Porting a Multi-threaded RTOS to Xtesna and Diamond Processors: Example µC/OS-II Port

Application Note
## Contents

1. Introduction ............................................................................................................. 1

2. Porting an RTOS to Tensilica Processors ............................................................... 1
   2.1 Processor Requirements ................................................................................... 2
   2.2 Architecture Specific OS Files ........................................................................... 2
       - OS_CPU.H ........................................................................................................ 2
       - OS_CPU_C.C ................................................................................................. 3
       - OS_CPU_A.S ................................................................................................. 6
   2.2 Processor Run-Time Files ................................................................................. 8

3. Application Example .............................................................................................. 11

4. Xplorer Workspace ................................................................................................ 13

5. Conclusion ............................................................................................................ 16
1 Introduction

This application note describes a port of a real-time operating system to Tensilica Xtensa and Diamond processors (version RA-2006.4) and is intended for software/firmware engineers who are considering porting an RTOS such as µC/OS-II. Although this application note describes a port of the µC/OS-II (v2.80) operating system, the general concepts apply to the porting of almost any real-time operating system to Tensilica processors.

The content of this application note is based upon concepts introduced in the Xtensa Microprocessor Programmer’s Guide. The Xtensa Microprocessor Programmer’s Guide discusses the Xtensa micro-architecture and describes general runtime and context switch software routines. Also, the Xtensa System Software Reference Manual discusses the Xtensa processor HAL – a set of useful processor defines and macros. This application note assumes that the reader is already familiar with these documents, hence no review is offered here. If you have not yet read these documents, please do so before reading this application note.

The µC/OS-II RTOS is a readily available, pre-emptive operating system that is primarily written in C and is hence relatively easy to port. It has been ported to more than 100 microprocessors, from 8-bit to 64-bit, and is considered very cost effective. It has also been validated for safety critical applications in the avionics, automotive and medical industry and is therefore a robust and reliable RTOS. For more details please refer to http://www.uC/OS-II-ii.com or in the text MicroC/OS-II The Real-Time Kernel (Second Edition) by Jean J. Labrosse.

2 Porting an RTOS to Tensilica Processors

A generic software architecture for a multitasking system is shown in Figure 1. There is a layer of software – Hardware Abstraction Layer (HAL) – that serves to abstract the RTOS code from the target CPU. While the RTOS is written in C and is processor independent, the HAL is processor dependent and is commonly written in assembler.
Porting a Multi-threaded RTOS to Xtensa and Diamond Processors

Porting an operating system to a processor often involves development of the HAL. This can be a significant challenge for software/firmware engineers because it requires intimate knowledge of the target processor.

To ease porting development Tensilica provides a “generic” context switch example listed and explained in the Xtensa Microprocessor Programmer’s Guide. The context switch example code is bundled with the Xtensa Xplorer as the “Multi-Threading Demo” (To find the demo in Xplorer, click on Help->Welcome, and select Samples). The context switch example contains basic routines for processor initialization from reset, interrupt/exception handlers, and context switch code. Often the context switch example code can be directly used in most OS ports.

The context switch example code is reused throughout the µC/OS-II port described in this application note. As you will see, there is very little new code that needed to be developed to complete the µC/OS-II port. This approach of reusing the context switch example code can greatly simplify any RTOS porting effort for Xtensa/Diamond processors.

2.1 Processor Requirements

The main processor requirements for a pre-emptive RTOS, such as µC/OS-II, to work correctly are a software and timer interrupt.

For a basic OS, task scheduling occurs whenever a RTOS service is requested. During scheduling, the RTOS performs a priority check with the highest priority given the execution thread. The way in which the RTOS service routine typically calls the scheduler is by generating a software interrupt. For a preemptive RTOS to function correctly, it must also have a periodic timer interrupt that causes task scheduling. This forces the operating system to periodically schedule and run the highest priority task that is ready to run.

Xtensa processors must be configured with a timer and software interrupt level-1 interrupt. All Diamond processors meet these basic RTOS requirements

2.2 Architecture Specific OS Files

A port of µC/OS-II requires the modification of the following RTOS files to match requirements of the specific architecture:

♦ os_cpu.h
♦ os_cpu_c.c
♦ os_cpu_a.S

The remaining RTOS source files are all written in C and can simply be recompiled but the files above are dependant upon the architecture of the processor and therefore represent the HAL depicted in Figure 1.

OS_CPU.H

OS_CPU.H contains processor specific constants, macros, and typedefs.

Type definition for generic data types for the Tensilica processors are as follows:
typedef unsigned char BOOLEAN;
typedef unsigned char INT8U; /* Unsigned 8 bit quantity */
typedef signed char INT8S; /* Signed 8 bit quantity */
typedef unsigned short INT16U; /* Unsigned 16 bit quantity */
typedef signed short INT16S; /* Signed 16 bit quantity */
typedef unsigned int INT32U; /* Unsigned 32 bit quantity */
typedef signed int INT32S; /* Signed 32 bit quantity */
typedef float FP32;       /* Single precision floating point */
typedef double FP64;       /* Double precision floating point */

OS_CPU.h contains #defines to determine how to enter and exit critical sections. When a critical section is entered, all processor interrupts should be disabled. The OS_ENTER_CRITICAL macro uses the rsil instruction which is an atomic operation that is used to set the interrupt level to 15 (masks all interrupts) while saving the value of the program status (PS) register. The intrinsic function, XT_RSIL maps to the rsil instruction and is defined in the xt_core.h header file. The saved value is written back to the PS register upon calling OS_EXIT_CRITICAL using the intrinsic function, XT_WSR_PS. Note the use of the ESYNC intrinsic function for both macros to allow the RSIL/WSR to complete before executing any further instructions.

#include <xtensa/tie/xt_core.h>
static  INT32U   OS_XTENSA_PS; //this is a global variable
/* Disables interrupts */
#define OS_ENTER_CRITICAL()  OS_XTENSA_PS =XT_RSIL(15) ; XT_ESYNC()
/* Enables interrupts */
#define OS_EXIT_CRITICAL()   XT_WSR_PS(OS_XTENSA_PS); XT_ESYNC()

The Xtensa processor has a stack which grows from high memory to low therefore:

#define OS_STK_GROWTH        1

Finally, a context switch is invoked by OS_TASK_SW(). This macro should generate a level-1 software interrupt. For the Diamond processors, the level-1 software interrupt is configured as Interrupt #7. Therefore, bit D7 of the INTENABLE register is asserted to set the software interrupt.

#include <xtensa/hal.h>
#define SOFTWARE0_INT_MASK    0x000000080
#define OS_TASK_SW()          xthal_set_intset (SOFTWARE0_INT_MASK)

The xthal_set_intset() function is a link time function of the Xtensa Hardware Abstraction Layer (HAL). The Xtensa HAL is a set of defines, macros, and functions that are useful in developing firmware for Diamond processors and Xtensa configuration independent code in general. Refer to the Xtensa System Software Reference Manual for further information regarding the Xtensa HAL.

OS_CPU.C.C

A µC/OS-II port requires modification of several C functions. Some functions are “hook” functions. µC/OS-II extends tasks by using “hooks”, allowing port or processor specific functionality on top of standard RTOS functionality. However, this port simply stubs most
of the hooks as they are not used. The only hook that is used is the OSTaskIdleHook.

```c
void OSTaskIdleHook (void)
{
    /* WAITI instruction saves power by setting the current interrupt
     * level, gating clocks to processor's logic and waiting for an
     * interrupt
     */
    asm("waiti 0");
}
```

Architecture and implementation specific C functions are also included in the OS_CPU_C file. The OSTaskStkInit() function is used to allocate and initialize a stack for a new task. The context switching example contains an add_task() function (contained in task.c and task.h) that has the exact functionality required by the OSTaskStkInit() function. Therefore, the add_task() code is reused and slightly modified to match the C syntax of the OSTaskStkInit() function for μC/OS-II. The add_task() function is described in section 7.1 of the Xtensa Microprocessor Programmer’s Guide. Note the use of Xtensa HAL constants (named as XCHAL_*) to setup the processor PS register.
As mentioned earlier, µC/OS-II may perform context switching during a timer or software interrupt. Context switch is accomplished by save/restoring task context and then switching task stacks. For the Diamond port, a level-1 interrupt is asserted for both timer and software interrupts (see the `OS_TASK_SW()` macro described earlier). The level-1 interrupt handler provided in the context switch example is reused for this port. The level-1 interrupt handler is exactly the same as provided in the context switch example and is not described in this document. Refer to the *Microprocessor Programming Guide* for further information on how the level-1 interrupt handler performs the context switch. The level-1 interrupt handler also exports `userStackPtr` and `newUserStackPtr` variables that are used to communicate task stack addresses with RTOS code.

The level-1 handler performs a register context save to the current stack, calls into the interrupt dispatch handlers, then to `OSIntCtxSW()` to switch task stacks, and finally restores register context from the new task stack upon return from `OSIntCtxSW()`.

The `OSCtxSW()` routine is shown below. First, the current stack is saved, prior to calling a stub function `OSTaskSwHook()` (This hook function allows piggyback of custom code that is executed for every task switch – it is stubbed out for this port). Then, the stack of

```c
/* Processor Specific */
/* frame defines in # of bytes */
#define FRAME_SIZE 124
#define FRAME_AR(x) (x * 4)
#define FRAME_PS 80
#define FRAME_PC 84

/***************************************************************************/
/*                            Initialize Stack for the Task */
/***************************************************************************/

OS_STK *OSTaskStkInit(void (*task)(void *pd), void *pdata, OS_STK *ptos,
INT16U opt)
{
    /*
This function performs these functions
1. Computation of the Initial Stack
2. Initialization of the Stack */

    OS_STK *sp1;
    INT32U i;

    opt = opt;

    sp1 = (OS_STK *)((INT32U)ptos & 0xffffffff);
    sp1 = sp1 - FRAME_SIZE / 4;

    for( i = 0 ; i < FRAME_SIZE / 4; i++ )
        sp1[i] = 0 ;

    sp1[ FRAME_PC/4 ] = (INT32U)task;
    sp1[ FRAME_PS/4 ] = XCHAL_PS_WOE_MASK | XCHAL_PS_UM_MASK |
                     XCHAL_PS_EXCM_MASK | (1 << XCHAL_PS_CALLINC_SHIFT);

    sp1[ FRAME_AR(6)/4 ] = (INT32U)pdata;
    sp1[ FRAME_AR(1)/4 ] = (INT32U)(sp1 + FRAME_SIZE / 4);
    return(sp1);
}
the highest priority task in ready state becomes the new stack.

```c
void OSCtxSw(void)
{
    OSTBCur->OSTCBStkPtr = userStackPtr;
    OSTaskSwHook();
    OSTBCur = OSTCBHighRdy;
    OSPrioCur = OSPrioHighRdy;
    newUserStackPtr = OSTCBHighRdy->OSTCBStkPtr;
}
```

A function is added to enable the context switch interrupts. The `set_ccompare` and `enable_ints` functions are taken from the context switch example (`interrupts.h` and `timer.h`) and are used to set the timer CCOMPARE0 register and enable interrupts respectively.

```c
void OSCtxSwIntsEnable(void)
{
    set_ccompare0( read_ccount() + TIMER0_INTERVAL );
    enable_ints(SOFTWARE0_INT_MASK|TIMER0_INT_MASK);
}
```

**OS_CPU_A.S**

This file primarily contains the code for `OSStartHighRdy()`. The `OSStartHighRdy()` function is executed prior to multitasking. This function calls the `OSTaskSwHook`, sets the µC/OS-II OSRunning flag to 1, enables context switch interrupts, and initializes the register context of the highest priority task.
.text
.align 4
OSTCBHighRdy:
    .word OSTCBHighRdy /* UC/OS-II variable */
OSRunning:
    .word OSRunning /* UC/OS-II variable */
#if OS_CPU_HOOKS_EN
OSTaskSwHook:
    .word OSTaskSwHook /* Defined in OS_CPU_C.C */
#endif /* OS_CPU_HOOKS_EN */
OSCtxSwIntsEnable:
    .word OSCtxSwIntsEnable /* Defined in OS_CPU_C.C */

.text
.align 4
OSStartHighRdy:
    entry a1, 32
    /*
    ** UC/OS-II-II requires a call to OSTaskSwHook()
    */
#if OS_CPU_HOOKS_EN
l32r a3, _OSTaskSwHook
    callx8 a3
#endif /* OS_CPU_HOOKS_EN */
    /*
    ** Set the OSRunning
    */
l32r a2, _OSRunning /* a2 = OSRunning = &ValueOfOSRunning */
movi a3, 1
    s8i a3, a2, 0 /* set ValueOfOSRunning = 1 = TRUE */
    /*
    ** Enable context switch interrupts
    */
l32r a3, _OSCtxSwIntsEnable
    callx8 a3
    /*
    ** Get the stack pointer of the task ready to run
    */
l32r a3, _OSTCBHighRdy /* a3 = *OSTCBHighRdy = &OSTCBHighRdy */
l32i a3, a3, 0 /* a3 = &OSTCBHighRdy = &ValueOfOSTCBHighRdy */
l32i a1, a3, 0 /* a1 = ValueOfOSTCBHighRdy */
Note that the instruction “rfe” is used to return from the routine. The “rfe” instruction uses the value in the EPC_1 (Exception Program Counter Level-1) register as the return address. The EPC_1 register is loaded from the task stack and was initialized as the starting address of the task when the task was created (see OSTaskStackInit() function described in a previous section). Upon executing rfe, multi-tasking begins with the processor executing the highest priority task.

2.3 Processor Run-Time Files

The HAL consists of OS files that require processor specific customization along with processor specific run-time files. The previous section covered all modifications of the µC/OS-II files. The run-time files include low-level reset and interrupt handlers. The run-
time files are taken directly from the Xtensa Microprocessor Programmer’s Guide Context Switch Example. Because the run-time files are covered in great detail in comments interspersed in the source file, as well as in the Xtensa Microprocessor Programmer’s Guide, they are not discussed in this application note. The basic run-time code for Diamond Cores (that is also used in this application note) is provided in the Multi-Threading demonstration that is included with Xplorer-DE.

The only run-time file that required modification was the interrupt dispatch code (intdisp.c) and the timer header file (timer.h). The intdisp.c file contains code that dispatches handler functions for each interrupt source. Since µC/OS-II uses a software and timer interrupt, these dispatch routines were modified.

The software interrupt dispatch routine simply calls the OSCtxSw() routine that was discussed in the previous section. Then, the software interrupt is cleared. Note the use of the Xtensa HAL function xthal_set_intclear to clear the INTERRUPT register.

```c
#define SOFTWARE0_INT_MASK 0x000000080

void L1_int7_software()
{
    OSIntCtxSw();
    xthal_set_intclear(SOFTWARE0_INT_MASK);
}
```

The timer interrupt dispatch routine increments the timer COMPARE register to cause an interrupt at the next periodic interval and then dispatches OS functions as required by µC/OS-II (OSTimeTick and OSIntExit) for the OS timer interrupt. Note that the OSIntExit function calls routines to check the highest priority ready task and then calls OSIntCtxSw() to cause a task switch.

```c
void L1_int6_timer0()
{
    /* Interrupt handler for Level 1 Timer Interrupt 0 */
    unsigned long old_ccompare;
    unsigned long diff;

    //These lines required by uC/OS-II
    OSIntNesting++;
    if (OSIntNesting == 1)
    {
        OSTCBCur->OSTCBStkPtr = userStackPtr;
    }

    //Increment timer compare register
    do {
        system_ticks++;
        old_ccompare = read_ccompare0();
        set_ccompare0( old_ccompare + TIMER0_INTERVAL );
        diff = read_ccount() - old_ccompare;
    } while ( diff > TIMER0_INTERVAL );

    //Call OS functions for timer interrupt
    OSTimeTick();
    OSIntExit();
}
```
The `timer.h` file contains the interval time for timer0. The interval time is increased to 0x8000 to provide sufficient cycles for OS functions such as scheduling.

```
#define TIMER0_INTERVAL 0x00008000
```

And that’s it......you have a working μC/OS-II port for the Xtensa/Diamond cores.
3 Application Example

The application example is contained in the `test.c` file. The main function of a µC/OS-II application example is shown below. The application initializes the OS, and initializes four tasks (Task1 to Task4) and sets their priorities (the lower the priority numbers, the higher the priority). Then, the OS is started.

```c
#define TASK_STK_SIZE 10000
#define TASK_NUM 3
#define TASK_1_PRIO 11
#define TASK_2_PRIO 12
#define TASK_3_PRIO 13

OS_STK TestTaskStk[TASK_NUM][TASK_STK_SIZE];

int main(void)
{
    OSInit();

    OSTaskCreate(Task1, (void *)0, &TestTaskStk[0][TASK_STK_SIZE], TASK_1_PRIO);
    OSTaskCreate(Task2, (void *)0, &TestTaskStk[1][TASK_STK_SIZE], TASK_2_PRIO);
    OSTaskCreate(Task3, (void *)0, &TestTaskStk[2][TASK_STK_SIZE], TASK_3_PRIO);

    OSStart();
    return(0);
}
```

The task1 function is listed below. These simple task increments a local variable, count, does a deep call check and then prints count. Note that the `printf` library function is not thread safe. Therefore, it is enclosed between operating system calls, `OSSchedLock` and `OSSchedUnlock`, to prevent a context switch while calling `printf`. The task then sleeps for 30 timer ticks. Tasks2 and 3 are similar to Task 1.

```c
void Task1(void *pdata)
{
    pdata = pdata;
    volatile count = 0;

    for(;;)
    {
        count++;
        OSSchedLock();
        printf("Task I - starting - %d\n", count);
        OSSchedUnlock();
        deep_call(10);
        OSSchedLock();
        printf("Task I - ending - %d\n", count);
        OSSchedUnlock();
        OSTimeDly(30);
    }
}
```
Applications with a deep calling stack can be particularly stressful for multi-tasking systems. Therefore, the test program uses the `deep_call` function to call itself iteratively for the number of times defined in the argument. The `deep_call` function performs a basic variable check to ensure that context is preserved while switching context. Also, a delay function is added to increase the likelihood that a task switch occurs while the call stack is deep. The use of the `global_fool_the_optimizer` variable is redundant and only used to prevent the compiler from optimizing away the calling depth.

```c
#define DELAY 1000
int global_fool_the_optimizer = 0;

void deep_call (int depth)
{
    int a = depth;
    int b = depth + 1;
    int c = depth + 2;
    int d = depth + 3;

    if (global_fool_the_optimizer)
    {
        a = b = c = d = 10;
    }

    // dummy delay
    for (temp=0; temp<DELAY; temp++) {};

    if (depth > 0)
        deep_call (depth - 1);

    if ((a != depth) || (b != depth + 1) || (c != depth + 2) || (d != depth+3))
    {
        OSSchedLock();
        printf("------failed-------\n");
        OSSchedUnlock();
    }
}
```

Upon simulating the example, the console output is shown below.

```
Task I - starting - 1
Task I - ending - 1
    Task II - starting - 1
    Task II - ending - 1
    Task III - starting - 1
    Task III - ending - 1
Task I - starting - 2
Task I - ending - 2
    Task III - starting - 2
    Task II - starting - 2
    Task II - ending - 2
    Task III - ending - 2
Task I - starting - 3
Task I - ending - 3
    Task III - starting - 3
    Task II - starting - 3
    Task II - ending - 3
    Task III - ending - 3
Etc. etc. etc. etc.
```
## 4 Xplorer Workspace

An Xplorer-DE workspace is provided along with this application note that contains all the code for the Hardware Abstraction Layer (HAL) of a working µC/OS-II (version 2.80). The Xplorer-DE workspace for the µC/OS-II includes sources listed in the table below:

<table>
<thead>
<tr>
<th>Code Taken From:</th>
<th>Code File Name:</th>
<th>Modifications:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xplorer-DE Sample</td>
<td>Internal.h</td>
<td></td>
</tr>
<tr>
<td>“Multi-Threading Demo”</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Interruptions.h</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Task.h</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Timer.h</td>
<td>Increased timer interval</td>
</tr>
<tr>
<td></td>
<td>DebugExceptionVector.S</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DebugExceptionVectorHandler.S</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Int2_medpri_dispatcher.S</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Int3_medpri_dispatcher.S</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Int4_highpri_handler.S</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Int5_highpri_handler.S</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Intdisp.c</td>
<td>Changed dispatch to call RTOS functions</td>
</tr>
<tr>
<td></td>
<td>Kernv.S</td>
<td></td>
</tr>
<tr>
<td></td>
<td>L1h.S</td>
<td></td>
</tr>
<tr>
<td></td>
<td>L2v.S</td>
<td></td>
</tr>
<tr>
<td></td>
<td>L3v.S</td>
<td></td>
</tr>
<tr>
<td></td>
<td>L4v.S</td>
<td></td>
</tr>
<tr>
<td></td>
<td>L5v.S</td>
<td></td>
</tr>
<tr>
<td></td>
<td>NMIExceptionVector.S</td>
<td></td>
</tr>
<tr>
<td></td>
<td>reset.S</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Userv.S</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Windv.S</td>
<td></td>
</tr>
<tr>
<td>µC/OS-II HAL</td>
<td>Os_cpu.h</td>
<td>Changed Defines/Macros for Xtensa/Diamond</td>
</tr>
<tr>
<td></td>
<td>os_cpu_c.c</td>
<td>Created C-level RTOS HAL functions for Xtensa/Diamond</td>
</tr>
<tr>
<td>Example Application Code</td>
<td>Test.c</td>
<td>Added simple multi-task sample application.</td>
</tr>
</tbody>
</table>
You may install this workspace in Xplorer-DE to reference the code described in this application note. Note that the HAL code discussed in this application note is slightly modified from the workspace code for ease of understanding.

The workspace has been tailored for the DC-108mini Diamond processor (version RA-2006.4). The workspace can be targeted for other Diamond processors; however, the context of MAC16, Booleans, HiFi2, Vectra LX will not be saved/restored upon context switch.

To duplicate a working example, you must add µC/OS-II code (provided in the text MicroC/OS-II The Real-Time Kernel Second Edition by Jean J. Labrosse) listed below:

- Os_core.c
- Os_dbg_r.c
- Os_mbox.c
- Os_mem.c
- Os_mutex.c
- Os_q.c
- Os_sem.c
- Os_task.c
- Os_time.c

The Xplorer-DE workspace also contains the appropriate C build properties listed below:

- Compile reset.S with –text-section-literals assembler option. Refer to section 4.12 “Building the Reset Vector” for explanation of why this is required.
- Use the nort (no run-time) linker support package since reset and interrupt vectors are provided
- Use –lc –lsim –lhal flags while linking to include the standard C library, simcall library, and link-time HAL function library respectively.

This workspace can also be used for Xtensa processors; however, the code is dependent on several configuration options. To use this workspace with Xtensa processors, you should prepare a configuration with options equal to the DC_108mini. The pertinent configuration options are defined below (configuration options that are not listed do not affect the port). The code can be used across processor configurations with different options, however some code modification will be required.
### Instruction options

<table>
<thead>
<tr>
<th>Feature</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>16-bit MAC with 40-bit Accumulator</td>
<td>no</td>
</tr>
<tr>
<td>Floating Point (coprocessor id 0)</td>
<td>no</td>
</tr>
<tr>
<td>Boolean Registers</td>
<td>no</td>
</tr>
<tr>
<td>Number of Coprocessors (NCP)</td>
<td>0</td>
</tr>
<tr>
<td>Vectra LX DSP coprocessor instruction family</td>
<td>no</td>
</tr>
<tr>
<td>HiFi2 Audio Engine DSP coprocessor instruction family</td>
<td>no</td>
</tr>
<tr>
<td>Interrupts enabled?</td>
<td>yes</td>
</tr>
<tr>
<td>Interrupt count</td>
<td>15</td>
</tr>
<tr>
<td>Int 6  type / priority level</td>
<td>Timer / 1</td>
</tr>
<tr>
<td>Int 7  type / priority level</td>
<td>Software / 1</td>
</tr>
<tr>
<td>Timer count</td>
<td>yes</td>
</tr>
<tr>
<td>Timer 0</td>
<td>6</td>
</tr>
</tbody>
</table>
5 Conclusion

This document shows how an RTOS, such as µC/OS-II, was ported to Xtensa and Diamond processors. The porting effort was easy due to reuse of generic context switch example code provided by Tensilica. Regardless of the RTOS that you wish to port, the first steps to a successful port include learning the context switch example and reuse as much of this code as possible. The context switch example code is provided in Xplorer bundle as a multi-threaded sample and is described in the Xtensa Microprocessor Programmer’s Guide.

Note that the µC/OS-II port that is provided is un-optimized and has been lightly tested for basic functionality. This port serves as a starting point that will require additional engineering effort by software engineers intending to create a production-worthy system.