Lecture 13: IO Hardware and Mechanisms

Seyed-Hosein Attarzadeh-Niaki

Based on the slides by Edward Lee

Review

- Network layers in embedded systems
- Wired and wireless networks

Connecting the Analog and Digital Worlds: Semantic Mismatch

Cyber
- Digital
- Discrete in time
- Sequential

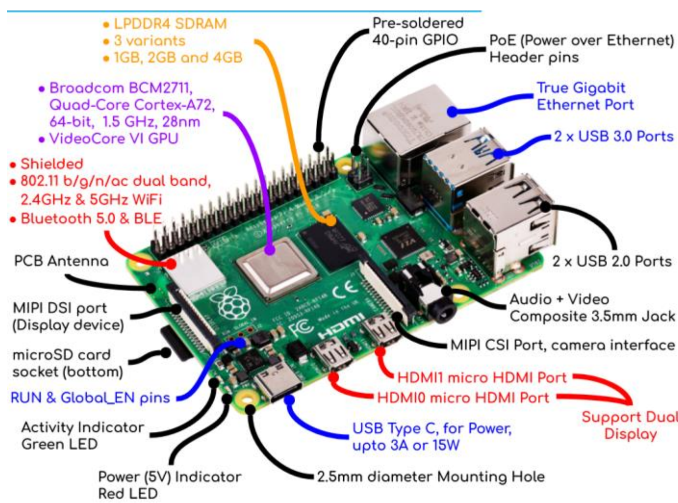
Physical
- Continuum
- Continuous in time
- Concurrent

Practical Issues

- Analog vs. digital
- Wired vs. wireless
- Serial vs. parallel
- Sampled or event triggered
- Bit rates
- Access control, security, authentication
- Physical connectors

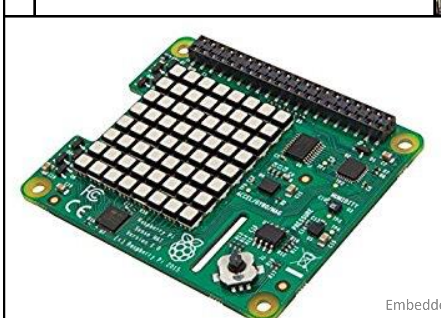
A Typical Microcomputer Board
Raspberry Pi 4

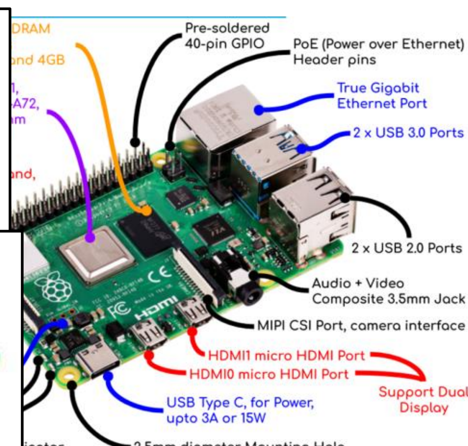
This board has analog and digital inputs and outputs. What are they? How do they work?



A Typical Microcomputer Board
Raspberry Pi 4

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

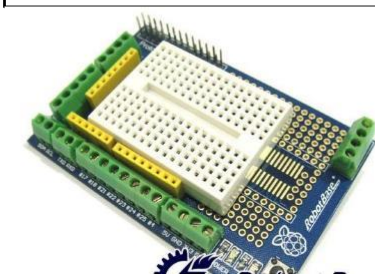


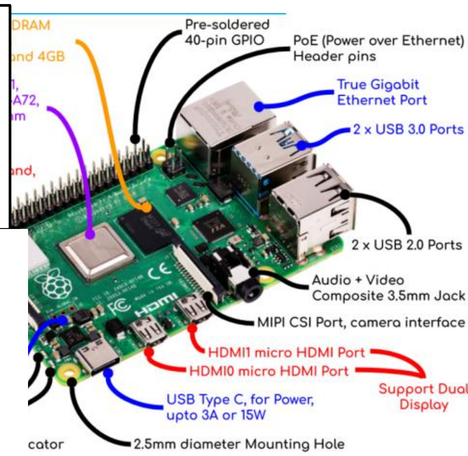


A Typical Microcomputer Board
Raspberry Pi 4

More interestingly, this one provides a protoshield to attach your own hardware.

How to do that?



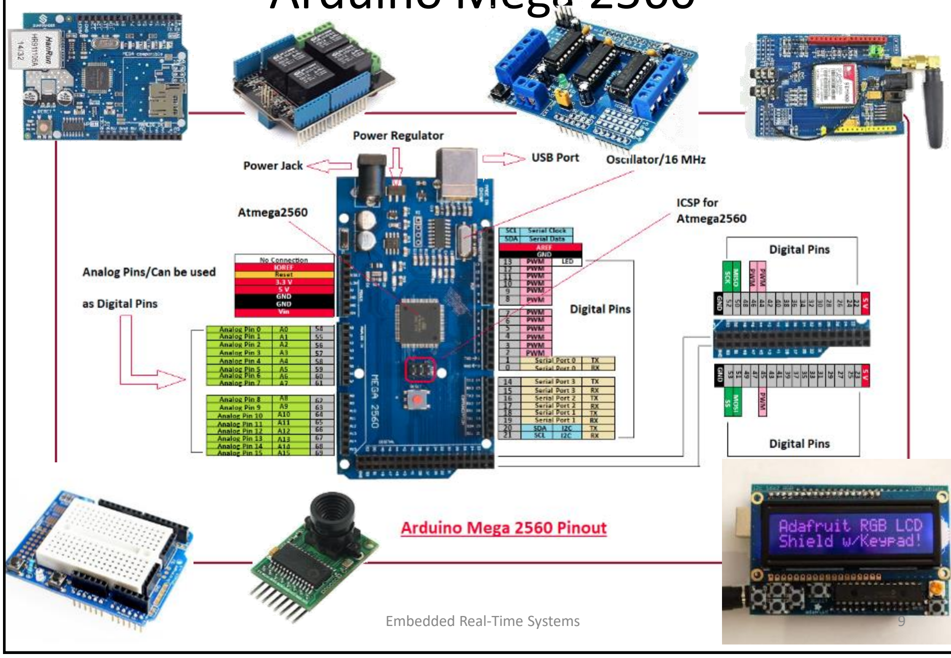


Raspberry Pi 4
Pin Layout

- One of many configurations with SPI buses, analog I/O, etc.
- Many GPIO pins can be reconfigured to be PWM drivers, timers, etc.

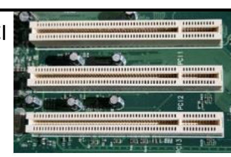
		Physical Pins			
Function	BCM	pin#	pin#	BCM	Function
3.3 Volts		1	2		5 Volts
GPIO/SDA1 (I2C)	2	3	4		5 Volts
GPIO/SCL1 (I2C)	3	5	6		GND
GPIO/GCLK	4	7	8	14	TX UART/GPIO
GND		9	10	15	RX UART/GPIO
GPIO	17	11	12	18	GPIO
GPIO	27	13	14		GND
GPIO	22	15	16	23	GPIO
3.3 Volts		17	18	24	GPIO
MOSI (SPI)	10	19	20		GND
MISO(SPI)	9	21	22	25	GPIO
SCLK(SPI)	11	23	24	8	CEO_N (SPI)
GND		25	26	7	CE1_N (SPI)
RESERVED		27	28		RESERVED
GPIO	5	29	30		GND
GPIO	6	31	32	12	GPIO
GPIO	13	33	34		GND
GPIO	19	35	36	16	GPIO
GPIO	26	37	38	20	GPIO
GND		39	40	21	GPIO

Arduino Mega 2560



Wired Connections
Parallel vs. Serial Digital Interfaces

PCI



Parallel (one wire per bit)

- ATA: Advanced Technology Attachment
- PCI: Peripheral Component Interface
- SCSI: Small Computer System Interface
- …

SCSI

Serial (one wire per direction)

- RS-232
- SPI: Serial Peripheral Interface bus
- I^{2}C: Inter-Integrated Circuit
- USB: Universal Serial Bus
- SATA: Serial ATA
- …

USB

Mixed (one or more "lanes")

- PCIe: PCI Express

RS-232

Input/Output Mechanisms in Sequential Software

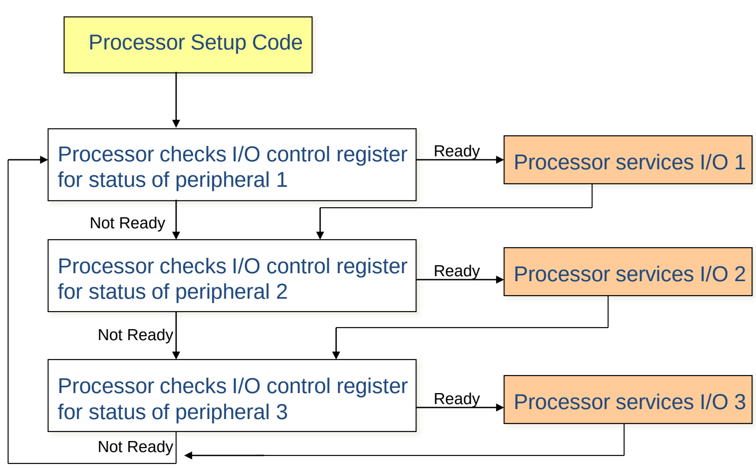
Polling

- 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



Example: Send a Sequence of Bytes

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

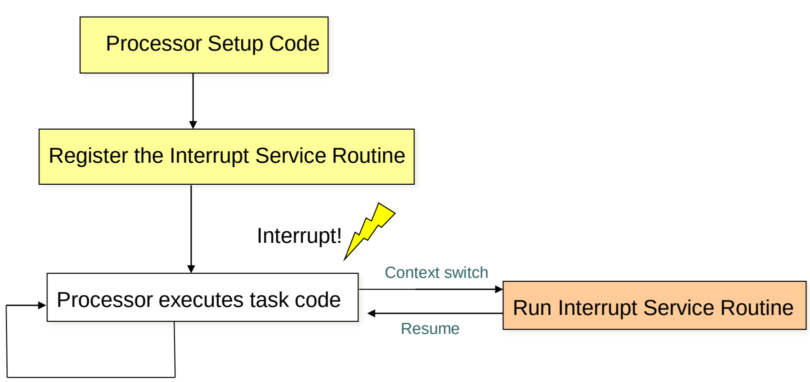
How long will this take to execute? Assume:

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

Each for loop iteration will consume about 2502 cycles.

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.

Embedded Real-Time Systems 15

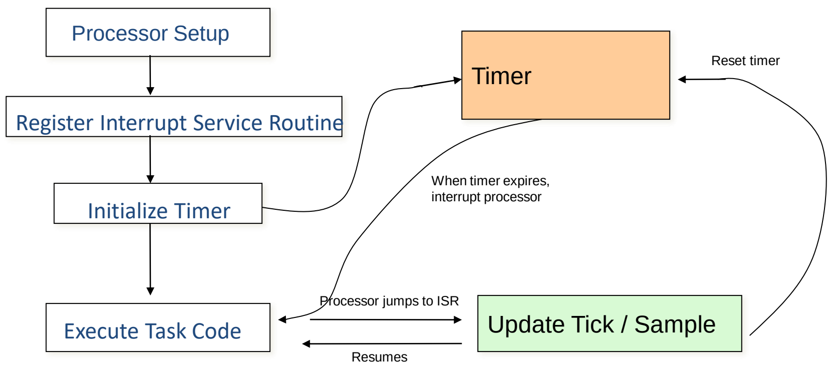
Interrupts are Evil

[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)

Embedded Real-Time Systems 16

Timed Interrupt



- A bad design pattern: put all the embedded code in ISRs invoked on (timed) interrupts.
- ISRs must be small.

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

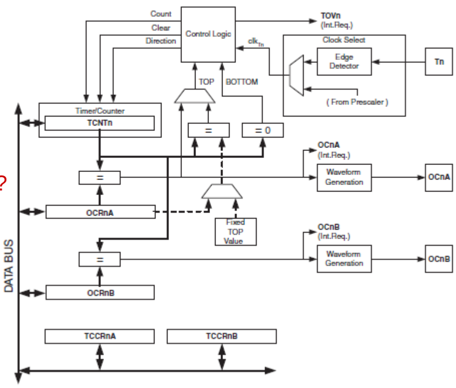
```c
#include <avr/io.h>

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



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

Source: ATmega168 Reference Manual


```c
#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))
```

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

```c
//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)
```

SEI		Global Interrupt Enable
CLI		Global Interrupt Disable

```c
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();
}
```

```c
// 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;
  }
}
```

```c
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) ;
}
```

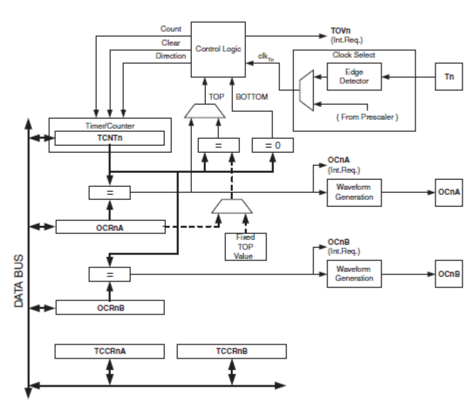

Setting up the timer interrupt hardware in C

```
#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

Set up a timer on an ARM 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
  }
}
```

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.

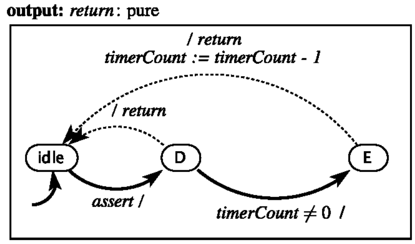
Concurrency in Computing

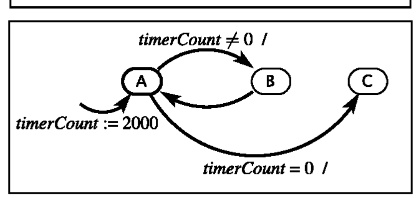
- Interrupt Handling
  - Reacting to external events (interrupts)
  - Exception handling (software interrupts)
- Processes
  - Creating the illusion of simultaneously running different programs (multitasking)
- Threads
  - How is a thread different from a process?
- Multiple processors (multi-cores)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

Modeling The Interrupt Handler Using State Machins

```
    volatile uint timerCount = 0;
    void ISR(void) {
       … disable interrupts
D →    if(timerCount != 0) {
E →       timerCount--;
       }
       … enable interrupts
    }
    int main(void) {
       // initialization code
       SysTickIntRegister(&ISR);
       … // other init
A →    timerCount = 2000;
B →    while(timerCount != 0) {
         … code to run for 2 seconds
       }
C →…whatever comes next
    }
```

variables: *timerCount*: uint
input: *assert*: pure
output: *return*: pure





A key question: Assuming interrupt can occur infinitely often, is position C always reached?

What kind of composition is needed here?

Asynchronous vs Synchronous Composition

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

Is synchronous composition the right model for this?

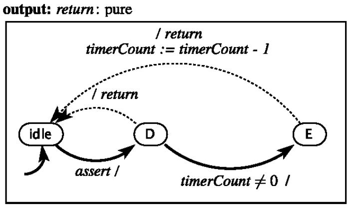
Is asynchronous composition (with interleaving semantics) the right model for this?

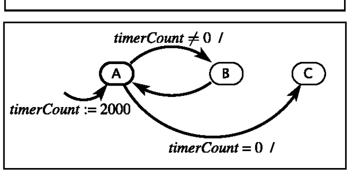
Answer: no to both.

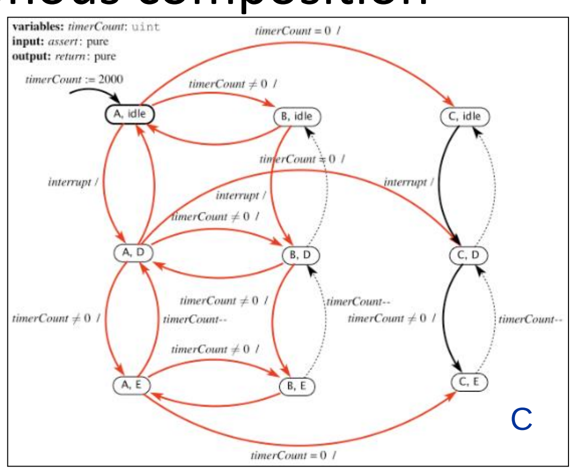
Asynchronous composition

$$S_C = S_A \times S_B$$

variables: *timerCount*: uint
input: *assert*: pure
output: *return*: pure





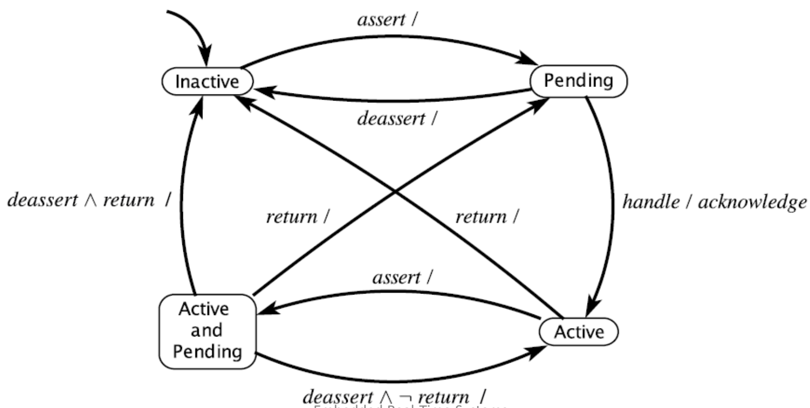


- This has transitions that will not occur in practice, such as A,D to B,D.
- Interrupts have priority over application code.

Modeling an interrupt controller

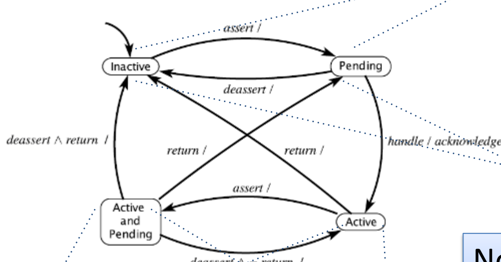
FSM model of a single interrupt handler in an interrupt controller:

input: *assert, deassert, handle, return*: pure
output: *acknowledge*



Modeling an interrupt controller

input: *assert, deassert, handle, return*: pure
output: *acknowledge*



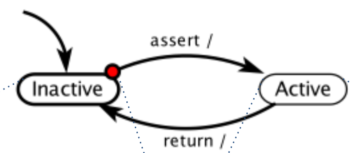

```
int main(void) {
   // initialization code
   SysTickIntRegister(&ISR);
   … // other init
   timerCount = 2000;
   while(timerCount != 0) {
    … code to run for 2 seconds
   }
}
```

Note that states can share refinements.

```
volatile uint timerCount = 0;
void ISR(void) {
   … disable interrupts
   if(timerCount != 0) {
      timerCount--;
   }
   … enable interrupts
}
```

Simplified Interrupt Controller Model

This abstraction assumes that an interrupt is always handled immediately upon being asserted:




```
    int main(void) {
       // initialization code
       SysTickIntRegister(&ISR);
       … // other init
A →    timerCount = 2000;
       while(timerCount != 0) {
B →     … code to run for 2 seconds
       }
C → }
```

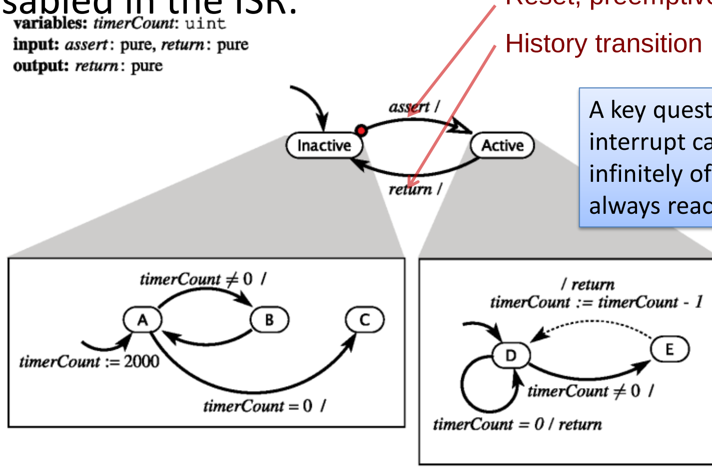
```
    volatile uint timerCount = 0;
    void ISR(void) {
D →    … disable interrupts
       if(timerCount != 0) {
E →       timerCount--;
       }
       … enable interrupts
    }
```


Hierarchical interrupt controller

This model assumes further that interrupts are disabled in the ISR:

Reset, preemptive transition

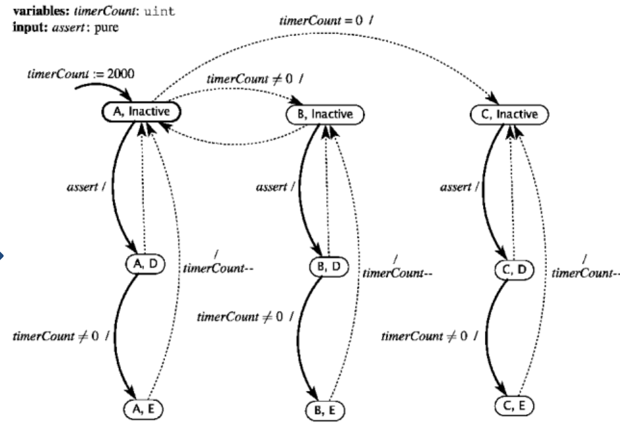
History transition

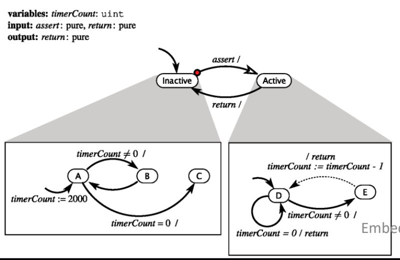


A key question: Assuming interrupt can occur infinitely often, is state C always reached?

Hierarchical Composition to Model Interrupts

History transition results in product state space, but hierarchy reduces the number of transitions compared to asynchronous composition.

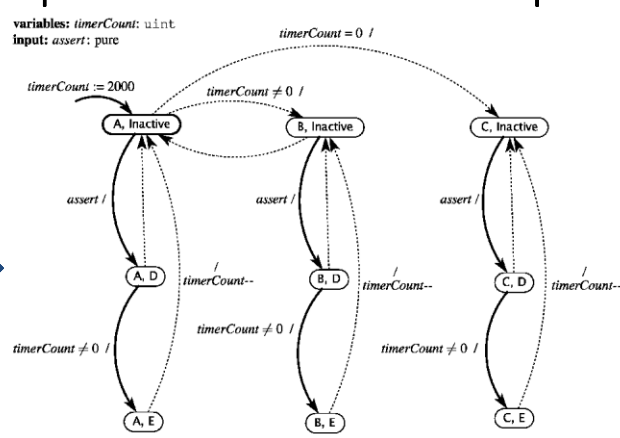


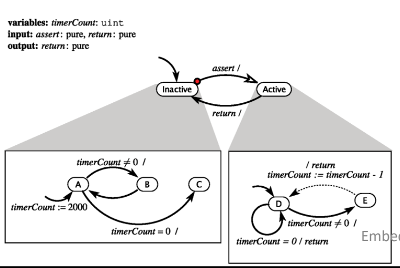


Examining this composition machine, it is clear that **C is not necessarily reached** if the interrupt occurs infinitely often. If assert is present on every reaction, C is never reached.

Hierarchical Composition to Model Interrupts

History transition results in product state space, but hierarchy reduces the number of transitions compared to asynchronous composition.





Under what assumptions/model of "assert" would C be reached?

In short…

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