

Lecture 5 objectives

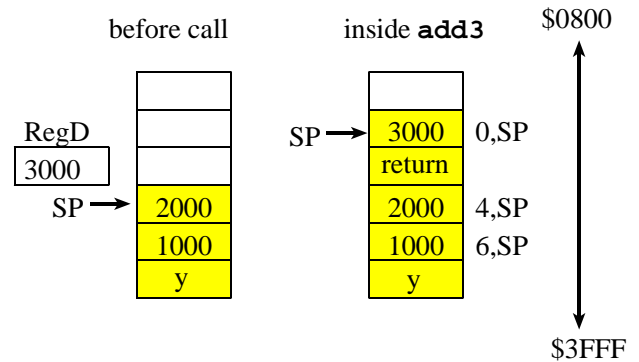
- Debugging using assembly language listings
- interface binary switch using pull-up resistor to an input port
- interface LED using a 7405 or PN2222
- draw flowchart of OC project
- profiling with the scope showing just how small a percentage of time is spent in the background

Debugging from an assembly perspective

```
short X1;
static short X2;
const short X3=3000;
short add3(short z1, short z2, short z3){
    short y;
    y = z1+z2+z3;
    return(y);
}
void main(void){ short y;
    X1 = 1000;
    X2 = 2000;
    y = add3(X1,X2,X3);
    EnableInterrupts;
    for(;;) {} /* wait forever */
}
```

assembly listing edited from main.lst

```
Function: add3
0000 3b          PSHD
0001 ec86       LDD    6,SP
0003 e384       ADDD   4,SP
0005 e380       ADDD   0,SP
0007 30        PULX
0008 3d        RTS
Function: main
0000 cc03e8     LDD    #1000
0003 7c0000     STD    X1
0006 59        LSLD
0007 7c0000     STD    X2
000a 49        LSRD
000b 6cac       STD    4,-SP
000d 59        LSLD
000e 3b        PSHD
000f cc0bb8     LDD    #3000
0012 0700       BSR    add3
0014 6ca3       STD    4,+SP
0016 10ef       CLI
0018 20fe       BRA    *+0 ;abs = 0018
```



Action

- 1) New 9S12DP512 project, select Serial Monitor
- 2) compile options, make S19, listing
- 3) copy paste example
- 4) Open PRM file (HCS12_Serial_Monitor_linker.prm)
 - See variables, programs and stack
- 5) Debug, single step (where is the stack?)

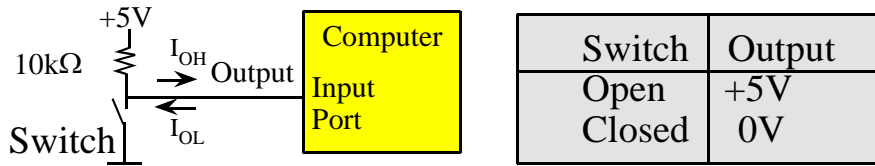


Figure 8.1. A simple switch interface.

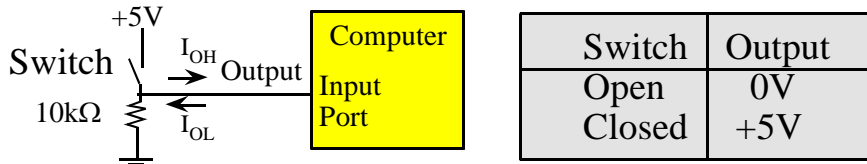


Figure 8.2. Another simple switch interface.

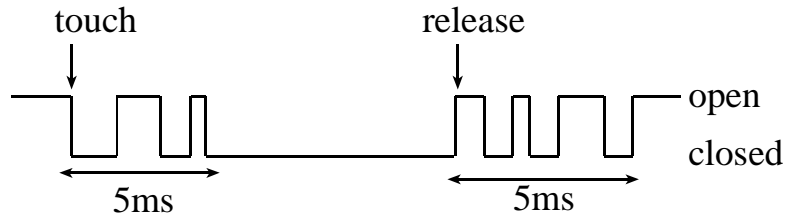


Figure 8.4. Switch timing showing bounce on touch and release.

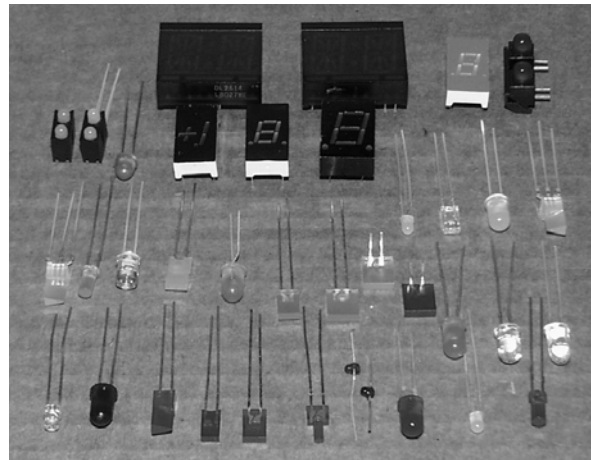


Figure 8.28. LED's come in a wide variety of shapes, sizes, colors.

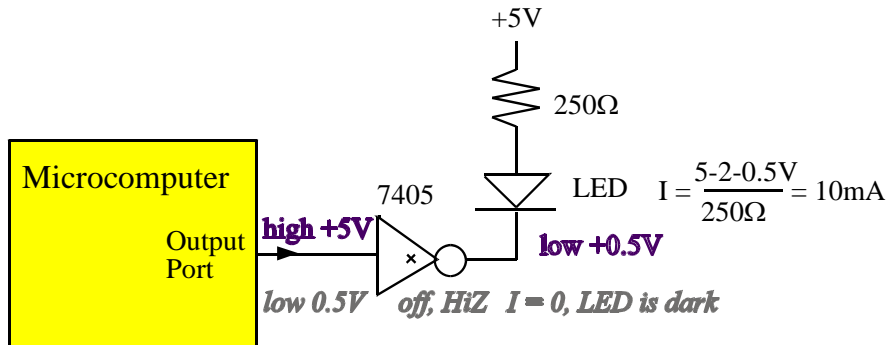


Figure 8.30. A single LED interface.

PN2222 $V_{ce} = 0.3V$

Red LED $1.5 \text{ min} < V < 2.0 \text{ max}$

Max LED Current $I = (5 - 1.5 - 0.3) / 220 = 15 \text{ mA}$

Min LED Current $I = (5 - 2.0 - 0.3) / 220 = 12 \text{ mA}$

$h_{fe} = 50$, so $I_b = 0.3 \text{ mA}$ (which is less than I_{OH} of the 6811)

Software must respond to events within a prescribed time.

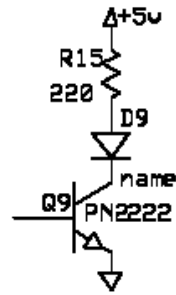
time between new keyboard inputs might be 10ms.

software latency

time from when new input is ready until the time software reads new data.

software latency must be less than 10ms.

interrupts guarantee an upper bound on the software response time.



Respond to infrequent but important events.

alarm conditions like low battery power and error conditions can be handled with interrupts.

Periodic interrupts, generated by the timer at a regular rate

clocks and timers

computer-based data acquisition and digital control systems.

Increase the overall bandwidth

buffer the data, spend less time waiting.

*uses a **first in first out queue***

4.1. What are interrupts?

An **interrupt** is the automatic transfer of software execution

in response to hardware that is asynchronous with the current software execution.

external I/O device (like a keyboard or printer) or

an internal event (like an op code fault, or a periodic timer.)

occurs the hardware needs service (busy to done state transition)

A **thread** is defined as the path of action of software as it executes.

a **background thread** interrupt service routine is called.

a new background thread is created for each interrupt request.

local variables and registers used in the interrupt service routine are unique

threads share globals

Each potential interrupt source has a separate **arm** bit. E.g., **C3I**

set the arm bits for those devices from which it wishes to accept interrupts,

deactivate the arm bits within those devices from which interrupts are not to be allowed.

Each potential interrupt source has a separate **flag** bit. E.g., **C3F**

hardware sets the flag when it wishes to request an interrupt

software clears the flag in ISR to signify it is processing the request

Interrupt **enable** bit, I, which is in the condition code register.

enable all armed interrupts by setting $I=0$, or **cli**

disable all interrupts by setting $I=1$. **sei**

$I=1$ does not dismiss the interrupt requests, rather it postpones

Shared open collector hardware request (polled)

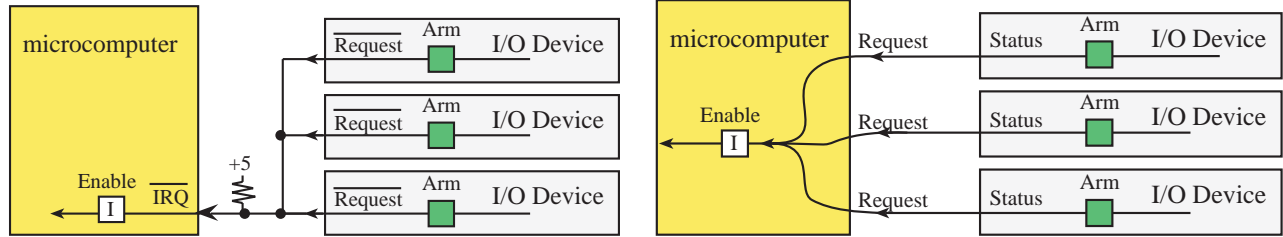


Figure 4.1. Wire-or negative-logic interrupt request line. Figure 4.2. Dedicated edge-triggered interrupt request lines.

share the same interrupt vector
 interrupt service routine must first determine which device requested the interrupt
 TDRE and RDRF have a shared vector, ISR must poll

Separate edge-triggered hardware request (vectored)

separate interrupt vectors
 jumps automatically to specific interrupt service routine

- 1) the software is simpler, therefore it will be easier to debug and it will run faster,
- 2) there will be less coupling in between modules, making it easier to debug and to reuse code,
- 3) it will be easier to implement priority such that higher priority requests are handled quickly

Three conditions must be true simultaneously for an interrupt to occur:

- 1) Initialization software will set the arm bit
 individual control bit for each possible flag that can interrupt
- 2) When it is convenient, the software will enable, I=0
 allow all interrupts now
- 3) Hardware action (busy to done) sets a flag **E.g., C3F**
 new input data ready,
 output device idle,
 periodic,
 alarm

What happens when an interrupt is processed?

- 1) the execution of the main program is suspended (the current instruction is finished),
 pushes registers on the stack
 sets the I bit
 gets the vector address from high memory
- 2) the interrupt service routine, or background thread is executed,
 clears the flag that requested the interrupt
 performs necessary operations
 communicates using global variables
- 3) the main program is resumed when the interrupt service routine executes **rti**.
 pulls the registers from the stack

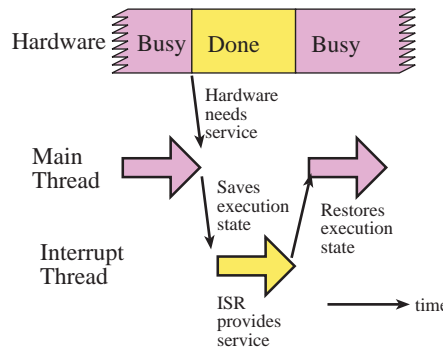


Figure 4.3. An interrupt causes the main thread to be suspended, and the interrupt thread is run.

What type of situations lend themselves to interrupt solutions?

Busy-wait

Predicable
Simple I/O
Fixed load
Dedicated, single thread
Single process
Nothing else to do

Interrupts

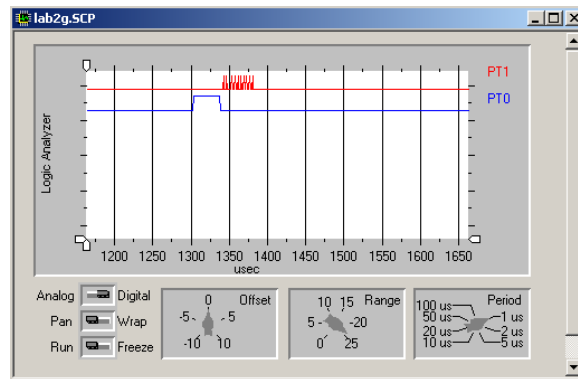
Variable arrival times
Complex I/O, different speeds
Variable load
Other functions to do
Multithread or multiprocess
Infrequent but important alarms
Program errors
Overflow, invalid op code
Illegal stack or memory access
Machine errors
Power failure, memory fault
Breakpoints for debugging
Real time clocks
Data acquisition and control

DMA

low latency
high bandwidth

Table 4.1. Each synchronization method has motivations for its use.

Run Lab2g on Texas, then on real board



Tasks to do in the ISR: Time critical operations, e.g.,

read new data from input device
write new data to output device
measure ADC input
set DAC output
increment time variables

Tasks to do in the foreground: not time critical, e.g.,

process/interpret data from input device
create data for output device
make calculations based on ADC input
determine the next DAC output
update displays seen by human eyes