C Preprocessor and AVR Headers

Table of contents

What the preprocessor is 1
`#include` 2
`#define` — macros 3
The `avr/io.h` register macros 4
Naming bits 6
See also 7

The preprocessor — `#include` and `#define` — and the three lines of `avr/io.h` that let `DDRA = 0xFF;` be written as if a hardware register were a variable. First met in 2026-09-10-more-c-programming-on-the-avr[1]; the port registers themselves are in avr-io-ports[2].

What the preprocessor is

C programs are first parsed by a preprocessor, which **replaces preprocessor directives with substitute text**. Only then does the compiler see the program. Directives start with `#`:

Directive	Does
`#include …`	paste in a file
`#define …`	define a macro — a textual substitution
`#if`, `#ifdef`, `#else`, `#undef`, …	conditional inclusion, un-definition, and others

The preprocessor does not know C. It edits text. Whatever it produces is what the compiler compiles, and every surprise in this note comes from forgetting that.

[1] courses/csse2010/2026-09-10-more-c-programming-on-the-avr.pdf
[2] courses/csse2010/avr-io-ports.pdf

`#include`

Form	Kind of header	Looks in
`#include <filename.h>`	system	the system include directories — where `avr/io.h` and `stdio.h` live
`#include "filename.h"`	user-defined	your own project directory first

The directive is replaced by the whole contents of the named file, at that point. After preprocessing there is one file, with the header pasted in at the top.

Header files

Header (`.h`) files are used for common macros, function prototypes etc — the C counterpart of `m324Adef.inc` in assembly (avr-io-ports[3]).

- `stdio.h` — prototypes of `printf()` etc. The AVR has a simplified version (no files); `printf()` only works if you say where the characters go, e.g. a serial port.
- `avr/io.h` — defines the AVR's useful constants and functions/macros. **Definitions depend on the actual device targeted**: it reads which chip is being compiled for and pulls in that chip's register map, so `DDRA` means the right address for the ATmega324A and a different one for another part. It also supplies `uint8_t` and the other fixed-width types.

Every program in the course starts with `#include <avr/io.h>`.

Headers against libraries

A header file is not the library. A header

- **defines** useful preprocessor **macros**,
- **declares** the functions that are in the library, as **function prototypes**, and
- **declares** variables that are available externally, with **`extern`**.

The library is the collection of object files that implements the functions (compilation-and-linking[4]).

[3] courses/csse2010/avr-io-ports.pdf
[4] courses/csse2010/compilation-and-linking.pdf

A prototype such as `int printf(const char *fmt, ...);` tells the compiler the name, the argument types and the return type. That is enough for the compiler to check a call and generate it. There is no code in it, so in `main.o` the call to `printf` is left as an **external reference**. The code is in the library, an object file `printf.o` inside it, and the **linker** joins it on and fills in the address. An `extern` declaration does the same for a variable: `extern int count;` says that `count` exists somewhere else and reserves nothing.

So the header goes to the compiler, through the preprocessor, and the library goes to the linker. The two kinds of mistake show up at different steps:

Missing	Shows up when	Typical message
the `#include`	compiling	implicit declaration of function `printf`
the library, or the `.c` file that defines the function	linking	undefined reference to `printf`

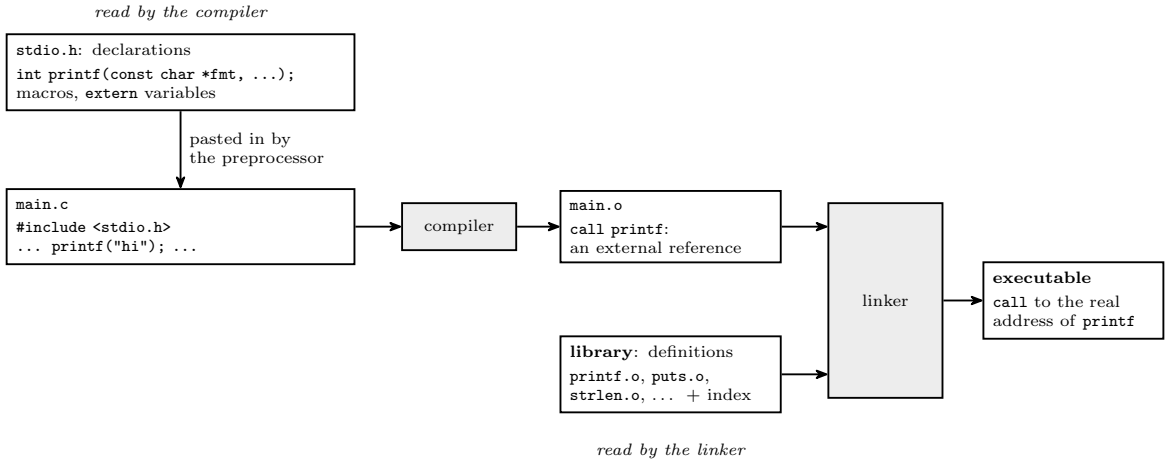


Figure 1: Where a header and a library each come into play, for a program that calls printf. The header holds only declarations. The preprocessor pastes it into main.c so that the compiler can check and generate the call, and in main.o that call is still an external reference. The library holds the definitions, as already compiled object files with an index. The linker finds printf.o through the index, links it in, and fills in the call. The header is used by the compiler and the library by the linker.

`#define` — macros

Macros are textual substitutions.

Macro	Effect
`#define pi 3.1415`	every later `pi` becomes `3.1415` — `A = pi * r²;` becomes `A = 3.1415 * r²;` (the superscript is shorthand; in real C the square would be `r * r`)
`#define EVER ;;`	`for(EVER)` becomes `for(;;)`, an infinite loop
`#define sum(a,b) (a+b)`	a macro **with parameters**: `sum(x, 2)` becomes `(x+2)`

A macro with parameters looks like a function call and is not one: no call happens, the arguments are pasted into the body as text, and the result is compiled in place.

The `SQUARE` trap

> Given `#define SQUARE(x) (x*x)`, what is `SQUARE(1+2)`?

5, not 9. Paste the text: `(1+2*1+2)`. Multiplication binds tighter than addition, so it is $1+2*1+2$. The argument was never evaluated first, because there is no function to evaluate it.

The fix is **parentheses around every parameter, and around the whole body**:

```
#define SQUARE(x) ((x)*(x))       /* SQUARE(1+2) -> ((1+2)*(1+2)) = 9 */
```

The outer pair matters as much as the inner: without it, `1 / SQUARE(a)` becomes `1 / (a)*(a)`, which is a, not $1/a^2$. **Write every macro this way.** The AVR headers below do — note the brackets around `(mem_addr)` and `(io_addr)`.

A second pitfall: an argument with a side effect is pasted **as many times as the parameter appears**. `SQUARE(i++)` becomes `((i++)*(i++))` and increments `i` twice.

The `avr/io.h` register macros

The definitions behind the port names, from `avr/io.h` and the device header it includes:

```
#define _MMIO_BYTE(mem_addr) \
                (*(volatile uint8_t *)(mem_addr))
#define _SFR_IO8(io_addr) \
                _MMIO_BYTE((io_addr) + 0x20)

/* Input Pins, Port A */
#define PINA      _SFR_IO8(0x00)
```

```
/* Data Direction Register, Port A */
#define DDRA      _SFR_IO8(0x01)

/* Data Register, Port A */
#define PORTA     _SFR_IO8(0x02)
```

(\ at the end of a line continues a directive onto the next line.)

Upshot: I/O register names can be used as variables.

```
DDRA = 0xFF;
PORTA = PINC | 0xF0;
```

Unrolling `DDRA`

Step	Text
as written	`DDRA = 0xFF;`
`#define DDRA`	`_SFR_IO8(0x01) = 0xFF;`
`#define _SFR_IO8`	`_MMIO_BYTE((0x01) + 0x20) = 0xFF;`
`#define _MMIO_BYTE`	`(*(volatile uint8_t *)((0x01) + 0x20)) = 0xFF;`

Reading the final line from the inside out:

1. `(0x01) + 0x20` is `0x21` — a number.
2. `(volatile uint8_t *)` casts it: *treat this number as the address of an 8-bit unsigned byte.* Now it is a pointer (c-pointers-and-arrays[5]).
3. The leading `*` dereferences it: *the byte at that address.*
4. `= 0xFF` stores into that byte.

So `DDRA = 0xFF;` is a store to data address `0x21`, and `x = PINC;` is a load from data address `0x26`. There is no variable called `DDRA` anywhere; the name dissolves before the compiler sees it.

[5] courses/csse2010/c-pointers-and-arrays.pdf

Why `+ 0x20`

The AVR's 64 I/O registers have **two addresses each** (avr-io-ports[6]): an *I/O-space* number 0–63, which is what `in` and `out` use, and a *data-space* address `0x20` higher, where `lds`/`sts` — and any C pointer — find the same register. `DDRA` is I/O `0x01` and data `0x21`. The macro takes the I/O number the datasheet lists and adds the offset, because a C pointer can only name data-space addresses. (The compiler, seeing a constant address in the I/O range, will usually emit `in`/`out` anyway.)

Why `volatile`

`volatile` tells the compiler **the value at this address can change without the program changing it**, and that **every read and write matters**. Without it the compiler is entitled to assume that a byte it wrote a moment ago still holds what it wrote, and that two reads with nothing between them give the same answer — both false for a port: `PINC` changes whenever the pins do, and `PORTA` is written for its effect on the pins, not to be read back. `volatile` forbids the optimisations that would break that. It is the one piece of the macro that is about hardware rather than addressing.

The same keyword belongs on any ordinary variable that an **interrupt handler** writes. To the code in `main`, a handler's write is as invisible as a pin changing, and for the same reason: nothing in `main`'s own flow of control explains it. See avr-interrupts[7].

Naming bits

The headers also define **bit numbers** for every flag and pin — for example `FLAG_NAME == 5` — so a mask is written from a name rather than a constant:

```
1 << FLAG_NAME              /* 00100000 when FLAG_NAME is 5 */
PORTA |= (1 << 5);          /* set one bit                  */
PORTA &= ~(1 << 5);         /* clear one bit                */
```

The shift and the `|=`/`&=` shorthands are in c-operators[8]; the mask patterns are in bit-masking[9]. `FLAG_NAME` is a placeholder name; the actual names for the ATmega324A's pins and flags come from the device header, as `#define WGM02 3` does in 2026-10-05-more-io[10].

[6] courses/csse2010/avr-io-ports.pdf
[7] courses/csse2010/avr-interrupts.pdf
[8] courses/csse2010/c-operators.pdf
[9] courses/csse2010/bit-masking.pdf
[10] courses/csse2010/2026-10-05-more-io.pdf

See also

- avr-io-ports[11] — the port registers, and the I/O-space versus data-space address columns
- c-pointers-and-arrays[12] — what `*(uint8_t *)addr` means
- c-operators[13] — `1 << n`, `|=`, `&=`
- bit-masking[14] · memory-models[15]
- avr-interrupts[16] — `volatile` on variables shared with an interrupt handler
- compilation-and-linking[17] — what happens after the preprocessor: object files, libraries and the linker · 2026-10-15-compilation-and-linking[18] — Lecture 22
- 2026-09-10-more-c-programming-on-the-avr[19] — Lecture 14

[11] courses/csse2010/avr-io-ports.pdf
[12] courses/csse2010/c-pointers-and-arrays.pdf
[13] courses/csse2010/c-operators.pdf
[14] courses/csse2010/bit-masking.pdf
[15] courses/csse2010/memory-models.pdf
[16] courses/csse2010/avr-interrupts.pdf
[17] courses/csse2010/compilation-and-linking.pdf
[18] courses/csse2010/2026-10-15-compilation-and-linking.pdf
[19] courses/csse2010/2026-09-10-more-c-programming-on-the-avr.pdf