

Scheduler

Purpose

You will design a timer-driven cooperative scheduler.

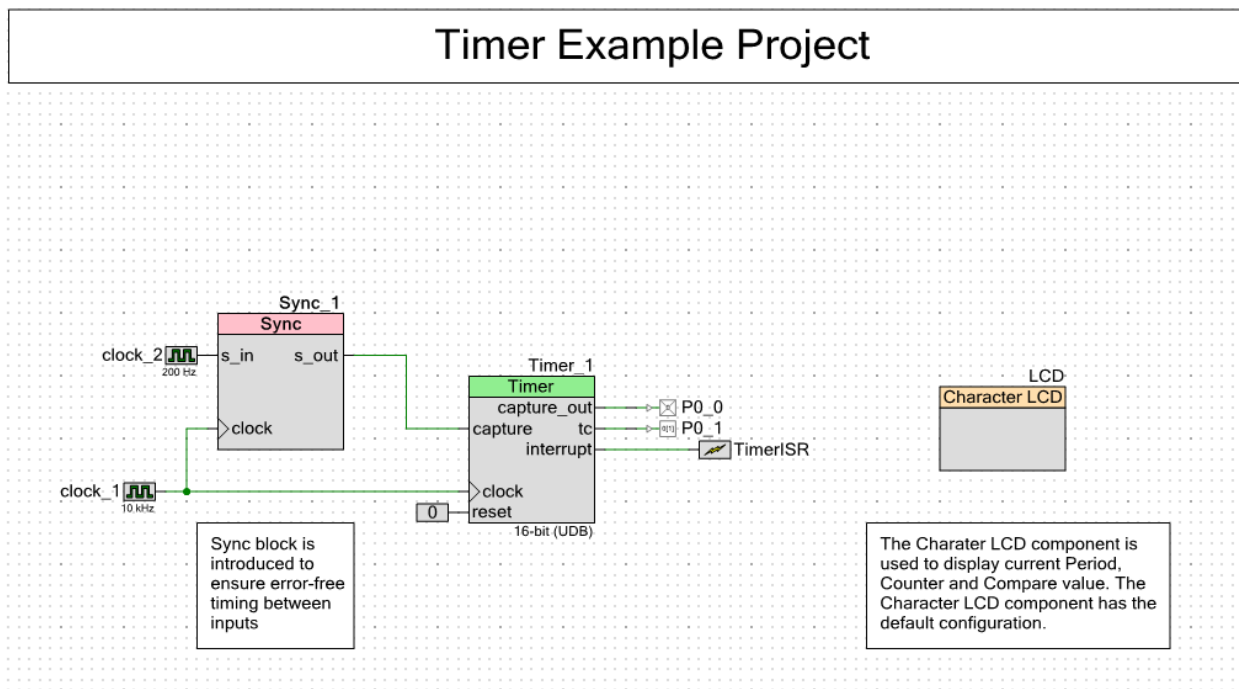
Setup

Open the Timer example project.

Procedure

We want to build a scheduler that runs a set of periodic tasks at the desired rates. This scheduler will only use the standard procedure call/return mechanism, which makes it much simpler to build. This scheduler cannot guard against execution time overruns by the tasks. You will run experiments to show the proper operation of the scheduler as well as what happens when a task overruns its time limit.

Copy the time01 project into your directory. Look at the .cysch file:



The timer's **capture_out** and **tc** outputs are sent to pins. The timer's interrupt signal is sent to an interrupt handler; the handler resets a status bit when the counter rolls over so that it continues to count.

Now look at **main.c**:

```

for(;;)
{
    /* To display Period on LCD */
    LCD_Position(0u, 7u);
    LCD_PrintInt16(Timer_1_ReadPeriod());

    /* To display Capture value on LCD */
    LCD_Position(0u, 12u);
    LCD_PrintInt16(Timer_1_ReadCapture());

    /* To display count on LCD */
    LCD_Position(1u, 0u);
    LCD_PrintInt16(Timer_1_ReadCounter());

    /* To display Interrupt count on LCD */
    LCD_Position(1u, 5u);
    LCD_PrintString("IntCnt:");
    LCD_PrintInt16(InterruptCnt);

    CyDelay(100u);
}

```

The example uses a loop to update the display but the timing of that loop isn't closely tied to the timer's period. The function `CyDelay(100u)` waits for the specified period (100 microseconds in this case) but that doesn't necessarily have anything to do with the timer state.

The timer's interrupt is handled by `CY_ISR(InterruptHandler) {}`. This routine must clear the timer's flag so that it will restart. While we could run the task code here, that code would then run inside the interrupt handler and with the interrupt priority of that handler, keeping lower-level interrupts from executing. We want to minimize the amount of time spent in any handler.

We are now ready to modify this example to create a timer-driven scheduler. We won't use the display in this example.

1. Change the timer's period to 5.

Q1: What is the timer's period in seconds?

2. Add these two C subroutines as our tasks:

```

void short_task() {
}

void long_task() {
    long i;
    for (i=0; i<99; i++);
}

```

3. Update the interrupt handler:

```

uint8 saw_tc = 0;

```

```

CY_ISR(InterruptHandler)
{
    Timer_1_STATUS;
    InterruptCnt++;
    saw_tc = 1;
}

```

The `saw_tc` variable allows the main routine to know when the counter has rolled over.

4. Finally, modify the main routine to execute the tasks once per timer period:

```

void main()
{
    uint8 toggleout = 0;
    /* Enable the global interrupt */
    CyGlobalIntEnable;

    /* Enable the Interrupt component connected to Timer interrupt */
    TimerISR_StartEx(InterruptHandler);

    /* Start the components */
    Timer_1_Start();

    scheduler_out_Write(toggleout);
    for(;;)
    {
        if (saw_tc) {
            saw_tc = 0; /* reset flag */
            /* toggle the state and send it to the pin */
            toggleout = ~toggleout;
            scheduler_out_Write(toggleout);
            /* run the tasks */
            short_task();
            long_task();
        }
    }
}

```

Run the task set and observe the output pin.

Q2: How often does `toggleout` change state?

Now that you have a scheduler, you can modify the timing of the tasks to determine how that timing affects the system schedule.

5. Modify the loop in `long_task()` to execute for more iterations. Increase the iteration count until you see the period of `toggleout` change.

Q3: How many iterations are executed in `long_task` when the period of `toggleout` is affected?

You Should Turn In

1. Answers to Q1, Q2, Q3.

As always, output snapshots are cool.