EE 3170 Microcontroller Applications

Lecture 13: Introduction to Microcontroller Hardware (Part II - Input/Output)
- Miller §6.1 - §6.12

Based on slides for ECE3170 by Profs. Davis, Kieckhafer, Tan, and Cischke

Objectives

- Describe the I/O ports of the 6811
- Explain the basic parts of I/O programming
- Give examples of I/O device drivers
- Understand simple I/O programs

Input/Output Architectures

- I/O Hardware Model
- 68HC11 I/O Organization
- Specific Port Examples
  - PORTB
  - PORTC
- Example of Polled I/O Operation
  - Observations
- Interrupt-Driven I/O
  - concept and requirements
  - generic response to an interrupt request
- Interrupts in the 68HC11
  - hardware setup
  - response to an interrupt request
- Example of Interrupt Driven I/O

How Does an Embedded System Interact with the World?

- An embedded system uses input/output devices to connect with the real world.
- Input devices get information from the external world, often through sensors.
- Output devices
  - control physical systems or
  - display information.
I/O Hardware Model

- **I/O ports** are the parts of a microcomputer that connect to its environment.
  - A set of pins for data input or output
  - Maybe a register attached to the pins
  - The control protocol for the pins

- **I/O Device**
  - A device attached to the I/O port
  - "Peripheral" to the computer
  - Generates data and sends it to a port AND/OR
  - Reads data from the port, and acts on it

- **Device drivers** are software routines that let higher level software use an I/O device.
  - You’re probably familiar with device drivers for your home computer.

### Bus Architecture: A Side Note

- **Some addr. & data bits share the same physical lines**
  - Low order address bits (A7…A0)
  - Data bits (D7…D0)
  - Lines are labeled: (A15…A8, AD7…AD0)

- **Bus usage within one bus cycle is time-multiplexed**
  - Issue address during 1st half of a cycle
  - Transfer data during 2nd half of the same cycle

- **This is a common technique to save lines & pins**
  - Transparent to the programmer
  - Lower performance solution
  - Clock period is long enough to include both phases in a single cycle

### What’s an I/O Port Address?

- **Most embedded microprocessors**, including the 6811, are memory mapped.
- This means that each I/O port has one or more dedicated “memory” addresses.
  - Data can then be written to the address for sending to the I/O device (output) or
  - Data can be read from the address for input to the processor.
- **The frame of reference is the computer.**
  - **Input** is to the computer from a device.
  - **Output** is from the computer to a device.
I/O Hardware Model

- **Input Port:**
  - receives data from the incoming pins
  - makes it available to the data bus
  - generally does not latch the data

- **Output Port:**
  - receives data from the data bus
  - makes it available on the outgoing pins
  - usually has a latch to hold the data for the device
    - value remains constant until a new value is output

I/O Organization

- 68HC11 uses Memory Mapped I/O =
  - Each I/O port behaves like a single word of memory
    - It recognizes an address as itself
    - It transfers data on the data bus
    - It responds to Load and Store commands
  - An I/O Port can have more than 1 register
    - A data register (latched or unlatched)
    - A command/status register
  - An I/O register can be
    - input only
    - output only
    - programmable (bidirectional)
  - Serial I/O port moves data at 1 bit per transfer
  - Parallel I/O Port moves data in word width chunks
    - Typically 8-16 bits in parallel
    - 68HC11 parallel I/O ports are 8-bits wide

Typical I/O Port

- Cnt! Data Addr
  - Com/Stat Reg
  - Com/Stat Reg addr decoder
  - Port Control Logic
  - Data Reg addr decoder
    - Data Reg
    - Device Keyboard, etc.

- I/O ports have Memory Locations/Register associated

Input Port

- Figure 6-19 Input port circuit.
**Output Port**

![Image of Output Port Circuit](image)

**Command & Status Register**

- This C/S register is part of the I/O port.
- Usually one bit of it is a ready flag.
- The C/S register has its own address.
  - A different address than the data port.
- Purposes of the C/S register:
  - Allow processor to read status info on the port.
    - e.g. Ready, Error, etc.
  - Allow the processor to write commands to the port.
    - To alter the way the port behaves.

---

**68HC11 I/O Organization**

- **68HC11 has 5 I/O ports**
  - PORTA -- 3 input, 4 output, and 1 bidirectional pin
    - also used for timer and pulse accumulator I/O
  - PORTB -- 8 output pins
  - PORTC -- 8 bidirectional pins
  - PORTD -- 6 bidirectional pins
    - also used for asynchronous serial comm
  - PORTE -- 8 input pins
    - also used for Analog to Digital (A/D) conversion
    - can handle analog voltage inputs

---

**68HC11 – E “family” I/O Ports**

<table>
<thead>
<tr>
<th>Port</th>
<th>Input Pins</th>
<th>Output Pins</th>
<th>Bidirectional Pins</th>
<th>Shared Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Port A</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>Timer</td>
</tr>
<tr>
<td>Port B</td>
<td>—</td>
<td>8</td>
<td>—</td>
<td>High-order address</td>
</tr>
<tr>
<td>Port C</td>
<td>—</td>
<td>—</td>
<td>8</td>
<td>Low-order address and data bus</td>
</tr>
<tr>
<td>Port D</td>
<td>—</td>
<td>—</td>
<td>8</td>
<td>Serial communications interface (SCI) and serial peripheral interface (SPI)</td>
</tr>
<tr>
<td>Port E</td>
<td>8</td>
<td>—</td>
<td>—</td>
<td>Analog-to-digital (A/D) converter</td>
</tr>
</tbody>
</table>
I/O Memory Mapping Addresses

- All built-in I/O and control regs are mapped into
  - a 64-byte block addresses
  - at $1000 … $103F
- The I/O port registers are a subset of that space
  - consisting of 11-bytes
  - not all at contiguous addresses in the block
- We will consider PORTB and PORTC in this chapter
  - They are the simplest and most general purpose
  - Others are considered in later chapters

I/O Registers

- Memory-Mapped I/O

<table>
<thead>
<tr>
<th>Name</th>
<th>Address</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>PORTA</td>
<td>$1000</td>
<td>Timer and Counter System</td>
</tr>
<tr>
<td>PORTB</td>
<td>$1004</td>
<td>Parallel Output</td>
</tr>
<tr>
<td>PORTC</td>
<td>$1003</td>
<td>Parallel Input/Output</td>
</tr>
<tr>
<td>PORTCL</td>
<td>$1005</td>
<td>PORTC Latch</td>
</tr>
<tr>
<td>PORTD</td>
<td>$1008</td>
<td>Serial Input/Output</td>
</tr>
<tr>
<td>PORTE</td>
<td>$100A</td>
<td>Analog-to-Digital Converters</td>
</tr>
<tr>
<td>PIOC</td>
<td>$1002</td>
<td>Parallel I/O Control</td>
</tr>
<tr>
<td>DDRC</td>
<td>$1007</td>
<td>Data Direction PORTC</td>
</tr>
<tr>
<td>DDRD</td>
<td>$1009</td>
<td>Data Direction PORTD</td>
</tr>
</tbody>
</table>

Parallel I/O Control Registers

<table>
<thead>
<tr>
<th>Bit 7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>BR7</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>PORTA</td>
</tr>
<tr>
<td>SAFP</td>
<td>SPC</td>
<td>VROM</td>
<td>RAMO</td>
<td>IOB</td>
<td>PC1</td>
<td>DDA</td>
<td>IVA</td>
<td>PIOC</td>
</tr>
<tr>
<td>BR7</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>PORTC</td>
</tr>
<tr>
<td>BR7</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>PORTB</td>
</tr>
<tr>
<td>BR7</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>DDRD</td>
</tr>
</tbody>
</table>

Parallel I/O Registers and Control Bits

<table>
<thead>
<tr>
<th>BR7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOR</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>DDRD</td>
</tr>
<tr>
<td>DOR</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>PACTL</td>
</tr>
<tr>
<td>DOR</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>SPIR</td>
</tr>
</tbody>
</table>

PORTB

- Unidirectional 8-bit output port
- $1004 = PORTB Data Register’s Address
- To do an output, simply store to $1004, e.g.
  - staa $1004 * Accumulator A → PORTB
  - stab $1004 * Accumulator B → PORTB
  - clr $1004 * $00 → PORTB
- Details
  - Output Pins hold a value until it is changed
  - At RESET, $00 → PORTB

$1004  0  0  0  0  0  0  0  0  PORTB
        0  0  0  0  0  0  0  0  RESET
PORTC

- Programmable (bidirectional) 8-bit I/O port
- Four Registers of Interest
  - $1003 = PORTC = the data register
  - $1007 = DDRC = Data Direction Register for C
  - $1002 = PIOC = Parallel I/O Control register
  - $1005 = PORTCL = PORTC Latch register

PORTC Reg

- The PORTC Register @ $1003
  - Individual Pins can be configured as input or output
- To do an I/O ops, load or store @ $1003, e.g.
  - staa $1003
    - Accumulator A → PORTC
  - ldaa $1003
    - PORTC → Accumulator A

Details
- A store to an input pin is ignored
- A load from an output pin returns last value stored
- At RESET, $00 → PORTC

DDRC Reg

- The DDRC Register @ $1007
  - Data Direction Register for port C
- Defines the direction of individual pins in PORTC
  - store to $1007 sets direction of each bit in PORTC
    - 0 = input pin,
    - 1 = output pin
- load from $1007 returns the last value stored
  - the current status of the PORTC pins
- At Reset: $00 → DDRC (initializes PORTC as input)

DDRC Reg

- PORTC Configuration Example:
  - ldaa #$F0
  - staa $1007
    - Stores $F0 = %1111,0000 into DDRC
    - Initializes PORTC such that
      - pins 7…4 = output pins
      - pins 3…0 = input pins
**PIOC Reg**

- The PIOC Register @ $1002
  - An I/O port control register
  - Contains control bits for several ports
- Bit 7 = STAF bit = Status Flag bit
  - There is an external ready signal STRA = Strobe A
  - STAF is set on a change in the value of STRA
    - either rising edge or falling edge sensitive
  - Edge sensitivity is determined by Bit 1 = EGA bit
    - EGA = 1 ⇒ STAF is rising edge sensitive
    - EGA = 0 ⇒ STAF is falling edge sensitive

**PORTCL Reg**

- The PORTC Latch Register @ $1005
  - Sometimes it is necessary to latch an input
    - at the instant the input device presents it
  - PORTCL latches the contents of PORTC
    - at the instant the STRA pin sets the STAF flag
- Last detail: clearing the STAF flag in PIOC reg
  - Two step op: read PIOC, then read PORTCL, e.g.
    - ldaa $1002
    - ldaa $1005

**Example: Figure 6-20**

- Read Two Push-Button Switches
- If both are closed, then light an LED, else turn it off

**Details**

- This is an asynchronous real-time application
  - The input data is always “valid” (or meaningful)
  - Do not need a “ready” flag (or status register)
  - Read the two switches in real-time
  - Set or Clear the LED bit in real-time
- Port Addresses
  - Input Port is at address $4002
  - Output Port is at address $4003
- Initial Conditions
  - Start with LED lit
Key Bit-Manipulation Instructions

- \( \text{bclr} \) address, mask
  - clear those bits of a mem word specified by the mask
  - i.e. \( \text{M[offset]} \leftarrow \text{M[offset]} \text{ AND (mask)} \)
  - Example:
    - \( \text{M[offset]} = 1101,0011 \)
    - \( \text{mask} = 0100,0001 \) (select bits 6 and 0)
    - \( \text{M[offset]} \leftarrow (1101,0011) \text{ AND (1011,1110)} = (1001,0110) \)

- \( \text{bset} \) works the same to set the selected bits

Key Bit-Conditional Instructions

- \( \text{brclr} \) offset, X, mask, target
  - examines a memory word, \( \text{M[X+offset]} \)
  - If all bits selected by the mask byte are clear (= 0)
  - Then branch to the target address
  - In other words:
    - IF \( \text{M[X+offset]} \text{ AND mask} = 00 \)
    THEN Branch to Target

- \( \text{brset} \) works the same if the selected bits are set to 1
Why I/O Synchronization?

- I/O Speed
  - I/O devices are typically much slower than processors
  - Can’t read from a device until it has the data ready
  - Can’t write to a device until it is ready to receive data
    - usually when it is done processing the previous data
  - Processor must “synchronize” with I/O devices
    - device sets a 1-bit “ready” flag when it is ready
    - proc. waits for flag to be set before accessing the port
    - flag is cleared when proc. accesses the port

How: Two Forms of I/O Synch.

- Polled I/O
  - proc. enters a “polling loop” to read the ready flag
  - when the flag is set, then processor accesses port
  - processor ends up waiting for the device

- Interrupt-Driven I/O
  - Ready flag is connected to an “Interrupt” pin on proc.
  - When Pin = 1, the processor
    - interrupts the current program
    - calls an Interrupt Service Routine (ISR) to access the port
    - then returns to the previous program

- DMA (Direct Memory Access)
  - ISR supports Control of an independent bus master
I/O Synchronization

- I/O Synchronization is determined by the programming model
  - Polled
  - Interrupt
  - DMA

- I/O Synchronization is constrained by interface hardware between CPU/Controller and I/O Controller/Devices

Polled I/O

Program Function
- User sets a number (0…F) with thumbwheel
- User pushes the ready button
  - Sets STAF-bit in PIOC via STRA pin (rising edge)
  - Latches input value into PORTCL
- Program loop continuously checks STAF bit of PIOC
  - Program reads PORTCL
  - Program outputs the input value to the display
Polled I/O Example

Program of Figure 6-29

*-----------------------------------------------
** SYMBOLOGY
* SYMBOL DEFINITIONS
**-----------------------------------------------

* #HCU11 REGISTERS
POTC EQU $1002  PARALLEL I/O CONTROL REGISTER
PORTC EQU $1003  I/O PORT C REGISTER
PORTCL EQU $1005  PORT C LATCH REGISTER
DDRC EQU $1007  DATA DIRECTION REGISTER C
IOPAT EQU $0F  I/O PATTERN, 0=IN 1=OUT

*-----------------------------------------------
** PROGRAM SECTION
*-----------------------------------------------

Polled I/O Example

();//read thumbwheel &
clear I/O flag

/initialize PORC

/initialize PORTC

/position data
for output

//control display

/end

/end

/end

*/

Polled I/O Example

*/

* INITIALIZE
* INITIALIZE PORT C
* INITIALIZE OUTPUTS TO ZEROS
CLR PORTC 0=LOW
* SET UP I NS AND O UT S
LDAA #$0F
STAA DDRC 0=IN, 1=OUT
* SET UP PIO C
LDAA #$02  STAP RESPONSES ON LOW TO HIGH
STAA PIOC ..TRANSITION ON STAP
Observations about Polled I/O

- Processor spends enormous amounts of time waiting for the I/O device to be ready
  - Human = seconds or tenths of seconds
  - Disk ≈ 10 ms for 1st word
    - @ 1GHz, 10 ms = 10 million clock cycles
  - Disk ≈ 0.1 ms between words
    - @ 1GHz, 0.1 ms = 100,000 clock cycles
- CPU spends all that time in the polling loop
  - It is not doing useful work

What Is Interrupt-Driven I/O?

- Problem with polling:
  - CPU spends most of its time waiting for the device.
- Solution:
  - Let device interrupt CPU when device is ready.
  - CPU drops what it was doing and services the port.
  - CPU returns to its original task.
- Interrupt Service Routine (ISR): routine that services the I/O port
  - Resembles a subroutine call
  - Done when device is ready

What Hardware Do We Need for Interrupts?

- Interrupt Request Pins (IRQ) on CPU
- Interrupt Acknowledge Pins (IACK)
- READY bit of port stat/com register is wired to an IRQ pin
- CPU must check IRQ pins
  - checked between Instructions
- CPU can ignore (mask) interrupt pins if an interrupt would be inconvenient
  - Mask bit holds current mask status
  - Mask manipulation instructions
How Does Hardware Respond to an IRQ?

- IF IRQ is masked, THEN ignore interrupt
- ELSE
  - Identify the originator of the interrupt
  - Find the address of the ISR
  - Push the process state on the stack
    - Program Counter - address of next instruction to be fetched
    - Program Status - e.g. Condition Code register
      - Most CPUs don’t save general purpose registers
      - Takes a lot of time
      - Let the ISR push any registers it needs
  - PC ← Address of ISR routine

How is Return from Interrupt Handled?

- Initiated by Return from Interrupt instruction
  - at end of Interrupt Service Routine
    - 68HC11 = RTI
  - Undo all stack ops done by the interrupt
  - The next instruction the CPU fetches will be the next instruction of the previous task.

Interrupts

- How to signal CPU
- How to protect the registers
- Where to find the subroutine
- How to come back

How Do We Know What Device Interrupted?

- Most CPU’s have 2 - 8 IRQ pins
- One pin usually Non-Maskable Interrupt (NMI)
  - usually for imminent disaster
- Remaining pins are prioritized (usually fixed priority)
  - usually more I/O devices than IRQ pins
  - may have simultaneous requests
  - must determine device that initiated IRQ
  - then find address of the proper ISR
How Do We Get the Address of the Interrupting Device?

- Static vector table or
- Direct bus vectored interrupts or
- Indirect bus vectored interrupts

Static Vector Table (Table 7-2)

- Put address of ISR in a fixed memory location
- Vector Table = List of all ISR addresses
- IRQ pin number points to Vector Table
- Vector Table entry points to ISR
- ISR can be anywhere in memory
- All devices sharing one IRQ pin must use same ISR
- CPU Hardware
  - Fetches vector table entry specified by pin number
  - Loads that value into Program Counter (PC)
  - Fetches 1st instruction of the ISR

Static Vector Table, p. 2

- Example:
  - Number the IRQ pins (0…N-1)
  - Assume:
    - Vector table starts @0000
    - 1 stored address = 2 Bytes
  - Then:
    - ISR addr for IRQ pin i is stored in M[2i:2i+1]

Direct Bus Vectored Interrupts

- Let the device send its ISR address to the CPU
- In response to IACK, device sends address on data bus
- CPU latches address into Program Counter
Indirect Bus Vectored Interrupts

- Device sends a pointer into the vector table rather than actual address of ISR
  - think of it as a device I.D. number
- CPU gets the actual address from the vector table
- The most common approach, why?

How Does the 6811 Handle Interrupts?

- Two interrupt Pins (active low)
  - IRQ -- maskable
  - XIRQ -- non-maskable
- Device must latch its request
  - Done via a flag latch in the I/O port
  - Example: the STAF flag in the PIOC
    - can be programmed to trigger an IRQ
  - Other external ports must implement their own latch
    - latch output is wired to IRQ
    - multiple devices can be Wire-ORed

The Interrupt Process – request (Skipped)

- Internal request
  - Pulse accumulator
  - Timer
  - SPI serial transfer
  - SCI serial system
  - Parallel I/O

Enabling and Disabling Interrupts

- 1 - Global IRQ Enable
  - Bit 4 of CC register = 1 bit

  Two instructions control the I bit
  - CLI ⇒ make I = 0 ⇒ enable IRQ
  - SEI ⇒ make I = 1 ⇒ disable IRQ

  When an IRQ occurs,
  - hardware immediately pushes CCs
  - and disables IRQ to prevent infinite loop
  - ISR can enable IRQ by executing a CLI, if desired
  - IRQ will be automatically enabled upon Return
    (when old CC is popped off stack)
Enabling and Disabling Interrupts, p.2

- 2 - Enabling Individual I/O Port Flags
  - Each I/O port can disable its flag from initiating an IRQ
  - For external ports it is up to the individual designer
  - For STAF flag in the PIOC it is controlled by STAI bit
    - STAI = 0  STAF interrupt is disabled
    - STAI = 1  STAF interrupt is enabled

<table>
<thead>
<tr>
<th></th>
<th>STAF</th>
<th>STAI</th>
<th>CWOM</th>
<th>HNDS</th>
<th>OIN</th>
<th>PLS</th>
<th>EGA</th>
<th>INVB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Hardware Response to IRQ

- Save Process State
  - PUSH:  PC, CC, A, B, X, Y

- PC ← Address of ISR
  - Obtained from a static vector table in ($FFC0 - $FFFF)
    - IRQ Pin vector is in M [FFF2:FFF3]
    - XIRQ Pin vector is in M [FFF4:FFF5]
    - RESET Pin vector is in M [FFF6:FFFF]

- Disable IRQ
  - can be enabled by ISR
  - automatically enabled on return from ISR

Software Response to IRQ

- ISR doesn’t have to save registers.
  - done by hardware

- ISR should execute a CLI instruction to re-enable IRQ as soon as it is safe to do so.

- Return from Interrupt (RTI) instruction
  - Pops:  Y, X, B, A, CC, PC
    - Popping PC returns to the previous task.
    - Popping CC
      - restores previous condition codes
      - restores previous IRQ enable bit
    - Result -- ISR leaves no residual effects on the task.

IRQ-Driven I/O Example
IRQ-Driven I/O Example

- Same Problem, Different Synchronization
  - User sets a number (0…F) with thumbwheel
  - User pushes the ready button (P.288)
    - sets STAF-bit in PIOC via STRA pin (rising edge)
    - Latches input value into PORTCL
    - Sets IRQ input: ↔ STAF Output
  - Program is Interrupted
    - ISR reads PORTCL
    - ISR outputs the input value to the display
    - ISR returns to main program

/* Initialization */
//initialize PORTC
//set up PIOC
//set up PIOC
//read thumbwheel & clear I/O flag
//position data for output
//control display

/* Copy Thumbwheel */
//copy thumbwheel
//do something else (something useful)
//return to main program
//Valid STAF

#include <reg51.h>

void main()
{
  P2 = 0x00; // Clear I/O flag
  P1 = 0x10; // Position data for output
  P0 = 0x00; // Control display
}

ISR(IRQ0)
{
  PORTC = P2; // Read thumbwheel
  P2 = 0x00; // Set I/O flag
  P1 = P1 & 0xffffffff; // Do something useful
  P0 = P0 & 0xffffffff; // Return to main program
}
IRQ-Driven I/O Example

```assembly
** MAIN PROGRAM
********************************************************************************
ORG $C100
* INITIALIZATION
********************************************************************************
LDS $10FF
* INITIALIZE STACK
SET UP INS AND OUTS
LDA $0FFA
STAA D80C 0=IN, 1=OUT
SET UP PIOC
LDAA $413 ENABLE STAP INTERRUPT
STAA PIOC LOW TO HIGH ON STAP
* TURN ON INTERRUPT SYSTEM
CLI
********************************************************************************
* WAIT FOR INTERRUPTS
********************************************************************************
HERE HERE DO NOTHING!
*"```

IRQ Summary

- **IRQ advantages**
  - Program can go about its business
  - Doesn’t continuously poll ports
  - Polling is a huge waste of CPU time
  - CPU services ports only when needed
  - Port management greatly simplified

- **IRQ Disadvantages**
  - Main program execution time indeterminate
    - depends on number and exact timing of IRQs
  - May make real-time deadlines harder to guarantee

```assembly
* VALID STAP INTERRUPT?
********************************************************************************
IRQSER LDS $880
BCLR PIOC-REQ, X,RST,RHST,RHT IRQ BRANCH ON NO
********************************************************************************
* COPY THUMBWHEEL SWITCH TO DISPLAY
********************************************************************************
* READ THUMBWHEEL SWITCH AND CLEAR I/O FLAG
LDAA PIOC TWO INSTRUCTIONS TO CLEAR STAP
LDAA PORTCL ..FLAG AND INPUT SSXSS----
* POSITION DATA FOR OUTPUT
STAA ----SSXSS
STAA ----SSXSS
STAA ----SSXSS
* CONTROL DISPLAY
OUTP STAPC OUTPUT TO PORT C
* RETURN TO MAIN PROGRAM
********************************************************************************
RTIA RTI
*"```