ECE 341 - Section 52

Process Speed Control Using Interrupts

Lab 5 Prelab

Submitted By:
Tristan Denning

1 Introduction

1.1 Goal and Background Information

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.

2 Plan

In Richard Wall's description of the project, he lists the following program structure to be implemented:

1. **Main**
   (a) calls system_init()
   (b) runs the infinite while(1) loop
2. **system_init** function that implements a fully nested interrupt scheme for Timer 1 and change notice interrupts using the following steps.
   (a) run "Cerebot_mx7cK_setup()"
   (b) Initialize IO port B for LEDA, LEDB, LEDC, and stepper motor set as outputs.
   (c) Initialize Timer 1 to generate an interrupt once each ms . Set the group priority for level 2 and the subgroup level for 0.
   (d) Initialize the PIC32MX7 system for CHANGE NOTICE detection. Set change notice interrupts to detect activity on BTN1 and BTN2 only, the group priority level 1 and the subgroup level 0.
   (e) Set the system for multiple vectored interrupts.
3. **change_notice_ISR** (replaces "read_buttons" function in Project 4)
   (a) Set LEDC on at the beginning of the ISR and off at the end of the ISR
   (b) Delay for 20 ms to allow the buttons to settle, i.e., debounce period
   (c) Determine the button status. Then decode the button status.
   (d) Clear the CN interrupt flag. (CNIF)
4. **Timer1_ISR**
   (a) Set for a 1ms interval, toggling LEDA each interval as done in Lab 4.
   (b) Decrement step_delay
   (c) When the step delay is 0, call the FSM and output functions, then reset the step_delay variable to the global step_period variable.
   (d) clear the T1 interrupt flag (T1IF)
5. **decode_buttons** (Same as Project 4)

6. **stepper_state_machine** (Same as Project 4)
7. **output_to_stepper_motor** (Same as Project4)
8. **Button debounce period**
   (a) Implement a hardware assisted software delay using the core timer that can be preempted

To visualize the controls and data that the program will need to manipulate, the following Data and Control Flow Diagrams will be useful: (Please see attached)