Micrium

Empowering Embedded Systems

µC/OS-II and µC/OS-View

for

The Atmel **AVR**

ATmega128

Application Note

AN-1128 Rev. C

www.Micrium.com

Table Of Contents

1.00 Introduction

This application note describes the μC/OS-II port for the ATmega128. However, the information provided in this application note should be portable to other processors in the AVR line.

We tested the port on an Atmel STK500/501 development board (see Figure 1-1) using an Atmel JTAGICE mkII (see Figure 1-2). The processor is assumed to run at 8 MHz using the on-chip oscillator. The example code changes the prescaler to divide by 1 from the factory default of divide by 8.

We assume that you have μC/OS-II V2.80 (or higher) and optionally, μC/OS-View V1.30 (or higher). If you didn't purchase μC/OS-View from Micriμm you can still run the example code by disabling invocations to that code.

We tested the code using two different sets of tools:

IAR's	EWAVR	(www.IAR.com)
ImageCraft's	ICCAVR	(www.ImageCraft.com)

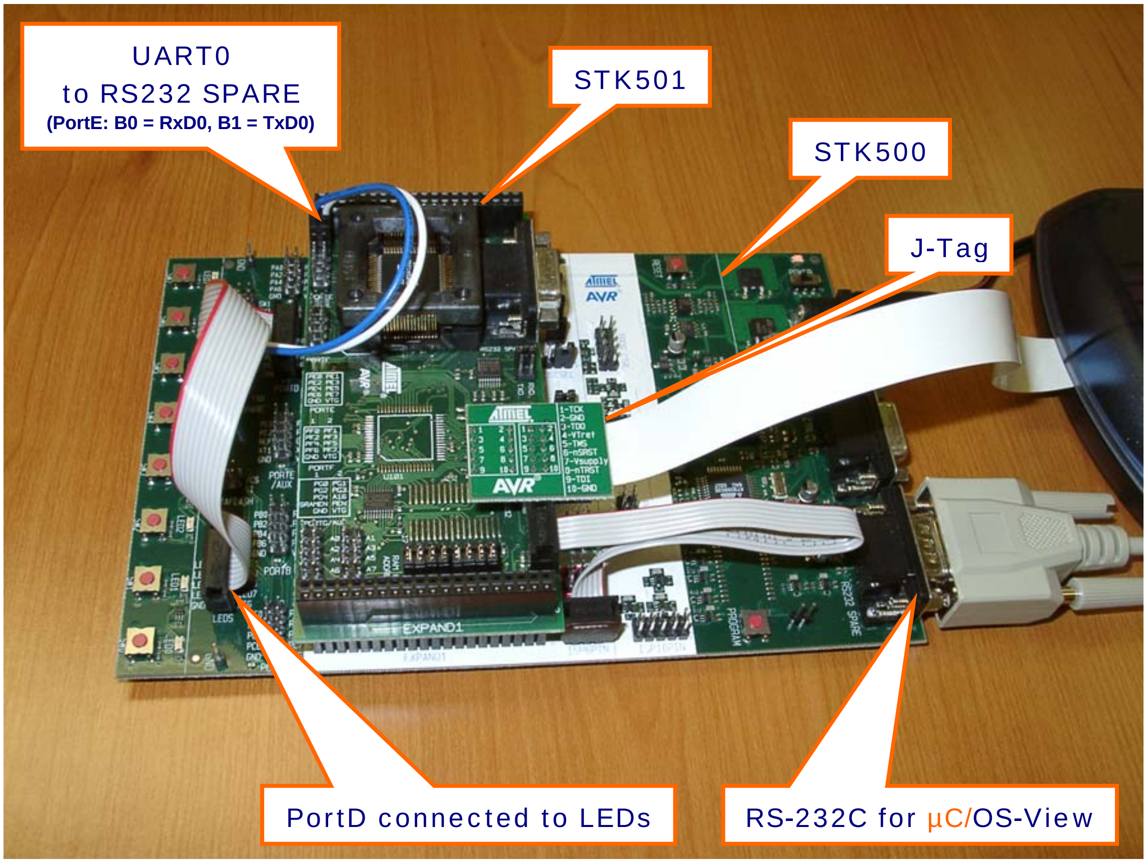


Figure 1-1, Atmel STK500/501 Setup for testing μC/OS-II and μC/OS-View

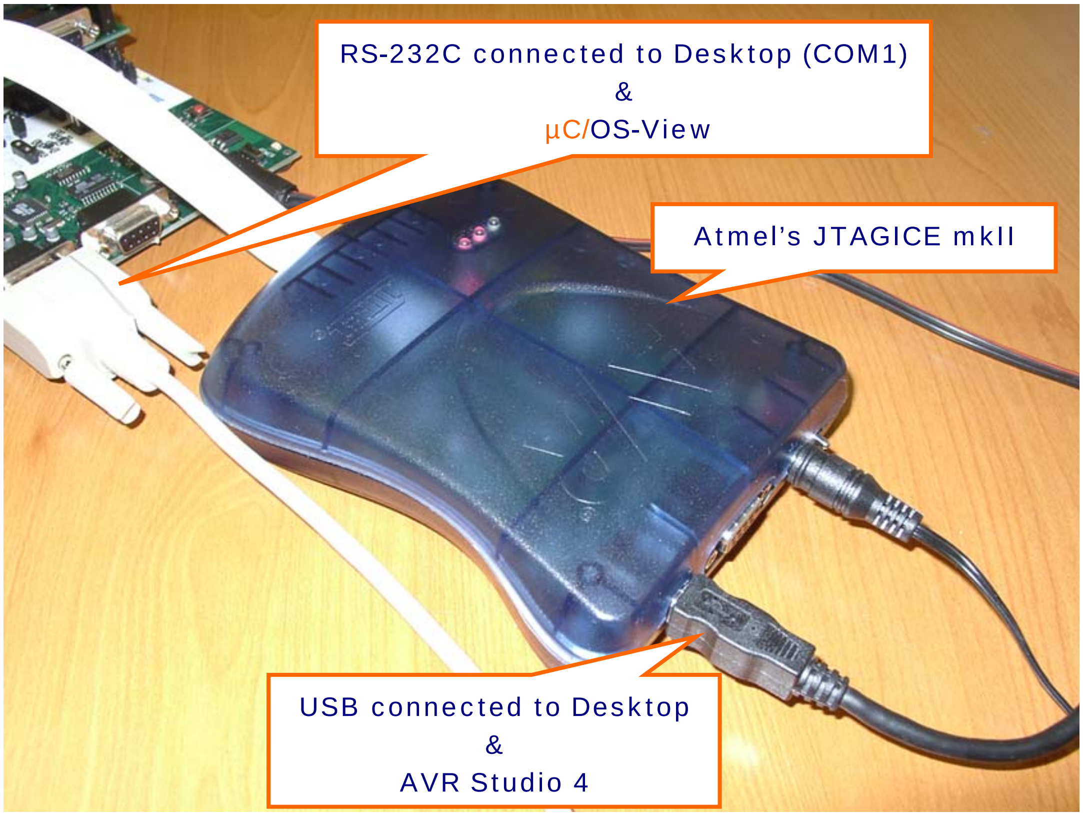


Figure 1-2, Atmel's JTAGICE mkII

1.01 Atmel AT90 (AVR)

The AVR core combines a rich instruction set with 32 general purpose working registers. All the 32 registers are directly connected to the Arithmetic Logic Unit (ALU), allowing two independent registers to be accessed in one single instruction executed in one clock cycle. The resulting architecture is more code efficient while achieving throughputs up to ten times faster than conventional CISC microcontrollers.

The ATmega128 provides the following features: 128K bytes of In-System Programmable Flash with Read-While-Write capabilities, 4K bytes EEPROM, 4K bytes SRAM, 53 general purpose I/O lines, 32 general purpose working registers, Real Time Counter (RTC), four flexible Timer/Counters with compare modes and PWM, 2 USARTs, a byte oriented Two-wire Serial Interface, an 8-channel, 10-bit ADC with optional differential input stage with programmable gain, programmable Watchdog Timer with Internal Oscillator, an SPI serial port, IEEE std. 1149.1 compliant JTAG test interface, also used for accessing the On-chip Debug system and programming and six software selectable power saving modes. The Idle mode stops the CPU while allowing the SRAM, Timer/Counters, SPI port, and interrupt system to continue functioning. A block diagram of the ATmega128 is shown in figure 1-3.

The Powerdown mode saves the register contents but freezes the OscillatorOscillator, disabling all other chip functions until the next interrupt or Hardware Reset. In Power-save mode, the asynchronous timer continues to run, allowing the user to maintain a timer base while the rest of the device is sleeping. The ADC Noise Reduction mode stops the CPU and all I/O modules except Asynchronous Timer and ADC, to minimize switching noise during ADC conversions. In Standby mode, the Crystal/Resonator Oscillator is running while the rest of the device is sleeping. This allows very fast start-up combined with low power consumption. In Extended Standby mode, both the main Oscillator and the Asynchronous Timer continue to run.

The device is manufactured using Atmel's high-density nonvolatile memory technology. The On-chip ISP Flash allows the program memory to be reprogrammed in-system through an SPI serial interface, by a conventional nonvolatile memory programmer, or by an On-chip Boot program running on the AVR core. The boot program can use any interface to download the application program in the application Flash memory. Software in the Boot Flash section will continue to run while the Application Flash section is updated, providing true Read-While-Write operation. By combining an 8-bit RISC CPU with In-System Self-Programmable Flash on a monolithic chip, the Atmel ATmega128 is a powerful microcontroller that provides a highly flexible and cost effective solution to many embedded control applications.

The ATmega128 AVR is supported with a full suite of program and system development tools including: C compilers, macro assemblers, program debugger/simulators, in-circuit emulators, and evaluation kits.

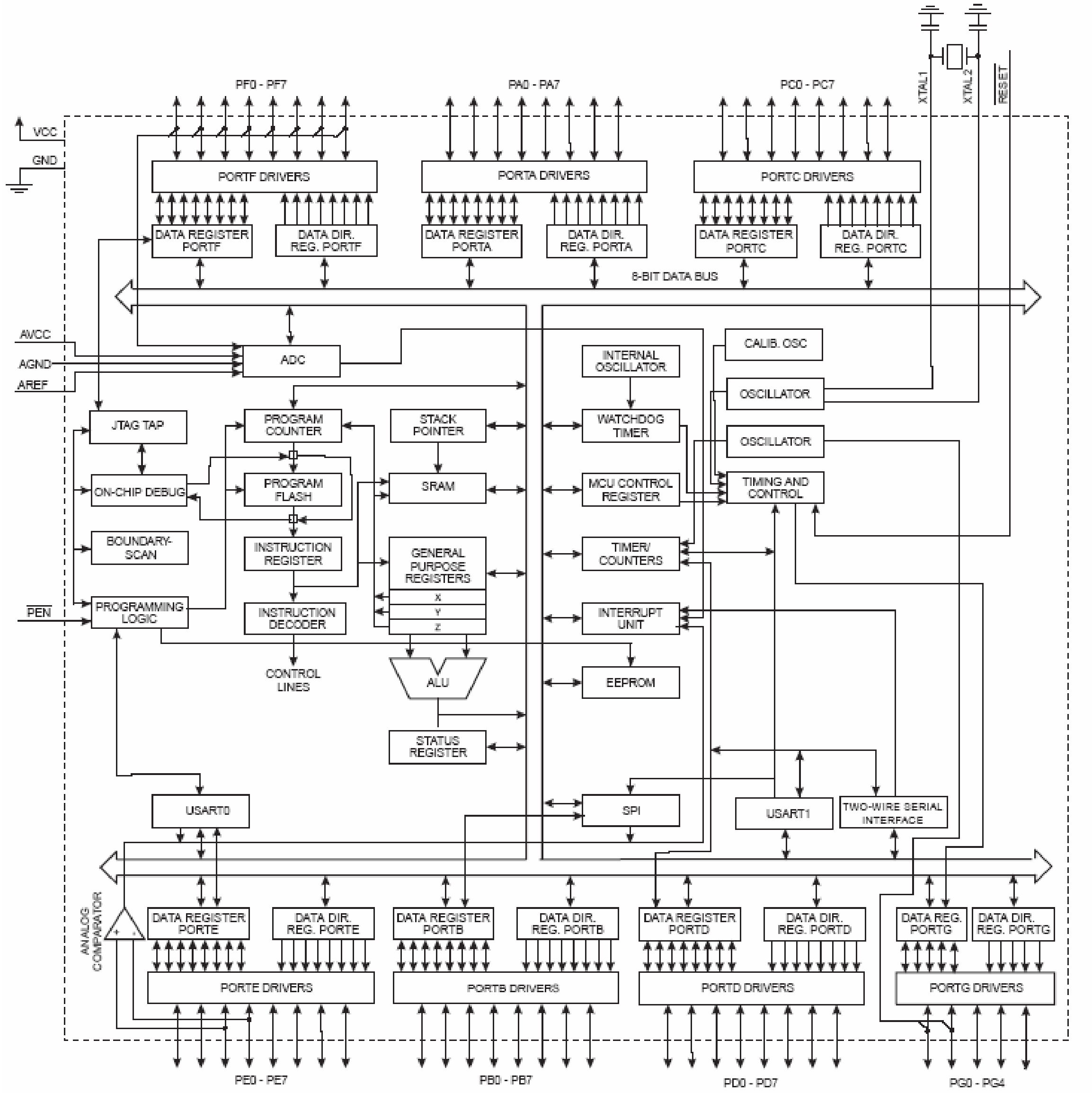


Figure 1-3, Atmel ATmega128

1.02 Test Setup

To test the μC/OS-II port for the AVR, we used an ATmega128 on an STK500/501 evaluation boards. Figure 1-4 and 1-5 show pictures of the boards and the connections from Port D to the LEDs on board.

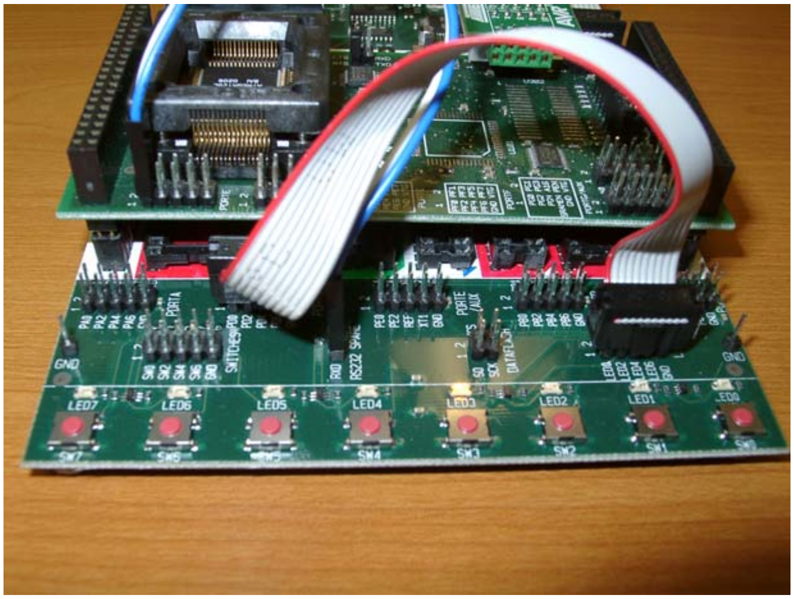


Figure 1-4, Atmel STK500/501 Setup for μC/OS-II port Example #1 (LED connections)

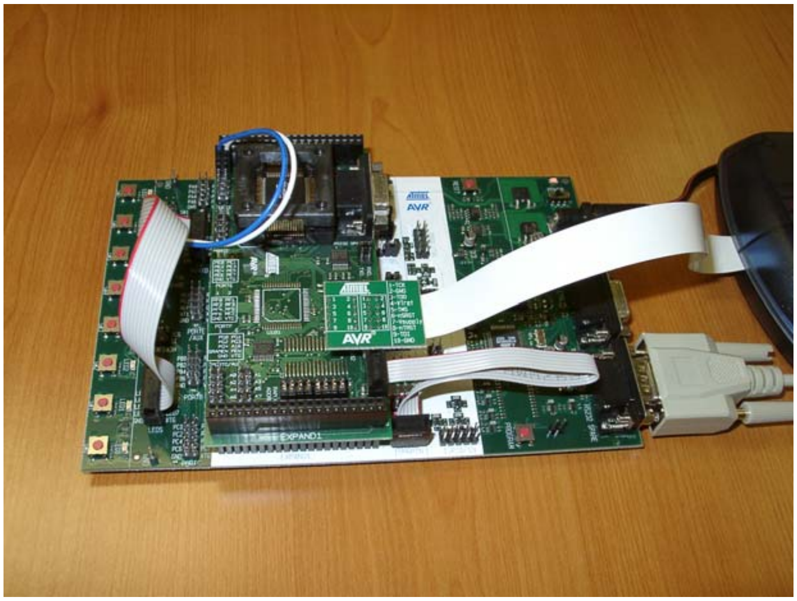


Figure 1-5, Atmel STK500/501 Setup for μC/OS-II port Example #1 (Top View)

1.03 Development Tools

We tested the µC/OS-II port on the IAR (www.IAR.com) compiler and the ImageCraft (www.ImageCraft.com) compiler.

IAR Compiler / Debugger

The IAR Embedded Workbench is a powerful Integrated Development Environment that allows you to develop and manage a complete embedded application project for a variety of target processors in a convenient Windows interface. This IDE is the framework where all necessary tools are integrated: a C/EC++ compiler, an assembler, a linker, an editor, a project manager, and the C-SPYTM Debugger.

The IAR C-SPY Debugger is a high-level language debugger for embedded applications. It is designed for use with the IAR compilers and assemblers, and is completely integrated in the IAR Embedded Workbench IDE, providing seamless switching between development and debugging. Some C-SPY Debuggers are available in a simulator, emulator, and ROM-monitor versions. The simulator version simulates the functions of the target processor entirely in software. The Emulator version provides control over an in-circuit emulator, which is connected to the host computer. The ROM-monitor version provides a low-cost solution to real-time debugging.

C-SPY can be extended with Plug-ins to provide Kernel Awareness capabilities during debugging. Micriµm offers such a plug-in for C-SPY called the:

µC/OS-II Kernel Awareness Plug-in for C-SPY.

C-SPY can actually interface with Atmel’s J-Tag emulator and thus, you can do all your code development and debugging from within the IAR environment.

ImageCraft Compiler

ImageCraft provides offer quality ANSI C tools wrapped in an easy-to-use modern GUI Development Environment. They also provide excellent customer support that other companies cannot match. Add in the low cost factor, and you have the best deal in C tools for modern 8/16 bit microcontrollers such as the AVR.

Atmel AVR Studio 4

AVR Studio® 4 is the new professional Integrated Development Environment (IDE) for writing and debugging AVR® applications in Windows 9x/NT/2000 environments. AVR Studio 4 includes an assembler and a simulator. The following AVR development tools are also supported: ICE50, ICE40, JTAGICE, ICE200, STK500/501/502 and AVRISP.

AVR Studio® is an Integrated Development Environment for writing and debugging AVR applications in Windows® 98/XP/ME/2000 and Windows NT® environments. AVR Studio provides a project management tool, source file editor and chip simulator. It also interfaces with In-Circuit Emulators and development boards available for the AVR 8-bit RISC family of microcontrollers.

- Integrated Development Environment for Writing, Compiling and Debugging Software
- Fully Symbolic Source-level
- Debugger Configurable Memory Views, Including SRAM, EEPROM, Flash, Registers, and I/Os
- Unlimited Number of Break Points
- Trace Buffer and Trigger Control
- Online HTML Help
- Variable Watch/Edit Window with Drag-and-drop Function
- Extensive Program Flow Control Options
- Simulator Port Activity Logging and Pin Input Stimuli
- File Parser Support for COFF, UBROF6, UBROF8, and Hex Files
- Support for C, Pascal, BASIC and Assembly Languages

Atmel JTAGICE mkII

The JTAGICE mkII from Atmel Corporation is together with AVR Studio a complete tool for doing On-Chip Debugging on all AVR 8-bit RISC microcontrollers with the JTAG interface.

Figure 1-6, Atmel JTAG ICE

The JTAG interface is a 4 wire Test Access Port (TAP) controller that is compliant with the IEEE 1149.1 standard. The IEEE standard was developed to enable a standard way to efficiently test circuit board connectivity (Boundary Scan). Atmel AVR devices have extended this functionality to include full Programming and On-Chip Debugging support.

The JTAGICE mkII uses the standard JTAG interface to enable the user to do real time emulation of the microcontroller while it is running in the target system.

The AVROCD (AVR On-Chip Debug) protocol gives the user complete control of the internal resources of the AVR microcontroller. The JTAG ICE gives perfect emulation at a fraction of the cost of traditional emulators.

- AVR Studio Compatible
- Supports all AVR Devices with JTAG Interface
- Exact Electrical Characteristics
- Emulates all Digital and Analog On-Chip Functions
- Complex Breakpoints Like Break on Change of Program Flow
- Data and Program Memory Breakpoints
- Supports Assembler and HLL Source Level Debugging
- Programming interface to flash, eeprom, fuses and lockbits.
- RS232 Interface to PC for Programming and Control
- Regulated Power Supply for 9-15V DC Power

You should note that the IAR C-SPY debugger currently directly supports the JTAGICE mkII and thus, you can do all you development from within the EWAVR environment whereas debugging with the ImageCraft tools requires the use of AVRstudio4 (which is a free package).

2.00 Directories and Files

µC/OS-II

The AVR port files (described in section 3) are placed in the following directory:

Using IAR's EWAVR:

`\Micrium\Software\uCOS-II\Ports\AVR\ATmega128\`**`IAR`**

The port files are:

`os_cpu.h`
`os_cpu_a.s90`
`os_cpu_c.c`
`os_cpu_i.s90`
`os_dbg.c`

Using ImageCraft's ICCAVR:

`\Micrium\Software\uCOS-II\Ports\AVR\ATmega128\`**`ICC`**

The port files are:

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

µC/OS-View

The AVR port files (described in section 4) are placed in the following directory:

Using IAR's EWAVR:

`\Micrium\Software\uCOSView\Ports\AVR\ATmega128\`**`IAR`**

The port files are:

`OS_VIEWc.c`
`OS_VIEWa.s90`
`OS_VIEW.h`

Using ImageCraft's ICCAVR:

`\Micrium\Software\uCOSView\Ports\AVR\ATmega128\`**`ICC`**

The port files are:

`OS_VIEWc.c`
`OS_VIEWa.s`
`OS_VIEW.h`

Sample Code

Sample code is found in the following directories:

```
\Micrium\Software\EvalBoards\Atmel\STK500\ATmega128\IAR\Ex1-OS
\Micrium\Software\EvalBoards\Atmel\STK500\ATmega128\IAR\Ex1-OS-View
```

and,

```
\Micrium\Software\EvalBoards\Atmel\STK500\ATmega256\ICC\Ex1-OS
```

3.00 µC/OS-II Port Files

Like all µC/OS-II ports, the port code is found in the following three to five files:

`os_cpu.h`	
`os_cpu_c.c`	
`os_cpu_a.s90`	(**`os_cpu_a.s`** for the ICC tools)
`os_cpu_i.s90`	(**`os_cpu_i.h`** for the ICC tools)
`os_dbg.c`	

`OS_DBG.C` is only needed for the IAR port because it's used by the kernel awareness plug-in in IAR's C-SPY and AVRstudio 4.

In the following sections we will assume IAR's EWAVR but we'll discuss differences between ImageCraft's ICCAVR as needed.

3.01 OS_CPU.H

3.01.01 OS_CPU.H, macros for 'externals'

`OS_CPU_GLOBALS` and `OS_CPU_EXT` allows us to declare global variables that are specific to this port (described later).

Listing 3-1, OS_CPU.H, Globals and Externs

```
#ifdef  OS_CPU_GLOBALS
#define OS_CPU_EXT
#else
#define OS_CPU_EXT  extern
#endif
```

3.02.02 OS_CPU.H, Data Types

Listing 3-2, OS_CPU.H, Data Types

```
typedef unsigned char  BOOLEAN;
typedef unsigned char  INT8U;
typedef signed   char  INT8S;
typedef unsigned int   INT16U;           // (1)
typedef signed   int   INT16S;
typedef unsigned long  INT32U;
typedef signed   long  INT32S;
typedef float          FP32;             // (2)
typedef double         FP64;

typedef unsigned char  OS_STK;           // (3)
typedef unsigned char  OS_CPU_SR;        // (4)
```

L3-2(1)	If you were to consult the IAR compiler documentation, you would find that an `int` is 16 bits and an `long` is 32 bits for the AVR. Most AVR compilers should have the same definitions.
L3-2(2)	Floating-point data types are included even though μC/OS-II doesn't make use of floating-point numbers.
L3-2(3)	A stack entry for the AVR processor is always 8 bits wide; thus, `OS_STK` is declared accordingly. All task stacks must be declared using `OS_STK` as its data type.
L3-2(4)	The status register (the `SREG`) on the AVR processor is 8 bits wide. The `OS_CPU_SR` data type is used when `OS_CRITICAL_METHOD` #3 is used (described below). In fact, this port only supports `OS_CRITICAL_METHOD` #3 because it's the preferred method for μC/OS-II ports.

3.01.03 OS_CPU.H, Critical Sections

μC/OS-II, as with all real-time kernels, needs to disable interrupts in order to access critical sections of code and re-enable interrupts when done. μC/OS-II defines two macros to disable and enable interrupts: `OS_ENTER_CRITICAL()` and `OS_EXIT_CRITICAL()`, respectively. μC/OS-II defines three ways to disable interrupts but, you only need to use one of the three methods for disabling and enabling interrupts. The book (MicroC/OS-II, The Real-Time Kernel) describes the three different methods. The one to choose depends on the processor and compiler. In most cases, the prefered method is `OS_CRITICAL_METHOD` #3.

`OS_CRITICAL_METHOD` #3 implements `OS_ENTER_CRITICAL()` by writing a function that will save the status register of the CPU in a variable. `OS_EXIT_CRITICAL()` invokes another function to restore the status register from the variable. In the book, Mr. Labrosse recommends that you call the functions expected in `OS_ENTER_CRITICAL()` and `OS_EXIT_CRITICAL()`: `OS_CPU_SR_Save()` and `OS_CPU_SR_Restore()`, respectively. The code for these two functions is declared in `OS_CPU_A.S` (described later).

Listing 3-3, OS_CPU.H, `OS_ENTER_CRITICAL()` and `OS_EXIT_CRITICAL()`

```
#define  OS_CRITICAL_METHOD    3


#if      OS_CRITICAL_METHOD == 3
#define  OS_ENTER_CRITICAL()  (cpu_sr = OS_CPU_SR_Save();)
#define  OS_EXIT_CRITICAL()   (OS_CPU_SR_Restore(cpu_sr);)
#endif
```

3.01.04 OS_CPU.H, Stack growth

The stacks on the AVR grows from high memory to low memory and thus, `OS_STK_GROWTH` is set to `1` to indicate this to µC/OS-II.

Listing 3-4, OS_CPU.H, Stack Growth

```
#define  OS_STK_GROWTH        1
```

3.01.05 OS_CPU.H, Task Level Context Switch

Task level context switches are performed when µC/OS-II invokes the macro `OS_TASK_SW()`. Because context switching is processor specific, `OS_TASK_SW()` needs to execute an assembly language function. In this case, `OSCtxSw()` which is declared in `OS_CPU_A.S` (described later).

Listing 3-5, OS_CPU.H, Task Level Context Switch

```
#define  OS_TASK_SW()         OSCtxSw()
```

3.01.06 OS_CPU.H, Global Variables

The AVR contains a hardware stack which is used to save the return address of functions during function calls as well as the return address of an interrupted function. Most AVR compilers also implement a 'software stack' which is used to pass arguments to functions. The software stack is maintained by the compiler. More on this will be explained later but, in order to create tasks, we need a way to tell µC/OS-II about the size of each of these two stacks. This is done by the global variables shown in listing 3-6.

Listing 3-6, OS_CPU.H, Global Variables used when creating a task

```
#if OS_CRITICAL_METHOD == 3
OS_CPU_EXT  INT16U  OSTaskStkSize;
OS_CPU_EXT  INT16U  OSTaskStkSizeHard;
#endif
```

3.02.07 OS_CPU.H, Function Prototypes

The prototypes in Listing 3-7 are for the functions used to disable and re-enable interrupts using `OS_CRITICAL_METHOD` #3 and are described later.

Listing 3-7, OS_CPU.H, Function Prototypes

```
#if OS_CRITICAL_METHOD == 3
OS_CPU_SR  OS_CPU_SR_Save(void);
void       OS_CPU_SR_Restore(OS_CPU_SR  cpu_sr);
#endif

void       OSStartHighRdy(void);
void       OSCtxSw(void);
void       OSIntCtxSw(void);
```

Also, as of V2.77, the prototypes for `OSCtxSw()`, `OSIntCtxSw()` and `OSStartHighRdy()` need to be placed in `OS_CPU.H`. In fact, it makes sense to do this since these are all port specific functions.

3.02 OS_CPU_C.C

A μC/OS-II port requires that you write ten fairly simple C functions:

```
OSInitHookBegin()
OSInitHookEnd()
OSTaskCreateHook()
OSTaskDelHook()
OSTaskIdleHook()
OSTaskStatHook()
OSTaskStkInit()
OSTaskSwHook()
OSTCBInitHook()
OSTimeTickHook()
```

Typically, μC/OS-II only requires `OSTaskStkInit()`. The other functions allow you to extend the functionality of the OS with your own functions. The functions that are highlighted will be discussed in this section.

Note that you will also need to set the `#define` constant `OS_CPU_HOOKS_EN` to `1` in `OS_CFG.H` in order for the compiler to use the functions declared in this file.

3.02.01 OS_CPU_C.C, OSTaskCreateHook()

This function is called by µC/OS-II's `OSTaskCreate()` or `OSTaskCreateExt()` when a task is created. `OSTaskCreateHook()` gives the opportunity to add code specific to the port when a task is created. In our case, we call the initialization function of µC/OS-View (an optional module available for µC/OS-II which performs task profiling at run-time, See www.micrium.com for details).

Note that for `OSView_TaskCreateHook()` to be called, the target resident code for µC/OS-View must be included as part of your build. In this case, you need to add a `#define OS_VIEW_MODULE 1` in `OS_CFG.H` of your application.

Note that if `OS_VIEW_MODULE` is `0`, we simply tell the compiler that `ptcb` is not actually used (i.e. `(void)ptcb`)) and thus avoid a compiler warning.

Listing 3-8, OS_CPU_C.C, OSTaskCreateHook()

```
void OSTaskCreateHook (OS_TCB *ptcb)
{
#if OS_VIEW_MODULE > 0
    OSView_TaskCreateHook(ptcb);
#else
    (void)ptcb;
#endif
}
```

3.02.02 OS_CPU_C.C, OSTaskStkInit()

Recall that a task is declared as shown in listing 3-9. For the IAR and ICC compilers, a pointer argument passed to a function is placed in the `R17:R16` registers.

Listing 3-9, µC/OS-II Task

```
void  MyTask (void *p_arg)
{
    /* Do something with 'p_arg', optional */
    while (1) {
        /* Task body */
    }
}
```

The code in Listing 3-10 initializes the stack frame for the task being created. The task received an optional argument '`p_arg`'. That's why '`p_arg`' is passed in `R17:R16` when the task is created. The initial value of most of the CPU registers is not important so, we decided to initialize them to values corresponding to their register number. This makes it convenient when debugging and examining stacks in RAM. The initial values are thus useful when the task is first created but, of course, the register values will most likely change as the task code is executed.

Figure 3-1 shows how the stack frame is initialized for each task when it's created. The AVR only provides a *hardware stack* to save/restore subroutine and interrupt return addresses. Most C compiler pass function arguments onto a stack so, the compiler manufacturer decided to simulate a stack for passing arguments. This is called a *software stack*.

When a task is created, you need to initialize the two global variables `OSTaskStkSize` and `OSTaskStkSizeHard` as shown below:

```
OSTaskStkSize     = 256;
OSTaskStkSizeHard =  64;
OSTaskCreateExt(. . . .);
```

IMPORTANT

1) You MUST initialize `OSTaskStkSize` and `OSTaskStkSizeHard` **BEFORE** calling `OSInit()` to the size of the Idle Task and the Statistic Task (if you set `OS_TASK_STAT_EN` to `1` in `OS_CFG.H`), which ever is the LARGEST.

2) You can set the task stacks to be different for each task by reloading `OSTaskStkSize` and `OSTaskStkSizeHard` before calling `OSTaskCreate()` or `OSTaskCreateExt()`.

3) You **MUST** make sure that you will not be interrupted when you create your tasks in case these variables are changed by other tasks or ISRs.

Listing 3-10, OS_CPU_C.C, `OSTaskStkInit()`

```
OS_STK  *OSTaskStkInit (void  (*task)(void *pd),
                        void   *p_arg,
                        OS_STK *ptos,
                        INT16U  opt)
{
    INT8U  *psoft_stk;
    INT8U  *phard_stk;                   /*     Used for AVR hardware stack          */
    INT16U  tmp;


    (void)opt;                           /*     'opt' is not used, prevent warning   */
    psoft_stk     = (INT8U *)ptos;
    phard_stk     = (INT8U *)ptos        /* (1)
                  - OSTaskStkSize        /*     Task stack size                      */
                  + OSTaskStkSizeHard;   /*     AVR return stack ("hardware stack")  */

#if IAR
    tmp           = (INT16U)task;        /* (2) Get Task address                     */
#endif

#if ICC
    tmp           = *(INT16U const *)task);
#endif

    *phard_stk--  = (INT8U)(tmp & 0xFF); /* (3) Put task address on "hardware stack" */
    tmp         >>= 8;
    *phard_stk--  = (INT8U)(tmp & 0xFF);

    *psoft_stk--  = (INT8U)0x00;         /* (4) R0    = 0x00                         */
    *psoft_stk--  = (INT8U)0x01;         /*     R1    = 0x01                         */
    *psoft_stk--  = (INT8U)0x02;         /*     R2    = 0x02                         */
    *psoft_stk--  = (INT8U)0x03;         /*     R3    = 0x03                         */
    *psoft_stk--  = (INT8U)0x04;         /*     R4    = 0x04                         */
    *psoft_stk--  = (INT8U)0x05;         /*     R5    = 0x05                         */
    *psoft_stk--  = (INT8U)0x06;         /*     R6    = 0x06                         */
    *psoft_stk--  = (INT8U)0x07;         /*     R7    = 0x07                         */
    *psoft_stk--  = (INT8U)0x08;         /*     R8    = 0x08                         */
    *psoft_stk--  = (INT8U)0x09;         /*     R9    = 0x09                         */
    *psoft_stk--  = (INT8U)0x10;         /*     R10   = 0x10                         */
    *psoft_stk--  = (INT8U)0x11;         /*     R11   = 0x11                         */
    *psoft_stk--  = (INT8U)0x12;         /*     R12   = 0x12                         */
    *psoft_stk--  = (INT8U)0x13;         /*     R13   = 0x13                         */
    *psoft_stk--  = (INT8U)0x14;         /*     R14   = 0x14                         */
    *psoft_stk--  = (INT8U)0x15;         /*     R15   = 0x15                         */
    tmp           = (INT16U)p_arg;
    *psoft_stk--  = (INT8U)(tmp & 0xFF); /* (5) 'p_arg' passed in R17:R16            */
    tmp         >>= 8;
    *psoft_stk--  = (INT8U)(tmp & 0xFF);
    *psoft_stk--  = (INT8U)0x18;         /*     R18   = 0x18                         */
    *psoft_stk--  = (INT8U)0x19;         /*     R19   = 0x19                         */
    *psoft_stk--  = (INT8U)0x20;         /*     R20   = 0x20                         */
    *psoft_stk--  = (INT8U)0x21;         /*     R21   = 0x21                         */
    *psoft_stk--  = (INT8U)0x22;         /*     R22   = 0x22                         */
    *psoft_stk--  = (INT8U)0x23;         /*     R23   = 0x23                         */
    *psoft_stk--  = (INT8U)0x24;         /*     R24   = 0x24                         */
    *psoft_stk--  = (INT8U)0x25;         /*     R25   = 0x25                         */
    *psoft_stk--  = (INT8U)0x26;         /*     R26   = 0x26                         */
    *psoft_stk--  = (INT8U)0x27;         /*     R27   = 0x27                         */
                                         /* (6) R28     R29:R28 (software stack)     */
                                         /*     R29                                  */
    *psoft_stk--  = (INT8U)0x30;         /*     R30   = 0x30                         */
    *psoft_stk--  = (INT8U)0x31;         /*     R31   = 0x31                         */
    *psoft_stk--  = (INT8U)0xAA;         /* (7) RAMPZ = 0xAA                         */
    *psoft_stk--  = (INT8U)0x80;         /* (8) SREG  = Interrupts enabled           */
    tmp           = (INT16U)phard_stk;
    *psoft_stk--  = (INT8U)(tmp >> 8);   /* (9) SPH                                  */
    *psoft_stk    = (INT8U) tmp;         /*     SPL                                  */
    return ((OS_STK *)psoft_stk);        /* (10)                                      */
}
```

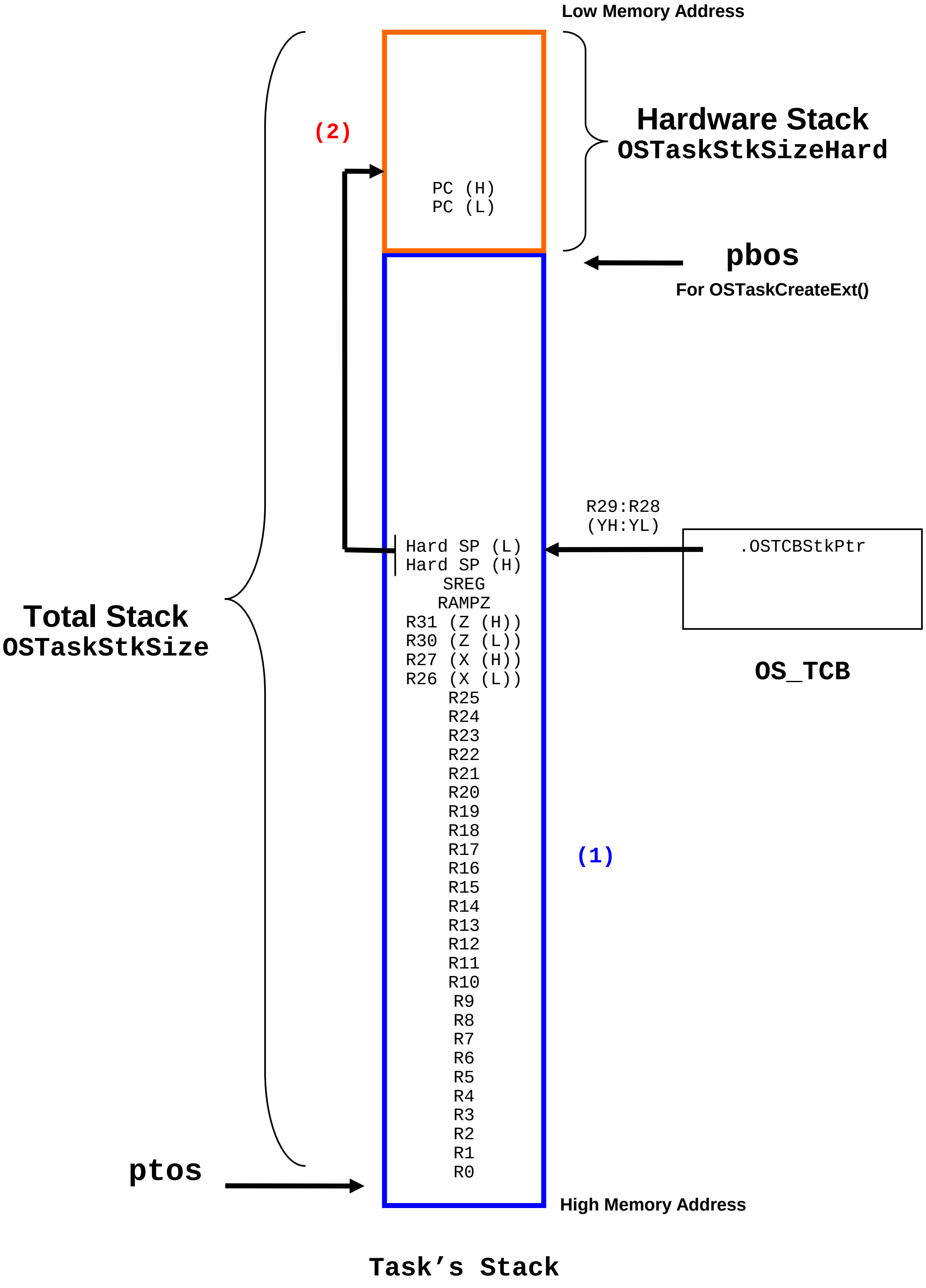


Figure 3-1, Atmel AVR’s dual stack

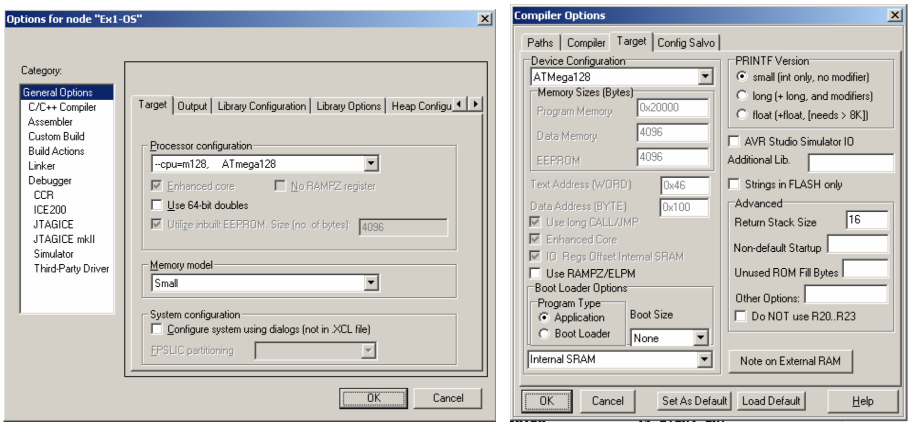
L3-10(1) We calculate the starting address of the hardware stack from the total stack size and the desired size of the hardware stack.

L3-10(2) We obtain the address of ‘task’ and we save it as a 16-bit value since function pointers on an ATmega128 are only 16 bit wide. You should note that for the ICC compiler, we need to do an extra level of indirection. This is a requirement of the ICC compiler.

L3-10(3) We store the address of the task onto the hardware stack. Note that we assume a 16 bit address (64K words, 128K bytes). It’s also important to note two things: first, the address of the task is pushed low byte first and second, that the hardware stack pointer points to the **NEXT empty** location on the stack.

On the IAR tools, we selected the following configuration (LEFT screen shot):
“—cpu=m128, ATmega128”

On the ICC tools, we selected the following configuration (RIGHT screen shot):
“ATmega128”



Processor configuration: --cpu=m128, ATmega128
☑ Enhanced core ☐ No RAMPZ register
☐ Use 64-bit doubles
☑ Utilize inbuilt EEPROM. Size (no. of bytes): 4096
Memory model: Small
System configuration
☐ Configure system using dialogs (not in .XCL file)
FPSLIC partitioning

Device Configuration: ATMega128
Memory Sizes (Bytes)
Program Memory: 0x20000
Data Memory: 4096
EEPROM: 4096
Text Address (WORD): 0x46
Data Address (BYTE): 0x100
☑ Use long CALL/JMP
☑ Enhanced Core
☑ IO Regs Offset Internal SRAM
☐ Use RAMPZ/ELPM
Boot Loader Options
Program Type: ● Application ○ Boot Loader
Boot Size: None
Internal SRAM
PRINTF Version
● small (int only, no modifier)
○ long (+ long, and modifiers)
○ float (+float, [needs > 8K])
☐ AVR Studio Simulator IO
Additional Lib.
☐ Strings in FLASH only
Advanced
Return Stack Size: 16
Non-default Startup
Unused ROM Fill Bytes
Other Options:
☐ Do NOT use R20..R23

Figure 3-2, Atmel AVR Compiler Options
(IAR on Left, **ICC** on Right**)**

L3-10(4) We initialize the register values on the task’s stack. Note that we set the values corresponding to the register number. This is useful during debug since it allows us to locate the stack frame in memory.

L3-10(5) As mentioned previously, `p_arg` is passed in `R17:R16` so we initialize those registers accordingly – `R17` with the high byte and `R16` with the low byte. Note that with the memory model selected, data pointers are 16 bit wide.

L3-10(6) `R29:R28` contains the address of the software stack. Because of this, `R29:R28` are not actually initialized here.

L3-10(7) We initialize the `RAMPZ`. If the software actually uses `RAMPZ` then the compiler will load the proper value in this register at run-time.

L3-10(8) We initialize the `SREG` with `0x80` to enable interrupts when the task is started.

L3-10(9) We save the contents of the hardware stack pointer onto the software stack. Note that we place the high byte of the stack pointer onto the stack first.

L3-10(10) We return the address of the new top of stack to the caller. This address will actually be saved in the task control block of the task being created. It's important to note that the pointer points to the **LAST byte placed** onto the stack.

3.02.03 OS_CPU_C.C, OSTaskSwHook()

`OSTaskSwHook()` is called when a context switch occurs. This function allows the port code to be extended and do things such as measuring the execution time of a task, output a pulse on a port pin when a contact switch occurs, etc. In this case, we call the μC/OS-View task switch hook called `OSView_TaskSwHook()`. This assumes that you have μC/OS-View as part of your build and that you set `OS_VIEW_MODULE` to `1` in `OS_CFG.H`.

Listing 3-11, OS_CPU_C.C, OSIntCtxSw()

```
void  OSCtxSwHook (void)
{
#if OS_VIEW_MODULE > 0
    OSView_TaskSwHook();
#endif
}
```

3.02.04 OS_CPU_C.C, OSTimeTickHook()

`OSTimeTickHook()` is called at the very beginning of `OSTimeTick()`. This function allows the port code to be extended and, in our case, we call the μC/OS-View function `OSView_TickHook()`. Again, this assumes that you have μC/OS-View as part of your build and that you set `OS_VIEW_MODULE` to `1` in `OS_CFG.H`.

Listing 3-12, OS_CPU_C.C, OSTimeTickHook()

```
void  OSTimeTickHook (void)
{
#if OS_VIEW_MODULE > 0
    OSView_TickHook();
#endif
}
```

3.03 OS_CPU_A.S90 (IAR), OS_CPU_A.S (ICC)

A µC/OS-II port requires that you write five fairly simple assembly language functions. These functions are needed because you normally cannot save/restore registers from C functions.

```
OS_CPU_SR_Save()
OS_CPU_SR_Restore()
OSStartHighRdy()
OSCtxSw()
OSIntCtxSw()
```

3.03.01 OS_CPU_SR_Save()

The code in listing 3-13 implements the saving of the `SREG` register and then disabling interrupts for `OS_CRITICAL_METHOD` #3.

When this function returns, `R16` contains the state of the `SREG` register prior to disabling interrupts. This will allow us to restore the `SREG` to its original state when we exit the critical section.

Listing 3-13, `OS_CPU_SR_Save()`

```
OS_CPU_SR_Save:   IN  R16,SREG   ; Get current state of interrupts disable flag
                  CLI            ; Disable interrupts
                  RET            ; Return original SREG value in R16
```

3.03.02 OS_CPU_SR_Restore()

The code in the listing below implements the function to restore the `SREG` register for `OS_CRITICAL_METHOD` #3. When called, it's assumed that `R16` contains the desired state of the `SREG` register.

Listing 3-14, `OS_CPU_SR_Restore()`

```
OS_CPU_SR_Restore: OUT SREG,R16   ; Restore SREG
                   RET            ; Return
```

3.03.03 OSStartHighRdy()

`OSStartHighRdy()` is called by `OSStart()` to start running the highest priority task that was created before calling `OSStart()`. `OSStart()` sets `OSTCBHighRdy` to point to the `OS_TCB` of the highest priority task.

Listing 3-15, OSStartHighRdy()

```
OSStartHighRdy:
    CALL    OSTaskSwHook        ; (1) Invoke user defined context switch hook

    LDS     R16,OSRunning       ; (2) Indicate that we are multitasking
    INC     R16
    STS     OSRunning,R16

    LDS     R30,OSTCBHighRdy    ; (3) Let Z point to TCB of highest priority task
    LDS     R31,OSTCBHighRdy+1  ;     ready to run

    LD      R28,Z+              ;     Load Y (R29:R28) pointer – The software SP
    LD      R29,Z+

    POP_SP                      ; (4) Restore stack pointer
    POP_SREG_INT                ; (5) Restore status register (DISABLE interrupts)
    POP_ALL                     ; (6) Restore all registers

    RETI                        ; (7)
```

L3-15(1) Before starting the highest priority task, we call `OSTaskSwHook()` in case a hook call has been declared.

L3-15(2) The μC/OS-II flag `OSRunning` is set to `TRUE` indicating that μC/OS-II will be running once the first task is started.

L3-15(3) We then get the pointer to the task's top-of-stack (was stored by `OSTaskCreate()` or `OSTaskCreateExt()`). This is the *software stack pointer*.

L3-15(4) We then pop the *hardware stack pointer* from the software stack. Note that this is done with the `POP_SP` macro which is declared in `OS_CPU_I.S90` for IAR, `OS_CPU_I.H` for ICC:

```
POP_SP          MACRO
                LD      R16,Y+
                OUT     SPL,R16
                LD      R16,Y+
                OUT     SPH,R16
                ENDM
```

L3-15(5) We now restore the contents of the `SREG` but, we clear bit #7 BEFORE we restore the actual `SREG` because we don't want to be interrupted while we are restoring the context of the first task. The reason we do this will become clear when we discuss L3-15(7). Restoring the `SREG` is done with the `POP_SREG_INT` macro as follows (also declared in `OS_CPU_I.S90` for IAR, `OS_CPU_I.H` for ICC):

```
POP_SREG_INT    MACRO
                LD      R16,Y+
                CBR     R16,BIT07
                OUT     SREG,R16
```

```
ENDM
```

L3-15(6) We now restore all the other registers that make up the context of the CPU. This is done by the `POP_ALL` macro as shown below (declared in `OS_CPU_I.S90` for IAR, `OS_CPU_I.H` for ICC).

Note that the RAMPZ registers is not an actual CPU registers since it's are found in the I/O space of the AVR. However, it is part of the AVR's context.

```
POP_ALL         MACRO
                LD      R16,Y+
                OUT     RAMPZ,R16
                LD      R31,Y+
                LD      R30,Y+
                LD      R27,Y+
                LD      R26,Y+
                LD      R25,Y+
                LD      R24,Y+
                LD      R23,Y+
                LD      R22,Y+
                LD      R21,Y+
                LD      R20,Y+
                LD      R19,Y+
                LD      R18,Y+
                LD      R17,Y+
                LD      R16,Y+
                LD      R15,Y+
                LD      R14,Y+
                LD      R13,Y+
                LD      R12,Y+
                LD      R11,Y+
                LD      R10,Y+
                LD      R9,Y+
                LD      R8,Y+
                LD      R7,Y+
                LD      R6,Y+
                LD      R5,Y+
                LD      R4,Y+
                LD      R3,Y+
                LD      R2,Y+
                LD      R1,Y+
                LD      R0,Y+
                ENDM
```

L3-15(7) Finally, we execute the `RETI` instruction that POPs the return address from the hardware stack *AND* sets bit #7 of the `SREG` enabling interrupts for the task. At this point, the CPU is executing the code of the most important task you created prior to calling `OSStart()`. Note that if we had used the `RET` instruction, the interrupt enable bit would not have been set.

3.03.04 OSCtxSw()

The code to perform a 'task level' context switch is shown below in pseudo-code. `OSCtxSw()` is called when a higher priority task is made ready to run by another task or, when the current task can no longer execute (e.g. it calls `OSTimeDly()`, `OSSemPend()` and the semaphore is not available, etc.).

```
Save the CPU registers onto the old task's stack;
OSPrioCur              = OSPrioHighRdy;
OSTCBCur->OSTCBStkPtr = SP;
OSTaskSwHook();
SP                     = OSTCBHighRdy->OSTCBStkPtr;
OSTCBCur               = OSTCBHighRdy;
Restore the CPU registers from the new task's stack;
```

The actual code for the task level context switch is shown in Listing 3-16.

Listing 3-16, `OSCtxSw()`

```
OSCtxSw:
    PUSH_ALL                    ;  (1) Save current task's context
    PUSH_SREG
    PUSH_SP

    LDS     R30,OSTCBCur        ;  (2) Z = OSTCBCur->OSTCBStkPtr
    LDS     R31,OSTCBCur+1
    ST      Z+,R28              ;      Save Y (R29:R28) pointer
    ST      Z+,R29

    CALL    OSTaskSwHook        ;  (3) Call user defined task switch hook

    LDS     R16,OSPrioHighRdy   ;  (4) OSPrioCur = OSPrioHighRdy
    STS     OSPrioCur,R16

    LDS     R30,OSTCBHighRdy    ;  (5) Let Z point to TCB of highest priority task
    LDS     R31,OSTCBHighRdy+1  ;      ready to run
    STS     OSTCBCur,R30        ;      OSTCBCur = OSTCBHighRdy
    STS     OSTCBCur+1,R31

    LD      R28,Z+              ;  (6) Restore Y pointer
    LD      R29,Z+

    POP_SP                      ;  (7) Restore stack pointer
    LD      R16,Y+              ;  (8) Restore status register
    SBRC    R16,7               ;      Skip next instruction in interrupts DISABLED
    RJMP    OSCtxSw_1

    OUT     SREG,R16            ;  (9) Interrupts of task to return to are DISABLED
    POP_ALL                     ; (10)
    RET                         ; (11)

OSCtxSw_1:
    CBR     R16,BIT07           ; (12) Interrupts of task to return to are ENABLED
    OUT     SREG,R16
    POP_ALL                     ; (13) Restore all registers
    RETI                        ; (14) Return from interrupt and enable interrupts
```

L3-16(1) We save the context of the task being switched out. We use the following three macros to accomplish this:

```
PUSH_ALL        MACRO
                ST      -Y,R0
                ST      -Y,R1
                ST      -Y,R2
                ST      -Y,R3
                ST      -Y,R4
                ST      -Y,R5
                ST      -Y,R6
                ST      -Y,R7
                ST      -Y,R8
                ST      -Y,R9
                ST      -Y,R10
                ST      -Y,R11
                ST      -Y,R12
                ST      -Y,R13
                ST      -Y,R14
                ST      -Y,R15
                ST      -Y,R16
                ST      -Y,R17
                ST      -Y,R18
                ST      -Y,R19
                ST      -Y,R20
                ST      -Y,R21
                ST      -Y,R22
                ST      -Y,R23
                ST      -Y,R24
                ST      -Y,R25
                ST      -Y,R26
                ST      -Y,R27
                ST      -Y,R30
                ST      -Y,R31
                IN      R16,RAMPZ
                ST      -Y,R16
                ENDM


PUSH_SREG       MACRO
                IN      R16,SREG
                ST      -Y,R16
                ENDM


PUSH_SP         MACRO
                IN      R16,SPH
                ST      -Y,R16
                IN      R16,SPL
                ST      -Y,R16
                ENDM
```

You should note that μC/OS-II ALWAYS calls `OSCtxSw()` with interrupts disabled and thus, bit 7 of `SREG` is always `0`.

L3-16(2) We save the current value of the software stack pointer into the TCB of the task being switched out.

L3-16(3) We call the user define `OSTaskSwHook()`. This call is not necessary if your application doesn't use `OSTaskSwHook()`.

L3-16(4) We set `OSPrioCur` to the value of `OSPrioHighRdy` (an 8 bit variable).

L3-16(5) We now make the current TCB the TCB of the task being switched in.

L3-16(6) We load the software stack pointer of the new task. This was stored in the TCB of the task.

L3-16(7) We restore the hardware stack pointer of the new task from the software stack.

L3-16(8) Here, we restore the value of the `SREG` from the task stack but, we don't actually change the `SREG` just yet. In fact, we examine bit #7 of the restore register to see if interrupts need to be enable when the task returns or not. The reason we do this is because if the context of the task was saved using `OSCtxSw()` then we must return to the caller of `OSCtxSw()` with interrupts disabled. However, if the context of the task was saved because the task was interrupted then, we need to return to the task with interrupts enabled.

L3-16(9) Bit 7 of `SREG` was cleared and thus, we need to return to the caller of `OSCtxSw()` with interrupts disabled. We thus simply copy the value retrieved from the stack into `SREG`.

L3-16(10) We now retrieve the remaining context of the task to switch in using the `POP_ALL` macro which we already saw.

L3-16(11) We return to the caller using the `RET` instruction which does not touch bit 7 of the `SREG` register. At this point, we return to the function that called `OSCtxSw()`.

L3-16(12) If bit 7 of the `SREG` register of the task to restore was set (the task was interrupted and we context switched to a higher priority task) then we need to clear it before saving it into the actual `SREG` because we will be using the `RETI` instruction to set the bit upon returning to the interrupted task.

L3-16(13) We now retrieve the remaining context of the task to switch in using the `POP_ALL` macro which we already saw.

L3-16(14) We return to the caller using the `RETI` instruction which sets bit 7 of the `SREG` register enable interrupts. At this point, the interrupted task is resumed.

3.03.05 OSIntCtxSw()

When an ISR completes, `OSIntExit()` is called to determine whether a more important task than the interrupted task needs to execute. If that's the case, `OSIntExit()` determines which task to run next and calls `OSIntCtxSw()` to perform the actual context switch to that task. You will notice that `OSIntCtxSw()` is identical to the second half of `OSCtxSw()`. The reason we have these as two separate functions is to simplify debugging. Specifically, if you wanted to set a breakpoint in `OSIntCtxSw()`, you would hit the breakpoint during a task level context switch (if `OSIntCtxSw()` was just a label in `OSCtxSw()`). Of course this would make debugging a bit difficult.

Listing 3-17, `OSIntCtxSw()`

```
OSIntCtxSw:
    CALL    OSTaskSwHook             ; Call user defined task switch hook

    LDS     R16,OSPrioHighRdy        ; OSPrioCur = OSPrioHighRdy
    STS     OSPrioCur,R16

    LDS     R30,OSTCBHighRdy         ; Let Z point to TCB of highest priority task
    LDS     R31,OSTCBHighRdy+1       ;     ready to run
    STS     OSTCBCur,R30             ; OSTCBCur = OSTCBHighRdy
    STS     OSTCBCur+1,R31

    LD      R28,Z+                   ; Restore Y pointer
    LD      R29,Z+

    POP_SP                           ; Restore stack pointer
    LD      R16,Y+                   ; Restore status register
    SBRC    R16,7                    ; Skip next instruction in interrupts DISABLED
    RJMP    OSIntCtxSw_1

    OUT     SREG,R16                 ; Interrupts of task to return to are DISABLED
    POP_ALL
    RET

OSIntCtxSw_1:
    CBR     R16,BIT07                ; Interrupts of task to return to are ENABLED
    OUT     SREG,R16
    POP_ALL                          ; Restore all registers
    RETI
```

3.04 OS_CPU_I.S90 (IAR), OS_CPU_I.H (ICC)

`OS_CPU_I.S90` (`OS_CPU_I.H` for ICC) is a file that should be included with **ALL** your assembly language file that declare interrupt service routines. The file contains macros and I/O port definitions for the ATmega128. This file avoids having to duplicate the macros in multiple files.

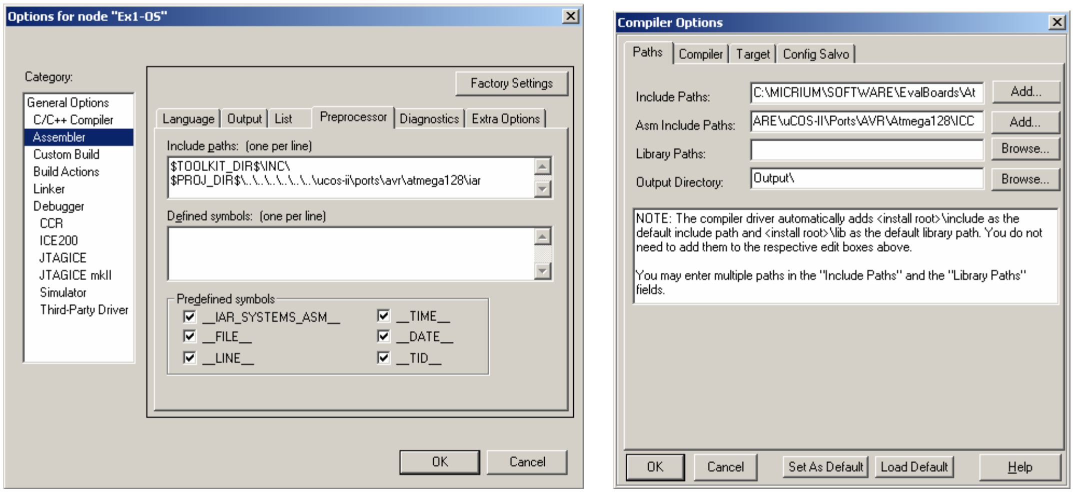
With the **IAR** tools, you simply include `OS_CPU_I.S90` as follows:

```
#include  <os_cpu_i.s90>
```

With the **ICC** tools, you simply include `OS_CPU_I.H` as follows:

```
.include  “os_cpu_i.h”
```

It’s **IMPORTANT** that you make sure that the path to the µC/OS-II port for the AVR is included in the search path of the assembler, as shown in figure 3-3.



Include paths: (one per line)

```
$TOOLKIT_DIR$\INC\
$PROJ_DIR$\..\..\..\..\..\..\ucos-ii\ports\avr\atmega128\iar
```

Defined symbols: (one per line)

Predefined symbols
☑ __IAR_SYSTEMS_ASM__ ☑ __TIME__
☑ __FILE__ ☑ __DATE__
☑ __LINE__ ☑ __TID__

Include Paths: C:\MICRIUM\SOFTWARE\EvalBoards\At
Asm Include Paths: ARE\uCOS-II\Ports\AVR\Atmega128\ICC
Library Paths:
Output Directory: Output\

NOTE: The compiler driver automatically adds <install root>\include as the default include path and <install root>\lib as the default library path. You do not need to add them to the respective edit boxes above.

You may enter multiple paths in the "Include Paths" and the "Library Paths" fields.

Figure 3-3, Assembler Search Path
(IAR on Left, **ICC** on Right**)**

3.05 OS_DBG.C

`OS_DBG.C` is a file that has been added in V2.62 to provide Kernel Aware debugger to extract information about µC/OS-II and its configuration. Specifically, `OS_DBG.C` contains a number of constants that are placed in ROM (code space) which the debugger can read and display. Because you may not be using a debugger that needs that file, you may omit it in your build.

For the **IAR** compiler, Micriµm has introduced a Windows-based ‘Plug-In’ module that makes use of this file and thus needs to be included if you use IAR’s C-Spy. This plug-in is available from Micriµm at www.Micrium.com.

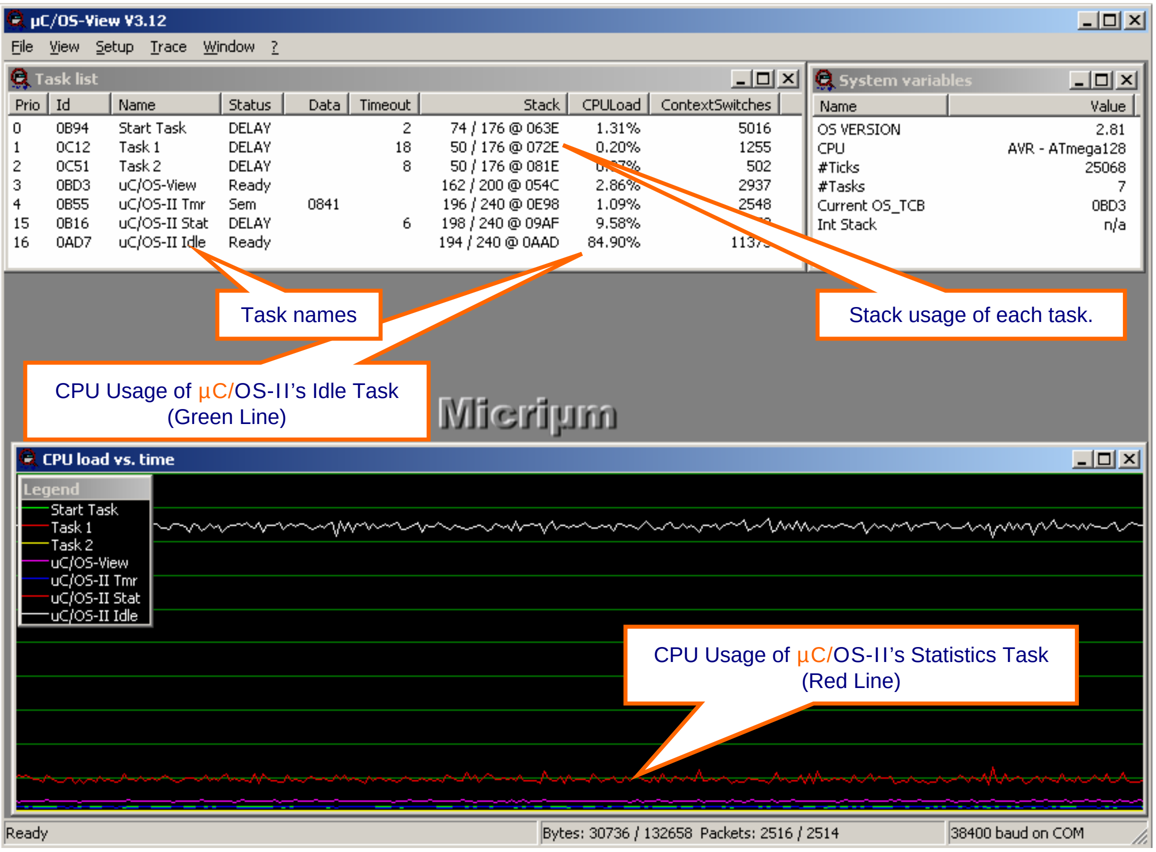
For the **ICC** compiler, Micriµm has introduced a Windows-based ‘Plug-In’ module that makes use of this file and thus needs to be included if you use Atmel’s AVRstudio 4. This plug-in is included with Atmel’s AVRstudio4 as part of their RTOS development kit.

4.00 µC/OS-View Port

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

µC/OS-View is an optional module you can purchase from Micriμm. If you did not purchase this module you can simply disable code compilation by defining `OS_VIEW_MODULE` to `0` in `OS_CFG.H` and also remove the `OS_VIEW*.*` from the project build files.

Figure 4-1 shows a screen shot of the µC/OS-View 'viewer' (the Windows application). The viewer communicates with the target to collect and display this run-time information. Especially useful if the display of the CPU usage (in percentage) for each task as well as worst case stack growth (for the software stack). You should note that the stack usage for the Idle and Statistics task is **not** reported correctly because of the way these tasks are created in `OS_CORE.C`. In other words, the idle stack should actually be using about 50 or so bytes off the software stack while the statistics task should be using about 100 or so bytes.



Prio	Id	Name	Status	Data	Timeout	Stack	CPULoad	ContextSwitches
0	0B94	Start Task	DELAY		2	74 / 176 @ 063E	1.31%	5016
1	0C12	Task 1	DELAY		18	50 / 176 @ 072E	0.20%	1255
2	0C51	Task 2	DELAY		8	50 / 176 @ 081E	[illegible]	502
3	0BD3	uC/OS-View	Ready			162 / 200 @ 054C	2.86%	2937
4	0B55	uC/OS-II Tmr	Sem	0841		196 / 240 @ 0E98	1.09%	2548
15	0B16	uC/OS-II Stat	DELAY		6	198 / 240 @ 09AF	9.58%	[illegible]
16	0AD7	uC/OS-II Idle	Ready			194 / 240 @ 0AAD	84.90%	[illegible]

Name	Value
OS VERSION	2.81
CPU	AVR - ATmega128
#Ticks	25068
#Tasks	7
Current OS_TCB	0BD3
Int Stack	n/a

Ready
Bytes: 30736 / 132658 Packets: 2516 / 2514
38400 baud on COM

Figure 4-1, µC/OS-View's 'Viewer' Screen Shot

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

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

μC/OS-View also allows you to:

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

4.01 μC/OS-View Port Files

The μC/OS-View port files are found in the following three files:

IAR's EWAVR:
`os_viewc.h`
`os_viewc.c`
`os_viewa.s90`

ImageCraft's ICCAVR:
`os_viewc.h`
`os_viewc.c`
`os_viewa.s`

4.02 OS_VIEWc.h

The μC/OS-View port header file configures certain aspects of the port as described below.

OS_VIEW_TMR_32_BITS

This `#define` constant indicates whether the timer used for measuring execution time is a 16-bit timer (when 0) or 32-bit timer (when 1). On the AVR, all timers are 16 bits and thus this value is 0.

OS_VIEW_RX_BUF_SIZE

This `#define` constant indicates the size of the receive buffer for commands sent by the Windows application. The default value is `20` and, you should not change the value of this `#define`.

OS_VIEW_TX_BUF_SIZE

This `#define` constant indicates the size of the transmit buffer for responses from the target system. If your application is tight on RAM, you can reduce this value. However, doing this will limit the number of tasks that can be reported back from the target. The default value is `255` and you should not change the value of this `#define`.

OS_VIEW_TX_STR_SIZE

This `#define` constant determines the largest string that can be sent to the Windows' 'Terminal Window'. If your application is tight with RAM, you could reduce the value of this `#define`. The default is `80`.

4.03 OS_VIEWc.c

This file contains the code that adapts μC/OS-View to the AVR. μC/OS-View requires the presence of one UART as well as a free-running timer.

This port was designed such that you can select any of the four UARTs (`UART0` or `UART1`) of the ATmega128 for use with μC/OS-View. The UART used is selected via a `#define` which you need to set in your application configuration. For the example code, we created a file called `app_cfg.h` for that and other purposes. In our tests, we used `UART0`.

```
#define   OS_VIEW_COMM_0                   0
#define   OS_VIEW_COMM_1                   1

#define   OS_VIEW_COMM_SEL       OS_VIEW_COMM_0
```

Similarly, you can use either Timer #1 or Timer #3 as the timer used by μC/OS-View. For the example code, we decided to use Timer #1 and this is specified in `app_cfg.h` as follows. Note that we assume that Timer #0 is used for the μC/OS-II tick rate.

```
#define  OS_VIEW_TMR_1                      1
#define  OS_VIEW_TMR_3                      3

#define  OS_VIEW_TMR_SEL        OS_VIEW_TMR_1
```

`OS_VIEWc.c` also declares a number of hardware specific functions to initialize the UART and timer and service interrupts from those devices.

4.04 OS_VIEWa.s90 (IAR) or OS_VIEW_a.s (ICC)

This file contains the assembly language code that adapts μC/OS-View to the AVR. Assembly language is necessary to properly handle the register saving/restoring order expected by μC/OS-II. However, we did the minimum amount of code as possible in assembly language and deferred most of the work to C functions.

You will notice that we use the same type of code as in `os_cpu_a.s90`.

5.00 Application code

The sample code is found in the following directories:

IAR's EWAVR:

`\Micrium\Software\EvalBoards\Atmel\STK500\ATmega128\`**`IAR`**`\Ex1-OS`
`\Micrium\Software\EvalBoards\Atmel\STK500\ATmega128\`**`IAR`**`\Ex1-OS-View`

In this directory, you will find the following files:

```
app.c
app_cfg.h
app_isr.s90
app_vect.s90
includes.h
os_cfg.h
```

ImageCraft's ICCAVR:

`\Micrium\Software\EvalBoards\Atmel\STK500\ATmega128\`**`ICC`**`\Ex1-OS`

In this directory, you will find the following files:

```
app.c
app_cfg.h
app_isr.s
app_vect.s
includes.h
os_cfg.h
```

5.01 app.c

`app.c` contains the application code for the example. The example code is designed to run on the STK500/501. Basically, the application consist of blinking three of the 8 LEDs on the STK500 board. Each of the LED is controlled by its own task.

As with most C programs, execution start in `main()` as shown in Listing 5-1.

Listing 5-1, main()

```
void  main (void)
{
#if (OS_TASK_NAME_SIZE > 14) && (OS_TASK_STAT_EN > 0)
    INT8U  err;
#endif


    /*---- Any initialization code prior to calling OSInit() goes HERE ----------*/

                        /* (1) Setup the Idle and Statistics Task sizes          */
    OSTaskStkSize     = OS_TASK_IDLE_STK_SIZE;        /* See app_cfg.h           */
    OSTaskStkSizeHard = OS_TASK_IDLE_STK_SIZE_HARD;

    OSInit();           /* (2) Initialize "uC/OS-II, The Real-Time Kernel"       */


    /*---- Any initialization code before starting multitasking -----------------*/


    OSTaskStkSize     = OS_TASK_START_STK_SIZE;        /* (3), see app_cfg.h     */
    OSTaskStkSizeHard = OS_TASK_START_STK_SIZE_HARD;
    OSTaskCreateExt(AppTaskStart,
                    (void *)0,
                    (OS_STK *)&AppTaskStartStk[OSTaskStkSize - 1],
                    OS_TASK_START_PRIO,
                    OS_TASK_START_PRIO,
                    (OS_STK *)&AppTaskStartStk[OSTaskStkSizeHard],
                    OSTaskStkSize - OSTaskStkSizeHard,
                    (void *)0,
                    OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR);

    /*---- Create any other task you want before we start multitasking ----------*/

    OSStart();         /* (4) Start multitasking (i.e. give control to uC/OS-II) */
}
```

L5-1(1) It's important that you initialize `OSTaskStkSize` and `OSTaskStkSizeHard` **BEFORE** calling `OSInit()` because these variable are necessary to establish the stack sizes for both the idle task (i.e. `OS_TaskIdle()`) and the statistic task (i.e. `OS_TaskStat()`). Because both of these tasks are create in `OSInit()`, you need to set `OSTaskStkSize` and `OSTaskStkSizeHard` to the highest requirements of these two tasks. In other words, if the statistic task requires `256` bytes of total stack space and `64` bytes of hardware stack space and the idle task only needs `128/32` respectively then you need to use `256` and `64`.

L5-1(2) As with all µC/OS-II based applications, you need to initialize the OS by calling `OSInit()`.

L5-1(3) You now need to create AT LEAST one application task before you startup µC/OS-II. In fact, you can create ALL your tasks here or, create one task that will create the other tasks. The later is actually the preferred method. It's important to load `OSTaskStkSize` and `OSTaskStkSizeHard` for **EACH** different task you create. In other words, each task can have different stack requirements.

L5-1(4) You can now start µC/OS-II and thus multitasking. If you created one task then µC/OS-II will start it.

In our case, the one task created in `main()` (i.e. `AppTaskStart()`) is the task that will run first. The code for `AppTaskStart()` is shown in listing 5-2.

Listing 5-2, AppTaskStart()

```
static  void  AppTaskStart (void *p_arg)
{
    INT8U  I;


    (void)p_arg;        /*     Prevent compiler warnings                          */

    BSP_Init();         /* (1) Initialize the Board Support Package               */

#if OS_TASK_STAT_EN > 0
    OSStatInit();       /* (2) Initialize uC/OS-II’s statistic task               */
#endif

#if OS_VIEW_MODULE > 0  /* (3) Initialize uC/OS-View                              */
    OSTaskStkSize     = OS_VIEW_TASK_STK_SIZE;
    OSTaskStkSizeHard = OS_VIEW_TASK_STK_SIZE_HARD;
    OSView_Init(38400);
    OSView_TerminalRxSetCallback(AppTerminalRx);
#endif

    AppTaskCreate();    /* (4) Create other application tasks                     */

    while (TRUE) {      /* (5) Task body, always written as an infinite loop.    */
        for (i = 1; i <= 8; i++) {
            LED_On(i);
            OSTimeDly(OS_TICKS_PER_SEC / 20);
            LED_Off(i);
        }
        for (i = 7; i > 1; i--) {
            LED_On(i);
            OSTimeDly(OS_TICKS_PER_SEC / 20);
            LED_Off(i);
        }
    }
}
```

L5-2(1) We start off by initializing the Board Support Package (BSP). The BSP in this example simply consist of functions used to control the LEDs on the evaluation board. The code for `BSP_Init()` is found in `BSP.C` in the `..\BSP` directory (discussed later). `BSP_Init()` calls `BSP_TickInit()` to initialize the µC/OS-II tick ISR. The `#define OS_TICKS_PER_SEC` in `OS_CFG.H` determines how fast we want the tick interrupt to occur. The actual value depends on the desired granularity for clock ticks. The code is placed in this file to keep the µC/OS-II port files generic. In other words, we didn’t want µC/OS-II’s port files to make any assumptions about which timer would be used to generate tick interrupts. For the example, we decided to use the AVR’s Timer #0 and we configured it for *compare mode*.

L5-2(2) We call `OSStatInit()` to determine the performance of the CPU. What this code does is run the Idle task for 1/10 of a second and measures how fast the CPU is running during that time. From there on, CPU usage will be determined every 1/10 second based on these calculations.

L5-2(3) If you purchased the μC/OS-View package from Micriμm, you can call its initialization functions here. Note that we assumed 38,400 baud but, you can change this value to any other standard baud rate.

L5-2(4) You can now create other tasks.

L5-2(5) This task now starts an infinite loop which blinks the LEDs from left to right and right to left (or up and down depending on the orientation of your board).

5.01.01 Creating a task with OSTaskCreate()

When you create a task using `OSTaskCreate()`, you need to have the following code:

```
OSTaskStkSize     = 256;
OSTaskStkSizeHard =  64;
OSTaskCreate(task,
             (void *)0,
             (OS_STK *)&TaskStk[OSTaskStkSize – 1],
             prio);
```

Note that we used 256/64 in the example above but, you would actually specify the desired size for your task's stack. `prio` is of course, the priority of the task you are creating.

5.01.02 Creating a task with OSTaskCreateExt()

When you create a task using `OSTaskCreateExt()`, you need to have the following code:

```
OSTaskStkSize     = 256;
OSTaskStkSizeHard =  64;
OSTaskCreateExt(task,
                (void *)0,
                (OS_STK *)&TaskStk[OSTaskStkSize – 1],
                prio,
                prio,
                (OS_STK *)&TaskStk[OSTaskStkSizeHard],
               OSTaskStkSize – OSTaskStkSizeHard,
                (void *)0,
                OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR);
```

Again, we used 256/64 in the example above but, you would actually specify the desired size for your task's stack. Also when you pass the 'bottom' of stack to `OSTaskCreate()`, you **MUST** pass it the 'last' location of the hardware stack. You do this because the statistic task can determine the amount of free space available on the software stack. For the statistics, you need to pass the size of the software stack which is of course, the difference between the total size of the stack minus the size allocated for the hardware stack. Unfortunately, the statistic task doesn't return statistics for both stacks. If you need statistics on stack usage of your hardware stacks, you can always create a function to do this.

5.02 app_cfg.h

`app_cfg.h` contains application specific configuration information. Specifically, we define the constants shown in Listing 5-4. We decided to declare the task sizes and task priorities in this file. You can thus change the size of each task in one convenient place and also, setup ALL your task priorities in one file.

You should note that all the stacks have the same size but you can certainly change those as needed.

Listing 5-4, app_cfg.h

```
/*
**********************************************************************
*                            STACK SIZES
**********************************************************************
*/

#define  OS_TASK_START_STK_SIZE          240
#define  OS_TASK_START_STK_SIZE_HARD      64

#define  OS_TASK_START_STK_SIZE          240
#define  OS_TASK_START_STK_SIZE_HARD      64

#define  OS_TASK_1_STK_SIZE              240
#define  OS_TASK_1_STK_SIZE_HARD          64

#define  OS_TASK_2_STK_SIZE              240
#define  OS_TASK_2_STK_SIZE_HARD          64

#define  OS_VIEW_TASK_STK_SIZE           200
#define  OS_VIEW_TASK_STK_SIZE_HARD       64

#define  OS_TASK_TMR_STK_SIZE            240
#define  OS_TASK_TMR_STK_SIZE_HARD        64

#define  OS_TASK_STAT_STK_SIZE           240
#define  OS_TASK_STAT_STK_SIZE_HARD       64

#define  OS_TASK_IDLE_STK_SIZE           240
#define  OS_TASK_IDLE_STK_SIZE_HARD       64

/*
***********************************************************************
*                           TASK PRIORITIES
***********************************************************************
*/

#define  OS_TASK_START_PRIO                0

#define  OS_TASK_1_PRIO                    1
#define  OS_TASK_2_PRIO                    2

#define  OS_VIEW_TASK_PRIO                 3
#define  OS_VIEW_TASK_ID                   3

#define  OS_TASK_TMR_PRIO                  4

/*       OS_TASK_STAT_PRIO       OS_LOWEST_PRIO - 1                  */
/*       OS_TASK_IDLE_PRIO       OS_LOWEST_PRIO                      */
```

```
/*
*********************************************************************
*                            uC/OS-View
*********************************************************************
*/

#define  OS_VIEW_PARSE_TASK               1
#define  OS_VIEW_TMR_32_BITS              0

#define  OS_VIEW_COMM_0                   0
#define  OS_VIEW_COMM_1                   1

#define  OS_VIEW_COMM_SEL     OS_VIEW_COMM_0

#define  OS_VIEW_TMR_1                    1
#define  OS_VIEW_TMR_3                    3

#define  OS_VIEW_TMR_SEL       OS_VIEW_TMR_1
```

5.03 app_isr.s90 (IAR), app_isr.s (ICC)

We decided to place ALL the interrupt service routines for the application in one file, `app_isr.s90`. However, this file currently only contains one ISR, the μC/OS-II tick ISR. Listing 5-5 shows the code for this ISR.

The code for ALL your ISRs **MUST** be **IDENTICAL** except for the portion in **RED** which would change based on the specific ISR.

When you add ISRs, you also need to add the appropriate entry in the interrupt vector table, see `app_vect.s90`.

Listing 5-5, AppTickISR()

```
BSP_TickISR:
     PUSH_ALL                     ;  (1) Save all registers and status register
     IN      R16,SREG             ;  (2) Save the SREG but with interrupts enabled
     SBR     R16,BIT07
     ST      -Y,R16
     PUSH_SP                      ;  (3) Save task's hardware SP onto task's stack

     LDS     R16,OSIntNesting     ;  (4) Notify uC/OS-II of ISR
     INC     R16
     STS     OSIntNesting,R16

     CPI     R16,1                ;  (5) if (OSIntNesting == 1) {
     BRNE    BSP_TickISR_1

     LDS     R30,OSTCBCur         ;         OSTCBCur->OSTCBStkPtr = Y
     LDS     R31,OSTCBCur+1
     ST      Z+,R28
     ST      Z+,R29               ;     }

BSP_TickISR_1:
;    OK to re-enable interrupts if you want to support interrupt nesting!

     CALL    BSP_TickISR_Handler ;  (6) Call tick ISR Handler written in C
     CALL    OSIntExit            ;  (7) Notify uC/OS-II about end of ISR

     POP_SP                       ;  (8) Restore the hardware SP from task's stack
     POP_SREG_INT                 ;  (9) Interrupts of task to return to are ENABLED
     POP_ALL                      ; (10) Restore all registers
     RETI                         ; (11) Return to interrupted task
```

L5-5(1) At the beginning of your ISRs, you need to save ALL the CPU registers onto the interrupted task's stack. This is done with the `PUSH_ALL` macro that we already discussed.

L5-5(2) When the ISR starts, the CPU disables further interrupts by clearing bit 7 of the `SREG` register. When we resume the interrupted task, we need to restore the interrupt enable bit and thus, we save the value of `SREG` register with bit 7 set.

L5-5(3) When the push the hardware stack pointer onto the software stack. Again, we use the `PUSH_SP` macro to accomplish this.

L5-5(4) You need to notify μC/OS-II about the start of an ISR. This is done by incrementing `OSIntNesting`.

L5-5(5) If this is the FIRST nested ISR, we need to save the software stack pointer into the TCB of the current task.

NOTE

At this point, you can re-enable interrupts (by setting bit 7 of the `SREG` register) if you need to support interrupt nesting. If you do, be sure that you allocated sufficient stack space for each task. However, we recommend that you do not re-enable interrupts and that you keep your ISR code as short as possible and simply signal a task and let it handle the majority of the interrupting device's code.

L5-5(6) We can now process the ISR. We call a function so that we can do this in C. We could just as well have handled the ISR in assembly language but C is typically easier to maintain. `BSP_TickISR_Handler()` is found in `APP.C`.

L5-5(7) When you are done servicing the ISR, you need to call `OSIntExit()`. `OSIntExit()` determines whether there is a more important task that needs to run because of the ISR. If there is, a context switch will occur and `OSIntExit()` will NOT return to this ISR code but instead, will exit through `OSIntCtxSw()`, see `OS_CPU_A.S90`.

L5-5(8) If the interrupted task is still the most important task to run then `OSIntExit()` returns and we restore the hardware stack pointer from the software stack.

L5-5(9) Since we will return using the `RETI` instruction, we clear bit 7 of the `SREG` register to ensure that we don't get interrupted as we restore the remaining of the context of the task being resumed. In fact, it would be OK to not disable interrupts except that if an interrupt occurred as we were in the process of restoring the context we could have more than one context on the stack frame of the task and thus risk overflowing the stack. Note that we use the `POP_SREG_INT` macro for this and it's shown below.

```
POP_SREG_INT    MACRO
                LD      R16,Y+
                CBR     R16,BIT07
                OUT     SREG,R16
                ENDM
```

L5-5(10) We then restore the remaining context of the task that was interrupted.

L5-5(11) Finally, we execute a return from interrupt instruction which pops the return address from the hardware stack and also re-enables interrupts.

5.04 app_vect.s90 (IAR), app_vect.s (ICC)

For your convenience, we included an interrupt vector table containing all the entries for the ATmega128 CPU as shown in listing 5-6. You can copy this file to your own project and add the interrupt vectors used in your application.

Listing 5-5, Interrupt Vector Table for the ATmega128 (IAR)

```
    COMMON  INTVEC


APP_INT_VECT_TBL:         ;
                          ; Vector #   Program Address   Interrupt Definition
                          ; --------   ---------------   -------------------------
    DS      4             ;     1          0x0000        Reset
    DS      4             ;     2          0x0002        IRQ0 Handler
    DS      4             ;     3          0x0004        IRQ1 Handler
    DS      4             ;     4          0x0006        IRQ2 Handler
    DS      4             ;     5          0x0008        IRQ3 Handler
    DS      4             ;     6          0x000A        IRQ4 Handler
    DS      4             ;     7          0x000C        IRQ5 Handler
    DS      4             ;     8          0x000E        IRQ6 Handler
    DS      4             ;     9          0x0010        IRQ7 Handler
    DS      4             ;    10          0x0012        Timer2 Compare Match
    DS      4             ;    11          0x0014        Timer2 Overflow
    DS      4             ;    12          0x0016        Timer1 Capture
    DS      4             ;    13          0x0018        Timer1 Compare A
    DS      4             ;    14          0x001A        Timer1 Compare B
    DS      4             ;    15          0x001C        Timer1 Overflow
    JMP     BSP_TickISR   ;    16          0x001E        Timer0 Compare
    DS      4             ;    17          0x0020        Timer0 Overflow
    DS      4             ;    18          0x0022        SPI, STC Transfer Complete
#if OS_VIEW_MODULE > 0
    JMP     OSView_RxISR  ;    19          0x0024        USART0 RX Complete
    DS      4             ;    20          0x0026        USART0 UDR Empty
    JMP     OSView_TxISR  ;    21          0x0028        USART0 TX Complete
#else
    DS      4             ;    19          0x0024        USART0 RX Complete
    DS      4             ;    20          0x0026        USART0 UDR Empty
    DS      4             ;    21          0x0028        USART0 TX Complete
#endif
    DS      4             ;    22          0x002A        ADC    Conversion Complete
    DS      4             ;    23          0x002C        EEPROM Ready
    DS      4             ;    24          0x002E        Analog Compare Interrupt
    DS      4             ;    25          0x0030        Timer1 Compare C
    DS      4             ;    26          0x0032        Timer3 Capture
    DS      4             ;    27          0x0034        Timer3 Compare A
    DS      4             ;    28          0x0036        Timer3 Compare B
    DS      4             ;    29          0x0038        Timer3 Compare C
    DS      4             ;    30          0x003A        Timer3 Overflow
    DS      4             ;    31          0x003C        USART1 RX Complete
    DS      4             ;    32          0x003E        USART1 UDR Empty
    DS      4             ;    33          0x0040        USART1 TX Complete
    DS      4             ;    34          0x0042        Two-wire Serial Interface
    DS      4             ;    35          0x0044        SPM Ready
```

5.05 includes.h

This file is a master include file which is used by `app.c` and possibly, other application files. The idea behind this file is to avoid having to specify each `.H` files in all the `.C` files. In other words, all your application files need to include is `includes.h`.

5.06 os_cfg.h

This is the configuration file for μC/OS-II which is used to specify what μC/OS-II features you would like in your application.

6.00 Board Support Package

The example code includes a simple Board Support Package (BSP) that sets up the tick interrupt as well as provides you with functions to control the LEDs on the STK500/501 evaluation board.

The BSP contains two files, `BSP.C` and `BSP.H` and these files are found in the following directory:

`\MICRIUM\SOFTWARE\EvalBoards\Atmel\STK500\ATmega128\`**`IAR`**`\BSP`

Or,

`\MICRIUM\SOFTWARE\EvalBoards\Atmel\STK500\ATmega128\`**`ICC`**`\BSP`

6.01 LED Controls

The BSP contains code to turn on, turn off or toggle any or all of the LEDs. These functions are:

```
LED_On(INT8U led_id)
LED_Off(INT8U led_id)
LED_Toggle(INT8U led_id)
```

There are advantages to isolating LED control in such functions because it allows you to re-use the same application code on different boards just by simply changing the LED control functions for that board. As you probably know, this is called encapsulation.

'`led_id`' is an identifier associated with each of the 8 LEDs on the STK500 board. When you specify `0`, you will affect ALL the LEDs. For example, `LED_On(0)` means that you want to turn ON all the LEDs. `LED0` on the board has an ID of `1`, `LED1` on the board has an ID of `2`, etc. Thus, to toggle `LED7` on the board, simply call `LED_Toggle(8)`.

6.02 Tick ISR (Using the ATmega128 Timer #0)

The BSP also contains code to initialize and handle the tick ISR needed for μC/OS-II. Initialization is handled by `BSP_InitTickISR()` and the ISR handler is `BSP_TickISR_Handler()` which is actually called by `BSP_TickISR()` (see `app_isr.s90` (IAR) or `app_isr.s` (ICC)).

Listing 6-1 shows the tick ISR handler. We decided to use Timer #0 or the ATmega128. Note that we **DON'T** need to clear the interrupt **IF** we use compare mode on the timer because the hardware does that automatically when it vectors to the ISR. This is **NOT** true for **ALL** interrupting devices so, make sure you clear the interrupt source **IF** you need to re-enable interrupts to service other interrupts. If you don't clear the interrupt before you re-enable interrupts, you would re-enter the ISR and your application will crash!

Listing 6-1, BSP_TickISR_Handler()

```
void  BSP_TickISR_Handler (void)
{
    OSTimeTick();
}
```

6.03 CPU Clock Frequency

`BSP.H` contains a declaration (see below) that tells the rest of the software the CPU clock frequency. For the ATmega128 used on the STK500/501, we assumed 3.684 MHz.

```
/*
**********************************************************************
*                          CPU CLOCK FREQUENCY
**********************************************************************
*/

#define  CPU_CLK_FREQ                3684000
```

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

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

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

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

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

ImageCraft
706 Colorado Ave., #10-88
Palo Alto, CA 94303
USA
+1 650 493 9326
+1 650 493 9329 (FAX)
e-mail: Info@imagecraft.com
WEB : www.ImageCraft.com