

# Lecture 7 — Stack Management

Jeff Zarnett & Mike Cooper-Stachowsky

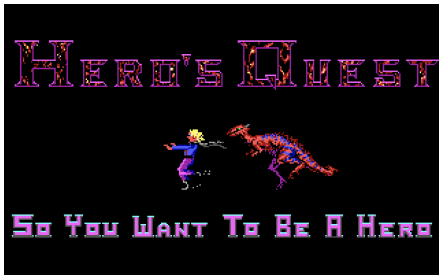
jzarnett@uwaterloo.ca, mstachowsky@uwaterloo.ca

Department of Electrical and Computer Engineering  
University of Waterloo

May 24, 2024

# So You Want To Manage a Stack

Stack management is a challenging but important subject.



Goals: learn about the stack, the Cortex M's stack pointers, and the Application-Binary Interface.

1. Local variables
2. Context variables: function args, return addresses...
3. Register data or other state storage

Remember that the stack is a LIFO data structure:

**Push** things on to the stack...

**Pop** things off of the stack...

Cannot remove things from the middle...

But I mean, it's just memory, so you *can* do bad things... but shouldn't.

---

```
int do_something( int x ) {  
    if ( x > 30 ) {  
        int y = 5;  
        return y+x;  
    }  
    return x - 10;  
}
```

---

- PUSH return address (but not x)
- If  $x > 30$ 
  - Push y
  - Add  $y + x$
  - Put result into Register R0
  - POP y
  - JUMP to return address
  - POP return address
- Otherwise:
  - Compute  $x - 10$
  - Put the result into Register R0
  - JUMP to return address
  - POP return address

# Function Call with Annotation



Some things in that list are a bit suspect. Why?

The compiler optimizes! Remove `int y = 5;` and make it return `5 + x`

Computations need memory locations so instead of adding `5 + x` somewhere else and then putting it into `R0`, make that the destination.

So the branch of `x > 30` probably looks more like:

- `LOAD x into R0`
- `R0 = R0 + 5`
- `JUMP to return address`
- `POP return address`

The example still gives us some idea about pushing and popping from the stack.

Every thread-local variable will have a stack location or a register.

When the function ends, everything from that function is popped off the stack.



PUSH and POP operations must be equal.



So no have memory leaks, though we can run out of space (overflow).

Typically we think of a stack like its physical representation.

Software stacks may grow down rather than up.

So PUSH decrements the stack pointer and POP increments it.

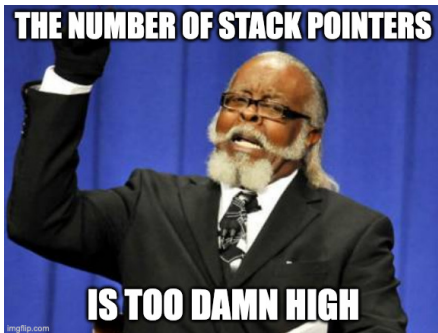
If the stack is empty and we push data: write it, decrement SP.

If it's not empty ("full"), decrement SP then write.

SP: “The” stack pointer, used by the processor to access the stack.

MSP: “Main” stack pointer, always used in interrupts, initialized via reset vector address 0x0.

PSP: “Process” stack pointer, never used in interrupts, initialized by user/programmer.



SP is directly connected to the CPU (need it).

PSP can never be used by privileged code; MSP is for privileged code.

At boot time SP is probably 0 (not good).

At load time memory segments get set up.

The vector reset table is used (see next slide).

The initial value of the MSP (SP) is 0x0...

Access it as: `uint32_t* MSP_INIT = *(uint32_t*) 0;`

# Reset Vector Table

Exception number	IRQ number	Offset	Vector
16+n	n	0x0040+4n	IRQn
.	.	.	.
.	.	.	.
.	.	.	.
18	2	0x004C	IRQ2
17	1	0x0048	IRQ1
16	0	0x0044	IRQ0
15	-1	0x0040	Systick
14	-2	0x003C	PendSV
13		0x0038	Reserved
12			Reserved for Debug
11	-5		SVCall
10		0x002C	Reserved
9			
8			
7			
6	-10		Usage fault
5	-11	0x0018	Bus fault
4	-12	0x0014	Memory management fault
3	-13	0x0010	Hard fault
2	-14	0x000C	NMI
1		0x0008	Reset
		0x0004	Initial SP value
		0x0000	

The ABI is rules developed by chip designers for compilers to know.



We'll look at the stack rules, but there's more complexity we're skipping.

ABI says the stack is full and descending.  
Any other organization is not following the rules.

Registers R0 - R3 are the “argument” and “result” registers.  
Precisely 4x4 bytes of space for arguments.

Each register is one of:

- A 4-byte type (e.g., `uint_32`)
- A smaller-than-4-byte type (space is wasted)
- A portion of a larger type (e.g., half of a `uint_64`)

Registers fill from R0 to R3.



---

```
uint_32 a, b;  
uint_64 c, d, e;
```

```
/* Initialization not shown */
```

```
foo( a, b ); // a goes in R0, b goes in R1
```

```
bar( a, b, c ); // a goes in R0, b in R1, c is split over R2 and R3.
```

```
big_fun( a, b, c, d, e ); // Where do they go?
```

---

---

```
uint_32 a, b;  
uint_64 c, d, e;  
  
/* Initialization not shown */  
  
foo( a, b ); // a goes in R0, b goes in R1  
  
bar( a, b, c ); // a goes in R0, b in R1, c is split over R2 and R3.  
  
big_fun( a, b, c, d, e ); // Where do they go?
```

---

You guessed it: additional function arguments go on the stack!

The compiler may need to add code to shuffle them around.

If the compiler is going to do it, do we need to know about it?

Yes, if we're going to call a C function from assembly!

```
MOV R0, #3  
MOV R1, #4  
BL  foo
```

The above is equivalent to calling `foo( 3, 4 );`

---

```
int f1( int x, int y ) {  
    return x + y;  
}  
  
int f2( int a, int b, int c ) {  
    int x = f1( 3, 4 );  
    return x + a + b + c;  
}
```

---

Can't put 3 and 4 in R0, R1 because they contain a, b that we still need.

Now what?



In this specific example the compiler can just replace the call to `f1` with the add operation and change the return statement to be: `return 7 + a + b + c.`

But suppose it did not or could not do that?

Solution is hardware saved registers!

R0 - R3, PC, LR, SP are pushed onto the stack by the hardware.  
So the compiler does not have to do this.

This is part of the “call stack” that helps us trace how we got here.

When a function ends, those values are popped & put back where they were.

Other registers R4 - R11 are called **scratch**.

We don't have to preserve them across function calls.

It will be our responsibility to store/restore them when switching between threads (context switch, scheduling).

The compiler doesn't manage them and things can change.

Any data type less than 4 bytes goes into R0.

Anything bigger than 4 bytes but less than 16 is spread over the needed amount of R0 - R3, like the input arguments.

After that, the compiler will put it elsewhere and return a pointer in R0.