EECE 218
Microcontrollers

Program organization
Interrupt driven I/O
Software Architecture

- How to organize the code for a microcontroller application?

Typical microcontroller app:

Application is reactive: reacts to changes in the environment
Software Architecture

Typical code:

```c
int main() {
    initialize();
    for(;;) { /* Loop forever */
        if (check_in_1()) handle_in_1();
        if (check_in_2()) handle_in_2();
        ...
        if (check_in_N()) handle_in_N();
    }
}
```

Problems:
- Programmer must anticipate I/O at specific points
- Response time may be too slow
- Logic is difficult to extend
Interrupts – Asynchronous I/O

- IT: hardware-generated event that indicates change in the ‘external world’ (incl. device)
- I(ntrrupt) S(ervice) R(outine): special subroutine, called when IT happens

Advantages:
- Allows coordination between program and asynchronous events
- Allows multi-tasking (e.g. calculations overlapping with I/O)

Disadvantage: Complexity
Interrupts

- Typical behavior

Physical device performs I/O

Main program starts I/O process here

Main program continues

ISR handles end of I/O

Main program continues

IT event

Call ISR
Code organization with interrupts

```c
void isr_handle_in_1() { .... }
void isr_handle_in_2() { .... }
....
void isr_handle_in_N() { ... }

int main() {
    initialize();
    for(;;) { wait_for_interrupt(); }
}
```

- Interrupts may come at any time
- System reacts as fast as possible
- Logic is simple(r) to extend
Interrupt machinery

Who can generate it?
- Most of the peripheral devices on the 9S12: ‘Built-in’: SCI, SPI, ECT, CAN, some parallel ports,
- ~IRQ and ~XIRQ lines (HW pins)
- Interrupt-like cases:
  - RESET (power on), clock monitor/watchdog
  - Bad instruction code, SWI instruction

How to control it:
- Built-ins and IRQ are maskable
  Globally via I bit in CCR (1=disabled/0=enabled)
  Locally, in the built-in device
- Note: IRQ is controlled by the I bit only
  Both local and global control must be enabled
- XIRQ is masked after RESET (X bit in CCR), clearing X bit enables XIRQ interrupts BUT it cannot be set again.
- Other cases (RESET, etc.) cannot be masked
(HW) Interrupt processing flow

- **XIRQ**
  - **X set?**
    - **N**
      - Stack CPU State
        - (CCR,A,B,X,Y,PC)
      - Set I(X) in CCR
      - Goto ISR
    - **Y** → Continue
- **IRQ**
  - **I set?**
    - **N** → Continue
    - **Y** → Continue
- **Built-in** → Continue
Locating the ISR

- Each type of ISR has a dedicated location in memory that contains the address of the corresponding ISR.
  
  Example:
  
  $FFD4/FFD5$ contains address of SCI1 ISR

  Table at $FF80$-$FFFF$: Interrupt Vector Table
  This is ROM/Flash --- cannot be changed from SW.

- In the lab: DBUG-12 (see manual)

  The IVT is remapped into RAM

  Example:
  
  $3E54/3E55$ contains address of SCI1 ISR

  How?

  $[FFD4/FFD5] \rightarrow \text{JMP} [3E54]$ (indirect jump)

  If ISR-s are needed: RAM table must be filled out (~DC.W~ ...)
Interrupt machinery

- Upon entering ISR I (or X) is set (disabled)
  If I is set, XIRQ can still happen
- Programmer can clear I in ISR! (Nesting IT-s)
  NOT recommended, because
  - Complex
  - Pending requests must be cleared before proceeding (to avoid infinite IT-ISR progression)
- To set/clear I: SEI (disable!) / CLI (enable!)
  » This is the Global I-mask
- To return from ISR:
  ALWAYS use RTI (return from interrupt)
  Expects special stack layout – 9 bytes – do not mess up stack!
Interrupt programming scheme

- ‘Main’ program:
  - Reset sets I (use SEI to make sure)
  - Program IT-generating device
  - Ensure enough stack is available (2/SR,9/ISR)
  - Reroute IT vectors (if in RAM/Lab)
  - When all IT-related work is done, CLI
  - Continue with main algorithm

- In ISR:
  - Ascertain identity of IT-requesting device (check device register, poll if multiple devices)
  - Acknowledge IT (clear device’s ‘status flag’)
  - Do all service in ISR (has its own context)
  - Return with an RTI
Interrupt priorities

Who gets serviced first?

- Why? Fast/slow devices, to allocate CPU cycles to ‘tasks’
- Hardware defined/programmable (9S12)
  - RESET has highest priority
  - XIRQ has higher priority than any other HW interrupt
- Multiple devices requesting the same:

  ![Diagram of CPU and devices]

  - Pull-up to Vcc (R)
  - In ISR: poll devices, one by one. Poll order establishes priority.