**Contents**

1. ABI
    1. Type Layout
        1. Deviations from the Standard
        2. Fixed-Point Support
    2. Register Layout
        1. Fixed Registers
        2. Call-Used Registers
        3. Call-Saved Registers
    3. Frame Layout
    4. Calling Convention
        1. Exceptions to the Calling Convention
    5. Reduced Tiny
2. Extensions
    1. Types
    2. Attributes
    3. Pragmas
    4. Address Spaces
    5. Inline Assembly
        1. Constraint Modifiers
        2. Constraints
        3. Print Operand Modifiers
        4. Special Sequences
        5. Assembly Operand Modifiers
3. Using avr-gcc
    1. Locating .rodata in Flash for AVR64* and AVR128* Devices
    2. Supporting "unsupported" Devices
        1. avr-gcc v5 and newer
        2. avr-gcc v4.9 and below
4. Libf7
    1. Implementation
    2. Known Problems
    3. Using 64-bit long double without proper AVR-LibC
    4. Shortcomings
    5. Other Implementations

# ABI

Application Binary Interface and implementation defined behaviour of avr-gcc. Object format bits are not discussed here. See also C Implementation-defined behaviour.

## Type Layout

Endianess: Little

| **default** | sizeof | Note |
|---|---|---|
| char | 1 | signed |
| short | 2 | |
| int | 2 | |
| long | 4 | |

| | | |
|---|---|---|
| long long | 8 | |
| size_t | 2 | unsigned int |
| ptrdiff_t | 2 | int |
| | | |
| void* | 2 | |
| float | 4 | |
| double | 4,8 | depends on configuration and command line options |
| long double | 8,4 | depends on configuration and command line options |
| wchar_t | 2 | |

## Deviations from the Standard

### double

### long double

In avr-gcc up to v9, `double` and `long double` are only 32 bits wide and implemented in the same way as `float`.

In avr-gcc v10 and higher, the layout of `double` and `long double` are determined by configure options `--with-double=` and `--with-long-double=`, respectively. The default layout of double is like float, and the default layout of long double is a 64-bit IEEE format, see GCC configure options for details. Depending on the configuration, command line options `-mdouble=32` and `-mdouble=64` are available so that the type layout of `double` can be chosen at compile time, similar for `-mlong-double=32` and `-mlong-double=64` for `long double`. In order to test in a program which type layout has been chosen, GCC built-in macros `__SIZEOF_DOUBLE__` and `__SIZEOF_LONG_DOUBLE__` can be used.

### 8-bit int with -mint8

With `-mint8` int is only 8 bits wide which does not comply to the C standard. Notice that `-mint8` is not a multilib option and neither supported by AVR-LibC (except `stdint.h`) nor by newlib.

| **`-mint8`** | sizeof | Note |
|---|---|---|
| char | 1 | signed |
| short | 1 | |
| int | 1 | |
| long | 2 | |
| long long | 4 | |
| size_t | 2 | long unsigned int |
| ptrdiff_t | 2 | long int |

## Fixed-Point Support

avr-gcc 4.8 and up supports fixed point arithmetic according to ISO/IEC TR 18037. The support is not complete. The type layouts are as follows:

| Type | sizeof | unsigned | signed | Note |
|---|---|---|---|---|
| `_Fract` | | | | |
| short | 1 | 0.8 | ±.7 | |
| — | 2 | 0.16 | ±.15 | |
| long | 4 | 0.32 | ±.31 | |
| long long | 8 | 0.64 | ±.63 | GCC extension |
| `_Accum` | | | | |
| short | 2 | 8.8 | ±8.7 | |
| — | 4 | 16.16 | ±16.15 | |
| long | 8 | 32.32 | ±32.31 | |
| long long | 8 | 16.48 | ±16.47 | GCC extension |

Overflow behaviour of the non-saturated arithmetic is unspecified.

Please notice that some private ports found on the web implement different layouts.

## Register Layout

Values that occupy more than one 8-bit register start in an even register.

### Fixed Registers

*Fixed Registers* are registers that won't be allocated by GCC's register allocator. Registers R0 and R1 are fixed and used implicitly while printing out assembler instructions:

**R0**
: is used as scratch register that need not to be restored after its usage. It must be saved and restored in interrupt service routine's (ISR) prologue and epilogue. In inline assembler you can use `__tmp_reg__` for the scratch register.

**R1**
: always contains zero. During an insn the content might be destroyed, e.g. by a MUL instruction that uses R0/R1 as implicit output register. If an insn destroys R1, the insn must restore R1 to zero afterwards. This register must be saved in ISR prologues and must then be set to zero because R1 might contain values other than zero. The ISR epilogue restores the value. In inline assembler you can use `__zero_reg__` for the zero register.

**T**
: the T flag in the status register (SREG) is used in the same way like the temporary scratch register R0.

User-defined global registers by means of global `register asm` and / or `-ffixed-`*n* won't be saved or restored in function pro- and epilogue.

### Call-Used Registers

The *call-used* or *call-clobbered* general purpose registers (GPRs) are registers that might be destroyed (clobbered) by a function call.

**R18–R27, R30, R31**
: These GPRs are call clobbered. An ordinary function may use them without restoring the contents. Interrupt service routines (ISRs) must save and restore each register they use.

**R0, T-Flag**
: The temporary register and the T-flag in SREG are also call-clobbered, but this knowledge is not exposed explicitly to the compiler (R0 is a fixed register).

### Call-Saved Registers

**R2–R17, R28, R29**
: The remaining GPRs are call-saved, i.e. a function that uses such a registers must restore its original content. This applies even if the register is used to pass a function argument.

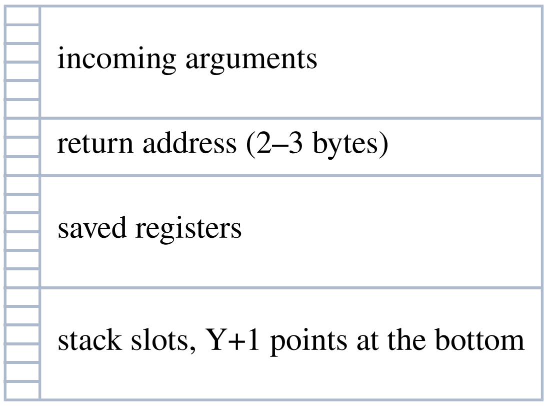
**R1**
: The zero-register is implicity call-saved (implicit because R1 is a fixed register).

## Frame Layout

During compilation the compiler may come up with an arbitrary number of *pseudo registers* which will be allocated to *hard registers* during register allocation.

- Pseudos that don't get a hard register will be put into a stack slot and loaded / stored as needed.
- In order to access stack locations, avr-gcc will set up a 16-bit frame pointer in R29:R28 (Y) because the stack pointer (SP) cannot be used to access stack slots.
- The stack grows downwards. Smaller addresses are at the bottom of the drawing at the right.
- Stack pointer and frame pointer are not aligned, i.e. 1-byte aligned.
- After the function prologue, the frame pointer will point one byte below the stack frame, i.e. Y+1 points to the bottom of the stack frame.
- Any of "incoming arguments", "saved registers" or "stack slots" in the drawing at the right may be empty.
- Even "return address" may be empty which happens for functions that are tail-called.

**Frame Layout after Function Prologue**


incoming arguments
return address (2–3 bytes)
saved registers
stack slots, Y+1 points at the bottom

## Calling Convention

- Neither function arguments nor function return values are promoted to wider types.
- An argument is passed either completely in registers or completely in memory.
- To find the register where a function argument is passed, initialize the register

number $R_n$ with R26 and follow this procedure:

    1. If the argument size is an odd number of bytes, round up the size to the next even number.
    2. Subtract the rounded size from the register number $R_n$.
    3. If the new $R_n$ is at least R8 and the size of the object is non-zero, then the low-byte of the argument is passed in $R_n$. Subsequent bytes of the argument are passed in the subsequent registers, i.e. in increasing register numbers.
    4. If the new register number $R_n$ is smaller than R8 or the size of the argument is zero, the argument will be passed in memory.
    5. If the current argument is passed in memory, stop the procedure: All subsequent arguments will also be passed in memory.
    6. If there are arguments left, goto 1. and proceed with the next argument.
- Return values with a size of 1 byte up to and including a size of 8 bytes will be returned in registers. Return values whose size is outside that range will be returned in memory. Sizes of structures up to 8 bytes are padded to the next power of two when they are returned in registers.
- If a return value cannot be returned in registers, the caller will allocate stack space and pass the address as implicit first pointer argument to the callee. The callee will put the return value into the space provided by the caller.
- If the return value of a function is returned in registers, the same registers are used as if the value was the first parameter of a non-varargs function. For example, an 8-bit value is returned in R24 and an 32-bit value is returned R22...R25.
- Arguments of varargs functions are passed on the stack. This applies even to the named arguments.

For example, suppose a function with the following prototype:

```
int func (char a, long b);
```

then

- a will be passed in R24.
- b will be passed in R20, R21, R22 and R23 with the LSB in R20 and the MSB in R23.
- The result is returned in R24 (LSB) and R25 (MSB).

## Exceptions to the Calling Convention

GCC comes with libgcc, a runtime support library. This library implements functions that are too complicated to be emit inline by GCC. What functions are used when depends on the target architecture, what instructions are available, how expensive they are and on the optimization level.

Functions in libgcc are implemented in C or hand-written assembly. In the latter case, some functions use a special ABI that allows better code generation by the compiler.

For example, the function that computes unsigned 8-bit quotient and remainder, `__udivmodqi4`, just returns the quotient and the remainder and clobbers R22 and R23. The compiler knows that the function does not destroy R30, for example, and may hold a value in R30 across the function call. This reduces the register pressure in functions that call `__udivmodqi4`.

| Function | Availability | Operation | Clobbers | Description |
|---|---|---|---|---|
| `__umulhisi3` | 4.7+ && MUL | `SI:22 = HI:26 * HI:18` | $R_{tmp}$ | Multiply 2 unsigned 16-bit integers to a 32-bit result |
| `__mulhisi3` | 4.7+ && MUL | `SI:22 = HI:26 * HI:18` | $R_{tmp}$ | Multiply 2 signed 16-bit integers to a 32-bit result |
| `__usmulhisi3` | 4.7+ && MUL | `SI:22 = HI:26 * HI:18` | $R_{tmp}$ | Multiply the signed 16-bit integer in R26 with the unsigned 16-bit integer in R18 to a 32-bit result |
| `__muluhisi3` | 4.7+ && MUL | `SI:22 = HI:26 * SI:18` | $R_{tmp}$ | Multiply an unsigned 16-bit integer with a 32-bit integer to a 32-bit result |
| `__mulshisi3` | 4.7+ && MUL | `SI:22 = HI:26 * SI:18` | $R_{tmp}$ | Multiply a signed 16-bit integer with a 32-bit integer to a 32-bit result |
| `__udivmodqi4` |  | `QI:24 = QI:24 / QI:22` `QI:25 = QI:24 % QI:22` | $R_{23}$ | Unsigned 8-bit integer quotient and remainder |
| `__divmodqi4` |  | `QI:24 = QI:24 / QI:22` `QI:25 = QI:24 % QI:22` | $R_{23}$, $R_{tmp}$, T | Signed 8-bit integer quotient and remainder |
| `__udivmodhi4` |  | `HI:22 = HI:24 / HI:22` `HI:24 = HI:24 % HI:22` | $R_{21}$, $R_{26...27}$ | Unsigned 16-bit |

| | | | | |
|---|---|---|---|---|
| | | | | integer quotient and remainder |
| `__divmodhi4` | | `HI:22 = HI:24 / HI:22`<br>`HI:24 = HI:24 % HI:22` | $R_{21}$, $R_{26...27}$, $R_{tmp}$, T | Signed 16-bit integer quotient and remainder |

The *Operation* column uses GCC's machine modes to describe how values in registers are interpreted.

| **Machine Modes** | **Q**arter, 8 bit | **H**alf, 16 bit | **S**ingle, 32 bit | **D**ouble, 64 bit | **P**artial **S**ingle, 24 bit |
|---|---|---|---|---|---|
| **I**nteger | QI | HI | SI | DI | PSI |
| **F**loat | – | – | SF | DF | – |
| Signed _**A**ccum | – | HA | SA | DA | – |
| Signed _Fract (**Q**-Format) | QQ | HQ | SQ | DQ | – |
| **U**nsigned _**A**ccum | – | UHA | USA | UDA | – |
| **U**nsigned _Fract (**Q**-Format) | UQQ | UHQ | USQ | UDQ | – |

## Reduced Tiny

On the Reduced Tiny cores (16 GPRs only) several modifications to the ABI above apply:

- Call-saved registers are: R18–R19, R28–R29.
- Fixed Registers are R16 (`__tmp_reg__`) and R17 (`__zero_reg__`).
- Registers used to pass arguments to functions and return values from functions are R25...R20 (instead of R25...R8).
- Values that occupy more than 4 registers are returned in memory (instead of more than 8 registers).

There is only limited library support both from libgcc and AVR-LibC, for example there is no float support and no printf support.

# Extensions

## Types

- Signed and unsigned 24-bit integers: `__int24` (v4.7), `__uint24` (v4.7).

## Attributes

- Variable: `progmem`, `absdata` (v7).
- Function: `interrupt`, `interrupt(n)` (v15), `signal`, `signal(n)` (v15), `naked`, `OS_main` (v4.4), `OS_task` (v4.4), `no_gccisr` (v8), `noblock` (v15).
- Type: (none).

## Pragmas

(none)

## Address Spaces

`__flash` (v4.7), `__flash1` ... `__flash5` (v4.7), `__memx` (v4.7).

Address spaces are supported as part of GNU-C. They are not supported for ISO C, and are not supported for C++.

avr-gcc puts objects in `__flash` into section `.progmem.data`, and objects in `__memx` into section `.progmemx.data`. These sections are handled in the default linker description file and need no further attention. This is different for the address spaces `__flashN`, where objects are put into section `.progmemN.data` but are **not mentioned** in the linker script, because there is no one-fits-all memory layout. This means you have to provide location directives for these sections. Suppose for example that an application uses address space `__flash2`, and therefore it has to locate the respective section somewhere in the range 0x20000-0x2ffff of flash memory. One way to achieve it is to use the following linker script augmentation:

```
SECTIONS
{
    .text :
    {
        . = MAX (ABSOLUTE(0x20000), ABSOLUTE(.));
        . = ALIGN(2);
        __progmem2_start = .;
        *(.progmem2.text*)
        *(.progmem2.data*)
        __progmem2_end = .;

        ASSERT (__progmem2_start == __progmem2_end || __progmem2_end <=
ABSOLUTE(0x30000),
                "progmem2.data exceeds 0x30000");
    }
}

INSERT AFTER .text
```

Store this text in a file `flash2.ld` and link with `-T flash2.ld`. This will locate in the order `text-progmem2-data` where `text` refers to "ordinary" code (startup-code, vector table, functions, progmem, jump-tables, etc.) and `data` refers to the data from which the startup-code initialized non-zero objects in static storage. If you want the order to be `text-data-progmem2` instead, then you would use `INSERT AFTER .data` in the snippet above.

## Inline Assembly

For introductions and tutorials on inline assembly, see

- GCC: How to Use Inline Assembly Language in C Code
- AVR-LibC: Inline Assembler Cookbook
- Roboternetz: Inine-Assembler in avr-gcc (Deutsch)
- Mikrocontroller.net: Assembler und Inline-Assembler (Deutsch)

### Constraint Modifiers

| Modifier | Meaning |
|---|---|
| `=` | An output operand, like in `"=r"`. Without &, the operand may overlap with input operands. |
| `&` | An output operand that may not overlap with any input operand, like in `"=&r"`. Referred to as *"early-clobber"*. |
| `+` | An output operand that is also an input operand, like in `"+r"`. |

### Constraints

| Constraint | Register | Range |
|---|---|---|
| `a` | Simple upper registers that support `FMUL` | R16...R23 |
| `b` | Base registers Y and Z | R28...R31 |
| `d` | Upper registers that support `LDI`, `ORI`, etc. | R16...R31 |
| `e` | Pointer registers X, Y, and Z | R26...R31 |
| `l` | Lower registers, empty on Reduced Tiny | R2...R15 |
| `r` | General purpose registers | R2...R31 |
| `w` | Registers for `ADIW` and `SBIW` | R24...R31 |
| `x` | X register | R26...R27 |
| `y` | Y register | R28...R29 |
| `z` | Z register | R30...R31 |

| Constraint | Constant | Range |
|---|---|---|
| `n` | Compile-time constant | |
| `s` | Symbolic operand known at link-time | Address of a function or static variable |
| `i` | Immediate operand known at link-time | Same as `"sn"` |
| `I` | Unsigned 6-bit integer constant | 0...63 |

| | | |
|---|---|---|
| `J` | Negative 6-bit integer constant | −63...0 |
| `M` | Unsigned 8-bit integer constant | 0...255 |
| `EF` | Floating-point constant | |
| `Ynn` | Fixed-point or integer constant | |

| Constraint | Explanation |
|---|---|
| `m` | Memory |
| `X` | Matches anything |
| `0`...`9` | Matches respective operand number |

- Specifying more than one constraint, like in `"az"`, specifies the union of all mentioned constraints.

## Print Operand Modifiers

| Modifier | Number of Arguments | Explanation | Suitable Constraints |
|---|---|---|---|
| `%a0` | 1 | Print pointer register as address `X`, `Y` or `Z`, like in `"LD r0, %a0+"` | `x`, `y`, `z`, `b`, `e` |
| `%i0` | 1 | Print compile-time RAM address as I/O address, like in `"OUT %i0, r0"` with argument `"n"(&RAMPZ)` | `n` |
| `%n0` | 1 | Print the negative of a compile-time constant | `n` |
| `%r0` | 1 | Print the register number of a register, like in `"CLR %r0+7"` for the MSB of a 64-bit register | reg |
| `%x0` | 1 | Print a function name without `gs()` modifier, like in `"%~CALL %x0"` with argument `"s"(main)` | `s` |
| `%A0` | 1 | Add 0 to the register number (no effect) | reg |
| `%B0` | 1 | Add 1 to the register number | reg |
| `%C0` | 1 | Add 2 to the register number | reg |
| `%D0` | 1 | Add 3 to the register number | reg |
| `%T0%t1` | 2 | Print the register that holds bit number `%1` of register `%0` | reg + `n` |
| `%T0%T1` | 2 | Print operands suitable for `BLD`/`BST`, like in `"BST %T0%T1"`, including the required `,` | reg + `n` |

- Register constraints are: `r`, `d`, `w`, `x`, `y`, `z`, `b`, `e`, `a`, `l`.

## Special Sequences

| Squence | Meaning |
|---|---|
| `%~` | `""` or `"r"`: `"%~call"` yields `"call"` on devices with `CALL`, and `"rcall"` on devices without `CALL` |
| `%!` | `""` or `"e"`: `"%!icall"` yields `"eicall"` on devices with `EICALL`, and `"icall"` on devices without `EICALL` |
| `%=` | A number that's unique for this inline assembly snip and the compilation unit. Used to compose unique local labels |
| `%%` | Insert a `%`, provided the inline asm has arguments |
| `\n` | Insert a line break |
| `\t` | Insert a TAB |
| `\"` | Insert a `"` |
| `\\` | Insert a `\` |
| `$` | Logical line separator, like in `"LDI %A0,1 $ LDI %B0,2"` |
| `__zero_reg__` | The register containing zero, see section Register Layout |
| `__tmp_reg__` | The scratch register, see section Register Layout |

Moreover, the following I/O addresses are defined provided the device supports the respective SFR: `__SREG__`, `__SP_L__`, `__SP_H__`, `__CCP__`, `__RAMPX__`, `__RAMPY__`, `__RAMPZ__`, `__RAMPD__`.

## Assembly Operand Modifiers

| Modifier | Explanation | Purpose |
|---|---|---|
| `lo8()` | 1st Byte of a link-time constant, bits 0...7 | Getting parts of a byte-address |
| `hi8()` | 2nd Byte of a link-time constant, bits 8...15 | |
| `hlo8()` | 3rd Byte of a link-time constant, bits 16...23 | |
| `hhi8()` | 4th Byte of a link-time constant, bits 24...31 | |
| `hh8()` | Same like `hlo8` | |
| `pm_lo8()` | 1st Byte of a link-time constant divided by 2, bits 1...8 | Getting parts of a word-address |
| `pm_hi8()` | 2nd Byte of a link-time constant divided by 2, bits 9...16 | |
| `pm_hh8()` | 3rd Byte of a link-time constant divided by 2, bits 17...24 | |
| `pm()` | Link-time constant divided by 2 in order to get a **p**rogram **m**emory (word) address, like in `lo8(pm(main))`. | word-address |

| | | |
|---|---|---|
| `gs()` | Function address divided by 2 in order to get a (word) addresses, like in `lo8(gs(main))`. Generate stub (trampoline) as needed. This is needed when computing the address of a function on devices with more than 128KiB of program memory that's supposed to be used in `EICALL`. For rationale, see GCC documentation. On devices with less program memory, `gs()` behaves like `pm()`. | function address for `[E]ICALL` |

When the argument of a modifier is not computable at assembler-time, the assembler has to encode the expression in an abstract form using relocs. Consequence is that only a very limited number of argument expressions is supported when they are not computable at assembler-time.

# Using avr-gcc

## Locating .rodata in Flash for AVR64* and AVR128* Devices

- See Application Note avr-gcc: Locate .rodata in Flash for AVR Devices like AVR64 and AVR128

## Supporting "unsupported" Devices

### avr-gcc v5 and newer

In contrast to older versions of the compiler that support `-mmcu=<mcu>` natively, avr-gcc v5+ comes with a bunch of spec files in `./lib/gcc/avr/<version>/device-specs`. These files are generated when the compiler is built and are part of each distribution since then. Spec files specify substitution and transformation rules for command line options for the compiler proper and for subprograms like assembler and linker.

Adding support for a new device consists in writing a new spec file for that device and supply it by means of

```
avr-gcc -mmcu=<mcu> -B <path-to-dir> ...
```

where `<path-to-dir>` is a directory containing a folder named `device-specs` which contains a file named `specs-<mcu>`. As a blue print, start with an already existing spec file for a device as closely related to `<mcu>` as possible. Also read the comments in that spec file.

Just like with older versions, you have to get the device headers which are realm of AVR-LibC from somewhere; same applies for the startup code in `crt<mcu>.o` and for the device library `lib<mcu>.a`. If you do not need or have a device library, `-nodevicelib` will do, but note that some non-standard functionality like EEPROM support is missing then.

Spec files allow to add support for new devices without the need to change the binares of the compiler, the assembler or the linker. **Spec files may depend on the versions of GCC and Binutils, and using an incompatible spec file may lead to errors or wrong or sub-optimal code.** For example, this is the case when newer tool versions support more or different options, but a spec file doesn't reflect that.

As the tools evolve, new features and command line options are added. When porting a device-specs file across one of the following features and versions, extra care must be taken:

- `-mmcu=avrxmega3` (GCC v8, PR81072), `-mavrxmega3` (Binutils v2.29, PR21472)
- `-mgas-isr-prologues` (GCC v8, PR81268), `-mgcc-isr` and pseudo-instruction `__gcc_isr` (Binutils v2.29, PR21683)
- `-mrodata-in-ram` and `-mflmap` (GCC v14, PR112944), `-mavrxmega2_flmap` and `-mavrxmega4_flmap` (Binutils v2.42, PR31124). This does only make a difference for AVR64* and AVR128* devices, see the GCC v14 Release Notes for details.

Notice that the compiler behaves differently depending on the Binutils features it finds during configuration.

### Using .atpack Device Pack Files from Atmel / Microchip

To make your life easier, Atmel / Microchip provides device-pack files at http://packs.download.atmel.com and https://packs.download.microchip.com. The files have extension `.atpack` but apart from that, they are just ZIP files, so you can unzip them and use them. These files contain all you need: Device header and device lib, startup-code, specs-file. Suppose you unzipped the pack to a folder `<atpack>`, then amongst others, following folders and files are present:

```
<atpack>
|--include
|   +--avr
|      +--io*.h
+--gcc
   +--dev
      +--<mcu>
         |--device-specs
         |   +--specs-<mcu>
         +--<mdir>
            |--lib<mcu>.a
            +--crt<mcu>.o
```

Where `<mcu>` comes from `-mmcu=<mcu>`, and `<mdir>` is the multilib-path as printed by `avr-gcc -mmcu=<mcu> -print-multi-directory`. This means we can support a device like, say, ATtiny424 by means of:

```
> avr-gcc -mmcu=attiny424 -B <atpack>/gcc/dev/attiny424 -isystem
<atpack>/include ...
```

### Known Issues

- With **Binutils 2.39 and older**: `crt<mcu>.o` from atpack defines symbol `__DATA_REGION_LENGTH__` to a value much narrower than the one from the default linker script. This may lead to linker errors because `.data` or `.bss` sections no more fit in the data memory segment. One way out is to (re)define this symbol to a larger value like `-Wl,--defsym,__DATA_REGION_LENGTH__=0xffa0`. You can place this in the command line for the link, or add it without `-Wl,` to the `*link_data_start` spec in `specs-<mcu>`. A different fix is to use a linker script file similar to the ones distributed with Binutils 2.40 and newer, that uses symbol `__DATA_REGION_ORGIN__` from `crt<mcu>.o` instead of hard-coded values like 0x802000. See also AVR-LibC Issue #971.
- With **GCC v7 and older**: Some devices belong to device family `avrxmega3` which is only supported since avr-gcc v8+. If you want to use such a device with avr-gcc v7 or lower, you'll have to rewrite the files so that they use core arch `avrxmega2` instead.

## avr-gcc v4.9 and below

avr-gcc and avr-as support the `-mmcu=<mcu>` command line option to generate code for a specific device `<mcu>`. Currently (2012), there are more than 200 known AVR devices and the hardware vendor keeps releasing new devices. If you need support for such a device and don't want to rebuild the tools, you can

1. Sit and wait until support for your `-mmcu=<mcu>` is added to the tools.
2. Use appropriate command line options to compile for your favourite `<mcu>`.

Approach 1 is comfortable but slow. Lazy developers that don't care for time-to-market will use it.

Approach 2 is preferred if you want to start development as soon as possible and don't want to wait until the tool chain with respective device support is released. **This approach is only possible if the compiler and Binutils already come with support for the core architecture of your device.**

When you feed code into the compiler and compile for a specific device, the compiler will only care for the respective core; it won't care for the exact device. It does not matter to the compiler how many I/O pins the device has, at what voltage it operates, how much RAM is present, how many timers or UARTs are on the silicon or in what package it is shipped. The only thing the compiler does with `-mmcu=<mcu>` is to build-in define a specific macro and to call the linker in a specific way, i.e. the compiler driver behaves a bit differently, but the sub-tools like compiler proper and assembler will generate exactly the same code.

Thus, you can support your device by setting these options by hand.

Additionally, we need the following to compile a C program:

- A device support header `avr/io.h` similar to the headers provided by AVR-LibC.
- Startup code for the device.

### The Device Header avr/io.h

This header and its subheaders contain almost all information about a particular device like SFR addresses, size of the interrupt table and interrupt names, etc.

After all, it's just text and you can write it yourself. Find a device that is already supported by AVR-LibC and that is similar enough to your new device to serve as a reasonable starting point for the new device description.

If you are lucky, the device is already supported by AVR-LibC but not yet by the compiler. In that case, you can use verbatim copies from AVR-LibC.

Yet another approach is to write the file from scratch or not to use `avr/io.h` like headers at all. I that case, you provide all needed definitions like, say, SP and size of the vector table yourself.

If your toolchain is distributed with AVR-LibC then `avr/io.h` is located in the installation directory at `./avr/include` i.e. you find a file `io.h` in `./avr/include/avr`. In that file you find the lines:

```
#if defined (__AVR_AT94K__)
#  include <avr/ioat94k.h>
#elif defined (__AVR_AT43USB320__)
#  include <avr/io43u32x.h>

/* many many more entries */

#else
#  if !defined(__COMPILING_AVR_LIBC__)
#    warning "device type not defined"
#  endif
#endif
```

Add an entry for `__AVR_mydevice__` and include your new file `avr/iomydevice.h`.

If you don't want to change the existing `avr/io.h` then copy it to a new directory and add that directory as system search path by means of `-isystem` whenever you compile or preprocess a C or assembler source that shall include the extended `avr/io.h`. Notice that the new directory will contain a subdirectory named `avr`.

## Compiling the Code

Let's start with a simple C program, `source.c`:

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

int var;

int main (void)
{
    return var + SP;
}
```

Your source directory then contains the following files:

```
source.c    gcrt1.S    macros.inc    sectionname.h
```

The startup code gcrt1.S and macros.inc are verbatim copies from AVR-LibC.

`sectionname.h` is included by `macros.inc` but we don't need it: Simply provide `sectionname.h` as an empty file.

For the matter of simplicity, we show how to compile for a device that is similar to ATmega8 so that we don't need to extend `avr/io.h` to show the work flow. In the case you copied `avr/io.h` to a new place, don't forget to add respective `-isystem` to the first two commands for `source.c` and `gcrt1.S`.

ATmega8 is a device in core family `avr4`, thus we compile and assemble our `source.c` for that core architecture. __`AVR_ATmega8`__ stands for the subheader selector you added to `avr/io.h`.

```
avr-gcc -mmcu=avr4 -D__AVR_ATmega8__ -c source.c -Os
```

Similarly, we assemble the startup code for our device by means of:

```
avr-gcc -mmcu=avr4 -D__AVR_ATmega8__ -c gcrt1.S -o crt0-
mydevice.o
```

Finally, we link the stuff together to get a working `source.elf` (assuming that RAM starts at address 0x124):

```
avr-gcc -mmcu=avr4 -Tdata 0x800124 source.o crt0-
mydevice.o -nostartfiles -o source.elf
```

Voilà!

# Libf7

Libf7 is an ad-hoc, AVR-specific, 64-bit floating point emulation written in GNU-C and (inline) assembly. It is hosted and deployed as part of libgcc. Hence, it will be part of any avr-gcc distribution from v10 onwards without any further ado.

## Implementation

- The emulated 64-bit floating point representation is IEEE compatible: Little endian, 11 bit for the encoded exponent, 52 bits for the encoded mantissa.
- The transcendental functions are implemented using MiniMax approximations, i.e. they minimize the maximum norm. Most of these functions use rational MiniMax approximations because they perform better than CORDIC (and they perform better than Taylor or Padé expansions, of course).
- Portability to other architectures or to other compilers was of no consideration; the implementation focuses solely on avr-gcc.

## Known Problems

- PR99184: Wrong double to 16-bit and 32-bit integer conversion. A work-around is to do an intermediate cast to an 64-bit integer type:

```
 double x = 2.9;
 int x_int = (int) (int64_t) x; // For [u]int64_t and uint32_t, do
#include <stdint.h>
 uint32_t x_u32 = (uint32_t) (uint64_t) x;
```

  The issue is fixed in v10.5+, v11.4+ and v12.3+.

The following long standing patches to avr-libc are needed:

- AVR-LibC #57071: Fix math.h and function names and symbols that block 64-bit double.
- AVR-LibC #49567: Use meta-info from --print-multi-lib and --print-multi-directory.

Without these additions to AVR-LibC, 64-bit double cannot work correctly and you will get non-working programs. The AVR-LibC patches were integrated February 2022 and should be available in AVR-LibC v2.2 or newer. Or you can build / use AVR-LibC from git master.

- AVR-LibC #929: A problem with the prototypes of `frexp`, `frexpf`, `frexpl` can lead to wrong code. This bug can be fixed locally by removing `const` attribute from the prototypes of these functions in `math.h`.

## Using 64-bit long double without proper AVR-LibC

Even without the mentioned AVR-LibC patches, you can use 64-bit long double arithmetic if:

- Prototypes for long double functions are provided like in the following example:

```
long double sinl (long double);

long double example (long double x)
{
    return sinl (x + 1.23L);
}
```

  Notice that you don't need prototypes for basic arithmetic like comparisons, addition, etc.

- For C++, you need `extern "C"` prototypes.
- The compiler is used in the default configuration, i.e. Libf7 has not been switched off, and long double layout is 64 bits wide. If you do not know how the compiler has been configured, you can use the following tests to check whether everything is all right:

```
#if __WITH_LIBF7_MATH_SYMBOLS__ != 1
#error Using 64-bit double requires avr-gcc v10+ and --with-libf7=math-symbols.
#endif
```

```
#if __SIZEOF_LONG_DOUBLE__ != 8
#error Only 64-bit long double is supported without the AVR-LibC
patches.
#endif

#if __WITH_DOUBLE_COMPARISON__ != 2
#error Wrong configuration of long double comparison.
#endif
```

- Without the required patches, AVR-LibC has no idea about the true size of `double` and `long double`. This means that when you pass such a variable to a varargs funtion like `printf` or `scanf`, you program might crash and do evil things because these function will pop the wrong number of arguments from the stack. Note: With the patches installed, `printf` will just print "`?`" for 64-bit float, reading the correct number of bytes from the stack so nothing is corrupted.

## Shortcomings

Libf7 is incomplete:

- Some functions are only avalable with newer versions of the compiler:
  - v13.3: Multiplication and `sqrt` for devices without MUL instruction, `atan2`, `fma`.
- Some functions from math.h like `lround`, `lrint`, Bessel and Gamma are not implemented. If you try to use them, you will get undefined references from the linker. If you really need it, you can provide such functions in your projects, or better still contribute them to GCC.

## Other Implementations

- fp64lib from *Uwe Bissinger*: Written in GNU assembly. Slightly less precise. Smaller stack footprint. Smaller code size and execution times. No build script / Makefile as it targets the Arduino ecosystem. fp64lib is not reentrant and cannot be used in multi-threaded programs.
- avr_f64.c from *Detlef* with improvements from *Florian Königstein*: Implemented in C. Resource consumption might be a multiple of what Libf7 consumes. Easy to integrate in own projects that use avr-gcc without native 64-bit double support. Precision is quite good except for some corner(?) cases where it might deteriorate. Could be compiled after fixing minor problems (missing `const` at `progmem`). Should also work with other compilers / targets.
- dannis64bit.S from *Peter Danegger*. Written in GNU assembly.