Task Scheduling on NIOS II

When creating your projects you may need to design code that has some real-time constraints. For example, you may wish to run a certain piece of your code every 10 milliseconds. This is especially useful for games, but could be useful for almost any software application. Consider a classic platformer game, such as Super Mario Bros. The user controls the in-game character and moves him left and right across the screen. This type of animation can be done in a loop by erasing and then redrawing the character at a shifted position. The loop will also contain some other processing, perhaps to detect collisions, determine if the game is over, etc.

```
loop while true:
      erase object
      object_x += 5
      draw object
      …other processing…
end loop
```

Constant Delays

The problem with the above code is that there is no delay between iterations so the object will move across the screen very, very fast – probably too fast to see. Perhaps we decide that the object should move every 10 milliseconds. (In this example we are moving an object around the screen, but these principles apply to any type of real-time recurring task). We could simply add a constant delay into the loop.

```
loop while true:
      erase object
      object_x += 5
      draw object
      …other processing…
      wait 10 milliseconds
end loop
```

The problem with this approach is that the delay between each iteration will be longer than 10 milliseconds, because it takes time to erase and redraw the object, and it takes time to do the other processing. One way to fix this would be to simply reduce the wait time by trial and error, until each loop iteration took about 10 milliseconds. The problem with this is the ‘other processing’. You can’t always rely upon it taking the same amount of time. The code might not be in your control – a group member could be responsible for the code and may forget to tell you it changed, or in industry it may be developed by someone you don’t even know. Even if you are the owner of the code – the execution time may change from one iteration to the next. For example, if you need to detect collisions with enemies, the processing time will vary depending on how many enemies are on the screen. Your character movement would be slowed down as more enemies appeared!

Timers

One good solution is to use a timer. Instead of stalling for a constant amount of time in each loop iteration, we wait until the timer has expired:

```
set timer period to 10 milliseconds
set time to continuous mode
start timer
loop while true:
     erase object
     object_x += 5
     draw object
     …other processing…
     wait until timer timeout
     reset timeout
end loop
```

In this example, we configure a timer to have a 10 millisecond period, and set it to be continuous. This means that after the timer expires it will immediately begin counting down again. After all of the processing in the loop is completed, the code waits for the timer to expire. Once it is expired you will need to reset the timeout bit, otherwise on the next iteration the timer will still appear to be expired. You can read about how to do this in Section 28 of the Embedded Peripherals IP User Guide (http://www.altera.com/literature/ug/ug_embedded_ip.pdf).

You will need to add an additional timer to your Qsys project, as the other timers in your project are probably already being used for the system clock and the timestamp timer. When adding the timer in Qsys, you can choose a 'full-featured' timer.

Multiple Tasks

The timer solution works great for moving the character around the screen, but things can get more complicated once you have multiple tasks to perform. Maybe you also need to move the enemies on the screen, but you want them to move slower than the character. You could just move the enemies a shorter distance than the character, perhaps move the character 2 pixels, and the enemies 1 pixel (or use decimals and round). This is probably the simplest solution for this application, but we will explore other methods as well, as they may be more useful for other applications.

Another method would be to move the character during every loop iteration, but only move the enemies every other iteration. This would make the enemies move at half the speed. If we wanted them only a little bit slower we could move the character every 2 iterations, and the enemies every 3 iterations. By using a counter in the loop and checking the modulus, we can create arbitrary ratios between different tasks.

Interrupts

Another solution is to use interrupts. An interrupt is a special signal sent by a hardware core to the processor to 'interrupt' it. Basically it is a request for the processor to stop what it is doing, and run a certain function instead. Once completed, the processor can resume what it was working on. The Interval Timer core is capable of generating interrupt requests to the processor (just make sure you connect the interrupt line in Qsys). The timer will generate an interrupt when the timer expires, and if you set it in continuous mode, it will generate an interrupt periodically. You can use this to run a specific function periodically.

Here are the steps of what you must do in software:

1. Register the interrupt. This creates a pairing between an interrupt line (the processor has many) and the function to execute (called the interrupt handler, or interrupt service routine). Use the `alt_irq_register()` function. You will need to provide the ID of your interrupt line (there will be a `<ip_core_name>_IRQ` constant in system.h), and a pointer to the interrupt handler function.

2. Configure your timer – set the period, whether it is continuous, etc. You will also need to configure the timer to produce interrupt requests. Again, check out the documentation for the timer (link above).

3. Enable the interrupt. This instructs the processor to begin responding to interrupts requests from a given interrupt line. Use the `alt_irq_enable()` function – you will need to provide the interrupt ID.

4. Start the timer!

This can be repeated for multiple timers, which will allow you to run multiple different functions at different periods.

IMPORTANT NOTE: When an interrupt occurs, the processor will immediately stop execution and run the interrupt handler function. It does not wait for the current function to finish, and it might even be interrupted mid instruction! For example if you have the line `x=x+1`, the processor may read `x` from memory, and then be interrupted before it has a chance to increment it and write it back to memory. If `x` is a global variable, and your interrupt function also modifies `x`, this can cause bugs that are VERY hard to track down. If you think you need to share global variables between your main code and your interrupt handler functions, talk to a TA first.