Micriμm

μC/OS-II

and the

AVR32 UC3 Architecture

(with the Atmel AVR32 EVK1100 Evaluation Board)

Application Note

AN-1300

www.Micrium.com

About Micriμm

Micriμm 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 Micriμm **μC/OS-II**, features the highest-quality source code available for today's embedded market. Micriμm 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. Micriμm's products consistently shorten time-to-market throughout all product development cycles. For additional information on Micriμm, please visit www.micrium.com.

About μC/OS-II

Thank you for your interest in **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 and now, has been ported to the AVR32 UC3 CPU.

μC/OS-II is small yet provides all the services you would expect from an RTOS: task management, time and timer management, semaphore and mutex, message mailboxes and queues, event flags an 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 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.

Manual Version

If any error is found in this document, please inform Micriμm in order for the appropriate corrections to be present in future releases.

Version	Date	By	Description
V 1.00	2007/05/08	FGK	Initial version.
V 1.01	2008/07/14	FGK	Update uC/OS-II version AVR32Studio and IAR toolchain update Update BSP

Software Versions

This document may or may not have been downloaded as part of an executable file containing the code described herein or any additional application or board support code. If so, then the versions of the Micriμm 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	V2.86	

Table of Contents

1.00 Introduction

This document, AN-1300, explains an example code for using **µC/OS-II** with the Atmel AVR32 EVK1100 Evaluation Board, which uses a Atmel AVR32 UC3-based microcontroller. The processor on the board has an internal 512-kB of flash and 64-kB of SRAM. Peripherals for several communications busses are available, including UART, SPI, I2C, USB device and USB OTG, Ethernet, among others.

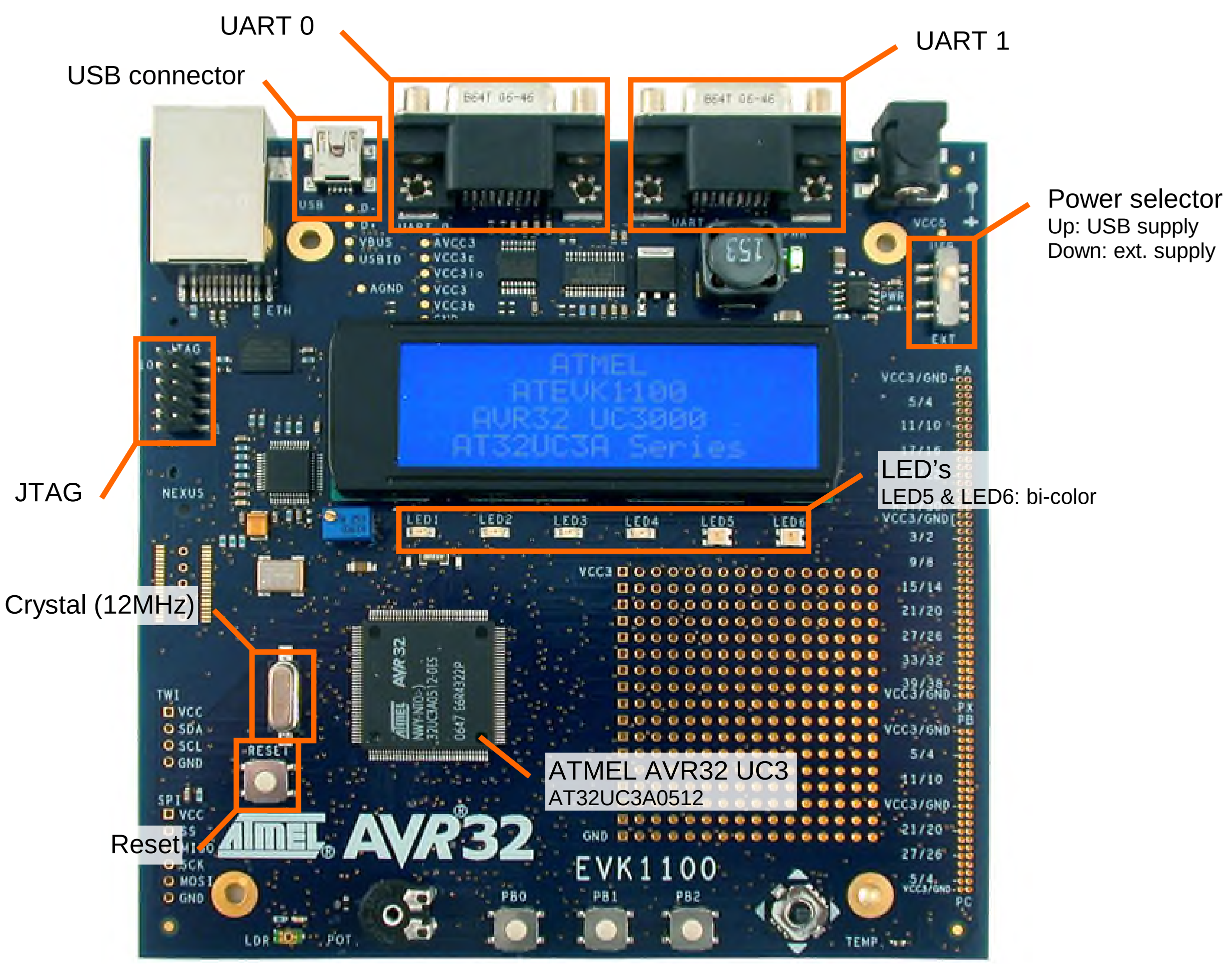


Figure 1-1, AVR32 UC3 EVK1100 Evaluation Board

The evaluation board, shown in figure 1-1, provides three user push buttons, a two-axis joystick, a potentiometer, six user LEDs including 2 bi-color, a thermistor, a photoresistor, a 64Mbit external dataflash, a 4x20 characters LCD, and a SD/MMC slot. Breakout headers are located on the right side of the board to provide access to the processor's GPIO pins for prototyping.

Figure 1-2 shows a block diagram with the relationship between your application, **µC/OS-II**, the port code and the BSP (Board Support Package). Relevant sections of this application note are referenced on the figure. The sections involving the specifics of the port for **µC/OS-II** on the AVR32 UC3 architecture are described on the Application Note AN-1030*, **µC/OS-II** and the AVR32 UC3 Processors. The AVR32 UC3 has been ported on both the IAR and AVR32Studio tools.

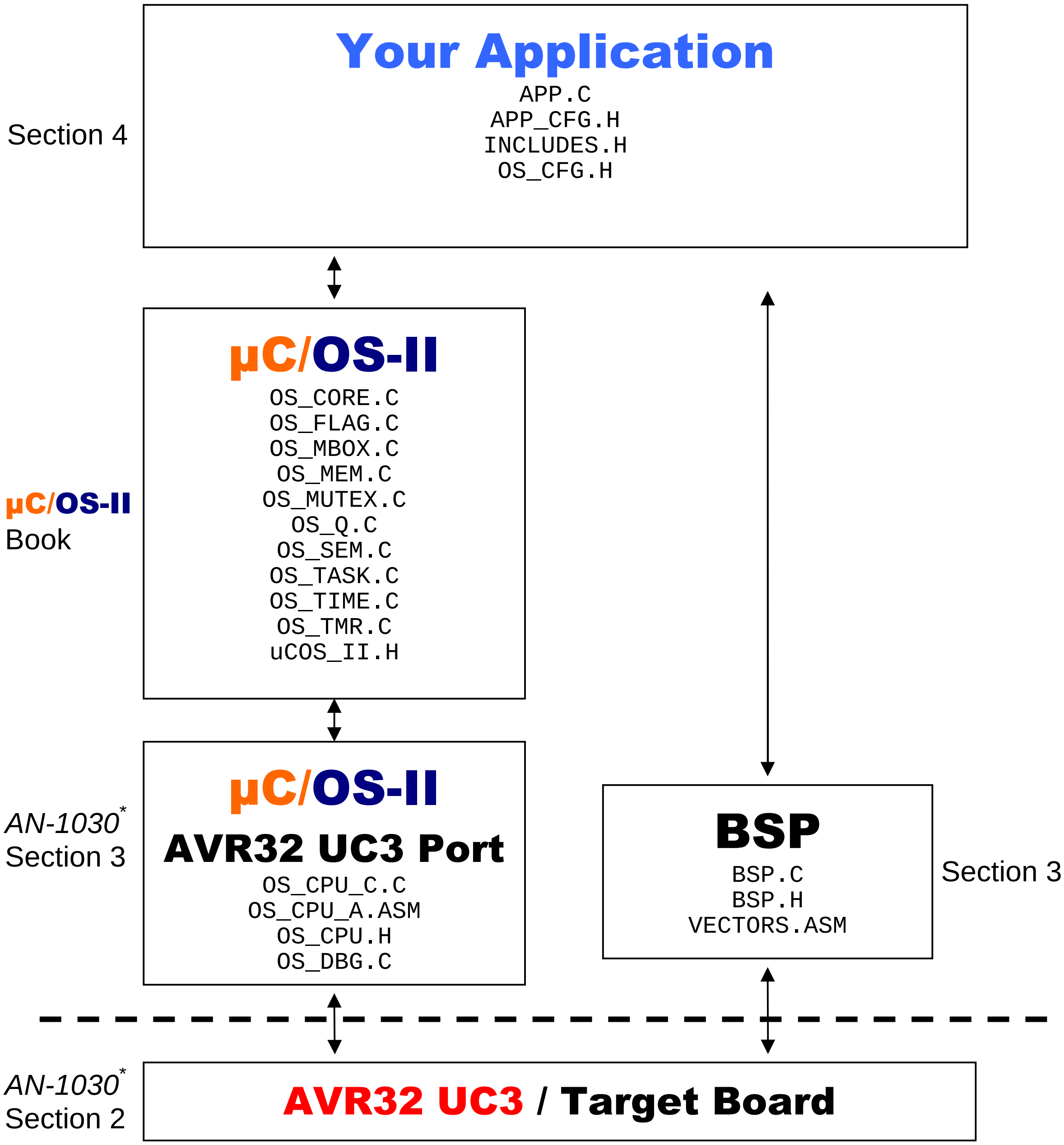


Figure 1-2, Relationship between modules.

* Please refer to the Application Note AN-1030 located in the AppNotes directory.

2.00 Opening and Loading the Project

The files located in the executable zip file named *Micrium-Atmel-uCOS-II-EVK1100.exe* are organized in the directory structure shown in Figure 2-1. The code files referred to herein follows the same directory structure.

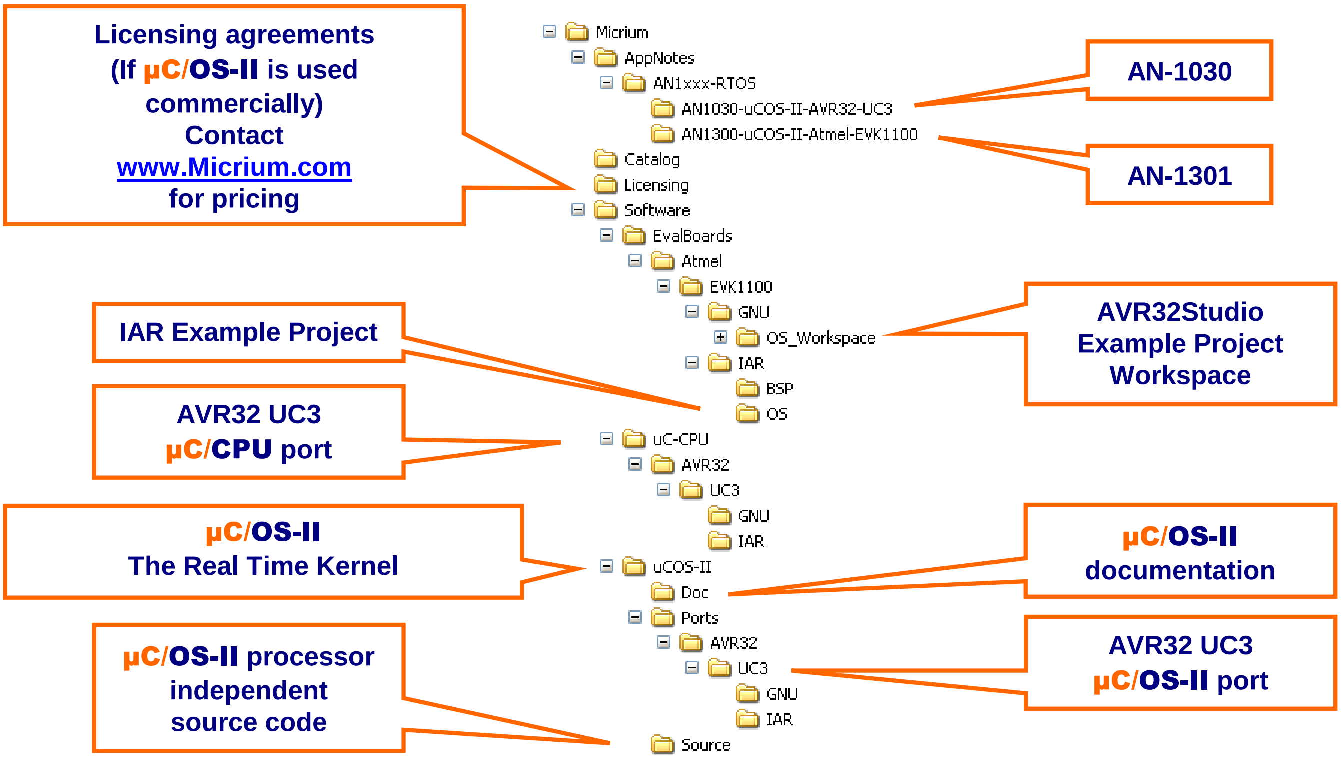


Figure 2-1, Directory Structure

Projects are included for both IAR EW and Atmel AVR32Studio. An IAR project file named OS.ewp is located in the IAR project's folder in the directory:

```
\Micrium\Software\EvalBoards\Atmel\EVK1100\IAR\OS
```

To view this example project, start an instance of IAR EW, and open the workspace file `OS.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. In addition, the workspace could also be opened by double-clicking on the file in a Windows Explorer window.

An AVR32Studio project workspace is located in the directory:

```
\Micrium\Software\EvalBoards\Atmel\EVK1100\GNU\OS_Workspace
```

To view this example project, start an instance of AVR32Studio. If the "Workspace Launcher" dialog appears, navigate to the project directory and select the workspace. Otherwise, choose the "Switch Workspace…" menu command under the "File" menu and select the workspace file after navigating to the project directory.

2.01 Programmer/Debugger

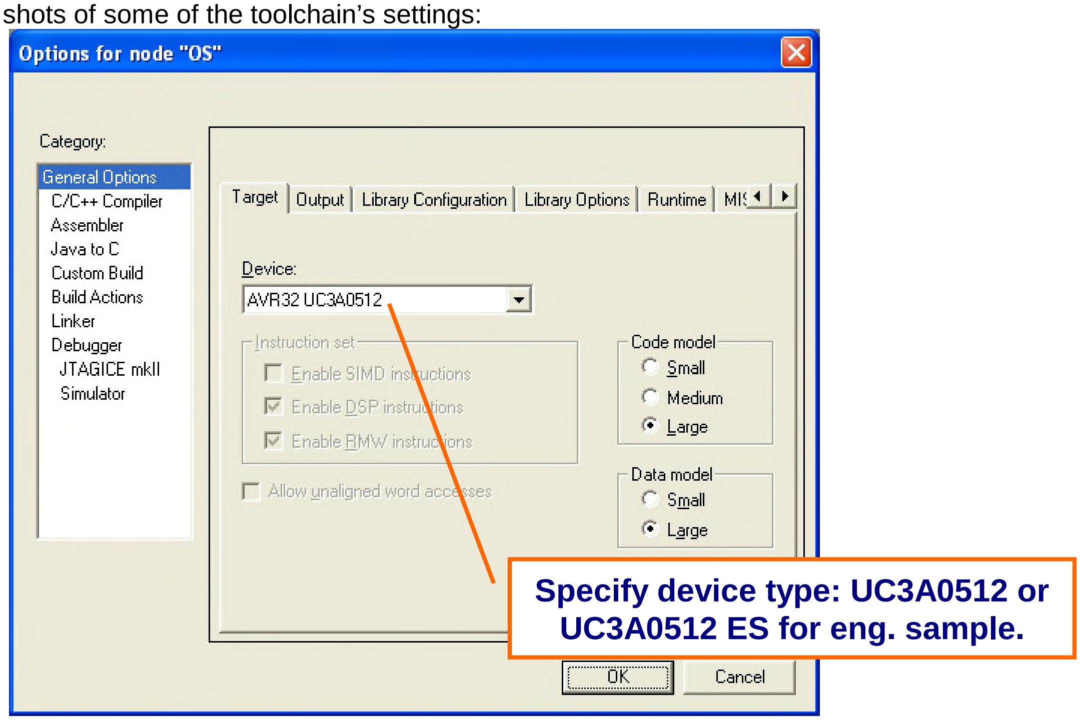
The AVR32 EVK1100 evaluation board may be programmed and debugged through the 10-pin debug port named JTAG on the board. This is done through the Atmel AVR JTAGICE mkII on-chip debugger, shown in Figure 2-2.

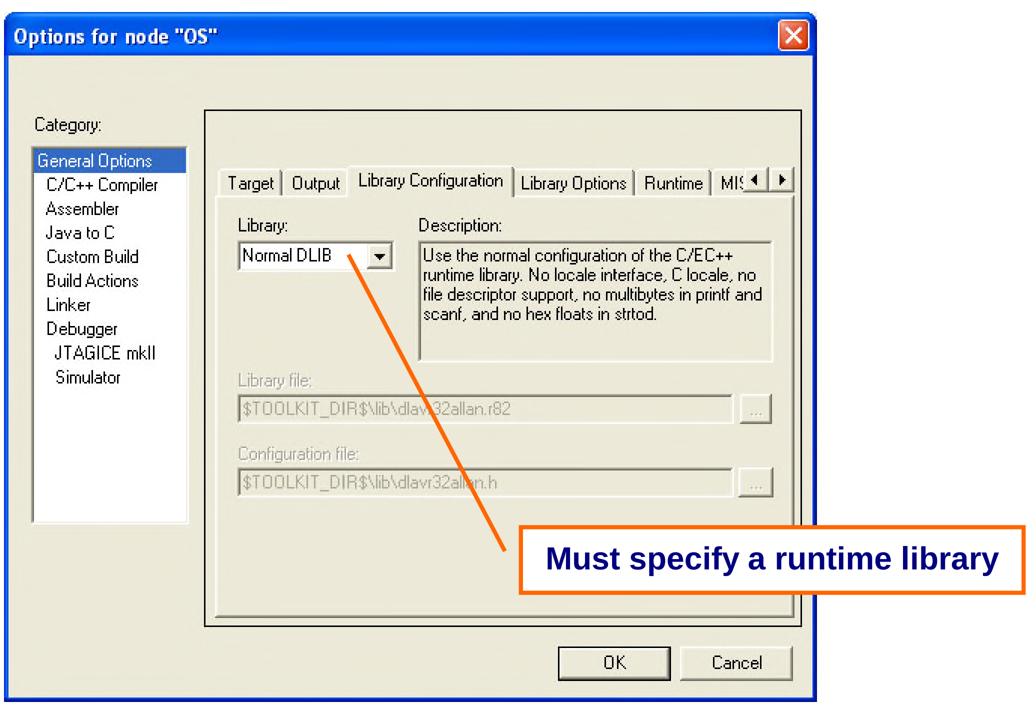
Figure 2-2, AVR JTAGICE mkII on-chip debugging tool

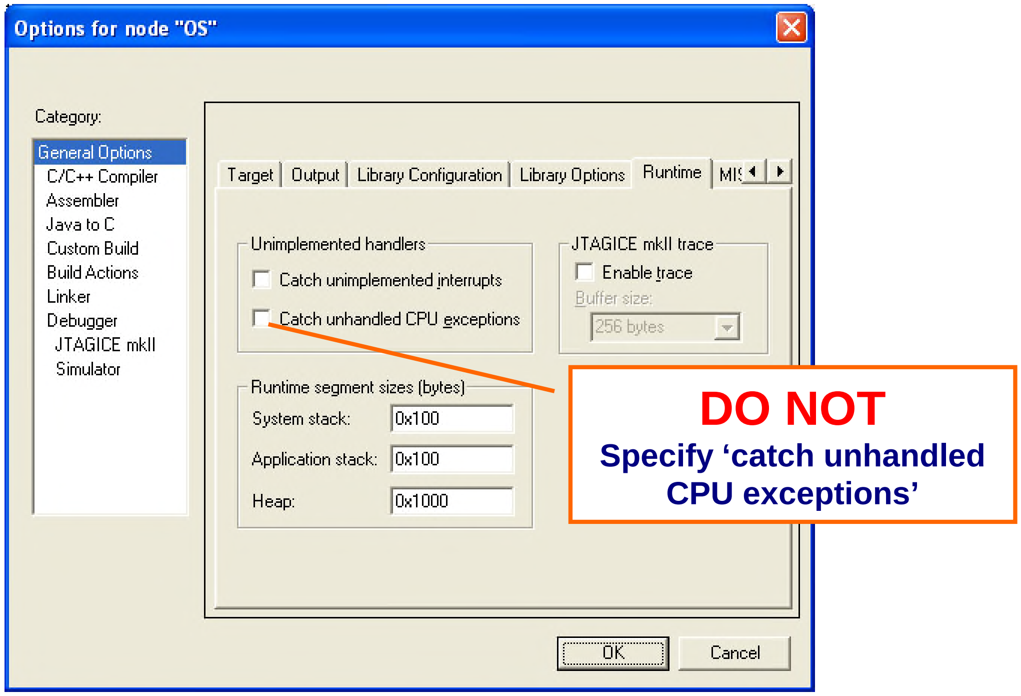
2.02 The IAR Embedded Workbench Toolchain

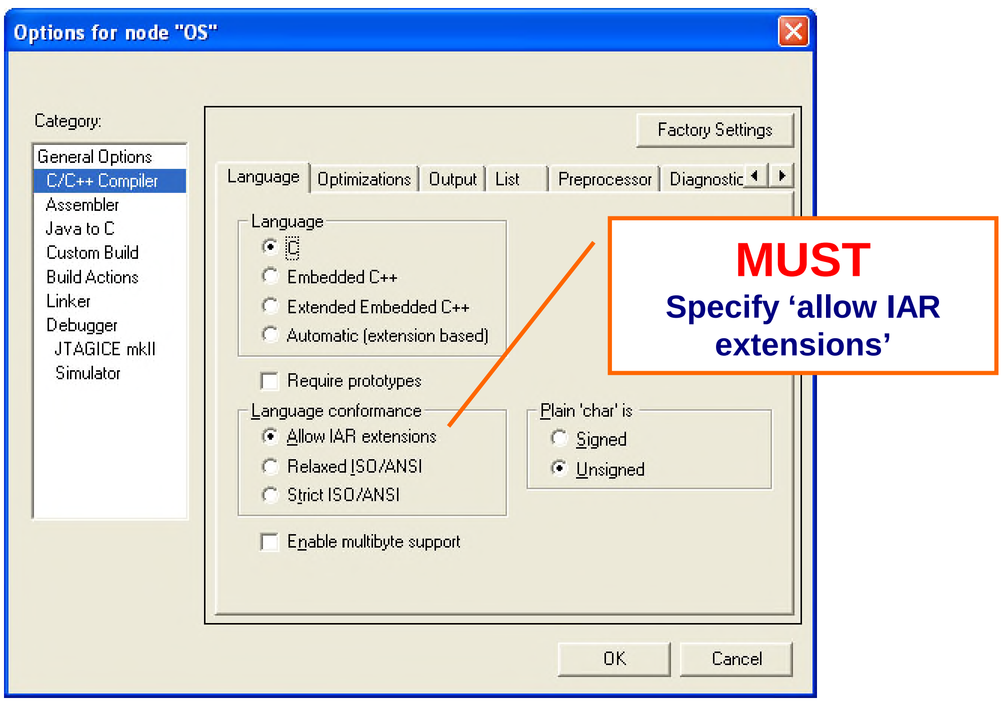
This **μC/OS-II** port was tested using the IAR Embedded Workbench for AVR32 V2.22A toolchain. The IAR Embedded Workbench IDE holds source files and libraries, manages dependencies and stores compiler, linker and other settings. The toolchain also contains the C-SPY Source-Level Debugger; a high performance graphical source-level debugger equipped with the latest features to shorten hardware bring-up and application development time.

Below are screen shots of some of the toolchain's settings:



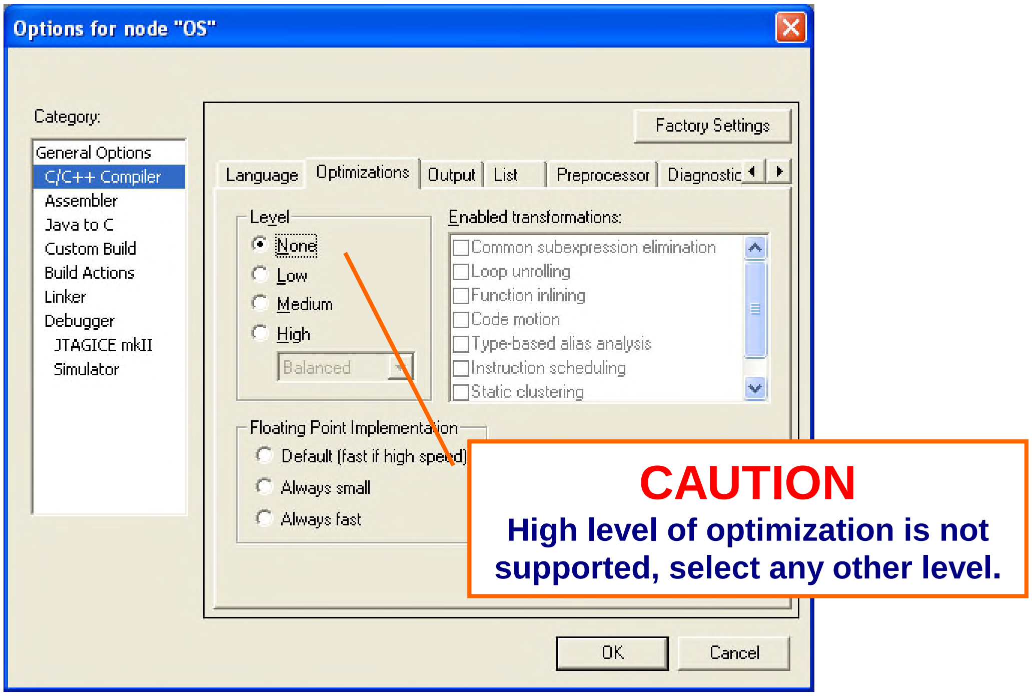


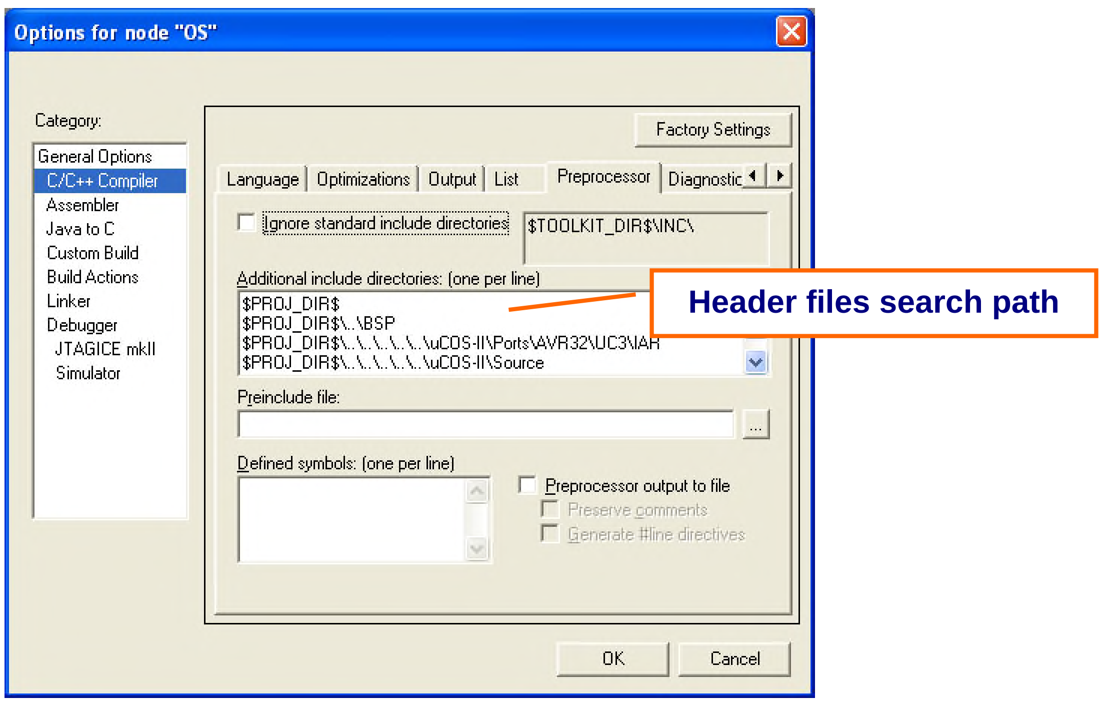




IMPORTANT

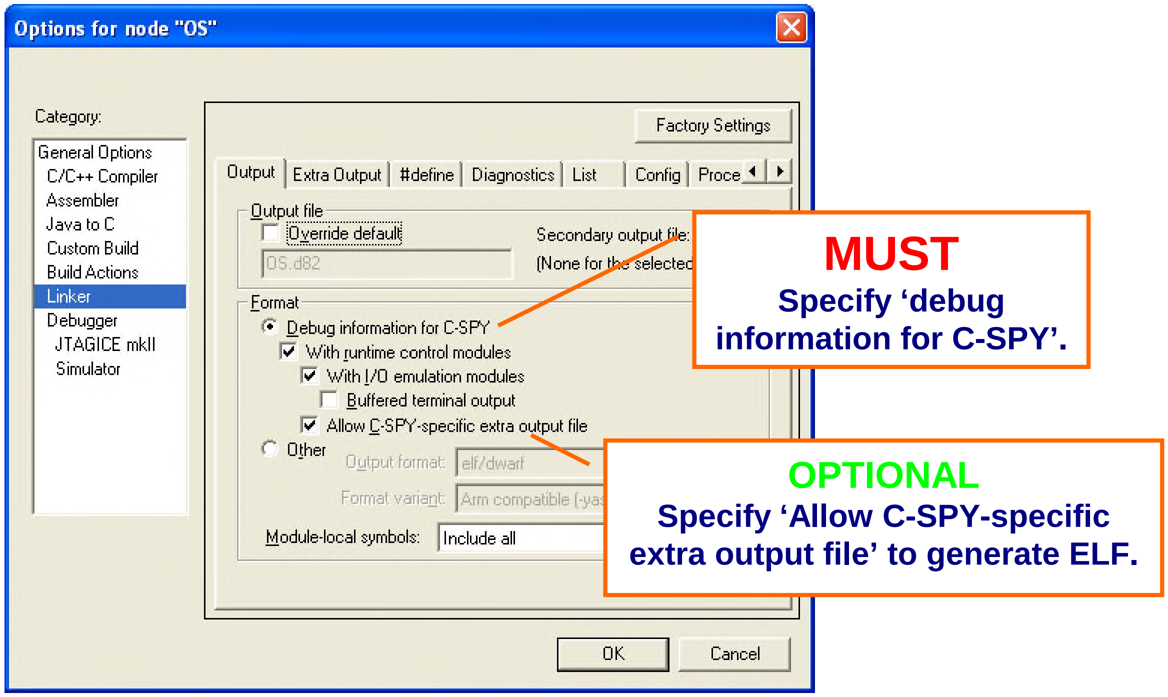
IAR extensions are required for the extended keyword used for context switching. `OS_CPU.H` defines an extern system call prototype for `OSCtxSw()` in the assembly function. The code in this port expects such calling convention in this function. If this prototype is altered, the port will **NOT** work.





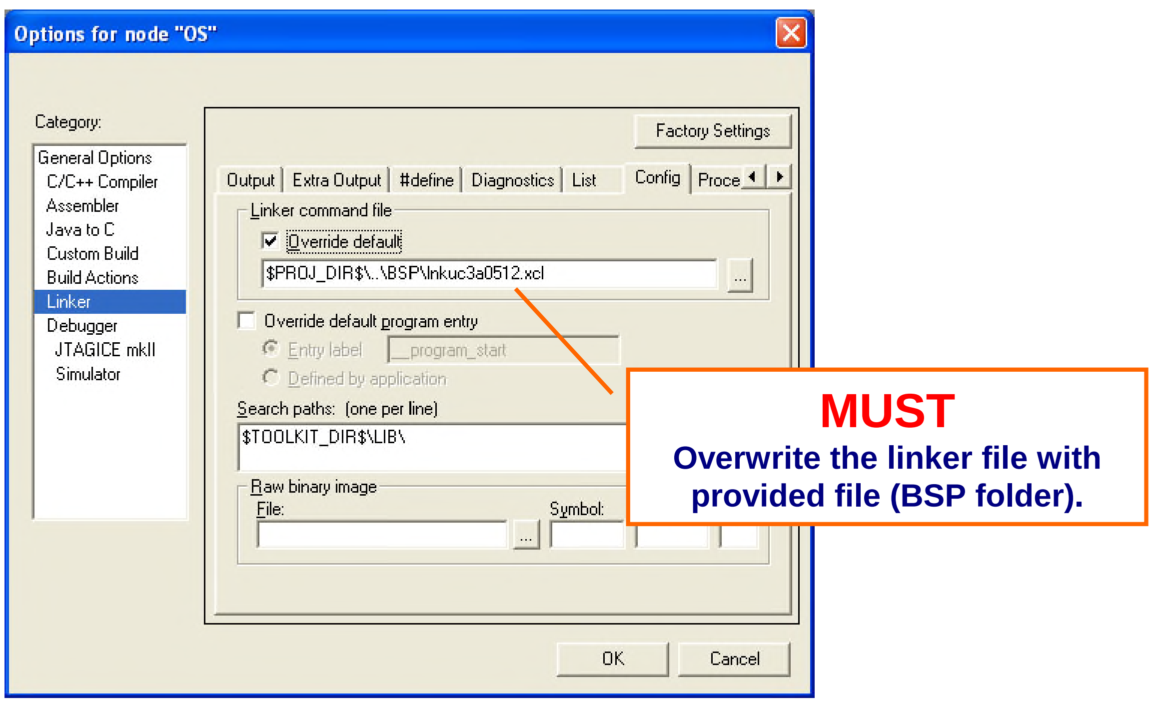
IMPORTANT

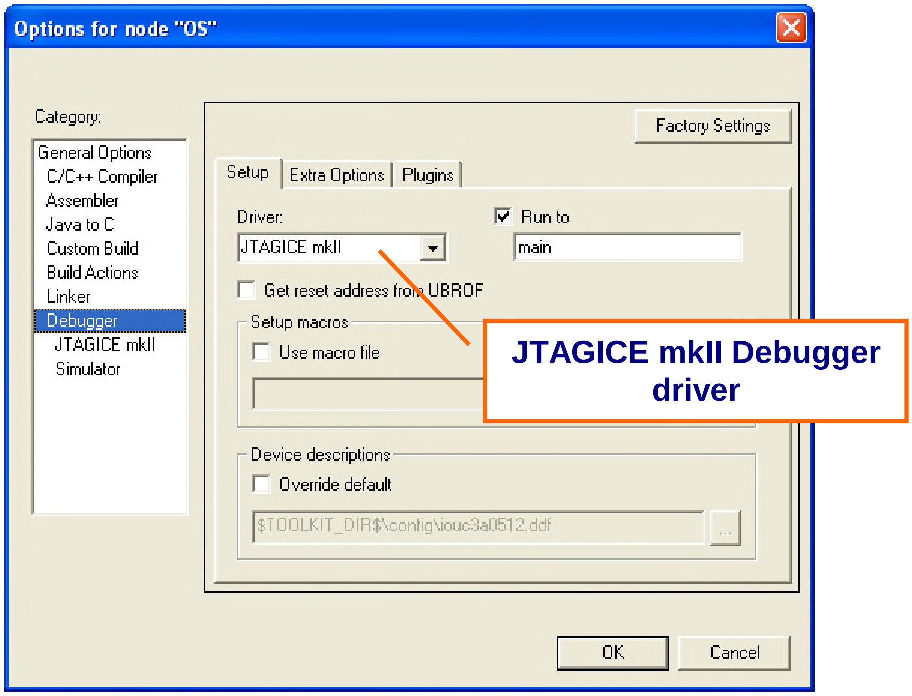
The additional include directories should contain the directory paths for all sections of the project: **μC/OS-II**, **μC/OS-II** Port, **μC/CPU**, **μC/CPU** Port, **BSP**, project root, and any other specific directory.



IMPORTANT

In order to use the JTAGICE mkII on-chip debugger, "debug information for C-SPY" must be selected in the linker/output options. The other options under the format section are not strictly necessary.




Options for node "OS"
Category:
General Options
C/C++ Compiler
Assembler
Java to C
Custom Build
Build Actions
Linker
Debugger
JTAGICE mkII
Simulator
Factory Settings
Setup
Extra Options
Plugins
Driver:
JTAGICE mkII
Run to
main
Get reset address from UBROF
Setup macros
Use macro file
Device descriptions
Override default
$TOOLKIT_DIR$\config\iouc3a0512.ddf
OK
Cancel
JTAGICE mkII Debugger driver

2.02.01 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. In order to enable the plug-in, the debugger must not be active. A **µC/OS-II** entry should be listed in the "Debugger\Plug-ins" section of the project options if the **µC/OS-II** Kernel Awareness Plug-In is installed.

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 showing the actively executing task. Each task displayed includes their pertinent information such as used stack space, task status, and task priority. An example task list is shown in Figure 2-4.

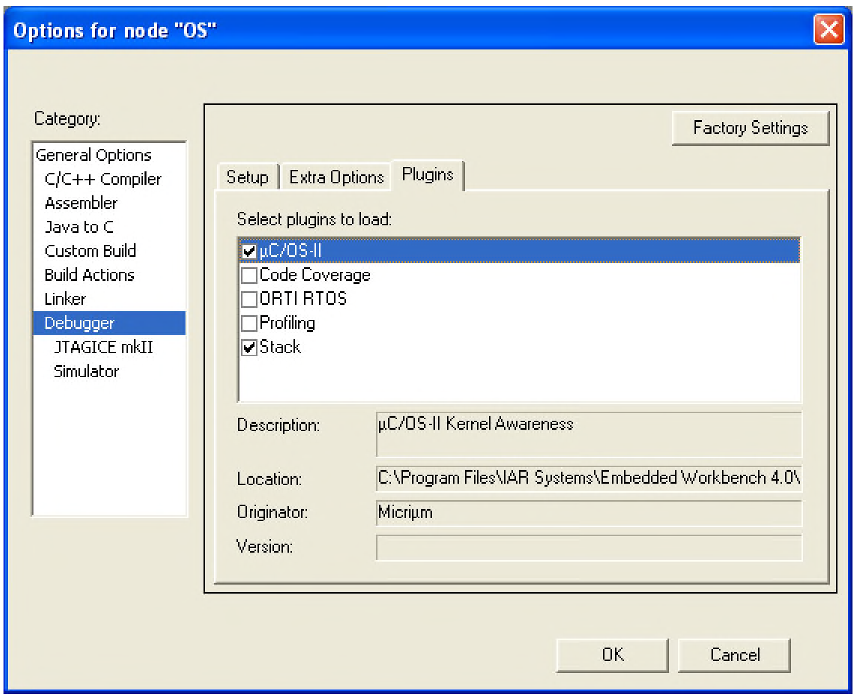
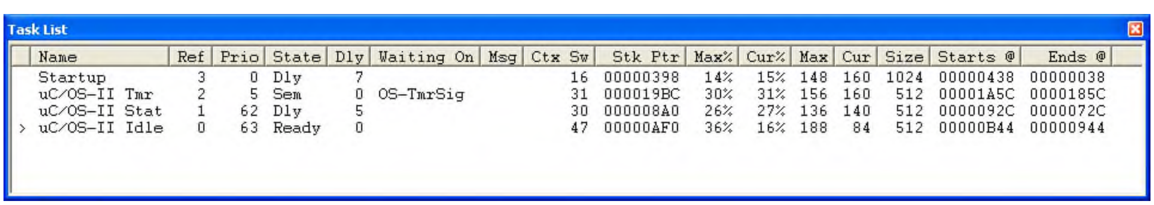


Figure 2-3. Enabling the µC/OS-II Kernel Awareness Plug-In


Task List

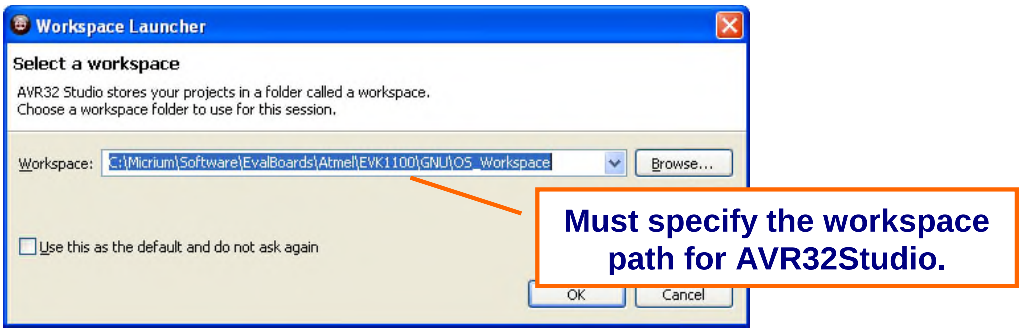
	Name	Ref	Prio	State	Dly	Waiting On	Msg	Ctx Sw	Stk Ptr	Max%	Cur%	Max	Cur	Size	Starts @	Ends @
	Startup	3	0	Dly	7			16	00000398	14%	15%	148	160	1024	00000438	00000038
	uC/OS-II Tmr	2	5	Sem	0	OS-TmrSig		31	000019BC	30%	31%	156	160	512	00001A5C	0000185C
	uC/OS-II Stat	1	62	Dly	5			30	000008A0	26%	27%	136	140	512	0000092C	0000072C
>	uC/OS-II Idle	0	63	Ready	0			47	00000AF0	36%	16%	188	84	512	00000B44	00000944

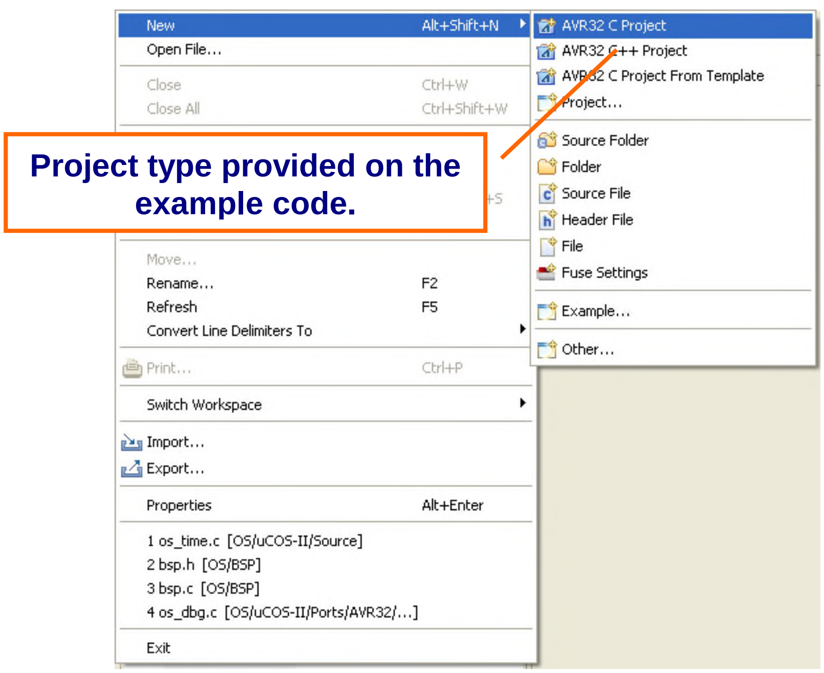
Figure 2-4. µC/OS-II Task List

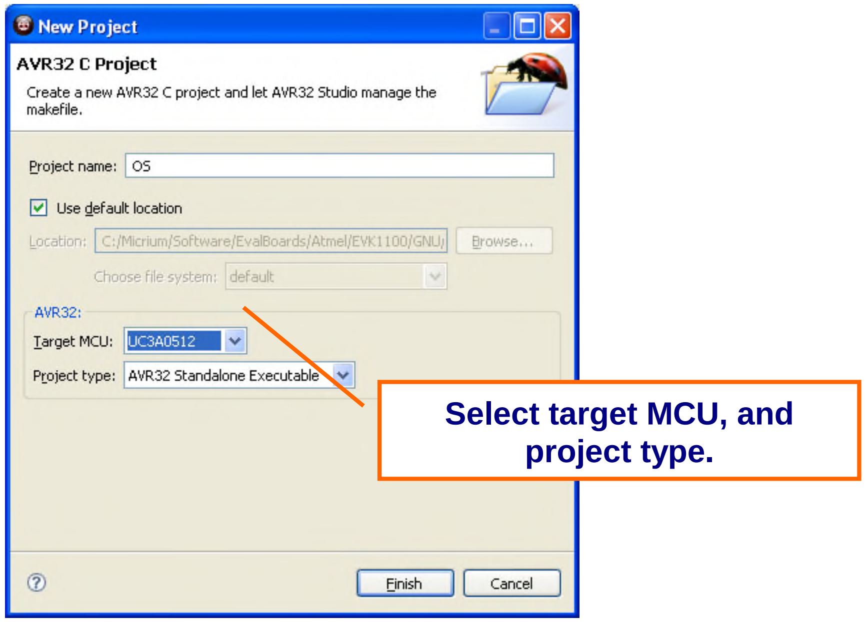
2.03 The AVR32Studio Toolchain

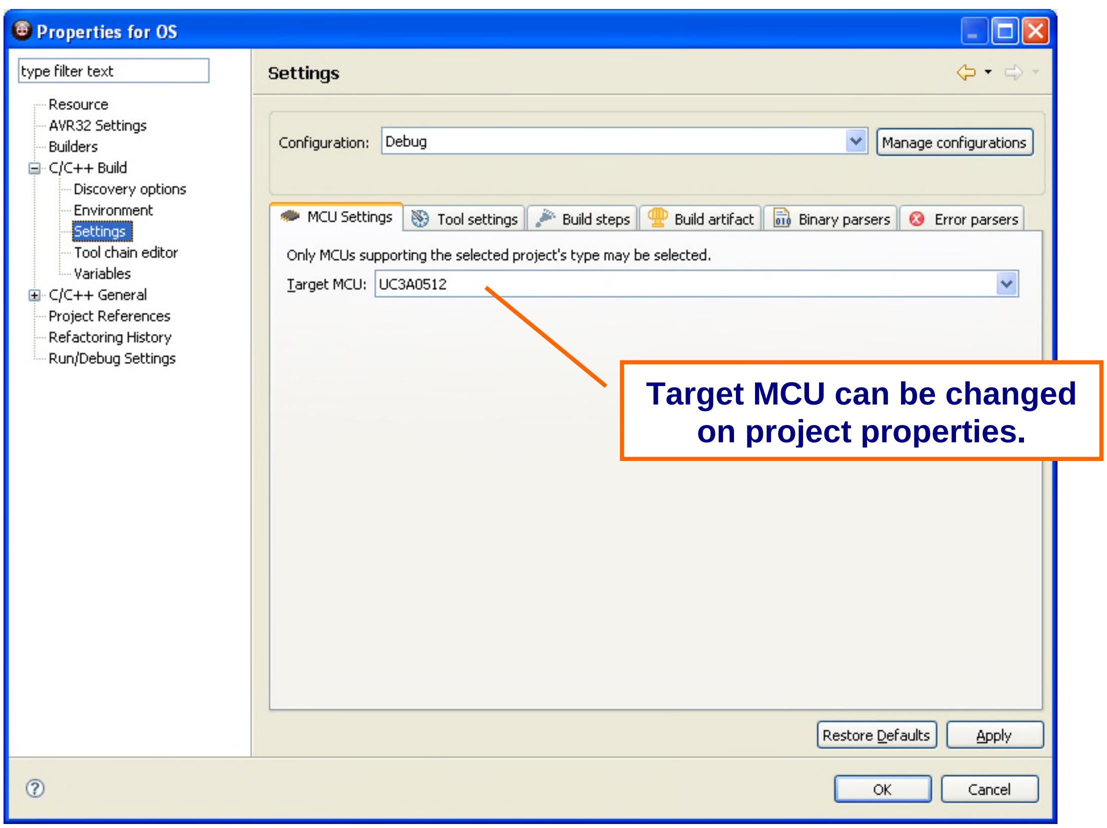
This **μC/OS-II** port was also tested using the AVR32Studio toolchain. The AVR32Studio IDE holds source files and libraries, manages dependencies and stores compiler, linker and other settings. The toolchain uses a GNU cross compiler, assembler, linker, debugger, and flash programmer for AVR32 devices.

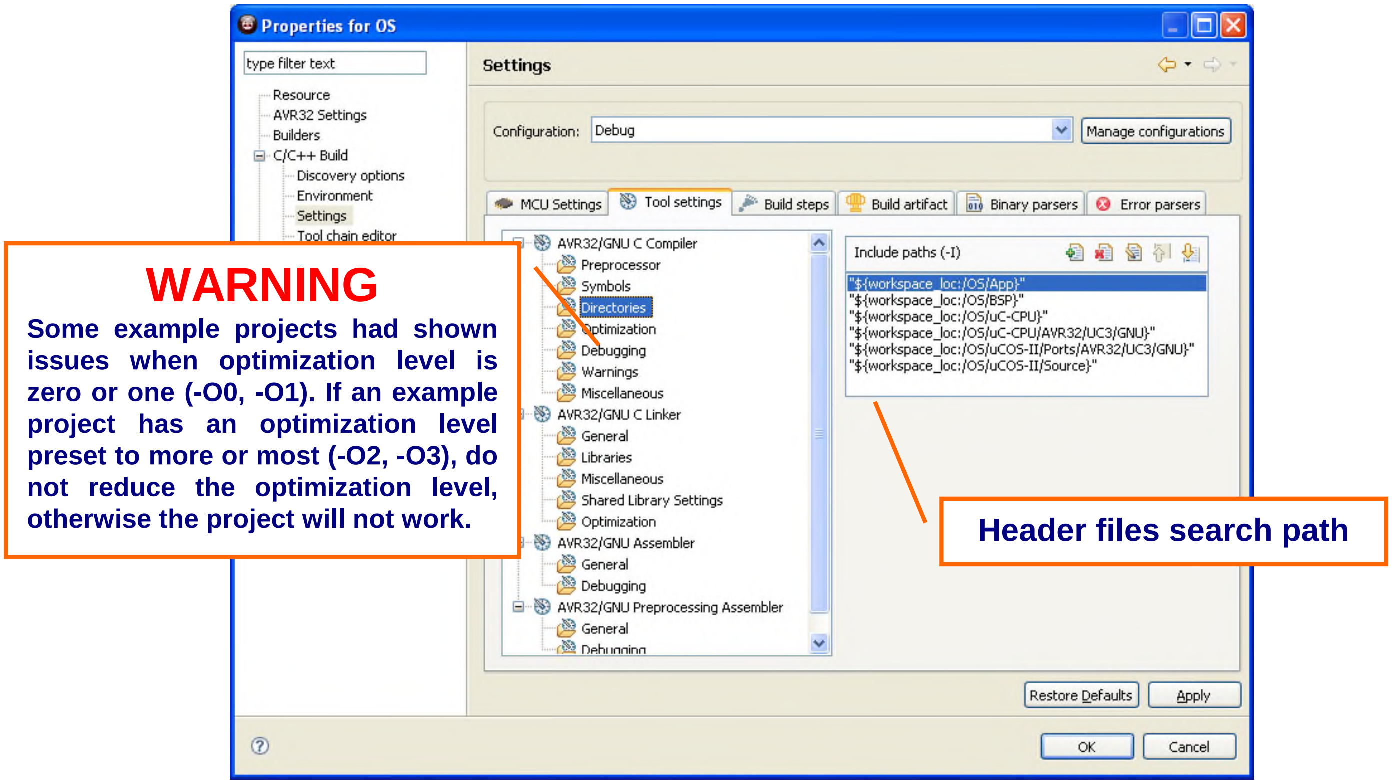
Below are screen shots of some of the toolchain's settings:

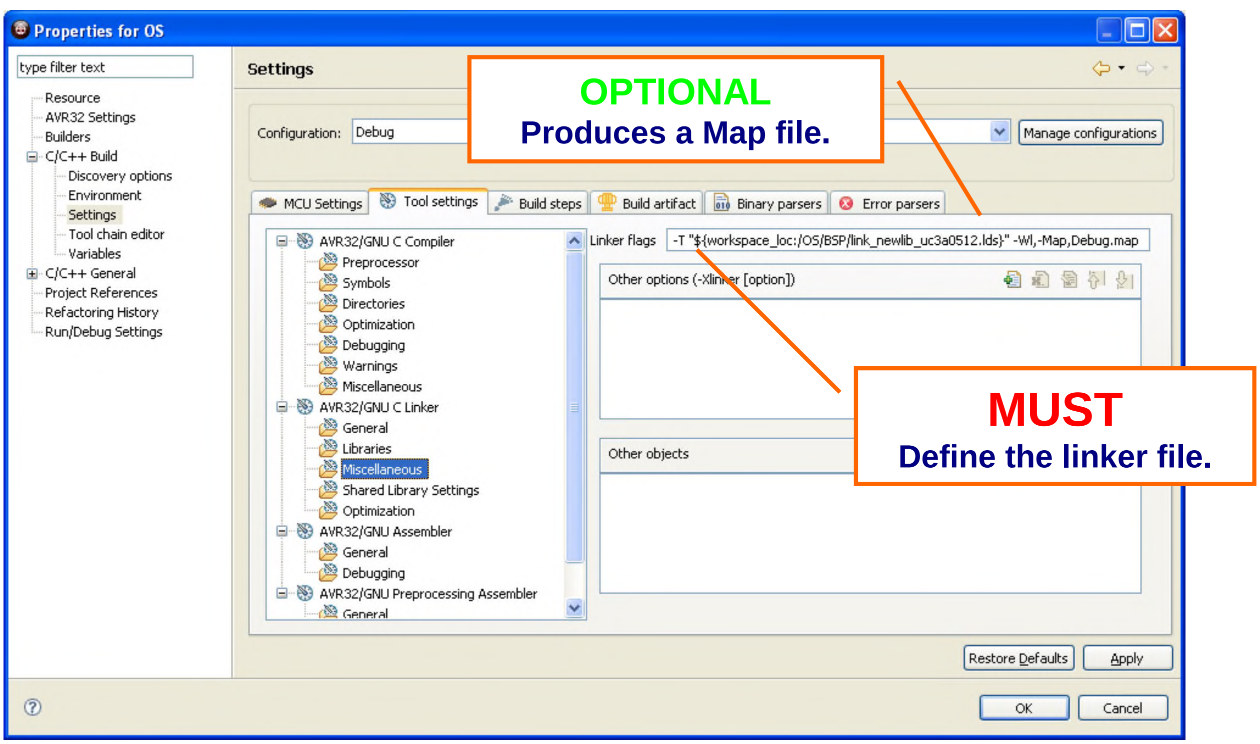





New Project
AVR32 C Project
Create a new AVR32 C project and let AVR32 Studio manage the makefile.
Project name: OS
Use default location
Location: C:/Micrium/Software/EvalBoards/Atmel/EVK1100/GNU/
Browse...
Choose file system: default
AVR32:
Target MCU: UC3A0512
Project type: AVR32 Standalone Executable
Finish
Cancel
Select target MCU, and project type.

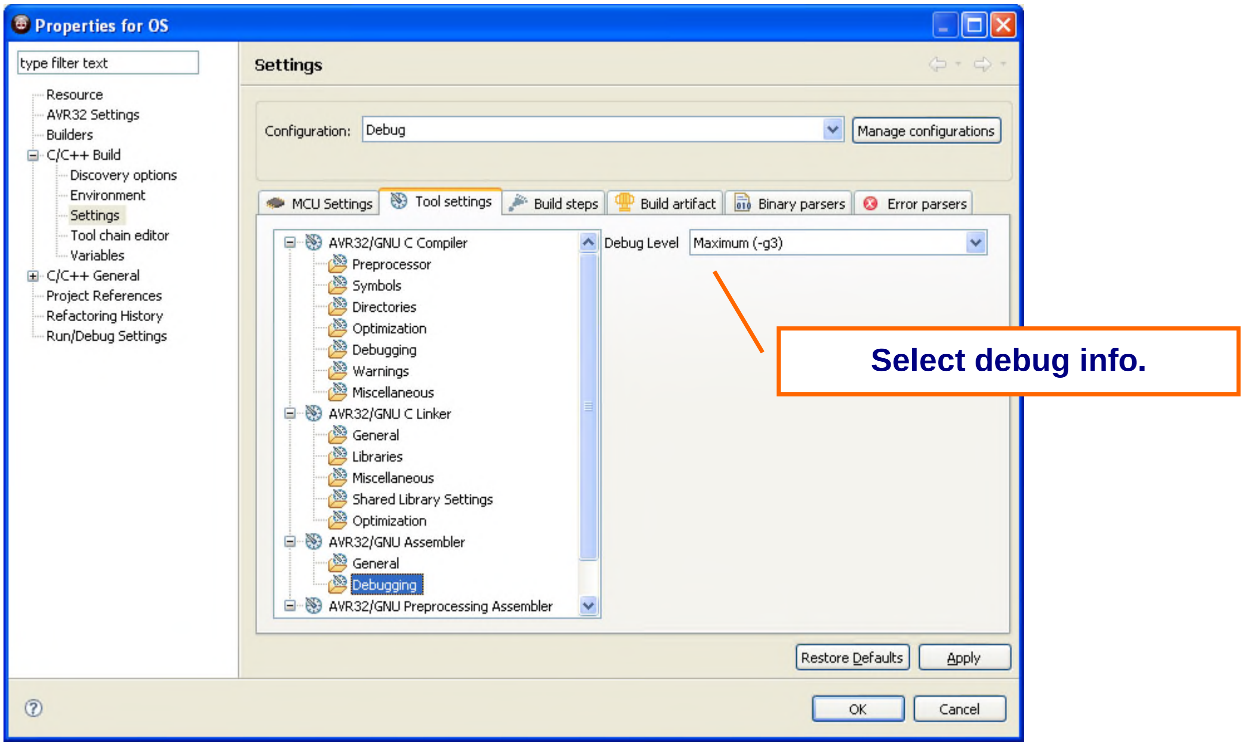

Properties for OS
type filter text
Resource
AVR32 Settings
Builders
C/C++ Build
Discovery options
Environment
Settings
Tool chain editor
Variables
C/C++ General
Project References
Refactoring History
Run/Debug Settings
Settings
Configuration: Debug
Manage configurations
MCU Settings
Tool settings
Build steps
Build artifact
Binary parsers
Error parsers
Only MCUs supporting the selected project's type may be selected.
Target MCU: UC3A0512
Restore Defaults
Apply
OK
Cancel
Target MCU can be changed on project properties.

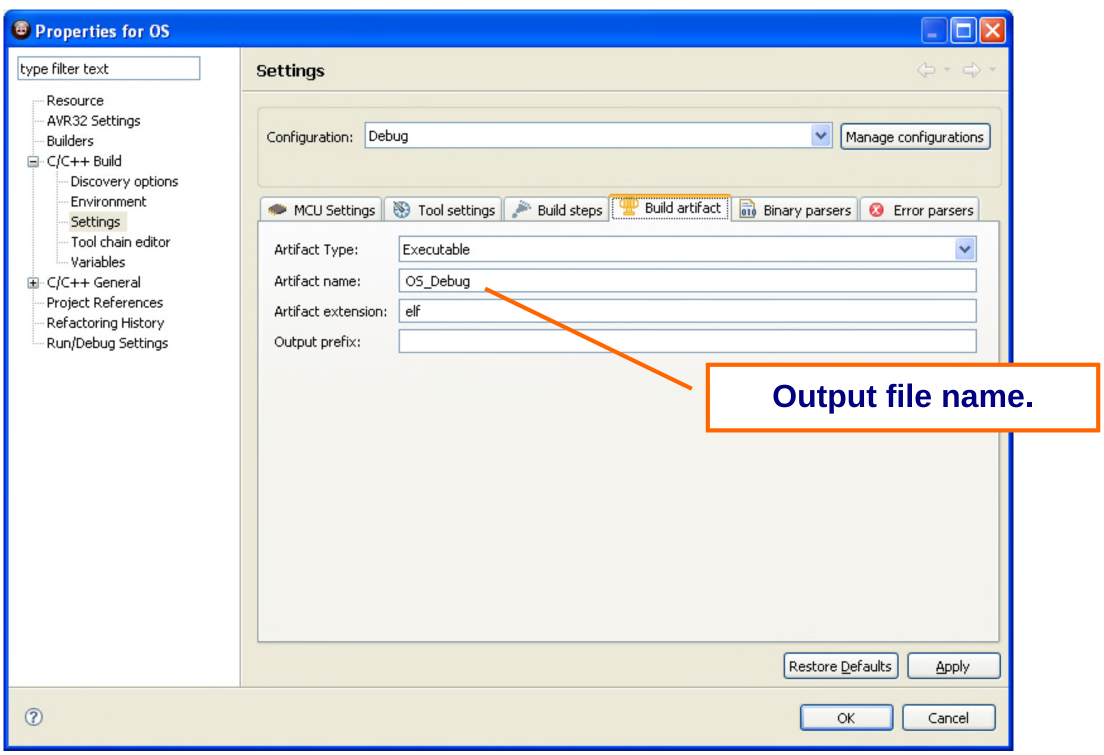


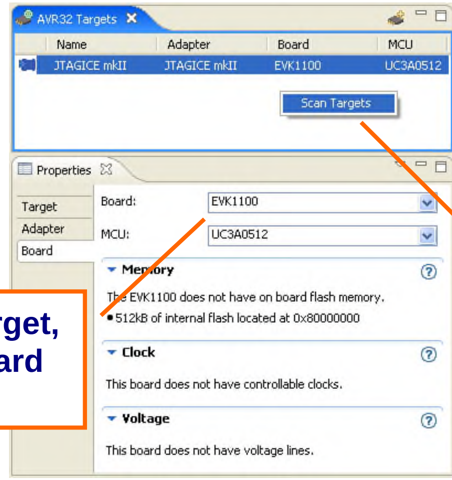


IMPORTANT

The linker file provided **MUST** be include in the linking process in order for the exception table and interrupt vectors be properly aligned in the memory space.


Properties for OS
type filter text
Resource
AVR32 Settings
Builders
C/C++ Build
Discovery options
Environment
Settings
Tool chain editor
Variables
C/C++ General
Project References
Refactoring History
Run/Debug Settings
Settings
Configuration:
Debug
Manage configurations
MCU Settings
Tool settings
Build steps
Build artifact
Binary parsers
Error parsers
AVR32/GNU C Compiler
Preprocessor
Symbols
Directories
Optimization
Debugging
Warnings
Miscellaneous
AVR32/GNU C Linker
General
Libraries
Miscellaneous
Shared Library Settings
Optimization
AVR32/GNU Assembler
General
Debugging
AVR32/GNU Preprocessing Assembler
Debug Level
Maximum (-g3)
Select debug info.
Restore Defaults
Apply
OK
Cancel


Properties for OS
type filter text
Resource
AVR32 Settings
Builders
C/C++ Build
Discovery options
Environment
Settings
Tool chain editor
Variables
C/C++ General
Project References
Refactoring History
Run/Debug Settings
Settings
Configuration:
Debug
Manage configurations
MCU Settings
Tool settings
Build steps
Build artifact
Binary parsers
Error parsers
Artifact Type:
Executable
Artifact name:
OS_Debug
Artifact extension:
elf
Output prefix:
Output file name.
Restore Defaults
Apply
OK
Cancel

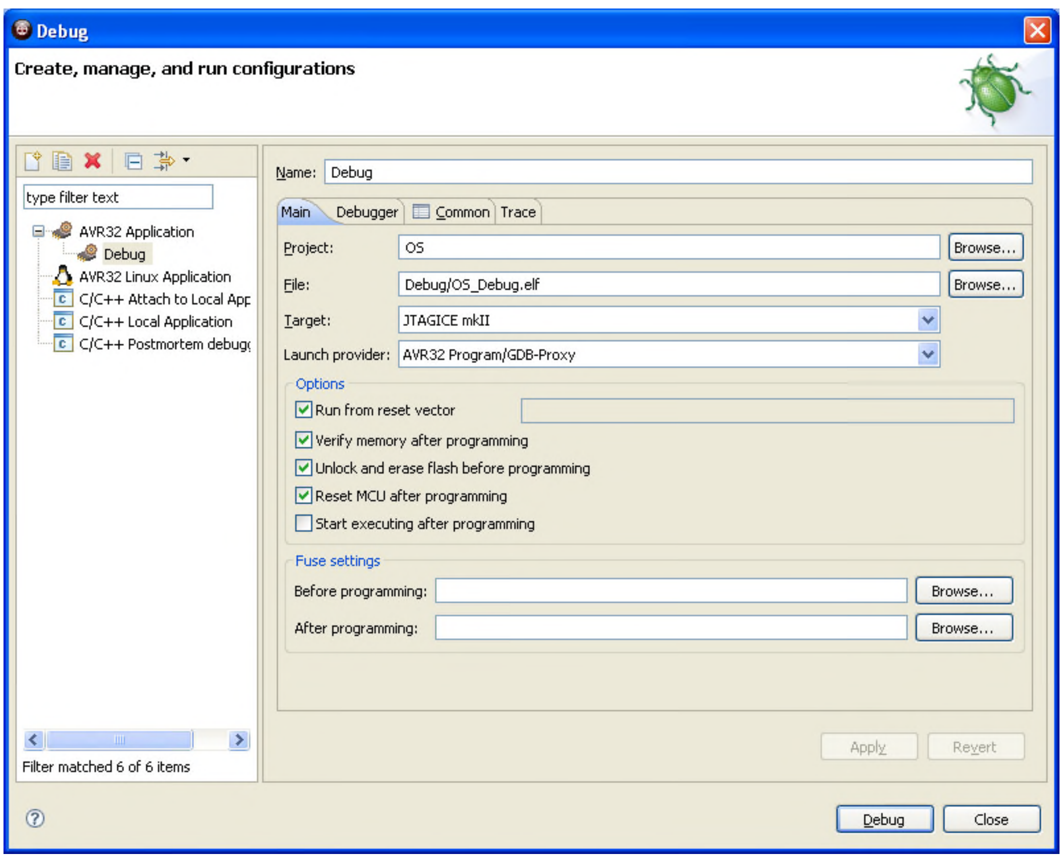


Select JTAGICE mkII target, then the evaluation board and MCU.

MUST

Scan for targets.

Debug options and configuration.



3.00 BSP (Board Support Package) for AVR32 UC3 EVK1100

It is often convenient to create a Board Support Package (BSP) for the target hardware. A BSP allows the encapsulation of commonly used functionality which eases the application port, since the functions provided can be freely used by the application. The BSP for the AVR32 UC3 EVK1100 evaluation board supports the following:

Configuration of CPU and peripheral clock frequencies;
Determine CPU and peripheral clock frequencies;
Configure the LED I/Os;
Configuration and handling of the μC/OS-II tick timer;
Interrupt handling for the μC/OS-II tick timer;
Initialization of the μC/OS-View and μC/Probe communication channel;
Configuration and handling of the μC/OS-View and μC/Probe measurement timer;
Interrupt handling for the μC/OS-View and μC/Probe data transfers;
Configuration of the interrupt services;
General I/O access functions.

The BSP consist of 2 files: `BSP.C` and `BSP.H`. The `VECTORS.ASM` contains the exception vectors and it is also included under the BSP section. Since in general only the supervisor call exception is required in the application, `VECTORS.ASM` would remain unaltered.

3.01 Directories and Files

The software described in this section of this application note is assumed to be placed in the following directory:

`\Micrium\Software\EvalBoards\Atmel\EVK1100\IAR\BSP`

`\Micrium\Software\EvalBoards\Atmel\EVK1100\GNU\$Project\BSP`

The section for each source code file is found as follows:

`BSP.H`	Section 3.02
`BSP.C`	Section 3.03
`VECTORS.ASM`	Section 3.04

3.02 BSP.H & BSP.C

The BSP implements several global functions to give the application access to important services available in the target board. Also, several local functions are defined to perform some atomic duty, such as clock setup, initialization of the interrupt controller, LED's, **μC/OS-II** tick timer, amount others. The discussion of the BSP will be limited to the global functions that might be called from the application code.

3.02.01 BSP, BSP_Init()

The application code must call `BSP_Init()` to initialize the BSP. `BSP_Init()` is responsible for the initialization of the services provided by the BSP.

```
void  BSP_Init (void)
{
    BSP_PM_OSC0Setup();                                       /* (1)                              */
    BSP_PM_ClkSwitch(AVR32_PM_MCSEL_OSC0);                    /* (2)                              */

    BSP_PM_PLL0Setup(8, 1, 1);                                /* (3)                              */
    BSP_PM_GClkSetup(1);                                      /* (4)                              */
    BSP_PM_ClkSelect(0, 0, 1, 0, 0, 0);                       /* (5)                              */

    BSP_FLASHC_SetWaitState(1);                               /* (6)                              */

    BSP_PM_ClkSwitch(AVR32_PM_MCSEL_PLL0);                    /* (7)                              */

    LED_Init();                                               /* (8)                              */

    BSP_INTC_Init();                                          /* (9)                              */
    BSP_INTC_IntReg(&BSP_TmrTickISR, BSP_IRQ_COMPARE, 3);     /* (10)                             */
#if ((OS_VIEW_MODULE > 0) || (uC_PROBE_COM_MODULE > 0))
    BSP_INTC_IntReg(&BSP_USARTRxTxISR, BSP_IRQ_USART, 0);     /* (11)                             */
#endif

    BSP_TmrInit();                                            /* (12)                             */
}
```

Listing 3-1, BSP.C, `BSP_Init()`

L3-1(1)-(2) Configure Oscillator 0 and switch the main clock to it.

L3-1(3) Configure PLL 0 to run at 48MHz (mul: 8, div: 1, and div by 2). Note that the external crystal frequency in the AVR32 UC3 EVK1100 evaluation board is 12MHz.

L3-1(4) Configure and activate Generic Clock 0 using PLL 0 as clock source.

L3-1(5) Configure PBA to half of main clock's frequency.

L3-1(6) Since HSB frequency > 30MHz, set the Flash Controller's wait state.

L3-1(7) Switch the main clock to the PLL 0.

L3-1(8) Initialize LED service.

L3-1(9) Initialize Interrupt support.

L3-1(10) Register **μC/OS-II** timer tick interrupt handler.

L3-1(11) If **μC/OS-View** or **μC/Probe** is part of the project build (`OS_VIEW_MODULE` or `uC_PROBE_COM_MODULE` is set to `1` in the `OS_CFG.H`), then register serial data transfer interrupt handler.

L3-1(12) Initialize **μC/OS-II** timer tick service.

3.02.02 BSP, BSP_INTC_Init()

This function initializes the interrupt controller with all interrupt groups using interrupt priority level 0 and with the unhandled interrupt user-handler. Therefore, if a non-registered interrupt becomes active, it will be caught by the unhandled interrupt user-handler.

```
static  void  BSP_INTC_Init (void)
{
    CPU_INT32U  int_grp;
    CPU_INT32U  int_req;


    for (int_grp = 0; int_grp < AVR32_INTC_NUM_INT_GRPS; int_grp++) {                 /* (1)  */
        for (int_req = 0; int_req < BSP_INTC_Handlers[int_grp].num_irqs; int_req++) {  /* (2)  */
            BSP_INTC_Handlers[int_grp].handlers[int_req] = &BSP_INTC_IntUnhandled;     /* (3)  */
        }

        AVR32_INTC.ipr[int_grp] =  OSIntPrioReg[BSP_INTC_INT0] |                      /* (4)  */
                                  (BSP_INTC_INT0 << BSP_INTC_IPR_INTLEVEL_OFFSET);
    }
}
```

Listing 3-2, BSP.C, `BSP_INTC_Init()`

L3-2(1) Loop thru all interrupt groups.

L3-2(2) Loop thru each interrupt request line inside an interrupt group.

L3-2(3) Assign `BSP_INTC_IntUnhandled()` as the default interrupt handler.

L3-2(4) Set each interrupt group's IPR to interrupt handler and priority of level 0.

`BSP_INTC_Init()` must be called prior to enable global interrupts, otherwise any unhandled interrupt will generate an unexpected behavior.

3.02.03 BSP, BSP_INTC_IntUnhandled()

The `BSP_INTC_IntUnhandled()` is an infinite loop that halts the execution of the code. Under proper registration and activation of an interrupt service, `BSP_INTC_IntUnhandled()` is never executed, since `BSP_INTC_IntReg()` loads the user-defined handler for the specific interrupt service. The existence of `BSP_INTC_IntUnhandled()` is for the case that an interrupt service has been activated but not registered with `BSP_INTC_IntReg()`. In this case, the execution is trapped by the `BSP_INTC_IntUnhandled()` instead of branching to an unknown address in memory.

```
static  void  BSP_INTC_IntUnhandled (void)
{
    while (1) {
        ;
    }
}
```

Listing 3-3, BSP.C, `BSP_INTC_IntUnhandled()`

3.02.04 BSP, BSP_INTC_IntReg()

Once the interrupt support has been initialized by `BSP_INTC_Init()`, the application can freely register user-defined interrupt handlers for any interrupt request. User-defined interrupt handlers are registered to an interrupt request by `BSP_INTC_IntReg()`.

```
CPU_INT32U  BSP_INTC_IntReg (CPU_FNCT_PTR handler, CPU_INT32U irq, CPU_INT32U int_level)
{
    CPU_INT32U  int_grp;
    CPU_INT32U  int_id;


    int_grp = irq / BSP_INTC_IRQS_PER_GRP;                                   /* (1)      */
    int_id  = irq % BSP_INTC_IRQS_PER_GRP;                                   /* (2)      */

    if (int_id > BSP_INTC_Handlers[int_grp].num_irqs) {                      /* (3)      */
        return (BSP_INTC_ERR_INVALID_IRQ);
    }

    BSP_INTC_Handlers[int_grp].handlers[int_id] = handler;                   /* (4)      */
    AVR32_INTC.ipr[int_grp]                     = OSIntPrioReg[int_level & 0x3] |  /* (5)      */
                                            ((int_level & 0x3) << BSP_INTC_IPR_INTLEVEL_OFFSET);

    return (BSP_INTC_ERR_NONE);
}
```

Listing 3-4, BSP.C, `BSP_INTC_IntReg()`

L3-4(1) Retrieve interrupt request group.

L3-4(2) Retrieve interrupt request line.

L3-4(3) Return an error if interrupt request line is out of the range of interrupt lines for the retrieved interrupt group.

L3-4(4) Store interrupt handler in the interrupt handlers' table.

L3-4(5) Set interrupt priority register with given interrupt priority level and autovector from `OSIntPrioReg`.

The `irq` argument is the interrupt request number and not the interrupt group number. If a group number is passed instead of a request number, the function may return an invalid IRQ error or it may just register a wrong interrupt request.

An interrupt request group has only one priority level attached to it. If interrupt requests from the same group are registered, the resultant interrupt priority level is the latest registered one.

3.02.05 BSP, BSP_INTC_IntGetHandler()

`BSP_INTC_IntGetHandler()` retrieves the interrupt handler associated with the current interrupt request and level. It is necessary to call `BSP_INTC_Init()` prior to enable global interrupts, otherwise any unhandled interrupt will generate an unexpected behavior.

```
CPU_FNCT_NONE  BSP_INTC_IntGetHandler (CPU_INT32U int_level)
{
    CPU_INT32U  int_grp;
    CPU_INT32U  int_req;
    CPU_INT32U  int_id;


    int_grp = AVR32_INTC_ICR_reg[BSP_INTC_INT3 - int_level];                /* (1)      */
    int_req = AVR32_INTC.irr[int_grp];                                      /* (2)      */
    int_id  = 32 - CPU_CntLeadZeros(int_req) - 1;                           /* (3)      */

    if (int_req == 0 ) {                                                    /* (4)      */
        return (0);                                                         /* (5)      */
    } else {
        return (BSP_INTC_Handlers[int_grp].handlers[int_id]);               /* (6)      */
    }
}
```

Listing 3-5, BSP.C, `BSP_INTC_IntGetHandler()`

L3-5(1) Retrieve interrupt request group causing interrupt of priority level `int_level`.

L3-5(2) Retrieve pending interrupt request lines.

L3-5(3) When multiple interrupt lines are active, the prioritization given by the interrupt controller for the interrupt groups is preserved selecting the highest interrupt line. This is achieved with the help of the counting the leading zeros function `CPU_ CntLeadZeros()`.

L3-5(4)-(5) If there is no request pending in the interrupt request register, a `NULL` pointer is returned to inform the caller function.

L3-5(6) Return the specific user-defined interrupt handler to the caller function.

3.02.06 BSP, General-Purpose Input/Output service functions

A GPIO line may be multiplexed with one or more peripheral functions. When the I/O line is assigned to a peripheral function (corresponding bit in GPER is at 0), the drive of the I/O line is controlled by the peripheral. The peripheral, depending on the value in PMR0 and PMR1, is responsible to determine whether the pin is driven or not. When the I/O line is controlled by the GPIO, the value of ODER (Output Driver Enable Register) determines if the pin is driven or not. When a bit in this register is at 1, the corresponding I/O line is driven by the GPIO. When the bit is at 0, the GPIO does not drive the line. The level driven on an I/O line can be determined by writing OVR (Output Value Register).

The `BSP_GPIO_SetFnct()` configures the peripheral function of a pin. Note the PA, PB, PC and PX ports do not directly correspond to the GPIO ports. The relationship between GPIO port and pin to the GPIO number is given by the following equations:

GPIO port = floor((GPIO number) / 32)
GPIO pin = GPIO number mod 32

```
void  BSP_GPIO_SetFnct (CPU_INT16U pin, CPU_INT08U fnct)
{
    volatile  avr32_gpio_port_t  *gpio_port;


    gpio_port        = &AVR32_GPIO.port[pin / 32];           /* (1)                              */

    switch (fnct) {
        case 0:                                              /* (2)                              */
             gpio_port->pmr0c = 1 << (pin % 32);
             gpio_port->pmr1c = 1 << (pin % 32);
             break;

        case 1:                                              /* (3)                              */
             gpio_port->pmr0s = 1 << (pin % 32);
             gpio_port->pmr1c = 1 << (pin % 32);
             break;

        case 2:                                              /* (4)                              */
             gpio_port->pmr0c = 1 << (pin % 32);
             gpio_port->pmr1s = 1 << (pin % 32);
             break;

        case 3:                                              /* (5)                              */
             gpio_port->pmr0s = 1 << (pin % 32);
             gpio_port->pmr1s = 1 << (pin % 32);
             break;
    }

    gpio_port->gperc = 1 << (pin % 32);                      /* (6)                              */
}
```

Listing 3-6, BSP.C, `BSP_GPIO_SetFnct()`

L3-6(1) Retrieve port register from the pin number.

L3-6(2) Configure the peripheral function A. Clear PMR0 and PMR1 (Peripheral Mux Register).

L3-6(3) Configure the peripheral function B. Set PMR0 and clear PMR1.

L3-6(4) Configure the peripheral function C. Clear PMR0 and set PMR1.

L3-6(5) Configure the peripheral function D. Set PMR0 and PMR1.

L3-6(6) Give peripheral control of the pin.

BSP_GPIO_SetPin(), BSP_GPIO_ClrPin(), and BSP_GPIO_TglPin(), shown in listing 3-7, are functions responsible for changing the logical level of a pin. These functions modify the level driven on an I/O line by altering the OVR (Output Value Register) thru OVRS, OVRC, and OVRT, respectively. Once the I/O level is altered, the output driver is enable and the control of the pin is given to the GPIO controller. Listing 3-8 shows the generic code for these functions.

```
    void  BSP_GPIO_SetPin(CPU_INT16U pin);
    void  BSP_GPIO_ClrPin(CPU_INT16U pin);
    void  BSP_GPIO_TglPin(CPU_INT16U pin);
```

Listing 3-7, BSP.C, GPIO control functions

```
void  BSP_GPIO_???Pin (CPU_INT16U pin)                        /* (1)                            */
{
    volatile  avr32_gpio_port_t  *gpio_port;


    gpio_port        = &AVR32_GPIO.port[pin / 32];            /* (2)                            */

    gpio_port->ovr?  = 1 << (pin % 32);                       /* (3)                            */
    gpio_port->oders = 1 << (pin % 32);                       /* (4)                            */
    gpio_port->gpers = 1 << (pin % 32);                       /* (5)                            */
}
```

Listing 3-8, BSP.C, BSP_GPIO_???Pin()

L3-8(1) Each GPIO control function has its own definition: BSP_GPIO_SetPin(), BSP_GPIO_ClrPin(), and BSP_GPIO_TglPin().

L3-8(2) Retrieve port register from the pin number.

L3-8(3) Access the respective output value register operation of the control function: OVRS, OVRC, and OVRT.

L3-8(4) Enable GPIO output driver for that pin.

L3-8(5) Give GPIO control of the pin.

3.02.07 BSP, LED service functions

A number of evaluation boards are equipped with LED's. For this reason, the BSP has a collection of control functions which creates a standard access to the LED's from the application perspective. The LED control functions use internally the `BSP_GPIO_???Pin()` control functions to change the state of the LED's. Listing 3-9 shows the available LED control functions.

```
    void  LED_On(CPU_INT08U led);
    void  LED_Off(CPU_INT08U led);
    void  LED_Toggle(CPU_INT08U led);
```

Listing 3-9, BSP.C, LED control functions' prototypes

3.02.08 BSP, OS Timer functions

The **μC/OS-II** clock tick ISR handler and its initialization function have been encapsulated in the BSP. This makes it easier to adapt the **μC/OS-II** port to different target hardware since these functions can be changed to select whichever timer or interrupt source for the clock tick that the application requires.

The AVR32 UC3 has an internal 32-bits free-running clock with interrupt capability. For this reason, it has been chosen as the OS tick source as oppose to dedicate one of the three available timer counters.

To initialize the OS tick source, `BSP_TmrInit()` must be called by the application code. Listing 3-10 shows the function source code.

```
static  void  BSP_TmrInit (void)
{
    CPU_INT32U  cycle;


    cycle  =  CPU_SysReg_Get_Count();                                           /* (1)         */
    cycle += (CPU_CLK_FREQ() / OS_TICKS_PER_SEC);                               /* (2)         */

    if (cycle == 0) {                                                           /* (3)         */
        cycle++;
    }

    CPU_SysReg_Set_Compare(cycle);                                              /* (4)         */
}
```

Listing 3-10, BSP.C, `BSP_TmrInit()`

L3-10(1) Retrieve current system Counter register value.

L3-10(2) Add number of clock ticks for the next interrupt trigger.

L3-10(3) If cycle ends up to be 0, make it 1 so interrupt generation feature does not get disabled.

L3-10(4) Save next interrupt clock cycle value into the system Compare register.

At the BSP initialization, `BSP_TmrTickISR()` is registered as the interrupt handler for the Compare interrupt. The next Compare interrupt trigger must be updated inside the interrupt handler, otherwise the next OS tick will not occur. This update is done by the `BSP_TmrReload()` function, shown in listing 3-12. Listing 3-11 shows the interrupt handler code.

```
void  BSP_TmrTickISR (void)
{
    BSP_TmrReload();                                                                /* (1)          */
    OSTimeTick();                                                                   /* (2)          */
}
```

Listing 3-11, BSP.C, `BSP_TmrTickISR()`

L3-11(1) Schedule next Compare interrupt.

L3-11(2) Signal a clock tick to the OS.

Note the similarity between `BSP_TmrReload()` and `BSP_TmrInit()`. The only difference between them is that `BSP_TmrInit()` retrieves the current value of the system Count register, and `BSP_TmrReload()` retrieves the current value of Compare register. This procedure guarantees the OS tick does not get skewed after every single register update. If the current system Count register had been used in the update process, the time between the interrupt trigger and the point into the handler that reads the current Count register would have been added in the next tick interrupt elapsed time.

```
static  void  BSP_TmrReload (void)
{
    CPU_INT32U  cycle;


    cycle  =  CPU_SysReg_Get_Compare();                                             /* (1)          */
    cycle += (CPU_CLK_FREQ() / OS_TICKS_PER_SEC);                                   /* (2)          */

    if (cycle == 0) {                                                               /* (3)          */
        cycle++;
    }

    CPU_SysReg_Set_Compare(cycle);                                                  /* (4)          */
}
```

Listing 3-12, BSP.C, `BSP_TmrReload()`

L3-12(1) Retrieve current system Compare register value.

L3-12(2) Add number of clock ticks for the next interrupt trigger.

L3-12(3) If cycle ends up to be 0, make it 1 so interrupt generation feature does not get disabled.

L3-12(4) Save next interrupt clock cycle value into the system Compare register.

4.00 Application Code

The sample application code makes use of the port presented in this application note as described in this section.

4.01 Directories and Files

The software described in this section of this application note is assumed to be placed in the following directory:

```
\Micrium\Software\EvalBoards\Atmel\EVK1100\IAR\OS

\Micrium\Software\EvalBoards\Atmel\EVK1100\GNU\OS_Workspace\App
```

The section for each source code file is found as follows:

`APP.C`	Section 4.02
`APP_CFG.H`	Section 4.03
`INCLUDES.H`	Section 4.04
`OS_CFG.H`	Section 4.05

4.02 APP.C

The sample application code is placed in the file called `APP.C`. An application can contain many more files. `APP.C` is where the `main()` is placed but, it could be placed in any other file in a final application.

`APP.C` is a standard test file for **µC/OS-II** examples. The two important functions are `main()` (listing 4-1) and `AppStartTask()` (listing 4-2).

```
int  main (void)
{
#if (OS_TASK_NAME_SIZE > 7) && (OS_TASK_STAT_EN > 0)
    INT8U  err;
#endif


    CPU_IntDis();

    OSInit();                                                               /* (1)          */

    OSTaskCreateExt(AppStartTask,                                           /* (2)          */
                    (void *)0,
                    (OS_STK *)&AppStartTaskStk[APP_TASK_START_STK_SIZE - 1],
                    APP_TASK_START_PRIO,
                    APP_TASK_START_PRIO,
                    (OS_STK *)&AppStartTaskStk[0],
                    APP_TASK_START_STK_SIZE,
                    (void *)0,
                    OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR);

#if (OS_TASK_NAME_SIZE > 7) && (OS_TASK_STAT_EN > 0)
    OSTaskNameSet(APP_TASK_START_PRIO, (INT8U *)"Startup", &err);           /* (3)          */
#endif

    OSStart();                                                              /* (4)          */

    return (0);
}
```

Listing 4-1, APP.C, `main()`

L4-1(1) As with all **µC/OS-II** based applications, the OS needs to be initialized by calling `OSInit()`.

L4-1(2) At least one task needs to be created. In this case, a task is created using the extended task create call. This allows **µC/OS-II** to have more information about the task. Specifically, with the IAR toolchain, the extra information allows the C-Spy debugger to display stack usage information when you use the **µC/OS-II** Kernel Awareness Plug-In.

L4-1(3) Tasks can be given a name and it can be displayed by the Kernel Aware debuggers such as IAR's C-SPY.

L4-1(4) Call `OSStart()` to start multitasking. Note that `OSStart()` does not return from this call.

```
static void  AppStartTask (void *p_arg)
{
    INT8U  i;


    (void)p_arg;                                                          /* (1)          */

    BSP_Init();                                                           /* (2)          */

#if OS_TASK_STAT_EN > 0
    OSStatInit();                                                         /* (3)          */
#endif

    AppTaskCreate();                                                      /* (4)          */


    while (1) {                                                           /* (5)          */
        for (i = 1; i <= 6; i++) {
            LED_On(i);
            OSTimeDlyHMSM(0, 0, 0, 200);
            LED_Off(i);
        }
        for (i = 4; i <= 7; i++) {
            LED_On(9 - i);
            OSTimeDlyHMSM(0, 0, 0, 200);
            LED_Off(9 - i);
        }
    }
}
```

Listing 4-2, APP.C, `AppStartTask()`

L4-2(1) Prevent compiler warning.

L4-2(2) Initializes the BSP (see Board Support Package section) for the target board.

L4-2(3) If task statistics is enable (`OS_TASK_STAT_EN` is set to `1` in the `OS_CFG.H`) then, the task statistics needs to be initialized. Note that the **μC/OS-II** clock tick needs to be enabled and initialized because `OSStatInit()` assumes the presence of clock ticks. In other words, if the tick ISR is not active when `OSStatInit()` is called, the application will end up in **μC/OS-II**'s idle task and not be able to run any other tasks.

L4-2(4) At this point, additional tasks can be created. All task initialization is placed in one function called `AppTaskCreate()`.

L4-2(5) Additional function can be performed by this task. Note that in order to switch between tasks, the task **MUST** call either one of the `OS???Pend()` functions or `OSTimedly???()` functions. In other words, a task **MUST** always be waiting for an event to occur. An event can be the reception of a signal or a message from another task or ISR, or simply be a wait for the passage of time. If the task does not perform any of these function calls, the OS will never have an opportunity to switch between different tasks.

4.03 APP_CFG.H

`APP_CFG.H` contains configuration options for the sample application code, such as `#define` constants, macros, prototypes, etc. that are specific to the application.

```
#define   APP_TASK_START_STK_SIZE          256            /* (1)                               */
```

Listing 4-3, Stack sizes

L4-3(1) Depth of the stack for the sample application task. Note that the depth of the stack is in number of registers it can hold. Since AVR32 registers are 32-bits wide, the amount of bytes required for the stack is four times the stack size.

```
#define   APP_TASK_START_PRIO                1            /* (1)                               */
#define   OS_TASK_TMR_PRIO                   5            /* (2)                               */
```

Listing 4-4, Task priorities

L4-4(1) Priority for the sample application task. Note that the lower the priority number, the higher the priority of the task.

L4-4(2) Priority for the OS Timer Management task.

4.04 INCLUDES.H

INCLUDES.H is a master include file and it is found at the top of all .C files. INCLUDES.H allows every .C file in the project to be written without concern about which header file is actually needed. The drawbacks to have a master include file are that INCLUDES.H may include header files that are not pertinent to the actual .C file being compiled and the compilation process may take longer. These inconveniences are offset by code portability. The INCLUDES.H can be edited to include additional header files, but any addition should be added at the end of the list. Listing 4-5 shows the typical contents of INCLUDES.H for an AVR32 **µC/OS-II** project.

```
#include      <stdio.h>
#include      <stdarg.h>

#include      <cpu.h>
#include      <app_cfg.h>
#include      <ucos_ii.h>

#if OS_VIEW_MODULE > 0
    #include  <os_viewc.h>
    #include  <os_view.h>
#endif

#include      <bsp.h>

#include      <avr32/io.h>
```

Listing 4-5, INCLUDES.H

4.05 OS_CFG.H

This is the **µC/OS-II** configuration file for the project. The book (MicroC/OS-II, The Real-Time Kernel) describes the configuration elements for the **µC/OS-II**.

```
#define OS_VIEW_MODULE            0                     /* (1)                          */

#define OS_TICKS_PER_SEC        100                     /* (2)                          */
```

Listing 4-6, OS_CFG.H

L4-6(1) **µC/OS-View** is part of the project build if it is set to 1.

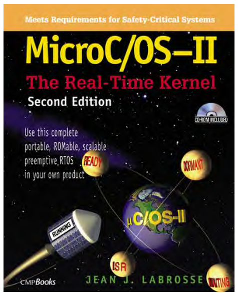
L4-6(2) Number of OS time ticks occurring in one second. This value can be changed as needed but it is typically set between 10 and 1000 Hz. The higher the tick rate, the more overhead **µC/OS-II** will impose on the application. However, a higher tick rate has a better tick granularity.

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

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



Contacts

Atmel Corporation
2325 Orchard Parkway
San Jose, CA 95131
USA
+1 408 441 0311
WEB: www.Atmel.com

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

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

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

Notes