memory management
the stack & the heap

memory management

So far:

data representations:

how are individual data elements represented in memory?

pointers and pointer arithmetic

to find out where data is allocated

Now:

memory management:

how is the memory as a whole organised and managed?

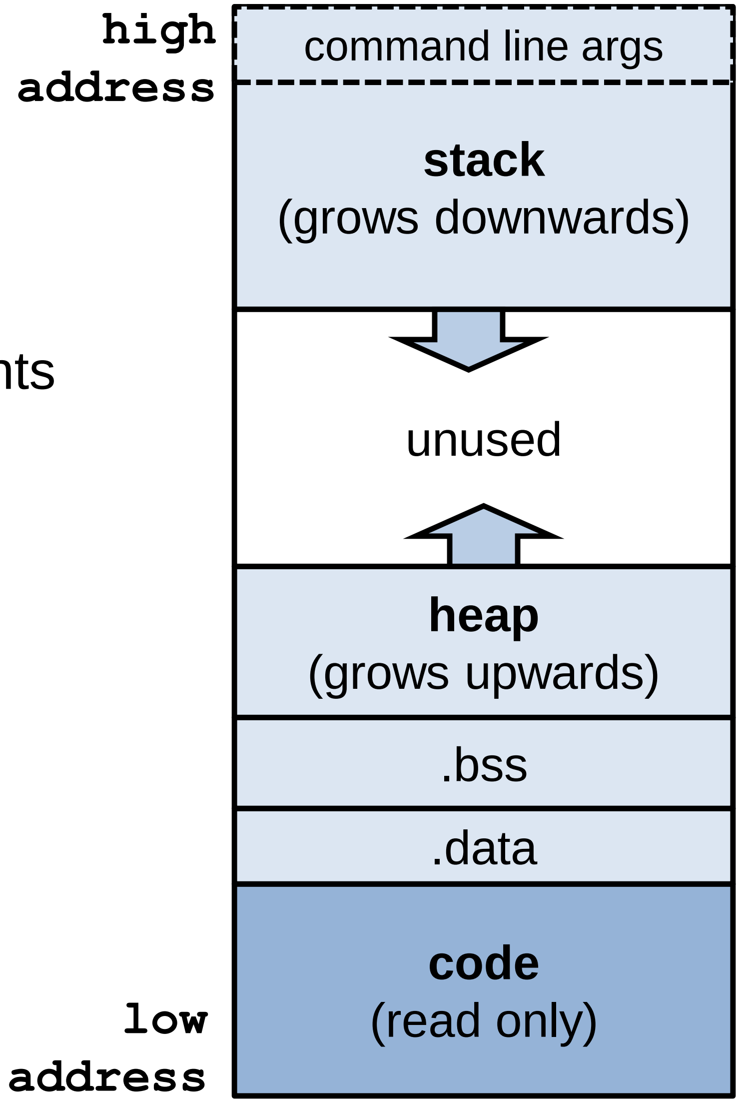
memory segments

The OS allocates memory for each *process* - ie. a running program – for data and code

This memory consists of different segments

- stack - for local variables
  - incl. command line arguments and environment variables
- heap - for dynamic memory
- data segment for
  - global uninitialised variables (.bss)
  - global initialised variables (.data)
- code segment

  typically read-only



memory segments

On Linux

```
> cat /proc/<pid>/maps
```

shows memory regions of process **`<pid>`**

With

```
> ps
```

you get a listing of all processes,

like the Taskbar in windows

(This is not exam material)

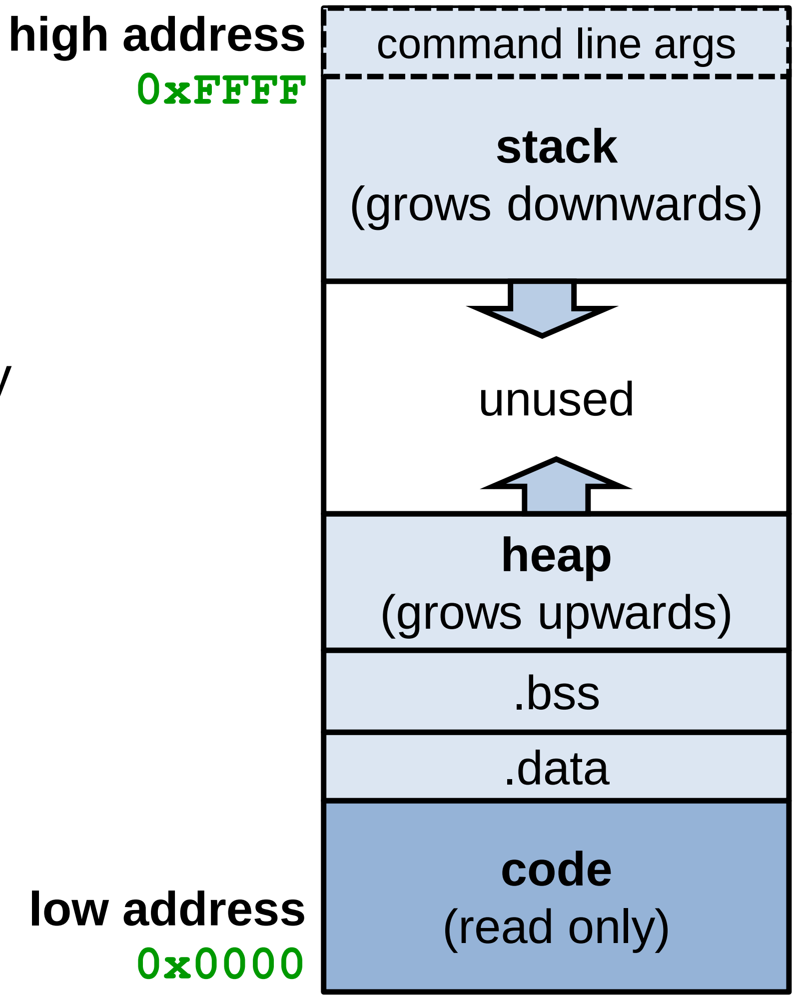
(Aside: real vs virtual memory)

Memory management depends on capabilities of

1. the hardware and
2. the operating system (OS)

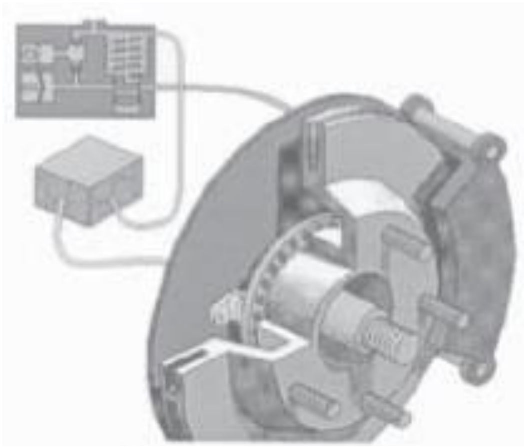
On primitive computers, which can only run a single process and have no real OS, the memory of the process may simply be all the *physical* memory

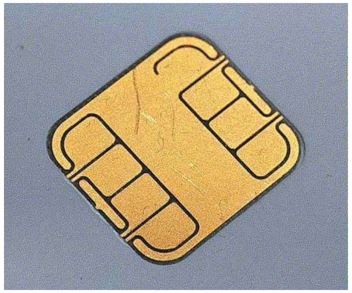
Eg, for an old 64K computer



(Aside: primitive computers)

These may only run
a single process which
then gets to use ***all***
of the memory





global variables (in `.bss` and `.data`)

These are the easy ones for the compiler to deal with.

```
#include <stdio.h>
long n = 12345;
char *string = "hello world\n";
int a[256];
...
```

Here

- the global variables **`n`**, **`string`** and the string literal **`"hello world\n"`**, will be allocated in data
- The uninitialised global array **`a`** will be allocated in .bss

  The segment .bss is initialised to all zeroes. NB this is a rare case where C will do a default initialisation for the programmer!

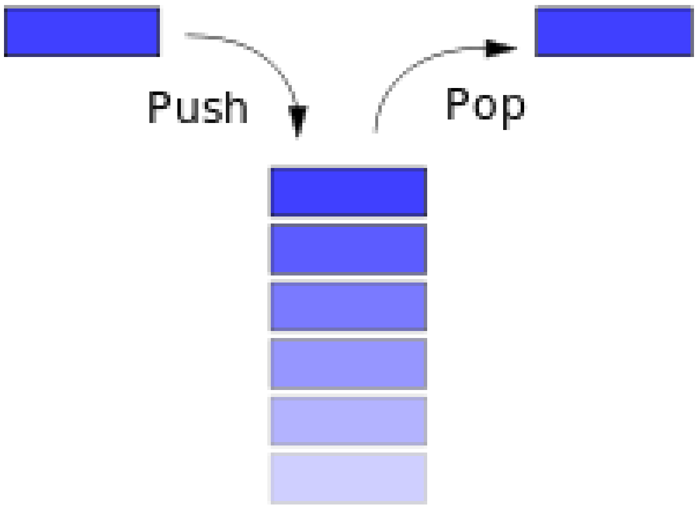
the stack

stack, pop, push

A stack (in Dutch: stapel) organises a set of elements in a Last In, First Out (LIFO) manner

The three basic operations on a stack are

- pushing a new element on the stack
- popping an element from the stack
- checking if the stack is empty



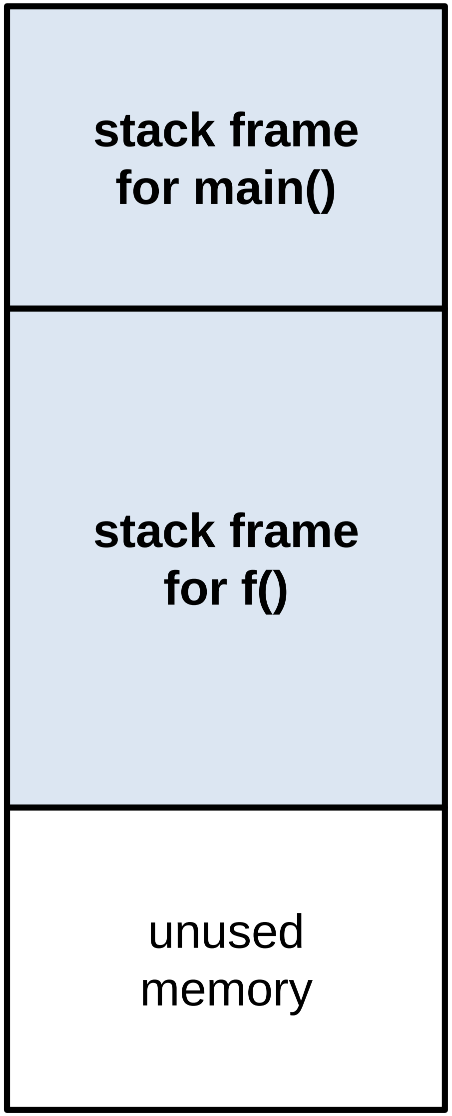
the stack

The stack consists of stack frames aka activation records, one for each function call,

- allocated when a function is called,
- de-allocated when it returns.

```
main(int i){
 char *msg ="hello";
 f(msg);
}

int f(char *p){
  int j;
  ..;
  return 5;
}
```

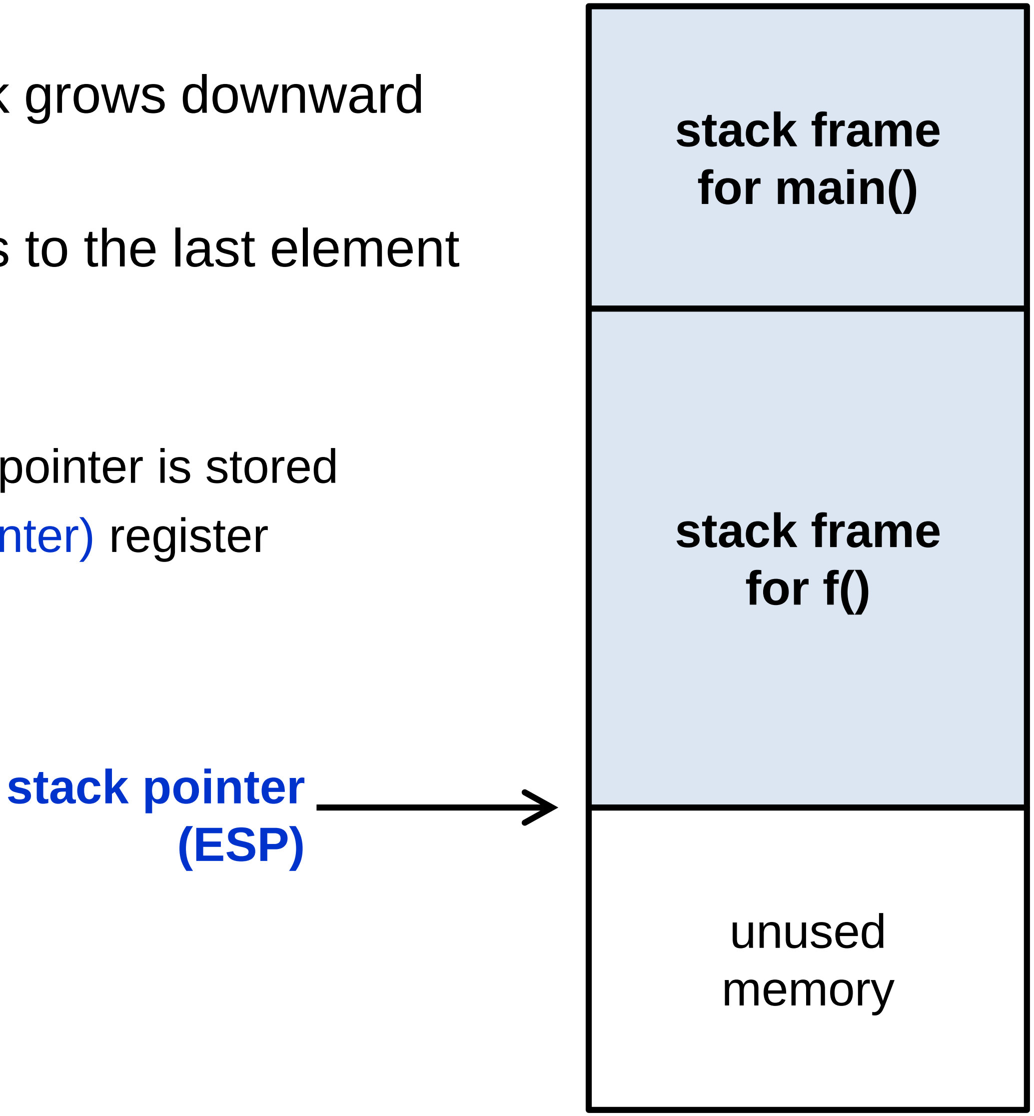


the stack

On most machines, the stack grows downward

The stack pointer (SP) points to the last element on the stack

On x86 architectures, the stack pointer is stored in the ESP (Extended Stack Pointer) register



the stack

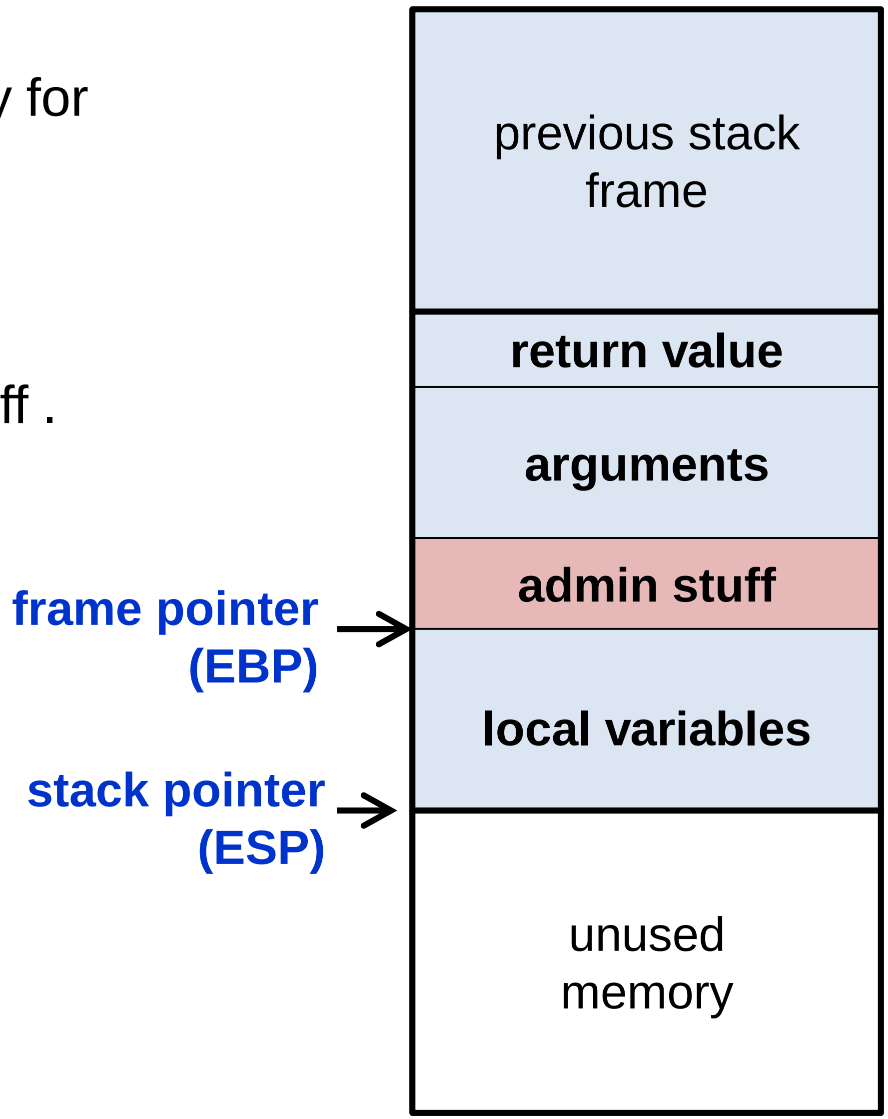
Each stack frame provides memory for

- arguments
- the return value
- local variables

of a function, plus some admin stuff .

The frame pointer provides a starting point to locate the local variables, using offsets.

On x86 architectures, it is stored in the EBP (Extended Base Pointer) register



the stack

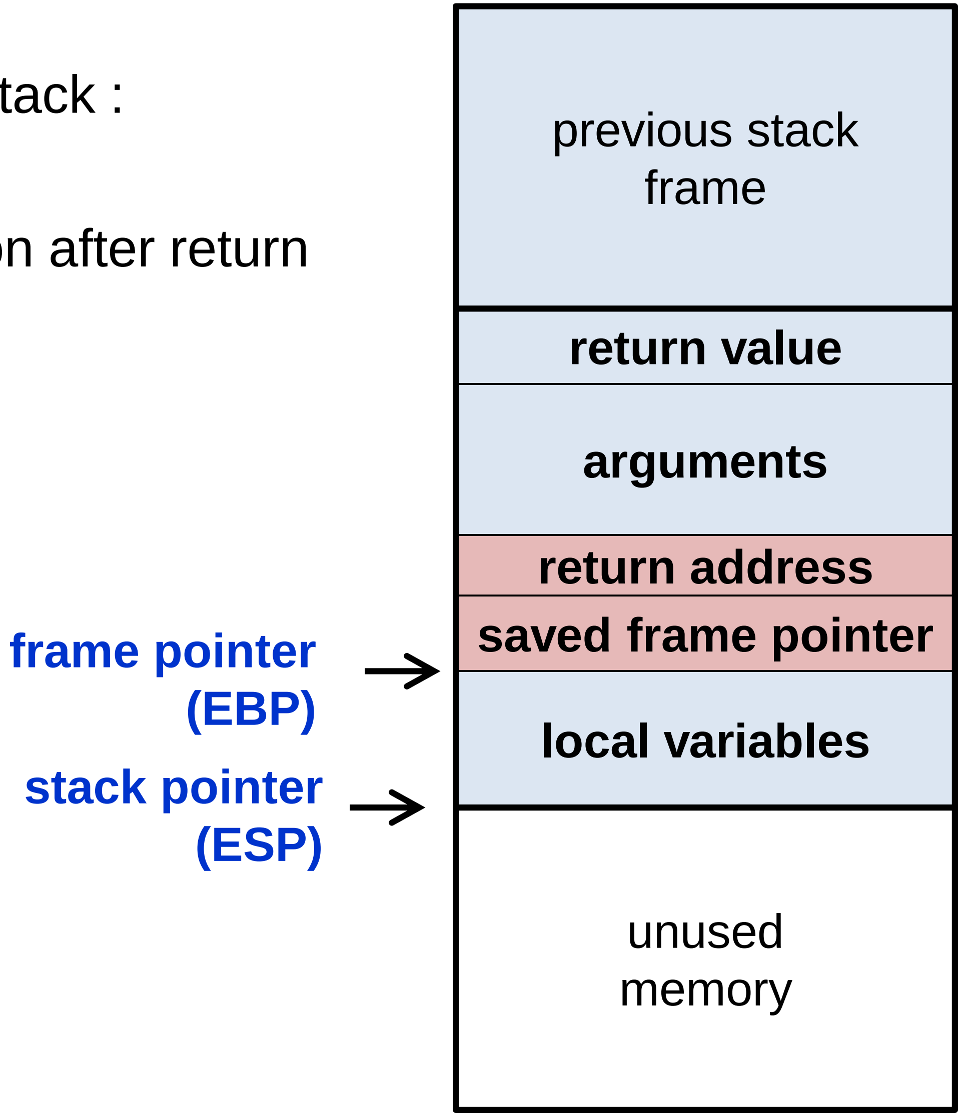
The admin stuff stored on the stack :

- return address

  ie where to resume execution after return

- previous frame pointer

  to locate previous frame

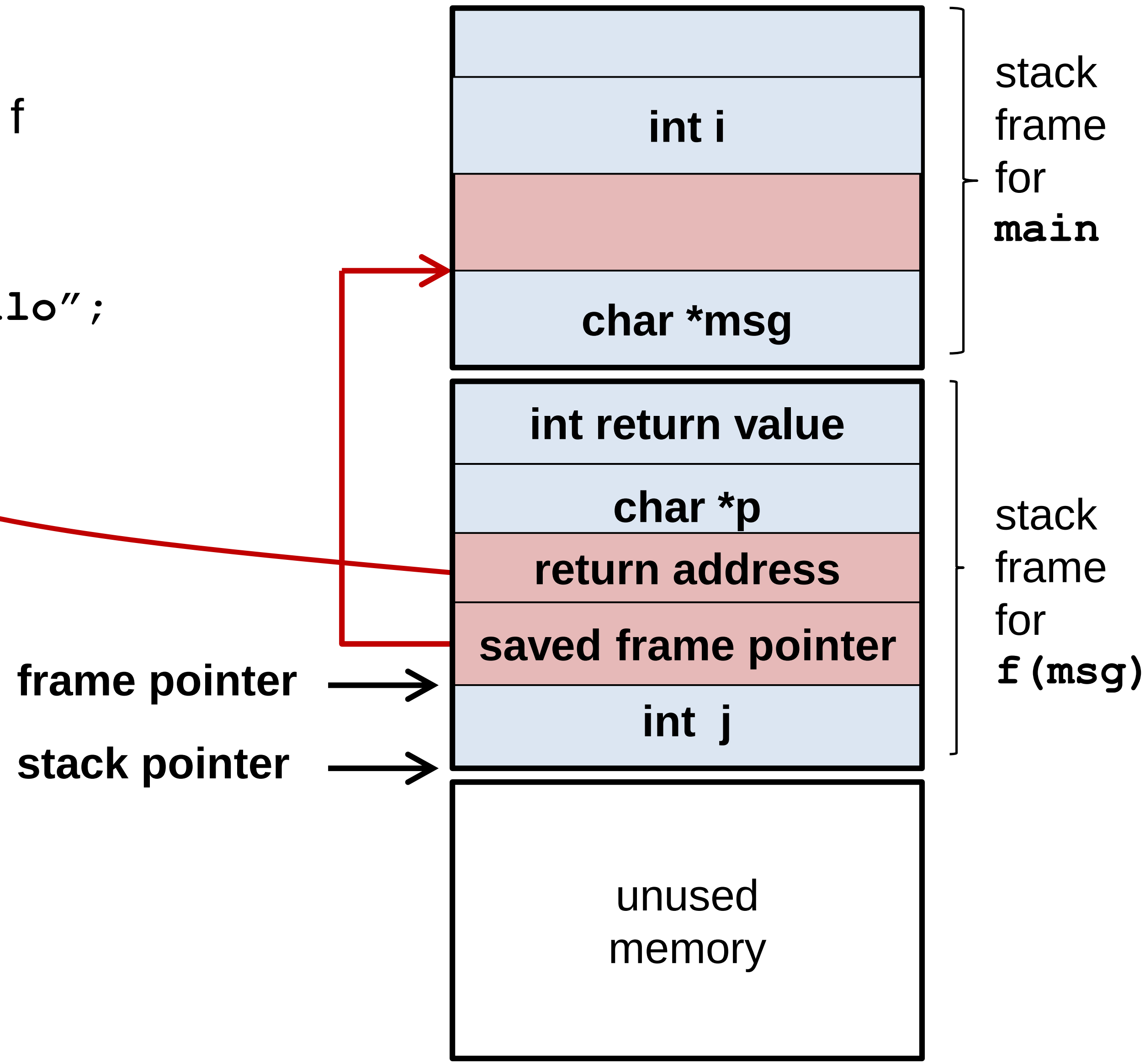


the stack

Stack during call to f

```
main(int i){
  char *msg ="hello";
  f(msg);
}
```

```
int f(char *p){
   int j;
   ..;
   return 5;
}
```



function calls

- When a function is called, a new stack frame is created
  - arguments are stored on the stack
  - current frame pointer and return address are recorded
  - memory for local variables is allocated
  - stack pointer is adjusted

- When a function returns, the top stack frame is removed
  - old frame pointer and return address are restored
  - stack pointer is adjusted
  - the caller can find the return value, if there is one, on top of the stack

- Because of recursion, there may be multiple frames for the same function on the stack
- Note that the variables that are stored in the current stack frame are precisely the variables that are in scope

security worries

- There is no default initialisation for stack variables
  - by reading unitialised local variables,

    you can read memory content used in earlier function calls

- There is only finite stack space
  - a function call may fail because there is no more memory

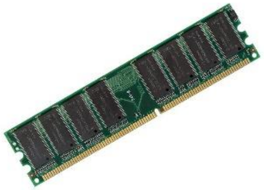
  In highly safety- or security-critical code, you may want to ensure that this cannot happen, or handle it in a safe way when it does.

- The stack mixes program data and control data
  - by overrunning buffers on the stack we can corrupt the return addresses!

  More on that the next weeks!

(Aside: hardware-specific details)

- The precise organisation of the stack depends on the machine architecture of the CPU

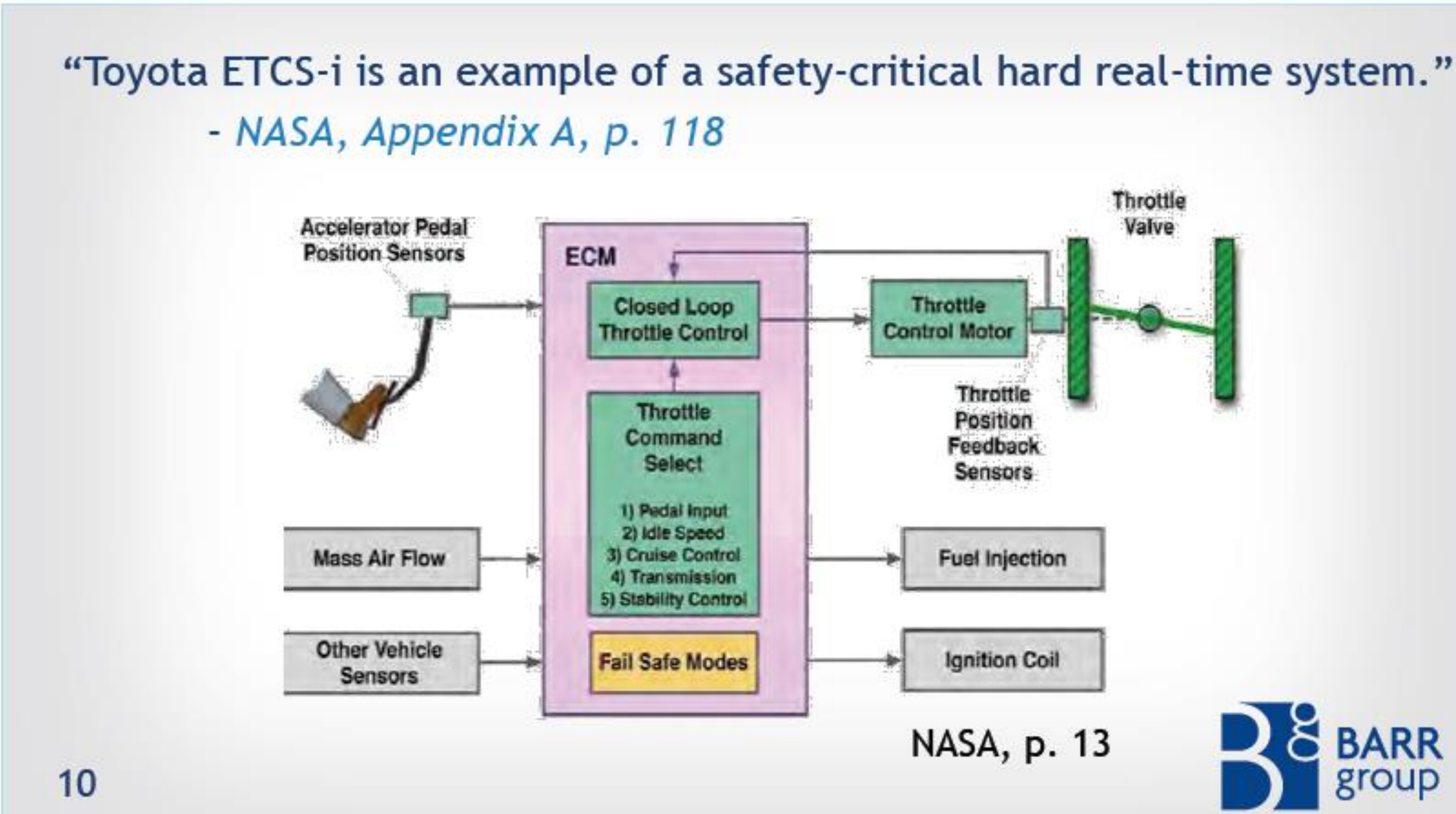


- Instead of storing data on the stack (in RAM)
  some data may be stored in a register (*in* the CPU)

Eg, for efficiency, the top values of the stack may be stored in CPU registers, or in the CPU cache, or the return value could be stored in a register instead of on the stack.

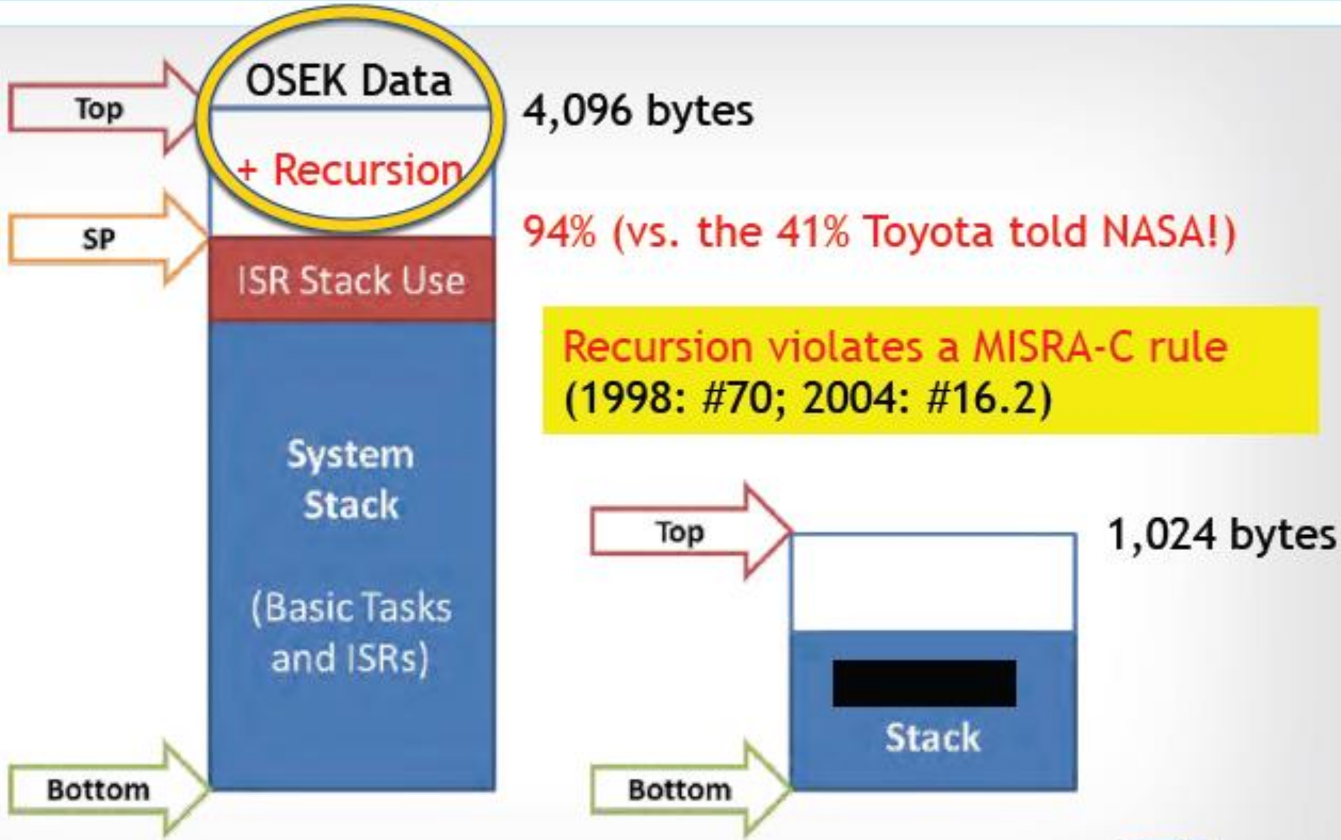
Example security problem caused by bad memory management

ELECTRONIC THROTTLE CONTROL (ETCS)



http://embeddedgurus.com/state-space/2014/02/are-we-shooting-ourselves-in-the-foot-with-stack-overflow/

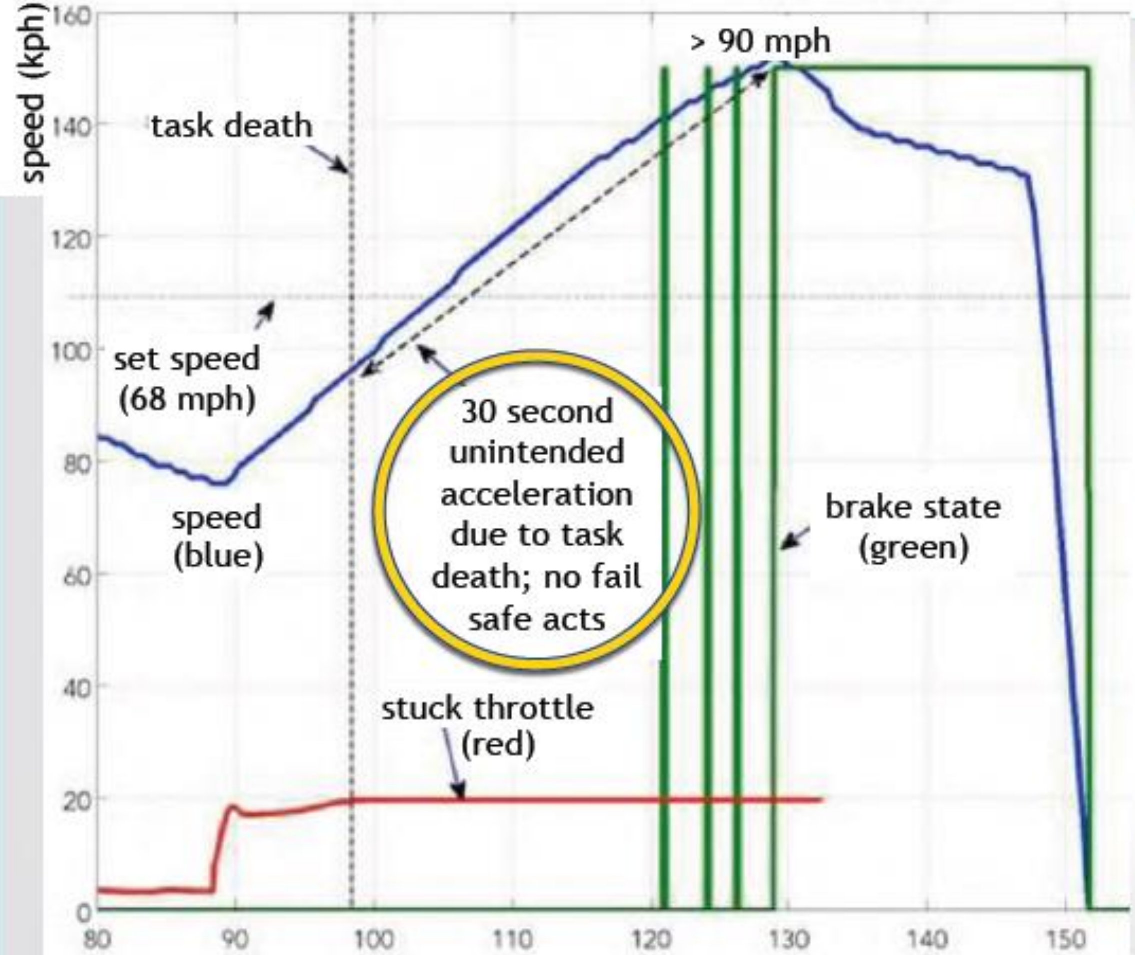
STACK ANALYSIS FOR 2005 CAMRY L4



Barr Chapter Regarding
Toyota's Stack Analysis

25

EXAMPLE OF UNINTENDED ACCELERATION



Source: Loudon Vehicle Testing

- Representative of task death in real-world
- Dead task also monitors accelerator pedal, so loss of throttle control
  - ✓ Confirmed in tests
- When this task's death begins with brake press (any amount), driver must <u>fully</u> remove foot from brake to end UA
  - ✓ Confirmed in tests

BARR group