I. Introduction

Key idea: interrupts provide a most useful mechanism in computer systems in which high-priority events may be handled most efficiently; also used to synchronize a program with real-time events.
- handles I/O more efficiently and increases MPU utilization (don’t need “busy” loops)
- Interrupts are used in real-time applications (i.e. scheduled/“timed” interrupts)
- multitasking: time division program execution
- provide notification of and/or recovery from software errors.

Interrupt: a high-priority, asynchronous event occurring in a computer system that requires (immediate) servicing by the processor.
- may be generated externally (by I/O hardware) or internally (I/O modules or software exceptions)
- require both hardware and software to be used effectively.

Interrupt Request (IRQ): a high-priority, asynchronous signal (message) sent from an I/O device/interface to the processor indicating that the I/O device/interface needs processor attention.

Interrupt Service Routine (ISR): an asynchronously executed (never "called" by the programmer as a subroutine) routine performed by the processor in response to an IRQ.

Interrupt states: a given interrupt request may be in one of a number of states...
- pending – asserted (active) but not yet being serviced
- in service – being processed by the MPU
- serviced / inactive
- various interrupt sources may be maskable (ignorable) or non-maskable (never ignored)
- maskable interrupts have an associated mask bit which enable or disable the interrupt
- multiple interrupt sources have different priority levels

II. Interrupt Details

Instruction execution timeline: controller-sequencer checks for interrupt condition between (but not within) instruction execution.
- thus: a pending interrupt can only be serviced after completion of any given instruction, never during!

Interrupt Components:
1. IRQ pin - the processor will have 1 or more input pins dedicated for external interrupt requests; may level or edge triggered, active high/rising or low/falling
2. IRQ flip-flop - latches interrupt request signal \( \Rightarrow \) pending interrupt (not yet serviced)
3. interrupt enable F/F - allows pending interrupt to actually reach the CPU if enabled
4. interrupt acknowledge (INTA) - signal generated by CPU in response to an enabled interrupt
5. interrupt return (RTI) - instruction used at end of ISR; note: do not use RTS!

III. Interrupt mechanism operation

Interrupt Operation: given an IRQ, the following sequence of events occurs.
1. CPU finishes current instruction
2. CPU pushes some/all registers on the stack (context saving)
3. CPU acknowledges pending interrupt and vectors to appropriate ISR
4. CPU executes code normally until RTI is encountered
5. CPU recalls some/all registers from stack, thus returning to next instruction in interrupted (main) program

Interrupt Latency: amount of time that transpires between IRQ signal and CPU execution of first instruction of ISR.
**ISR Vector** (starting address of an ISR): may be defined in a *vector table* or provided by hardware device
- each interrupt has an associated *vector* and *vector address*, these are stored in a *vector table*
  - *vector address*: fixed memory location(s) where ISR vector is stored
  - *vector table*: collection of all vector addresses for a given processor
- may also be supplied to CPU by I/O device via a handshaking procedure; more flexible but more complicated

**IV. Interrupt Programming**

three steps:
1. write the ISR
   - must end with RTI instruction (not RTS!)
   - must preserve all used registers (save before & restore after using)
   - must reset hardware that caused interrupt as appropriate
   - should be as short as possible (time-wise); "get in, get it done, get out"
2. setup the interrupt vector to the above ISR
   - typically done with ORG/FDB assembler directives
   Note: we can't do this directly with the HC11EVB but will do this in EET360
3. enable the interrupt
   - may require manipulation of a processor flag bit (I) and/or I/O control register
   - typically done in main program during initialization

**Intermodule communication:**
- an ISR cannot exchange data with interrupted program using registers
- must use global data memory (variables)
- beware of critical regions within interrupted programs

**V. Other Issues**
- multiple interrupt sources
  - each I/O device must have a "I did it" status feature to determine an interrupt's source
- interrupt masking
  - each interrupt source must have a "local" enable/disable feature
- interrupt nesting
  - nested ISRs are typically disallowed by CPU hardware; however, may be circumvented by programmer
- interrupt priorities & priority resolution
  - multiple interrupt sources may/should have a *priority* assignment & should be resolved in priority order; may be implemented in software (poll order within ISR) or in hardware (PIC, etc.)
- software interrupts
  - typical CPU designs provide 1 or more instructions to "simulate" an interrupt, although this is an entirely synchronous event (ex: SWI); this is commonly used for debugging purposes
- exceptions
  - some CPU designs provide more generic interrupt features implemented as *exceptions* (ex: catching a division by 0)