



Introduction to Embedded Systems

Sanjit A. Seshia

UC Berkeley
EECS 149/249A
Fall 2015

© 2008-2015: E. A. Lee, A. L. Sangiovanni-Vincentelli, S. A. Seshia. All rights reserved.

Chapter 10: Input and Output, Interrupts

Connecting the Analog and Digital Worlds

Semantic mismatch:

Cyber:

- Digital
- Discrete in time
- Sequential

Physical:

- Continuum
- Continuous in time
- Concurrent

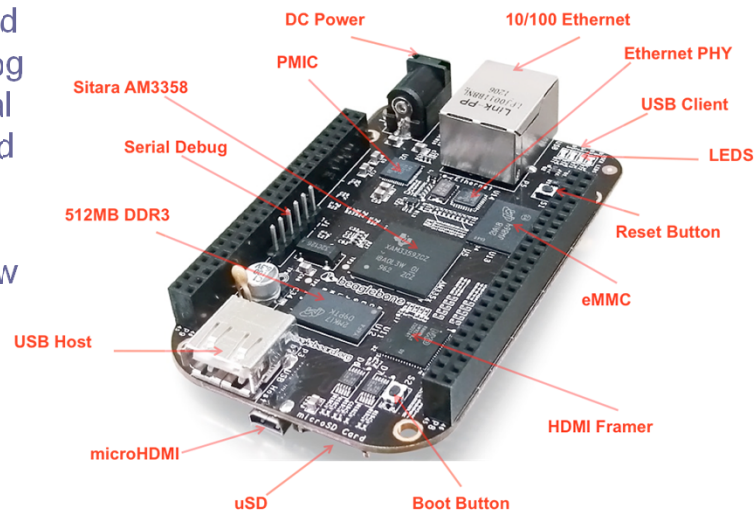
Practical Issues

- Analog vs. digital
- Wired vs. wireless
- Serial vs. parallel
- Sampled or event triggered
- Bit rates
- Access control, security, authentication
- Physical connectors
- Electrical requirements (voltages and currents)

EECS 149/249A, UC Berkeley: 3

A Typical Microcomputer Board Beaglebone Black from Texas Instruments

This board has analog and digital inputs and outputs. What are they? How do they work?



EECS 149/249A, UC Berkeley: 4

Wired Connections Parallel vs. Serial Digital Interfaces

- Parallel (one wire per bit)
 - ATA: Advanced Technology Attachment
 - PCI: Peripheral Component Interface
 - SCSI: Small Computer System Interface
 - ...
- Serial (one wire per direction)
 - RS-232
 - SPI: Serial Peripheral Interface bus
 - I²C: Inter-Integrated Circuit
 - USB: Universal Serial Bus
 - SATA: Serial ATA
 - ...
- Mixed (one or more “lanes”)
 - PCIe: PCI Express



EECS 149/249A, UC Berkeley: 5

Wired Connections Parallel vs. Serial Digital Interfaces

Parallel connectors have been steadily replaced by serial ones.

Why?

EECS 149/249A, UC Berkeley: 6

Input/Output Mechanisms in Software

o Polling

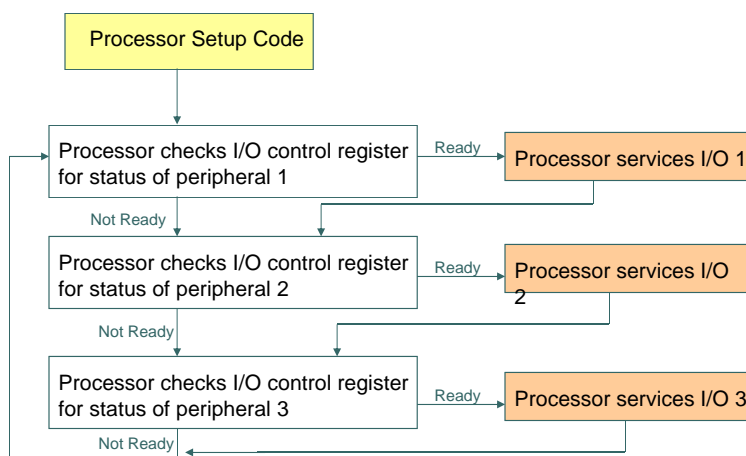
- Main loop uses each I/O device periodically.
- If output is to be produced, produce it.
- If input is ready, read it.

o Interrupts

- External hardware alerts the processor that input is ready.
- Processor suspends what it is doing.
- Processor invokes an interrupt service routine (ISR).
- ISR interacts with the application concurrently.

EECS 149/249A, UC Berkeley: 7

Polling



EECS 149/249A, UC Berkeley: 8

Example Using a Serial Interface

In an Atmel AVR 8-bit microcontroller, to send a byte over a serial port, the following C code will do:

```
while(!(UCSR0A & 0x20));  
UDR0 = x;
```

- x is a variable of type uint8.
- UCSR0A and UDR0 are variables defined in a header.
- They refer to memory-mapped registers in the UART (Universal Asynchronous Receiver-Transmitter)

EECS 149/249A, UC Berkeley: 9

Send a Sequence of Bytes

```
for(i = 0; i < 8; i++) {  
    while(!(UCSR0A & 0x20));  
    UDR0 = x[i];  
}
```

How long will this take to execute? Assume:

- 57600 baud serial speed.
- $8/57600 = 139$ microseconds.
- Processor operates at 18 MHz.

Each for loop iteration will consume about 2502 cycles.

EECS 149/249A, UC Berkeley: 10

Receiving via UART

Again, on an Atmel AVR:

```
while(!(UCSR0A & 0x80));  
return UDR0;
```

- Wait until the UART has received an incoming byte.
- *The programmer must ensure there will be one!*
- If reading a sequence of bytes, how long will this take?

Under the same assumptions as before, it will take about 2502 cycles to receive each byte.

EECS 149/249A, UC Berkeley: 11

Input Mechanisms in Software

○ Polling

- Main loop uses each I/O device periodically.
- If output is to be produced, produce it.
- If input is ready, read it.

○ Interrupts

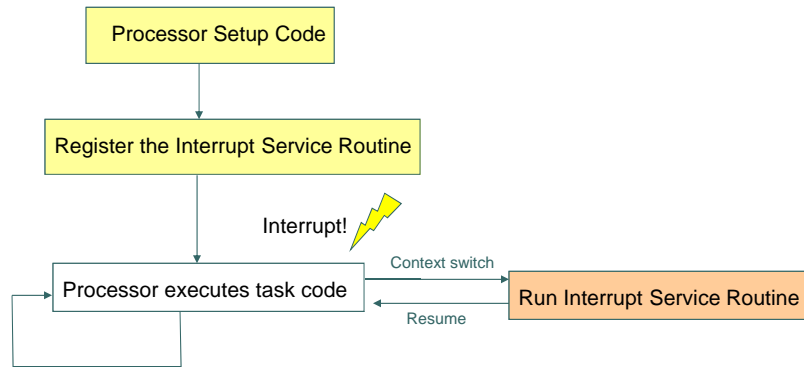
- External hardware alerts the processor that input is ready.
- Processor suspends what it is doing.
- Processor invokes an interrupt service routine (ISR).
- ISR interacts with the application concurrently.

EECS 149/249A, UC Berkeley: 12

Interrupts

- Interrupt Service Routine

Short subroutine that handles the interrupt



EECS 149/249A, UC Berkeley: 13

Interrupts

The most typical and general program setup for the Reset and Interrupt Vector Addresses in ATmega168 is:

Address	Labels	Code	Comments
0x0000	jmp	RESET	; Reset Handler
0x0002	jmp	EXT_INT0	; IRQ0 Handler
0x0004	jmp	EXT_INT1	; IRQ1 Handler
0x0006	jmp	PCINT0	; PCINT0 Handler
0x0008	jmp	PCINT1	; PCINT1 Handler
0x000A	jmp	PCINT2	; PCINT2 Handler
0x000C	jmp	WDT	; Watchdog Timer Handler
0x000E	jmp	TIM2_COMPA	; Timer2 Compare A Handler
0x0010	jmp	TIM2_COMPB	; Timer2 Compare B Handler
0x0012	jmp	TIM2_OVF	; Timer2 Overflow Handler
0x0014	jmp	TIM1_CAPT	; Timer1 Capture Handler

Program memory addresses,
not data memory addresses.

Triggers:

- A level change on an interrupt request pin
- Writing to an interrupt pin configured as an output (“software interrupt”)

Source: ATmega168 Reference Manual

Responses:

- Disable interrupts.
- Push the current program counter onto the stack.
- Execute the instruction at a designated address in the flash memory.

Design of interrupt service routine:

- Save and restore any registers it uses.
- Re-enable interrupts before returning from interrupt.

EECS 149/249A, UC Berkeley: 14

Berkeley Microblaze Personality Memory Map

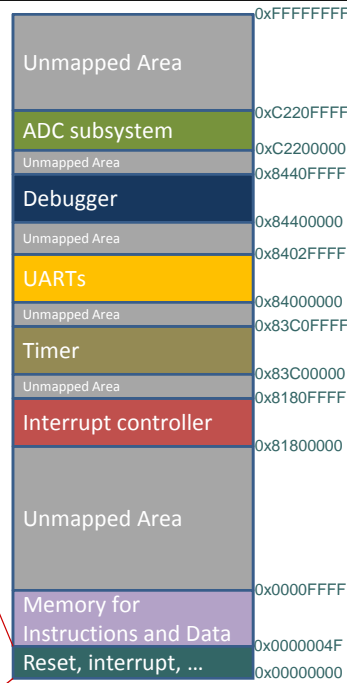
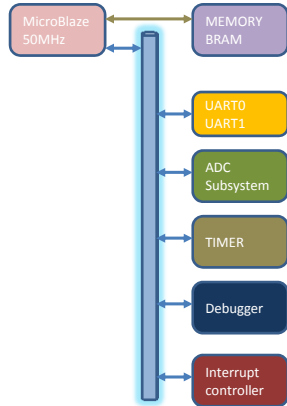


Table 3-4: Interrupt and Exception Handling

On	Hardware jumps to	Software Labels
Start / Reset	0x0	_start
User exception	0x8	_exception_handler
Interrupt	0x10	_interrupt_handler
Break (HW/SW)	0x18	-
Hardware exception	0x20	_hw_exception_handler
Reserved by Xilinx for future use	0x28 - 0x4F	-

EECS 149/249A, UC Berkeley: 15

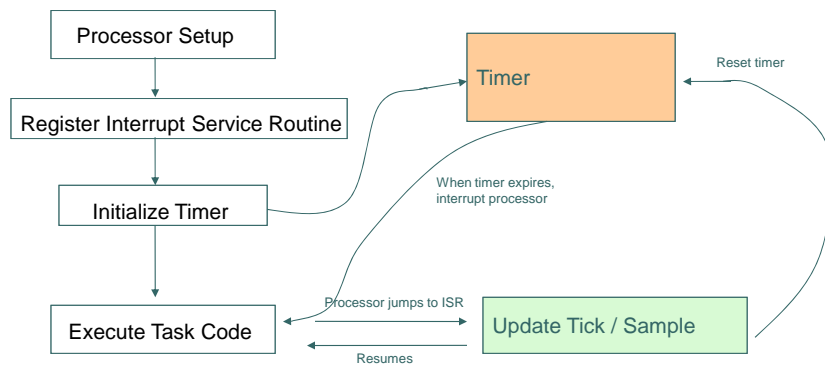
Microblaze Interrupt Policy

- “The interrupt return address (the PC associated with the instruction in the decode stage at the time of the interrupt) is automatically loaded into general purpose register R14.”
- “In addition, the processor also disables future interrupts by clearing the IE [Interrupt Enable] bit in the MSR [Machine Status Register]. The IE bit is automatically set again when executing the RTID instruction.”

Source: Microblaze datasheet

EECS 149/249A, UC Berkeley: 16

Timed Interrupt



EECS 149/249A, UC Berkeley: 17

Example: Set up a timer on an ATmega168 to trigger an interrupt every 1ms.

The frequency of the processor in the command module is 18.432 MHz.

1. Set up an interrupt to occur once every millisecond. Toward the beginning of your program, set up and enable the timer1 interrupt with the following code:

```

TCCR1A = 0x00;
TCCR1B = 0x0C;
OCR1A = 71;
TIMSK1 = 0x02;
  
```

The first two lines of the code put the timer in a mode in which it generates an interrupt and resets a counter when the timer value reaches the value of OCR1A, and select a prescaler value of 256, meaning that the timer runs at 1/256th the speed of the processor. The third line sets the reset value of the timer. To generate an interrupt every 1ms, the interrupt frequency will be 1000 Hz. To calculate the value for OCR1A, use the following formula:

```

OCR1A = (processor_frequency / (prescaler *
interrupt_frequency)) - 1
OCR1A = (18432000 / (256 * 1000)) - 1 = 71
  
```

The fourth line of the code enables the timer interrupt. See the ATmega168 datasheet for more information on these control registers.

- o TCCR: Timer/Counter Control Register
- o OCR: output compare register
- o TIMSK: Timer Interrupt Mask

The “prescaler” value divides the system clock to drive the timer.

Setting a non-zero bit in the timer interrupt mask causes an interrupt to occur when the timer resets.

Source: iRobot Command Module Reference Manual v6

EECS 149/249A, UC Berkeley: 18

Setting up the timer interrupt hardware in C

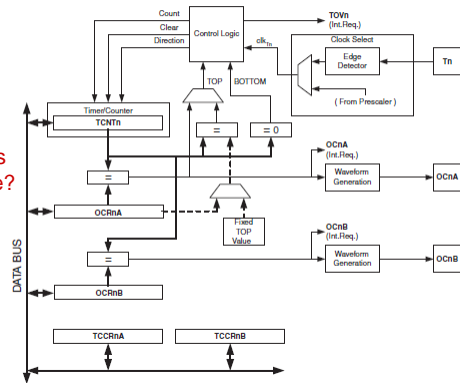
```
#include <avr/io.h>
```

```
int main (void) {
    TCCR1A = 0x00;
    TCCR1B = 0x0C;
    OCR1A = 71;
    TIMSK1 = 0x02;
    ...
}
```

memory-mapped register.
But how is this proper C code?

This code sets the hardware up to trigger an interrupt every 1ms. How do we handle the interrupt?

Figure 16-1. 8-bit Timer/Counter Block Diagram



Source: ATmega168 Reference Manual

```
#define _MMIO_BYTE(mem_addr) (*(volatile uint8_t *) (mem_addr))
#define _SFR_IO8(io_addr) _MMIO_BYTE((io_addr) + 0x20)
#define _SFR_MEM8(mem_addr) _MMIO_BYTE(mem_addr)
#define _BV(bit) (1 << (bit))
```

```
//Timer defines (iomx8.h)
#define TCCR1A _SFR_MEM8 (0x80)
#define TCCR1B _SFR_MEM8 (0x81)
/* TCCR1B */
#define WGM12 3
#define CS12 2
```

```
//Enable interrupts (interrupt.h)
#define sei() __asm__ __volatile__ ("sei" ::)
//Disable interrupts (interrupt.h)
#define cli() __asm__ __volatile__ ("cli" ::)
#define SIGNAL(signame)
void signame (void) __attribute__((signal)); \
void signame (void)
```

VALUE	USER DEFINED NAME
SEI	Global Interrupt Enable
CLI	Global Interrupt Disable

```
void initialize(void) {
    cli();

    // Set I/O pins
    DDRB = 0x10;
    PORTB = 0xCF;
    .....

    // Set up timer 1 to generate an interrupt every 1 ms
    TCCR1A = 0x00;
    TCCR1B = (_BV(WGM12) | _BV(CS12));
    OCR1A = 71;
    TIMSK1 = _BV(OCIE1A);

    // Set up the serial port with rx interrupt
    .....

    // Turn on interrupts
    sei();
}
```

```
// Global variables
volatile uint16_t timer_cnt = 0;
volatile uint8_t timer_on = 0;

// Timer 1 interrupt to time delays in ms
SIGNAL(SIG_OUTPUT_COMPARE1A) {
    if(timer_cnt) {
        timer_cnt--;
    } else {
        timer_on = 0;
    }
}
```

```
void delayMs(uint16_t time_ms) {
    timer_on = 1;
    timer_cnt = time_ms;
    while(timer_on);
}
```

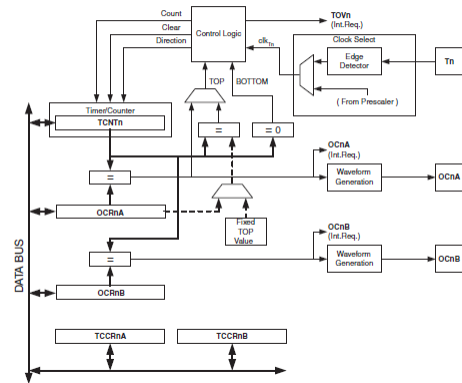
Setting up the timer interrupt hardware in C

```
#include <avr/io.h>

int main (void) {
    TCCR1A = 0x00;
    TCCR1B = 0x0C;
    OCR1A = 71;
    TIMSK1 = 0x02;
    ...
}

(*(volatile uint8_t *) (0x80)) = 0x00;
```

Figure 16-1. 8-bit Timer/Counter Block Diagram



Source: ATmega168 Reference Manual

EECS 149/249A, UC Berkeley: 21

Example 2: Set up a timer on a Luminary Micro board to trigger an interrupt every 1ms.

```
// Setup and enable SysTick with interrupt every 1ms
void initTimer(void) {
    SysTickPeriodSet(SysCtlClockGet() / 1000);
    SysTickEnable();
    SysTickIntEnable();
}

// Disable SysTick
void disableTimer(void) {
    SysTickIntDisable();
    SysTickDisable();
}
```

Number of cycles per sec.

Start SysTick counter

Enable SysTick timer interrupt

Source: Stellaris Peripheral Driver Library User's Guide

EECS 149/249A, UC Berkeley: 22

Example: Do something for 2 seconds then stop

```
volatile uint timer_count;
void ISR(void) {
    timer_count--;
}
```

static variable: declared outside main() puts them in statically allocated memory (not on the stack)

```
int main(void) {
    // initialization code
    SysTickIntRegister(&ISR);
    ... // other init
    timer_count = 2000;
    while(timer_count != 0) {
        ... code to run for 2 seconds
    }
}
```

volatile: C keyword to tell the compiler that this variable may change at any time, not (entirely) under the control of this program.

Interrupt service routine

Registering the ISR to be invoked on every SysTick interrupt

EECS 149/249A, UC Berkeley: 23

Concurrency

```
volatile uint timer_count;
void ISR(void) {
    timer_count--;
}
```

```
int main(void) {
    // initialization code
    SysTickIntRegister(&ISR);
    ... // other init
    timer_count = 2000;
    while(timer_count != 0) {
        ... code to run for 2 seconds
    }
}
```

concurrent code: logically runs at the same time. In this case, between any two machine instructions in main() an interrupt can occur and the upper code can execute.

What could go wrong?


EECS 149/249A, UC Berkeley: 24

Concurrency

```
volatile uint timer_count;
void ISR(void) {
    timer_count--;
}

int main(void) {
    // initialization code
    SysTickIntRegister(&ISR);
    ... // other init
    timer_count = 2000;
    while(timer_count != 0) {
        ... code to run for 2 seconds
    }
}
```

what if the interrupt
occurs twice during
the execution of this
code?



What could go wrong?

EECS 149/249A, UC Berkeley: 25

Improved Example

```
volatile uint timer_count = 0;
void ISR(void) {
    if(timer_count != 0) {
        timer_count--;
    }
}

int main(void) {
    // initialization code
    SysTickIntRegister(&ISR);
    ... // other init
    timer_count = 2000;
    while(timer_count != 0) {
        ... code to run for 2 seconds
    }
}
```

EECS 149/249A, UC Berkeley: 26

Reasoning about concurrent code

```
volatile uint timer_count = 0;
void ISR(void) {
    if(timer_count != 0) {
        timer_count--;
    }
}
int main(void) {
    // initialization code
    SysTickIntRegister(&ISR);
    ... // other init
    timer_count = 2000;
    while(timer_count != 0) {
        ... code to run for 2 seconds
    }
}
```

can an interrupt occur here? If it can, what happens?

EECS 149/249A, UC Berkeley: 27

Issues to Watch For

- Interrupt service routine execution time
- Context switch time
- Nesting of higher priority interrupts
- Interactions between ISR and the application
- Interactions between ISRs
- ...

EECS 149/249A, UC Berkeley: 28

A question:

What's the difference between

Concurrency
and
Parallelism

EECS 149/249A, UC Berkeley: 29

Concurrency and Parallelism

A program is said to be **concurrent** if different parts of the program *conceptually execute simultaneously*.

A program is said to be **parallel** if different parts of the program *physically execute simultaneously* on distinct hardware.

A parallel program is concurrent, but a concurrent program need not be parallel.

EECS 149/249A, UC Berkeley: 30

Concurrency in Computing

- Interrupt Handling
 - Reacting to external events (interrupts)
 - Exception handling (software interrupts)
- Processes
 - Creating the illusion of simultaneously running different programs (multitasking)
- Threads
 - How is a thread different from a process?
- Multiple processors (multi-cores)
- ...

EECS 149/249A, UC Berkeley: 31

Summary

Interrupts introduce a great deal of nondeterminism into a computation. Very careful reasoning about the design is necessary.

EECS 149/249A, UC Berkeley: 32