Introduction to Embedded Systems

Edward A. Lee & Sanjit Seshia
UC Berkeley
EECS 124
Spring 2008

Lecture 8: Concurrency 1: Threads

Definition and Uses

Threads are sequential procedures that share memory.

Uses:

- Reacting to external events (interrupts)
- Exception handling (software interrupts)
- Creating the illusion of simultaneously running different programs (multitasking)
- Exploiting parallelism in the hardware (e.g. multicore machines).

Memory architectures: Issues

- Persistence
- Bounding the stack
- Scratchpad memories
- Caches
- Absolute and relative addresses
- Virtual memory
- Heaps
  - allocation/deallocation
  - fragmentation
  - garbage collection
- Segmented memory spaces
- …

These issues loom larger in embedded systems than in general-purpose computing.

EECS 124, UC Berkeley: 3

Memory usage: Understanding the stack. Find the flaw in this program

```
int x = 2;

int* foo(int y) {
  int z;
  z = y * x;
  return &z;
}

int main(void) {
  int* result = foo(10);
  ...
}
```

statically allocated: compiler assigns a memory location.

arguments on the stack

automatic variables on the stack

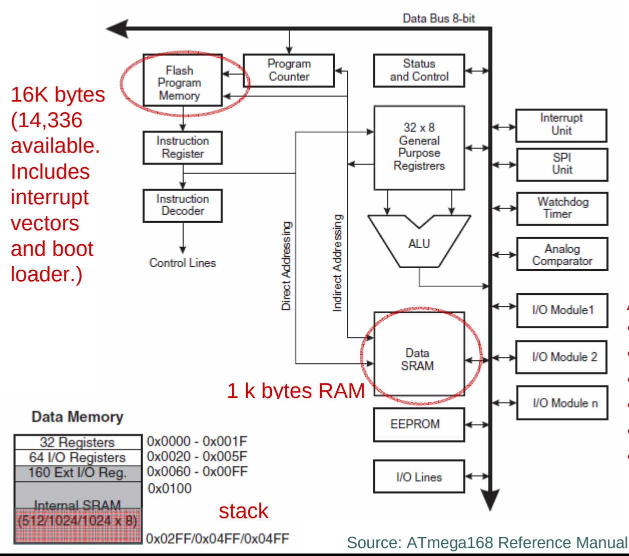
program counter and copies of all registers on the stack

This program returns a pointer to a variable on the stack. What if another procedure call occurs before the returned pointer is de-referenced?

EECS 124, UC Berkeley: 4

ATMega168 Architecture

16K bytes (14,336 available. Includes interrupt vectors and boot loader.)



Source: ATmega168 Reference Manual

Example of a microcontroller architecture. Used in iRobot command module.

Additional I/O in iRobot:
- Two 8-bit timer/counters
- One 16-bit timer/counter
- 6 PWM channels
- 8-channel, 10-bit ADC
- One serial UART
- 2-wire serial interface

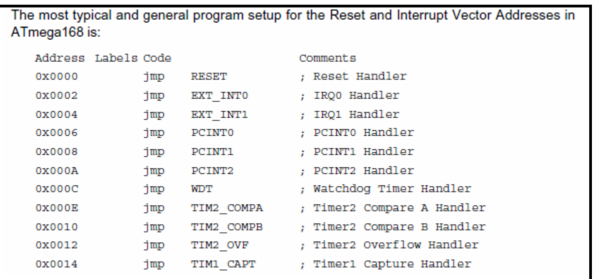
One more word on memory: Heaps

An operating system typically offers a way to dynamically allocate memory on a “heap”.

Memory management (malloc() and free()) can lead to many problems with embedded systems:
- Memory leaks (allocated memory is never freed)
- Memory fragmentation (allocatable pieces get smaller)

Automatic techniques (“garbage collection”) typically require stopping everything and reorganizing the allocated memory. This is deadly for real-time programs.

Focus on concurrency, starting with 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
```

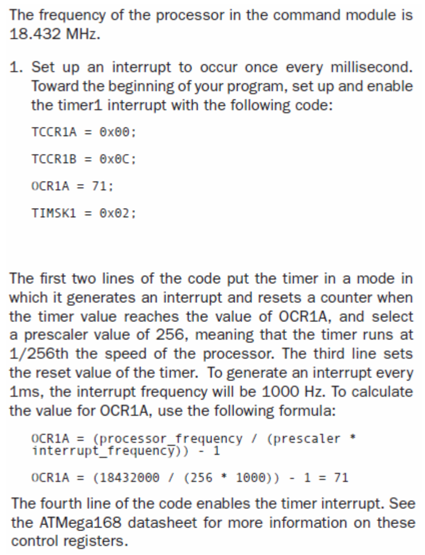
Source: ATmega168 Reference Manual

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

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.

Example: Set up a timer on the iRobot Command Module 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

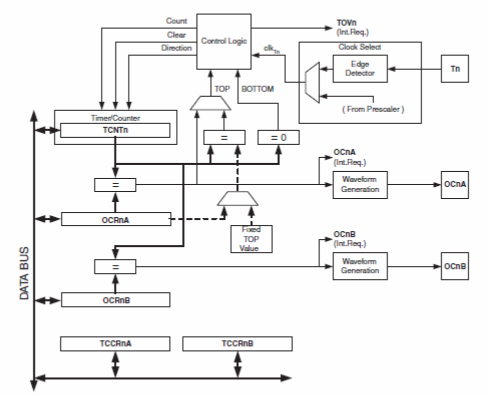
Setting up the timer interrupt hardware in C

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

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

memory-mapped register

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

Example: Do something for 2 seconds then stop

```
#include <avr/interrupt.h>
volatile uint16_t timer_count = 0;
volatile uint8_t timer_running = 0;
SIGNAL(SIG_OUTPUT_COMPARE1A) {
  if(timer_running) {
    if(timer_count) {
      timer_count--;
    } else {
      timer_running = 0;
    }
  }
}
int main(void) {
  ... set up from above
  timer_count = 2000;
  timer_running = 1;
  while(timer_running) {
    ... code to run for 2 seconds
  }
}
```

static variables: 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.

macro defining an interrupt service routine for interrupt vector that reacts to a timer interrupt. This macro takes care of saving register state (via compiler directives).

Source: iRobot Command Module Reference Manual v6

Concurrency

```
#include <avr/interrupt.h>
volatile uint16_t timer_count = 0;
volatile uint8_t timer_running = 0;
SIGNAL(SIG_OUTPUT_COMPARE1A) {
  if(timer_running) {
    if(timer_count) {
      timer_count--;
    } else {
      timer_running = 0;
    }
  }
}
int main(void) {
  ... set up from above
  timer_count = 2000;
  timer_running = 1;
  while(timer_running) {
    ... 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.

Source: iRobot Command Module Reference Manual v6

Reasoning about concurrent code

```
#include <avr/interrupt.h>
volatile uint16_t timer_count = 0;
volatile uint8_t timer_running = 0;
SIGNAL(SIG_OUTPUT_COMPARE1A) {
  if(timer_running) {
    if(timer_count) {
      timer_count--;
    } else {
      timer_running = 0;
    }
  }
}
int main(void) {
  ... set up from above
  timer_count = 2000;
  timer_running = 1;
  while(timer_running) {
    ... code to run for 2 seconds
  }
}
```

what if the interrupt occurs right here?

Source: iRobot Command Module Reference Manual v6

Reasoning about concurrent code

```
#include <avr/interrupt.h>
volatile uint16_t timer_count = 0;
volatile uint8_t timer_running = 0;
SIGNAL(SIG_OUTPUT_COMPARE1A) {
  if(timer_running) {
    if(timer_count) {
      timer_count--;
    } else {
      timer_running = 0;
    }
  }
}
int main(void) {
  ... set up from above
  timer_count = 2000;
  timer_running = 1;
  while(timer_running) {
    ... code to run for 2 seconds
  }
}
```

what if the interrupt occurs right here?

Source: iRobot Command Module Reference Manual v6

Reasoning about concurrent code

```
#include <avr/interrupt.h>
volatile uint16_t timer_count = 0;
volatile uint8_t timer_running = 0;
SIGNAL(SIG_OUTPUT_COMPARE1A) {
  if(timer_running) {
    if(timer_count) {
      timer_count--;
    } else {
      timer_running = 0;
    }
  }
}
int main(void) {
  ... set up from above
  timer_count = 2000;
  timer_running = 1;
  while(timer_running) {
    ... code to run for 2 seconds
  }
}
```

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

Source: iRobot Command Module Reference Manual v6

Reasoning about concurrent code

```
#include <avr/interrupt.h>
volatile uint16_t timer_count = 0;
volatile uint8_t timer_running = 0;
SIGNAL(SIG_OUTPUT_COMPARE1A) {
  if(timer_running) {
    if(timer_count) {
      timer_count--;
    } else {
      timer_running = 0;
    }
  }
}
int main(void) {
  ... set up from above
  timer_count = 2000;
  timer_running = 1;
  while(timer_running) {
    ... code to run for 2 seconds
  }
}
```

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

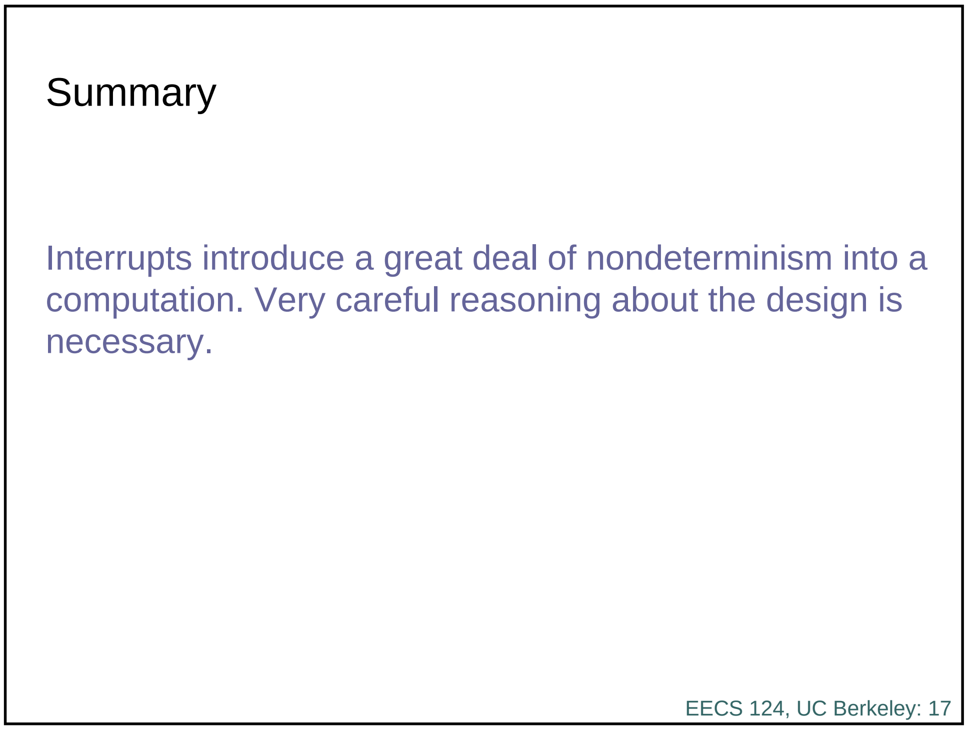
Source: iRobot Command Module Reference Manual v6

Reasoning about concurrent code

```
#include <avr/interrupt.h>
volatile uint16_t timer_count = 0;
volatile uint8_t timer_running = 0;
SIGNAL(SIG_OUTPUT_COMPARE1A) {
  if(timer_running) {
    if(timer_count) {
      timer_count--;
    } else {
      timer_running = 0;
    }
  }
}
int main(void) {
  ... set up from above
  timer_count = 2000;
  timer_running = 1;
  while(timer_running) {
    ... code to run for 2 seconds
  }
}
```

What is it about this code that makes it work?

Source: iRobot Command Module Reference Manual v6



Summary

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