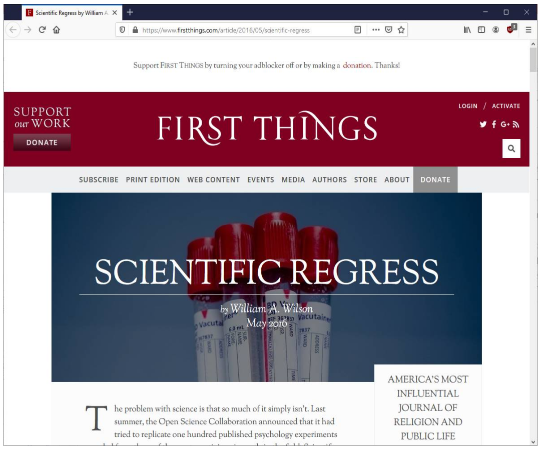
Informal discussion at the start of class on the matter of accuracy of measurement and the surprising results when measuring rare events. The content of this slide is not an official part of the course—will not be tested or graded.



Scientific Regress by William A.

https://www.firstthings.com/article/2016/05/scientific-regress

Support FIRST THINGS by turning your adblocker off or by making a donation. Thanks!

SUPPORT our WORK

DONATE

FIRST THINGS

LOGIN / ACTIVATE

SUBSCRIBE PRINT EDITION WEB CONTENT EVENTS MEDIA AUTHORS STORE ABOUT DONATE

SCIENTIFIC REGRESS

by William A. Wilson
May 2016

The problem with science is that so much of it simply isn't. Last summer, the Open Science Collaboration announced that it had tried to replicate one hundred published psychology experiments

AMERICA'S MOST INFLUENTIAL JOURNAL OF RELIGION AND PUBLIC LIFE

https://www.firstthings.com/article/2016/05/scientific-regress

From the article:
Suppose that there are a hundred and one stones in a certain field. One of them has a diamond inside it, and, luckily, you have a diamond-detecting device that advertises 99 percent accuracy. After an hour or so of moving the device around, examining each stone in turn, suddenly alarms flash and sirens wail while the device is pointed at a promising-looking stone. What is the probability that the stone contains a diamond?

Prof. dDB's Analysis:

100 stones with no diamonds, 1 stone with a diamond.
But since the tester is not perfect. I'll describe this situation with probabilities. . .
Let $P(X)$ be the probability that event X has happened.
Let $P(A|B)$ be the conditional probability of A given that B is true.
Let T_+ be the event of a test turning out positive, suggesting the stone has a diamond.
Let D be the event of randomly picking the stone with the diamond from the pile.
Let $\bar{D}$ be the event of randomly picking a stone with no diamond from the pile.

What I am given is that. . . $P(D) = 1/101$ and $P(\bar{D}) = 100/101$.
I am also given $P(T_+|D) = 99/100$ and $P(T_+|\bar{D}) = 1/100$
But what I want to know is $P(D|T_+)$. This is a call for Bayes theorem. (If given $P(B|A)$ and individual probabilities $P(A)$ and $P(B)$ then you can find $P(A|B)$ by Bayes theorem.)
Bayes theorem at Wikipedia: https://en.wikipedia.org/wiki/Bayes%27_theorem

Bayes theorem:

$$P(D|T_+) = \frac{P(T_+|D)P(D)}{P(T_+)} = \frac{P(T_+|D)P(D)}{P(T_+|D)P(D) + P(T_+|\bar{D})P(\bar{D})} = \frac{\left(\frac{99}{100}\right)\left(\frac{1}{101}\right)}{\left(\frac{99}{100}\right)\left(\frac{1}{101}\right) + \left(\frac{1}{100}\right)\left(\frac{100}{101}\right)} = \frac{99}{199} \simeq \boxed{\frac{1}{2}}$$

The moral of the story: Accuracy needs to be interpreted in context.
(Also, a senior-graduate level course in stochastic systems is very relevant for an engineer!)

1

Here is another program that uses the same hardware source of interrupts.

The loop does gadfly output. The relative time is updated by the interrupt service routine. The `loop()` routine and the interrupt service routine communicate via global variable `relativeTime`.

```
#include <Time.h>
#include <TimeLib.h>

// This program demonstrates how a tic clock can keep relative time

#define NOT_AN_INTERRUPT -1  //Required due to a bug in the IDE
#define TICS_PER_SECOND 60

//Global variables
int led_pin = 13;
int intr_pin =  2;
volatile time_t relativeTime=0;
volatile int tics = 0;

void tic_isr() {
  ++tics;
  if (tics >= TICS_PER_SECOND) {
    tics = 0;
    ++relativeTime;  //seconds since reset
  }
}

void setup() {
   pinMode(led_pin, OUTPUT);
   digitalWrite (intr_pin, LOW);  // disable internal pull-up resistor
   attachInterrupt(digitalPinToInterrupt(intr_pin), tic_isr, RISING);
}

void loop () {
  //if relativeTime is even, illuminate the LED, otherwise extinquish it
  if (relativeTime % 2 == 0) {
    digitalWrite(led_pin, HIGH);
  }
  else {
    digitalWrite(led_pin, LOW);
  }
}
```

Interrupt service routine

2

The agenda—understanding interrupt-driven I/O (and by extension, multitasking)

An example to give some context

Next

Memory capabilities needed for subroutines (functions, procedures, interrupts, are types of subroutines)

Sources of interrupts including counter-timer systems

Advantages of using interrupt-driven I/O—so obvious this section is hardly needed.
--Alternatives to interrupt driven I/O are gadfly (uncontrolled—annoying) I/O or various polling techniques, all of which waste processor cycles prodigiously.
--Interrupts are foundational to object-oriented programming
--Many embedded systems that use interrupts have very little other code to run!

Risks of interrupt-driven I/O
--density limit
--latency and resolution limits
--interval restrictions
--critical regions in code
--deadlock

3

Memory capabilities needed for subroutines—stack operations

These illustrations were done for Matlab code, but the same applies for any language including assembly, c/c++, Python.

What happens when you call a function?

Example:

Script filename `junk_word.m`

```
% Call a function that makes a display
write_yep;
disp(“OK, now do it again!”)
write_yep;
```

Function filename write_yep.m

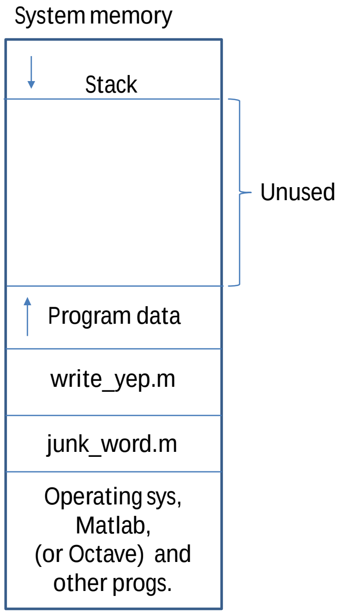
```
function write_yep
% Writes the word “yep”
disp(“yep”);
```

Command Window

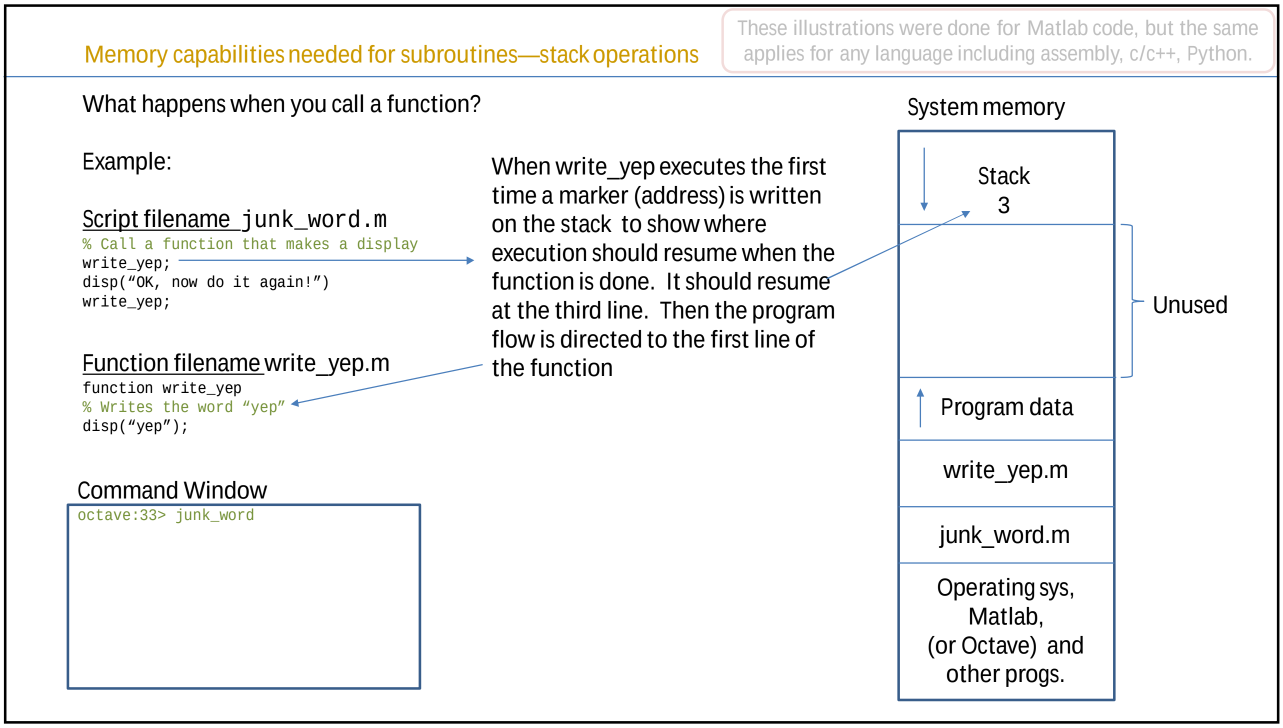
```
octave:33>
```

When Matlab starts the script the memory is set up as shown.

The stack is a scratch-pad area in which data may be temporarily stored. It “grows downward” as it is used. It is “lifted upwards” when data is removed from it.



4



Memory capabilities needed for subroutines—stack operations

These illustrations were done for Matlab code, but the same applies for any language including assembly, c/c++, Python.

What happens when you call a function?

Example:

Script filename junk_word.m

```
% Call a function that makes a display
write_yep;
disp("OK, now do it again!")
write_yep;
```

Function filename write_yep.m

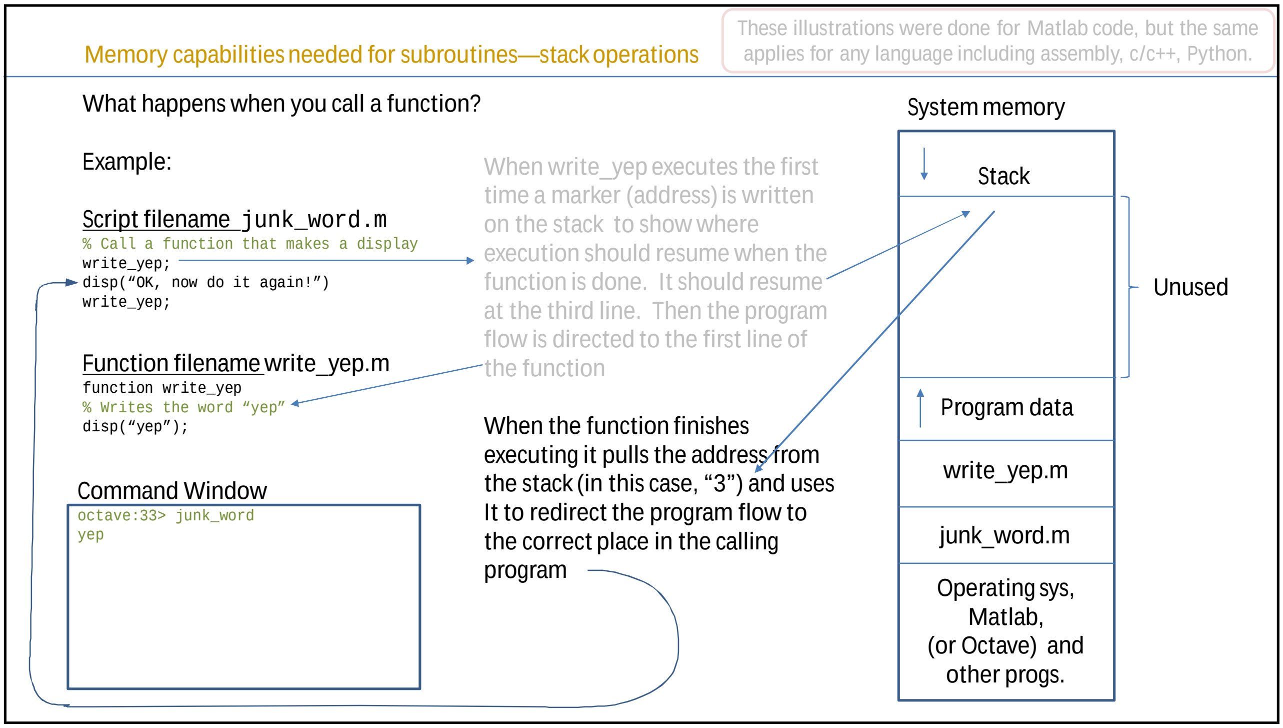
```
function write_yep
% Writes the word "yep"
disp("yep");
```

When write_yep executes the first time a marker (address) is written on the stack to show where execution should resume when the function is done. It should resume at the third line. Then the program flow is directed to the first line of the function

Command Window

```
octave:33> junk_word
```

5



Memory capabilities needed for subroutines—stack operations

These illustrations were done for Matlab code, but the same applies for any language including assembly, c/c++, Python.

What happens when you call a function?

Example:

Script filename junk_word.m

```
% Call a function that makes a display
write_yep;
disp("OK, now do it again!")
write_yep;
```

Function filename write_yep.m

```
function write_yep
% Writes the word "yep"
disp("yep");
```

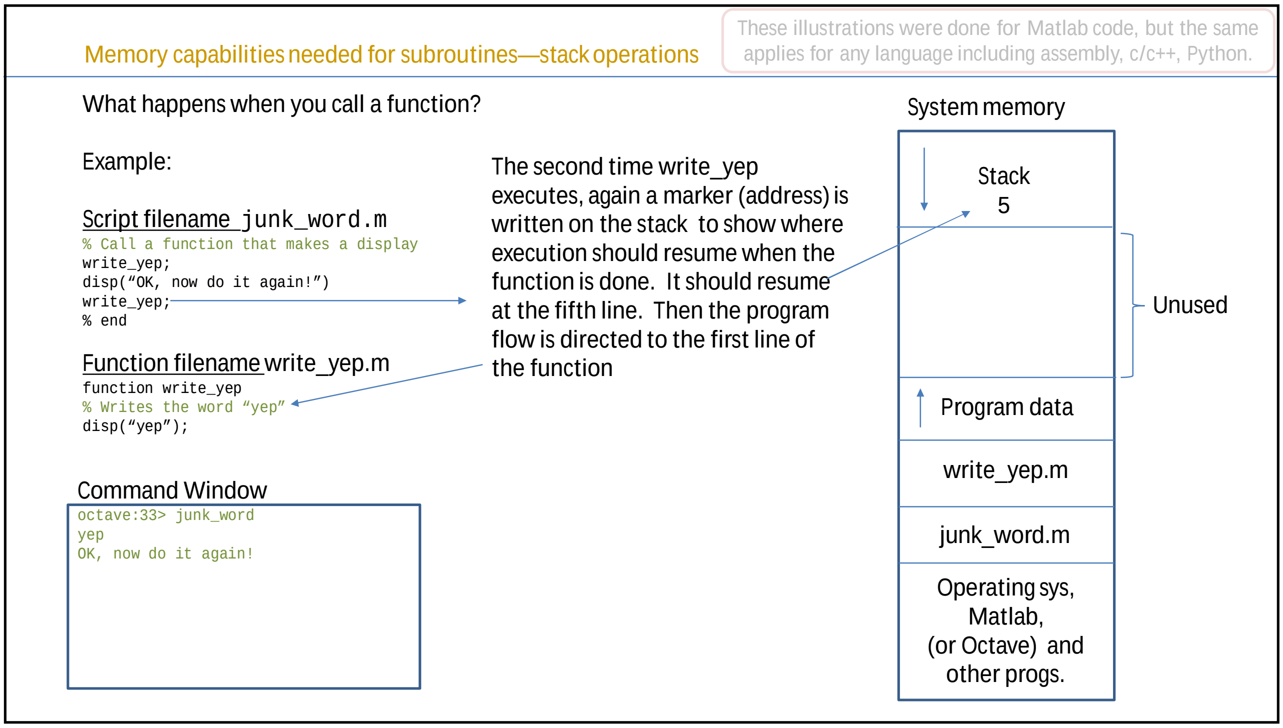
When write_yep executes the first time a marker (address) is written on the stack to show where execution should resume when the function is done. It should resume at the third line. Then the program flow is directed to the first line of the function

When the function finishes executing it pulls the address from the stack (in this case, "3") and uses It to redirect the program flow to the correct place in the calling program

Command Window

```
octave:33> junk_word
yep
```

6



Memory capabilities needed for subroutines—stack operations

These illustrations were done for Matlab code, but the same applies for any language including assembly, c/c++, Python.

What happens when you call a function?

Example:

Script filename junk_word.m

```
% Call a function that makes a display
write_yep;
disp("OK, now do it again!")
write_yep;
% end
```

Function filename write_yep.m

```
function write_yep
% Writes the word "yep"
disp("yep");
```

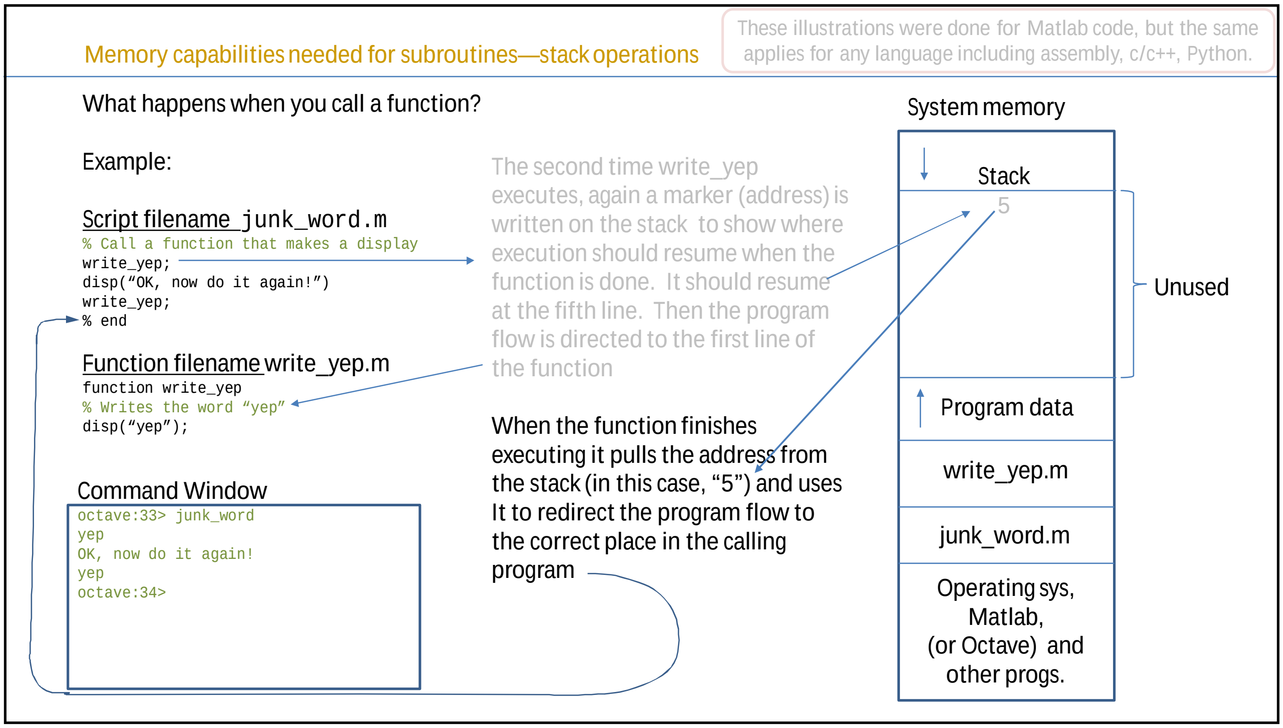
The second time write_yep executes, again a marker (address) is written on the stack to show where execution should resume when the function is done. It should resume at the fifth line. Then the program flow is directed to the first line of the function

Command Window

```
octave:33> junk_word
yep
OK, now do it again!
```

System memory

7



Memory capabilities needed for subroutines—stack operations

These illustrations were done for Matlab code, but the same applies for any language including assembly, c/c++, Python.

What happens when you call a function?

Example:

Script filename junk_word.m

```
% Call a function that makes a display
write_yep;
disp("OK, now do it again!")
write_yep;
% end
```

Function filename write_yep.m

```
function write_yep
% Writes the word "yep"
disp("yep");
```

The second time write_yep executes, again a marker (address) is written on the stack to show where execution should resume when the function is done. It should resume at the fifth line. Then the program flow is directed to the first line of the function

When the function finishes executing it pulls the address from the stack (in this case, "5") and uses It to redirect the program flow to the correct place in the calling program

Command Window

```
octave:33> junk_word
yep
OK, now do it again!
yep
octave:34>
```

System memory

8

Memory capabilities needed for subroutines—stack operations

These illustrations were done for Matlab code, but the same applies for any language including assembly, c/c++, Python.

What happens when you call a function that needs parameters?

Example:

Script filename junk_sum.m

```
% Call a function that adds numbers;
add_two_nums(4,6);
disp("OK, now do it again!")
add_two_nums(9,7);
% end
```

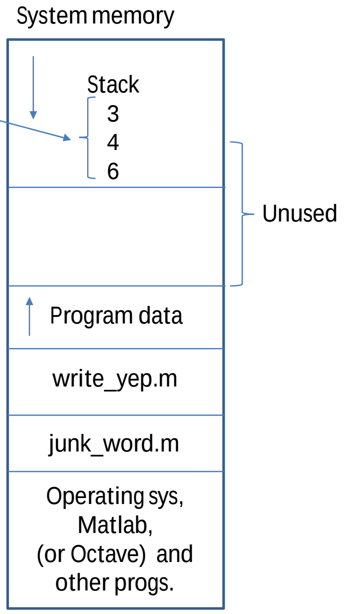
Function filename add_two_nums.m

```
function add_two_nums(a,b)
% displays the sum of a + b
disp(a + b);
```

Upon being called the return address is put on the stack and also the parameters, in the order given. The stack expands downward as far as needed.

Command Window

```
octave:40> junk_sum
```


System memory

Stack
3
4
6

Unused

Program data

write_yep.m

junk_word.m

Operating sys, Matlab, (or Octave) and other progs.

9

Memory capabilities needed for subroutines—stack operations

These illustrations were done for Matlab code, but the same applies for any language including assembly, c/c++, Python.

What happens when you call a function that needs parameters?

Example:

Script filename junk_sum.m

```
% Call a function that adds numbers;
add_two_nums(4,6);
disp("OK, now do it again!")
add_two_nums(9,7);
% end
```

Function filename add_two_nums.m

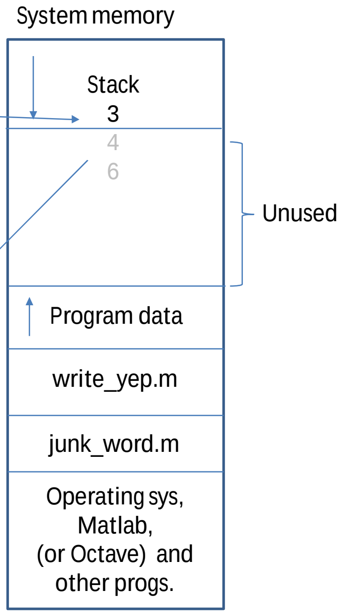
```
function add_two_nums(a,b)
% displays the sum of a + b
disp(a + b);
```

Upon being called the return address is put on the stack and also the parameters, in the order given. The stack expands downward as far as needed.

When the function starts executing it pulls (copies) the parameters from the stack in reverse order according to the list on the first line of the function. (in this case, b = 6 and a = 4) and uses those to execute. The parameters remain in memory but can no longer be accessed.

Command Window

```
octave:40> junk_sum
```


System memory

Stack
3
4
6

Unused

Program data

write_yep.m

junk_word.m

Operating sys, Matlab, (or Octave) and other progs.

10

Memory capabilities needed for subroutines—stack operations

These illustrations were done for Matlab code, but the same applies for any language including assembly, c/c++, Python.

What happens when you call a function that needs parameters?

Example:

Script filename `junk_sum.m`

```
% Call a function that adds numbers;
add_two_nums(4,6);
disp("OK, now do it again!")
add_two_nums(9,7);
% end
```

Function filename add_two_nums.m

```
function add_two_nums(a,b)
% displays the sum of a + b
disp(a + b);
```

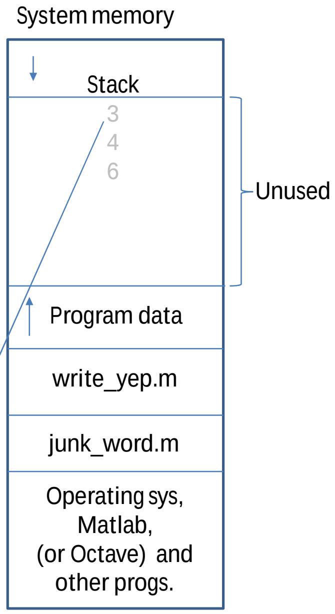
Command Window

```
octave:40> junk_sum
 10
```

Upon being called the return address is put on the stack and also the parameters, in the order given. The stack expands downward as far as needed.

When the function starts executing it pulls (copies) the parameters from the stack in reverse order according to the list on the first line of the function. (in this case, b = 6 and a = 4) and uses those to execute.

When the function is done it pulls the return address (in this case, “3”) and transfers program flow to the calling program.



11

Memory capabilities needed for subroutines—stack operations

These illustrations were done for Matlab code, but the same applies for any language including assembly, c/c++, Python.

What happens when you call a function that needs parameters?

Example:

Script filename `junk_sum.m`

```
% Call a function that adds numbers;
add_two_nums(4,6);
disp("OK, now do it again!")
add_two_nums(9,7);
% end
```

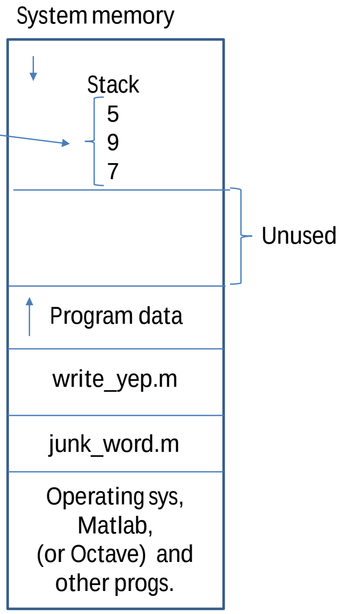
Function filename add_two_nums.m

```
function add_two_nums(a,b)
% displays the sum of a + b
disp(a + b);
```

Command Window

```
octave:40> junk_sum
 10
OK, now do it again!
```

Upon being called the second time the return address is put on the stack and also the parameters, in the order given. The stack expands downward as far as needed.



12

Memory capabilities needed for subroutines—stack operations

These illustrations were done for Matlab code, but the same applies for any language including assembly, c/c++, Python.

What happens when you call a function that needs parameters?

Example:

Script filename junk_sum.m

```
% Call a function that adds numbers;
add_two_nums(4,6);
disp("OK, now do it again!")
add_two_nums(9,7);
% end
```

Function filename add_two_nums.m

```
function add_two_nums(a,b)
% displays the sum of a + b
disp(a + b);
```

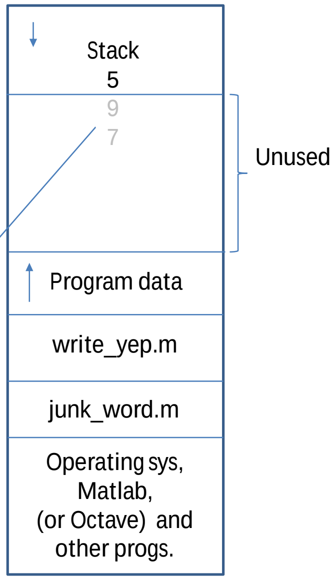
Command Window

```
octave:40> junk_sum
 10
OK, now do it again!
```

Upon being called the second time the return address is put on the stack and also the parameters, in the order given. The stack expands downward as far as needed.

When the function starts executing it pulls (copies) the parameters from the stack in reverse order according to the list on the first line of the function. (in this case, b = 7 and a = 9) and uses those to execute. The parameters remain

System memory



Stack

5

9

7

Unused

Program data

write_yep.m

junk_word.m

Operating sys, Matlab, (or Octave) and other progs.

Memory capabilities needed for subroutines—stack operations

These illustrations were done for Matlab code, but the same applies for any language including assembly, c/c++, Python.

What happens when you call a function that needs parameters?

Example:

Script filename junk_sum.m

```
% Call a function that adds numbers;
add_two_nums(4,6);
disp("OK, now do it again!")
add_two_nums(9,7);
% end
```

Function filename add_two_nums.m

```
function add_two_nums(a,b)
% displays the sum of a + b
disp(a + b);
```

Command Window

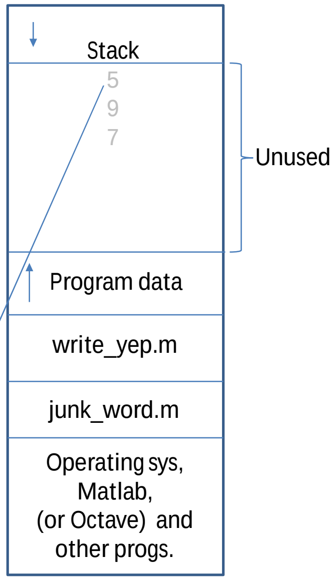
```
octave:40> junk_sum
 10
OK, now do it again!
 16
octave:41>
```

Upon being called the second time the return address is put on the stack and also the parameters, in the order given. The stack expands downward as far as needed.

When the function starts executing it pulls (copies) the parameters from the stack in reverse order according to the list on the first line of the function. (in this case, b = 6 and a = 4) and uses those to execute. The parameters remain

When the function is done it pulls the return address (in this case, “5”) and transfers program flow to the calling program.

System memory



Stack

5

9

7

Unused

Program data

write_yep.m

junk_word.m

Operating sys, Matlab, (or Octave) and other progs.

Memory capabilities needed for subroutines—stack operations

These illustrations were done for Matlab code, but the same applies for any language including assembly, c/c++, Python.

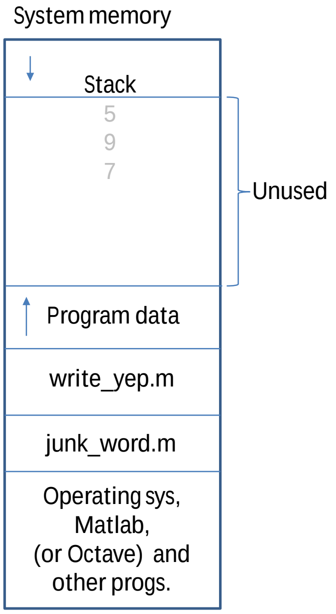
What happens when you call a function that. . .

. . . needs parameters and. . .
. . . needs to return a parameter to the calling program?

After the function is done executing it pulls the return address as usual, but before returning control to the calling program it pushes the return parameters onto the stack. The returning program knows it must pull them because of the form of the function call

```
[return_parameters] = function(parameter_list);
```

If a function is written such that it produces return parameters they will always get pushed to the stack before return. The calling program will pull the parameters off the stack (indeed it must). Pulled parameters will be stored in the variables indicated by the program call, or if the return parameters were not directed to a variable by the programmer, they will be discarded to the bit-bucket.



15

Memory capabilities needed for subroutines—stack operations

What happens when an interrupt is requested via an interrupt pin?

Step 1) If the pin has been set up to receive interrupts, and if the interrupt is enabled, an interrupt service request (ISR) bit is set in a special register in the CPU's hardware.

Step 2) Whatever *atomic operation* is in progress finishes executing. Then the ISR bit-register is checked.
If no bits are set, execution continues with the next machine instruction as normal, otherwise go to step 3.
Machine instructions and *critical regions* are atomic operations. (Interrupt service routines are normally critical.)
A *critical region* is a section of code that runs with interrupts disabled for some reason.
Critical regions may be deliberately created or prevented using *disable interrupt* and *enable interrupt* instructions.
In some cases the compiler will automatically create critical regions at the machine instruction level.

Step 3) If one or more ISR bits are set, the highest priority ISR bit will receive attention. The corresponding interrupt service routine (which must have been set up previously) will be called in a style similar to a subroutine.
All interrupts will be automatically disabled upon recognition of the highest priority interrupt.
A return address is pushed to the stack. In addition, some or all of the CPU's registers will be pushed to the stack.
The interrupt service routine may only use those registers that have been pushed, or it may push more registers via its own instructions if needed. The ISR bit will be reset when the first instruction of the interrupt service routine begins executing. (The interrupt may now be requested—set—again!)

Step 4) The interrupt service routine runs. (Because all interrupts are disabled it will run to completion without further interruption.) However the interrupt service routine may be programmed to re-enable any or all other interrupts (dangerous but powerful). Any information the interrupt service routine needs to manipulate will have to come from either persistent or global variables or I/O operations.

16

Memory capabilities needed for subroutines—stack operations

What happens when an interrupt is requested via an interrupt pin?

Step 1) If the pin has been set up to receive interrupts, and if the interrupt is enabled,
an interrupt service request (ISR) bit is set in a special register in the CPU's hardware.

Step 2) Whatever *atomic operation* finishes executing the ISR bit-register is checked.
If no bits are set, continue with the next machine instruction as normal, otherwise go to step 3.
Machine instructions and *critical regions* are atomic operations. (Interrupt service routines are normally atomic too.)
A *critical region* is a section of code that runs with interrupts disabled for some reason.
Critical regions may be deliberately created using *disable interrupt* and *enable interrupt* instructions.
In some other cases the compiler will automatically create critical regions at the machine instruction level.

Step 3) If one or more ISR bits are set, the highest priority ISR bit will receive attention. The corresponding interrupt service routine (which must have been set up previously) will be called in a style similar to a subroutine.
All interrupts will be automatically disabled upon recognition of the highest priority interrupt.
A return address is pushed to the stack. In addition, some or all of the CPU's registers will be pushed to the stack.
The interrupt service routine may only use those registers that have been pushed, or it may push more registers via its own instructions if needed. The ISR bit will be reset when the first instruction of the interrupt service routine begins executing. (The interrupt may now be requested—set—again!)

Step 4) The interrupt service routine normally runs atomically. (Because all interrupts are disabled it will run to completion without further interruption.) However the interrupt service routine may be programmed to re-enable any or all other interrupts (dangerous but powerful). Any information the interrupt service routine needs to manipulate will have to come from either persistent or global variables or I/O operations.

Step 5) The interrupt service routine finishes with a machine-level "return from interrupt" command.
This command pulls the former contents of all CPU registers that were initially pushed to the stack and returns these registers back to their original state.
Then the command returns program control back to the routine that was interrupted. The routine that was interrupted will continue executing as if nothing happened, except it has been delayed a small amount.

Step 6) The normal programming of the CPU can access processing that was performed by the interrupt service routines by using global variables.

17

The agenda—understanding interrupt-driven I/O (and by extension, multitasking)

An example to give some context

Memory capabilities needed for subroutines (functions, procedures, interrupts, are types of subroutines)

Next

Sources of interrupts including counter-timer systems

Advantages of using interrupt-driven I/O—so obvious this section is hardly needed.
--Alternatives to interrupt driven I/O are gadfly (uncontrolled—annoying) I/O or various polling techniques, all of which waste processor cycles prodigiously.
--Interrupts are foundational to object-oriented programming
--Many embedded systems that use interrupts have very little other code to run!

Risks of interrupt-driven I/O
--density limit
--latency and resolution limits
--interval restrictions
--critical regions in code
--deadlock

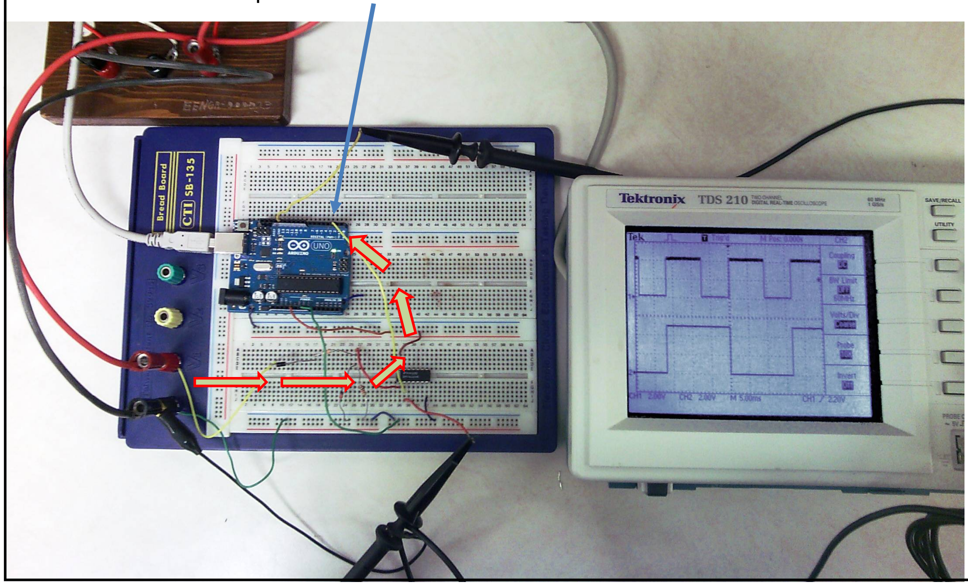
18

Detail Slide

Sources of interrupts

Hardware interrupt

Arrives via an I/O port connected to the CPU



19

Sources of interrupts

Hardware interrupt

Arrives via an I/O port connected to the CPU

Some ports have external GPIO pins that can be set up to support interrupts (e.g. pin 2 in the previous example)

20

Detail Slide

Sources of interrupts

Hardware interrupt

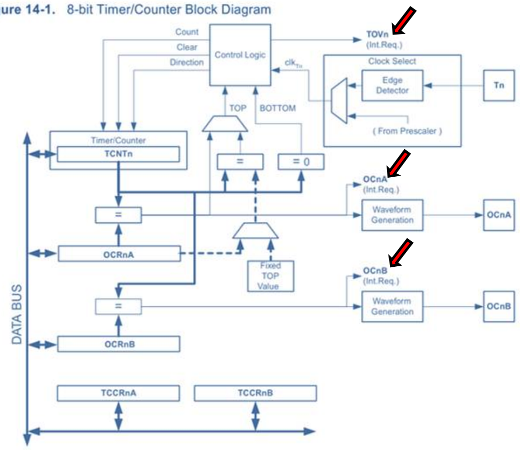
Arrives via an I/O port connected to the CPU

Some ports have external GPIO pins that can be set up to support interrupts (e.g. pin 2 in the previous example)

Additionally, virtually all microcontrollers and SoC systems have internal ports that can generate interrupts.

These internal ports are connected to counters and compare registers and form a so-called *counter-timer-system*.

Figure 14-1. 8-bit Timer/Counter Block Diagram



The counter-timer systems is configured via a set of registers. Registers are also readable so that the main program (or loop) has a way to respond to the counted or timed events.

TCNT0 – Timer/Counter Register

Bit	7	6	5	4	3	2	1	0	
0x26 (0x46)	TCNT0[7:0]								TCNT0
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

The Timer/Counter Register gives direct access, both for read and write operations, to the Timer/Counter unit 8-bit counter. Writing to the TCNT0 Register blocks (removes) the Compare Match on the following timer clock. Modifying the counter (TCNT0) while the counter is running, introduces a risk of missing a Compare Match between TCNT0 and the OCR0x Registers.

OCR0A – Output Compare Register A

Bit	7	6	5	4	3	2	1	0	
0x27 (0x47)	OCR0A[7:0]								OCR0A
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

The Output Compare Register A contains an 8-bit value that is continuously compared with the counter value (TCNT0). A match can be used to generate an Output Compare interrupt, or to generate a waveform output on the OC0A pin.

OCR0B – Output Compare Register B

Bit	7	6	5	4	3	2	1	0	
0x28 (0x48)	OCR0B[7:0]								OCR0B
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

The Output Compare Register B contains an 8-bit value that is continuously compared with the counter value (TCNT0). A match can be used to generate an Output Compare interrupt, or to generate a waveform output on the OC0B pin.

Illustration is from the AVR datasheet

21

Sources of interrupts

Hardware interrupt

Arrives via an I/O port connected to the CPU

Some ports have external GPIO pins that can be set up to support interrupts (e.g. pin 2 in the previous example)

Additionally, virtually all microcontrollers and SoC systems have internal ports that can generate interrupts.

These internal ports are connected to counters and compare registers and form a so-called *counter-timer-system*.

Software interrupt

Is triggered by a program instruction. Example in C: `SWI Printer_Status`

Here `Printer_Status` is a "label" (A symbol that is defined as an address to the interrupt service routine.)

22

Sources of interrupts

Hardware interrupt

Arrives via an I/O port connected to the CPU

- Some ports have external GPIO pins that can be set up to support interrupts (e.g. pin 2 in the previous example)
- Additionally, virtually all microcontrollers and SoC systems have internal ports that can generate interrupts.
  - These internal ports are connected to counters and compare registers and form a so-called *counter-timer-system*.

Software interrupt

Is triggered by a program instruction. E.g. `SWI <label>`

- In CPUs that do not have hardware-supported privilege rings (e.g Arduino) a software interrupt is tantamount to a normal subroutine call except that it also stacks ALL CPU registers—a minor convenience sometimes.

23

Sources of interrupts

Hardware interrupt

Arrives via an I/O port connected to the CPU

- Some ports have external GPIO pins that can be set up to support interrupts (e.g. pin 2 in the previous example)
- Additionally, virtually all microcontrollers and SoC systems have internal ports that can generate interrupts.
  - These internal ports are connected to counters and compare registers and form a so-called *counter-timer-system*.

Software interrupt

Is triggered by a program instruction. E.g. `SWI <label>`

- In CPUs that do not have hardware-supported privilege rings (e.g Arduino) a software interrupt is tantamount to a normal subroutine call except that it also stacks ALL CPU registers—a minor convenience sometimes.

- In CPUs that have hardware-supported privilege rings (e.g. Raspberry Pi.) software interrupts are the only way to call privileged routines because the CPU state and all hardware resources are protected by stacking ALL registers to a hardware-protected stack. (A lower privileged routine has no access to CPU registers, ports, or memory used by higher privileged routines.)

24

Sources of interrupts

Hardware interrupt

Arrives via an I/O port connected to the CPU

Some ports have external GPIO pins that can be set up to support interrupts (e.g. pin 2 in the previous example)

Additionally, virtually all microcontrollers and SoC systems have internal ports that can generate interrupts.

These internal ports are connected to counters and compare registers and form a so-called *counter-timer-system*.

Software interrupt

Is triggered by a program instruction. E.g. `SWI <label>`

In CPUs that do not have hardware-supported privilege rings (e.g Arduino) a software interrupt is tantamount to a normal subroutine call except that it also stacks ALL CPU registers—a minor convenience sometimes.

In CPUs that have hardware-supported privilege rings (e.g. Raspberry Pi.) software interrupts are the only way to call privileged routines because the CPU state and all hardware resources are protected by stacking ALL registers to a hardware-protected stack. (A lower privileged routine has no access to CPU registers, ports, or memory used by higher privileged routines.)

Arduino has no software interrupt instruction. However one can write to a hardware interrupt pin and thus create the equivalent action. But why not just use a normal call? There are no privilege rings to cut through.

Raspbian takes all interrupt resources to itself. Python has no inter. access whatever. Access to interrupts is only via the OS (slow).

25

Sources of interrupts

Hardware interrupt

Arrives via an I/O port connected to the CPU

Some ports have external GPIO pins that can be set up to support interrupts (e.g. pin 2 in the previous example)

Additionally, virtually all microcontrollers and SoC systems have internal ports that can generate interrupts.

These internal ports are connected to counters and compare registers and form a so-called *counter-timer-system*.

Software interrupt

Is triggered by a program instruction. E.g. `SWI <label>`

In CPUs that do not have hardware-supported privilege rings (e.g Arduino) a software interrupt is tantamount to a normal subroutine call except that it also stacks ALL CPU registers—a minor convenience sometimes.

In CPUs that have hardware-supported privilege rings (e.g. Raspberry Pi.) software interrupts are the only way to call privileged routines because the CPU state and all hardware resources are protected by stacking ALL registers to a hardware-protected stack. (A lower privileged routine has no access to CPU registers, ports, or memory used by higher privileged routines.)

Arduino has no software interrupt instruction. However one can write to a hardware interrupt pin and thus create the equivalent action. But why not just use a normal call? There are no privilege rings to cut through.

Raspbian takes all interrupt resources to itself. Python has no inter. access whatever. Access to interrupts is only via the OS (slow).

Exception (a.k.a. trap)

Arrives from inside the CPU via hardware but not via a complete port. It calls an interrupt service routine as for any intr.

26

Detail Slide

Sources of interrupts

Hardware interrupt

Arrives via an I/O port connected to the CPU

Some ports have external GPIO pins that can be set up to support interrupts (e.g. pin 2 in the previous example)

Additionally, virtually all microcontrollers and SoC systems have internal ports that can generate interrupts.

These internal ports are connected to counters and compare registers and form a so-called *counter-timer-system*.

Software interrupt

Is triggered by a program instruction. E.g. `SWI <label>`

In CPUs that do not have hardware-supported privilege rings (e.g Arduino) a software interrupt is tantamount to a normal subroutine call except that it also stacks ALL CPU registers—a minor convenience sometimes.

Arduino has no software interrupt instruction. However one can write to a hardware interrupt pin and thus create the equivalent action. But why not just use a normal call? There are no privilege rings to cut through.

In CPUs that have hardware-supported privilege rings (e.g. Raspberry Pi.) software interrupts are the only way to call privileged routines because the CPU state and all hardware resources are protected by stacking ALL registers to a hardware-protected stack. (A lower privileged routine has no access to CPU registers, ports, or memory used by higher privileged routines.)

Raspbian takes all interrupt resources to itself. Python has no inter. access whatever. Access to interrupts is only via the OS (slow).

Exception (a.k.a. trap)

Arrives from inside the CPU *via hardware* but not via a complete port. It calls an interrupt service routine as for any intr.

The request is *handled like a software interrupt* (controlled access to privileged resources).

The return usually is to the operating system, which proceeds to kill the offending program.

Examples: Corrupted OS file, divide by zero error. (If not configured, ignored.)

27

SUMMARY SLIDE

Sources of interrupts

Hardware interrupt

Arrives via an I/O port connected to the CPU

Some ports have external GPIO pins that can be set up to support interrupts (e.g. pin 2 in the previous example)

Additionally, virtually all microcontrollers and SoC systems have internal ports that can generate interrupts.

These internal ports are connected to counters and compare registers and form a so-called *counter-timer-system*.

Software interrupt

Is triggered by a program instruction. E.g. `SWI <label>`

In CPUs that do not have hardware-supported privilege rings (e.g Arduino) a software interrupt is tantamount to a normal subroutine call except that it also stacks ALL CPU registers—a minor convenience sometimes.

Arduino has no software interrupt instruction. However one can write to a hardware interrupt pin and thus create the equivalent action. But why not just use a normal call? There are no privilege rings to cut through.

In CPUs that have hardware-supported privilege rings (e.g. Raspberry Pi.) software interrupts are the only way to call privileged routines because the CPU state and all hardware resources are protected by stacking ALL registers to a hardware-protected stack. (A lower privileged routine has no access to CPU registers, ports, or memory used by higher privileged routines.)

Raspbian takes all interrupt resources to itself. Python has no inter. access whatever. Access to interrupts is only via the OS (slow).

Exception (a.k.a. trap) Arduino has no exception interrupts

Arrives from inside the CPU *via hardware* but not via a complete port. It calls an interrupt service routine as for any intr.

The request is *handled like a software interrupt* (controlled access to privileged resources).

The return usually is to the operating system, which proceeds to kill the offending program.

Examples: Corrupted OS file, divide by zero error. (If not configured, ignored.)

28