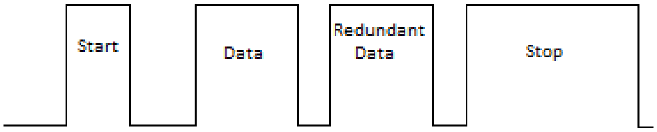
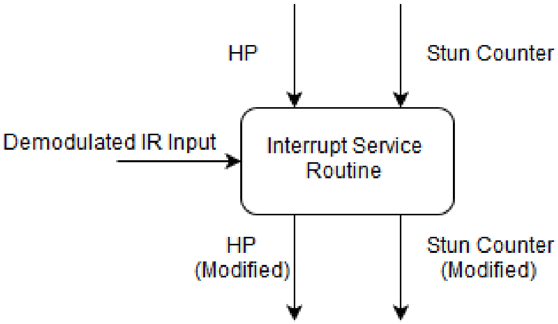
Infrared Data Reception for the Alpha, Delta, and Gamma Blades

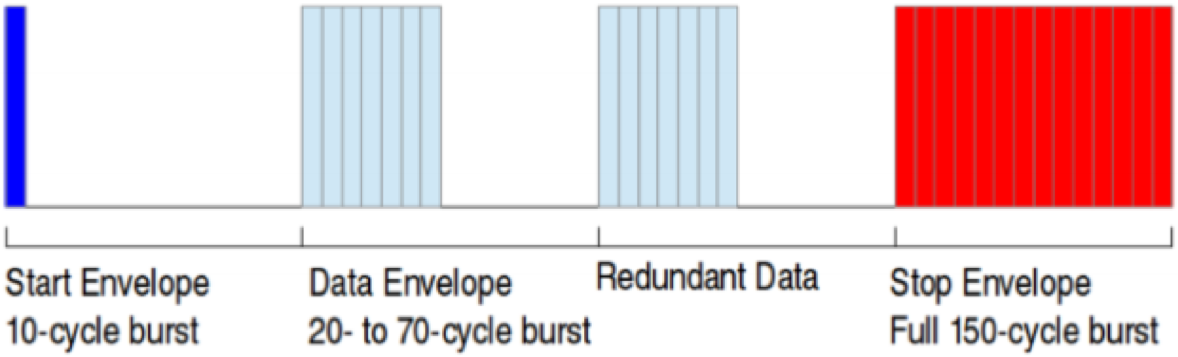
Christian Coffield
ECEN4013
September 27, 2015





Introduction

The Omega Blade, comprised of four individual sword (Alpha, Beta, Delta, and Gamma) is an object in the M.A.G.E. (Mobile Active Gaming Environment) gaming system. As such, it complies with the associated infrared protocol, M.I.R.P. (M.A.G.E. Infra-Red Protocol). This protocol states that objects involved in the game must be capable of receiving and transmitting data in the form of IR packets transmitted at a frequency of 56kHz, and process the data received into commands that affect the object's status. Data is transmitted in a packet as follows:



The three types of packets that are currently implemented into M.I.R.P. are damage, healing, and stun. These packets are sorted by the length of their data envelope, as shown in the table below.

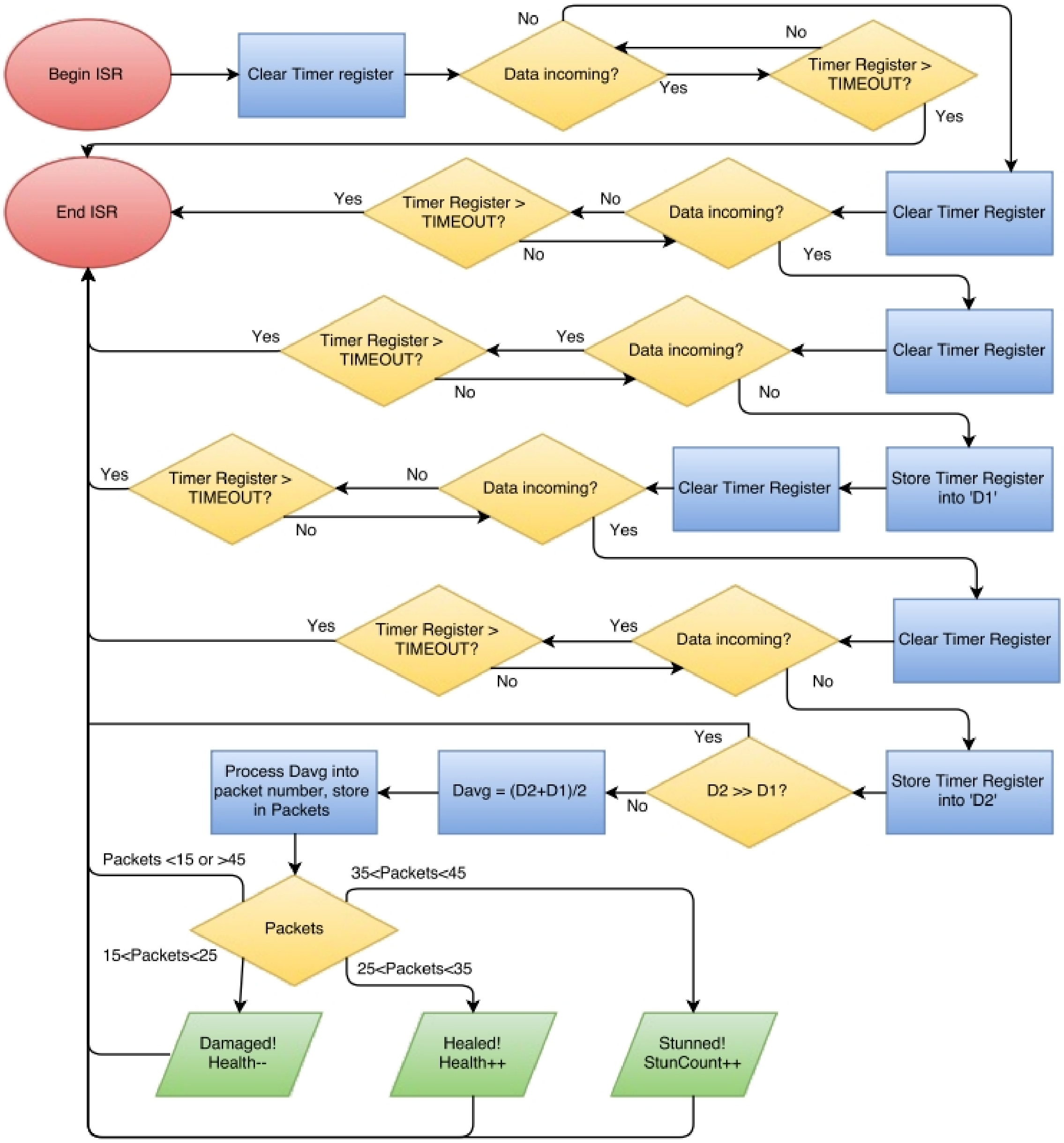
Number of On Cycles	Type of packet	Description
20	Damage	Reduce the object's health by 1 per packet.
30	Healing	Increase the object's health by 1 per packet.
40	Stun	Prevent the device from outputting damage, for 100ms per packet.

There are a few ways to accept the data, but the two most prominent in the microcontroller world are interrupts and polling.

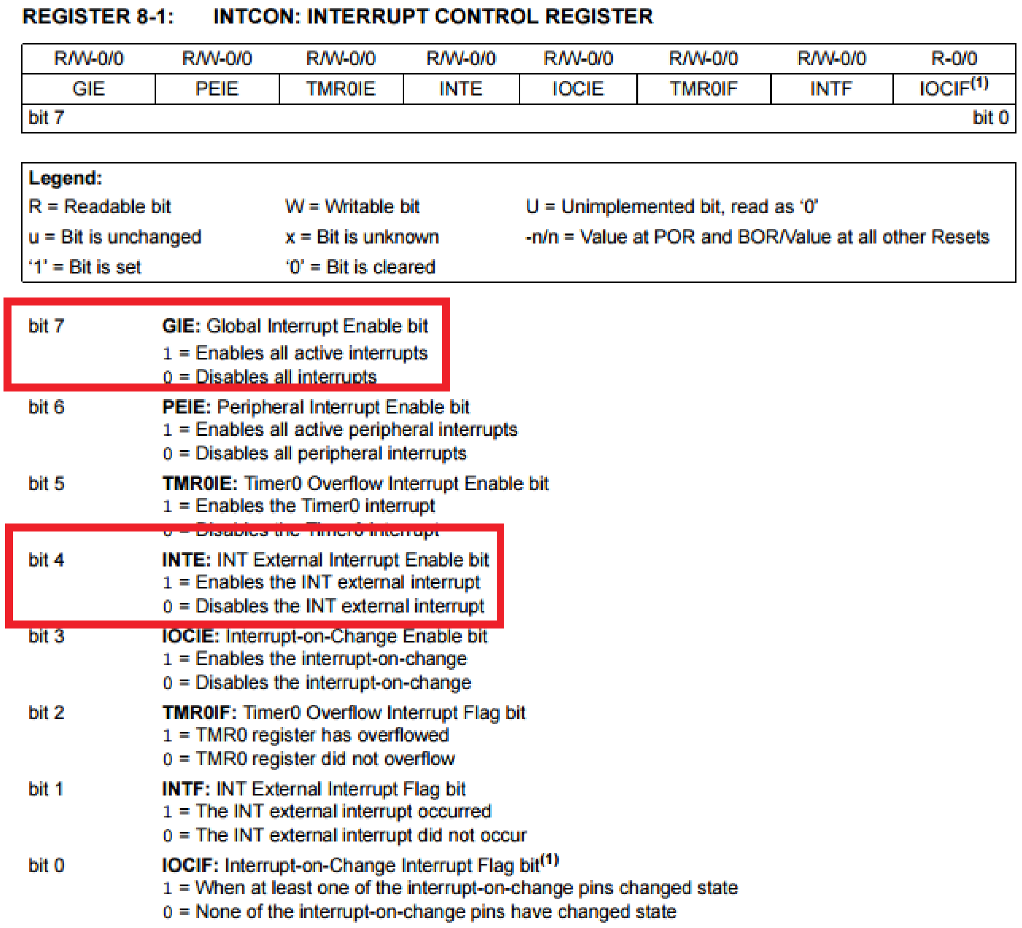
Interrupts

The method of choice for accepting data that does not occur at a defined interval is interrupts. Interrupts serve as a way to temporarily cease normal operation to process a high-priority input. An example of such would be stopping an infinite loop that drives a robot forward if the robot bumps into something, at which point the robot would stop, turn, and resume driving forward.

Because of the random nature of the data received in MAGE, interrupts are capable of pausing the standard operation of listening for the user to put out data (by swinging their sword in this case) if they take a hit from another player, in order to process what kind of hit they took. This would work similarly to the following flowchart, activated at any point in the main program, when data is received.



A single pin on the microcontroller would be set up as an interrupt pin, using the microcontroller’s INTCON register to allow for external interrupts and the IOCxP/IOCxN registers for whichever port the interrupt is to be set on to listen for positive and negative edge switches. Any pin on the primary microcontroller selection is configurable as an external interrupt-on-change pin, and nearly all microcontrollers have some form of external interrupt.



REGISTER 8-1: INTCON: INTERRUPT CONTROL REGISTER

R/W-0/0	R/W-0/0	R/W-0/0	R/W-0/0	R/W-0/0	R/W-0/0	R/W-0/0	R-0/0
GIE	PEIE	TMR0IE	INTE	IOCIE	TMR0IF	INTF	IOCIF[1]
bit 7							bit 0

Legend:		
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'
u = Bit is unchanged	x = Bit is unknown	-n/n = Value at POR and BOR/Value at all other Resets
'1' = Bit is set	'0' = Bit is cleared	

bit 7 **GIE:** Global Interrupt Enable bit
1 = Enables all active interrupts
0 = Disables all interrupts

bit 6 **PEIE:** Peripheral Interrupt Enable bit
1 = Enables all active peripheral interrupts
0 = Disables all peripheral interrupts

bit 5 **TMR0IE:** Timer0 Overflow Interrupt Enable bit
1 = Enables the Timer0 interrupt
0 = Disables the Timer0 interrupt

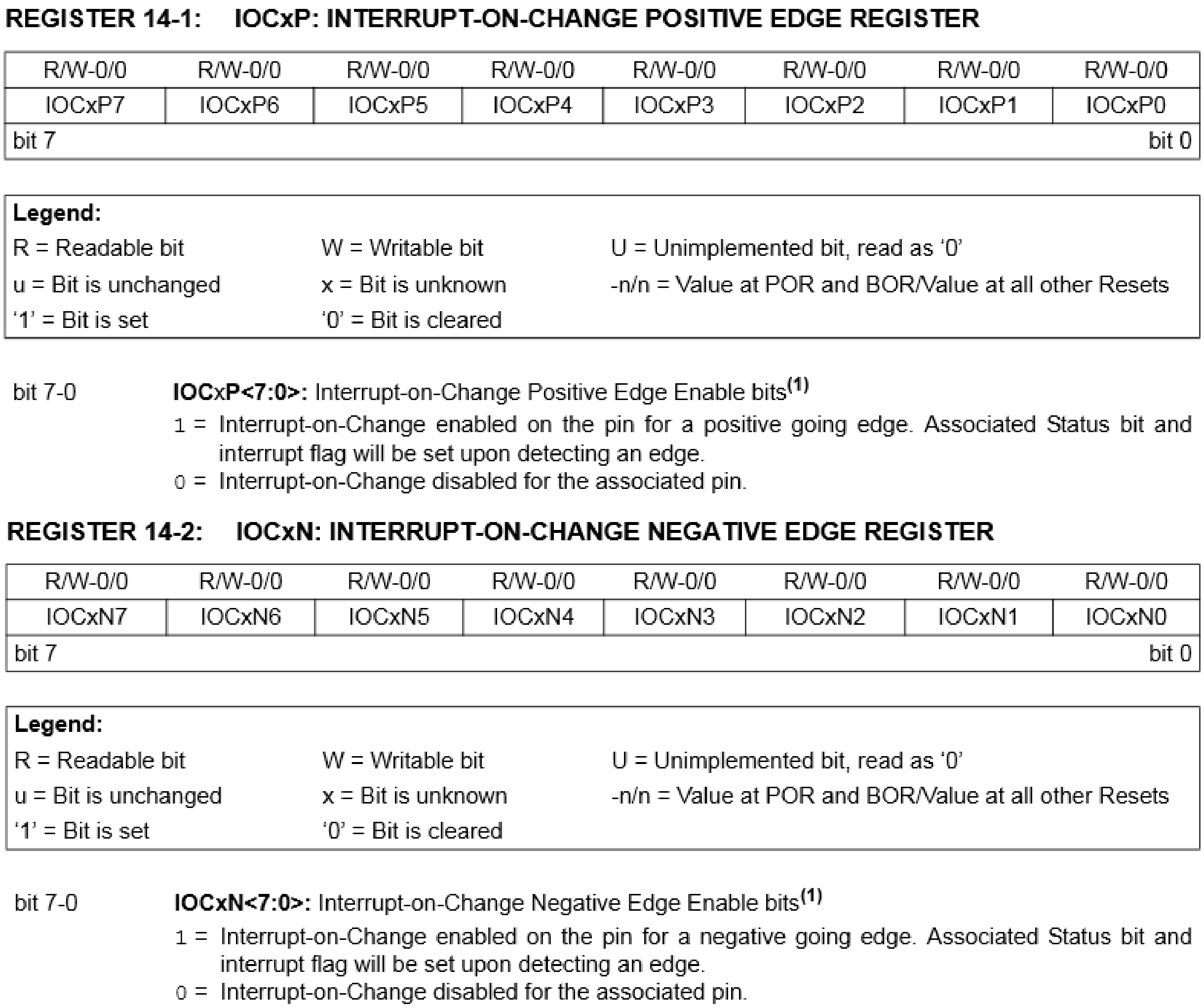
bit 4 **INTE:** INT External Interrupt Enable bit
1 = Enables the INT external interrupt
0 = Disables the INT external interrupt

bit 3 **IOCIE:** Interrupt-on-Change Enable bit
1 = Enables the interrupt-on-change
0 = Disables the interrupt-on-change

bit 2 **TMR0IF:** Timer0 Overflow Interrupt Flag bit
1 = TMR0 register has overflowed
0 = TMR0 register did not overflow

bit 1 **INTF:** INT External Interrupt Flag bit
1 = The INT external interrupt occurred
0 = The INT external interrupt did not occur

bit 0 **IOCIF:** Interrupt-on-Change Interrupt Flag bit[1]
1 = When at least one of the interrupt-on-change pins changed state
0 = None of the interrupt-on-change pins have changed state



REGISTER 14-1: IOCxP: INTERRUPT-ON-CHANGE POSITIVE EDGE REGISTER

R/W-0/0	R/W-0/0	R/W-0/0	R/W-0/0	R/W-0/0	R/W-0/0	R/W-0/0	R/W-0/0
IOCxP7	IOCxP6	IOCxP5	IOCxP4	IOCxP3	IOCxP2	IOCxP1	IOCxP0
bit 7							bit 0

Legend:		
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as ‘0’
u = Bit is unchanged	x = Bit is unknown	-n/n = Value at POR and BOR/Value at all other Resets
‘1’ = Bit is set	‘0’ = Bit is cleared	

bit 7-0 **IOCxP<7:0>:** Interrupt-on-Change Positive Edge Enable bits[1]

1 = Interrupt-on-Change enabled on the pin for a positive going edge. Associated Status bit and interrupt flag will be set upon detecting an edge.

0 = Interrupt-on-Change disabled for the associated pin.

REGISTER 14-2: IOCxN: INTERRUPT-ON-CHANGE NEGATIVE EDGE REGISTER

R/W-0/0	R/W-0/0	R/W-0/0	R/W-0/0	R/W-0/0	R/W-0/0	R/W-0/0	R/W-0/0
IOCxN7	IOCxN6	IOCxN5	IOCxN4	IOCxN3	IOCxN2	IOCxN1	IOCxN0
bit 7							bit 0

Legend:		
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as ‘0’
u = Bit is unchanged	x = Bit is unknown	-n/n = Value at POR and BOR/Value at all other Resets
‘1’ = Bit is set	‘0’ = Bit is cleared	

bit 7-0 **IOCxN<7:0>:** Interrupt-on-Change Negative Edge Enable bits[1]

1 = Interrupt-on-Change enabled on the pin for a negative going edge. Associated Status bit and interrupt flag will be set upon detecting an edge.

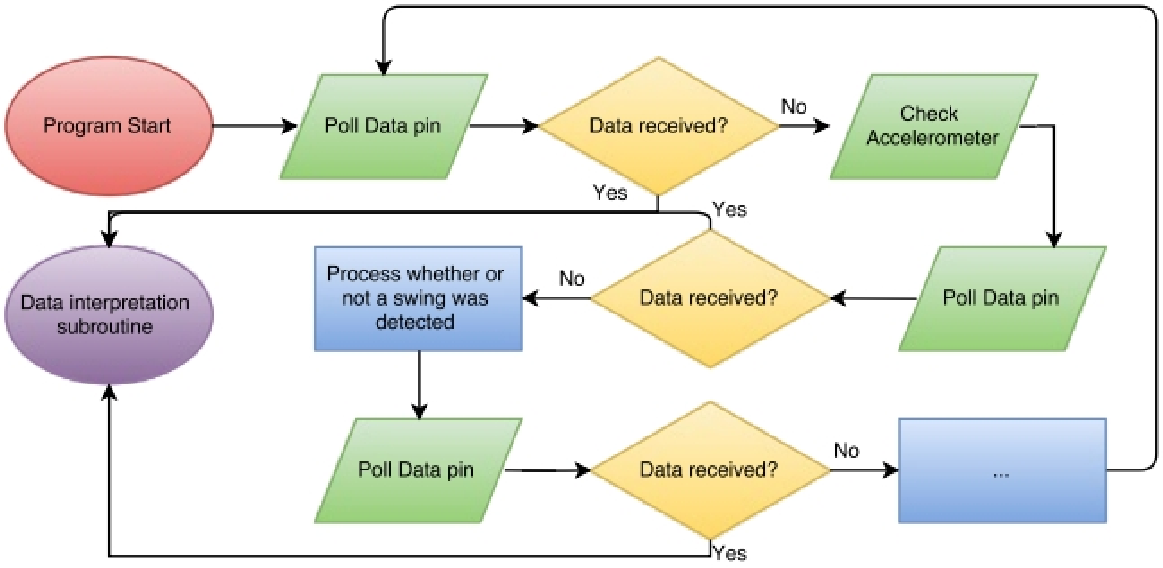
0 = Interrupt-on-Change disabled for the associated pin.

Upon interrupt, the associated interrupt service routine will listen for a complete information packet and record the length of pulses appropriately, in number of clock cycles. If at any point the ISR does not detect a complete packet, including start, data, redundant data, and stop, it will discard the packet as incomplete. Otherwise, the ISR records the lengths of the two data pulses. If one is substantially longer than the other, the ISR will determine the data as inconsistent and discard it. Otherwise, it will average the data and redundant data widths and then convert them from clock cycles into on-pulses, and sort the averaged number of on pulses into a type of packet. From there it increments or decrements an appropriate variable, and then exits the ISR.

Polling

An alternative to interrupt-based data interpretation is polling. Polling is in essence pausing the main program intermittently to check a pin for data. While it is easier to implement than interrupts on a device where timing is of the essence, it is less effective when data can come in a very short length and without any reliable timing.

Because of how short a complete packet in MIRP is (2678.5 microseconds at the maximum, or 2.7 milliseconds), a polling would need to occur very frequently. Microcontrollers at present tend to clock in at a maximum of 32MHz. This results in a 125 nanosecond instruction cycle. It’s more than fast enough to catch data with little to no loss, assuming one polls frequently. T main loop would execute something like the following.



While the above graphic is a minor exaggeration, because the microprocessor can run at many, many times the speed at which the data pin must be polled (32MHz max compared to 56kHz max, with the sampling theorem requiring at least a 112kHz sample rate) it does provide insight on a major dilemma in terms of stuns. While stunned, polling cannot check the data pin; however, even during a blocking loop to stun the user, an interrupt can allow for more stun packets, damage, or health to get through. Generally, polling is an outdated programming practice.

Discussion

While both polling and interrupts are viable methods of receiving data, interrupts are the clear winner of the two. Because the user can never tell when they will be receiving data, polling it at a defined interval is impossible; that means that the data input would have to be polled constantly at an interval for which no data could be missed. This interval is such a small length of time (if we assume the most pulses that could ever be missed is 9, which would be all but one of the start packet, is just shy of 170 microseconds) that a polling command would have to execute every few lines of code.

With that in mind, interrupts are not specific to any timing-sensitive application. At virtually the instant data is received, the microcontroller can stop its normal operation and process what it was given. This is convenient, because it means fewer clock cycles go to waste with checking a pin for activity, and there is much lower probability of missing an input.

The primary concern with using interrupts is the risky nature of modifying and accessing variables mid-interrupt service routine. Because the main program can be cut off at any time, and that includes while using a variable that is altered by an ISR, there is a significant risk of corruption. This is most substantial when modifying a variable in both the main program and the ISR. As a way to safeguard data corruption, Priyadeep Kaur of Cypress Semiconductor [5] recommends reducing modification of a variable to as few subroutines as possible. With that in mind, a proposed solution is to only modify variables in the ISR whenever possible. Thus, only incrementing and decrementing the HP counter in the ISR is preferred. The only point of contention is the stun counter, which must be handled outside of the ISR to allow for more data to come through but incremented inside of the ISR when data is received. However, this risk can be alleviated by disabling interrupts in the main program any time the stun counter is accessed and modified, and then re-enabling them whenever finished. This prevents an ISR from modifying a variable for the very short duration for which it is needed in the main loop.

The most commonly used language for the microcontroller programming world is C. It is so widely used because of a few factors, including accessibility and hardware independence. By far the most compelling reason to use it, however, and the reason it is the language of choice for accepting data on the Omega Blade, is its compliance with low level control. Because interrupt and pin-polling both require access to registers on a very low level, something that can easily modify registers is required. C is also capable of higher level computing and method calls, which is a key component in integrating the various segments of the Omega Blade project.