Frequently Asked Questions

FAQ Index

My program doesn't recognize a variable updated within an interrupt routine

When using the optimizer, in a loop like the following one:

```
uint8_t flag;
...
ISR(SOME_vect) {
  flag = 1;
}
...

        while (flag == 0) {
                ...
        }
```

the compiler will typically access `flag` only once, and optimize further accesses completely away, since its code path analysis shows that nothing inside the loop could change the value of `flag` anyway. To tell the compiler that this variable could be changed outside the scope of its code path analysis (e. g. from within an interrupt routine), the variable needs to be declared like:

```
volatile uint8_t flag;
```

Back to **FAQ Index**.

I get "undefined reference to..." for functions like "sin()"

In order to access the mathematical functions that are declared in <**`math.h`**>, the linker needs to be told to also link the mathematical library, `libm.a`.

Typically, system libraries like `libm.a` are given to the final C compiler command line that performs the linking step by adding a flag `-lm` at the end. (That is, the initial *lib* and the filename suffix from the library are written immediately after a *-l* flag. So for a `libfoo.a` library, `-lfoo` needs to be provided.) This will make the linker search the library in a path known to the system.

An alternative would be to specify the full path to the `libm.a` file at the same place on the command line, i. e. *after* all the object files (`*.o`). However, since this requires knowledge of where the build system will exactly find those library files, this is deprecated for system libraries.

Back to **FAQ Index**.

How to permanently bind a variable to a register?

This can be done with

```
register unsigned char counter asm("r3");
```

Typically, it should be safe to use r2 through r7 that way.

Registers r8 through r15 can be used for argument passing by the compiler in case many or long arguments are being passed to callees. If this is not the case throughout the entire application, these registers could be used for register variables as well.

Extreme care should be taken that the entire application is compiled with a consistent set of register-allocated variables, including possibly used library functions.

See **C Names Used in Assembler Code** for more details.

Back to **FAQ Index**.

How to modify MCUCR or WDTCR early?

The method of early initialization (`MCUCR`, `WDTCR` or anything else) is different (and more flexible) in the current version. Basically, write a small assembler file which looks like this:

```
;; begin xram.S

#include <avr/io.h>

        .section .init1,"ax",@progbits

        ldi r16,_BV(SRE) | _BV(SRW)
        out _SFR_IO_ADDR(MCUCR),r16

;; end xram.S
```

Assemble it, link the resulting `xram.o` with other files in your program, and this piece of code will be inserted in initialization code, which is run right after reset. See the linker script for comments about the new `.init`*N* sections (which one to use, etc.).

The advantage of this method is that you can insert any initialization code you want (just remember that this is very early startup -- no stack and no `__zero_reg__` yet), and no program memory space is wasted if this feature is not used.

There should be no need to modify linker scripts anymore, except for some very special cases. It is best to leave `__stack` at its default value (end of internal SRAM -- faster, and required on some devices like ATmega161 because of errata), and add `-Wl,-Tdata,0x801100` to start the data section above the stack.

For more information on using sections, see **Memory Sections**. There is also an example for **Using Sections in C Code**. Note that in C code, any such function would preferably be placed into section .init3 as the code in .init2 ensures the internal register `__zero_reg__` is already cleared.

Back to **FAQ Index**.

What is all this _BV() stuff about?

When performing low-level output work, which is a very central point in microcontroller programming, it is quite common that a particular bit needs to be set or cleared in some IO register. While the device documentation provides mnemonic names for the various bits in the IO registers, and the **AVR device-specific IO definitions** reflect these names in definitions for numerical constants, a way is needed to convert a bit number (usually within a byte register) into a byte value that can be assigned directly to the register. However, sometimes the direct bit numbers are needed as well (e. g. in an `SBI()` instruction), so the definitions cannot usefully be made as byte values in the first place.

So in order to access a particular bit number as a byte value, use the **`_BV()`** macro. Of course, the implementation of this macro is just the usual bit shift (which is done by the compiler anyway, thus doesn't impose any run-time penalty), so the following applies:

```
_BV(3) => 1 << 3 => 0x08
```

However, using the macro often makes the program better readable.

"BV" stands for "bit value", in case someone might ask you. :-)

Example: clock timer 2 with full IO clock (`CS2`*x* = 0b001), toggle OC2 output on compare match (`COM2`*x* = 0b01), and clear timer on compare match (`CTC2` = 1). Make OC2 (`PD7`) an output.

```
        TCCR2 = _BV(COM20)|_BV(CTC2)|_BV(CS20);
        DDRD = _BV(PD7);
```

Back to **FAQ Index**.

Can I use C++ on the AVR?

Basically yes, C++ is supported (assuming your compiler has been configured and compiled to support it, of course). Source files ending in .cc, .cpp or .C will automatically cause the compiler frontend to invoke the C++ compiler. Alternatively, the C++ compiler could be explicitly called by the name `avr-c++`.

However, there's currently no support for `libstdc++`, the standard support library needed for a complete C++ implementation. This imposes a number of restrictions on the C++ programs that can be compiled. Among them are:

- Obviously, none of the C++ related standard functions, classes, and template classes are available.
- The operators `new` and `delete` are not implemented, attempting to use them will cause the linker to complain about undefined external references. (This could perhaps be fixed.)
- Some of the supplied include files are not C++ safe, i. e. they need to be wrapped into
  ```
  extern "C" { . . . }
  ```
  (This could certainly be fixed, too.)
- Exceptions are not supported. Since exceptions are enabled by default in the C++ frontend, they explicitly need to be turned off using `-fno-exceptions` in the compiler options. Failing this, the linker will complain about an undefined external reference to `__gxx_personality_sj0`.

Constructors and destructors *are* supported though, including global ones.

When programming C++ in space- and runtime-sensitive environments like microcontrollers, extra care should be taken to avoid unwanted side effects of the C++ calling conventions like implied copy constructors that could be called upon function invocation etc. These things could easily add up into a considerable amount of time and program memory wasted. Thus, casual inspection of the generated assembler code (using the `-S` compiler option) seems to be warranted.

Back to **FAQ Index**.

Shouldn't I initialize all my variables?

Global and static variables are guaranteed to be initialized to 0 by the C standard. `avr-gcc` does this by placing the appropriate code into section .init4 (see **The .initN Sections**). With respect to the standard, this sentence is somewhat simplified (because the standard allows for machines where the actual bit pattern used differs from all bits being 0), but for the AVR target, in general, all integer-type variables are set to 0, all pointers to a NULL pointer, and all floating-point variables to 0.0.

As long as these variables are not initialized (i. e. they don't have an equal sign and an initialization expression to the right within the definition of the variable), they go into the **.bss** section of the file. This section simply records the size of the variable, but otherwise doesn't consume space, neither within the object file nor within flash memory. (Of course, being a variable, it will consume space in the target's SRAM.)

In contrast, global and static variables that have an initializer go into the **.data** section of the file. This will cause them to consume space in the object file (in order to record the initializing value), *and* in the flash ROM of the target device. The latter is needed since the flash ROM is the only way that the compiler can tell the target device the value this variable is going to be initialized to.

Now if some programmer "wants to make doubly sure" their variables really get a 0 at program startup, and adds an initializer just containing 0 on the right-hand side, they waste space. While this waste of space applies to virtually any platform C is implemented on, it's usually not noticeable on larger machines like PCs, while the waste of flash ROM storage can be very painful on a small microcontroller like the AVR.

So in general, variables should only be explicitly initialized if the initial value is non-zero.

Note:
Recent versions of GCC are now smart enough to detect this situation, and revert variables that are explicitly initialized to 0 to the .bss section. Still, other compilers might not do that optimization, and as the C standard guarantees the initialization, it is safe to rely on it.

Back to **FAQ Index**.

Why do some 16-bit timer registers sometimes get trashed?

Some of the timer-related 16-bit IO registers use a temporary register (called TEMP in the Atmel datasheet) to guarantee an atomic access to the register despite the fact that two separate 8-bit IO transfers are required to actually move the data. Typically, this includes access to the current timer/counter value register (`TCNT`*n*), the input capture register (`ICR`*n*), and write access to the output compare registers (`OCR`*nM*). Refer to the actual datasheet for each device's set of registers that involves the TEMP register.

When accessing one of the registers that use TEMP from the main application, and possibly any other one from within an interrupt routine, care must be taken that no access from within an interrupt context could clobber the TEMP register data of an in-progress transaction that has just started elsewhere.

To protect interrupt routines against other interrupt routines, it's usually best to use the **ISR()** macro when declaring the interrupt function, and to ensure that interrupts are still disabled when accessing those 16-bit timer registers.

Within the main program, access to those registers could be encapsulated in calls to the **cli()** and **sei()** macros. If the status of the global interrupt flag before accessing one of those registers is uncertain, something like the following example code can be used.

```
uint16_t
read_timer1(void)
{
        uint8_t sreg;
        uint16_t val;

        sreg = SREG;
        cli();
        val = TCNT1;
        SREG = sreg;

        return val;
}
```

Back to **FAQ Index**.

How do I use a #define'd constant in an asm statement?

So you tried this:

```
asm volatile("sbi 0x18,0x07;");
```

Which works. When you do the same thing but replace the address of the port by its macro name, like this:

```
asm volatile("sbi PORTB,0x07;");
```

you get a compilation error: `"Error: constant value required"`.

`PORTB` is a precompiler definition included in the processor specific file included in **`avr/io.h`**. As you may know, the precompiler will not touch strings and `PORTB`, instead of `0x18`, gets passed to the assembler. One way to avoid this problem is:

```
asm volatile("sbi %0, 0x07" : "I" (_SFR_IO_ADDR(PORTB)):);
```

Note:
For C programs, rather use the standard C bit operators instead, so the above would be expressed as `PORTB |= (1 << 7)`. The optimizer will take care to transform this into a single SBI instruction, assuming the operands allow for this.

Back to **FAQ Index**.

Why does the PC randomly jump around when single-stepping through my program in avr-gdb?

When compiling a program with both optimization (`-O`) and debug information (`-g`) which is fortunately possible in `avr-gcc`, the code watched in the debugger is optimized code. While it is not guaranteed, very often this code runs with the exact same optimizations as it would run without the `-g` switch.

This can have unwanted side effects. Since the compiler is free to reorder code execution as long as the semantics do not change, code is often rearranged in order to make it possible to use a single branch instruction for conditional operations. Branch instructions can only cover a short range for the target PC (-63 through +64 words from the current PC). If a branch instruction cannot be used directly, the compiler needs to work around it by combining a skip instruction together with a relative jump (`rjmp`) instruction, which will need one additional word of ROM.

Another side effect of optimization is that variable usage is restricted to the area of code where it is actually used. So if a variable was placed in a register at the beginning of some function, this same register can be re-used later on if the compiler notices that the first variable is no longer used inside that function, even though the variable is still in lexical scope. When trying to examine the variable in `avr-gdb`, the displayed result will then look garbled.

So in order to avoid these side effects, optimization can be turned off while debugging. However, some of these optimizations might also have the side effect of uncovering bugs that would otherwise not be obvious, so it must be noted that turning off optimization can easily change the bug pattern. In most cases, you are better off leaving optimizations enabled while debugging.

Back to **FAQ Index**.

How do I trace an assembler file in avr-gdb?

When using the `-g` compiler option, `avr-gcc` only generates line number and other debug information for C (and C++) files that pass the compiler. Functions that don't have line number information will be completely skipped by a single `step` command in `gdb`. This includes functions linked from a standard library, but by default also functions defined in an assembler source file, since the `-g` compiler switch does not apply to the assembler.

So in order to debug an assembler input file (possibly one that has to be passed through the C preprocessor), it's the assembler that needs to be told to include line-number information into the output file. (Other debug information like data types and variable allocation cannot be generated, since unlike a compiler, the assembler basically doesn't know about this.) This is done using the (GNU) assembler option `--gstabs`.

Example:

```
$ avr-as -mmcu=atmega128 --gstabs -o foo.o foo.s
```

When the assembler is not called directly but through the C compiler frontend (either implicitly by passing a source file ending in .S, or explicitly using `-x assembler-with-cpp`), the compiler frontend needs to be told to pass the `--gstabs` option down to the assembler. This is done using `-Wa,--gstabs`. Please take care to *only* pass this option when compiling an assembler input file. Otherwise, the assembler code that results from the C compilation stage will also get line number information, which confuses the debugger.

Note:
You can also use `-Wa,-gstabs` since the compiler will add the extra `'-'` for you.

Example:

```
$ EXTRA_OPTS="-Wall -mmcu=atmega128 -x assembler-with-cpp"
$ avr-gcc -Wa,--gstabs ${EXTRA_OPTS} -c -o foo.o foo.S
```

Also note that the debugger might get confused when entering a piece of code that has a non-local label before, since it then takes this label as the name of a new function that appears to have been entered. Thus, the best practice to avoid this confusion is to only use non-local labels when declaring a new function, and restrict anything else to local labels. Local labels consist just of a number only. References to these labels consist of the number, followed by the letter **b** for a backward reference, or **f** for a forward reference. These local labels may be re-used within the source file, references will pick the closest label with the same number and given direction.

Example:

```
myfunc: push    r16
        push    r17
        push    r18
        push    YL
        push    YH
        ...
        eor     r16, r16        ; start loop
        ldi     YL, lo8(sometable)
        ldi     YH, hi8(sometable)
        rjmp    2f              ; jump to loop test at end
1:      ld      r17, Y+         ; loop continues here
        ...
        breq    1f              ; return from myfunc prematurely
        ...
        inc     r16
2:      cmp     r16, r18
        brlo    1b              ; jump back to top of loop

1:      pop     YH
        pop     YL
        pop     r18
        pop     r17
        pop     r16
        ret
```

Back to **FAQ Index**.

How do I pass an IO port as a parameter to a function?

Consider this example code:

```
#include <inttypes.h>
#include <avr/io.h>

void
set_bits_func_wrong (volatile uint8_t port, uint8_t mask)
{
    port |= mask;
}
```

```
void
set_bits_func_correct (volatile uint8_t *port, uint8_t mask)
{
    *port |= mask;
}

#define set_bits_macro(port,mask) ((port) |= (mask))

int main (void)
{
    set_bits_func_wrong (PORTB, 0xaa);
    set_bits_func_correct (&PORTB, 0x55);
    set_bits_macro (PORTB, 0xf0);

    return (0);
}
```

The first function will generate object code which is not even close to what is intended. The major problem arises when the function is called. When the compiler sees this call, it will actually pass the value of the `PORTB` register (using an `IN` instruction), instead of passing the address of `PORTB` (e.g. memory mapped io addr of `0x38`, io port `0x18` for the mega128). This is seen clearly when looking at the disassembly of the call:

```
    set_bits_func_wrong (PORTB, 0xaa);
 10a:   6a ea           ldi     r22, 0xAA       ; 170
 10c:   88 b3           in      r24, 0x18       ; 24
 10e:   0e 94 65 00     call    0xca
```

So, the function, once called, only sees the value of the port register and knows nothing about which port it came from. At this point, whatever object code is generated for the function by the compiler is irrelevant. The interested reader can examine the full disassembly to see that the function's body is completely fubar.

The second function shows how to pass (by reference) the memory mapped address of the io port to the function so that you can read and write to it in the function. Here's the object code generated for the function call:

```
    set_bits_func_correct (&PORTB, 0x55);
 112:   65 e5           ldi     r22, 0x55       ; 85
 114:   88 e3           ldi     r24, 0x38       ; 56
 116:   90 e0           ldi     r25, 0x00       ; 0
 118:   0e 94 7c 00     call    0xf8
```

You can clearly see that `0x0038` is correctly passed for the address of the io port. Looking at the disassembled object code for the body of the function, we can see that the function is indeed performing the operation we intended:

```
void
set_bits_func_correct (volatile uint8_t *port, uint8_t mask)
{
  f8:   fc 01           movw    r30, r24
    *port |= mask;
  fa:   80 81           ld      r24, Z
  fc:   86 2b           or      r24, r22
  fe:   80 83           st      Z, r24
}
 100:   08 95           ret
```

Notice that we are accessing the io port via the `LD` and `ST` instructions.

The `port` parameter must be volatile to avoid a compiler warning.

Note:

Because of the nature of the `IN` and `OUT` assembly instructions, they can not be used inside the function when passing the port in this way. Readers interested in the details should consult the *Instruction Set* datasheet.

Finally we come to the macro version of the operation. In this contrived example, the macro is the most efficient method with respect to both execution speed and code size:

```
    set_bits_macro (PORTB, 0xf0);
 11c:   88 b3           in      r24, 0x18       ; 24
 11e:   80 6f           ori     r24, 0xF0       ; 240
 120:   88 bb           out     0x18, r24       ; 24
```

Of course, in a real application, you might be doing a lot more in your function which uses a passed by reference io port address and thus the use of a function over a macro could save you some code space, but still at a cost of execution speed.

Care should be taken when such an indirect port access is going to one of the 16-bit IO registers where the order of write access is critical (like some timer registers). All versions of avr-gcc up to 3.3 will generate instructions that use the wrong access order in this situation (since with normal memory operands where the order doesn't matter, this sometimes yields shorter code).

See http://mail.nongnu.org/archive/html/avr-libc-dev/2003-01/msg00044.html for a possible workaround.

avr-gcc versions after 3.3 have been fixed in a way where this optimization will be disabled if the respective pointer variable is declared to be `volatile`, so the correct behaviour for 16-bit IO ports can be forced that way.

Back to **FAQ Index**.

What registers are used by the C compiler?

- **Data types:**
  `char` is 8 bits, `int` is 16 bits, `long` is 32 bits, `long` long is 64 bits, `float` and `double` are 32 bits (this is the only supported floating point format), pointers are 16 bits (function pointers are word addresses, to allow addressing up to 128K program memory space). There is a `-mint8` option (see **Options for the C compiler avr-gcc**) to make `int` 8 bits, but that is not supported by avr-libc and violates C standards (`int` *must* be at least 16 bits). It may be removed in a future release.

- **Call-used registers (r18-r27, r30-r31):**
  May be allocated by gcc for local data. You *may* use them freely in assembler subroutines. Calling C subroutines can clobber any of them - the caller is responsible for saving and restoring.

- **Call-saved registers (r2-r17, r28-r29):**
  May be allocated by gcc for local data. Calling C subroutines leaves them unchanged. Assembler subroutines are responsible for saving and restoring these registers, if changed. r29:r28 (Y pointer) is used as a frame pointer (points to local data on stack) if necessary. The requirement for the callee to save/preserve the contents of these registers even applies in situations where the compiler assigns them for argument passing.

- **Fixed registers (r0, r1):**
  Never allocated by gcc for local data, but often used for fixed purposes:

r0 - temporary register, can be clobbered by any C code (except interrupt handlers which save it), *may* be used to remember something for a while within one piece of assembler code

r1 - assumed to be always zero in any C code, *may* be used to remember something for a while within one piece of assembler code, but *must* then be cleared after use (`clr r1`). This includes any use of the `[f]mul[s[u]]` instructions, which return their result in r1:r0. Interrupt handlers save and clear r1 on entry, and restore r1 on exit (in case it was non-zero).

- **Function call conventions:**
  Arguments - allocated left to right, r25 to r8. All arguments are aligned to start in even-numbered registers (odd-sized arguments, including `char`, have one free register above them). This allows making better use of the `movw` instruction on the enhanced core.

If too many, those that don't fit are passed on the stack.

Return values: 8-bit in r24 (not r25!), 16-bit in r25:r24, up to 32 bits in r22-r25, up to 64 bits in r18-r25. 8-bit return values are zero/sign-extended to 16 bits by the called function (`unsigned char` is more efficient than `signed char` - just `clr r25`). Arguments to functions with variable argument lists (printf etc.) are all passed on stack, and `char` is extended to `int`.

Warning:
There was no such alignment before 2000-07-01, including the old patches for gcc-2.95.2. Check your old assembler subroutines, and adjust them accordingly.

Back to **FAQ Index**.

How do I put an array of strings completely in ROM?

There are times when you may need an array of strings which will never be modified. In this case, you don't want to waste ram storing the constant strings. The most obvious (and incorrect) thing to do is this:

```
#include <avr/pgmspace.h>

PGM_P array[2] PROGMEM = {
    "Foo",
    "Bar"
};

int main (void)
{
    char buf[32];
    strcpy_P (buf, array[1]);
    return 0;
}
```

The result is not what you want though. What you end up with is the array stored in ROM, while the individual strings end up in RAM (in the .data section).

To work around this, you need to do something like this:

```
#include <avr/pgmspace.h>

const char foo[] PROGMEM = "Foo";
const char bar[] PROGMEM = "Bar";

PGM_P array[2] PROGMEM = {
    foo,
    bar
};

int main (void)
{
    char buf[32];
    PGM_P p;
    int i;

    memcpy_P(&p, &array[i], sizeof(PGM_P));
    strcpy_P(buf, p);
    return 0;
}
```

Looking at the disassembly of the resulting object file we see that array is in flash as such:

```
00000026 <array>:
  26:   2e 00           .word   0x002e  ; ????
  28:   2a 00           .word   0x002a  ; ????

0000002a <bar>:
  2a:   42 61 72 00                                     Bar.

0000002e <foo>:
  2e:   46 6f 6f 00                                     Foo.
```

`foo` is at addr 0x002e.
`bar` is at addr 0x002a.
`array` is at addr 0x0026.

Then in main we see this:

```
    memcpy_P(&p, &array[i], sizeof(PGM_P));
  70:   66 0f           add     r22, r22
  72:   77 1f           adc     r23, r23
  74:   6a 5d           subi    r22, 0xDA       ; 218
  76:   7f 4f           sbci    r23, 0xFF       ; 255
  78:   42 e0           ldi     r20, 0x02       ; 2
  7a:   50 e0           ldi     r21, 0x00       ; 0
  7c:   ce 01           movw    r24, r28
  7e:   81 96           adiw    r24, 0x21       ; 33
  80:   08 d0           rcall   .+16            ; 0x92
```

This code reads the pointer to the desired string from the ROM table `array` into a register pair.

The value of `i` (in r22:r23) is doubled to accommodate for the word offset required to access array[], then the address of array (0x26) is added, by subtracting the negated address (0xffda). The address of variable `p` is computed by adding its offset within the stack frame (33) to the Y pointer register, and **`memcpy_P`** is called.

```
   strcpy_P(buf, p);
  82:    69 a1          ldd     r22, Y+33       ; 0x21
  84:    7a a1          ldd     r23, Y+34       ; 0x22
  86:    ce 01          movw    r24, r28
  88:    01 96          adiw    r24, 0x01       ; 1
  8a:    0c d0          rcall   .+24            ; 0xa4
```

This will finally copy the ROM string into the local buffer `buf`.

Variable `p` (located at Y+33) is read, and passed together with the address of buf (Y+1) to **`strcpy_P`**. This will copy the string from ROM to `buf`.

Note that when using a compile-time constant index, omitting the first step (reading the pointer from ROM via **`memcpy_P`**) usually remains unnoticed, since the compiler would then optimize the code for accessing `array` at compile-time.

Back to **FAQ Index**.

How to use external RAM?

Well, there is no universal answer to this question; it depends on what the external RAM is going to be used for.

Basically, the bit `SRE` (SRAM enable) in the `MCUCR` register needs to be set in order to enable the external memory interface. Depending on the device to be used, and the application details, further registers affecting the external memory operation like `XMCRA` and `XMCRB`, and/or further bits in `MCUCR` might be configured. Refer to the datasheet for details.

If the external RAM is going to be used to store the variables from the C program (i. e., the .data and/or .bss segment) in that memory area, it is essential to set up the external memory interface early during the **device initialization** so the initialization of these variable will take place. Refer to **How to modify MCUCR or WDTCR early?** for a description how to do this using few lines of assembler code, or to the chapter about memory sections for an **example written in C**.

The explanation of **malloc()** contains a **discussion** about the use of internal RAM vs. external RAM in particular with respect to the various possible locations of the *heap* (area reserved for **malloc()**). It also explains the linker command-line options that are required to move the memory regions away from their respective standard locations in internal RAM.

Finally, if the application simply wants to use the additional RAM for private data storage kept outside the domain of the C compiler (e. g. through a `char *` variable initialized directly to a particular address), it would be sufficient to defer the initialization of the external RAM interface to the beginning of **`main()`**, so no tweaking of the .init3 section is necessary. The same applies if only the heap is going to be located there, since the application start-up code does not affect the heap.

It is not recommended to locate the stack in external RAM. In general, accessing external RAM is slower than internal RAM, and errata of some AVR devices even prevent this configuration from working properly at all.

Back to **FAQ Index**.

Which -O flag to use?

There's a common misconception that larger numbers behind the `-O` option might automatically cause "better" optimization. First, there's no universal definition for "better", with optimization often being a speed vs. code size trade off. See the **detailed discussion** for which option affects which part of the code generation.

A test case was run on an ATmega128 to judge the effect of compiling the library itself using different optimization levels. The following table lists the results. The test case consisted of around 2 KB of strings to sort. Test #1 used **qsort()** using the standard library **strcmp()**, test #2 used a function that sorted the strings by their size (thus had two calls to **strlen()** per invocation).

When comparing the resulting code size, it should be noted that a floating point version of fvprintf() was linked into the binary (in order to print out the time elapsed) which is entirely not affected by the different optimization levels, and added about 2.5 KB to the code.

Optimization flags	Size of .text	Time for test #1	Time for test #2
-O3	6898	903 Âμs	19.7 ms
-O2	6666	972 Âμs	20.1 ms
-Os	6618	955 Âμs	20.1 ms
-Os -mcall-prologues	6474	972 Âμs	20.1 ms

(The difference between 955 Âμs and 972 Âμs was just a single timer-tick, so take this with a grain of salt.)

So generally, it seems `-Os -mcall-prologues` is the most universal "best" optimization level. Only applications that need to get the last few percent of speed benefit from using `-O3`.

Back to **FAQ Index**.

How do I relocate code to a fixed address?

First, the code should be put into a new **named section**. This is done with a section attribute:

```
__attribute__ ((section (".bootloader")))
```

In this example, .bootloader is the name of the new section. This attribute needs to be placed after the prototype of any function to force the function into the new section.

```
void boot(void) __attribute__ ((section (".bootloader")));
```

To relocate the section to a fixed address the linker flag `--section-start` is used. This option can be passed to the linker using the **-Wl compiler option**:

```
-Wl,--section-start=.bootloader=0x1E000
```

The name after section-start is the name of the section to be relocated. The number after the section name is the beginning address of the named section.

Back to **FAQ Index**.

My UART is generating nonsense! My ATmega128 keeps crashing! Port F is completely broken!

Well, certain odd problems arise out of the situation that the AVR devices as shipped by Atmel often come with a default fuse bit configuration that doesn't match the user's expectations. Here is a list of things to care for:

- All devices that have an internal RC oscillator ship with the fuse enabled that causes the device to run off this oscillator, instead of an external crystal. This often remains unnoticed until the first attempt is made to use something critical in timing, like UART communication.
- The ATmega128 ships with the fuse enabled that turns this device into ATmega103 compatibility mode. This means that some ports are not fully usable, and in particular that the internal SRAM is located at lower addresses. Since by default, the stack is located at the top of internal SRAM, a program compiled for an ATmega128 running on such a device will immediately crash upon the first function call (or rather, upon the first function return).
- Devices with a JTAG interface have the `JTAGEN` fuse programmed by default. This will make the respective port pins that are used for the JTAG interface unavailable for regular IO.

Back to **FAQ Index**.

Why do all my "foo...bar" strings eat up the SRAM?

By default, all strings are handled as all other initialized variables: they occupy RAM (even though the compiler might warn you when it detects write attempts to these RAM locations), and occupy the same amount of flash ROM so they can be initialized to the actual string by startup code. The compiler can optimize multiple identical strings into a single one, but obviously only for one compilation unit (i. e., a single C source file).

That way, any string literal will be a valid argument to any C function that expects a `const char *` argument.

Of course, this is going to waste a lot of SRAM. In **Program Space String Utilities**, a method is described how such constant data can be moved out to flash ROM. However, a constant string located in flash ROM is no longer a valid argument to pass to a function that expects a `const char *`-type string, since the AVR processor needs the special instruction `LPM` to access these strings. Thus, separate functions are needed that take this into account. Many of the standard C library functions have equivalents available where one of the string arguments can be located in flash ROM. Private functions in the applications need to handle this, too. For example, the following can be used to implement simple debugging messages that will be sent through a UART:

```
#include <inttypes.h>
#include <avr/io.h>
#include <avr/pgmspace.h>

int
uart_putchar(char c)
{
  if (c == '\n')
    uart_putchar('\r');
  loop_until_bit_is_set(USR, UDRE);
  UDR = c;
  return 0; /* so it could be used for fdevopen(), too */
}

void
debug_P(const char *addr)
{
  char c;

  while ((c = pgm_read_byte(addr++)))
    uart_putchar(c);
}

int
main(void)
{
  ioinit(); /* initialize UART, ... */
  debug_P(PSTR("foo was here\n"));
  return 0;
}
```

Note:
By convention, the suffix **_P** to the function name is used as an indication that this function is going to accept a "program-space string". Note also the use of the **PSTR()** macro.

Back to **FAQ Index**.

Why does the compiler compile an 8-bit operation that uses bitwise operators into a 16-bit operation in assembly?

Bitwise operations in Standard C will automatically promote their operands to an int, which is (by default) 16 bits in avr-gcc.

To work around this use typecasts on the operands, including literals, to declare that the values are to be 8 bit operands.

This may be especially important when clearing a bit:

```
var &= ~mask;  /* wrong way! */
```

The bitwise "not" operator (~) will also promote the value in `mask` to an int. To keep it an 8-bit value, typecast before the "not" operator:

```
var &= (unsigned char)~mask;
```

Back to **FAQ Index**.

How to detect RAM memory and variable overlap problems?

You can simply run `avr-nm` on your output (ELF) file. Run it with the `-n` option, and it will sort the symbols numerically (by default, they are sorted alphabetically).

Look for the symbol `_end`, that's the first address in RAM that is not allocated by a variable. (avr-gcc internally adds 0x800000 to all data/bss variable addresses, so please ignore this offset.) Then, the run-time initialization code initializes the stack pointer (by default) to point to the last available address in (internal) SRAM. Thus, the region between `_end` and the end of SRAM is what is available for stack. (If your application uses **malloc()**, which e. g. also can happen inside **printf()**, the heap for dynamic memory is also located there. See **Memory Areas and Using malloc()**.)

The amount of stack required for your application cannot be determined that easily. For example, if you recursively call a function and forget to break that recursion, the amount of stack required is infinite. :-) You can look at the generated assembler code (`avr-gcc ... -S`), there's a comment in each generated assembler file that tells you the frame size for each generated function. That's the amount of stack required for this function, you have to add up that for all functions where you know that the calls could be nested.

Back to **FAQ Index**.

Is it really impossible to program the ATtinyXX in C?

While some small AVRs are not directly supported by the C compiler since they do not have a RAM-based stack (and some do not even have RAM at all), it is possible anyway to use the general-purpose registers as a RAM replacement since they are mapped into the data memory region.

Bruce D. Lightner wrote an excellent description of how to do this, and offers this together with a toolkit on his web page:

http://lightner.net/avr/ATtinyAvrGcc.html

Back to **FAQ Index**.

What is this "clock skew detected" message?

It's a known problem of the MS-DOS FAT file system. Since the FAT file system has only a granularity of 2 seconds for maintaining a file's timestamp, and it seems that some MS-DOS derivative (Win9x) perhaps rounds up the current time to the next second when calculating the timestamp of an updated file in case the current time cannot be represented in FAT's terms, this causes a situation where `make` sees a "file coming from the future".

Since all make decisions are based on file timestamps, and their dependencies, make warns about this situation.

Solution: don't use inferior file systems / operating systems. Neither Unix file systems nor HPFS (aka NTFS) do experience that problem.

Workaround: after saving the file, wait a second before starting `make`. Or simply ignore the warning. If you are paranoid, execute a `make clean all` to make sure everything gets rebuilt.

In networked environments where the files are accessed from a file server, this message can also happen if the file server's clock differs too much from the network client's clock. In this case, the solution is to use a proper time keeping protocol on both systems, like NTP. As a workaround, synchronize the client's clock frequently with the server's clock.

Back to **FAQ Index**.

Why are (many) interrupt flags cleared by writing a logical 1?

Usually, each interrupt has its own interrupt flag bit in some control register, indicating the specified interrupt condition has been met by representing a logical 1 in the respective bit position. When working with interrupt handlers, this interrupt flag bit usually gets cleared automatically in the course of processing the interrupt, sometimes by just calling the handler at all, sometimes (e. g. for the U[S]ART) by reading a particular hardware register that will normally happen anyway when processing the interrupt.

From the hardware's point of view, an interrupt is asserted as long as the respective bit is set, while global interrupts are enabled. Thus, it is essential to have the bit cleared before interrupts get re-enabled again (which usually happens when returning from an interrupt handler).

Only few subsystems require an explicit action to clear the interrupt request when using interrupt handlers. (The notable exception is the TWI interface, where clearing the interrupt indicates to proceed with the TWI bus hardware handshake, so it's never done automatically.)

However, if no normal interrupt handlers are to be used, or in order to make extra sure any pending interrupt gets cleared before re-activating global interrupts (e. g. an external edge-triggered one), it can be necessary to explicitly clear the respective hardware interrupt bit by software. This is usually done by writing a logical 1 into this bit position. This seems to be illogical at first, the bit position already carries a logical 1 when reading it, so why does writing a logical 1 to it *clear* the interrupt bit?

The solution is simple: writing a logical 1 to it requires only a single `OUT` instruction, and it is clear that only this single interrupt request bit will be cleared. There is no need to perform a read-modify-write cycle (like, an `SBI` instruction), since all bits in these control registers are interrupt bits, and writing a logical 0 to the remaining bits (as it is done by the simple `OUT` instruction) will not alter them, so there is no risk of any race condition that might accidentally clear another interrupt request bit. So instead of writing

```
TIFR |= _BV(TOV0); /* wrong! */
```

simply use

```
TIFR = _BV(TOV0);
```

Back to **FAQ Index**.

Why have "programmed" fuses the bit value 0?

Basically, fuses are just a bit in a special EEPROM area. For technical reasons, erased E[E]PROM cells have all bits set to the value 1, so unprogrammed fuses also have a logical 1. Conversely, programmed fuse cells read out as bit value 0.

Back to **FAQ Index**.

Which AVR-specific assembler operators are available?

See **Pseudo-ops and operators**.

Back to **FAQ Index**.

Why are interrupts re-enabled in the middle of writing the stack pointer?

When setting up space for local variables on the stack, the compiler generates code like this:

```
/* prologue: frame size=20 */
        push r28
        push r29
        in r28,__SP_L__
        in r29,__SP_H__
        sbiw r28,20
        in __tmp_reg__,__SREG__
        cli
        out __SP_H__,r29
        out __SREG__,__tmp_reg__
        out __SP_L__,r28
/* prologue end (size=10) */
```

It reads the current stack pointer value, decrements it by the required amount of bytes, then disables interrupts, writes back the high part of the stack pointer, writes back the saved `SREG` (which will eventually re-enable interrupts if they have been enabled before), and finally writes the low part of the stack pointer.

At the first glance, there's a race between restoring `SREG`, and writing `SPL`. However, after enabling interrupts (either explicitly by setting the `I` flag, or by restoring it as part of the entire `SREG`), the AVR hardware executes (at least) the next instruction still with interrupts disabled, so the write to `SPL` is guaranteed to be executed with interrupts disabled still. Thus, the emitted sequence ensures interrupts will be disabled only for the minimum time required to guarantee the integrity of this operation.

Back to **FAQ Index**.

Why are there five different linker scripts?

From a comment in the source code:

Which one of the five linker script files is actually used depends on command line options given to ld.

A .x script file is the default script A .xr script is for linking without relocation (-r flag) A .xu script is like .xr but *do* create constructors (-Ur flag) A .xn script is for linking with -n flag (mix text and data on same page). A .xbn script is for linking with -N flag (mix text and data on same page).

Back to **FAQ Index**.

How to add a raw binary image to linker output?

The GNU linker `avr-ld` cannot handle binary data directly. However, there's a companion tool called `avr-objcopy`. This is already known from the output side: it's used to extract the contents of the linked ELF file into an Intel Hex load file.

`avr-objcopy` can create a relocatable object file from arbitrary binary input, like

```
avr-objcopy -I binary -O elf32-avr foo.bin foo.o
```

This will create a file named `foo.o`, with the contents of `foo.bin`. The contents will default to section .data, and two symbols will be created named `_binary_foo_bin_start` and `_binary_foo_bin_end`. These symbols can be referred to inside a C source to access these data.

If the goal is to have those data go to flash ROM (similar to having used the PROGMEM attribute in C source code), the sections have to be renamed while copying, and it's also useful to set the section flags:

```
avr-objcopy --rename-section .data=.progmem.data,contents,alloc,load,readonly,data -I binary -O elf32-avr foo.bin foo.o
```

Note that all this could be conveniently wired into a Makefile, so whenever `foo.bin` changes, it will trigger the recreation of `foo.o`, and a subsequent relink of the final ELF file.

Below are two Makefile fragments that provide rules to convert a .txt file to an object file, and to convert a .bin file to an object file:

```
$(OBJDIR)/%.o : %.txt
	@echo Converting $<
	@cp $(<) $(*).tmp
	@echo -n 0 | tr 0 '\000' >> $(*).tmp
	@$(OBJCOPY) -I binary -O elf32-avr \
	--rename-section .data=.progmem.data,contents,alloc,load,readonly,data \
	--redefine-sym _binary_$*_tmp_start=$* \
	--redefine-sym _binary_$*_tmp_end=$*_end \
	--redefine-sym _binary_$*_tmp_size=$*_size_sym \
	$(*).tmp $(@)
	@echo "extern const char" $(*)"[] PROGMEM;" > $(*).h
	@echo "extern const char" $(*)_end"[] PROGMEM;" >> $(*).h
	@echo "extern const char" $(*)_size_sym"[];" >> $(*).h
	@echo "#define $(*)_size ((int)$(*)_size_sym)" >> $(*).h
	@rm $(*).tmp

$(OBJDIR)/%.o : %.bin
	@echo Converting $<
	@$(OBJCOPY) -I binary -O elf32-avr \
	--rename-section .data=.progmem.data,contents,alloc,load,readonly,data \
	--redefine-sym _binary_$*_bin_start=$* \
	--redefine-sym _binary_$*_bin_end=$*_end \
	--redefine-sym _binary_$*_bin_size=$*_size_sym \
	$(<) $(@)
	@echo "extern const char" $(*)"[] PROGMEM;" > $(*).h
	@echo "extern const char" $(*)_end"[] PROGMEM;" >> $(*).h
	@echo "extern const char" $(*)_size_sym"[];" >> $(*).h
	@echo "#define $(*)_size ((int)$(*)_size_sym)" >> $(*).h
```

Back to **FAQ Index**.

How do I perform a software reset of the AVR?

The canonical way to perform a software reset of the AVR is to use the watchdog timer. Enable the watchdog timer to the shortest timeout setting, then go into an infinite, do-nothing loop. The watchdog will then reset the processor.

The reason why this is preferable over jumping to the reset vector, is that when the watchdog resets the AVR, the registers will be reset to their known, default settings. Whereas jumping to the reset vector will leave the registers in their previous state, which is generally not a good idea.

CAUTION! Older AVRs will have the watchdog timer disabled on a reset. For these older AVRs, doing a soft reset by enabling the watchdog is easy, as the watchdog will then be disabled after the reset. On newer AVRs, once the watchdog is enabled, then it **stays enabled, even after a reset**! For these newer AVRs a function needs to be added to the .init3 section (i.e. during the startup code, before main()) to disable the watchdog early enough so it does not continually reset the AVR.

Here is some example code that creates a macro that can be called to perform a soft reset:

```
#include <avr/wdt.h>

...

#define soft_reset()        \
do                          \
{                           \
    wdt_enable(WDTO_15MS);  \
    for(;;)                 \
    {                       \
    }                       \
} while(0)
```

For newer AVRs (such as the ATmega1281) also add this function to your code to then disable the watchdog after a reset (e.g., after a soft reset):

```
#include <avr/wdt.h>

...

// Function Pototype
void wdt_init(void) __attribute__((naked)) __attribute__((section(".init3")));

...

// Function Implementation
void wdt_init(void)
{
    MCUSR = 0;
    wdt_disable();

    return;
}
```

Back to **FAQ Index**.

I am using floating point math. Why is the compiled code so big? Why does my code not work?

You are not linking in the math library from AVR-LibC. GCC has a library that is used for floating point operations, but it is not optimized for the AVR, and so it generates big code, or it could be incorrect. This can happen even when you are not using any floating point math functions from the Standard C library, but you are just doing floating point math operations.

When you link in the math library from AVR-LibC, those routines get replaced by hand-optimized AVR assembly and it produces much smaller code.

See **I get "undefined reference to..." for functions like "sin()"** for more details on how to link in the math library.

Back to **FAQ Index**.

What pitfalls exist when writing reentrant code?

Reentrant code means the ability for a piece of code to be called simultaneously from two or more threads.

Attention to re-enterability is needed when using a multi-tasking operating system, or when using interrupts since an interrupt is really a temporary thread.

The code generated natively by gcc is reentrant. But, only some of the libraries in avr-libc are explicitly reentrant, and some are known not to be reentrant. In general, any library call that reads and writes global variables (including I/O registers) is not reentrant. This is because more than one thread could read or write the same storage at the same time, unaware that other threads are doing the same, and create inconsistent and/or erroneous results.

A library call that is known not to be reentrant will work if it is used only within one thread *and* no other thread makes use of a library call that shares common storage with it.

Below is a table of library calls with known issues.

Library call	Reentrant Issue	Workaround/Alternative
rand(), **random()**	Uses global variables to keep state information.	Use special reentrant versions: **rand_r()**, **random_r()**.
strtod(), **strtol()**, **strtoul()**	Uses the global variable `errno` to return success/failure.	Ignore `errno`, or protect calls with **cli()**/sei() or **ATOMIC_BLOCK()** if the application can tolerate it. Or use sccanf() or sccanf_P() if possible.
malloc(), **realloc()**, **calloc()**, **free()**	Uses the stack pointer and global variables to allocate and free memory.	Protect calls with **cli()**/sei() or **ATOMIC_BLOCK()** if the application can tolerate it. If using an OS, use the OS provided memory allocator since the OS is likely modifying the stack pointer anyway.
fdevopen(), **fclose()**	Uses **calloc()** and **free()**.	Protect calls with **cli()**/sei() or **ATOMIC_BLOCK()** if the application can tolerate it. Or use **fdev_setup_stream()** or **FDEV_SETUP_STREAM()**. Note: **fclose()** will only call **free()** if the stream has been opened with **fdevopen()**.
eeprom_*(), boot_*()	Accesses I/O registers.	Protect calls with **cli()**/sei(), **ATOMIC_BLOCK()**, or use OS locking.
pgm_*_far()	Accesses I/O register RAMPZ.	Starting with GCC 4.3, RAMPZ is automatically saved for ISRs, so nothing further is needed if only using interrupts. Some OSes may automatically preserve RAMPZ during context switching. Check the OS documentation before assuming it does. Otherwise, protect calls with **cli()**/sei(), **ATOMIC_BLOCK()**, or use explicit OS locking.
printf(), **printf_P()**, **vprintf()**, vprintf_P(), **puts()**, **puts_P()**	Alters flags and character count in global FILE `stdout`.	Use only in one thread. Or if returned character count is unimportant, do not use the *_P versions. Note: Formatting to a string output, e.g. **sprintf()**, **sprintf_P()**, **snprintf()**, **snprintf_P()**, **vsprintf()**, **vsprintf_P()**, **vsnprintf()**, **vsnprintf_P()**, is thread safe. The formatted string could then be followed by an **fwrite()** which simply calls the lower layer to send the string.
fprintf(), **fprintf_P()**, **vfprintf()**, **vfprintf_P()**, **fputs()**, **fputs_P()**	Alters flags and character count in the FILE argument. Problems can occur if a global FILE is used from multiple threads.	Assign each thread its own FILE for output. Or if returned character count is unimportant, do not use the *_P versions.
assert()	Contains an embedded **fprintf()**. See above for **fprintf()**.	See above for **fprintf()**.
clearerr()	Alters flags in the FILE argument.	Assign each thread its own FILE for output.
getchar(), **gets()**	Alters flags, character count, and unget buffer in global FILE `stdin`.	Use only in one thread. ***
fgetc(), **ungetc()**, **fgets()**, **scanf()**		

fgets(), scanf(), scanf_P(), fscanf(), fscanf_P(), vscanf(), vfscanf(), vfscanf_P(), fread()	Alters flags, character count, and unget buffer in the FILE argument.	Assign each thread its own FILE for input. *** Note: Scanning from a string, e.g. sscanf() and sscanf_P(), are thread safe.

*** It's not clear one would ever want to do character input simultaneously from more than one thread anyway, but these entries are included for completeness.

An effort will be made to keep this table up to date if any new issues are discovered or introduced.

Back to **FAQ Index**.

Why are some addresses of the EEPROM corrupted (usually address zero)?

The two most common reason for EEPROM corruption is either writing to the EEPROM beyond the datasheet endurance specification, or resetting the AVR while an EEPROM write is in progress.

EEPROM writes can take up to tens of milliseconds to complete. So that the CPU is not tied up for that long of time, an internal state-machine handles EEPROM write requests. The EEPROM state-machine expects to have all of the EEPROM registers setup, then an EEPROM write request to start the process. Once the EEPROM state-machine has started, changing EEPROM related registers during an EEPROM write is guaranteed to corrupt the EEPROM write process. The datasheet always shows the proper way to tell when a write is in progress, so that the registers are not changed by the user's program. The EEPROM state-machine will **always** complete the write in progress unless power is removed from the device.

As with all EEPROM technology, if power fails during an EEPROM write the state of the byte being written is undefined.

In older generation AVRs the EEPROM Address Register (EEAR) is initialized to zero on reset, be it from Brown Out Detect, Watchdog or the Reset Pin. If an EEPROM write has just started at the time of the reset, the write will be completed, but now at address zero instead of the requested address. If the reset occurs later in the write process both the requested address and address zero may be corrupted.

To distinguish which AVRs may exhibit the corrupt of address zero while a write is in process during a reset, look at the "initial value" section for the EEPROM Address Register. If EEAR shows the initial value as 0x00 or 0x0000, then address zero and possibly the one being written will be corrupted. Newer parts show the initial value as "undefined", these will not corrupt address zero during a reset (unless it was address zero that was being written).

EEPROMs have limited write endurance. The datasheet specifies the number of EEPROM writes that are guaranteed to function across the full temperature specification of the AVR, for a given byte. A read should always be performed before a write, to see if the value in the EEPROM actually needs to be written, so not to cause unnecessary EEPROM wear.

AVRs use a paging mechanism for doing EEPROM writes. This is almost entirely transparent to the user with one exception: When a byte is written to the EEPROM, the entire EEPROM page is also transparently erased and (re)written, which will cause wear to bytes that the programmer did not explicitly write. If it is desired to extend EEPROM write lifetimes, in an attempt not to exceed the datasheet EEPROM write endurance specification for a given byte, then writes must be in multiples of the EEPROM page size, and not sequential bytes. The EEPROM write page size varies with the device. The EEPROM page size is found in the datasheet section on Memory Programming, generally before the Electrical Specifications near the end of the datasheet.

The failure mechanism for an overwritten byte/page is generally one of "stuck" bits, i. e. a bit will stay at a one or zero state regardless of the byte written. Also a write followed by a read may return the correct data, but the data will change with the passage of time, due the EEPROM's inability to hold a charge from the excessive write wear.

Back to **FAQ Index**.

Why is my baud rate wrong?

Some AVR datasheets give the following formula for calculating baud rates:

```
(F_CPU/(UART_BAUD_RATE*16L)-1)
```

Unfortunately that formula does not work with all combinations of clock speeds and baud rates due to integer truncation during the division operator.

When doing integer division it is usually better to round to the nearest integer, rather than to the lowest. To do this add 0.5 (i. e. half the value of the denominator) to the numerator before the division, resulting in the formula:

```
((F_CPU + UART_BAUD_RATE * 8L) / (UART_BAUD_RATE * 16L) - 1)
```

This is also the way it is implemented in **<util/setbaud.h>: Helper macros for baud rate calculations**.

Back to **FAQ Index**.