# Procedures and the Stack

## Table of contents


Procedures . . . 1
`call` and `ret` . . . 2
Why a stack . . . 2
The AVR stack . . . 3
`push` and `pop` . . . 5
Arguments and results . . . 6
A C function's stack frame . . . 7
The stack and interrupts . . . 7
Pitfalls . . . 9
See also . . . 9


What a **procedure** is at the machine level, how `call` and `ret` get into it and back out, and the **stack** — the structure that makes returning possible from any depth of nesting, holds saved registers and arguments, and has to be set up before any of that works. First met in 2026-09-14-flow-of-control[1]; branches and jumps are in avr-branches-and-jumps[2], and data memory, where the stack lives, is in memory-models[3] and avr-architecture[4].

### Procedures

A **procedure** is a group of instructions that performs some task, which

- **may have operands** — *arguments* or *parameters*,
- **may produce result(s)**,
- **can be invoked (called) from multiple places** in a program — even from within itself, a **recursive** procedure — and

[1] courses/csse2010/2026-09-14-flow-of-control.pdf
[2] courses/csse2010/avr-branches-and-jumps.pdf
[3] courses/csse2010/memory-models.pdf
[4] courses/csse2010/avr-architecture.pdf

- **after invocation, control returns to the statement after the call.**

It is the same thing as a C **function** (c-language-basics[5]), a Python function, a Java **method**, or a **subroutine**; the names differ by language, the mechanism does not. Good use of procedures is most of what makes a program of any size readable, and the AVR C compiler turns every function into exactly what follows.

The return is the part with machinery behind it. A jump goes somewhere; a call goes somewhere **and remembers where it came from**, so that a `ret` at the far end can come back — to a different place each time, depending on who called.

### `call` and `ret`

| Instruction | | Size | Cycles |
|---|---|---|---|
| `call label` | direct call — target is an absolute address | 32-bit | 4 |
| `rcall label` | relative call — target is PC ± 2 K words | 16-bit | 3 |
| `icall` | indirect call — target is the address in **Z** | 16-bit | 3 |
| `ret` | return to the instruction after the most recent call | 16-bit | 4 |

The three calls mirror `jmp`, `rjmp` and `ijmp` exactly — same reach, same sizes — and each costs **one cycle more** than its jump. That cycle is the push. What a call does, in order:

1. compute the **return address** — the address of the instruction *after* the call;
2. **push** it onto the stack;
3. load the PC with the target.

And `ret` undoes it: **pop** the top of the stack into the PC. Nothing in the procedure needs to know who called it, because the address to go back to was left on the stack by the caller.

### Why a stack

The return address has to be stored *somewhere*. There are three candidates:

[5] courses/csse2010/c-language-basics.pdf

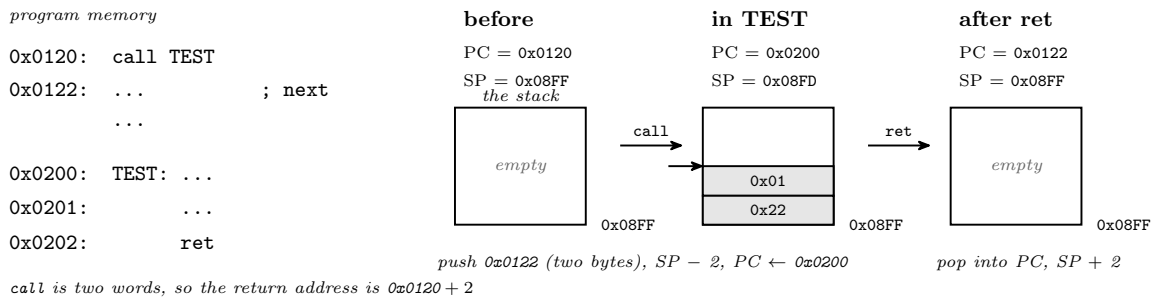



Figure 1: A call and its return, followed through the three registers involved. The PC, the stack pointer and the top of the stack are shown before the call, during the procedure and after the return, with example addresses. `call` records the address of its own successor on the stack and lowers the stack pointer by two, because a program-memory address needs two bytes; then it loads the PC with the procedure's address. `ret` reverses both steps and execution resumes at the recorded address, which is the instruction after the call. The procedure body never touches the return address, so it does not matter what the procedure does in between, provided it leaves the stack as it found it.

| Where | Fails when |
|---|---|
| A single fixed register or memory address for all calls | a procedure calls another procedure — the second call overwrites the first's return address, and the outer procedure can never return |
| One memory location per procedure | a procedure calls **itself** — recursion needs two live return addresses for one procedure |
| **A stack** | never, until memory runs out |

A **stack** is a last-in, first-out store: **push** puts a value on top, **pop** takes the top value off. Calls and returns nest — the most recent call is always the first to return — so a stack of return addresses always has the right one on top. Nested calls are drawn step by step in 2026-09-14-flow-of-control[6].

## The AVR stack

On the AVR the stack is simply a region of **data memory** (memory-models[7]), and the **stack pointer** is a 16-bit address held in two I/O registers, `SPH:SPL`, that says where the top currently

[6] courses/csse2010/2026-09-14-flow-of-control.pdf
[7] courses/csse2010/memory-models.pdf

is. Two conventions fix how it works:

- **The stack grows downward.** A push stores at the address in `SP` and then decrements `SP`; a pop increments `SP` and then loads from it. So `SP` always points at the next *free* cell, just below the most recently pushed byte.
- **It starts at the top of SRAM.** Program variables are allocated from the bottom of SRAM upward; the stack is started at the last address, `RAMEND` (`0x08FF` on the ATmega324A), and grows down toward them. Neither has to know in advance how much room the other needs.

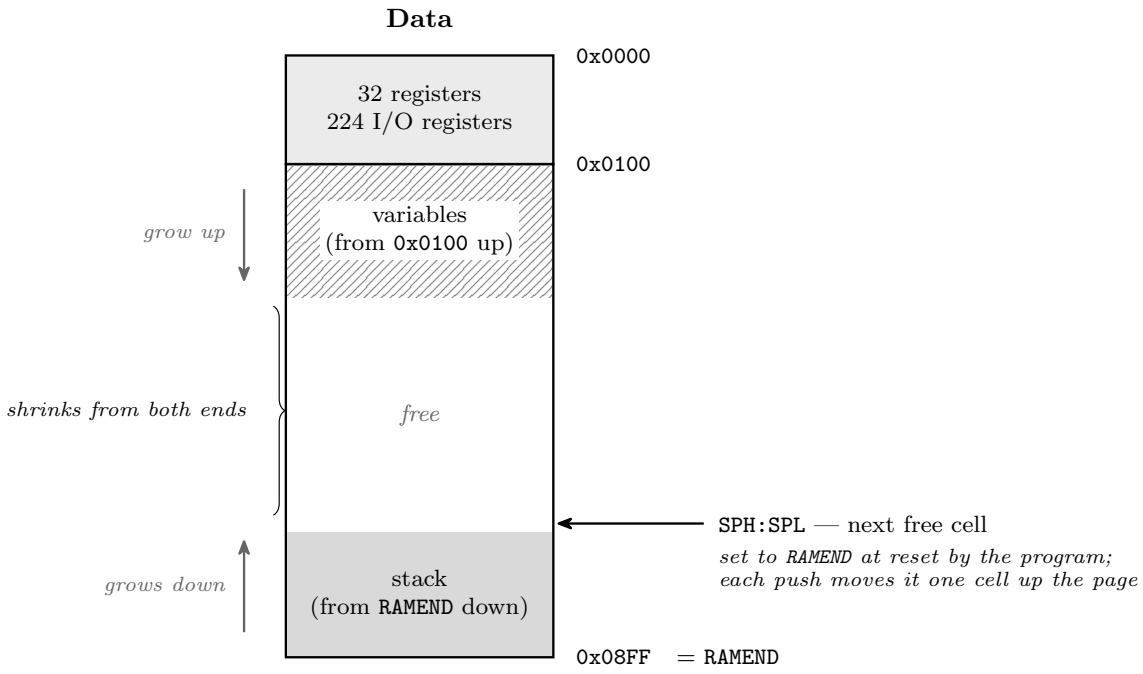



Figure 2: Where the stack lives in the ATmega324A's data memory. The general-purpose registers and I/O registers occupy the first 256 addresses; SRAM begins at `0x0100`. Global variables are laid out from the bottom of SRAM upward, and the stack starts at the top, `RAMEND`, and grows downward toward them, with `SPH:SPL` holding the address of the next free cell. The gap between them is the free memory, and it shrinks from both ends: every nested call, every `push` and every local variable of a running function takes cells off the top, and the stack overflows when it reaches the variables — silently, since nothing checks. That is why the stack pointer must be set to `RAMEND` before the first `call`: the reset value would put it somewhere else entirely.

### Initialising the stack pointer

**An assembly program must initialise the stack before using it.** These four lines belong at the top of `RESET:` in any program that calls anything:

```
.def temp=r16
    ldi  temp, low(RAMEND)
    out  SPL, temp
    ldi  temp, high(RAMEND)
    out  SPH, temp
```

`RAMEND` is defined by the device include file (avr-io-ports[8]), and `low()`/`high()` split the 16-bit value because `SPL` and `SPH` are two separate 8-bit I/O registers. **In C the compiler emits this itself** as part of the start-up code that runs before `main`; it is one of the things a C program on the AVR gets for free and an assembly program does not. The `RESET` template in avr-program-structure[9] does not include it only because nothing in the Lab 10 programs made a call.

### `push` and `pop`

The two stack instructions are also available directly:

| | | Cycles |
|---|---|---|
| **`push rr`** | store `rr` at `SP`, then `SP ← SP - 1` | 2 |
| **`pop rd`** | `SP ← SP + 1`, then load `rd` from `SP` | 2 |

Their main use is **saving registers a procedure is about to clobber**, so that the caller gets them back unchanged:

```
TEST:
    push r16            ; save what the caller had in r16 and r17
    push r17
    ...                 ; use r16, r17 freely
    pop  r17            ; restore - in REVERSE order
    pop  r16
    ret
```

[8] courses/csse2010/avr-io-ports.pdf
[9] courses/csse2010/avr-program-structure.pdf

The order matters because the stack is LIFO: the last register pushed is the first one back. And the count matters even more — **every push must be matched by a pop before the `ret`**, because `ret` pops whatever is on top and treats it as the return address. One push too many and the procedure returns to an address made of a register's contents; one too few and it returns into the middle of the saved data.

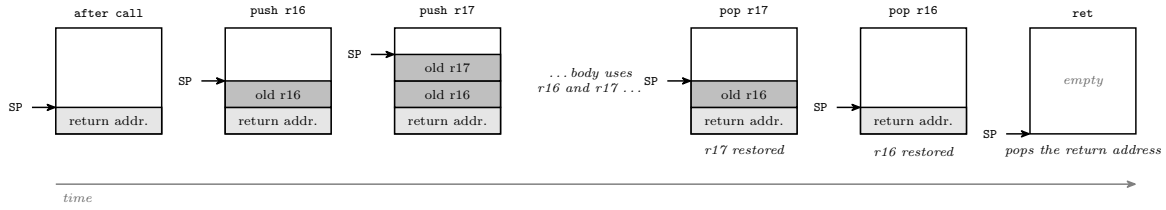



Figure 3: Saving and restoring registers around a procedure body, with the stack drawn after each instruction. The return address is already on the stack when the procedure starts; the two pushes go on top of it, and the two pops must come off in the reverse order to land in the right registers. When `ret` runs the return address is back on top, exactly as `call` left it. If the pops were in the same order as the pushes, `r16` and `r17` would come back swapped; if one pop were missing, `ret` would pop the saved `r16` and jump to whatever address that byte pair happened to spell.

## Arguments and results

Arguments are passed and results returned in one of two ways: **in registers, or on the stack.**

- **Registers** are the fast way for a few small values: the caller puts the argument in an agreed register before the `call`, the procedure reads it there and leaves its result in an agreed register for the caller to pick up. Agreement is the whole trick — both sides must use the same register, and a procedure that uses registers for arguments cannot also expect them preserved.
- **The stack** is the general way: the caller pushes the arguments, then calls; the procedure finds them on the stack just *above* the return address, and results can be returned the same way or in registers. The return address, the arguments and any saved registers and locals together are the procedure's **stack frame**. It costs pushes and pops, but it works for any number of arguments and for recursion, since every live call has its own frame.

The AVR C compiler uses both: the first few arguments of a function travel in registers, `r24` down, and the rest on the stack, and a function's local variables live in its stack frame. Those details are the compiler's convention, and they are why a C function call is not free — see the `main` costs in c-language-basics[10].

[10] courses/csse2010/c-language-basics.pdf

## A C function's stack frame

A C compiler puts a function's **local variables in its stack frame**, next to the return address and any arguments passed on the stack. The frame is **pushed when the function is called and popped when it returns**. A local therefore exists only while its call is running, and each call, including each level of a recursion, gets its own copy at its own address. An optimising compiler keeps a small local in a register when it can. The frame is where it goes when it needs memory, and the lifetime is the same either way.

In address order, from high to low, which is the order they are pushed:

1. **arguments**, pushed by the caller before the call, if any are passed on the stack;
2. the **return address**, pushed by `call`;
3. **locals**, made by the function itself when it starts.

A `static` **local is not in the frame.** It is stored with the globals, at a fixed address, so it outlives the call and still holds its value the next time the function runs. Globals sit at the bottom of SRAM, the heap grows up above them, and the stack grows down from `RAMEND`. A stack that grows far enough to meet the heap has **overflowed**. How the compiler lays out the whole data memory is in compilation-and-linking[11].

## The stack and interrupts

An interrupt uses the same stack in the same way. The hardware pushes the return address exactly as `call` would, and the handler's `reti` pops it, like `ret` (avr-interrupts[12], 2026-09-21-interrupts[13]). Three things follow:

- **The stack pointer must be set before interrupts are enabled**, not merely before the first `call`. An assembly program with no procedures at all still needs `SP` initialised once it uses an interrupt.
- **A handler's pushes and pops must balance, and it must restore every register it uses.** This is the same rule as for a procedure, but stricter: an interrupt can arrive between any two instructions, so there is no caller to agree anything with (polling-and-interrupts[14]).
- **Every interrupt that is being handled holds stack space**: two bytes of return address plus whatever its handler pushes, and more again if handlers are allowed to nest.

[11] courses/csse2010/compilation-and-linking.pdf
[12] courses/csse2010/avr-interrupts.pdf
[13] courses/csse2010/2026-09-21-interrupts.pdf
[14] courses/csse2010/polling-and-interrupts.pdf

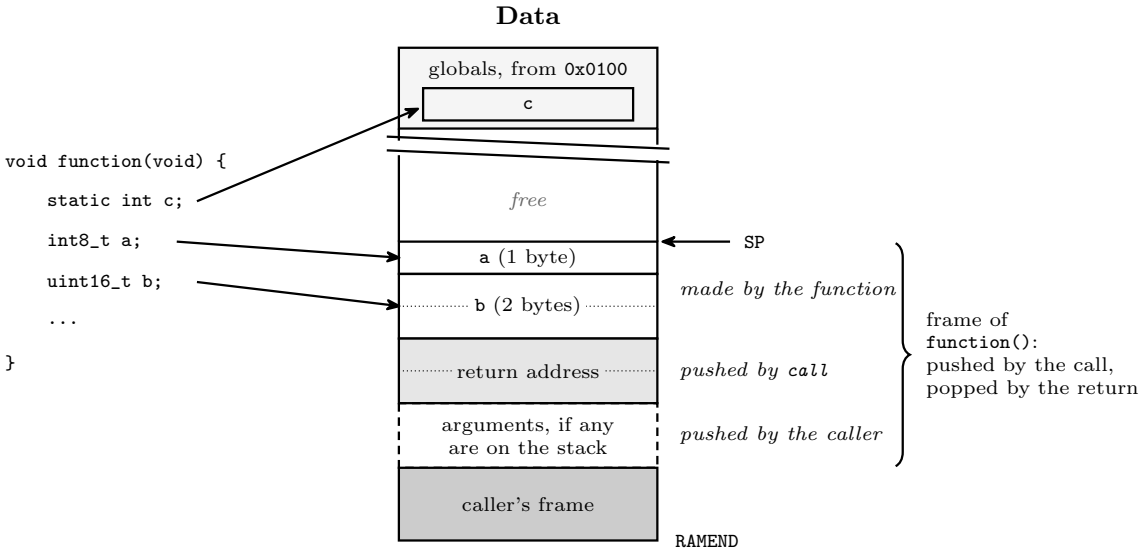



Figure 4: The stack frame of a C function while it runs, with the lowest address at the top as in the course's memory maps. The caller pushed any stack arguments, call pushed the return address, and the function made room for its locals a and b above that. SP points at the next free cell, just above the frame. When the function returns, the whole frame is popped and SP goes back to where it was before the call, so a and b no longer exist. The static local c is not in the frame at all. It is stored among the globals at a fixed address, which is why it keeps its value from one call to the next.

## Pitfalls

- **Forgetting to initialise `SP`** in an assembly program. The first `call` pushes to wherever the reset value points; the `ret` then reads garbage. In C, not a concern.
- **Unbalanced push/pop** inside a procedure — see Figure 3. `ret` does not check what it pops.
- **Stack overflow.** Deep recursion, or big local arrays in C, walk the stack down into the heap and the variables. Nothing on the chip detects it; the symptom is variables changing value for no visible reason.
- **Returning a pointer to a local.** The local's frame is popped by the return, so the pointer points at stack space that the next call will reuse.
- **Clobbering the caller's registers.** A procedure that uses `r16` without saving it changes `r16` for whoever called it. Either save and restore, or document which registers the procedure destroys — and be consistent.

## See also

- avr-branches-and-jumps[15] — `jmp`, `rjmp`, `ijmp`, and the conditional branches
- avr-program-structure[16] — the `RESET` template the stack initialisation goes into
- memory-models[17] · avr-architecture[18] — data memory, `RAMEND`, and the I/O registers `SPH`/`SPL`
- avr-instructions[19] — `push`, `pop`, `call`, `ret` among the rest
- c-language-basics[20] — functions in C, which compile to exactly this
- compilation-and-linking[21] — where the compiler puts globals, locals, statics and the heap · 2026-10-15-compilation-and-linking[22] — Lecture 22
- avr-interrupts[23] · polling-and-interrupts[24] — the same stack, used by the hardware
- 2026-09-14-flow-of-control[25] — Lecture 15 · 2026-09-21-interrupts[26] — Lecture 17, the stack under interrupts

[15] courses/csse2010/avr-branches-and-jumps.pdf
[16] courses/csse2010/avr-program-structure.pdf
[17] courses/csse2010/memory-models.pdf
[18] courses/csse2010/avr-architecture.pdf
[19] courses/csse2010/avr-instructions.pdf
[20] courses/csse2010/c-language-basics.pdf
[21] courses/csse2010/compilation-and-linking.pdf
[22] courses/csse2010/2026-10-15-compilation-and-linking.pdf
[23] courses/csse2010/avr-interrupts.pdf
[24] courses/csse2010/polling-and-interrupts.pdf
[25] courses/csse2010/2026-09-14-flow-of-control.pdf
[26] courses/csse2010/2026-09-21-interrupts.pdf