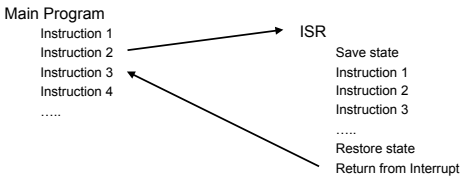
Interrupts

- Fundamental concept in computation
- Interrupt execution of a program to "handle" an event
  - Don't have to rely on program relinquishing control
  - Can code program without worrying about others
- Issues
  - What can interrupt and when?
  - Where is the code that knows what to do?
  - How long does it take to handle interruption?
  - Can an interruption be, in turn, interrupted?
  - How does the interrupt handling code communicate its results?
  - How is data shared between interrupt handlers and programs?

What is an Interrupt?

- Reaction to something in I/O (human, comm link)
- Usually asynchronous to processor activities
- "interrupt handler" or "interrupt service routine" (ISR) invoked to take care of condition causing interrupt
  - Change value of internal variable (count)
  - Read a data value (sensor, receive)
  - Write a data value (actuator, send)



Interrupts

- Code sample that does not interrupt

```
char SPI_SlaveReceive(void)
{
/* Wait for reception complete */
while(!(SPSR & (1<<SPIF)))
;
/* Return data register */
return SPDR;
}
```

- Instead of busy waiting until a byte is received the processor can generate an interrupt when it sets SPIF

```
SIGNAL(SIG_SPI) {
    RX_Byte = SPDR
}
```


Saving and Restoring Context

- Processor and compiler dependent
- Where to find ISR code?
  - Different interrupts have separate ISRs
- Who does dispatching?
  - Direct
    - Different address for each interrupt type
    - Supported directly by processor architecture
  - Indirect
    - One top-level ISR
    - Switch statement on interrupt type
  - A mix of these two extremes?

Saving and Restoring Context

- How much context to save?
  - Registers, flags, program counter, etc.
  - Save all or part?
  - Agreement needed between ISR and program
- Where should it be saved?
  - Stack, special memory locations, shadow registers, etc.
  - How much room will be needed on the stack?
  - Nested interrupts may make stack reach its limit – what then?
- Restore context when ISR completes

Ignoring Interrupts

- Can interrupts be ignored?
  - It depends on the cause of the interrupt
  - No, for nuclear power plant temperature warning
  - Yes, for keypad on cell phone (human timescale is long)
- When servicing another interrupt
  - Ignore others until done
  - Can't take too long – keep ISRs as short as possible
    - Just do a quick count, or read, or write – not a long computation
- Interrupt disabling
  - Will ignored interrupt "stick"?
    - Rising edge sets a flip-flop
  - Or will it be gone when you get to it?
    - Level changes again and its as if it never happened
  - Don't forget to re-enable

Prioritizing Interrupts

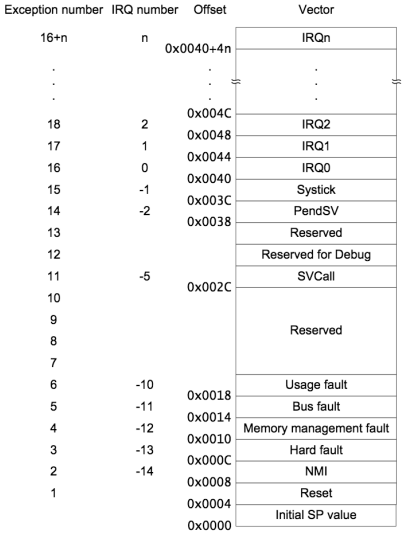
- When multiple interrupts happen simultaneously
  - Which is serviced first?
  - Fixed or flexible priority?
- Priority interrupts
  - Higher priority can interrupt
  - Lower priority can't
- Maskable interrupts
  - "don't bother me with that right now"
  - Not all interrupts are maskable, some are non-maskable

Interrupts in the ARM Cortex M processors

- External interrupts
  - From I/O pins of microcontroller
- Internal interrupts
  - Timers
    - Output compare
    - Input capture
    - Overflow
  - Communication units
    - Receiving something
    - Done sending
  - ADC
    - Completed conversion

Interrupt Jump Vector Table

- Fixed location in memory to find first instruction for each type of interrupt
- Only room for one instruction
  - JMP to location of complete ISR
- On system reset, the vector table is fixed at address 0x00000000.



On system reset, the vector table is fixed at address 0x00000000.

Chain of Events on Interrupt

- Finish executing current instruction
- Disable all interrupts — CLI
- Push program counter on to stack
- Jump to interrupt vector table
- Jump to start of complete ISR
- Save any context that ISR may otherwise change
  - Registers and flags must be saved within ISR and restored before it returns – **this is very important!**
- Re-enable interrupts if nested interrupts are ok
- Complete ISR's code
- Re-enable interrupts upon return
- Jump back to next instruction before interruption

Automatic

Compiler

SEI

RETI

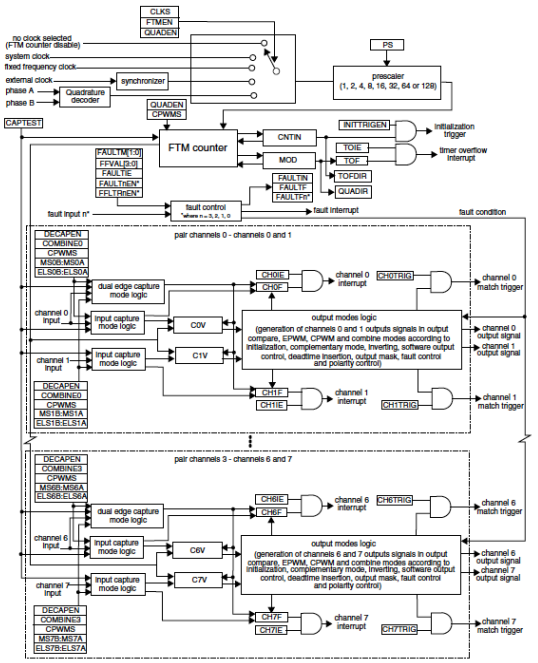
Shared Data Problem

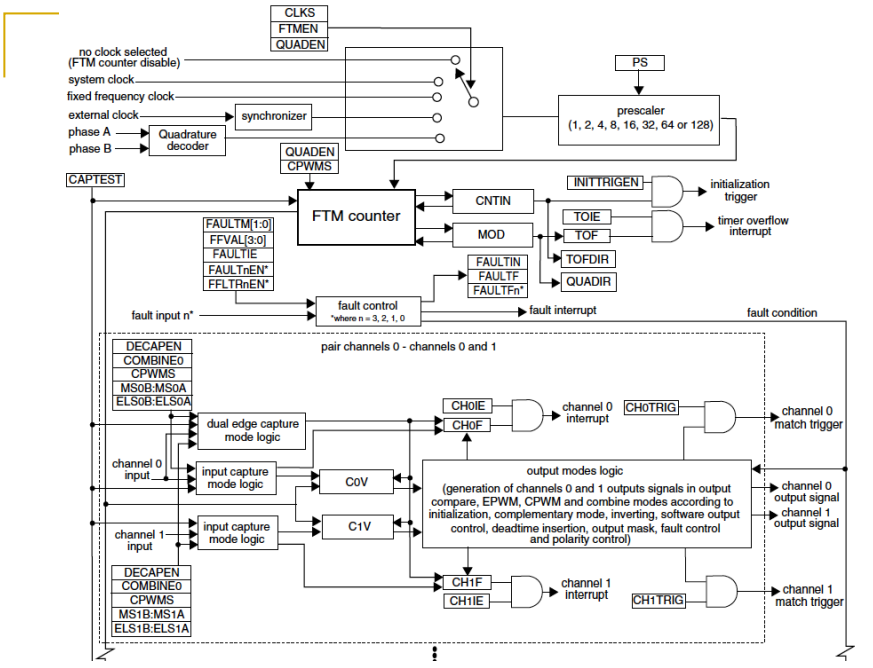
- When you use interrupts you create the opportunity for multiple sections of code to update a variable.
- This might cause a problems in your logic if an interrupt updates a variable between two lines of code that are directly dependent on each other (e.g. if statement)
- One solution is to create critical sections where you disable the interrupts for a short period of time while you complete your logic on the shared variable

```
cli();
…..critical section code goes here…..
sei();
```


Flex Timer

- Flex Timer Module (FTM)
  - Clear timer on compare mat (auto reload)
  - Prescaler divide-by 1, 2, 4, 8, 16, 32, 64, or 128
  - Overflow and compare match interrupts
  - Many PWM modes
  - Registers
    - Configuration
    - Count value
    - Output compare value





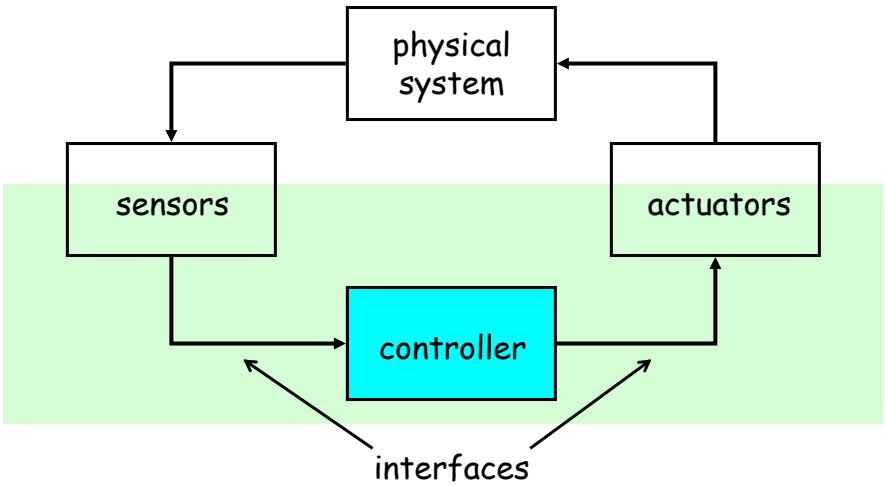
Interfacing

- Connecting the computation capabilities of a microcontroller to external signals
  - Transforming variable values into voltages and vice-versa
  - Digital and analog
- Issues
  - How many signals can be controlled?
  - How can digital and/or analog inputs be used to measure different physical phenomena?
  - How can digital and/or analog inputs be used to control different physical phenomena?

Controlling and reacting to the environment

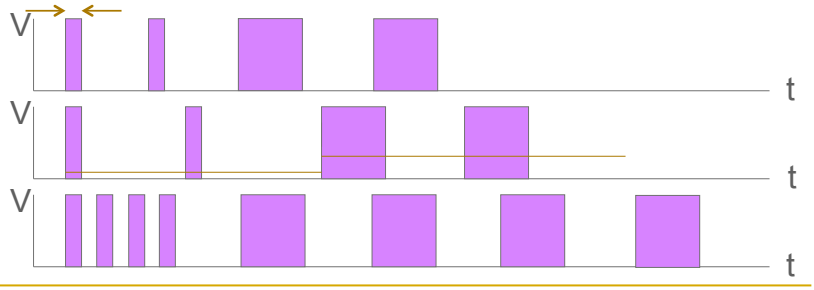
- To control or react to the environment we need to interface the microcontroller to peripheral devices
  - Microcontroller may contain specialized interfaces to sensors and actuators
- Things we want to measure or control
  - light, temperature, sound, pressure, velocity, position
- Sensors
  - e.g., switches, photoresistors, accelerometers, compass, sonar
- Actuators
  - e.g., motors, relays, LEDs, sonar, displays, buzzers

Typical control system



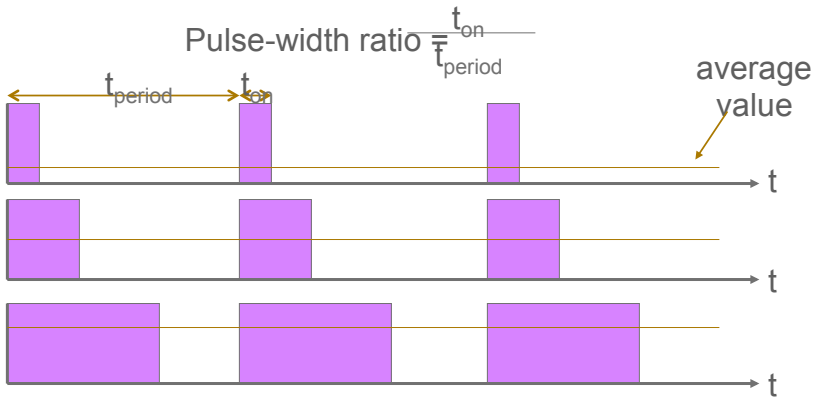
Digital to analog conversion

- Map binary values to analog outputs (voltages)
- Most devices have a digital interface – use time to encode value
- Time-varying digital signals – almost arbitrary resolution
  - pulse-code modulation (data = number or width of pulses)
  - pulse-width modulation (data = duty-cycle of pulses)
  - frequency modulation (data = rate at which pulses occur)



Pulse-width modulation

- Pulse a digital signal to get an average "analog" value
- The longer the pulse width, the higher the voltage



Why pulse-width modulation works

- Most mechanical systems are low-pass filters
  - Consider frequency components of pulse-width modulated signal
  - Low frequency components affect components
    - They pass through
  - High frequency components are too fast to fight inertia
    - They are "filtered out"
- Electrical RC-networks are low-pass filters
  - Time constant ($\tau = RC$) sets "cutoff" frequency that separates low and high frequencies

Why pulse-width modulation works

- LEDs
  - NOT low pass filters
  - But, your vision is, effectively
    - Persistence of vision
    - The reason motion pictures and video work
  - Teensy: PWM is controlled with the analogWrite(pin, value) function.
  - analogWrite(3, 50);
  - analogWrite(5, 140);
  - Here are the actual waveforms this code creates on pins 3 and 5: