# Interrupt Theory

*(Revised: 10/3/2012)*

## 1. Interrupts

An interrupt is an unexpected hardware initiated subroutine call or jump that temporarily suspends the running of the current program. Interrupts are used to ensure adequate service response times by the processing. Sometimes, with software polling routines, service times by the processor cannot be guaranteed, and data may be lost. The use of interrupts guarantees that the processor will service the request within a specified time period, reducing the likelihood of lost data.

Interrupts occur when a peripheral device asserts an interrupt flag that is sampled each machine cycle. Since the Microchip MIPS executes an instruction each core processor clock cycle, the processor can immediately branch to a special interrupt service routine (ISR) written to handle that particular interrupt. It is the responsibility of the ISR to save the context of the processor core registers prior to making any changes to these registers. In addition to the core registers, the address of the next instruction that would have been executed if the interrupt had not occurred must be saved. The entire processor context is saved in stack space of memory. Upon executing the code required to service the interrupt generating event, the processor must be prepared to return to what it was doing previously. This requires that the context be restored from the stack before returning to the next instruction in the preempted code.

Interrupts fall into two major categories, internally generated and externally generated. Internally generated interrupts are in response to processor resources such as a timer flag. There are two types of externally generated interrupts: direct and indirect. An interrupt generated in response to a completed serial port transmission and received data ready are examples of indirect external interrupts. There are four pins on the PIC32 that can be used to generate an interrupt in direct response to an input signal. These inputs can be programmed to generate an interrupt on a rising (logic 0 to logic 1) or falling (logic 1 to logic 0) signal edge. In addition, 21 IO pins can be used for a CHANGE NOTICE interrupts that are generated whenever there is a change of logic level on any one of the configured pins. There are up to 21 possible pins that can be used to generate a CN interrupt. (See Table 1.1 of the PIC32MX5XX/6XX/7XX user’s guide.)

## 2. Vectored Interrupts

Vectored interrupts direct the computer to make an unconditional subroutine call to a fixed memory address. It is expected that data programmed into memory at that address and subsequent locations will either perform the instructions to service the interrupt or execute code that will cause the processor to jump to a new address in memory to execute code to service the interrupt. Refer to Chapter 5 of the course text for additional information on the PIC32 Interrupt vector tables. Also suggested reading is Section 8 of the Pic32 Reference Manual that covers interrupts.

### A. Context Switching

The interrupt handler routine must generate a prologue and an epilogue to properly configure, save and restore all of the core registers, along with General Purpose Registers. At a worst case, all of the modifiable General Purpose Registers must be saved and restored by the prologue and epilogue.
Example 8-8: Prologue Without a Dedicated General Purpose Register Set in Assembly Code

```
rdpgpr sp, sp
mfc0 k0, Cause
mfc0 k1, EPC
ori k0, k0, 0xa
addiu sp, sp, -76
sw k1, 0(sp)
mfc0 k1, Status
sw k1, 4(sp)
ins k1, k0, 10, 6
ins k1, zero, 1, 4
mtc0 k1, Status
sw a8, 8(sp)
sw a0, 12(sp)
sw a1, 16(sp)
sw a2, 20(sp)
sw a3, 24(sp)
sw v0, 28(sp)
sw v1, 32(sp)
sw t0, 36(sp)
sw t1, 40(sp)
sw t2, 44(sp)
sw t3, 48(sp)
sw t4, 52(sp)
sw t5, 56(sp)
sw t6, 60(sp)
sw t7, 64(sp)
sw t8, 68(sp)
sw t9, 72(sp)
addu a8, sp, zero

// start interrupt handler code here
```

Listing 1 Prolog context saving.

Example 8-10: Epilogue Without a Dedicated General Purpose Register Set in Assembly Code

```
addu sp, a8, zero
lw t9, 72(sp)
lw t8, 68(sp)
lw t7, 64(sp)
lw t6, 60(sp)
lw t5, 56(sp)
lw t4, 52(sp)
lw t3, 48(sp)
lw t2, 44(sp)
lw t1, 40(sp)
lw t0, 36(sp)
lw v1, 32(sp)
lw v0, 28(sp)
lw a3, 24(sp)
lw a2, 20(sp)
lw a1, 16(sp)
lw a0, 12(sp)
lw a8, 8(sp)
di
lw k0, 0(sp)
mtc0 k0, EPC
lw k0, 4(sp)
mtc0 k0, Status
```

Listing 2. Epilog context restoration
3. Interrupt Limitations

Interrupts have four attributes: priority, latency, frequency, and persistency. These attributes are discussed in detail in the following sections. Interrupts can also be classified by their pattern of occurrence: either aperiodic (sporadic) or periodic. The occurrences of sporadic interrupts are not predictable.

4. Signal Characteristics

A. Magnitude

All signals used must conditioned to meet the processor input electrical specifications. Pins that are designated as “5 volt tolerant” can tolerate inputs up to 5 volts with doing damage to the processor. Pins that can also be used as analog inputs are not 5 volt tolerant. Voltages above 0.7*VDD are sensed as a logic one by the processor. Signals below 0.45 volts are sensed as a logic zero. Pins associated with a black box in the figure below are 5 volt tolerant.

![Figure 1. PIC32 64 pin package pin assignments](image-url)
B. Latency

The time interval from when the interrupt is first asserted to the time the CPU recognizes it. This will depend much upon whether interrupts are disabled, prioritized and what the processor is currently executing. At times, a processor might ignore requests whilst executing an instruction stream (read-write-modify cycle). The figure that matters most is the longest possible interrupt latency time.

*Interrupt latency* refers to the time required for an interrupt to take place after it has been requested. Generally, interrupts of the same priority are disabled when an interrupt service routine is entered. Sometimes interrupts must stay disabled until the interrupt service routine is completed, other times the interrupts can be re-enabled once the interrupt service routine has at least disabled its own cause of interrupt. In any case, if several interrupt routines are operating at the same priority, this introduces interrupt latency while the next routine is waiting for the previous routine to allow more interrupts to take place. If a number of devices have interrupt service routines, and all interrupts are of the same priority, then pending interrupts can not take place until at least the interrupt service routine in progress is finished, or at least until it changes the interrupt priority. As a rule of thumb, Z-World usually suggests that 100 µs be allowed for interrupt latency on Z180- or Rabbit-based controllers. This can result if, for example, there are five active interrupt routines, and each turns off the interrupts for at most 20 µs.

Figure 2 shows the components that make up latency computations. Before a processor begins the interrupt process, it must first complete the machine instruction that it is currently processing. On some processors with instruction queues, the next instruction in the process being interrupted must also be executed. The process of saving the current context is also associated with latency. Latency can be extended if additional code is required to implement nested interrupt schemes as well as additional time to complete lower priority interrupts in non-nested interrupt schemes. See *Interrupt Management* below.

![Diagram of interrupt latency](image)

**Figure 2. Breakdown of interrupt latency**

C. Frequency

Frequency is a measure of the rate that periodic interrupts occur or the inverse of the shortest interval between sporadic interrupts. Three things limit the frequency of interrupt signals: the speed of the processor, the execution time of the ISR, and the worst case latency for a particular ISR. The factors that influence latency are discussed in *Interrupt Management* section of this chapter.

D. Persistence

Persistence is the duration of the interrupting signal. The required persistence when polling for signals to trigger event is the processor execution time of all code between two successive polling operations. Signals whose persistence spans multiple polling operations may falsely generate multiple events unless mitigated using software or external hardware.

Since most processors latch external interrupts using flip-flops internal to the processor the required persistence for these interrupts is generally one or two instructions cycles. When interrupt signals are latched, one must consider how the flip-flops are reset so to be able to capture the next
signal trigger. As discussed in previous sections of this chapter, some internally generated interrupts are cleared only when a specific register is read. The external interrupts and certain internal interrupts are automatically cleared when the ISR is serviced.

5. Protected Code

Controlling the stepper motor requires that the outputs PG0 through PG3 be modified each step. Since only four of the eight output pins are needed for stepper motor control, the other four pins are available for additional controls or indications. In Lab 4, you are required to toggle PG6 and PG7 so you can observe the real-time operations of the stepper motor speed controls. This means that some pins of the same IO port could change at different times. Fortunately, for the students, in the Lab 4 experiment, all events are synchronized. However that is not always the case and the states of some pins must be unaltered while modifying other pins.

Consider the read-modify-write sequence represented by the three lines of code in Listing 3 that performs the read-modify-write operation to change the stepper motor outputs. For the case of sporadic interrupts or other timer interrupts run a rate that is not a multiple of the first timer interrupt and it too alters Port G. For the later case, eventually the second timer interrupt will occur between lines two and three of the code. When (and not if) this happens, the data that is written back to port G and the shadow register is outdated because it was modified by the interrupt after it was read in the second line.

Listing 3. Protecting read-modify-write sequences

```c
#define STEPPER_MASK (BIT_7 | BIT_8 | BIT_9 | BIT_10)
int stepper_output;

// Following two lines must be protected by disabling interrupts
// only is there is a possibility of an interrupt occurring while
// executing these instructions
stepper_output = (LATB & STEPPER_MASK); // Read statement
LATB = stepper_output | (stepper_code << 7); // Modify and write

// Enable interrupts
```

We must prevent one interrupt function from making changes to global variable or I/O ports that are being modified by the function that was interrupted. This requires that the three lines of code written above must be protected from interrupts. The easiest way to prevent interrupts that will protect a segment of code is to promote the processor operating level to a level that is higher than the interrupt that could cause a problem. The following table lists the effects of setting the filter for the four levels of priority.

6. Interrupt Management

To this point we have discussed the four attributes of interrupts: priority frequency, persistence, and latency. How interrupts are be managed can result in very different latency. Fundamentally, there are two types of management schemes: fully nested and non-nested. Non-nested interrupt schemes are usually the result of systems with single level software priorities and multi level hardware priorities.

A. Non-nested Priority Management Systems

Figure 3 illustrates that low priority interrupts can impose large latencies on high priority interrupts. Interrupt T1 shown in Figure 3 is the lowest priority interrupt and by virtue of when it occurs, it has started to run before interrupt 2 occurs. While interrupt T1 is running, interrupt T3. Since T3 is the highest standing interrupt when interrupt T1 finishes, T3 will run next. While T2 is running, interrupt T4 occurs and will begin running when T3 has finished. After T4 is completed, interrupt T2 that occurred shortly after T1 started running, will finally get a chance to run. The latency associated with T2 can be as long as the sum of the run times for all higher interrupts plus the longest run time of an interrupt with a lower priority.
The latency for an interrupt of any priority can be computed using the following expression:

\[
T_{LAT} = T_{LI} + T_{SC} + \text{MAX}(T_{SC-LP} + T_{EX-LP} + T_{RC-LP}) + \\
\sum(T_{SC-SSP-HHP} + T_{EX-SP-HHP} + T_{RC-SSP-HHP})
\]  

(1)

where,

\(T_{LI} = \text{Execution time of the longest executing assembler instruction}\)

\(T_{SC} = \text{Time to save context of the interrupt for which the latency is being calculated}\)

\(T_{SC-LP} + T_{EX-LP} + T_{RC-LP} = \text{Time to save context of interrupts plus execution time plus time to restore context of lower software priority}\)

\(T_{SC-SSP-HHP} + T_{EX-SP-HHP} + T_{RC-SSP-HHP} = \text{Execution time plus time to save context plus time to restore context of interrupts of same software priority but higher hardware priority}\)

B. Nested Priority Management Operations

Nesting interrupts schemes as illustrated in Figure 4 is where some interrupts are allowed to preempt interrupts that have a lower priority. Although this scheme results in shortened worse case latency, the execution time of low priority interrupts can be extended to include the sum of all higher priority interrupts. This is the case for interrupts T1 through T3 illustrated in Figure 4. The problem can be further exacerbated if the frequency of high priority interrupts causes them to run multiple times before the low priority interrupt is able to finish.
Figure 4, Nested interrupt management scheme

The latency for an interrupt of any priority for the fully nested scheme can be computed using the expression shown in equation 2:

\[
T_{LAT} = T_{LI} + T_{SC} + \text{MAX} \left( T_{SC-SSP} + T_{SSP-EXC} + T_{RC-SSP} \right) + \\
\text{MAX} \left( T_{SC-HHP} + T_{SP-HHP} + T_{RC-HHP} \right) + \\
\sum \left( T_{SC-HSP} + T_{EX-HSP} + T_{RC-HSP} \right)
\]  

(2)

where,

\( T_{LI} \) = Execution time of the longest executing assembler instruction

\( T_{SC} \) = Time to save context of the interrupt for which the latency is being calculated

\( T_{SC-SSP} + T_{EX-SCP} + T_{RC-SSP} \) = Execution time of plus time to save context of interrupts of same software priority that is currently executing

\( T_{SC-HHP} + T_{EX-HHP} + T_{RC-HHP} \) = Execution time of plus time to save context of interrupts of same software priority but higher hardware priority

\( T_{SC-HSP} + T_{EX-HSP} + T_{RC-HSP} \) = Execution time of plus time to save and restore context of interrupts of higher software priority.

On some processors, additional code is required to enable interrupts while executing an interrupt. The time to execute this additional code maybe considered part of the interrupt latency in some text books.
Figure 5. Latency details for nested interrupts