Introduction to Embedded Systems

Edward A. Lee
UC Berkeley
EECS 149/249A
Fall 2016

Chapter 10: Input and Output, Interrupts

Connecting the Analog and Digital Worlds

Semantic mismatch:

Cyber:
- Digital
- Discrete in time
- Sequential

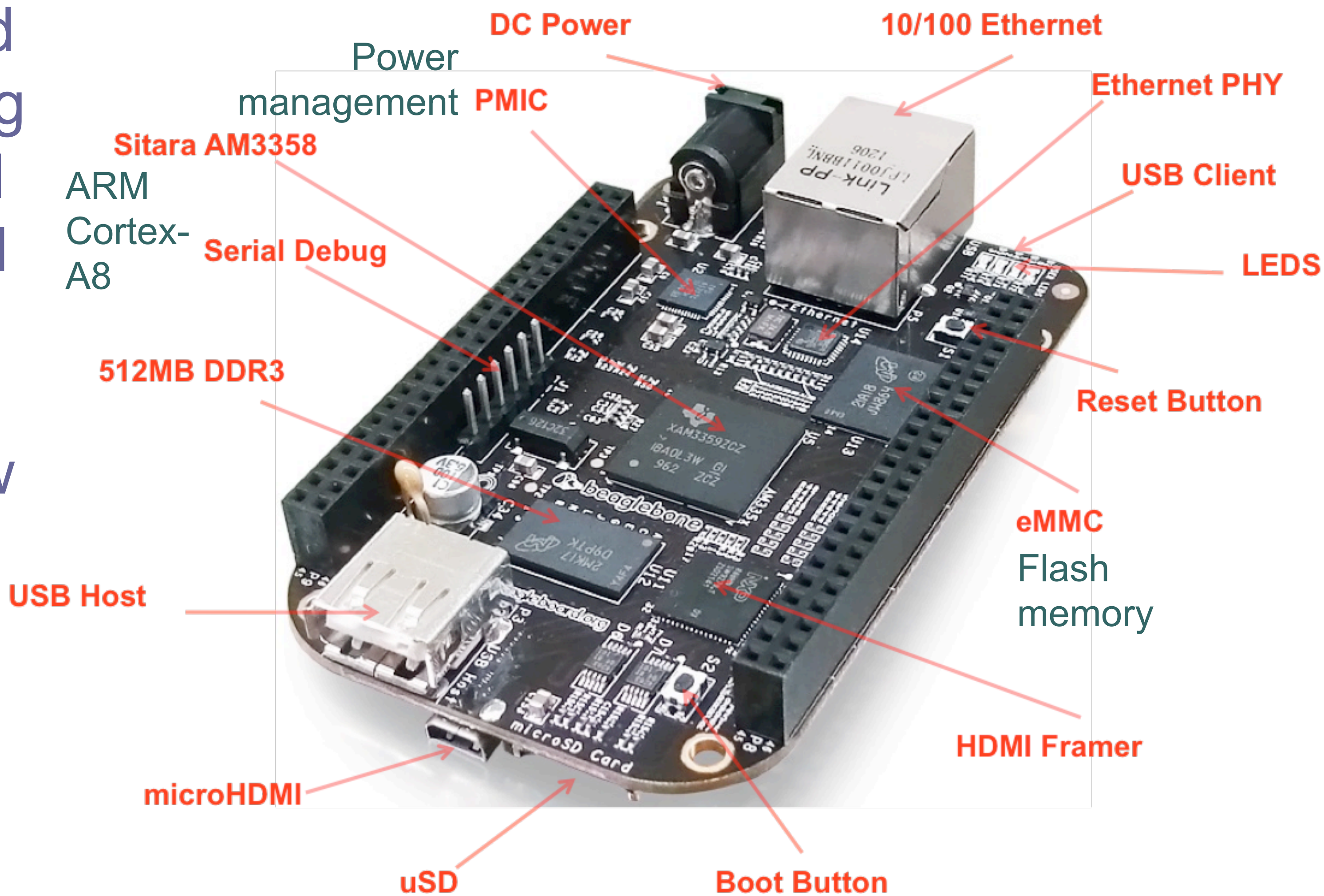
Physical:
- Continuum
- Continuous in time
- Concurrent

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)

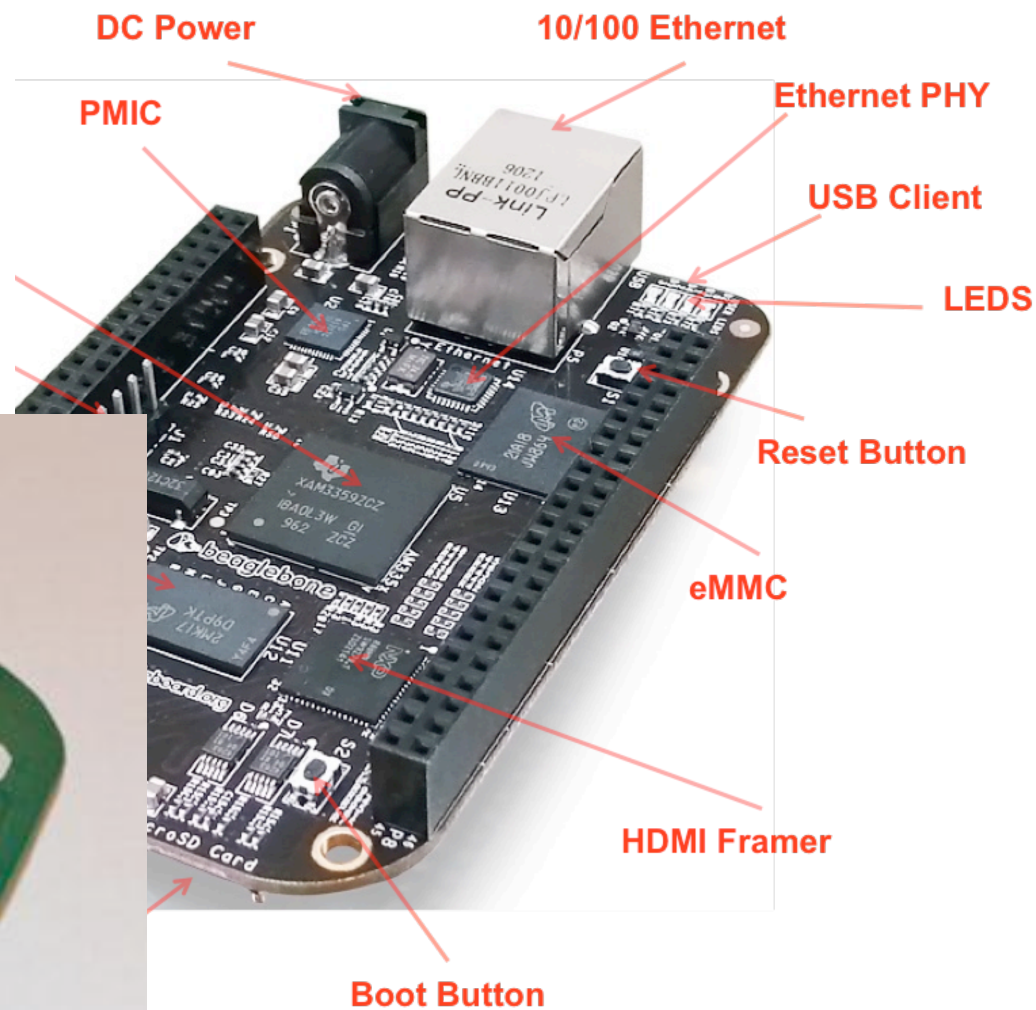
A Typical Microcomputer Board
Beaglebone Black from Texas Instruments

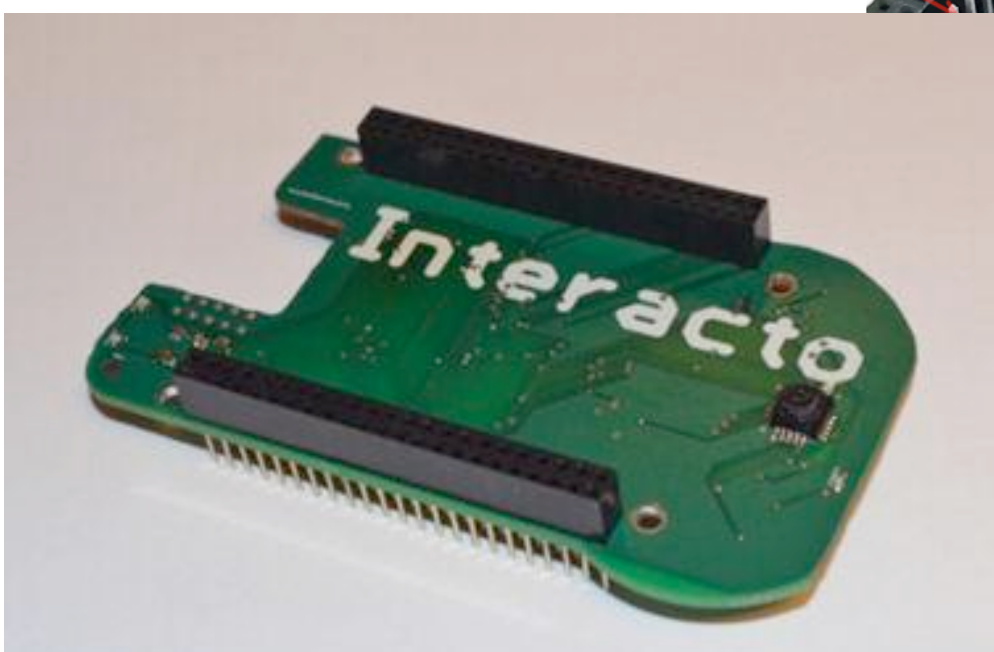
This board has analog and digital inputs and outputs. What are they? How do they work?



A Typical Microcomputer Board
Beaglebone Black from Texas Instruments

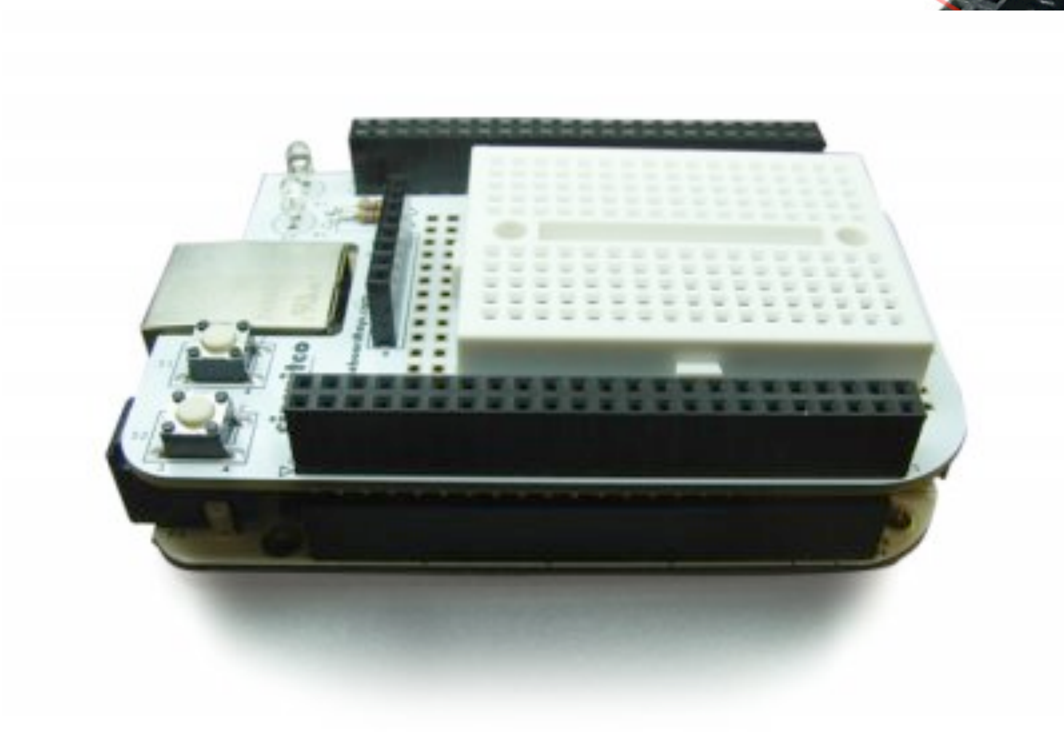
A “cape” is a daughter card that fits on the board. Arduino “shields” are similar. This one provides an accelerometer, gyro, and magnetometer.

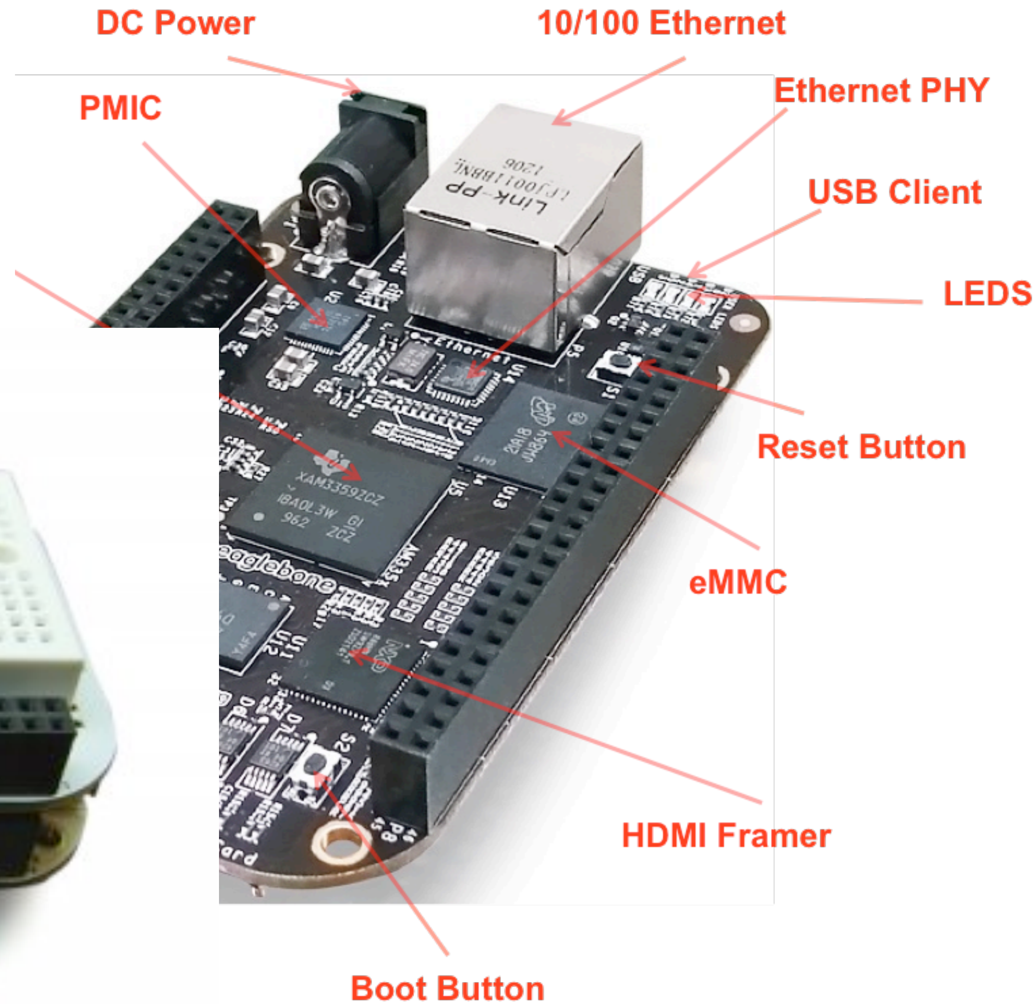




A Typical Microcomputer Board
Beaglebone Black from Texas Instruments

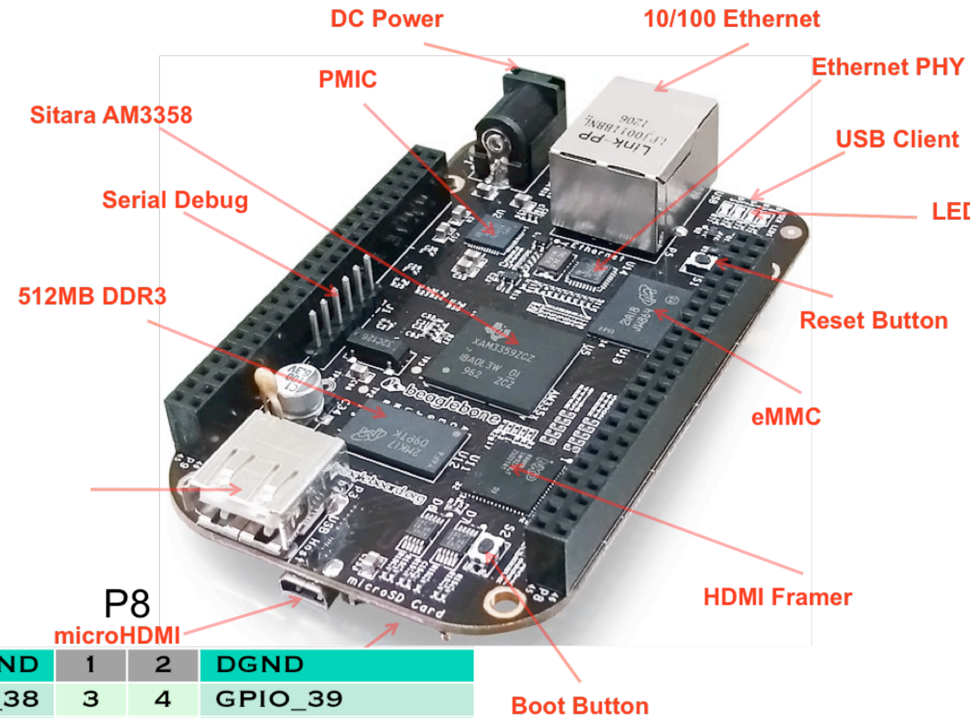
More interestingly, this one provides a protoboard to attach your own hardware. How to do that?





Beaglebone Black Header Configuration

One of eight configurations with SPI buses, analog I/O, etc.



P9

DGND	1	2	DGND
VDD_3V3	3	4	VDD_3V3
VDD_5V	5	6	VDD_5V
SYS_5V	7	8	SYS_5V
PWR_BUT	9	10	SYS_RESETN
GPIO_30	11	12	GPIO_60
GPIO_31	13	14	GPIO_40
GPIO_48	15	16	GPIO_51
SPI0_CS0	17	18	SPI0_D1
SPI1_CS1	19	20	SPI1_CS0
SPI0_D0	21	22	SPI0_SCLK
GPIO_49	23	24	GPIO_15
GPIO_117	25	26	GPIO_14
GPIO_125	27	28	SPI1_CS0
SPI1_D0	29	30	SPI1_D1
SPI1_SCLK	31	32	VDD_ADC
AIN4	33	34	GNDA_ADC
AIN6	35	36	AIN5
AIN2	37	38	AIN3
AIN0	39	40	AIN1
GPIO_20	41	42	SPI1_CS1
DGND	43	44	DGND
DGND	45	46	DGND

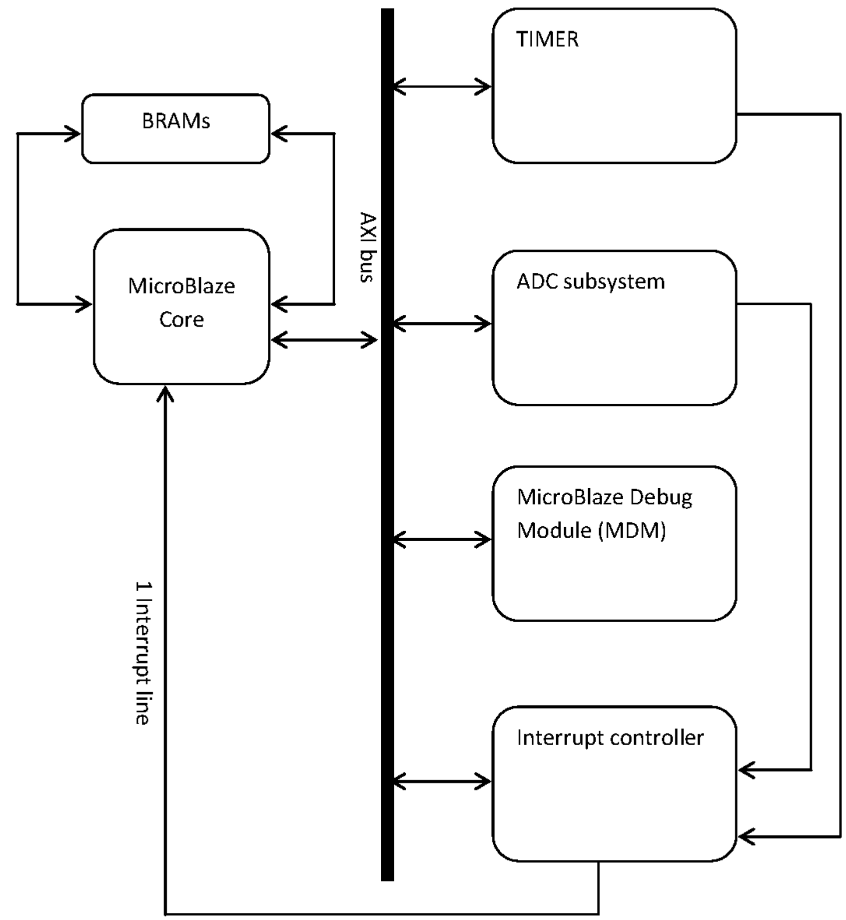
P8

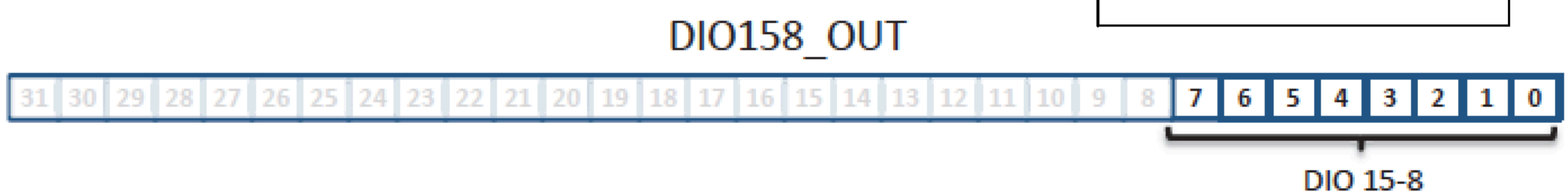
DGND	1	2	DGND
GPIO_38	3	4	GPIO_39
GPIO_34	5	6	GPIO_35
GPIO_66	7	8	GPIO_67
GPIO_69	9	10	GPIO_68
GPIO_45	11	12	GPIO_44
GPIO_23	13	14	GPIO_26
GPIO_47	15	16	GPIO_46
GPIO_27	17	18	GPIO_65
GPIO_22	19	20	GPIO_63
GPIO_62	21	22	GPIO_37
GPIO_36	23	24	GPIO_33
GPIO_32	25	26	GPIO_61
GPIO_86	27	28	GPIO_88
GPIO_87	29	30	GPIO_89
GPIO_10	31	32	GPIO_11
GPIO_9	33	34	GPIO_81
GPIO_8	35	36	GPIO_80
GPIO_78	37	38	GPIO_79
GPIO_76	39	40	GPIO_77
GPIO_74	41	42	GPIO_75
GPIO_72	43	44	GPIO_73
GPIO_70	45	46	GPIO_71

Many GPIO pins can be reconfigured to be PWM drivers, timers, etc.

Memory-Mapped Peripherals on the "Berkeley Personality"

DIO158_OUT is a C preprocessor macro defined in a header file in your IDE project. It defines the memory address of this register.





Bit	Field	Access	Values
7 – 0	Digital Output	Write	myRIO DIO 15-8 output state

Figure 1.10: `DIO158_OUT` (DIO 15-8 Out) register. myRIO MXP Connector B pins 8-15. Bit 0 writes MXP Connector B DIO8, and bit 7 writes DIO15.

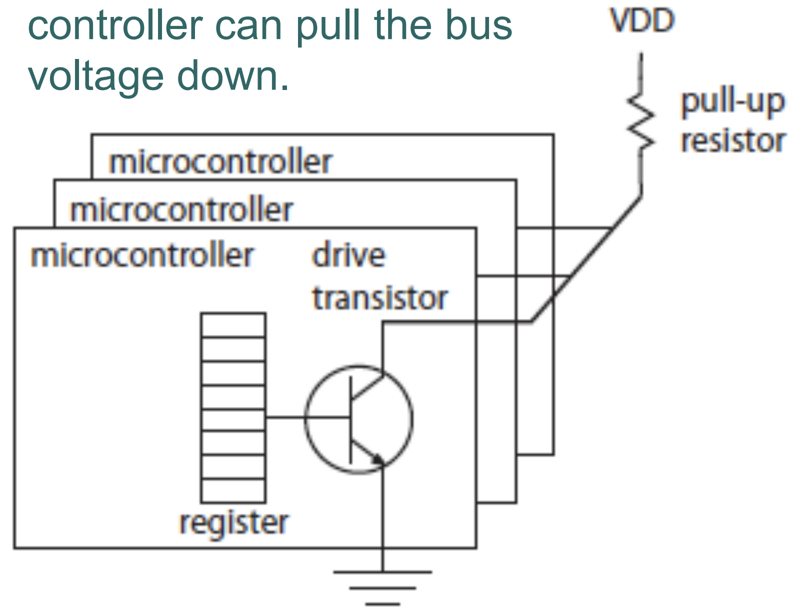
Simple Digital I/O: GPIO

Open collector circuits are often used on GPIO (general-purpose I/O) pins of a microcontroller.

The same pin can be used for input and output. And multiple users can connect to the same bus.

Why is the current limited?

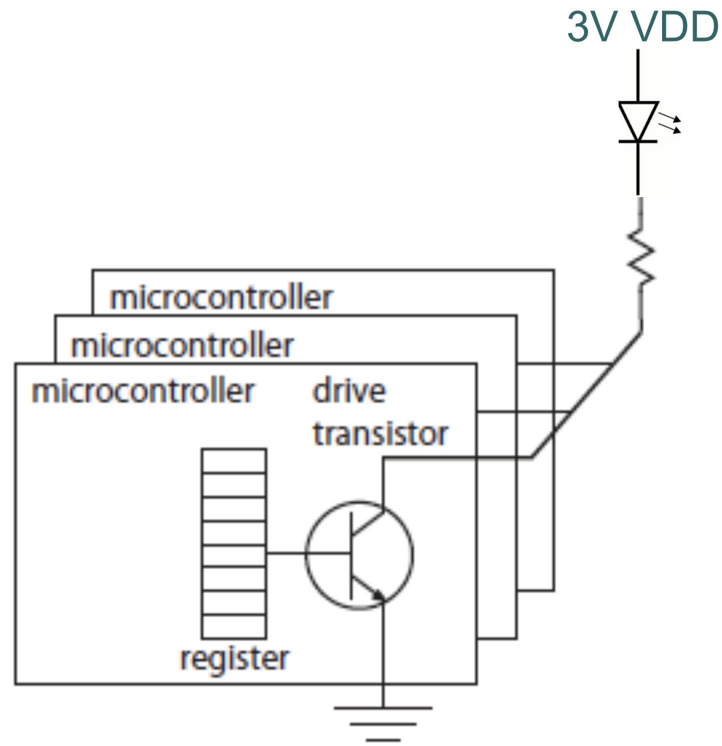
GPIO pins configured for bus output. Any one controller can pull the bus voltage down.



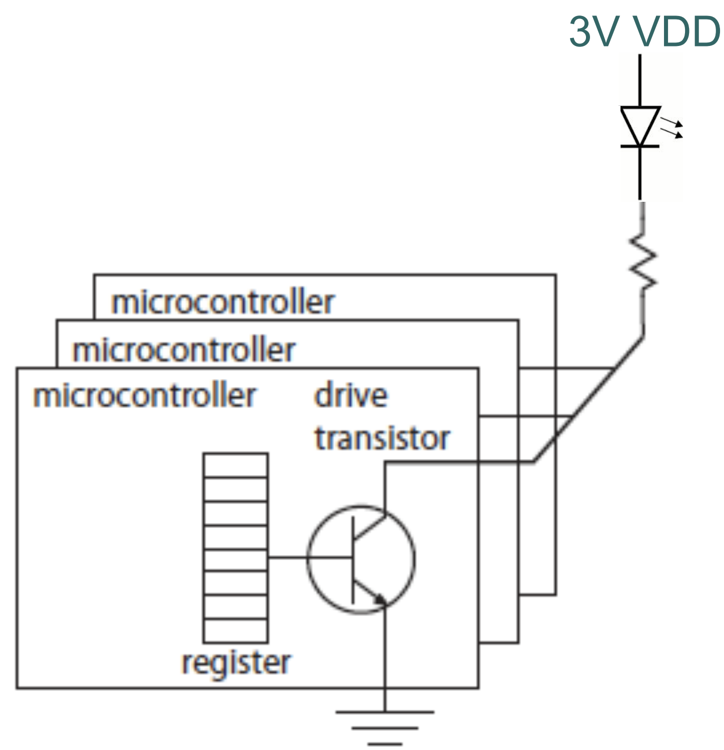
Example: Turn on an LED

Assume GPIO pins can sink up to 18 mA. Assume the LED, when forward biased (turned on), has a voltage drop of 2 volts.

What resistor should you use?



Example: Turn on an LED



Ohm's law:

$V = IR$

When LED is on, V = 1 volt.

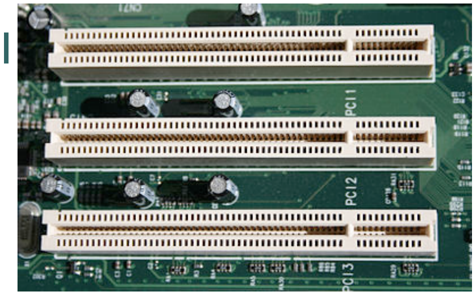
To limit to 18mA,

R ≥ 1/0.018 ≈ 56 ohms

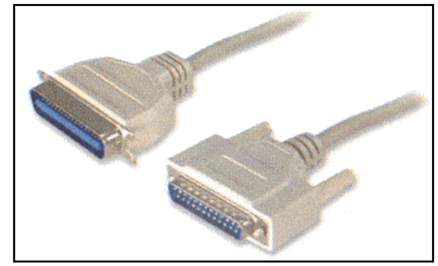
Wired Connections
Parallel vs. Serial Digital Interfaces

- 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
  - I^2C: Inter-Integrated Circuit
  - USB: Universal Serial Bus
  - SATA: Serial ATA
  - …
- Mixed (one or more "lanes")
  - PCIe: PCI Express

PCI



SCSI



USB

RS-232

Wired Connections
Parallel vs. Serial Digital Interfaces

Parallel connectors have been largely replaced by serial ones.

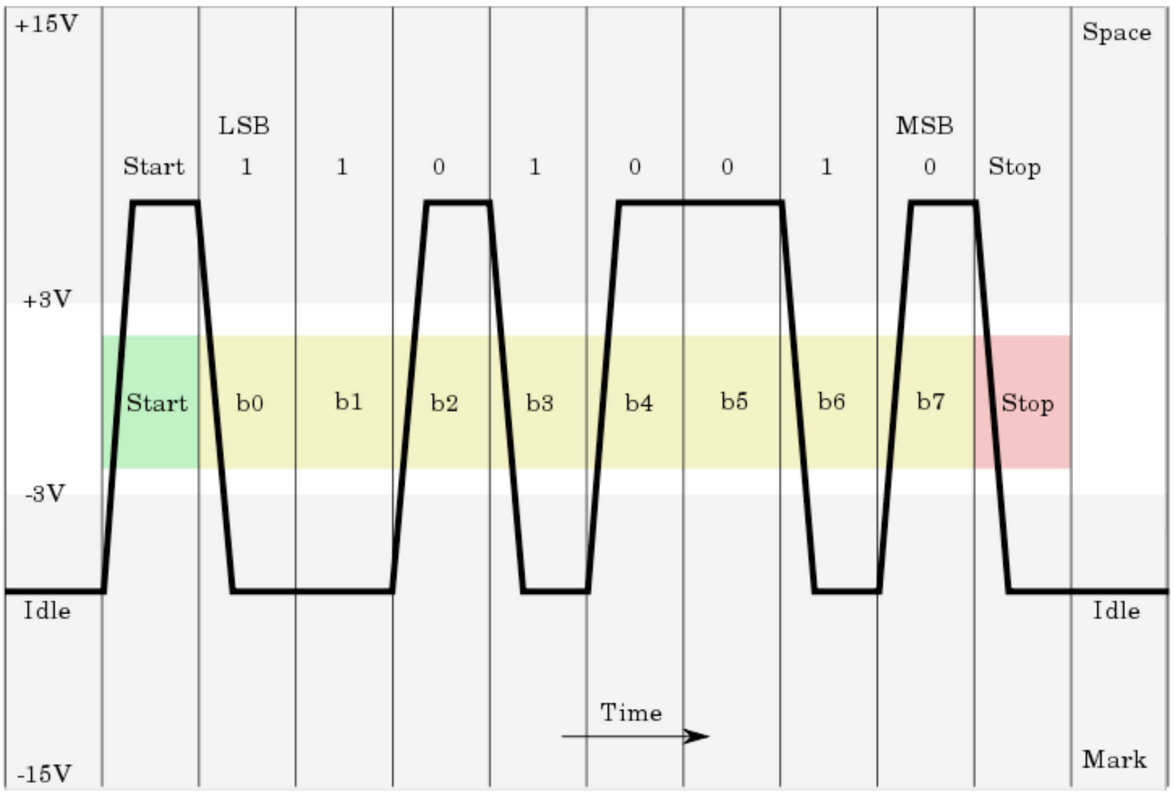
Why?

Serial Interfaces

The old but persistent RS-232 standard supports asynchronous serial connections (no common clock).

How does it work?

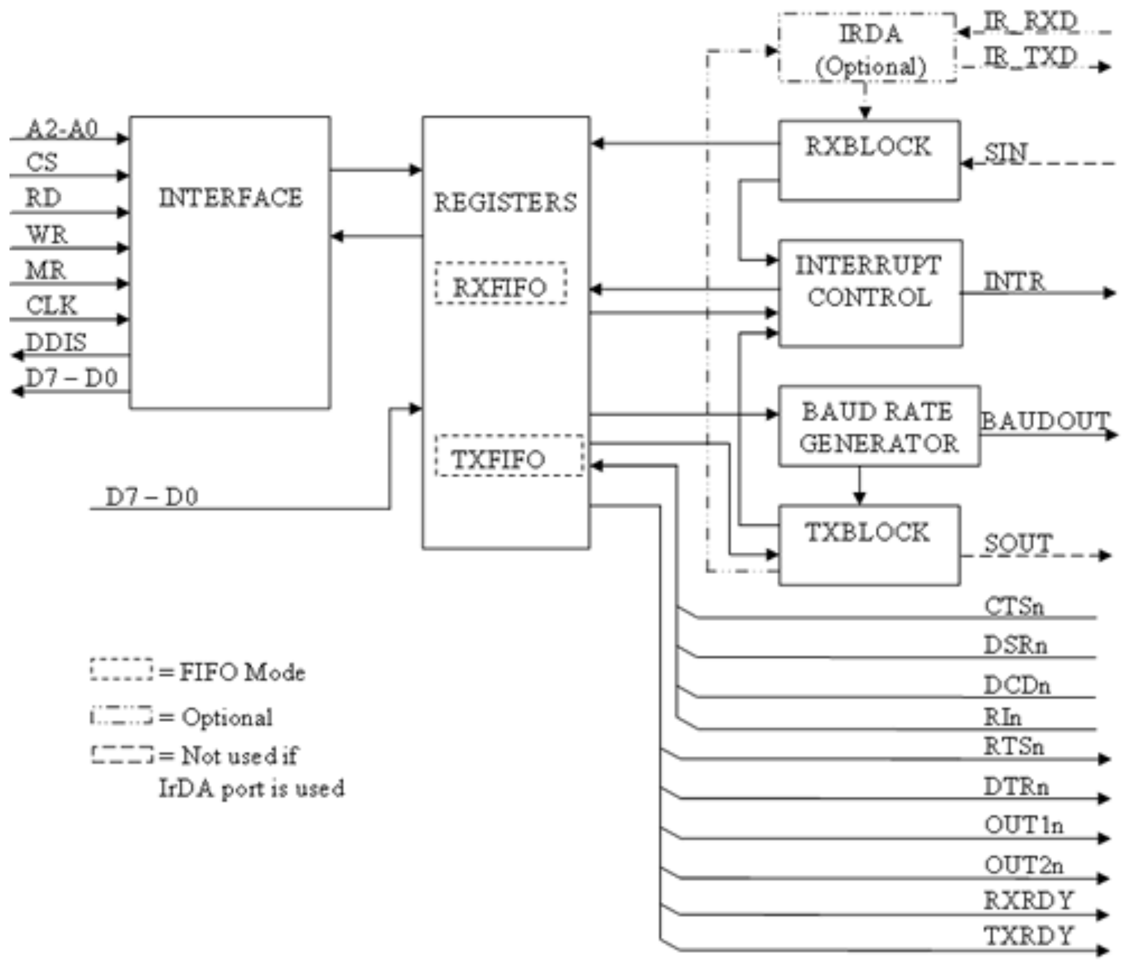
Many uses of RS-232 are being replaced by USB, which is electrically simpler but with a more complex protocol, or bluetooth, which is wireless.



Uppercase ASCII "K" character (0x4b) with 1 start bit, 8 data bits, 1 stop bit.

UART: Universal Asynchronous Receiver-Transmitter

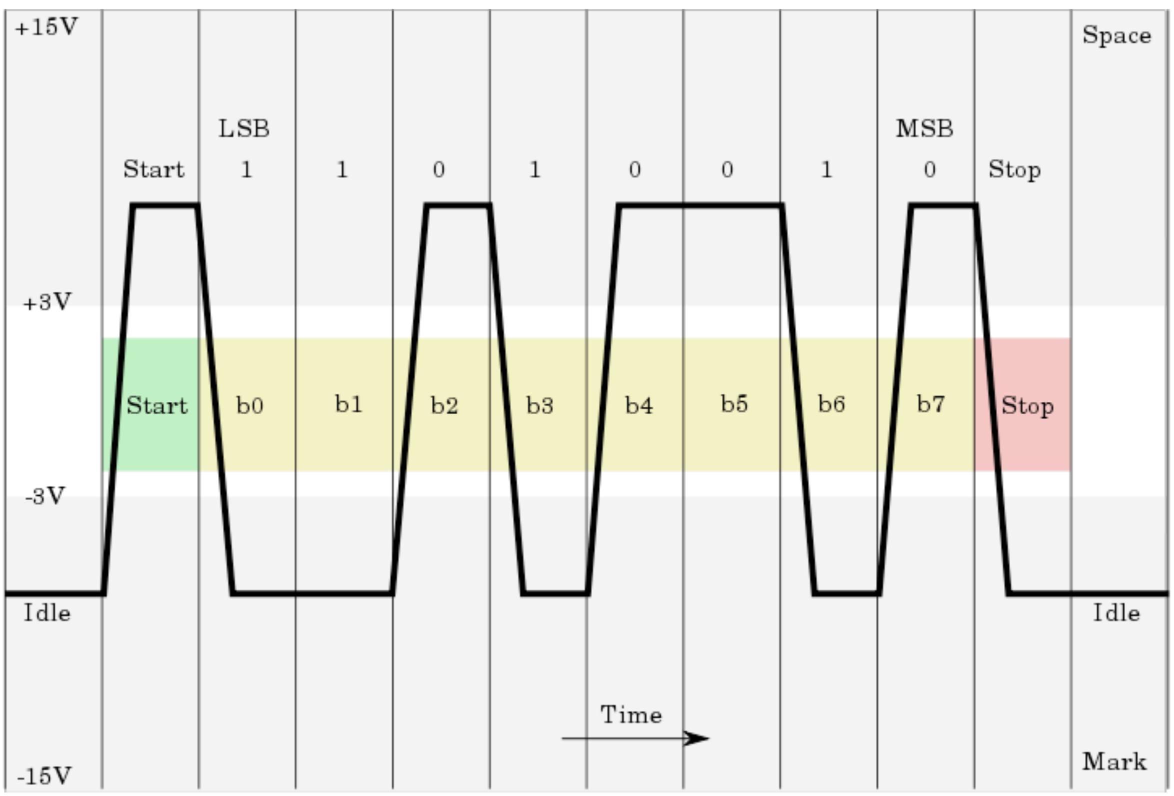
- Convert serial data to parallel data, and vice versa.
- Uses shift registers to load store data
- Can raise interrupt when data is ready
- Commonly used with RS-232 interface



Variant: USART: Universal Synchronous/Asynchronous Receiver-Transmitter

Speed Limitations

RS-232 relies on the clock in the transmitter being close enough in frequency to the clock on the receiver that upon detecting the start bit, it can just sample 8 more times and will see the remaining bits.



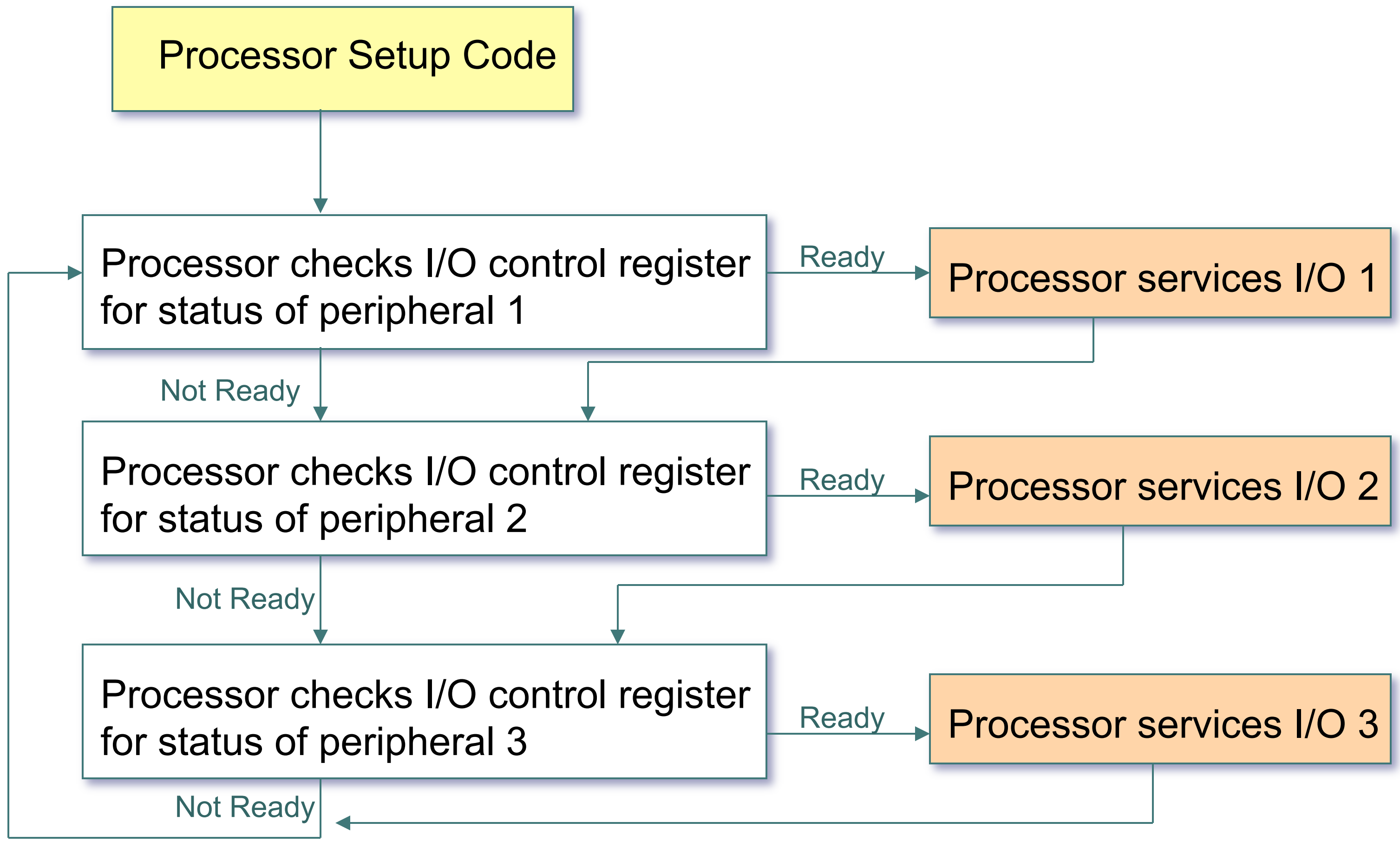
USB achieves higher speeds by beginning every packet with synchronization sequence of 8 bits. The receiver clock locks to this for the rest of the packet.

Input/Output 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.

Polling


Processor Setup Code
Processor checks I/O control register for status of peripheral 1
Ready
Processor services I/O 1
Not Ready
Processor checks I/O control register for status of peripheral 2
Ready
Processor services I/O 2
Not Ready
Processor checks I/O control register for status of peripheral 3
Ready
Processor services I/O 3
Not Ready

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)

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.

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.

Input Mechanisms in Software

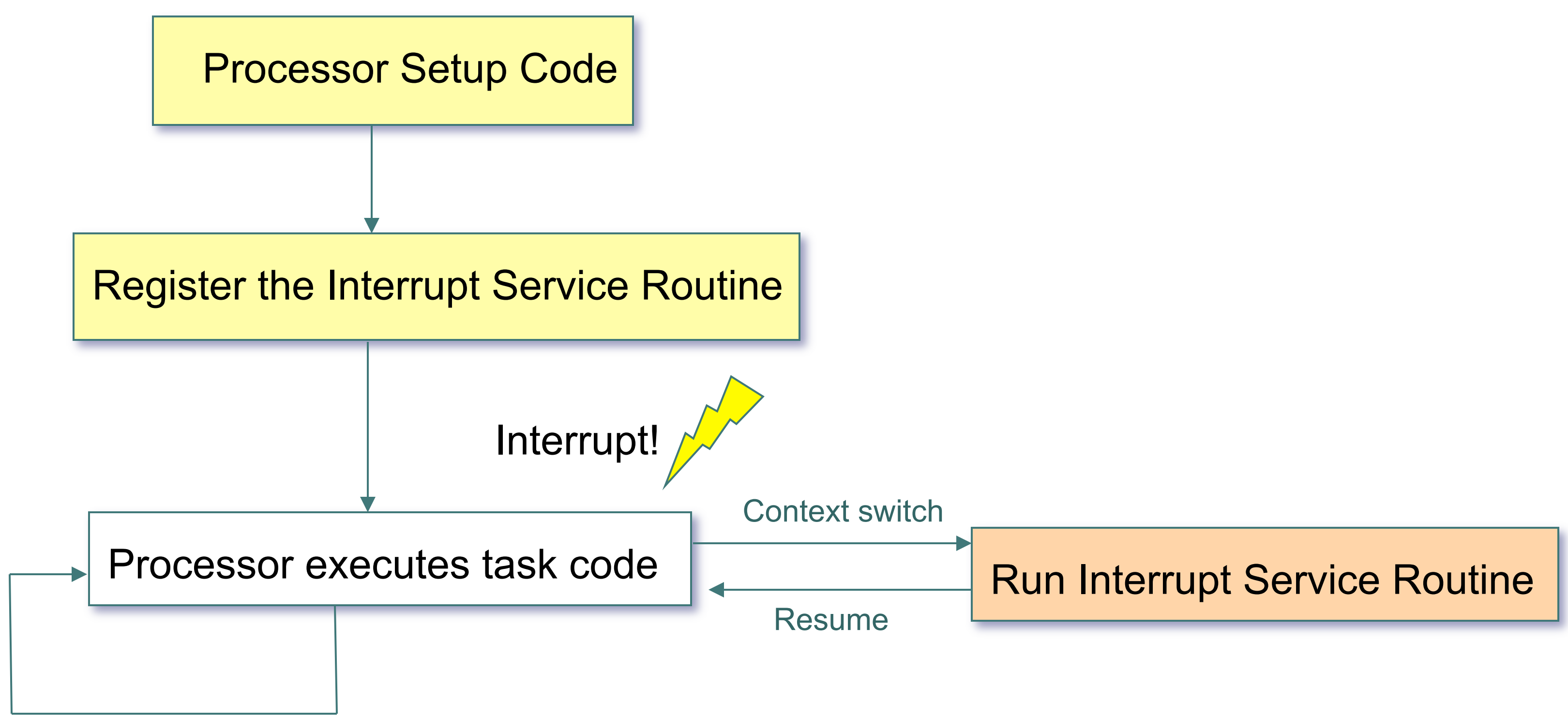
- Polling
  - Main loop uses each I/O device periodically.
  - If output is to be produced, produce it.
  - If input is ready, read it.

- 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.

Interrupts

- Interrupt Service Routine

  Short subroutine that handles the interrupt



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.

Source: ATmega168 Reference Manual

Triggers:

- A level change on an interrupt request pin
- Writing to an interrupt pin configured as an output ("software interrupt") or executing special instruction

Responses:

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

Design of interrupt service routine:

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

Berkeley Microblaze Personality Memory Map

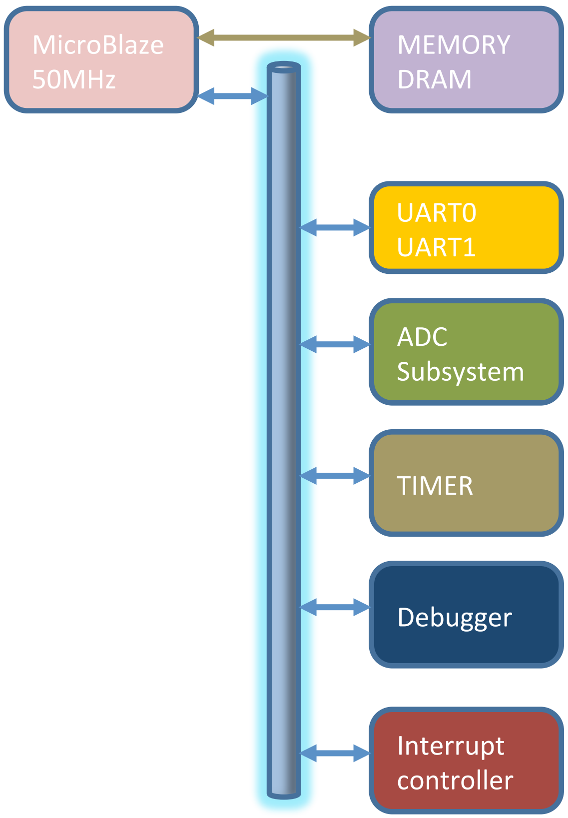
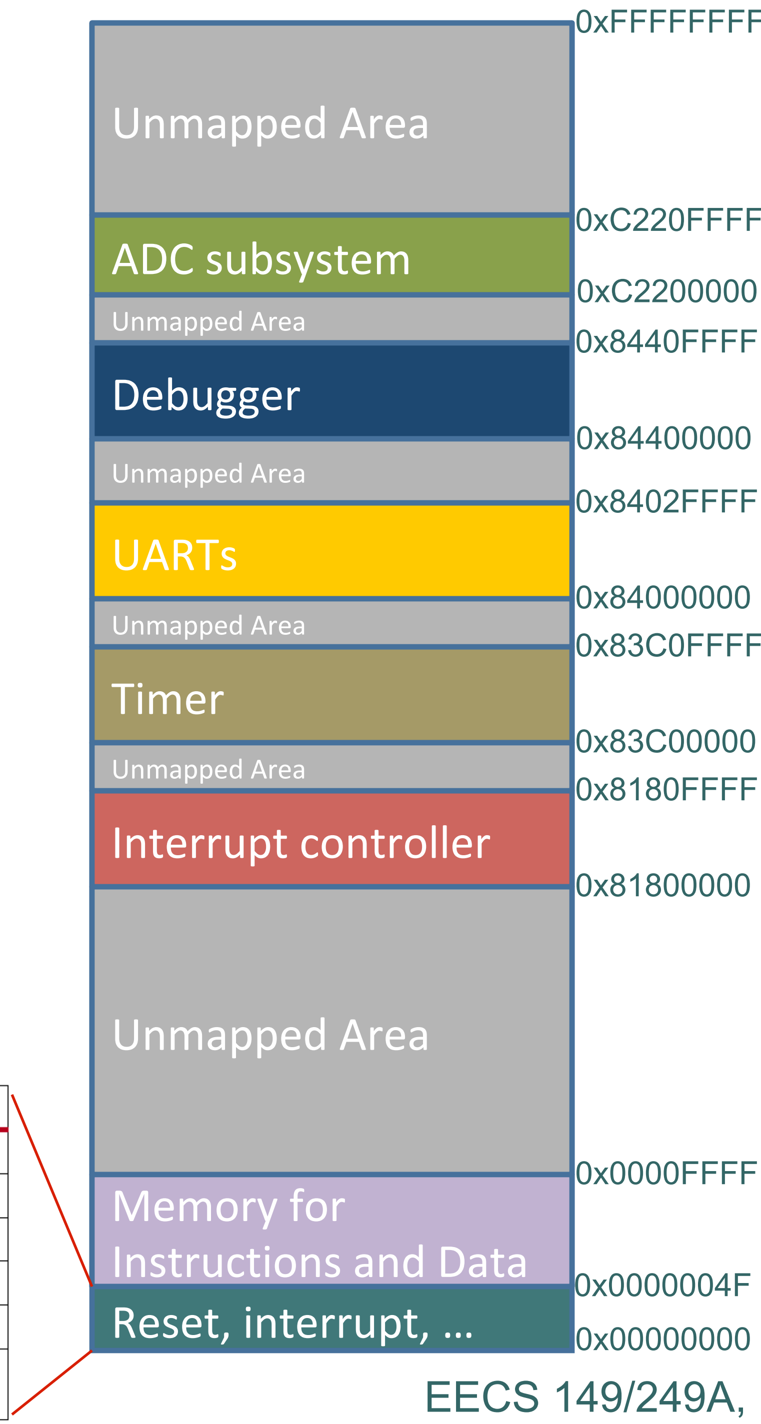


Table 3-4: **Interrupt and Exception Handling**

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	-



Microblaze Interrupt Policy

"MicroBlaze supports one external interrupt source (connected to the Interrupt input port). The processor only reacts to interrupts if the Interrupt Enable (IE) bit in the Machine Status Register (MSR) is set to 1. On an interrupt, the instruction in the execution stage completes while the instruction in the decode stage is replaced by a branch to the interrupt vector (address 0x10). 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 bit in the MSR. The IE bit is automatically set again when executing the RTID instruction."

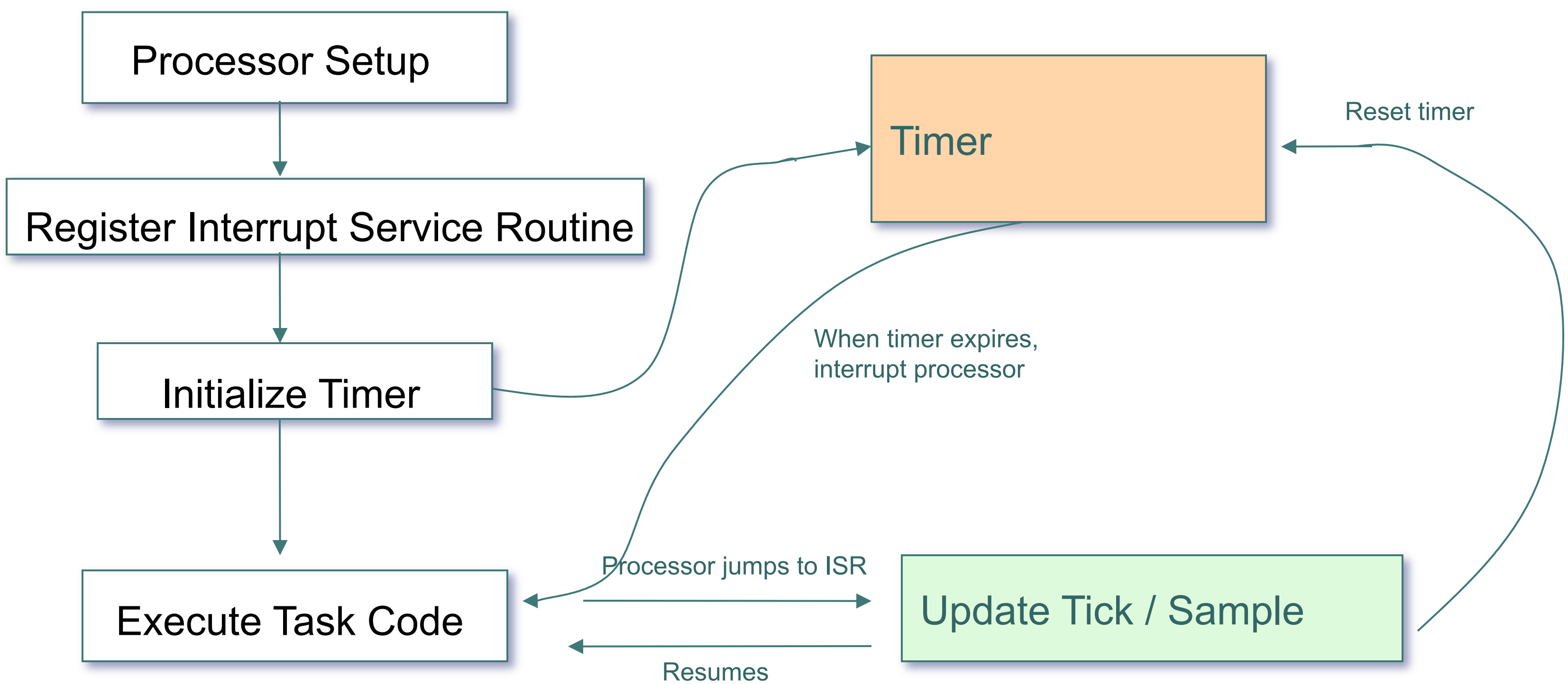
Source: Microblaze datasheet

Interrupts are Evil

[I]n one or two respects modern machinery is basically more difficult to handle than the old machinery. Firstly, we have got the interrupts, occurring at unpredictable and irreproducible moments; compared with the old sequential machine that pretended to be a fully deterministic automaton, this has been a dramatic change, and many a systems programmer's grey hair bears witness to the fact that we should not talk lightly about the logical problems created by that feature.

(Dijkstra, "The humble programmer" 1972)

Timed Interrupt



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.

- TCCR: Timer/Counter Control Register
- OCR: output compare register
- 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

Setting up the timer interrupt hardware in C

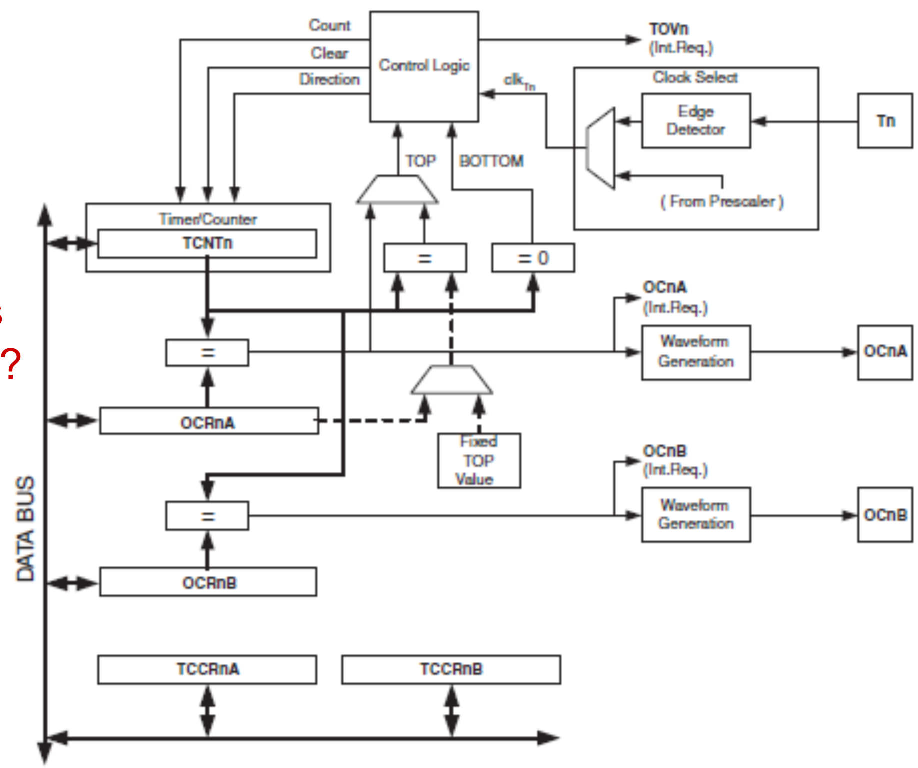
```
#include <avr/io.h>

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

memory-mapped register.

But how is this proper C code?

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



Source: ATmega168 Reference Manual

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

```
#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))
```

```
//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)
```

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

CLZ		Clear Zero Flag
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) ;
}
```

```
#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))
```

```
//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)
```

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

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) ;
}
```

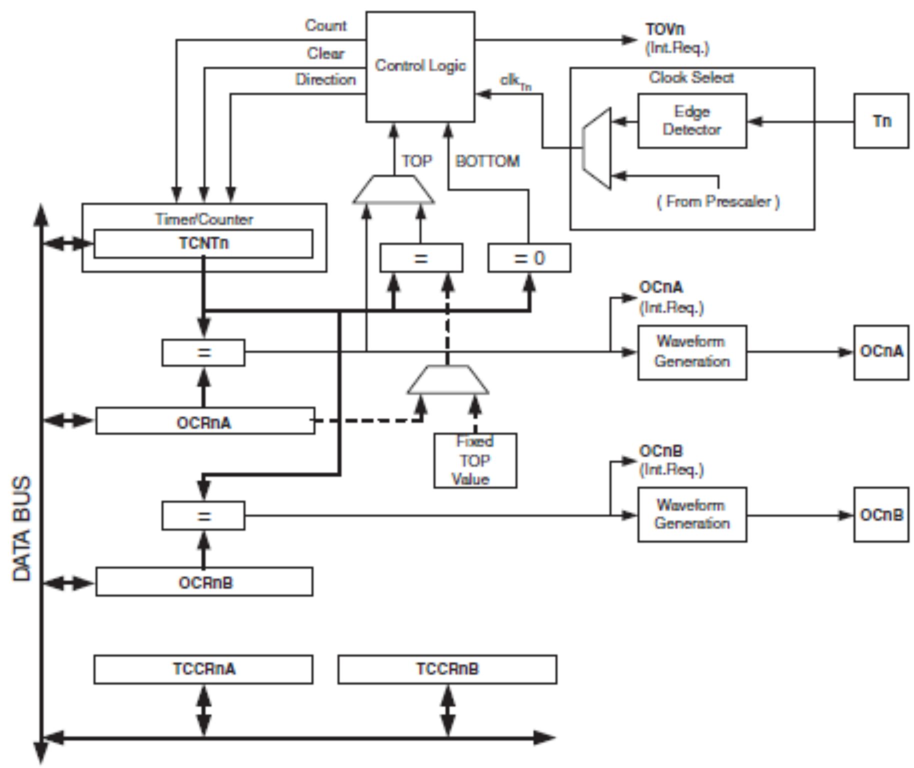
Setting up the timer interrupt hardware in C

```
#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

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

Example: Do something for 2 seconds then stop

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


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

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

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

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?

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?

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
  }
}
```

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?

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
- …

A question:

What's the difference between

Concurrency

and

Parallelism

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.

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)

. . .

Summary

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