Lecture 05.02 Interrupts

Embedded computing frequently requires a program to respond to events the timing of which is unkown, beforehand. These events include

- digital user input such as keypad and button presses;
- other digital input such as limit switch detection; and
- analog input such as sensor values.

One way to handle these types of events is to frequently poll the analog and digital inputs in a program's main loop. However, there are two drawbacks to this method: (1) events can be missed if the inputs are polled too infrequently, (2) it is difficult to control polling timing in a loop that contains other processes, and (3) the main loop's timing can be affected by the additional time it takes to handle the event.

The first and second concerns are mitigated by using another method: threaded interrupt handling. A new thread (in addition to the main thread) is created to handle an interrupt. This thread processes an *interrupt service routine* (ISR) which checks to see if there has been an *interrupt request* (IRQ) and, if so, responds to the event. The IRQ can be expressed in memory or externally on a *programmable interrupt controller* (PIC). In either case, the ISR frequently checks for the corresponding IRQ and, once serviced, clears it. We call an IRQ-ISR pair an *interrupt*.

ISR
IRQ
PIC
interrupts

Even when only a single core is available, interrupts can be given high-priority by the OS scheduler (via simultaneous multithreading) such that the program will be responsive to interrupts. Of course, unless the threads are run on distinct cores, an interrupt thread *does* add to the time of the iteration of the main loop (our third concern from above). For some applications (such as that of Lab Exercise 05), this additional time is negligible. But for many real-time applications, this will be problematic. Mitigation can be achieved by using a *timer interrupt*, which will be explored in Lab Exercise 06.

timer interrupt