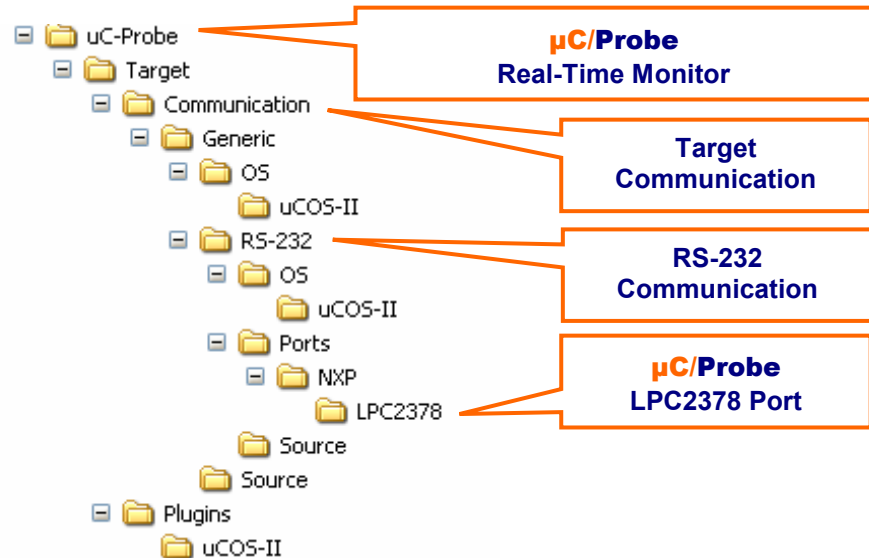


Figure 2-2. Directory Structure (continued)

2.03 Using the IAR Projects

Two IAR projects are located in the directory (marked “IAR Example Project (with μC/Probe)” in Figure 2-3):

/Micrium/Software/EvalBoards/NXP/LPC2378-SK/IAR/OS-Probe

The first example project, *LPC2378-SK-OS-Probe.ewp*, is intended for EWARM v4.4x. To view this example, start an instance of IAR EWARM v4.4x, and open the workspace file *LPC2378-SK-OS-Probe.eww*. To do this, select the “Open” menu command under the “File” menu, select the “Workspace...” submenu command and select the workspace file after navigating to the project directory. The project tree shown in Figure 2-4 should appear. (In addition, the workspace should be openable by double-clicking on the file itself in a Windows Explorer window.)

The second example project, *LPC2378-SK-OS-Probe-v5.ewp*, is intended for EWARM v5.1x. To view this example, start an instance of IAR EWARM v5.1x, and open the workspace file *LPC2378-SK-OS-Probe-v5.eww*. To do this, select the “Open” menu command under the “File” menu, select the “Workspace...” submenu command and select the workspace file after navigating to the project directory. The project tree for this project will be essentially identical to the project tree for the EWARM v4.4x project.

IAR EWARM Versions

Be certain to open the proper project for your version of EWARM. IAR EWARM v4.4x will NOT open a v5.1x project. And though IAR EWARM v5.1x will open a v4.4x project, many errors will be generated upon compilation.

2.03.01 Project Options

The IAR projects are setup to compile in ARM mode. However, both could be re-configured to generate 16-bit Thumb instructions wherever possible, thereby reducing code size. This setting may be changed by opening the project settings dialog box. To display this dialog box, choose the “Options” menu item from

the “Project” menu. The location of the configuration option within this dialog is different for EWARM v4.4x and EWARM v5.1x; Figures 2-3 highlight this location for both toolchain versions.

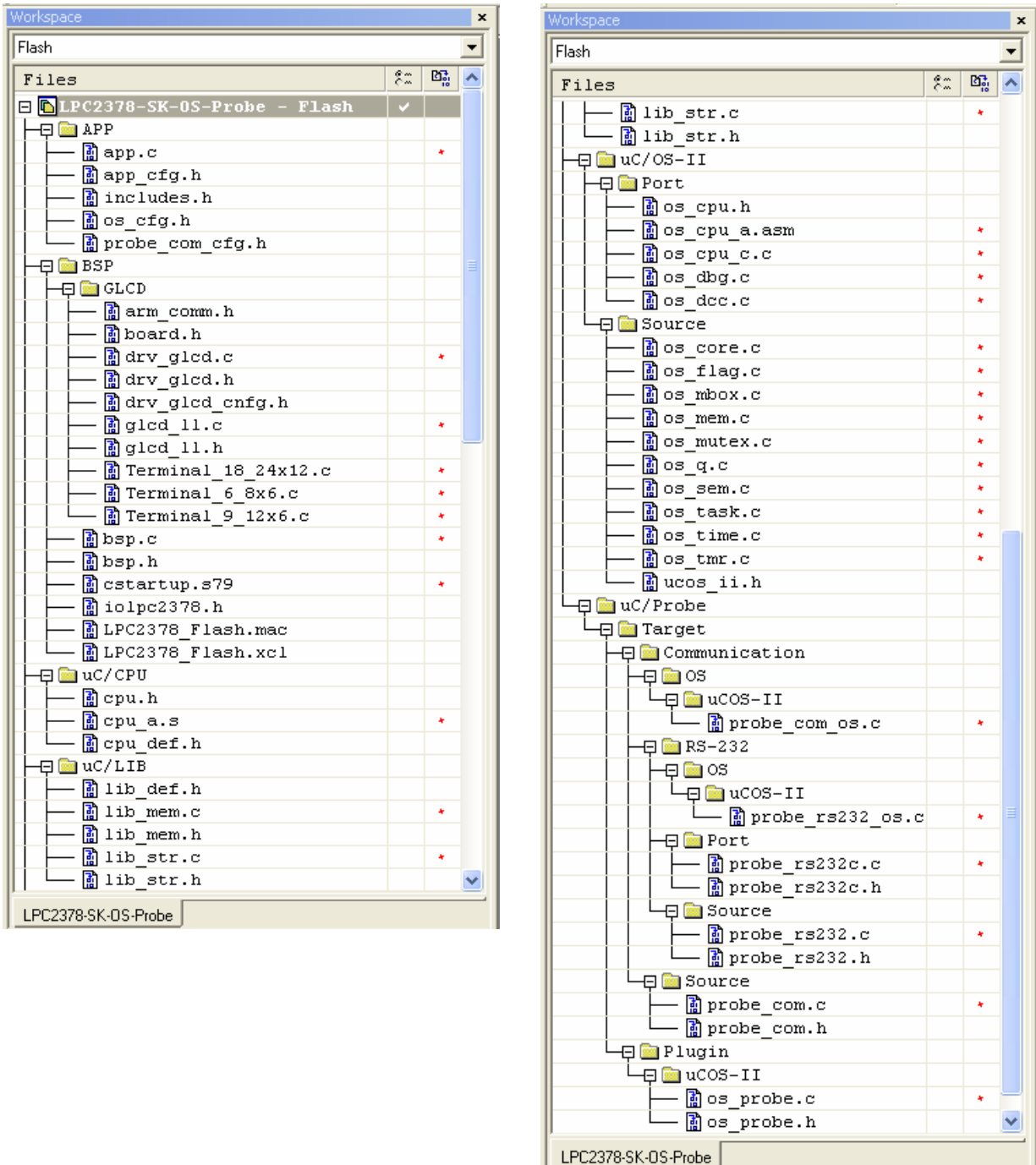


Figure 2-3. IAR EWARM Project Tree for *LPC2378-SK-OS-Probe.ewp*.

Once the connections described in Section 2.01 are made between your PC and the LPC2378-SK Evaluation Board, the code can be built and loaded onto the board. To build the code, choose the “Rebuild All” menu item from the “Project” menu. To load the code through the J-TAG debugger onto the

connected evaluation board, select the “Debug” menu item from the “Project” menu. The project is setup to use a J-Link debugger; if you wish to use a different debugger, please select the appropriate DLL in the project options dialog box (select “Debugger” in the list box).

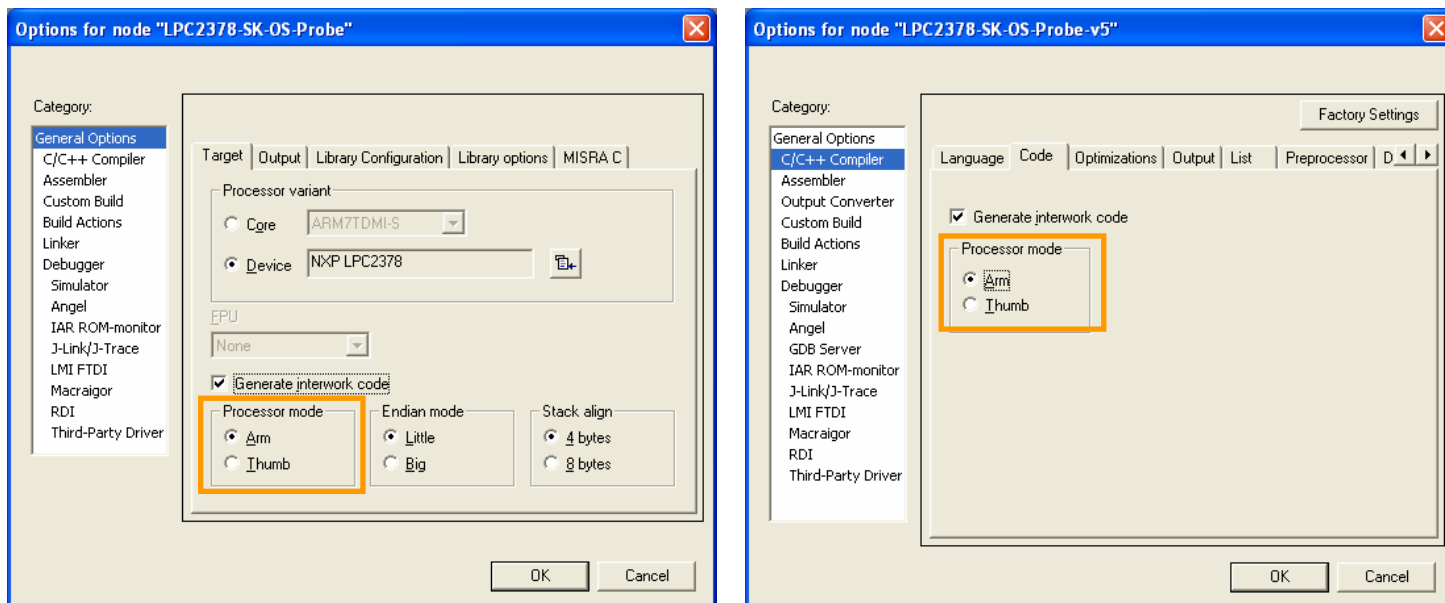


Figure 2-4. Project Options: ARM/Thumb Selection.
 EWARM v4.4x (left); EWARM v5.1x (right)

2.03.02 μC/OS-II Kernel Awareness

When running the IAR C-Spy debugger, the μC/OS-II Kernel Awareness Plug-In can be used to provide useful information about the status of μC/OS-II objects and tasks. If the μC/OS-II Kernel Awareness Plug-In is currently enabled, then a “μC/OS-II” menu should be displayed while debugging. Otherwise, the plug-in can be enabled. Stop the debugger (if it is currently active) and select the “Options” menu item from the “Project” menu. Select the “Debugger” entry in the list box and then select the “Plugins” tab pane. Find the μC/OS-II entry in the list and select the check box beside the entry, as shown in Figure 2-4.

When the code is reloaded onto the evaluation board, the “μC/OS-II” menu should appear. Options are included to display lists of kernel objects such as semaphores, queues, and mailboxes, including for each entry the state of the object. Additionally, a list of the current tasks may be displayed, including for each task pertinent information such as used stack space, task status, and task priority, in addition to showing the actively executing task. An example task list for this project is shown in Figure 2-5.

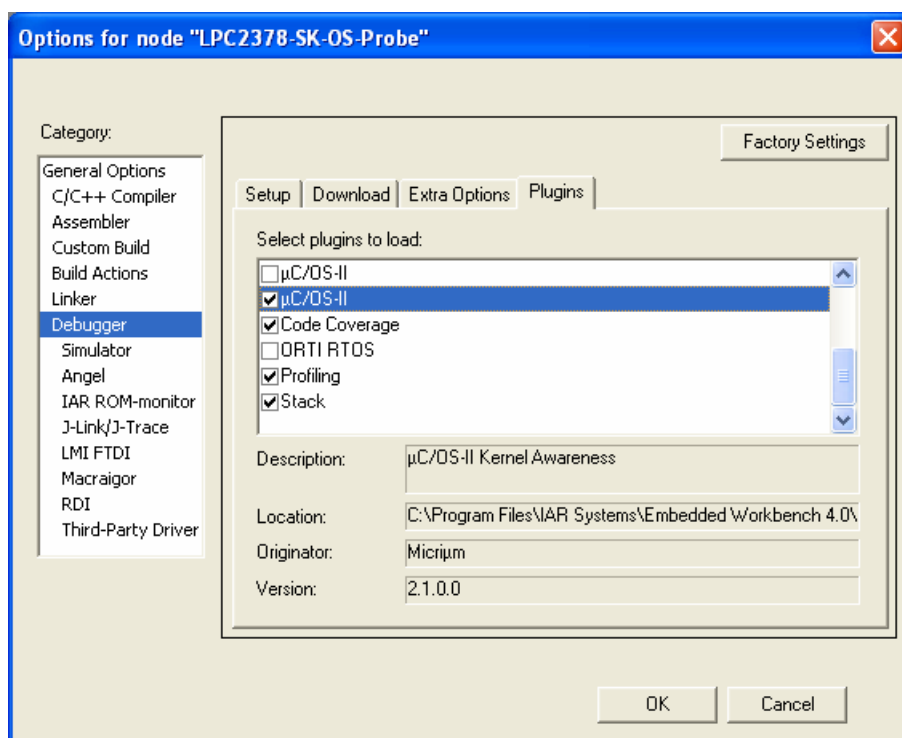


Figure 2-5. Enabling the µC/OS-II Kernel Awareness Plug-In (v4.4x)

Task List																
Name	Ref	Prio	State	Dly	Waiting On	Msg	Ctx Sw	Stk Ptr	Max%	Cur%	Max	Cur	Size	Starts @	Ends @	
Keyboard	7	4	Dly	4			49092	40000CEC	12%	9%	124	100	1024	40000D50	40000950	
Start Task	3	5	Dly	10			106724	400003D8	16%	10%	168	104	1024	40000440	40000040	
uC/Probe OS	4	7	Dly	4			49092	40002A80	13%	10%	136	112	1024	40002AF0	400026F0	
User I/F	6	8	Mbox	5	?		1251068	400008E0	12%	8%	168	112	1296	40000950	40000440	
Probe Str	8	9	Dly	4			24483	40001214	11%	9%	144	120	1296	4000128C	40000D7C	
Probe RS-232	5	10	Sem	0	Probe RS-232		1	40002FB4	25%	21%	132	108	512	40003020	40002E20	
uC/OS-II Tmr	2	29	Sem	0	OS-TmrSig		24547	400025B8	25%	21%	132	108	512	40002624	40002424	
uC/OS-II Stat	1	30	Dly	2			24546	4000168C	23%	18%	120	96	512	400016EC	400014EC	
> uC/OS-II Idle	0	31	Ready	0			1349672	400018B0	20%	13%	104	68	512	400018F4	400016F4	

Figure 2-6. µC/OS-II Task List for *LPC2378-SK-OS-Probe.ewp*

2.04 Example Application

The example applications contain application tasks which respond to the push buttons and update the LCD. In addition, either can be used with the Micrium's real-time monitor, **μC/Probe**, as covered in Section 6.

2.04.01 Application Information

Once the program is loaded onto the target, the start-up screen, as shown in Figure 2-9, will be displayed on the LCD. After two seconds, the LCD will switch to the first of a series of screens providing information about **μC/OS-II**, the LPC2378 platform, and the tasks managed by **μC/OS-II**. The additional screens, as shown in Figures 2-10 through 2-14, are accessible in two ways. Firstly, pushing the left and right push buttons (PB1 and PB2) will cause the screen number to decrement or increment, respectively. Secondly, toggling the joystick left or right will cause the screen number to decrement or increment, respectively.

Toggling the joystick up or down will affect the vertical scroll of the display. The brightness of the display is determined by the setting of the potentiometer; turning the knob counter-clockwise causes the screen to grow brighter.

Stack Out of Range Notification

While debugging this project (or any other **μC/OS-II** project), IAR may log a SVC stack pointer out-of-range notification in the "Debug Log" window. This is actually normal behavior and does **NOT** indicate an error. IAR EWARM does not understand that the SVC stack pointer points to the stack for the current task stack.



Figure 2-7. Start-up Screen
This screen is displayed when the application starts.

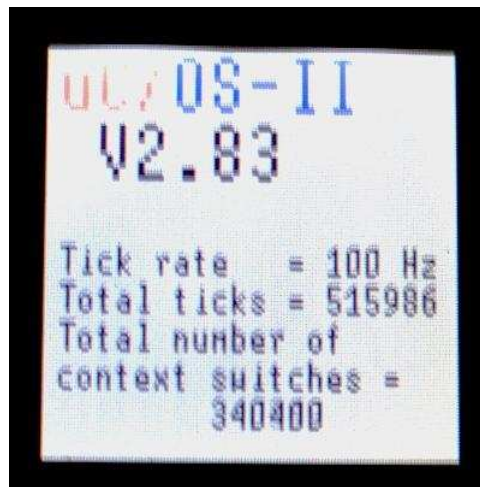


Figure 2-8. μC/OS-II Information
This screen lists the tick rate, the total number of ticks, and the total number of context switches.

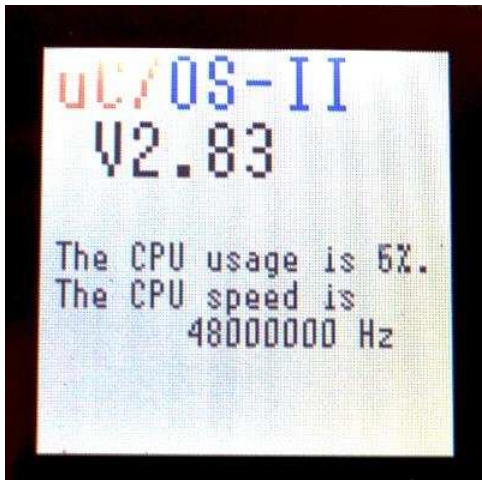


Figure 2-9. CPU Usage & Speed
This screen displays the percent CPU usage and CPU speed.

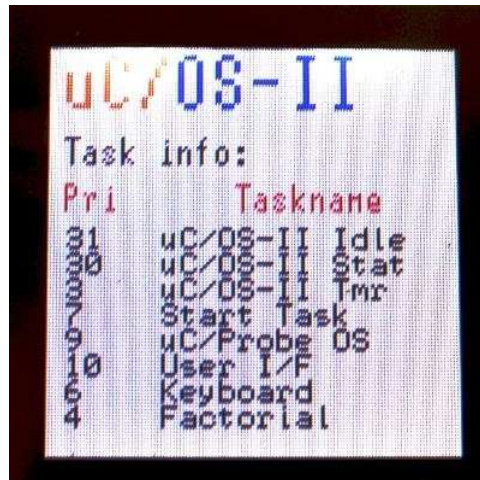


Figure 2-10. μC/OS-II Tasks
This screen lists the μC/OS-II task priorities and task names.

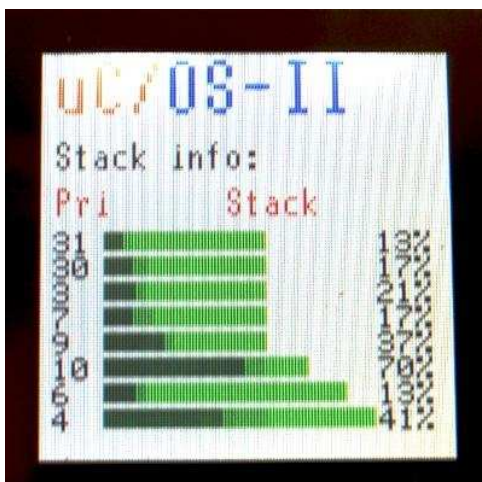


Figure 2-11. μC/OS-II Task Stacks
This screen displays, as a horizontal bar graph, the task stack usage for each task. Light green (■) denotes the total stack space. Medium green (■) denotes the maximum stack usage. Dark green (■) denotes the current stack usage.

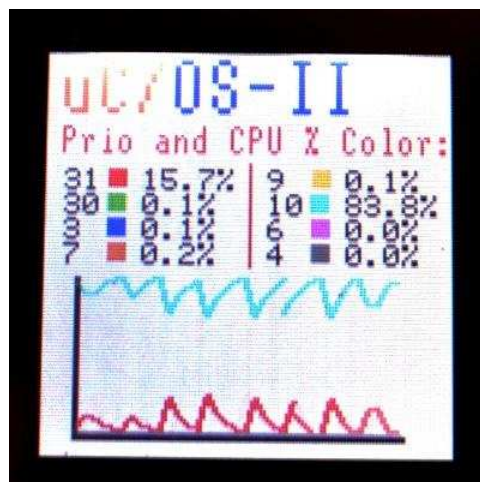


Figure 2-12. μC/OS-II CPU Usage
This screen graphs the CPU usage for each task versus time. The key indexes the graph colors by the task priority (see Figure 2-12). The two most evident series are the user interface task (priority 10, cyan) and the idle task (priority 31, red).

2.04.02 Additional Application Information

The project is configured so that code is loaded into Flash and the stacks and data are loaded into RAM, as shown in Table 2-1. The tasks that run in the example application are listed in Table 2-2.

Memory Range	Size	Segment(s)
0x00000000–0x0007CFFF	500 kB	Code (in Flash)
0x40000000–0x4007FFF	32 kB	Stacks and data (in RAM)

Table 2-1. Memory Setup

Task Name	Priority	Function
AppTaskKbd() "Keyboard"	4	Reads status of push buttons and joystick, passing new input to AppTaskUserIF().
AppTaskStart() "Start Task"	5	Initializes μ C/Probe; reads potentiometer input to update the LCD backlight level; updates array that holds graph information.
"Probe RS-232"	6	Parses packets from μ C/Probe.
"uC/Probe OS"	7	Updates CPU usage for μ C/Probe.
AppTaskUserIF() "User I/F"	8	Updates LCD.
AppTaskProbeStr() "Probe Str"	9	Outputs strings to the Windows μ C/Probe program.
AppTaskFactorial() "Factorial"	10	Provides a task with variable stack usage; recursively calculates the factorial of small integers
"uC/OS-II Tmr"	29	Manages timers.
"uC/OS-II Stat"	30	Collect stack usage statistics.
"uC/OS-II Idle"	31	Executes when no other task is executing.

Table 2-2. Example Application Tasks

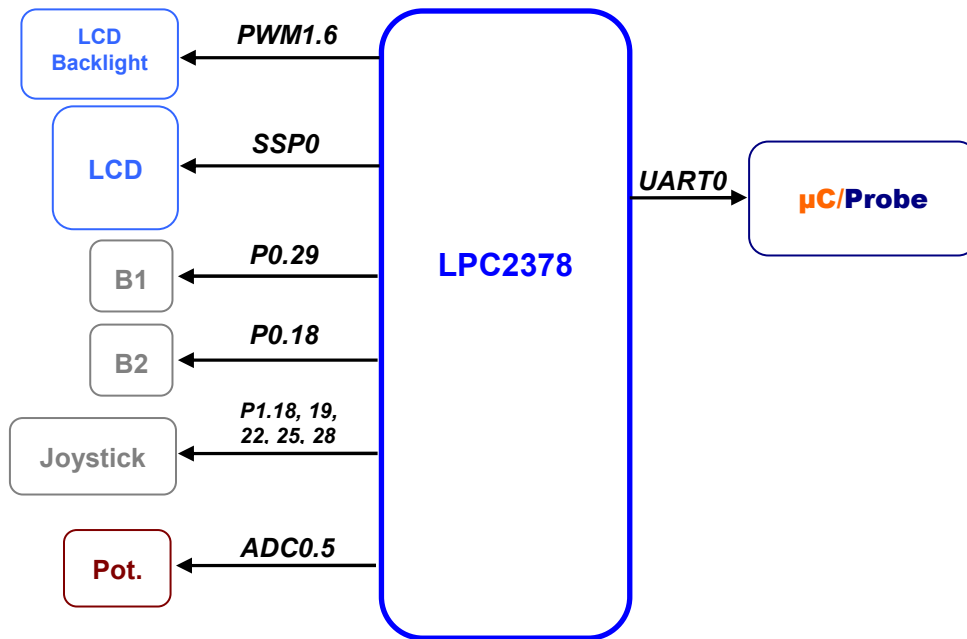


Figure 2-13. Example Application Hardware Use

3. Directories and Files

Application Notes

`\Micrium\AppNotes\AN1xxx-RTOS\AN1014-uCOS-II-ARM`

This directory contains *AN-1014.pdf*, the application note describing the ARM port for μC/OS-II, and *AN-1014-PPT.pdf*, a supplement to *AN-1014.pdf*.

`\Micrium\AppNotes\AN1xxx-RTOS\AN1077-uCOS-II-NXP-LPC2378-SK`

This directory contains this application note, *AN-1077.pdf*.

Licensing Information

`\Micrium\Licensing`

Licensing agreements are located in this directory. Any source code accompanying this appnote is provided for evaluation purposes only. If you choose to use μC/OS-II in a commercial product, you must contact Micrium regarding the necessary licensing.

μC/OS-II Files

`\Micrium\Software\uCOS-II\Doc`

This directory contains documentation for μC/OS-II.

`\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 toolchain. These files could easily be modified to work with other toolchains (i.e., compiler/assembler/linker/locator/debugger); however, the modified files should be placed into a different directory. The following files are in this directory:

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

With this port, μC/OS-II can be used 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 (as discussed in Section 2.30) to generate ARM-mode code without needing to change either the port or the application code. The ARM/Thumb port is described in application note *AN-1014* which is available from the Micrium web site.

`\Micrium\Software\uCOS-II\Source`

This directory contains the processor-independent source code for μC/OS-II.

μC/Probe Files

`\Micrium\Software\uC-Probe\Communication\Generic\`

This directory contains the μC/Probe generic communication module, the target-side code responsible for responding to requests from the μC/Probe Windows application (including requests over RS-232).

`\Micrium\Software\uC-Probe\Communication\Generic\Source`

This directory contains *probe_com.c* and *probe_com.h*, the source code for the generic communication module.

\Micrium\Software\uC-Probe\Communication\Generic\OS\uCOS-II

This directory contains *probe_com_os.c*, which is the µC/OS-II port for the µC/Probe generic communication module.

\Micrium\Software\uC-Probe\Communication\Generic\Source\RS-232

This directory contains the RS-232 specific code for µC/Probe generic communication module, the target-side code responsible for responding to requests from the µC/Probe Windows application over RS-232

\Micrium\Software\uC-Probe\Communication\Generic\Source\RS-232\Source

This directory contains *probe_rs232.c* and *probe_rs232.h*, the source code for the generic communication module RS-232 code.

\Micrium\Software\uC-Probe\Communication\Generic\Source\RS-232\Ports\NXP\LPC2378

This directory contains *probe_rs232c.c* and *probe_rs232c.h*, the NXP LPC2378 port for the RS-232 communications.

\Micrium\Software\uC-Probe\Communication\Generic\Source\RS-232\OS\uCOS-II

This directory contains *probe_rs232_os.c*, which is the µC/OS-II port for the µC/Probe RS-232 communication module.

µC/OS-View Files

\Micrium\Software\uCOSView\Source

This directory contains the processor-independent code for µC/OS-View:

- *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*

µC/CPU Files

\Micrium\Software\uC-CPU

This directory contains *cpu_def.h*, which declares #define constants for CPU alignment, endianness, and other generic CPU properties.

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

This directory contains *cpu.h* and *cpu_a.s*. *cpu.h* defines the Micrium portable data types for 8-, 16-, and 32-bit signed and unsigned integers (such as CPU_INT16U, a 16-bit unsigned integer). These allow code to be independent of processor and compiler word size definitions. *cpu_a.s* contains generic assembly code for ARM7 and 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().

μ C/LIB Files

`\Micrium\Software\uC-LIB`

This directory contains *lib_def.h*, which provides `#defines` for useful constants (like `DEF_TRUE` and `DEF_DISABLED`) and macros.

`\Micrium\Software\uC-LIB\Doc`

This directory contains the documentation for μ C/LIB.

Application Code

`\Micrium\Software\EvalBoards\NXP\LPC2378-SK\IAR\OS-Probe`

This directory contains the source code for the example application:

- *app.c* contains the test code for the example application including calls to the functions that start multitasking within μ C/OS-II, register tasks with the kernel, and update the user interface (the LCD). *app_cfg.h* is a configuration file specifying stack sizes and priorities for all user tasks and `#defines` for important global application constants.
- *includes.h* is the master include file used by the application.
- *os_cfg.h* is the μ C/OS-II configuration file.
- *LPC2378-SK-OS-Probe.wsp* is an example μ C/Probe workspace.
- *LPC2378-SK-OS-Probe.** are the IAR EWARM v4.4x project files.
- *LPC2378-SK-OS-Probe-v5.** are the IAR EWARM v5.1x project files.

`\Micrium\Software\EvalBoards\NXP\LPC2378-SK\IAR\BSP`

This directory contains the Board Support Package for the IAR LPC-P2378-SK evaluation board:

- *bsp.c* contains the board support package functions which initialize critical processor functions (e.g., the PLL) and provide support for peripherals such as the push button and potentiometer. *bsp.h* contains prototypes for functions that may be called by the user.
- *cstartup.s79* is the IAR EWARM v4.4x startup file. This file performs critical processor initialization (such as the initialization of task stacks), readying the platform to enter `main()`.
- *cstartup.s* is the IAR EWARM v5.1x startup file. This file performs critical processor initialization (such as the initialization of task stacks), readying the platform to enter `main()`.
- *LPC2378_Flash.xcl* is a IAR EWARM v4.4x linker file which contains information about the placement of data and code segments in the processor's memory map.
- *LPC2378_Flash.icf* is a IAR EWARM v5.1x linker file which contains information about the placement of data and code segments in the processor's memory map.

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

This directory contains the driver for the LCD provided by IAR with the supporting files containing defines and functions expected by the driver itself. This driver has been lightly modified to allow some functionality to move into the BSP (namely, the LCD backlight control). In addition, two interface functions were added.

- *drv_glcd.c*, *drv_glcd.h*, and *drv_glcd_cfg.h* constitute the core of the driver. *glcd_ll.c* and *glcd_ll.h* provide low-level initialization and access.
- *Terminal_18_24x12.c*, *Terminal_6_8x6.c*, and *Terminal_9_12x6.c* are three bit-mapped fonts that can be used by the LCD driver.
- *arm_comm.h* and *board.h* include defines expected by the driver.

4. Application Code

The example application described in this appnote, *AN-1079*, is a simple demonstration of **μC/OS-II** and **μC/Probe** for the NXP LPC2378 processor on the IAR LPC-P2378-SK evaluation board. The basic procedure for setting up and using each of these can be gleaned from an inspection of the application code contained in *app.c*, which should serve as a beginning template for further use of these software modules. Being but a basic demonstration of software and hardware functionality, this code will make evident the power and convenience of **μC/OS-II** “The Real-Time Kernel” used on the NXP LPC2378 processor without the clutter or confusion of a more complex example.

4.01 *app.c*

Five functions of interest are located in *app.c*:

1. **main()** is the entry point for the application, as it is with most C programs. This function initializes the operating system, creates the primary application task, **AppTaskStart()**, begins multitasking, and exits.
2. **AppTaskStart()**, after creating the user interface tasks, enters an infinite loop in which it sets the LED backlight based on the potentiometer output and buffers task CPU usage data which may be graphed on the LCD (see Figure 2-14).
3. **AppTaskKbd()** polls the user inputs—the push buttons and the joystick—and, if new input is detected, sends a message to **AppTaskUserIF()**.
4. **AppTaskUserIF()** updates the LCD.
5. **AppTaskFactorial()** is a task with highly variable stack usage. This function calculates the factorial of a small integer using a recursive routine, delaying before each recursion.
6. **AppTaskProbeStr()** outputs strings to the **μC/Probe** Windows application via RS-232, which will appear in the Serial Output window. For more information, see Section 6.

```

void main (void)                                     /* Note 1 */
{
    CPU_INT08U err;

    BSP_IntDisAll();                                 /* Note 2 */

    OSInit();                                         /* Note 3 */

    OSTaskCreateExt(AppTaskStart,                    /* Note 4 */
        (void *)0,
        (OS_STK *)&AppTaskStartStk[APP_TASK_START_STK_SIZE - 1],
        APP_TASK_START_PRIO,
        APP_TASK_START_PRIO,
        (OS_STK *)&AppTaskStartStk[0],
        APP_TASK_START_STK_SIZE,
        (void *)0,
        OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR);

#ifdef OS_TASK_NAME_SIZE > 13                         /* Note 5 */
    OSTaskNameSet(APP_TASK_START_PRIO, "Start Task", &err);
#endif

    OSStart();                                       /* Note 6 */
}

```

Listing 4-1, main ()

Listing 4-1, Note 1: As with most C applications, the code starts in `main()`.

Listing 4-1, Note 2: All interrupts are disabled to make sure the application does not get interrupted until it is fully initialized.

Listing 4-1, Note 3: `OSInit()` must be called before creating a task or any other kernel object, as must be done with all µC/OS-II applications.

Listing 4-1, Note 4: At least one task must be created (in this case, using `OSTaskCreateExt()` to obtain additional information about the task). In addition, 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 statistic task, `OS_TaskStat()` if you set `OS_TASK_STAT_EN` to 1 in `os_cfg.h`.

Listing 4-1, Note 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, the `AppTaskStart()` is given the name "Start Task". Because C-Spy can work with the Kernel Awareness Plug-In available from Micrium, task names can be displayed during debugging.

Listing 4-1, Note 6: Finally multitasking under µC/OS-II is started by calling `OSStart()`. µC/OS-II will then begin executing `AppTaskStart()` since that is the highest-priority task created (both `OS_TaskStat()` and `OS_TaskIdle()` having lower priorities).

```
static void AppTaskStart (void *p_arg)
{
    CPU_INT16U      adc;
    CPU_INT32U      idx;
    CPU_INT32U      jdx;

    (void)p_arg;

    BSP_Init();                                /* Note 1 */

    #if (OS_TASK_STAT_EN > 0)
        OSStatInit();                          /* Note 2 */
    #endif

    #if (uC_PROBE_OS_PLUGIN > 0)
        OSProbe_Init();                        /* Note 3 */
        OSProbe_SetCallback(AppProbeCallback);
        OSProbe_SetDelay(50);
    #endif

    #if (uC_PROBE_COM_MODULE > 0)
        ProbeCom_Init();                      /* Note 4 */
        ProbeRS232_Init(115200);
        ProbeRS232_RxIntEn();
    #endif

    AppUserIFMbox1 = OSMboxCreate((void *)0); /* Note 5 */
    AppUserIFMbox2 = OSMboxCreate((void *)0);

    AppTaskCreate();

    for (idx = 0; idx < CPU_USAGE_NUM_TASKS; idx++) {
        for (jdx = 0; jdx < CPU_USAGE_BUF_SIZE; jdx++) {
            AppTaskCPUUsage[idx][jdx] = 0;
        }
    }

    while (DEF_TRUE) {                         /* Note 6 */
        adc = ADC_GetStatus(1);
        LCD_LightSet((adc >> 4) + 0x40);
        OSTimeDlyHMSM(0, 0, 0, 250);

        for (idx = 0; idx < CPU_USAGE_NUM_TASKS; idx++) {
            if (AppTaskCPUUsageIdx == CPU_USAGE_BUF_SIZE - 1) {
                AppTaskCPUUsage[idx][0] = OSProbe_TaskCPUUsage[idx];
            } else {
                AppTaskCPUUsage[idx][AppTaskCPUUsageIdx + 1] = OSProbe_TaskCPUUsage[idx];
            }
        }

        if (AppTaskCPUUsageIdx == CPU_USAGE_BUF_SIZE - 1) {
            AppTaskCPUUsageIdx = 0;
        } else {
            AppTaskCPUUsageIdx++;
        }
    }
}
```

Listing 4-2, AppTaskStart()

Listing 4-2, Note 1: `BSP_Init()` initializes the Board Support Package—the I/Os, tick interrupt, etc. See Section 5 for details.

Listing 4-2, Note 2: `OSStatInit()` initializes μ C/OS-II's statistic task. This only occurs if you enable the statistic task by setting `OS_TASK_STAT_EN` to 1 in `os_cfg.h`. The statistic task measures

overall CPU usage (expressed as a percentage) and performs stack checking for all the tasks that have been created with `OSTaskCreateExt()` with the stack checking option set.

Listing 4-2, Note 3: `OSProbe_Init()` initializes the **μC/Probe** plug-in for **μC/OS-II**, which maintains CPU usage statistics for each task.

Listing 4-2, Note 4: `ProbeCom_Init()` initializes the **μC/Probe** generic communication module; `ProbeRS232_Init()` initializes the RS-232 communication module. After these have been initialized, the **μC/Probe** Windows program will be able to download data from the processor. For more information, see Section 6.

Listing 4-2, Note 5: These two `OS_EVENTS` provide for communication between `AppTaskUserIF()` and `AppTaskKbd()`. If one of the push buttons is pressed or if the joystick is toggled left or right, then the new state of the display is passed by `AppTaskKbd()` to `AppTaskUserIF()` in `AppUserIFMbox1`. If the joystick is toggled up or down, then the new scroll of the display is passed by `AppTaskKbd()` to `AppTaskUserIF()` in `AppUserIFMbox2`.

Listing 4-2, Note 6: Any task managed by **μC/OS-II** must either enter an infinite loop 'waiting' for some event to occur or terminate itself. This task enters an infinite loop in which it sets the LED backlight based on the potentiometer output and buffers task CPU usage data which may be graphed on the LCD (see Figure 2-14).

4.02 *os_cfg.h*

The file *os_cfg.h* 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 Jean Labrosse's book, *μC/OS-II, The Real-Time Kernel, 2nd Edition*. *os_cfg.h* assumes you have **μC/OS-II** V2.83 or higher but also works with previous versions of **μC/OS-II**.

- `OS_APP_HOOKS_EN` is set to 1 so that the cycle counters in the `OS_TCBs` will be maintained.
- Task sizes for the Idle (`OS_TASK_IDLE_STK_SIZE`), statistics `OS_TASK_STAT_STK_SIZE`) and timer (`OS_TASK_TMR_STK_SIZE`) task are set to 128 `OS_STK` elements (each is 4 bytes) and thus each task stack is 512 bytes. If you add code to the examples make sure you account for additional stack usage.
- `OS_DEBUG_EN` is set to 1 to provide valuable information about **μC/OS-II** objects to IAR's C-Spy through the Kernel Awareness plug-in. Setting `OS_DEBUG_EN` to 0 should save some code space (though it will not save much).
- `OS_LOWEST_PRIO` is set to 31, allowing up to 32 total tasks.
- `OS_MAX_TASKS` determines the number of "application" tasks and is currently set to 16 allowing 8 more tasks to be added to the example code.
- `OS_TICKS_PER_SEC` is set to 1000 Hz. This value can be changed as needed and the proper tick rate will be adjusted in *bsp.c* if you change this value. You would typically set the tick rate

between 10 and 1000 Hz. The higher the tick rate, the more overhead **μC/OS-II** will impose on the application. However, you will have better tick granularity with a higher tick rate.

5. 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 LPC2378-SK evaluation board. 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*).

5.01 IAR EWARM v4.4x-Specific BSP Files

The BSP includes two files intended specifically for use with IAR EWARM v4.4x: *LPC2378_Flash.xcl* and *cstartup.s79*. These serve to define the memory map and initialize the processor prior to loading or executing code. If the example application is to be used with other toolchains, 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. IAR requires a linker command file, such as *LPC378_Flash*, that provides directives to accomplish this. With this file, the data and execution stacks are mapped to RAM while code is mapped to flash.

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.

5.02 IAR EWARM v5.1x-Specific BSP Files

The BSP includes two files intended specifically for use with IAR EWARM v5.1x: *LPC2378_Flash.icf* and *cstartup.s*. These files serve the same purpose as their IAR EWARM v4.4x counterparts. The linker specification file (extension **.icf* for EWARM v5.1x) uses a completely different format than its predecessor (extension **.xcl* for EWARM v4.4x), but the information is essentially identical. Except for some minor changes to the EWARM v5.1x assembler, *cstartup.s* is basically identical to *cstartup.s79*.

5.03 RVMDK-Specific BSP Files

The BSP includes three files intended specifically for use with Keil μVision3 (RV-MDK) tools: *LPC2378_Flash.scats*, *init.s* and *vectors.s*. The first file serves to define the memory map of the processor, equivalent to *LPC2378_Flash.xcl* for the IAR v4.4x project.

In *init.s* is code which will be executed prior to calling `main`. This does nothing more than setup the various exception stacks. After executing, this function branches to the `__main` function, in which the processor is further readied for entering application code.

The ARM exception vectors are defined in *vector.s*.

5.04 BSP, *bsp.c* and *bsp.h*

The file *bsp.c* implements several global functions, each providing some important service, be that the initialization of processor functions for μ C/OS-II to operate or the toggling of an LED. Several local functions are defined as well to perform some atomic duty, initializing the I/O for the LED or initialize the μ C/OS-II tick timer. The discussion of the BSP will be limited to the discussion of the global functions that might be called from user code (and may be called from the example application).

The global functions defined in *bsp.c* (and prototyped in *bsp.h*) may be roughly divided into two categories: critical processor initialization and user interface services. Four functions constitute the former:

- **BSP_Init()** is called by the application code to initialize critical processor features (particularly the μ C/OS-II tick interrupt) after multitasking has started (i.e., *OS_Start()* has been called). This function should be called before any other BSP functions are used. See Listing 5-1 for more details.
- **BSP_IntDisAll()** is called to disable all interrupts, thereby preventing any interrupts until the processor is ready to handle them.
- **BSP_CPU_ClkFreq()** returns the clock frequency in Hz.
- **BSP_CPU_PclkFreq()** returns the clock frequency in Hz or a peripheral clock; an ID for the peripheral clock (as defined in *bsp.h*) is accepted as the argument.

Four function provide access to user interface components:

- **LCD_LightSet()** accepts an 8-bit integer between 0x00 and 0x80, using this value to set the LCD backlight brightness. The PWM which drives the LCD brightness is configured with a period of 0x80, so value of 0x00 is off and a value of 0x80 is the brightest possible setting.
- **PB_GetStatus()** takes as its argument the ID of a push button and returns *DEF_TRUE* if the push button is being pressed and *DEF_FALSE* if the push button is not being pressed. The valid IDs are 1 and 2.
- **Joystick_GetStatus()** returns the direction(s) that the joystick is currently toggled. One or more of five bits may be set; the constants *JOYSTICK_CENTER*, *JOYSTICK_LEFT*, *JOYSTICK_RIGHT*, *JOYSTICK_UP*, and *JOYSTICK_DOWN* defined in *bsp.h* correspond to the five different possibilities.
- **ADC_GetStatus()** takes as its argument the ID of a ADC and returns the 10-bit number that, when divided by 0x3FF, is equal to the ratio of the input voltage divided by the reference voltage. The only valid ID is 1.

5.05 Processor Initialization Function

```
void BSP_Init (void)
{
    PLL_Init();                /* Note 1 */
    MAM_Init();                /* Note 2 */

    GPIO_Init();              /* Note 3 */
    ADC_Init();               /* Note 4 */
    LCD_LightInit();          /* Note 5 */

    VIC_Init();               /* Note 6 */
    Tmr_TickInit();           /* Note 7 */
}
```

Listing 5-1, BSP_Init()

Listing 5-1, Note 1: The PLL is setup to generate a 72 MHz CPU clock. All peripheral clocks are set to half the CPU clock.

Listing 5-1, Note 2: The Memory Acceleration Module (MAM) is setup.

Listing 5-1, Note 3: The general purpose I/O ports are setup for the UART, joystick and PBs.

Listing 5-1, Note 4: The ADC used for the potentiometer is initialized..

Listing 5-1, Note 5: The PWM used for the LCD backlight is initialized..

Listing 5-1, Note 6: The interrupt controller is initialized, including the disabling of all interrupts and the assignment of a dummy ISR handler to each interrupt vector to catch spurious interrupts.

Listing 5-1, Note 7: The μC/OS-II tick interrupt source is initialized.

Listings 5-2 and 5-3 give the μC/OS-II timer tick initialization function, `Tmr_TickInit()`, the tick ISR handler, `Tmr_TickISR_Handler()`. These may serve as examples for initializing an interrupt and servicing that interrupt.

```
static void Tmr_TickInit (void)
{
    CPU_INT32U  pclk_freq;
    CPU_INT32U  rld_cnts;

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

    pclk_freq    = BSP_CPU_PclkFreq();             /* Note 2 */

    rld_cnts     = pclk_freq / OS_TICKS_PER_SEC;

    TOTCR        = (1 << 1);
    TOTCR        = 0;
    TOPC         = 0;

    TOMR0        = rld_cnts;                       /* Note 3 */
    TOMCR        = 3;

    TOCCR        = 0;
    TOEMR        = 0;
    TOTCR        = 1;                             /* Note 4 */
}
```

Listing 5-2, Tmr_TickInit()

Listing 5-2, Note 1: The timer interrupt vector is set and the interrupt is enabled.

Listing 5-2, Note 2: The peripheral clock frequency is calculated, and this clock frequency and desired tick rate—OS_TICKS_PER_SEC—are used to determine the number of clocks between interrupts.

Listing 5-2, Note 3: The timer is setup to generate a periodic interrupt and then reset to zero.

Listing 5-2, Note 4: The timer is started.

```
void Tmr_TickISR_Handler (void)
{
    TOIR = 0xFF;                                   /* Note 1 */

    OSTimeTick();                                  /* Note 2 */
}
```

Listing 5-3, Tmr_TickISR_Handler()

Listing 5-3, Note 1: The interrupt is cleared.

Listing 5-3, Note 2: OSTimeTick() informs µC/OS-II of the tick interrupt.

5.06 LCD Driver, *glcd.c* and *glcd.h*

IAR includes a sample driver for the LCD controller, which is used (with some additions) for this example application. The original driver had six global functions prototyped in *glcd.h*:

1. **GLCD_SendCmd()** sends a command to the display. The commands are included in the `enum GLCD_Cmd_t`, also in *glcd.h*. The example application only uses this function to send the display scroll command; otherwise this is only called indirectly (i.e., it is called by functions called in the application).
2. **GLCD_PowerUpInit()** initializes the display. The argument of this function is a pointer to initial screen data or is `NULL`.
3. **GLCD_SetFont()** sets the font that will be used for displaying text. The first argument is one of `Terminal_18_24_12`, `Terminal_6_8_6`, or `Terminal_9_12_6` (unless you define your own font). The second and third arguments are the font color and the background color, respectively.
4. **GLCD_SetWindow()** sets the current window area. This is the area, for instance, in which text will be displayed.
5. **GLCD_TextSetPos()** sets the position of text inside the current window area. The text position will change as characters are written to the display so that additional characters will appear in the appropriate location.
6. **GLCD_TextSetTabSize()** sets the tab size, in characters.

Two functions were added to provide additional capabilities:

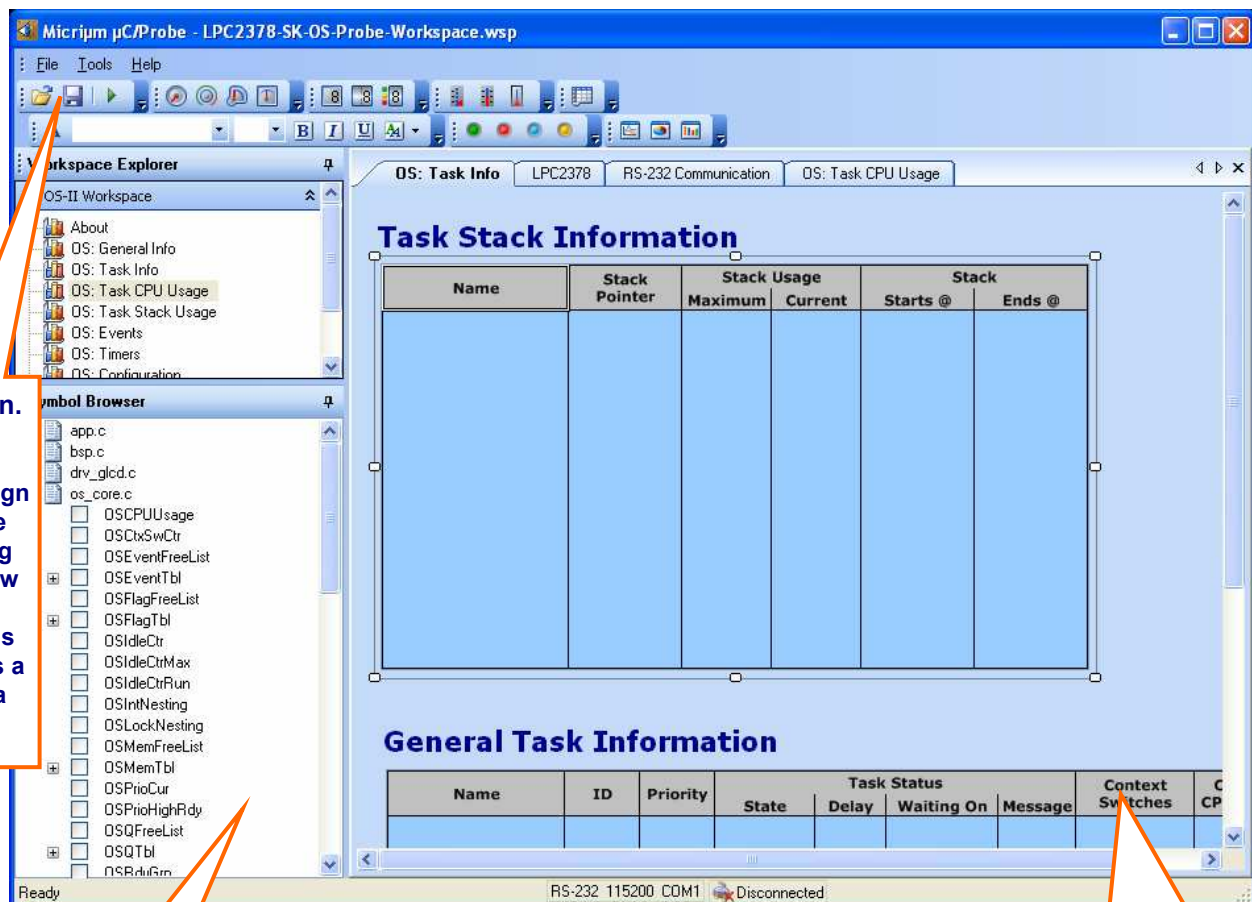
1. **GLCD_DrawRectangle()** draws a rectangle with the specified upper-left vertex, height, width, and color.
2. **GLCD_DrawLine()** draws a line between two specified points in the specified color.

The driver also provides an implementation of the `putchar()` function which writes a character to the LCD. Because this function is used by the default `printf()`, formatted output can be made by simply calling `printf()`.

6. μC/Probe

μC/Probe is a Windows program which retrieves the values of global variables from a connected embedded target and displays the values in an engineer-friendly format. To accomplish this, an ELF file, created by the user's compiler and containing the names and addresses of all the global symbols on the target, is monitored by **μC/Probe**. The user places components (such as gauges, labels, and charts) into a Data Screen in a **μC/Probe** workspace and assigns each one of these a variable from the Symbol Browser, which lists all symbols from the ELF file. The symbols associated with components placed on an open Data Screen will be updated after the user presses the start button (assuming the user's PC is connected to the target).

μC/Probe currently interfaces with a target processor with a RS-232. A small section of code resident on the target receives commands from the Windows application and responds to those commands. The commands ask for a certain number of bytes located at a certain address, for example, "Send 16 bytes beginning at 0x0040102C". The Windows application, upon receiving the response, updates the appropriate component(s) on the screens with the new values.



Start Button.
This button switches between Design and Run-Time Views. During Run-Time View (when data is collected), this will appear as a stop button (a blue square).

Symbol Browser.
Contains all symbols from the ELF files added to the workspace.

Figure 6-1. μC/Probe Windows Program

Data Screen.
Components are placed onto the data screen and assigned symbols during Design View. During Run-Time View, these components are updated with values of those symbols from the target

To use **µC/Probe** with the example project (or your application), do the following:

1. **Download and Install µC/Probe.** A trial version of **µC/Probe** can be downloaded from the Micrium website at

<http://www.micrium.com/products/probe/probe.html>

2. **Open µC/Probe.** After downloading and installing this program, open the example **µC/Probe** workspace for **µC/OS-II**, named *OS-Probe-Workspace.wsp*, which should be located in your installation directory at

/Program Files/Micrium/uC-Probe/Target/Plugins/uCOS-II/Workspace

3. **Connect Target to PC.** Currently, **µC/Probe** can use RS-232 to retrieve information from the target. You should connect a RS-232 cable between your target and computer.
4. **Load Your ELF File.** The example projects included with this application note are already configured to output an ELF file. (If you are using your own project, please refer to Appendix A of the **µC/Probe** user manual for directions for generating an ELF file with your compiler.) This file should be in

<Project Directory>/<Configuration Name>/exe/


where *<Project Directory>* is the directory in which the IAR EWARM project is located (extension *.ewp) and *<Configuration Name>* is the name of the configuration in that project which was built to generate the ELF file and which will be loaded onto the target. The ELF file will be named

<Project Name>.elf

in EWARM v4.4x and

<Project Name>.out

in EWARM v5.1x unless you specify otherwise. To load this ELF file, right-click on the symbol browser and choose “Add Symbols”.

5. **Configure the RS-232 Options.** In **µC/Probe**, choose the “Options” menu item on the “Tools” menu. A dialog box as shown in Figure 6-2 (left) should appear. Choose the “RS-232” radio button. Next, select the “RS-232” item in the options tree, and choose the appropriate COM port and baud rate. The baud rate for the projects accompanying this appnote is 115200.
6. **Start Running.** You should now be ready to run **µC/Probe**. Just press the run button () to see the variables in the open data screens update. Figure 6-3 displays two screens in the **µC/OS-II** workspace which display detailed information about each task’s state.

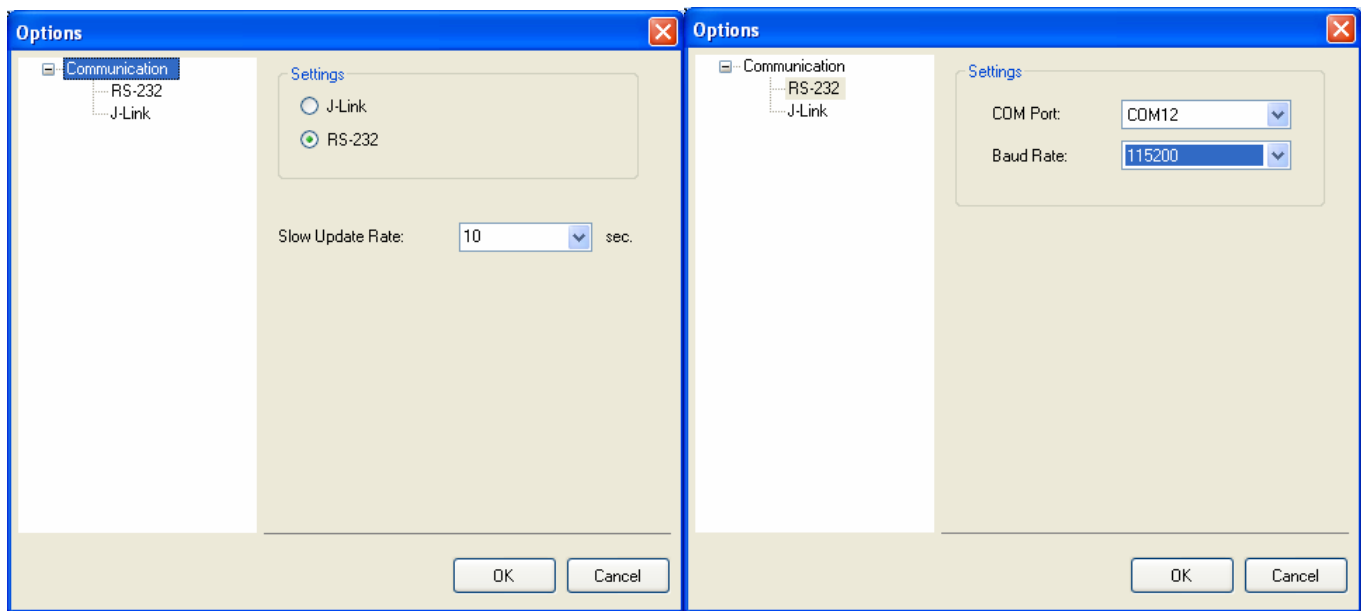


Figure 6.2. μ C/Probe Options

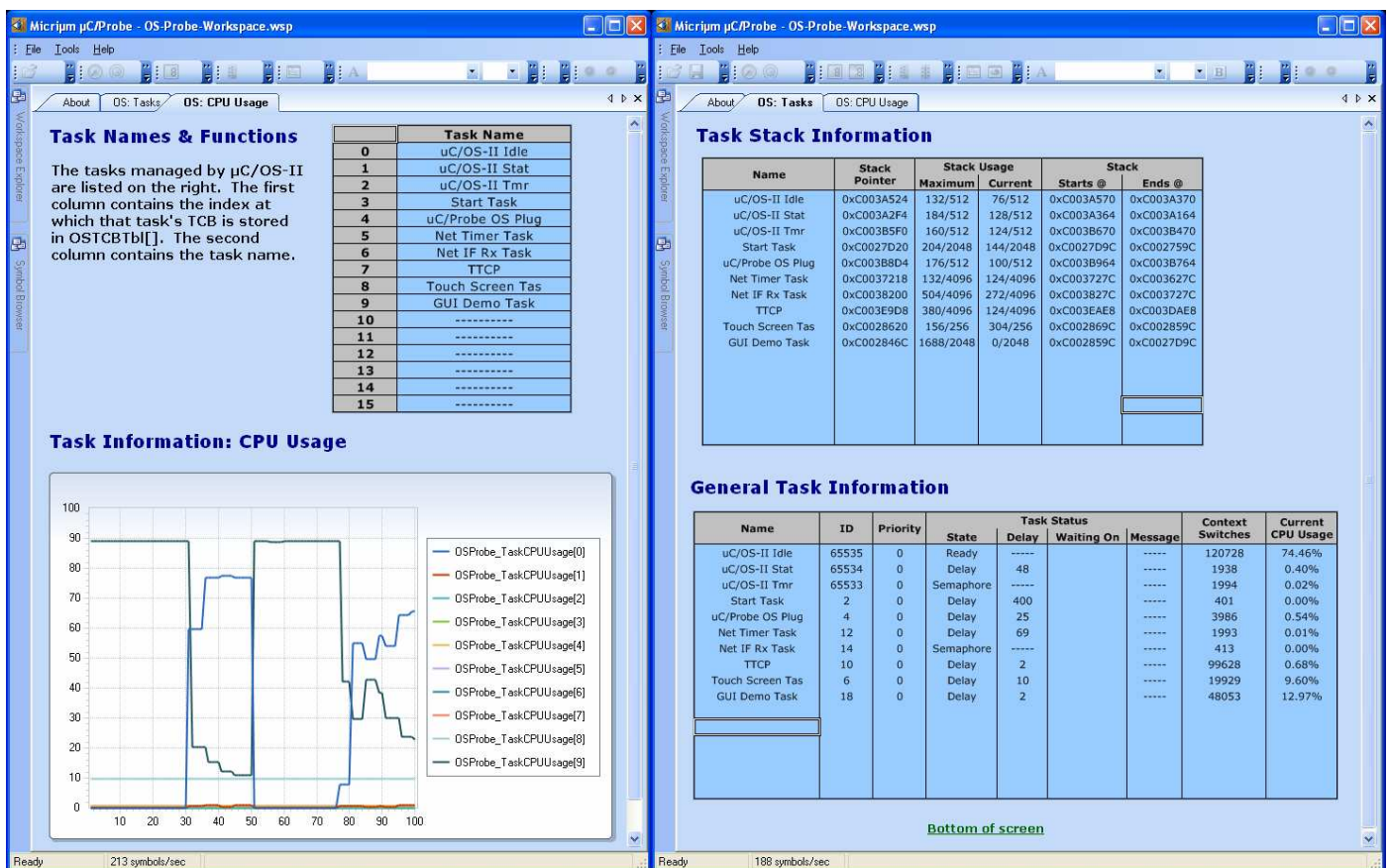


Figure 6-3. μ C/Probe Run-Time: μ C/OS-II Task Information

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 Micrium 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. Hillside Blvd
Foster City, CA 94404
USA

+1 650 287 4250
+1 650 287 4253 (FAX)

e-mail: Info@IAR.com
WEB : <http://www.IAR.com>

Micrium

949 Crestview Circle
Weston, FL 33327
USA

+1 954 217 2036
+1 954 217 2037 (FAX)

e-mail: Jean.Labrosse@Micrium.com
WEB : <http://www.Micrium.com>

NXP

1110 Ringwood Court
San Jose, CA 95131

+1 408 474 8142

WEB: www.nxp.com