μC/OS-II
and
The Atmel AVR

Application Note
AN-1128

www.Micrium.com
## Table Of Contents

1.00 Atmel AT90 (AVR) 3
1.01 Test Setup 5
1.02 Development Tools 6

2.00 Directories and Files 9

3.00 μC/OS-II Port Files 10
3.01 OS_CPU.H 10
3.02 OS_CPU_C.C 11
3.03 OS_CPU_A.S 14

4.00 Application code 17
4.01 app.c 17
4.02 includes.h 17
4.03 os_cfg.h 18
4.04 vectors.s 18

References 19

Contacts 19
In this application note, we assume the ATmega128 chip but, the port can be easily adapted to other AVR parts.

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 1-1.

The Powerdown mode saves the register contents but freezes the Oscillator, 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 1-1, Atmel ATmega128
1.01 Test Setup

To test the µC/OS-II port for the AVR, we used an ATmega128 on an STK500/501 evaluation boards. Figure 1-2 and 1-3 show pictures of the boards and the connections from Port D to the LEDs on board.

Figure 1-2, Atmel STK500/501 Setup for µC/OS-II port Example #1 (LED connections)

Figure 1-3, Atmel STK500/501 Setup for µC/OS-II port Example #1 (Top View)
1.02 Development Tools

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

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. Micrium 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's ANSI C tools offer quality code generation wrapped in an easy-to-use modern GUI Development Environment. They 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 the modern 8/16 bit microcontrollers.

In addition to their C tools, they offer a line of debuggers and hardware development tools through partnerships with other companies. For hardware, besides carrying microcontroller development boards, they carry CANDIP kits (CAN for the AVR), the Atmel ICE-200 In Circuit Emulator and more.

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 JTAG ICE**
The JTAG ICE 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 1-4, Atmel JTAG ICE](image)

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 JTAG ICE 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
2.00 Directories and Files

The AVR port files for the IAR compiler are placed in the following directory:

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

The port files are:

- os_cpu.h
- os_cpu_a.s90
- os_cpu_c.c
- os_dbg.c

The AVR port files for the ImageCraft compiler are placed in the following directory:

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

The port files are:

- os_cpu.h
- os_cpu_a.s
- os_cpu_c.c

Sample code is found in the following directory:

\Micrium\Software\EvalBoards\Atmel\STK500\ATmega128\IAR\Ex1
\Micrium\Software\EvalBoards\Atmel\STK500\ATmega128\ICC\Ex1

This code assumes that you have an Atmel AVR J-Tag emulator and the STK500/STK501 evaluation boards. The code creates four tasks. Three of the tasks simply blink LEDs on the STK500/501. For this, it’s assumed that port D is connected to the LEDs.
3.00 µC/OS-II Port Files

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

- os_cpu.h
- os_cpu_c.c
- os_cpu_a.s
- 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.

3.01 OS_CPU.H

OS_CPU.H contains the 'standard' declarations expected by µC/OS-II. We will simply describe some of the unique items in this file.

**OS_CRITICAL_METHOD**

OS_CRITICAL_METHOD is set to 3 which is the preferred method to disable interrupts. Specifically, OS_ENTER_CRITICAL() calls a function called OS_CPU_SR_Save() which is actually declared in the assembly language file OS_CPU_A.S. OS_CPU_SR_Save() simply obtains the current value of the interrupt disable flag and returns it into a local variable called cpu_sr. Before returning, however, OS_CPU_SR_Save() disables interrupts. OS_EXIT_CRITICAL() calls a function called OS_CPU_SR_Restore() to restore the previously saved interrupt disable flag of the CPU from the local variable cpu_sr.

**OS_TASK_SW()**

OS_TASK_SW() is a macro that gets replaced with a call to OSCtxSw() which is declared in OS_CPU_A.S. There is no need to invoke OSCtxSw() as if it was an interrupt service routine because the context switch can be 'simulated' using a function.

**OSTaskStkSize**

OSTaskStkSize is a global variable that allows us to specify the 'software stack' size of a task. This will be explained later.

**OSTaskHardStkSize**

OSTaskHardStkSize is a global variable that allows us to specify the 'hardware stack' size of a task. This will be explained later.

**OSTickISR_Init(void)**

This prototype is used to specify that a function will be declared to initialize the hardware to generate a tick ISR. OSTickISR_Init() is assumed to reside in your application. For the example code, we decided to place it in APP.C.
**OSTickISR_Handler(void)**

This prototype is used to specify that a C function will be declared to handle the tick ISR. `OSTickISR_Handler()` is assumed to reside in your application. For the example code, we decided to place it in *APP.C*.

### 3.02 OS_CPU_C.C

`OS_CPU_C.C` contains the 'standard' declarations expected by µC/OS-II for hook functions and stack frame initialization. We will simply describe some of the unique items in this file.

**OSTaskCreateHook()**

`OSTaskCreateHook()` is called whenever a task is being created. The code in this function checks to see if µC/OS-View is being used in your product. If you have µC/OS-View, `OSTaskCreateHook()` simply calls the µC/OS-View hook function `OSView_TaskCreateHook()`. For more details about µC/OS-View, consult the Micrium website.

**OSTaskSwHook()**

`OSTaskSwHook()` is called whenever µC/OS-II performs a context switch. The code in this function checks to see if µC/OS-View is being used in your product. If you have µC/OS-View, `OSTaskSwHook()` simply calls the µC/OS-View hook function `OSView_TaskSwHook()`.

**OSTimeTickHook()**

`OSTimeTickHook()` is called by `OSTimeTick()` to process when a µC/OS-II tick occurs. The code for `OSTimeTickHook()` checks to see if µC/OS-View is being used in your product. If you have µC/OS-View, `OSTaskSwHook()` simply calls the µC/OS-View hook function `OSView_TaskSwHook()`.

**OSTaskStkInit()**

`OSTaskStkInit()` is called by `OSTaskCreate()` and `OSTaskCreateExt()` to initialize the stack frame for each task. The stack frame for the AVR is a bit unique in that it contains actually two stacks as shown in figure 3-1, a software stack and a hardware stack.

Both the IAR and ICC compilers create a ‘software stack’ that is used to pass arguments to functions. The ‘hardware stack’ is used by the CPU to save the PC (Program Counter) during function calls and ISRs.

When you create a task with µC/OS-II, you pass the create call (either `OSTaskCreate()` or `OSTaskCreateExt()`) the pointer to the top-of-stack. For the AVR, this is a high memory address because the stack grows downwards. Because of the dual stack requirement of the AVR, we created two ‘global’ variables that tells the task create functions how to distribute the total stack space between the software and hardware stack. Specifically, you need to set `OSTaskStkSize` to the TOTAL stack space (i.e. software and hardware) of the task and, `OSTaskHardStkSize` which defines the size of the hardware stack prior to calling `OSTaskCreate()` or `OSTaskCreateExt()`. Of course, `OSTaskHardStkSize` must NOT be greater than `OSTaskStkSize`. Also, the software stack needs to be large enough to hold all the CPU registers shown in figure 3-1 (34 bytes). In fact, it would not be prudent to only reserve 34 bytes for each software stacks because a task may call nested functions with arguments and thus, the task’s stack would grow beyond the amount of memory you would have allocated. The exact size of a stack is application dependent.
Figure 3-1, Atmel AVR's dual stack
Creating a task with OSTaskCreate()
When you create a task using OSTaskCreate(), you should thus have the following code:

```c
OSTaskStkSize   = 256;
OSTaskHardStkSize = 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.

Creating a task with OSTaskCreateExt()
When you create a task using OSTaskCreateExt(), you should thus have the following code:

```c
OSTaskStkSize   = 256;
OSTaskHardStkSize = 64;
OSTaskCreateExt(task,
                 (void *)0,
                 (OS_STK *)&TaskStk[OSTaskStkSize – 1],
                 prio,
                 prio,
                 (OS_STK *)&TaskStk[OSTaskHardStkSize],
                 OSTaskStkSize – OSTaskHardSize,
                 (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.
3.03 OS_CPU_A.S

OS_CPU_A.S contains code for the following functions:

- OS_CPU_SR_Save()
- OS_CPU_SR_Restore()
- OSStartHighRdy()
- OSCtxSw()
- OSIntCtxSw()
- OSTickISR()

OS_CPU_SR  OS_CPU_SR_Save(void)
This function saves the state of the current interrupt enable/disable status in a local variable on the caller’s stack. Before returning, this function disables interrupts.

void  OS_CPU_SR_Restore(OS_CPU_SR  cpu_sr)
This function restores the state of the interrupt enable/disable status from the variable cpu_sr.

void  OSStartHighRdy(void)
This function starts multitasking by restoring the stack frame of the highest priority task created before you called OSStart(). When this function returns, the CPU will resume execution at the beginning of that task.

void  OSCtxSw(void)
This function is called by OS_Sched() to perform a task level context switch. In other words, the context switch was caused by a task and not by an ISR.

void  OSIntCtxSw(void)
This function is called by OSIntExit() to perform a context switch caused by an ISR. You should note that it’s assumed that the context of the task to suspend was saved by the ISR preamble code.

void  OSTickISR(void)
This function is invoked by the timer that is used to generate tick interrupts. In fact, you should model ALL your ISRs from this code. In other words, ALL your ISRs should be written in assembly language. However, this doesn’t mean that ALL of the ISR handling should be done in assembly language. Specifically, your assembly language ISR should call a C based handler to perform the actual processing associated with the ISR. Your ISRs should thus look as shown below:
Micrium
μC/OS-II for Atmel AVR

_YourISR::

PUSH_ALL ; (1)
IN R16,SREG ; (2)
SBR R16,BIT07
ST -Y,R16
PUSH_SP ; (3)
LDS R16,_OSIntNesting ; (4)
INC R16
STS _OSIntNesting,R16
CPI R16,1 ; (5)
BRNE YourISR_1
LDS R30,_OSTCBCur ; (6)
LDS R31,_OSTCBCur+1
ST Z+,R28
ST Z+,R29

YourISR_1:

CALL _YourISR_Handler ; (7)
CALL _OSIntExit ; (8)
PUSH_SP ; (9)
PUSH_SREG
PUSH_ALL
RET ; (10)

(1) Save ALL the CPU registers as outlined in the macro PUSH_ALL which is declared at the top of the file.

(2) Save the contents of the SREG but, with bit #7 set. In other words, save the SREG with interrupts enabled.

(3) Save the contents of the CPU’s stack pointer onto the interrupted task’s stack. Note that we use another macro, PUSH_SP, which is also declared at the top of the file.

(4) Increment the μC/OS-II global variable OSIntNesting. This notifies μC/OS-II about the fact that we just started an ISR.

(5) If the value of OSIntNesting is 1, this is the first nested ISR and thus, you MUST save the contents of the software stack pointer into the current task’s OS_TCB.

(6) The software stack (the AVR’s Y register or, R29:R28 are saved at OSTCBCur->OSTCBBstkPtr.

(7) You can now call the code that handles the ISR. This is YOUR code which could (and should) be written in C for clarity.

(8) When your C handler returns, you MUST call OSIntExit() which allows μC/OS-II to run the scheduler and determine if the ISR needs to return to the interrupted task or, perform a context switch to a more important task that was made ready-to-run by the ISR.
(9) If OSIntExit() returns, it means that the interrupted task is still the most important task and thus, we need to restore the CPU registers which were saved at the beginning of the ISR. Note that we use assembly language macros to perform the work of restoring the AVR’s registers. Macros are used for clarity.

(10) Finally, you ISR MUST return using a RET instruction and NOT an RETI instruction.
4.00 Application code

The sample code is found in the following directories:

\Micrium\Software\EvalBoards\Atmel\STK500\ATmega128\IAR\Ex1
\Micrium\Software\EvalBoards\Atmel\STK500\ATmega128\ICC\Ex1

In this directory, you will find the following files:

- app.c
- includes.h
- os_cfg.h
- vectors.s

4.01 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.

In this file, two functions have a prefix OS (OSTickISR_Init() and OSTickISR_Handler()) and thus, you would think they ought to be part of the OS (i.e. µC/OS-II).

OSTickISR_Init() is called by the first task that runs after calling OSStart(). OSTickISR_Init() is used to initialize µC/OS-II's system tick. 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.

OSTickISR_Handler() is a function that is called by OSTickISR() to process a system tick. Again, the code for this function 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. In the case of the example code, we decided to use timer #0. OSTickISR_Handler() needs to reload TCNT0 with the proper number of counts until the next interrupt. In fact, Timer #0 could be configured to a different mode which would auto-reload itself. In any case, you MUST call OSTimeTick() in OSTickISR_Handler() to properly process the tick.

4.02 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.
4.03 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.

4.04 vectors.s

This file contains the interrupt vector table for the AVR. Currently, this file only contains the entry for the tick ISR which is generated by timer #0. If you add other ISRs in your system, you should add the pointer to the ISRs in this file.
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
ISBN 0-87930-604-1

Contacts

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

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

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

R&D 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