ECE 341 - Section 52

Process Speed Control Using Interrupts

Lab 5

Submitted By:
Tristan Denning

Contents

1 Introduction

The goal of this project is to implement an embedded system that runs entirely using foreground processes rather than background processes as done in previous projects. Foreground processes will be implemented through the use of Interrupts.

In prior projects, tasks were executed sequentially in the same order as written in code. In this project, the scheduling of tasks will be controlled by interrupts and their priority. Put simply, an interrupt is a signal triggered by some event. When an interrupt is set high, the associated Interrupt Service Routine preempts whatever next background (or lower priority) task would occur otherwise. There are two types of events that trigger interrupts: deterministic and sporadic. Deterministic interrupts are predictable, such as a timer flag being set. Sporadic interrupts are less predictable, such as the pressing of a button. In Project 5, the Change Notice interrupt will be invoked by the pressing of BTN2, and BTN3, and the Timer1 interrupt will be invoked every millisecond. Interrupts can also be invoked in software by setting the corresponding bit in the interrupt flag register.

Background tasks are those that reside in the while(1) loop, and have the lowest priority, 0. Foreground processes preempt background processes by means of interrupts that can be assigned ascending priorities 1-7. These are called Interrupt Service Routines (ISR). Within each priority group, 4 subgroup priorities exist. This way, an ISR with the same group priority as another could preempt the other if it had a higher subgroup priority. If multiple interrupts are triggered simultaneously, the associated ISRs will execute in descending order of their assigned priority.

Variables are neither passed to or returned from ISRs. Instead, ISRs manipulate data by changing global variables. Also, ISRs are not “called” like a typical C function. When a function is declared as an ISR, it cannot be called by any other C function.

Preemptive systems can be either nested or non-nested. In a non-nested system, all ISRs have a priority of 1, and there is no possibility of one ISR preempting another. The more widely used method is a nested system. In a nested system, ISRs are assigned differing priorities, and can preempt each other depending on their priority. Whenever an interrupt is triggered, the context (state) of the CPU is saved before the first task in the associated ISR. This process is called the prolog. The epilog restores the state saved during the prolog of the associated ISR, and occurs after the final task within the ISR. The interrupt flag for any ISR must be cleared before the epilog, or the ISR will repeat. The epilog and prolog are protected from interrupts. If an interrupt of higher priority occurs during either of these stages, the associated ISR will not execute until the prolog or epilog of the current ISR has completed.

In preemptive programming, some sections of code may need protection from being interrupted. These sections are called *critical sections*. Critical sections can be protected by bracketing them with the INTEnableInterrupts() and INTDisableInterrupts() functions. When the disable function is called, all Interrupts are essentially ignored until the enable function is called again.

2 Implementation

Before initializing resources, a few global variables need to be declared for the Interrupt Service Routines to communicate with. ISRs do not take arguments or return variables, so they can only manipulate data by use of global variables. For this project, the following global variables are created:

```
    unsigned int buttons, dir, mode, sm_code, step_delay;
    unsigned int step_ctrl = 1;
```

The first step in main is to initialize the required resources. The first function call in main is system_init(). Within the initialize function, buttons 1 and 2 are set as inputs, and the stepper motor coils and LEDs are set as outputs and cleared. Then, the initialize functions are called for the Change Notice and Timer1 interrupts. After the interrupts are initialized, the program is configured to allow nested interrupts with the INTEnableSystemMultiVectoredInt() function. Finally, the interrupts are enabled with the INTEnableInterrupts() function.

Listing 1 - Initialize Function

```
void system_init(void)
{
    Cerebot_mx7cK_setup(); // Initialize processor board
    PORTSetPinsDigitalOut(IOPORT_B, SM_LEDS | SM_COILS);
    PORTSetPinsDigitalIn(IOPORT_G, BTN1 | BTN2);
    LATBCLR = (SM_LEDS | SM_COILS);
    timer1_interrupt_initialize();
    cn_interrupt_initialize();

    INTEnableSystemMultiVectoredInt();    //done only once
    INTEnableInterrupts();                //done as needed

}
```

Within cn_interrupt_initialize(), the CN interrupt is enabled for BTN1 and BTN2 so that changing the state of either one of them will trigger the CN interrupt. Then the group and subgroup priorities are set to 1 and 0 respectively. A dummy variable is used to store the value of buttons 1 and 2. Then the CN interrupt flag is cleared and enabled.

Listing 2 - Change Notice Interrupt Initialization

```
void cn_interrupt_initialize(void)
{
  unsigned int dummy;     // used to store PORT value

  // Enable CN Interrupt for BTN1 and BTN2
  mCNOpen(CN_ON,(CN8_ENABLE | CN9_ENABLE), 0);

  // Set CN interrupts priority level 1 sub priority level 0
  mCNSetIntPriority(1);     // Group priority (1 to 7)
  mCNSetIntSubPriority(0);  // Subgroup priority (0 to 3)

  // read port to clear difference
  dummy = PORTReadBits(IOPORT_G, BTN1 | BTN2);
  mCNClearIntFlag();    // Clear CN interrupt flag
  mCNIntEnable(1);    // Enable CN interrupts
}
```

In timer1_interrupt_initialize(), Timer 1 is configured to use the internal clock with a 1:1 prescale and a PR1 value set for a 1 millisecond period. Then the group and subgroup priorities are set to 2 and 0 respectively. Finally Timer 1 interrupts are enabled with mT1IntEnable(1).

Listing 3 - Timer1 Interrupt Initialization

```
void timer1_interrupt_initialize(void)
{
    //configure Timer 1 with internal clock, 1:1 prescale, PR1 for
   1 ms period
    OpenTimer1(T1_ON | T1_SOURCE_INT | T1_PS_1_1, T1_INTR_RATE-1);

    // set up the timer interrupt with a priority of 2, sub
   priority 0
    mT1SetIntPriority(2); // Group priority range: 1 to 7
    mT1SetIntSubPriority(0); // Subgroup priority range: 0 to 3
    mT1IntEnable(1); // Enable T1 interrupts

}
```

With both interrupt sources initialized, the next step is to create each corresponding Interrupt Service Routine.

The CN ISR, when triggered, calls a button_debounce() function to wait for any button bouncing to settle. The button debounce function consists of the simple hardware-assisted software delay used in previous labs. It creates a delay period of 20 milliseconds that follows any state change of the buttons. Then the present state of the buttons are read and sent to the decode_buttons function used in previous projects. The decode_buttons() function updates the global variables for direction, mode and step delay for a given button state. Lastly, the CN interrupt flag must be cleared before exiting the ISR. LED C is flashed once for the duration of the CN ISR for testing and verification. See listing 4:

Listing 4 - CN Interrupt Service Routine

```
    unsigned int dummy;              //To hold port read value
    LATBSET = LEDC;                  //Flash LED C every button
   press
    button_debounce(20);             //WAIT 20 ms for BTN debounce
   period
    buttons = PORTG & (BTN1 | BTN2);
    decode_buttons(buttons, &step_delay, &dir, &mode);

    dummy = PORTReadBits(IOPORT_G, BTN1 | BTN2);

  /* Required to clear the interrupt flag in the ISR */
  mCNClearIntFlag();  // Macro function
    LATBCLR = LEDC;
```

The Timer1 ISR decrements and checks the step_ctrl global variable every 1 millisecond, as dictated in the Timer1 interrupt initialization. The step_ctrl variable is initialized to 1 so the ISR's tasks are first executed after the first millisecond of program run time. Within the step_ctrl check-loop, the stepper motor code is determined by the software FSM used in previous labs. This time, the FSM takes the values of the global variables for direction and mode changed in the CN ISR. Then the output function is called to send the global variable, sm_code, to the stepper motor just as in previous labs. Then the step_ctrl variable is set to the appropriate step_delay as determined in the decode_buttons() function. The last step in the ISR is to clear the Timer1 interrupt flag. LED A is toggled once at the beginning of the ISR for testing and verification purposes. See Listing 5:

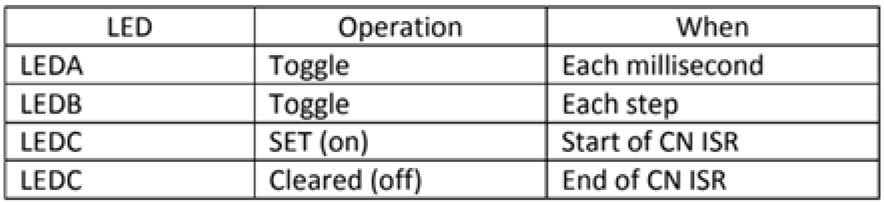
Listing 5 - Timer1 Interrupt Service Routine

```
void __ISR(_TIMER_1_VECTOR, IPL2) Timer1Handler(void)
{
  LATBINV = LEDA;      // toggle LEDA each millisecond
    step_ctrl--;      // decrement step_ctrl variable

    if (step_ctrl == 0)
    {
        sm_code = sw_fsm(dir, mode);
        output_sm_code(sm_code);
        step_ctrl = step_delay;
    }

    mT1ClearIntFlag();   // Macro function to clear the interrupt flag
}
```

3 Testing and Verification

LEDs A, B and C were toggled at various points in the program to indicate functionality. The following table describes when each LED is toggled.

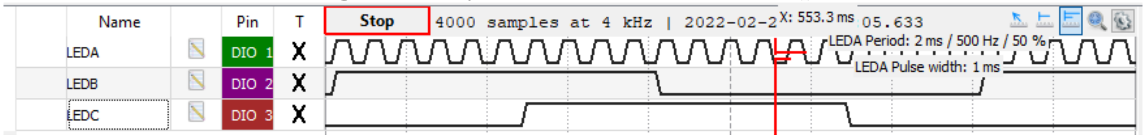
Instrumentation LED Operations



LED	Operation	When
LEDA	Toggle	Each millisecond
LEDB	Toggle	Each step
LEDC	SET (on)	Start of CN ISR
LEDC	Cleared (off)	End of CN ISR

LEDA should toggle every millisecond, indicating the Timer1 interrupt flag being set high. LEDB is toggled each step and should show the corresponding step delay. LEDC will turn on for a duration of at least 20 milliseconds (for button debounce) then turn off. The following is a screen capture of an oscilloscope reading the pins associated with LEDs A, B, and C.

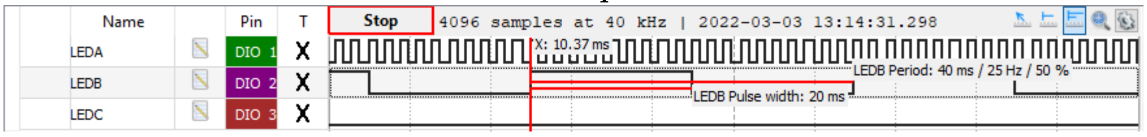
Logic Analyzer - Nested Interrupts



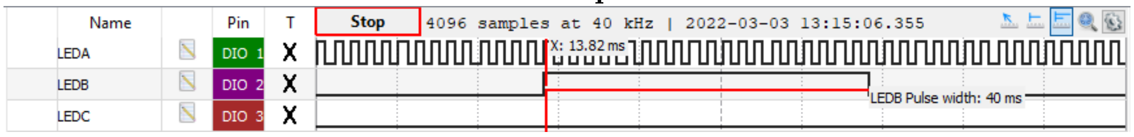
This demonstrates that the Timer1 ISR is indeed operating at a higher priority than the CN ISR. LEDC toggles, indicating that a button was changed, but the toggling period of LEDA is unaffected.

In addition to verifying priority levels, the delay period for each button state was also verified with the logic analyzer by measuring the period of LEDB. Each delay period was measured to be exactly what was expected for each of the button states 00, 01, 10, 11: 20, 40, 30 and 24 ms respectively.

Button Input 00


Name
Pin
T
Stop
4096 samples at 40 kHz | 2022-03-03 13:14:31.298
X: 10.37 ms
LEDA
LEDB
LEDC
DIO 1
DIO 2
DIO 3
LEDB Period: 40 ms / 25 Hz / 50 %
LEDB Pulse width: 20 ms

Button Input 01

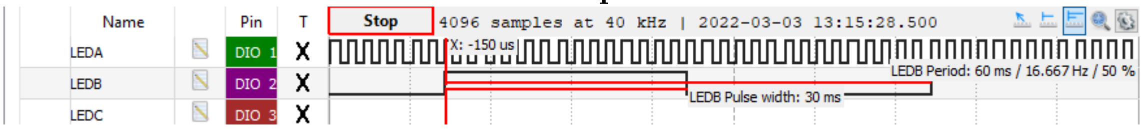


4096 samples at 40 kHz | 2022-03-03 13:15:06.355

X: 13.82 ms

LEDB Pulse width: 40 ms

Button Input 10



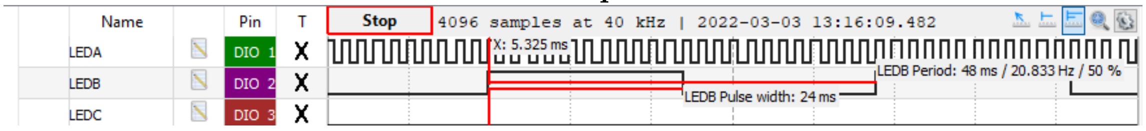
4096 samples at 40 kHz | 2022-03-03 13:15:28.500

X: -150 us

LEDB Period: 60 ms / 16.667 Hz / 50 %

LEDB Pulse width: 30 ms

Button Input 11



4096 samples at 40 kHz | 2022-03-03 13:16:09.482

X: 5.325 ms

LEDB Period: 48 ms / 20.833 Hz / 50 %

LEDB Pulse width: 24 ms

The stepper motor is running at more accurate speeds than in previous labs. This because the FSM and output functions reside in an ISR that preempts all other code. Therefore they can execute without waiting on other functions to execute. In previous labs, the read and decode functions were called between each step, adding unneeded latency between each step.

4 Conclusion

The objective of this lab was fulfilled. An interrupt-based scheduling construct was implemented to control a stepper motor. This lab demonstrated the usefulness of preemptive scheduling and its impact on timing accuracy.

5 Questions

1. Discuss the advantages and disadvantages of a real-time control system that uses polling versus an interrupt-based system. When might you choose one over the other?

   (a) Real-time control systems that rely on polling have the advantage of being simple, and require little memory to run effectively. Interrupt-based systems require more memory allocation to save the state of the CPU whenever an ISR is triggered. A disadvantage of a polling-system is excessive polling. This increases CPU usage and latency between other functions that may be time-sensitive. The Change Notice interrupt fixes this issue by setting its flag whenever the value of its associated pin(s) changes. For simple deterministic events, an interrupt-based control system may not be advantageous. Adding interrupts to deal with deterministic events may result in higher latency and slower program execution than if a simple polling scheme was used.

2. What is the worst-case latency for a system that uses polling? What about an interrupt-based system?

   (a) The worst-case latency for a system that uses polling depends on the overall number of sequential tasks that are executed. The worst-case latency for an interrupt-based system depends on the number of interrupts and how nested the interrupt scheme becomes. With 7 main priority groups and 4 subgroups, one ISR may be preempted by multiple other ISRs, which would cause significant latency for the initial ISR to complete.