

# *RTOS DAC Profiling*

**Organization:**  
Liebert Corporation

**Written By:**  
Kevin Eschhofen

**Revision Level-Date:**  
1 - 6/20/2005



Table of contents

**1 INTRODUCTION..... 1**

**2 TASK PROFILING ..... 1**

**3 INTERRUPT PROFILING..... 3**

**4 TASK AND INTERRUPT PROFILING ..... 6**

**5 HARDWARE..... 7**



# 1 Introduction

In a Real Time Operating Systems (RTOSs) it can be helpful to have a window into the OS system, providing visual information for profiling and diagnostic purposes. This window would be provided through an oscilloscope displaying signal information from a Digital to Analog Converter (DAC). This can especially be helpful when the system has no profiling tools. The signaling can provide valuable diagnostic information on software operations, coordination, tracking and timing information. Task signaling displays task execution and timing relative to other tasks. Interrupt signaling shows interrupt timing, duration and interaction with tasks (when task signaling is displayed).

To provide the output of these signals, a 4 channel, 8bit Digital to Analog Converter (DAC) is used. The DAC should be fast enough for both bus accesses and output settling time (approximately 1microS). Fast bus writes reduce DAC write time and overhead of signal profiling on the system. Fast settling times provide better signal response time and waveform representation. One channel is dedicated for OS task signaling and the other is dedicated for interrupt signaling. The remaining channels are open for other diagnostics. After development is complete, the DAC can be removed (not populated) to reduce cost.

The DAC signaling happens in the following manner. At a selected point in the software (i.e. at the beginning of a task), code will update an individual DAC output, signaling the occurrence of an event, which event has occurred and how long it ran. At another point in the software, code will update the DAC output to signal the end of an event.

# 2 Task Profiling

For tasks profiling, the DAC has to be updated whenever task switching occurs. This signals the start of the task. The level of the DAC output indicates which task is running and its priority. When the task finishes execution, the DAC is updated with a value at or near zero to indicate idle task time (No system OS tasks are running). One should note that the oscilloscope ground reference should be adjusted accordingly as the OS idle task signal will not come out to be exactly zero. If a task is currently running and is preempted by another task, the new task's priority is signaled out the DAC (see Figure 2-1). When the second task is finished, task switching will return execution to the previous task and the DAC will be updated with the previous task's priority.

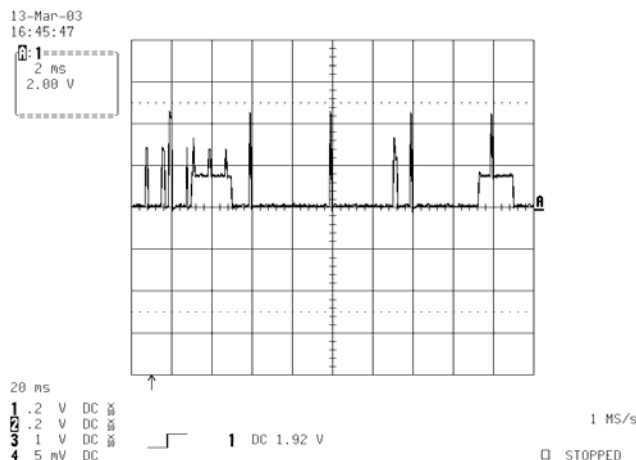


Figure 2-1: Task Profile

The signaling to the DAC has to take place when tasks switch. The preferred place to do this in uCOS/II is in the TaskSwitchHook routine. Of course, the Task switch option has to be enabled for this to work. Once enabled, every time a task switch takes place the TaskSwitchHook routine will be called. The Code to

update the DAC has to be in or called by this routine. Having it in the TaskSwitchHook routine reduces the overhead of another subroutine call. Another less desirable option is to put the DAC update code in the OS code itself. It does make the code more time efficient by less encapsulated.

The code to update the DAC for task signaling is shown in Listing 2-1 . This can be put in the task switch hook function. In this case the function is called (actually it is “inlined”) in the OSSched (shown in Listing 2-2 ) and OSIntExit (shown in Listing 3-2 ).

```
inline void writeTaskPrio2DAC(int TempOSPrio)
{
    anaTaskPrio = 256 - (TempOSPrio*4); //Calculate task prioity for DAC
    port8F01 = anaTaskPrio;           //Write task priority to DAC port.
    port8F04 = anaTaskPrio;           //Update DAC outputs by writing to latch location.
}
```

**Listing 2-1: DAC Task update code**

```
void OSSched (void)
{
    INT8U y;

    OS_ENTER_CRITICAL();
    if ((OSLockNesting | OSIntNesting) == 0) { /* Task scheduling must be enabled and not ISR level */
        y = OSUnMapTbl[OSRdyGrp]; /* Get pointer to highest priority task ready to run */
        OSPrioHighRdy = (INT8U)((y << 3) + OSUnMapTbl[OSRdyTbl[y]]);
        if (OSPrioHighRdy != OSPrioCur) { /* No context switch if current task is highest ready */
            OSTCBHighRdy = OSTCBPrioTbl[OSPrioHighRdy];
            OSCtxSwCtr++; /* Increment context switch counter */

            writeTaskPrio2DAC( OSPrioHighRdy );

            OS_TASK_SW(); /* Perform a context switch */
        }
    }
    OS_EXIT_CRITICAL();
}
```

**Listing 2-2: Task update in the OSSched code**

The following formula is used to calculate the task value to be writing to the DAC:

DAC output for task priority = 256 - (OSTaskPriority \*4)

The priority is multiplied by 4 to scale it for better DAC resolution. This operation could be done with a left shift of two for faster execution in a processor that has long multiply operations. The scaled priority was then subtracted from 256 to invert the signal. This results in the higher DAC outputs representing higher priority tasks. Remember that the lower the task priority number is the higher the priority of the task.

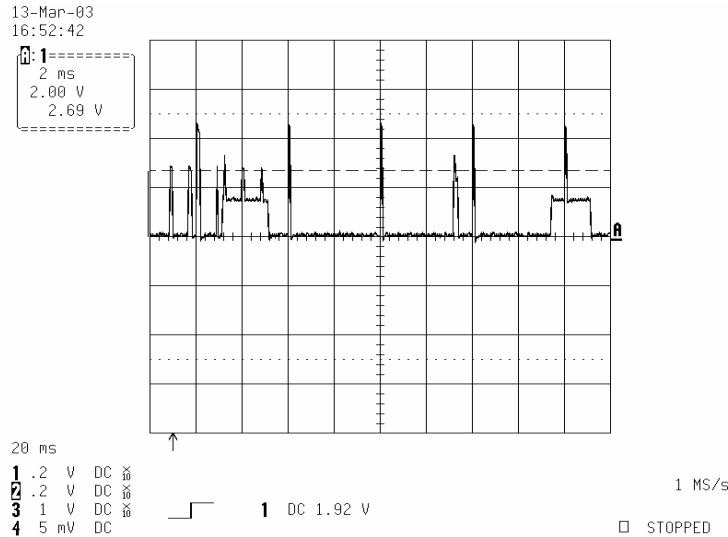
Given the task priority, we can calculate its voltage level expected on the oscilloscope. Being that we are using a 5 volt DAC, Volts/bit = 5/255 = 19.6mV/bit. If a task’s priority is 30, the value output to the DAC would be calculated by:

DAC output for task priority =  $256 - (30 * 4) = 136$

The resulting voltage would be:

DAC Volts = DAC value \* DAC Volts/bit =  $136 * 19.6\text{mv/bit} = 2.66 \text{ Volts}$ .

Figure 2-2 displays the task with a voltage level of 2.66 volts (at the dotted line), the Task with a priority of 30.



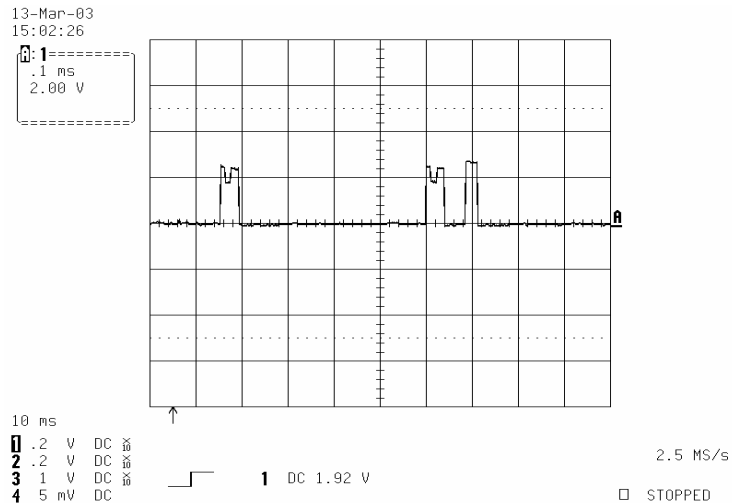
**Figure 2-2: Task Profile, With Task Priority 30 Level**

### 3 Interrupt Profiling

For interrupts the DAC is updated with the interrupt number at the beginning of an interrupt (OSIntEnter) and again at the end of the interrupt (OSIntExit). The value written to the DAC represents the interrupt source or number. One should note that in some processors the interrupt number does not necessarily indicate interrupt priority. At the end of the interrupt execution, the DAC is updated to zero. Interrupt number or source may be application dependent. The following formula is used when writing to the DAC.

DAC output for interrupt priority =  $256 - (\text{Interrupt Source} * 3)$

Once an interrupt signal level is calculated it is output to the DAC. Because interrupts can nest, the interrupt signal level is stored into an array buffer for later processing. Figure 3-1 shows an interrupt profile with nested interrupts. When a second interrupt occurs while the first interrupt is still executing (nests), the interrupt DAC signal is updated to the new interrupt level and stored on the buffer array. When the second interrupt finishes, the interrupt signal level for the first interrupt is retrieved from the buffer array and output to the DAC. The first position in the array buffer is initialized to zero to represent interrupt idle time. When the last of interrupts finishes execution, the zero will be output to the DAC, indicating no interrupts are currently running.



**Figure 3-1: Interrupt Profile with Nested Interrupts**

Being that there is no “switch hook” operation for interrupt handling, the interrupt signaling is embedded into the OSIntEnter and OSIntExit routines. Listing 3-1 shows the code for signaling the start of the interrupt. Listing 3-2 shows the code for signaling the end of the interrupt.

```

void OSIntEnter (void)
{
    OS_ENTER_CRITICAL();
    OSIntNesting++;          /* Increment ISR nesting level          */

    //This code has been added to the normal OS routine.
    anaIntPrio = 256 - (IntSource * 3);    //Output interrupt priority to DAC
    anaIntPrioBuffer[OSIntNesting] = anaIntPrio; //Store interrupt priority for later signaling in OSIntExit.
    port8F00 = anaIntPrio;                //Write task priority to DAC port.
    port8F04 = anaIntPrio;                //Update DAC outputs by writing to latch location.
    //End of added code.

    OS_EXIT_CRITICAL();
}

```

**Listing 3-1: Update DAC interrupt signal upon entry**

```

void OSIntExit (void)
{
    OS_ENTER_CRITICAL();

    //This code has been added to the normal OS routine.
    anaIntPrio = anaIntPrioBuffer[OSIntNesting - 1]; //Retrieve previous interrupt priority.
    port8F00 = anaIntPrio; //Write task priority to DAC port.
    port8F04 = anaIntPrio; //Update DAC outputs by writing to latch location.
    //End of added code.

    if ((--OSIntNesting | OSLockNesting) == 0) { /* Reschedule only if all ISRs completed & not locked
    */
        OSIntExitY = OSUnMapTbl[OSRdyGrp];
        OSPrioHighRdy = (INT8U)((OSIntExitY << 3) + OSUnMapTbl[OSRdyTbl[OSIntExitY]]);

        if (OSPrioHighRdy != OSPrioCur) { /* No context switch if current task is highest ready */
            OSTCBHighRdy = OSTCBPrioTbl[OSPrioHighRdy];
            OSCtxSwCtr++; /* Keep track of the number of context switches */

            writeTaskPrio2DAC( OSPrioHighRdy );
            OSIntCtxSw(); /* Perform interrupt level context switch */
        }
    }
    OS_EXIT_CRITICAL();
}

```

### Listing 3-2: Update DAC interrupt signal upon exit

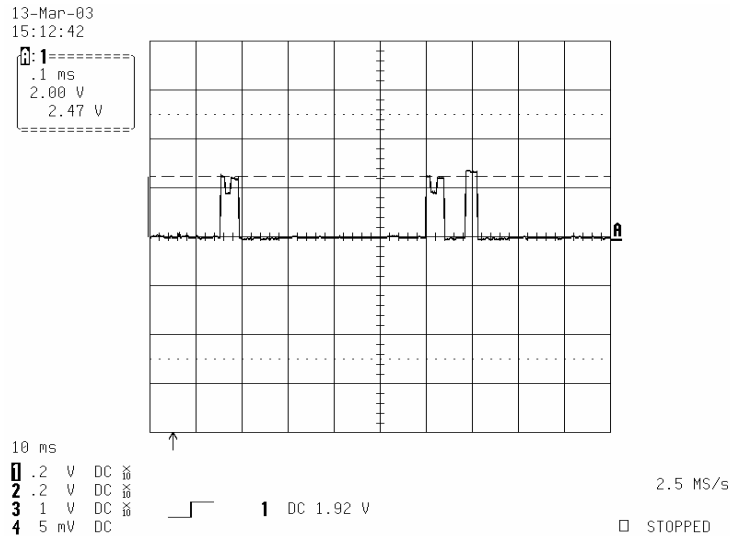
Given the interrupt number, we can calculate its voltage level expected on the oscilloscope. Being that we are using a 5 volt DAC, Volts/bit =  $5/255 = 19.6\text{mV/bit}$ . If an interrupt's number is 44, the value output to the DAC would be calculated by:

$$\text{DAC output for interrupt number} = 256 - (44 * 3) = 124$$

The resulting voltage would be:

$$\text{DAC Volts} = \text{DAC value} * \text{DAC Volts/bit} = 124 * 19.6\text{mV/bit} = 2.43 \text{ Volts.}$$

Figure 2-2 displays the interrupt with a voltage level of 2.43 volts (at the dotted line), the Interrupt with a number of 44.

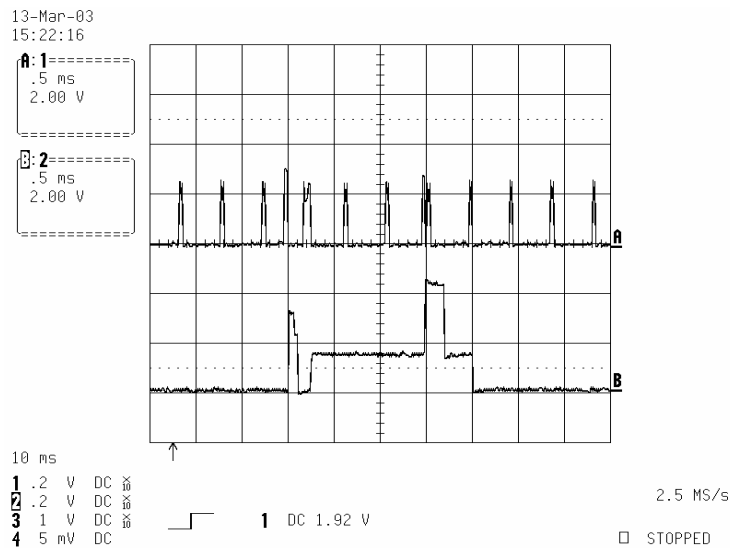


**Figure 3-2: Interrupt Profile, With Interrupt Number 47(0x2F) Level**

## 4 Task and Interrupt profiling

The following is an expanded scope plot of both the interrupt signaling (trace: A) and the task signaling (trace: B). The plot has been expanded to provide better viewing of the tasks running. From this display one can see interaction between interrupts and tasks. For instance an interrupt may post to a task, from the task one can see the task wakeup and run. Also one can see the task running in the time left over after interrupt execution.

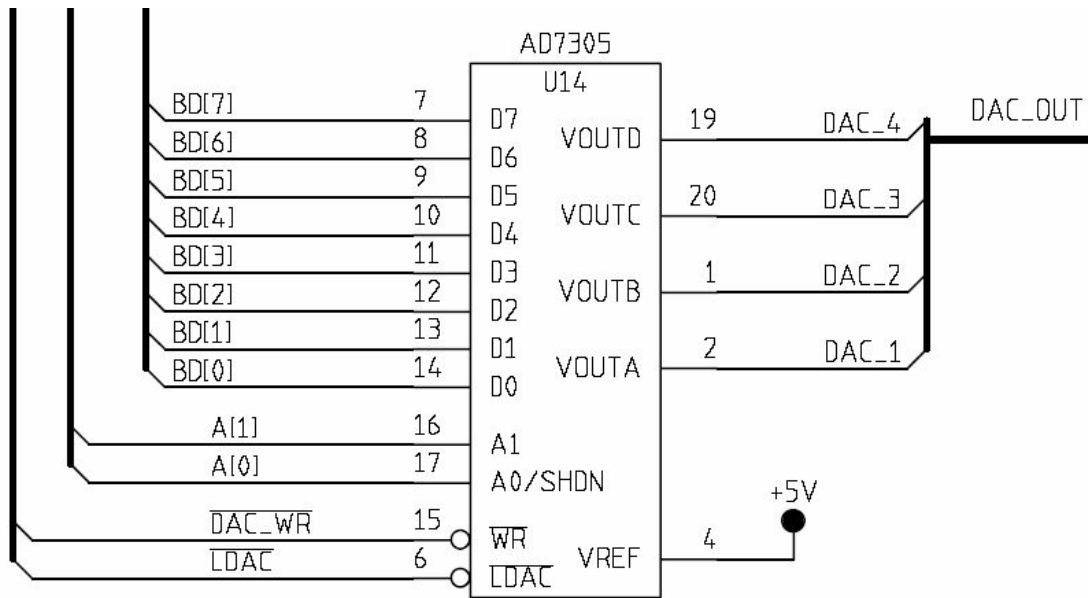
Another advantage to DAC signaling is that one may be able to view a possible cause of a processor reset or crash. The task or interrupt that was running should be displayed. This would require the oscilloscope to trigger on a signal drop out.



**Figure 4-1: Task and Interrupt Profile**

## 5 Hardware

The DAC is an eight bit, four channel DAC. The address bus selected one of the four channels to be updated. DAC\_WR is the write control line to the device. When driven low data is written the respective channel in the device. LDAC is the Latch line for the output. When data is written to the device data does not show up on the output until the latch line is toggled. Toggling this line latched the channels to the output lines. A CPLD was used to address decode the LDAC line. Writing to the address would toggle the line. Each time a channel is updated. The previous code writes a value to the DAC channel, and then writes to the address decode LDAC line to update the DAC output. The DAC provides 0-5VDC out.



### Limitations:

One limitation to the DAC signaling is the fact that the signals do not show the overhead time for interrupt handling. This is the time for the processor to recognize an interrupt, break normal code execution and start to process the interrupt. In most cases these times are minimal, but may need to be considered if idle time is low or the number of interrupts executed is high.