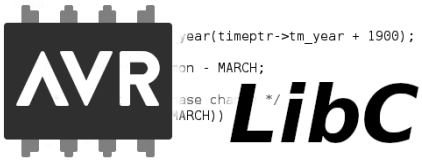

AVR-LibC

2.3.0

Generated by Doxygen 1.9.6

1 AVR-LibC

1.1 Introduction

The latest version of this document is always available from https://avrdudes.github.io/avr-libc/

This documentation is distributed under the same licensing conditions as the entire library itself, see License below.

The AVR-LibC package provides a subset of the standard C library for Microchip (formerly Atmel) AVR 8-bit RISC microcontrollers. In addition, the library provides the basic startup code needed by most applications.

There is a wealth of information in this document which goes beyond simply describing the interfaces and routines provided by the library. We hope that this document provides enough information to get a new AVR developer up to speed quickly using the freely available development tools: Binutils, GCC, AVR-LibC and many others.

If you think you've found a bug, or have a suggestion for an improvement, either in this documentation or in the library itself, please use the bug tracker to ensure the issue won't be forgotten. Before reporting a new issue, you might want to try reading the Frequently Asked Questions chapter of this document.

1.2 General Information about this Library

In general, it has been the goal to stick as best as possible to established standards while implementing this library. Commonly, this refers to the C library as described by the ANSI X3.159-1989 and ISO/IEC 9899:1990 ("ANSI-C") standard, as well as parts of their successor ISO/IEC 9899:1999 ("C99"). Some additions have been inspired by other standards like IEEE Std 1003.1-1988 ("POSIX.1"), while other extensions are purely AVR-specific (like the entire program-space string interface).

Unless otherwise noted, functions of this library are *not* guaranteed to be reentrant. In particular, any functions that store local state are known to be non-reentrant, as well as functions that manipulate I/O registers like the EEPROM access routines. If these functions are used within both standard and interrupt contexts, then undefined behaviour will result. See the FAQ for a more detailed discussion.

1.3 Supported Compilers

The current AVR-LibC can be used with the most recent AVR GCC back to GCC 4.8.

Always use AVR-LibC with the GCC and Binutils version for which the library has been configured and built.

1.4 Supported Devices

The following is a list of AVR devices currently supported by the library. Note that the actual support for a particular device depends on whether it is supported by the compiler at library compile-time.

megaAVR Devices:

ATmega48, ATmega48A, ATmega48PA, ATmega48PB, ATmega48P, ATmega8, ATmega8A, ATmega88, ATmega88A, ATmega88P, ATmega88PA, ATmega88PB, ATmega8515, ATmega8535 ATmega16, ATmega161, ATmega162, ATmega163, ATmega164A, ATmega164P, ATmega164PA, ATmega165, ATmega165A, ATmega165P, ATmega165PA, ATmega168, ATmega168A, ATmega168P, ATmega168PA, ATmega168PB, ATmega16A, ATmega32, ATmega32A, ATmega323, ATmega324A, ATmega324P, ATmega324PA, ATmega324PB, ATmega325, ATmega325A, ATmega325P, ATmega325PA, ATmega3250, ATmega3250A, ATmega3250P, ATmega3250PA, ATmega328, ATmega328P, ATmega328PB, ATmega64, ATmega64A, ATmega640, ATmega644, ATmega644A, ATmega644P, ATmega644PA, ATmega645, ATmega645A, ATmega645P, ATmega6450, ATmega6450A, ATmega6450P, ATmega103, ATmega128, ATmega128A, ATmega1280, ATmega1281, ATmega1284, ATmega1284P, ATmega2560, ATmega2561

megaAVR 0-Series Devices:

ATmega808, ATmega809, ATmega1608, ATmega1609, ATmega3208, ATmega3209, ATmega4808, ATmega4809

tinyAVR Devices:

ATtiny11 [1], ATtiny12 [1], ATtiny13, ATtiny13A, ATtiny15 [1], ATtiny22, ATtiny2313, ATtiny2313A, ATtiny24, ATtiny24A, ATtiny25, ATtiny26, ATtiny261, ATtiny261A, ATtiny28 [1], ATtiny4313, ATtiny43U, ATtiny44, ATtiny44A, ATtiny441, ATtiny45, ATtiny461, ATtiny461A, ATtiny48, ATtiny828, ATtiny84, ATtiny84A, ATtiny841, ATtiny85, ATtiny861, ATtiny861A, ATtiny88, ATtiny1634

tinyAVR 0-Series Devices:

ATtiny202, ATtiny204, ATtiny402, ATtiny404, ATtiny406, ATtiny804, ATtiny806, ATtiny807, ATtiny1604, ATtiny1606, ATtiny1607

tinyAVR 1-Series Devices:

ATtiny212, ATtiny214, ATtiny412, ATtiny414, ATtiny416, ATtiny417, ATtiny814, ATtiny816, ATtiny817, ATtiny1614, ATtiny1616, ATtiny1617, ATtiny3214, ATtiny3216, ATtiny3217

tinyAVR 2-Series Devices:

ATtiny424, ATtiny426, ATtiny427, ATtiny824, ATtiny826, ATtiny827, ATtiny1624, ATtiny1626, ATtiny1627, ATtiny3224, ATtiny3226, ATtiny3227

Reduced tinyAVR Devices with only 16 GPRs:

ATtiny4, ATtiny5, ATtiny9, ATtiny10, ATtiny102, ATtiny104, ATtiny20, ATtiny40

Automotive AVR Devices:

ATtiny416auto, ATtiny87, ATtiny167, ATA5505, ATA5272, ATA5700M322, ATA5702M322, ATA5782, ATA5787, ATA5790, ATA5790N, ATA5791, ATA5831, ATA5835, ATA5795, ATA6285, ATA6286, ATA6289, ATA6612C, ATA6613C, ATA6614Q, ATA6616C, ATA6617C, ATA664251, ATA8210, ATA8510

Automotive CAN AVR Devices:

ATmega16M1, ATmega32C1, ATmega32M1, ATmega64C1, ATmega64M1

CAN AVR Devices:

AT90CAN32, AT90CAN64, AT90CAN128

LCD AVR Devices:

ATmega169, ATmega169A, ATmega169P, ATmega169PA, ATmega329, ATmega329A, ATmega329P, ATmega329PA, ATmega3290, ATmega3290A, ATmega3290P, ATmega3290PA, ATmega649, ATmega649A, ATmega6490, ATmega6490A, ATmega6490P, ATmega649P

Lighting AVR Devices:

AT90PWM1, AT90PWM2, AT90PWM2B, AT90PWM216, AT90PWM3, AT90PWM3B, AT90PWM316, AT90PWM161, AT90PWM81

Smart Battery AVR Devices:

ATmega8HVA, ATmega16HVA, ATmega16HVA2, ATmega16HVB, ATmega16HVBREVB, ATmega32HVB, ATmega32HVBREVB, ATmega64HVE, ATmega64HVE2, ATmega406

USB AVR Devices:

ATmega8U2, ATmega16U2, ATmega16U4, ATmega32U2, ATmega32U4, ATmega32U6, AT43USB320, AT43USB355, AT76C711 [3], AT90SCR100, AT90USB82, AT90USB162, AT90USB646, AT90USB647, AT90USB1286, AT90USB1287

XMEGA Devices:

ATxmega8E5, ATxmega16A4, ATxmega16D4, ATxmega16E5, ATxmega32A4, ATxmega32D3, ATxmega32D4, ATxmega32E5, ATxmega64A1, ATxmega64A3, ATxmega64D3, ATxmega64D4, ATxmega128A1, ATxmega128A3, ATxmega128D3, ATxmega128D4, ATxmega192A3, ATxmega192D3, ATxmega256A3, ATxmega256A3B, ATxmega256D3

USB XMEGA Devices:

ATxmega16A4U, ATxmega16C4, ATxmega32A4U, ATxmega32C3, ATxmega32C4, ATxmega64A1U, ATxmega64A3U, ATxmega64A4U, ATxmega64B1, ATxmega64B3, ATxmega64C3, ATxmega128A1U, ATxmega128A3U, ATxmega128A4U, ATxmega128B1, ATxmega128B3, ATxmega128C3, ATxmega192A3U, ATxmega192C3, ATxmega256A3U, ATxmega256A3BU, ATxmega256C3, ATxmega384C3, ATxmega384D3

AVR-Dx Devices:

AVR16DD14, AVR16DD20, AVR16DD28, AVR16DD32, AVR32DA28, AVR32DA28S, AVR32DA32, AVR32DA32S, AVR32DA48, AVR32DA48S, AVR32DB28, AVR32DB32, AVR32DB48, AVR32DD14, AVR32DD20, AVR32DD28, AVR32DD32, AVR64DA28, AVR64DA28S, AVR64DA32, AVR64DA32S, AVR64DA48, AVR64DA48S, AVR64DA64, AVR64DA64S, AVR64DB28, AVR64DB32, AVR64DB48, AVR64DB64, AVR64DD14, AVR64DD20, AVR64DD28, AVR64DD32, AVR128DA28, AVR128DA28S, AVR128DA32, AVR128DA32S, AVR128DA48, AVR128DA48S, AVR128DA64, AVR128DA64S,

AVR128DB28, AVR128DB32, AVR128DB48, AVR128DB64

USB AVR-Dx Devices:
AVR16DU14, AVR16DU20, AVR16DU28, AVR16DU32, AVR32DU14, AVR32DU20, AVR32DU28, AVR32DU32, AVR64DU28, AVR64DU32

AVR-Ex Devices:
AVR16EA28, AVR16EA32, AVR16EA48, AVR16EB14, AVR16EB20, AVR16EB28, AVR16EB32, AVR32EA28, AVR32EA32, AVR32EA48, AVR32EB14, AVR32EB20, AVR32EB28, AVR32EB32, AVR64EA28, AVR64EA32, AVR64EA48

AVR-Lx Devices:
AVR16LA14, AVR16LA20, AVR16LA28, AVR16LA32, AVR32LA14, AVR32LA20, AVR32LA28, AVR32LA32

AVR-SD Functional Safety Devices:
AVR32SD20, AVR32SD28, AVR32SD32

Wireless AVR Devices:
ATmega64RFR2, ATmega644RFR2, ATmega128RFA1, ATmega128RFR2, ATmega1284RFR2, ATmega256RFR2, ATmega2564RFR2, AT86RF401

Miscellaneous Devices:
AT94K [2], M3000 [4]

Classic AVR Devices:
AT90S1200 [1], AT90S2313, AT90S2323, AT90S2333, AT90S2343, AT90S4414, AT90S4433, AT90S4434, AT90S8515, AT90C8534, AT90S8535

Note [1]
Assembly only. There is no direct support for these devices to be programmed in C since they do not have a RAM based stack. Still, it could be possible to program them in C, see the FAQ for an option.

Note [2]
The AT94K devices are a combination of FPGA and AVR microcontroller. [TRoth-2002/11/12: Not sure of the level of support for these. More information would be welcomed.]

Note [3]
The AT76C711 is a USB to fast serial interface bridge chip using an AVR core.

Note [4]
The M3000 is a motor controller AVR ASIC from Intelligent Motion Systems (IMS) / Schneider Electric.

1.5 AVR-LibC License

AVR-LibC can be freely used and redistributed, provided the following license conditions are met.

2 Frequently Asked Questions

2.1 FAQ Index

2.2 Why doesn't my program recognize a variable updated in an interrupt routine?

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

```
uint8_t flag; // volatile is missing

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 qualified as volatile:

```
volatile uint8_t flag;
```

Back to FAQ Index.

2.3 How to permanently bind a variable to a register?

This can be done with

```
register uint8_t counter __asm("r3");
```

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

Registers r8 through r25 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.

Warning

Extreme care should be taken that the entire application is compiled with a consistent set of register-allocated variables including possibly used library functions. This can be achieved by compiling each module with `-ffixed-r3` or `-ffixed-3`. Notice that when you are using library functions from libgcc (the avr-gcc runtime library) or AVR-LibC, then these libraries were generated *without* the requirement to avoid specific registers. Hence when you are using libraries from the distribution, you must make sure that none of the reserved registers is used in the generated binary.

Also notice that global register variables can't be volatile, because only variables in memory can be volatile, and register variables are not located in memory.

Back to FAQ Index.

2.4 How to modify MCUCR or WDTCR early?

Basically, write a small function which looks like this:

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

static __attribute__((used, unused, naked, section(".init3")))
void init_MCUCR (void);

void init_MCUCR (void)
{
    MCUCR = _BV(SRE) | _BV(SRW);
}
```

Do not call this function by hand! This piece of code will be inserted in startup code, which is run right after reset. For the meaning of the attributes, see How do I perform a software reset of the AVR?

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 In C/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.

2.5 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 -save-temps compiler option) seems to be warranted.

Back to FAQ Index.

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

So you tried this:

```
asm volatile ("sbi 0x18, 7" ::: "memory");
```

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, 7" ::: "memory");
```

you get a syntax error from the assembler: `"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` gets passed to the assembler instead of `0x18`. One way to avoid this problem is:

```
asm volatile ("sbi %0, 7" :: "I" (_SFR_IO_ADDR(PORTB)) : "memory");
```

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.

There are situation though where the address of a special function register (SFR) is required in inline assembly. When the register can be accessed by `LDS` and `STS`, one can use the RAM address of the SFR:

```
asm volatile ("sts %0, __zero_reg__" :: "n" (& PORTB) : "memory");
```

When the I/O address of the register is required, one way is to use `_SFR_IO_ADDR` to get the I/O address like in the example above. A different approach is to use inline asm print modifier `%i` supported since avr-gcc v4.7:

```
asm volatile ("out %i0, __zero_reg__" :: "n" (& PORTB) : "memory");
```

The `%i0` will print the address of `PORTB` as an I/O address.

Back to FAQ Index.

2.7 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. It is guaranteed that the code runs with the exact same optimizations as it would run without the `-g` switch.

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, or at least optimization level `-Og` can be used which was introduced to improve good debugging experience while it still provides a reasonable amount of optimization.

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.

2.8 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
    ...
    clr     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    3f              ; return from myfunc prematurely
    ...
    inc     r16
2:  cmp     r16, r18
    brlo    1b              ; jump back to top of loop
3:  pop     YH
    pop     YL
    pop     r18
    pop     r17
    pop     r16
    ret
```

Back to FAQ Index.

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

Consider this example code:

```
#include <stdint.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;
```

```
}

int main (void)
{
    set_bits_func_wrong (PORTB, 0xaa);
    set_bits_func_correct (&PORTB, 0x55);
    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
 10c:   88 b3           in      r24, 0x18
 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
 114:   88 e3           ldi     r24, 0x38
 116:   90 e0           ldi     r25, 0x00
 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.

In this contrived example, the most efficient method with respect to both execution speed and code size is to inline the function:

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

Back to FAQ Index.

2.10 What registers are used by the compiler?

See also the Type Layout, Register Layout and Calling Convention sections in the avr-gcc Wiki.

Data types

`char` is 8 bits, `int` and `short` are 16 bits, `long` is 32 bits, `long long` is 64 bits, `float` is 32 bits, `double` and `long double` are 32 bits or 64 bits, 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` and `short` 8 bits, `long` 16 bits and `long long` 32 bits. But that is not supported by AVR-LibC (except for `stdint.h` and `avr/pgmspace.h`, but no 64-bit integer types are available) and violates C standards (`int` *must* be at least 16 bits).

Call-used registers (r18-r27, r30-r31)

May be allocated by gcc for local data. You *may* use them freely in assembly subroutines. Calling C subroutines can clobber any of them - the caller is responsible for saving and restoring.
For the Reduced Core architecture (ATtiny10 and relatives), r20-r27 and r30-31 are call-clobbered.

Call-saved registers (r2-r17, r28-r29)

May be allocated by gcc for local data. Calling C subroutines leaves them unchanged. Assembly 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.
For the Reduced Core architecture (ATtiny10 etc.), r18-r19 and r28-r29 are call-saved. Registers r0 through r15 do not exist.

Fixed registers (r0, r1)

Never allocated by gcc for local data, but often used for fixed purposes:

- **r0** (`__tmp_reg__`) — temporary register, can be clobbered by any code (except interrupt handlers which save it), *may* be used to remember something for a while within one piece of assembly code
- **r1** (`__zero_reg__`) — 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 __zero_reg__`). This includes any use of the `[f]mul[s[u]]` instructions, which return their result in r1:r0. Interrupt handlers save and clear `__zero_reg__` on entry, and restore it on exit (in case it was non-zero).
- **T flag** — the `T` flag in the status register (`SREG`) can be used the same way like `__tmp_reg__`.

For the Reduced Core architecture (ATtiny10 etc.), `__tmp_reg__` is r16, and `__zero_reg__` is r17.

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.
On AVRrc, r25 to r20 are used to pass values.

- Return values: 8-bit in r24, 16-bit in r25:r24, up to 32 bits in r22-r25, up to 64 bits in r18-r25.
- Arguments to functions with a variable number of lists like `printf` get all their values on the stack. `char` is extended to `int`, and `float` is extended to `double`.
- When an argument is passed on the stack, all subsequent arguments are also passed on the stack.
- An argument is either passed completely in registers or completely on the stack.
- Arguments with a size of zero or with a size larger than 8 bytes (4 bytes on AVR_TINY) are returned in memory. The caller provides the memory location as implicit first argument to the callee.
- When an argument is returned in registers, its size is padded to the next power of 2.

Back to FAQ Index.

2.11 How are the RAMPX, RAMPY, RAMPZ and RAMPD registers handled?

See `The RAMPX, RAMPY, RAMPZ and RAMPD special function registers` in the GCC documentation.

Back to FAQ Index.

2.12 How is the EIND special function register handled?

See `The EIND special function register` in the GCC documentation.

Back to FAQ Index.

2.13 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.

2.14 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 μs	19.7 ms
-O2	6666	972 μs	20.1 ms
-Os	6618	955 μs	20.1 ms
-Os -mcall-prologues	6474	972 μs	20.1 ms

(The difference between 955 μs and 972 μs 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.

2.15 How do I relocate code to a fixed address?

First, put the function into a new, orphan named section. This is done with a `section` attribute that specifies the name of the input section with the prototype of the function:

```
__attribute__ ((noinline, noclone, section (".bootloader")))
void boot (void);
```

The `noinline` and `noclone` attributes are required to make sure that the function is not (partially) inlined into the caller, which does not have a respective section attribute.

Second, locate the section to the desired fixed address by means of linking with, say

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

see the -Wl compiler option. The name after `--section-start` is the name of the section to be located, and the number specifies the beginning address of the named section.

This will only work when the section is an orphan section, i.e. a section that is not mentioned in the linker script. For sections that *are* mentioned in the linker script, like for example `.text.bootloader`, this will not work because `--section-start` refers to an output section, but the output section for input section `.text.bootloader` is the `.text` section.

To verify that everything went as expected, generate the disassembly with `avr-objdump ... -j .bootloader`. The top of the list file will show

```
  1 .bootloader   00000004  00002000  00002000  00000204  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
```

Or display section properties with `avr-readelf --section-details`

```
$ avr-readelf -t main.elf
Section Headers:
  [Nr] Name
       Type            Addr     Off    Size   ES   Lk Inf Al
       Flags
  [ 2] .bootloader
       PROGBITS        00002000 000204 000004 00   0   0  1
       [00000006]: ALLOC, EXEC
```

A different way to locate the section is by means of a custom linker script. The avr-gcc Wiki has an example that locates the `.progmem2.data` section which is used by the compiler for variables in address-space `__flash2`.

Back to FAQ Index.

2.16 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.

2.17 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 AVR 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.

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

uint16_t read_timer1 (void)
{
    uint8_t sreg = SREG;
    uint16_t val;

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

    return val;
}
```

Back to FAQ Index.

2.18 Shouldn't I initialize all my variables?

Variables in static storage are guaranteed to be initialized by the C standard. This includes global and static variables without explicit initializer, which are initialized to 0. `avr-gcc` does this by placing the appropriate code into section `.init4`. 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 explicitly initialized, or their initializer is all zeros, they go into the .bss output section. 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.)

Back to FAQ Index.

2.19 Why do all my string literals eat up the SRAM?

By default, string literals are handled like all other initialized constants in static storage: They are put into a `.rodata` input section.

On old devices, the `.rodata` input sections are allocated to the .data output section, which means that such objects occupy memory in the flash ROM *and* in the SRAM according to the following rules:

Table 2 Non-Zero Data in Static Storage

Read-Only	Input Section	Output Section	Memory Region(s)	Case
No	.data	.data	text *and* data	All
Yes	.rodata	**.data**	text *and* data	3
		.rodata	text	1
		.rodata	rodata	2

Cases

1. Devices that see all of program memory in the RAM address space: `avrtiny` (AVRrc), `avrxmega3` (AVR16 and AVR32 devices, 0-series, 1-series and 2-series).
2. Devices that see only a part of program memory in the RAM address space: AVR64 and AVR128 devices without `-mrodata-in-ram`.
3. All other devices, i.e. devices that don't see (parts of) the program memory in the RAM address space, or when `-mrodata-in-ram` is specified. The place in the flash ROM (`.text`) is occupied by the initializer values that are used by the startup code to initialize the objects in the `.data` output section.

The question remains: Why is case 3. allocating constant variables into the SRAM? For a better understanding, consider the following code:

```
char get_first_char (const char *str)
{
    return str[0];
}
```

What code should the compiler generate for that function? Notice that it is perfectly fine to pass a string that is not constant through the `const char *str` parameter, which only states that `get_first_char` won't change `str` like in the following use cases:

```
char func1 (int i)
{
```

```
    char str[10];
    sprintf (str, "%d", i);
    return get_first_char (str);
}

char func2 (void)
{
    return get_first_char ("Hello");
}
```

In any case the compiler will generate code like this:

```
get_first_char:
    movw r30, r24
    ld   r24, Z
    ret
```

For the devices from 1. and 2. the "`ld r24,Z`" will read from flash or from SRAM depending on the address passed in R24. But on the old 3. devices only LPM can read from flash, and therefore the tools place "`Hello`" in RAM so that the code works for all possible string addresses. The same applies to all other constants in static storage.

Of course, this is going to waste a lot of SRAM on the old devices.

For ways how to place string literals in flash ROM (and to access them), see also the next section How do I put an array of strings completely in ROM?.

Note

What follows is only needed when the `.rodata` input sections are located in RAM, which is only the case for the "old" AVR devices. See the table above.

2.19.1 Use PROGMEM

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 old AVR devices need 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 <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_char(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.

2.19.2 Use __flash

As an alternative, the GNU-C named address-space `__flash` can be used. Notice the proper address-space qualification of `debug_F`, the usage of FSTR instead of `PSTR`, and that reading from a `__flash` address can be accomplished by open-coded C code like in `*addr++`.

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

void debug_F (const __flash char *addr)
{
  char c;

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

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

Back to FAQ Index.

2.20 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. For the rationale behind storing constant strings in RAM, see the previos section Why do all my string literals eat up the SRAM? and Data in Program Space

2.20.1 With PROGMEM

The most obvious (and incorrect) thing to do is this:

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

// array[] will be in flash ROM, but "Foo" and "Bar" are still in RAM.
const char* const 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 some `.rodata` input section).

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

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

static const char s_foo[] PROGMEM = "Foo";
static const char s_bar[] PROGMEM = "Bar";

const char* const array[2] PROGMEM =
{
    s_foo, s_bar
};

void func (uint8_t i)
{
    char buf[32];

    const char *p = pgm_read_ptr (&array[i]);
    strcpy_P (buf, p);
}
```

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

```
00000026 <array>:
  26:   2e 00 2a 00

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

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

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

2.20.2 With named address-spaces

An alternative is to use the named address-space `__flash`, which is supported since avr-gcc v4.7 and in GNU-C99 (`-std=gnu99`) and up:

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

const __flash char* const __flash array[] =
{
    FLIT("Foo"), FLIT("Bar")
};

int compare (const char *str, uint8_t i)
{
    return strcmp_F (str, array[i]);
}
```

Moreover, there is no more need for `pgm_read_xxx()`: The (addresses of) the string literals can be read directly by means of `array[i]`. Depending on the application, the string array can be defined as

```
const __flash char array[][4] =
{
    "Foo", "Bar"
};
```

Back to FAQ Index.

2.21 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 ... -save-temps`), there's a comment in each generated assembler file that tells you the frame size for each generated function.

GCC also supports `-fstack-usage` that generates an `.su` text file for each compilation unit:

```
int main (void)
{
    return 0;
}

$ avr-gcc -c main.c -Os -fstack-usage
$ cat main.su

main.c:3:5:main 2   static
```

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.

2.22 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.

2.23 What features are (not) supported on AVRrc Reduced Core devices?

On the reduced AVRrc core with only 16 general purpose registers, the following features are not supported:

- The printf and scanf families of functions
- Floating-point arithmetic
- Fixed-point arithmetic (though 8-bit and 16-bit arithmetic should work with GCC v15+)
- Arithmetic involving 64-bit computations
- Named address-spaces like `__flash`

Back to FAQ Index.

2.24 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.

2.25 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.

2.26 How to use a Compact Vector Table?

Some devices support a Compact Vector Table:

- Devices from the 0-series, 1-series, 2-series
- AVR-Dx, AVR-Ex and AVR-SD devices

A CVT consists of only four vectors: 0=Reset, 1=NMI, 2=Prio1 IRQ, 3=Prio0 IRQs. The easiest way is to link and compile with `-mcvt` which is supported since avr-gcc v15. This will link the `crt`*mcu*`-cvt.o` variant of the startup code that provides a CVT and activates it by setting the `CPUINT_CTRLA.CPUINT_CVT` bit. A C code will look something like:

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

ISR (__vector_2)
{
    /* Code for Prio1 IRQ */
}

ISR_N (3)
static void prio0_isr (void)
{
    /* Code for Prio0 IRQs */
}
```

When you are using an older compiler but at least AVR-LibC v2.3, you can link with `-nostartfiles` *path*`/crt`*mcu*`-cvt.o` where:

- *mcu* is the name of the device like `attiny202` for ATtiny202.
- *path* is of the form *prefix*`/avr/lib/`*mlib* where
  - *prefix* is the install path
  - *mlib* is the multilib path that can be inferred with `$(avr-gcc -mmcu=`*mcu*` -print-multi-directory)` like `avrxmega3/short-calls` for `-mmcu=attiny202`.

Building `crt`*mcu*`-cvt.o`
When you are using AVR-LibC older than v2.3, then you can build `crt`*mcu*`-cvt.o` yourself from the current `gcrt1.S` source with `-D USE_COMPACT_VECTOR_TABLE`, and you have to activate the CVT by hand (or you can implement `__init_cvt` as, say, a static constructor or as a naked function in section `.init3`).

Downloading `crt`*mcu*`-cvt.o`
An alternative to building `crt`*mcu*`-cvt.o` yourself is to download a pre-built toolchain like avr-gcc-15.1 from the `WinAVR repo`. Notice that you can use that tarball on any operating system since you are only going to use target-specific files. Unpack the downloaded archive as explained there to some folder *prefix* on your machine.
Then link with `-nostartfiles` *path*`/crt`*mcu*`-cvt.o -L` *path* where *path* is like above. The "-L *path*" is needed so that `lib`*mcu*`.a` from the downloaded toolchain is used. It contains parts of the startup code like `__init_cvt` and `__call_main`.

Back to FAQ Index.

2.27 How to use a local (static, namespace) function as ISR?

In order to use a non-global function as an ISR, there is ISR_N from `avr/interrupt.h`:

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

ISR_N (INT0_vect_num)
static void int0_isr ()
{
    // Code
}

namespace
{
    ISR_N (ADC_vect_num)
    void adc0_isr ()
    {
        // Code
    }
};
```

ISR_N was added in AVR-LibC v2.3. It uses the `signal(N)` function attribute added in `GCC v15`.

Back to FAQ Index.

2.28 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.

2.29 Which AVR-specific assembler operators are available?

See Pseudo-Ops and Operand Modifiers.

Back to FAQ Index.

2.30 How to call an assembly function from C/C++?

Just declare the function like you would in C/C++:

```
extern     uint16_t asm_add (uint16_t a, uint8_t b); // in C
extern "C" uint16_t asm_add (uint16_t a, uint8_t b); // in C++
```

and then call it like any ordinary C/C++ function:

```
uint16_t use_asm_func (uint16_t a)
{
    return asm_add (a, 42);
}
```

An example implementation of an assembly function that returns *a* + *b* would read something like:

```
#ifdef __AVR_TINY__
#define zero_reg r17
#else
#define zero_reg r1
#endif
```

```
.text
/* According to the AVR GCC ABI:
   - Param a is passed in R25:R24.
   - Param b is passed in R22.
   - Return value is in R25:R24.
   - zero_reg always contains a value of zero.  */
.global asm_add
.type asm_add, @function
asm_add:
    add r24, r22
    adc r24, zero_reg
    ret
.size asm_add, . - asm_add
```

Notice that the assembly function complies to the avr-gcc calling convention.

See also

- What registers are used by the compiler?
- How can I use a function that doesn't comply to the ABI?

Back to FAQ Index.

2.31 How to call a C/C++ function from assembly?

The actual call is simple:

```
#ifdef __AVR_HAVE_JMP_CALL__
#define XCALL call
#define XJMP  jmp
#else
#define XCALL rcall
#define XJMP  rjmp
#endif

.text
   ...
   XCALL c_func  /* Ordinary call */
   ...
   XJMP c_func   /* Tail-call */
```

Since `__AVR_HAVE_JMP_CALL__` is a macro built-in defined by the C preprocessor, you have to invoke the assembler by means of the avr-gcc driver program. On the C/C++ side, you have to make sure that the `c_func` function is actually present:

- In C/C++, `c_func` must be declared as `extern`.
- C++ will provide the mangled name like `_Z4funci` for `int c_func(int)`. When you want a plain `c_func` symbol, then define the function as `extern "C"`.
- Make sure that the function is not optimized away: When you are using global optimizations, then function attributes like `externally_visible` and `used` may be needed.

The passed arguments and the return value must comply to the avr-gcc ABI, see also What registers are used by the compiler? In particular C/C++ functions expect that register R1 (on AVRrc: R17) contains the value 0.

When calling a function in libgcc, GCC's runtime support library, then you'll have to check which calling convention is actually in use. These function are implemented in assembly, and some of them don't use the C/C++ calling convention. An example is the 32 = 16 × 16 multiplication routine `__umulhisi3`. For an (incomplete) list, see the avr-gcc Wiki.

Back to FAQ Index.

2.32 How can I use a function that doesn't comply to the ABI?

Suppose you have a function `f_nonabi` written in assembly that doesn't comply to the avr-gcc ABI. How can such a function be used from C/C++?

For the matter of the discussion, suppose that the function `f_nonabi`

- Takes a 16-bit integer value in R31:R30
- Returns a 16-bit integer value in R23:R22
- Clobbers R1 and R2
- Changes memory contents

There are basically two approaches: Write an interface function in assembly, or write an interface in inline assembly.

2.32.1 Interface in Assembly

The iterface function would look something like this, assuming that the AVR device supports the `MOVW` and `CALL` instructions:

```
.global call_f_nonabi
.type call_f_nonabi,"function"
call_f_nonabi:
    push r2          ; ABI: R2 is callee-saved
    movw r30, r24    ; ABI: int is passed in R25:R24
    call f_nonabi
    movw r24, r22    ; ABI: Return result in R25:R24
    clr r1           ; ABI: R1 (__zero_reg__) must hold 0.
    pop r2           ; ABI: R2 is callee-saved
    ret
```

Then provide C/C++ with an interface to `call_f_nonabi`:

```
extern     int call_f_nonabi (int);  // C
extern "C" int call_f_nonabi (int);  // C++
```

2.32.2 Interface in Inline Assembly

GCC supports `local register variables` as a GNU extension:

```
static inline __attribute__((__always_inline__))
int call_f_nonabi (int arg)
{
    register int r22 __asm("r22");
    __asm volatile ("%~call f_nonabi"   "\n\t"
                    "clr __zero_reg__"
                    : "=r" (r22) : "z" (arg) : "r2", "memory");
    return r22;
}
```

Notes:

- Whether or not `volatile` and the `"memory"` clobber are needed depend on what `f_nonabi` actually does.
- There's no need for a local register variable for `num` since the `"z"` constraint is R30 and R31.

- R1 is a fixed register, and therefore clobbering it has no effect. The value of 0 in R1 has to be restored by hand.
- For the meaning of "%~call" see inline asm special sequences.
- GCC v16 introduces hard-register constraints as an alternative to local register variables. Using that feature, the function would look like this:

```
static inline __attribute__((__always_inline__))
int call_f_nonabi (int arg)
{
    int ret;
    __asm volatile ("%~call f_nonabi"   "\n\t"
                    "clr __zero_reg__"
                    : "={r22}" (ret) : "z" (arg) : "r2", "memory");
    return ret;
}
```

Back to FAQ Index.

2.33 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.

2.34 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.

2.35 How to add a raw binary image to linker output?

2.35.1 Using #embed

Before answering this question, lets mention the `#embed` directive introduced in C23. While `#embed` doesn't strictly conform to the requirement of the question, it is so much more convenient that you still may want to use it. `#embed` embeds the contents of a file into a compilation unit. Here is an example:

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

static const char abba_txt[] PROGMEM =
{
#embed "abba.txt"
    // Append \0 so that abba_txt becomes a proper C string
    // (assuming abba.txt is a text file).
    , '\0'
};

int main (void)
{
    printf ("== abba.txt has %d bytes ==\n%S\n",
            (int) sizeof (abba_txt) - 1, abba_txt);
}
```

2.35.2 Using .incbin

The GNU assembler supports the `.incbin` directive, which includes the content of a file verbatim as if it was specified by `.byte` directives:

```
.macro ARRAY_START name
    .pushsection .progmem.data.\name,"a","progbits"
    .global \name
    .type \name,"object"
    \name:
.endm

.macro ARRAY_END name
    .global \name\()_size
    \name\()_size = . - \name
    .size \name, . - \name
    .popsection
.endm

/* Defines abba_txt[] from the bytes of abba.txt. */
ARRAY_START abba_txt
    .incbin "abba.txt"
    /* Append a \0 to the array. */
    .byte 0
ARRAY_END abba_txt
```

The code defines the assembly macros `ARRAY_START` and `ARRAY_END` for convenience. They define the array and a symbol for the array size. A C header file provides the symbols to a C program:

```
#ifndef ABBA_H
#define ABBA_H
extern char abba_txt[];
// Notice that abba_txt_size is just a symbol
// that doesn't occupy memory.
extern char abba_txt_size[];
#endif /* ABBA_H */
```

This header is then included in order to use the data:

```
#include <stdio.h>
#include <stdint.h>
```

```
#include "abba.h"

int main (void)
{
    printf ("start: 0x%x\n", (uintptr_t) abba_txt);
    printf ("size:  %u\n", (uintptr_t) abba_txt_size);
    printf ("array: \"%S\"\n", abba_txt);
    return 0;
}
```

The simplest way to assemble the assembly module is by letting avr-gcc do the work:

```
$ avr-gcc -x assembler-with-cpp abba.asm -x none -o abba.o -mmcu=...
```

The "`-x assembler-with-cpp`" and "`-x none`" are only needed when the file extension of the assembly file is neither `.S` nor `.sx`. In such cases, avr-gcc must be provided with the input language.

2.35.3 Using objcopy

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 fo
```

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.

2.36 How do I perform a software reset of the AVR?

The canonical way to perform a software reset of non-XMega AVR's 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.

XMega parts have a specific bit `RST_SWRST_bm` in the `RST.CTRL` register, that generates a hardware reset. RST_SWRST_bm is protected by the XMega Configuration Change Protection system.

The reason why using the watchdog timer or `RST_SWRST_bm` is preferable over jumping to the reset vector, is that when the watchdog or `RST_SWRST_bm` 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>

static inline __attribute__((__always_inline__))
void soft_reset (void)
{
    wdt_enable (WDTO_15MS);
    for(;;) {}
}
```

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
static  __attribute__((used, unused, naked, section(".init3")))
void wdt_init (void);

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

```
}
```

The code is placed in section `.init3` so that it is executed as part of the normal startup procedure. The `naked` attribute is required so that the code does not `return` (Code in init sections is executed as it is located; the code is not called, and code from one init section falls through to the code in the next one). The `used` attribute makes sure that the compiler does not throw the seemingly unused function away. The `unused` attributes avoids warnings about unused code.

Back to FAQ Index.

2.37 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.
`strtof`, `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 scanf or scanf_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.

Library Call	Reentrant Issue	Workaround / Alternative
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, 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.
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, 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.
time.h	Uses global state in __latitude, __longitude, __utc_offset, __asc_store, __tm_store, __dst_ptr, __system_time.	Some functions have reentrant versions, like: asctime_r, ctime_r, gmtime_r, iso_week_date_r, isotime_r, localtime_r.

Note

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.

2.38 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.

The failure mechanism for an overwritten byte 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.

2.39 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.

2.40 On a device with more than 128 KiB of flash, how to make function pointers work?

Function pointers beyond the "magical" 128 KiB barrier(s) on larger devices are supposed to be resolved through so-called *trampolines* by the linker, so the actual pointers used in the code can remain 16 bits wide.

In order for this to work, the option `-mrelax` must be given on the compiler command-line that is used to link the final ELF file.

See also the avr-gcc online documentation on the `EIND special function register` and indirect calls.

Back to FAQ Index.

2.41 Why is assigning ports in a "chain" a bad idea?

Suppose a number of IO port registers should get the value `0xff` assigned. Conveniently, it is implemented like this:

```
DDRB = DDRD = 0xff;
```

According to the rules of the C language, this causes 0xff to be assigned to `DDRD`, then `DDRD` is read back, and the value is assigned to `DDRB`. The compiler stands no chance to optimize the readback away, as an IO port register is declared "volatile". Thus, chaining that kind of IO port assignments would better be avoided, using explicit assignments instead:

```
DDRB = 0xff;
DDRD = 0xff;
```

Even worse ist this, e. g. on an ATmega1281:

```
DDRA = DDRB = DDRC = DDRD = DDRE = DDRF = DDRG = 0xff;
```

The same happens as outlined above. However, when reading back register `DDRG`, this register only implements 6 out of the 8 bits, so the two topmost (unimplemented) bits read back as 0! Consequently, all remaining `DDR`*x* registers get assigned the value 0x3f, which does not match the intention of the developer in any way.

2.42 Which header files are included in my program?

Suppose we have a simple program like

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

int main (void)
{
    return 0;
}
```

and we want to know which files this `#include` triggers. Just add option `-H` to the compiler options and check what is printed on standard output:

```
$ avr-gcc -H -S main.c -mmcu=atmega8
. <install>/avr/include/avr/pgmspace.h
.. <install>/avr/include/inttypes.h
... <install>/lib/gcc/avr/<version>/include/stdint.h
.... <install>/avr/include/stdint.h
.. <install>/lib/gcc/avr/<version>/include/stddef.h
.. <install>/avr/include/avr/io.h
... <install>/avr/include/avr/sfr_defs.h
... <install>/avr/include/avr/iom8.h
... <install>/avr/include/avr/portpins.h
...
```

where <install> denotes the installation path, <version> denotes the GCC version, and the number of dots indicates the include level, e.g. inttypes.h is included by pgmspace.h.

When -v is added to the compiler options, then the search paths are also displayed (amongst other stuff):

```
#include "..." search starts here:
#include <...> search starts here:
 <install>/bin/../lib/gcc/avr/<version>/include
 <install>/bin/../lib/gcc/avr/<version>/include-fixed
 <install>/bin/../lib/gcc/avr/<version>/../../../../avr/include
End of search list.
```

After resolving the ..'s for *"parent directory"*, the last directory becomes <install>/avr/include.

Back to FAQ Index.

2.43 Which macros are defined in my program? Where are they defined, and to what value?

For a documentation of compiler built-in macros see AVR built-in macros in the GCC online documentation.

One way to find all the defined macros is to add -save-temps and -g3 to the compiler options. This saves the temporary files like the pre-processed source code in an .i file (for C sources), an .ii (for C++), or a .s (for assembly). A debug level of DWARF3 or higher is required to include the macro definitions in the file. With lower debug levels, only the preprocessed source will be present.

For a module with a simple #include <avr/pgmspace.h>, the saved intermediate file might look something like:

```
# 0 "<built-in>"
#define __STDC__ 1
```

⇒ The __STDC__ macro is defined built-in in the compiler.

```
# 0 "<command-line>"
#define __AVR_DEVICE_NAME__ atmega8
```

⇒ The __AVR_DEVICE_NAME__ macro is defined on the command line by means of -D __AVR_DEVICE↩
_NAME__=atmega8. In this special case, the -D option is added by the specs file specs-atmega8.

```
# 81 "<install>/avr/include/avr/pgmspace.h" 3
#define __PGMSPACE_H_ 1

#define __need_size_t
```

⇒ The __PGMSPACE_H_ macro is defined in line 81 of that header file. When there is no line note directly above the definition, go up until you find a line note and add the respective number of lines. For example, the __need_size_t macro is defined in line 84 of that file.

Back to FAQ Index.

2.44 What ISR names are available for my device?

The HTML documentation comes with exhaustive Vector → AVRs and AVR → Vectors tables. In order to get a vector number for the `ISR_N` macro, just append `_num` to the vector name as used with `ISR`. For example use `ISR_N(ADC0_vect_num)` instead of ISR(ADC0_vect).

One way to find the available vector names is to pre-process a small file, and to grep for possible names, like in:

```
$ echo "#include <avr/io.h>" | avr-gcc -xc - -mmcu=atmega8 -E -dM  | grep _VECTOR
#define INT0_vect _VECTOR(1)
#define INT1_vect _VECTOR(2)
#define TIMER2_COMP_vect _VECTOR(3)
#define TIMER2_OVF_vect _VECTOR(4)
#define TIMER1_CAPT_vect _VECTOR(5)
...
```

Explanation:

`echo "#include <avr/io.h>"`
: This prints `#include <avr/io.h>` to the standard output, which is picked up by the following command as a C program to be preprocessed.

`avr-gcc -xc - -mmcu=atmega8 -E -dM`
: Set the input language to C, read the program from standard input (specified by a dash), preprocess, and print all macro definitions to the standard output.

`grep _VECTOR`
: Only print lines with `_VECTOR` in them.
The output above was actually generated with an additional `| sort -t '(' -k 2n` so that the vectors are printed in order.
In order to find the respective vector numbers, use `grep _vect_num` instead.

Back to FAQ Index.

2.45 What are the versions of the tools in my toolchain?

The versions of the tools can be obtained by running the following commands and taking the first line of the standard output:

Table 4 Commands to obtain Tool Versions

Version of	Command
Compiler	`avr-gcc --version`
Assembler	`avr-as --version`
LD Linker[1]	`avr-ld --version`
CC Linker[2]	`echo "" | avr-gcc -xc - -Wl,--version 2> /dev/null`
AVR-LibC	`echo "#include <avr/version.h>" | avr-gcc -xc - -E -dM | grep AVR_LIBC_VERSION_STRING`

Linker[1] is the linker as called directly (`avr-ld`) whereas Linker[2] is the linker as called though the compiler driver `avr-gcc`. The latter is the usual way to invoke the linker. When these two versions are not the same, then the resulting toolchain may have inconsistencies.

When you have more then one toolchain installed, then make sure you are calling the command from the intended toolchain, e.g. by specifying the absolute path to the tool.

Also notice that software may come with vendor specific patches, so that the tool behaves differently than the stock version.

The FAQ document you are reading right now is part of an AVR-LibC that has been configured for the following tool versions:

Table 5 Tool Versions

Software	Version
Compiler	`avr-gcc (GCC) 14.2.0`
Assembler	`GNU assembler (GNU Binutils) 2.43.1`
LD Linker	`GNU ld (GNU Binutils) 2.43.1`
CC Linker	`GNU ld (GNU Binutils) 2.43.1`
AVR-LibC	`2.3.0`

- GCC 14 documentation
- GCC 14 release notes

The following features have been detected when building AVR-LibC. They influence how pars of the library behave.

Table 6 Toolchain Features affecting AVR-LibC

Feature	Supported	Affects
`__builtin_avr_delay_cycles()`	yes	`<util/delay.h>`
text starts at `__TEXT_REGION_ORIGIN__`	yes	Starup code
data starts at `__DATA_REGION_ORIGIN__`	yes	Starup code
Compiler supports `-mcvt`	no	CVT convenience
Compiler supports attribute `signal(n)`	no	`<avr/interrupt.h>`, ISR_N, local functions as ISRs
64-bit floating-point	yes: `-mdouble=64`, `-mlong-double=32`	`<math.h>`, Libraries
Fixed-point	yes	`<stdfix.h>`, Libraries
Device library `libmcu.a`	yes	Startup code, Libraries

Back to FAQ Index.

2.46 Where can I find the source code of the tools?

Here are web views of the GIT repos of the projects and some points of interest.

Binutils

- Top source: `https://sourceware.org/git/binutils-gdb.git`
  The linker is located in `ld/`, the assembler in `gas/` and the other tools are in `binutils/`.
- Linker script: `ld/scripttempl/avr.sc`

- AVR opcodes: include/opcodes/avr.h
- Latest manual

GCC

- Top source: https://gcc.gnu.org/git/gcc.git
- AVR backend: gcc/config/avr
- libgcc (AVR part): libgcc/config/avr
- Online manuals

AVR-LibC

- Top source: https://github.com/avrdudes/avr-libc.git
- Startup code: crt1/gcrt1.S
- Online manuals

Back to FAQ Index.

2.47 Where can I report a problem?

The projects have issue trackers:

- GCC Bugzilla and the GCC bug-reporting guide
- Binutils Bugzilla and the Binutils bug-reporting guide
- AVR-LibC issues

Please make sure to use the most recent tool version before filing an issue. When reporting a problem, please make sure that the report is self-contained and that the artifact can be reproduced by others.

Back to FAQ Index.

2.48 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 (CS2x = 0b001).
// Toggle OC2 output on compare match (COM2x = 0b01).
// Clear timer on compare match (CTC2 = 1).
TCCR2 = _BV(COM20) | _BV(CTC2) | _BV(CS20);

// Make OC2 (PD7) an output.
DDRD = _BV(PD7);
```

Back to FAQ Index.

3 Toolchain Overview

3.1 Introduction

Welcome to the open source software development toolset for the Microchip (formerly Atmel) AVR!

There is not a single tool that provides everything needed to develop software for the AVR. It takes many tools working together. Collectively, this group of tools is called a toolset, or commonly a toolchain, as the tools are chained together to produce the final executable application for the AVR microcontroller.

The following sections provide an overview of all of these tools. You may be used to cross-compilers that provide everything with a GUI front-end, and not know what goes on "underneath the hood". You may be coming from a desktop or server computer background and not used to embedded systems. Or you may be just learning about the most common software development toolchain available on Unix and Linux systems. Hopefully the following overview will be helpful in putting everything in perspective.

3.2 FSF and GNU

According to its website, "the Free Software Foundation (FSF), established in 1985, is dedicated to promoting computer users' rights to use, study, copy, modify, and redistribute computer programs. The FSF promotes the development and use of free software, particularly the GNU operating system, used widely in its GNU/Linux variant." The FSF remains the primary sponsor of the GNU project.

The GNU Project was launched in 1984 to develop a complete Unix-like operating system which is free software: the GNU system. GNU is a recursive acronym for "GNU's Not Unix"; it is pronounced guh-noo, approximately like canoe.

One of the main projects of the GNU system is the GNU Compiler Collection, or GCC, and its sister project, GNU Binutils. These two open source projects provide a foundation for a software development toolchain. Note that these projects were designed to originally run on Unix-like systems.

3.3 GCC

GCC stands for GNU Compiler Collection. GCC is a highly flexible compiler system. It has different compiler front-ends for different languages. It has many back-ends that generate assembly code for many different processors and host operating systems. All share a common "middle-end", containing the generic parts of the compiler, including a lot of optimizations.

In GCC, a *host* system is the system (processor/OS) that the compiler runs on. A *target* system is the system that the compiler compiles code for. And, a *build* system is the system that the compiler is built (from source code) on. If a compiler has the same system for *host* and for *target*, it is known as a *native* compiler. If a compiler has different systems for *host* and *target*, it is known as a cross-compiler. (And if all three, *build*, *host*, and *target* systems are different, it is known as a Canadian cross compiler, but we won't discuss that here.) When GCC is built to execute on a *host* system such as FreeBSD, Linux, or Windows, and it is built to generate code for the AVR microcontroller *target*, then it is a cross compiler, and this version of GCC is commonly known as "AVR GCC". In documentation, or discussion, AVR GCC is used when referring to GCC targeting specifically the AVR, or something that is AVR specific about GCC. The term "GCC" is usually used to refer to something generic about GCC, or about GCC as a whole.

GCC is different from most other compilers. GCC focuses on translating a high-level language to the target assembly only. AVR GCC has three available compilers for the AVR: C language, C++, and Ada. The compiler itself does not assemble or link the final code.

GCC is also known as a "driver" program, in that it knows about, and drives other programs seamlessly to create the final output. The assembler and the linker are parts of another open source project called GNU Binutils. GCC knows how to drive the GNU assembler (gas) to assemble the output of the compiler. GCC knows how to drive the GNU linker (ld) to link all of the object modules into a final executable.

The two projects, GCC and Binutils, are very much interrelated and many of the same volunteers work on both open source projects.

When GCC is built for the AVR target, the actual program names are prefixed with "avr-". So the actual executable name for AVR GCC is: avr-gcc. The name "avr-gcc" is used in documentation and discussion when referring to the program itself and not just the whole AVR GCC system.

See the GCC Web Site and GCC User Manual for more information about GCC.

3.4 GNU Binutils

The name GNU Binutils stands for "Binary Utilities". It contains the GNU assembler (gas), and the GNU linker (ld), but also contains many other utilities that work with binary files that are created as part of the software development toolchain.

Again, when these tools are built for the AVR target, the actual program names are prefixed with "avr-". For example, the assembler program name, for a native assembler is "as" (even though in documentation the GNU assembler is commonly referred to as "gas"). But when built for an AVR target, it becomes "avr-as". Below is a list of the programs that are included in Binutils:

`avr-as`
: The Assembler.

`avr-ld`
: The Linker.

`avr-ar`
: Create, modify, and extract from libraries (archives).

`avr-ranlib`
: Generate index to library (archive) contents.

`avr-objcopy`
: Copy and translate object files to different formats.

`avr-objdump`
: Display information from object files including disassembly.

`avr-size`
: List section sizes and total size.

`avr-nm`
: List symbols from object files.

`avr-strings`
: List printable strings from files.

`avr-strip`
: Discard symbols from files.

`avr-readelf`
: Display the contents of ELF format files.

`avr-addr2line`
: Convert addresses to file and line.

`avr-c++filt`
: Filter to demangle encoded C++ symbols.

3.5 AVR-LibC

GCC and Binutils provides a lot of the tools to develop software, but there is one critical component that they do not provide: a Standard C Library.

There are different open source projects that provide a Standard C Library depending upon your system time, whether for a native compiler (GNU Libc), for some other embedded system (Newlib), or for some versions of Linux (uCLibc). The open source AVR toolchain has its own Standard C Library project: AVR-LibC.

AVR-LibC provides many of the same functions found in a regular Standard C Library and many additional library functions that are specific to AVR. Some of the Standard C Library functions that are commonly used on a PC environment have limitations or additional issues that a user needs to be aware of when used on an embedded system.

AVR-LibC also contains the most documentation about the whole AVR toolchain.

3.6 Building Software

Even though GCC, Binutils, and AVR-LibC are the core projects that are used to build software for the AVR, there is another piece of software that ties it all together: Make. GNU Make is a program that makes things, and mainly software. Make interprets and executes a Makefile that is written for a project. A Makefile contains dependency rules, showing which output files are dependent upon which input files, and instructions on how to build output files from input files.

Some distributions of the toolchains, and other AVR tools such as MFile, contain a Makefile template written for the AVR toolchain and AVR applications that you can copy and modify for your application.

See the GNU Make User Manual for more information.

3.7 AVRDUDE

After creating your software, you'll want to program your device. You can do this by using the program AVRDUDE which can interface with various hardware devices to program your processor.

AVRDUDE is a very flexible package. All the information about AVR processors and various hardware programmers is stored in a text database. This database can be modified by any user to add new hardware or to add an AVR processor if it is not already listed.

3.8 GDB / Insight / DDD

The GNU Debugger (GDB) is a command-line debugger that can be used with the rest of the AVR toolchain. Insight is GDB plus a GUI written in Tcl/Tk. Both GDB and Insight are configured for the AVR and the main executables are prefixed with the target name: avr-gdb, and avr-insight. There is also a "text mode" GUI for GDB: avr-gdbtui. DDD (Data Display Debugger) is another popular GUI front end to GDB, available on Unix and Linux systems.

3.9 AVaRICE

AVaRICE is a back-end program to AVR GDB and interfaces to the AVR JTAG In-Circuit Emulator (ICE), to provide emulation capabilities.

3.10 SimulAVR

SimulAVR is an AVR simulator used as a back-end with AVR GDB.

3.11 AVRtest

AVRtest is a fast AVR core simulator used to test AVR GCC and AVR-LibC.

3.12 Utilities

There are also other optional utilities available that may be useful to add to your toolset.

`SRecord` is a collection of powerful tools for manipulating EPROM load files. It reads and writes numerous EPROM file formats, and can perform many different manipulations.

`MFile` is a simple Makefile generator is meant as an aid to quickly customize a Makefile to use for your AVR application.

3.13 Toolchain Distributions (Distros)

All of the various open source projects that comprise the entire toolchain are normally distributed as source code. It is left up to the user to build the tool application from its source code. This can be a very daunting task to any potential user of these tools.

Luckily, there are people who help out in this area. Volunteers take the time to build the application from source code on particular host platforms and sometimes packaging the tools for convenient installation by the end user. These packages contain the binary executables of the tools, pre-made and ready to use. These packages are known as "distributions" of the AVR toolchain, or by a more shortened name, "distros".

AVR toolchain distros are available on FreeBSD, Windows, Mac OS X, and certain flavors of Linux.

For details on the present distribution, see the FAQ.

3.14 Open Source

All of these tools, from the original source code in the multitude of projects, to the various distros, are put together by many, many volunteers. All of these projects could always use more help from other people who are willing to volunteer some of their time. There are many different ways to help, for people with varying skill levels, abilities, and available time.

You can help to answer questions in mailing lists such as the avr-gcc-list, or on forums at the AVR Freaks website. This helps many people new to the open source AVR tools.

If you think you found a bug in any of the tools, it is always a big help to submit a good bug report to the proper project. A good bug report always helps other volunteers to analyze the problem and to get it fixed for future versions of the software.

You can also help to fix bugs in various software projects, or to add desirable new features.

Volunteers are always welcome! :-)

4 Using the GNU tools

This is a short summary of the AVR-specific aspects of using the GNU tools. Normally, the generic documentation of these tools is fairly large and maintained in `texinfo` files. Command-line options are explained in detail in the manual page.

4.1 Options for the C compiler avr-gcc

4.1.1 Machine-specific options for the AVR

The following machine-specific options are recognized by the C compiler frontend. The preprocessor will define the macros `__AVR` and `__AVR__` (to the value 1) when compiling for an AVR target. The macro `AVR` will be defined as well, except in strict ANSI mode.

There are many options supported by avr-gcc, which also depend on the compiler version. For a complete overview, please see the documentation of avr-gcc's command line options. Here are links to supported options of the respective release series:

Table 7 GCC Release Notes and Online Documentation

Release Notes	Online Documentation
	`Current` (work in progress)
`GCC v15`	`GCC v15`
`GCC v14`	`GCC v14`
`GCC v13`	`GCC v13`
`GCC v12`	`GCC v12`
`GCC v11`	`GCC v11`
`GCC v10`	`GCC v10`
`GCC v9`	`GCC v9`
`GCC v8`	`GCC v8`
`GCC v7`	`GCC v7`
`GCC v6`	`GCC v6`
`GCC v5`	`GCC v5`
`GCC v4.9`	`GCC v4.9`
`GCC v4.8`	`GCC v4.8`

GCC v4.8 is the oldest compiler version supported by AVR-LibC.

Apart from the documentation of `command line options`, the linked pages also contain:

- The documentation of `built-in macros` like `__AVR_ARCH__`, `__AVR_ATmega328P__` and `__AVR_HAVE_MUL__`, just to mention a few.
- How the compiler treats the `RAMPX, RAMPY, RAMPZ and RAMPD special function registers` on devices that have (one of) them.
- How the compiler treats the `EIND special function register` on devices with more than 128 KiB of program memory, and how indirect calls are realized on such devices.

See also:

- `AVR variable attributes`

- AVR function attributes
- AVR named address-spaces
- AVR built-in functions

-mmcu=*arch*
-mmcu=*mcu*
Compile code for architecture *arch* resp. AVR device *mcu*.
Since GCC v5, the compiler no more supports indivitual devices, instead the compiler comes with device specs files that describe which options to use with each sub-processes like pre-processor, compiler proper, assembler and linker.
The purpose of these specs files is to add support for AVR devices that the compiler does not yet support.
The easiest way to add support for an unsupported device is to use device support from an atpack archive as provided by the hardware manufacturer. Apart from the *mcu* specific specs file, it provides device headers `io*.h`, startup code `crt`*mcu*`.o` and device library `lib`*mcu*`.a`.

4.1.2 Selected general compiler options

The following general gcc options might be of some interest to AVR users.

-O*n*
Optimization level *n*.
-O0
reduces compilation time and makes debugging produce the expected results. This is the default. Turning off all optimizations will prevent some warnings from being issued since the generation of those warnings depends on code analysis steps that are only performed when optimizing (unreachable code, unused variables). Moreover, the delay routines in <util/delay.h> require optimization to be turned on.
-O2
optimizes for speed, but without enabling very expensive optimizations like `-O3` does.
-Os
turns on all `-O2` optimizations except those that often increase code size. In most cases, this is the preferred optimization level for AVR programs.
-Og
optimizes debugging experience. This should be the optimization level of choice for the standard edit-compile-debug cycle, offering a reasonable level of optimization while maintaining fast compilation and a good debugging experience.
-O3
attempts to inline all "simple" functions and might unroll some loops. For the AVR target, this will normally constitute a large pessimization due to the code increase.
-O
is equivalent to `-O1`. The compiler tries to reduce code size and execution time, without performing any optimizations that take a great deal of compilation time.
See also the respective FAQ entry for issues regarding debugging optimized code.
-Wp,*preprocessor-options*
-Wa,*assembler-options*
-Wl,*linker-options*
Pass the listed options to the C preprocessor, the assembler or the linker, respectively. Several options can be passed at once when they are separated by a `,` (comma).
-g
Generate debugging information that can be used by avr-gdb. GCC v12 changed the default from STABS to DWARF. Different DWARF levels can be selected by `-g2` or `-gdwarf-3`.
The compiler may use GNU extensions to the DWARF format. When a debugger has problems with that, try `-gstrict-dwarf`.

-x *lang*
-x none
Compile the following files in language *lang*. Values for *lang* are: `c`, `c++`, `assembler`, `assembler-with-cpp` and `none`.
For example, GCC does not recognize the `.asm` file extension as assembly source. With `-x assembler-with-cpp file.asm`, the compiler first runs the C preprocessor on `file.asm` (so that `#include <avr/io.h>` can be used in assembly), and then calls the assembler.
Another use case is to compile a C file that's read from standard input, which is specified as `-` (dash) instead of the name of a source file. As no source file name is specified, the compiler must be told which language to use: The command

```
$ echo '#include <avr/io.h>' | avr-gcc -xc - -mmcu=atmega8 -E -dM | grep _VECTOR
```

pre-processes the C file `#include <avr/io.h>` and writes all macro definitions to stdout. The output is then filtered by `grep` to show all possible ISR vector names for ATmega8.
`-x none` returns to the default for the following inputs, i.e. infer the respective source languages from the file extensions.

-save-temps
-save-temps=obj
-save-temps=cwd
-fverbose-asm
-dumpbase *base*
-dumpdir *dir*
Don't remove temporary, intermediate files like C preprocessor output and assembly code generated by the compiler. The intermediate files have file extension `.i` (preprocessed C), `.ii` (preprocessed C++) and `.s` (preprocessed assembly, compiler-generated assembly).
`-dumpbase` and `-dumpdir` can be used to adjust file names and locations.
With `-fverbose-asm`, the compiler adds the high level source code to the assembly output. Compiling without debugging information (`-g0`) improves legibility of the generated assembly.
The preprocessed files can be used to check if macro expansions work as expected. With `-g3` or higher, the preprocessed files will also contain all macro definitions and indications where they are defined: Built-in, on the command line, or in some header file as indicated by `#line` notes.

-l*name*
Locate the archive library named `lib`*name*`.a`, and use it to resolve currently unresolved symbols from it. The library is searched along a path that consists of builtin pathname entries that have been specified at configure time (e. g. `/usr/local/avr/lib` on Unix systems), possibly extended by pathname entries as specified by `-L` options (that must precede the `-l` options on the command-line).

-L*path*
Additional directory *path* to look for archive libraries requested by `-l` options.

-ffunction-sections
-fdata-sections
Put each function resp. object in static storage in its own input section. This is used with `-Wl,--gc-sections` so the linker can better garbage-collect unused sections, which are sections that are neither referenced by other sections, nor are marked as `KEEP`, nor are referenced by an entry symbol.

-mrelax
Replace `JMP` and `CALL` instructions with the faster and shorter `RJMP` and `RCALL` instructions if possible. That optimization is performed by the linker, and the assembler must not resolve some expressions, which is all arranged by `-mrelax`.

-Tbss *org*
-Tdata *org*
-Ttext *org*
Start the `.bss`, `.data`, or `.text` section at VMA address *org*, respectively.

-T *scriptfile*
Use *scriptfile* as the linker script, replacing or augmenting the default linker script.
Default linker scripts are stored in a system-specific location (e. g. under `/usr/local/avr/lib/ldscripts` on Unix systems), and consist of the AVR architecture name (`avr2` through `avrxmega7`) with the suffix `.x` appended. They describe how the various memory sections will be linked together and which input sections go into them. Notice that the default linker scripts are part of the linker binary, changing them on file will have no effect.

For a simple linker script augmentation, see the avr-gcc Wiki on named address spaces.

-nostdlib
Don't link against standard libraries.

-nodefaultlibs
Don't link against default libraries.

-nodevicelib
Don't link against AVR-LibC's lib*mcu*.a that contains EEPROM support and other stuff. This can be used when no such library is available.

-nostartfiles
Don't link against AVR-LibC's startup code crt*mcu*.o.
Notice that parts of the startup code are provided by libgcc.a. To get rid of that, one can -nostdlib or -nodefaultlibs; however that also removes other code like functions required for arithmetic. To just get rid of the startup bits, define the respective symbols, for example
-Wl,--defsym,__do_clear_bss=0 and similar for __do_copy_data, __do_global_ctors and __do_global_dtors.

-funsigned-char
This option changes the binary interface!
Make any unqualfied char type an unsigned char. Without this option, they default to a signed char.

-funsigned-bitfields
This option changes the binary interface!
Make any unqualified bitfield type unsigned. By default, they are signed.

-fshort-enums
This option changes the binary interface!
Allocate to an enum type only as many bytes as it needs for the declared range of possible values. Specifically, the enum type will be equivalent to the smallest integer type which has enough room.

-fpack-struct
This option changes the binary interface!
Pack all structure members together without holes.

-fno-jump-tables
Do not generate tablejump instructions. By default, jump tables can be used to optimize switch statements. When turned off, sequences of compare statements are used instead. Jump tables are usually faster to execute on average, but in particular for switch statements, where most of the jumps would go to the default label, they might waste a bit of flash memory.
Note: Sinve GCC v4.9.2, tablejump code uses the ELPM instruction to read from jump tables. In older version, use the -fno-jump-tables switch when compiling a bootloader for devices with more than 64 KiB of code memory.

-ffreestanding
Assume a "freestanding" environment as per the C standard. This turns off automatic builtin functions (though they can still be reached by prepending __builtin_ to the actual function name). It also makes the compiler not complain when main() is declared with a void return type. (In most cases, main() won't even return anyway, and hence using a return type of int has no downsides at all).
This option turns off all optimizations normally done by the compiler which assume that functions known by a certain name behave as described by the standard. For example, applying the function strlen() to a literal string will normally cause the compiler to immediately replace that call by the actual length of the string, while with -ffreestanding, it will always call strlen() at run-time.

4.2 Options for the assembler avr-as

Note

The preferred way to assemble a file is by means of using avr-gcc:

- avr-gcc, which is a driver program to call sub-programs like the compiler proper or the assembler, knows which options it has to add to the assembler's command line, like: -mmcu=*arch*, -mno-skip-bug, etc.
- avr-gcc will call the C preprocessor on the assembler input for sources with extensions .S and .sx. For other extensions, use

```
$ avr-gcc -x assembler-with-cpp file.asm ...
```

This allows to use C preprocessor directives like

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

in assembly sources.

4.2.1 Machine-specific assembler options

-mmcu=*architecture*
-mmcu=*mcu*
: `avr-as` does not support all *mcu*s supported by the compiler. As explained in the note above, the preferred way to run the assembler is by using the compiler driver `avr-gcc`.

-mall-opcodes
: Turns off opcode checking, and allows any possible AVR opcode to be assembled.

-mno-skip-bug
: Don't emit a warning when trying to skip a 2-word instruction with a `CPSE/SBIC/SBIS/SBRC/SBRS` instruction. Early AVR devices suffered from a hardware bug where these instructions could not be properly skipped.

-mno-wrap
: For `RJMP/RCALL` instructions, don't allow the target address to wrap around for devices that have more than 8 KiB of memory.

--gstabs
: Generate `.stabs` debugging symbols for assembler source lines. This enables `avr-gdb` to trace through assembler source files. This option *must not* be used when assembling sources that have been generated by the C compiler; these files already contain the appropriate line number information from the C source files.

-a[cdhlmns=*file*]
: Turn on the assembler listing. The sub-options are:

  - `c` omit false conditionals
  - `d` omit debugging directives
  - `h` include high-level source
  - `l` include assembly
  - `m` include macro expansions
  - `n` omit forms processing
  - `s` include symbols
  - `=file` set the name of the listing file

  The various sub-options can be combined into a single `-a` option list; *=file* must be the last one in that case.

4.2.2 Examples for assembler options passed through the C compiler

Remember that assembler options can be passed from the C compiler frontend using `-Wa` (see gcc_minusW above), so in order to include the C source code into the assembler listing in file `foo.lss`, when compiling `foo.c`, the following compiler command-line can be used:

```
$ avr-gcc -mmcu=atmega8 -c foo.c -o foo.o -Wa,-ahls=foo.lss
```

In order to pass an assembler file through the C preprocessor first, and have the assembler generate line number debugging information for it, the following command can be used:

```
$ avr-gcc -c -x assembler-with-cpp -o foo.o foo.asm -Wa,--gstabs
```

Note that on Unix systems that have case-distinguishing file systems, specifying a file name with the suffix `.S` (upper-case letter `S`) will make the compiler automatically assume `-x assembler-with-cpp`, while using `.s` would pass the file directly to the assembler (no preprocessing done).

4.3 Controlling the linker avr-ld

Note

It is highly recommended to use the compiler driver `avr-gcc` or `avr-g++` for linking.

- The driver knows which options to pass down to the linker. This includes the correct multilib path, support libraries like `libc.a`, `libm.a`, `libgcc.a` and `libmcu.a`, as well as options for LTO (link-time optimization) and options for plugins (that call back the compiler to compile LTO byte code), startup code and many more.
- The driver program understands options like `-l`*`lib`*, `-L`*`path`*, `Ttext`, `Tdata`, `Tbss` and `-T` *`script`*, so no `-Wl` is required for them.

4.3.1 Selected linker options

A number of the standard options might be of interest to AVR users.

`--defsym` *`symbol=expr`*
Define a global symbol *symbol* using *expr* as the value.

`-M`
Print a linker map to `stdout`.

`-Map` *`mapfile`*
Print a linker map to *mapfile*.

`--cref`
Output a cross reference table to the map file (in case `-Map` is also present), or to `stdout`.

`--gc-sections`
Only keep input sections that are referenced (by other sections or the entry symbol), and that are not marked as `KEEP` in the linker script. This is used to reduce code size, usually together with compiler option `-ffunction-sections` so that input section granularity is on function level rather than on the level of compilation units.

`--section-start` *`sectionname=org`*
Start section *sectionname* at absolute address *org*.

4.3.2 Passing linker options from the C compiler

By default, all unknown non-option arguments on the `avr-gcc` command-line (i. e., all filename arguments that don't have a suffix that is handled by `avr-gcc`) are passed straight to the linker. Thus, all files ending in `.o` (object files) and `.a` (object libraries) are provided to the linker.

System libraries are usually not passed by their explicit filename but rather using the `-l` option which uses an abbreviated form of the archive filename (see above). AVR-LibC ships system libraries, `libc.a`, `libm.a` and `libmcu.a`. All these standard libraries will always be searched for unresolved references when the linker is started using the C/C++ compiler frontend, i. e., there are always implied options `-lc` and `-lm`. When a device library exists, which is the case since GCC v5, then `-l`*`mcu`* will be automatically added, too.

Conventionally, Makefiles use the `make` macro `LIBS` to keep track of `-l` (and possibly `-L`) options that should only be appended to the C compiler command-line when linking the final binary. In contrast, the macro `LDFLAGS` is used to store other command-line options to the C compiler that should be passed as options during the linking stage. The difference is that options are placed early on the command-line, while libraries are put at the end since they are to be used to resolve global symbols that are still unresolved at this point.

Specific linker flags can be passed from the C compiler command-line using the `-Wl` compiler option, see gcc_minusW above. This option requires that there be no spaces in the appended linker option, while some of the linker options above (like `-Map` or `--defsym`) would require a space. In these situations, the space can be replaced by an equal sign as well. For example, the following command-line can be used to compile `foo.c` into an executable, and also produce a link map that contains a cross-reference list in the file `foo.map`:

```
$ avr-gcc -mmcu=atmega8 foo.c -o foo.elf -Wl,-Map,foo.map -Wl,--cref
```

Alternatively, a comma as a placeholder will be replaced by a space before passing the option to the linker. So for a device with external SRAM, the following command-line would cause the linker to place the data segment at address 0x2000 in the SRAM:

```
$ avr-gcc -mmcu=atmega128 foo.c -o foo.elf -Wl,-Tdata,0x802000
```

See the explanation of the data section for why 0x800000 needs to be added to the actual value. Note that the stack will still remain in internal RAM, through the symbol `__stack` that is provided by the run-time startup code. This is probably a good idea anyway (since internal RAM access is faster), and even required for some early devices that had hardware bugs preventing them from using a stack in external RAM. Note also that the heap for malloc() will still be placed after all the variables in the data section, so in this situation, no stack/heap collision can occur.

In order to relocate the stack from its default location at the top of interns RAM, the value of the symbol `__stack` can be changed on the linker command-line. As the linker is typically called from the compiler frontend, this can be achieved using a compiler option like

```
-Wl,--defsym=__stack=0x8003ff
```

The above will make the code use stack space from RAM address 0x3ff downwards. Symbol `__stack` is picked up by the startup code to initialize the `SP` register. The amount of stack space available then depends on the bottom address of internal RAM for a particular device. It is the responsibility of the application to ensure the stack does not grow out of bounds, as well as to arrange for the stack to not collide with variable allocations made by the compiler (sections .data and .bss).

5 Building and Installing the GNU Tool Chain

This chapter shows how to build and install, from source code, a complete development environment for the AVR processors using the GNU toolset. There are two main sections, one for Linux, FreeBSD, and other Unix-like operating systems, and another section for Windows.

5.1 Required AVR Tools

GNU Binutils
Project Home: https://sourceware.org/binutils
Source Downloads: https://sourceware.org/pub/binutils/releases
FTP: anonymous@ftp.gnu.org/gnu/binutils
Git: git://sourceware.org/git/binutils-gdb.git
GitHub Mirror: https://github.com/bminor/binutils-gdb
Installation

GCC
Project Home https://gcc.gnu.org
Mirrors Site: https://gcc.gnu.org/mirrors.html
FTP: anonymous@ftp.gnu.org/gnu/gcc
Git: git://gcc.gnu.org/git/gcc.git
GitHub Mirror: https://github.com/gcc-mirror/gcc
Installation: https://gcc.gnu.org/install
Installation

AVR-LibC
New Project Home: https://avrdudes.github.io/avr-libc
New Source Downloads ($\geq$ v2.2): https://github.com/avrdudes/avr-libc/releases
Git: https://github.com/avrdudes/avr-libc.git
GitHub: https://github.com/avrdudes/avr-libc
Old Project Home: http://savannah.gnu.org/projects/avr-libc
Old Source Downloads ($\leq$ v2.1): https://download-mirror.savannah.gnu.org/releases/avr-libc
Installation

5.2 Optional AVR Tools

You can develop programs for AVR devices without the following tools. They may or may not be of use for you.

AVRDUDE
Git: https://github.com/avrdudes/avrdude.git
GitHub: https://github.com/avrdudes/avrdude
Installation

GDB
The GNU Debugger GDB is hosted together with GNU Binutils. When you don't want or need GDB, you can configure Binutils with `--disable-gdb`.

SimulAVR
http://savannah.gnu.org/projects/simulavr
Installation

AVRtest
GitHub: https://github.com/sprintersb/atest
Installation
AVRtest is an AVR core simulator that can be used to run the avr-gcc testsuite. It can also be used to run parts of the AVR-LibC testsuite. For details, see the README.

AVaRICE
GitHub: https://github.com/avrdudes/avarice
Installation

5.3 Building and Installing under Linux, FreeBSD, and Others

The default behaviour for most of these tools is to install every thing under the `/usr/local` directory. In order to keep the AVR tools separate from the base system, it is usually better to install everything into `/usr/local/avr`.

If the /usr/local/avr directory does not exist, you should create it before trying to install anything. You will need root access to install there. If you don't have or want root access to the system, you can alternatively install in your home directory, for example, in $HOME/local/avr. Where you install is a completely arbitrary decision, but should be consistent for all the tools.

It is usually the best to use the latest released version of each of the tools.

Warning

If you have CC set to anything other than avr-gcc in your environment, this will cause the configure script to fail. It is best to not have CC set at all.

5.3.1 Build Scripts

There are scripts that can simplify the process of building an AVR toolchain for Linux, and/or for Windows in Canadian cross configuration.

- GitHub: avr-gcc-build from Zak Kemble.
- GitHub: make-avr-gcc

5.3.2 Preparations

5.3.2.1 Required Tools on the Build Machine

In order to be able to build and install the compiler and other tools, the build machine requires some software packages. The Debian package list is something like:

- perl gawk binutils gcc-multilib python3 python3-pip gzip make tar zstd autoconf automake gettext gperf dejagnu autogen guile-3.0 expect tcl flex texinfo git diffutils patch git-email

5.3.2.2 Install Location

You specify the installation directory by using the --prefix=dir option with the configure script. It is important to install all the AVR tools in the same directory, or some of the tools will not work correctly. To ensure consistency and simplify the discussion, we will use $PREFIX to refer to whatever directory you wish to install in. You can set this as an environment variable if you wish as such (using a Bourne-like shell):

```
$ PREFIX=$HOME/local/avr
$ export PREFIX
```

Note

Be sure that you have your PATH environment variable set to search the directory you install everything in *before* you start installing anything. For example, if you use --prefix=$PREFIX, you must have $↩ PREFIX/bin in your exported PATH. As such:

```
$ PATH=$PATH:$PREFIX/bin
$ export PATH
```

5.3.2.3 Directory Layout

The instructions below build Binutils, GCC and AVR-LibC *outside* of the source tree, because:

- When something goes wrong, you can just remove the build directory and start all over again with a fresh build folder.
- You may want to build the tools with different configure options, e.g. build the tools for a Linux host and then build a Canadian cross to run on a Windows host.
- GCC does not support configuring anywhere in the source tree, so we'll have to use a separate build folder outside the source tree, anyway.

The instructions below assume that you have set up a directory tree like

```
+--source
+--build
```

in some place where you have write access, like in your home directory.

After successful downloads and builds, the tree will be something like:

```
+--source
|   +--gcc-<version>
|   +--binutils-<version>
|   +--avr-libc-<version>
+-- build
    +--gcc-<version>-avr
    +--binutils-<version>-avr
    +--avr-libc-<version>
```

5.3.3 GNU Binutils for the AVR target

The **Binutils** package provides all the low-level utilities needed in building and manipulating object files. Once installed, your environment will have an AVR assembler (`avr-as`), linker (`avr-ld`), and librarian (`avr-ar` and `avr-ranlib`). In addition, you get tools which extract data from object files (`avr-objcopy`), disassemble object file information (`avr-objdump`), and strip information from object files (`avr-strip`). Before we can build the C compiler, these tools need to be in place.

Download and unpack the source files:

```
$ # in ./source
$ tar xfj binutils-<version>.tar.bz2
```

Replace `<version>` with the version of the package you downloaded.

If you obtained a gzip compressed file (`.tar.gz` or .tgz), use `gunzip` instead of `bunzip2`, or `tar xfz` *`file`*`.tar.gz`.

The next step is to configure and build the tools. This is done by supplying arguments to the `configure` script that enable the AVR-specific options. When you also want GDB, just drop `--disable-gdb`.

```
$ # in ./build
$ mkdir binutils-<version>-avr
$ cd binutils-<version>-avr
$ ../../source/binutils-<version>/configure --prefix=$PREFIX --target=avr \
    --disable-nls --disable-sim --disable-gdb --disable-werror
```

When `configure` is run, it generates a lot of messages while it determines what is available on your operating system. When it finishes, it will have created several `Makefile`s that are custom tailored to your platform and that are run with the `make` command.

```
$ make
```

BSD users should note that the project's `Makefile` uses GNU make syntax. This means FreeBSD users may need to build the tools by using `gmake`.

If the tools compiled cleanly, you're ready to install them. If you specified a destination that isn't owned by your account, you'll need `root` access to install them. To install:

```
$ make install
```

You should now have the programs from Binutils installed into `$PREFIX/bin`. Don't forget to set your PATH environment variable before going to build avr-gcc. To check that the correct assembler is found, run

```
$ avr-as --version
```

which should print the `<version>` of the used Binutils sources.

5.3.4 GCC for the AVR target

Warning

You **must** install avr-binutils and make sure your path is set properly before installing avr-gcc.

Before we can configure the compiler, we have to prepare the sources. GCC depends on some external host libraries, namely `GMP`, `MPFR`, `MPC` and `ISL`. You can build and install the appropriate versions of the required prerequisites by hand and provide their location by means of `--with-gmp=` etc. Though in most situations it is easier to let GCC download and build these libraries as part of the configure and build process. All what's needed is an internet connection when running `./contrib/download_prerequisites`:

```
$ # in ./source
$ tar xfj gcc-<version>.tar.bz2
$ cd gcc-<version>
$ ./contrib/download_prerequisites

$ # in ./build
$ mkdir gcc-<version>-avr
$ cd gcc-<version>-avr
$ ../../source/gcc-<version>/configure --prefix=$PREFIX --target=avr --enable-languages=c,c++ \
    --disable-nls --disable-libssp --disable-libcc1 \
    --with-gnu-as --with-gnu-ld --with-dwarf2
$ make
$ make install # or make install-strip
```

The GCC binaries may consume quite some disc space. In most cases, you don't need the debug information in the compiler proper, and installing with

```
$ make install-strip
```

can save you some space.

5.3.5 AVR-LibC

Warning

> You **must** install avr-binutils, avr-gcc and make sure your path is set properly before configuring abd building AVR-LibC.

If you have obtained the latest AVR-LibC from git, you will have to run the `./bootstrap` script before using either of the build methods described below.

To build and install AVR-LibC:

```
$ # in ./source
$ tar xfz avr-libc-<version>.tar.gz

$ # in ./build
$ mkdir avr-libc-<version>
$ cd avr-libc-<version>
$ ../../source/avr-libc-<version>/configure --prefix=$PREFIX \
    --build=x86_64-pc-linux-gnu --host=avr
$ make
$ make install
```

Where the `--build` platform can be guessed by running

```
$ ./source/avr-libc-<version>/config.guess
```

When you want to build and install with extra options, you can for example:

```
$ make CFLAGS="-Werror" CCASFLAGS="-Werror"
$ make install prefix=/home/me/install/my-avr-gcc
```

In order to build the examples:

```
$ # in ./build/doc/examples
$ make all-examples
$ make example-largedemo # etc.
```

5.3.5.1 AVR-LibC Documentation

Additional prerequisites:

- Doxygen v1.9.6
- `fig2dev` from the `Xfig` project
- A LaTeX installation like "TeX Live" with `pdflatex`
- Python can improve the rendering of some man pages, though Python is not strictly required to build the documentation.

When `fig2dev` is not present on the system, one way is to build it from source:

```
$ wget https://sourceforge.net/projects/mcj/files/fig2dev-3.2.9a.tar.xz/download -O fig2dev-3.2.9a.tar.xz
$ tar xfJ fig2dev-3.2.9a.tar.xz
```

and then configure, make and install as usual.

For building the documentation, Doxygen v1.9.6 is required. Here are the steps for building and installing it in a non-system location (so that no root permissions are needed). The first step is to get the required Doxygen source version:

```
$ # in ./source
$ git clone --branch Release_1_9_6 --depth 1 https://github.com/doxygen/doxygen.git doxygen-1.9.6
```

Then configure and build it. The installation is in some folder `/someplace` which you can adjust as you like:

```
$ # in ./build
$ mkdir doxygen-1.9.6
$ cd doxygen-1.9.6
$ cmake -DCMAKE_INSTALL_PREFIX:PATH=/someplace/doxygen-1.9.6 ../../source/doxygen-1.9.6
$ make
$ make install
```

There are several ways to provide Doxygen 1.9.6 to AVR-LibC:

- Configure AVR-LibC with

  `--with-doxygen=/someplace/doxygen-1.9.6/bin/doxygen`

- or add `/someplace/doxygen-1.9.6/bin` to `PATH` prior to configuring AVR-LibC.

The AVR-LibC configure script will print the used Doxygen version. Make sure it is actually Doxygen 1.9.6!

In order to build the documentation, you can add `--enable-doc` to the AVR-LibC configure options, or you can invoke building the documentation by hand:

```
$ # in ./build/avr-libc-<version>/doc/api
$ make html
$ make dox-pdf
$ make install-dox-html
```

Note that Doxygen doesn't care about dependencies, thus when you made changes to the sources and want to re-build the documentation, better run `make clean` in `build/avr-libc-<version>/doc/api` before building the doc. Moreover, the Doxygen version is highly critical, and the exact version of Doxygen v1.9.6 is recommended.

5.3.6 AVRDUDE

AVRDUDE has been ported to Windows (via MinGW or Cygwin), Linux and Solaris. Other Unix systems should be trivial to port to.

AVRDUDE is part of the FreeBSD ports system. To install it, simply do the following:

```
# cd /usr/ports/devel/avrdude
# make install
```

Installation into the default location usually requires root permissions. However, running the program only requires access permissions to the appropriate `ppi(4)` device.

Building and installing on other systems should use the `configure` system, as such:

```
$ gunzip -c avrdude-<version>.tar.gz | tar xf -
$ cd avrdude-<version>
$ mkdir obj-avr
$ cd obj-avr
$ ../configure --prefix=$PREFIX
$ make
$ make install
```

5.3.7 SimulAVR

SimulAVR also uses the `configure` system, so to build and install:

```
$ gunzip -c simulavr-<version>.tar.gz | tar xf -
$ cd simulavr-<version>
$ mkdir obj-avr
$ cd obj-avr
$ ../configure --prefix=$PREFIX
$ make
$ make install
```

You might want to have already installed avr-binutils, avr-gcc and AVR-LibC if you want to have the test programs built in the simulavr source.

5.3.8 AVRtest

Building AVRtest is done by running its Makefile. No configure script is required. To get the sources and make:

```
$ git clone https://github.com/sprintersb/atest.git <folder>
$ cd <folder>
$ make
```

The Makefile will build some object files like `exit-atmega128.o` which require an avr-gcc that supports such devices. There are also pre-built AVRtest binaries for Windows.

5.3.9 AVaRICE

These install notes are not applicable to avarice-1.5 or older. You probably don't want to use anything that old anyway since there have been many improvements and bug fixes since the 1.5 release.

AVaRICE also uses the `configure` system, so to build and install:

```
$ gunzip -c avarice-<version>.tar.gz | tar xf -
$ cd avarice-<version>
$ mkdir obj-avr
$ cd obj-avr
$ ../configure --prefix=$PREFIX
$ make
$ make install
```

AVaRICE uses the BFD library for accessing various binary file formats. You may need to tell the configure script where to find the lib and headers for the link to work. This is usually done by invoking the configure script like this (Replace `<hdr_path>` with the path to the `bfd.h` file on your system. Replace `<lib_path>` with the path to `libbfd.a` on your system.):

```
$ CPPFLAGS=-I<hdr_path> LDFLAGS=-L<lib_path> ../configure --prefix=$PREFIX
```

5.4 Building and Installing under Windows

Building and installing the toolchain under Windows requires more effort because all of the tools required for building, and the programs themselves, are mainly designed for running under a POSIX environment such as Unix and Linux. Windows does not natively provide such an environment.

There are two projects available that provide such an environment, Cygwin and MinGW. There are advantages and disadvantages to both. Cygwin provides a very complete POSIX environment that allows one to build many Linux based tools from source with very little or no source modifications. However, POSIX functionality is provided in the form of a DLL that is linked to the application. This DLL has to be redistributed with your application, and there are issues if the Cygwin DLL already exists on the installation system and different versions of the DLL. On the other hand, MinGW can compile code as native Win32 applications. However, this means that programs designed for Unix and Linux (i.e. that use POSIX functionality) will not compile as MinGW does not provide that POSIX layer for you. Therefore, most programs that compile on both types of host systems, usually must provide some sort of abstraction layer to allow an application to be built cross-platform.

MinGW does provide somewhat of a POSIX environment, called MSYS, that allows you to build Unix and Linux applications as they would normally do, with a `configure` step and a `make` step. Cygwin also provides such an environment. This means that building the AVR toolchain is very similar to how it is built in Linux, described above. The main differences are in what the PATH environment variable gets set to, pathname differences, and the tools that are required to build the projects under Windows. We'll take a look at the tools next.

5.4.1 Tools Required for Building the Toolchain for Windows

These are the tools that are currently used to build an AVR tool chain. This list may change, either the version of the tools, or the tools themselves, as improvements are made.

MinGW

Download the MinGW Automated Installer, 2013-10-04 (or later) https://sourceforge.net/projects/mingw/files

- Run `mingw-get-setup.exe`
- In the installation wizard, keep the default values and press the "Next" button for all installer pages except for the pages explicitly listed below.
- In the installer page "Repository Catalogues", select the "Download latest repository catalogues" radio button, and press the "Next" button
- In the installer page "License Agreement", select the "I accept the agreement" radio button, and press the "Next" button
- In the installer page "Select Components", be sure to select these items:
  - C compiler (default checked)
  - C++ compiler
  - Ada compiler
  - MinGW Developer Toolkit (which includes "MSYS Basic System").
- Install.

Install Cygwin

Install everything, all users, UNIX line endings. This will take a *long* time. A fat internet pipe is highly recommended. It is also recommended that you download all to a directory first, and then install from that directory to your machine.

GMP, MPFR, MPC and ISL are required to build GCC. By far the easiest way to use them is by letting GCC download the sources locally by means of running the `./contrib/download_prewrequisites` script from the GCC top source. GCC will configure and build these libs during `configure` and `make`.

Doxygen is required to build AVR-LibC documentation.

- **Install Doxygen**

  - Version 1.9.6
  - https://www.doxygen.nl
  - Download version 1.9.6 and install.

fig2dev is required to build graphics in the AVR-LibC documentation.

- **Install fig2dev**

  - Version 3.2 patchlevel 5c
  - From WinFig 4.62: http://winfig.com/downloads
  - Download the zip file version of WinFig
  - Unzip the download file and install fig2dev.exe in a location of your choice, somewhere in the PATH.
  - You may have to unzip and install related DLL files for fig2dev. In the version above, you have to install QtCore4.dll and QtGui4.dll.

MikTeX is required to build various documentation.

- **Install MiKTeX**

  - Version 2.9
  - https://miktex.org
  - Download and install.

Set the `TEMP` and `TMP` environment variables to **`c:\\temp`** or to the short filename version. This helps to avoid NTVDM errors during building.

5.4.2 Building the Toolchain for Windows

All directories in the PATH environment variable should be specified using their short filename (8.3) version. This will also help to avoid NTVDM errors during building. These short filenames can be specific to each machine.

Build the tools below in MinGW/MSYS.

- **Binutils**

  - Open source code package and patch as necessary.
  - Configure and build in a directory outside of the source code tree.
  - Set PATH, in order:
    - <MikTex executables>
    - /usr/local/bin
    - /usr/bin
    - /bin
    - /mingw/bin
    - c:/cygwin/bin

    - <install directory>/bin
  - Configure

```
CFLAGS=-D__USE_MINGW_ACCESS  \
../$archivedir/configure \
    --prefix=$installdir \
    --target=avr \
    --disable-nls \
    --enable-doc \
    --datadir=$installdir/doc/binutils \
    2>&1 | tee binutils-configure.log
```

  - Make

```
make all html install install-html 2>&1 | tee binutils-make.log
```

  - Manually change documentation location.

- **GCC**

  - Open source code package and patch as necessary.
  - Configure and build in a directory outside of the source code tree.
  - Set PATH, in order:
    - <MikTex executables>
    - /usr/local/bin
    - /usr/bin
    - /bin
    - /mingw/bin
    - c:/cygwin/bin
    - <install directory>/bin
  - Configure

```
LDFLAGS='-L /usr/local/lib -R /usr/local/lib' \
CFLAGS='-D__USE_MINGW_ACCESS'  \
../gcc-$version/configure \
    --prefix=$installdir \
    --target=$target \
    --enable-languages=c,c++ \
    --with-dwarf2 \
    --enable-doc \
    --with-docdir=$installdir/doc/$project \
    --disable-shared \
    --disable-libada \
    --disable-libssp \
    --disable-libcc1 \
    --disable-nls \
    2>&1 | tee $project-configure.log
```

  - Make

```
make all html install 2>&1 | tee $package-make.log
```

- **AVR-LibC**

  - Open source code package.
  - Configure and build at the top of the source code tree.
  - Set PATH, in order:
    - /usr/local/bin
    - /mingw/bin
    - /bin
    - <MikTex / TeX live executables>

    - <install directory>/bin
    - <Doxygen executables>
    - <fig2dev executable>
    - c:/cygwin/bin
  - Configure

```
./configure \
    --host=avr \
    --prefix=$installdir \
    --enable-doc \
    --disable-versioned-doc \
    --enable-html-doc \
    --enable-pdf-doc \
    --enable-man-doc \
    --mandir=$installdir/man \
    --datadir=$installdir \
    2>&1 | tee $package-configure.log
```

  - Make

```
make all install 2>&1 | tee $package-make.log
```

  - Manually change location of man page documentation.
  - Move the examples to the top level of the install tree.
  - Convert line endings in examples to Windows line endings.
  - Convert line endings in header files to Windows line endings.

- **AVRDUDE**

  - Open source code package.
  - Configure and build at the top of the source code tree.
  - Set PATH, in order:
    - <MikTex executables>
    - /usr/local/bin
    - /usr/bin
    - /bin
    - /mingw/bin
    - c:/cygwin/bin
    - <install directory>/bin
  - Set location of LibUSB headers and libraries

```
export CPPFLAGS="-I../../libusb-win32-device-bin-$libusb_version/include"
export CFLAGS="-I../../libusb-win32-device-bin-$libusb_version/include"
export LDFLAGS="-L../../libusb-win32-device-bin-$libusb_version/lib/gcc"
```

  - Configure

```
./configure \
    --prefix=$installdir \
    --datadir=$installdir \
    --sysconfdir=$installdir/bin \
    --enable-doc \
    --disable-versioned-doc \
    2>&1 | tee $package-configure.log
```

  - Make

```
make -k all install 2>&1 | tee $package-make.log
```

  - Convert line endings in avrdude config file to Windows line endings.
  - Delete backup copy of avrdude config file in install directory if exists.

- **Insight/GDB**
  - Open source code package and patch as necessary.
  - Configure and build in a directory outside of the source code tree.
  - Set PATH, in order:
    - <MikTex executables>
    - /usr/local/bin
    - /usr/bin
    - /bin
    - /mingw/bin
    - c:/cygwin/bin
    - <install directory>/bin
  - Configure

```
CFLAGS=-D__USE_MINGW_ACCESS  \
LDFLAGS='-static' \
../$archivedir/configure \
    --prefix=$installdir \
    --target=avr \
     --with-gmp=/usr/local \
     --with-mpfr=/usr/local \
     --enable-doc \
     2>&1 | tee insight-configure.log
```

  - Make

```
make all install 2>&1 | tee $package-make.log
```

- **SRecord**
  - Open source code package.
  - Configure and build at the top of the source code tree.
  - Set PATH, in order:
    - <MikTex executables>
    - /usr/local/bin
    - /usr/bin
    - /bin
    - /mingw/bin
    - c:/cygwin/bin
    - <install directory>/bin
  - Configure

```
./configure \
    --prefix=$installdir \
    --infodir=$installdir/info \
    --mandir=$installdir/man \
    2>&1 | tee $package-configure.log
```

  - Make

```
make all install 2>&1 | tee $package-make.log
```

Build the tools below in Cygwin.

- **AVaRICE**

- Open source code package.
- Configure and build in a directory outside of the source code tree.
- Set PATH, in order:
  - <MikTex executables>
  - /usr/local/bin
  - /usr/bin
  - /bin
  - <install directory>/bin
- Set location of LibUSB headers and libraries

```
export CPPFLAGS=-I$startdir/libusb-win32-device-bin-$libusb_version/include
export CFLAGS=-I$startdir/libusb-win32-device-bin-$libusb_version/include
export LDFLAGS="-static -L$startdir/libusb-win32-device-bin-$libusb_version/lib/gcc "
```

- Configure

```
../$archivedir/configure \
    --prefix=$installdir \
    --datadir=$installdir/doc \
    --mandir=$installdir/man \
    --infodir=$installdir/info \
    2>&1 | tee avarice-configure.log
```

- Make

```
make all install 2>&1 | tee avarice-make.log
```

- **SimulAVR**

  - Open source code package.
  - Configure and build in a directory outside of the source code tree.
  - Set PATH, in order:
    - <MikTex executables>
    - /usr/local/bin
    - /usr/bin
    - /bin
    - <install directory>/bin
  - Configure

```
export LDFLAGS="-static"
../$archivedir/configure \
    --prefix=$installdir \
    --datadir=$installdir \
    --disable-tests \
    --disable-versioned-doc \
    2>&1 | tee simulavr-configure.log
```

  - Make

```
make -k all install 2>&1 | tee simulavr-make.log
make pdf install-pdf 2>&1 | tee simulavr-pdf-make.log
```

5.5 Canadian Cross Builds

It is also possible to build avr-gcc for host Windows on a Linux build system. Suppose you have installed a `i686-w64-mingw32-gcc` toolchain that can compile code to run on `host=i686-w64-mingw32`. Then the steps to build a toolchain for Windows are:

1. Build and install the AVR toolchain for the Linux build machine as explained above. Make sure that running the command

   ```
   avr-gcc --version
   ```

   prints the compiler version according to the used GCC sources. The native AVR cross compiler is required during configure and to build the AVR target libraries like libgcc. Similarly, the version of the found AVR Binutils programs must match the version of the used Binutils sources.

2. Determine the name of the `--build` platform like `x86_64-pc-linux-gnu`, for example by running the `config.guess` script as shipped with the top level GCC sources (and also with Binutils sources, and AVR-LibC sources after `./bootstrap`).

3. Use different build and install directories, e.g. `./build/binutils-<version>-avr-mingw32` to build Binutils and `--prefix=$PREFIX-mingw32` as install path.

4. Configure, build and install **Binutils** and **GCC** like for the native build, but add the following configure options:

   ```
   --build=x86_64-pc-linux-gnu --host=i686-w64-mingw32
   ```

   This assumes that the required host libraries like GMP are being built in one go with the compiler. This is accomplished by running the `contrib/download_prerequisites` script from the toplevel GCC sources, just like with the native build.

5. There is no need to build **AVR-LibC** again because it is a pure target library. It can be installed by means of running

   ```
   $ # in ./build/avr-libc-<version>
   $ make install prefix=$PREFIX-mingw32
   ```

In order to "install" the toolchain on Windows, the canadian cross installed in `$PREFIX-mingw32` can be moved to the desired location on the Windows computer. The compiler can be used by calling it by its absolute path, or by adding the `$PREFIX-mingw32/bin` directory to the `PATH` environment variable.

5.6 Using Git

Most of the sources of the projects above are now managed with the `git` distributed version-control tools. When you want to build from the newest development branch, you can clone the repo, like with

```
$ git clone <repo> [dirname]
```

Replace `<repo>` with the URL of the Git repository, e.g. `https://github.com/avrdudes/avr-libc.git` for AVR-LibC. Notice that when building AVR-LibC from the repo source, you have to run `./bootstrap` from the top level AVR-LibC sources prior to `configure`.

Useful options for git clone:

`dirname`
: Specify an optional directory name for the cloned repository, like:

  ```
  $ git clone https://github.com/avrdudes/avr-libc.git ./source/avr-libc-main
  ```

  Without `dirname`, the name of the git file like `avr-libc` is used.

`--depth 1`
: An ordinary clone will clone the complete repository with all its branches and their history. To speed up the cloning and save some disc space, you can just clone the top of the history to some depth.

`--branch` *`branch`*
: The default branch is the head of the latest development, which is `master` for GCC and Binutils, and `main` for AVR-LibC.

  When you want a different ref, like GCC's `releases/gcc-14` for the head of the GCC v14 branch, or `releases/gcc-14.1.0` for the GCC v14.1 release tag, then you can specify that as *`branch`*. To see the available refs, you can use

  ```
  $ git ls-remote <repo>
  ```

6 Data in Program Space

6.1 Introduction

So you have some constant data and you're running out of room to store it? Many AVRs have limited amount of RAM in which to store data, but may have more Flash space available. The AVR is a Harvard architecture processor, where Flash is used for the program, RAM is used for data, and they each have separate address spaces. It is a challenge to get constant data to be stored in the Program Space, and to retrieve that data to use it in the AVR application.

The problem is exacerbated by the fact that the C Language was not designed for Harvard architectures, it was designed for Von Neumann architectures where code and data exist in the same address space. This means that any compiler for a Harvard architecture processor, like the AVR, has to use other means to operate with separate address spaces.

GCC has a special keyword, `__attribute__` that is used to attach different attributes to things such as function declarations, variables, and types. This keyword is followed by an attribute specification in double parentheses. In AVR GCC, there is a special attribute called `progmem`. This attribute is used on data definitions, and tells the compiler to place the data in the Program Memory (Flash).

AVR-LibC provides a simple macro PROGMEM that is defined as the attribute syntax of GCC with the `__progmem__` attribute. The PROGMEM macro is defined in the <avr/pgmspace.h> system header which also provides macros and inline functions to access such data.

An alternative approach is taken by named address-spaces like __flash and __flashx as proposed by the N1275 draft to the ISO/IEC DTR 18037 "Embedded C" specification. Named address-spaces are supported in `avr-gcc` since v4.7 (__flashx since `avr-gcc` v15) as part of the GNU-C99 language dialect (`-std=gnu99` and up), see the `avr-gcc documentation`.

AVRs with a linear address space

There are AVR devices that provide a linear address space where the program memory is seen in the RAM address space and can be accessed with `LD` instructions. The respective device families are:

- Devices from the AVRrc core in `avrtiny`.
- AVR16 and AVR32 devices, and devices from the 0-series, 1-series, 2-series in `avrxmega3`.
- AVR64 devices in `avrxmega2` and AVR128 devices in `avrxmega4`. For these devices, only a 32 KiB portion of the program flash is visible in the RAM address space. And only *without* `-mrodata-in-ram` will `.rodata` be located in flash.

In all of these cases, complications like PROGMEM or __flash are not needed, though they are working as usual.

6.2 Why is GCC putting const Data into RAM to begin with?

From a technical point of view, GCC is putting constant data in static storage[1] into the .rodata input section, as opposed to non-const data which is put into `.data` or `.bss`. But the question is then: Why is the linker (script) putting the `.rodata` sections into RAM? In order to better understand this, take the following code:

```
#include <stdbool.h>

extern const char c_one;
extern char c_two;

bool is_one (const char *pc)
{
    return *pc == '1';
}

int test1 (void)
{
    if (is_one (&c_one))
        return 1;
    else if (is_one (&c_two))
        return 2;
    else
        return 0;
}
```

This is a completely valid C99 compilation unit.

Function `is_one` takes a `const char*` pointer argument because it is just reading through `pc` and does not modify the pointed-to object. Without the `const` qualifier for the pointed-to object it was not possible to use the function with constant pointers like `&c_one`, because the code would no more be const-correct.

Moreover, it is completely fine to pass the address of a non-const object like `&c_two` to a function that won't change the pointed-to object, and hence takes a pointer-to-const.

The big question in now: What assembly / machine code should a compiler generate for `is_one()`?

- AVR GCC is using `LD(pc)` in order to read `*pc`:

  ```
  is_one:
      movw r30, r24  ; move pc from r25:r24 to Z
      ldi  r24, 1    ; return value := true
      ld   r25, Z    ; r25 := *Z using LD
      cpi  r25, '1'  ; is r25 == '1' ?
      breq .L2       ; yes: then goto return
      ldi  r24, 0    ; no:  then return value := false
  .L2:
      ret            ; return value (r24)
  ```

  This works when `c_one` and `c_two` are located in RAM[2], so that the `LD` instruction can be used.

- A different approach would be to use `LD(pc)` when `*pc` is located in RAM, and `LPM(pc)` when `*pc` is located in flash; something like:

  ```
  if is_ram_pointer(pc)
      r25 = LD(pc)
  else
      r25 = LPM(pc)
  ```

  The drawbacks are obvious: Such code is expensive, because it has to discriminate at run-time whether `pc` points to RAM or to flash. Plus, there must be some means to tell which kind of pointer `pc` actually is. For example, the high bit of the address could be used to encode the information.

  This approach is taken by `avr-gcc`'s named address-space __memx, which uses 24-bit pointers and encodes the information in the high byte.

So when the compiler takes the first approach of always using `LD`, what will happen when we put `is_one` in PROGMEM?

The code will just not work![3]

To use `is_one()` for pointers to RAM as well as pointers to progmem (or `__flash` for that matter), `is_one()` needs a second argument that tells which kind of pointer is being passed, and it has to adjust the code accordingly; something like:

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

bool is_one (const char *pc, bool is_ram_addr)
{
    char c = is_ram_addr
        ? pgm_read_char (pc)
        : *pc;

    return c == '1';
}
```

Notes

1. In C++, const static storage data might be written to. For example, in

   ```
   volatile int vi;
   const int i2 = vi;
   ```

   the variable `i2` is read-only for the C++ program, but `i2` must not be put into `.rodata` because it cannot be initialized at load-time. Due to its initializer that is not computable at load-time, `i2` has to be put into RAM and will be initialized (written to) at run-time by the startup code. `avr-g++` will diagnose when an attempt is made to put `i2` in PROGMEM.

2. More precisely, these variables have to be located in the RAM *address space* for the code to work. For example, some AVR devices see (a part of) the program memory in the RAM address space, and hence can use the LD instruction to access program memory.
   For example, an ATmega3208 sees the program memory range of 0x0...0x7fff in the RAM address space at addresses 0x4000...0xbfff. So all the linker script has to do is to provide an appropriate VMA of 0x4000+LMA for `.rodata` objects. This is the case for devices from the `avrxmega3` and `avrtiny` families. Since `GCC v14` / Binutils v2.42 it is also the case for `AVR64` and `AVR128` devices when they use the default `avrxmega2_flmap` or `avrxmega4_flmap` emulation, i.e. without `-mrodata-in-ram`.

3. The code does actually work for Reduced Tiny devices because the compiler is implementing attribute `progmem` in a different way for the reduced core (AVRrc). See the `GCC documentation on progmem`.

6.3 Storing and Retrieving Data in the Program Space

6.3.1 With Attribute PROGMEM and pgm_read() Functions

Let's say you have some global data:

```
char mydata[2][8] =
{
    { 2, 3, 5,  7, 11, 13, 17, 19 },
    { 1, 4, 9, 16, 25, 36, 49, 64 }
};
```

and later in your code you access this data in a function and store a single byte into a variable `value` like so:

```
char value = mydata[i][j];
```

Now you want to store your data in Program Memory. Use the PROGMEM macro found in `<avr/pgmspace.h>` and put it after the declaration of the variable, but before the initializer, like so:

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

const char mydata[2][8] PROGMEM =
{
    { 2, 3, 5,  7, 11, 13, 17, 19 },
    { 1, 4, 9, 16, 25, 36, 49, 64 }
};
```

That's it! Now your data is in the Program Space. You can compile, link, and check the map file to verify that `mydata` is placed in the correct section.

Now that your data resides in the Program Space, your code to access (read) the data will no longer work. The code that gets generated will retrieve the data that is located at the address of the `mydata` array, plus offsets indexed by the `i` and `j` variables. However, the final address that is calculated where to the retrieve the data points to the Data Space! Not the Program Space where the data is actually located. It is likely that you will be retrieving some garbage. The problem is that `avr-gcc` does not intrinsically know that the data resides in the Program Space.

The solution is fairly simple. The "rule of thumb" for accessing data stored in the Program Space is to access the data as you normally would (as if the variable is stored in Data Space), like so:

```
char value = mydata[i][j];
```

then take the address of the data:

```
... &(mydata[i][j]);
```

then use the appropriate `pgm_read_*` function, and the address of your data becomes the parameter to that function:

```
char value = pgm_read_char (&(mydata[i][j]));
```

The `pgm_read_*` functions take an address that points to the Program Space, and retrieves the data that is stored at that address. This is why you take the address of the offset into the array. This address becomes the parameter to the function so it can generate the correct code to retrieve the data from the Program Space. There are different `pgm_read_*` functions to read different types of data at the address given.

6.3.2 With Named Address-Space __flash

The same code in terms of address-space __flash is:

```
const __flash char mydata[2][8] =
{
    { 2, 3, 5,  7, 11, 13, 17, 19 },
    { 1, 4, 9, 16, 25, 36, 49, 64 }
};
```

In order to read from `mydata`, no special code is required:

```
char value = mydata[i][j];
```

You can also pass qualified addresses around, like in

```
char get_first (const __flash char *array)
{
    return array[0];
}

char get_mydata_nth_first (uint8_t n)
{
    return get_first (mydata[n]);
}
```

6.4 PROGMEM and __flash: The Differences

So what's are the ups and down of using PROGMEM or __flash ?

1. Named address-spaces are only available in GNU-C99 and up, and with avr-gcc v4.7 or newer. To date, GCC does not support named address-spaces in C++, whereas the pgm_read functions work in C++ just as well as in C.

2. Qualifiers like __flash are easier to port. For example, avr-gcc does not support named address-spaces for the Reduced Tiny devices like ATtiny10. This can be handled with the builtin macro __FLASH:

   ```
   #ifndef __FLASH
   #define __flash // empty
   #endif

   // Code that uses __flash
   ```

   (Notice that on Reduced Tiny, section .rodata is located in program memory as opposed to many other AVR cores that have .rodata in RAM. Hence *not* using __flash does *not* cause a loss of performance.)

3. __flash is transparent to the compiler, for example an access like value = mydata[1][1] can be optimized to value = 4, whereas accesses through pgm_read cannot be optimized.

4. Qualifiers like __flash can be used in pointer targets, like in

   ```
   char read_c (const __flash char *c)
   {
       return *c; // Compiles to LPM
   }
   ```

   whereas this is not possible for PROGMEM, which is an attribute in GCC and not a qualifier.

5. The analog of PSTR for address-spaces is FSTR or FXSTR. As of v14, avr-gcc still rejects to put local static compound literals into an address-space (GCC PR84163). While constructs like

   ```
   const __flash char *ptext = FSTR ("Text");
   ```

   will work and are qualifier-correct, i.e. avr-gcc will not raise a diagnostic with -Waddr-space-convert.

6. PSTR cannot be used on global scope, whereas a similar construct with address-spaces is possible: Take for example the code discussed in the next section that declares an array to string literals. With address-space we can write:

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

   const __flash char* const __flash string_table[] =
   {
       FLIT("String 1"),
       FLIT("String 2"),
       FLIT("String 3")
   };
   ```

   Notice that the __flash left of the * refers to strings pointed-to by string_table, whereas the __flash right of the * refers to string_table itself, i.e. the string literals as well as the table are in __flash.

7. For functions that take a pointer to program space like strcpy_P, there are address-space correct variants like strcpy_F and strcpy_FX that work nicely with -Waddr-space-convert.

6.5 Storing and Retrieving Strings in the Program Space

6.5.1 With __flash

For a solution with the __flash address-space, see the section above, the FLIT examples, or have a look at the FAQ: How to put an array of strings completely in ROM?

6.5.2 With PROGMEM

Now that you can successfully store and retrieve simple data from Program Space you want to store and retrieve strings from Program Space. And specifically you want to store and array of strings to Program Space. So you start off with your array, like so:

```
const char* const string_table[] =
{
    "String 1",
    "String 2",
    "String 3"
};
```

and then you add your PROGMEM macro to the end of the declaration:

```
const char* const string_table[] PROGMEM =
{
    "String 1",
    "String 2",
    "String 3"
};
```

Right? WRONG!

Unfortunately, with GCC attributes, they affect only the declaration that they are attached to. So in this case, we successfully put the `string_table` variable, the array itself, in the Program Space. This DOES NOT put the actual strings themselves into Program Space. At this point, the strings are still in the Data Space, which is probably not what you want.

In order to put the strings in Program Space, you have to have explicit declarations for each string, and put each string in Program Space:

```
static const char string_1[] PROGMEM = "String 1";
static const char string_2[] PROGMEM = "String 2";
static const char string_3[] PROGMEM = "String 3";
```

Then use the new symbols in your table, like so:

```
const char* const string_table[] PROGMEM =
{
    string_1,
    string_2,
    string_3
};
```

Now this has the effect of putting `string_table` in Program Space, where `string_table` is an array of pointers to characters (strings), where each pointer is a pointer to the Program Space, where each string is also stored.

Retrieving the strings are a different matter. You probably don't want to pull the string out of Program Space, byte by byte, using the `pgm_read_byte()` macro or `prgm_read_char()` function. There are other functions declared in the <avr/pgmspace.h> header file that work with strings that are stored in the Program Space.

For example, if you want to copy the string from Program Space to a buffer in RAM (like an automatic variable inside a function, that is allocated on the stack), you can do this:

```
void foo (void)
{
    char buffer[10];

    for (uint8_t i = 0; i < 3; i++)
    {
        strcpy_P (buffer, (const char*) pgm_read_ptr (& string_table[i]));

        // Display buffer on LCD.
    }
}
```

Here, the `string_table` array is stored in Program Space, so we access it normally, as if it was stored in Data Space, then take the address of the location we want to access, and use the address as a parameter to `pgm_read_ptr`. We use the `pgm_read_ptr` macro to read the string pointer out of the `string_table` array. Remember that a pointer is 16-bits, or word size. The pgm_read_ptr macro will return a `void*`. This pointer is an address in Program Space pointing to the string that we want to copy. This pointer is then used as a parameter to the function strcpy_P. The function strcpy_P is just like the regular `strcpy` function, except that it copies a string from Program Space (the second parameter) to a buffer in the Data Space (the first parameter).

There are many string functions available that work with strings located in Program Space. All of these special string functions have a suffix of `_P` in the function name, and are declared in the <avr/pgmspace.h> header file.

7 Memory Sections

Section are used to organize code and data of a program on the binary level.

The (compiler-generated) assembly code assigns code, data and other entities like debug information to so called input sections. These sections serve as input to the linker, which bundles similar sections together to output sections like `.text` and `.data` according to rules defined in the linker description file.

The final ELF binary is then used by programming tools like `avrdude`, simulators, debuggers and other programs, for example programs from the GNU Binutils family like `avr-size`, `avr-objdump` and `avr-readelf`.

Sections may have extra properties like section alignment, section flags, section type and rules to locate them or to assign them to memory regions.

7.1 Concepts

7.1.1 Named Sections

Named sections are sections that can be referred to by their name. The name and other properties can be provided with the `.section` directive like in

```
.section name, "flags", @type
```

or with the `.pushsection` directive, which directs the assembler to assemble the following code into the named section.

An example of a section that is not referred to by its name is the COMMON section. In order to put an object in that section, special directives like `.comm name,size` or `.lcomm name,size` have to be used.

Directives like `.text` are basically the same like `.section .text`, where the assembler assumes appropriate section flags and type; same for directives `.data` and `.bss`.

7.1.1.1 Section Flags

The *section flags* can be specified with the `.section` and `.pushsection` directives, see section type for an example. Section flags of output sections can be specified in the linker description file, and the linker implements heuristics to determine the section flags of output sections from the various input section that go into it.

Table 8 Section Flags

Flag	Meaning
`a`	The section will be **a**llocated, i.e. it occupies space on the target hardware
`w`	The section contains data that can be **w**ritten at run-time. Sections that only contain read-only entities don't have the `w` flag set
`x`	The section contains e**x**ecutable code, though the section may also contain non-executable objects
`M`	A **m**ergeable section
`S`	A **s**tring section
`G`	A section **g**roup, like used with `comdat` objects

The last three flags are listed for completeness. They are used by the compiler, for example for header-only C++ modules and to ensure that multiple instantiations of the same template in different compilation units does occur at most once in the executable file.

7.1.1.2 Section Type

The *section type* can be specified with the `.section` and `.pushsection` directives, like in

```
.section .text.myfunc,"ax",@progbits
.pushsection ".data.myvar", "a", "progbits"
```

On ELF level, the section type is stored in the section header like `Elf32_Shdr.sh_type = SHT_PROGBITS`.

Table 9 Section Types

Type	Meaning
`@progbits`	The section contains data that will be loaded to the target, like objects in the `.text` and `.data` sections.
`@nobits`	The section does not contain data that needs to be transferred to the target device, like data in the `.bss` and `.noinit` sections. The section still occupies space on the target.
`@note`	The section is a note, like for example the .note.gnu.avr.deviceinfo section.

7.1.1.3 Section Alignment

The *alignment* of a section is the maximum over the alignments of the objects in the section.

7.1.1.4 Subsections

Subsections are compartments of named sections and are introduced with the `.subsection` directive. Subsections are located in order of increasing index in their input section. The default subsection after switching to a new section is subsection 0.

Note

A common misconception is that a section like `.text.module.func` were a subsection of `.text.module`. This is not the case. These two sections are independent, and there is no subset relation. The sections may have different flags and type, and they may be assigned to different output sections.

7.1.2 Orphan Sections

Orphan sections are sections that are not mentioned in the linker description file. When an input section is orphan, then the GNU linker implicitly generates an output section of the same name. The linker implements various heuristics to determine sections flags, section type and location of orphaned sections. One use of orphan sections is to locate code to a fixed address.

Like for any other output section, the start address can be specified by means of linking with
`-Wl,--section-start,`*`secname=address`*

7.1.3 LMA: Load Memory Address

The LMA of an object is the address where a loader like `avrdude` puts the object when the binary is being uploaded to the target device.

7.1.4 VMA: Virtual Memory Address

The VMA is the address of an object as used by the running program.

VMA and LMA may be different: Suppose a small ATmega8 program with executable code that extends from byte address 0x0 to 0x20f, and one variable `my_var` in static storage. The default linker script puts the content of the `.data` output section after the `.text` output section and into the `text` segment. The startup code then copies `my_data` from its LMA location beginning at 0x210 to its VMA location beginning at 0x800060, because C/C++ requires that all data in static storage must have been initialized when `main` is entered.

The internal SRAM of ATmega8 starts at RAM address 0x60, which is offset by 0x800000 in order to linearize the address space (VMA 0x60 is a flash address). The AVR program only ever uses the lower 16 bits of VMAs in static storage so that the offset of 0x800000 is masked out. But code like `"LDI r24,hh8(my_data)"` actually sets R24 to 0x80 and reveals that `my_data` is an object located in RAM.

7.2 The Linker Script: Building Blocks

The linker description file is the central hub to channel functions and static storage objects of a program to the various memory spaces and address ranges of a device.

7.2.1 Input Sections and Output Sections

Input sections are sections that are inputs to the linker. Functions and static variables but also additional notes and debug information are assigned to different input sections by means of assembler directives like `.section` or `.text`. The linker takes all these sections and assigns them to output sections as specified in the linker script.

Output sections are defined in the linker description file. Contrary to the unlimited number of input sections a program can come up with, there is only a handful of output sections like `.text` and `.data`, that roughly correspond to the memory spaces of the target device.

One step in the final link is to *locate* the sections, that is the linker/locator determines at which memory location to put the output sections, and how to arrange the many input sections within their assigned output section. *Locating* means that the linker assigns Load Memory Addresses — addresses as used by a loader like `avrdude` — and Virtual Memory Addresses, which are the addresses as used by the running program.

While it is possible to directly assign LMAs and VMAs to output sections in the linker script, the default linker scripts provided by Binutils assign *memory regions* (aka. *memory segments*) to the output sections. This has some advantages like a linker script that is easier to maintain. An output sections can be assigned to more than one memory region. For example, non-zero data in static storage (`.data`) goes to

1. the `data` region (VMA), because such variables occupy RAM which has to be allocated
2. the `text` region (LMA), because the initializers for such data has to be kept in some non-volatile memory (program ROM), so that the startup code can initialize that data so that the variables have their expected initial values when `main()` is entered.

The `SECTIONS{}` portion of a linker script models the input and output section, and it assigns the output section to the memory regions defined in the `MEMORY{}` part.

7.2.2 Memory Regions

The *memory regions* defined in the default linker script model and correspond to the different kinds of memories of a device.

Table 10 Memory Regions of the Default Linker Script

Region	Virtual Address[1]	Flags	Purpose
`text`	`0`[2]	`rx`	Executable code, vector table, data in PROGMEM, __flash, __flashx and __memx, startup code, linker stubs, initializers for `.data`
`data`	`0x800000`[2]	`rw`	Data in static storage
`rodata`[3]	`0xa00000`[2]	`r`	Read-only data in static storage
`eeprom`	`0x810000`	`rw`	EEPROM data
`fuse`	`0x820000`	`rw`	Fuse bytes
`lock`	`0x830000`	`rw`	Lock bytes
`signature`	`0x840000`	`rw`	Device signature
`user_signatures`	`0x850000`	`rw`	User signature

Notes

1. The VMAs for regions other than `text` are offset in order to linearize the non-linear memory address space

of the AVR Harvard architecture. The target code only ever uses the lower 16 bits of the VMA to access objects in non-`text` regions.

2. The addresses for regions `text`, `data and` rodata are actually defined as symbols like `__TEXT_REGION_ORIGIN__`, so that they can be adjusted by means of, say `-Wl,--defsym,__DATA_REGION_ORIGIN__=0x800060`. Same applies for the lengths of all the regions, which is `__`*NAME*`_REGION_LENGTH__` for region *name*.

3. The `rodata` region is only present in the `avrxmega2_flmap` and `avrxmega4_flmap` emulations, which is the case for Binutils since v2.42 for the AVR64 and AVR128 devices without `-mrodata-in-ram`.

7.3 Output Sections of the Default Linker Script

This section describes the various output sections defined in the default linker description files.

Table 11 Output Sections and Memory Regions

Output Section	Purpose	Memory Region	
		LMA	VMA
`.text`	Executable code, data in progmem	text	text
`.data`	Non-zero data in static storage	text	data
`.bss`	Zero data in static storage	—	data
`.noinit`	Non-initialized data in static storage	—	data
`.rodata`[1]	Read-only data in static storage	text	LMA + offset[3]
`.rodata`[2]	Read-only data in static storage	0x8000 $*$ `__flmap`[4]	rodata
`.eeprom`	Data in EEPROM	Note[5]	eeprom
`.fuse`	Fuse bytes		fuse
`.lock`	Lock bytes		lock
`.signature`	Signature bytes		signature
	User signature bytes		user_signatures

Notes

1. On `avrxmega3` and `avrtiny` devices.
2. On `AVR64` and `AVR128` devices without `-mrodata-in-ram`.
3. With an offset `__RODATA_PM_OFFSET__` of 0x4000 or 0x8000 depending on the device.
4. The value of symbol `__flmap` defaults to the last 32 KiB block of program memory, see the GCC `v14 release notes`.
5. The LMA actually equals the VMA, but is unused. The flash loader like avrdude knows where to put the data.

7.3.1 The .text Output Section

The `.text` output section contains the actual machine instructions which make up the program, but also additional code like jump tables and lookup tables placed in program memory with the PROGMEM attribute.

The `.text` output section contains the input sections described below. Input sections that are not used by the tools are omitted. A $*$ wildcard stands for any sequence of characters, including empty ones, that are valid in a section name.

.vectors
The .vectors sections contains the interrupt vector table which consists of jumps to weakly defined labels: To __init for the first entry at index 0, and to __vector_N for the entry at index $N \geq 1$. The default value for __vector_N is __bad_interrupt, which jumps to weakly defined __vector_default, which jumps to __vectors, which is the start of the .vectors section.
Implementing an interrupt service routine (ISR) is performed with the help of the ISR macro in C/C++ code.

.progmem.data
.progmem.data.*
.progmem.gcc.*
This section is used for read-only data declared with attribute PROGMEM, and for data in address-space __flash.
The compiler assumes that the .progmem.data sections are located in the lower 64 KiB of program memory. When it does not fit in the lower 64 KiB block, then the program reads garbage except address-space __flashx or pgm_read_*_far is used. In that case however, data can be located in the .progmemx.data section which does not require to be located in the lower program memory.

.trampolines
Linker stubs for indirect jumps and calls on devices with more than 128 KiB of program memory. This section must be located in the same 128 KiB block like the interrupt vector table. For some background on linker stubs, see the GCC documentation on EIND.

.text
.text.*
Executable code. This is where almost all of the executable code of an application will go.

.ctors
.dtors
Tables with addresses of static constructors and destructors, like C++ static constructors and functions declared with attribute constructor.

The .init*N* Sections
These sections are used to hold the startup code from reset up through the start of main().
The .init$N$ sections are executed in order from 0 to 9: The code from one init section falls through to the next higher init section. This is the reason for why code in these sections must be naked (more precisely, it must not contain return instructions), and why code in these sections must never be called explicitly.
When several modules put code in the same init section, the order of execution is not specified.

Table 12 The .init*N* Sections

Section	Performs	Hosted By	Symbol[1]
.init0	Weakly defines the __init label which is the jump target of the first vector in the interrupt vector table. When the user defines the __init() function, it will be jumped to instead.	AVR-LibC[2]	
.init1	Unused	—	
.init2	•Clears __zero_reg__ •Initializes EIND to hh8(pm(__vectors)) on devices that have it •Initializes RAMPX, RAMPY, RAMPZ and RAMPD on devices that have all of them	AVR-LibC	
.init2	Initializes the stack pointer to the value of weak symbol __stack, which has a default value of RAMEND as defined in avr/io.h. On devices that have it, set SPLIM to one less than the value of symbol __heap_start	AVR-LibC	__init_sp[3]
.init3	Initializes the NVMCTRLB.FLMAP bit-field on devices that have it, except when -mrodata-in-ram is specified	AVR-LibC	__do_flmap_init[v2.2]

Section	Performs	Hosted By	Symbol[1]
.init3	Sets the CPUINT_CTRLA.CPUINT_CVT bit provided a compact vector table is in effect (-mcvt).	AVR-LibC	__init_cvt[v2.3]
.init3	Paints the RAM from __heap_start up to RAMEND in preparation for _get_ram_unused().	AVR-LibC	__init_ram_color[v2.3]
.init4	Initializes data in static storage: Initializes .data and clears .bss	libgcc	__do_copy_data __do_clear_bss
.init5	Unused	—	
.init6	Run static C++ constructors and functions defined with __attribute__↩ ((constructor)).	libgcc	__do_global_ctors
.init7	Unused	—	
.init8	Unused	—	
.init9	Calls main and then jumps to exit	AVR-LibC	__call_main[3]

Notes

1. Code in the .init3, .init4 and .init6 sections is optional; it will only be present when there is something to do. This will be tracked by the compiler — or has to be tracked by the assembly programmer — which pulls in the code from the respective library by means of the mentioned symbols, e.g. by linking with -Wl,-u,__do_flmap_init or by means of

   ```
   .global __do_copy_data
   ```

   Conversely, when the respective code is not desired for some reason, the symbol can be satisfied by defining it with, say, -Wl,--defsym,__do_copy_data=0 so that the code is not pulled in any more.

2. The code is provided by gcrt1.S.

3. Since AVR-LibC v2.3, the startup code pulls in the respective code from libmcu.a. Prior to that, the code was located in gcrt1.S.

The .fini*N* Sections

Shutdown code. These sections are used to hold the exit code executed after return from main() or a call to exit().

The .fini*N* sections are executed in descending order from 9 to 0 in a fallthrough manner.

Table 13 The .fini*N* Sections

Section	Performs	Hosted By	Symbol
.fini9	Defines _exit and weakly defines the exit label	libgcc	
.fini8	Run functions registered with atexit()	AVR-LibC	
.fini7	Unused	—	
.fini6	Run static C++ destructors and functions defined with __attribute__↩ ((destructor))	libgcc	__do_global_dtors
.fini5	Unused	—	
.fini4	Weakly defines __abort, used by abort()	AVR-LibC	
.fini3...1	Unused	—	
.fini0	Globally disables interrupts and enters an infinite loop to label __stop_program	libgcc	

It is unlikely that ordinary code uses the fini sections. When there are no static destructors and atexit() is not used, then the respective code is not pulled in from the libraries, and the fini code just consumes four bytes: a CLI and a RJMP to itself. Common use cases of fini code is when running the GCC test suite where it reduces fallout, and in simulators to determine (un)orderly termination of a simulated program.

.progmemx.*

Read-only data in program memory without the requirement that it must reside in the lower 64 KiB. The compiler uses this section for data in the named address-spaces __flashx and __memx. Data can be accessed with pgm_read_*_far when it is not in a named address-space:

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

const __memx int array1[] = { 1, 4, 9, 16, 25, 36 };

PROGMEM_FAR
const int array2[] = { 2, 3, 5, 7, 11, 13, 17 };

int add (uint8_t id1, uint8_t id2)
{
    uint_farptr_t p_array2 = pgm_get_far_address (array2);
    int val2 = pgm_read_int_far (p_array2 + sizeof(int) * id2);

    return val2 + array1[id1];
}
```

For data in address-space __flashx there are support functions in <avr/flash.h>.

The `.progmemx.*` input section pattern was added in Binutils v2.30. In older versions, there was a catch all `.progmem*` pattern for all data in program space, which will cause problems when progmem data pushes the `.trampolines` out of the lower 128 KiB segment.

`.jumptables*`
Used to place jump tables in some cases.

7.3.2 The .data Output Section

This section contains data in static storage which has an initializer that is not all zeroes. This includes the following input sections:

`.data*`
Read-write data

`.rodata*`
Read-only data. These input sections are only included on devices that host read-only data in RAM.

It is possible to tell the linker the SRAM address of the beginning of the `.data` section. This is accomplished by linking with

```
avr-gcc ... -Tdata addr -Wl,--defsym,__DATA_REGION_START__=addr
```

Note that `addr` must be offset by adding 0x800000 to the real SRAM address so that the linker knows that the address is in the SRAM memory segment. Thus, if you want the `.data` section to start at `0x1100`, pass `0x801100` as the address to the linker.

Note

When using malloc() in the application (which could even happen inside library calls), additional adjustments are required.

7.3.3 The .bss Output Section

Data in static storage that will be zeroed by the startup code. These are data objects without explicit initializer, and data objects with initializers that are all zeroes.

Input sections are `.bss*` and `COMMON`. Common symbols are defined with directives `.comm` or `.lcomm`.

7.3.4 The .noinit Output Section

Data objects in static storage that should not be initialized by the startup code. As the C/C++ standard requires that *all* data in static storage is initialized — which includes data without explicit initializer, which will be initialized to all zeroes — such objects have to be put into section `.noinit` by hand:

```
__attribute__ ((section (".noinit")))
int foo;
```

The only input section in this output section is `.noinit`. Only data without initializer can be put in this section.

7.3.5 The .rodata Output Section

This section contains read-only data in static storage from `.rodata*` input sections. This output section is only present for devices where read-only data remains in program memory, which are the devices where (parts of) the program memory are visible in the RAM address space. This is currently the case for the emulations `avrtiny`, `avrxmega3`, `avrxmega2_flmap` and `avrxmega4_flmap`.

7.3.6 The .eeprom Output Section

This is where EEPROM variables are stored, for example variables declared with the `EEMEM` attribute. The only input section (pattern) is `.eeprom*`.

7.3.7 The .fuse, .lock and .signature Output Sections

These sections contain fuse bytes, lock bytes and device signature bytes, respectively. The respective input section patterns are `.fuse*`, `.lock*` and `.signature*`.

7.3.8 The .note.gnu.avr.deviceinfo Section

This section is actually *not mentioned* in the default linker script, which means it is an orphan section and hence the respective output section is implicit.

The startup code from AVR-LibC puts device information in that section to be picked up by simulators or tools like `avr-size`, `avr-objdump`, `avr-readelf`, etc,

The section is contained in the ELF file but not loaded onto the target. Source of the device specific information are the device header file and compiler builtin macros. The layout conforms to the standard ELF note section layout and is laid out as follows.

```
#include <elf.h>

typedef struct
{
    Elf32_Word n_namesz;      /* AVR_NOTE_NAME_LEN */
    Elf32_Word n_descsz;      /* size of avr_desc */
    Elf32_Word n_type;        /* 1 - the only AVR note type */
} Elf32_Nhdr;

#define AVR_NOTE_NAME_LEN 4

struct note_gnu_avr_deviceinfo
{
    Elf32_Nhdr nhdr;
```

```
    char note_name[AVR_NOTE_NAME_LEN]; /* = "AVR\0" */

    struct
    {
        Elf32_Word flash_start;
        Elf32_Word flash_size;
        Elf32_Word sram_start;
        Elf32_Word sram_size;
        Elf32_Word eeprom_start;
        Elf32_Word eeprom_size;
        Elf32_Word offset_table_size;
        /* Offset table containing byte offsets into
           string table that immediately follows it.
           index 0: Device name byte offset */
        Elf32_Off offset_table[1];
        /* Standard ELF string table.
           index 0 : NULL
           index 1 : Device name
           index 2 : NULL */
        char strtab[2 + strlen(__AVR_DEVICE_NAME__)];
    } avr_desc;
};
```

The contents of this section can be displayed with

- `avr-objdump -P avr-deviceinfo` *file*, which is supported since Binutils v2.43.
- `avr-readelf -n` *file*, which displays all notes.

7.4 Symbols in the Default Linker Script

Most of the symbols like `main` are defined in the code of the application, but some symbols are defined in the default linker script.

The following two groups of symbols have architecture specific default values and can be adjusted by defining the respective symbol. One use case is to adjust the `.data` region when external RAM should be used. Most of the symbols get their default value from the startup code according to the memory layout of the respective device.

__*name*_REGION_ORIGIN__
: Describes the physical properties of memory region *name*, where *name* is one of `TEXT` or `DATA`. The address is a VMA and offset as explained above.
The linker script only supplies a default for the symbol values when they have not been defined by other means, like for example in the startup code or by `--defsym`. For example, to let the code start at address `0x100`, one can link with

```
avr-gcc ... -Ttext=0x100 -Wl,--defsym,__TEXT_REGION_ORIGIN__=0x100
```

Notice that `__DATA_REGION_ORIGIN__` was only introduced since Binutils `v2.40`. Prior to that, a core specific default value was used for the beginning of the data region, and thus defining this symbol had no effect.

__*name*_REGION_LENGTH__
: Describes the physical properties of memory region *name*, where *name* is one of: `TEXT`, `DATA`, `EEPROM`, `LOCK`, `FUSE`, `SIGNATURE` or `USER_SIGNATURE`.
Only a default is supplied when the symbol is not yet defined by other means. Most of these symbols are weakly defined in the startup code.

The following symbols are provided by the default linker script. The application can read them to infer properties of the binary.

__data_start

__data_end
Start and (one past the) end VMA address of the .data section in RAM.

__data_load_start
__data_load_end
Start and (one past the) end LMA address of the .data section initializers located in program memory. Used together with the VMA addresses above by the startup code to copy data initializers from program memory to RAM.

__bss_start
__bss_end
Start and (one past the) end VMA address of the .bss section. The startup code clears this part of the RAM.

__rodata_start
__rodata_end
__rodata_load_start
__rodata_load_end
Start and (one past the) end VMA resp. LMA address of the .rodata output section. These symbols are only defined when .rodata is not output to the text region, which is the case for emulations avrxmega2_flmap and avrxmega4_flmap.

__heap_start
One past the last object located in static storage. Immediately follows the .noinit section (which immediately follows .bss, which immediately follows .data). Used by malloc() tunables to determine the start of the heap. On devices that have it, the startup code sets SPLIM to the value of __heap_start-1.

_etext
One byte past the last byte in the .text output section.

Code that computes a checksum over all relevant code and data in program memory has to consider:

- The range from the beginning of the .text section (address 0x0 in the default layout) up to __data_load_end.
- For emulations that have the rodata memory region (avrxmega2_flmap and avrxmega4_flmap), the range from __rodata_load_start to __rodata_load_end has also to be taken into account, too.

7.5 Output Sections and Code Size

The avr-size program (part of Binutils), coming from a Unix background, doesn't account for the .data initialization space added to the .text section, so in order to know how much flash the final program will consume, one needs to add the values for both, .text and .data (but not .bss), while the amount of pre-allocated SRAM is the sum of .data and .bss.

Memory usage and free memory can also be displayed with

```
avr-objdump -P mem-usage code.elf
```

7.6 Using Sections

7.6.1 In C/C++ Code

The following example shows how to read and reset the MCUCR special function register on ATmega328. This SFR holds to reset source like "watchdog reset" or "external reset", and should be read early, prior to the initialization of RAM and execution of static constructors which may take some time. This means the code has to be placed prior to .init4 which initializes static storage, but after .init2 which initializes __zero_reg__. As the code runs

prior to the initialization of static storage, variable `mcucr` must be placed in section `.noinit` so that it won't be overridden by that part of the startup code:

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

__attribute__((section(".noinit")))
uint8_t mcucr;

__attribute__((used, unused, naked, section(".init3")))
static void read_MCUCR (void)
{
    mcucr = MCUCR;
    MCUCR = 0;
}
```

- The `used` attribute tells the compiler that the function is used although it is never called.
- The `unused` attribute tells the compiler that it is fine that the function is unused, and silences respective diagnostics about the seemingly unused functions.
- The `naked` attribute is required because the code is located in an init section. The function *must not have a `RET` statement* because the function is never called. According to the GCC documentation, the only code supported in naked functions is inline assembly, but the code above is simple enough so that GCC can deal with it.

7.6.2 In Assembly Code

Example:

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

.section .init3,"ax",@progbits
    lds     r0, MCUCR

.pushsection .noinit,"a",@nobits
mcucr:
    .type   mcucr, @object
    .size   mcucr, 1
    .space  1
.popsection                     ; Proceed with .init3

    sts     mcucr, r0
    sts     MCUCR, __zero_reg__ ; Initialized in .init2

.text
    .global main
    .type   main, @function
    lds     r24,    mcucr
    clr     r25
    rjmp    putchar
    .size main, .-main
```

- The `"ax"` flags tells that the sections is **a**llocatable (consumes space on the target hardware) and is e**x**ecutable.
- The `@progbits` type tells that the section contains bits that have to be uploaded to the target hardware.

For more details, see the gas user manual on the `.section` directive.

8 Memory Areas and Using malloc()

8.1 Introduction

Many of the devices that are possible targets of AVR-LibC have a minimal amount of RAM. The smallest parts supported by the C environment come with 32 bytes of RAM. This needs to be shared between initialized and uninitialized variables (sections `.data` and `.bss`), the dynamic memory allocator, and the stack that is used for calling subroutines and storing local (automatic) variables.

Also, unlike larger architectures, there is no hardware-supported memory management which could help in separating the mentioned RAM regions from being overwritten by each other.

The standard RAM layout is to place `.data` variables first, from the beginning of the internal RAM, followed by `.bss`. The stack is started from the top of internal RAM, growing downwards. The so-called "heap" available for the dynamic memory allocator will be placed beyond the end of `.bss`. Thus, there's no risk that dynamic memory will ever collide with the RAM variables (unless there were bugs in the implementation of the allocator). There is still a risk that the heap and stack could collide if there are large requirements for either dynamic memory or stack space. The former can even happen if the allocations aren't all that large but dynamic memory allocations get fragmented over time such that new requests don't quite fit into the "holes" of previously freed regions. Large stack space requirements can arise in a C function containing large and/or numerous local variables or when recursively calling functions.

Note

The pictures shown in this document represent typical situations where the RAM locations refer to an ATmega128. The memory addresses used are not displayed in a linear scale.

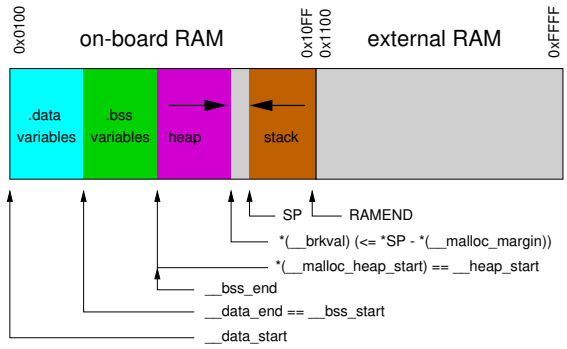

Figure 1 RAM map of a device with internal RAM

On a simple device like a microcontroller, it is a challenge to implement a dynamic memory allocator that is simple enough, so the code size requirements will remain low, yet powerful enough to avoid unnecessary memory fragmentation and to get it all done with reasonably few CPU cycles. Microcontrollers are often low on space and also run at much lower speeds than the typical PC these days.

The memory allocator implemented in AVR-LibC tries to cope with all of these constraints, and offers some tuning options that can be used if there are more resources available than in the default configuration.

8.2 Internal vs. external RAM

Obviously, the constraints are much harder to satisfy in the default configuration where only internal RAM is available. Extreme care must be taken to avoid a stack-heap collision, both by making sure functions aren't nesting too deeply, and don't require too much stack space for local variables, as well as by being cautious with allocating too much dynamic memory.

If external RAM is available, it is strongly recommended to move the heap into the external RAM, regardless of whether or not the variables from the `.data` and `.bss` sections are also going to be located there. The stack should always be kept in internal RAM. Some devices even require this, and in general, internal RAM can be accessed faster since no extra wait states are required. When using dynamic memory allocation and stack and heap are separated in distinct memory areas, this is the safest way to avoid a stack-heap collision.

8.3 Tunables for malloc()

There are a number of variables that can be tuned to adapt the behavior of malloc() to the expected requirements and constraints of the application. Any changes to these tunables should be made before the very first call to malloc(). Note that some library functions might also use dynamic memory (notably those from the <stdio.h>: Standard IO facilities), so make sure the changes will be done early enough in the startup sequence.

The variables `__malloc_heap_start` and `__malloc_heap_end` can be used to restrict the malloc() function to a certain memory region. These variables are statically initialized to point to `__heap_start` and `__heap_end`, respectively, where `__heap_start` is filled in by the linker to point just beyond `.bss`, and `__heap_end` is set to 0 which makes malloc() assume the heap is below the stack.

If the heap is going to be moved to external RAM, `__malloc_heap_end` *must* be adjusted accordingly. This can either be done at run-time, by writing directly to this variable, or it can be done automatically at link-time, by adjusting the value of the symbol `__heap_end`.

The following example shows a linker command to relocate the entire `.data` and `.bss` segments, and the heap to location 0x1100 in external RAM. The heap will extend up to address 0xffff.

```
avr-gcc ... -Wl,--section-start,.data=0x801100,--defsym=__heap_end=0x80ffff ...
```

Note

See explanation for offset 0x800000. See the chapter about using gcc for the `-Wl` options.

The linker manual states that `-Tdata=<x>` is equivalent to `--section-start,.data=<x>`. However, you have to use the latter when the compiler is affected by PR30417. That PR is fixed in GCC v4.9.4, v5.5 and v6.2+.

Symbols `__DATA_REGION_ORIGIN__` and `__DATA_REGION_LENGTH__` should also be adjusted, see Symbols in the Default Linker Script. For the given example, the symbol values would be 0x801100 and 0xef00, respectively.

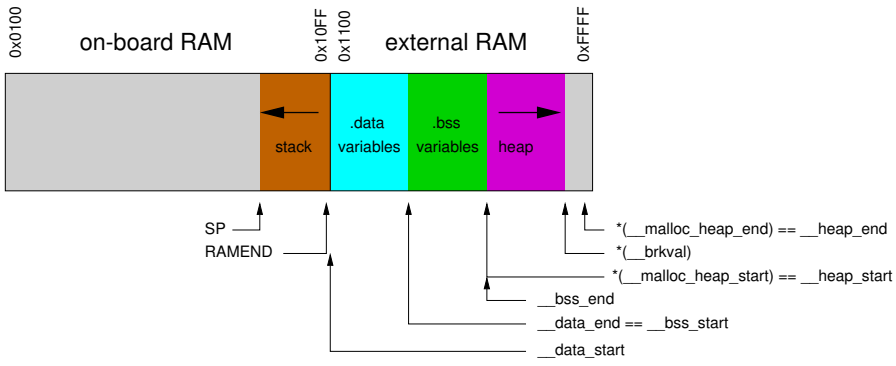

Figure 2 Internal RAM: stack only, external RAM: variables and heap

If dynamic memory should be placed in external RAM, while keeping the variables in internal RAM, something like the following could be used. Note that for demonstration purposes, the assignment of the various regions has not been made adjacent in this example, so there are "holes" below and above the heap in external RAM that remain completely unaccessible by regular variables or dynamic memory allocations (shown in light bisque color in the picture below).

```
avr-gcc ... -Wl,--defsym=__heap_start=0x802000,--defsym=__heap_end=0x803fff ...
```

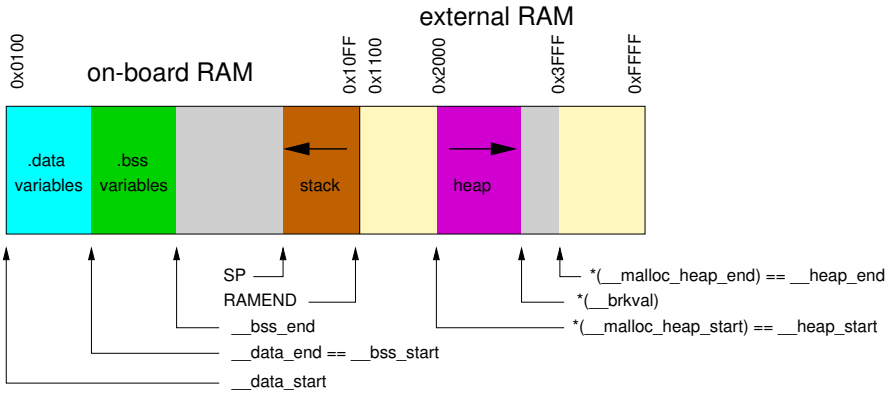

Figure 3 Internal RAM: variables and stack, external RAM: heap

If `__malloc_heap_end` is 0, the allocator attempts to detect the bottom of stack in order to prevent a stack-heap collision when extending the actual size of the heap to gain more space for dynamic memory. It will not try to go beyond the current stack limit, decreased by `__malloc_margin` bytes. Thus, all possible stack frames of interrupt routines that could interrupt the current function, plus all further nested function calls must not require more stack space, or they will risk colliding with the data segment.

The default value of `__malloc_margin` is set to 32.

8.4 Implementation details

Dynamic memory allocation requests will be returned with a two-byte header prepended that records the size of the allocation. This is later used by free(). The returned address points just beyond that header. Thus, if the application accidentally writes before the returned memory region, the internal consistency of the memory allocator is compromised.

The implementation maintains a simple freelist that accounts for memory blocks that have been returned in previous calls to free(). Note that all of this memory is considered to be successfully added to the heap already, so no further checks against stack-heap collisions are done when recycling memory from the freelist.

The freelist itself is not maintained as a separate data structure, but rather by modifying the contents of the freed memory to contain pointers chaining the pieces together. That way, no additional memory is required to maintain this list except for a variable that keeps track of the lowest memory segment available for reallocation. Since both, a chain pointer and the size of the chunk need to be recorded in each chunk, the minimum chunk size on the freelist is four bytes.

When allocating memory, first the freelist is walked to see if it could satisfy the request. If there's a chunk available on the freelist that will fit the request exactly, it will be taken, disconnected from the freelist, and returned to the caller. If no exact match could be found, the closest match that would just satisfy the request will be used. The chunk will normally be split up into one to be returned to the caller, and another (smaller) one that will remain on the freelist. In case this chunk was only up to two bytes larger than the request, the request will simply be altered internally to also account for these additional bytes since no separate freelist entry could be split off in that case.

If nothing could be found on the freelist, heap extension is attempted. This is where `__malloc_margin` will be considered if the heap is operating below the stack, or where `__malloc_heap_end` will be verified otherwise.

If the remaining memory is insufficient to satisfy the request, NULL will eventually be returned to the caller.

When calling free(), a new freelist entry will be prepared. An attempt is then made to aggregate the new entry with possible adjacent entries, yielding a single larger entry available for further allocations. That way, the potential for heap fragmentation is hopefully reduced. When deallocating the topmost chunk of memory, the size of the heap is reduced.

A call to realloc() first determines whether the operation is about to grow or shrink the current allocation. When shrinking, the case is easy: the existing chunk is split, and the tail of the region that is no longer to be used is passed to the standard free() function for insertion into the freelist. Checks are first made whether the tail chunk is large enough to hold a chunk of its own at all, otherwise realloc() will simply do nothing, and return the original region.

When growing the region, it is first checked whether the existing allocation can be extended in-place. If so, this is done, and the original pointer is returned without copying any data contents. As a side-effect, this check will also record the size of the largest chunk on the freelist.

If the region cannot be extended in-place, but the old chunk is at the top of heap, and the above freelist walk did not reveal a large enough chunk on the freelist to satisfy the new request, an attempt is made to quickly extend this topmost chunk (and thus the heap), so no need arises to copy over the existing data. If there's no more space available in the heap (same check is done as in malloc()), the entire request will fail.

Otherwise, malloc() will be called with the new request size, the existing data will be copied over, and free() will be called on the old region.

9 AVR-LibC and Assembler Programs

9.1 Introduction

There might be several reasons to write code for AVR microcontrollers using plain assembler source code. Among them are:

- Code for devices that do not have RAM and are thus not supported by the C compiler.
- Code for very time-critical applications.
- Special tweaks that cannot be done in C.

Usually, all but the first could probably be done easily using the inline assembler facility of the compiler.

Although AVR-LibC is primarily targeted to support programming AVR microcontrollers using the C (and C++) language, there's limited support for direct assembler usage as well. The benefits of it are:

- Use of the C preprocessor and thus the ability to use the same symbolic constants that are available to C programs, as well as a flexible macro concept that can use any valid C identifier as a macro (whereas the assembler's macro concept is basically targeted to use a macro in place of an assembler instruction).
- Use of the runtime framework like automatically assigning interrupt vectors. For devices that have RAM, initializing the RAM variables can also be utilized.

9.2 Invoking the Compiler

For the purpose described in this document, the assembler and linker are usually not invoked manually, but rather using the C compiler frontend (`avr-gcc`) that in turn will call the assembler and linker as required.

This approach has the following advantages:

- There is basically only one program to be called directly, `avr-gcc`, regardless of the actual source language used.
- The invocation of the C preprocessor will be automatic, and will include the appropriate options to locate required include files in the filesystem.
- The invocation of the linker will be automatic, and will include the appropriate options to locate additional libraries as well as the application start-up code (`crt`*XXX*`.o`) and linker script.

Note that the invocation of the C preprocessor will be automatic when the filename provided for the assembler file ends in `.S` (the capital letter "S"). This would even apply to operating systems that use case-insensitive filesystems since the actual decision is made based on the case of the filename suffix given on the command-line, not based on the actual filename from the file system.

As an alternative to using `.S`, the suffix `.sx` is recognized for this purpose (starting with GCC v4.3). This is primarily meant to be compatible with other compiler environments that have been providing this variant before in order to cope with operating systems where filenames are case-insensitive (and, with some versions of `make` that could not distinguish between `.s` and `.S` on such systems).

Alternatively, the language can explicitly be specified using the `-x assembler-with-cpp` option before the assembly source files. After the asm source file specifications a `-x none` must be added when non-assembly source files are following. Otherwise, the driver program would assume that they contain assembly code, too.

9.3 Example Program

The following annotated example features a simple 100 kHz square wave generator using an AT90S1200 clocked with a 10.7 MHz crystal. Pin PD6 will be used for the square wave output.

```
#include <avr/io.h>           // Note [1]

work    =   16                // Note [2]
tmp     =   17

inttmp  =   19
intsav  =   0

SQUARE  =   PD6               // Note [3]

#define IO(x) _SFR_IO_ADDR(x)

// Note [4]:
// 100 kHz => 200000 edges/s
tmconst = 10700000 / 200000

// # clocks in ISR until TCNT0 is set
fuzz = 8

.text

.global main                        // Note [5]
main:
    rcall   ioinit
1:  rjmp    1b                      // Note [6]

.global TIMER0_OVF_vect             // Note [7]
TIMER0_OVF_vect:
    ldi     inttmp, 256 - tmconst + fuzz
    out     IO(TCNT0), inttmp       // Note [8]

    in      intsav, IO(SREG)        // Note [9]

    sbic    IO(PORTD), SQUARE
    rjmp    1f
    sbi     IO(PORTD), SQUARE
    rjmp    2f
1:  cbi     IO(PORTD), SQUARE
2:
    out     IO(SREG), intsav
    reti

ioinit:
    sbi     IO(DDRD), SQUARE

    ldi     work, _BV(TOIE0)
    out     IO(TIMSK), work

    ldi     work, _BV(CS00)         // tmr0:  CK/1
    out     IO(TCCR0), work

    ldi     work, 256 - tmconst
    out     IO(TCNT0), work

    sei

    ret

.global __vector_default            // Note [10]
__vector_default:
    reti
```

Note [1]

As in C programs, this includes the central processor-specific file containing the IO port definitions for the device. Note that not all include files can be included into assembler sources.

Note [2]

Assignment of registers to symbolic names used locally. Another option would be to use a C preprocessor macro instead:

```
#define work 16
```

Note [3]

Our bit number for the square wave output. Note that the right-hand side consists of a CPP macro which will be substituted by its value (6 in this case) before actually being passed to the assembler.

Note [4]

The assembler uses integer operations in the host-defined integer size (32 bits or longer) when evaluating expressions. This is in contrast to the C compiler that uses the C type `int` by default in order to calculate constant integer expressions.

In order to get a 100 kHz output, we need to toggle the PD6 line 200000 times per second. Since we use timer 0 without any prescaling options in order to get the desired frequency and accuracy, we already run into serious timing considerations: while accepting and processing the timer overflow interrupt, the timer already continues to count. When pre-loading the `TCCNT0` register, we therefore have to account for the number of clock cycles required for interrupt acknowledge and for the instructions to reload `TCCNT0` (4 clock cycles for interrupt acknowledge, 2 cycles for the jump from the interrupt vector, 2 cycles for the 2 instructions that reload `TCCNT0`). This is what the constant `fuzz` is for.

Note [5]

External functions need to be declared to be `.global`. `main` is the application entry point that will be jumped to from the initialization routine in `crts1200.o`.

Note [6]

The main loop is just a single jump back to itself. Square wave generation itself is completely handled by the timer 0 overflow interrupt service. A `sleep` instruction (using idle mode) could be used as well, but probably would not conserve much energy anyway since the interrupt service is executed quite frequently.

Note [7]

Interrupt functions can get the usual names that are also available to C programs. The linker will then put them into the appropriate interrupt vector slots. Note that they must be declared `.global` in order to be acceptable for this purpose. This will only work if `<avr/io.h>` has been included. Note that the assembler or linker have no chance to check the correct spelling of an interrupt function, so it should be double-checked. (When analyzing the resulting object file using `avr-objdump` or `avr-nm`, a name like `__vector_`*N* should appear, with *N* being a small integer number.)

Note [8]

As explained in the section about special function registers, the actual IO port address should be obtained using the macro `_SFR_IO_ADDR`. (The AT90S1200 does not have RAM thus the memory-mapped approach to access the IO registers is not available. It would be slower than using `in` / `out` instructions anyway.)

Since the operation to reload `TCCNT0` is time-critical, it is even performed before saving `SREG`. Obviously, this requires that the instructions involved would not change any of the flag bits in `SREG`.

Note [9]

Interrupt routines must not clobber the global CPU state. Thus, it is usually necessary to save at least the state of the flag bits in `SREG`. (Note that this serves as an example here only since actually, all the following instructions would not modify `SREG` either, but that's not commonly the case.)

Also, it must be made sure that registers used inside the interrupt routine do not conflict with those used outside. In the case of a RAM-less device like the AT90S1200, this can only be done by agreeing on a set of registers to be used exclusively inside the interrupt routine; there would not be any other chance to "save" a register anywhere.

If the interrupt routine is to be linked together with C modules, care must be taken to follow the register usage guidelines imposed by the C compiler. Also, any register modified inside the interrupt service needs to be saved, usually on the stack.

Note [10]

As explained in Interrupts, a global "catch-all" interrupt handler that gets all unassigned interrupt vectors can be installed using the name `__vector_default`. This must be `.global`, and obviously, should end in a `reti` instruction. (By default, a jump to location 0 would be implied instead.)

9.4 Assembler Directives

The directives available in the assembler are described in the GNU assembler (gas) manual at Assembler Directives.

As gas comes from a Unix origin, its directives and overall assembler syntax is slightly different than the one being used by other assemblers. Numeric constants follow the C notation (prefix `0x` for hexadecimal constants, `0b` for binary constants), expressions use a C-like syntax.

Some common directives include:

Table 14 Assembler Directives: Sections

Section Ops	Description
`.section` *name*`,"`*flags*`",@`*typ*	Put the following objects into named section *name*. Set section flags *flags* and section type to *typ*
`.pushsection ...` `.popsection`	Like `.section`, but also pushes the current section and subsection onto the section stack. The current section and subsection can be restored with `.popsection`.
`.subsection` *int*	Put the following code into subsection number *int* which is some integer. Subsections are located in order of increasing index within their input section. The default after switching to a new section by means of `.section` or `.pushsection` is subsection 0. Subsections only exist on the assembly level, i.e. subsection ordering doesn't work across different modules.

`.text`
`.data`
`.bss`
	Put the following code into the `.text` section, `.data` section or `.bss` section, respectively. The assembler knows the right section flags and section type, for example the `.text` directive is basically the same like `.section .text,"ax",@progbits`. The directives support an optional subsection argument, see `.subsection` above.

Table 15 Assembler Directives: Symbols

Symbol Ops	Description
`.global` *sym* `.globl` *sym*	Globalize symbol *sym* so that it can be referred to in other modules. When a symbol is used without prior declaration or definition, the symbol is implicitly global. The `.global` directive can also be used to refer to that symbol, so that the linker pulls in code that defines the symbol (provided such a symbol definition exists). For example, code that puts objects in the `.data` section and that assumes that the startup code initializes that area, would use `.global __do_copy_data`.
`.weak` *syms*	Declare symbols *syms* as weak symbols, where *syms* is a comma-separated list of symbols. This applies only when the symbols are also defined in the same module. When the linker encounters a weak symbol that is also defined as `.global` in a different module, then the linker will use the latter without raising a diagnostic about multiple symbol definitions.
`.type` *sym*`,@`*kind*	Set the type of symbol *sym* to *kind*. Commonly used symbol types are `@function` for function symbols like `main` and `@object` for data symbols. This has an effect for disassemblers, debuggers and tools that show function / object properties.
`.size` *sym*`,`*size*	Set the size associated with symbol *sym* to expression *size*. The linker works on the level of sections, it does not even know what functions are. This directive serves bookkeeping, and may be useful for debuggers, disassemblers or tools that show which function / object consumes how much memory.
`.set` *sym*`,` *expr* `.equ` *sym*`,` *expr* *sym* `=` *expr*	Set the value of symbol *sym* to the value of expression *expr*. When a global symbol is set multiple times, the value stored in the object file is the last value stored into the symbol.

Symbol Ops	Description
`.extern`	Ignored for compatibility with other assemblers.
`.org`	Advance the location pointer to a specified offset *relative* to the beginning of the input section. The location counter cannot be moved backwards. This is a fairly pointless directive in an assembler environment that uses relocatable object files. The linker determines the final location of the objects. See the FAQ on how to relocate code to a fixed address.

Table 16 Assembler Directives: Data

Data Ops	Description	Alias
`.byte` *list*	Allocate bytes specified by a list of comma-separated expressions.	
`.2byte` *list*	Similar to `.byte`, but for 16-bit values.	`.word`
`.4byte` *list*	Similar to `.byte`, but for 32-bit values.	`.long`
`.8byte` *list*	Similar to `.byte`, but for 64-bit values.	`.qword`
`.ascii "`*string*`"`	Allocate a string of characters without \0 termination.	
`.asciz "`*string*`"`	Allocate a \0 terminated string.	`.string`
`.base64 "`*string*`"`	Used for more efficient encoding of (large) blocks of binary data, see PR31964.	
`.float` *list*	Allocate IEEE-754 single 32-bit floating-point values specified in the comma-separated *list*.	
`.double` *list*	Same, but for IEEE-754 double 64-bit floats.	
`.space` *num*`[,`*val*`]`	Allocate *num* bytes with value *val* where *val* is optional and defaults to zero.	`.skip`
`.zero` *num*	Insert *num* zero bytes.	

Table 17 Assembler Directives: Alignment

Alignment Ops	Description	Alias
`.balign` *val*	Align the following code to *val* bytes, where *val* is an absolute expression that evaluates to a power of 2.	`.align`
`.p2align` *expo*	Align the following code to 2^{expo} bytes.	

Moreover, there is the **`.macro`** directive, which starts an assembler macro. The GNU assembler implements a powerful macro processor which even supports recursive macro definitions. For an example, see the gas documentation for `.macro`. A gas `.macro` can further be combined with C preprocessor directives. For some real-world examples, see the AVR-LibC source `asmdef.h`.

9.5 Operand Modifiers

There are some AVR-specific operators available like `lo8()`, `hi8()`, `pm()`, `gs()` etc. For an overview, see the documentation of the operand modifiers in the inline assembly Cookbook.

Example:

```
ldi  r24, lo8(gs(somefunc))
ldi  r25, hi8(gs(somefunc))
call      something
subi r24, lo8(-(my_var))
sbci r25, hi8(-(my_var))
```

This passes the address of function `somefunc` as the first parameter to function `something`, and *adds* the address of variable `my_var` to the 16-bit return value of `something`.

9.6 Using the C Runtime

The C runtime is a collection of code that is required for a correct operation of a C/C++ program, and that is not generated by the compiler but provided by support libraries and startup-code. For example, variables in static storage like `int two = 2;` have to be initialized so that when `main` starts, these variables hold their required initial value.

This support code can also be used in assembly programs. The code can be divided into two classes: Code that is used per default, and code that has to be explicitly pulled in from a target library. As explained above, the assembler should be invoked via `avr-gcc`, so that the C runtime and support libraries are available.

9.6.1 Code that is used per Default

9.6.1.1 The Interrupt Vector Table

The code is linked against the interrupt vector table as provided by AVR-LibC, hence code like the following will work out of the box:

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

.global INT0_vect

INT0_vect:
    ;; Code for INT0 ISR
    reti
```

This will add the respective entry to the interrupt vector table e.g. `__vector_1` in the case of an ATmega8.

9.6.1.2 The __init Label and Code

The startup code from AVR-LibC also sets up the stackpointer according to the value of symbol `__stack`, and it sets `__zero_reg__` (R1) to zero. This code starts at `__init` which is jumped to from the reset vector via the very first entry in the vector table. `__init` label is located in section `.init0`, so that all the following init sections will also be executed in a fallthrough manner until section `.init9` which finally calls `main`. Thus, the following code will be executed without any further ado:

```
.text
.global main

main:
    ;; Code for main
```

When you prefer to implement all this in a custom manner, then you can link with option `-nostartfiles`.

9.6.2 Code that has to be pulled in by Hand

9.6.2.1 Initializing .data

In order to initialize variables in the `.data` section, startup routine `__do_copy_data` from libgcc has to be pulled in. It will run during `.init4`.

```
.data
.type   var, @object
.size   ccc, 2
var:
    .2byte  1234

.global __do_copy_data
```

9.6.2.2 Clearing .bss

Similarly, variables that clears the `.bss` section can be pulled in by referencing `__do_clear_bss` from libgcc. It will run during `.init4`.

```
.bss
.type  var0, @object
.size  var0, 2
var0:
    .zero  2

.global __do_clear_bss
```

9.6.2.3 Running static Constructors

Suppose you have a function `func` that you want to register as a constructor. The code that runs the constructors is `__do_global_ctors` from libgcc that runs during `.init6`. The addresses of the constructors are listed in section `.ctors:`

```
;; Define void func (void) as an ordinary function.
.type   func, @function
func:
    ;; Code
    ret
.size  func, .-func

;; Add func to the list of static constructors
.section .ctors,"a",@progbits
.p2align 1
.word  gs(func)

;; Pull in code to call the functions listed in .ctors
.global __do_global_ctors
```

Similarly, static destructors are run by `__do_global_dtors`.

9.6.2.4 Other Routines

All function in libgcc, libc and libm can be used, for example the 16-bit signed integer division routine:

```
    ;; R23:22 = R25:24 div R23:22
    ;; R25:24 = R25:24 mod R23:22
    ;; Clobbers: R21, R26, R27, SREG.
    call __divmodhi4
```

Notice that not all functions are following the avr-gcc calling convention, and availability of some functions depends on the instruction set.

10 Inline Assembler Cookbook

AVR-GCC
Inline Assembler Cookbook

10.1 About this Document

The GNU C/C++ compiler for AVR RISC processors offers to embed assembly language code into C/C++ programs. This cool feature may be used for manually optimizing time critical parts of the software, or to use specific processor instructions which are not available in the C language.

It's assumed that you are familiar with writing AVR assembler programs, because this is not an AVR assembler programming tutorial. It's not a C/C++ tutorial either.

Note that this document does not cover files written completely in assembly language, refer to AVR-LibC and Assembler Programs for this.

This document describes version 4.8 of the compiler or newer.

Herne, 17th of May 2002 Harald Kipp harald.kipp-at-egnite.de

10.2 The Anatomy of a GCC asm Statement

A GCC inline assembly statement starts with the keyword `asm`, `__asm` or `__asm__`, where the first one is not available in strict ANSI mode.

In its simplest form, the inline assembly statement has no operands and injects just one instruction into the code stream, like in

```
__asm ("nop");
```

In its generic form, an asm statements can have one of the following three forms:

__asm (code-string);
: A simple asm without operands: `code-string` is a string literal that will be added as is into the generated assembly code. This even applies to the % character. The only replacement is that \n and \t are interpreted as newline resp. TAB character.
This type of asm statement may occur at top level, outside any function as global asm. When its placement relative to functions is important, consider `-fno-toplevel-reorder`.
In `GCC v15` and up, `extended inline asm` can be used with some limitations outside of functions as well.

__asm volatile (code-string : output-operands : input-operands : clobbers);
: An asm with operands: This is the most widely used form of an asm statement. It must be located in a function.
`output-operands`, `input-operands` and `clobbers` are comma-separated lists of operands resp. clobber specifications. Any of them may be empty, for example when the asm has no outputs. At least one **:** (colon) must be present, otherwise it will be a simple asm without operands and without `%` replacements.

__asm goto (code-string : : input-operands : clobbers : labels);
: An asm goto statement: Like the asm above, but `labels` is a comma-separated list of C/C++ code labels which would be valid in a `goto` statement. And `output-operands` must be empty, because it is impossible to generate output reloads after the code has transferred control to one of the labels.
As there are no output operands, asm goto is implicitly volatile. When `volatile` is specified explicitly, the `goto` keyword may be placed after or before the `volatile`.

Notes on the various parts:

Volatility
: Keyword `volatile` is optional and means that the asm statement has side effects that are not expressed in terms of the operands or clobbers. The asm statement must not be optimized away or reordered with respect to other volatile statements like volatile memory accesses or other volatile asm.
Any asm statement without `output-operands` is implicitly volatile.
A non-volatile asm statement with output operands that are all unused may be optimized away when all output operands are unused.
Instead of `volatile`, `__volatile` or `__volatile__` can be used as well.

`code-string`
: A string literal that contains the code that is to be injected in the assembly code generated by the compiler. `%-expressions` are replaced by the string representations of the operands, and the number of lines is determined to estimate the code size of the asm.
Apart from that, **the compiler does not analyze the code provided in the code template.**
This means that the code appears to the compiler *as if it was executed in one parallel chunk, all at once*. It is important to keep that in mind, in particular for cases where input and output operands may overlap.

`output-operands`
`input-operands`
: A comma-separated list of operands, which may take the following forms. In any case, the first operand can be referred to as `"%0"` in `code-string`, the second one as `"%1"` etc.

"constraints" (expr)

`expr` is a C expression that's an input or output (or both) to the asm statement. An output expression must be an lvalue, i.e. it must be valid to assign a value to it.

`"constraints"` is a string literal with constraints and constraint modifiers. For example, constraint `"r"` stands for *general-purpose register*. A simple input operand would be

```
"r" (value + 1)
```

The compiler computes `value + 1` and supplies it in some general-purpose register R2...R31. In many cases, an upper d-register R16...R31 is required for instructions like `LDI` or `ANDI`. A respective output operand specification is

```
"=d" (result)
```

Notice that this operand may overlap with input operands!

When an operand is written before all input operands are consumed, then in almost all cases the output operand requires an early-clobber modifier `&` so that it won't overlap with any input operand:

```
"=&d" (result)
```

An operand that's both an output and an input can be expressed with the `+` constraint modifier:

```
"+d" (result)
```

Such an operand is both output and input, and hence it won't overlap with other operands.

[name] "constraints" (expr)

Like above. In addition, a named operand can be referred to as `%[name]` in `code-string`. This is useful in long asm statements with many operands.

clobbers

A comma-separated list of string literals like `"16"`, "r16" or "memory".

The first two clobbers mean that the asm destroys register R16. Only the lower-case form is allowed, and register names like Z are not recognized. A register that's clobbered will not be used as (part of) an input or output operand.

`"memory"` means that the asm touches memory in some way. When the asm writes to some RAM location for example, the compiler must not optimize RAM accesses across the asm because the memory may change.

Clobbering `__tmp_reg__` by means of `"r0"` has no effect, but such a clobber may be added to indicate to the reader that the asm clobbers R0.

Clobbering `__zero_reg__` by means of `"r1"` has no effect. When the asm destroys the zero register, for example by means of a `MUL` instruction, then the code must restore the register at the end by means of `"clr __zero_reg__"`.

Before we start with the first examples, we list all the bells and whistles that can be used to compose an inline assembly statement: special sequences, constraints, constraint modifiers, print modifiers operand modifiers and the size of an asm.

10.3 The Size of an asm

The code size of an asm statement is the number of lines multiplied by 4 bytes, the maximal possible AVR instruction length. The length is needed when (conditional) jumps cross the asm statement in order to compute (upper bounds for) jump offsets of PC-relative jumps.

The number of lines is one plus the number of line breaks in `code-string`. These may be physical line breaks from `\n` characters and logical line breaks from `$` characters.

10.4 Special Sequences

There are special sequences that can be used in the assembly template.

Resolution of `%` is only performed when the inline asm has operands or clobbers. For example, `asm("%%")` inserts *two* `%`'s whereas `asm("%%":)` only inserts one.

Table 18 Inline asm Special Sequences

Sequence	Description
__SREG__	The I/O address of the status register SREG at 0x3F
__tmp_reg__	The temporary register R0 (R16 on reduced Tiny)
__zero_reg__	The zero register R1, always zero (R17 on reduced Tiny)
$	A logical line separator, used to separate multiple instructions in one physical line
\n	A physical newline, used to separate multiple instructions
\t	A TAB character, can be used for better legibility of the generated asm
\"	A " character (double quote)
\\	A \ character (backslash)
%%	A % charater (percent)
%~	"r" or "", used to construct call or rcall by means of "%~call", depending on the architecture
%!	"" or "e", used to construct indirect calls like icall or eicall by means of "%!icall", depending on the architecture
%=	A number that's unique for the compilation unit and the respective inline asm code, used to construct unique labels
Comment	**Description**
; text	A single-line assembly comment that extends to the end of the physical line
/* text */	A multi-line C comment

- Moreover, the following I/O addresses are defined provided the device supports the respective SFR: __SP_L__, __SP_H__, __CCP__, __RAMPX__, __RAMPY__, __RAMPZ__, __RAMPD__.
- Register __tmp_reg__ may be freely used by inline assembly code and need not be restored at the end of the code.
- Register __zero_reg__ contains a value of zero. When that value is destroyed, for example by a MUL instruction, its value has to be restored at the end of the code by means of

  ```
  clr __zero_reg__
  ```

- In inline asm without operands (i.e without a single colon), a % will always insert a single %. No %-codes are available.

Sequences like __SREG__ are not evaluated as part of the inline asm, they are just copied to the asm code as they are. At the top of each assembly file, the compiler prints definitions like

```
__SREG__ = 0x3f
```

so that they can also be used in inline assembly.

10.5 Constraints

The most up-to-date and detailed information on constraints for the AVR can be found in the avr-gcc Wiki.

Table 19 Inline asm Operand Constraints

Constraint	Registers	Range
a	Simple upper registers that support FMUL	R16 ... R23
b	Base pointer registers that support LDD, STD	Y, Z (R28 ... R31)
d	Upper registers	R16 ... R31

Constraint	Registers	Range
e	Pointer registers that support LD, ST	X, Y, Z (R26 ... R31)
l	Lower registers	R2 ... R15
r	Any register	R2 ... R31
w	Upper registers that support ADIW	R24 ... R31
x	X pointer registers	R26, R27
y	Y pointer registers	R28, R29
z	Z pointer registers	R30, R31
Constraint	**Constant**	**Range**
I	6-bit unsigned integer constant	0 to 63
J	6-bit negative integer constant	-63 to 0
M	8-bit unsigned integer constant	0 to 255
n	Integer constant	
i	Immediate value known at link-time, like the address of a variable in static storage	
F	floating-point constant	
Ynn	Fixed-point or integer constant	
Constraint	**Explanation**	**Notes**
m	A memory location	
X	Any valid operand	
0 ... 9	Matches the respective operand number	

GCC v16 also supports hard-register constraints. They allow to specify a register in a constraint. For example, "{r20}" specifies register(s) starting at R20.

- Constraints without a modifier specify input operands.
- Constraints with a modifier specify output operands.
- More than one constraint like in "rn" specifies the union of the specified constraints; "r" and "n" in this case.
- All constraints listed above are single-letter constraints, except Ynn which is a 3-letter constraint.
- According to the avr-gcc ABI, values that occupy more than one register will start at an even register number.

Constraint modifiers are:

Table 20 Constraint Modifiers

Modifier	Meaning
=	Output-only operand. Without & it may overlap with input operands
+	Output operand that's also an input
=&	"Early-clobber". Register should be used for output only and won't overlap with any input operand(s)

Examples:

"a" (10)
Load the value of 10 into a register in the range of the a constraint, i.e. R16...R23. As int is a 16-bit value, so is 10, and the constant will occupy two consecutive hard registers.

"a" ((uint8_t) 10)
Same, but this time the constant only occupies one register.

"I" (10)
The literal value `10` can be used in the according `%` operand.

"J" (10)
An error: The value 10 is not in the range -63...0 of the `J` constraint.

"=a" (10)
An error: `10` is not an lvalue and cannot be written to.

"a" (var)
The variable `var` will be loaded into a register in the range R16...R23. A 4-byte value will occupy 4 consecutive registers etc.

"+a" (var)
The variable `var` will be loaded into a register in the range R16...R23. It is an input as well as an output of the inline asm, i.e. it must be an lvalue. Otherwise, e.g. when `var` is read-only, the compiler will complain.

"I" (var)
In most cases, this is an error because `var` is not known at compile-time. Only when `__builtin_constant_p(var)` is true and `var` is in the range 0...63, then `var` is accepted by the compiler.

The selection of the proper constraint depends on the range of the constants or registers, which must be acceptable to the AVR instruction they are used with. The C compiler doesn't check any line of your assembler code. But it is able to check the constraint against your C expression. However, if you specify the wrong constraints, then the compiler may silently pass wrong code to the assembler. And, of course, the assembler will fail with some cryptic output or internal errors, or in the worst case wrong code may be the result.

For example, if you specify the constraint `"r"` and you are using this register with an `ORI` instruction, then the compiler may select any register. This will fail if the compiler chooses `R2` to `R15`. (It will never choose `R0` or `R1`, because these are uses for special purposes.) That's why the correct constraint in that case is `"d"`. On the other hand, if you use the constraint `"M"`, the compiler will make sure that you don't pass anything else but an 8-bit unsigned integer value known at compile-time.

The following table shows all AVR assembler mnemonics which require operands, and the related constraints.

Table 21 AVR Instructions and Constraints

Mnemonic	Constraints		Mnemonic	Constraints
adc	r,r		add	r,r
adiw	w,I		and	r,r
andi	d,M		asr	r
bclr	I		bld	r,I
brbc	I,label		brbs	I,label
bset	I		bst	r,I
call	i		cbi	I,I
cbr	d,I		clr	r
com	r		cp	r,r
cpc	r,r		cpi	d,M
cpse	r,r		dec	r
elpm	r,z		eor	r,r
fmul	a,a		fmuls	a,a
fmulsu	a,a		in	r,I
inc	r		jmp	i
lac	z,r		las	z,r
lat	z,r		ld	r,e
ldd	r,b		ldi	d,M
lds	r,i		lpm	r,z
lsl	r		lsr	r
mov	r,r		movw	r,r

mul	r,r		muls	r,r
mulsu	a,a		neg	r
or	r,r		ori	d,M
out	I,r		pop	r
push	r		rcall	i
rjmp	i		rol	r
ror	r		sbc	r,r
sbci	d,M		sbi	I,I
sbic	I,I		sbiw	w,I
sbr	d,M		sbrc	r,I
sbrs	r,I		ser	d
st	e,r		std	b,r
sts	i,r		sub	r,r
subi	d,M		swap	r
tst	r		xch	z,r

10.6 Print Modifiers

The %-operands in the inline assembly template can be adjusted by special print-modify characters. The one-letter modifier follows the % and precedes the operand number like in "%a0", or precedes the name in named operands like in "%a[address]".

Table 22 Inline asm Print 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" (&SREG)	n
%n0	1	Print the negative of a compile-time integer 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.

10.7 Operand Modifiers

Table 23 Assembly Code Operand Modifiers

Modifier	Explanation	Purpose
lo8()	1st Byte of a link-time constant, bits 0...7	

Getting parts of a byte-address

Modifier	Explanation	Purpose
`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	
`pm_hi8()`	2nd Byte of a link-time constant divided by 2, bits 9...16	Getting parts of a word-address
`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) addresses, 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))`. **G**enerate **s**tub (trampoline) as needed. This is required to calculate the address of a code label on devices with more than 128 KiB of program memory that's supposed to be used in `EICALL`. For rationale, see the 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, then the assembler has to encode the expression in an abstract form using RELOCs. The consequence is that only a very limited number of argument expressions is supported when they are not computable at assembler-time.

10.8 Examples

Some examples show the assembly code as generated by the compiler. It's the code from the `.s` files as generated with option `-save-temps`. Adding the high-level source to the generated assembly can be turned on with `-fverbose-asm` since GCC v8.

10.8.1 Swapping Nibbles

The first example uses the `swap` instruction to swap the nibbles of a byte. Input and output of `swap` are located in the same general purpose register. This means the input operand, operand 1 below, must be located in the same register(s) like operand 0, so that the right constraint for operand 1 is `"0"`:

```
asm ("swap" : "=r" (value) : "0" (value));
```

All side effects of the code are described by the constraints and the clobbers, so that there is no need for this asm to be volatile. In particular, this asm may be optimized out when the output value is unused.
A shorter pattern to state that `value` is both input and output is by means of constraint modifier `+`

```
asm ("swap" : "+r" (value));
```

10.8.2 Swapping Bytes

Swapping nibbles was a piece of cake, so let's swap the bytes of a 16-bit value. In order to access the constituent bytes of the 16-bit input and output values, we use the print modifiers `%A` and `%B`.

The asm is placed in a small C test case so that we can inspect the resulting assembly code as generated by the compiler with `-save-temps`.

```
void callee (int, int);

void func (int param)
{
    int swapped;

    asm ("mov %A0, %B1" "\n\t"
         "mov %B0, %A1"
         : "=r" (swapped) : "r" (param));

    callee (param, swapped);
}
```

The `"\n\t"` sequence adds a line feed that is required between the two instructions, and a TAB to align the two instructions in the generated assembly. There is no `"\n\t"` after the last instruction because that would just increase the size of the asm.
The generated assembly works as expected. The compiler wraps it in `#APP` / `#NOAPP` annotations:

```
func:
/* #APP */
    mov r22, r25     ;  swapped, param
    mov r23, r24     ;  swapped, param
/* #NOAPP */
    jmp callee
```

Wrong! **While the generated code above is correct, the inline asm itself is not!**
We see this with a slightly adjusted test case where the arguments of `callee` have been swapped, but that uses the same inline asm:

```
void func (int param)
{
    int swapped;

    asm ("mov %A0, %B1" "\n\t"
         "mov %B0, %A1"
         : "=r" (swapped) : "r" (param));

    callee (swapped, param);
}
```

The result is the following assembly:

```
func:
    movw r22,r24
/* #APP */
    mov r24, r25     ;  swapped, param
    mov r25, r24     ;  swapped, param
/* #NOAPP */
    jmp callee
```

which is obviously wrong, because after the code from the inline asm, the low byte of `swapped` and the high byte will always have the same value of `r25`.

The reason is that the output operand overlaps the input, *and* the output is changed before all of the input operands are consumed. This is a so-called *early-clobber* situation. There are two possible solutions to this predicament:

- Mark the output operand with the early-clobber constraint modifier:

  ```
  asm ("mov %A0, %B1" "\n\t"
       "mov %B0, %A1"
       : "=&r" (swapped) : "r" (param));
  ```

- Use constraints and a code sequence that expect input and output in the same registers:

  ```
  asm ("eor %A0, %B0" "\n\t"
       "eor %B0, %A0" "\n\t"
       "eor %A0, %B0"
       : "=r" (swapped) : "0" (param));
  ```

10.8.3 Accessing Memory

Accessing memory requires that the AVR instructions that perform the memory access are provided with the appropriate memory address.

1. The address can be provided directly, like __SREG__, 0x3f, as a symbol, or as a symbol plus a constant offset.
2. Provide the address by means of an inline asm operand.

Approach 1 is simpler as it does not require an asm operand, while approach 2 is in many cases more powerful because macros defined per, say, `#include <avr/io.h>` can be used as operands, whereas such headers are not included in the assembly code as generated by the compiler.

Reading a SFR like `PORTB` can be performed by

```
asm volatile ("in %0, %i1" : "=r" (result) : "I" (& PORTB));
```

or

```
asm volatile ("in %0, %1" : "=r" (result) : "I" (_SFR_IO_ADDR (PORTB)));
```

where the former with print modifier `%i` is only supported since GCC v4.7. Macro `_SFR_IO_ADDR` is provided by `avr/sfr_defs.h` which is included by `avr/io.h`.

When the address is not an I/O address, then `LDS` or `LD` must be used, depending on whether the address is known at link-time or only at run-time. For example, the following macro provides the functionality to clear an SFR. The code discriminates between the possibilities that

- The SFR address is known at compile-time and is an I/O address.
- The SFR address is known at compile-time but is not in the I/O range.
- The SFR address is not known at compile-time.

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

#define CLEAR_REG(sfr)                          \
do {                                            \
  if (__builtin_constant_p (& (sfr))            \
      && _SFR_IO_REG_P (sfr))                   \
    asm volatile ("out %i0, __zero_reg__"       \
                  :: "I" (& (sfr)) : "memory"); \
  else if (__builtin_constant_p (& (sfr)))      \
    asm volatile ("sts %0, __zero_reg__"        \
                  :: "n" (& (sfr)) : "memory"); \
  else                                          \
    asm volatile ("st %a0, __zero_reg__"        \
                  :: "e" (& (sfr)) : "memory"); \
} while (0)
```

The last case with constraint `"e"` works because `&sfr` is a 16-bit value, and 16-bit values (and larger) start in even registers. Therefore, the address will be located in R27:R26, R29:R28 or in R31:R30, which print modifier `%a` will print as X, Y or Z, respectively. The address will never end up in, say, R30:R29.

The test case

```
void clear_3_regs (uint8_t volatile *psfr)
{
    CLEAR_REG (PORTB);
    CLEAR_REG (UDR0);
    CLEAR_REG (*psfr);
}
```

compiles for ATmega328 and with optimization turned on to

```
clear_3_regs:
    movw r30,r24
/* #APP */
    out 0x5, __zero_reg__
    sts 198, __zero_reg__
    st Z,    __zero_reg__   ;  psfr
/* #NOAPP */
    ret
```

As `__builtin_constant_p` is used to infer whether the address of the SFR is known at compile-time, extra care must be taken when the functionality is implemented as an inline function:

```
static inline __attribute__((__always_inline__))
void clear_reg (uint8_t volatile *psfr)
{
  // !!! The following cast is required to make __builtin_constant_p
  // !!! work as expected in the inline function.
  uintptr_t addr = (uintptr_t) psfr;

  if (__builtin_constant_p (addr)
      && _SFR_IO_REG_P (* psfr))
    asm volatile ("out %i0, __zero_reg__"
                  :: "I" (addr) : "memory");
  else if (__builtin_constant_p (addr))
    asm volatile ("sts %0, __zero_reg__"
                  :: "n" (addr) : "memory");
  else
    asm volatile ("st %a0, __zero_reg__"
                  :: "e" (addr) : "memory");
}

void clear_3_pregs (uint8_t volatile *psfr)
{
  clear_reg (& PORTB);
  clear_reg (& UDR0);
  clear_reg (psfr);
}
```

Casting the address `psfr` to an integer type in the inline function is required so that the compiler will recognize constant addresses.
Also notice that we have to pass the *address of the SFR* to the inline function. Passing the SFR directly like in the marco approach won't work for obvious reasons.

10.8.4 Accessing Bytes of wider Expressions

Finally, an example that atomically increments a 16-bit integer. The code is wrapped in `IN SREG` / `CLI` / `OUT SREG` to make it atomic. It reads the 16-bit value `data` from its absolute address, increments it and then writes it back:

```
uint16_t volatile data;

void inc_data (void)
{
    uint16_t tmp;
    asm volatile ("in __tmp_reg__, __SREG__"    "\n\t"
                  "cli"                         "\n\t"
                  "lds %A[temp], %[addr]"       "\n\t"
                  "lds %B[temp], %[addr]+1"     "\n\t"
#ifdef __AVR_TINY__
                  // Reduced Tiny does not have ADIW.
                  "subi %A[temp], lo8(-1)"      "\n\t"
                  "sbci %B[temp], hi8(-1)"      "\n\t"
#else
                  "adiw %[temp], 1"             "\n\t"
#endif
                  "sts %[addr]+1, %B[temp]"     "\n\t"
                  "sts %[addr],   %A[temp]"     "\n\t"
```

```
                  "out __SREG__, __tmp_reg__"
#ifdef __AVR_TINY__
                  // No need to restrict tmp to a "w" register. And on
                  // avr-gcc v13.2 and older, "w" contains no regs.
                  : [temp] "=d" (tmp), "+m" (data)
#else
                  : [temp] "=w" (tmp), "+m" (data)
#endif
                  : [addr] "i" (& data));
}
```

Notice there are three different ways required to access the different bytes of the involved 16-bit entities:

- For the 16-bit general purpose register `%[temp]`, print modifiers `%A` and `%B` are used.
- For the 16-bit value `data` in static storage, `%[addr]+1` is used to access the high byte. The resulting expression `data+1` is computable at link-time and evaluated by the linker.
- In the compilation variant for Reduced Tiny, the bytes of the 16-bit subtrahend `-1` are accessed with the operand modifiers `lo8` and `hi8` that are evaluated by the assembler because `-1` is known at assembler-time.

`data` is located in static storage, hence its address is known to the linker and fits constraint `"i"`.

The sole purpose of operand `"+m" (data)` is to describe the effect of the asm on data memory: It changes `data`. Notice that there is no `"memory"` clobber, because that operand already describes all memory side effects, and it does this in a less intrusive way than a catch-all `"memory"`. The operand is not used in the asm template; but in principle it would be possible to use it as operand with `LDS` and `STS` instead of operand `[addr] "i" (& data)`. However, there are many situations where a memory operand constrained by `"m"` takes a form that cannot be used with AVR instructions because there are no matching print modifiers, or because it is not known a priori what specific form the memory operand takes. In such cases, one would take the address of the operand and supply it as address in a pointer register to the inline asm. The compiler generates the required instructions for address computation, and the inline asm knows that it can use `LD` and `ST`.

10.8.5 Jumping and Branching

When an inline asm contains jumps, then it also requires labels. When the label is inside the asm, then care must be taken that the label is unique in the compilation unit even when the inline asm is used multiple times, e.g. when the code is located in an unrolled loop or a function has multiple incarnations due to cloning, or simply because a macro or inline function that contains an asm statement is used more than once.
There are two kinds of labels that can be used:

- Local labels of the form *n*`:` where *n* is some (small, non-negative) number. They can be targeted by means of *n*`b` or *n*`f`, depending on whether the jump direction is **b**ackwards or **f**orwards. Such a numeric labels may be present more than once. The taken label is the first one with the specified number in the respective direction:

```
// Loop until bit PORTB.7 is set.
asm volatile ("1: sbrs %i[sfr], %[bitno]"  "\n\t"
              "rjmp 1b"
              :: [sfr] "I" (& PORTB), [bitno] "n" (PB7));
```

- Local labels that contain the sequence `%=` which yields some number that's unique amongst all asm incarnations in the respective compilation unit:

```
// Loop until bit PORTB.7 is set.
asm volatile (".Loop.%=: sbrs %i[sfr], %[bitno]"  "\n\t"
              "rjmp .Loop.%="
              :: [sfr] "I" (& PORTB), [bitno] "n" (PB7));
```

Which form is used is a matter of taste. In practice, the first variant is often preferred in short sequences, whereas the second form is usually seen in longer algorithms.

For labels that are defined in the surrounding C/C++ code, asm goto has to be used. The print modifier `%x0` prints `panic` as a raw label, not as `gs(panic)` like it would be the case with `%0`.

```
int main (void)
{
    asm goto ("tst __zero_reg__" "\n\t"
              "brne %x0"
              :::: panic);
    /* ...Application code here... */
    return 0;
panic:
    // __zero_reg__ is supposed to contain 0, but doesn't.
    return 1;
}
```

This assumes that the jump offset can be encoded in the `brne` instruction in all situations. When static analysis cannot prove that the jump offset fits, then a jumpity jump has to be used:

```
asm goto ("tst   __zero_reg__" "\n\t"
          "breq  1f"           "\n\t"
          "%~jmp %x0"          "\n"
          "1: ;; all fine"
          :::: panic);
```

Sequence `"%~jmp"` yields `"rjmp"` or `"jmp"` depending on the architecture. Notice that a `jmp` can be relaxed to an `rjmp` with option `-mrelax` provided the jump offset fits.

10.8.6 Interfacing non-ABI Functions

Suppose we want to interface a non- ABI assembly function `mul_8_16` that multiplies R24 with R27:R26, clobbers R0, R1 and R25, and returns the 24-bit result in R20:R19:R18. One way to implement such an interface would be to provide an assembly function that performs the required copying and call to `mul_8_16`. Such a function would destroy some of the performance gain obtained by using assembly for `mul_8_16`: Additional copying back and forth and extra `CALL` and `RET` instructions.

10.8.6.1 Binding local Variables to Registers

The compiler comes to the rescue. We can bind local variables to the required registers.

One use of GCC's `asm` keyword is to bind local register variables to hardware registers.
Such bindings of local variables to registers are only guaranteed during inline asm which has these variables as operands.

Here is an example how interfacing `mul_8_16` could look like:

```
extern void mul_8_16 (void); // Non-ABI function. Don't call in C/C++!

static inline __attribute__((__always_inline__))
uint24_t mul_8_16_gccabi (uint8_t val8, uint16_t val16)
{
    register uint8_t r24 __asm("r24") = val8;
    register uint24_t r18 __asm("r18");

    asm ("%~call %x[func]"  "\n\t"
         "clr    __zero_reg__"
         : "=r" (r18)
         : "r" (r24), "x" (val16), [func] "i" (mul_8_16)
         : "r25", "r0");

    return r18;
}
```

- The 8-bit parameter is bound to R24, and the 24-bit return value is bound to R18...R20.
- The `register` keyword is mandatory.
- The hard register is specified as a string literal for the lower case register name or register number, like `"18"` or `"r18"`. Specifications like `"R18"`, `18` or `"Z"` are not supported.
- The 16-bit parameter of `mul_8_16` happens to be required in R27:R26, which is the X register for which there is register constraint `"x"`. Therefore, no register binding is required for `val16`.
- As `mul_8_16` clobbers the zero register R1, it has to be restored by means of

  ```
  clr __zero_reg__
  ```

- The asm is pure arithmetic and hence not volatile. (It might be advisable to make it volatile anyway, so that it won't be reordered across `sei()` or `cli()` instructions.)

Let's have a look at how this performs in a test case:

```
void use_mul_8_16_gccabi (uint8_t val, uint8_t a, uint8_t b)
{
    if (mul_8_16_gccabi (val, a * b) >= 0x2010)
        __builtin_abort();
}
```

For ATmega8 we get the following assembly:

```
use_mul_8_16_gccabi:
    mul  r22,r20
    movw r26,r0
    clr  __zero_reg__
/* #APP */
    rcall mul_8_16
    clr   __zero_reg__
/* #NOAPP */
    cpi  r18,16
    sbci r19,32
    cpc  r20,__zero_reg__
    brlo .L1
    rcall abort
.L1:
    ret
```

No superfluous register moves. Great!

10.8.6.2 Hard-Register Constraints

GCC v16 supports `hard-register constraints` that are easier to use than local register variables. With that feature, the code from above can be simplified to:

```
extern void mul_8_16 (void); // Non-ABI function. Don't call in C/C++!

static inline __attribute__((__always_inline__))
uint24_t mul_8_16_gccabi (uint8_t val8, uint16_t val16)
{
    uint24_t result;

    asm ("%~call %x[func]"  "\n\t"
         "clr    __zero_reg__"
         : "={r18}" (result)
         : "{r24}" (val8), "{r26}" (val16), [func] "i" (mul_8_16)
         : "r25", "r0");

    return result;
}
```

10.9 Specifying the Assembly Name of Static Objects

Sometimes, it is desirable to use a different name for an object or function rather than the (mangled) name from the C/C++ implementation. Just add an asm specifier with the desired name as a string literal at the end of the declaration.

For example, this is how `avr/eeprom.h` desclares the `eeprom_read_double()` function:

```
#if __SIZEOF_DOUBLE__ == 4
double eeprom_read_double (const double*)  __asm("eeprom_read_dword");
#elif __SIZEOF_DOUBLE__ == 8
double eeprom_read_double (const double*)  __asm("eeprom_read_qword");
#endif
```

- It uses the implementation of `eeprom_read_dword` for `eeprom_read_double`, provided `double` is a 32-bit type.
- It uses the implementation of `eeprom_read_qword` for 64-bit doubles.

10.10 What won't work

GCC inline asm has some limitations.

10.10.1 Setting a Register in one asm and using it in a different one

Sequences like the following are not supposed to work:

```
char var;

void set_var (char c)
{
    __asm ("inc r24");
    __asm ("sts var, r24");
}
```

- There is no guarantee whatsoever that the value in R24 will survive from one asm to the next. **Such code might work in many situations, but it is still wrong.** The compiler may very well put instructions between the asm statements that change R24.
- R24 is changed without noticing the compiler. When R24 contains other data, then that data will be trashed.

A correct code would be

```
__asm ("inc %0"    "\n\t"
       "sts var, %0"
       :: "r" (c) : "memory");
```

or

```
__asm ("inc %1"    "\n\t"
       "sts %0, %1"
       : "=m" (var) : "r" (c));
```

10.10.2 Referring to a Local Register Variable that's not an Operand to the asm

Code like the following won't work:

```
register uint8_t reg2 __asm("r2");
__asm ("clr r2");
```

That code assumes that the local register variable `reg2` is bound to register R2 in the inline asm, but that is not the case. Only when a local register variable is used as an operand it will be loaded to the specified register:

```
__asm ("clr r2" : "=r" (reg2));
// or
__asm ("clr %0" : "=r" (reg2));
```

10.10.3 Letting an Operand cross the Boundaries of the Y Register

It is not possible to bind a value to a local register variable that crosses the boundaries of the Y register. For example, trying to bind a 32-bit value to R31:R28 by means of

```
register uint32_t r28 __asm ("28");
```

will result in an error message like

```
error: register specified for 'r28' isn't suitable for data type
```

Similarly, an operand described by a constraint will be located either completely below the Y register, as part of Y register, or above it.

10.10.4 Using Matching Constraints "=0"..."=9" with Output Operands

Suppose we want an inline asm that returns the low byte of a 16-bit value `val16`:

```
asm ("" : "=1" (lo8) : "r" (val16));
```

The diagnostic will be:

```
error: matching constraint not valid in output operand
```

11 How to Build a Library

11.1 Introduction

So you keep reusing the same functions that you created over and over? Tired of cut and paste going from one project to the next? Would you like to reduce your maintenance overhead? Then you're ready to create your own library! Code reuse is a very laudable goal. With some upfront investment, you can save time and energy on future projects by having ready-to-go libraries. This chapter describes some background information, design considerations, and practical knowledge that you will need to create and use your own libraries.

11.2 How the Linker Works

The compiler compiles a single high-level language file (C language, for example) into a single object module file. The linker (ld) can only work with object modules to link them together. Object modules are the smallest unit that the linker works with.

Typically, on the linker command line, you will specify a set of object modules (that has been previously compiled) and then a list of libraries, including the Standard C Library. The linker takes the set of object modules that you specify on the command line and links them together. Afterwards there will probably be a set of "undefined references". A reference is essentially a function call. An undefined reference is a function call, with no defined function to match the call.

The linker will then go through the libraries, in order, to match the undefined references with function definitions that are found in the libraries. If it finds the function that matches the call, the linker will then link in the object module in which the function is located. This part is important: the linker links in THE ENTIRE OBJECT MODULE in which the function is located. Remember, the linker knows nothing about the functions internal to an object module, other than symbol names (such as function names). The smallest unit the linker works with is object modules.

When there are no more undefined references, the linker has linked everything and is done and outputs the final application.

11.3 How to Design a Library

How the linker behaves is very important in designing a library. Ideally, you want to design a library where only the functions that are called are the only functions to be linked into the final application. This helps keep the code size to a minimum. In order to do this, with the way the linker works, is to only write one function per code module. This will compile to one function per object module. This is usually a very different way of doing things than writing an application!

There are always exceptions to the rule. There are generally two cases where you would want to have more than one function per object module.

The first is when you have very complementary functions that it doesn't make much sense to split them up. For example, malloc() and free(). If someone is going to use malloc(), they will very likely be using free() (or at least should be using free()). In this case, it makes more sense to aggregate those two functions in the same object module.

The second case is when you want to have an Interrupt Service Routine (ISR) in your library that you want to link in. The problem in this case is that the linker looks for unresolved references and tries to resolve them with code in libraries. A reference is the same as a function call. But with ISRs, there is no function call to initiate the ISR. The ISR is placed in the Interrupt Vector Table (IVT), hence no call, no reference, and no linking in of the ISR. In order to do this, you have to trick the linker in a way. Aggregate the ISR, with another function in the same object module, but have the other function be something that is required for the user to call in order to use the ISR, like perhaps an initialization function for the subsystem, or perhaps a function that enables the ISR in the first place.

11.4 Creating a Library

The librarian program is called `ar` (for "archiver") and is found in the GNU Binutils project. This program will have been built for the AVR target and will therefore be named `avr-ar`.

The job of the librarian program is simple: aggregate a list of object modules into a single library (archive) and create an index for the linker to use. The name that you create for the library filename must follow a specific pattern: lib*name*.a. The *name* part is the unique part of the filename that you create. It makes it easier if the *name* part relates to what the library is about. This *name* part must be prefixed by "lib", and it must have a file extension of .a, for "archive". The reason for the special form of the filename is for how the library gets used by the toolchain, as we will see later on.

Note

The filename is case-sensitive. Use a lowercase "lib" prefix, and a lowercase ".a" as the file extension.

The command line is fairly simple:

```
avr-ar rcs <library name> <list of object modules>
```

The `r` command switch tells the program to insert the object modules into the archive with replacement. The `c` command line switch tells the program to create the archive. And the `s` command line switch tells the program to write an object-file index into the archive, or update an existing one. This last switch is very important as it helps the linker to find what it needs to do its job.

Note

The command line switches are case sensitive! There are uppercase switches that have completely different actions.

MFile and the WinAVR distribution contain a Makefile Template that includes the necessary command lines to build a library. You will have to manually modify the template to switch it over to build a library instead of an application.

See the GNU Binutils manual for more information on the `ar` program.

11.5 Using a Library

To use a library, use the `-l` switch on your linker command line. The string immediately following the `-l` is the unique part of the library filename that the linker will link in. For example, if you use:

```
-lm
```

this will expand to the library filename:

```
libm.a
```

which happens to be the math library included in AVR-LibC.

If you use this on your linker command line:

```
-lprintf_flt
```

then the linker will look for a library called:

```
libprintf_flt.a
```

This is why naming your library is so important when you create it!

The linker will search libraries in the order that they appear on the command line. Whichever function is found first that matches the undefined reference, it will be linked in.

There are also command line switches that tell GCC which directory to look in (`-L`) for the libraries that are specified to be linked in with `-l`.

See the GNU Binutils manual for more information on the GNU linker (ld) program.

12 Benchmarks

- Benchmarks for libc functions
- Benchmarks for IEEE single floating-point functions from <math.h>
- Benchmarks for IEEE double floating-point functions from <math.h>
- Benchmarks for fixed-point functions from <stdfix.h>

The results below can only give a rough estimate of the resources necessary for using certain library functions. There is a number of factors which can both increase or reduce the effort required:

- Expenses for preparation of operands and their stack are not considered.
- In the table, the size includes all dependent functions.
- Expenses of time of performance of some functions essentially depend on parameters of a call, for example, qsort() is recursive, and sprintf() receives parameters in a stack.
- Different versions of the compiler can give a significant difference in code size and execution time. For example, the dtostre() function, compiled with avr-gcc 3.4.6, requires 930 bytes. After transition to avr-gcc 4.2.3, the size become 1088 bytes.

12.1 A few of libc Functions

avr-gcc version is 15.2.1

The size of a function is given in view of all picked up functions. By default AVR-LibC is compiled with `-mcall-prologues`. In parenthesis the size without taking into account the code for the prologue and epilogue routines is shown. Both sizes can coincide when no prologue and epilogue routines are present, and then only one code size is shown.

- `qsort` sorts an array of `char` with 100 elements.
- The used size of `double` is 4, which plays a role in the float type promotions of arguments of varargs functions like `sprintf`.
- For an overview of the different AVR architectures, see `avr-gcc: Command Line Options`.

Function	Units	avr2	avr25	avr4	avr6
`atoi ("12345")`	Flash bytes	82	78	74	76
	Stack bytes	4	4	4	6
	Cycles	155	149	149	167
`atol ("12345")`	Flash bytes	126	118	106	110
	Stack bytes	5	5	4	6
	Cycles	221	209	205	223
`ftostre (1.2345f, s, 6, 0)`	Flash bytes	1128 (1016)	1058 (948)	1058 (948)	1078 (968)
	Stack bytes	20	20	20	22
	Cycles	1339	1178	1178	1191
`ftostrf (1.2345f, 15, 6, s)`	Flash bytes	1648 (1536)	1524 (1414)	1524 (1414)	1548 (1438)
	Stack bytes	40	40	40	43
	Cycles	1645	1469	1469	1486
`ktoa (123.45k, s, 2)`	Flash bytes	316	306	298	306
	Stack bytes	8	8	8	10
	Cycles	479	466	466	488
`itoa (12345, s, 10)`	Flash bytes	110	102	102	106
	Stack bytes	2	2	2	3
	Cycles	879	875	875	880

Function	Units	avr2	avr25	avr4	avr6
`ltoa (12345678L, s, 10)`	Flash bytes	138	130	130	136
	Stack bytes	2	2	2	3
	Cycles	2766	2762	2762	2767
`lltoa (12345678LL, s, 10)`	Flash bytes	206	194	194	202
	Stack bytes	3	3	3	4
	Cycles	2239	2209	2209	2214
`ulltoa_base10 (12345678ULL, s)`	Flash bytes	182	172	168	168
	Stack bytes	11	11	11	12
	Cycles	1908	1852	1458	1461
`malloc (1)`	Flash bytes	668	606	606	610
	Stack bytes	6	6	6	7
	Cycles	101	97	97	100
`realloc ((void*) 0, 1)`	Flash bytes	1242 (1130)	1114 (1004)	1114 (1004)	1096
	Stack bytes	6	6	6	7
	Cycles	101	97	97	100
`qsort (s, sizeof(s), 1, cmp)`	Flash bytes	1130 (1018)	952 (842)	938 (828)	984 (874)
	Stack bytes	146	146	146	153
	Cycles	54683	50635	45705	47930
`rand ()`	Flash bytes	132	120	120	124
	Stack bytes	2	2	2	3
	Cycles	95	90	90	93
`random ()`	Flash bytes	678 (566)	634 (524)	630 (520)	584
	Stack bytes	16	16	18	22
	Cycles	1410	1394	805	820
`sprintf_min (s, "%d", 12345)`	Flash bytes	1218 (1106)	1086 (976)	1082 (972)	1102
	Stack bytes	59	59	59	62
	Cycles	2077	1940	1935	1887
`sprintf (s, "%d", 12345)`	Flash bytes	1636 (1524)	1494 (1384)	1478 (1368)	1514
	Stack bytes	59	59	61	64
	Cycles	1885	1795	1796	1734
`sprintf_flt (s, "%e", 1.2345)`	Flash bytes	3250 (3138)	3000 (2890)	2976 (2866)	3088 (2978)
	Stack bytes	68	68	69	72
	Cycles	3089	2866	2873	2695
`sscanf_min ("12345", "%d", &i)`	Flash bytes	2804 (2692)	2502 (2392)	2498 (2388)	2728
	Stack bytes	57	57	57	61
	Cycles	1380	1294	1294	1311
`sscanf ("12345", "%d", &i)`	Flash bytes	1798 (1686)	1614 (1504)	1614 (1504)	1716
	Stack bytes	57	57	57	61
	Cycles	1380	1294	1294	1311
`sscanf ("point,color", "%[a-z]", s)`	Flash bytes	1798 (1686)	1614 (1504)	1614 (1504)	1716
	Stack bytes	90	90	90	94
	Cycles	2748	2601	2601	2502
`sscanf_flt ("1.2345", "%e", &x)`	Flash bytes	4866 (4754)	4438 (4328)	4414 (4304)	4714 (4604)
	Stack bytes	40	40	40	43
	Cycles	410	364	364	341
`strtof ("1.2345", &end)`	Flash bytes	1648 (1536)	1490 (1380)	1462 (1352)	1698 (1588)
	Stack bytes	24	24	24	37
	Cycles	1289	1172	969	1282
`strtol ("12345", &end, 0)`	Flash bytes	390	368	350	364
	Stack bytes	14	14	12	15
	Cycles	606	583	351	373
`strtoll ("12345", &end, 0)`	Flash bytes	578 (466)	540 (430)	516 (406)	540 (430)
	Stack bytes	20	20	18	21
	Cycles	832	785	488	513

12.2 Math Functions from libm

The following tables contain benchmark values for some floating-point functions over the indicated range(s) of input values.

Notice that the values for relative error and the Worst Case Execution Time $\text{Cycles}_{\text{max}}$ are only *lower bounds*. The best achievable accuracy for IEEE single with its 23 fractional bits in the mantissa is $\log_{10}(2^{-24} \approx 6{\cdot}10^{-8}) \approx -7.22$.

The poor performance of `sinf`, `cosf` and `tanf` occurs for values that are close to the poles (if any) or close to the non-zero zeros.

Table 25 libm benchmarks for ATmega128 (avr51, with MUL)

Function	Size	x_0	x_1	$Cycles_{avr}$	$Cycles_{max}$	$\log_{10}(Err_{max})$
acosf	1102	-1	1	1957	2464	-6.66
asinf	1092	-1	1	1896	2454	-6.47
atanf	1058	-10	10	2879	3073	-6.92
cbrtf	514	-1e+06	1e+06	2573	2665	-6.92
ceilf	258	-1e+05	1e+05	108	177	-7.22
cosf	904	-1.57	1.58	1775	2126	**-2.86**
coshf	1366	-20	20	3053	3439	-5.78
expf	1320	-20	20	2588	3247	-5.78
floorf	258	-1e+05	1e+05	108	180	-7.22
frexpf	154	-1e+05	1e+05	40	40	-7.22
logf	1076	0	100	2392	2866	-6.69
log10f	1076	0	100	2397	2866	-6.62
log2f	1052	0	100	2252	2723	-6.83
modff	484	-1e+05	1e+05	365	456	-7.22
roundf	236	-1e+05	1e+05	111	156	-7.22
sinf	910	0	3.15	1744	2146	**-3.67**
sincosf	976	-1.57	3.15	3482	3883	**-3.43**
sinhf	1466	-20	20	3043	3461	-5.78
sqrtf	256	0	1e+06	474	510	-7.22
tanf	1178	0	3.15	2178	2946	**-2.86**
tanhf	1494	-20	20	3148	3620	-6.37
truncf	234	-1000	1000	140	178	-7.22

Table 26 libm benchmarks for ATmega128 (avr51, with MUL)

Function	Size	x_0	x_1	y_0	y_1	$Cycles_{avr}$	$Cycles_{max}$	$\log_{10}(Err_{max})$
+	380	-1e+10	1e+10	-1e+10	1e+10	102	256	-7.22
*	380	-1e+10	1e+10	-1e+10	1e+10	129	139	-7.22
/	390	-1e+10	1e+10	-1e+10	1e+10	469	501	-7.22
atan2f	1206	-10	10	-10	10	2882	3455	-6.82
fdimf	446	-1e+10	1e+10	-1e+10	1e+10	75	218	-7.22
fmaxf	62	-1e+10	1e+10	-1e+10	1e+10	30	34	$-\infty$
fminf	62	-1e+10	1e+10	-1e+10	1e+10	30	34	$-\infty$
fmodf	312	-1e+10	1e+10	-1e+10	1e+10	88	324	-7.22
hypotf	1092	-1e+10	1e+10	-1e+10	1e+10	850	927	-6.92
ldexpf	238	-1e+10	1e+10	-10	10	40	40	-7.22
powf	1858	0	1e+04	-10	10	5182	5833	**-4.65**
__builtin_powf	732	0	1e+04	-10	10	648	1223	-6.39

For devices without MUL instruction, the following applies:

- The execution times for multiplication and for the transcendental functions are roughly twice the time for devices that have MUL.
- The execution times for the remaining functions are roughly the same.
- The maximal relative errors are the same, i.e. independent of MUL.

Table 27 libm benchmarks for AT90S8515 (avr2, no MUL)

Function	Size	x_0	x_1	$Cycles_{avr}$	$Cycles_{max}$	$\log_{10}(Err_{max})$
acosf	1104	-1	1	3513	3888	-6.66
asinf	1096	-1	1	3452	3879	-6.47
atanf	1054	-10	10	5280	5541	-6.92
cbrtf	538	-1e+06	1e+06	2702	2795	-6.92
ceilf	250	-1e+05	1e+05	105	174	-7.22
cosf	906	-1.57	1.58	3441	3798	**-2.86**
coshf	1348	-20	20	4966	5346	-5.78
expf	1312	-20	20	4512	5140	-5.78
floorf	250	-1e+05	1e+05	105	177	-7.22
frexpf	150	-1e+05	1e+05	39	39	-7.22
logf	1076	0	100	4562	5023	-6.69
log10f	1076	0	100	4568	5035	-6.62
log2f	1060	0	100	4205	4632	-6.83
modff	490	-1e+05	1e+05	365	456	-7.22
roundf	230	-1e+05	1e+05	109	154	-7.22
sinf	912	0	3.15	3408	3818	**-3.67**
sincosf	978	-1.57	3.15	6813	7225	**-3.43**
sinhf	1434	-20	20	4941	5358	-5.78
sqrtf	252	0	1e+06	474	510	-7.22
tanf	1164	0	3.15	4080	4785	**-2.86**
tanhf	1462	-20	20	5055	5544	-6.37
truncf	226	-1000	1000	137	175	-7.22

Table 28 libm benchmarks for AT90S8515 (avr2, no MUL)

Function	Size	x_0	x_1	y_0	y_1	$Cycles_{avr}$	$Cycles_{max}$	$\log_{10}(Err_{max})$
+	376	-1e+10	1e+10	-1e+10	1e+10	102	253	-7.22
*	378	-1e+10	1e+10	-1e+10	1e+10	346	386	-7.22
/	374	-1e+10	1e+10	-1e+10	1e+10	467	499	-7.22
atan2f	1192	-10	10	-10	10	5287	5844	-6.82
fdimf	436	-1e+10	1e+10	-1e+10	1e+10	74	219	-7.22
fmaxf	66	-1e+10	1e+10	-1e+10	1e+10	31	36	$-\infty$
fminf	66	-1e+10	1e+10	-1e+10	1e+10	30	36	$-\infty$
fmodf	302	-1e+10	1e+10	-1e+10	1e+10	86	322	-7.22
hypotf	1068	-1e+10	1e+10	-1e+10	1e+10	1297	1387	-6.92
ldexpf	232	-1e+10	1e+10	-10	10	39	39	-7.22
powf	1820	0	1e+04	-10	10	9490	10180	**-4.65**
__builtin_powi	734	0	1e+04	-10	10	1143	2140	-6.39

12.3 Math Functions for IEEE double from LibF7

The following tables contain benchmark values for some IEEE double floating-point functions over the indicated range(s) of input values. LibF7 is a IEEE double implementation hosted by libgcc since GCC v10.

The code sizes include all dependencies with the exception of potential prologue and epilogue routines (__↩ prologue_saves__, __epilogue_restores__).

The sizes of functions don't add up. For example, sinl, cosl, asinl, acosl and sqrtl together occupy only 4744 bytes of code *including* the prologue and epilogue routines. With `-mrelax` the code size reduces further to around 4400 bytes.

Notice that the values for relative error and the Worst Case Execution Time $\text{Cycles}_{\text{max}}$ are only *lower bounds*. The best achievable accuracy for IEEE double with its 52 fractional bits in the mantissa is $\log_{10}(2^{-53} \approx 1.1 \cdot 10^{-16}) \approx -15.95$.

Table 29 LibF7 Benchmarks for ATmega128 (avr51, with MUL)

Function	Size	x_0	x_1	$\text{Cycles}_{\text{avr}}$	$\text{Cycles}_{\text{max}}$	$\log_{10}(\text{Err}_{\text{max}})$
acosl	3342	-1	1	16235	17997	-15.65
asinl	3342	-1	1	16124	18008	-15.65
atanl	2820	-10	10	20363	21255	-15.65
cbrtl	4102	-1e+06	1e+06	32368	33415	-15.33
ceill	1772	-1e+05	1e+05	1430	1768	-15.95
cosl	3900	-1.57	1.58	11241	13649	-15.66
coshl	3654	-20	20	20036	21466	-15.65
expl	3562	-20	20	16965	18417	-15.65
floorl	1710	-1e+05	1e+05	1354	1690	-15.95
frexpl	1256	-1e+05	1e+05	748	765	-15.95
logl	2944	0	100	15099	15825	-15.65
log10l	2954	0	100	15746	16437	-15.59
log2l	2954	0	100	15751	16418	-15.65
roundl	1796	-1e+05	1e+05	1664	1760	-15.95
sinl	3898	0	3.15	11976	14028	-15.65
sincosl	3880	-1.57	3.15	19913	22412	-15.65
sinhl	3862	-20	20	19842	21536	-15.51
sqrtl	1410	0	1e+06	3009	3087	-15.65
tanl	3946	0	3.15	22448	24870	-15.65
tanhl	3678	-20	20	20646	22347	-14.70
truncl	1710	-1000	1000	1118	1173	-15.95

Table 30 LibF7 Benchmarks for ATmega128 (avr51, with MUL)

Function	Size	x_0	x_1	y_0	y_1	$\text{Cycles}_{\text{avr}}$	$\text{Cycles}_{\text{max}}$	$\log_{10}(\text{Err}_{\text{max}})$
+	1470	-1e+10	1e+10	-1e+10	1e+10	1531	1667	-15.95
*	1600	-1e+10	1e+10	-1e+10	1e+10	1620	1661	-15.65
/	1572	-1e+10	1e+10	-1e+10	1e+10	3563	3756	-15.65
atan2l	3208	-10	10	-10	10	21115	22010	-15.65
fdiml	1730	-1e+10	1e+10	-1e+10	1e+10	1371	1802	-15.95
fmaxl	234	-1e+10	1e+10	-1e+10	1e+10	169	188	$-\infty$
fminl	234	-1e+10	1e+10	-1e+10	1e+10	170	189	$-\infty$
fmodl	2808	-1e+10	1e+10	-1e+10	1e+10	5175	5947	-15.95
hypotl	2312	-1e+10	1e+10	-1e+10	1e+10	5006	5166	-15.66
ldexpl	1254	-1e+10	1e+10	-10	10	738	756	-15.95
powl	4080	0	1e+04	-10	10	32440	34066	-14.46
__builtin_pow	2224	0	1e+04	-10	10	3396	6173	-15.12

12.4 Fixed-Point Functions from <stdfix.h>

The following tables contain benchmark values for some fixed-point functions over the indicated range of input values.

- V+ denotes the smallest value that is larger than V for the considered fixed-point type. Similarly, V− denotes the largest value that is smaller than V for the considered type.
- The code sizes include all dependencies.

Notice that the values for absolute error Err_{max}, and the Worst Case Execution Times $Cycles_{max}$ are only *lower bounds*.

Table 31 Fixed-Point Benchmarks for ATmega128 (avr51, with MUL)

Function	Size	x_0	x_1	$Cycles_{avr}$	$Cycles_{max}$	Err_{max}
log2uhk	78	0+	10	52	75	1.25e-02
log21puhr	32	0	1-	22	22	4.28e-03
sinuhk_deg	272	0	256-	53	55	6.44e-05
cosuhk_deg	318	0	256-	73	77	6.44e-05
sqrthk	92	0	256-	293	309	3.91e-03
sqrtuhk	70	0	256-	277	293	3.91e-03
sqrthr	42	0	1-	100	100	3.84e-03
sqrtuhr	38	0	1-	98	98	3.90e-03
acosk	404	-1	1	416	572	5.39e-05
acosuk	328	0	1	385	526	4.51e-05
asink	386	-1	1	414	589	4.95e-05
asinuk	328	0	1	381	539	4.43e-05
atank	368	-1	1	242	264	4.04e-05
atank	368	1	10	888	913	4.65e-05
atanuk	298	0	1	203	206	2.52e-05
atanur	152	0	1-	188	191	2.46e-05
exp2k	214	-10	10	255	317	1.09e-02
exp2uk	164	0	10	245	293	1.06e-02
exp2m1ur	112	0	1-	177	180	2.13e-05
log2uk	184	0+	10	257	305	6.03e-05
log21pur	114	0	1-	212	215	2.87e-05
cospi2k	182	-4	4	249	258	4.52e-05
sinpi2k	182	-4	4	247	256	4.54e-05
sinpi2ur	120	0	1-	215	219	2.80e-05
sqrtk	128	0	65536-	572	635	1.53e-05
sqrtuk	94	0	65536-	550	613	1.53e-05
sqrtr	88	0	1-	290	313	1.53e-05
sqrtur	66	0	1-	274	297	1.53e-05

Table 32 Fixed-Point Benchmarks for ATtiny88 (avr25, no MUL)

Function	Size	x_0	x_1	$Cycles_{avr}$	$Cycles_{max}$	Err_{max}
log2uhk	114	0+	10	339	380	1.25e-02

Function	Size	x_0	x_1	$Cycles_{avr}$	$Cycles_{max}$	Err_{max}
log21puhr	70	0	1-	312	332	4.28e-03
sqrthk	88	0	256-	291	307	3.91e-03
sqrtuhk	68	0	256-	276	292	3.91e-03
sqrthr	54	0	1-	102	105	3.84e-03
sqrtuhr	50	0	1-	100	104	3.90e-03
acosk	446	-1	1	1097	1233	5.39e-05
acosuk	378	0	1	1070	1197	4.51e-05
asink	428	-1	1	1095	1249	4.95e-05
asinuk	378	0	1	1066	1210	4.43e-05
atank	412	-1	1	877	973	4.04e-05
atank	412	1	10	1538	1630	4.65e-05
atanuk	352	0	1	844	924	2.52e-05
atanur	212	0	1-	830	910	2.46e-05
exp2k	234	-10	10	993	1074	1.09e-02
exp2uk	184	0	10	982	1050	1.06e-02
exp2m1ur	134	0	1-	916	939	2.13e-05
log2uk	214	0+	10	1222	1294	6.03e-05
log21pur	146	0	1-	1187	1228	2.87e-05
cospi2k	204	-4	4	1235	1285	4.52e-05
sinpi2k	204	-4	4	1233	1283	4.54e-05
sinpi2ur	146	0	1-	1203	1248	2.80e-05
sqrtk	124	0	65536-	570	633	1.53e-05
sqrtuk	92	0	65536-	549	612	1.53e-05
sqrtr	84	0	1-	288	311	1.53e-05
sqrtur	64	0	1-	273	296	1.53e-05

13 Problems with Reordering Code

13.1 Problems with Reordering Code

Author

Jan Waclawek

This is an essay about PR53049.

Programs contain sequences of statements, and a naive compiler would execute them exactly in the order as they are written. But an optimizing compiler is free to *reorder* the statements — or even parts of them — if the resulting "net effect" is the same. The "measure" of the "net effect" is what the standard calls "side effects", and is accomplished exclusively through accesses (reads and writes) to variables qualified as `volatile`. So, as long as all volatile reads and writes are to the same addresses and in the same order (and writes write the same values), the program is correct, regardless of other operations in it. One important point to note here is, that time duration between consecutive volatile accesses is not considered at all.

Unfortunately, there are also operations which are not covered by volatile accesses. An example of this in AVR-GCC/AVR-LibC are the `cli()` and `sei()` macros defined in <avr/interrupt.h>, which convert directly to the respective assembler mnemonics through the `__asm__()` statement. They constitute a variable access by means of their memory clobber, and they are (implicitly) volatile because they don't have an output operand. So the compiler may not reorder these inline asm statements with respect to other memory accesses or volatile actions. However, such asm statement may still be reordered with other statement that are neither volatile nor access memory.

Note that even a volatile asm instruction can be moved relative to other code, including across (expensive) arithmetic and jump instructions [...]

See also

http://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html

However, not even a volatile memory barrier like

```
__asm __volatile__ ("" ::: "memory");
```

keeps GCC from reordering non-volatile, non-memory accesses across such barriers. Peter Dannegger provided a nice example of this effect:

```
#define cli() __asm volatile( "cli" ::: "memory" )
#define sei() __asm volatile( "sei" ::: "memory" )

unsigned int ivar;

void test2 (unsigned int val)
{
  val = 65535U / val;

  cli();

  ivar = val;

  sei();
}
```

avr-gcc v5.4 or v14 compile with optimisations switched on (`-Os`) to

```
00000112 <test2>:
 112: bc 01        movw r22, r24
 114: f8 94        cli
 116: 8f ef        ldi r24, 0xFF ; 255
 118: 9f ef        ldi r25, 0xFF ; 255
 11a: 0e 94 96 00  call 0x12c ; 0x12c <__udivmodhi4>
 11e: 70 93 01 02  sts 0x0201, r23
 122: 60 93 00 02  sts 0x0200, r22
 126: 78 94        sei
 128: 08 95        ret
```

where the potentially slow division is moved across `cli()`, resulting in interrupts to be disabled longer than intended. Note, that the volatile access occurs in order with respect to `cli()` or `sei()`; so the "net effect" required by the standard is achieved as intended, it is "only" the timing which is off. However, for most of embedded applications, timing is an important, sometimes critical factor.

See also

https://www.mikrocontroller.net/topic/65923

Unfortunately, at the moment, in avr-gcc (nor in the C standard), there is no mechanism to enforce complete match of written and executed code ordering — except maybe of switching the optimization completely off (`-O0`), or writing all the critical code in assembly.

Note

The artifact with the `__udivmodhi4` function is specific to avr-gcc and how the compiler represents the division internally. On other target platforms that are using a library function for division or whatever expensive operation, this effect will not occur. The reason is that avr-gcc does not represent the library call as a function call but rather like an ordinary instruction. Outcome is that the GCC middle-end concludes that the division is cheap (because the backend has an instruction for it) but in fact it's not.

A work around for the code from above would be to enforce that the division happens prior to the `cli()`:

```
val = 65535U / val;
__asm __volatile__ ("" : "+r" (val));
cli();
```

- The `volatile` forces the asm statement prior to the `cli`.
- The asm has `val` as input operand, hence the division must be carried out prior to the asm because `val` is set by the division.

Notice that this work around does not work in general due to a variety of reasons:

- The division might be located in an inlined function.
- The variable might be read-only or may not be appropriate as an asm operand.
- There may be more such instruction prior to the division, and it is not practical to treat all of them like this.

To sum it up:

- volatile memory barriers don't ensure statements with no volatile accesses to be reordered across the barrier

14 Acknowledgments

This document tries to tie together the labors of a large group of people. Without these individuals' efforts, we wouldn't have a terrific, ***free*** set of tools to develop AVR projects. We all owe thanks to:

- The GCC Team, which produced a very capable set of development tools for an amazing number of platforms and processors.
- Denis Chertykov [denisc@overta.ru] for making the AVR-specific changes to the GNU tools.
- Denis Chertykov and Marek Michalkiewicz [marekm@linux.org.pl] for developing the standard libraries and startup code for **AVR-GCC**.
- Uros Platise for developing the AVR programmer tool, **uisp**.
- Joerg Wunsch [joerg@FreeBSD.ORG] for adding all the AVR development tools to the FreeBSD [http://www.freebsd.org] ports tree and for providing the basics for the demo project.
- Brian Dean [bsd@bsdhome.com] for developing **avrdude**, an alternative to **uisp**. **Avrdude** was previously called **avrprog**.
- Eric Weddington [eweddington@cso.atmel.com] for maintaining the **WinAVR** package and thus making the continued improvements to the open source AVR toolchain available to many users.
- Rich Neswold for writing the original avr-tools document (which he graciously allowed to be merged into this document) and his improvements to the demo project.
- Theodore A. Roth for having been a long-time maintainer of many of the tools (**AVR-LibC**, the AVR port of **GDB**, **AVaRICE**, **uisp**, **avrdude**).
- All the people who currently maintain the tools, and/or have submitted suggestions, patches and bug reports. (See the AUTHORS files of the various tools.)
- And lastly, all the users who use the software. If nobody used the software, we would probably not be very motivated to continue to develop it. Keep those bug reports coming. ;-)

15 Global Index

16 Deprecated List

Global cbi (port, bit)

Global enable_external_int (mask)

Global inb (port)

Global inp (port)

Global INTERRUPT (signame)

Global ISR_ALIAS (vector, target_vector)

For new code, the use of ISR(..., ISR_ALIASOF(...)) or ISR_N is recommended. Notice that using ISR_N does *not* impose a JMP/RJMP overhead.

Global outb (port, val)

Global outp (val, port)

Global sbi (port, bit)

Global SIGNAL (vector)

Do not use SIGNAL() in new code. Use ISR() or ISR_N() instead.

Global timer_enable_int (unsigned char ints)

17 Module Index

17.1 Modules

Here is a list of all modules:

18 Data Structure Index

18.1 Data Structures

Here are the data structures with brief descriptions:

19 File Index

19.1 File List

Here is a list of all documented files with brief descriptions:

20 Module Documentation

20.1 <alloca.h>: Allocate space in the stack

Functions

- void * alloca (size_t __size)

20.1.1 Detailed Description

```
#include <alloca.h>
```

20.1.2 Function Documentation

20.1.2.1 alloca()

```
void * alloca (
            size_t __size )
```

Allocate *__size* bytes of space in the stack frame of the caller.

This temporary space is automatically freed when the function that called alloca() returns to its caller. AVR-LibC defines the alloca() as a macro, which is translated into the inlined `__builtin_alloca()` function. The fact that the code is inlined, means that it is impossible to take the address of this function, or to change its behaviour by linking with a different library.

Returns

alloca() returns a pointer to the beginning of the allocated space. If the allocation causes stack overflow, program behaviour is undefined.

Warning

Avoid use alloca() inside the list of arguments of a function call.

20.2 <assert.h>: Diagnostics

Macros

- #define assert(expression)

20.2.1 Detailed Description

```
#include <assert.h>
```

This header file defines a debugging aid.

As there is no standard error output stream available for many applications using this library, the generation of a printable error message is not enabled by default. These messages will only be generated if the application defines the macro

```
__ASSERT_USE_STDERR
```

before including the `<assert.h>` header file. By default, only abort() will be called to halt the application.

20.2.2 Macro Definition Documentation

20.2.2.1 assert

```
#define assert(
             expression )
```

Parameters

expression	Expression to test for.

The assert() macro tests the given expression and if it is false, the calling process is terminated by calling abort(). When the macro __ASSERT_USE_STDERR was defined prior to including <assert.h>, then a diagnostic message is written to stderr.

If expression is true, the assert() macro does nothing.

The assert() macro may be removed at compile time by defining NDEBUG as a macro (e.g., by using the compiler option -DNDEBUG).

20.3 <ctype.h>: Character Operations

Character classification routines

These functions perform character classification. They return true or false status depending whether the character passed to the function falls into the function's classification (i.e. isdigit() returns true if its argument is any value '0' though '9', inclusive). If the input is not an unsigned char value, all of this function return false.

- int isalnum (int __c)
- int isalpha (int __c)
- int isascii (int __c)
- int isblank (int __c)
- int iscntrl (int __c)
- int isdigit (int __c)
- int isgraph (int __c)
- int islower (int __c)
- int isprint (int __c)
- int ispunct (int __c)
- int isspace (int __c)
- int isupper (int __c)
- int isxdigit (int __c)

Character conversion routines

This realization permits all possible values of integer argument. The toascii() function clears all highest bits. The tolower() and toupper() functions return an input argument as is, if it is not an unsigned char value.

- int toascii (int __c)
- int tolower (int __c)
- int toupper (int __c)

20.3.1 Detailed Description

These functions perform various operations on characters.

```
#include <ctype.h>
```

20.3.2 Function Documentation

20.3.2.1 isalnum()

```
int isalnum (
            int __c )
```

Checks for an alphanumeric character. It is equivalent to `(isalpha(c) || isdigit(c))`.

20.3.2.2 isalpha()

```
int isalpha (
            int __c )
```

Checks for an alphabetic character. It is equivalent to `(isupper(c) || islower(c))`.

20.3.2.3 isascii()

```
int isascii (
            int __c )
```

Checks whether `c` is a 7-bit unsigned char value that fits into the ASCII character set.

20.3.2.4 isblank()

```
int isblank (
            int __c )
```

Checks for a blank character, that is, a space or a tab.

20.3.2.5 iscntrl()

```
int iscntrl (
            int __c )
```

Checks for a control character.

20.3.2.6 isdigit()

```
int isdigit (
            int __c )
```

Checks for a digit (0 through 9).

20.3.2.7 isgraph()

```
int isgraph (
            int __c )
```

Checks for any printable character except space.

20.3.2.8 islower()

```
int islower (
            int __c )
```

Checks for a lower-case character.

20.3.2.9 isprint()

```
int isprint (
            int __c )
```

Checks for any printable character including space.

20.3.2.10 ispunct()

```
int ispunct (
            int __c )
```

Checks for any printable character which is not a space or an alphanumeric character.

20.3.2.11 isspace()

```
int isspace (
            int __c )
```

Checks for white-space characters. For the AVR-LibC library, these are: space, form-feed ('\f'), newline ('\n'), carriage return ('\r'), horizontal tab ('\t'), and vertical tab ('\v').

20.3.2.12 isupper()

```
int isupper (
            int __c )
```

Checks for an uppercase letter.

20.3.2.13 isxdigit()

```
int isxdigit (
            int __c )
```

Checks for a hexadecimal digits, i.e. one of 0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F.

20.3.2.14 toascii()

```
int toascii (
            int __c )
```

Converts `c` to a 7-bit unsigned char value that fits into the ASCII character set, by clearing the high-order bits.

Warning

Many people will be unhappy if you use this function. This function will convert accented letters into random characters.

20.3.2.15 tolower()

```
int tolower (
            int __c )
```

Converts the letter c to lower case, if possible.

20.3.2.16 toupper()

```
int toupper (
            int __c )
```

Converts the letter c to upper case, if possible.

20.4 <errno.h>: System Errors

Macros

- #define EDOM 33
- #define ERANGE 34
- #define EINVAL 35

Variables

- int errno

20.4.1 Detailed Description

```
#include <errno.h>
```

Some functions in the library set the global variable errno when an error occurs. The file, <errno.h>, provides symbolic names for various error codes.

20.4.2 Macro Definition Documentation

20.4.2.1 EDOM

```
#define EDOM 33
```

Domain error.

20.4.2.2 EINVAL

```
#define EINVAL 35
```

Invalid value error.

20.4.2.3 ERANGE

```
#define ERANGE 34
```

Range error.

20.4.3 Variable Documentation

20.4.3.1 errno

```
int errno  [extern]
```

Error code for last error encountered by library.

The variable `errno` holds the last error code encountered by a library function. This variable must be cleared by the user prior to calling a library function.

Warning

> The `errno` global variable is not safe to use in a threaded or multi-task system. A race condition can occur if a task is interrupted between the call which sets `error` and when the task examines `errno`. If another task changes `errno` during this time, the result will be incorrect for the interrupted task.

20.5 <inttypes.h>: Integer Type conversions

Far pointers for memory access > 64K

- typedef int32_t int_farptr_t
- typedef uint32_t uint_farptr_t

macros for printf and scanf format specifiers

For C++, these are only included if __STDC_LIMIT_MACROS is defined before including <inttypes.h>.

- #define PRId8 "d"
- #define PRIdLEAST8 "d"
- #define PRIdFAST8 "d"
- #define PRIi8 "i"
- #define PRIiLEAST8 "i"
- #define PRIiFAST8 "i"
- #define PRId16 "d"
- #define PRIdLEAST16 "d"
- #define PRIdFAST16 "d"
- #define PRIi16 "i"
- #define PRIiLEAST16 "i"
- #define PRIiFAST16 "i"
- #define PRId32 "ld"
- #define PRIdLEAST32 "ld"

- #define PRIdFAST32 "ld"
- #define PRIi32 "li"
- #define PRIiLEAST32 "li"
- #define PRIiFAST32 "li"
- #define PRIdPTR PRId16
- #define PRIiPTR PRIi16
- #define PRIo8 "o"
- #define PRIoLEAST8 "o"
- #define PRIoFAST8 "o"
- #define PRIu8 "u"
- #define PRIuLEAST8 "u"
- #define PRIuFAST8 "u"
- #define PRIx8 "x"
- #define PRIxLEAST8 "x"
- #define PRIxFAST8 "x"
- #define PRIX8 "X"
- #define PRIXLEAST8 "X"
- #define PRIXFAST8 "X"
- #define PRIo16 "o"
- #define PRIoLEAST16 "o"
- #define PRIoFAST16 "o"
- #define PRIu16 "u"
- #define PRIuLEAST16 "u"
- #define PRIuFAST16 "u"
- #define PRIx16 "x"
- #define PRIxLEAST16 "x"
- #define PRIxFAST16 "x"
- #define PRIX16 "X"
- #define PRIXLEAST16 "X"
- #define PRIXFAST16 "X"
- #define PRIo32 "lo"
- #define PRIoLEAST32 "lo"
- #define PRIoFAST32 "lo"
- #define PRIu32 "lu"
- #define PRIuLEAST32 "lu"
- #define PRIuFAST32 "lu"
- #define PRIx32 "lx"
- #define PRIxLEAST32 "lx"
- #define PRIxFAST32 "lx"
- #define PRIX32 "lX"
- #define PRIXLEAST32 "lX"
- #define PRIXFAST32 "lX"
- #define PRIoPTR PRIo16
- #define PRIuPTR PRIu16
- #define PRIxPTR PRIx16
- #define PRIXPTR PRIX16
- #define SCNd8 "hhd"
- #define SCNdLEAST8 "hhd"
- #define SCNdFAST8 "hhd"
- #define SCNi8 "hhi"
- #define SCNiLEAST8 "hhi"
- #define SCNiFAST8 "hhi"
- #define SCNd16 "d"
- #define SCNdLEAST16 "d"
- #define SCNdFAST16 "d"

- #define SCNi16 "i"
- #define SCNiLEAST16 "i"
- #define SCNiFAST16 "i"
- #define SCNd32 "ld"
- #define SCNdLEAST32 "ld"
- #define SCNdFAST32 "ld"
- #define SCNi32 "li"
- #define SCNiLEAST32 "li"
- #define SCNiFAST32 "li"
- #define SCNdPTR SCNd16
- #define SCNiPTR SCNi16
- #define SCNo8 "hho"
- #define SCNoLEAST8 "hho"
- #define SCNoFAST8 "hho"
- #define SCNu8 "hhu"
- #define SCNuLEAST8 "hhu"
- #define SCNuFAST8 "hhu"
- #define SCNx8 "hhx"
- #define SCNxLEAST8 "hhx"
- #define SCNxFAST8 "hhx"
- #define SCNo16 "o"
- #define SCNoLEAST16 "o"
- #define SCNoFAST16 "o"
- #define SCNu16 "u"
- #define SCNuLEAST16 "u"
- #define SCNuFAST16 "u"
- #define SCNx16 "x"
- #define SCNxLEAST16 "x"
- #define SCNxFAST16 "x"
- #define SCNo32 "lo"
- #define SCNoLEAST32 "lo"
- #define SCNoFAST32 "lo"
- #define SCNu32 "lu"
- #define SCNuLEAST32 "lu"
- #define SCNuFAST32 "lu"
- #define SCNx32 "lx"
- #define SCNxLEAST32 "lx"
- #define SCNxFAST32 "lx"
- #define SCNoPTR SCNo16
- #define SCNuPTR SCNu16
- #define SCNxPTR SCNx16

20.5.1 Detailed Description

```
#include <inttypes.h>
```

This header file includes the exact-width integer definitions from <stdint.h>, and extends them with additional facilities provided by the implementation.

Currently, the extensions include two additional integer types that could hold a "far" pointer (i.e. a code pointer that can address more than 64 KB), as well as standard names for all printf and scanf formatting options that are supported by the <stdio.h>: Standard IO facilities. As the library does not support the full range of conversion specifiers from ISO 9899:1999, only those conversions that are actually implemented will be listed here.

The idea behind these conversion macros is that, for each of the types defined by <stdint.h>, a macro will be supplied that portably allows formatting an object of that type in printf() or scanf() operations. Example:

```
#include <inttypes.h>

uint8_t smallval;
int32_t longval;
...
printf("The hexadecimal value of smallval is %" PRIx8
       ", the decimal value of longval is %" PRId32 ".\n",
   smallval, longval);
```

20.5.2 Macro Definition Documentation

20.5.2.1 PRId16

```
#define PRId16 "d"
```

decimal printf format for int16_t

20.5.2.2 PRId32

```
#define PRId32 "ld"
```

decimal printf format for int32_t

20.5.2.3 PRId8

```
#define PRId8 "d"
```

decimal printf format for int8_t

20.5.2.4 PRIdFAST16

```
#define PRIdFAST16 "d"
```

decimal printf format for int_fast16_t

20.5.2.5 PRIdFAST32

```
#define PRIdFAST32 "ld"
```

decimal printf format for int_fast32_t

20.5.2.6 PRIdFAST8

```
#define PRIdFAST8 "d"
```

decimal printf format for int_fast8_t

20.5.2.7 PRIdLEAST16

```
#define PRIdLEAST16 "d"
```

decimal printf format for int_least16_t

20.5.2.8 PRIdLEAST32

```
#define PRIdLEAST32 "ld"
```

decimal printf format for int_least32_t

20.5.2.9 PRIdLEAST8

```
#define PRIdLEAST8 "d"
```

decimal printf format for int_least8_t

20.5.2.10 PRIdPTR

```
#define PRIdPTR PRId16
```

decimal printf format for intptr_t

20.5.2.11 PRIi16

```
#define PRIi16 "i"
```

integer printf format for int16_t

20.5.2.12 PRIi32

```
#define PRIi32 "li"
```

integer printf format for int32_t

20.5.2.13 PRIi8

```
#define PRIi8 "i"
```

integer printf format for int8_t

20.5.2.14 PRIiFAST16

```
#define PRIiFAST16 "i"
```

integer printf format for int_fast16_t

20.5.2.15 PRIiFAST32

```
#define PRIiFAST32 "li"
```

integer printf format for int_fast32_t

20.5.2.16 PRIiFAST8

```
#define PRIiFAST8 "i"
```

integer printf format for int_fast8_t

20.5.2.17 PRIiLEAST16

```
#define PRIiLEAST16 "i"
```

integer printf format for int_least16_t

20.5.2.18 PRIiLEAST32

```
#define PRIiLEAST32 "li"
```

integer printf format for int_least32_t

20.5.2.19 PRIiLEAST8

```
#define PRIiLEAST8 "i"
```

integer printf format for int_least8_t

20.5.2.20 PRIiPTR

```
#define PRIiPTR PRIi16
```

integer printf format for intptr_t

20.5.2.21 PRIo16

```
#define PRIo16 "o"
```

octal printf format for uint16_t

20.5.2.22 PRIo32

```
#define PRIo32 "lo"
```

octal printf format for uint32_t

20.5.2.23 PRIo8

```
#define PRIo8 "o"
```

octal printf format for uint8_t

20.5.2.24 PRIoFAST16

```
#define PRIoFAST16 "o"
```

octal printf format for uint_fast16_t

20.5.2.25 PRIoFAST32

```
#define PRIoFAST32 "lo"
```

octal printf format for uint_fast32_t

20.5.2.26 PRIoFAST8

```
#define PRIoFAST8 "o"
```

octal printf format for uint_fast8_t

20.5.2.27 PRIoLEAST16

```
#define PRIoLEAST16 "o"
```

octal printf format for uint_least16_t

20.5.2.28 PRIoLEAST32

```
#define PRIoLEAST32 "lo"
```

octal printf format for uint_least32_t

20.5.2.29 PRIoLEAST8

```
#define PRIoLEAST8 "o"
```

octal printf format for uint_least8_t

20.5.2.30 PRIoPTR

```
#define PRIoPTR PRIo16
```

octal printf format for uintptr_t

20.5.2.31 PRIu16

```
#define PRIu16 "u"
```

decimal printf format for uint16_t

20.5.2.32 PRIu32

```
#define PRIu32 "lu"
```

decimal printf format for uint32_t

20.5.2.33 PRIu8

```
#define PRIu8 "u"
```

decimal printf format for uint8_t

20.5.2.34 PRIuFAST16

```
#define PRIuFAST16 "u"
```

decimal printf format for uint_fast16_t

20.5.2.35 PRIuFAST32

```
#define PRIuFAST32 "lu"
```

decimal printf format for uint_fast32_t

20.5.2.36 PRIuFAST8

```
#define PRIuFAST8 "u"
```

decimal printf format for uint_fast8_t

20.5.2.37 PRIuLEAST16

```
#define PRIuLEAST16 "u"
```

decimal printf format for uint_least16_t

20.5.2.38 PRIuLEAST32

```
#define PRIuLEAST32 "lu"
```

decimal printf format for uint_least32_t

20.5.2.39 PRIuLEAST8

```
#define PRIuLEAST8 "u"
```

decimal printf format for uint_least8_t

20.5.2.40 PRIuPTR

```
#define PRIuPTR PRIu16
```

decimal printf format for uintptr_t

20.5.2.41 PRIx16

```
#define PRIx16 "x"
```

hexadecimal printf format for uint16_t

20.5.2.42 PRIX16

```
#define PRIX16 "X"
```

uppercase hexadecimal printf format for uint16_t

20.5.2.43 PRIx32

```
#define PRIx32 "lx"
```

hexadecimal printf format for uint32_t

20.5.2.44 PRIX32

```
#define PRIX32 "lX"
```

uppercase hexadecimal printf format for uint32_t

20.5.2.45 PRIx8

```
#define PRIx8 "x"
```

hexadecimal printf format for uint8_t

20.5.2.46 PRIX8

```
#define PRIX8 "X"
```

uppercase hexadecimal printf format for uint8_t

20.5.2.47 PRIxFAST16

```
#define PRIxFAST16 "x"
```

hexadecimal printf format for uint_fast16_t

20.5.2.48 PRIXFAST16

```
#define PRIXFAST16 "X"
```

uppercase hexadecimal printf format for uint_fast16_t

20.5.2.49 PRIxFAST32

```
#define PRIxFAST32 "lx"
```

hexadecimal printf format for uint_fast32_t

20.5.2.50 PRIXFAST32

```
#define PRIXFAST32 "lX"
```

uppercase hexadecimal printf format for uint_fast32_t

20.5.2.51 PRIxFAST8

```
#define PRIxFAST8 "x"
```

hexadecimal printf format for uint_fast8_t

20.5.2.52 PRIXFAST8

```
#define PRIXFAST8 "X"
```

uppercase hexadecimal printf format for uint_fast8_t

20.5.2.53 PRIxLEAST16

```
#define PRIxLEAST16 "x"
```

hexadecimal printf format for uint_least16_t

20.5.2.54 PRIXLEAST16

```
#define PRIXLEAST16 "X"
```

uppercase hexadecimal printf format for uint_least16_t

20.5.2.55 PRIxLEAST32

```
#define PRIxLEAST32 "lx"
```

hexadecimal printf format for uint_least32_t

20.5.2.56 PRIXLEAST32

```
#define PRIXLEAST32 "lX"
```

uppercase hexadecimal printf format for uint_least32_t

20.5.2.57 PRIxLEAST8

```
#define PRIxLEAST8 "x"
```

hexadecimal printf format for uint_least8_t

20.5.2.58 PRIXLEAST8

```
#define PRIXLEAST8 "X"
```

uppercase hexadecimal printf format for uint_least8_t

20.5.2.59 PRIxPTR

```
#define PRIxPTR PRIx16
```

hexadecimal printf format for uintptr_t

20.5.2.60 PRIXPTR

```
#define PRIXPTR PRIX16
```

uppercase hexadecimal printf format for uintptr_t

20.5.2.61 SCNd16

```
#define SCNd16 "d"
```

decimal scanf format for int16_t

20.5.2.62 SCNd32

```
#define SCNd32 "ld"
```

decimal scanf format for int32_t

20.5.2.63 SCNd8

```
#define SCNd8 "hhd"
```

decimal scanf format for int8_t

20.5.2.64 SCNdFAST16

```
#define SCNdFAST16 "d"
```

decimal scanf format for int_fast16_t

20.5.2.65 SCNdFAST32

```
#define SCNdFAST32 "ld"
```

decimal scanf format for int_fast32_t

20.5.2.66 SCNdFAST8

```
#define SCNdFAST8 "hhd"
```

decimal scanf format for int_fast8_t

20.5.2.67 SCNdLEAST16

```
#define SCNdLEAST16 "d"
```

decimal scanf format for int_least16_t

20.5.2.68 SCNdLEAST32

```
#define SCNdLEAST32 "ld"
```

decimal scanf format for int_least32_t

20.5.2.69 SCNdLEAST8

```
#define SCNdLEAST8 "hhd"
```

decimal scanf format for int_least8_t

20.5.2.70 SCNdPTR

```
#define SCNdPTR SCNd16
```

decimal scanf format for intptr_t

20.5.2.71 SCNi16

```
#define SCNi16 "i"
```

generic-integer scanf format for int16_t

20.5.2.72 SCNi32

```
#define SCNi32 "li"
```

generic-integer scanf format for int32_t

20.5.2.73 SCNi8

```
#define SCNi8 "hhi"
```

generic-integer scanf format for int8_t

20.5.2.74 SCNiFAST16

```
#define SCNiFAST16 "i"
```

generic-integer scanf format for int_fast16_t

20.5.2.75 SCNiFAST32

```
#define SCNiFAST32 "li"
```

generic-integer scanf format for int_fast32_t

20.5.2.76 SCNiFAST8

```
#define SCNiFAST8 "hhi"
```

generic-integer scanf format for int_fast8_t

20.5.2.77 SCNiLEAST16

```
#define SCNiLEAST16 "i"
```

generic-integer scanf format for int_least16_t

20.5.2.78 SCNiLEAST32

```
#define SCNiLEAST32 "li"
```

generic-integer scanf format for int_least32_t

20.5.2.79 SCNiLEAST8

```
#define SCNiLEAST8 "hhi"
```

generic-integer scanf format for int_least8_t

20.5.2.80 SCNiPTR

```
#define SCNiPTR SCNi16
```

generic-integer scanf format for intptr_t

20.5.2.81 SCNo16

```
#define SCNo16 "o"
```

octal scanf format for uint16_t

20.5.2.82 SCNo32

```
#define SCNo32 "lo"
```

octal scanf format for uint32_t

20.5.2.83 SCNo8

```
#define SCNo8 "hho"
```

octal scanf format for uint8_t

20.5.2.84 SCNoFAST16

```
#define SCNoFAST16 "o"
```

octal scanf format for uint_fast16_t

20.5.2.85 SCNoFAST32

```
#define SCNoFAST32 "lo"
```

octal scanf format for uint_fast32_t

20.5.2.86 SCNoFAST8

```
#define SCNoFAST8 "hho"
```

octal scanf format for uint_fast8_t

20.5.2.87 SCNoLEAST16

```
#define SCNoLEAST16 "o"
```

octal scanf format for uint_least16_t

20.5.2.88 SCNoLEAST32

```
#define SCNoLEAST32 "lo"
```

octal scanf format for uint_least32_t

20.5.2.89 SCNoLEAST8

```
#define SCNoLEAST8 "hho"
```

octal scanf format for uint_least8_t

20.5.2.90 SCNoPTR

```
#define SCNoPTR SCNo16
```

octal scanf format for uintptr_t

20.5.2.91 SCNu16

```
#define SCNu16 "u"
```

decimal scanf format for uint16_t

20.5.2.92 SCNu32

```
#define SCNu32 "lu"
```

decimal scanf format for uint32_t

20.5.2.93 SCNu8

```
#define SCNu8 "hhu"
```

decimal scanf format for uint8_t

20.5.2.94 SCNuFAST16

```
#define SCNuFAST16 "u"
```

decimal scanf format for uint_fast16_t

20.5.2.95 SCNuFAST32

```
#define SCNuFAST32 "lu"
```

decimal scanf format for uint_fast32_t

20.5.2.96 SCNuFAST8

```
#define SCNuFAST8 "hhu"
```

decimal scanf format for uint_fast8_t

20.5.2.97 SCNuLEAST16

```
#define SCNuLEAST16 "u"
```

decimal scanf format for uint_least16_t

20.5.2.98 SCNuLEAST32

```
#define SCNuLEAST32 "lu"
```

decimal scanf format for uint_least32_t

20.5.2.99 SCNuLEAST8

```
#define SCNuLEAST8 "hhu"
```

decimal scanf format for uint_least8_t

20.5.2.100 SCNuPTR

```
#define SCNuPTR SCNu16
```

decimal scanf format for uintptr_t

20.5.2.101 SCNx16

```
#define SCNx16 "x"
```

hexadecimal scanf format for uint16_t

20.5.2.102 SCNx32

```
#define SCNx32 "lx"
```

hexadecimal scanf format for uint32_t

20.5.2.103 SCNx8

```
#define SCNx8 "hhx"
```

hexadecimal scanf format for uint8_t

20.5.2.104 SCNxFAST16

```
#define SCNxFAST16 "x"
```

hexadecimal scanf format for uint_fast16_t

20.5.2.105 SCNxFAST32

```
#define SCNxFAST32 "lx"
```

hexadecimal scanf format for uint_fast32_t

20.5.2.106 SCNxFAST8

```
#define SCNxFAST8 "hhx"
```

hexadecimal scanf format for uint_fast8_t

20.5.2.107 SCNxLEAST16

```
#define SCNxLEAST16 "x"
```

hexadecimal scanf format for uint_least16_t

20.5.2.108 SCNxLEAST32

```
#define SCNxLEAST32 "lx"
```

hexadecimal scanf format for uint_least32_t

20.5.2.109 SCNxLEAST8

```
#define SCNxLEAST8 "hhx"
```

hexadecimal scanf format for uint_least8_t

20.5.2.110 SCNxPTR

```
#define SCNxPTR SCNx16
```

hexadecimal scanf format for uintptr_t

20.5.3 Typedef Documentation

20.5.3.1 int_farptr_t

```
typedef int32_t int_farptr_t
```

signed integer type that can hold a pointer > 64 KiB

20.5.3.2 uint_farptr_t

```
typedef uint32_t uint_farptr_t
```

unsigned integer type that can hold a pointer > 64 KiB, see also pgm_get_far_address()

20.6 <math.h>: Mathematics

Macros

- #define M_E 2.7182818284590452354
- #define M_LOG2E 1.4426950408889634074
- #define M_LOG10E 0.43429448190325182765
- #define M_LN2 0.69314718055994530942
- #define M_LN10 2.30258509299404568402
- #define M_PI 3.14159265358979323846
- #define M_PI_2 1.57079632679489661923
- #define M_PI_4 0.78539816339744830962
- #define M_1_PI 0.31830988618379067154
- #define M_2_PI 0.63661977236758134308
- #define M_2_SQRTPI 1.12837916709551257390
- #define M_SQRT2 1.41421356237309504880
- #define M_SQRT1_2 0.70710678118654752440
- #define NAN __builtin_nan("")
- #define nanf(__tag) __builtin_nanf(__tag)
- #define nan(__tag) __builtin_nan(__tag)
- #define nanl(__tag) __builtin_nanl(__tag)
- #define INFINITY __builtin_inff()
- #define HUGE_VALF __builtin_huge_valf()
- #define HUGE_VAL __builtin_huge_val()
- #define HUGE_VALL __builtin_huge_vall()

Functions

- float cosf (float x)
- double cos (double x)
- long double cosl (long double x)
- float sinf (float x)
- double sin (double x)
- long double sinl (long double x)
- void sincosf (float x, float *psin, float *pcos)
- void sincos (double x, double *psin, double *pcos)
- void sincosl (long double x, long double *psin, long double *pcos)
- float tanf (float x)
- double tan (double x)
- long double tanl (long double x)
- float fabsf (float __x)
- double fabs (double __x)
- long double fabsl (long double __x)
- float fmodf (float x, float y)
- double fmod (double x, double y)
- long double fmodl (long double x, long double y)
- float modff (float x, float *iptr)
- double modf (double x, double *iptr)
- long double modfl (long double x, long double *iptr)
- float sqrtf (float x)
- double sqrt (double x)
- long double sqrtl (long double x)
- float cbrtf (float x)
- double cbrt (double x)
- long double cbrtl (long double x)
- float hypotf (float x, float y)
- double hypot (double x, double y)
- long double hypotl (long double x, long double y)
- float floorf (float x)
- double floor (double x)
- long double floorl (long double x)
- float ceilf (float x)
- double ceil (double x)
- long double ceill (long double x)
- float frexpf (float x, int *pexp)
- double frexp (double x, int *pexp)
- long double frexpl (long double x, int *pexp)
- float ldexpf (float x, int iexp)
- double ldexp (double x, int iexp)
- long double ldexpl (long double x, int iexp)
- float expf (float x)
- double exp (double x)
- long double expl (long double x)
- float coshf (float x)
- double cosh (double x)
- long double coshl (long double x)
- float sinhf (float x)
- double sinh (double x)
- long double sinhl (long double x)
- float tanhf (float x)
- double tanh (double x)

- long double tanhl (long double x)
- float acosf (float x)
- double acos (double x)
- long double acosl (long double x)
- float asinf (float x)
- double asin (double x)
- long double asinl (long double x)
- float atanf (float x)
- double atan (double x)
- long double atanl (long double x)
- float atan2f (float y, float x)
- double atan2 (double y, double x)
- long double atan2l (long double y, long double x)
- float logf (float x)
- double log (double x)
- long double logl (long double x)
- float log10f (float x)
- double log10 (double x)
- long double log10l (long double x)
- float log2f (float x)
- double log2 (double x)
- long double log2l (long double x)
- float powf (float x, float y)
- double pow (double x, double y)
- long double powl (long double x, long double y)
- int isnanf (float x)
- int isnan (double x)
- int isnanl (long double x)
- int isinff (float x)
- int isinf (double x)
- int isinfl (long double x)
- static int isfinitef (float __x)
- static int isfinite (double __x)
- static int isfinitel (long double __x)
- static float copysignf (float __x, float __y)
- static double copysign (double __x, double __y)
- static long double copysignl (long double __x, long double __y)
- int signbitf (float x)
- int signbit (double x)
- int signbitl (long double x)
- float fdimf (float x, float y)
- double fdim (double x, double y)
- long double fdiml (long double x, long double y)
- float fmaf (float x, float y, float z)
- double fma (double x, double y, double z)
- long double fmal (long double x, long double y, long double z)
- float fmaxf (float x, float y)
- double fmax (double x, double y)
- long double fmaxl (long double x, long double y)
- float fminf (float x, float y)
- double fmin (double x, double y)
- long double fminl (long double x, long double y)
- float truncf (float x)
- double trunc (double x)
- long double truncl (long double x)

- float roundf (float x)
- double round (double x)
- long double roundl (long double x)
- long lroundf (float x)
- long lround (double x)
- long lroundl (long double x)
- long lrintf (float x)
- long lrint (double x)
- long lrintl (long double x)

Non-Standard Math Functions

- float squaref (float x)
- double square (double x)
- long double squarel (long double x)

20.6.1 Detailed Description

```
#include <math.h>
```

This header file declares basic mathematics constants and functions.

Notes:

- Math functions do not raise exceptions and do not change the `errno` variable. Therefore the majority of them are declared with `const` attribute, for better optimization by GCC.
- 64-bit floating-point arithmetic is only available in `GCC v10` and up. The size of the `double` and `long double` type can be selected at compile-time with options like `-mdouble=64` and `-mlong-double=32`. Whether such options are available, and their default values, depend on how the compiler has been configured.
- The implementation of 64-bit floating-point arithmetic has some shortcomings and limitations, see the `avr-gcc Wiki` for details.

See also some benchmarks for IEEE single and IEEE double.

20.6.2 Macro Definition Documentation

20.6.2.1 HUGE_VAL

```
#define HUGE_VAL __builtin_huge_val()
```

`double` infinity constant.

20.6.2.2 HUGE_VALF

```
#define HUGE_VALF __builtin_huge_valf()
```

`float` infinity constant.

20.6.2.3 HUGE_VALL

```
#define HUGE_VALL __builtin_huge_vall()
```

`long double` infinity constant.

20.6.2.4 INFINITY

```
#define INFINITY __builtin_inff()
```

`float` infinity constant.

20.6.2.5 M_1_PI

```
#define M_1_PI 0.31830988618379067154
```

The constant $1/\pi$.

20.6.2.6 M_2_PI

```
#define M_2_PI 0.63661977236758134308
```

The constant $2/\pi$.

20.6.2.7 M_2_SQRTPI

```
#define M_2_SQRTPI 1.12837916709551257390
```

The constant 2/sqrt(π).

20.6.2.8 M_E

```
#define M_E 2.7182818284590452354
```

The constant Euler's number *e*.

20.6.2.9 M_LN10

```
#define M_LN10 2.30258509299404568402
```

The constant natural logarithm of 10.

20.6.2.10 M_LN2

```
#define M_LN2 0.69314718055994530942
```

The constant natural logarithm of 2.

20.6.2.11 M_LOG10E

```
#define M_LOG10E 0.43429448190325182765
```

The constant logarithm of Euler's number *e* to base 10.

20.6.2.12 M_LOG2E

```
#define M_LOG2E 1.4426950408889634074
```

The constant logarithm of Euler's number *e* to base 2.

20.6.2.13 M_PI

```
#define M_PI 3.14159265358979323846
```

The constant π.

20.6.2.14 M_PI_2

```
#define M_PI_2 1.57079632679489661923
```

The constant $\pi/2$.

20.6.2.15 M_PI_4

```
#define M_PI_4 0.78539816339744830962
```

The constant $\pi/4$.

20.6.2.16 M_SQRT1_2

```
#define M_SQRT1_2 0.70710678118654752440
```

The constant *1/sqrt*(2).

20.6.2.17 M_SQRT2

```
#define M_SQRT2 1.41421356237309504880
```

The square root of 2.

20.6.2.18 NAN

```
#define NAN __builtin_nan("")
```

The `double` representation of a constant quiet NaN.

20.6.2.19 nan

```
#define nan(
             __tag ) __builtin_nan(__tag)
```

The `double` representation of a constant quiet NaN. `__tag` is a string constant like `""` or `"123"`.

20.6.2.20 nanf

```
#define nanf(
             __tag ) __builtin_nanf(__tag)
```

The `float` representation of a constant quiet NaN. `__tag` is a string constant like `""` or `"123"`.

20.6.2.21 nanl

```
#define nanl(
             __tag ) __builtin_nanl(__tag)
```

The `long double` representation of a constant quiet NaN. `__tag` is a string constant like `""` or `"123"`.

20.6.3 Function Documentation

20.6.3.1 acos()

```
double acos (
             double x )
```

The acos() function computes the principal value of the arc cosine of *x*. The returned value is in the range [0, π] radians or NaN.

20.6.3.2 acosf()

```
float acosf (
             float x )
```

The acosf() function computes the principal value of the arc cosine of *x*. The returned value is in the range [0, π] radians. A domain error occurs for arguments not in the range [-1, +1]. The relative error is bounded by $1.9 \cdot 10^{-7}$.

20.6.3.3 acosl()

```
long double acosl (
             long double x )
```

The acosl() function computes the principal value of the arc cosine of *x*. The returned value is in the range [0, π] radians or NaN.

Since

AVR-LibC v2.2

20.6.3.4 asin()

```
double asin (
             double x )
```

The asin() function computes the principal value of the arc sine of *x*. The returned value is in the range [-π/2, π/2] radians or NaN.

20.6.3.5 asinf()

```
float asinf (
             float x )
```

The asinf() function computes the principal value of the arc sine of *x*. The returned value is in the range [-π/2, π/2] radians. A domain error occurs for arguments not in the range [-1, +1]. The relative error is bounded by $3.3 \cdot 10^{-7}$.

20.6.3.6 asinl()

```
long double asinl (
             long double x )
```

The asinl() function computes the principal value of the arc sine of *x*. The returned value is in the range [-π/2, π/2] radians or NaN.

Since

AVR-LibC v2.2

20.6.3.7 atan()

```
double atan (
             double x )
```

The atan() function computes the principal value of the arc tangent of *x*. The returned value is in the range [-π/2, π/2] radians.

20.6.3.8 atan2()

```
double atan2 (
             double y,
             double x )
```

The atan2() function computes the principal value of the arc tangent of *y* / *x*, using the signs of both arguments to determine the quadrant of the return value. The returned value is in the range [-π, +π] radians.

20.6.3.9 atan2f()

```
float atan2f (
             float y,
             float x )
```

The atan2f() function computes the principal value of the arc tangent of *y* / *x*, using the signs of both arguments to determine the quadrant of the return value. The returned value is in the range [-π, +π] radians.

20.6.3.10 atan2l()

```
long double atan2l (
            long double y,
            long double x )
```

The atan2l() function computes the principal value of the arc tangent of *y* / *x*, using the signs of both arguments to determine the quadrant of the return value. The returned value is in the range [-π, +π] radians.

Since

AVR-LibC v2.2

20.6.3.11 atanf()

```
float atanf (
            float x )
```

The atanf() function computes the principal value of the arc tangent of *x*. The returned value is in the range [-π/2, π/2] radians. The relative error is bounded by $1.3 \cdot 10^{-7}$.

20.6.3.12 atanl()

```
long double atanl (
            long double x )
```

The atanl() function computes the principal value of the arc tangent of *x*. The returned value is in the range [-π/2, π/2] radians.

Since

AVR-LibC v2.2

20.6.3.13 cbrt()

```
double cbrt (
            double x )
```

The cbrt() function returns the cube root of *x*.

20.6.3.14 cbrtf()

```
float cbrtf (
            float x )
```

The cbrtf() function returns the cube root of *x*.

20.6.3.15 cbrtl()

```
long double cbrtl (
            long double x )
```

The cbrtl() function returns the cube root of *x*.

Since

AVR-LibC v2.2

20.6.3.16 ceil()

```
double ceil (
            double x )
```

The ceil() function returns the smallest integral value greater than or equal to *x*, expressed as a floating-point number.

20.6.3.17 ceilf()

```
float ceilf (
            float x )
```

The ceilf() function returns the smallest integral value greater than or equal to *x*, expressed as a floating-point number.

20.6.3.18 ceill()

```
long double ceill (
            long double x )
```

The ceill() function returns the smallest integral value greater than or equal to *x*, expressed as a floating-point number.

Since

AVR-LibC v2.2

20.6.3.19 copysign()

```
static double copysign (
            double __x,
            double __y )  [inline], [static]
```

The copysign() function returns *__x* but with the sign of *__y*. They work even if *__x* or *__y* are NaN or zero.

20.6.3.20 copysignf()

```
static float copysignf (
            float __x,
            float __y )  [inline], [static]
```

The copysignf() function returns __x but with the sign of __y. They work even if __x or __y are NaN or zero.

20.6.3.21 copysignl()

```
static long double copysignl (
            long double __x,
            long double __y )  [inline], [static]
```

The copysignl() function returns __x but with the sign of __y. They work even if __x or __y are NaN or zero.

Since

AVR-LibC v2.2

20.6.3.22 cos()

```
double cos (
            double x )
```

The cos() function returns the cosine of *x*, measured in radians.

20.6.3.23 cosf()

```
float cosf (
            float x )
```

The cosf() function returns the cosine of *x*, measured in radians.

20.6.3.24 cosh()

```
double cosh (
            double x )
```

The cosh() function returns the hyperbolic cosine of *x*.

20.6.3.25 coshf()

```
float coshf (
            float x )
```

The coshf() function returns the hyperbolic cosine of *x*.

20.6.3.26 coshl()

```
long double coshl (
            long double x )
```

The coshl() function returns the hyperbolic cosine of *x*.

Since

AVR-LibC v2.2

20.6.3.27 cosl()

```
long double cosl (
            long double x )
```

The cosl() function returns the cosine of *x*, measured in radians.

Since

AVR-LibC v2.2

20.6.3.28 exp()

```
double exp (
            double x )
```

The exp() function returns the exponential value of *x*.

20.6.3.29 expf()

```
float expf (
            float x )
```

The expf() function returns the exponential value of *x*.

20.6.3.30 expl()

```
long double expl (
            long double x )
```

The expl() function returns the exponential value of *x*.

Since

AVR-LibC v2.2

20.6.3.31 fabs()

```
double fabs (
            double __x )  [inline]
```

The fabs() function computes the absolute value of a floating-point number *x*.

20.6.3.32 fabsf()

```
float fabsf (
            float __x )  [inline]
```

The fabsf() function computes the absolute value of a floating-point number *x*.

20.6.3.33 fabsl()

```
long double fabsl (
            long double __x )  [inline]
```

The fabsl() function computes the absolute value of a floating-point number *x*.

Since

AVR-LibC v2.2

20.6.3.34 fdim()

```
double fdim (
            double x,
            double y )
```

The fdim() function returns *max(x - y, 0)*. If *x* or *y* or both are NaN, NaN is returned.

20.6.3.35 fdimf()

```
float fdimf (
            float x,
            float y )
```

The fdimf() function returns *max(x - y, 0)*. If *x* or *y* or both are NaN, NaN is returned.

20.6.3.36 fdiml()

```
long double fdiml (
            long double x,
            long double y )
```

The fdiml() function returns *max(x - y, 0)*. If *x* or *y* or both are NaN, NaN is returned.

Since

AVR-LibC v2.2, GCC v15.2

20.6.3.37 floor()

```
double floor (
            double x )
```

The floor() function returns the largest integral value less than or equal to *x*, expressed as a floating-point number.

20.6.3.38 floorf()

```
float floorf (
            float x )
```

The floorf() function returns the largest integral value less than or equal to *x*, expressed as a floating-point number.

20.6.3.39 floorl()

```
long double floorl (
            long double x )
```

The floorl() function returns the largest integral value less than or equal to *x*, expressed as a floating-point number.

Since

AVR-LibC v2.2

20.6.3.40 fma()

```
double fma (
            double x,
            double y,
            double z )
```

The fma() function performs floating-point multiply-add. This is the operation $(x * y) + z$, but the intermediate result is not rounded to the destination type. This can sometimes improve the precision of a calculation.

20.6.3.41 fmaf()

```
float fmaf (
            float x,
            float y,
            float z )
```

The fmaf() function performs floating-point multiply-add. This is the operation $(x * y) + z$, but the intermediate result is not rounded to the destination type. This can sometimes improve the precision of a calculation.

20.6.3.42 fmal()

```
long double fmal (
             long double x,
             long double y,
             long double z )
```

The fmal() function performs floating-point multiply-add. This is the operation *(x ∗ y) + z*, but the intermediate result is not rounded to the destination type. This can sometimes improve the precision of a calculation.

Since

AVR-LibC v2.2

20.6.3.43 fmax()

```
double fmax (
             double x,
             double y )
```

The fmax() function returns the greater of the two values *x* and *y*. If an argument is NaN, the other argument is returned. If both arguments are NaN, NaN is returned.

20.6.3.44 fmaxf()

```
float fmaxf (
             float x,
             float y )
```

The fmaxf() function returns the greater of the two values *x* and *y*. If an argument is NaN, the other argument is returned. If both arguments are NaN, NaN is returned.

20.6.3.45 fmaxl()

```
long double fmaxl (
             long double x,
             long double y )
```

The fmaxl() function returns the greater of the two values *x* and *y*. If an argument is NaN, the other argument is returned. If both arguments are NaN, NaN is returned.

Since

AVR-LibC v2.2

20.6.3.46 fmin()

```
double fmin (
            double x,
            double y )
```

The fmin() function returns the lesser of the two values *x* and *y*. If an argument is NaN, the other argument is returned. If both arguments are NaN, NaN is returned.

20.6.3.47 fminf()

```
float fminf (
            float x,
            float y )
```

The fminf() function returns the lesser of the two values *x* and *y*. If an argument is NaN, the other argument is returned. If both arguments are NaN, NaN is returned.

20.6.3.48 fminl()

```
long double fminl (
            long double x,
            long double y )
```

The fminl() function returns the lesser of the two values *x* and *y*. If an argument is NaN, the other argument is returned. If both arguments are NaN, NaN is returned.

Since

AVR-LibC v2.2

20.6.3.49 fmod()

```
double fmod (
            double x,
            double y )
```

The function fmod() returns the floating-point remainder of *x / y*.

20.6.3.50 fmodf()

```
float fmodf (
            float x,
            float y )
```

The function fmodf() returns the floating-point remainder of *x / y*.

20.6.3.51 fmodl()

```
long double fmodl (
             long double x,
             long double y )
```

The function fmodl() returns the floating-point remainder of *x* / *y*.

Since

AVR-LibC v2.2

20.6.3.52 frexp()

```
double frexp (
             double x,
             int * pexp )
```

The frexp() function breaks a floating-point number into a normalized fraction and an integral power of 2. It stores the integer in the `int` object pointed to by *pexp*.

If *x* is a normal float point number, the frexp() function returns the value `v`, such that `v` has a magnitude in the interval [1/2, 1) or zero, and *x* equals `v` times 2 raised to the power *pexp*. If *x* is zero, both parts of the result are zero. If *x* is not a finite number, the frexp() returns *x* as is and stores 0 by *pexp*.

20.6.3.53 frexpf()

```
float frexpf (
             float x,
             int * pexp )
```

The frexpf() function breaks a floating-point number into a normalized fraction and an integral power of 2. It stores the integer in the `int` object pointed to by *pexp*.

If *x* is a normal float point number, the frexpf() function returns the value `v`, such that `v` has a magnitude in the interval [1/2, 1) or zero, and *x* equals `v` times 2 raised to the power *pexp*. If *x* is zero, both parts of the result are zero. If *x* is not a finite number, the frexpf() returns *x* as is and stores 0 by *pexp*.

Note

This implementation permits a zero pointer as a directive to skip a storing the exponent.

20.6.3.54 frexpl()

```
long double frexpl (
            long double x,
            int * pexp )
```

The frexpl() function breaks a floating-point number into a normalized fraction and an integral power of 2. It stores the integer in the `int` object pointed to by *pexp*.

If *x* is a normal float point number, the frexpl() function returns the value `v`, such that `v` has a magnitude in the interval [1/2, 1) or zero, and *x* equals `v` times 2 raised to the power *pexp*. If *x* is zero, both parts of the result are zero. If *x* is not a finite number, the frexpl() returns *x* as is and stores 0 by *pexp*.

Since

AVR-LibC v2.2

20.6.3.55 hypot()

```
double hypot (
            double x,
            double y )
```

The hypot() function returns *sqrt(x∗x + y∗y)*. This is the length of the hypotenuse of a right triangle with sides of length *x* and *y*, or the distance of the point (*x*, *y*) from the origin. Using this function instead of the direct formula is wise, since the error is much smaller. No underflow with small *x* and *y*. No overflow if result is in range.

20.6.3.56 hypotf()

```
float hypotf (
            float x,
            float y )
```

The hypotf() function returns *sqrtf(x∗x + y∗y)*. This is the length of the hypotenuse of a right triangle with sides of length *x* and *y*, or the distance of the point (*x*, *y*) from the origin. Using this function instead of the direct formula is wise, since the error is much smaller. No underflow with small *x* and *y*. No overflow if result is in range.

20.6.3.57 hypotl()

```
long double hypotl (
            long double x,
            long double y )
```

The hypotl() function returns *sqrtl(x∗x + y∗y)*. This is the length of the hypotenuse of a right triangle with sides of length *x* and *y*, or the distance of the point (*x*, *y*) from the origin. Using this function instead of the direct formula is wise, since the error is much smaller. No underflow with small *x* and *y*. No overflow if result is in range.

Since

AVR-LibC v2.2

20.6.3.58 isfinite()

```
static int isfinite (
            double __x )  [inline], [static]
```

The isfinite() function returns a nonzero value if __x is finite: not plus or minus infinity, and not NaN.

20.6.3.59 isfinitef()

```
static int isfinitef (
            float __x )  [inline], [static]
```

The isfinitef() function returns a nonzero value if __x is finite: not plus or minus infinity, and not NaN.

20.6.3.60 isfinitel()

```
static int isfinitel (
            long double __x )  [inline], [static]
```

The isfinitel() function returns a nonzero value if __x is finite: not plus or minus infinity, and not NaN.

Since

AVR-LibC v2.2

20.6.3.61 isinf()

```
int isinf (
            double x )
```

The function isinf() returns 1 if the argument *x* is positive infinity, -1 if *x* is negative infinity, and 0 otherwise.

20.6.3.62 isinff()

```
int isinff (
            float x )
```

The function isinff() returns 1 if the argument *x* is positive infinity, -1 if *x* is negative infinity, and 0 otherwise.

20.6.3.63 isinfl()

```
int isinfl (
            long double x )
```

The function isinfl() returns 1 if the argument *x* is positive infinity, -1 if *x* is negative infinity, and 0 otherwise.

Since

AVR-LibC v2.2

20.6.3.64 isnan()

```
int isnan (
            double x )
```

The function isnan() returns 1 if the argument *x* represents a "not-a-number" (NaN) object, otherwise 0.

20.6.3.65 isnanf()

```
int isnanf (
            float x )
```

The function isnanf() returns 1 if the argument *x* represents a "not-a-number" (NaN) object, otherwise 0.

20.6.3.66 isnanl()

```
int isnanl (
            long double x )
```

The function isnanl() returns 1 if the argument *x* represents a "not-a-number" (NaN) object, otherwise 0.

Since

AVR-LibC v2.2

20.6.3.67 ldexp()

```
double ldexp (
            double x,
            int iexp )
```

The ldexp() function multiplies a floating-point number by an integral power of 2. It returns the value of *x* times 2 raised to the power *iexp*.

20.6.3.68 ldexpf()

```
float ldexpf (
            float x,
            int iexp )
```

The ldexpf() function multiplies a floating-point number by an integral power of 2. It returns the value of *x* times 2 raised to the power *iexp*.

20.6.3.69 ldexpl()

```
long double ldexpl (
            long double x,
            int iexp )
```

The ldexpl() function multiplies a floating-point number by an integral power of 2. It returns the value of *x* times 2 raised to the power *iexp*.

Since

AVR-LibC v2.2

20.6.3.70 log()

```
double log (
            double x )
```

The log() function returns the natural logarithm of argument *x*.

20.6.3.71 log10()

```
double log10 (
            double x )
```

The log10() function returns the logarithm of argument *x* to base 10.

20.6.3.72 log10f()

```
float log10f (
            float x )
```

The log10f() function returns the logarithm of argument *x* to base 10. The relative error is bounded by $2.8 \cdot 10^{-7}$.

20.6.3.73 log10l()

```
long double log10l (
            long double x )
```

The log10l() function returns the logarithm of argument *x* to base 10.

Since

AVR-LibC v2.2

20.6.3.74 log2()

```
double log2 (
            double x )
```

The log2() function returns the logarithm of argument *x* to base 2.

Since

AVR-LibC v2.3

20.6.3.75 log2f()

```
float log2f (
            float x )
```

The log2f() function returns the logarithm of argument *x* to base 2. The relative error is bounded by $1.8 \cdot 10^{-7}$.

Since

AVR-LibC v2.3

20.6.3.76 log2l()

```
long double log2l (
            long double x )
```

The log2l() function returns the logarithm of argument *x* to base 2.

Since

AVR-LibC v2.3

20.6.3.77 logf()

```
float logf (
            float x )
```

The logf() function returns the natural logarithm of argument *x*. The relative error is bounded by $2.3 \cdot 10^{-7}$.

20.6.3.78 logl()

```
long double logl (
            long double x )
```

The logl() function returns the natural logarithm of argument *x*.

Since

AVR-LibC v2.2

20.6.3.79 lrint()

```
long lrint (
            double x )
```

The lrint() function rounds *x* to the nearest integer, rounding the halfway cases to the even integer direction. (That is both 1.5 and 2.5 values are rounded to 2). This function is similar to rint() function, but it differs in type of return value and in that an overflow is possible.

Returns

The rounded long integer value. If *x* is not a finite number or an overflow was, this realization returns the LONG_MIN value (0x80000000).

20.6.3.80 lrintf()

```
long lrintf (
            float x )
```

The lrintf() function rounds *x* to the nearest integer, rounding the halfway cases to the even integer direction. (That is both 1.5 and 2.5 values are rounded to 2). This function is similar to rintf() function, but it differs in type of return value and in that an overflow is possible.

Returns

The rounded long integer value. If *x* is not a finite number or an overflow was, this realization returns the LONG_MIN value (0x80000000).

20.6.3.81 lrintl()

```
long lrintl (
            long double x )
```

The lrintl() function rounds *x* to the nearest integer, rounding the halfway cases to the even integer direction. (That is both 1.5 and 2.5 values are rounded to 2). This function is similar to rintl() function, but it differs in type of return value and in that an overflow is possible.

Returns

The rounded long integer value. If *x* is not a finite number or an overflow was, this realization returns the LONG_MIN value (0x80000000).

Since

AVR-LibC v2.2

20.6.3.82 lround()

```
long lround (
            double x )
```

The lround() function rounds *x* to the nearest integer, but rounds halfway cases away from zero (instead of to the nearest even integer). This function is similar to round() function, but it differs in type of return value and in that an overflow is possible.

Returns

The rounded long integer value. If *x* is not a finite number or an overflow was, this realization returns the LONG_MIN value (0x80000000).

20.6.3.83 lroundf()

```
long lroundf (
            float x )
```

The lroundf() function rounds *x* to the nearest integer, but rounds halfway cases away from zero (instead of to the nearest even integer). This function is similar to round() function, but it differs in type of return value and in that an overflow is possible.

Returns

The rounded long integer value. If *x* is not a finite number or an overflow was, this realization returns the LONG_MIN value (0x80000000).

20.6.3.84 lroundl()

```
long lroundl (
            long double x )
```

The lroundl() function rounds *x* to the nearest integer, but rounds halfway cases away from zero (instead of to the nearest even integer). This function is similar to round() function, but it differs in type of return value and in that an overflow is possible.

Returns

The rounded long integer value. If *x* is not a finite number or an overflow was, this realization returns the LONG_MIN value (0x80000000).

Since

AVR-LibC v2.2

20.6.3.85 modf()

```
double modf (
             double x,
             double * iptr )
```

The modf() function breaks the argument *x* into integral and fractional parts, each of which has the same sign as the argument. It stores the integral part as a `double` in the object pointed to by *iptr*.

The modf() function returns the signed fractional part of *x*.

20.6.3.86 modff()

```
float modff (
             float x,
             float * iptr )
```

The modff() function breaks the argument *x* into integral and fractional parts, each of which has the same sign as the argument. It stores the integral part as a `float` in the object pointed to by *iptr*.

The modff() function returns the signed fractional part of *x*.

Note

This implementation skips writing by zero pointer. However, the GCC 4.3 can replace this function with inline code that does not permit to use NULL address for the avoiding of storing.

20.6.3.87 modfl()

```
long double modfl (
             long double x,
             long double * iptr )
```

The modfl() function breaks the argument *x* into integral and fractional parts, each of which has the same sign as the argument. It stores the integral part as a `long double` in the object pointed to by *iptr*.

The modf() function returns the signed fractional part of *x*.

Since

AVR-LibC v2.2

20.6.3.88 pow()

```
double pow (
             double x,
             double y )
```

The function pow() returns the value of *x* to the exponent *y*.
Notice that for integer exponents, there is the more efficient `double __builtin_powi(double x, int y)`.

20.6.3.89 powf()

```
float powf (
            float x,
            float y )
```

The function powf() returns the value of *x* to the exponent *y*.
Notice that for integer exponents, there is the more efficient `float __builtin_powif(float x, int y)`.

20.6.3.90 powl()

```
long double powl (
            long double x,
            long double y )
```

The function powl() returns the value of *x* to the exponent *y*.
Notice that for integer exponents, there is the more efficient `long double __builtin_powil(long double x, int y)`

Since

AVR-LibC v2.2

20.6.3.91 round()

```
double round (
            double x )
```

The round() function rounds *x* to the nearest integer, but rounds halfway cases away from zero (instead of to the nearest even integer). Overflow is impossible.

Returns

The rounded value. If *x* is an integral or infinite, *x* itself is returned. If *x* is `NaN`, then `NaN` is returned.

20.6.3.92 roundf()

```
float roundf (
            float x )
```

The roundf() function rounds *x* to the nearest integer, but rounds halfway cases away from zero (instead of to the nearest even integer). Overflow is impossible.

Returns

The rounded value. If *x* is an integral or infinite, *x* itself is returned. If *x* is `NaN`, then `NaN` is returned.

20.6.3.93 roundl()

```
long double roundl (
            long double x )
```

The roundl() function rounds *x* to the nearest integer, but rounds halfway cases away from zero (instead of to the nearest even integer). Overflow is impossible.

Returns

The rounded value. If *x* is an integral or infinite, *x* itself is returned. If *x* is NaN, then NaN is returned.

Since

AVR-LibC v2.2

20.6.3.94 signbit()

```
int signbit (
            double x )
```

The signbit() function returns a nonzero value if the value of *x* has its sign bit set. This is not the same as `*x* < 0.0', because IEEE 754 floating point allows zero to be signed. The comparison '-0.0 < 0.0' is false, but `signbit (-0.0)' will return a nonzero value.

20.6.3.95 signbitf()

```
int signbitf (
            float x )
```

The signbitf() function returns a nonzero value if the value of *x* has its sign bit set. This is not the same as `*x* < 0.0', because IEEE 754 floating point allows zero to be signed. The comparison '-0.0 < 0.0' is false, but `signbit (-0.0)' will return a nonzero value.

20.6.3.96 signbitl()

```
int signbitl (
            long double x )
```

The signbitl() function returns a nonzero value if the value of *x* has its sign bit set. This is not the same as `*x* < 0.0', because IEEE 754 floating point allows zero to be signed. The comparison '-0.0 < 0.0' is false, but `signbit (-0.0)' will return a nonzero value.

Since

AVR-LibC v2.2

20.6.3.97 sin()

```
double sin (
            double x )
```

The sin() function returns the sine of *x*, measured in radians.

20.6.3.98 sincos()

```
void sincos (
            double x,
            double * psin,
            double * pcos )
```

The sincos() function returns the sine of *x* in *psin, and the cosine of *x* in *pcos. The angle *x* is measured in radians. As an example, the performance gain of the IEEE double version of sincos() compared to a sin() plus a cos() call is around 5000 cycles for *x* = 2.0.

Since

GCC v15.3 (IEEE double), AVR-LibC v2.3

20.6.3.99 sincosf()

```
void sincosf (
            float x,
            float * psin,
            float * pcos )
```

The sincosf() function returns the sine of *x* in *psin, and the cosine of *x* in *pcos. The angle *x* is measured in radians. A sincosf() call is a bit faster than calling sinf() and cosf() individually, but it consumes a bit more program memory.

Since

AVR-LibC v2.3

20.6.3.100 sincosl()

```
void sincosl (
            long double x,
            long double * psin,
            long double * pcos )
```

The sincosl() function returns the sine of *x* in *psin, and the cosine of *x* in *pcos. The angle *x* is measured in radians. As an example, the performance gain of the IEEE double version of sincosl() compared to a sinl() plus a cosl() call is around 5000 cycles for *x* = 2.0.

Since

GCC v15.3 (IEEE double), AVR-LibC v2.3

20.6.3.101 sinf()

```
float sinf (
            float x )
```

The sinf() function returns the sine of *x*, measured in radians.

20.6.3.102 sinh()

```
double sinh (
             double x )
```

The sinh() function returns the hyperbolic sine of *x*.

20.6.3.103 sinhf()

```
float sinhf (
             float x )
```

The sinhf() function returns the hyperbolic sine of *x*.

20.6.3.104 sinhl()

```
long double sinhl (
             long double x )
```

The sinhl() function returns the hyperbolic sine of *x*.

20.6.3.105 sinl()

```
long double sinl (
             long double x )
```

The sinl() function returns the sine of *x*, measured in radians.

Since

AVR-LibC v2.2

20.6.3.106 sqrt()

```
double sqrt (
             double x )
```

The sqrt() function returns the non-negative square root of *x*.

20.6.3.107 sqrtf()

```
float sqrtf (
             float x )
```

The sqrtf() function returns the non-negative square root of *x*.

20.6.3.108 sqrtl()

```
long double sqrtl (
            long double x )
```

The sqrtl() function returns the non-negative square root of *x*.

Since

AVR-LibC v2.2

20.6.3.109 square()

```
double square (
            double x )
```

The function square() returns $x * x$.

Note

This function does not belong to the C standard definition.

20.6.3.110 squaref()

```
float squaref (
            float x )
```

The function squaref() returns $x * x$.

Note

This function does not belong to the C standard definition.

20.6.3.111 squarel()

```
long double squarel (
            long double x )
```

The function squarel() returns $x * x$.

Note

This function does not belong to the C standard definition.

Since

AVR-LibC v2.2

20.6.3.112 tan()

```
double tan (
            double x )
```

The tan() function returns the tangent of *x*, measured in radians.

20.6.3.113 tanf()

```
float tanf (
            float x )
```

The tanf() function returns the tangent of *x*, measured in radians.

20.6.3.114 tanh()

```
double tanh (
            double x )
```

The tanh() function returns the hyperbolic tangent of *x*.

20.6.3.115 tanhf()

```
float tanhf (
            float x )
```

The tanhf() function returns the hyperbolic tangent of *x*.

20.6.3.116 tanhl()

```
long double tanhl (
            long double x )
```

The tanhl() function returns the hyperbolic tangent of *x*.

Since

AVR-LibC v2.2

20.6.3.117 tanl()

```
long double tanl (
            long double x )
```

The tanl() function returns the tangent of *x*, measured in radians.

Since

AVR-LibC v2.2

20.6.3.118 trunc()

```
double trunc (
             double x )
```

The trunc() function rounds *x* to the nearest integer not larger in absolute value.

20.6.3.119 truncf()

```
float truncf (
             float x )
```

The truncf() function rounds *x* to the nearest integer not larger in absolute value.

20.6.3.120 truncl()

```
long double truncl (
             long double x )
```

The truncl() function rounds *x* to the nearest integer not larger in absolute value.

Since

AVR-LibC v2.2

20.7 <setjmp.h>: Non-local goto

Functions

- int setjmp (jmp_buf __jmpb)
- void longjmp (jmp_buf __jmpb, int __ret)

20.7.1 Detailed Description

While the C language has the dreaded `goto` statement, it can only be used to jump to a label in the same (local) function. In order to jump directly to another (non-local) function, the C library provides the setjmp and longjmp functions. setjmp and longjmp are useful for dealing with errors and interrupts encountered in a low-level subroutine of a program.

Note

setjmp and longjmp make programs hard to understand and maintain. If possible, an alternative should be used.

longjmp can destroy changes made to global register variables (see How to permanently bind a variable to a register?).

For a very detailed discussion of setjmp/longjmp, see Chapter 7 of *Advanced Programming in the UNIX Environment*, by W. Richard Stevens.

Example:

```
#include <setjmp.h>

jmp_buf env;

int main (void)
{
    if (setjmp (env))
    {
        // Handle error ...
    }

    while (1)
    {
       // Main processing loop which calls foo() somewhere ...
    }
}

void foo (void)
{
    // blah, blah, blah ...

    if (err)
    {
        longjmp (env, 1);
    }
}
```

20.7.2 Function Documentation

20.7.2.1 longjmp()

```
void longjmp (
             jmp_buf __jmpb,
             int __ret )
```

Non-local jump to a saved stack context.

```
#include <setjmp.h>
```

longjmp() restores the environment saved by the last call of setjmp() with the corresponding *__jmpb* argument. After longjmp() is completed, program execution continues as if the corresponding call of setjmp() had just returned the value *__ret*.

Note

longjmp() cannot cause 0 to be returned. If longjmp() is invoked with a second argument of 0, 1 will be returned instead.

Parameters

__jmpb	Information saved by a previous call to setjmp().
__ret	Value to return to the caller of setjmp().

This function never returns.

20.7.2.2 setjmp()

```
int setjmp (
              jmp_buf __jmpb )
```

Save stack context for non-local goto.

```
#include <setjmp.h>
```

setjmp() saves the stack context/environment in *__jmpb* for later use by longjmp(). The stack context will be invalidated if the function which called setjmp() returns.

Parameters

__jmpb	Variable of type `jmp_buf` which holds the stack information such that the environment can be restored.

Returns

Returns 0 if returning directly, and non-zero when returning from longjmp() using the saved context.

20.8 <stdint.h>: Standard Integer Types

Exact-width integer types

Integer types having exactly the specified width.

- typedef signed char int8_t
- typedef unsigned char uint8_t
- typedef signed int int16_t
- typedef unsigned int uint16_t
- typedef __int24 int24_t
- typedef __uint24 uint24_t
- typedef signed long int int32_t
- typedef unsigned long int uint32_t
- typedef signed long long int int64_t
- typedef unsigned long long int uint64_t

Integer types capable of holding object pointers

These allow you to declare variables of the same size as a pointer.

- typedef int16_t intptr_t
- typedef uint16_t uintptr_t
- typedef int_least24_t intptr24_t
- typedef uint_least24_t uintptr24_t

Minimum-width integer types

Integer types having at least the specified width.

- typedef int8_t int_least8_t
- typedef uint8_t uint_least8_t
- typedef int16_t int_least16_t
- typedef uint16_t uint_least16_t
- typedef int24_t int_least24_t
- typedef uint24_t uint_least24_t
- typedef int32_t int_least32_t
- typedef uint32_t uint_least32_t
- typedef int64_t int_least64_t
- typedef uint64_t uint_least64_t

Fastest minimum-width integer types

Integer types being usually fastest having at least the specified width.

- typedef int8_t int_fast8_t
- typedef uint8_t uint_fast8_t
- typedef int16_t int_fast16_t
- typedef uint16_t uint_fast16_t
- typedef int24_t int_fast24_t
- typedef uint24_t uint_fast24_t
- typedef int32_t int_fast32_t
- typedef uint32_t uint_fast32_t
- typedef int64_t int_fast64_t
- typedef uint64_t uint_fast64_t

Greatest-width integer types

Types designating integer data capable of representing any value of any integer type in the corresponding signed or unsigned category.

- typedef int64_t intmax_t
- typedef uint64_t uintmax_t

Limits of specified-width integer types

C++ implementations should define these macros only when __STDC_LIMIT_MACROS is defined before <stdint.h> is included.

- #define INT8_MAX 0x7f
- #define INT8_MIN (-INT8_MAX - 1)
- #define UINT8_MAX (__CONCAT(INT8_MAX, U) * 2U + 1U)
- #define INT16_MAX 0x7fff
- #define INT16_MIN (-INT16_MAX - 1)
- #define UINT16_MAX (__CONCAT(INT16_MAX, U) * 2U + 1U)
- #define INT24_MAX __INT24_MAX__
- #define INT24_MIN __INT24_MIN__
- #define UINT24_MAX __UINT24_MAX__
- #define INT32_MAX 0x7fffffffL
- #define INT32_MIN (-INT32_MAX - 1L)
- #define UINT32_MAX (__CONCAT(INT32_MAX, U) * 2UL + 1UL)
- #define INT64_MAX 0x7fffffffffffffffLL
- #define INT64_MIN (-INT64_MAX - 1LL)
- #define UINT64_MAX (__CONCAT(INT64_MAX, U) * 2ULL + 1ULL)

Limits of minimum-width integer types

- #define INT_LEAST8_MAX INT8_MAX
- #define INT_LEAST8_MIN INT8_MIN
- #define UINT_LEAST8_MAX UINT8_MAX
- #define INT_LEAST16_MAX INT16_MAX
- #define INT_LEAST16_MIN INT16_MIN
- #define UINT_LEAST16_MAX UINT16_MAX
- #define INT_LEAST24_MAX INT24_MAX
- #define INT_LEAST24_MIN INT24_MIN
- #define UINT_LEAST24_MAX UINT24_MAX
- #define INT_LEAST32_MAX INT32_MAX
- #define INT_LEAST32_MIN INT32_MIN
- #define UINT_LEAST32_MAX UINT32_MAX
- #define INT_LEAST64_MAX INT64_MAX
- #define INT_LEAST64_MIN INT64_MIN
- #define UINT_LEAST64_MAX UINT64_MAX

Limits of fastest minimum-width integer types

- #define INT_FAST8_MAX INT8_MAX
- #define INT_FAST8_MIN INT8_MIN
- #define UINT_FAST8_MAX UINT8_MAX
- #define INT_FAST16_MAX INT16_MAX
- #define INT_FAST16_MIN INT16_MIN
- #define UINT_FAST16_MAX UINT16_MAX
- #define INT_FAST24_MAX INT24_MAX
- #define INT_FAST24_MIN INT24_MIN
- #define UINT_FAST24_MAX UINT24_MAX
- #define INT_FAST32_MAX INT32_MAX
- #define INT_FAST32_MIN INT32_MIN
- #define UINT_FAST32_MAX UINT32_MAX
- #define INT_FAST64_MAX INT64_MAX
- #define INT_FAST64_MIN INT64_MIN
- #define UINT_FAST64_MAX UINT64_MAX

Limits of integer types capable of holding object pointers

- #define INTPTR_MAX INT16_MAX
- #define INTPTR_MIN INT16_MIN
- #define UINTPTR_MAX UINT16_MAX
- #define INTPTR24_MAX INT24_MAX
- #define INTPTR24_MIN INT24_MIN
- #define UINTPTR24_MAX UINT24_MAX

Limits of greatest-width integer types

- #define INTMAX_MAX INT64_MAX
- #define INTMAX_MIN INT64_MIN
- #define UINTMAX_MAX UINT64_MAX

Limits of other integer types

C++ implementations should define these macros only when __STDC_LIMIT_MACROS is defined before <stdint.h> is included.

- #define PTRDIFF_MAX INT16_MAX
- #define PTRDIFF_MIN INT16_MIN
- #define SIG_ATOMIC_MAX INT8_MAX
- #define SIG_ATOMIC_MIN INT8_MIN
- #define SIZE_MAX UINT16_MAX

Macros for integer constants

C++ implementations should define these macros only when __STDC_CONSTANT_MACROS is defined before <stdint.h> is included.

These definitions are valid for integer constants without suffix and for macros defined as integer constant without suffix.

- #define INT8_C(value) ((int8_t) value)
- #define UINT8_C(value) ((uint8_t) __CONCAT(value, U))
- #define INT16_C(value) value
- #define UINT16_C(value) __CONCAT(value, U)
- #define INT24_C(value) ((int24_t) __CONCAT(value, L))
- #define UINT24_C(value) ((uint24_t) __CONCAT(value, UL))
- #define INT32_C(value) __CONCAT(value, L)
- #define UINT32_C(value) __CONCAT(value, UL)
- #define INT64_C(value) __CONCAT(value, LL)
- #define UINT64_C(value) __CONCAT(value, ULL)
- #define INTMAX_C(value) __CONCAT(value, LL)
- #define UINTMAX_C(value) __CONCAT(value, ULL)

20.8.1 Detailed Description

```
#include <stdint.h>
```

Use [u]intN_t if you need exactly N bits.

Since these typedefs are mandated by the C99 standard, they are preferred over rolling your own typedefs.

Note

The 24-bit types and constants are supported since AVR-LibC v2.3. In older versions, they can still be used by means of the compiler built-in types and macros `__int24`, `__uint24`, `__INT24_MIN__`, `__INT24_MAX__` and `__UINT24_MAX__` supported since GCC v4.7.

20.8.2 Macro Definition Documentation

20.8.2.1 INT16_C

```
#define INT16_C(
              value ) value
```

Define a constant of type int16_t

20.8.2.2 INT16_MAX

```
#define INT16_MAX 0x7fff
```

Largest positive value an int16_t can hold.

20.8.2.3 INT16_MIN

```
#define INT16_MIN (-INT16_MAX - 1)
```

Smallest negative value an int16_t can hold.

20.8.2.4 INT24_C

```
#define INT24_C(
              value ) ((int24_t) __CONCAT(value, L))
```

Define a constant of type int24_t

Since

AVR-LibC v2.3, int24

20.8.2.5 INT24_MAX

```
#define INT24_MAX __INT24_MAX__
```

Largest positive value an int24_t can hold.

Since

AVR-LibC v2.3, int24

20.8.2.6 INT24_MIN

```
#define INT24_MIN __INT24_MIN__
```

Smallest negative value an int24_t can hold.

Since

AVR-LibC v2.3, int24

20.8.2.7 INT32_C

```
#define INT32_C(
              value ) __CONCAT(value, L)
```

Define a constant of type int32_t

20.8.2.8 INT32_MAX

```
#define INT32_MAX 0x7fffffffL
```

Largest positive value an int32_t can hold.

20.8.2.9 INT32_MIN

```
#define INT32_MIN (-INT32_MAX - 1L)
```

Smallest negative value an int32_t can hold.

20.8.2.10 INT64_C

```
#define INT64_C(
              value ) __CONCAT(value, LL)
```

Define a constant of type int64_t

20.8.2.11 INT64_MAX

```
#define INT64_MAX 0x7fffffffffffffffLL
```

Largest positive value an int64_t can hold.

20.8.2.12 INT64_MIN

```
#define INT64_MIN (-INT64_MAX - 1LL)
```

Smallest negative value an int64_t can hold.

20.8.2.13 INT8_C

```
#define INT8_C(
              value ) ((int8_t) value)
```

Define a constant of type int8_t

20.8.2.14 INT8_MAX

```
#define INT8_MAX 0x7f
```

Largest positive value an int8_t can hold.

20.8.2.15 INT8_MIN

```
#define INT8_MIN (-INT8_MAX - 1)
```

Smallest negative value an int8_t can hold.

20.8.2.16 INT_FAST16_MAX

```
#define INT_FAST16_MAX INT16_MAX
```

Largest positive value an int_fast16_t can hold.

20.8.2.17 INT_FAST16_MIN

```
#define INT_FAST16_MIN INT16_MIN
```

Smallest negative value an int_fast16_t can hold.

20.8.2.18 INT_FAST24_MAX

```
#define INT_FAST24_MAX INT24_MAX
```

Largest positive value an int_fast24_t can hold.

Since

AVR-LibC v2.3, int24

20.8.2.19 INT_FAST24_MIN

```
#define INT_FAST24_MIN INT24_MIN
```

Smallest negative value an int_fast24_t can hold.

Since

AVR-LibC v2.3, int24

20.8.2.20 INT_FAST32_MAX

```
#define INT_FAST32_MAX INT32_MAX
```

Largest positive value an int_fast32_t can hold.

20.8.2.21 INT_FAST32_MIN

```
#define INT_FAST32_MIN INT32_MIN
```

Smallest negative value an int_fast32_t can hold.

20.8.2.22 INT_FAST64_MAX

```
#define INT_FAST64_MAX INT64_MAX
```

Largest positive value an int_fast64_t can hold.

20.8.2.23 INT_FAST64_MIN

```
#define INT_FAST64_MIN INT64_MIN
```

Smallest negative value an int_fast64_t can hold.

20.8.2.24 INT_FAST8_MAX

```
#define INT_FAST8_MAX INT8_MAX
```

Largest positive value an int_fast8_t can hold.

20.8.2.25 INT_FAST8_MIN

```
#define INT_FAST8_MIN INT8_MIN
```

Smallest negative value an int_fast8_t can hold.

20.8.2.26 INT_LEAST16_MAX

```
#define INT_LEAST16_MAX INT16_MAX
```

Largest positive value an int_least16_t can hold.

20.8.2.27 INT_LEAST16_MIN

```
#define INT_LEAST16_MIN INT16_MIN
```

Smallest negative value an int_least16_t can hold.

20.8.2.28 INT_LEAST24_MAX

```
#define INT_LEAST24_MAX INT24_MAX
```

Largest positive value an int_least24_t can hold.

Since

AVR-LibC v2.3, int24

20.8.2.29 INT_LEAST24_MIN

```
#define INT_LEAST24_MIN INT24_MIN
```

Smallest negative value an int_least24_t can hold.

Since

AVR-LibC v2.3, int24

20.8.2.30 INT_LEAST32_MAX

```
#define INT_LEAST32_MAX INT32_MAX
```

Largest positive value an int_least32_t can hold.

20.8.2.31 INT_LEAST32_MIN

```
#define INT_LEAST32_MIN INT32_MIN
```

Smallest negative value an int_least32_t can hold.

20.8.2.32 INT_LEAST64_MAX

```
#define INT_LEAST64_MAX INT64_MAX
```

Largest positive value an int_least64_t can hold.

20.8.2.33 INT_LEAST64_MIN

```
#define INT_LEAST64_MIN INT64_MIN
```

Smallest negative value an int_least64_t can hold.

20.8.2.34 INT_LEAST8_MAX

```
#define INT_LEAST8_MAX INT8_MAX
```

Largest positive value an int_least8_t can hold.

20.8.2.35 INT_LEAST8_MIN

```
#define INT_LEAST8_MIN INT8_MIN
```

Smallest negative value an int_least8_t can hold.

20.8.2.36 INTMAX_C

```
#define INTMAX_C(
              value ) __CONCAT(value, LL)
```

Define a constant of type intmax_t

20.8.2.37 INTMAX_MAX

```
#define INTMAX_MAX INT64_MAX
```

Largest positive value an intmax_t can hold.

20.8.2.38 INTMAX_MIN

```
#define INTMAX_MIN INT64_MIN
```

Smallest negative value an intmax_t can hold.

20.8.2.39 INTPTR24_MAX

```
#define INTPTR24_MAX INT24_MAX
```

Largest positive value an intptr24_t can hold.

Since

AVR-LibC v2.3, int24

20.8.2.40 INTPTR24_MIN

```
#define INTPTR24_MIN INT24_MIN
```

Smallest negative value an intptr24_t can hold.

Since

AVR-LibC v2.3, int24

20.8.2.41 INTPTR_MAX

```
#define INTPTR_MAX INT16_MAX
```

Largest positive value an intptr_t can hold.

20.8.2.42 INTPTR_MIN

```
#define INTPTR_MIN INT16_MIN
```

Smallest negative value an intptr_t can hold.

20.8.2.43 PTRDIFF_MAX

```
#define PTRDIFF_MAX INT16_MAX
```

Largest positive value a ptrdiff_t can hold.

20.8.2.44 PTRDIFF_MIN

```
#define PTRDIFF_MIN INT16_MIN
```

Smallest negative value a ptrdiff_t can hold.

20.8.2.45 SIG_ATOMIC_MAX

```
#define SIG_ATOMIC_MAX INT8_MAX
```

Largest positive value a sig_atomic_t can hold.

20.8.2.46 SIG_ATOMIC_MIN

```
#define SIG_ATOMIC_MIN INT8_MIN
```

Smallest negative value a sig_atomic_t can hold.

20.8.2.47 SIZE_MAX

```
#define SIZE_MAX UINT16_MAX
```

Largest value a size_t can hold.

20.8.2.48 UINT16_C

```
#define UINT16_C(
              value ) __CONCAT(value, U)
```

Define a constant of type uint16_t

20.8.2.49 UINT16_MAX

```
#define UINT16_MAX (__CONCAT(INT16_MAX, U) * 2U + 1U)
```

Largest value an uint16_t can hold.

20.8.2.50 UINT24_C

```
#define UINT24_C(
              value ) ((uint24_t) __CONCAT(value, UL))
```

Define a constant of type uint24_t

Since

AVR-LibC v2.3, int24

20.8.2.51 UINT24_MAX

```
#define UINT24_MAX __UINT24_MAX__
```

Largest value an uint24_t can hold.

Since

AVR-LibC v2.3, int24

20.8.2.52 UINT32_C

```
#define UINT32_C(
              value ) __CONCAT(value, UL)
```

Define a constant of type uint32_t

20.8.2.53 UINT32_MAX

```
#define UINT32_MAX (__CONCAT(INT32_MAX, U) * 2UL + 1UL)
```

Largest value an uint32_t can hold.

20.8.2.54 UINT64_C

```
#define UINT64_C(
              value ) __CONCAT(value, ULL)
```

Define a constant of type uint64_t

20.8.2.55 UINT64_MAX

```
#define UINT64_MAX (__CONCAT(INT64_MAX, U) * 2ULL + 1ULL)
```

Largest value an uint64_t can hold.

20.8.2.56 UINT8_C

```
#define UINT8_C(
              value ) ((uint8_t) __CONCAT(value, U))
```

Define a constant of type uint8_t

20.8.2.57 UINT8_MAX

```
#define UINT8_MAX (__CONCAT(INT8_MAX, U) * 2U + 1U)
```

Largest value an uint8_t can hold.

20.8.2.58 UINT_FAST16_MAX

```
#define UINT_FAST16_MAX UINT16_MAX
```

Largest value an uint_fast16_t can hold.

20.8.2.59 UINT_FAST24_MAX

```
#define UINT_FAST24_MAX UINT24_MAX
```

Largest value an uint_fast24_t can hold.

Since

AVR-LibC v2.3, int24

20.8.2.60 UINT_FAST32_MAX

```
#define UINT_FAST32_MAX UINT32_MAX
```

Largest value an uint_fast32_t can hold.

20.8.2.61 UINT_FAST64_MAX

```
#define UINT_FAST64_MAX UINT64_MAX
```

Largest value an uint_fast64_t can hold.

20.8.2.62 UINT_FAST8_MAX

```
#define UINT_FAST8_MAX UINT8_MAX
```

Largest value an uint_fast8_t can hold.

20.8.2.63 UINT_LEAST16_MAX

```
#define UINT_LEAST16_MAX UINT16_MAX
```

Largest value an uint_least16_t can hold.

20.8.2.64 UINT_LEAST24_MAX

```
#define UINT_LEAST24_MAX UINT24_MAX
```

Largest value an uint_least24_t can hold.

Since

AVR-LibC v2.3, int24

20.8.2.65 UINT_LEAST32_MAX

```
#define UINT_LEAST32_MAX UINT32_MAX
```

Largest value an uint_least32_t can hold.

20.8.2.66 UINT_LEAST64_MAX

```
#define UINT_LEAST64_MAX UINT64_MAX
```

Largest value an uint_least64_t can hold.

20.8.2.67 UINT_LEAST8_MAX

```
#define UINT_LEAST8_MAX UINT8_MAX
```

Largest value an uint_least8_t can hold.

20.8.2.68 UINTMAX_C

```
#define UINTMAX_C(
              value ) __CONCAT(value, ULL)
```

Define a constant of type uintmax_t

20.8.2.69 UINTMAX_MAX

```
#define UINTMAX_MAX UINT64_MAX
```

Largest value an uintmax_t can hold.

20.8.2.70 UINTPTR24_MAX

```
#define UINTPTR24_MAX UINT24_MAX
```

Largest value an uintptr24_t can hold.

Since

AVR-LibC v2.3, int24

20.8.2.71 UINTPTR_MAX

```
#define UINTPTR_MAX UINT16_MAX
```

Largest value an uintptr_t can hold.

20.8.3 Typedef Documentation

20.8.3.1 int16_t

```
typedef signed int int16_t
```

16-Bit signed integer type.

20.8.3.2 int24_t

```
typedef __int24 int24_t
```

24-Bit signed integer type.

Since

AVR-LibC v2.3, int24

20.8.3.3 int32_t

```
typedef signed long int int32_t
```

32-Bit signed integer type.

20.8.3.4 int64_t

```
typedef signed long long int int64_t
```

64-Bit signed integer type.

Note

This type is not available when the compiler option `-mint8` is in effect.

20.8.3.5 int8_t

```
typedef signed char int8_t
```

8-Bit signed integer type.

20.8.3.6 int_fast16_t

```
typedef int16_t int_fast16_t
```

Fastest signed integer type with at least 16 bits.

20.8.3.7 int_fast24_t

```
typedef int24_t int_fast24_t
```

Fastest signed integer type with at least 24 bits.

Since

AVR-LibC v2.3, int24

20.8.3.8 int_fast32_t

```
typedef int32_t int_fast32_t
```

Fastest signed integer type with at least 32 bits.

20.8.3.9 int_fast64_t

```
typedef int64_t int_fast64_t
```

Fastest signed integer type with at least 64 bits.

Note

This type is not available when the compiler option -mint8 is in effect.

20.8.3.10 int_fast8_t

```
typedef int8_t int_fast8_t
```

Fastest signed integer type with at least 8 bits.

20.8.3.11 int_least16_t

```
typedef int16_t int_least16_t
```

Signed integer type with at least 16 bits.

20.8.3.12 int_least24_t

```
typedef int24_t int_least24_t
```

Signed integer type with at least 24 bits.

Since

AVR-LibC v2.3, int24

20.8.3.13 int_least32_t

```
typedef int32_t int_least32_t
```

Signed integer type with at least 32 bits.

20.8.3.14 int_least64_t

```
typedef int64_t int_least64_t
```

Signed integer type with at least 64 bits.

Note

This type is not available when the compiler option `-mint8` is in effect.

20.8.3.15 int_least8_t

```
typedef int8_t int_least8_t
```

Signed integer type with at least 8 bits.

20.8.3.16 intmax_t

```
typedef int64_t intmax_t
```

Largest signed integer available.

20.8.3.17 intptr24_t

```
typedef int_least24_t intptr24_t
```

Signed 24-bit pointer compatible type.

Since

AVR-LibC v2.3, int24

20.8.3.18 intptr_t

```
typedef int16_t intptr_t
```

Signed pointer compatible type.

20.8.3.19 uint16_t

```
typedef unsigned int uint16_t
```

16-Bit unsigned integer type.

20.8.3.20 uint24_t

```
typedef __uint24 uint24_t
```

24-Bit unsigned integer type.

Since

AVR-LibC v2.3, int24

20.8.3.21 uint32_t

```
typedef unsigned long int uint32_t
```

32-Bit unsigned integer type.

20.8.3.22 uint64_t

```
typedef unsigned long long int uint64_t
```

64-Bit unsigned integer type.

Note

This type is not available when the compiler option `-mint8` is in effect.

20.8.3.23 uint8_t

```
typedef unsigned char uint8_t
```

8-Bit unsigned integer type.

20.8.3.24 uint_fast16_t

```
typedef uint16_t uint_fast16_t
```

Fastest unsigned integer type with at least 16 bits.

20.8.3.25 uint_fast24_t

```
typedef uint24_t uint_fast24_t
```

Fastest unsigned integer type with at least 24 bits.

Since

AVR-LibC v2.3, int24

20.8.3.26 uint_fast32_t

```
typedef uint32_t uint_fast32_t
```

Fastest unsigned integer type with at least 32 bits.

20.8.3.27 uint_fast64_t

```
typedef uint64_t uint_fast64_t
```

Fastest unsigned integer type with at least 64 bits.

Note

This type is not available when the compiler option -mint8 is in effect.

20.8.3.28 uint_fast8_t

```
typedef uint8_t uint_fast8_t
```

Fastest unsigned integer type with at least 8 bits.

20.8.3.29 uint_least16_t

```
typedef uint16_t uint_least16_t
```

Unsigned integer type with at least 16 bits.

20.8.3.30 uint_least24_t

```
typedef uint24_t uint_least24_t
```

Unsigned integer type with at least 24 bits.

Since

AVR-LibC v2.3, int24

20.8.3.31 uint_least32_t

```
typedef uint32_t uint_least32_t
```

Unsigned integer type with at least 32 bits.

20.8.3.32 uint_least64_t

```
typedef uint64_t uint_least64_t
```

Unsigned integer type with at least 64 bits.

Note

This type is not available when the compiler option `-mint8` is in effect.

20.8.3.33 uint_least8_t

```
typedef uint8_t uint_least8_t
```

Unsigned integer type with at least 8 bits.

20.8.3.34 uintmax_t

```
typedef uint64_t uintmax_t
```

Largest unsigned integer available.

20.8.3.35 uintptr24_t

```
typedef uint_least24_t uintptr24_t
```

Unsigned 24-bit pointer compatible type.

Since

AVR-LibC v2.3, int24

20.8.3.36 uintptr_t

```
typedef uint16_t uintptr_t
```

Unsigned pointer compatible type.

20.9 <stdio.h>: Standard IO facilities

Macros

- #define stdin (__iob[0])
- #define stdout (__iob[1])
- #define stderr (__iob[2])
- #define EOF (-1)
- #define fdev_set_udata(stream, u) do { (stream)->udata = u; } while(0)
- #define fdev_get_udata(stream) ((stream)->udata)
- #define fdev_setup_stream(stream, put, get, rwflag)
- #define _FDEV_SETUP_READ __SRD
- #define _FDEV_SETUP_WRITE __SWR
- #define _FDEV_SETUP_RW (__SRD|__SWR)
- #define _FDEV_ERR (-1)
- #define _FDEV_EOF (-2)
- #define FDEV_SETUP_STREAM(put, get, rwflag)
- #define fdev_close() ((void)0)
- #define putc(__c, __stream) fputc(__c, __stream)
- #define putchar(__c) fputc(__c, stdout)
- #define getc(__stream) fgetc(__stream)
- #define getchar() fgetc(stdin)

Typedefs

- typedef struct __file FILE

Functions

- int fclose (FILE *__stream)
- int vfprintf (FILE *__stream, const char *__fmt, va_list __ap)
- int vfprintf_P (FILE *__stream, const char *__fmt, va_list __ap)
- int fputc (int __c, FILE *__stream)
- int printf (const char *__fmt,...)
- int printf_P (const char *__fmt,...)
- int vprintf (const char *__fmt, va_list __ap)
- int sprintf (char *__s, const char *__fmt,...)
- int sprintf_P (char *__s, const char *__fmt,...)
- int snprintf (char *__s, size_t __n, const char *__fmt,...)
- int snprintf_P (char *__s, size_t __n, const char *__fmt,...)
- int vsprintf (char *__s, const char *__fmt, va_list __ap)
- int vsprintf_P (char *__s, const char *__fmt, va_list __ap)
- int vsnprintf (char *__s, size_t __n, const char *__fmt, va_list __ap)
- int vsnprintf_P (char *__s, size_t __n, const char *__fmt, va_list __ap)
- int fprintf (FILE *__stream, const char *__fmt,...)
- int fprintf_P (FILE *__stream, const char *__fmt,...)
- int fputs (const char *__str, FILE *__stream)
- int fputs_P (const char *__str, FILE *__stream)
- int puts (const char *__str)
- int puts_P (const char *__str)
- size_t fwrite (const void *__ptr, size_t __size, size_t __nmemb, FILE *__stream)
- int fgetc (FILE *__stream)
- int ungetc (int __c, FILE *__stream)
- char * fgets (char *__str, int __size, FILE *__stream)
- char * gets (char *__str)
- size_t fread (void *__ptr, size_t __size, size_t __nmemb, FILE *__stream)
- void clearerr (FILE *__stream)
- int feof (FILE *__stream)
- int ferror (FILE *__stream)
- int vfscanf (FILE *__stream, const char *__fmt, va_list __ap)
- int vfscanf_P (FILE *__stream, const char *__fmt, va_list __ap)
- int fscanf (FILE *__stream, const char *__fmt,...)
- int fscanf_P (FILE *__stream, const char *__fmt,...)
- int scanf (const char *__fmt,...)
- int scanf_P (const char *__fmt,...)
- int vscanf (const char *__fmt, va_list __ap)
- int sscanf (const char *__buf, const char *__fmt,...)
- int sscanf_P (const char *__buf, const char *__fmt,...)
- int fflush (FILE *stream)
- FILE * fdevopen (int(*put)(char, FILE *), int(*get)(FILE *))

20.9.1 Detailed Description

```
#include <stdio.h>
```

Introduction to the Standard IO facilities This file declares the standard IO facilities that are implemented in AVR-LibC. Due to the nature of the underlying hardware, only a limited subset of standard IO is implemented. There is no actual file implementation available, so only device IO can be performed. Since there's no operating system, the application needs to provide enough details about their devices in order to make them usable by the standard IO facilities.

Due to space constraints, some functionality has not been implemented at all (like some of the `printf` conversions that have been left out). Nevertheless, potential users of this implementation should be warned: the `printf` and `scanf` families of functions, although usually associated with presumably simple things like the famous "Hello, world!" program, are actually fairly complex which causes their inclusion to eat up a fair amount of code space. Also, they are not fast due to the nature of interpreting the format string at run-time. Whenever possible, resorting to the (sometimes non-standard) predetermined conversion facilities that are offered by AVR-LibC will usually cost much less in terms of speed and code size.

Tunable options for code size vs. feature set In order to allow programmers a code size vs. functionality tradeoff, the function vfprintf() which is the heart of the printf family can be selected in different flavours using linker options. See the documentation of vfprintf() for a detailed description. The same applies to vfscanf() and the `scanf` family of functions.

Outline of the chosen API The standard streams `stdin`, `stdout`, and `stderr` are provided, but contrary to the C standard, since AVR-LibC has no knowledge about applicable devices, these streams are not already pre-initialized at application startup. Also, since there is no notion of "file" whatsoever to AVR-LibC, there is no function `fopen()` that could be used to associate a stream to some device. (See note 1.) Instead, the function `fdevopen()` is provided to associate a stream to a device, where the device needs to provide a function to send a character, to receive a character, or both. There is no differentiation between "text" and "binary" streams inside AVR-LibC. Character `\n` is sent literally down to the device's `put()` function. If the device requires a carriage return (`\r`) character to be sent before the linefeed, its `put()` routine must implement this (see note 2).

As an alternative method to fdevopen(), the macro fdev_setup_stream() might be used to setup a user-supplied FILE structure.

It should be noted that the automatic conversion of a newline character into a carriage return - newline sequence breaks binary transfers. If binary transfers are desired, no automatic conversion should be performed, but instead any string that aims to issue a CR-LF sequence must use `"\r\n"` explicitly.

For convenience, the first call to `fdevopen()` that opens a stream for reading will cause the resulting stream to be aliased to `stdin`. Likewise, the first call to `fdevopen()` that opens a stream for writing will cause the resulting stream to be aliased to both, `stdout`, and `stderr`. Thus, if the open was done with both, read and write intent, all three standard streams will be identical. Note that these aliases are indistinguishable from each other, thus calling `fclose()` on such a stream will also effectively close all of its aliases (note 3).

It is possible to tie additional user data to a stream, using fdev_set_udata(). The backend put and get functions can then extract this user data using fdev_get_udata(), and act appropriately. For example, a single put function could be used to talk to two different UARTs that way, or the put and get functions could keep internal state between calls there.

Format strings in flash ROM All the `printf` and `scanf` family functions come in three flavours: the standard name, where the format string is expected to be in SRAM, as well as two versions with the suffix "↩ _P" and "_F" where the format string is expected to reside in the flash ROM. The macros PSTR (explained in <avr/pgmspace.h>: Program Space Utilities) and FSTR (explained in <avr/flash.h>) become very handy for declaring these format strings.

Running stdio without malloc() By default, fdevopen() requires malloc(). As this is often not desired in the limited environment of a microcontroller, an alternative option is provided to run completely without malloc().

The macro fdev_setup_stream() is provided to prepare a user-supplied FILE buffer for operation with stdio.

Example `#include <stdio.h>`

```
static FILE mystdout = FDEV_SETUP_STREAM (uart_putchar, NULL,
                                          _FDEV_SETUP_WRITE);
static int
uart_putchar (char c, FILE *stream)
{
    if (c == '\n')
        uart_putchar ('\r', stream);
    loop_until_bit_is_set (UCSRA, UDRE);
    UDR = c;
    return 0;
}

int main (void)
{
    init_uart();
    stdout = &mystdout;
    printf ("Hello, world!\n");
    return 0;
}
```

This example uses the initializer form FDEV_SETUP_STREAM() rather than the function-like fdev_setup_stream(), so all data initialization happens during C start-up.

If streams initialized that way are no longer needed, they can be destroyed by first calling the macro fdev_close(), and then destroying the object itself. No call to fclose() should be issued for these streams. While calling fclose() itself is harmless, it will cause an undefined reference to free() and thus cause the linker to pull in the malloc module into the application.

Note 1:
It might have been possible to implement a device abstraction that is compatible with `fopen()` but since this would have required to parse a string, and to take all the information needed either out of this string, or out of an additional table that would need to be provided by the application, this approach was not taken.

Note 2:
This basically follows the Unix approach: if a device such as a terminal needs special handling, it is in the domain of the terminal device driver to provide this functionality. Thus, a simple function suitable as `put()` for `fdevopen()` that talks to a UART interface might look like this:

```
int uart_putchar (char c, FILE *stream)
{
    if (c == '\n')
        uart_putchar ('\r', stream);
    loop_until_bit_is_set (UCSRA, UDRE);
    UDR = c;
    return 0;
}
```

Note 3:
This implementation has been chosen because the cost of maintaining an alias is considerably smaller than the cost of maintaining full copies of each stream. Yet, providing an implementation that offers the complete set of standard streams was deemed to be useful. Not only that writing `printf()` instead of `fprintf(mystream, ...)` saves typing work, but since avr-gcc needs to resort to pass all arguments of variadic functions on the stack (as opposed to passing them in registers for functions that take a fixed number of parameters), the ability to pass one parameter less by implying `stdin` or stdout will also save some execution time.

20.9.2 Macro Definition Documentation

20.9.2.1 _FDEV_EOF

```
#define _FDEV_EOF (-2)
```

Return code for an end-of-file condition during device read.

To be used in the get function of fdevopen().

20.9.2.2 _FDEV_ERR

```
#define _FDEV_ERR (-1)
```

Return code for an error condition during device read.

To be used in the get function of fdevopen().

20.9.2.3 _FDEV_SETUP_READ

```
#define _FDEV_SETUP_READ __SRD
```

fdev_setup_stream() with read intent

20.9.2.4 _FDEV_SETUP_RW

```
#define _FDEV_SETUP_RW (__SRD|__SWR)
```

fdev_setup_stream() with read/write intent

20.9.2.5 _FDEV_SETUP_WRITE

```
#define _FDEV_SETUP_WRITE __SWR
```

fdev_setup_stream() with write intent

20.9.2.6 EOF

```
#define EOF (-1)
```

`EOF` declares the value that is returned by various standard IO functions in case of an error. Since the AVR platform (currently) doesn't contain an abstraction for actual files, its origin as "end of file" is somewhat meaningless here.

20.9.2.7 fdev_close

```
#define fdev_close( ) ((void)0)
```

This macro frees up any library resources that might be associated with `stream`. It should be called if `stream` is no longer needed, right before the application is going to destroy the `stream` object itself.

(Currently, this macro evaluates to nothing, but this might change in future versions of the library.)

20.9.2.8 fdev_get_udata

```
#define fdev_get_udata(
              stream ) ((stream)->udata)
```

This macro retrieves a pointer to user defined data from a FILE stream object.

20.9.2.9 fdev_set_udata

```
#define fdev_set_udata(
              stream,
              u ) do { (stream)->udata = u; } while(0)
```

This macro inserts a pointer to user defined data into a FILE stream object.

The user data can be useful for tracking state in the put and get functions supplied to the fdevopen() function.

20.9.2.10 FDEV_SETUP_STREAM

```
#define FDEV_SETUP_STREAM(
              put,
              get,
              rwflag )
```

Initializer for a user-supplied stdio stream.

This macro acts similar to fdev_setup_stream(), but it is to be used as the initializer of a variable of type FILE.

The remaining arguments are to be used as explained in fdev_setup_stream().

20.9.2.11 fdev_setup_stream

```
#define fdev_setup_stream(
              stream,
              put,
              get,
              rwflag )
```

Setup a user-supplied buffer as an stdio stream.

This macro takes a user-supplied buffer `stream`, and sets it up as a stream that is valid for stdio operations, similar to one that has been obtained dynamically from fdevopen(). The buffer to setup must be of type FILE.

The arguments `put` and `get` are identical to those that need to be passed to fdevopen().

The `rwflag` argument can take one of the values _FDEV_SETUP_READ, _FDEV_SETUP_WRITE, or _FDEV_SETUP_RW, for read, write, or read/write intent, respectively.

Note

No assignments to the standard streams will be performed by fdev_setup_stream(). If standard streams are to be used, these need to be assigned by the user. See also under Running stdio without malloc().

20.9.2.12 getc

```
#define getc(
             __stream ) fgetc(__stream)
```

The macro `getc` used to be a "fast" macro implementation with a functionality identical to fgetc(). For space constraints, in AVR-LibC, it is just an alias for `fgetc`.

20.9.2.13 getchar

```
#define getchar(
              void ) fgetc(stdin)
```

The macro `getchar` reads a character from `stdin`. Return values and error handling is identical to fgetc().

20.9.2.14 putc

```
#define putc(
             __c,
             __stream ) fputc(__c, __stream)
```

The macro `putc` used to be a "fast" macro implementation with a functionality identical to fputc(). For space constraints, in AVR-LibC, it is just an alias for `fputc`.

20.9.2.15 putchar

```
#define putchar(
             __c ) fputc(__c, stdout)
```

The macro `putchar` sends character `c` to `stdout`.

20.9.2.16 stderr

```
#define stderr (__iob[2])
```

Stream destined for error output. Unless specifically assigned, identical to `stdout`.

If `stderr` should point to another stream, the result of another `fdevopen()` must be explicitly assigned to it without closing the previous `stderr` (since this would also close `stdout`).

20.9.2.17 stdin

```
#define stdin (__iob[0])
```

Stream that will be used as an input stream by the simplified functions that don't take a `stream` argument.

The first stream opened with read intent using `fdevopen()` will be assigned to `stdin`.

20.9.2.18 stdout

```
#define stdout (__iob[1])
```

Stream that will be used as an output stream by the simplified functions that don't take a `stream` argument.

The first stream opened with write intent using `fdevopen()` will be assigned to both, `stdin`, and `stderr`.

20.9.3 Typedef Documentation

20.9.3.1 FILE

```
typedef struct __file FILE
```

`FILE` is the opaque structure that is passed around between the various standard IO functions.

20.9.4 Function Documentation

20.9.4.1 clearerr()

```
void clearerr (
            FILE * __stream )
```

Clear the error and end-of-file flags of `stream`.

20.9.4.2 fclose()

```
int fclose (
            FILE * __stream )
```

This function closes `stream`, and disallows and further IO to and from it.

When using fdevopen() to setup the stream, a call to fclose() is needed in order to free the internal resources allocated.

If the stream has been set up using fdev_setup_stream() or FDEV_SETUP_STREAM(), use fdev_close() instead.

It currently always returns 0 (for success).

20.9.4.3 fdevopen()

```
FILE * fdevopen (
            int(*)(char, FILE *) put,
            int(*)(FILE *) get )
```

This function is a replacement for `fopen()`.

It opens a stream for a device where the actual device implementation needs to be provided by the application. If successful, a pointer to the structure for the opened stream is returned. Reasons for a possible failure currently include that neither the `put` nor the `get` argument have been provided, thus attempting to open a stream with no IO intent at all, or that insufficient dynamic memory is available to establish a new stream.

If the `put` function pointer is provided, the stream is opened with write intent. The function passed as `put` shall take two arguments, the first a character to write to the device, and the second a pointer to FILE, and shall return 0 if the output was successful, and a nonzero value if the character could not be sent to the device.

If the `get` function pointer is provided, the stream is opened with read intent. The function passed as `get` shall take a pointer to FILE as its single argument, and return one character from the device, passed as an `int` type. If an error occurs when trying to read from the device, it shall return `_FDEV_ERR`. If an end-of-file condition was reached while reading from the device, `_FDEV_EOF` shall be returned.

If both functions are provided, the stream is opened with read and write intent.

The first stream opened with read intent is assigned to `stdin`, and the first one opened with write intent is assigned to both, `stdout` and `stderr`.

fdevopen() uses calloc() (und thus malloc()) in order to allocate the storage for the new stream.

Note

If the macro __STDIO_FDEVOPEN_COMPAT_12 is declared before including <stdio.h>, a function prototype for fdevopen() will be chosen that is backwards compatible with AVR-LibC version 1.2 and before. This is solely intented for providing a simple migration path without the need to immediately change all source code. Do not use for new code.

20.9.4.4 feof()

```
int feof (
             FILE * __stream )
```

Test the end-of-file flag of `stream`. This flag can only be cleared by a call to clearerr().

20.9.4.5 ferror()

```
int ferror (
             FILE * __stream )
```

Test the error flag of `stream`. This flag can only be cleared by a call to clearerr().

20.9.4.6 fflush()

```
int fflush (
             FILE * stream )
```

Flush `stream`.

This is a null operation provided for source-code compatibility only, as the standard IO implementation currently does not perform any buffering.

20.9.4.7 fgetc()

```
int fgetc (
             FILE * __stream )
```

The function `fgetc` reads a character from `stream`. It returns the character, or `EOF` in case end-of-file was encountered or an error occurred. The routines feof() or ferror() must be used to distinguish between both situations.

20.9.4.8 fgets()

```
char * fgets (
             char * __str,
             int __size,
             FILE * __stream )
```

Read at most `size - 1` bytes from `stream`, until a newline character was encountered, and store the characters in the buffer pointed to by `str`. Unless an error was encountered while reading, the string will then be terminated with a `NUL` character.

If an error was encountered, the function returns NULL and sets the error flag of `stream`, which can be tested using ferror(). Otherwise, a pointer to the string will be returned.

20.9.4.9 fprintf()

```
int fprintf (
             FILE * __stream,
             const char * __fmt,
              ...  )
```

The function `fprintf` performs formatted output to `stream`. See `vfprintf()` for details.

20.9.4.10 fprintf_P()

```
int fprintf_P (
             FILE * __stream,
             const char * __fmt,
              ...  )
```

Variant of `fprintf()` that uses a `fmt` string that resides in program memory. See also fprintf_F.

20.9.4.11 fputc()

```
int fputc (
             int __c,
             FILE * __stream )
```

The function `fputc` sends the character `c` (though given as type `int`) to `stream`. It returns the character, or `EOF` in case an error occurred.

20.9.4.12 fputs()

```
int fputs (
             const char * __str,
             FILE * __stream )
```

Write the string pointed to by `str` to stream `stream`.

Returns 0 on success and EOF on error.

20.9.4.13 fputs_P()

```
int fputs_P (
             const char * __str,
             FILE * __stream )
```

Variant of fputs() where `str` resides in program memory. See also fputs_F.

20.9.4.14 fread()

```
size_t fread (
             void * __ptr,
             size_t __size,
             size_t __nmemb,
             FILE * __stream )
```

Read `nmemb` objects, `size` bytes each, from `stream`, to the buffer pointed to by `ptr`.

Returns the number of objects successfully read, i. e. `nmemb` unless an input error occurred or end-of-file was encountered. feof() and ferror() must be used to distinguish between these two conditions.

20.9.4.15 fscanf()

```
int fscanf (
             FILE * __stream,
             const char * __fmt,
              ...  )
```

The function `fscanf` performs formatted input, reading the input data from `stream`.

See vfscanf() for details.

20.9.4.16 fscanf_P()

```
int fscanf_P (
             FILE * __stream,
             const char * __fmt,
              ...  )
```

Variant of fscanf() using a `fmt` string in program memory. See also fscanf_F.

20.9.4.17 fwrite()

```
size_t fwrite (
             const void * __ptr,
             size_t __size,
             size_t __nmemb,
             FILE * __stream )
```

Write `nmemb` objects, `size` bytes each, to `stream`. The first byte of the first object is referenced by `ptr`.

Returns the number of objects successfully written, i. e. `nmemb` unless an output error occurred.

20.9.4.18 gets()

```
char * gets (
             char * __str )
```

Similar to fgets() except that it will operate on stream `stdin`, and the trailing newline (if any) will not be stored in the string. It is the caller's responsibility to provide enough storage to hold the characters read.

20.9.4.19 printf()

```
int printf (
             const char * __fmt,
              ...  )
```

The function `printf` performs formatted output to stream `stdout`. See `vfprintf()` for details.

20.9.4.20 printf_P()

```
int printf_P (
             const char * __fmt,
              ...  )
```

Variant of `printf()` that uses a `fmt` string that resides in program memory. See also printf_F.

20.9.4.21 puts()

```
int puts (
             const char * __str )
```

Write the string pointed to by `str`, and a trailing newline character, to `stdout`.

20.9.4.22 puts_P()

```
int puts_P (
             const char * __str )
```

Variant of puts() where `str` resides in program memory. See also puts_F.

20.9.4.23 scanf()

```
int scanf (
             const char * __fmt,
              ...  )
```

The function `scanf` performs formatted input from stream `stdin`.

See vfscanf() for details.

20.9.4.24 scanf_P()

```
int scanf_P (
             const char * __fmt,
              ...  )
```

Variant of scanf() where `fmt` resides in program memory. See also scanf_F.

20.9.4.25 snprintf()

```
int snprintf (
             char * __s,
             size_t __n,
             const char * __fmt,
              ...  )
```

Like `sprintf()`, but instead of assuming `s` to be of infinite size, no more than `n` characters (including the trailing NUL character) will be converted to `s`.

Returns the number of characters that would have been written to `s` if there were enough space.

20.9.4.26 snprintf_P()

```
int snprintf_P (
             char * __s,
             size_t __n,
             const char * __fmt,
              ...  )
```

Variant of `snprintf()` that uses a `fmt` string that resides in program memory. See also snprintf_F.

20.9.4.27 sprintf()

```
int sprintf (
             char * __s,
             const char * __fmt,
              ...  )
```

Variant of `printf()` that sends the formatted characters to string `s`.

20.9.4.28 sprintf_P()

```
int sprintf_P (
             char * __s,
             const char * __fmt,
              ...  )
```

Variant of `sprintf()` that uses a `fmt` string that resides in program memory. See also sprintf_F.

20.9.4.29 sscanf()

```
int sscanf (
             const char * __buf,
             const char * __fmt,
              ...  )
```

The function `sscanf` performs formatted input, reading the input data from the buffer pointed to by `buf`.

See vfscanf() for details.

20.9.4.30 sscanf_P()

```
int sscanf_P (
             const char * __buf,
             const char * __fmt,
              ...  )
```

Variant of sscanf() using a `fmt` string in program memory. See also sscanf_F.

20.9.4.31 ungetc()

```
int ungetc (
             int __c,
             FILE * __stream )
```

The ungetc() function pushes the character `c` (converted to an unsigned char) back onto the input stream pointed to by `stream`. The pushed-back character will be returned by a subsequent read on the stream.

Currently, only a single character can be pushed back onto the stream.

The ungetc() function returns the character pushed back after the conversion, or `EOF` if the operation fails. If the value of the argument `c` character equals `EOF`, the operation will fail and the stream will remain unchanged.

20.9.4.32 vfprintf()

```
int vfprintf (
             FILE * __stream,
             const char * __fmt,
             va_list __ap )
```

`vfprintf` is the central facility of the `printf` family of functions. It outputs values to `stream` under control of a format string passed in `fmt`. The actual values to print are passed as a variable argument list `ap`.

`vfprintf` returns the number of characters written to `stream`, or `EOF` in case of an error. Currently, this will only happen if `stream` has not been opened with write intent.

The format string is composed of zero or more directives: ordinary characters (not `%`), which are copied unchanged to the output stream; and conversion specifications, each of which results in fetching zero or more subsequent arguments. Each conversion specification is introduced by the `%` character. The arguments must properly correspond (after type promotion) with the conversion specifier. After the `%`, the following appear in sequence:

- Zero or more of the following flags:
  - `#` The value should be converted to an "alternate form". For c, d, i, s, and u conversions, this option has no effect. For o conversions, the precision of the number is increased to force the first character of the output string to a zero (except if a zero value is printed with an explicit precision of zero). For x and X conversions, a non-zero result has the string `0x' (or `0X' for X conversions) prepended to it.
  - `0` (zero) Zero padding. For all conversions, the converted value is padded on the left with zeros rather than blanks. If a precision is given with a numeric conversion (d, i, o, u, i, x, and X), the 0 flag is ignored.
  - `-` A negative field width flag; the converted value is to be left adjusted on the field boundary. The converted value is padded on the right with blanks, rather than on the left with blanks or zeros. A - overrides a 0 if both are given.
  - ' ' (space) A blank should be left before a positive number produced by a signed conversion (d, or i).
  - `+` A sign must always be placed before a number produced by a signed conversion. A + overrides a space if both are used.
- An optional decimal digit string specifying a minimum field width. If the converted value has fewer characters than the field width, it will be padded with spaces on the left (or right, if the left-adjustment flag has been given) to fill out the field width.
- An optional precision, in the form of a period . followed by an optional digit string. If the digit string is omitted, the precision is taken as zero. This gives the minimum number of digits to appear for d, i, o, u, x, and X conversions, or the maximum number of characters to be printed from a string for `s` conversions.
- An optional `l` or `h` length modifier, that specifies that the argument for the d, i, o, u, x, or X conversion is a `"long int"` rather than `int`. The `h` is ignored, as `"short int"` is equivalent to `int`.
- A character that specifies the type of conversion to be applied.

The conversion specifiers and their meanings are:

- `diouxX` The int (or appropriate variant) argument is converted to signed decimal (d and i), unsigned octal (o), unsigned decimal (u), or unsigned hexadecimal (x and X) notation. The letters "abcdef" are used for x conversions; the letters "ABCDEF" are used for X conversions. The precision, if any, gives the minimum number of digits that must appear; if the converted value requires fewer digits, it is padded on the left with zeros.
- `p` The `void*` argument is taken as an unsigned integer, and converted similarly as a `%#x` command would do.

- `c` The `int` argument is converted to an "`unsigned char`", and the resulting character is written.
- `s` The `char*` argument is expected to be a pointer to an array of character type (pointer to a string). Characters from the array are written up to (but not including) a terminating NUL character; if a precision is specified, no more than the number specified are written. If a precision is given, no null character need be present; if the precision is not specified, or is greater than the size of the array, the array must contain a terminating NUL character.
- `%` A `%` is written. No argument is converted. The complete conversion specification is "%%".
- `eE` The double argument is rounded and converted in the format "`[-]d.ddde±dd`" where there is one digit before the decimal-point character and the number of digits after it is equal to the precision; if the precision is missing, it is taken as 6; if the precision is zero, no decimal-point character appears. An *E* conversion uses the letter `'E'` (rather than `'e'`) to introduce the exponent. The exponent always contains two digits; if the value is zero, the exponent is 00.
- `fF` The double argument is rounded and converted to decimal notation in the format "`[-]ddd.ddd`", where the number of digits after the decimal-point character is equal to the precision specification. If the precision is missing, it is taken as 6; if the precision is explicitly zero, no decimal-point character appears. If a decimal point appears, at least one digit appears before it.
- `gG` The double argument is converted in style `f` or `e` (or `F` or `E` for `G` conversions). The precision specifies the number of significant digits. If the precision is missing, 6 digits are given; if the precision is zero, it is treated as 1. Style `e` is used if the exponent from its conversion is less than -4 or greater than or equal to the precision. Trailing zeros are removed from the fractional part of the result; a decimal point appears only if it is followed by at least one digit.
- `S` Similar to the `s` format, except the pointer is expected to point to a program-memory (ROM) string instead of a RAM string.

In no case does a non-existent or small field width cause truncation of a numeric field; if the result of a conversion is wider than the field width, the field is expanded to contain the conversion result.

Since the full implementation of all the mentioned features becomes fairly large, three different flavours of vfprintf() can be selected using linker options:

- The default vfprintf() implements all the mentioned functionality except floating point conversions.
- A minimized version of vfprintf() that only implements the very basic integer and string conversion facilities, but only the # additional option can be specified using conversion flags (these flags are parsed correctly from the format specification, but then are simply ignored).
- A version with floating point-support, but with the following twists:
  - The argument will be converted to IEEE single for output.
  - When `long double` is a 64-bit type and `double` is a 32-bit type, then the former will only be printed as a single '?'. Rationale is to avoid 64-bit floating point arithmetic in printf when the used selected `-mdouble=32`. In order to print IEEE double, it can be converted to IEEE single by hand.

The respective version can be requested using the following link options:

Classic approach

The minimal version can be requested with the following options:

```
-Wl,-u,vfprintf -lprintf_min
```

If the full functionality including the floating point conversions is required, the following options should be used:

```
-Wl,-u,vfprintf -lprintf_flt
```

This approach will always link the selected printf code, even when the application doesn't use printf.

Since AVR-LibC v2.3

The minimal version can be requested with the following options:

```
-Wl,--defsym,vfprintf=vfprintf_min
```

•The full functionality including the floating point conversions can be requested with:

```
-Wl,--defsym,vfprintf=vfprintf_flt
```

The difference to the "classic" approach is that when no printf is used in the application and `-Wl,--gc-sections` is added to the linker options, then the printf code will not be pulled in. See `issue #654` for an example.

Limitations:

- The specified width and precision can be at most 255.

Notes:

- For floating-point conversions, if you link the default or minimized version of vfprintf(), the symbol `?` will be output. The same applies for the float version with `-mdouble=32` and when an IEEE double arguments is passed. For default version the width field and the "pad to left" (symbol *minus*) option will work in this case.
- The `hh` length modifier is ignored (`char` argument is promouted to `int`). More exactly, this realization does not check the number of `h` symbols.
- But the `ll` length modifier will to abort the output, as this realization does not operate `long long` arguments.
- The variable width or precision field (an asterisk `*` symbol) is not realized and will to abort the output.

20.9.4.33 vfprintf_P()

```
int vfprintf_P (
            FILE * __stream,
            const char * __fmt,
            va_list __ap )
```

Variant of `vfprintf()` that uses a `fmt` string that resides in program memory. See also vfprintf_F.

20.9.4.34 vfscanf()

```
int vfscanf (
            FILE * stream,
            const char * fmt,
            va_list ap )
```

Formatted input. This function is the heart of the **scanf** family of functions.

Characters are read from *stream* and processed in a way described by *fmt*. Conversion results will be assigned to the parameters passed via *ap*.

The format string *fmt* is scanned for conversion specifications. Anything that doesn't comprise a conversion specification is taken as text that is matched literally against the input. White space in the format string will match any white space in the data (including none), all other characters match only itself. Processing is aborted as soon as the data and format string no longer match, or there is an error or end-of-file condition on *stream*.

Most conversions skip leading white space before starting the actual conversion.

Conversions are introduced with the character **%**. Possible options can follow the **%**:

- a * indicating that the conversion should be performed but the conversion result is to be discarded; no parameters will be processed from ap,
- the character h indicating that the argument is a pointer to short int (rather than int),
- the 2 characters hh indicating that the argument is a pointer to char (rather than int).
- the character l indicating that the argument is a pointer to long int (rather than int, for integer type conversions), or a pointer to float (for floating point conversions),

In addition, a maximal field width may be specified as a nonzero positive decimal integer, which will restrict the conversion to at most this many characters from the input stream. This field width is limited to at most 255 characters which is also the default value (except for the c conversion that defaults to 1).

The following conversion flags are supported:

- % Matches a literal % character. This is not a conversion.
- d Matches an optionally signed decimal integer; the next pointer must be a pointer to int.
- i Matches an optionally signed integer; the next pointer must be a pointer to int. The integer is read in base 16 if it begins with **0x** or **0X**, in base 8 if it begins with **0**, and in base 10 otherwise. Only characters that correspond to the base are used.
- o Matches an octal integer; the next pointer must be a pointer to unsigned int.
- u Matches an optionally signed decimal integer; the next pointer must be a pointer to unsigned int.
- x Matches an optionally signed hexadecimal integer; the next pointer must be a pointer to unsigned int.
- f Matches an optionally signed floating-point number; the next pointer must be a pointer to float.
- e, g, F, E, G Equivalent to f.
- s Matches a sequence of non-white-space characters; the next pointer must be a pointer to char, and the array must be large enough to accept all the sequence and the terminating NUL character. The input string stops at white space or at the maximum field width, whichever occurs first.
- c Matches a sequence of width count characters (default 1); the next pointer must be a pointer to char, and there must be enough room for all the characters (no terminating NUL is added). The usual skip of leading white space is suppressed. To skip white space first, use an explicit space in the format.
- [Matches a nonempty sequence of characters from the specified set of accepted characters; the next pointer must be a pointer to char, and there must be enough room for all the characters in the string, plus a terminating NUL character. The usual skip of leading white space is suppressed. The string is to be made up of characters in (or not in) a particular set; the set is defined by the characters between the open bracket [character and a close bracket] character. The set excludes those characters if the first character after the open bracket is a circumflex ^. To include a close bracket in the set, make it the first character after the open bracket or the circumflex; any other position will end the set. The hyphen character - is also special; when placed between two other characters, it adds all intervening characters to the set. To include a hyphen, make it the last character before the final close bracket. For instance, [^]0-9-] means the set of *everything except close bracket, zero through nine, and hyphen*. The string ends with the appearance of a character not in the (or, with a circumflex, in) set or when the field width runs out. Note that usage of this conversion enlarges the stack expense.
- p Matches a pointer value (as printed by p in printf()); the next pointer must be a pointer to void.
- n Nothing is expected; instead, the number of characters consumed thus far from the input is stored through the next pointer, which must be a pointer to int. This is not a conversion, although it can be suppressed with the * flag.

These functions return the number of input items assigned, which can be fewer than provided for, or even zero, in the event of a matching failure. Zero indicates that, while there was input available, no conversions were assigned; typically this is due to an invalid input character, such as an alphabetic character for a `d` conversion. The value `EOF` is returned if an input failure occurs before any conversion such as an end-of-file occurs. If an error or end-of-file occurs after conversion has begun, the number of conversions which were successfully completed is returned.

By default, all the conversions described above are available except the floating-point conversions and the width is limited to 255 characters. The floating-point conversion will be available in the extended version. Also in this case the width is not limited (exactly, it is limited to 65535 characters).

A third version is available for environments that are tight on space. In addition to the restrictions of the standard one, this version implements no `%[` specification.

To link a program against the respective version, use the following compiler flags in the link stage:

- **Classic approach**
  - In order to use the extended version, link with:
    ```
    -Wl,-u,vfscanf -lscanf_flt
    ```
  - In order to use the minimal version, link with:
    ```
    -Wl,-u,vfscanf -lscanf_min
    ```

  This approach will always link the requested scanf implementation, even when the application doesn't use scanf.
- **Since AVR-LibC v2.3**
  - In order to use the extended version, link with:
    ```
    -Wl,--defsym,vfscanf=vfscanf_flt
    ```
  - In order to use the minimal version, link with:
    ```
    -Wl,--defsym,vfscanf=vfscanf_min
    ```

  This approach will not link the requested scanf implementation provided the application doesn't use scanf, and `-Wl,--gc-sections` is added to the link options.

20.9.4.35 vfscanf_P()

```
int vfscanf_P (
            FILE * __stream,
            const char * __fmt,
            va_list __ap )
```

Variant of vfscanf() using a `fmt` string in program memory. See also vfscanf_F.

20.9.4.36 vprintf()

```
int vprintf (
            const char * __fmt,
            va_list __ap )
```

The function `vprintf` performs formatted output to stream `stdout`, taking a variable argument list as in vfprintf().

See vfprintf() for details.

20.9.4.37 vscanf()

```
int vscanf (
            const char * __fmt,
            va_list __ap )
```

The function `vscanf` performs formatted input from stream `stdin`, taking a variable argument list as in vfscanf().

See vfscanf() for details.

20.9.4.38 vsnprintf()

```
int vsnprintf (
            char * __s,
            size_t __n,
            const char * __fmt,
            va_list __ap )
```

Like `vsprintf()`, but instead of assuming `s` to be of infinite size, no more than `n` characters (including the trailing NUL character) will be converted to `s`.

Returns the number of characters that would have been written to `s` if there were enough space.

20.9.4.39 vsnprintf_P()

```
int vsnprintf_P (
            char * __s,
            size_t __n,
            const char * __fmt,
            va_list __ap )
```

Variant of `vsnprintf()` that uses a `fmt` string that resides in program memory. See also vsnprintf_F.

20.9.4.40 vsprintf()

```
int vsprintf (
            char * __s,
            const char * __fmt,
            va_list __ap )
```

Like `sprintf()` but takes a variable argument list for the arguments.

20.9.4.41 vsprintf_P()

```
int vsprintf_P (
            char * __s,
            const char * __fmt,
            va_list __ap )
```

Variant of `vsprintf()` that uses a `fmt` string that resides in program memory. See also vsprintf_F.

20.10 <stdlib.h>: General utilities

Data Structures

- struct div_t
- struct ldiv_t

Macros

- #define EXIT_SUCCESS 0
- #define EXIT_FAILURE 1
- #define RAND_MAX 0x7FFF

Typedefs

- typedef int(* __compar_fn_t) (const void *, const void *)

Functions

- void abort (void)
- int abs (int __i)
- long labs (long __i)
- long long llabs (long long __i)
- void * bsearch (const void *__key, const void *__base, size_t __nmemb, size_t __size, __compar_fn_t __compar)
- div_t div (int __num, int __denom) __asm__("__divmodhi4")
- ldiv_t ldiv (long __num, long __denom) __asm__("__divmodsi4")
- void qsort (void *__base, size_t __nmemb, size_t __size, __compar_fn_t __compar)
- long strtol (const char *__nptr, char **__endptr, int __base)
- long long strtoll (const char *__nptr, char **__endptr, int __base)
- unsigned long strtoul (const char *__nptr, char **__endptr, int __base)
- unsigned long long strtoull (const char *__nptr, char **__endptr, int __base)
- long atol (const char *__s)
- int atoi (const char *__s)
- void exit (int __status)
- void * malloc (size_t __size)
- void free (void *__ptr)
- void * calloc (size_t __nele, size_t __size)
- void * realloc (void *__ptr, size_t __size)
- float strtof (const char *__nptr, char **__endptr)
- double strtod (const char *__nptr, char **__endptr)
- long double strtold (const char *__nptr, char **__endptr)
- int atexit (void(*func)(void))
- float atoff (const char *__nptr)
- double atof (const char *__nptr)
- long double atofl (const char *__nptr)
- int rand (void)
- void srand (unsigned int __seed)
- int rand_r (unsigned long *__ctx)

Variables

- size_t __malloc_margin
- char * __malloc_heap_start
- char * __malloc_heap_end

Non-standard (i.e. non-ISO C) functions.

- char * ltoa (long val, char *s, int radix)
- char * utoa (unsigned int val, char *s, int radix)
- char * ultoa (unsigned long val, char *s, int radix)
- char * ulltoa (unsigned long long val, char *s, int radix)
- char * ulltoa_base10 (unsigned long long val, char *s)
- char * lltoa (long long val, char *s, int radix)
- long random (void)
- void srandom (unsigned long __seed)
- long random_r (unsigned long *__ctx)
- unsigned char sqrtu16_floor (unsigned int radic)
- unsigned int sqrtu32_floor (unsigned long radic)
- unsigned long sqrtu64_floor (unsigned long long radic)
- char * itoa (int val, char *s, int radix)
- #define RANDOM_MAX 0x7FFFFFFF

Conversion functions for double arguments.

- char * ftostre (float __val, char *__s, unsigned char __prec, unsigned char __flags)
- char * dtostre (double __val, char *__s, unsigned char __prec, unsigned char __flags)
- char * ldtostre (long double __val, char *__s, unsigned char __prec, unsigned char __flags)
- char * ftostrf (float __val, signed char __width, unsigned char __prec, char *__s)
- char * dtostrf (double __val, signed char __width, unsigned char __prec, char *__s)
- char * ldtostrf (long double __val, signed char __width, unsigned char __prec, char *__s)
- #define DTOSTR_ALWAYS_SIGN 0x01 /* put '+' or ' ' for positives */
- #define DTOSTR_PLUS_SIGN 0x02 /* put '+' rather than ' ' */
- #define DTOSTR_UPPERCASE 0x04 /* put 'E' rather 'e' */

20.10.1 Detailed Description

```
#include <stdlib.h>
```

This file declares some basic C macros and functions as defined by the ISO standard, plus some AVR-specific extensions.

For some functions, benchmarks are available.

20.10.2 Macro Definition Documentation

20.10.2.1 DTOSTR_ALWAYS_SIGN

```
#define DTOSTR_ALWAYS_SIGN 0x01 /* put '+' or ' ' for positives */
```

Bit value that can be passed in `flags` to ftostre(), dtostre() and ldtostre().

20.10.2.2 DTOSTR_PLUS_SIGN

```
#define DTOSTR_PLUS_SIGN 0x02 /* put '+' rather than ' ' */
```

Bit value that can be passed in `flags` to ftostre(), dtostre() and ldtostre().

20.10.2.3 DTOSTR_UPPERCASE

```
#define DTOSTR_UPPERCASE 0x04 /* put 'E' rather 'e' */
```

Bit value that can be passed in `flags` to ftostre(), dtostre() and ldtostre().

20.10.2.4 EXIT_FAILURE

```
#define EXIT_FAILURE 1
```

Unsuccessful termination for exit(); evaluates to a non-zero value.

20.10.2.5 EXIT_SUCCESS

```
#define EXIT_SUCCESS 0
```

Successful termination for exit(); evaluates to 0.

20.10.2.6 RAND_MAX

```
#define RAND_MAX 0x7FFF
```

Highest number that can be generated by rand().

20.10.2.7 RANDOM_MAX

```
#define RANDOM_MAX 0x7FFFFFFF
```

Highest number that can be generated by random().

20.10.3 Typedef Documentation

20.10.3.1 __compar_fn_t

```
typedef int(* __compar_fn_t) (const void *, const void *)
```

Comparision function type for qsort() and bsearch(), just for convenience.

20.10.4 Function Documentation

20.10.4.1 abort()

```
void abort (
            void  )
```

The abort() function causes abnormal program termination to occur. This realization disables interrupts and execution is effectively halted by entering an infinite loop. Static destructors and atexit() registered functions are not executed.

20.10.4.2 abs()

```
int abs (
            int __i )  [inline]
```

The abs() function computes the absolute value of the integer `i`.

Note

The abs() and labs() functions are builtins of gcc.

20.10.4.3 atexit()

```
int atexit (
            void(*)(void) func )
```

The atexit() function registers function *func* to be run as part of the `exit()` function during .fini8. atexit() calls malloc().

20.10.4.4 atof()

```
double atof (
            const char * nptr )
```

The atof() function converts the initial portion of the string pointed to by *nptr* to `double` representation.

It is equivalent to calling

```
strtod(nptr, (char**) 0);
```

20.10.4.5 atoff()

```
float atoff (
            const char * nptr )
```

The atoff() function converts the initial portion of the string pointed to by *nptr* to `float` representation.

It is equivalent to calling

```
strtof(nptr, (char**) 0);
```

20.10.4.6 atofl()

```
long double atofl (
            const char * nptr )
```

The atofl() function converts the initial portion of the string pointed to by *nptr* to `long double` representation.

It is equivalent to calling

```
strtold(nptr, (char**) 0);
```

20.10.4.7 atoi()

```
int atoi (
            const char * __s )
```

The atoi() function converts the initial portion of the string pointed to by `s` to integer representation. In contrast to

```
(int)strtol(s, (char **)NULL, 10);
```

this function does not detect overflow (`errno` is not changed and the result value is not predictable), uses smaller memory (flash and stack) and works more quickly.

20.10.4.8 atol()

```
long atol (
            const char * __s )
```

The atol() function converts the initial portion of the string pointed to by `s` to long integer representation. In contrast to

```
strtol(s, (char **)NULL, 10);
```

this function does not detect overflow (`errno` is not changed and the result value is not predictable), uses smaller memory (flash and stack) and works more quickly.

20.10.4.9 bsearch()

```
void * bsearch (
            const void * __key,
            const void * __base,
            size_t __nmemb,
            size_t __size,
            __compar_fn_t __compar )
```

The bsearch() function searches an array of `nmemb` objects, the initial member of which is pointed to by `base`, for a member that matches the object pointed to by `key`. The size of each member of the array is specified by `size`.

The contents of the array should be in ascending sorted order according to the comparison function referenced by `compar`. The `compar` routine is expected to have two arguments which point to the key object and to an array member, in that order, and should return an integer less than, equal to, or greater than zero if the key object is found, respectively, to be less than, to match, or be greater than the array member.

The bsearch() function returns a pointer to a matching member of the array, or a null pointer if no match is found. If two members compare as equal, which member is matched is unspecified.

20.10.4.10 calloc()

```
void * calloc (
            size_t __nele,
            size_t __size )
```

Allocate `nele` elements of `size` each. Identical to calling malloc() using `nele * size` as argument (provided the product doesn't overflow), except the allocated memory will be cleared to zero.

20.10.4.11 div()

```
div_t div (
            int __num,
            int __denom )
```

The div() function computes the value `num/denom` and returns the quotient and remainder in a structure named `div_t` that contains two int members named `quot` and `rem`.

20.10.4.12 dtostre()

```
char * dtostre (
            double __val,
            char * __s,
            unsigned char __prec,
            unsigned char __flags )
```

The dtostre() function is similar to the ftostre() function, except that it converts a `double` value instead of a `float` value.

dtostre() is currently only supported when `double` is a 32-bit type.

20.10.4.13 dtostrf()

```
char * dtostrf (
            double __val,
            signed char __width,
            unsigned char __prec,
            char * __s )
```

The dtostrf() function is similar to the ftostrf() function, except that converts a `double` value instead of a `float` value.

ldtostre() is currently only supported when `double` is a 32-bit type.

20.10.4.14 exit()

```
void exit (
            int __status )
```

The exit() function terminates the application. Since there is no environment to return to, `status` is ignored, and code execution will eventually reach an infinite loop, thereby effectively halting all code processing. Before entering the infinite loop, interrupts are globally disabled.

Global destructors will be called before halting execution, see the .fini sections.

20.10.4.15 free()

```
void free (
            void * __ptr )
```

The free() function makes the memory referenced by `ptr` available for future allocations. The memory must have been allocated by a call to malloc(), realloc(), calloc() or other functions like strdup() or fdevopen() that allocate dynamic memory on the heap. If `ptr` is NULL, no action occurs.

20.10.4.16 ftostre()

```
char * ftostre (
            float __val,
            char * __s,
            unsigned char __prec,
            unsigned char __flags )
```

The ftostre() function converts the `float` value passed in `val` into an ASCII representation that will be stored under `s`. The caller is responsible for providing sufficient storage in `s`.

Conversion is done in the format `"[-]d.ddde±dd"` where there is one digit before the decimal-point character and the number of digits after it is equal to the precision `prec`; if the precision is zero, no decimal-point character appears. If `flags` has the DTOSTR_UPPERCASE bit set, the letter `'E'` (rather than `'e'`) will be used to introduce the exponent. The exponent always contains two digits; if the value is zero, the exponent is `"00"`.

If `flags` has the DTOSTR_ALWAYS_SIGN bit set, a space character will be placed into the leading position for positive numbers.

If `flags` has the DTOSTR_PLUS_SIGN bit set, a plus sign will be used instead of a space character in this case.

The ftostre() function returns the pointer to the converted string `s`.

20.10.4.17 ftostrf()

```
char * ftostrf (
            float __val,
            signed char __width,
            unsigned char __prec,
            char * __s )
```

The ftostrf() function converts the `float` value passed in `val` into an ASCII representation that will be stored in `s`. The caller is responsible for providing sufficient storage in `s`.

Conversion is done in the format `"[-]d.ddd"`. The minimum field width of the output string (including the possible `'.'` and the possible sign for negative values) is given in `width`, and `prec` determines the number of digits after the decimal sign. `width` is signed value, negative for left adjustment.

The ftostrf() function returns the pointer to the converted string `s`.

20.10.4.18 itoa()

```
char * itoa (
            int val,
            char * s,
            int radix )
```

Convert an integer to a string.

The function itoa() converts the integer value from `val` into an ASCII representation that will be stored under `s`. The caller is responsible for providing sufficient storage in `s`.

Note

The minimal size of the buffer `s` depends on the choice of radix. For example, if the radix is 2 (binary), you need to supply a buffer with a minimal length of 8 * sizeof (int) + 1 characters, i.e. one character for each bit plus one for the string terminator. Using a larger radix will require a smaller minimal buffer size.

Warning

If the buffer is too small, you risk a buffer overflow.

Conversion is done using the `radix` as base, which may be a number between 2 (binary conversion) and up to 36. If `radix` is greater than 10, the next digit after `'9'` will be the letter `'a'`.

If radix is 10 and val is negative, a minus sign will be prepended.

The itoa() function returns the pointer passed as `s`.

Note

Decimal conversions can be sped up by using the ktoa() function from <stdfix.h> that converts fixed-point values to decimal ASCII, like in `ktoa((accum) val, s, FXTOA_TRUNC)` that converts `val` to a decimal ASCII representation with zero fractional digits. For example, converting 1000 using itoa() takes around 700 cycles whereas ktoa() does the job in less than 300 cycles.

20.10.4.19 labs()

```
long labs (
            long __i )  [inline]
```

The labs() function computes the absolute value of the long integer `i`.

Note

The abs() and labs() functions are builtins of gcc.

20.10.4.20 ldiv()

```
ldiv_t ldiv (
            long __num,
            long __denom )
```

The ldiv() function computes the value num/denom and returns the quotient and remainder in a structure named ldiv_t that contains two long integer members named quot and rem.

20.10.4.21 ldtostre()

```
char * ldtostre (
            long double __val,
            char * __s,
            unsigned char __prec,
            unsigned char __flags )
```

The ldtostre() function is similar to the ftostre() function, except that it converts a long double value instead of a float value.

ldtostre() is currently only supported when long double is a 32-bit type.

20.10.4.22 ldtostrf()

```
char * ldtostrf (
            long double __val,
            signed char __width,
            unsigned char __prec,
            char * __s )
```

The ldtostrf() function is similar to the ftostrf() function, except that converts a long double value instead of a float value.

ldtostre() is currently only supported when long double is a 32-bit type.

20.10.4.23 llabs()

```
long long llabs (
            long long __i )  [inline]
```

The llabs() function computes the absolute value of the 64-bit integer i.

Since

AVR-LibC v2.3

20.10.4.24 lltoa()

```
char * lltoa (
            long long val,
            char * s,
            int radix )
```

Convert a signed 64-bit integer to a string.

The function lltoa() writes the ASCII representation of the signed 64-bit integer val to a string starting at s. Except for decimal conversions with a negative val, the effect is the same like with ulltoa().

Parameters

val	A signed 64-bit integral value for which the ASCII representation is computed.
s	The location to which the string representation should be stored. The caller is responsible for providing sufficient storage in `s`. The minimal size of the buffer `s` depends on the choice of the radix. For example, if the radix is 10 (decimal), the function will write at most 21 characters (including the terminating '\0').
radix	The Conversion is done using the `radix` as base, which may be a number between 2 (binary conversion) and up to 36. If `radix` is greater than 10, the next digit after `'9'` will be the letter `'a'`.

Returns

The lltoa() function returns the pointer passed as `s`.

Since

AVR-LibC v2.3

20.10.4.25 ltoa()

```
char * ltoa (
            long val,
            char * s,
            int radix )
```

Convert a long integer to a string.

The function ltoa() converts the long integer value from `val` into an ASCII representation that will be stored under `s`. The caller is responsible for providing sufficient storage in `s`.

Note

The minimal size of the buffer `s` depends on the choice of radix. For example, if the radix is 2 (binary), you need to supply a buffer with a minimal length of 8 * sizeof (long int) + 1 characters, i.e. one character for each bit plus one for the string terminator. Using a larger radix will require a smaller minimal buffer size.

Warning

If the buffer is too small, you risk a buffer overflow.

Conversion is done using the `radix` as base, which may be a number between 2 (binary conversion) and up to 36. If `radix` is greater than 10, the next digit after `'9'` will be the letter `'a'`.

If radix is 10 and val is negative, a minus sign will be prepended.

The ltoa() function returns the pointer passed as `s`.

20.10.4.26 malloc()

```
void * malloc (
            size_t size )
```

The malloc() function allocates `size` bytes of memory. If malloc() fails, a NULL pointer is returned.

Note that malloc() does *not* initialize the returned memory to zero bytes. For that, see calloc().

See the chapter about malloc() usage for implementation details.

20.10.4.27 qsort()

```
void qsort (
            void * __base,
            size_t __nmemb,
            size_t __size,
            __compar_fn_t __compar )
```

The qsort() function is a modified partition-exchange sort, or quicksort.

The qsort() function sorts an array of `nmemb` objects, the initial member of which is pointed to by `base`. The size of each object is specified by `size`. The contents of the array base are sorted in ascending order according to a comparison function pointed to by `compar`, which requires two arguments pointing to the objects being compared.

The comparison function must return an integer less than, equal to, or greater than zero if the first argument is considered to be respectively less than, equal to, or greater than the second.

20.10.4.28 rand()

```
int rand (
            void  )
```

The rand() function computes a sequence of pseudo-random integers in the range of 0 to RAND_MAX (as defined by the header file <stdlib.h>).

The srand() function sets its argument `seed` as the seed for a new sequence of pseudo-random numbers to be returned by rand(). These sequences have a period of $2^{32} - 1$ and are repeatable by calling srand() with the same seed value.

If no seed value is provided, the functions are automatically seeded with a value of 1.

rand() achieves a score of 100% in the `bbattery_SmallCrush` tests from the `TestU01` suite.

For the resource consumptions, see the libc benchmarks.

20.10.4.29 rand_r()

```
int rand_r (
            unsigned long * __ctx )
```

Variant of rand() that stores the context in the user-supplied variable located at `ctx` instead of a static library variable so the function becomes re-entrant.

20.10.4.30 random()

```
long random (
             void  )
```

The random() function computes a sequence of pseudo-random integers in the range of 0 to RANDOM_MAX (as defined by the header file <stdlib.h>).

The srandom() function sets its argument `seed` as the seed for a new sequence of pseudo-random numbers to be returned by random(). These sequences have a period of $2^{31} - 2$ and are repeatable by calling srandom() with the same seed value.

If no seed value is provided, the functions are automatically seeded with a value of 1.

For the resource consumptions, see the libc benchmarks.

20.10.4.31 random_r()

```
long random_r (
             unsigned long * __ctx )
```

Variant of random() that stores the context in the user-supplied variable located at `ctx` instead of a static library variable so the function becomes re-entrant.

20.10.4.32 realloc()

```
void * realloc (
             void * __ptr,
             size_t __size )
```

The realloc() function tries to change the size of the region allocated at `ptr` to the new `size` value. It returns a pointer to the new region. The returned pointer might be the same as the old pointer, or a pointer to a completely different region.

The contents of the returned region up to either the old or the new size value (whatever is less) will be identical to the contents of the old region, even in case a new region had to be allocated.

It is acceptable to pass `ptr` as NULL, in which case realloc() will behave identical to malloc().

If the new memory cannot be allocated, realloc() returns NULL, and the region at `ptr` will not be changed.

20.10.4.33 sqrtu16_floor()

```
unsigned char sqrtu16_floor (
             unsigned int radic )
```

Returns

Returns the square root of the 16-bit value `radic`, rounded down to the next integral value.

20.10.4.34 sqrtu32_floor()

```
unsigned int sqrtu32_floor (
             unsigned long radic )
```

Returns

Returns the square root of the 32-bit value `radic`, rounded down to the next integral value.

20.10.4.35 sqrtu64_floor()

```
unsigned long sqrtu64_floor (
             unsigned long long radic )
```

Returns

Returns the square root of the 64-bit value `radic`, rounded down to the next integral value.

20.10.4.36 srand()

```
void srand (
             unsigned int __seed )
```

Pseudo-random number generator seeding; see rand().

20.10.4.37 srandom()

```
void srandom (
             unsigned long __seed )
```

Pseudo-random number generator seeding; see random().

20.10.4.38 strtod()

```
double strtod (
             const char * __nptr,
             char ** __endptr )
```

The strtod() function is similar to strtof(), except that the conversion result is of type `double` instead of `float`.

20.10.4.39 strtof()

```
float strtof (
             const char * nptr,
             char ** endptr )
```

The strtof() function converts the initial portion of the string pointed to by *nptr* to `float` representation.

The expected form of the string is an optional plus (`'+'`) or minus sign (`'-'`) followed by a sequence of digits optionally containing a decimal-point character, optionally followed by an exponent. An exponent consists of an `'E'` or `'e'`, followed by an optional plus or minus sign, followed by a sequence of digits.

Leading white-space characters in the string are skipped.

The strtof() function returns the converted value, if any.

If *endptr* is not `NULL`, a pointer to the character after the last character used in the conversion is stored in the location referenced by *endptr*.

If no conversion is performed, zero is returned and the value of *nptr* is stored in the location referenced by *endptr*.

If the correct value would cause overflow, plus or minus `INFINITY` is returned (according to the sign of the value), and `ERANGE` is stored in `errno`. If the correct value would cause underflow, zero is returned and `ERANGE` is stored in `errno`.

20.10.4.40 strtol()

```
long strtol (
             const char * __nptr,
             char ** __endptr,
             int __base )
```

The strtol() function converts the string in `nptr` to a long value. The conversion is done according to the given base, which must be between 2 and 36 inclusive, or be the special value 0.

The string may begin with an arbitrary amount of white space (as determined by isspace()) followed by a single optional `'+'` or `'-'` sign. If `base` is zero or 16, the string may then include a `"0x"` or `"0X"` prefix, and the number will be read in base 16; otherwise, a zero base is taken as 10 (decimal) unless the next character is `'0'`, in which case it is taken as 8 (octal).

Similarly, prefixes `"0b"` and `"0B"` signify base 2, and `"0o"` and `"0O"` signify base 8.

The remainder of the string is converted to a long value in the obvious manner, stopping at the first character which is not a valid digit in the given base. (In bases above 10, the letter `'A'` in either upper or lower case represents 10, `'B'` represents 11, and so forth, with `'Z'` representing 35.)

If `endptr` is not NULL, strtol() stores the address of the first invalid character in `*endptr`. If there were no digits at all, however, strtol() stores the original value of `nptr` in `endptr`. (Thus, if `*nptr` is not `'\0'` but `**endptr` is `'\0'` on return, the entire string was valid.)

The strtol() function returns the result of the conversion, unless the value would underflow or overflow. If no conversion could be performed, 0 is returned. If an overflow or underflow occurs, `errno` is set to ERANGE and the function return value is clamped to `LONG_MIN` or `LONG_MAX`, respectively.

20.10.4.41 strtold()

```
long double strtold (
            const char * __nptr,
            char ** __endptr )
```

The strtold() function is similar to strtof(), except that the conversion result is of type `long double` instead of `float`.

20.10.4.42 strtoll()

```
long long strtoll (
            const char * __nptr,
            char ** __endptr,
            int __base )
```

The strtoll() function converts the string in `nptr` to a long long value. The conversion is done according to the given base, which must be between 2 and 36 inclusive, or be the special value 0.

The string may begin with an arbitrary amount of white space (as determined by isspace()) followed by a single optional `'+'` or `'-'` sign. If `base` is zero or 16, the string may then include a `"0x"` or `"0X"` prefix, and the number will be read in base 16; otherwise, a zero base is taken as 10 (decimal) unless the next character is `'0'`, in which case it is taken as 8 (octal).

Similarly, prefixes `"0b"` and `"0B"` signify base 2, and `"0o"` and `"0O"` signify base 8.

The remainder of the string is converted to a long long value in the obvious manner, stopping at the first character which is not a valid digit in the given base. (In bases above 10, the letter `'A'` in either upper or lower case represents 10, `'B'` represents 11, and so forth, with `'Z'` representing 35.)

If `endptr` is not NULL, strtoll() stores the address of the first invalid character in `*endptr`. If there were no digits at all, however, strtoll() stores the original value of `nptr` in `endptr`. (Thus, if `*nptr` is not `'\0'` but `**endptr` is `'\0'` on return, the entire string was valid.)

The strtoll() function returns the result of the conversion, unless the value would underflow or overflow. If no conversion could be performed, 0 is returned. If an overflow or underflow occurs, `errno` is set to ERANGE and the function return value is clamped to `LLONG_MIN` or `LLONG_MAX`, respectively.

Since

AVR-LibC v2.3

20.10.4.43 strtoul()

```
unsigned long strtoul (
            const char * __nptr,
            char ** __endptr,
            int __base )
```

The strtoul() function converts the string in nptr to an unsigned long value. The conversion is done according to the given base, which must be between 2 and 36 inclusive, or be the special value 0.

The string may begin with an arbitrary amount of white space (as determined by isspace()) followed by a single optional '+' or '-' sign. If base is zero or 16, the string may then include a "0x" or "0X" prefix, and the number will be read in base 16; otherwise, a zero base is taken as 10 (decimal) unless the next character is '0', in which case it is taken as 8 (octal).

Similarly, prefixes "0b" and "0B" signify base 2, and "0o" and "0O" signify base 8.

The remainder of the string is converted to an unsigned long value in the obvious manner, stopping at the first character which is not a valid digit in the given base. (In bases above 10, the letter 'A' in either upper or lower case represents 10, 'B' represents 11, and so forth, with 'Z' representing 35.)

If endptr is not NULL, strtoul() stores the address of the first invalid character in *endptr. If there were no digits at all, however, strtoul() stores the original value of nptr in endptr. (Thus, if *nptr is not '\0' but **endptr is '\0' on return, the entire string was valid.)

The strtoul() function returns either the result of the conversion or, if there was a leading minus sign, the negation of the result of the conversion, unless the original (non-negated) value would overflow; in the latter case, strtoul() returns ULONG_MAX, and errno is set to ERANGE. If no conversion could be performed, 0 is returned.

20.10.4.44 strtoull()

```
unsigned long long strtoull (
            const char * __nptr,
            char ** __endptr,
            int __base )
```

The strtoull() function converts the string in nptr to an unsigned long long value. The conversion is done according to the given base, which must be between 2 and 36 inclusive, or be the special value 0.

The string may begin with an arbitrary amount of white space (as determined by isspace()) followed by a single optional '+' or '-' sign. If base is zero or 16, the string may then include a "0x" or "0X" prefix, and the number will be read in base 16; otherwise, a zero base is taken as 10 (decimal) unless the next character is '0', in which case it is taken as 8 (octal).

Similarly, prefixes "0b" and "0B" signify base 2, and "0o" and "0O" signify base 8.

The remainder of the string is converted to an unsigned long long value in the obvious manner, stopping at the first character which is not a valid digit in the given base. (In bases above 10, the letter 'A' in either upper or lower case represents 10, 'B' represents 11, and so forth, with 'Z' representing 35.)

If endptr is not NULL, strtoull() stores the address of the first invalid character in *endptr. If there were no digits at all, however, strtoull() stores the original value of nptr in endptr. (Thus, if *nptr is not '\0' but **endptr is '\0' on return, the entire string was valid.)

The strtoull() function returns either the result of the conversion or, if there was a leading minus sign, the negation of the result of the conversion, unless the original (non-negated) value would overflow; in the latter case, strtoull() returns ULLONG_MAX, and errno is set to ERANGE. If no conversion could be performed, 0 is returned.

Since

AVR-LibC v2.3

20.10.4.45 ulltoa()

```
char * ulltoa (
            unsigned long long val,
            char * s,
            int radix )
```

Convert an unsigned 64-bit integer to a string.

The function ulltoa() writes the ASCII representation of the unsigned 64-bit integer `val` to a string starting at `s`.

A very rough estimation of the execution time is

$$Cycles \approx 950 + 23 \cdot N + 8.3 \cdot N^2 \cdot \log(radix) \pm 400$$

where *N* denotes the number of digits in the result, and *log* stands for the Natural Logarithm. This means a decimal conversion can take up to 9000 cycles, a hexadecimal conversions can take up to 7800 cycles, and a binary conversion can take more than 27000 cycles.

Parameters

val	An unsigned 64-bit integral value for which the ASCII representation is computed.
s	The location to which the string representation should be stored. The caller is responsible for providing sufficient storage in `s`. The minimal size of the buffer `s` depends on the choice of the radix. For example, if the radix is 10 (decimal), the function will write at most 21 characters (including the terminating '\0').
radix	The Conversion is done using the `radix` as base, which may be a number between 2 (binary conversion) and up to 36. If `radix` is greater than 10, the next digit after `'9'` will be the letter `'a'`.

Returns

The ulltoa() function returns the pointer passed as `s`.

Since

AVR-LibC v2.3

20.10.4.46 ulltoa_base10()

```
char * ulltoa_base10 (
            unsigned long long val,
            char * s )
```

Convert an unsigned 64-bit integer to a decimal string.

The function ulltoa_base10() writes the decimal ASCII representation of the unsigned 64-bit integer `val` to a string starting at `s`. The effect is the same like for `ulltoa(val, s, 10)`.

This function can be used for decimal ASCII conversions when ulltoa() is not fast enough. It consumes no more than 3300 cycles (no more than 2800 cycles with `MUL`), where ulltoa() may consume up to 9000 cycles for a decimal conversion.

Parameters

val	An unsigned 64-bit integral value for which the decimal ASCII representation is computed.
s	The location to which the string representation should be stored. The caller is responsible for providing sufficient storage in `s`. The function will write at most 21 characters (including the terminating '\0').

Returns

The ulltoa_base10() function returns the pointer passed as `s`.

Since

AVR-LibC v2.3

20.10.4.47 ultoa()

```
char * ultoa (
            unsigned long val,
            char * s,
            int radix )
```

Convert an unsigned long integer to a string.

The function ultoa() converts the unsigned long integer value from `val` into an ASCII representation that will be stored under `s`. The caller is responsible for providing sufficient storage in `s`.

Note

The minimal size of the buffer `s` depends on the choice of radix. For example, if the radix is 2 (binary), you need to supply a buffer with a minimal length of 8 * sizeof (unsigned long int) + 1 characters, i.e. one character for each bit plus one for the string terminator. Using a larger radix will require a smaller minimal buffer size.

Warning

If the buffer is too small, you risk a buffer overflow.

Conversion is done using the `radix` as base, which may be a number between 2 (binary conversion) and up to 36. If `radix` is greater than 10, the next digit after `'9'` will be the letter `'a'`.

The ultoa() function returns the pointer passed as `s`.

20.10.4.48 utoa()

```
char * utoa (
            unsigned int val,
            char * s,
            int radix )
```

Convert an unsigned integer to a string.

The function utoa() converts the unsigned integer value from `val` into an ASCII representation that will be stored under `s`. The caller is responsible for providing sufficient storage in `s`.

Note

The minimal size of the buffer `s` depends on the choice of radix. For example, if the radix is 2 (binary), you need to supply a buffer with a minimal length of 8 * sizeof (unsigned int) + 1 characters, i.e. one character for each bit plus one for the string terminator. Using a larger radix will require a smaller minimal buffer size.

Warning

If the buffer is too small, you risk a buffer overflow.

Conversion is done using the `radix` as base, which may be a number between 2 (binary conversion) and up to 36. If `radix` is greater than 10, the next digit after `'9'` will be the letter `'a'`.

The utoa() function returns the pointer passed as `s`.

Note

Decimal conversions can be sped up by using the uktoa() function from <stdfix.h> that converts fixed-point values to decimal ASCII, like in `uktoa((unsigned accum) val, s, FXTOA_TRUNC)` that converts `val` to a decimal ASCII representation with zero fractional digits. For example, converting 1000 using utoa() takes around 700 cycles whereas uktoa() does the job in less than 300 cycles.

20.10.5 Variable Documentation

20.10.5.1 __malloc_heap_end

```
char* __malloc_heap_end  [extern]
```

tunable for malloc(). Default value is __heap_end, which is weakly defined to 0 in the startup code.

20.10.5.2 __malloc_heap_start

```
char* __malloc_heap_start  [extern]
```

tunable for malloc(). Default value is __heap_start.

20.10.5.3 __malloc_margin

```
size_t __malloc_margin  [extern]
```

tunable for malloc(). Default value is 32 bytes.

20.11 <stdfix.h>: ISO/IEC TR 18037 Fixed-Point Arithmetic

Macros

- #define FXTOA_ALL 0x1f
- #define FXTOA_ROUND 0x00
- #define FXTOA_TRUNC 0x80
- #define FXTOA_NTZ 0x40
- #define FXTOA_DOT 0x00
- #define FXTOA_COMMA 0x20

ASCII Conversions (not in ISO/IEC TR18037)

- char * hktoa (short accum x, char *buf, unsigned char mode)
- char * hrtoa (short fract x, char *buf, unsigned char mode)
- char * ktoa (accum x, char *buf, unsigned char mode)
- char * rtoa (fract x, char *buf, unsigned char mode)
- char * uhktoa (unsigned short accum x, char *buf, unsigned char mode)
- char * uhrtoa (unsigned short fract x, char *buf, unsigned char mode)
- char * uktoa (unsigned accum x, char *buf, unsigned char mode)
- char * urtoa (unsigned fract x, char *buf, unsigned char mode)

Absolute Value

- short fract abshr (short fract val)
- fract absr (fract val)
- long fract abslr (long fract val)
- long long fract absllr (long long fract val)
- short accum abshk (short accum val)
- accum absk (accum val)
- long accum abslk (long accum val)
- long long accum absllk (long long accum val)

Bit-Conversions to Integer

- signed char bitshr (short fract val)
- unsigned char bitsuhr (unsigned short fract val)
- int bitsr (fract val)
- unsigned int bitsur (unsigned fract val)
- long bitslr (long fract val)
- unsigned long bitsulr (unsigned long fract val)
- long long bitsllr (long long fract val)
- unsigned long long bitsullr (unsigned long long fract val)
- int bitshk (short accum val)
- unsigned int bitsuhk (unsigned short accum val)
- long bitsk (accum val)
- unsigned long bitsuk (unsigned accum val)
- long long bitslk (long accum val)
- unsigned long long bitsulk (unsigned long accum val)
- long long bitsllk (long long accum val)
- unsigned long long bitsullk (unsigned long long accum val)

Bit-Conversions to Fixed-Point

- short fract hrbits (signed char val)
- unsigned short fract uhrbits (unsigned char val)
- fract rbits (int val)
- unsigned fract urbits (unsigned int val)
- long fract lrbits (long val)
- unsigned long fract ulrbits (unsigned long val)
- long long fract llrbits (long long val)
- unsigned long long fract ullrbits (unsigned long long val)
- short accum hkbits (int val)
- unsigned short accum uhkbits (unsigned int val)
- accum kbits (long val)
- unsigned accum ukbits (unsigned long val)
- long accum lkbits (long long val)
- unsigned long accum ulkbits (unsigned long long val)
- long long accum llkbits (long long val)
- unsigned long long accum ullkbits (unsigned long long val)

Count Left-Shift

- int countlshr (short fract val)
- int countlsuhr (unsigned short fract val)
- int countlsr (fract val)
- int countlsur (unsigned fract val)
- int countlslr (long fract val)
- int countlsulr (unsigned long fract val)
- int countlsllr (long long fract val)
- int countlsullr (unsigned long long fract val)
- int countlshk (short accum val)
- int countlsuhk (unsigned short accum val)
- int countlsk (accum val)
- int countlsuk (unsigned accum val)
- int countlslk (long accum val)
- int countlsulk (unsigned long accum val)
- int countlsllk (long long accum val)
- int countlsullk (unsigned long long accum val)

Division

- fract rdivi (int num, int denom)
- long fract lrdivi (long int num, long int denom)
- unsigned fract urdivi (unsigned int num, unsigned int denom)
- unsigned long fract ulrdivi (unsigned long int num, unsigned long int denom)

Rounding

- short fract roundhr (short fract val, int bit)
- unsigned short fract rounduhr (unsigned short fract val, int bit)
- fract roundr (fract val, int bit)
- unsigned fract roundur (unsigned fract val, int bit)
- long fract roundlr (long fract val, int bit)
- unsigned long fract roundulr (unsigned long fract val, int bit)
- long long fract roundllr (long long fract val, int bit)
- unsigned long long fract roundullr (unsigned long long fract val, int bit)
- short accum roundhk (short accum val, int bit)
- unsigned short accum rounduhk (unsigned short accum val, int bit)
- accum roundk (accum val, int bit)
- unsigned accum rounduk (unsigned accum val, int bit)
- long accum roundlk (long accum val, int bit)
- unsigned long accum roundulk (unsigned long accum val, int bit)
- long long accum roundllk (long long accum val, int bit)
- unsigned long long accum roundullk (unsigned long long accum val, int bit)

Square Root and Transcendental Functions

- accum acosk (accum x)
- unsigned accum acosuk (unsigned accum x)
- accum asink (accum x)
- unsigned accum asinuk (unsigned accum x)
- accum atank (accum x)
- unsigned accum atanuk (unsigned accum x)
- unsigned fract atanur (unsigned fract x)
- accum exp2k (accum x)
- unsigned accum exp2uk (unsigned accum x)
- unsigned fract exp2m1ur (unsigned fract x)
- accum log2uk (unsigned accum x)
- short accum log2uhk (unsigned short accum x)
- unsigned short fract log21puhr (unsigned short fract x)
- unsigned fract log21pur (unsigned fract x)
- accum cospi2k (accum deg)
- accum sinpi2k (accum deg)
- fract sinuhk_deg (unsigned short accum deg)
- fract cosuhk_deg (unsigned short accum deg)
- unsigned fract sinpi2ur (unsigned fract x)
- short accum sqrthk (short accum radic)
- short fract sqrthr (short fract radic)
- accum sqrtk (accum radic)
- long fract sqrtlr (long fract radic)
- fract sqrtr (fract radic)
- unsigned short accum sqrtuhk (unsigned short accum radic)
- unsigned short fract sqrtuhr (unsigned short fract radic)
- unsigned accum sqrtuk (unsigned accum radic)
- unsigned long fract sqrtulr (unsigned long fract radic)
- unsigned fract sqrtur (unsigned fract radic)

Type-Generic Functions

- type absfx (type val)
- int countlsfx (type val)
- type roundfx (type val, int bit)

Functions reading from PROGMEM

- static short fract pgm_read_hr (const short fract *addr)
- static unsigned short fract pgm_read_uhr (const unsigned short fract *addr)
- static fract pgm_read_r (const fract *addr)
- static unsigned fract pgm_read_ur (const unsigned fract *addr)
- static long fract pgm_read_lr (const long fract *addr)
- static unsigned long fract pgm_read_ulr (const unsigned long fract *addr)
- static long long fract pgm_read_llr (const long long fract *addr)
- static unsigned long long fract pgm_read_ullr (const unsigned long long fract *addr)
- static short accum pgm_read_hk (const short accum *addr)
- static unsigned short accum pgm_read_uhk (const unsigned short accum *addr)
- static accum pgm_read_k (const accum *addr)
- static unsigned accum pgm_read_uk (const unsigned accum *addr)
- static long accum pgm_read_lk (const long accum *addr)
- static unsigned long accum pgm_read_ulk (const unsigned long accum *addr)
- static long long accum pgm_read_llk (const long long accum *addr)
- static unsigned long long accum pgm_read_ullk (const unsigned long long accum *addr)

Functions reading from PROGMEM_FAR

- static short fract pgm_read_hr_far (uint_farptr_t addr)
- static unsigned short fract pgm_read_uhr_far (uint_farptr_t addr)
- static fract pgm_read_r_far (uint_farptr_t addr)
- static unsigned fract pgm_read_ur_far (uint_farptr_t addr)
- static long fract pgm_read_lr_far (uint_farptr_t addr)
- static unsigned long fract pgm_read_ulr_far (uint_farptr_t addr)
- static long long fract pgm_read_llr_far (uint_farptr_t addr)
- static unsigned long long fract pgm_read_ullr_far (uint_farptr_t addr)
- static short accum pgm_read_hk_far (uint_farptr_t addr)
- static unsigned short accum pgm_read_uhk_far (uint_farptr_t addr)
- static accum pgm_read_k_far (uint_farptr_t addr)
- static unsigned accum pgm_read_uk_far (uint_farptr_t addr)
- static long accum pgm_read_lk_far (uint_farptr_t addr)
- static unsigned long accum pgm_read_ulk_far (uint_farptr_t addr)
- static long long accum pgm_read_llk_far (uint_farptr_t addr)
- static unsigned long long accum pgm_read_ullk_far (uint_farptr_t addr)

EEPROM Read Functions

- short fract eeprom_read_hr (const short fract *__p)
- unsigned short fract eeprom_read_uhr (const unsigned short fract *__p)
- fract eeprom_read_r (const fract *__p)
- unsigned fract eeprom_read_ur (const unsigned fract *__p)
- long fract eeprom_read_lr (const long fract *__p)
- unsigned long fract eeprom_read_ulr (const unsigned long fract *__p)
- long long fract eeprom_read_llr (const long long fract *__p)
- unsigned long long fract eeprom_read_ullr (const unsigned long long fract *__p)
- short accum eeprom_read_hk (const short accum *__p)
- unsigned short accum eeprom_read_uhk (const unsigned short accum *__p)
- accum eeprom_read_k (const accum *__p)
- unsigned accum eeprom_read_uk (const unsigned accum *__p)
- long accum eeprom_read_lk (const long accum *__p)
- unsigned long accum eeprom_read_ulk (const unsigned long accum *__p)
- long long accum eeprom_read_llk (const long long accum *__p)
- unsigned long long accum eeprom_read_ullk (const unsigned long long accum *__p)

EEPROM Write Functions

- void eeprom_write_hr (short fract *__p, short fract __value)
- void eeprom_write_uhr (unsigned short fract *__p, unsigned short fract __value)
- void eeprom_write_r (fract *__p, fract __value)
- void eeprom_write_ur (unsigned fract *__p, unsigned fract __value)
- void eeprom_write_lr (long fract *__p, long fract __value)
- void eeprom_write_ulr (unsigned long fract *__p, unsigned long fract __value)
- void eeprom_write_llr (long long fract *__p, long long fract __value)
- void eeprom_write_ullr (unsigned long long fract *__p, unsigned long long fract __value)
- void eeprom_write_hk (short accum *__p, short accum __value)
- void eeprom_write_uhk (unsigned short accum *__p, unsigned short accum __value)
- void eeprom_write_k (accum *__p, accum __value)
- void eeprom_write_uk (unsigned accum *__p, unsigned accum __value)
- void eeprom_write_lk (long accum *__p, long accum __value)
- void eeprom_write_ulk (unsigned long accum *__p, unsigned long accum __value)
- void eeprom_write_llk (long long accum *__p, long long accum __value)
- void eeprom_write_ullk (unsigned long long accum *__p, unsigned long long accum __value)

EEPROM Update Functions

- void eeprom_update_hr (short fract *__p, short fract __value)
- void eeprom_update_uhr (unsigned short fract *__p, unsigned short fract __value)
- void eeprom_update_r (fract *__p, fract __value)
- void eeprom_update_ur (unsigned fract *__p, unsigned fract __value)
- void eeprom_update_lr (long fract *__p, long fract __value)
- void eeprom_update_ulr (unsigned long fract *__p, unsigned long fract __value)
- void eeprom_update_llr (long long fract *__p, long long fract __value)
- void eeprom_update_ullr (unsigned long long fract *__p, unsigned long long fract __value)
- void eeprom_update_hk (short accum *__p, short accum __value)
- void eeprom_update_uhk (unsigned short accum *__p, unsigned short accum __value)
- void eeprom_update_k (accum *__p, accum __value)
- void eeprom_update_uk (unsigned accum *__p, unsigned accum __value)
- void eeprom_update_lk (long accum *__p, long accum __value)
- void eeprom_update_ulk (unsigned long accum *__p, unsigned long accum __value)
- void eeprom_update_llk (long long accum *__p, long long accum __value)
- void eeprom_update_ullk (unsigned long long accum *__p, unsigned long long accum __value)

20.11.1 Detailed Description

```
#include <stdfix.h>
```

As an extension, GNU C supports fixed-point types as defined in the N1169 draft of ISO/IEC DTR 18037.

Since

avr-gcc v4.8

Two groups of fixed-point data types are added:

- The *fract types* and the *accum types*. The data value of a *fract* type has no integral part, hence values of a *fract* type are between -1.0 and +1.0.
- The value range of an *accum* type depends on the number of integral bits in the data type.

Table 39 Fixed-Point Type Layout

Const Suffix	Type	Size	Q-Format	Epsilon
hr	short fract	1	s.7	$7.81 \cdot 10^{-3}$
r	fract	2	s.15	$3.05 \cdot 10^{-5}$
lr	long fract	4	s.31	$4.66 \cdot 10^{-10}$
llr	long long fract	8	s.63	$1.08 \cdot 10^{-19}$
hk	short accum	2	s8.7	$7.81 \cdot 10^{-3}$
k	accum	4	s16.15	$3.05 \cdot 10^{-5}$
lk	long accum	8	s32.31	$4.66 \cdot 10^{-10}$
llk	long long accum	8	s16.47	$7.11 \cdot 10^{-15}$
uhr	unsigned short fract	1	0.8	$3.91 \cdot 10^{-3}$
ur	unsigned fract	2	0.16	$1.53 \cdot 10^{-5}$
ulr	unsigned long fract	4	0.32	$2.33 \cdot 10^{-10}$
ullr	unsigned long long fract	8	0.64	$5.42 \cdot 10^{-20}$
uhk	unsigned short accum	2	8.8	$3.91 \cdot 10^{-3}$
uk	unsigned accum	4	16.16	$1.53 \cdot 10^{-5}$
ulk	unsigned long accum	8	32.32	$2.33 \cdot 10^{-10}$
ullk	unsigned long long accum	8	16.48	$3.55 \cdot 10^{-15}$

Remarks

- Upper case constant suffixes are also supported.
- The `long long` fixed-point types are avr-gcc extensions.

See also some benchmarks.

20.11.2 Macro Definition Documentation

20.11.2.1 FXTOA_ALL

```
#define FXTOA_ALL 0x1f
```

Include all significant digits in the result of a fixed-point to decimal ASCII conversion. The result has no trailing zeros.

To be used in the *mode* parameter of such a conversion. For details and examples, see uktoa().

Since

AVR-LibC v2.3

20.11.2.2 FXTOA_COMMA

```
#define FXTOA_COMMA 0x20
```

The fixed-point to decimal ASCII conversion routines use a comma (`,`) for the decimal point.

For details and examples, see uktoa().

Since

AVR-LibC v2.3

20.11.2.3 FXTOA_DOT

```
#define FXTOA_DOT 0x00
```

The fixed-point to decimal ASCII conversion routines use a dot (`.`) for the decimal point. This is the default, i.e. FXTOA_DOT can be omitted.

For details, and examples, see uktoa().

Since

AVR-LibC v2.3

20.11.2.4 FXTOA_NTZ

```
#define FXTOA_NTZ 0x40
```

A flag to select that the result of a fixed-point to decimal ASCII conversion has no trailing zeros.

To be used in the *mode* parameter of such a conversion. For details and examples, see uktoa().

Since

AVR-LibC v2.3

20.11.2.5 FXTOA_ROUND

```
#define FXTOA_ROUND 0x00
```

A flag to select rounding to nearest in a fixed-point to decimal ASCII conversion. Rounding mode is the default, i.e. FXTOA_ROUND can be omitted.

To be used in the *mode* parameter of such a conversion. For details and examples, see uktoa().

Since

AVR-LibC v2.3

20.11.2.6 FXTOA_TRUNC

```
#define FXTOA_TRUNC 0x80
```

A flag to select truncation (rounding to zero) in a fixed-point to decimal ASCII conversion.

To be used in the *mode* parameter of such a conversion. For details and examples, see uktoa().

Since

AVR-LibC v2.3

20.11.3 Function Documentation

20.11.3.1 absfx()

```
type absfx (
            type val )
```

Computes the absolute value of fixed-point value `val`. When the result does not fit into the range of the return type, the result is saturated.

20.11.3.2 abshk()

```
short accum abshk (
            short accum val )
```

Computes the absolute value of `val`. When the result does not fit into the range of the return type, the result is saturated.

20.11.3.3 abshr()

```
short fract abshr (
            short fract val )
```

Computes the absolute value of `val`. When the result does not fit into the range of the return type, the result is saturated.

20.11.3.4 absk()

```
accum absk (
             accum val )
```

Computes the absolute value of `val`. When the result does not fit into the range of the return type, the result is saturated.

20.11.3.5 abslk()

```
long accum abslk (
             long accum val )
```

Computes the absolute value of `val`. When the result does not fit into the range of the return type, the result is saturated.

20.11.3.6 absllk()

```
long long accum absllk (
             long long accum val )
```

Computes the absolute value of `val`. When the result does not fit into the range of the return type, the result is saturated.

20.11.3.7 absllr()

```
long long fract absllr (
             long long fract val )
```

Computes the absolute value of `val`. When the result does not fit into the range of the return type, the result is saturated.

20.11.3.8 abslr()

```
long fract abslr (
             long fract val )
```

Computes the absolute value of `val`. When the result does not fit into the range of the return type, the result is saturated.

20.11.3.9 absr()

```
fract absr (
             fract val )
```

Computes the absolute value of `val`. When the result does not fit into the range of the return type, the result is saturated.

20.11.3.10 acosk()

```
accum acosk (
             accum x )
```

Compute the arcus cosine of x. The returned value is in the range $[0, \pi]$. For invalid values of x the returned value is -65536 = kbits (0x80000000).

The absolute error is bounded by $5.5 \cdot 10^{-5} \approx 2^{-14.1}$.

Since

AVR-LibC v2.3

20.11.3.11 acosuk()

```
unsigned accum acosuk (
             unsigned accum x )
```

Compute the arcus cosine of x. The returned value is in the range $[0, \pi/2]$. For invalid values of x the returned value is 32768 = ukbits (0x80000000).

The absolute error is bounded by $4.6 \cdot 10^{-5} \approx 2^{-14.4}$.

Since

AVR-LibC v2.3

20.11.3.12 asink()

```
accum asink (
             accum x )
```

Compute the arcus sine of x. The returned value is in the range $[-\pi/2, \pi/2]$. For invalid values of x the returned value is -65536 = kbits (0x80000000).

The absolute error is bounded by $5.1 \cdot 10^{-5} \approx 2^{-14}$.

Since

AVR-LibC v2.3

20.11.3.13 asinuk()

```
unsigned accum asinuk (
            unsigned accum x )
```

Compute the arcus sine of x. The returned value is in the range $[0, \pi/2]$. For invalid values of x the returned value is 32768 = ukbits (0x80000000).

The absolute error is bounded by $4.5 \cdot 10^{-5} \approx 2^{-14.4}$.

Since

AVR-LibC v2.3

20.11.3.14 atank()

```
accum atank (
            accum x )
```

Compute the arcus tangent of x. The returned value is in the range $(-\pi/2, \pi/2 \approx 1.5708)$.

Since

AVR-LibC v2.3

20.11.3.15 atanuk()

```
unsigned accum atanuk (
            unsigned accum x )
```

Compute the arcus tangent of x. The returned value is in the range $[0, \pi/2 \approx 1.5708)$.

Since

AVR-LibC v2.3

20.11.3.16 atanur()

```
unsigned fract atanur (
            unsigned fract x )
```

Compute the arcus tangent of x. The returned value is in the range $[0, \pi/4 \approx 0.7854]$.

The absolute error is bounded by $2.6 \cdot 10^{-5} \approx 2^{-15}$. The worst case execution time (WCET) is around 210 cycles when MUL is available, and around 1000 cycles when MUL is not available (measured with avr-gcc v15).

Since

AVR-LibC v2.3

20.11.3.17 bitshk()

```
int bitshk (
             short accum val )
```

Return an integer value of the same size and signedness, and with the same bit representation like `val`.

20.11.3.18 bitshr()

```
signed char bitshr (
             short fract val )
```

Return an integer value of the same size and signedness, and with the same bit representation like `val`.

20.11.3.19 bitsk()

```
long bitsk (
             accum val )
```

Return an integer value of the same size and signedness, and with the same bit representation like `val`.

20.11.3.20 bitslk()

```
long long bitslk (
             long accum val )
```

Return an integer value of the same size and signedness, and with the same bit representation like `val`.

20.11.3.21 bitsllk()

```
long long bitsllk (
             long long accum val )
```

Return an integer value of the same size and signedness, and with the same bit representation like `val`.

20.11.3.22 bitsllr()

```
long long bitsllr (
             long long fract val )
```

Return an integer value of the same size and signedness, and with the same bit representation like `val`.

20.11.3.23 bitslr()

```
long bitslr (
             long fract val )
```

Return an integer value of the same size and signedness, and with the same bit representation like `val`.

20.11.3.24 bitsr()

```
int bitsr (
             fract val )
```

Return an integer value of the same size and signedness, and with the same bit representation like `val`.

20.11.3.25 bitsuhk()

```
unsigned int bitsuhk (
             unsigned short accum val )
```

Return an integer value of the same size and signedness, and with the same bit representation like `val`.

20.11.3.26 bitsuhr()

```
unsigned char bitsuhr (
             unsigned short fract val )
```

Return an integer value of the same size and signedness, and with the same bit representation like `val`.

20.11.3.27 bitsuk()

```
unsigned long bitsuk (
             unsigned accum val )
```

Return an integer value of the same size and signedness, and with the same bit representation like `val`.

20.11.3.28 bitsulk()

```
unsigned long long bitsulk (
             unsigned long accum val )
```

Return an integer value of the same size and signedness, and with the same bit representation like `val`.

20.11.3.29 bitsullk()

```
unsigned long long bitsullk (
             unsigned long long accum val )
```

Return an integer value of the same size and signedness, and with the same bit representation like `val`.

20.11.3.30 bitsullr()

```
unsigned long long bitsullr (
             unsigned long long fract val )
```

Return an integer value of the same size and signedness, and with the same bit representation like `val`.

20.11.3.31 bitsulr()

```
unsigned long bitsulr (
            unsigned long fract val )
```

Return an integer value of the same size and signedness, and with the same bit representation like `val`.

20.11.3.32 bitsur()

```
unsigned int bitsur (
            unsigned fract val )
```

Return an integer value of the same size and signedness, and with the same bit representation like `val`.

20.11.3.33 cospi2k()

```
accum cospi2k (
            accum deg )
```

Cosine of $x \cdot \pi/2$ radians.

The absolute error is bounded by $4.6 \cdot 10^{-5} \approx 2^{-14.4}$. The worst case execution time (WCET) is around 300 cycles when MUL is available, and around 1400 cycles when MUL is not available.

Since

AVR-LibC v2.3

20.11.3.34 cosuhk_deg()

```
fract cosuhk_deg (
            unsigned short accum deg )
```

Cosine of the angle *deg* where *deg* is specified in degrees, i.e. in the range $[0°, 256°)$. The returned value is in the range (-1, +1), i.e. is never -1.

The absolute error is bounded by $6.5 \cdot 10^{-5} \approx 2^{-13.9}$. The worst case execution time (WCET) is around 90 cycles.

Since

AVR-LibC v2.3

20.11.3.35 countlsfx()

```
int countlsfx (
            type val )
```

- If `val` is non-zero, the return value is the largest integer `k` for which the expression `val << k` does not overflow.
- If `val` is zero, an integer value is returned that is at least as large as `N - 1`, where `N` is the total number of bits of the type of the argument.

20.11.3.36 countlshk()

```
int countlshk (
            short accum val )
```

- If `val` is non-zero, the return value is the largest integer `k` for which the expression `val << k` does not overflow.
- If `val` is zero, an integer value is returned that is at least as large as `N - 1`, where `N` is the total number of bits of the type of the argument.

20.11.3.37 countlshr()

```
int countlshr (
            short fract val )
```

- If `val` is non-zero, the return value is the largest integer `k` for which the expression `val << k` does not overflow.
- If `val` is zero, an integer value is returned that is at least as large as `N - 1`, where `N` is the total number of bits of the type of the argument.

20.11.3.38 countlsk()

```
int countlsk (
            accum val )
```

- If `val` is non-zero, the return value is the largest integer `k` for which the expression `val << k` does not overflow.
- If `val` is zero, an integer value is returned that is at least as large as `N - 1`, where `N` is the total number of bits of the type of the argument.

20.11.3.39 countlslk()

```
int countlslk (
            long accum val )
```

- If `val` is non-zero, the return value is the largest integer `k` for which the expression `val << k` does not overflow.
- If `val` is zero, an integer value is returned that is at least as large as `N - 1`, where `N` is the total number of bits of the type of the argument.

20.11.3.40 countlsllk()

```
int countlsllk (
            long long accum val )
```

- If `val` is non-zero, the return value is the largest integer `k` for which the expression `val << k` does not overflow.
- If `val` is zero, an integer value is returned that is at least as large as `N - 1`, where `N` is the total number of bits of the type of the argument.

20.11.3.41 countlsllr()

```
int countlsllr (
            long long fract val )
```

- If `val` is non-zero, the return value is the largest integer `k` for which the expression `val << k` does not overflow.
- If `val` is zero, an integer value is returned that is at least as large as `N - 1`, where `N` is the total number of bits of the type of the argument.

20.11.3.42 countlslr()

```
int countlslr (
            long fract val )
```

- If `val` is non-zero, the return value is the largest integer `k` for which the expression `val << k` does not overflow.
- If `val` is zero, an integer value is returned that is at least as large as `N - 1`, where `N` is the total number of bits of the type of the argument.

20.11.3.43 countlsr()

```
int countlsr (
            fract val )
```

- If `val` is non-zero, the return value is the largest integer `k` for which the expression `val << k` does not overflow.
- If `val` is zero, an integer value is returned that is at least as large as `N - 1`, where `N` is the total number of bits of the type of the argument.

20.11.3.44 countlsuhk()

```
int countlsuhk (
            unsigned short accum val )
```

- If `val` is non-zero, the return value is the largest integer `k` for which the expression `val << k` does not overflow.
- If `val` is zero, an integer value is returned that is at least as large as `N - 1`, where `N` is the total number of bits of the type of the argument.

20.11.3.45 countlsuhr()

```
int countlsuhr (
            unsigned short fract val )
```

- If `val` is non-zero, the return value is the largest integer `k` for which the expression `val << k` does not overflow.
- If `val` is zero, an integer value is returned that is at least as large as `N - 1`, where `N` is the total number of bits of the type of the argument.

20.11.3.46 countlsuk()

```
int countlsuk (
            unsigned accum val )
```

- If `val` is non-zero, the return value is the largest integer `k` for which the expression `val << k` does not overflow.
- If `val` is zero, an integer value is returned that is at least as large as `N - 1`, where `N` is the total number of bits of the type of the argument.

20.11.3.47 countlsulk()

```
int countlsulk (
            unsigned long accum val )
```

- If `val` is non-zero, the return value is the largest integer `k` for which the expression `val << k` does not overflow.
- If `val` is zero, an integer value is returned that is at least as large as `N - 1`, where `N` is the total number of bits of the type of the argument.

20.11.3.48 countlsullk()

```
int countlsullk (
             unsigned long long accum val )
```

- If `val` is non-zero, the return value is the largest integer `k` for which the expression `val << k` does not overflow.
- If `val` is zero, an integer value is returned that is at least as large as `N - 1`, where `N` is the total number of bits of the type of the argument.

20.11.3.49 countlsullr()

```
int countlsullr (
             unsigned long long fract val )
```

- If `val` is non-zero, the return value is the largest integer `k` for which the expression `val << k` does not overflow.
- If `val` is zero, an integer value is returned that is at least as large as `N - 1`, where `N` is the total number of bits of the type of the argument.

20.11.3.50 countlsulr()

```
int countlsulr (
             unsigned long fract val )
```

- If `val` is non-zero, the return value is the largest integer `k` for which the expression `val << k` does not overflow.
- If `val` is zero, an integer value is returned that is at least as large as `N - 1`, where `N` is the total number of bits of the type of the argument.

20.11.3.51 countlsur()

```
int countlsur (
             unsigned fract val )
```

- If `val` is non-zero, the return value is the largest integer `k` for which the expression `val << k` does not overflow.
- If `val` is zero, an integer value is returned that is at least as large as `N - 1`, where `N` is the total number of bits of the type of the argument.

20.11.3.52 eeprom_read_hk()

```
short accum eeprom_read_hk (
            const short accum * __p )
```

Read a `short accum` from EEPROM address *__p*.

Since

AVR-LibC v2.3

20.11.3.53 eeprom_read_hr()

```
short fract eeprom_read_hr (
            const short fract * __p )
```

Read a `short fract` from EEPROM address *__p*.

Since

AVR-LibC v2.3

20.11.3.54 eeprom_read_k()

```
accum eeprom_read_k (
            const accum * __p )
```

Read an `accum` from EEPROM address *__p*.

Since

AVR-LibC v2.3

20.11.3.55 eeprom_read_lk()

```
long accum eeprom_read_lk (
            const long accum * __p )
```

Read a `long accum` from EEPROM address *__p*.

Since

AVR-LibC v2.3

20.11.3.56 eeprom_read_llk()

```
long long accum eeprom_read_llk (
            const long long accum * __p )
```

Read a `long long accum` from EEPROM address *__p*.

Since

AVR-LibC v2.3

20.11.3.57 eeprom_read_llr()

```
long long fract eeprom_read_llr (
            const long long fract * __p )
```

Read a `long long fract` from EEPROM address *__p*.

Since

AVR-LibC v2.3

20.11.3.58 eeprom_read_lr()

```
long fract eeprom_read_lr (
            const long fract * __p )
```

Read a `long fract` from EEPROM address *__p*.

Since

AVR-LibC v2.3

20.11.3.59 eeprom_read_r()

```
fract eeprom_read_r (
            const fract * __p )
```

Read a `fract` from EEPROM address *__p*.

Since

AVR-LibC v2.3

20.11.3.60 eeprom_read_uhk()

```
unsigned short accum eeprom_read_uhk (
            const unsigned short accum * __p )
```

Read an `unsigned short accum` from EEPROM address *__p*.

Since

AVR-LibC v2.3

20.11.3.61 eeprom_read_uhr()

```
unsigned short fract eeprom_read_uhr (
            const unsigned short fract * __p )
```

Read an `unsigned short fract` from EEPROM address *__p*.

Since

AVR-LibC v2.3

20.11.3.62 eeprom_read_uk()

```
unsigned accum eeprom_read_uk (
            const unsigned accum * __p )
```

Read an `unsigned accum` from EEPROM address *__p*.

Since

AVR-LibC v2.3

20.11.3.63 eeprom_read_ulk()

```
unsigned long accum eeprom_read_ulk (
            const unsigned long accum * __p )
```

Read an `unsigned long accum` from EEPROM address *__p*.

Since

AVR-LibC v2.3

20.11.3.64 eeprom_read_ullk()

```
unsigned long long accum eeprom_read_ullk (
            const unsigned long long accum * __p )
```

Read an `unsigned long long accum` from EEPROM address *__p*.

Since

AVR-LibC v2.3

20.11.3.65 eeprom_read_ullr()

```
unsigned long long fract eeprom_read_ullr (
            const unsigned long long fract * __p )
```

Read an `unsigned long long fract` from EEPROM address *__p*.

Since

AVR-LibC v2.3

20.11.3.66 eeprom_read_ulr()

```
unsigned long fract eeprom_read_ulr (
            const unsigned long fract * __p )
```

Read an `unsigned long fract` from EEPROM address *__p*.

Since

AVR-LibC v2.3

20.11.3.67 eeprom_read_ur()

```
unsigned fract eeprom_read_ur (
            const unsigned fract * __p )
```

Read an `unsigned fract` from EEPROM address *__p*.

Since

AVR-LibC v2.3

20.11.3.68 eeprom_update_hk()

```
void eeprom_update_hk (
             short accum * __p,
             short accum __value )
```

Update a `short accum` at EEPROM address *__p*.

Since

AVR-LibC v2.3

20.11.3.69 eeprom_update_hr()

```
void eeprom_update_hr (
             short fract * __p,
             short fract __value )
```

Update a `short fract` at EEPROM address *__p*.

Since

AVR-LibC v2.3

20.11.3.70 eeprom_update_k()

```
void eeprom_update_k (
             accum * __p,
             accum __value )
```

Update an `accum` at EEPROM address *__p*.

Since

AVR-LibC v2.3

20.11.3.71 eeprom_update_lk()

```
void eeprom_update_lk (
             long accum * __p,
             long accum __value )
```

Update a `long accum` at EEPROM address *__p*.

Since

AVR-LibC v2.3

20.11.3.72 eeprom_update_llk()

```
void eeprom_update_llk (
             long long accum * __p,
             long long accum __value )
```

Update a `long long accum` at EEPROM address *__p*.

Since

AVR-LibC v2.3

20.11.3.73 eeprom_update_llr()

```
void eeprom_update_llr (
             long long fract * __p,
             long long fract __value )
```

Update a `long long fract` at EEPROM address *__p*.

Since

AVR-LibC v2.3

20.11.3.74 eeprom_update_lr()

```
void eeprom_update_lr (
             long fract * __p,
             long fract __value )
```

Update a `long fract` at EEPROM address *__p*.

Since

AVR-LibC v2.3

20.11.3.75 eeprom_update_r()

```
void eeprom_update_r (
             fract * __p,
             fract __value )
```

Update a `fract` at EEPROM address *__p*.

Since

AVR-LibC v2.3

20.11.3.76 eeprom_update_uhk()

```
void eeprom_update_uhk (
            unsigned short accum * __p,
            unsigned short accum __value )
```

Update an `unsigned short accum` at EEPROM address *__p*.

Since

AVR-LibC v2.3

20.11.3.77 eeprom_update_uhr()

```
void eeprom_update_uhr (
            unsigned short fract * __p,
            unsigned short fract __value )
```

Update an `unsigned short fract` at EEPROM address *__p*.

Since

AVR-LibC v2.3

20.11.3.78 eeprom_update_uk()

```
void eeprom_update_uk (
            unsigned accum * __p,
            unsigned accum __value )
```

Update an `unsigned accum` at EEPROM address *__p*.

Since

AVR-LibC v2.3

20.11.3.79 eeprom_update_ulk()

```
void eeprom_update_ulk (
            unsigned long accum * __p,
            unsigned long accum __value )
```

Update an `unsigned long accum` at EEPROM address *__p*.

Since

AVR-LibC v2.3

20.11.3.80 eeprom_update_ullk()

```
void eeprom_update_ullk (
             unsigned long long accum * __p,
             unsigned long long accum __value )
```

Update an `unsigned long long accum` at EEPROM address *__p*.

Since

AVR-LibC v2.3

20.11.3.81 eeprom_update_ullr()

```
void eeprom_update_ullr (
             unsigned long long fract * __p,
             unsigned long long fract __value )
```

Update an `unsigned long long fract` at EEPROM address *__p*.

Since

AVR-LibC v2.3

20.11.3.82 eeprom_update_ulr()

```
void eeprom_update_ulr (
             unsigned long fract * __p,
             unsigned long fract __value )
```

Update an `unsigned long fract` at EEPROM address *__p*.

Since

AVR-LibC v2.3

20.11.3.83 eeprom_update_ur()

```
void eeprom_update_ur (
             unsigned fract * __p,
             unsigned fract __value )
```

Update an `unsigned fract` at EEPROM address *__p*.

Since

AVR-LibC v2.3

20.11.3.84 eeprom_write_hk()

```
void eeprom_write_hk (
             short accum * __p,
             short accum __value )
```

Write a `short accum` to EEPROM address *__p*.

Since

AVR-LibC v2.3

20.11.3.85 eeprom_write_hr()

```
void eeprom_write_hr (
             short fract * __p,
             short fract __value )
```

Write a `short fract` to EEPROM address *__p*.

Since

AVR-LibC v2.3

20.11.3.86 eeprom_write_k()

```
void eeprom_write_k (
             accum * __p,
             accum __value )
```

Write an `accum` to EEPROM address *__p*.

Since

AVR-LibC v2.3

20.11.3.87 eeprom_write_lk()

```
void eeprom_write_lk (
             long accum * __p,
             long accum __value )
```

Write a `long accum` to EEPROM address *__p*.

Since

AVR-LibC v2.3

20.11.3.88 eeprom_write_llk()

```
void eeprom_write_llk (
            long long accum * __p,
            long long accum __value )
```

Write a `long long accum` to EEPROM address *__p*.

Since

AVR-LibC v2.3

20.11.3.89 eeprom_write_llr()

```
void eeprom_write_llr (
            long long fract * __p,
            long long fract __value )
```

Write a `long long fract` to EEPROM address *__p*.

Since

AVR-LibC v2.3

20.11.3.90 eeprom_write_lr()

```
void eeprom_write_lr (
            long fract * __p,
            long fract __value )
```

Write a `long fract` to EEPROM address *__p*.

Since

AVR-LibC v2.3

20.11.3.91 eeprom_write_r()

```
void eeprom_write_r (
            fract * __p,
            fract __value )
```

Write a `fract` to EEPROM address *__p*.

Since

AVR-LibC v2.3

20.11.3.92 eeprom_write_uhk()

```
void eeprom_write_uhk (
            unsigned short accum * __p,
            unsigned short accum __value )
```

Write an `unsigned short accum` to EEPROM address *__p*.

Since

AVR-LibC v2.3

20.11.3.93 eeprom_write_uhr()

```
void eeprom_write_uhr (
            unsigned short fract * __p,
            unsigned short fract __value )
```

Write an `unsigned short fract` to EEPROM address *__p*.

Since

AVR-LibC v2.3

20.11.3.94 eeprom_write_uk()

```
void eeprom_write_uk (
            unsigned accum * __p,
            unsigned accum __value )
```

Write an `unsigned accum` to EEPROM address *__p*.

Since

AVR-LibC v2.3

20.11.3.95 eeprom_write_ulk()

```
void eeprom_write_ulk (
            unsigned long accum * __p,
            unsigned long accum __value )
```

Write an `unsigned long accum` to EEPROM address *__p*.

Since

AVR-LibC v2.3

20.11.3.96 eeprom_write_ullk()

```
void eeprom_write_ullk (
             unsigned long long accum * __p,
             unsigned long long accum __value )
```

Write an `unsigned long long accum` to EEPROM address *__p*.

Since

AVR-LibC v2.3

20.11.3.97 eeprom_write_ullr()

```
void eeprom_write_ullr (
             unsigned long long fract * __p,
             unsigned long long fract __value )
```

Write an `unsigned long long fract` to EEPROM address *__p*.

Since

AVR-LibC v2.3

20.11.3.98 eeprom_write_ulr()

```
void eeprom_write_ulr (
             unsigned long fract * __p,
             unsigned long fract __value )
```

Write an `unsigned long fract` to EEPROM address *__p*.

Since

AVR-LibC v2.3

20.11.3.99 eeprom_write_ur()

```
void eeprom_write_ur (
             unsigned fract * __p,
             unsigned fract __value )
```

Write an `unsigned fract` to EEPROM address *__p*.

Since

AVR-LibC v2.3

20.11.3.100 exp2k()

```
accum exp2k (
             accum x )
```

Compute 2^x with saturation.

The WCET is at least the one of exp2m1ur().

Since

AVR-LibC v2.3

20.11.3.101 exp2m1ur()

```
unsigned fract exp2m1ur (
             unsigned fract x )
```

Compute $2^x - 1$. The returned value is in the range [0, 1).

The absolute error is bounded by $2.2 \cdot 10^{-5} \approx 2^{-15.4}$. The worst case execution time (WCET) is around 200 cycles when MUL is available, and around 1000 cycles when MUL is not available (measured with avr-gcc v15).

Since

AVR-LibC v2.3

20.11.3.102 exp2uk()

```
unsigned accum exp2uk (
             unsigned accum x )
```

Compute 2^x with saturation.

The WCET is at least the one of exp2m1ur().

Since

AVR-LibC v2.3

20.11.3.103 hkbits()

```
short accum hkbits (
             int val )
```

Return a fixed-point value of the same size and signedness, and with the same bit representation like `val`.

20.11.3.104 hktoa()

```
char * hktoa (
             short accum x,
             char * buf,
             unsigned char mode )
```

Convert fixed-point value `x` to a decimal ASCII representation. The result is written to `buf`, and the user is responsible for providing enough memory in `buf`. Returns `buf`.

For the meaning of `mode`, and for the (maximal) number of character written by this function, see uktoa().

Since

AVR-LibC v2.3

20.11.3.105 hrbits()

```
short fract hrbits (
             signed char val )
```

Return a fixed-point value of the same size and signedness, and with the same bit representation like `val`.

20.11.3.106 hrtoa()

```
char * hrtoa (
             short fract x,
             char * buf,
             unsigned char mode )
```

Convert fixed-point value `x` to a decimal ASCII representation. The result is written to `buf`, and the user is responsible for providing enough memory in `buf`. Returns `buf`.

For the meaning of `mode`, and for the (maximal) number of character written by this function, see uktoa().

Since

AVR-LibC v2.3

20.11.3.107 kbits()

```
accum kbits (
             long val )
```

Return a fixed-point value of the same size and signedness, and with the same bit representation like `val`.

20.11.3.108 ktoa()

```
char * ktoa (
            accum x,
            char * buf,
            unsigned char mode )
```

Convert fixed-point value `x` to a decimal ASCII representation. The result is written to `buf`, and the user is responsible for providing enough memory in `buf`. Returns `buf`.

For the meaning of `mode`, and for the (maximal) number of character written by this function, see uktoa().

Since

AVR-LibC v2.3

20.11.3.109 lkbits()

```
long accum lkbits (
            long long val )
```

Return a fixed-point value of the same size and signedness, and with the same bit representation like `val`.

20.11.3.110 llkbits()

```
long long accum llkbits (
            long long val )
```

Return a fixed-point value of the same size and signedness, and with the same bit representation like `val`.

20.11.3.111 llrbits()

```
long long fract llrbits (
            long long val )
```

Return a fixed-point value of the same size and signedness, and with the same bit representation like `val`.

20.11.3.112 log21puhr()

```
unsigned short fract log21puhr (
            unsigned short fract x )
```

Return $\log_2(1+x)$, the logarithm to base 2 of the value 1 + `x`. The result is in the range [0, 1).

The absolute error is bounded by $4.3 \cdot 10^{-3} \approx 2^{-7.5}$. The worst case execution time (WCET) is around 25 cycles when MUL is available, and around 340 cycles when MUL is not available.

Since

AVR-LibC v2.3

20.11.3.113 log21pur()

```
unsigned fract log21pur (
            unsigned fract x )
```

Return $\log_2(1+x)$, the logarithm to base 2 of the value 1 + x. The result is in the range [0, 1).

The absolute error is bounded by $3 \cdot 10^{-5} \approx 2^{-15}$. The worst case execution time (WCET) is around 250 cycles when MUL is available, and around 1300 cycles when MUL is not available.

Since

AVR-LibC v2.3

20.11.3.114 log2uhk()

```
short accum log2uhk (
            unsigned short accum x )
```

Return $\log_2(x)$, the logarithm to base 2 of the value x. The returned value for x = 0 is -128.

The absolute error is bounded by $8 \cdot 10^{-3} \approx 2^{-7}$. The worst case execution time (WCET) is around 60 cycles plus the WCET of log21puhr().

Since

AVR-LibC v2.3

20.11.3.115 log2uk()

```
accum log2uk (
            unsigned accum x )
```

Returns $\log_2(x)$, the logarithm to base 2 of the value x. The returned value for x = 0 is -32768.

The absolute error is bounded by $4.5 \cdot 10^{-5} \approx 2^{-14.5}$. The worst case execution time (WCET) is around 150 cycles more than the WCET of log21pur().

Since

AVR-LibC v2.3

20.11.3.116 lrbits()

```
long fract lrbits (
            long val )
```

Return a fixed-point value of the same size and signedness, and with the same bit representation like val.

20.11.3.117 lrdivi()

```
long fract lrdivi (
            long int num,
            long int denom )
```

The lrdivi() function computes the value of num/denom and returns the result of the long fract type. The return value is rounded towards zero, and is saturated on overflow. If denom is zero, the behavior is undefined.

Since

AVR-LibC v2.3

20.11.3.118 pgm_read_hk()

```
static short accum pgm_read_hk (
            const short accum * addr )  [inline], [static]
```

Read a short accum from 16-bit address addr. The address is in the lower 64 KiB of program memory.

Since

AVR-LibC v2.3

20.11.3.119 pgm_read_hk_far()

```
static short accum pgm_read_hk_far (
            uint_farptr_t addr )  [inline], [static]
```

Read a short accum from far address addr. The address is in the program memory.

Since

AVR-LibC v2.3

20.11.3.120 pgm_read_hr()

```
static short fract pgm_read_hr (
            const short fract * addr )  [inline], [static]
```

Read a short fract from 16-bit address addr. The address is in the lower 64 KiB of program memory.

Since

AVR-LibC v2.3

20.11.3.121 pgm_read_hr_far()

```
static short fract pgm_read_hr_far (
            uint_farptr_t addr )  [inline], [static]
```

Read a `short fract` from far address `addr`. The address is in the program memory.

Since

AVR-LibC v2.3

20.11.3.122 pgm_read_k()

```
static accum pgm_read_k (
            const accum * addr )  [inline], [static]
```

Read an `accum` from 16-bit address `addr`. The address is in the lower 64 KiB of program memory.

Since

AVR-LibC v2.3

20.11.3.123 pgm_read_k_far()

```
static accum pgm_read_k_far (
            uint_farptr_t addr )  [inline], [static]
```

Read an `accum` from far address `addr`. The address is in the program memory.

Since

AVR-LibC v2.3

20.11.3.124 pgm_read_lk()

```
static long accum pgm_read_lk (
            const long accum * addr )  [inline], [static]
```

Read a `long accum` from 16-bit address `addr`. The address is in the lower 64 KiB of program memory.

Since

AVR-LibC v2.3

20.11.3.125 pgm_read_lk_far()

```
static long accum pgm_read_lk_far (
            uint_farptr_t addr )  [inline], [static]
```

Read a `long accum` from far address `addr`. The address is in the program memory.

Since

AVR-LibC v2.3

20.11.3.126 pgm_read_llk()

```
static long long accum pgm_read_llk (
            const long long accum * addr )  [inline], [static]
```

Read a `long long accum` from 16-bit address `addr`. The address is in the lower 64 KiB of program memory.

Since

AVR-LibC v2.3

20.11.3.127 pgm_read_llk_far()

```
static long long accum pgm_read_llk_far (
            uint_farptr_t addr )  [inline], [static]
```

Read a `long long accum` from far address `addr`. The address is in the program memory.

Since

AVR-LibC v2.3

20.11.3.128 pgm_read_llr()

```
static long long fract pgm_read_llr (
            const long long fract * addr )  [inline], [static]
```

Read a `long long fract` from 16-bit address `addr`. The address is in the lower 64 KiB of program memory.

Since

AVR-LibC v2.3

20.11.3.129 pgm_read_llr_far()

```
static long long fract pgm_read_llr_far (
            uint_farptr_t addr )  [inline], [static]
```

Read a `long long fract` from far address `addr`. The address is in the program memory.

Since

AVR-LibC v2.3

20.11.3.130 pgm_read_lr()

```
static long fract pgm_read_lr (
            const long fract * addr )  [inline], [static]
```

Read a `long fract` from 16-bit address `addr`. The address is in the lower 64 KiB of program memory.

Since

AVR-LibC v2.3

20.11.3.131 pgm_read_lr_far()

```
static long fract pgm_read_lr_far (
            uint_farptr_t addr )  [inline], [static]
```

Read a `long fract` from far address `addr`. The address is in the program memory.

Since

AVR-LibC v2.3

20.11.3.132 pgm_read_r()

```
static fract pgm_read_r (
            const fract * addr )  [inline], [static]
```

Read a `fract` from 16-bit address `addr`. The address is in the lower 64 KiB of program memory.

Since

AVR-LibC v2.3

20.11.3.133 pgm_read_r_far()

```
static fract pgm_read_r_far (
            uint_farptr_t addr )  [inline], [static]
```

Read a `fract` from far address `addr`. The address is in the program memory.

Since

AVR-LibC v2.3

20.11.3.134 pgm_read_uhk()

```
static unsigned short accum pgm_read_uhk (
            const unsigned short accum * addr )  [inline], [static]
```

Read an `unsigned short accum` from 16-bit address `addr`. The address is in the lower 64 KiB of program memory.

Since

AVR-LibC v2.3

20.11.3.135 pgm_read_uhk_far()

```
static unsigned short accum pgm_read_uhk_far (
            uint_farptr_t addr )  [inline], [static]
```

Read an `unsigned short accum` from far address `addr`. The address is in the program memory.

Since

AVR-LibC v2.3

20.11.3.136 pgm_read_uhr()

```
static unsigned short fract pgm_read_uhr (
            const unsigned short fract * addr )  [inline], [static]
```

Read an `unsigned short fract` from 16-bit address `addr`. The address is in the lower 64 KiB of program memory.

Since

AVR-LibC v2.3

20.11.3.137 pgm_read_uhr_far()

```
static unsigned short fract pgm_read_uhr_far (
            uint_farptr_t addr )  [inline], [static]
```

Read an `unsigned short fract` from far address `addr`. The address is in the program memory.

Since

AVR-LibC v2.3

20.11.3.138 pgm_read_uk()

```
static unsigned accum pgm_read_uk (
            const unsigned accum * addr )  [inline], [static]
```

Read an `unsigned accum` from 16-bit address `addr`. The address is in the lower 64 KiB of program memory.

Since

AVR-LibC v2.3

20.11.3.139 pgm_read_uk_far()

```
static unsigned accum pgm_read_uk_far (
            uint_farptr_t addr )  [inline], [static]
```

Read an `unsigned accum` from far address `addr`. The address is in the program memory.

Since

AVR-LibC v2.3

20.11.3.140 pgm_read_ulk()

```
static unsigned long accum pgm_read_ulk (
            const unsigned long accum * addr )  [inline], [static]
```

Read an `unsigned long accum` from 16-bit address `addr`. The address is in the lower 64 KiB of program memory.

Since

AVR-LibC v2.3

20.11.3.141 pgm_read_ulk_far()

```
static unsigned long accum pgm_read_ulk_far (
            uint_farptr_t addr )  [inline], [static]
```

Read an `unsigned long accum` from far address `addr`. The address is in the program memory.

Since

AVR-LibC v2.3

20.11.3.142 pgm_read_ullk()

```
static unsigned long long accum pgm_read_ullk (
            const unsigned long long accum * addr )  [inline], [static]
```

Read an `unsigned long long accum` from 16-bit address `addr`. The address is in the lower 64 KiB of program memory.

Since

AVR-LibC v2.3

20.11.3.143 pgm_read_ullk_far()

```
static unsigned long long accum pgm_read_ullk_far (
            uint_farptr_t addr )  [inline], [static]
```

Read an `unsigned long long accum` from far address `addr`. The address is in the program memory.

Since

AVR-LibC v2.3

20.11.3.144 pgm_read_ullr()

```
static unsigned long long fract pgm_read_ullr (
            const unsigned long long fract * addr )  [inline], [static]
```

Read an `unsigned long long fract` from 16-bit address `addr`. The address is in the lower 64 KiB of program memory.

Since

AVR-LibC v2.3

20.11.3.145 pgm_read_ullr_far()

```
static unsigned long long fract pgm_read_ullr_far (
            uint_farptr_t addr )  [inline], [static]
```

Read an `unsigned long long fract` from far address `addr`. The address is in the program memory.

Since

AVR-LibC v2.3

20.11.3.146 pgm_read_ulr()

```
static unsigned long fract pgm_read_ulr (
            const unsigned long fract * addr )  [inline], [static]
```

Read an `unsigned long fract` from 16-bit address `addr`. The address is in the lower 64 KiB of program memory.

Since

AVR-LibC v2.3

20.11.3.147 pgm_read_ulr_far()

```
static unsigned long fract pgm_read_ulr_far (
            uint_farptr_t addr )  [inline], [static]
```

Read an `unsigned long fract` from far address `addr`. The address is in the program memory.

Since

AVR-LibC v2.3

20.11.3.148 pgm_read_ur()

```
static unsigned fract pgm_read_ur (
            const unsigned fract * addr )  [inline], [static]
```

Read an `unsigned fract` from 16-bit address `addr`. The address is in the lower 64 KiB of program memory.

Since

AVR-LibC v2.3

20.11.3.149 pgm_read_ur_far()

```
static unsigned fract pgm_read_ur_far (
            uint_farptr_t addr )  [inline], [static]
```

Read an `unsigned fract` from far address `addr`. The address is in the program memory.

Since

AVR-LibC v2.3

20.11.3.150 rbits()

```
fract rbits (
            int val )
```

Return a fixed-point value of the same size and signedness, and with the same bit representation like `val`.

20.11.3.151 rdivi()

```
fract rdivi (
            int num,
            int denom )
```

The rdivi() function computes the value `num/denom` and returns the result of the `fract` type. The return value is rounded towards zero, and is saturated on overflow. If `denom` is zero, the behavior is undefined.

Since

AVR-LibC v2.3

20.11.3.152 roundfx()

```
type roundfx (
            type val,
            int bit )
```

Round `val` to `bit` fractional bits. When the result does not fit into the range of the return type, the result is saturated.

As an extension, `bit` may be in the range `-IBIT < bit < FBIT`. For example, `bit = -1` rounds to an even value.

20.11.3.153 roundhk()

```
short accum roundhk (
            short accum val,
            int bit )
```

Round `val` to `bit` fractional bits. When the result does not fit into the range of the return type, the result is saturated.

As an extension, `bit` may be in the range `-IBIT < bit < FBIT`. For example, `bit = -1` rounds to an even value.

20.11.3.154 roundhr()

```
short fract roundhr (
             short fract val,
             int bit )
```

Round `val` to `bit` fractional bits. When the result does not fit into the range of the return type, the result is saturated.

20.11.3.155 roundk()

```
accum roundk (
             accum val,
             int bit )
```

Round `val` to `bit` fractional bits. When the result does not fit into the range of the return type, the result is saturated.

As an extension, `bit` may be in the range `-IBIT < bit < FBIT`. For example, `bit = -1` rounds to an even value.

20.11.3.156 roundlk()

```
long accum roundlk (
             long accum val,
             int bit )
```

Round `val` to `bit` fractional bits. When the result does not fit into the range of the return type, the result is saturated.

As an extension, `bit` may be in the range `-IBIT < bit < FBIT`. For example, `bit = -1` rounds to an even value.

20.11.3.157 roundllk()

```
long long accum roundllk (
             long long accum val,
             int bit )
```

Round `val` to `bit` fractional bits. When the result does not fit into the range of the return type, the result is saturated.

As an extension, `bit` may be in the range `-IBIT < bit < FBIT`. For example, `bit = -1` rounds to an even value.

20.11.3.158 roundllr()

```
long long fract roundllr (
             long long fract val,
             int bit )
```

Round `val` to `bit` fractional bits. When the result does not fit into the range of the return type, the result is saturated.

20.11.3.159 roundlr()

```
long fract roundlr (
             long fract val,
             int bit )
```

Round `val` to `bit` fractional bits. When the result does not fit into the range of the return type, the result is saturated.

20.11.3.160 roundr()

```
fract roundr (
             fract val,
             int bit )
```

Round `val` to `bit` fractional bits. When the result does not fit into the range of the return type, the result is saturated.

20.11.3.161 rounduhk()

```
unsigned short accum rounduhk (
             unsigned short accum val,
             int bit )
```

Round `val` to `bit` fractional bits. When the result does not fit into the range of the return type, the result is saturated.

As an extension, `bit` may be in the range `-IBIT < bit < FBIT`. For example, `bit = -1` rounds to an even value.

20.11.3.162 rounduhr()

```
unsigned short fract rounduhr (
             unsigned short fract val,
             int bit )
```

Round `val` to `bit` fractional bits. When the result does not fit into the range of the return type, the result is saturated.

20.11.3.163 rounduk()

```
unsigned accum rounduk (
             unsigned accum val,
             int bit )
```

Round `val` to `bit` fractional bits. When the result does not fit into the range of the return type, the result is saturated.

As an extension, `bit` may be in the range `-IBIT < bit < FBIT`. For example, `bit = -1` rounds to an even value.

20.11.3.164 roundulk()

```
unsigned long accum roundulk (
            unsigned long accum val,
            int bit )
```

Round `val` to `bit` fractional bits. When the result does not fit into the range of the return type, the result is saturated.

As an extension, `bit` may be in the range `-IBIT < bit < FBIT`. For example, `bit = -1` rounds to an even value.

20.11.3.165 roundullk()

```
unsigned long long accum roundullk (
            unsigned long long accum val,
            int bit )
```

Round `val` to `bit` fractional bits. When the result does not fit into the range of the return type, the result is saturated.

As an extension, `bit` may be in the range `-IBIT < bit < FBIT`. For example, `bit = -1` rounds to an even value.

20.11.3.166 roundullr()

```
unsigned long long fract roundullr (
            unsigned long long fract val,
            int bit )
```

Round `val` to `bit` fractional bits. When the result does not fit into the range of the return type, the result is saturated.

20.11.3.167 roundulr()

```
unsigned long fract roundulr (
            unsigned long fract val,
            int bit )
```

Round `val` to `bit` fractional bits. When the result does not fit into the range of the return type, the result is saturated.

20.11.3.168 roundur()

```
unsigned fract roundur (
            unsigned fract val,
            int bit )
```

Round `val` to `bit` fractional bits. When the result does not fit into the range of the return type, the result is saturated.

20.11.3.169 rtoa()

```
char * rtoa (
            fract x,
            char * buf,
            unsigned char mode )
```

Convert fixed-point value `x` to a decimal ASCII representation. The result is written to `buf`, and the user is responsible for providing enough memory in `buf`. Returns `buf`.

For the meaning of `mode`, and for the (maximal) number of character written by this function, see uktoa().

Since

AVR-LibC v2.3

20.11.3.170 sinpi2k()

```
accum sinpi2k (
            accum deg )
```

Sine of $x \cdot \pi/2$ radians.

The absolute error is bounded by $4.6 \cdot 10^{-5} \approx 2^{-14.4}$. The worst case execution time (WCET) is around 300 cycles when MUL is available, and around 1400 cycles when MUL is not available.

Since

AVR-LibC v2.3

20.11.3.171 sinpi2ur()

```
unsigned fract sinpi2ur (
            unsigned fract x )
```

Sine of $x \cdot \pi/2$ redians. The returned value is in the range [0, 1).

The absolute error is bounded by $2.9 \cdot 10^{-5} \approx 2^{-15}$. The worst case execution time (WCET) is around 260 cycles when MUL is available, and around 1400 cycles when MUL is not available.

Since

AVR-LibC v2.3

20.11.3.172 sinuhk_deg()

```
fract sinuhk_deg (
            unsigned short accum deg )
```

Sine of the angle *deg* where *deg* is specified in degrees, i.e. in the range $[0°, 256°)$. The returned value is in the range (-1, +1), i.e. is never -1.

The absolute error is bounded by $6.5 \cdot 10^{-5} \approx 2^{-13.9}$. The worst case execution time (WCET) is around 70 cycles.

Since

AVR-LibC v2.3

20.11.3.173 sqrthk()

```
short accum sqrthk (
            short accum radic )
```

Square root of the value `radic`. Negative values are returned unchanged. The worst case execution time (WCET) is around 310 cycles.

Since

AVR-LibC v2.3

20.11.3.174 sqrthr()

```
short fract sqrthr (
            short fract radic )
```

Square root of the value `radic` rounded down.

Since

AVR-LibC v2.3

20.11.3.175 sqrtk()

```
accum sqrtk (
            accum radic )
```

Square root of the value `radic`. Negative values are returned unchanged. The worst case execution time (WCET) is around 640 cycles.

Since

AVR-LibC v2.3

20.11.3.176 sqrtlr()

```
long fract sqrtlr (
             long fract radic )
```

Square root of the value `radic`. Negative values are returned unchanged. The worst case execution time (WCET) is around 1080 cycles.

Since

AVR-LibC v2.3

20.11.3.177 sqrtr()

```
fract sqrtr (
             fract radic )
```

Square root of the value `radic`. Negative values are returned unchanged. The worst case execution time (WCET) is around 320 cycles.

Since

AVR-LibC v2.3

20.11.3.178 sqrtuhk()

```
unsigned short accum sqrtuhk (
             unsigned short accum radic )
```

Square root of the value `radic` rounded down. The worst case execution time (WCET) is around 300 cycles.

Since

AVR-LibC v2.3

20.11.3.179 sqrtuhr()

```
unsigned short fract sqrtuhr (
             unsigned short fract radic )
```

Square root of the value `radic` rounded down. The result is in the range [0, 1).

The absolute error is in the range $[-3.9 \cdot 10^{-3} \approx 2^{-8}, 0]$. The worst case execution time (WCET) is around 120 cycles.

Since

AVR-LibC v2.3

20.11.3.180 sqrtuk()

```
unsigned accum sqrtuk (
            unsigned accum radic )
```

Square root of the value `radic`. The worst case execution time (WCET) is around 620 cycles.

Since

AVR-LibC v2.3

20.11.3.181 sqrtulr()

```
unsigned long fract sqrtulr (
            unsigned long fract radic )
```

Square root of the value `radic`. The result is in the range [0, 1). The worst case execution time (WCET) is around 1060 cycles.

Since

AVR-LibC v2.3

20.11.3.182 sqrtur()

```
unsigned fract sqrtur (
            unsigned fract radic )
```

Square root of the value `radic`. The result is in the range [0, 1).

The absolute error is in the range $[-1.5 \cdot 10^{-5} \approx 2^{-16}, 0]$. The worst case execution time (WCET) is around 320 cycles.

Since

AVR-LibC v2.3

20.11.3.183 uhkbits()

```
unsigned short accum uhkbits (
            unsigned int val )
```

Return a fixed-point value of the same size and signedness, and with the same bit representation like `val`.

20.11.3.184 uhktoa()

```
char * uhktoa (
            unsigned short accum x,
            char * buf,
            unsigned char mode )
```

Convert fixed-point value `x` to a decimal ASCII representation. The result is written to `buf`, and the user is responsible for providing enough memory in `buf`. Returns `buf`.

For the meaning of `mode`, and for the (maximal) number of character written by this function, see uktoa().

Since

AVR-LibC v2.3

20.11.3.185 uhrbits()

```
unsigned short fract uhrbits (
            unsigned char val )
```

Return a fixed-point value of the same size and signedness, and with the same bit representation like `val`.

20.11.3.186 uhrtoa()

```
char * uhrtoa (
            unsigned short fract x,
            char * buf,
            unsigned char mode )
```

Convert fixed-point value `x` to a decimal ASCII representation. The result is written to `buf`, and the user is responsible for providing enough memory in `buf`. Returns `buf`.

For the meaning of `mode`, and for the (maximal) number of character written by this function, see uktoa().

Since

AVR-LibC v2.3

20.11.3.187 ukbits()

```
unsigned accum ukbits (
            unsigned long val )
```

Return a fixed-point value of the same size and signedness, and with the same bit representation like `val`.

20.11.3.188 uktoa()

```
char * uktoa (
             unsigned accum x,
             char * buf,
             unsigned char mode )
```

Convert fixed-point value `x` to a decimal ASCII representation. The result is written to `buf`, and the user is responsible for providing enough memory in `buf`. Returns `buf`.

The format of the output is controlled by the *mode* parameter. It is composed from the format flags below together with *Digs*, the number of fractional digits. The `mode` is the ORed result from *Digs* and a combination of FXTOA_ALL, FXTOA_ROUND or FXTOA_TRUNC, FXTOA_DOT or FXTOA_COMMA, and FXTOA_NTZ, like in:

```
uktoa (x, buf, FXTOA_ROUND | FXTOA_NTZ | Digs);
```

- Supported values for *Digs* are in the range 0...30.

FXTOA_ALL
Include all significant digits. *Digs* will be ignored.

FXTOA_ROUND
Round to nearest for *Digs* fractional digits. Rounding for *Digs* ≥ 5 has no effect, i.e. for such *Digs* values the rounded result will be the same like the truncated result. Rounding mode is the default, i.e. FXTOA_ROUND can be omitted.

FXTOA_TRUNC
Like FXTOA_ALL, but truncate the result (round to zero) after *Digs* fractional digits.

FXTOA_NTZ
• With FXTOA_NTZ the result has **n**o **t**railing **z**eros. When the result represents an integral value, then no decimal point and no fractional digits are present.
• Without FXTOA_NTZ and with FXTOA_ROUND or FXTOA_TRUNC, the result has the specified number of fractional digits.
• FXTOA_NTZ has no effect with FXTOA_ALL.
• FXTOA_NTZ has no effect on the required buffer size.

FXTOA_DOT
The decimal point is a dot (`.`). This is the default, i.e. FXTOA_DOT can be omitted.

FXTOA_COMMA
The decimal point is a comma (`,`).

Table 40 Examples

Value	Mode	Result
1.8uk	0	"2"
1.8uk	1	"1.8"
1.8uk	2	"1.80"
1.8uk	3	"1.800"
1.8uk	FXTOA_NTZ | 0	"2"
1.8uk	FXTOA_NTZ | 1	"1.8"
1.8uk	FXTOA_NTZ | 2	"1.8"
1.8uk	FXTOA_NTZ | 3	"1.8"
1.8uk	FXTOA_TRUNC | FXTOA_COMMA | 0	"1"
1.8uk	FXTOA_TRUNC | FXTOA_COMMA | 1	"1,7"
1.8uk	FXTOA_TRUNC | FXTOA_COMMA | 2	"1,79"
1.8uk	FXTOA_TRUNC | FXTOA_COMMA | 3	"1,799"
1.8uk	FXTOA_ALL	"1.79998779296875"

The following table helps with providing enough memory in *buf*.

Notice that the required size of *buf* is independent of FXTOA_NTZ. This is the case since with FXTOA_NTZ the required buffer size may be larger that the size of the returned string.

Table 41 Maximum Number of Bytes written to buf

Type	Function	FXTOA_ALL	All other Modes	Exact Maximal Value	
`unsigned accum`	uktoa	23	7 + *Digs*	2^{16}-2^{-16}	65535.↩ 9999847412109375
`accum`	ktoa	23	8 + *Digs*	2^{16}-2^{-15}	65535.↩ 999969482421875
`unsigned short accum`	uhktoa	13	5 + *Digs*	2^{8}-2^{-8}	255.99609375
`short accum`	hktoa	13	6 + *Digs*	2^{8}-2^{-7}	255.9921875
`unsigned fract`	urtoa	19	3 + *Digs*	1-2^{-16}	0.9999847412109375
`fract`	rtoa	19	4 + *Digs*	1-2^{-15}	0.999969482421875
`unsigned short fract`	uhrtoa	11	3 + *Digs*	1-2^{-8}	0.99609375
`short fract`	hrtoa	11	4 + *Digs*	1-2^{-7}	0.9921875

Since

AVR-LibC v2.3

20.11.3.189 ulkbits()

```
unsigned long accum ulkbits (
            unsigned long long val )
```

Return a fixed-point value of the same size and signedness, and with the same bit representation like `val`.

20.11.3.190 ullkbits()

```
unsigned long long accum ullkbits (
            unsigned long long val )
```

Return a fixed-point value of the same size and signedness, and with the same bit representation like `val`.

20.11.3.191 ullrbits()

```
unsigned long long fract ullrbits (
            unsigned long long val )
```

Return a fixed-point value of the same size and signedness, and with the same bit representation like `val`.

20.11.3.192 ulrbits()

```
unsigned long fract ulrbits (
            unsigned long val )
```

Return a fixed-point value of the same size and signedness, and with the same bit representation like `val`.

20.11.3.193 ulrdivi()

```
unsigned long fract ulrdivi (
            unsigned long int num,
            unsigned long int denom )
```

The ulrdivi() function computes the value of `num/denom` and returns the result of the `unsigned long fract` type. The return value is rounded towards zero, and is saturated on overflow. If `denom` is zero, the behavior is undefined.

Since

AVR-LibC v2.3

20.11.3.194 urbits()

```
unsigned fract urbits (
            unsigned int val )
```

Return a fixed-point value of the same size and signedness, and with the same bit representation like `val`.

20.11.3.195 urdivi()

```
unsigned fract urdivi (
            unsigned int num,
            unsigned int denom )
```

The urdivi() function computes the value `num/denom` and returns the result of the `unsigned fract` type. The return value is rounded towards zero, and is saturated on overflow. If `denom` is zero, the behavior is undefined.

Since

AVR-LibC v2.3

20.11.3.196 urtoa()

```
char * urtoa (
            unsigned fract x,
            char * buf,
            unsigned char mode )
```

Convert fixed-point value `x` to a decimal ASCII representation. The result is written to `buf`, and the user is responsible for providing enough memory in `buf`. Returns `buf`.

For the meaning of `mode`, and for the (maximal) number of character written by this function, see uktoa().

Since

AVR-LibC v2.3

20.12 <string.h>: Strings

Macros

- #define _FFS(x)

Functions

- int ffs (int __val)
- int ffsl (long __val)
- int ffsll (long long __val)
- void * memccpy (void *, const void *, int, size_t)
- void * memchr (const void *, int, size_t)
- int memcmp (const void *, const void *, size_t)
- void * memcpy (void *, const void *, size_t)
- void * memmem (const void *, size_t, const void *, size_t)
- void * memmove (void *, const void *, size_t)
- void * memrchr (const void *, int, size_t)
- void * memset (void *, int, size_t)
- char * strcat (char *, const char *)
- char * strchr (const char *, int)
- char * strchrnul (const char *, int)
- int strcmp (const char *, const char *)
- char * strcpy (char *, const char *)
- char * stpcpy (char *, const char *)
- int strcasecmp (const char *, const char *)
- char * strcasestr (const char *, const char *)
- size_t strcspn (const char *__s, const char *__reject)
- char * strdup (const char *s1)
- char * strndup (const char *s, size_t n)
- size_t strlcat (char *, const char *, size_t)
- size_t strlcpy (char *, const char *, size_t)
- size_t strlen (const char *)
- char * strlwr (char *)
- char * strncat (char *, const char *, size_t)
- int strncmp (const char *, const char *, size_t)
- char * strncpy (char *, const char *, size_t)
- int strncasecmp (const char *, const char *, size_t)
- size_t strnlen (const char *, size_t)
- char * strpbrk (const char *__s, const char *__accept)
- char * strrchr (const char *, int)
- char * strrev (char *)
- char * strsep (char **, const char *)
- size_t strspn (const char *__s, const char *__accept)
- char * strstr (const char *, const char *)
- char * strtok (char *, const char *)
- char * strtok_r (char *, const char *, char **)
- char * strupr (char *)

20.12.1 Detailed Description

```
#include <string.h>
```

The string functions perform string operations on NULL terminated strings.

Note

If the strings you are working on reside in program memory (flash), then you will need to use the string functions provided by one of the following headers:

- <avr/pgmspace.h>: Program Space Utilities
- <avr/flash.h>: Utilities for named address-spaces __flash and __flashx

When the strings are defined with PROGMEM or PROGMEM_FAR, then use the first header. When they are defined with __flash or __flashx, then use the second one.

20.12.2 Macro Definition Documentation

20.12.2.1 _FFS

```
#define _FFS(
             x )
```

This macro finds the first (least significant) bit set in the input value.

This macro is very similar to the function ffs() except that it evaluates its argument at compile-time, so it should only be applied to compile-time constant expressions where it will reduce to a constant itself. Application of this macro to expressions that are not constant at compile-time is not recommended, and might result in a huge amount of code generated.

Returns

The _FFS() macro returns the position of the first (least significant) bit set in the word val, or 0 if no bits are set. The least significant bit is position 1. Only 16 bits of argument are evaluated.

20.12.3 Function Documentation

20.12.3.1 ffs()

```
int ffs (
            int val )
```

This function finds the first (least significant) bit set in the input value.

Returns

The ffs() function returns the position of the first (least significant) bit set in the word val, or 0 if no bits are set. The least significant bit is position 1.

Note

For expressions that are constant at compile time, consider using the _FFS macro instead.

20.12.3.2 ffsl()

```
int ffsl (
            long __val )
```

Same as ffs(), for an argument of type long.

20.12.3.3 ffsll()

```
int ffsll (
            long long __val )
```

Same as ffs(), for an argument of type `long long`.

20.12.3.4 memccpy()

```
void * memccpy (
            void * dest,
            const void * src,
            int val,
            size_t len )
```

Copy memory area.

The memccpy() function copies no more than `len` bytes from memory area `src` to memory area `dest`, stopping when the character `val` is found.

Returns

The memccpy() function returns a pointer to the next character in `dest` after `val`, or `NULL` if `val` was not found in the first `len` characters of `src`.

20.12.3.5 memchr()

```
void * memchr (
            const void * src,
            int val,
            size_t len )
```

Scan memory for a character.

The memchr() function scans the first len bytes of the memory area pointed to by src for the character val. The first byte to match val (interpreted as an unsigned character) stops the operation.

Returns

The memchr() function returns a pointer to the matching byte or NULL if the character does not occur in the given memory area.

20.12.3.6 memcmp()

```
int memcmp (
            const void * s1,
            const void * s2,
            size_t len )
```

Compare memory areas.

The memcmp() function compares the first len bytes of the memory areas s1 and s2. The comparision is performed using unsigned char operations.

Returns

The memcmp() function returns an integer less than, equal to, or greater than zero if the first len bytes of s1 is found, respectively, to be less than, to match, or be greater than the first len bytes of s2.

Note

Be sure to store the result in a 16 bit variable since you may get incorrect results if you use an unsigned char or char due to truncation.

Warning

This function is not -mint8 compatible, although if you only care about testing for equality, this function should be safe to use.

20.12.3.7 memcpy()

```
void * memcpy (
            void * dest,
            const void * src,
            size_t len )
```

Copy a memory area.

The memcpy() function copies len bytes from memory area src to memory area dest. The memory areas may not overlap. Use memmove() if the memory areas do overlap.

Returns

The memcpy() function returns a pointer to dest.

20.12.3.8 memmem()

```
void * memmem (
            const void * s1,
            size_t len1,
            const void * s2,
            size_t len2 )
```

The memmem() function finds the start of the first occurrence of the substring `s2` of length `len2` in the memory area `s1` of length `len1`.

Returns

The memmem() function returns a pointer to the beginning of the substring, or `NULL` if the substring is not found. If `len2` is zero, the function returns `s1`.

20.12.3.9 memmove()

```
void * memmove (
            void * dest,
            const void * src,
            size_t len )
```

Copy memory area.

The memmove() function copies len bytes from memory area src to memory area dest. The memory areas may overlap.

Returns

The memmove() function returns a pointer to dest.

20.12.3.10 memrchr()

```
void * memrchr (
            const void * src,
            int val,
            size_t len )
```

The memrchr() function is like the memchr() function, except that it searches backwards from the end of the `len` bytes pointed to by `src` instead of forwards from the front. (Glibc, GNU extension.)

Returns

The memrchr() function returns a pointer to the matching byte or `NULL` if the character does not occur in the given memory area.

20.12.3.11 memset()

```
void * memset (
            void * dest,
            int val,
            size_t len )
```

Fill memory with a constant byte.

The memset() function fills the first len bytes of the memory area pointed to by dest with the constant byte val.

Returns

The memset() function returns a pointer to the memory area dest.

20.12.3.12 stpcpy()

```
char * stpcpy (
            char * dest,
            const char * src )
```

Copy a string.

The stpcpy() function copies the string pointed to by `src` (including the terminating '\0' character) to the array pointed to by `dest`. The strings may not overlap, and the destination string `dest` must be large enough to receive the copy.

Returns

The stpcpy() function returns a pointer to the **end** of the string `dest` (that is, the address of the terminating null byte) rather than the beginning.

Since

AVR-LibC v2.3

20.12.3.13 strcasecmp()

```
int strcasecmp (
            const char * s1,
            const char * s2 )
```

Compare two strings ignoring case.

The strcasecmp() function compares the two strings `s1` and `s2`, ignoring the case of the characters.

Returns

The strcasecmp() function returns an integer less than, equal to, or greater than zero if `s1` is found, respectively, to be less than, to match, or be greater than `s2`. A consequence of the ordering used by strcasecmp() is that if `s1` is an initial substring of `s2`, then `s1` is considered to be "less than" `s2`.

20.12.3.14 strcasestr()

```
char * strcasestr (
            const char * s1,
            const char * s2 )
```

The strcasestr() function finds the first occurrence of the substring `s2` in the string `s1`. This is like strstr(), except that it ignores case of alphabetic symbols in searching for the substring. (Glibc, GNU extension.)

Returns

The strcasestr() function returns a pointer to the beginning of the substring, or `NULL` if the substring is not found. If `s2` points to a string of zero length, the function returns `s1`.

20.12.3.15 strcat()

```
char * strcat (
            char * dest,
            const char * src )
```

Concatenate two strings.

The strcat() function appends the src string to the dest string overwriting the '\0' character at the end of dest, and then adds a terminating '\0' character. The strings may not overlap, and the dest string must have enough space for the result.

Returns

The strcat() function returns a pointer to the resulting string dest.

20.12.3.16 strchr()

```
char * strchr (
            const char * src,
            int val )
```

Locate character in string.

Returns

The strchr() function returns a pointer to the first occurrence of the character `val` in the string `src`, or `NULL` if the character is not found.

Here "character" means "byte" – these functions do not work with wide or multi-byte characters.

20.12.3.17 strchrnul()

```
char * strchrnul (
            const char * s,
            int c )
```

The strchrnul() function is like strchr() except that if `c` is not found in `s`, then it returns a pointer to the null byte at the end of `s`, rather than `NULL`. (Glibc, GNU extension.)

Returns

The strchrnul() function returns a pointer to the matched character, or a pointer to the null byte at the end of `s` (i.e., `s+strlen(s)`) if the character is not found.

20.12.3.18 strcmp()

```
int strcmp (
            const char * s1,
            const char * s2 )
```

Compare two strings.

The strcmp() function compares the two strings `s1` and `s2`.

Returns

The strcmp() function returns an integer less than, equal to, or greater than zero if `s1` is found, respectively, to be less than, to match, or be greater than `s2`. A consequence of the ordering used by strcmp() is that if `s1` is an initial substring of `s2`, then `s1` is considered to be "less than" `s2`.

20.12.3.19 strcpy()

```
char * strcpy (
            char * dest,
            const char * src )
```

Copy a string.

The strcpy() function copies the string pointed to by src (including the terminating '\0' character) to the array pointed to by dest. The strings may not overlap, and the destination string dest must be large enough to receive the copy.

Returns

The strcpy() function returns a pointer to the destination string dest.

See also

strncpy(), stpcpy(), strcpy_P(), strcpy_F().

20.12.3.20 strcspn()

```
size_t strcspn (
            const char * s,
            const char * reject )
```

The strcspn() function calculates the length of the initial segment of `s` which consists entirely of characters not in `reject`.

Returns

The strcspn() function returns the number of characters in the initial segment of `s` which are not in the string `reject`. The terminating zero is not considered as a part of string.

20.12.3.21 strdup()

```
char * strdup (
            const char * s1 )
```

Duplicate a string.

The strdup() function allocates memory and copies into it the string addressed by `s1`, including the terminating null character.

Warning

The strdup() function calls malloc() to allocate the memory for the duplicated string! The user is responsible for freeing the memory by calling free().

Returns

The strdup() function returns a pointer to the resulting string dest. If malloc() cannot allocate enough storage for the string, strdup() will return `NULL`.

Warning

Be sure to check the return value of the strdup() function to make sure that the function has succeeded in allocating the memory!

20.12.3.22 strlcat()

```
size_t strlcat (
            char * dst,
            const char * src,
            size_t siz )
```

Concatenate two strings.

Appends `src` to string `dst` of size `siz` (unlike strncat(), `siz` is the full size of `dst`, not space left). At most `siz-1` characters will be copied. Always `'\0'` terminated (unless `siz` <= `strlen(dst)`).

Returns

The strlcat() function returns strlen(src) + MIN(siz, strlen(initial dst)). If retval >= siz, truncation occurred.

Appends src to string dst of size siz (unlike strncat(), siz is the full size of dst, not space left). At most siz-1 characters will be copied. Always NULL terminates (unless siz <= strlen(dst)).

Returns

The strlcat() function returns strlen(src) + MIN(siz, strlen(initial dst)). If retval >= siz, truncation occurred.

20.12.3.23 strlcpy()

```
size_t strlcpy (
            char * dst,
            const char * src,
            size_t siz )
```

Copy a string.

Copy `src` to string `dst` of size `siz`. At most `siz-1` characters will be copied. Always '\0' terminated (unless `siz` == 0).

Returns

The strlcpy() function returns strlen(src). If retval >= siz, truncation occurred.

Copy src to string dst of size siz. At most siz-1 characters will be copied. Always NULL terminates (unless siz == 0).

Returns

The strlcpy() function returns strlen(src). If retval >= siz, truncation occurred.

20.12.3.24 strlen()

```
size_t strlen (
            const char * src )
```

Calculate the length of a string.

The strlen() function calculates the length of the string `src`, not including the terminating '\0' character.

Returns

The strlen() function returns the number of characters in `src`.

20.12.3.25 strlwr()

```
char * strlwr (
            char * s )
```

Convert a string to lower case.

The strlwr() function will convert a string to lower case. Only the upper case alphabetic characters [A .. Z] are converted. Non-alphabetic characters will not be changed.

Returns

The strlwr() function returns a pointer to the converted string. Conversion is perfomed in-place.

20.12.3.26 strncasecmp()

```
int strncasecmp (
            const char * s1,
            const char * s2,
            size_t len )
```

Compare two strings ignoring case.

The strncasecmp() function is similar to strcasecmp(), except it only compares the first `len` characters of `s1`.

Returns

The strncasecmp() function returns an integer less than, equal to, or greater than zero if `s1` (or the first `len` bytes thereof) is found, respectively, to be less than, to match, or be greater than `s2`. A consequence of the ordering used by strncasecmp() is that if `s1` is an initial substring of `s2`, then `s1` is considered to be "less than" `s2`.

20.12.3.27 strncat()

```
char * strncat (
            char * dest,
            const char * src,
            size_t len )
```

Concatenate two strings.

The strncat() function is similar to strcat(), except that only the first `len` characters of `src` are appended to `dest`.

Returns

The strncat() function returns a pointer to the resulting string `dest`.

20.12.3.28 strncmp()

```
int strncmp (
            const char * s1,
            const char * s2,
            size_t len )
```

Compare two strings.

The strncmp() function is similar to strcmp(), except it only compares the first (at most) `len` characters of `s1` and `s2`.

Returns

The strncmp() function returns an integer less than, equal to, or greater than zero if `s1` (or the first `len` bytes thereof) is found, respectively, to be less than, to match, or be greater than `s2`.

20.12.3.29 strncpy()

```
char * strncpy (
            char * dest,
            const char * src,
            size_t len )
```

Copy a string.

The strncpy() function is similar to strcpy(), except that not more than `len` bytes of `src` are copied. Thus, if there is no null byte among the first `len` bytes of `src`, the result will not be null-terminated.

In the case where the length of `src` is less than that of `len`, the remainder of `dest` will be padded with nulls (`'\0'`).

Returns

The strncpy() function returns a pointer to the destination string `dest`.

20.12.3.30 strndup()

```
char * strndup (
            const char * s,
            size_t len )
```

Duplicate a string.

The strndup() function is similar to strdup(), but copies at most `len` bytes. If `s` is longer than `len`, only `len` bytes are copied, and a terminating null byte (`'\0'`) is added.

Memory for the new string is obtained with malloc(), and can be freed with free().

Returns

The strndup() function returns the location of the newly malloc'ed memory, or `NULL` if the allocation failed.

20.12.3.31 strnlen()

```
size_t strnlen (
            const char * src,
            size_t len )
```

Determine the length of a fixed-size string.

The strnlen() function returns the number of characters in the string pointed to by `src`, not including the terminating '\0' character, but at most `len`. In doing this, strnlen() looks only at the first `len` characters at `src` and never beyond `src` + `len`.

Returns

The strnlen function returns strlen(src), if that is less than `len`, or `len` if there is no '\0' character among the first `len` characters pointed to by `src`.

20.12.3.32 strpbrk()

```
char * strpbrk (
            const char * s,
            const char * accept )
```

The strpbrk() function locates the first occurrence in the string `s` of any of the characters in the string `accept`.

Returns

The strpbrk() function returns a pointer to the character in `s` that matches one of the characters in `accept`, or `NULL` if no such character is found. The terminating zero is not considered as a part of string: if one or both args are empty, the result will be `NULL`.

20.12.3.33 strrchr()

```
char * strrchr (
            const char * src,
            int val )
```

Locate character in string.

The strrchr() function returns a pointer to the last occurrence of the character val in the string src.

Here "character" means "byte" – these functions do not work with wide or multi-byte characters.

Returns

The strrchr() function returns a pointer to the matched character or `NULL` if the character is not found.

20.12.3.34 strrev()

```
char * strrev (
            char * s )
```

Reverse a string.

The strrev() function reverses the order of the string.

Returns

The strrev() function returns a pointer to the beginning of the reversed string.

20.12.3.35 strsep()

```
char * strsep (
            char ** sp,
            const char * delim )
```

Parse a string into tokens.

The strsep() function locates, in the string referenced by *sp, the first occurrence of any character in the string delim (or the terminating '\0' character) and replaces it with a '\0'. The location of the next character after the delimiter character (or NULL, if the end of the string was reached) is stored in *sp. An ``empty'' field, i.e. one caused by two adjacent delimiter characters, can be detected by comparing the location referenced by the pointer returned in *sp to '\0'.

Returns

The strsep() function returns a pointer to the original value of *sp. If *sp is initially NULL, strsep() returns NULL.

20.12.3.36 strspn()

```
size_t strspn (
            const char * s,
            const char * accept )
```

The strspn() function calculates the length of the initial segment of s which consists entirely of characters in accept.

Returns

The strspn() function returns the number of characters in the initial segment of s which consist only of characters from accept. The terminating zero is not considered as a part of string.

20.12.3.37 strstr()

```
char * strstr (
            const char * s1,
            const char * s2 )
```

Locate a substring.

The strstr() function finds the first occurrence of the substring s2 in the string s1. The terminating '\0' characters are not compared.

Returns

The strstr() function returns a pointer to the beginning of the substring, or NULL if the substring is not found. If s2 points to a string of zero length, the function returns s1.

20.12.3.38 strtok()

```
char * strtok (
            char * s,
            const char * delim )
```

Parses the string s into tokens.

strtok parses the string s into tokens. The first call to strtok should have s as its first argument. Subsequent calls should have the first argument set to NULL. If a token ends with a delimiter, this delimiting character is overwritten with a '\0' and a pointer to the next character is saved for the next call to strtok. The delimiter string delim may be different for each call.

Returns

The strtok() function returns a pointer to the next token or NULL when no more tokens are found.

Note

strtok() is NOT reentrant. For a reentrant version of this function see strtok_r().

20.12.3.39 strtok_r()

```
char * strtok_r (
            char * string,
            const char * delim,
            char ** last )
```

Parses string into tokens.

strtok_r parses string into tokens. The first call to strtok_r should have string as its first argument. Subsequent calls should have the first argument set to NULL. If a token ends with a delimiter, this delimiting character is overwritten with a '\0' and a pointer to the next character is saved for the next call to strtok_r. The delimiter string delim may be different for each call. last is a user allocated char* pointer. It must be the same while parsing the same string. strtok_r is a reentrant version of strtok().

Returns

The strtok_r() function returns a pointer to the next token or NULL when no more tokens are found.

20.12.3.40 strupr()

```
char * strupr (
            char * s )
```

Convert a string to upper case.

The strupr() function will convert a string to upper case. Only the lower case alphabetic characters [a .. z] are converted. Non-alphabetic characters will not be changed.

Returns

The strupr() function returns a pointer to the converted string. The pointer is the same as that passed in since the operation is perform in place.

20.13 <time.h>: Time

Data Structures

- struct tm
- struct week_date

Macros

- #define ONE_HOUR 3600
- #define ONE_DEGREE 3600
- #define ONE_DAY 86400
- #define UNIX_OFFSET 946684800
- #define NTP_OFFSET 3155673600

Typedefs

- typedef uint32_t time_t

Enumerations

- enum _WEEK_DAYS_ {
  SUNDAY , **MONDAY** , **TUESDAY** , **WEDNESDAY** ,
  THURSDAY , **FRIDAY** , **SATURDAY** }
- enum _MONTHS_ {
  JANUARY , **FEBRUARY** , **MARCH** , **APRIL** ,
  MAY , **JUNE** , **JULY** , **AUGUST** ,
  SEPTEMBER , **OCTOBER** , **NOVEMBER** , **DECEMBER** }

Functions

- time_t time (time_t *timer)
- int32_t difftime (time_t time1, time_t time0)
- time_t mktime (struct tm *timeptr)
- time_t mk_gmtime (const struct tm *timeptr)
- struct tm * gmtime (const time_t *timer)
- void gmtime_r (const time_t *timer, struct tm *timeptr)
- struct tm * localtime (const time_t *timer)
- void localtime_r (const time_t *timer, struct tm *timeptr)
- char * asctime (const struct tm *timeptr)
- void asctime_r (const struct tm *timeptr, char *buf)
- char * ctime (const time_t *timer)
- void ctime_r (const time_t *timer, char *buf)
- char * isotime (const struct tm *tmptr)
- void isotime_r (const struct tm *, char *)
- size_t strftime (char *s, size_t maxsize, const char *format, const struct tm *timeptr)
- void set_dst (int(*)(const time_t *, int32_t *))
- void set_zone (int32_t)
- void set_system_time (time_t timestamp)
- void system_tick (void)
- uint8_t is_leap_year (int16_t year)

- uint8_t month_length (int16_t year, uint8_t month)
- uint8_t week_of_year (const struct tm *timeptr, uint8_t start)
- uint8_t week_of_month (const struct tm *timeptr, uint8_t start)
- struct week_date * iso_week_date (int year, int yday)
- void iso_week_date_r (int year, int yday, struct week_date *)
- uint32_t fatfs_time (const struct tm *timeptr)
- void set_position (int32_t latitude, int32_t longitude)
- int16_t equation_of_time (const time_t *timer)
- int32_t daylight_seconds (const time_t *timer)
- time_t solar_noon (const time_t *timer)
- time_t sun_rise (const time_t *timer)
- time_t sun_set (const time_t *timer)
- float solar_declinationf (const time_t *timer)
- double solar_declination (const time_t *timer)
- long double solar_declinationl (const time_t *timer)
- int8_t moon_phase (const time_t *timer)
- uint32_t gm_sidereal (const time_t *timer)
- uint32_t lm_sidereal (const time_t *timer)

20.13.1 Detailed Description

```
#include <time.h>
```

Introduction to the Time functions This file declares the time functions implemented in AVR-LibC.

The implementation aspires to conform with ISO/IEC 9899 (C90). However, due to limitations of the target processor and the nature of its development environment, a practical implementation must of necessity deviate from the standard.

Section 7.23.2.1 `clock()`
The type `clock_t`, the macro `CLOCKS_PER_SEC`, and the function `clock()` are not implemented. We consider these items belong to operating system code, or to application code when no operating system is present.

Section 7.23.2.3 `mktime()`
The standard specifies that `mktime()` should return `(time_t) -1` if the time cannot be represented. This implementation always returns a 'best effort' representation, which means that there are corner cases where non-representable times are not mapped to `-1`. The purpose is to avoid 64-bit arithmetic.

Section 7.23.2.4 `time()`
The standard specifies that `time()` should return `(time_t) -1` if the time is not available. Since the application must initialize the time system, this functionality is not implemented.

Section 7.23.2.2, `difftime()`
Due to the lack of a 64-bit double, the function difftime() returns a `long` integer. In most cases this change will be invisible to the user, handled automatically by the compiler.

Section 7.23.1.4 `struct tm`
Per the standard, struct `tm->tm_isdst` is greater than zero when Daylight Saving time is in effect. This implementation further specifies that, when positive, the value of `tm_isdst` represents the amount time is advanced during Daylight Saving time.

Section 7.23.3.5 `strftime()`
Only the 'C' locale is supported, therefore the modifiers 'E' and 'O' are ignored. The 'Z' conversion is also ignored, due to the lack of time zone name.

In addition to the above departures from the standard, there are some behaviors which are different from what is often expected, though allowed under the standard.

There is no 'platform standard' method to obtain the current time, time zone, or daylight savings 'rules' in the AVR environment. Therefore the application must initialize the time system with this information. The functions `set_zone()`, `set_dst()`, and `set_system_time()` are provided for initialization. Once initialized, system time is maintained by calling the function `system_tick()` at one second intervals.

Though not specified in the standard, it is often expected that `time_t` is a signed integer representing an offset in seconds from Midnight Jan 1 1970, i.e. 'Unix time'. This implementation uses an unsigned 32-bit integer offset from Midnight Jan 1 2000. The use of this 'epoch' helps to simplify the conversion functions, while the 32-bit value allows time to be properly represented until Tue Feb 7 06:28:15 2136 UTC. The macros `UNIX_OFFSET` and `NTP_OFFSET` are defined to assist in converting to and from Unix and NTP time stamps.

Unlike desktop counterparts, it is impractical to implement or maintain the 'zoneinfo' database. Therefore no attempt is made to account for time zone, daylight saving, or leap seconds in past dates. All calculations are made according to the currently configured time zone and daylight saving 'rule'.

In addition to C standard functions, re-entrant versions of `ctime()`, `asctime()`, `gmtime()` and `localtime()` are provided which, in addition to being re-entrant, have the property of claiming less permanent storage in RAM. An additional time conversion, `isotime()` and its re-entrant version, uses far less storage than either `ctime()` or `asctime()`.

Along with the usual smattering of utility functions, such as `is_leap_year()`, this library includes a set of functions related the sun and moon, as well as sidereal time functions.

20.13.2 Macro Definition Documentation

20.13.2.1 NTP_OFFSET

```
#define NTP_OFFSET 3155673600
```

Difference between the Y2K and the NTP epochs, in seconds. To convert a Y2K timestamp to NTP:

```
unsigned long ntp;
time_t y2k;

y2k = time(NULL);
ntp = y2k + NTP_OFFSET;
```

20.13.2.2 ONE_DAY

```
#define ONE_DAY 86400
```

One day, expressed in seconds

20.13.2.3 ONE_DEGREE

```
#define ONE_DEGREE 3600
```

Angular degree, expressed in arc seconds

20.13.2.4 ONE_HOUR

```
#define ONE_HOUR 3600
```

One hour, expressed in seconds

20.13.2.5 UNIX_OFFSET

```
#define UNIX_OFFSET 946684800
```

Difference between the Y2K and the UNIX epochs, in seconds. To convert a Y2K timestamp to UNIX:

```
long unix;
time_t y2k;

y2k = time(NULL);
unix = y2k + UNIX_OFFSET;
```

20.13.3 Typedef Documentation

20.13.3.1 time_t

```
typedef uint32_t time_t
```

`time_t` represents seconds elapsed from Midnight, Jan 1 2000 UTC (the Y2K 'epoch'). Its range allows this implementation to represent time up to Tue Feb 7 06:28:15 2136 UTC.

20.13.4 Enumeration Type Documentation

20.13.4.1 _MONTHS_

```
enum _MONTHS_
```

Enumerated labels for the months.

20.13.4.2 _WEEK_DAYS_

```
enum _WEEK_DAYS_
```

Enumerated labels for the days of the week.

20.13.5 Function Documentation

20.13.5.1 asctime()

```
char * asctime (
             const struct tm * timeptr )
```

The asctime function converts the broken-down time of timeptr, into an ASCII string in the form

```
Sun Mar 23 01:03:52 2013
```

20.13.5.2 asctime_r()

```
void asctime_r (
             const struct tm * timeptr,
             char * buf )
```

Re entrant version of `asctime()`.

20.13.5.3 ctime()

```
char * ctime (
             const time_t * timer )
```

The ctime function is equivalent to `asctime(localtime(timer))`.

20.13.5.4 ctime_r()

```
void ctime_r (
             const time_t * timer,
             char * buf )
```

Re entrant version of `ctime()`.

20.13.5.5 daylight_seconds()

```
int32_t daylight_seconds (
             const time_t * timer )
```

Computes the amount of time the sun is above the horizon, at the location of the observer.

Note: At observer locations inside a polar circle, this value can be zero during the winter, and can exceed `ONE_DAY` during the summer.

The returned value is in seconds.

20.13.5.6 difftime()

```
int32_t difftime (
             time_t time1,
             time_t time0 )
```

The difftime function returns the difference between two binary time stamps, time1 - time0.

20.13.5.7 equation_of_time()

```
int16_t equation_of_time (
            const time_t * timer )
```

Computes the difference between apparent solar time and mean solar time. The returned value is in seconds.

20.13.5.8 fatfs_time()

```
uint32_t fatfs_time (
            const struct tm * timeptr )
```

Convert a Y2K time stamp into a FAT file system time stamp.

20.13.5.9 gm_sidereal()

```
uint32_t gm_sidereal (
            const time_t * timer )
```

Returns Greenwich Mean Sidereal Time, as seconds into the sidereal day. The returned value will range from 0 through 86399 seconds.

20.13.5.10 gmtime()

```
struct tm * gmtime (
            const time_t * timer )
```

The gmtime function converts the time stamp pointed to by timer into broken-down time, expressed as UTC.

20.13.5.11 gmtime_r()

```
void gmtime_r (
            const time_t * timer,
            struct tm * timeptr )
```

Re entrant version of `gmtime()`.

20.13.5.12 is_leap_year()

```
uint8_t is_leap_year (
            int16_t year )
```

Return 1 if year is a leap year, zero if it is not.

20.13.5.13 iso_week_date()

```
struct week_date * iso_week_date (
            int year,
            int yday )
```

Return a week_date structure with the ISO_8601 week based date corresponding to the given year and day of year. See http://en.wikipedia.org/wiki/ISO_week_date for more information.

20.13.5.14 iso_week_date_r()

```
void iso_week_date_r (
             int year,
             int yday,
             struct week_date * iso )
```

Re-entrant version of `iso_week_date()`.

20.13.5.15 isotime()

```
char * isotime (
             const struct tm * tmptr )
```

The isotime function constructs an ASCII string in the form

```
2013-03-23 01:03:52
```

20.13.5.16 isotime_r()

```
void isotime_r (
             const struct tm * tmptr,
             char * buffer )
```

Re entrant version of isotime()

20.13.5.17 lm_sidereal()

```
uint32_t lm_sidereal (
             const time_t * timer )
```

Returns Local Mean Sidereal Time, as seconds into the sidereal day. The returned value will range from 0 through 86399 seconds.

20.13.5.18 localtime()

```
struct tm * localtime (
             const time_t * timer )
```

The localtime function converts the time stamp pointed to by timer into broken-down time, expressed as Local time.

20.13.5.19 localtime_r()

```
void localtime_r (
             const time_t * timer,
             struct tm * timeptr )
```

Re entrant version of `localtime()`.

20.13.5.20 mk_gmtime()

```
time_t mk_gmtime (
            const struct tm * timeptr )
```

This function 'compiles' the elements of a broken-down time structure, returning a binary time stamp. The elements of `timeptr` are interpreted as representing UTC.

The original values of the tm_wday and tm_yday elements of the structure are ignored, and the original values of the other elements are not restricted to the ranges stated for struct `tm`.

Unlike `mktime()`, this function *does not* modify the elements of `timeptr`.

20.13.5.21 mktime()

```
time_t mktime (
            struct tm * timeptr )
```

This function 'compiles' the elements of a broken-down time structure, returning a binary time stamp. The elements of timeptr are interpreted as representing Local Time.

The original values of the `tm_wday` and tm_yday elements of the structure are ignored, and the original values of the other elements are not restricted to the ranges stated for struct tm.

The element `tm_isdst` is used for input and output. If set to 0 or a positive value on input, this requests calculation for Daylight Savings Time being off or on, respectively. If set to a negative value on input, it requests calculation to return whether Daylight Savings Time is in effect or not according to the other values.

On successful completion, the values of all elements of timeptr are set to the appropriate range.

20.13.5.22 month_length()

```
uint8_t month_length (
            int16_t year,
            uint8_t month )
```

Return the length of month, given the year and month, where month is in the range 1 to 12.

20.13.5.23 moon_phase()

```
int8_t moon_phase (
            const time_t * timer )
```

Returns an approximation to the phase of the moon. The sign of the returned value indicates a waning or waxing phase. The magnitude of the returned value indicates the percentage illumination.

20.13.5.24 set_dst()

```
void set_dst (
            int(*)(const time_t *, int32_t *) d )
```

Specify the Daylight Saving function.

The Daylight Saving function should examine its parameters to determine whether Daylight Saving is in effect, and return a value appropriate for `tm_isdst`.

Working examples for the USA and the EU are available:

```
#include <util/eu_dst.h>
```

for the European Union, and

```
#include <util/usa_dst.h>
```

for the United States.

If a Daylight Saving function is not specified, the system will ignore Daylight Saving.

20.13.5.25 set_position()

```
void set_position (
             int32_t latitude,
             int32_t longitude )
```

Set the geographic coordinates of the 'observer', for use with several of the following functions. Parameters are passed as seconds of North Latitude, and seconds of East Longitude.

For New York City:

```
set_position (40.7142 * ONE_DEGREE, -74.0064 * ONE_DEGREE);
```

20.13.5.26 set_system_time()

```
void set_system_time (
             time_t timestamp )
```

Initialize the system time. Examples are...

From a Clock / Calendar type RTC:

```
struct tm rtc_time;

read_rtc(&rtc_time);
rtc_time.tm_isdst = 0;
set_system_time( mktime(&rtc_time) );
```

From a Network Time Protocol time stamp:

```
set_system_time(ntp_timestamp - NTP_OFFSET);
```

From a UNIX time stamp:

```
set_system_time(unix_timestamp - UNIX_OFFSET);
```

20.13.5.27 set_zone()

```
void set_zone (
             int32_t  )
```

Set the 'time zone'. The parameter is given in seconds East of the Prime Meridian. Example for New York City:

```
set_zone(-5 * ONE_HOUR);
```

If the time zone is not set, the time system will operate in UTC only.

20.13.5.28 solar_declination()

```
double solar_declination (
             const time_t * timer )
```

Returns the declination of the sun in radians.

This implementation is only available when `double` is a 32-bit type.

20.13.5.29 solar_declinationf()

```
float solar_declinationf (
            const time_t * timer )
```

Returns the declination of the sun in radians.

20.13.5.30 solar_declinationl()

```
long double solar_declinationl (
            const time_t * timer )
```

Returns the declination of the sun in radians.

This implementation is only available when `long double` is a 32-bit type.

20.13.5.31 solar_noon()

```
time_t solar_noon (
            const time_t * timer )
```

Computes the time of solar noon, at the location of the observer.

20.13.5.32 strftime()

```
size_t strftime (
            char * s,
            size_t maxsize,
            const char * format,
            const struct tm * timeptr )
```

A complete description of `strftime()` is beyond the pale of this document. Refer to ISO/IEC document 9899 for details.

All conversions are made using the C Locale, ignoring the E or O modifiers. Due to the lack of a time zone 'name', the 'Z' conversion is also ignored.

20.13.5.33 sun_rise()

```
time_t sun_rise (
            const time_t * timer )
```

Return the time of sunrise, at the location of the observer. See the note about `daylight_seconds()`.

20.13.5.34 sun_set()

```
time_t sun_set (
            const time_t * timer )
```

Return the time of sunset, at the location of the observer. See the note about `daylight_seconds()`.

20.13.5.35 system_tick()

```
void system_tick (
             void  )
```

Maintain the system time by calling this function at a rate of 1 Hertz.

It is anticipated that this function will typically be called from within an Interrupt Service Routine, (though that is not required). It therefore includes code which makes it simple to use from within a naked ISR, avoiding the cost of saving and restoring all the CPU registers.

Such an ISR may resemble the following example:

```
ISR(RTC_OVF_vect, ISR_NAKED)
{
    system_tick();
    reti();
}
```

20.13.5.36 time()

```
time_t time (
             time_t * timer )
```

The `time` function returns the systems current time stamp. If timer is not a null pointer, the return value is also assigned to the object it points to.

20.13.5.37 week_of_month()

```
uint8_t week_of_month (
             const struct tm * timeptr,
             uint8_t start )
```

Return the calendar week of month, where the first week is considered to begin on the day of week specified by `start`. The returned value may range from zero to 5.

20.13.5.38 week_of_year()

```
uint8_t week_of_year (
             const struct tm * timeptr,
             uint8_t start )
```

Return the calendar week of year, where week 1 is considered to begin on the day of week specified by `start`. The returned value may range from zero to 52.

20.14 <avr/boot.h>: Bootloader Support Utilities

Macros

- #define BOOTLOADER_SECTION __attribute__ ((__section__(".bootloader")))
- #define boot_spm_interrupt_enable() (__SPM_REG |= (uint8_t)_BV(SPMIE))
- #define boot_spm_interrupt_disable() (__SPM_REG &= (uint8_t)~_BV(SPMIE))
- #define boot_is_spm_interrupt() (__SPM_REG & (uint8_t)_BV(SPMIE))
- #define boot_rww_busy() (__SPM_REG & (uint8_t)_BV(__COMMON_ASB))
- #define boot_spm_busy() (__SPM_REG & (uint8_t)_BV(__SPM_ENABLE))
- #define boot_spm_busy_wait() do{}while(boot_spm_busy())
- #define GET_LOW_FUSE_BITS (0x0000)
- #define GET_LOCK_BITS (0x0001)
- #define GET_EXTENDED_FUSE_BITS (0x0002)
- #define GET_HIGH_FUSE_BITS (0x0003)
- #define boot_lock_fuse_bits_get(address)
- #define boot_signature_byte_get(addr)
- #define boot_page_fill(address, data) __boot_page_fill_normal(address, data)
- #define boot_page_erase(address) __boot_page_erase_normal(address)
- #define boot_page_write(address) __boot_page_write_normal(address)
- #define boot_rww_enable() __boot_rww_enable()
- #define boot_lock_bits_set(lock_bits) __boot_lock_bits_set(lock_bits)
- #define boot_page_fill_safe(address, data)
- #define boot_page_erase_safe(address)
- #define boot_page_write_safe(address)
- #define boot_rww_enable_safe()
- #define boot_lock_bits_set_safe(lock_bits)

20.14.1 Detailed Description

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

The macros in this module provide a C language interface to the bootloader support functionality of certain AVR processors. These macros are designed to work with all sizes of flash memory.

Global interrupts are not automatically disabled for these macros. It is left up to the programmer to do this. See the code example below. Also see the processor datasheet for caveats on having global interrupts enabled during writing of the Flash.

Note

Not all AVR processors provide bootloader support. See your processor datasheet to see if it provides bootloader support.

API Usage Example

The following code shows typical usage of the boot API.

```
#include <stdint.h>
#include <avr/interrupt.h>
#include <avr/eeprom.h>
#include <avr/pgmspace.h>

void boot_program_page (uint32_t page, uint8_t *buf)
{
    // Disable interrupts.
    uint8_t sreg = SREG;
    cli();

    eeprom_busy_wait ();

    boot_page_erase (page);
    boot_spm_busy_wait ();      // Wait until the memory is erased.

    for (uint16_t i = 0; i < SPM_PAGESIZE; i += 2)
    {
        // Set up little-endian word.
        uint16_t w = *buf++;
        w += (*buf++) << 8;

        boot_page_fill (page + i, w);
    }

    boot_page_write (page);     // Store buffer in flash page.
    boot_spm_busy_wait();       // Wait until the memory is written.

    // Reenable RWW-section again. We need this if we want to jump back
    // to the application after bootloading.
    boot_rww_enable ();

    // Re-enable interrupts (if they were ever enabled).
    SREG = sreg;
}
```

20.14.2 Macro Definition Documentation

20.14.2.1 boot_is_spm_interrupt

```
#define boot_is_spm_interrupt( ) (__SPM_REG & (uint8_t)_BV(SPMIE))
```

Check if the SPM interrupt is enabled.

20.14.2.2 boot_lock_bits_set

```
#define boot_lock_bits_set(
            lock_bits ) __boot_lock_bits_set(lock_bits)
```

Set the bootloader lock bits.

Parameters

lock_bits	A mask of which Boot Loader Lock Bits to set.

Note

In this context, a 'set bit' will be written to a zero value. Note also that only BLBxx bits can be programmed by this command.

For example, to disallow the SPM instruction from writing to the Boot Loader memory section of flash, you would use this macro as such:

```
boot_lock_bits_set (_BV (BLB11));
```

Note

Like any lock bits, the Boot Loader Lock Bits, once set, cannot be cleared again except by a chip erase which will in turn also erase the boot loader itself.

20.14.2.3 boot_lock_bits_set_safe

```
#define boot_lock_bits_set_safe(
              lock_bits )
```

Value:

```
do { \
    boot_spm_busy_wait();                       \
    eeprom_busy_wait();                         \
    boot_lock_bits_set (lock_bits);             \
} while (0)
```

Same as boot_lock_bits_set() except waits for eeprom and spm operations to complete before setting the lock bits.

20.14.2.4 boot_lock_fuse_bits_get

```
#define boot_lock_fuse_bits_get(
              address )
```

Value:

```
(({                                             \
    uint8_t __result;                                       \
    __asm__ __volatile__                                    \
    (                                                       \
        "sts %1, %2\n\t"                                   \
        "lpm %0, Z\n\t"                                    \
        : "=r" (__result)                                   \
        : "i" (_SFR_MEM_ADDR(__SPM_REG)),                   \
          "r" ((uint8_t)(__BOOT_LOCK_BITS_SET)),            \
          "z" ((uint16_t)(address))                         \
    );                                                      \
    __result;                                               \
}))
```

Read the lock or fuse bits at `address`.

Parameter `address` can be any of GET_LOW_FUSE_BITS, GET_LOCK_BITS, GET_EXTENDED_FUSE_BITS, or GET_HIGH_FUSE_BITS.

Note

The lock and fuse bits returned are the physical values, i.e. a bit returned as 0 means the corresponding fuse or lock bit is programmed.

20.14.2.5 boot_page_erase

```
#define boot_page_erase(
              address ) __boot_page_erase_normal(address)
```

Erase the flash page that contains address.

Note

address is a byte address in flash, not a word address.

20.14.2.6 boot_page_erase_safe

```
#define boot_page_erase_safe(
              address )
```

Value:

```
do { \
    boot_spm_busy_wait();                       \
    eeprom_busy_wait();                         \
    boot_page_erase (address);                  \
} while (0)
```

Same as boot_page_erase() except it waits for eeprom and spm operations to complete before erasing the page.

20.14.2.7 boot_page_fill

```
#define boot_page_fill(
              address,
              data ) __boot_page_fill_normal(address, data)
```

Fill the bootloader temporary page buffer for flash address with data word.

Note

The address is a byte address. The data is a word. The AVR writes data to the buffer a word at a time, but addresses the buffer per byte! So, increment your address by 2 between calls, and send 2 data bytes in a word format! The LSB of the data is written to the lower address; the MSB of the data is written to the higher address.

20.14.2.8 boot_page_fill_safe

```
#define boot_page_fill_safe(
              address,
              data )
```

Value:

```
do { \
    boot_spm_busy_wait();                       \
    eeprom_busy_wait();                         \
    boot_page_fill(address, data);              \
} while (0)
```

Same as boot_page_fill() except it waits for eeprom and spm operations to complete before filling the page.

20.14.2.9 boot_page_write

```
#define boot_page_write(
              address ) __boot_page_write_normal(address)
```

Write the bootloader temporary page buffer to flash page that contains address.

Note

address is a byte address in flash, not a word address.

20.14.2.10 boot_page_write_safe

```
#define boot_page_write_safe(
              address )
```

Value:

```
do { \
    boot_spm_busy_wait();                       \
    eeprom_busy_wait();                         \
    boot_page_write (address);                  \
} while (0)
```

Same as boot_page_write() except it waits for eeprom and spm operations to complete before writing the page.

20.14.2.11 boot_rww_busy

```
#define boot_rww_busy( ) (__SPM_REG & (uint8_t)_BV(__COMMON_ASB))
```

Check if the RWW section is busy.

20.14.2.12 boot_rww_enable

```
#define boot_rww_enable( ) __boot_rww_enable()
```

Enable the Read-While-Write memory section.

20.14.2.13 boot_rww_enable_safe

```
#define boot_rww_enable_safe( )
```

Value:

```
do { \
    boot_spm_busy_wait();                       \
    eeprom_busy_wait();                         \
    boot_rww_enable();                          \
} while (0)
```

Same as boot_rww_enable() except waits for eeprom and spm operations to complete before enabling the RWW mameory.

20.14.2.14 boot_signature_byte_get

```
#define boot_signature_byte_get(
              addr )
```

Value:

```
  (({                                 \
      uint8_t __result;                          \
      __asm__ __volatile__                       \
      (                                          \
        "sts %1, %2"   "\n\t"                    \
        "lpm %0, Z"                              \
        : "=r" (__result)                        \
        : "i" (_SFR_MEM_ADDR(__SPM_REG)),        \
          "r" ((uint8_t)(__BOOT_SIGROW_READ)),   \
          "z" ((uint16_t)(addr))                 \
      );                                         \
      __result;                                  \
}))
```

Read the Signature Row byte at `address`. For some MCU types, this function can also retrieve the factory-stored oscillator calibration bytes.

Parameter `address` can be 0-0x1f as documented by the datasheet.

Note

The values are MCU type dependent.

20.14.2.15 boot_spm_busy

```
#define boot_spm_busy( ) (__SPM_REG & (uint8_t)_BV(__SPM_ENABLE))
```

Check if the SPM instruction is busy.

20.14.2.16 boot_spm_busy_wait

```
#define boot_spm_busy_wait( ) do{}while(boot_spm_busy())
```

Wait while the SPM instruction is busy.

20.14.2.17 boot_spm_interrupt_disable

```
#define boot_spm_interrupt_disable( ) (__SPM_REG &= (uint8_t)~_BV(SPMIE))
```

Disable the SPM interrupt.

20.14.2.18 boot_spm_interrupt_enable

```
#define boot_spm_interrupt_enable( ) (__SPM_REG |= (uint8_t)_BV(SPMIE))
```

Enable the SPM interrupt.

20.14.2.19 BOOTLOADER_SECTION

```
#define BOOTLOADER_SECTION __attribute__ ((__section__(".bootloader")))
```

Used to declare a function or variable to be placed into a new section called .bootloader. This section and its contents can then be relocated to any address (such as the bootloader NRWW area) at link-time.

20.14.2.20 GET_EXTENDED_FUSE_BITS

```
#define GET_EXTENDED_FUSE_BITS (0x0002)
```

address to read the extended fuse bits, using boot_lock_fuse_bits_get

20.14.2.21 GET_HIGH_FUSE_BITS

```
#define GET_HIGH_FUSE_BITS (0x0003)
```

address to read the high fuse bits, using boot_lock_fuse_bits_get

20.14.2.22 GET_LOCK_BITS

```
#define GET_LOCK_BITS (0x0001)
```

address to read the lock bits, using boot_lock_fuse_bits_get

20.14.2.23 GET_LOW_FUSE_BITS

```
#define GET_LOW_FUSE_BITS (0x0000)
```

address to read the low fuse bits, using boot_lock_fuse_bits_get

20.15 <avr/cpufunc.h>: Special AVR CPU functions

Macros

- #define _NOP() __asm__ __volatile__("nop")
- #define _MemoryBarrier() __asm__ __volatile__("" ::: "memory")

Functions

- void ccp_write_io (volatile void *__ioaddr, uint8_t __value)
- void ccp_write_spm (volatile void *__ioaddr, uint8_t __value)

20.15.1 Detailed Description

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

This header file contains macros that access special functions of the AVR CPU which do not fit into any of the other header files.

20.15.2 Macro Definition Documentation

20.15.2.1 _MemoryBarrier

```
#define _MemoryBarrier( ) __asm__ __volatile__("" ::: "memory")
```

Implement a read/write *memory barrier*. A memory barrier instructs the compiler to not cache any memory data in registers beyond the barrier. This can sometimes be more effective than blocking certain optimizations by declaring some object with a `volatile` qualifier.

See Problems with Reordering Code for things to be taken into account with respect to compiler optimizations.

20.15.2.2 _NOP

```
#define _NOP( ) __asm__ __volatile__("nop")
```

Execute a *no operation* (NOP) CPU instruction. This should not be used to implement delays, better use the functions from <util/delay_basic.h> or <util/delay.h> for this. For debugging purposes, a NOP can be useful to have an instruction that is guaranteed to be not optimized away by the compiler, so it can always become a breakpoint in the debugger.

20.15.3 Function Documentation

20.15.3.1 ccp_write_io()

```
void ccp_write_io (
            volatile void * __ioaddr,
            uint8_t __value )
```

Write *__value* to IO Register Protected (CCP) 8-bit IO register at *__ioaddr*. See also `_PROTECTED_WRITE()`.

20.15.3.2 ccp_write_spm()

```
void ccp_write_spm (
            volatile void * __ioaddr,
            uint8_t __value )
```

Write *__value* to SPM Instruction Protected (CCP) 8-bit IO register at *__ioaddr*. See also `_PROTECTED_WRITE_SPM()`.

20.16 <avr/eeprom.h>: EEPROM handling

Macros

- #define EEMEM __attribute__((__section__(".eeprom")))
- #define eeprom_is_ready()
- #define eeprom_busy_wait() do {} while (!eeprom_is_ready())

EEPROM Read Functions

- uint8_t eeprom_read_byte (const uint8_t *__p)
- char eeprom_read_char (const char *__p)
- uint8_t eeprom_read_u8 (const uint8_t *__p)
- int8_t eeprom_read_i8 (const int8_t *__p)
- uint16_t eeprom_read_word (const uint16_t *__p)
- uint16_t eeprom_read_u16 (const uint16_t *__p)
- int16_t eeprom_read_i16 (const int16_t *__p)
- uint24_t eeprom_read_u24 (const uint24_t *__p)
- int24_t eeprom_read_i24 (const int24_t *__p)
- uint32_t eeprom_read_dword (const uint32_t *__p)
- uint32_t eeprom_read_u32 (const uint32_t *__p)
- int32_t eeprom_read_i32 (const int32_t *__p)
- uint64_t eeprom_read_qword (const uint64_t *__p)
- uint64_t eeprom_read_u64 (const uint64_t *__p)
- int64_t eeprom_read_i64 (const int64_t *__p)
- float eeprom_read_float (const float *__p)
- double eeprom_read_double (const double *__p)
- long double eeprom_read_long_double (const long double *__p)
- void eeprom_read_block (void *__dst, const void *__src, size_t __n)

EEPROM Write Functions

- void eeprom_write_byte (uint8_t *__p, uint8_t __value)
- void eeprom_write_char (char *__p, char __value)
- void eeprom_write_u8 (uint8_t *__p, uint8_t __value)
- void eeprom_write_i8 (int8_t *__p, int8_t __value)
- void eeprom_write_word (uint16_t *__p, uint16_t __value)
- void eeprom_write_u16 (uint16_t *__p, uint16_t __value)
- void eeprom_write_i16 (int16_t *__p, int16_t __value)
- void eeprom_write_u24 (uint24_t *__p, uint24_t __value)
- void eeprom_write_i24 (int24_t *__p, int24_t __value)
- void eeprom_write_dword (uint32_t *__p, uint32_t __value)
- void eeprom_write_u32 (uint32_t *__p, uint32_t __value)
- void eeprom_write_i32 (int32_t *__p, int32_t __value)
- void eeprom_write_qword (uint64_t *__p, uint64_t __value)
- void eeprom_write_u64 (uint64_t *__p, uint64_t __value)
- void eeprom_write_i64 (int64_t *__p, int64_t __value)
- void eeprom_write_float (float *__p, float __value)
- void eeprom_write_double (double *__p, double __value)
- void eeprom_write_long_double (long double *__p, long double __value)
- void eeprom_write_block (const void *__src, void *__dst, size_t __n)

EEPROM Update Functions

- void eeprom_update_byte (uint8_t *__p, uint8_t __value)
- void eeprom_update_char (char *__p, char __value)
- void eeprom_update_u8 (uint8_t *__p, uint8_t __value)
- void eeprom_update_i8 (int8_t *__p, int8_t __value)
- void eeprom_update_word (uint16_t *__p, uint16_t __value)
- void eeprom_update_u16 (uint16_t *__p, uint16_t __value)
- void eeprom_update_i16 (int16_t *__p, int16_t __value)
- void eeprom_update_u24 (uint24_t *__p, uint24_t __value)
- void eeprom_update_i24 (int24_t *__p, int24_t __value)
- void eeprom_update_dword (uint32_t *__p, uint32_t __value)
- void eeprom_update_u32 (uint32_t *__p, uint32_t __value)
- void eeprom_update_i32 (int32_t *__p, int32_t __value)
- void eeprom_update_qword (uint64_t *__p, uint64_t __value)
- void eeprom_update_u64 (uint64_t *__p, uint64_t __value)
- void eeprom_update_i64 (int64_t *__p, int64_t __value)
- void eeprom_update_float (float *__p, float __value)
- void eeprom_update_double (double *__p, double __value)
- void eeprom_update_long_double (long double *__p, long double __value)
- void eeprom_update_block (const void *__src, void *__dst, size_t __n)

IAR C Compatibility Defines

- #define _EEPUT(addr, val) eeprom_write_byte ((uint8_t *)(addr), (uint8_t)(val))
- #define __EEPUT(addr, val) eeprom_write_byte ((uint8_t *)(addr), (uint8_t)(val))
- #define _EEGET(var, addr) (var) = eeprom_read_byte ((const uint8_t *)(addr))
- #define __EEGET(var, addr) (var) = eeprom_read_byte ((const uint8_t *)(addr))

20.16.1 Detailed Description

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

This header file declares the interface to some simple library routines suitable for handling the data EEPROM contained in the AVR microcontrollers. The implementation uses a simple polled mode interface. Applications that require interrupt-controlled EEPROM access to ensure that no time will be wasted in spinloops will have to deploy their own implementation.

Notes:

- Date is stores and retrieved in little endian format and with no padding bytes or alignment requirements.
- In addition to the write functions there is a set of update functions. This functions read each byte first and skip the burning if the old value is the same with new. The scanning direction is from high address to low, to obtain quick return in common cases.
- Similar functions for fixed-point types are supplied by <stdfix.h>.
- All of the read/write functions first make sure the EEPROM is ready to be accessed. Since this may cause long delays if a write operation is still pending, time-critical applications should first poll the EEPROM e. g. using eeprom_is_ready() before attempting any actual I/O. But this functions does not wait until SELFPRGEN in SPMCSR becomes zero. Do this manually, if your software contains the Flash burning.
- As these functions modify IO registers, they are known to be non-reentrant. If any of these functions are used from both, standard and interrupt context, the applications must ensure proper protection (e.g. by disabling interrupts before accessing them).
- All write functions force erase_and_write programming mode.
- For Xmega the EEPROM start address is 0, like other architectures. The reading functions add the 0x2000 value to use EEPROM mapping into data space.

20.16.2 Macro Definition Documentation

20.16.2.1 __EEGET

```
#define __EEGET(
              var,
              addr ) (var) = eeprom_read_byte ((const uint8_t *)(addr))
```

Read a byte from EEPROM. Compatibility define for IAR C.

20.16.2.2 __EEPUT

```
#define __EEPUT(
              addr,
              val ) eeprom_write_byte ((uint8_t *)(addr), (uint8_t)(val))
```

Write a byte to EEPROM. Compatibility define for IAR C.

20.16.2.3 _EEGET

```
#define _EEGET(
              var,
              addr ) (var) = eeprom_read_byte ((const uint8_t *)(addr))
```

Read a byte from EEPROM. Compatibility define for IAR C.

20.16.2.4 _EEPUT

```
#define _EEPUT(
              addr,
              val ) eeprom_write_byte ((uint8_t *)(addr), (uint8_t)(val))
```

Write a byte to EEPROM. Compatibility define for IAR C.

20.16.2.5 EEMEM

```
#define EEMEM __attribute__((__section__(".eeprom")))
```

Attribute expression causing a variable to be allocated within the .eeprom section.

20.16.2.6 eeprom_busy_wait

```
#define eeprom_busy_wait( ) do {} while (!eeprom_is_ready())
```

Loops until the eeprom is no longer busy.

Returns

Nothing.

20.16.2.7 eeprom_is_ready

```
#define eeprom_is_ready( )
```

Returns

1 if EEPROM is ready for a new read/write operation, 0 if not.

20.16.3 Function Documentation

20.16.3.1 eeprom_read_block()

```
void eeprom_read_block (
             void * __dst,
             const void * __src,
             size_t __n )
```

Read a block of *__n* bytes from EEPROM address *__src* to SRAM *__dst*.

20.16.3.2 eeprom_read_byte()

```
uint8_t eeprom_read_byte (
             const uint8_t * __p )
```

Read one byte from EEPROM address *__p*.

20.16.3.3 eeprom_read_char()

```
char eeprom_read_char (
             const char * __p )
```

Read a char from EEPROM address *__p*.

Since

AVR-LibC v2.3

20.16.3.4 eeprom_read_double()

```
double eeprom_read_double (
             const double * __p )
```

Read one double value from EEPROM address *__p*.

Since

AVR-LibC v2.2

20.16.3.5 eeprom_read_dword()

```
uint32_t eeprom_read_dword (
            const uint32_t * __p )
```

Read one 32-bit double word from EEPROM address *__p*.

20.16.3.6 eeprom_read_float()

```
float eeprom_read_float (
            const float * __p )
```

Read one float value from EEPROM address *__p*.

20.16.3.7 eeprom_read_i16()

```
int16_t eeprom_read_i16 (
            const int16_t * __p )
```

Read a signed 16-bit integer from EEPROM address *__p*.

Since

AVR-LibC v2.3

20.16.3.8 eeprom_read_i24()

```
int24_t eeprom_read_i24 (
            const int24_t * __p )
```

Read a signed 24-bit integer from EEPROM address *__p*.

Since

AVR-LibC v2.3

20.16.3.9 eeprom_read_i32()

```
int32_t eeprom_read_i32 (
            const int32_t * __p )
```

Read a signed 32-bit integer from EEPROM address *__p*.

Since

AVR-LibC v2.3

20.16.3.10 eeprom_read_i64()

```
int64_t eeprom_read_i64 (
            const int64_t * __p )
```

Read a signed 64-bit integer from EEPROM address *__p*.

Since

AVR-LibC v2.3

20.16.3.11 eeprom_read_i8()

```
int8_t eeprom_read_i8 (
            const int8_t * __p )
```

Read a signed 8-bit integer from EEPROM address *__p*.

Since

AVR-LibC v2.3

20.16.3.12 eeprom_read_long_double()

```
long double eeprom_read_long_double (
            const long double * __p )
```

Read one long double value from EEPROM address *__p*.

Since

AVR-LibC v2.2

20.16.3.13 eeprom_read_qword()

```
uint64_t eeprom_read_qword (
            const uint64_t * __p )
```

Read one 64-bit quad word from EEPROM address *__p*.

Since

AVR-LibC v2.2

20.16.3.14 eeprom_read_u16()

```
uint16_t eeprom_read_u16 (
            const uint16_t * __p )
```

Read an unsigned 16-bit integer from EEPROM address *__p*.

Since

AVR-LibC v2.3

20.16.3.15 eeprom_read_u24()

```
uint24_t eeprom_read_u24 (
            const uint24_t * __p )
```

Read an unsigned 24-bit integer from EEPROM address *__p*.

Since

AVR-LibC v2.3

20.16.3.16 eeprom_read_u32()

```
uint32_t eeprom_read_u32 (
            const uint32_t * __p )
```

Read an unsigned 32-bit integer from EEPROM address *__p*.

Since

AVR-LibC v2.3

20.16.3.17 eeprom_read_u64()

```
uint64_t eeprom_read_u64 (
            const uint64_t * __p )
```

Read an unsigned 64-bit integer from EEPROM address *__p*.

Since

AVR-LibC v2.3

20.16.3.18 eeprom_read_u8()

```
uint8_t eeprom_read_u8 (
            const uint8_t * __p )
```

Read an unsigned 8-bit integer from EEPROM address *__p*.

Since

AVR-LibC v2.3

20.16.3.19 eeprom_read_word()

```
uint16_t eeprom_read_word (
            const uint16_t * __p )
```

Read one 16-bit word from EEPROM address *__p*.

20.16.3.20 eeprom_update_block()

```
void eeprom_update_block (
            const void * __src,
            void * __dst,
            size_t __n )
```

Update a block of *__n* bytes at EEPROM address *__dst* from *__src*.

Note

The argument order is mismatch with common functions like strcpy().

20.16.3.21 eeprom_update_byte()

```
void eeprom_update_byte (
            uint8_t * __p,
            uint8_t __value )
```

Update a byte *__value* at EEPROM address *__p*.

20.16.3.22 eeprom_update_char()

```
void eeprom_update_char (
            char * __p,
            char __value )
```

Update a char *at* EEPROM address *__p*.

Since

AVR-LibC v2.3

20.16.3.23 eeprom_update_double()

```
void eeprom_update_double (
             double * __p,
             double __value )
```

Update a double *__value* at EEPROM address *__p*.

Since

AVR-LibC v2.2

20.16.3.24 eeprom_update_dword()

```
void eeprom_update_dword (
             uint32_t * __p,
             uint32_t __value )
```

Update a 32-bit double word *__value* at EEPROM address *__p*.

20.16.3.25 eeprom_update_float()

```
void eeprom_update_float (
             float * __p,
             float __value )
```

Update a float *__value* at EEPROM address *__p*.

20.16.3.26 eeprom_update_i16()

```
void eeprom_update_i16 (
             int16_t * __p,
             int16_t __value )
```

Update a signed 16-bit integer *at* EEPROM address *__p*.

Since

AVR-LibC v2.3

20.16.3.27 eeprom_update_i24()

```
void eeprom_update_i24 (
             int24_t * __p,
             int24_t __value )
```

Update a signed 24-bit integer *at* EEPROM address *__p*.

Since

AVR-LibC v2.3

20.16.3.28 eeprom_update_i32()

```
void eeprom_update_i32 (
             int32_t * __p,
             int32_t __value )
```

Update a signed 32-bit integer *at* EEPROM address *__p*.

Since

AVR-LibC v2.3

20.16.3.29 eeprom_update_i64()

```
void eeprom_update_i64 (
             int64_t * __p,
             int64_t __value )
```

Update a signed 64-bit integer *at* EEPROM address *__p*.

Since

AVR-LibC v2.3

20.16.3.30 eeprom_update_i8()

```
void eeprom_update_i8 (
             int8_t * __p,
             int8_t __value )
```

Update a signed 8-bit integer *at* EEPROM address *__p*.

Since

AVR-LibC v2.3

20.16.3.31 eeprom_update_long_double()

```
void eeprom_update_long_double (
             long double * __p,
             long double __value )
```

Update a long double *__value* at EEPROM address *__p*.

Since

AVR-LibC v2.2

20.16.3.32 eeprom_update_qword()

```
void eeprom_update_qword (
             uint64_t * __p,
             uint64_t __value )
```

Update a 64-bit quad word *__value* at EEPROM address *__p*.

Since

AVR-LibC v2.2

20.16.3.33 eeprom_update_u16()

```
void eeprom_update_u16 (
             uint16_t * __p,
             uint16_t __value )
```

Update an unsigned 16-bit integer *at* EEPROM address *__p*.

Since

AVR-LibC v2.3

20.16.3.34 eeprom_update_u24()

```
void eeprom_update_u24 (
             uint24_t * __p,
             uint24_t __value )
```

Update an unsigned 24-bit integer *at* EEPROM address *__p*.

Since

AVR-LibC v2.3

20.16.3.35 eeprom_update_u32()

```
void eeprom_update_u32 (
             uint32_t * __p,
             uint32_t __value )
```

Update an unsigned 32-bit integer *at* EEPROM address *__p*.

Since

AVR-LibC v2.3

20.16.3.36 eeprom_update_u64()

```
void eeprom_update_u64 (
            uint64_t * __p,
            uint64_t __value )
```

Update an unsigned 64-bit integer *at* EEPROM address *__p*.

Since

AVR-LibC v2.3

20.16.3.37 eeprom_update_u8()

```
void eeprom_update_u8 (
            uint8_t * __p,
            uint8_t __value )
```

Update an unsigned 8-bit integer *at* EEPROM address *__p*.

Since

AVR-LibC v2.3

20.16.3.38 eeprom_update_word()

```
void eeprom_update_word (
            uint16_t * __p,
            uint16_t __value )
```

Update a word *__value* at EEPROM address *__p*.

20.16.3.39 eeprom_write_block()

```
void eeprom_write_block (
            const void * __src,
            void * __dst,
            size_t __n )
```

Write a block of *__n* bytes to EEPROM address *__dst* from *__src*.

Note

The argument order is mismatch with common functions like strcpy().

20.16.3.40 eeprom_write_byte()

```
void eeprom_write_byte (
             uint8_t * __p,
             uint8_t __value )
```

Write a byte *__value* to EEPROM address *__p*.

20.16.3.41 eeprom_write_char()

```
void eeprom_write_char (
             char * __p,
             char __value )
```

Write a char to EEPROM address *__p*.

Since

AVR-LibC v2.3

20.16.3.42 eeprom_write_double()

```
void eeprom_write_double (
             double * __p,
             double __value )
```

Write a double *__value* to EEPROM address *__p*.

Since

AVR-LibC v2.2

20.16.3.43 eeprom_write_dword()

```
void eeprom_write_dword (
             uint32_t * __p,
             uint32_t __value )
```

Write a 32-bit double word *__value* to EEPROM address *__p*.

20.16.3.44 eeprom_write_float()

```
void eeprom_write_float (
             float * __p,
             float __value )
```

Write a float *__value* to EEPROM address *__p*.

20.16.3.45 eeprom_write_i16()

```
void eeprom_write_i16 (
             int16_t * __p,
             int16_t __value )
```

Write a signed 16-bit integer to EEPROM address *__p*.

Since

AVR-LibC v2.3

20.16.3.46 eeprom_write_i24()

```
void eeprom_write_i24 (
             int24_t * __p,
             int24_t __value )
```

Write a signed 24-bit integer to EEPROM address *__p*.

Since

AVR-LibC v2.3

20.16.3.47 eeprom_write_i32()

```
void eeprom_write_i32 (
             int32_t * __p,
             int32_t __value )
```

Write a signed 32-bit integer to EEPROM address *__p*.

Since

AVR-LibC v2.3

20.16.3.48 eeprom_write_i64()

```
void eeprom_write_i64 (
             int64_t * __p,
             int64_t __value )
```

Write a signed 64-bit integer to EEPROM address *__p*.

Since

AVR-LibC v2.3

20.16.3.49 eeprom_write_i8()

```
void eeprom_write_i8 (
             int8_t * __p,
             int8_t __value )
```

Write a signed 8-bit integer to EEPROM address *__p*.

Since

AVR-LibC v2.3

20.16.3.50 eeprom_write_long_double()

```
void eeprom_write_long_double (
             long double * __p,
             long double __value )
```

Write a long double *__value* to EEPROM address *__p*.

Since

AVR-LibC v2.2

20.16.3.51 eeprom_write_qword()

```
void eeprom_write_qword (
             uint64_t * __p,
             uint64_t __value )
```

Write a 64-bit quad word *__value* to EEPROM address *__p*.

Since

AVR-LibC v2.2

20.16.3.52 eeprom_write_u16()

```
void eeprom_write_u16 (
             uint16_t * __p,
             uint16_t __value )
```

Write an unsigned 16-bit integer to EEPROM address *__p*.

Since

AVR-LibC v2.3

20.16.3.53 eeprom_write_u24()

```
void eeprom_write_u24 (
             uint24_t * __p,
             uint24_t __value )
```

Write an unsigned 24-bit integer to EEPROM address *__p*.

Since

AVR-LibC v2.3

20.16.3.54 eeprom_write_u32()

```
void eeprom_write_u32 (
             uint32_t * __p,
             uint32_t __value )
```

Write an unsigned 32-bit integer to EEPROM address *__p*.

Since

AVR-LibC v2.3

20.16.3.55 eeprom_write_u64()

```
void eeprom_write_u64 (
             uint64_t * __p,
             uint64_t __value )
```

Write an unsigned 64-bit integer to EEPROM address *__p*.

Since

AVR-LibC v2.3

20.16.3.56 eeprom_write_u8()

```
void eeprom_write_u8 (
             uint8_t * __p,
             uint8_t __value )
```

Write an unsigned 8-bit integer to EEPROM address *__p*.

Since

AVR-LibC v2.3

20.16.3.57 eeprom_write_word()

```
void eeprom_write_word (
             uint16_t * __p,
             uint16_t __value )
```

Write a word *__value* to EEPROM address *__p*.

20.17 <avr/flash.h>: Utilities for named address-spaces __flash and __flashx

AVR Named Address-Spaces

- __flash
- __flashx
- __memx

Functions from stdio.h, but with a format string in address-space __flash

- int vfprintf_F (FILE *stream, const __flash char *fmt, va_list ap)
- int printf_F (const __flash char *fmt,...)
- int sprintf_F (char *s, const __flash char *fmt,...)
- int snprintf_F (char *s, size_t n, const __flash char *fmt,...)
- int vsprintf_F (char *s, const __flash char *fmt, va_list ap)
- int vsnprintf_F (char *s, size_t n, const __flash char *fmt, va_list ap)
- int fprintf_F (FILE *stream, const __flash char *fmt,...)
- int fputs_F (const __flash char *str, FILE *stream)
- int puts_F (const __flash char *str)
- int vfscanf_F (FILE *stream, const __flash char *fmt, va_list ap)
- int fscanf_F (FILE *stream, const __flash char *fmt,...)
- int scanf_F (const __flash char *fmt,...)
- int sscanf_F (const char *buf, const __flash char *fmt,...)

Functions from string.h, but one argument is in address-space __flash

- const __flash void * memchr_F (const __flash void *, int, size_t)
- int memcmp_F (const void *, const __flash void *, size_t)
- void * memccpy_F (void *, const __flash void *, int val, size_t)
- void * memcpy_F (void *, const __flash void *, size_t)
- void * memmem_F (const void *, size_t, const __flash void *, size_t)
- const __flash void * memrchr_F (const __flash void *, int val, size_t len)
- static size_t strlen_F (const __flash char *src)
- char * strcat_F (char *, const __flash char *)
- const __flash char * strchr_F (const __flash char *, int val)
- const __flash char * strchrnul_F (const __flash char *, int val)
- int strcmp_F (const char *, const __flash char *)
- char * strcpy_F (char *, const __flash char *)
- char * stpcpy_F (char *, const __flash char *)
- int strcasecmp_F (const char *, const __flash char *)
- char * strcasestr_F (const char *, const __flash char *)
- size_t strcspn_F (const char *s, const __flash char *reject)
- size_t strlcat_F (char *, const __flash char *, size_t)
- size_t strlcpy_F (char *, const __flash char *, size_t)

- size_t strnlen_F (const __flash char *, size_t)
- int strncmp_F (const char *, const __flash char *, size_t)
- int strncasecmp_F (const char *, const __flash char *, size_t)
- char * strncat_F (char *, const __flash char *, size_t)
- char * strncpy_F (char *, const __flash char *, size_t)
- char * strpbrk_F (const char *, const __flash char *accept)
- const __flash char * strrchr_F (const __flash char *, int val)
- char * strsep_F (char **sp, const __flash char *delim)
- size_t strspn_F (const char *s, const __flash char *accept)
- char * strstr_F (const char *, const __flash char *)
- char * strtok_F (char *s, const __flash char *delim)
- char * strtok_rF (char *s, const __flash char *delim, char **last)

Functions from string.h, but one argument is in 24-bit address-space __flashx

- void * memcpy_FX (void *dest, const __flashx void *src, size_t len)
- int memcmp_FX (const void *s1, const __flashx void *s2, size_t)
- size_t strlen_FX (const __flashx char *src)
- size_t strnlen_FX (const __flashx char *src, size_t len)
- char * strcpy_FX (char *dest, const __flashx char *src)
- char * stpcpy_FX (char *dest, const __flashx char *src)
- char * strncpy_FX (char *dest, const __flashx char *src, size_t len)
- char * strcat_FX (char *dest, const __flashx char *src)
- size_t strlcat_FX (char *dst, const __flashx char *src, size_t siz)
- char * strncat_FX (char *dest, const __flashx char *src, size_t len)
- int strcmp_FX (const char *s1, const __flashx char *s2)
- int strncmp_FX (const char *s1, const __flashx char *s2, size_t n)
- int strcasecmp_FX (const char *s1, const __flashx char *s2)
- int strncasecmp_FX (const char *s1, const __flashx char *s2, size_t n)
- const __flashx char * strchr_FX (const __flashx char *s, int val)
- char * strstr_FX (const char *s1, const __flashx char *s2)
- size_t strlcpy_FX (char *, const __flashx char *, size_t)

More efficient reading of 64-bit values from __flash and __flashx

- static uint64_t flash_read_u64 (const __flash uint64_t *addr)
- static int64_t flash_read_i64 (const __flash int64_t *addr)
- static double flash_read_double (const __flash double *addr)
- static long double flash_read_long_double (const __flash long double *addr)
- static uint64_t flashx_read_u64 (const __flashx uint64_t *addr)
- static int64_t flashx_read_i64 (const __flashx int64_t *addr)
- static double flashx_read_double (const __flashx double *addr)
- static long double flashx_read_long_double (const __flashx long double *addr)

Macros

- #define FSTR(str) ({ static const __flash char c[ ] = (str); &c[0]; })
- #define FXSTR(str) ({ static const __flashx char c[ ] = (str); &c[0]; })
- #define FLIT(str) ((const __flash char[ ]) { str })
- #define FXLIT(str) ((const __flashx char[ ]) { str })

Convenience macros for functions from stdio.h, that allocate the format string with FSTR

- #define vfprintf_FSTR(stream, fmt, ap) vfprintf_F(stream, FSTR(fmt), ap)
- #define printf_FSTR(fmt, ...) printf_F(FSTR(fmt), ##__VA_ARGS__)
- #define sprintf_FSTR(s, fmt, ...) sprintf_F(s, FSTR(fmt), ##__VA_ARGS__)
- #define snprintf_FSTR(s, n, fmt, ...) snprintf_F(s, n, FSTR(fmt), ##__VA_ARGS__)
- #define vsprintf_FSTR(s, fmt, ap) vsprintf_F(s, FSTR(fmt), ap)
- #define vsnprintf_FSTR(s, n, fmt, ap) vsnprintf_F(s, n, FSTR(fmt), ap);
- #define fprintf_FSTR(stream, fmt, ...) fprintf_F(stream, FSTR(fmt), ##__VA_ARGS__)
- #define fputs_FSTR(str, stream) fputs_F(FSTR(str), stream);
- #define puts_FSTR(str) puts_F(FSTR(str));
- #define vfscanf_FSTR(stream, fmt, ap) vfscanf_F(stream, FSTR(fmt), ap);
- #define fscanf_FSTR(stream, fmt, ...) fscanf_F(stream, FSTR(fmt), ##__VA_ARGS__)
- #define scanf_FSTR(fmt, ...) scanf_F(FSTR(fmt), ##__VA_ARGS__)
- #define sscanf_FSTR(buf, fmt, ...) sscanf_F(buf, FSTR(fmt), ##__VA_ARGS__)

20.17.1 Detailed Description

Since

AVR-LibC v2.3

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

The functions and macros in this module provide interfaces for a program to use data stored in program space (flash memory) by means of the __flash and __flashx named address-spaces supported by avr-gcc.

Purpose The prototypes and macros provided by this header allow to write C programs that are address-space correct, i.e. they will compile without diagnostics due to `-Waddr-space-convert`.

For example, you can call printf_P with a format string located in RAM resulting in non-functional code, and the compiler won't complain. This is different with printf_F which will trigger a diagnostic when `-Waddr-space-convert` is on, and you feed in a format string that does not carry the __flash named address-space qualifier.

Structure of this Header This header provides:

- Functions from `stdio.h`, `string.h` resp. `avr/pgmspace.h`, but where one input string resides in the 16-bit address-space __flash. For example, the strlen_F function works the same like the "progmem" function strlen_P but uses a prototype that describes the flash string as `const __flash char*`.
- Functions from `string.h` resp. `avr/pgmspace.h`, but where one input string resides in 24-bit address-space __flashx. For example, the strlen_FX function works the same like the "far" function strlen_PF but uses a prototype that describes the flash string as `const __flashx char*`.
- Some macros for convenience.

Examples

```
#include <stdbool.h>
#include <avr/flash.h>

// Array of string literals where the array
// and also the literals reside in __flash.
const __flash char* const __flash pets[] =
{
    FLIT("gnu"), FLIT("cat"), FLIT("bat"), FLIT("rat")
};

void test_pet (const char *pet, const __flash char *what)
{
    const __flash char *yesno;
    bool is_what = ! strcmp_F (pet, what);
    yesno = is_what ? FSTR("yes") : FSTR("no");

    // %S denotes a string in lower 64 KiB flash.
    printf_FSTR ("%s is a %S? %S!\n", pet, what, yesno);
}

int main (void)
{
    char pet[] = "cat";
    for (size_t i = 0; i < sizeof (pets) / sizeof (*pets); ++i)
    {
        pet[0] ^= 1;
        test_pet (pet, pets[i]);
    }
  return 0;
}
```

It will print

```
bat is a gnu? no!
cat is a cat? yes!
bat is a bat? yes!
cat is a rat? no!
```

provided stdout has been set up appropriately.

Efficiency The internal handling of 64-bit values in avr-gcc is such that it splits them into single byte operations for the purpose of moving them around. This can lead to quite some overhead when reading such values from named address-spaces. To that end, `avr/flash.h` provides some inline functions like flash_read_u64 and flashx_read_double that work similar to the pgm_read_u64 and pgm_read_double_far functions, but are coming with proper address-space qualification.

Limitations

- Named address-spaces are supported by avr-gcc as part of GNU-C (e.g. `-std=gnu99`). They are not available in Standard C, and are not supported in C++.
- Address-spaces __flash and __memx are supported since avr-gcc v4.7 (Release 2012), whereas __flashx is available since avr-gcc v15 (Release 2025).
- Named address-spaces are not supported for Reduced Tiny (core AVRrc, `-mmcu=avrtiny`), like ATtiny10, ATtiny102 or ATtiny40.

Further Reading

- avr-gcc: Named Address-Spaces

20.17.2 Macro Definition Documentation

20.17.2.1 FLIT

```
#define FLIT(
              str ) ((const __flash char[]) { str })
```

Turn string literal `str` into a compound literal in address-space __flash. This macro can be used to construct arrays of string pointers: Suppose the following 2-dimensional array of animal names:

```
// Each string occupies 13 bytes, hence
// animals[] occupies 9 * 13 = 117 bytes.
const __flash char animals[][13] =
{
    "hippopotamus",
    "cat", "pig", "gnu", "bat",
    "dog", "cow", "fox", "rat"
};
```

A more memory-friendly way to represent the strings is an array of string *pointers*, like in:

```
// Occupies 9*2 + 13 + 8*4 = 63 bytes
const char* const animals2[] =
{
    "hippopotamus",
    "cat", "pig", "gnu", "bat",
    "dog", "cow", "fox", "rat"
};
```

`animals2`[] occupies only 63 bytes (9 $*$ 2 bytes for the string addresses plus the lengths of the very strings). However, all objects are located in the generic address-space. When all objects should be put in a non-generic address-space like __flash, then the type of the literals has to be forced to be `const __flash char[]`. This can be accomplished with FLIT:

```
// Occupies 9*2 + 13 + 8*4 = 63 bytes
const __flash char* const __flash animals3[] =
{
    FLIT("hippopotamus"),
    FLIT("cat"), FLIT("pig"), FLIT("gnu"), FLIT("bat"),
    FLIT("dog"), FLIT("cow"), FLIT("fox"), FLIT("rat")
};
```

Notice the two __flash's in the declarator: The left one says that the pointed-to strings are in __flash, whereas the right one says that the `animals3`[] array itself resides in __flash.

Unfortunately, to date (avr-gcc v15), FLIT can only be used at global scope. Though what works in a function is using `constexpr` as introduced in C23:

```
void func (int i)
{
    static constexpr __flash char s_hip[] = "hippopotamus";
    static constexpr __flash char s_cat[] = "cat";
    // ...
    static const __flash char* const __flash animals4[] =
    {
        s_hip, s_cat, // ...
    };
    printf_FSTR ("Animal %d = %S\n", i, animals4[i]);
}
```

20.17.2.2 fprintf_FSTR

```
#define fprintf_FSTR(
              stream,
              fmt,
              ...  ) fprintf_F(stream, FSTR(fmt), ##__VA_ARGS__)
```

A convenience macro that wraps fprintf_F's format string with FSTR.

20.17.2.3 fputs_FSTR

```
#define fputs_FSTR(
              str,
              stream ) fputs_F(FSTR(str), stream);
```

A convenience macro that wraps fputs_F's string with FSTR.

20.17.2.4 fscanf_FSTR

```
#define fscanf_FSTR(
              stream,
              fmt,
              ...  ) fscanf_F(stream, FSTR(fmt), ##__VA_ARGS__)
```

A convenience macro that wraps fscanf_F's format string with FSTR.

20.17.2.5 FSTR

```
#define FSTR(
              str ) ({ static const __flash char c[] = (str); &c[0]; })
```

Used to get a pointer to a static string in address-space __flash. This macro can only be used in the context of a funcion, like in:

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

void say_hello (int x)
{
    printf_F (FSTR ("Hello number %d\n"), x);

    // Same effect, but more convenient.
    printf_FSTR ("Hello number %d\n", x);
}
```

FSTR works similar to the PSTR macro but returns a pointer to the 16-bit named address-space __flash, whereas PSTR returns a 16-bit address in the generic address-space (but isn't).

20.17.2.6 FXLIT

```
#define FXLIT(
              str ) ((const __flashx char[]) { str })
```

Turn string literal `str` into a compound literal in address-space __flashx. This macro can be used to construct arrays of string pointers.

For example code and usage, see the FLIT macro.

20.17.2.7 FXSTR

```
#define FXSTR(
              str ) ({ static const __flashx char c[] = (str); &c[0]; })
```

Used to get a pointer to a static string in the 24-bit address-space __flashx. This macro can only be used in the context of a funcion, like in:

```
#include <stdbool.h>
#include <avr/flash.h>

bool text_contains_dog (const char *text)
{
    return strstr_FX (text, FXSTR ("dog")) != NULL;
}
```

FXSTR works similar to the PSTR_FAR macro but returns a pointer to the 24-bit named address-space __flashx, whereas PSTR_FAR returns a 32-bit integer that represents an address in the generic address-space (but isn't).

20.17.2.8 printf_FSTR

```
#define printf_FSTR(
              fmt,
              ... ) printf_F(FSTR(fmt), ##__VA_ARGS__)
```

A convenience macro that wraps printf_F's format string with FSTR.

20.17.2.9 puts_FSTR

```
#define puts_FSTR(
              str ) puts_F(FSTR(str));
```

A convenience macro that wraps puts_F's string with FSTR.

20.17.2.10 scanf_FSTR

```
#define scanf_FSTR(
              fmt,
              ... ) scanf_F(FSTR(fmt), ##__VA_ARGS__)
```

A convenience macro that wraps scanf_F's format string with FSTR.

20.17.2.11 snprintf_FSTR

```
#define snprintf_FSTR(
              s,
              n,
              fmt,
              ... ) snprintf_F(s, n, FSTR(fmt), ##__VA_ARGS__)
```

A convenience macro that wraps snprintf_F's format string with FSTR.

20.17.2.12 sprintf_FSTR

```
#define sprintf_FSTR(
              s,
              fmt,
              ...  ) sprintf_F(s, FSTR(fmt), ##__VA_ARGS__)
```

A convenience macro that wraps sprintf_F's format string with FSTR.

20.17.2.13 sscanf_FSTR

```
#define sscanf_FSTR(
              buf,
              fmt,
              ...  ) sscanf_F(buf, FSTR(fmt), ##__VA_ARGS__)
```

A convenience macro that wraps sscanf_F's format string with FSTR.

20.17.2.14 vfprintf_FSTR

```
#define vfprintf_FSTR(
              stream,
              fmt,
              ap ) vfprintf_F(stream, FSTR(fmt), ap)
```

A convenience macro that wraps vfprintf_F's format string with FSTR.

20.17.2.15 vfscanf_FSTR

```
#define vfscanf_FSTR(
              stream,
              fmt,
              ap ) vfscanf_F(stream, FSTR(fmt), ap);
```

A convenience macro that wraps vfscanf_F's format string with FSTR.

20.17.2.16 vsnprintf_FSTR

```
#define vsnprintf_FSTR(
              s,
              n,
              fmt,
              ap ) vsnprintf_F(s, n, FSTR(fmt), ap);
```

A convenience macro that wraps vsnprintf_F's format string with FSTR.

20.17.2.17 vsprintf_FSTR

```
#define vsprintf_FSTR(
              s,
              fmt,
              ap ) vsprintf_F(s, FSTR(fmt), ap)
```

A convenience macro that wraps vsprintf_F's format string with FSTR.

20.17.3 Function Documentation

20.17.3.1 flash_read_double()

```
double flash_read_double (
            const __flash double * addr )  [inline], [static]
```

Read a `double` from 16-bit __flash address `addr`.

20.17.3.2 flash_read_i64()

```
int64_t flash_read_i64 (
            const __flash int64_t * addr )  [inline], [static]
```

Read an `int64_t` from 16-bit __flash address `addr`.

20.17.3.3 flash_read_long_double()

```
long double flash_read_long_double (
            const __flash long double * addr )  [inline], [static]
```

Read a `long double` from 16-bit __flash address `addr`.

20.17.3.4 flash_read_u64()

```
uint64_t flash_read_u64 (
            const __flash uint64_t * addr )  [inline], [static]
```

Read an `uint64_t` from 16-bit __flash address `addr`.

20.17.3.5 flashx_read_double()

```
double flashx_read_double (
            const __flashx double * addr )  [inline], [static]
```

Read a `double` from 24-bit __flashx address `addr`.

20.17.3.6 flashx_read_i64()

```
int64_t flashx_read_i64 (
            const __flashx int64_t * addr )  [inline], [static]
```

Read an `int64_t` from 24-bit __flashx address `addr`.

20.17.3.7 flashx_read_long_double()

```
long double flashx_read_long_double (
            const __flashx long double * addr )  [inline], [static]
```

Read a `long double` from 24-bit __flashx address `addr`.

20.17.3.8 flashx_read_u64()

```
uint64_t flashx_read_u64 (
            const __flashx uint64_t * addr )  [inline], [static]
```

Read an `uint64_t` from 24-bit __flashx address `addr`.

20.17.3.9 fprintf_F()

```
int fprintf_F (
            FILE * stream,
            const __flash char * fmt,
             ...  )
```

Variant of `fprintf` that uses a `fmt` string that resides in address-space __flash. See also fprintf_FSTR.

20.17.3.10 fputs_F()

```
int fputs_F (
            const __flash char * str,
            FILE * stream )
```

Variant of fputs where `str` resides in address-space __flash. See also fputs_FSTR.

20.17.3.11 fscanf_F()

```
int fscanf_F (
            FILE * stream,
            const __flash char * fmt,
             ...  )
```

Variant of fscanf using a `fmt` string in address-space __flash. See also fscanf_FSTR.

20.17.3.12 memccpy_F()

```
void * memccpy_F (
            void * dest,
            const __flash void * src,
            int val,
            size_t len )
```

This function is similar to memccpy except that `src` is pointer to a string in address-space __flash.

20.17.3.13 memchr_F()

```
const __flash void * memchr_F (
            const __flash void * s,
            int val,
            size_t len )
```

Scan flash memory for a character.

The memchr_F function scans the first `len` bytes of the flash memory area pointed to by `s` for the character `val`. The first byte to match `val` (interpreted as an unsigned character) stops the operation.

Returns

The memchr_F function returns a pointer to the matching byte or `NULL` if the character does not occur in the given memory area.

20.17.3.14 memcmp_F()

```
int memcmp_F (
            const void * s1,
            const __flash void * s2,
            size_t len )
```

Compare memory areas.

The memcmp_F function compares the first `len` bytes of the memory areas `s1` and flash `s2`. The comparision is performed using unsigned char operations.

Returns

The memcmp_F function returns an integer less than, equal to, or greater than zero if the first `len` bytes of `s1` is found, respectively, to be less than, to match, or be greater than the first `len` bytes of `s2`.

20.17.3.15 memcmp_FX()

```
int memcmp_FX (
            const void * s1,
            const __flashx void * s2,
            size_t len )
```

Compare memory areas.

The memcmp_FX function compares the first `len` bytes of the memory areas `s1` and __flashx `s2`. The comparision is performed using unsigned char operations. It is an equivalent of memcmp_F function, except that it is capable working on all Flash including the extended area above 64 KiB.

Returns

The memcmp_FX function returns an integer less than, equal to, or greater than zero if the first `len` bytes of `s1` is found, respectively, to be less than, to match, or be greater than the first `len` bytes of `s2`.

20.17.3.16 memcpy_F()

```
void * memcpy_F (
            void * dest,
            const __flash void * src,
            size_t n )
```

The memcpy_F function is similar to memcpy, except the src string resides in address-space __flash.

Returns

The memcpy_F function returns a pointer to dest.

20.17.3.17 memcpy_FX()

```
void * memcpy_FX (
            void * dest,
            const __flashx void * src,
            size_t n )
```

Copy a memory block from address-space __flashx to SRAM.

The memcpy_FX function is similar to memcpy, except the data is copied from address-space __flashx and is addressed using an according pointer.

Parameters

dest	A pointer to the destination buffer.
src	A pointer to the origin of data in __flashx.
n	The number of bytes to be copied.

Returns

The memcpy_FX function returns a pointer to *dst*.

20.17.3.18 memmem_F()

```
void * memmem_F (
             const void * s1,
             size_t len1,
             const __flash void * s2,
             size_t len2 )
```

The memmem_F function is similar to memmem except that `s2` is pointer to a string in address-space __flash.

20.17.3.19 memrchr_F()

```
const __flash void * memrchr_F (
             const __flash void * src,
             int val,
             size_t len )
```

The memrchr_F function is like the memchr_F function, except that it searches backwards from the end of the `len` bytes pointed to by `src` instead of forwards from the front. (Glibc, GNU extension.)

Returns

The memrchr_F function returns a pointer to the matching byte or `NULL` if the character does not occur in the given memory area.

20.17.3.20 printf_F()

```
int printf_F (
             const __flash char * fmt,
              ...  )
```

Variant of `printf` that uses a `fmt` string that resides in address-space __flash. See also printf_FSTR.

20.17.3.21 puts_F()

```
int puts_F (
             const __flash char * str )
```

Variant of puts where `str` resides in address-space __flash. See also puts_FSTR.

20.17.3.22 scanf_F()

```
int scanf_F (
             const __flash char * fmt,
              ...  )
```

Variant of scanf where `fmt` resides in address-space __flash. See also scanf_FSTR.

20.17.3.23 snprintf_F()

```
int snprintf_F (
             char * s,
             size_t n,
             const __flash char * fmt,
              ...  )
```

Variant of `snprintf` that uses a `fmt` string that resides in address-space __flash. See also snprintf_FSTR.

20.17.3.24 sprintf_F()

```
int sprintf_F (
             char * s,
             const __flash char * fmt,
              ...  )
```

Variant of `sprintf` that uses a `fmt` string that resides in address-space __flash. See also sprintf_FSTR.

20.17.3.25 sscanf_F()

```
int sscanf_F (
             const char * buf,
             const __flash char * fmt,
              ...  )
```

Variant of sscanf using a `fmt` string in address-space __flash. See also sscanf_FSTR.

20.17.3.26 stpcpy_F()

```
char * stpcpy_F (
             char * dest,
             const __flash char * src )  [inline]
```

The stpcpy_F function is similar to stpcpy except that `src` is a pointer to a string in address-space __flash.

Returns

stpcpy_F returns a pointer to the **end** of the string `dest` (that is, the address of the terminating null byte) rather than the beginning.

20.17.3.27 stpcpy_FX()

```
char * stpcpy_FX (
             char * dst,
             const __flashx char * src )
```

Duplicate a string from address-space __flashx.

The stpcpy_FX function is similar to stpcpy except that *src* is a string located in address-space __flashx.

Parameters

dst	A pointer to the destination string in SRAM.
src	A pointer to the source string in __flashx.

Returns

The stpcpy_PF() function returns a pointer to the terminating '\0' character of the destination string *dst*.

20.17.3.28 strcasecmp_F()

```
int strcasecmp_F (
            const char * s1,
            const __flash char * s2 )
```

Compare two strings ignoring case.

The strcasecmp_F function compares the two strings `s1` and `s2`, ignoring the case of the characters.

Parameters

s1	A pointer to a string in SRAM.
s2	A pointer to a string in address-space __flash.

Returns

The strcasecmp_F function returns an integer less than, equal to, or greater than zero if `s1` is found, respectively, to be less than, to match, or be greater than `s2`. A consequence of the ordering used by strcasecmp_F is that if `s1` is an initial substring of `s2`, then `s1` is considered to be "less than" `s2`.

20.17.3.29 strcasecmp_FX()

```
int strcasecmp_FX (
            const char * s1,
            const __flashx char * s2 )
```

Compare two strings ignoring case.

The strcasecmp_FX function compares the two strings *s1* and *s2*, ignoring the case of the characters.

Parameters

s1	A pointer to the first string in SRAM.
s2	A pointer to the second string in address-space __flashx.

Returns

The strcasecmp_FX function returns an integer less than, equal to, or greater than zero if *s1* is found, respectively, to be less than, to match, or be greater than *s2*.

20.17.3.30 strcasestr_F()

```
char * strcasestr_F (
            const char * s1,
            const __flash char * s2 )
```

This function is similar to strcasestr except that `s2` is pointer to a string in address-space __flash.

20.17.3.31 strcat_F()

```
char * strcat_F (
            char * dest,
            const __flash char * src )
```

The strcat_F function is similar to strcat except that the `src` string must be located in address-space __flash.

Returns

The strcat function returns a pointer to the resulting string `dest`.

20.17.3.32 strcat_FX()

```
char * strcat_FX (
            char * dst,
            const __flashx char * src )
```

Concatenates two strings.

The strcat_FX function is similar to strcat except that the *src* string must be located in address-space __flashx.

Parameters

dst	A pointer to the destination string in SRAM.
src	A pointer to the string located in __flashx to be appended.

Returns

The strcat_FX function returns a pointer to the resulting string *dst*.

20.17.3.33 strchr_F()

```
const __flash char * strchr_F (
            const __flash char * s,
            int val )
```

Locate character in a string in address-space __flash.

The strchr_F function locates the first occurrence of `val` (converted to a char) in the string pointed to by `s` in address-space __flash. The terminating NULL character is considered to be part of the string.

The strchr_F function is similar to strchr except that `s` is pointer to a string in address-space __flash.

Returns

The strchr_F function returns a pointer to the matched character or `NULL` if the character is not found.

20.17.3.34 strchr_FX()

```
const __flashx char * strchr_FX (
            const __flashx char * s,
            int val )
```

Locate a character in a string located in address-space __flashx.

The strchr_FX function locates the first occurrence of `val` (converted to a char) in the string pointed to by `s` in address-space __flashx. The terminating null character is considered to be part of the string.

The strchr_FX function is similar to strchr except that `s` is a pointer to a string in address-space __flashx that's *not required* to be located in the lower 64 KiB block like it is the case for strchr_F.

Returns

The strchr_FX function returns a far pointer to the matched character or `0` if the character is not found.

20.17.3.35 strchrnul_F()

```
const __flash char * strchrnul_F (
            const __flash char * s,
            int c )
```

The strchrnul_F function is like strchr_F except that if `c` is not found in `s`, then it returns a pointer to the NULL byte at the end of `s`, rather than `NULL`. (Glibc, GNU extension.)

Returns

The strchrnul_F function returns a pointer to the matched character, or a pointer to the NULL byte at the end of `s` (i.e., `s+strlen(s)`) if the character is not found.

20.17.3.36 strcmp_F()

```
int strcmp_F (
            const char * s1,
            const __flash char * s2 )  [inline]
```

The strcmp_F function is similar to strcmp except that `s2` is pointer to a string in address-space __flash.

Returns

The strcmp_F function returns an integer less than, equal to, or greater than zero if `s1` is found, respectively, to be less than, to match, or be greater than `s2`. A consequence of the ordering used by strcmp_F is that if `s1` is an initial substring of `s2`, then `s1` is considered to be "less than" `s2`.

20.17.3.37 strcmp_FX()

```
int strcmp_FX (
            const char * s1,
            const __flashx char * s2 )
```

Compares two strings.

The strcmp_FX function is similar to strcmp except that *s2* is a pointer to a string in address-space __flashx.

Parameters

s1	A pointer to the first string in SRAM.
s2	A pointer to the second string in __flashx.

Returns

The strcmp_FX function returns an integer less than, equal to, or greater than zero if *s1* is found, respectively, to be less than, to match, or be greater than *s2*.

20.17.3.38 strcpy_F()

```
char * strcpy_F (
            char * dest,
            const __flash char * src )  [inline]
```

The strcpy_F function is similar to strcpy except that src is a pointer to a string in address-space __flash.

Returns

The strcpy_F function returns a pointer to the destination string dest.

20.17.3.39 strcpy_FX()

```
char * strcpy_FX (
            char * dst,
            const __flashx char * src )
```

Duplicate a string from address-space __flashx.

The strcpy_FX function is similar to strcpy except that *src* is a string located in address-space __flashx.

Parameters

dst	A pointer to the destination string in SRAM.
src	A pointer to the source string in __flashx.

Returns

The strcpy_FX function returns a pointer to the destination string *dst*.

20.17.3.40 strcspn_F()

```
size_t strcspn_F (
            const char * s,
            const __flash char * reject )
```

The strcspn_F function calculates the length of the initial segment of `s` which consists entirely of characters not in `reject`. This function is similar to strcspn except that `reject` is a pointer to a string in address-space __flash.

Returns

The strcspn_F function returns the number of characters in the initial segment of `s` which are not in the string `reject`. The terminating zero is not considered as a part of string.

20.17.3.41 strlcat_F()

```
size_t strlcat_F (
            char * dst,
            const __flash char * src,
            size_t siz )
```

Concatenate two strings.

The strlcat_F function is similar to strlcat, except that the `src` string must be located in address-space __flash.

Appends `src` to string `dst` of size `siz` (unlike strncat, `siz` is the full size of `dst`, not space left). At most `siz-1` characters will be copied. Always NULL terminates (unless `siz <= strlen(dst)`).

Returns

The strlcat_F function returns strlen(src) + MIN(siz, strlen(initial dst)). If retval >= siz, truncation occurred.

20.17.3.42 strlcat_FX()

```
size_t strlcat_FX (
            char * dst,
            const __flashx char * src,
            size_t n )
```

Concatenate two strings.

The strlcat_FX function is similar to strlcat, except that the *src* string must be located in address-space __flashx.

Appends src to string dst of size *n* (unlike strncat, *n* is the full size of *dst*, not space left). At most *n-1* characters will be copied. Always NULL terminates (unless *n* <= strlen(*dst*)).

Parameters

dst	A pointer to the destination string in SRAM.
src	A pointer to the source string in __flashx.
n	The total number of bytes allocated to the destination string.

Returns

The strlcat_FX function returns strlen(*src*) + MIN(*n*, strlen(initial *dst*)). If retval >= *n*, truncation occurred.

20.17.3.43 strlcpy_F()

```
size_t strlcpy_F (
            char * dst,
            const __flash char * src,
            size_t siz )
```

Copy a string from address-space __flash to RAM.

Copy `src` to string `dst` of size `siz`. At most `siz-1` characters will be copied. Always NULL terminates (unless `siz` == 0). The strlcpy_F function is similar to strlcpy except that the `src` is pointer to a string in address-space __flash.

Returns

The strlcpy_F function returns strlen(src). If retval >= siz, truncation occurred.

20.17.3.44 strlcpy_FX()

```
size_t strlcpy_FX (
            char * dst,
            const __flashx char * src,
            size_t len )
```

Copy a string from address-space __flashx to RAM.

Copy src to string dst of length len. At most len-1 characters will be copied. Always NULL terminates (unless len == 0).

Returns

The strlcpy_FX function returns strlen_FX(src). If retval >= len, truncation occurred.

20.17.3.45 strlen_F()

```
size_t strlen_F (
            const __flash char * src )  [inline], [static]
```

The strlen_F function is similar to strlen, except that `src` is a pointer to a string in address-space __flash.

Returns

The strlen_F function returns the number of characters in `src`.

Note

strlen_F is implemented as an inline function in the avr/flash.h header file, which will check if the length of the string is a constant and known at compile time. If it is not known at compile time, the macro will issue a call to a libc function which will then calculate the length of the string as usual.

20.17.3.46 strlen_FX()

```
size_t strlen_FX (
            const __flashx char * s )
```

Obtain the length of a string located in address-space __flashx.

The strlen_FX function is similar to strlen, except that *s* is a pointer to a string in address-space __flashx.

Returns

The strlen_FX function returns the number of characters in *s*.

20.17.3.47 strncasecmp_F()

```
int strncasecmp_F (
            const char * s1,
            const __flash char * s2,
            size_t n )
```

Compare two strings ignoring case.

The strncasecmp_F function is similar to strcasecmp_F, except it only compares the first `n` characters of `s1`.

Parameters

s1	A pointer to a string in SRAM.
s2	A pointer to a string in address-space __flash.
n	The maximum number of bytes to compare.

Returns

The strncasecmp_F function returns an integer less than, equal to, or greater than zero if `s1` (or the first `n` bytes thereof) is found, respectively, to be less than, to match, or be greater than `s2`. A consequence of the ordering used by strncasecmp_F is that if `s1` is an initial substring of `s2`, then `s1` is considered to be "less than" `s2`.

20.17.3.48 strncasecmp_FX()

```
int strncasecmp_FX (
            const char * s1,
            const __flashx char * s2,
            size_t n )
```

Compare two strings ignoring case.

The strncasecmp_FX function is similar to strcasecmp_FX, except it only compares the first *n* characters of *s1*, and the string *s2* is located in address-space __flashx.

Parameters

s1	A pointer to a string in SRAM.
s2	A pointer to a string in __flashx.
n	The maximum number of bytes to compare.

Returns

The strncasecmp_FX function returns an integer less than, equal to, or greater than zero if *s1* (or the first *n* bytes thereof) is found, respectively, to be less than, to match, or be greater than *s2*.

20.17.3.49 strncat_F()

```
char * strncat_F (
            char * dest,
            const __flash char * src,
            size_t len )
```

Concatenate two strings.

The strncat_F function is similar to strncat, except that the `src` string must be located in address-space __flash.

Returns

The strncat_F function returns a pointer to the resulting string dest.

20.17.3.50 strncat_FX()

```
char * strncat_FX (
            char * dst,
            const __flashx char * src,
            size_t n )
```

Concatenate two strings.

The strncat_FX function is similar to strncat, except that the *src* string must be located in address-space __flashx.

Parameters

dst	A pointer to the destination string in SRAM.
src	A pointer to the source string in __flashx.
n	The maximum number of bytes to append.

Returns

The strncat_FX function returns a pointer to the resulting string *dst*.

20.17.3.51 strncmp_F()

```
int strncmp_F (
            const char * s1,
            const __flash char * s2,
            size_t n )
```

The strncmp_F function is similar to strcmp_F except it only compares the first (at most) n characters of s1 and s2.

Returns

The strncmp_F function returns an integer less than, equal to, or greater than zero if s1 (or the first n bytes thereof) is found, respectively, to be less than, to match, or be greater than s2.

20.17.3.52 strncmp_FX()

```
int strncmp_FX (
            const char * s1,
            const __flashx char * s2,
            size_t n )
```

Compare two strings with limited length.

The strncmp_FX function is similar to strcmp_FX except it only compares the first (at most) *n* characters of *s1* and *s2*.

Parameters

s1	A pointer to the first string in SRAM.
s2	A pointer to the second string in address-space __flashx.
n	The maximum number of bytes to compare.

Returns

The strncmp_FX function returns an integer less than, equal to, or greater than zero if *s1* (or the first *n* bytes thereof) is found, respectively, to be less than, to match, or be greater than *s2*.

20.17.3.53 strncpy_F()

```
char * strncpy_F (
            char * dest,
            const __flash char * src,
            size_t n )
```

The strncpy_F function is similar to strcpy_F except that not more than `n` bytes of src are copied. Thus, if there is no null byte among the first `n` bytes of `src`, the result will not be null-terminated.

In the case where the length of `src` is less than that of `n`, the remainder of `dest` will be padded with nulls.

Returns

The strncpy_F function returns a pointer to the destination string dest.

20.17.3.54 strncpy_FX()

```
char * strncpy_FX (
            char * dst,
            const __flashx char * src,
            size_t n )
```

Duplicate a string from address-space __flashx until a limited length.

The strncpy_FX function is similar to strcpy_FX except that not more than *n* bytes of *src* are copied. Thus, if there is no null byte among the first *n* bytes of *src*, the result will not be null-terminated.

In the case where the length of *src* is less than that of *n*, the remainder of *dst* will be padded with nulls.

Parameters

dst	A pointer to the destination string in SRAM.
src	A far pointer to the source string in address-space __flashx.
n	The maximum number of bytes to copy.

Returns

The strncpy_FX function returns a pointer to the destination string *dst*.

20.17.3.55 strnlen_F()

```
size_t strnlen_F (
            const __flash char * src,
            size_t len )
```

Determine the length of a fixed-size string.

The strnlen_F function is similar to strnlen, except that `src` is a pointer to a string in address-space __flash.

Returns

The strnlen_F function returns strlen_F(src), if that is less than `len`, or `len` if there is no '\0' character among the first `len` characters pointed to by `src`.

20.17.3.56 strnlen_FX()

```
size_t strnlen_FX (
            const __flashx char * s,
            size_t len )
```

Determine the length of a fixed-size string in address-space __flashx.

The strnlen_FX function is similar to strnlen, except that *s* is a pointer to a string in address-space __flashx.

Parameters

s	The address of a string in __flashx.
len	The maximum number of length to return.

Returns

The strnlen_FX function returns strlen_FX(*s*), if that is less than *len*, or *len* if there is no '\0' character among the first *len* characters pointed to by *s*.

20.17.3.57 strpbrk_F()

```
char * strpbrk_F (
            const char * s,
            const __flash char * accept )
```

The strpbrk_F function locates the first occurrence in the string `s` of any of the characters in the __flash string `accept`. This function is similar to strpbrk except that `accept` is a pointer to a string in address-space __flash.

Returns

The strpbrk_F function returns a pointer to the character in `s` that matches one of the characters in `accept`, or `NULL` if no such character is found. The terminating zero is not considered as a part of string: If one or both args are empty, the result will `NULL`.

20.17.3.58 strrchr_F()

```
const __flash char * strrchr_F (
            const __flash char * s,
            int val )
```

Locate character in string.

The strrchr_F function returns a pointer to the last occurrence of the character `val` in the __flash string `s`.

Returns

The strrchr_F function returns a pointer to the matched character or `NULL` if the character is not found.

20.17.3.59 strsep_F()

```
char * strsep_F (
            char ** sp,
            const __flash char * delim )
```

Parse a string into tokens.

The strsep_F function locates, in the string referenced by `*sp`, the first occurrence of any character in the string `delim` (or the terminating '\0' character) and replaces it with a '\0'. The location of the next character after the delimiter character (or `NULL`, if the end of the string was reached) is stored in `*sp`. An ``empty'' field, i.e. one caused by two adjacent delimiter characters, can be detected by comparing the location referenced by the pointer returned in `*sp` to '\0'. This function is similar to strsep except that `delim` is a pointer to a string in address-space __flash.

Returns

The strsep_F function returns a pointer to the original value of `*sp`. If `*sp` is initially `NULL`, strsep_F returns `NULL`.

20.17.3.60 strspn_F()

```
size_t strspn_F (
            const char * s,
            const __flash char * accept )
```

The strspn_F function calculates the length of the initial segment of `s` which consists entirely of characters in `accept`. This function is similar to strspn except that `accept` is a pointer to a string in address-space __flash.

Returns

The strspn_F function returns the number of characters in the initial segment of `s` which consist only of characters from `accept`. The terminating zero is not considered as a part of string.

20.17.3.61 strstr_F()

```
char * strstr_F (
            const char * s1,
            const __flash char * s2 )
```

Locate a substring.

The strstr_F function finds the first occurrence of the substring `s2` in the string `s1`. The terminating '\0' characters are not compared. The strstr_F function is similar to strstr except that `s2` is pointer to a string in address-space __flash.

Returns

The strstr_F function returns a pointer to the beginning of the substring, or NULL if the substring is not found. If `s2` points to a string of zero length, the function returns `s1`.

20.17.3.62 strstr_FX()

```
char * strstr_FX (
            const char * s1,
            const __flashx char * s2 )
```

Locate a substring.

The strstr_FX function finds the first occurrence of the substring `s2` in the string `s1`. The terminating '\0' characters are not compared. The strstr_FX function is similar to strstr except that `s2` points to a string located in address-space __flashx.

Returns

The strstr_FX function returns a pointer to the beginning of the substring, or NULL if the substring is not found. If `s2` points to a string of zero length, the function returns `s1`.

20.17.3.63 strtok_F()

```
char * strtok_F (
            char * s,
            const __flash char * delim )
```

Parses the string into tokens.

strtok_F parses the string `s` into tokens. The first call to strtok_F should have `s` as its first argument. Subsequent calls should have the first argument set to NULL. If a token ends with a delimiter, this delimiting character is overwritten with a '\0' and a pointer to the next character is saved for the next call to strtok_F. The delimiter string `delim` may be different for each call.

The strtok_F function is similar to strtok except that `delim` is pointer to a string in address-space __flash.

Returns

The strtok_F function returns a pointer to the next token or NULL when no more tokens are found.

Note

strtok_F is NOT reentrant. For a reentrant version of this function see strtok_rF.

20.17.3.64 strtok_rF()

```
char * strtok_rF (
            char * string,
            const __flash char * delim,
            char ** last )
```

Parses string into tokens.

The strtok_rF function parses `string` into tokens. The first call to strtok_rF should have `string` as its first argument. Subsequent calls should have the first argument set to NULL. If a token ends with a delimiter, this delimiting character is overwritten with a '\0' and a pointer to the next character is saved for the next call to strtok_rF. The delimiter string `delim` may be different for each call. `last` is a user allocated `char*` pointer. It must be the same while parsing the same string. strtok_rF is a reentrant version of strtok_F.

The strtok_rF function is similar to strtok_r except that `delim` is pointer to a string in address-space __flash.

Returns

The strtok_rF function returns a pointer to the next token or NULL when no more tokens are found.

20.17.3.65 vfprintf_F()

```
int vfprintf_F (
            FILE * stream,
            const __flash char * fmt,
            va_list ap )
```

Variant of vfprintf that uses a fmt string that resides in address-space __flash. See also vfprintf_FSTR.

20.17.3.66 vfscanf_F()

```
int vfscanf_F (
            FILE * stream,
            const __flash char * fmt,
            va_list ap )
```

Variant of vfscanf using a fmt string in address-space __flash. See also vfscanf_FSTR.

20.17.3.67 vsnprintf_F()

```
int vsnprintf_F (
            char * s,
            size_t n,
            const __flash char * fmt,
            va_list ap )
```

Variant of vsnprintf that uses a fmt string that resides in address-space __flash. See also vsnprintf_FSTR.

20.17.3.68 vsprintf_F()

```
int vsprintf_F (
            char * s,
            const __flash char * fmt,
            va_list ap )
```

Variant of vsprintf that uses a fmt string that resides in address-space __flash. See also vsprintf_FSTR.

20.17.4 Variable Documentation

20.17.4.1 __flash

```
__flash
```

A **named address-space** for data in the lower 64 KiB of program memory

Pointers to __flash are 16 bits wide.

Objects in __flash are located in section .progmem.data, which is located *prior* to the code sections by the default linker description files. Similar to PROGMEM, the assertion is that all objects in that section will fit in the lower 64 KiB flash segment (flash byte addresses 0x0 ... 0xffff).

The compiler defines the built-in macro __FLASH when this address-space is available. It is supported as part of GNU-C (-std=gnu99 etc.), and is not available on Reduced Tiny (core AVRrc).

Since

avr-gcc v4.7 (Release 2012)

See also

avr-gcc: __flash.

20.17.4.2 __flashx

`__flashx`

A **named address-space** for data in program memory

Pointers to __flashx are 24 bits wide.

Objects in __flashx are located in section .progmemx.data, which is located *after* the code sections by the default linker description files. There is no restriction on the address range occupied by objects in that section, and __flashx supports reading across the 64 KiB segment boundaries.

The compiler defines the built-in macro `__FLASHX` when this address-space is available. It is supported as part of GNU-C (`-std=gnu99` etc.), and is not available on Reduced Tiny (core AVRrc).

Since

`avr-gcc v15` (Release 2025)

See also

`avr-gcc:` `__flashx`.

20.17.4.3 __memx

`__memx`

A **named address-space** for data in program memory or RAM

Pointers to __memx are 24 bits wide.

Objects in __memx are located in section .progmemx.data, which is located *after* the code sections by the default linker description files. There is no restriction on the address range occupied by objects in that section, and __memx supports reading across the 64 KiB flash segment boundaries.

__memx pointers can also hold RAM addresses, and reading from __memx reads from RAM or from program memory depending on the most significant bit of the address.

The compiler defines the built-in macro `__MEMX` when this address-space is available. It is supported as part of GNU-C (`-std=gnu99` etc.), and is not available on Reduced Tiny (core AVRrc). To date, AVR-LibC does not have support for functions operating on __memx.

Since

`avr-gcc v4.7` (Release 2012)

See also

`avr-gcc:` `__memx`.

20.18 <avr/fuse.h>: Fuse Support

Macros

- #define FUSEMEM __attribute__((__used__, __section__ (".fuse")))
- #define FUSES

20.18.1 Detailed Description

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

The <avr/fuse.h> header is included by <avr/io.h>.

Introduction

The Fuse API allows a user to specify the fuse settings for the specific AVR device they are compiling for. These fuse settings will be placed in a special section in the ELF output file, after linking.

Programming tools can take advantage of the fuse information embedded in the ELF file, by extracting this information and determining if the fuses need to be programmed before programming the Flash and EEPROM memories. This also allows a single ELF file to contain all the information needed to program an AVR.

To use the Fuse API, include the <avr/io.h> header file, which in turn automatically includes the individual I/O header file and the <avr/fuse.h> file. These other two files provides everything necessary to set the AVR fuses.

Fuse API

Each I/O header file must define the FUSE_MEMORY_SIZE macro which is defined to the number of fuse bytes that exist in the AVR device.

A new type, __fuse_t, is defined as a structure. The number of fields in this structure are determined by the number of fuse bytes in the FUSE_MEMORY_SIZE macro:

- If FUSE_MEMORY_SIZE == 1, there is only a single field: byte, of type uint8_t.
- If FUSE_MEMORY_SIZE == 2, there are two fields: low, and high, of type uint8_t.
- If FUSE_MEMORY_SIZE == 3, there are three fields: low, high, and extended, of type uint8_t.
- If FUSE_MEMORY_SIZE > 3, there is a single field: byte, which is an array of uint8_t with the size of the array being FUSE_MEMORY_SIZE.

A convenience macro, FUSEMEM, is defined as a GCC attribute for a custom-named section of ".fuse".

A convenience macro, FUSES, is defined that declares a variable, __fuse, of type __fuse_t with the attribute defined by FUSEMEM. This variable allows the end user to easily set the fuse data.

Note

If a device-specific I/O header file has previously defined FUSEMEM, then FUSEMEM is not redefined. If a device-specific I/O header file has previously defined FUSES, then FUSES is not redefined.

Each AVR device I/O header file has a set of defined macros which specify the actual fuse bits available on that device. The AVR fuses have inverted values, logical 1 for an unprogrammed (disabled) bit and logical 0 for a programmed (enabled) bit. The defined macros for each individual fuse bit represent this in their definition by a bit-wise inversion of a mask. For example, the FUSE_EESAVE fuse in the ATmega128 is defined as:

```
#define FUSE_EESAVE  ~_BV(3)
```

The _BV macro creates a bit mask from a bit number. It is then inverted to represent logical values for a fuse memory byte. To combine the fuse bits macros together to represent a whole fuse byte, use the bitwise AND operator, like so:

```
(FUSE_BOOTSZ0 & FUSE_BOOTSZ1 & FUSE_EESAVE & FUSE_SPIEN & FUSE_JTAGEN)
```

Warning

Many device headers define fuse macros for **not inverted** fuse bits, like for example devices from the 0-series, 1-series and 2-series. Make sure you are using the right logic operations when using fuse values, or otherwise you can damage a device.

Each device I/O header file also defines macros that provide default values for each fuse byte that is available. LFUSE_DEFAULT is defined for a Low Fuse byte. HFUSE_DEFAULT is defined for a High Fuse byte. EFUSE↩_DEFAULT is defined for an Extended Fuse byte.

If FUSE_MEMORY_SIZE > 3, then the I/O header file defines macros that provide default values for each fuse byte like so:

```
FUSE0_DEFAULT
FUSE1_DEFAULT
FUSE2_DEFAULT
FUSE3_DEFAULT
FUSE4_DEFAULT
...
```

API Usage Example

Putting all of this together is easy. Using C99's designated initializers:

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

FUSES =
{
    .low = LFUSE_DEFAULT,
    .high = FUSE_BOOTSZ0 & FUSE_BOOTSZ1 & FUSE_EESAVE & FUSE_SPIEN & FUSE_JTAGEN,
    .extended = EFUSE_DEFAULT
};
```

Or, using the variable directly instead of the FUSES macro,

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

__fuse_t __fuse FUSEMEM =
{
    .low = LFUSE_DEFAULT,
    .high = FUSE_BOOTSZ0 & FUSE_BOOTSZ1 & FUSE_EESAVE & FUSE_SPIEN & FUSE_JTAGEN,
    .extended = EFUSE_DEFAULT
};
```

If you are compiling in C++, you cannot use the designated initializers so you must do:

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

FUSES =
{
    LFUSE_DEFAULT, // .low
    FUSE_BOOTSZ0 & FUSE_BOOTSZ1 & FUSE_EESAVE & FUSE_SPIEN & FUSE_JTAGEN, // .high
    EFUSE_DEFAULT  // .extended
};
```

However there are a number of caveats that you need to be aware of to use this API properly.

Be sure to include <avr/io.h> to get all of the definitions for the API. The FUSES macro defines a global variable to store the fuse data. This variable is assigned to its own linker section. Assign the desired fuse values immediately in the variable initialization.

The .fuse section in the ELF file will get its values from the initial variable assignment ONLY. This means that you can NOT assign values to this variable in functions and the new values will not be put into the ELF `.fuse` section.

The global variable is declared in the `FUSES` macro has two leading underscores, which means that it is reserved for the "implementation", meaning the library, so it will not conflict with a user-named variable.

You must initialize ALL fields in the `__fuse_t` structure. This is because the fuse bits in all bytes default to a logical 1, meaning unprogrammed. Normal uninitialized data defaults to all logical zeros. So it is vital that all fuse bytes are initialized, even with default data. If they are not, then the fuse bits may not programmed to the desired settings.

Be sure to have the `-mmcu=`*`device`* flag in your compile command line and your linker command line to have the correct device selected and to have the correct I/O header file included when you include <avr/io.h>.

You can print out the contents of the .fuse section in the ELF file by using this command line:

```
avr-objdump -s -j .fuse <ELF file>
```

The section contents shows the address on the left, then the data going from lower address to a higher address, left to right.

20.18.2 Macro Definition Documentation

20.18.2.1 FUSEMEM

```
#define FUSEMEM __attribute__((__used__, __section__ (".fuse")))
```

20.18.2.2 FUSES

```
#define FUSES
```

A convenience macro. On Xmega devices, it is defined as

```
#define FUSES NVM_FUSES_t __fuse FUSEMEM
```

Otherwise, the definition is:

```
#define FUSES __fuse_t __fuse FUSEMEM
```

20.19 <avr/interrupt.h>: Interrupts

Global manipulation of the interrupt flag

The global interrupt flag is maintained in the I bit of the status register (SREG).

Handling interrupts frequently requires attention regarding atomic access to objects that could be altered by code running within an interrupt context, see <util/atomic.h>.

Frequently, interrupts are being disabled for periods of time in order to perform certain operations without being disturbed; see Problems with Reordering Code for things to be taken into account with respect to compiler optimizations.

- #define sei() __asm__ __volatile__ ("sei" ::: "memory")
- #define cli() __asm__ __volatile__ ("cli" ::: "memory")

Macros for writing interrupt handler functions

- #define ISR(vector, attributes)
- #define ISR_N(vector_num, attributes)
- #define SIGNAL(vector)
- #define EMPTY_INTERRUPT(vector)
- #define ISR_ALIAS(vector, target_vector)
- #define reti() __asm__ __volatile__ ("reti" ::: "memory")
- #define BADISR_vect

ISR attributes

- #define ISR_BLOCK
- #define ISR_NOBLOCK
- #define ISR_NAKED
- #define ISR_FLATTEN
- #define ISR_NOICF
- #define ISR_NOGCCISR
- #define ISR_ALIASOF(target_vector)

20.19.1 Detailed Description

Note

This discussion of interrupts was originally taken from Rich Neswold's document. See Acknowledgments.

Introduction to AVR-LibC's interrupt handling It's nearly impossible to find compilers that agree on how to handle interrupt code. Since the C language tries to stay away from machine dependent details, each compiler writer is forced to design their method of support.

In the AVR-GCC environment, the vector table is predefined to point to interrupt routines with predetermined names. By using the appropriate name, your routine will be called when the corresponding interrupt occurs. The device library provides a set of default interrupt routines, which will get used if you don't define your own.

Patching into the vector table is only one part of the problem. The compiler uses, by convention, a set of registers when it's normally executing compiler-generated code. It's important that these registers, as well as the status register, get saved and restored.

These details seem to make interrupt routines a little messy, but all these details are handled by the Interrupt API. An interrupt routine is defined with ISR() or ISR_N(). These macros register and mark the routine as an interrupt handler for the specified peripheral. The following is an example definition of a handler for the ADC interrupt.

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

ISR (ADC_vect)
{
    // user code here
}

// Alternative using the ISR_N macro with avr-gcc v15

ISR_N (ADC_vect_num)
[static] void my_adc_handler (void)
{
    // user code here
}
```

Refer to the chapter explaining assembler programming for an explanation about interrupt routines written solely in assembly.

Catch-all interrupt vector If an unexpected interrupt occurs (interrupt is enabled but no handler is installed, which usually indicates a bug), then the default action is to reset the device by jumping to the reset vector. You can override this by supplying a function named `BADISR_vect` which should be defined with ISR() as such. The name `BADISR_vect` is actually an alias for `__vector_default`. The latter must be used inside assembly code in case <avr/interrupt.h> is not included.

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

ISR (BADISR_vect)
{
    // user code here
}
```

Nested interrupts The AVR hardware clears the global interrupt flag in SREG when an interrupt request is serviced. Thus, normally interrupts will remain disabled inside the handler until the handler exits, where the RETI instruction (that is emitted by the compiler as part of the normal function epilogue for an ISR) will eventually re-enable further interrupts. For that reason, interrupt handlers normally do not nest. For most interrupt handlers, this is the desired behaviour, for some it is even required in order to prevent infinitely recursive interrupts (like UART interrupts, or level-triggered external interrupts). In rare circumstances though it might be desired to re-enable the global interrupt flag as early as possible in the interrupt handler, in order to not defer any other interrupt more than absolutely needed. This could be done using an `sei()` instruction right at the beginning of the interrupt handler, but this still leaves few instructions inside the compiler-generated function prologue to run with global interrupts disabled. The compiler can be instructed to insert a `SEI` instruction right at the beginning of an interrupt handler by declaring the handler the following way:

```
ISR (XXX_vect, ISR_NOBLOCK)
{
  ...
```

```
}

// or

ISR_N (XXX_vect_num, ISR_NOBLOCK)
[static] void my_XXX_handler (void)
{
  ...
}
```

where `XXX_vect` is the name of a valid interrupt vector for the MCU type in question, as explained below.

Two vectors sharing the same code In some circumstances, the actions to be taken upon two different interrupts might be completely identical so a single implementation for the ISR would suffice. For example, pin-change interrupts arriving from two different ports could logically signal an event that is independent from the actual port (and thus interrupt vector) where it happened. Sharing interrupt vector code can be accomplished using the ISR_ALIASOF() attribute to the ISR macro:

```
ISR (PCINT0_vect)
{
  // Code to handle the event.
}

ISR (PCINT1_vect, ISR_ALIASOF(PCINT0_vect));

\note There is no body to the aliased ISR.

// Alternative using ISR_N

ISR_N (PCINT0_vect_num)
ISR_N (PCINT1_vect_num)
[static] void my_pcint_handler (void)
{
  // Code to handle the event.
}

// or

ISR (PCINT0_vect,
     [attributes]
     ISR_N (PCINT1_vect_num)
     ISR_N (PCINT2_vect_num))
{
  // Code to handle the event.
}
```

Note that the ISR_ALIASOF() feature requires GCC 4.2 or above (or a patched version of GCC 4.1.x). See the documentation of the ISR_ALIAS() macro for an implementation which is less elegant but could be applied to all compiler versions. The ISR_N() macro requires GCC v15 or higher.

Empty interrupt service routines In rare circumstances, in interrupt vector does not need any code to be implemented at all. The vector must be declared anyway, so when the interrupt triggers it won't execute the BADISR_vect code (which by default restarts the application).

This could for example be the case for interrupts that are solely enabled for the purpose of getting the controller out of sleep_mode().

A handler for such an interrupt vector can be declared using the EMPTY_INTERRUPT() macro:

```
EMPTY_INTERRUPT(ADC_vect);
```

Note

There is no body to this macro.

Manually defined ISRs In some circumstances, the compiler-generated prologue and epilogue of the ISR might not be optimal for the job, and a manually defined ISR could be considered particularly to speedup the interrupt handling.

One solution to this could be to implement the entire ISR as manual assembly code in a separate (assembly) file. See Combining C and assembly source files for an example of how to implement it that way.

Another solution is to still implement the ISR in C language but take over the compiler's job of generating the prologue and epilogue. This can be done using the ISR_NAKED attribute to the ISR() and ISR_N() macros. Note that the compiler does not generate *anything* as prologue or epilogue, so the final reti() must be provided by the actual implementation. SREG must be manually saved if the ISR code modifies it, and the compiler-implied assumption of `__zero_reg__` always being 0 could be wrong (e. g. when interrupting right after of a `MUL` instruction).

Warning

According to the GCC documentation, only inline assembly is supported in `naked` functions, like with ISR_NAKED.

```
ISR (TIMER1_OVF_vect, ISR_NAKED)
{
  PORTB |= _BV(0);  // results in SBI which does not affect SREG
  reti();
}

// or

ISR_N (TIMER1_OVF_vect_num, ISR_NAKED)
[static] void my_ovf_handler (void)
{
  PORTB |= _BV(0);  // results in SBI which does not affect SREG
  reti();
}
```

Choosing the vector: Interrupt vector names The interrupt is chosen by supplying one of the vector names in the following table. Apart from the *NAME*`_vect` macros listed below, for each such ISR name there is also a macro *NAME*`_vect_num` defined which resolves to the IRQ number and that can be used in the ISR_N macro.

There are currently two different styles present for naming the vectors.

- Starting with AVR-LibC v1.4, the style of interrupt vector names is a short phrase for the vector description followed by `_vect`. The short phrase matches the vector name as described in the datasheet of the respective device (and in the hardware manufacturer's XML/ATDF files), with spaces replaced by an underscore and other non-alphanumeric characters dropped. Using the suffix `_vect` is intended to improve portability to other C compilers available for the AVR that use a similar naming convention.
- A **deprecated** form that uses names starting with `SIG_`, followed by a relatively verbose but arbitrarily chosen name describing the interrupt vector. This has been the only available style in AVR-LibC up to version 1.2.x. This historical naming style is not recommended for new projects, and some headers require that the macro `__AVR_LIBC_DEPRECATED_ENABLE__` is defined so that the `SIG_` names ISR names are available.

Note

The ISR() macro cannot really spell-check the argument passed to them. Thus, by misspelling one of the names below used in ISR(), a function will be created that, while possibly being usable as an interrupt function, is not actually wired into the interrupt vector table. The compiler will generate a warning if it detects a suspiciously looking name of an ISR() function (i.e. one that after macro replacement does not start with `"__vector"`).

See also What ISR names are available for my device? in the FAQ for how find all the vector names for a specific device.

Go down to the MCU → Vector Names table.
Skip tables.

Table 57 Due to its sheer size, the following table is only available in the HTML version of the documentation.

Vector Name	Description	Applicable for Device

Note

For the following devices, only the deprecated `SIG_` names are available: AT43USB320, AT43USB355, AT76C711, AT90C8534, AT94K, M3000.

Go up to the Vector Name → MCUs table.
Skip this table.

Table 58 Due to its sheer size, the following table is only available in the HTML version of the documentation.

AVR Device	Vector Names

Note

For the following devices, only the deprecated `SIG_` names are available: AT43USB320, AT43USB355, AT76C711, AT90C8534, AT94K, M3000.

Go up to the Vector Name → MCUs table.
Go up to the MCU → Vector Names table.

20.19.2 Macro Definition Documentation

20.19.2.1 BADISR_vect

```
#define BADISR_vect
#include <avr/interrupt.h>
```

This is a vector which is aliased to `__vector_default`, the vector executed when an IRQ fires with no accompanying ISR handler. This may be used along with the ISR() macro to create a catch-all for undefined but used ISRs for debugging purposes. It cannot be used with ISR_N since there is no associated interrupt number.

20.19.2.2 cli

```
#define cli( ) __asm__ __volatile__ ("cli" ::: "memory")
```

Disables all interrupts by clearing the global interrupt mask. This function actually compiles into a single line of assembly, so there is no function call overhead. However, the macro also implies a *memory barrier* which can cause additional loss of optimization.

In order to implement atomic access to multi-byte objects, consider using the macros from <util/atomic.h>, rather than implementing them manually with cli() and sei().

20.19.2.3 EMPTY_INTERRUPT

```
#define EMPTY_INTERRUPT(
              vector )
```

Defines an empty interrupt handler function. This will not generate any prolog or epilog code and will only return from the ISR. Do not define a function body as this will define it for you. Example:

```
EMPTY_INTERRUPT(ADC_vect);
```

20.19.2.4 ISR

```
#define ISR(
              vector,
              attributes )
```

Introduces an interrupt handler function (interrupt service routine) that runs with the `SREG.I` flag unchanged by default with no attributes specified. (On most devices this means that global interrupts are disabled upon servicing the IRQ.)

The `attributes` are optional and alter the behaviour and resultant generated code of the interrupt routine. Multiple attributes may be used for a single function, with a space separating each attribute.

Valid attributes are ISR_BLOCK, ISR_NOBLOCK, ISR_NAKED, ISR_FLATTEN, ISR_NOICF, ISR_NOGCCISR and ISR_ALIASOF(vect).

`vector` must be one of the interrupt vector names that are valid for the particular MCU type.

See also the ISR_N macro for an alternative way to introduce an ISR.

20.19.2.5 ISR_ALIAS

```
#define ISR_ALIAS(
              vector,
              target_vector )
```

Aliases a given vector to another one in the same manner as the ISR_ALIASOF attribute for the ISR() macro.

Note

This macro creates a trampoline function for the aliased macro. This will result in a two cycle penalty for the aliased vector compared to the ISR the vector is aliased to, due to the JMP/RJMP opcode used.

Deprecated For new code, the use of ISR(..., ISR_ALIASOF(...)) or ISR_N is recommended. Notice that using ISR_N does *not* impose a JMP/RJMP overhead.

Example:

```
ISR (INT0_vect)
{
    PORTB = 42;
}

ISR_ALIAS (INT1_vect, INT0_vect);

// Alternative using ISR_N

ISR_N (INT0_vect_num)
ISR_N (INT1_vect_num)
static void my_int01_handler (void)
{
    PORTB = 42;
}

// or

ISR (INT0_vect,
     [attributes]
     ISR_N (INT1_vect_num))
{
    PORTB = 42;
}
```

20.19.2.6 ISR_ALIASOF

```
#define ISR_ALIASOF(
              target_vector )
```

The ISR is linked to another ISR, specified by the vect parameter.

Use this attribute in the `attributes` parameter of the ISR macro. Example:

```
ISR (INT0_vect)
{
    PORTB = 42;
}

ISR (INT1_vect, ISR_ALIASOF (INT0_vect));
```

Notice that the ISR_ALIASOF macro implements its own IRQ handler that jumps to the aliased ISR, which means there is a run-time overhead of a JMP/RJMP instruction. For an alternative without overhead, see the ISR_N macro.

20.19.2.7 ISR_BLOCK

```
#define ISR_BLOCK
```

Identical to an ISR with no attributes specified. Global interrupts are initially disabled by the AVR hardware when entering the ISR, without the compiler modifying this state.

Use this attribute in the `attributes` parameter of the ISR and ISR_N macros.

20.19.2.8 ISR_FLATTEN

```
#define ISR_FLATTEN
```

The compiler will try to inline all called function into the ISR.

Use this attribute in the `attributes` parameter of the ISR and ISR_N macros.

20.19.2.9 ISR_N

```
#define ISR_N(
              vector_num,
              attributes )
```

Introduces an interrupt handler function (interrupt service routine) that runs with the SREG.I flag unchanged by default with no attributes specified. (On most devices this means that global interrupts are disabled upon servicing the IRQ.)

vector_num must be a positive interrupt vector number that is valid for the particular MCU type. For available vector numbers, see for example the MCU → Vector Names table.

Contrary to the ISR macro, ISR_N does not provide a declarator for the ISR. ISR_N may be specified more than once, which can be used to define aliases. For example, the following definition provides an ISR for IRQ numbers 3 and 4 on an ATmega328:

```
ISR_N (PCINT0_vect_num)
ISR_N (PCINT1_vect_num)
static void my_isr_handler (void)
{
  // Code
}
```

The attributes are optional and alter the behaviour and resultant generated code of the interrupt routine. Multiple attributes may be used for a single function, with a space separating each attribute.

Valid attributes are ISR_BLOCK, ISR_NOBLOCK, ISR_NAKED, ISR_FLATTEN, ISR_NOICF and ISR_NOGCCISR.

Since

AVR-LibC v2.3, GCC v15

20.19.2.10 ISR_NAKED

```
#define ISR_NAKED
```

ISR is created with no prologue or epilogue code. The user code is responsible for preservation of the machine state including the SREG register, as well as placing a reti() at the end of the interrupt routine.

Use this attribute in the attributes parameter of the ISR and ISR_N macros.

Note

According to GCC documentation, the only code supported in naked functions is inline assembly.

20.19.2.11 ISR_NOBLOCK

```
#define ISR_NOBLOCK
```

ISR runs with global interrupts initially enabled. The interrupt enable flag is activated by the compiler as early as possible within the ISR to ensure minimal processing delay for nested interrupts.

This may be used to create nested ISRs, however care should be taken to avoid stack overflows, or to avoid infinitely entering the ISR for those cases where the AVR hardware does not clear the respective interrupt flag before entering the ISR.

Use this attribute in the attributes parameter of the ISR and ISR_N macros.

20.19.2.12 ISR_NOGCCISR

```
#define ISR_NOGCCISR
```

Do not generate __gcc_isr pseudo instructions for this ISR. This has an effect with GCC 8 and newer only.

Use this attribute in the attributes parameter of the ISR and ISR_N macros.

20.19.2.13 ISR_NOICF

```
#define ISR_NOICF
```

Avoid identical-code-folding optimization against this ISR. This has an effect with GCC 5 and newer only.

Use this attribute in the attributes parameter of the ISR and ISR_N macros.

20.19.2.14 reti

```
#define reti( ) __asm__ __volatile__ ("reti" ::: "memory")
```

Returns from an interrupt routine, enabling global interrupts. This should be the last command executed before leaving an ISR defined with the ISR_NAKED attribute.

This macro actually compiles into a single line of assembly, so there is no function call overhead.

Note

According to the GCC documentation, the only code supported in naked functions is inline assembly.

20.19.2.15 sei

```
#define sei( ) __asm__ __volatile__ ("sei" ::: "memory")
```

Enables interrupts by setting the global interrupt mask. This function actually compiles into a single line of assembly, so there is no function call overhead. However, the macro also implies a *memory barrier* which can cause additional loss of optimization.

In order to implement atomic access to multi-byte objects, consider using the macros from <util/atomic.h>, rather than implementing them manually with cli() and sei().

20.19.2.16 SIGNAL

```
#define SIGNAL(
              vector )
```

Introduces an interrupt handler function that runs with global interrupts initially disabled.

This is the same as the ISR macro without optional attributes.

Deprecated Do not use SIGNAL() in new code. Use ISR() or ISR_N() instead.

20.20 <avr/io.h>: AVR device-specific IO definitions

Macros

- #define _PROTECTED_WRITE(reg, value)
- #define _PROTECTED_WRITE_SPM(reg, value)

20.20.1 Detailed Description

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

This header file includes the appropriate IO definitions for the device that has been specified by the `-mmcu=` compiler command-line option. This is done by diverting to the appropriate file `<avr/ioXXXX.h>` which should never be included directly. Some register names common to all AVR devices are defined directly within `<avr/common.h>`, which is included in `<avr/io.h>`, but most of the details come from the respective include file.

Note that this header always includes the following ones:

```
#include <avr/sfr_defs.h>
#include <avr/portpins.h>
#include <avr/common.h>
#include <avr/version.h>
```

See <avr/sfr_defs.h>: Special function registers for more details about that header file.

Included are definitions of the IO register set and their respective bit values as specified in the device manual. Note that due to inconsistencies in naming conventions, so even identical functions sometimes get different names on different devices.

Also included are the specific names useable for interrupt service routines as documented here.

Finally, the following macros are defined:

RAMEND
: The last on-chip RAM address.

XRAMEND
: The last possible RAM location that is addressable. This is equal to RAMEND for devices that do not allow for external RAM. For devices that allow external RAM, this will be larger than RAMEND.

E2END
: The last EEPROM address.

FLASHEND
: The last byte address in the Flash program space.

SPM_PAGESIZE
: For devices with bootloader support, the flash pagesize (in bytes) to be used for the `SPM` instruction.

E2PAGESIZE
: The size of the EEPROM page.

20.20.2 Macro Definition Documentation

20.20.2.1 _PROTECTED_WRITE

```
#define _PROTECTED_WRITE(
              reg,
              value )
```

Write value `value` to IO register `reg` that is protected through the Xmega or AVRrc configuration change protection (CCP) mechanism. This implements the timed sequence that is required for CCP.

This macro requires that the address of `reg` is a compile-time constant. When that is not the case, the `ccp_write_io()` function can be used.

Example to modify the CPU clock:

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

_PROTECTED_WRITE (CLK_PSCTRL, CLK_PSADIV0_bm);
_PROTECTED_WRITE (CLK_CTRL, CLK_SCLKSEL0_bm);
```

20.20.2.2 _PROTECTED_WRITE_SPM

```
#define _PROTECTED_WRITE_SPM(
              reg,
              value )
```

Write value `value` to register `reg` that is protected through the Xmega or ATtiny102/104 configuration change protection (CCP) key for self programming (SPM). This implements the timed sequence that is required for CCP.

This macro requires that the address of `reg` is a compile-time constant. When that is not the case, the `ccp_write_spm()` function can be used.

Example to modify the CPU clock:

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

_PROTECTED_WRITE_SPM (NVMCTRL_CTRLA, NVMCTRL_CMD_PAGEERASEWRITE_gc);
```

20.21 <avr/lock.h>: Lockbit Support

Introduction

The Lockbit API allows a user to specify the lockbit settings for the specific AVR device they are compiling for. These lockbit settings will be placed in a special section in the ELF output file, after linking.

Programming tools can take advantage of the lockbit information embedded in the ELF file, by extracting this information and determining if the lockbits need to be programmed after programming the Flash and EEPROM memories. This also allows a single ELF file to contain all the information needed to program an AVR.

To use the Lockbit API, include the <avr/io.h> header file, which in turn automatically includes the individual I/O header file and the <avr/lock.h> file. These other two files provides everything necessary to set the AVR lockbits.

Lockbit API

Each I/O header file may define up to 3 macros that controls what kinds of lockbits are available to the user.

If __LOCK_BITS_EXIST is defined, then two lock bits are available to the user and 3 mode settings are defined for these two bits.

If __BOOT_LOCK_BITS_0_EXIST is defined, then the two BLB0 lock bits are available to the user and 4 mode settings are defined for these two bits.

If __BOOT_LOCK_BITS_1_EXIST is defined, then the two BLB1 lock bits are available to the user and 4 mode settings are defined for these two bits.

If __BOOT_LOCK_APPLICATION_TABLE_BITS_EXIST is defined then two lock bits are available to set the locking mode for the Application Table Section (which is used in the XMEGA family).

If __BOOT_LOCK_APPLICATION_BITS_EXIST is defined then two lock bits are available to set the locking mode for the Application Section (which is used in the XMEGA family).

If __BOOT_LOCK_BOOT_BITS_EXIST is defined then two lock bits are available to set the locking mode for the Boot Loader Section (which is used in the XMEGA family).

The AVR lockbit modes have inverted values, logical 1 for an unprogrammed (disabled) bit and logical 0 for a programmed (enabled) bit. The defined macros for each individual lock bit represent this in their definition by a bit-wise inversion of a mask. For example, the LB_MODE_3 macro is defined as:

```
    #define LB_MODE_3   (0xFC)
```

To combine the lockbit mode macros together to represent a whole byte, use the bitwise AND operator, like so:

```
(LB_MODE_3 & BLB0_MODE_2)
```

<avr/lock.h> also defines a macro that provides a default lockbit value: LOCKBITS_DEFAULT which is defined to be 0xFF.

See the AVR device specific datasheet for more details about these lock bits and the available mode settings.

A convenience macro, LOCKMEM, is defined as a GCC attribute for a custom-named section of ".lock".

A convenience macro, LOCKBITS, is defined that declares a variable, __lock, of type unsigned char with the attribute defined by LOCKMEM. This variable allows the end user to easily set the lockbit data.

Note

If a device-specific I/O header file has previously defined LOCKMEM, then LOCKMEM is not redefined. If a device-specific I/O header file has previously defined LOCKBITS, then LOCKBITS is not redefined. LOCKBITS is currently known to be defined in the I/O header files for the XMEGA devices.

API Usage Example

Putting all of this together is easy:

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

LOCKBITS = (LB_MODE_1 & BLB0_MODE_3 & BLB1_MODE_4);

int main(void)
{
    return 0;
}
```

Or:

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

unsigned char __lock __attribute__((section (".lock"))) =
    (LB_MODE_1 & BLB0_MODE_3 & BLB1_MODE_4);

int main(void)
{
    return 0;
}
```

However there are a number of caveats that you need to be aware of to use this API properly.

Be sure to include <avr/io.h> to get all of the definitions for the API. The LOCKBITS macro defines a global variable to store the lockbit data. This variable is assigned to its own linker section. Assign the desired lockbit values immediately in the variable initialization.

The .lock section in the ELF file will get its values from the initial variable assignment ONLY. This means that you can NOT assign values to this variable in functions and the new values will not be put into the ELF .lock section.

The global variable is declared in the LOCKBITS macro has two leading underscores, which means that it is reserved for the "implementation", meaning the library, so it will not conflict with a user-named variable.

You must initialize the lockbit variable to some meaningful value, even if it is the default value. This is because the lockbits default to a logical 1, meaning unprogrammed. Normal uninitialized data defaults to all locgial zeros. So it is vital that all lockbits are initialized, even with default data. If they are not, then the lockbits may not programmed to the desired settings and can possibly put your device into an unrecoverable state.

Be sure to have the -mmcu=*device* flag in your compile command line and your linker command line to have the correct device selected and to have the correct I/O header file included when you include <avr/io.h>.

You can print out the contents of the .lock section in the ELF file by using this command line:

```
avr-objdump -s -j .lock <ELF file>
```

20.22 <avr/pgmspace.h>: Program Space Utilities

Functions reading from PROGMEM

- static char pgm_read_char (const char *addr)
- static unsigned char pgm_read_unsigned_char (const unsigned char *addr)
- static signed char pgm_read_signed_char (const signed char *addr)
- static uint8_t pgm_read_u8 (const uint8_t *addr)
- static int8_t pgm_read_i8 (const int8_t *addr)
- static short pgm_read_short (const short *addr)
- static unsigned short pgm_read_unsigned_short (const unsigned short *addr)

- static uint16_t pgm_read_u16 (const uint16_t *addr)
- static int16_t pgm_read_i16 (const int16_t *addr)
- static int pgm_read_int (const int *addr)
- static signed pgm_read_signed (const signed *addr)
- static unsigned pgm_read_unsigned (const unsigned *addr)
- static signed int pgm_read_signed_int (const signed int *addr)
- static unsigned int pgm_read_unsigned_int (const unsigned int *addr)
- static int24_t pgm_read_i24 (const int24_t *addr)
- static uint24_t pgm_read_u24 (const uint24_t *addr)
- static uint32_t pgm_read_u32 (const uint32_t *addr)
- static int32_t pgm_read_i32 (const int32_t *addr)
- static long pgm_read_long (const long *addr)
- static unsigned long pgm_read_unsigned_long (const unsigned long *addr)
- static long long pgm_read_long_long (const long long *addr)
- static unsigned long long pgm_read_unsigned_long_long (const unsigned long long *addr)
- static uint64_t pgm_read_u64 (const uint64_t *addr)
- static int64_t pgm_read_i64 (const int64_t *addr)
- static float pgm_read_float (const float *addr)
- static double pgm_read_double (const double *addr)
- static long double pgm_read_long_double (const long double *addr)

Functions reading from PROGMEM_FAR

- static char pgm_read_char_far (uint_farptr_t addr)
- static unsigned char pgm_read_unsigned_char_far (uint_farptr_t addr)
- static signed char pgm_read_signed_char_far (uint_farptr_t addr)
- static uint8_t pgm_read_u8_far (uint_farptr_t addr)
- static int8_t pgm_read_i8_far (uint_farptr_t addr)
- static short pgm_read_short_far (uint_farptr_t addr)
- static unsigned short pgm_read_unsigned_short_far (uint_farptr_t addr)
- static uint16_t pgm_read_u16_far (uint_farptr_t addr)
- static int16_t pgm_read_i16_far (uint_farptr_t addr)
- static int pgm_read_int_far (uint_farptr_t addr)
- static unsigned pgm_read_unsigned_far (uint_farptr_t addr)
- static unsigned int pgm_read_unsigned_int_far (uint_farptr_t addr)
- static signed pgm_read_signed_far (uint_farptr_t addr)
- static signed int pgm_read_signed_int_far (uint_farptr_t addr)
- static long pgm_read_long_far (uint_farptr_t addr)
- static unsigned long pgm_read_unsigned_long_far (uint_farptr_t addr)
- static int24_t pgm_read_i24_far (uint_farptr_t addr)
- static uint24_t pgm_read_u24_far (uint_farptr_t addr)
- static uint32_t pgm_read_u32_far (uint_farptr_t addr)
- static int32_t pgm_read_i32_far (uint_farptr_t addr)
- static long long pgm_read_long_long_far (uint_farptr_t addr)
- static unsigned long long pgm_read_unsigned_long_long_far (uint_farptr_t addr)
- static uint64_t pgm_read_u64_far (uint_farptr_t addr)
- static int64_t pgm_read_i64_far (uint_farptr_t addr)
- static float pgm_read_float_far (uint_farptr_t addr)
- static double pgm_read_double_far (uint_farptr_t addr)
- static long double pgm_read_long_double_far (uint_farptr_t addr)

Functions with a PROGMEM argument

Similar to the functions from <string.h>, but with one string argument located in the lower 64 KiB segment of program memory.
For similar functions with address-space __flash, see <avr/flash.h>

- static size_t strlen_P (const char *s)
- const void * memchr_P (const void *, int __val, size_t __len)
- int memcmp_P (const void *, const void *, size_t)
- void * memccpy_P (void *, const void *, int __val, size_t)
- void * memcpy_P (void *, const void *, size_t)
- void * memmem_P (const void *, size_t, const void *, size_t)
- const void * memrchr_P (const void *, int __val, size_t __len)
- char * strcat_P (char *, const char *)
- const char * strchr_P (const char *, int __val)
- const char * strchrnul_P (const char *, int __val)
- int strcmp_P (const char *, const char *)
- char * strcpy_P (char *, const char *)
- char * stpcpy_P (char *, const char *)
- int strcasecmp_P (const char *, const char *)
- char * strcasestr_P (const char *, const char *)
- size_t strcspn_P (const char *__s, const char *__reject)
- size_t strlcat_P (char *, const char *, size_t)
- size_t strlcpy_P (char *, const char *, size_t)
- size_t strnlen_P (const char *, size_t)
- int strncmp_P (const char *, const char *, size_t)
- int strncasecmp_P (const char *, const char *, size_t)
- char * strncat_P (char *, const char *, size_t)
- char * strncpy_P (char *, const char *, size_t)
- char * strpbrk_P (const char *__s, const char *__accept)
- const char * strrchr_P (const char *, int __val)
- char * strsep_P (char **__sp, const char *__delim)
- size_t strspn_P (const char *__s, const char *__accept)
- char * strstr_P (const char *, const char *)
- char * strtok_P (char *__s, const char *__delim)
- char * strtok_rP (char *__s, const char *__delim, char **__last)

Functions with a PROGMEM_FAR argument

Similar to the functions from <string.h>, but with one string argument located in program memory.
For similar functions with address-space __flashx, see <avr/flash.h>

- size_t strlen_PF (uint_farptr_t src)
- size_t strnlen_PF (uint_farptr_t src, size_t len)
- void * memcpy_PF (void *dest, uint_farptr_t src, size_t len)
- char * strcpy_PF (char *dest, uint_farptr_t src)
- char * stpcpy_PF (char *dest, uint_farptr_t src)
- char * strncpy_PF (char *dest, uint_farptr_t src, size_t len)
- char * strcat_PF (char *dest, uint_farptr_t src)
- size_t strlcat_PF (char *dst, uint_farptr_t src, size_t siz)
- char * strncat_PF (char *dest, uint_farptr_t src, size_t len)
- int strcmp_PF (const char *s1, uint_farptr_t s2)

- int strncmp_PF (const char *s1, uint_farptr_t s2, size_t n)
- int strcasecmp_PF (const char *s1, uint_farptr_t s2)
- int strncasecmp_PF (const char *s1, uint_farptr_t s2, size_t n)
- uint_farptr_t strchr_PF (uint_farptr_t, int __val)
- char * strstr_PF (const char *s1, uint_farptr_t s2)
- size_t strlcpy_PF (char *dst, uint_farptr_t src, size_t siz)
- int memcmp_PF (const void *, uint_farptr_t, size_t)

Templates

- template<typename T >
  T pgm_read< T > (const T *addr)
- template<typename T >
  T pgm_read_far< T > (uint_farptr_t addr)

Macros

- #define PROGMEM_FAR __attribute__((__section__(".progmemx.data")))
- #define PROGMEM __attribute__((__progmem__))
- #define PSTR(str) ({ static const PROGMEM char c[ ] = (str); &c[0]; })
- #define PSTR_FAR(str) ({ static const PROGMEM_FAR char c[ ] = (str); pgm_get_far_address(c[0]); })
- #define pgm_get_far_address(var)

Macros reading from PROGMEM

- #define pgm_read_byte_near(__addr) __LPM ((uint16_t)(__addr))
- #define pgm_read_word_near(__addr) __LPM_word ((uint16_t)(__addr))
- #define pgm_read_dword_near(__addr) __LPM_dword ((uint16_t)(__addr))
- #define pgm_read_qword_near(__addr) __LPM_qword ((uint16_t)(__addr))
- #define pgm_read_float_near(addr) pgm_read_float (addr)
- #define pgm_read_ptr_near(__addr) ((void*) __LPM_word ((uint16_t)(__addr)))
- #define pgm_read_byte(__addr) pgm_read_byte_near(__addr)
- #define pgm_read_word(__addr) pgm_read_word_near(__addr)
- #define pgm_read_dword(__addr) pgm_read_dword_near(__addr)
- #define pgm_read_qword(__addr) pgm_read_qword_near(__addr)
- #define pgm_read_ptr(__addr) pgm_read_ptr_near(__addr)

Macros reading from PROGMEM_FAR

- #define pgm_read_byte_far(__addr) __ELPM (__addr)
- #define pgm_read_word_far(__addr) __ELPM_word (__addr)
- #define pgm_read_dword_far(__addr) __ELPM_dword (__addr)
- #define pgm_read_qword_far(__addr) __ELPM_qword (__addr)
- #define pgm_read_ptr_far(__addr) ((void*) __ELPM_word (__addr))

20.22.1 Detailed Description

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

The functions in this module provide interfaces for a program to access data stored in program space (flash memory) of the device.
For a different approach using named address-spaces like __flash, see <avr/flash.h>. For functions to read fixed-point values from program memory, see <stdfix.h>.

Note

These functions are an attempt to provide some compatibility with header files that come with IAR C, to make porting applications between different compilers easier. This is not 100% compatibility though.

If you are working with strings which are completely based in RAM, use the standard string functions described in <string.h>: Strings.

If possible, put your constant tables in the lower 64 KiB and use pgm_read_byte, pgm_read_char or pgm_read_u8 etc. instead of pgm_read_byte_far since it is more efficient that way, and you can still use the upper 64 KiB for executable code. All functions that are suffixed with a `_P` *require* their arguments to be in the lower 64 KiB of the flash ROM, as they do not use `ELPM` instructions. This is normally not a big concern as the linker setup arranges any program space constants declared using PROGMEM to be placed right after the interrupt vectors, and in front of any executable code. However, it can become a problem if there are too many of these constants, or for bootloaders on devices with more than 64 KiB of ROM. *All these functions will not work in that situation.*

For **Xmega** devices, make sure the NVM controller command register (`NVM.CMD` or `NVM_CMD`) is set to 0x00 (NOP) before using any of these functions.

20.22.2 Macro Definition Documentation

20.22.2.1 pgm_get_far_address

```
#define pgm_get_far_address(
            var )
```

This macro evaluates to a uint_farptr_t 32-bit "far" pointer (only 24 bits used) to data even beyond the 64 KiB limit for the 16-bit ordinary pointer. It is similar to the '&' operator, with some limitations. Example:

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

// Section .progmemx.data is located after all the code sections.
PROGMEM_FAR
const int data[] = { 2, 3, 5, 7, 9, 11 };

int get_data (uint8_t idx)
{
    uint_farptr_t pdata = pgm_get_far_address (data[0]);
    return pgm_read_int_far (pdata + idx * sizeof(int));
}
```

Comments:

- The overhead is minimal and it's mainly due to the 32-bit size operation.
- 24 bit sizes guarantees the code compatibility for use in future devices.
- `var` has to be resolved at link-time as an existing symbol, i.e. a simple variable name, an array name, or an array or structure element provided the offset is known at compile-time, and `var` is located in static storage, etc.

20.22.2.2 pgm_read_byte

```
#define pgm_read_byte(
              __addr ) pgm_read_byte_near(__addr)
```

Read a byte from the program space with a 16-bit address.

20.22.2.3 pgm_read_byte_far

```
#define pgm_read_byte_far(
              __addr ) __ELPM (__addr)
```

Read a byte from the program space with a 32-bit (far) address.

20.22.2.4 pgm_read_byte_near

```
#define pgm_read_byte_near(
              __addr ) __LPM ((uint16_t)(__addr))
```

Read a byte from the program space with a 16-bit address.

20.22.2.5 pgm_read_dword

```
#define pgm_read_dword(
              __addr ) pgm_read_dword_near(__addr)
```

Read a double word from the program space with a 16-bit address.

20.22.2.6 pgm_read_dword_far

```
#define pgm_read_dword_far(
              __addr ) __ELPM_dword (__addr)
```

Read a double word from the program space with a 32-bit (far) address.

20.22.2.7 pgm_read_dword_near

```
#define pgm_read_dword_near(
              __addr )  __LPM_dword ((uint16_t)(__addr))
```

Read a double word from the program space with a 16-bit address.

20.22.2.8 pgm_read_float_near

```
#define pgm_read_float_near(
              addr ) pgm_read_float (addr)
```

Read a `float` from the program space with a 16-bit address.

20.22.2.9 pgm_read_ptr

```
#define pgm_read_ptr(
              __addr ) pgm_read_ptr_near(__addr)
```

Read a pointer from the program space with a 16-bit address.

20.22.2.10 pgm_read_ptr_far

```
#define pgm_read_ptr_far(
              __addr ) ((void*) __ELPM_word (__addr))
```

Read a pointer from the program space with a 32-bit (far) address.

20.22.2.11 pgm_read_ptr_near

```
#define pgm_read_ptr_near(
              __addr )  ((void*) __LPM_word ((uint16_t)(__addr)))
```

Read a pointer from the program space with a 16-bit address.

20.22.2.12 pgm_read_qword

```
#define pgm_read_qword(
              __addr ) pgm_read_qword_near(__addr)
```

Read a quad-word from the program space with a 16-bit address.

Since

AVR-LibC v2.2

20.22.2.13 pgm_read_qword_far

```
#define pgm_read_qword_far(
              __addr ) __ELPM_qword (__addr)
```

Read a quad-word from the program space with a 32-bit (far) address.

Since

AVR-LibC v2.2

20.22.2.14 pgm_read_qword_near

```
#define pgm_read_qword_near(
              __addr ) __LPM_qword ((uint16_t)(__addr))
```

Read a quad-word from the program space with a 16-bit address.

Since

AVR-LibC v2.2

20.22.2.15 pgm_read_word

```
#define pgm_read_word(
              __addr ) pgm_read_word_near(__addr)
```

Read a word from the program space with a 16-bit address.

20.22.2.16 pgm_read_word_far

```
#define pgm_read_word_far(
              __addr ) __ELPM_word (__addr)
```

Read a word from the program space with a 32-bit (far) address.

20.22.2.17 pgm_read_word_near

```
#define pgm_read_word_near(
              __addr ) __LPM_word ((uint16_t)(__addr))
```

Read a word from the program space with a 16-bit address.

20.22.2.18 PROGMEM

```
#define PROGMEM __attribute__((__progmem__))
```

Attribute to use in order to declare a read-only object in static storage being located in the lower 64 KiB of flash ROM.

Objects in this section will be located in the `.progmem.data` section. In order to access PROGMEM objects:

Reduced Tiny (AVRrc) devices
: No extra code is needed. Use vanilla C/C++ code to access.

All other devices (non-AVRrc)
: (Inline) assembly must be used in order to read from `PROGMEM` objects, like for example by means of the `pgm_read_xxx` functions and macros as declared in this header.

For an alternative, see the `__flash` named address-space which is supported since `avr-gcc v4.7`, and <avr/flash.h> which exists since AVR-LibC v2.3.

20.22.2.19 PROGMEM_FAR

```
#define PROGMEM_FAR __attribute__((__section__(".progmemx.data")))
```

Attribute to use in order to declare a read-only object being located in far flash ROM. This is similar to PROGMEM, except that it puts the static storage object in section `.progmemx.data`. In order to access the object, the `pgm_read_*_far` and `_PF` functions declared in this header can be used. In order to get its address, see pgm_get_far_address().

It only makes sense to put read-only objects in this section, though the compiler does not diagnose when this is not the case.

As an alternative available since AVR-LibC v2.3 and avr-gcc v15, you can use the 24-bit address-space __flashx and functions from <avr/flash.h> that work the same like the `_far` functions.

Since

AVR-LibC v2.2

20.22.2.20 PSTR

```
#define PSTR(
             str ) ({ static const PROGMEM char c[] = (str); &c[0]; })
```

Used to declare a static pointer to a string in program space.

20.22.2.21 PSTR_FAR

```
#define PSTR_FAR(
             str ) ({ static const PROGMEM_FAR char c[] = (str); pgm_get_far_address(c[0]);
})
```

Used to define a string literal in far program space, and to return its address of type uint_farptr_t.

Since

AVR-LibC v2.2

20.22.3 Function Documentation

20.22.3.1 memccpy_P()

```
void * memccpy_P (
             void * dest,
             const void * src,
             int val,
             size_t len )
```

This function is similar to memccpy() except that `src` points to a string in the lower 64 KiB of program space.

20.22.3.2 memchr_P()

```
const void * memchr_P (
             const void * s,
             int val,
             size_t len )
```

Scan flash memory for a character.

The memchr_P() function scans the first `len` bytes of the flash memory area pointed to by `s` for the character `val`. The first byte to match `val` (interpreted as an unsigned character) stops the operation. `s` is located in the lower 64 KiB of program memory.

Returns

The memchr_P() function returns a pointer to the matching byte or `NULL` if the character does not occur in the given memory area.

20.22.3.3 memcmp_P()

```
int memcmp_P (
             const void * s1,
             const void * s2,
             size_t len )
```

Compare memory areas.

The memcmp_P() function compares the first `len` bytes of the memory areas `s1` and flash `s2`. The comparision is performed using unsigned char operations. `s2` is located in the lower 64 KiB of program memory.

Returns

The memcmp_P() function returns an integer less than, equal to, or greater than zero if the first `len` bytes of `s1` is found, respectively, to be less than, to match, or be greater than the first `len` bytes of `s2`.

20.22.3.4 memcmp_PF()

```
int memcmp_PF (
             const void * s1,
             uint_farptr_t s2,
             size_t len )
```

Compare memory areas.

The memcmp_PF() function compares the first `len` bytes of the memory areas `s1` and flash `s2`. The comparision is performed using unsigned char operations. It is an equivalent of memcmp_P() function, except that it is capable working on all FLASH including the extended area above 64kB.

Returns

The memcmp_PF() function returns an integer less than, equal to, or greater than zero if the first `len` bytes of `s1` is found, respectively, to be less than, to match, or be greater than the first `len` bytes of `s2`.

20.22.3.5 memcpy_P()

```
void * memcpy_P (
            void * dest,
            const void * src,
            size_t n )
```

The memcpy_P() function is similar to memcpy(), except the src string resides in the lower 64 KiB of program space.

Returns

The memcpy_P() function returns a pointer to dest.

20.22.3.6 memcpy_PF()

```
void * memcpy_PF (
            void * dest,
            uint_farptr_t src,
            size_t n )
```

Copy a memory block from flash to SRAM.

The memcpy_PF() function is similar to memcpy(), except the data is copied from the program space and is addressed using a far pointer.

Parameters

dest	A pointer to the destination buffer
src	A far pointer to the origin of data in flash memory
n	The number of bytes to be copied

Returns

The memcpy_PF() function returns a pointer to *dst*.

20.22.3.7 memmem_P()

```
void * memmem_P (
            const void * s1,
            size_t len1,
            const void * s2,
            size_t len2 )
```

The memmem_P() function is similar to memmem() except that `s2` is pointer to a string in the lower 64 KiB of program space.

20.22.3.8 memrchr_P()

```
const void * memrchr_P (
            const void * src,
            int val,
            size_t len )
```

The memrchr_P() function is like the memchr_P() function, except that it searches backwards from the end of the `len` bytes pointed to by `src` instead of forwards from the front. (Glibc, GNU extension.) `src` is located in the lower 64 KiB of program memory.

Returns

The memrchr_P() function returns a pointer to the matching byte or `NULL` if the character does not occur in the given memory area.

20.22.3.9 pgm_read< T >()

```
template<typename T >
T pgm_read< T > (
            const T * addr )
```

Read an object of type `T` from program memory address `addr` and return it. This template is only available when macro `__pgm_read_template__` is defined.

Since

AVR-LibC v2.2

20.22.3.10 pgm_read_char()

```
static char pgm_read_char (
            const char * addr )  [inline], [static]
```

Read a `char` from 16-bit address `addr`. The address is in the lower 64 KiB of program memory.

Since

AVR-LibC v2.2

20.22.3.11 pgm_read_char_far()

```
static char pgm_read_char_far (
            uint_farptr_t addr )  [inline], [static]
```

Read a `char` from far address `addr`. The address is in the program memory.

Since

AVR-LibC v2.2

20.22.3.12 pgm_read_double()

```
static double pgm_read_double (
            const double * addr )  [inline], [static]
```

Read a `double` from 16-bit address `addr`. The address is in the lower 64 KiB of program memory.

Since

AVR-LibC v2.2

20.22.3.13 pgm_read_double_far()

```
static double pgm_read_double_far (
            uint_farptr_t addr )  [inline], [static]
```

Read a `double` from far address `addr`. The address is in the program memory.

Since

AVR-LibC v2.2

20.22.3.14 pgm_read_far< T >()

```
template<typename T >
T pgm_read_far< T > (
            uint_farptr_t addr )
```

Read an object of type `T` from program memory address `addr` and return it. This template is only available when macro `__pgm_read_template__` is defined.

Since

AVR-LibC v2.2

20.22.3.15 pgm_read_float()

```
static float pgm_read_float (
            const float * addr )  [inline], [static]
```

Read a `float` from 16-bit address `addr`. The address is in the lower 64 KiB of program memory.

20.22.3.16 pgm_read_float_far()

```
static float pgm_read_float_far (
            uint_farptr_t addr )  [inline], [static]
```

Read a `float` from far address `addr`. The address is in the program memory.

20.22.3.17 pgm_read_i16()

```
static int16_t pgm_read_i16 (
            const int16_t * addr )  [inline], [static]
```

Read an `int16_t` from 16-bit address `addr`. The address is in the lower 64 KiB of program memory.

Since

AVR-LibC v2.2

20.22.3.18 pgm_read_i16_far()

```
static int16_t pgm_read_i16_far (
            uint_farptr_t addr )  [inline], [static]
```

Read an `int16_t` from far address `addr`. The address is in the program memory.

Since

AVR-LibC v2.2

20.22.3.19 pgm_read_i24()

```
static int24_t pgm_read_i24 (
            const int24_t * addr )  [inline], [static]
```

Read an `int24_t` from 16-bit address `addr`. The address is in the lower 64 KiB of program memory.

Since

AVR-LibC v2.2

20.22.3.20 pgm_read_i24_far()

```
static int24_t pgm_read_i24_far (
            uint_farptr_t addr )  [inline], [static]
```

Read an `int24_t` from far address `addr`. The address is in the program memory.

Since

AVR-LibC v2.2

20.22.3.21 pgm_read_i32()

```
static int32_t pgm_read_i32 (
            const int32_t * addr )  [inline], [static]
```

Read an `int32_t` from 16-bit address `addr`. The address is in the lower 64 KiB of program memory.

Since

AVR-LibC v2.2

20.22.3.22 pgm_read_i32_far()

```
static int32_t pgm_read_i32_far (
            uint_farptr_t addr )  [inline], [static]
```

Read an `int32_t` from far address `addr`. The address is in the program memory.

Since

AVR-LibC v2.2

20.22.3.23 pgm_read_i64()

```
static int64_t pgm_read_i64 (
            const int64_t * addr )  [inline], [static]
```

Read an `int64_t` from 16-bit address `addr`. The address is in the lower 64 KiB of program memory.

Since

AVR-LibC v2.2

20.22.3.24 pgm_read_i64_far()

```
static int64_t pgm_read_i64_far (
            uint_farptr_t addr )  [inline], [static]
```

Read an `int64_t` from far address `addr`. The address is in the program memory.

Since

AVR-LibC v2.2

20.22.3.25 pgm_read_i8()

```
static int8_t pgm_read_i8 (
            const int8_t * addr )  [inline], [static]
```

Read an `int8_t` from 16-bit address `addr`. The address is in the lower 64 KiB of program memory.

Since

AVR-LibC v2.2

20.22.3.26 pgm_read_i8_far()

```
static int8_t pgm_read_i8_far (
            uint_farptr_t addr )  [inline], [static]
```

Read an `int8_t` from far address `addr`. The address is in the program memory.

Since

AVR-LibC v2.2

20.22.3.27 pgm_read_int()

```
static int pgm_read_int (
            const int * addr )  [inline], [static]
```

Read an `int` from 16-bit address `addr`. The address is in the lower 64 KiB of program memory.

Since

AVR-LibC v2.2

20.22.3.28 pgm_read_int_far()

```
static int pgm_read_int_far (
            uint_farptr_t addr )  [inline], [static]
```

Read an `int` from far address `addr`. The address is in the program memory.

Since

AVR-LibC v2.2

20.22.3.29 pgm_read_long()

```
static long pgm_read_long (
            const long * addr )  [inline], [static]
```

Read a `long` from 16-bit address `addr`. The address is in the lower 64 KiB of program memory.

Since

AVR-LibC v2.2

20.22.3.30 pgm_read_long_double()

```
static long double pgm_read_long_double (
            const long double * addr )  [inline], [static]
```

Read a `long double` from 16-bit address `addr`. The address is in the lower 64 KiB of program memory.

Since

AVR-LibC v2.2

20.22.3.31 pgm_read_long_double_far()

```
static long double pgm_read_long_double_far (
            uint_farptr_t addr )  [inline], [static]
```

Read a `long double` from far address `addr`. The address is in the program memory.

Since

AVR-LibC v2.2

20.22.3.32 pgm_read_long_far()

```
static long pgm_read_long_far (
            uint_farptr_t addr )  [inline], [static]
```

Read a `long` from far address `addr`. The address is in the program memory.

Since

AVR-LibC v2.2

20.22.3.33 pgm_read_long_long()

```
static long long pgm_read_long_long (
            const long long * addr )  [inline], [static]
```

Read a `long long` from 16-bit address `addr`. The address is in the lower 64 KiB of program memory.

Since

AVR-LibC v2.2

20.22.3.34 pgm_read_long_long_far()

```
static long long pgm_read_long_long_far (
            uint_farptr_t addr )  [inline], [static]
```

Read a `long long` from far address `addr`. The address is in the program memory.

Since

AVR-LibC v2.2

20.22.3.35 pgm_read_short()

```
static short pgm_read_short (
            const short * addr )  [inline], [static]
```

Read a `short` from 16-bit address `addr`. The address is in the lower 64 KiB of program memory.

Since

AVR-LibC v2.2

20.22.3.36 pgm_read_short_far()

```
static short pgm_read_short_far (
            uint_farptr_t addr )  [inline], [static]
```

Read a `short` from far address `addr`. The address is in the program memory.

Since

AVR-LibC v2.2

20.22.3.37 pgm_read_signed()

```
static signed pgm_read_signed (
            const signed * addr )  [inline], [static]
```

Read a `signed` from 16-bit address `addr`. The address is in the lower 64 KiB of program memory.

Since

AVR-LibC v2.2

20.22.3.38 pgm_read_signed_char()

```
static signed char pgm_read_signed_char (
            const signed char * addr )  [inline], [static]
```

Read a `signed char` from 16-bit address `addr`. The address is in the lower 64 KiB of program memory.

Since

AVR-LibC v2.2

20.22.3.39 pgm_read_signed_char_far()

```
static signed char pgm_read_signed_char_far (
            uint_farptr_t addr )  [inline], [static]
```

Read a `signed char` from far address `addr`. The address is in the program memory.

Since

AVR-LibC v2.2

20.22.3.40 pgm_read_signed_far()

```
static signed pgm_read_signed_far (
            uint_farptr_t addr )  [inline], [static]
```

Read a `signed` from far address `addr`. The address is in the program memory.

Since

AVR-LibC v2.2

20.22.3.41 pgm_read_signed_int()

```
static signed int pgm_read_signed_int (
            const signed int * addr )  [inline], [static]
```

Read a `signed int` from 16-bit address `addr`. The address is in the lower 64 KiB of program memory.

Since

AVR-LibC v2.2

20.22.3.42 pgm_read_signed_int_far()

```
static signed int pgm_read_signed_int_far (
            uint_farptr_t addr )  [inline], [static]
```

Read a `signed int` from far address `addr`. The address is in the program memory.

Since

AVR-LibC v2.2

20.22.3.43 pgm_read_u16()

```
static uint16_t pgm_read_u16 (
            const uint16_t * addr )  [inline], [static]
```

Read an `uint16_t` from 16-bit address `addr`. The address is in the lower 64 KiB of program memory.

Since

AVR-LibC v2.2

20.22.3.44 pgm_read_u16_far()

```
static uint16_t pgm_read_u16_far (
            uint_farptr_t addr )  [inline], [static]
```

Read an `uint16_t` from far address `addr`. The address is in the program memory.

Since

AVR-LibC v2.2

20.22.3.45 pgm_read_u24()

```
static uint24_t pgm_read_u24 (
            const uint24_t * addr )  [inline], [static]
```

Read an uint24_t from 16-bit address addr. The address is in the lower 64 KiB of program memory.

Since

AVR-LibC v2.2

20.22.3.46 pgm_read_u24_far()

```
static uint24_t pgm_read_u24_far (
            uint_farptr_t addr )  [inline], [static]
```

Read an uint24_t from far address addr. The address is in the program memory.

Since

AVR-LibC v2.2

20.22.3.47 pgm_read_u32()

```
static uint32_t pgm_read_u32 (
            const uint32_t * addr )  [inline], [static]
```

Read an uint32_t from 16-bit address addr. The address is in the lower 64 KiB of program memory.

Since

AVR-LibC v2.2

20.22.3.48 pgm_read_u32_far()

```
static uint32_t pgm_read_u32_far (
            uint_farptr_t addr )  [inline], [static]
```

Read an uint32_t from far address addr. The address is in the program memory.

Since

AVR-LibC v2.2

20.22.3.49 pgm_read_u64()

```
static uint64_t pgm_read_u64 (
            const uint64_t * addr )  [inline], [static]
```

Read an `uint64_t` from 16-bit address `addr`. The address is in the lower 64 KiB of program memory.

Since

AVR-LibC v2.2

20.22.3.50 pgm_read_u64_far()

```
static uint64_t pgm_read_u64_far (
            uint_farptr_t addr )  [inline], [static]
```

Read an `uint64_t` from far address `addr`. The address is in the program memory.

Since

AVR-LibC v2.2

20.22.3.51 pgm_read_u8()

```
static uint8_t pgm_read_u8 (
            const uint8_t * addr )  [inline], [static]
```

Read an `uint8_t` from 16-bit address `addr`. The address is in the lower 64 KiB of program memory.

Since

AVR-LibC v2.2

20.22.3.52 pgm_read_u8_far()

```
static uint8_t pgm_read_u8_far (
            uint_farptr_t addr )  [inline], [static]
```

Read an `uint8_t` from far address `addr`. The address is in the program memory.

Since

AVR-LibC v2.2

20.22.3.53 pgm_read_unsigned()

```
static unsigned pgm_read_unsigned (
            const unsigned * addr )  [inline], [static]
```

Read an `unsigned` from 16-bit address `addr`. The address is in the lower 64 KiB of program memory.

Since

AVR-LibC v2.2

20.22.3.54 pgm_read_unsigned_char()

```
static unsigned char pgm_read_unsigned_char (
            const unsigned char * addr )  [inline], [static]
```

Read an `unsigned char` from 16-bit address `addr`. The address is in the lower 64 KiB of program memory.

Since

AVR-LibC v2.2

20.22.3.55 pgm_read_unsigned_char_far()

```
static unsigned char pgm_read_unsigned_char_far (
            uint_farptr_t addr )  [inline], [static]
```

Read an `unsigned char` from far address `addr`. The address is in the program memory.

Since

AVR-LibC v2.2

20.22.3.56 pgm_read_unsigned_far()

```
static unsigned pgm_read_unsigned_far (
            uint_farptr_t addr )  [inline], [static]
```

Read an `unsigned` from far address `addr`. The address is in the program memory.

Since

AVR-LibC v2.2

20.22.3.57 pgm_read_unsigned_int()

```
static unsigned int pgm_read_unsigned_int (
            const unsigned int * addr )  [inline], [static]
```

Read an `unsigned int` from 16-bit address `addr`. The address is in the lower 64 KiB of program memory.

Since

AVR-LibC v2.2

20.22.3.58 pgm_read_unsigned_int_far()

```
static unsigned int pgm_read_unsigned_int_far (
            uint_farptr_t addr )  [inline], [static]
```

Read an `unsigned int` from far address `addr`. The address is in the program memory.

Since

AVR-LibC v2.2

20.22.3.59 pgm_read_unsigned_long()

```
static unsigned long pgm_read_unsigned_long (
            const unsigned long * addr )  [inline], [static]
```

Read an `unsigned long` from 16-bit address `addr`. The address is in the lower 64 KiB of program memory.

Since

AVR-LibC v2.2

20.22.3.60 pgm_read_unsigned_long_far()

```
static unsigned long pgm_read_unsigned_long_far (
            uint_farptr_t addr )  [inline], [static]
```

Read an `unsigned long` from far address `addr`. The address is in the program memory.

Since

AVR-LibC v2.2

20.22.3.61 pgm_read_unsigned_long_long()

```
static unsigned long long pgm_read_unsigned_long_long (
            const unsigned long long * addr )  [inline], [static]
```

Read an `unsigned long long` from 16-bit address `addr`. The address is in the lower 64 KiB of program memory.

Since

AVR-LibC v2.2

20.22.3.62 pgm_read_unsigned_long_long_far()

```
static unsigned long long pgm_read_unsigned_long_long_far (
            uint_farptr_t addr )  [inline], [static]
```

Read an `unsigned long long` from far address `addr`. The address is in the program memory.

Since

AVR-LibC v2.2

20.22.3.63 pgm_read_unsigned_short()

```
static unsigned short pgm_read_unsigned_short (
            const unsigned short * addr )  [inline], [static]
```

Read an `unsigned short` from 16-bit address `addr`. The address is in the lower 64 KiB of program memory.

Since

AVR-LibC v2.2

20.22.3.64 pgm_read_unsigned_short_far()

```
static unsigned short pgm_read_unsigned_short_far (
            uint_farptr_t addr )  [inline], [static]
```

Read an `unsigned short` from far address `addr`. The address is in the program memory.

Since

AVR-LibC v2.2

20.22.3.65 stpcpy_P()

```
char * stpcpy_P (
            char * dest,
            const char * src )
```

The stpcpy_P() function is similar to stpcpy() except that `src` points to a string in the lower 64 KiB of program space.

Returns

The stpcpy_P() function returns a pointer to the terminating '\0' character of destination string `dest`.

Since

AVR-LibC 2.3

20.22.3.66 stpcpy_PF()

```
char * stpcpy_PF (
            char * dst,
            uint_farptr_t src )
```

Duplicate a string.

The stpcpy_PF() function is similar to stpcpy() except that *src* is a far pointer to a string in program space.

Parameters

dst	A pointer to the destination string in SRAM
src	A far pointer to the source string in Flash

Returns

The stpcpy_PF() function returns a pointer to the terminating '\0' character of the destination string *dst*.

Since

AVR-LibC 2.3

20.22.3.67 strcasecmp_P()

```
int strcasecmp_P (
            const char * s1,
            const char * s2 )
```

Compare two strings ignoring case.

The strcasecmp_P() function compares the two strings `s1` and `s2`, ignoring the case of the characters.

Parameters

s1	A pointer to a string in the device's SRAM.
s2	A pointer to a string in the lower 64 KiB of the device's Flash.

Returns

The strcasecmp_P() function returns an integer less than, equal to, or greater than zero if `s1` is found, respectively, to be less than, to match, or be greater than `s2`. A consequence of the ordering used by strcasecmp_P() is that if `s1` is an initial substring of `s2`, then `s1` is considered to be "less than" `s2`.

20.22.3.68 strcasecmp_PF()

```
int strcasecmp_PF (
            const char * s1,
            uint_farptr_t s2 )
```

Compare two strings ignoring case.

The strcasecmp_PF() function compares the two strings *s1* and *s2*, ignoring the case of the characters.

Parameters

s1	A pointer to the first string in SRAM
s2	A far pointer to the second string in Flash

Returns

The strcasecmp_PF() function returns an integer less than, equal to, or greater than zero if *s1* is found, respectively, to be less than, to match, or be greater than *s2*.

20.22.3.69 strcasestr_P()

```
char * strcasestr_P (
            const char * s1,
            const char * s2 )
```

This function is similar to strcasestr() except that `s2` points to a string in the lower 64 KiB of program space.

20.22.3.70 strcat_P()

```
char * strcat_P (
            char * dest,
            const char * src )
```

The strcat_P() function is similar to strcat() except that the *src* string must be located in the lower 64 KiB of program space (flash).

Returns

The strcat() function returns a pointer to the resulting string *dest*.

20.22.3.71 strcat_PF()

```
char * strcat_PF (
            char * dst,
            uint_farptr_t src )
```

Concatenates two strings.

The strcat_PF() function is similar to strcat() except that the *src* string must be located in program space (flash) and is addressed using a far pointer

Parameters

dst	A pointer to the destination string in SRAM
src	A far pointer to the string to be appended in Flash

Returns

The strcat_PF() function returns a pointer to the resulting string *dst*.

20.22.3.72 strchr_P()

```
const char * strchr_P (
            const char * s,
            int val )
```

Locate character in program space string.

The strchr_P() function locates the first occurrence of `val` (converted to a char) in the string pointed to by `s` in the lower 64 KiB of program space. The terminating null character is considered to be part of the string.

The strchr_P() function is similar to strchr() except that `s` points to a string in the lower 64 KiB of program space.

Returns

The strchr_P() function returns a pointer to the matched character or `NULL` if the character is not found.

20.22.3.73 strchr_PF()

```
uint_farptr_t strchr_PF (
            uint_farptr_t s,
            int val )
```

Locate character in far program space string.

The strchr_PF() function locates the first occurrence of `val` (converted to a char) in the string pointed to by `s` in far program space. The terminating null character is considered to be part of the string.

The strchr_PF() function is similar to strchr() except that `s` is a far pointer to a string in program space that's *not required* to be located in the lower 64 KiB block like it is the case for strchr_P().

Returns

The strchr_PF() function returns a far pointer to the matched character or `0` if the character is not found.

20.22.3.74 strchrnul_P()

```
const char * strchrnul_P (
            const char * s,
            int c )
```

The strchrnul_P() function is like strchr_P() except that if `c` is not found in `s`, then it returns a pointer to the null byte at the end of `s`, rather than `NULL`. (Glibc, GNU extension.)

Returns

The strchrnul_P() function returns a pointer to the matched character, or a pointer to the null byte at the end of `s` (i.e., `s+strlen(s)`) if the character is not found.

20.22.3.75 strcmp_P()

```
int strcmp_P (
            const char * s1,
            const char * s2 )
```

The strcmp_P() function is similar to strcmp() except that `s2` is pointer to a string in the lower 64 KiB of program space.

Returns

The strcmp_P() function returns an integer less than, equal to, or greater than zero if `s1` is found, respectively, to be less than, to match, or be greater than `s2`. A consequence of the ordering used by strcmp_P() is that if `s1` is an initial substring of `s2`, then `s1` is considered to be "less than" `s2`.

20.22.3.76 strcmp_PF()

```
int strcmp_PF (
            const char * s1,
            uint_farptr_t s2 )
```

Compares two strings.

The strcmp_PF() function is similar to strcmp() except that *s2* is a far pointer to a string in program space.

Parameters

s1	A pointer to the first string in SRAM
s2	A far pointer to the second string in Flash

Returns

The strcmp_PF() function returns an integer less than, equal to, or greater than zero if *s1* is found, respectively, to be less than, to match, or be greater than *s2*.

20.22.3.77 strcpy_P()

```
char * strcpy_P (
            char * dest,
            const char * src )  [inline]
```

The strcpy_P() function is similar to strcpy() except that `src` points to a string in the lower 64 KiB of program space.

Returns

The strcpy_P() function returns a pointer to the destination string dest.

20.22.3.78 strcpy_PF()

```
char * strcpy_PF (
            char * dst,
            uint_farptr_t src )
```

Duplicate a string.

The strcpy_PF() function is similar to strcpy() except that *src* is a far pointer to a string in program space.

Parameters

dst	A pointer to the destination string in SRAM
src	A far pointer to the source string in Flash

Returns

The strcpy_PF() function returns a pointer to the destination string *dst*.

20.22.3.79 strcspn_P()

```
size_t strcspn_P (
            const char * s,
            const char * reject )
```

The strcspn_P() function calculates the length of the initial segment of `s` which consists entirely of characters not in `reject`. This function is similar to strcspn() except that `reject` points to a string in the lower 64 KiB of program space.

Returns

The strcspn_P() function returns the number of characters in the initial segment of `s` which are not in the string `reject`. The terminating zero is not considered as a part of string.

20.22.3.80 strlcat_P()

```
size_t strlcat_P (
            char * dst,
            const char * src,
            size_t siz )
```

Concatenate two strings.

The strlcat_P() function is similar to strlcat(), except that the `src` string must be located in the lower 64 KiB of program space (flash).

Appends `src` to string `dst` of size `siz` (unlike strncat(), `siz` is the full size of `dst`, not space left). At most `siz-1` characters will be copied. Always NULL terminates (unless `siz <= strlen(dst)`).

Returns

The strlcat_P() function returns strlen(src) + MIN(siz, strlen(initial dst)). If retval >= siz, truncation occurred.

20.22.3.81 strlcat_PF()

```
size_t strlcat_PF (
            char * dst,
            uint_farptr_t src,
            size_t n )
```

Concatenate two strings.

The strlcat_PF() function is similar to strlcat(), except that the *src* string must be located in program space (flash) and is addressed using a far pointer.

Appends src to string dst of size *n* (unlike strncat(), *n* is the full size of *dst*, not space left). At most *n-1* characters will be copied. Always NULL terminates (unless *n* <= strlen(*dst*)).

Parameters

dst	A pointer to the destination string in SRAM
src	A far pointer to the source string in Flash
n	The total number of bytes allocated to the destination string

Returns

The strlcat_PF() function returns strlen(*src*) + MIN(*n*, strlen(initial *dst*)). If retval >= *n*, truncation occurred.

20.22.3.82 strlcpy_P()

```
size_t strlcpy_P (
            char * dst,
```

```
            const char * src,
            size_t siz )
```

Copy a string from progmem to RAM.

Copy `src` to string `dst` of size `siz`. At most `siz-1` characters will be copied. Always NULL terminates (unless `siz` == 0). The strlcpy_P() function is similar to strlcpy() except that `src` points to a string in the lower 64 KiB of program memory.

Returns

The strlcpy_P() function returns strlen(src). If retval >= siz, truncation occurred.

20.22.3.83 strlcpy_PF()

```
size_t strlcpy_PF (
            char * dst,
            uint_farptr_t src,
            size_t siz )
```

Copy a string from progmem to RAM.

Copy src to string dst of size siz. At most siz-1 characters will be copied. Always NULL terminates (unless siz == 0).

Returns

The strlcpy_PF() function returns strlen(src). If retval >= siz, truncation occurred.

20.22.3.84 strlen_P()

```
size_t strlen_P (
            const char * src )  [inline], [static]
```

The strlen_P() function is similar to strlen(), except that src is a pointer to a string in program space.

Returns

The strlen_P() function returns the number of characters in src.

20.22.3.85 strlen_PF()

```
size_t strlen_PF (
            uint_farptr_t s )
```

Obtain the length of a string.

The strlen_PF() function is similar to strlen(), except that *s* is a far pointer to a string in program space.

Parameters

s	A far pointer to the string in flash

Returns

The strlen_PF() function returns the number of characters in *s*.

20.22.3.86 strncasecmp_P()

```
int strncasecmp_P (
            const char * s1,
            const char * s2,
            size_t n )
```

Compare two strings ignoring case.

The strncasecmp_P() function is similar to strcasecmp_P(), except it only compares the first `n` characters of `s1`.

Parameters

s1	A pointer to a string in the device's SRAM.
s2	A pointer to a string in the thwer 64 KiB of the device's Flash.
n	The maximum number of bytes to compare.

Returns

The strncasecmp_P() function returns an integer less than, equal to, or greater than zero if `s1` (or the first `n` bytes thereof) is found, respectively, to be less than, to match, or be greater than `s2`. A consequence of the ordering used by strncasecmp_P() is that if `s1` is an initial substring of `s2`, then `s1` is considered to be "less than" `s2`.

20.22.3.87 strncasecmp_PF()

```
int strncasecmp_PF (
            const char * s1,
            uint_farptr_t s2,
            size_t n )
```

Compare two strings ignoring case.

The strncasecmp_PF() function is similar to strcasecmp_PF(), except it only compares the first *n* characters of *s1* and the string in flash is addressed using a far pointer.

Parameters

s1	A pointer to a string in SRAM
s2	A far pointer to a string in Flash
n	The maximum number of bytes to compare

Returns

The strncasecmp_PF() function returns an integer less than, equal to, or greater than zero if *s1* (or the first *n* bytes thereof) is found, respectively, to be less than, to match, or be greater than *s2*.

20.22.3.88 strncat_P()

```
char * strncat_P (
            char * dest,
            const char * src,
            size_t len )
```

Concatenate two strings.

The strncat_P() function is similar to strncat(), except that the `src` string must be located in the lower 64 KiB of program space (flash).

Returns

The strncat_P() function returns a pointer to the resulting string `dest`.

20.22.3.89 strncat_PF()

```
char * strncat_PF (
            char * dst,
            uint_farptr_t src,
            size_t n )
```

Concatenate two strings.

The strncat_PF() function is similar to strncat(), except that the *src* string must be located in program space (flash) and is addressed using a far pointer.

Parameters

dst	A pointer to the destination string in SRAM
src	A far pointer to the source string in Flash
n	The maximum number of bytes to append

Returns

The strncat_PF() function returns a pointer to the resulting string *dst*.

20.22.3.90 strncmp_P()

```
int strncmp_P (
            const char * s1,
```

```
            const char * s2,
            size_t n )
```

The strncmp_P() function is similar to strcmp_P() except it only compares the first (at most) n characters of s1 and s2. `s2` is located in the lower 64 KiB of program memory.

Returns

The strncmp_P() function returns an integer less than, equal to, or greater than zero if s1 (or the first n bytes thereof) is found, respectively, to be less than, to match, or be greater than s2.

20.22.3.91 strncmp_PF()

```
int strncmp_PF (
            const char * s1,
            uint_farptr_t s2,
            size_t n )
```

Compare two strings with limited length.

The strncmp_PF() function is similar to strcmp_PF() except it only compares the first (at most) *n* characters of *s1* and *s2*.

Parameters

s1	A pointer to the first string in SRAM
s2	A far pointer to the second string in Flash
n	The maximum number of bytes to compare

Returns

The strncmp_PF() function returns an integer less than, equal to, or greater than zero if *s1* (or the first *n* bytes thereof) is found, respectively, to be less than, to match, or be greater than *s2*.

20.22.3.92 strncpy_P()

```
char * strncpy_P (
            char * dest,
            const char * src,
            size_t n )
```

The strncpy_P() function is similar to strcpy_P() except that not more than n bytes of src are copied. Thus, if there is no null byte among the first n bytes of src, the result will not be null-terminated. `src` is located in the lower 64 KiB of program memory.

In the case where the length of src is less than that of n, the remainder of dest will be padded with nulls.

Returns

The strncpy_P() function returns a pointer to the destination string `dest`.

20.22.3.93 strncpy_PF()

```
char * strncpy_PF (
            char * dst,
            uint_farptr_t src,
            size_t n )
```

Duplicate a string until a limited length.

The strncpy_PF() function is similar to strcpy_PF() except that not more than *n* bytes of *src* are copied. Thus, if there is no null byte among the first *n* bytes of *src*, the result will not be null-terminated.

In the case where the length of *src* is less than that of *n*, the remainder of *dst* will be padded with nulls.

Parameters

dst	A pointer to the destination string in SRAM
src	A far pointer to the source string in Flash
n	The maximum number of bytes to copy

Returns

The strncpy_PF() function returns a pointer to the destination string *dst*.

20.22.3.94 strnlen_P()

```
size_t strnlen_P (
            const char * src,
            size_t len )
```

Determine the length of a fixed-size string.

The strnlen_P() function is similar to strnlen(), except that `src` points to a string in the lower 64 KiB of program space.

Returns

The strnlen_P function returns strlen_P(src), if that is less than `len`, or `len` if there is no '\0' character among the first `len` characters pointed to by `src`.

20.22.3.95 strnlen_PF()

```
size_t strnlen_PF (
            uint_farptr_t s,
            size_t len )
```

Determine the length of a fixed-size string.

The strnlen_PF() function is similar to strnlen(), except that *s* is a far pointer to a string in program space.

Parameters

s	A far pointer to the string in Flash
len	The maximum number of length to return

Returns

The strnlen_PF function returns strlen_PF(*s*), if that is less than *len*, or *len* if there is no '\0' character among the first *len* characters pointed to by *s*.

20.22.3.96 strpbrk_P()

```
char * strpbrk_P (
            const char * s,
            const char * accept )
```

The strpbrk_P() function locates the first occurrence in the string `s` of any of the characters in the flash string `accept`. This function is similar to strpbrk() except that `accept` points to a string in the lower 64 KiB of program space.

Returns

The strpbrk_P() function returns a pointer to the character in `s` that matches one of the characters in `accept`, or `NULL` if no such character is found. The terminating zero is not considered as a part of string: if one or both args are empty, the result will `NULL`.

20.22.3.97 strrchr_P()

```
const char * strrchr_P (
            const char * s,
            int val )
```

Locate character in string.

The strrchr_P() function returns a pointer to the last occurrence of the character `val` in the string `s`. `s` is located in the lower 64 KiB of program memory.

Returns

The strrchr_P() function returns a pointer to the matched character or `NULL` if the character is not found.

20.22.3.98 strsep_P()

```
char * strsep_P (
            char ** sp,
            const char * delim )
```

Parse a string into tokens.

The strsep_P() function locates, in the string referenced by *sp, the first occurrence of any character in the string delim (or the terminating '\0' character) and replaces it with a '\0'. The location of the next character after the delimiter character (or NULL, if the end of the string was reached) is stored in *sp. An ``empty'' field, i.e. one caused by two adjacent delimiter characters, can be detected by comparing the location referenced by the pointer returned in *sp to '\0'. This function is similar to strsep() except that delim points to a string in the lower 64 KiB of program space.

Returns

The strsep_P() function returns a pointer to the original value of *sp. If *sp is initially NULL, strsep_P() returns NULL.

20.22.3.99 strspn_P()

```
size_t strspn_P (
            const char * s,
            const char * accept )
```

The strspn_P() function calculates the length of the initial segment of s which consists entirely of characters in accept. This function is similar to strspn() except that accept points to a string in the lower 64 KiB of program space.

Returns

The strspn_P() function returns the number of characters in the initial segment of s which consist only of characters from accept. The terminating zero is not considered as a part of string.

20.22.3.100 strstr_P()

```
char * strstr_P (
            const char * s1,
            const char * s2 )
```

Locate a substring.

The strstr_P() function finds the first occurrence of the substring s2 in the string s1. The terminating '\0' characters are not compared. The strstr_P() function is similar to strstr() except that s2 points to a string in the lower 64 KiB of program space.

Returns

The strstr_P() function returns a pointer to the beginning of the substring, or NULL if the substring is not found. If s2 points to a string of zero length, the function returns s1.

20.22.3.101 strstr_PF()

```
char * strstr_PF (
            const char * s1,
            uint_farptr_t s2 )
```

Locate a substring.

The strstr_PF() function finds the first occurrence of the substring `s2` in the string `s1`. The terminating '\0' characters are not compared. The strstr_PF() function is similar to strstr() except that `s2` is a far pointer to a string in program space.

Returns

The strstr_PF() function returns a pointer to the beginning of the substring, or NULL if the substring is not found. If `s2` points to a string of zero length, the function returns `s1`.

20.22.3.102 strtok_P()

```
char * strtok_P (
            char * s,
            const char * delim )
```

Parses the string into tokens.

strtok_P() parses the string `s` into tokens. The first call to strtok_P() should have `s` as its first argument. Subsequent calls should have the first argument set to NULL. If a token ends with a delimiter, this delimiting character is overwritten with a '\0' and a pointer to the next character is saved for the next call to strtok_P(). The delimiter string `delim` may be different for each call.

The strtok_P() function is similar to strtok() except that `delim` points to a string in the lower 64 KiB of program space.

Returns

The strtok_P() function returns a pointer to the next token or NULL when no more tokens are found.

Note

strtok_P() is NOT reentrant. For a reentrant version of this function see strtok_rP().

20.22.3.103 strtok_rP()

```
char * strtok_rP (
            char * string,
            const char * delim,
            char ** last )
```

Parses string into tokens.

The strtok_rP() function parses `string` into tokens. The first call to strtok_rP() should have string as its first argument. Subsequent calls should have the first argument set to NULL. If a token ends with a delimiter, this delimiting character is overwritten with a '\0' and a pointer to the next character is saved for the next call to strtok_rP(). The delimiter string `delim` may be different for each call. `last` is a user allocated char* pointer. It must be the same while parsing the same string. strtok_rP() is a reentrant version of strtok_P().

The strtok_rP() function is similar to strtok_r() except that `delim` points to a string in the lower 64 KiB of program space.

Returns

The strtok_rP() function returns a pointer to the next token or NULL when no more tokens are found.

20.23 <avr/power.h>: Power Reduction Management

Macros

- #define clock_prescale_get() (clock_div_t)(CLKPR & (uint8_t)((1<<CLKPS0)|(1<<CLKPS1)|(1<<CLKPS2)|(1<<CLKPS3))

Functions

- static void power_all_enable ()
- static void power_all_disable ()
- void clock_prescale_set (clock_div_t __x)

20.23.1 Detailed Description

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

Many AVRs contain a Power Reduction Register (PRR) or Registers (PRRx) that allow you to reduce power consumption by disabling or enabling various on-board peripherals as needed. Some devices have the XTAL Divide Control Register (XDIV) which offer similar functionality as System Clock Prescale Register (CLKPR).

There are many macros in this header file that provide an easy interface to enable or disable on-board peripherals to reduce power. See the table below.

Note

Not all AVR devices have a Power Reduction Register (for example the ATmega8). On those devices without a Power Reduction Register, the power reduction macros are not available..

Not all AVR devices contain the same peripherals (for example, the LCD interface), or they will be named differently (for example, USART and USART0). Please consult your device's datasheet, or the header file, to find out which macros are applicable to your device.

For device using the XTAL Divide Control Register (XDIV), when prescaler is used, Timer/Counter0 can only be used in asynchronous mode. Keep in mind that Timer/Counter0 source shall be less than ¼th of peripheral clock. Therefore, when using a typical 32.768 kHz crystal, one shall not scale the clock below 131.072 kHz.

Table 74 Power Macros

Power Macro	Description
power_aca_disable()	Disable the Analog Comparator on PortA
power_aca_enable()	Enable the Analog Comparator on PortA
power_adc_enable()	Enable the Analog to Digital Converter module
power_adc_disable()	Disable the Analog to Digital Converter module
power_adca_disable()	Disable the Analog to Digital Converter module on PortA
power_adca_enable()	Enable the Analog to Digital Converter module on PortA
power_evsys_disable()	Disable the EVSYS module
power_evsys_enable()	Enable the EVSYS module
power_hiresc_disable()	Disable the HIRES module on PortC
power_hiresc_enable()	Enable the HIRES module on PortC
power_lcd_enable()	Enable the LCD module
power_lcd_disable()	Disable the LCD module
power_pga_enable()	Enable the Programmable Gain Amplifier module
power_pga_disable()	Disable the Programmable Gain Amplifier module
power_pscr_enable()	Enable the Reduced Power Stage Controller module
power_pscr_disable()	Disable the Reduced Power Stage Controller module
power_psc0_enable()	Enable the Power Stage Controller 0 module

Power Macro	Description
power_psc0_disable()	Disable the Power Stage Controller 0 module
power_psc1_enable()	Enable the Power Stage Controller 1 module
power_psc1_disable()	Disable the Power Stage Controller 1 module
power_psc2_enable()	Enable the Power Stage Controller 2 module
power_psc2_disable()	Disable the Power Stage Controller 2 module
power_ram0_enable()	Enable the SRAM block 0
power_ram0_disable()	Disable the SRAM block 0
power_ram1_enable()	Enable the SRAM block 1
power_ram1_disable()	Disable the SRAM block 1
power_ram2_enable()	Enable the SRAM block 2
power_ram2_disable()	Disable the SRAM block 2
power_ram3_enable()	Enable the SRAM block 3
power_ram3_disable()	Disable the SRAM block 3
power_rtc_disable()	Disable the RTC module
power_rtc_enable()	Enable the RTC module
power_spi_enable()	Enable the Serial Peripheral Interface module
power_spi_disable()	Disable the Serial Peripheral Interface module
power_spic_disable()	Disable the SPI module on PortC
power_spic_enable()	Enable the SPI module on PortC
power_spid_disable()	Disable the SPI module on PortD
power_spid_enable()	Enable the SPI module on PortD
power_tc0c_disable()	Disable the TC0 module on PortC
power_tc0c_enable()	Enable the TC0 module on PortC
power_tc0d_disable()	Disable the TC0 module on PortD
power_tc0d_enable()	Enable the TC0 module on PortD
power_tc0e_disable()	Disable the TC0 module on PortE
power_tc0e_enable()	Enable the TC0 module on PortE
power_tc0f_disable()	Disable the TC0 module on PortF
power_tc0f_enable()	Enable the TC0 module on PortF
power_tc1c_disable()	Disable the TC1 module on PortC
power_tc1c_enable()	Enable the TC1 module on PortC
power_twic_disable()	Disable the Two Wire Interface module on PortC
power_twic_enable()	Enable the Two Wire Interface module on PortC
power_twie_disable()	Disable the Two Wire Interface module on PortE
power_twie_enable()	Enable the Two Wire Interface module on PortE
power_timer0_enable()	Enable the Timer 0 module
power_timer0_disable()	Disable the Timer 0 module
power_timer1_enable()	Enable the Timer 1 module
power_timer1_disable()	Disable the Timer 1 module
power_timer2_enable()	Enable the Timer 2 module
power_timer2_disable()	Disable the Timer 2 module
power_timer3_enable()	Enable the Timer 3 module
power_timer3_disable()	Disable the Timer 3 module
power_timer4_enable()	Enable the Timer 4 module
power_timer4_disable()	Disable the Timer 4 module
power_timer5_enable()	Enable the Timer 5 module
power_timer5_disable()	Disable the Timer 5 module
power_twi_enable()	Enable the Two Wire Interface module
power_twi_disable()	Disable the Two Wire Interface module
power_usart_enable()	Enable the USART module
power_usart_disable()	Disable the USART module
power_usart0_enable()	Enable the USART 0 module
power_usart0_disable()	Disable the USART 0 module
power_usart1_enable()	Enable the USART 1 module
power_usart1_disable()	Disable the USART 1 module
power_usart2_enable()	Enable the USART 2 module

Power Macro	Description
power_usart2_disable()	Disable the USART 2 module
power_usart3_enable()	Enable the USART 3 module
power_usart3_disable()	Disable the USART 3 module
power_usartc0_disable()	Disable the USART0 module on PortC
power_usartc0_enable()	Enable the USART0 module on PortC
power_usartd0_disable()	Disable the USART0 module on PortD
power_usartd0_enable()	Enable the USART0 module on PortD
power_usarte0_disable()	Disable the USART0 module on PortE
power_usarte0_enable()	Enable the USART0 module on PortE
power_usartf0_disable()	Disable the USART0 module on PortF
power_usartf0_enable()	Enable the USART0 module on PortF
power_usb_enable()	Enable the USB module
power_usb_disable()	Disable the USB module
power_usi_enable()	Enable the Universal Serial Interface module
power_usi_disable()	Disable the Universal Serial Interface module
power_vadc_enable()	Enable the Voltage ADC module
power_vadc_disable()	Disable the Voltage ADC module
power_all_enable()	Enable all modules
power_all_disable()	Disable all modules

Some of the newer AVRs contain a System Clock Prescale Register (CLKPR) that allows you to decrease the system clock frequency and the power consumption when the need for processing power is low. On some earlier AVRs (ATmega103, ATmega64, ATmega128), similar functionality can be achieved through the XTAL Divide Control Register. Below are two macros and an enumerated type that can be used to interface to the Clock Prescale Register or XTAL Divide Control Register.

Note

Not all AVR devices have a clock prescaler. On those devices without a Clock Prescale Register or XTAL Divide Control Register, these macros are not available.

```
typedef enum
{
    clock_div_1 = 0,
    clock_div_2 = 1,
    clock_div_4 = 2,
    clock_div_8 = 3,
    clock_div_16 = 4,
    clock_div_32 = 5,
    clock_div_64 = 6,
    clock_div_128 = 7,
    clock_div_256 = 8,
    clock_div_1_rc = 15, // ATmega128RFA1 only
} clock_div_t;
```

Clock prescaler setting enumerations for device using System Clock Prescale Register.

```
typedef enum
{
    clock_div_1 = 1,
    clock_div_2 = 2,
    clock_div_4 = 4,
    clock_div_8 = 8,
    clock_div_16 = 16,
    clock_div_32 = 32,
    clock_div_64 = 64,
    clock_div_128 = 128
} clock_div_t;
```

Clock prescaler setting enumerations for device using XTAL Divide Control Register.

20.23.2 Macro Definition Documentation

20.23.2.1 clock_prescale_get

```
#define clock_prescale_get( ) (clock_div_t)(CLKPR & (uint8_t)((1<<CLKPS0)|(1<<CLKPS1)|(1<<CLKPS2)|(1<<CLKPS3
```

Gets and returns the clock prescaler register setting. The return type is `clock_div_t`.

Note

For device with XTAL Divide Control Register (XDIV), return can actually range from 1 to 129. Care should be taken has the return value could differ from the typedef enum clock_div_t. This should only happen if clock_prescale_set was previously called with a value other than those defined by `clock_div_t`.

20.23.3 Function Documentation

20.23.3.1 clock_prescale_set()

```
clock_prescale_set (
            clock_div_t x )
```

Set the clock prescaler register select bits, selecting a system clock division setting. This function is inlined, even if compiler optimizations are disabled.

The type of `x` is `clock_div_t`.

Note

For device with XTAL Divide Control Register (XDIV), `x` can actually range from 1 to 129. Thus, one does not need to use `clock_div_t` type as argument.

20.23.3.2 power_all_disable()

```
void power_all_disable ( )  [inline], [static]
```

Disable all modules.

20.23.3.3 power_all_enable()

```
void power_all_enable ( )  [inline], [static]
```

Enable all modules.

20.24 Additional notes from <avr/sfr_defs.h>

The <avr/sfr_defs.h> file is included by all of the <avr/ioXXXX.h> files, which use macros defined here to make the special function register definitions look like C variables or simple constants, depending on the _SFR_ASM_COMPAT define. Some examples from <avr/iocanxx.h> to show how to define such macros:

```
#define PORTA   _SFR_IO8(0x02)
#define EEAR    _SFR_IO16(0x21)
#define UDR0    _SFR_MEM8(0xC6)
#define TCNT3   _SFR_MEM16(0x94)
#define CANIDT  _SFR_MEM32(0xF0)
```

If _SFR_ASM_COMPAT is not defined, C programs can use names like PORTA directly in C expressions (also on the left side of assignment operators) and GCC will do the right thing (use short I/O instructions if possible). The __SFR_OFFSET definition is not used in any way in this case.

Define _SFR_ASM_COMPAT as 1 to make these names work as simple constants (addresses of the I/O registers). This is necessary when included in preprocessed assembler (*.S) source files, so it is done automatically if _↩ _ASSEMBLER__ is defined. By default, all addresses are defined as if they were memory addresses (used in lds/sts instructions). To use these addresses in in/out instructions, you must subtract 0x20 from them.

For more backwards compatibility, insert the following at the start of your old assembler source file:

```
#define __SFR_OFFSET 0
```

This automatically subtracts 0x20 from I/O space addresses, but it's a hack, so it is recommended to change your source: wrap such addresses in macros defined here, as shown below. After this is done, the __SFR_OFFSET definition is no longer necessary and can be removed.

Real example - this code could be used in a boot loader that is portable between devices with SPMCR at different addresses.

```
<avr/iom163.h>: #define SPMCR _SFR_IO8(0x37)
<avr/iom128.h>: #define SPMCR _SFR_MEM8(0x68)

#if _SFR_IO_REG_P(SPMCR)
    out _SFR_IO_ADDR(SPMCR), r24
#else
    sts _SFR_MEM_ADDR(SPMCR), r24
#endif
```

You can use the in/out/cbi/sbi/sbic/sbis instructions, without the _SFR_IO_REG_P test, if you know that the register is in the I/O space (as with SREG, for example). If it isn't, the assembler will complain (I/O address out of range 0...0x3f), so this should be fairly safe.

If you do not define __SFR_OFFSET (so it will be 0x20 by default), all special register addresses are defined as memory addresses (so SREG is 0x5f), and (if code size and speed are not important, and you don't like the ugly #if above) you can always use lds/sts to access them. But, this will not work if __SFR_OFFSET != 0x20, so use a different macro (defined only if __SFR_OFFSET == 0x20) for safety:

```
sts _SFR_ADDR(SPMCR), r24
```

In C programs, all 3 combinations of _SFR_ASM_COMPAT and __SFR_OFFSET are supported - the _SFR_↩ ADDR(SPMCR) macro can be used to get the address of the SPMCR register (0x57 or 0x68 depending on device).

20.25 <avr/sfr_defs.h>: Special function registers

Modules

- Additional notes from <avr/sfr_defs.h>

Bit manipulation

- #define _BV(bit) (1 << (bit))

IO register bit manipulation

- #define bit_is_set(sfr, bit) (_SFR_BYTE(sfr) & _BV(bit))
- #define bit_is_clear(sfr, bit) (!(_SFR_BYTE(sfr) & _BV(bit)))
- #define loop_until_bit_is_set(sfr, bit) do { } while (bit_is_clear(sfr, bit))
- #define loop_until_bit_is_clear(sfr, bit) do { } while (bit_is_set(sfr, bit))

20.25.1 Detailed Description

When working with microcontrollers, many tasks usually consist of controlling internal peripherals, or external peripherals that are connected to the device. The entire IO address space is made available as *memory-mapped IO*, i.e. it can be accessed using all the MCU instructions that are applicable to normal data memory. For most AVR devices, the IO register space is mapped into the data memory address space with an offset of 0x20 since the bottom of this space is reserved for direct access to the MCU registers. (Actual SRAM is available only behind the IO register area, starting at some specific address depending on the device.)

For example the user can access memory-mapped IO registers as if they were globally defined variables like this:

```
PORTA = 0x33;
unsigned char foo = PINA;
```

The compiler will choose the correct instruction sequence to generate based on the address of the register being accessed.

The advantage of using the memory-mapped registers in C programs is that it makes the programs more portable to other C compilers for the AVR platform.

Note that special care must be taken when accessing some of the 16-bit timer IO registers where access from both the main program and within an interrupt context can happen. See Why do some 16-bit timer registers sometimes get trashed?.

Porting programs that use the deprecated sbi/cbi macros

Access to the AVR single bit set and clear instructions are provided via the standard C bit manipulation commands. The sbi and cbi macros are no longer directly supported. sbi (sfr,bit) can be replaced by sfr |= _BV(bit) .

i.e.: sbi(PORTB, PB1); is now PORTB |= _BV(PB1);

This actually is more flexible than having sbi directly, as the optimizer will use a hardware sbi if appropriate, or a read/or/write operation if not appropriate. You do not need to keep track of which registers sbi/cbi will operate on.

Likewise, cbi (sfr,bit) is now sfr &= ~(_BV(bit));

20.25.2 Macro Definition Documentation

20.25.2.1 _BV

```
#define _BV(
              bit ) (1 << (bit))
#include <avr/io.h>
```

Converts a bit number into a byte value.

Note

The bit shift is performed by the compiler which then inserts the result into the code. Thus, there is no run-time overhead when using _BV().

20.25.2.2 bit_is_clear

```
#define bit_is_clear(
              sfr,
              bit ) (!(_SFR_BYTE(sfr) & _BV(bit)))
#include <avr/io.h>
```

Test whether bit `bit` in IO register `sfr` is clear. This will return non-zero if the bit is clear, and a 0 if the bit is set.

20.25.2.3 bit_is_set

```
#define bit_is_set(
              sfr,
              bit ) (_SFR_BYTE(sfr) & _BV(bit))
#include <avr/io.h>
```

Test whether bit `bit` in IO register `sfr` is set. This will return a 0 if the bit is clear, and non-zero if the bit is set.

20.25.2.4 loop_until_bit_is_clear

```
#define loop_until_bit_is_clear(
              sfr,
              bit ) do { } while (bit_is_set(sfr, bit))
#include <avr/io.h>
```

Wait until bit `bit` in IO register `sfr` is clear.

20.25.2.5 loop_until_bit_is_set

```
#define loop_until_bit_is_set(
              sfr,
              bit ) do { } while (bit_is_clear(sfr, bit))
#include <avr/io.h>
```

Wait until bit `bit` in IO register `sfr` is set.

20.26 <avr/signature.h>: Signature Support

Introduction

The <avr/signature.h> header file allows the user to automatically and easily include the device's signature data in a special section of the final linked ELF file.

This value can then be used by programming software to compare the on-device signature with the signature recorded in the ELF file to look for a match before programming the device.

API Usage Example

Usage is very simple; just include the header file:

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

This will declare a constant unsigned char array and it is initialized with the three signature bytes, MSB first, that are defined in the device I/O header file. This array is then placed in the .signature section in the resulting linked ELF file.

The three signature bytes that are used to initialize the array are these defined macros in the device I/O header file, from MSB to LSB: SIGNATURE_2, SIGNATURE_1, SIGNATURE_0.

This header file should only be included once in an application.

20.27 <avr/sleep.h>: Power Management and Sleep Modes

Functions

- void set_sleep_mode (uint8_t mode)
- void sleep_enable (void)
- void sleep_disable (void)
- void sleep_cpu (void)
- void sleep_mode (void)
- void sleep_bod_disable (void)

20.27.1 Detailed Description

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

Use of the `SLEEP` instruction can allow an application to reduce its power consumption considerably. AVR devices can be put into different sleep modes. Refer to the datasheet for the details relating to the device you are using.

There are several macros provided in this header file to actually put the device into sleep mode. The simplest way is to optionally set the desired sleep mode using `set_sleep_mode()` (it usually defaults to idle mode where the CPU is put on sleep but all peripheral clocks are still running), and then call `sleep_mode()`. This macro automatically sets the sleep enable bit, goes to sleep, and clears the sleep enable bit.

Example:

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

...
  set_sleep_mode(<mode>);
  sleep_mode();
```

Note that unless your purpose is to completely lock the CPU (until a hardware reset), interrupts need to be enabled before going to sleep.

As the `sleep_mode()` macro might cause race conditions in some situations, the individual steps of manipulating the sleep enable (SE) bit, and actually issuing the `SLEEP` instruction, are provided in the macros `sleep_enable()`, `sleep_disable()`, and `sleep_cpu()`. This also allows for test-and-sleep scenarios that take care of not missing the interrupt that will awake the device from sleep.

Example:

```
#include <avr/interrupt.h>
#include <avr/sleep.h>

...
  set_sleep_mode(<mode>);
  cli();
  if (some_condition)
  {
    sleep_enable();
    sei();
    sleep_cpu();
    sleep_disable();
  }
  sei();
```

This sequence ensures an atomic test of `some_condition` with interrupts being disabled. If the condition is met, sleep mode will be prepared, and the `SLEEP` instruction will be scheduled immediately after an `SEI` instruction. As the instruction right after the `SEI` is guaranteed to be executed before an interrupt could trigger, it is sure the device will really be put to sleep.

Some devices have the ability to disable the Brown Out Detector (BOD) before going to sleep. This will also reduce power while sleeping. If the specific AVR device has this ability then an additional macro is defined: `sleep_bod_disable()`. This macro generates inlined assembly code that will correctly implement the timed sequence for disabling the BOD before sleeping. However, there is a limited number of cycles after the BOD has been disabled that the device can be put into sleep mode, otherwise the BOD will not truly be disabled. Recommended practice is to disable the BOD (`sleep_bod_disable()`), set the interrupts (`sei()`), and then put the device to sleep (`sleep_cpu()`), like so:

```
#include <avr/interrupt.h>
#include <avr/sleep.h>

...
  set_sleep_mode(<mode>);
  cli();
  if (some_condition)
  {
    sleep_enable();
    sleep_bod_disable();
    sei();
    sleep_cpu();
    sleep_disable();
  }
  sei();
```

20.27.2 Function Documentation

20.27.2.1 set_sleep_mode()

```
void set_sleep_mode (
             uint8_t mode )
```

Set the sleep mode control register to the specified sleep mode *mode*. The name of the sleep mode register depends on the device, see the data sheet for details.

Parameters

mode	The sleep mode to be set. The available sleep modes like SLEEP_MODE_PWR_SAVE, SLEEP_MODE_PWR_DOWN, SLEEP_MODE_PWR_OFF etc. depend on the device and are defined in avr/io.h.

20.27.2.2 sleep_bod_disable()

```
void sleep_bod_disable (
             void  )
```

Disable BOD before going to sleep. Not available on all devices.

20.27.2.3 sleep_cpu()

```
void sleep_cpu (
             void  )
```

Put the device into sleep mode. The SE bit must be set beforehand, and it is recommended to clear it afterwards.

20.27.2.4 sleep_disable()

```
void sleep_disable (
             void  )
```

Clear the SE (sleep enable) bit.

20.27.2.5 sleep_enable()

```
void sleep_enable (
             void  )
```

Put the device in sleep mode. How the device is brought out of sleep mode depends on the specific mode selected with the set_sleep_mode() function. See the data sheet for your device for more details.

Set the SE (sleep enable) bit.

20.27.2.6 sleep_mode()

```
void sleep_mode (
             void  )
```

Put the device into sleep mode, taking care of setting the SE bit before, and clearing it afterwards. All this command does is to run

```
sleep_enable()
sleep_cpu()
sleep_disable()
```

20.28 <avr/version.h>: AVR-LibC version macros

Macros

- #define __AVR_LIBC_VERSION_STRING__ "2.3.0"
- #define __AVR_LIBC_VERSION__ 20300UL
- #define __AVR_LIBC_DATE_STRING__ "20251223"
- #define __AVR_LIBC_DATE__ 20251223UL
- #define __AVR_LIBC_MAJOR__ 2
- #define __AVR_LIBC_MINOR__ 3
- #define __AVR_LIBC_REVISION__ 0

20.28.1 Detailed Description

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

This header file defines macros that contain version numbers and strings describing the current version of AVR-↩ LibC.

The version number itself basically consists of three pieces that are separated by a dot: the major number, the minor number, and the revision number. For development versions (which use an odd minor number), the string representation additionally gets the date code (YYYYMMDD) appended.

This file will also be included by <avr/io.h>. That way, portable tests can be implemented using <avr/io.h> that can be used in code that wants to remain backwards-compatible to library versions prior to the date when the library version API had been added, as referenced but undefined C preprocessor macros automatically evaluate to 0.

20.28.2 Macro Definition Documentation

20.28.2.1 __AVR_LIBC_DATE__

```
#define __AVR_LIBC_DATE__ 20251223UL
```

Numerical representation of the release date.

20.28.2.2 __AVR_LIBC_DATE_STRING__

```
#define __AVR_LIBC_DATE_STRING__ "20251223"
```

String literal representation of the release date.

20.28.2.3 __AVR_LIBC_MAJOR__

```
#define __AVR_LIBC_MAJOR__ 2
```

Library major version number.

20.28.2.4 __AVR_LIBC_MINOR__

```
#define __AVR_LIBC_MINOR__ 3
```

Library minor version number.

20.28.2.5 __AVR_LIBC_REVISION__

```
#define __AVR_LIBC_REVISION__ 0
```

Library revision number.

20.28.2.6 __AVR_LIBC_VERSION__

```
#define __AVR_LIBC_VERSION__ 20300UL
```

Numerical representation of the current library version.

In the numerical representation, the major number is multiplied by 10000, the minor number by 100, and all three parts are then added. It is intended to provide a monotonically increasing numerical value that can easily be used in numerical checks.

20.28.2.7 __AVR_LIBC_VERSION_STRING__

```
#define __AVR_LIBC_VERSION_STRING__ "2.3.0"
```

String literal representation of the current library version.

20.29 <avr/builtins.h>: avr-gcc builtins documentation

Functions

- void __builtin_avr_sei (void)
- void __builtin_avr_cli (void)
- void __builtin_avr_sleep (void)
- void __builtin_avr_wdr (void)
- uint8_t __builtin_avr_swap (uint8_t __b)
- uint16_t __builtin_avr_fmul (uint8_t __a, uint8_t __b)
- int16_t __builtin_avr_fmuls (int8_t __a, int8_t __b)
- int16_t __builtin_avr_fmulsu (int8_t __a, uint8_t __b)
- float __builtin_powif (float base, int expo)
- double __builtin_powi (double base, int expo)
- long double __builtin_powil (long double base, int expo)

20.29.1 Detailed Description

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

Note

This file only documents some avr-gcc builtins. For functions built-in in the compiler, there should be no prototype declarations.

See also the GCC documentation for a full list of avr-gcc builtins.

20.29.2 Function Documentation

20.29.2.1 __builtin_avr_cli()

```
void __builtin_avr_cli (
             void  )
```

Disables all interrupts by clearing the global interrupt mask.

20.29.2.2 __builtin_avr_fmul()

```
uint16_t __builtin_avr_fmul (
             uint8_t __a,
             uint8_t __b )
```

Emits an FMUL (fractional multiply unsigned) instruction.

20.29.2.3 __builtin_avr_fmuls()

```
int16_t __builtin_avr_fmuls (
             int8_t __a,
             int8_t __b )
```

Emits an FMUL (fractional multiply signed) instruction.

20.29.2.4 __builtin_avr_fmulsu()

```
int16_t __builtin_avr_fmulsu (
             int8_t __a,
             uint8_t __b )
```

Emits an FMUL (fractional multiply signed with unsigned) instruction.

20.29.2.5 __builtin_avr_sei()

```
void __builtin_avr_sei (
             void  )
```

Enables interrupts by setting the global interrupt mask.

20.29.2.6 __builtin_avr_sleep()

```
void __builtin_avr_sleep (
             void  )
```

Emits a `SLEEP` instruction.

20.29.2.7 __builtin_avr_swap()

```
uint8_t __builtin_avr_swap (
              uint8_t __b )
```

Emits a SWAP (nibble swap) instruction on __b.

20.29.2.8 __builtin_avr_wdr()

```
void __builtin_avr_wdr (
              void  )
```

Emits a WDR (watchdog reset) instruction.

20.29.2.9 __builtin_powi()

```
double __builtin_powi (
              double base,
              int expo )
```

Returns *base* raised to the power of *expo*. Since avr-gcc v15 this function is implemented in assembly.

20.29.2.10 __builtin_powif()

```
float __builtin_powif (
              float base,
              int expo )
```

Returns *base* raised to the power of *expo*. Since avr-gcc v15 this function is implemented in assembly. See also the benchmarks.

20.29.2.11 __builtin_powil()

```
long double __builtin_powil (
              long double base,
              int expo )
```

Returns *base* raised to the power of *expo*. Since avr-gcc v15 this function is implemented in assembly. See also the benchmarks.

20.30 <avr/wdt.h>: Watchdog timer handling

Macros

- #define wdt_reset() __asm__ __volatile__ ("wdr")
- #define wdt_enable(timeout)
- #define WDTO_8MS -1
- #define WDTO_15MS 0
- #define WDTO_30MS 1
- #define WDTO_60MS 2
- #define WDTO_120MS 3
- #define WDTO_250MS 4
- #define WDTO_500MS 5
- #define WDTO_1S 6
- #define WDTO_2S 7
- #define WDTO_4S 8
- #define WDTO_8S 9

Functions

- static void wdt_enable (const uint8_t value)
- static void wdt_disable (void)

20.30.1 Detailed Description

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

This header file declares the interface to some inline macros handling the watchdog timer present in many AVR devices. In order to prevent the watchdog timer configuration from being accidentally altered by a crashing application, a special timed sequence is required in order to change it. The macros within this header file handle the required sequence automatically before changing any value. Interrupts will be disabled during the manipulation.

Note

Depending on the fuse configuration of the particular device, further restrictions might apply, in particular it might be disallowed to turn off the watchdog timer.

Note that for newer devices (ATmega88 and newer, effectively any AVR that has the option to also generate interrupts), the watchdog timer remains active even after a system reset (except a power-on condition), using the fastest prescaler value (approximately 15 ms). It is therefore required to turn off the watchdog early during program startup, the datasheet recommends a sequence like the following:

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

uint8_t mcusr_mirror __attribute__((section (".noinit")));

__attribute__((used, unused, naked, section(".init3")))
static void get_mcusr (void)
{
    mcusr_mirror = MCUSR;
    MCUSR = 0;
    wdt_disable();
}
```

Saving the value of MCUSR in `mcusr_mirror` is only needed if the application later wants to examine the reset source, but in particular, clearing the watchdog reset flag before disabling the watchdog is required, according to the datasheet.

20.30.2 Macro Definition Documentation

20.30.2.1 wdt_enable

```
#define wdt_enable(
              timeout )
```

Enable the watchdog timer, configuring it for expiry after `timeout` (which is a combination of the `WDP0` through `WDP2` bits to write into the `WDTCR` register; For those devices that have a `WDTCSR` register, it uses the combination of the `WDP0` through `WDP3` bits).

See also the symbolic constants `WDTO_15MS` et al.

20.30.2.2 wdt_reset

```
#define wdt_reset( ) __asm__ __volatile__ ("wdr")
```

Reset the watchdog timer. When the watchdog timer is enabled, a call to this instruction is required before the timer expires, otherwise a watchdog-initiated device reset will occur.

20.30.2.3 WDTO_120MS

```
#define WDTO_120MS 3
```

See WDTO_15MS

20.30.2.4 WDTO_15MS

```
#define WDTO_15MS 0
```

A value to be passed to wdt_enable() for the specified watchdog timeout duration.

20.30.2.5 WDTO_1S

```
#define WDTO_1S 6
```

See WDTO_15MS

20.30.2.6 WDTO_250MS

```
#define WDTO_250MS 4
```

See WDTO_15MS

20.30.2.7 WDTO_2S

```
#define WDTO_2S 7
```

See WDTO_15MS

20.30.2.8 WDTO_30MS

```
#define WDTO_30MS 1
```

See WDTO_15MS

20.30.2.9 WDTO_4S

```
#define WDTO_4S 8
```

See WDTO_15MS Note: This is only available on the ATtiny2313, ATtiny24, ATtiny44, ATtiny84, ATtiny84A, ATtiny25, ATtiny45, ATtiny85, ATtiny261, ATtiny461, ATtiny861, ATmega48∗, ATmega88∗, ATmega168∗, ATmega328∗, ATmega164P, ATmega324P, ATmega324PB, ATmega644P, ATmega644, ATmega640, ATmega1280, ATmega1281, ATmega2560, ATmega2561, ATmega8HVA, ATmega16HVA, ATmega32HVB, ATmega406, ATmega1284P, AT90PWM1, AT90PWM2, AT90PWM2B, AT90PWM3, AT90PWM3B, AT90PWM216, AT90PWM316, AT90PWM81, AT90PWM161, AT90USB82, AT90USB162, AT90USB646, AT90USB647, AT90USB1286, AT90USB1287, ATtiny48, ATtiny88.

Note: This value does *not* match the bit pattern of the respective control register. It is solely meant to be used together with wdt_enable().

20.30.2.10 WDTO_500MS

```
#define WDTO_500MS 5
```

See WDTO_15MS

20.30.2.11 WDTO_60MS

```
#define WDTO_60MS 2
```

See WDTO_15MS

20.30.2.12 WDTO_8MS

```
#define WDTO_8MS -1
```

Symbolic constants for the watchdog timeout. Since the watchdog timer is based on a free-running RC oscillator, the times are approximate only and apply to a supply voltage of 5 V. At lower supply voltages, the times will increase. For older devices, the times will be as large as three times when operating at Vcc = 3 V, while the newer devices (e. g. ATmega128, ATmega8) only experience a negligible change.

Possible timeout values are: 15 ms, 30 ms, 60 ms, 120 ms, 250 ms, 500 ms, 1 s, 2 s. (Some devices also allow for 4 s and 8 s.) Symbolic constants are formed by the prefix WDTO_, followed by the time.

Example that would select a watchdog timer expiry of approximately 500 ms:

```
wdt_enable(WDTO_500MS);
```

20.30.2.13 WDTO_8S

```
#define WDTO_8S 9
```

See WDTO_15MS Note: This is only available on the ATtiny2313, ATtiny24, ATtiny44, ATtiny84, ATtiny84A, ATtiny25, ATtiny45, ATtiny85, ATtiny261, ATtiny461, ATtiny861, ATmega48*, ATmega88*, ATmega168*, ATmega328*, ATmega164P, ATmega324P, ATmega324PB, ATmega644P, ATmega644, ATmega640, ATmega1280, ATmega1281, ATmega2560, ATmega2561, ATmega8HVA, ATmega16HVA, ATmega32HVB, ATmega406, ATmega1284P, ATmega2564RFR2, ATmega256RFR2, ATmega1284RFR2, ATmega128RFR2, ATmega644RFR2, ATmega64↩ RFR2 AT90PWM1, AT90PWM2, AT90PWM2B, AT90PWM3, AT90PWM3B, AT90PWM216, AT90PWM316, AT90PWM81, AT90PWM161, AT90USB82, AT90USB162, AT90USB646, AT90USB647, AT90USB1286, AT90↩ USB1287, ATtiny48, ATtiny88, ATxmega16a4u, ATxmega32a4u, ATxmega16c4, ATxmega32c4, ATxmega128c3, ATxmega192c3, ATxmega256c3.

Note: This value does *not* match the bit pattern of the respective control register. It is solely meant to be used together with wdt_enable().

20.30.3 Function Documentation

20.30.3.1 wdt_disable()

```
static void wdt_disable (
             void  )  [inline], [static]
```

Disable the watchdog timer.

20.30.3.2 wdt_enable()

```
static void wdt_enable (
             const uint8_t value )  [inline], [static]
```

Enable the watchdog timer, configuring it for expiry after `timeout` (which is a combination of the `WDP0` through `WDP2` bits to write into the `WDTCR` register; For those devices that have a `WDTCSR` register, it uses the combination of the `WDP0` through `WDP3` bits).

See also the symbolic constants `WDTO_15MS` et al.

20.31 <util/delay.h>: Convenience functions for busy-wait delay loops

Macros

- #define F_CPU 1000000UL

Functions

- static void _delay_ms (double __ms)
- static void _delay_us (double __us)

20.31.1 Detailed Description

```
#define F_CPU 1000000UL  // 1 MHz
//#define F_CPU 14.7456e6
#include <util/delay.h>
```

The functions in this header are meant as convenience functions where actual time values can be specified as a delay time, rather than a number of cycles to wait for.

This requires that the clock frequency of the device is provided in the `F_CPU` macro in units of Hertz. The macro must be defined before including the <util/delay.h> header. It can be defined in the source code like indicated above, or it can be defined on the command line / in a Makefile by means of `-D F_CPU=...`

The functions in this header file are wrappers around the basic busy-wait functions from `<util/delay_basic.h>`, or, when supported by the compiler, then `__builtin_avr_delay_cycles()` is used instead.

In any case, the delay functions provided by this header will not disable interrupts, which means hat the delay time will be longer than specified when interrupts occur while a delay function is running.

In order for these functions to work as intended, compiler optimizations *must* be enabled, and the delay time *must* be an expression that is a known constant at compile-time. If these requirements are not met, the resulting delay will be much longer (and basically unpredictable), and applications that otherwise do not use floating-point calculations will experience severe code bloat by the floating-point library routines linked into the application. The idea is that compile-time constant expressions will be eliminated by compiler optimization so floating-point expressions can be used to calculate the number of delay cycles needed based on the CPU frequency passed by the macro F_CPU.

The functions available allow the specification of microsecond, and millisecond delays directly, using the application-supplied macro F_CPU as the CPU clock frequency in Hertz.

20.31.2 Macro Definition Documentation

20.31.2.1 F_CPU

```
#define F_CPU 1000000UL
```

CPU frequency in Hz.

The macro F_CPU specifies the CPU frequency in Hertz to be considered by the delay functions. This macro is normally supplied by the environment (e.g. from within a project header, or the project's Makefile). The value 1 MHz here is only provided as a fallback default if no such user-provided definition could be found.

In terms of the delay functions, the CPU frequency can be given as a floating-point constant (e.g. 3.6864e6 for 3.6864 MHz). However, the macros in <util/setbaud.h> require it to be an integer value.

20.31.3 Function Documentation

20.31.3.1 _delay_ms()

```
static void _delay_ms (
             double __ms )  [inline], [static]
```

Perform a delay of `__ms` milliseconds.

The macro F_CPU is supposed to be defined to a constant defining the CPU clock frequency in Hertz.

- If `__builtin_avr_delay_cycles()` is supported by the compiler, then the maximal possible delay is 4294967.04 / f_{CPU} milliseconds where f_{CPU} denotes the CPU frequency in units of 1 MHz. This is around 71 minutes / f_{CPU}. Values greater than that are saturated to this value.
- Otherwise, _delay_loop_2() is used as a fallback, and the maximal possible delay is 262.14 / f_{CPU} milliseconds. When the user requests a delay which exceeds the maximum possible one, _delay_ms() provides a decreased resolution functionality. In this mode _delay_ms() will work with a resolution of 1/10 ms, providing delays up to 6.5535 seconds (independent from CPU frequency). The user will not be informed about decreased resolution.

Conversion of `__ms` into clock cycles may not always result in an integral value. By default, the clock cycles are rounded up to the next integer. This ensures that the user gets at least `__ms` microseconds of delay. Alternatively, by defining the macro `__DELAY_ROUND_DOWN__`, or `__DELAY_ROUND_CLOSEST__`, before including this header file, the algorithm can be made to round down, or round to closest integer, respectively.

Note

The implementation of _delay_ms() based on `__builtin_avr_delay_cycles()` is not backward compatible with older implementations. In order to get a functionality backward compatible with previous versions, the macro `__DELAY_BACKWARD_COMPATIBLE__` must be defined before including this header file.

20.31.3.2 _delay_us()

```
static void _delay_us (
             double __us )  [inline], [static]
```

Perform a delay of __us microseconds.

The macro F_CPU is supposed to be defined to a constant defining the CPU clock frequency in Hertz.

- If __builtin_avr_delay_cycles() is supported by the compiler, then the maximal possible delay is 4294967040 / f_{CPU} microseconds where f_{CPU} denotes the CPU frequency in units of 1 MHz. This is around 71 minutes / f_{CPU}. Values greater than that are saturated to this value.
- Otherwise, _delay_loop_1() is used as a fallback, and the maximal possible delay is 768 / f_{CPU} microseconds. If the user requests a delay greater than the maximal possible one, _delay_us() will automatically call _delay_ms() instead. The user will not be informed about this case.

Conversion of __us into clock cycles may not always result in an integral value. By default, the clock cycles are rounded up to next integer. This ensures that the user gets at least __us microseconds of delay. Alternatively, by defining the macro __DELAY_ROUND_DOWN__, or __DELAY_ROUND_CLOSEST__, before including this header file, the algorithm can be made to round down, or round to closest integer, respectively.

Note

The implementation of _delay_us() based on __builtin_avr_delay_cycles() is not backward compatible with older implementations. In order to get a functionality backward compatible with previous versions, the macro __DELAY_BACKWARD_COMPATIBLE__ must be defined before including this header file.

20.32 <util/ram-usage.h>: Determine dynamic RAM usage

Functions

- uint16_t _get_ram_unused (void)

Symbols

- __heap_start
- __ram_color_end
- __ram_color_value

20.32.1 Detailed Description

```
#include <util/ram-usage.h>
```

This header provides a single function, _get_ram_unused(), that can be used during development to get a rough estimate of how much RAM might be used by an application. This works as follows:

1. The startup code paints the RAM location with a specific value. This happens in .init3, and only when _get_ram_unused() is actually used. The coloring extends from __heap_start (which is the RAM location right after static storage) up to and including RAMEND as defined in <avr/io.h>.
2. The application calls _get_ram_unused() and determines how much of the colored bytes are still intact. This can be used to calculate a **lower bound** of how much RAM is used by the application.

Since

AVR-LibC v2.3

20.32.2 Function Documentation

20.32.2.1 _get_ram_unused()

```
uint16_t _get_ram_unused (
             void  )  [inline]
```

Determines how much of the initial RAM coloring is still intact. It can be used to get a **lower bound** of how much RAM might be used by an application.

The function is written in such a way that it has a small register foot print, so that it is not a big issue to call it in an ISR. Though that's not required for the intended purpose: _get_ram_unused() can simply be called after some time has elapsed and enough ISRs and other functions have been invoked.

Returns

The value returned by _get_ram_unused() is an *upper bound* for how much bytes of RAM are still *unused at the time of invocation*.
The return value will never increase with time (except for very rare occasions where __ram_color_value is written to the top of the stack).

Limitations

- The start of the coloring is hard coded as __heap_start, which is the beginning of the RAM area after static storage.
- The algorithm assumes that the stack is located *after static storage* and grows downwards towards __heap_start.
- The current implementation is **not compatible with malloc** et al. (alloca() is no problem though, since it allocates on the stack.)

20.32.3 Variable Documentation

20.32.3.1 __heap_start

```
__heap_start  [extern]
```

A symbol defined in the default linker script. It points one byte past static storage (.data, .bss, .noinit).

20.32.3.2 __ram_color_end

```
__ram_color_end
```

A weak symbol that defaults to `RAMEND + 1`. It points one byte past the last location that is colored by the startup code. It can be adjusted by say

```
avr-gcc ... -Wl,-defsym,__ram_color_end=<value>
```

in the link command, or by means of a global inline assembly statement like:

```
__asm (".global __ram_color_end\n"
       "__ram_color_end = <value>");
```

20.32.3.3 __ram_color_value

```
__ram_color_value
```

A weak symbol that defaults to 0xaa. It represents the "color" that's being used by the startup code to paint the RAM, and the value that _get_ram_unused() will check against. It can be adjusted by say

```
avr-gcc ... -Wl,-defsym,__ram_color_value=<value>
```

in the link command.

20.33 <util/atomic.h> Atomically and Non-Atomically Executed Code Blocks

Macros

- #define ATOMIC_BLOCK(type)
- #define NONATOMIC_BLOCK(type)
- #define ATOMIC_RESTORESTATE
- #define ATOMIC_FORCEON
- #define NONATOMIC_RESTORESTATE
- #define NONATOMIC_FORCEOFF

20.33.1 Detailed Description

```
#include <util/atomic.h>
```

Note

> The macros in this header file require the ISO/IEC 9899:1999 ("ISO C99") feature of for loop variables that are declared inside the for loop itself. For that reason, this header file can only be used if the standard level of the compiler (option –std=) is set to either `c99`, `gnu99` or higher.

The macros in this header file deal with code blocks that are guaranteed to be executed Atomically or Non-↩ Atomically. The term "Atomic" in this context refers to the inability of the respective code to be interrupted.

These macros operate via automatic manipulation of the Global Interrupt Status (I) bit of the SREG register. Exit paths from both block types are all managed automatically without the need for special considerations, i.e. the interrupt status will be restored to the same value it had when entering the respective block (unless ATOMIC_↩ FORCEON or NONATOMIC_FORCEOFF are used).

Warning

> The features in this header are implemented by means of a for loop. This means that commands like `break` and `continue` that are located in an atomic block refer to the atomic for loop, not to a loop construct that hosts the atomic block.

A typical example that requires atomic access is a 16 (or more) bit variable that is shared between the main execution path and an ISR. While declaring such a variable as volatile ensures that the compiler will not optimize accesses to it away, it does not guarantee atomic access to it. Assuming the following example:

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

volatile uint16_t ctr;

ISR(TIMER1_OVF_vect)
```

```
{
  ctr--;
}

...
int
main(void)
{
   ...
   ctr = 0x200;
   start_timer();
   while (ctr != 0)
     // wait
       ;
   ...
}
```

There is a chance where the main context will exit its wait loop when the variable `ctr` just reached the value 0xFF. This happens because the compiler cannot natively access a 16-bit variable atomically in an 8-bit CPU. So the variable is for example at 0x100, the compiler then tests the low byte for 0, which succeeds. It then proceeds to test the high byte, but that moment the ISR triggers, and the main context is interrupted. The ISR will decrement the variable from 0x100 to 0xFF, and the main context proceeds. It now tests the high byte of the variable which is (now) also 0, so it concludes the variable has reached 0, and terminates the loop.

Using the macros from this header file, the above code can be rewritten like:

```
#include <stdint.h>
#include <avr/interrupt.h>
#include <avr/io.h>
#include <util/atomic.h>

volatile uint16_t ctr;

ISR(TIMER1_OVF_vect)
{
  ctr--;
}

...
int
main(void)
{
   ...
   ctr = 0x200;
   start_timer();
   sei();
   uint16_t ctr_copy;
   do
   {
     ATOMIC_BLOCK(ATOMIC_FORCEON)
     {
       ctr_copy = ctr;
     }
   }
   while (ctr_copy != 0);
   ...
}
```

This will install the appropriate interrupt protection before accessing variable `ctr`, so it is guaranteed to be consistently tested. If the global interrupt state were uncertain before entering the ATOMIC_BLOCK, it should be executed with the parameter ATOMIC_RESTORESTATE rather than ATOMIC_FORCEON.

See Problems with Reordering Code for things to be taken into account with respect to compiler optimizations.

20.33.2 Macro Definition Documentation

20.33.2.1 ATOMIC_BLOCK

```
#define ATOMIC_BLOCK(
              type )
```

Creates a block of code that is guaranteed to be executed atomically. Upon entering the block the Global Interrupt Status flag in SREG is disabled, and re-enabled upon exiting the block from any exit path.

Two possible macro parameters are permitted, ATOMIC_RESTORESTATE and ATOMIC_FORCEON.

20.33.2.2 ATOMIC_FORCEON

```
#define ATOMIC_FORCEON
```

This is a possible parameter for ATOMIC_BLOCK. When used, it will cause the ATOMIC_BLOCK to force the state of the SREG register on exit, enabling the Global Interrupt Status flag bit. This saves a small amount of flash space, a register, and one or more processor cycles, since the previous value of the SREG register does not need to be saved at the start of the block.

Care should be taken that ATOMIC_FORCEON is only used when it is known that interrupts are enabled before the block's execution or when the side effects of enabling global interrupts at the block's completion are known and understood.

20.33.2.3 ATOMIC_RESTORESTATE

```
#define ATOMIC_RESTORESTATE
```

This is a possible parameter for ATOMIC_BLOCK. When used, it will cause the ATOMIC_BLOCK to restore the previous state of the SREG register, saved before the Global Interrupt Status flag bit was disabled. The net effect of this is to make the ATOMIC_BLOCK's contents guaranteed atomic, without changing the state of the Global Interrupt Status flag when execution of the block completes.

20.33.2.4 NONATOMIC_BLOCK

```
#define NONATOMIC_BLOCK(
              type )
```

Creates a block of code that is executed non-atomically. Upon entering the block the Global Interrupt Status flag in SREG is enabled, and disabled upon exiting the block from any exit path. This is useful when nested inside ATOMIC_BLOCK sections, allowing for non-atomic execution of small blocks of code while maintaining the atomic access of the other sections of the parent ATOMIC_BLOCK.

Two possible macro parameters are permitted, NONATOMIC_RESTORESTATE and NONATOMIC_FORCEOFF.

20.33.2.5 NONATOMIC_FORCEOFF

```
#define NONATOMIC_FORCEOFF
```

This is a possible parameter for NONATOMIC_BLOCK. When used, it will cause the NONATOMIC_BLOCK to force the state of the SREG register on exit, disabling the Global Interrupt Status flag bit. This saves a small amount of flash space, a register, and one or more processor cycles, since the previous value of the SREG register does not need to be saved at the start of the block.

Care should be taken that NONATOMIC_FORCEOFF is only used when it is known that interrupts are disabled before the block's execution or when the side effects of disabling global interrupts at the block's completion are known and understood.

20.33.2.6 NONATOMIC_RESTORESTATE

```
#define NONATOMIC_RESTORESTATE
```

This is a possible parameter for NONATOMIC_BLOCK. When used, it will cause the NONATOMIC_BLOCK to restore the previous state of the SREG register, saved before the Global Interrupt Status flag bit was enabled. The net effect of this is to make the NONATOMIC_BLOCK's contents guaranteed non-atomic, without changing the state of the Global Interrupt Status flag when execution of the block completes.

20.34 <util/crc16.h>: CRC Computations

Functions

- static uint16_t _crc16_update (uint16_t __crc, uint8_t __data)
- static uint16_t _crc_xmodem_update (uint16_t __crc, uint8_t __data)
- static uint16_t _crc_ccitt_update (uint16_t __crc, uint8_t __data)
- static uint8_t _crc_ibutton_update (uint8_t __crc, uint8_t __data)
- static uint8_t _crc8_ccitt_update (uint8_t __crc, uint8_t __data)

20.34.1 Detailed Description

```
#include <util/crc16.h>
```

This header file provides optimized inline functions for calculating cyclic redundancy checks (CRC) using common polynomials.

A typical application would look like:

```
// Dallas iButton test vector.
uint8_t serno[] = { 0x02, 0x1c, 0xb8, 0x01, 0, 0, 0, 0xa2 };

int
checkcrc (void)
{
    uint8_t crc = 0, i;

    for (i = 0; i < sizeof serno / sizeof serno[0]; i++)
        crc = _crc_ibutton_update (crc, serno[i]);

    return crc; // must be 0
}
```

References:

See the Dallas Semiconductor app note 27 for 8051 assembler example and general CRC optimization suggestions. The table on the last page of the app note is the key to understanding these implementations.

Jack Crenshaw's "Implementing CRCs" article in the January 1992 issue of *Embedded Systems Programming*. This may be difficult to find, but it explains CRC's in very clear and concise terms. Well worth the effort to obtain a copy.

The hexadecimal values shown beneath the polynomials may be in little-endian or big-endian notation. The leading term is implicit. For details, see the respective implementation and the references.

20.34.2 Function Documentation

20.34.2.1 _crc16_update()

```
static uint16_t _crc16_update (
             uint16_t __crc,
             uint8_t __data )  [inline], [static]
```

Optimized CRC-16 calculation.

Polynomial: $x^{16} + x^{15} + x^2 + 1$ (0xa001, big-endian)
Initial value: `0xffff`

This CRC is normally used in disk-drive controllers.

The following is the equivalent functionality written in C.

```
static inline uint16_t
_crc16_update (uint16_t crc, uint8_t a)
{
    crc ^= a;
    for (int i = 0; i < 8; ++i)
    {
        if (crc & 1)
            crc = (crc » 1) ^ 0xA001;
        else
            crc = crc » 1;
    }

    return crc;
}
```

20.34.2.2 _crc8_ccitt_update()

```
static uint8_t _crc8_ccitt_update (
             uint8_t __crc,
             uint8_t __data )  [inline], [static]
```

Optimized CRC-8-CCITT calculation.

Polynomial: $x^8 + x^2 + x + 1$ (0xE0, big-endian)

For use with simple CRC-8
Initial value: 0x0

For use with CRC-8-ROHC
Initial value: 0xff
Reference: http://tools.ietf.org/html/rfc3095#section-5.9.1

For use with CRC-8-ATM/ITU
Initial value: 0xff
Final XOR value: 0x55
Reference: http://www.itu.int/rec/T-REC-I.432.1-199902-I/en

The C equivalent has been originally written by Dave Hylands. Assembly code is based on `_crc_ibutton_update` optimization.

The following is the equivalent functionality written in C.

```
static inline uint8_t
_crc8_ccitt_update (uint8_t inCrc, uint8_t inData)
{
    uint8_t data = inCrc ^ inData;

    for (int i = 0; i < 8; i++)
    {
        if ((data & 0x80) != 0)
        {
            data «= 1;
            data ^= 0x07;
        }
        else
        {
            data «= 1;
        }
    }
    return data;
}
```

20.34.2.3 _crc_ccitt_update()

```
static uint16_t _crc_ccitt_update (
             uint16_t __crc,
             uint8_t __data )  [inline], [static]
```

Optimized CRC-CCITT calculation.

Polynomial: $x^{16} + x^{12} + x^5 + 1$ (0x8408, big-endian)
Initial value: `0xffff`

This is the CRC used by PPP and IrDA.

See RFC1171 (PPP protocol) and IrDA IrLAP 1.1

Note

Although the CCITT polynomial is the same as that used by the Xmodem protocol, they are quite different. The difference is in how the bits are shifted through the algorithm. Xmodem shifts the MSB of the CRC and the input first, while CCITT shifts the LSB of the CRC and the input first.

The following is the equivalent functionality written in C.

```
static inline uint16_t
_crc_ccitt_update (uint16_t crc, uint8_t data)
{
    data ^= lo8 (crc);
    data ^= data « 4;

    return ((((uint16_t)data « 8) | hi8 (crc)) ^ (uint8_t)(data » 4)
            ^ ((uint16_t)data « 3));
}
```

20.34.2.4 _crc_ibutton_update()

```
static uint8_t _crc_ibutton_update (
             uint8_t __crc,
             uint8_t __data )  [inline], [static]
```

Optimized Dallas (now Maxim) iButton 8-bit CRC calculation.

Polynomial: $x^8 + x^5 + x^4 + 1$ (0x8C, big-endian)
Initial value: 0x0

See http://www.maxim-ic.com/appnotes.cfm/appnote_number/27

The following is the equivalent functionality written in C.

```
static inline uint8_t
_crc_ibutton_update (uint8_t crc, uint8_t data)
{
    crc = crc ^ data;
    for (uint8_t i = 0; i < 8; i++)
    {
        if (crc & 0x01)
            crc = (crc » 1) ^ 0x8C;
        else
            crc »= 1;
    }

    return crc;
}
```

20.34.2.5 _crc_xmodem_update()

```
static uint16_t _crc_xmodem_update (
             uint16_t __crc,
             uint8_t __data )  [inline], [static]
```

Optimized CRC-XMODEM calculation.

Polynomial: $x^{16} + x^{12} + x^5 + 1$ (0x1021, little-endian)
Initial value: 0x0

This is the CRC used by the Xmodem-CRC protocol.

The following is the equivalent functionality written in C.

```
static inline uint16_t
_crc_xmodem_update (uint16_t crc, uint8_t data)
{
    crc = crc ^ ((uint16_t)data « 8);
    for (int i = 0; i < 8; i++)
    {
        if (crc & 0x8000)
            crc = (crc « 1) ^ 0x1021;
        else
            crc «= 1;
    }

    return crc;
}
```

20.35 <util/delay_basic.h>: Basic busy-wait delay loops

Functions

- void _delay_loop_1 (uint8_t __count)
- void _delay_loop_2 (uint16_t __count)

20.35.1 Detailed Description

```
#include <util/delay_basic.h>
```

The functions in this header file implement simple delay loops that perform a busy-waiting. They are typically used to facilitate short delays in the program execution. They are implemented as count-down loops with a well-known CPU cycle count per loop iteration. As such, no other processing can occur simultaneously. It should be kept in mind that the functions described here do not disable interrupts.

In general, for long delays, the use of hardware timers is much preferable, as they free the CPU, and allow for concurrent processing of other events while the timer is running. However, in particular for very short delays, the overhead of setting up a hardware timer is too much compared to the overall delay time.

Two inline functions are provided for the actual delay algorithms.

20.35.2 Function Documentation

20.35.2.1 _delay_loop_1()

```
void _delay_loop_1 (
             uint8_t __count )
```

Delay loop using an 8-bit counter `__count`, so up to 256 iterations are possible. (The value 256 would have to be passed as 0.) The loop executes three CPU cycles per iteration, not including the overhead the compiler needs to setup the counter register.

Thus, delays of up to 768 / f_{CPU} microseconds can be achieved, where f_{CPU} denotes the CPU speed in units of 1 MHz.

As an alternative, consider `__builtin_avr_delay_cycles` which allows to specify the cycle count and can implement delays of up to 71.5 / f_{CPU} seconds.

20.35.2.2 _delay_loop_2()

```
void _delay_loop_2 (
             uint16_t __count )
```

Delay loop using a 16-bit counter `__count`, so up to 65536 iterations are possible. (The value 65536 would have to be passed as 0.) The loop executes four CPU cycles per iteration, not including the overhead the compiler requires to setup the counter register pair.

Thus, delays of up to 262.1 / f_{CPU} milliseconds can be achieved, where f_{CPU} denotes the CPU speed in units of 1 MHz.

As an alternative, consider `__builtin_avr_delay_cycles` which allows to specify the cycle count and can implement delays of up to 71.5 / f_{CPU} seconds.

20.36 <util/eu_dst.h>: Daylight Saving function for the European Union.

Functions

- int eu_dst (const time_t *timer, int32_t *z)

20.36.1 Detailed Description

```
#include <util/eu_dst.h>
```

Daylight Saving Time for the European Union

20.36.2 Function Documentation

20.36.2.1 eu_dst()

```
int eu_dst (
              const time_t * timer,
              int32_t * z )
```

To utilize this function, call

```
set_dst(eu_dst);
```

Given the time stamp and time zone parameters provided, the Daylight Saving function must return a value appropriate for the tm structures' tm_isdst element. That is:

- 0 : If Daylight Saving is not in effect.
- -1 : If it cannot be determined if Daylight Saving is in effect.
- A positive integer: Represents the number of seconds a clock is advanced for Daylight Saving. This will typically be ONE_HOUR.

Daylight Saving 'rules' are subject to frequent change. For production applications it is recommended to write your own DST function, which uses 'rules' obtained from, and modifiable by, the end user (perhaps stored in EEPROM).

20.37 <util/parity.h>: Parity bit generation

Functions

- static uint8_t parity_even_bit (uint8_t __val)

20.37.1 Detailed Description

```
#include <util/parity.h>
```

This header file contains optimized assembler code to calculate the parity bit for a byte.

20.37.2 Function Documentation

20.37.2.1 parity_even_bit()

```
uint8_t parity_even_bit (
            uint8_t val )  [inline], [static]
```

Returns

1 if `val` has an odd number of bits set, and 0 otherwise.

20.38 <util/setbaud.h>: Helper macros for baud rate calculations

Macros

- #define BAUD_TOL 2
- #define UBRR_VALUE
- #define UBRRL_VALUE
- #define UBRRH_VALUE
- #define USE_2X 0

20.38.1 Detailed Description

```
#define F_CPU 11059200
#define BAUD 38400
#include <util/setbaud.h>
```

This header file requires that on entry values are already defined for F_CPU and BAUD. In addition, the macro BAUD_TOL will define the baud rate tolerance (in percent) that is acceptable during the calculations. The value of BAUD_TOL will default to 2 %.

This header file defines macros suitable to setup the UART baud rate prescaler registers of an AVR. All calculations are done using the C preprocessor. Including this header file causes no other side effects so it is possible to include this file more than once (supposedly, with different values for the BAUD parameter), possibly even within the same function.

Assuming that the requested BAUD is valid for the given F_CPU then the macro UBRR_VALUE is set to the required prescaler value. Two additional macros are provided for the low and high bytes of the prescaler, respectively↩ : UBRRL_VALUE is set to the lower byte of the UBRR_VALUE and UBRRH_VALUE is set to the upper byte. An additional macro USE_2X will be defined. Its value is set to 1 if the desired BAUD rate within the given tolerance could only be achieved by setting the U2X bit in the UART configuration. It will be defined to 0 if U2X is not needed.

Example usage:

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

#define F_CPU 4000000

static void
uart_9600(void)
{
#define BAUD 9600
#include <util/setbaud.h>
    UBRRH = UBRRH_VALUE;
    UBRRL = UBRRL_VALUE;
```

```
#if USE_2X
    UCSRA |= (1 « U2X);
#else
    UCSRA &= ~(1 « U2X);
#endif
}

static void
uart_38400(void)
{
#undef BAUD  // avoid compiler warning
#define BAUD 38400
#include <util/setbaud.h>
    UBRRH = UBRRH_VALUE;
    UBRRL = UBRRL_VALUE;
#if USE_2X
    UCSRA |= (1 « U2X);
#else
    UCSRA &= ~(1 « U2X);
#endif
}
```

In this example, two functions are defined to setup the UART to run at 9600 Bd, and 38400 Bd, respectively. Using a CPU clock of 4 MHz, 9600 Bd can be achieved with an acceptable tolerance without setting U2X (prescaler 25), while 38400 Bd require U2X to be set (prescaler 12).

20.38.2 Macro Definition Documentation

20.38.2.1 BAUD_TOL

```
#define BAUD_TOL 2
```

Input and output macro for <util/setbaud.h>

Define the acceptable baud rate tolerance in percent. If not set on entry, it will be set to its default value of 2.

20.38.2.2 UBRR_VALUE

```
#define UBRR_VALUE
```

Output macro from <util/setbaud.h>

Contains the calculated baud rate prescaler value for the UBRR register.

20.38.2.3 UBRRH_VALUE

```
#define UBRRH_VALUE
```

Output macro from <util/setbaud.h>

Contains the upper byte of the calculated prescaler value (UBRR_VALUE).

20.38.2.4 UBRRL_VALUE

```
#define UBRRL_VALUE
```

Output macro from <util/setbaud.h>

Contains the lower byte of the calculated prescaler value (UBRR_VALUE).

20.38.2.5 USE_2X

```
#define USE_2X 0
```

Output macro from <util/setbaud.h>

Contains the value 1 if the desired baud rate tolerance could only be achieved by setting the U2X bit in the UART configuration. Contains 0 otherwise.

20.39 <util/twi.h>: TWI bit mask definitions

TWSR values

Mnemonics:
TW_MT_xxx - master transmitter
TW_MR_xxx - master receiver
TW_ST_xxx - slave transmitter
TW_SR_xxx - slave receiver

- #define TW_START 0x08
- #define TW_REP_START 0x10
- #define TW_MT_SLA_ACK 0x18
- #define TW_MT_SLA_NACK 0x20
- #define TW_MT_DATA_ACK 0x28
- #define TW_MT_DATA_NACK 0x30
- #define TW_MT_ARB_LOST 0x38
- #define TW_MR_ARB_LOST 0x38
- #define TW_MR_SLA_ACK 0x40
- #define TW_MR_SLA_NACK 0x48
- #define TW_MR_DATA_ACK 0x50
- #define TW_MR_DATA_NACK 0x58
- #define TW_ST_SLA_ACK 0xA8
- #define TW_ST_ARB_LOST_SLA_ACK 0xB0
- #define TW_ST_DATA_ACK 0xB8
- #define TW_ST_DATA_NACK 0xC0
- #define TW_ST_LAST_DATA 0xC8
- #define TW_SR_SLA_ACK 0x60
- #define TW_SR_ARB_LOST_SLA_ACK 0x68
- #define TW_SR_GCALL_ACK 0x70
- #define TW_SR_ARB_LOST_GCALL_ACK 0x78
- #define TW_SR_DATA_ACK 0x80
- #define TW_SR_DATA_NACK 0x88
- #define TW_SR_GCALL_DATA_ACK 0x90
- #define TW_SR_GCALL_DATA_NACK 0x98
- #define TW_SR_STOP 0xA0
- #define TW_NO_INFO 0xF8
- #define TW_BUS_ERROR 0x00
- #define TW_STATUS_MASK
- #define TW_STATUS (TWSR & TW_STATUS_MASK)

R/∼W bit in SLA+R/W address field.

- #define TW_READ 1
- #define TW_WRITE 0

20.39.1 Detailed Description

```
#include <util/twi.h>
```

This header file contains bit mask definitions for use with the AVR TWI interface.

20.39.2 Macro Definition Documentation

20.39.2.1 TW_BUS_ERROR

```
#define TW_BUS_ERROR 0x00
```

illegal start or stop condition

20.39.2.2 TW_MR_ARB_LOST

```
#define TW_MR_ARB_LOST 0x38
```

arbitration lost in SLA+R or NACK

20.39.2.3 TW_MR_DATA_ACK

```
#define TW_MR_DATA_ACK 0x50
```

data received, ACK returned

20.39.2.4 TW_MR_DATA_NACK

```
#define TW_MR_DATA_NACK 0x58
```

data received, NACK returned

20.39.2.5 TW_MR_SLA_ACK

```
#define TW_MR_SLA_ACK 0x40
```

SLA+R transmitted, ACK received

20.39.2.6 TW_MR_SLA_NACK

```
#define TW_MR_SLA_NACK 0x48
```

SLA+R transmitted, NACK received

20.39.2.7 TW_MT_ARB_LOST

```
#define TW_MT_ARB_LOST 0x38
```

arbitration lost in SLA+W or data

20.39.2.8 TW_MT_DATA_ACK

```
#define TW_MT_DATA_ACK 0x28
```

data transmitted, ACK received

20.39.2.9 TW_MT_DATA_NACK

```
#define TW_MT_DATA_NACK 0x30
```

data transmitted, NACK received

20.39.2.10 TW_MT_SLA_ACK

```
#define TW_MT_SLA_ACK 0x18
```

SLA+W transmitted, ACK received

20.39.2.11 TW_MT_SLA_NACK

```
#define TW_MT_SLA_NACK 0x20
```

SLA+W transmitted, NACK received

20.39.2.12 TW_NO_INFO

```
#define TW_NO_INFO 0xF8
```

no state information available

20.39.2.13 TW_READ

```
#define TW_READ 1
```

SLA+R address

20.39.2.14 TW_REP_START

```
#define TW_REP_START 0x10
```

repeated start condition transmitted

20.39.2.15 TW_SR_ARB_LOST_GCALL_ACK

```
#define TW_SR_ARB_LOST_GCALL_ACK 0x78
```

arbitration lost in SLA+RW, general call received, ACK returned

20.39.2.16 TW_SR_ARB_LOST_SLA_ACK

```
#define TW_SR_ARB_LOST_SLA_ACK 0x68
```

arbitration lost in SLA+RW, SLA+W received, ACK returned

20.39.2.17 TW_SR_DATA_ACK

```
#define TW_SR_DATA_ACK 0x80
```

data received, ACK returned

20.39.2.18 TW_SR_DATA_NACK

```
#define TW_SR_DATA_NACK 0x88
```

data received, NACK returned

20.39.2.19 TW_SR_GCALL_ACK

```
#define TW_SR_GCALL_ACK 0x70
```

general call received, ACK returned

20.39.2.20 TW_SR_GCALL_DATA_ACK

```
#define TW_SR_GCALL_DATA_ACK 0x90
```

general call data received, ACK returned

20.39.2.21 TW_SR_GCALL_DATA_NACK

```
#define TW_SR_GCALL_DATA_NACK 0x98
```

general call data received, NACK returned

20.39.2.22 TW_SR_SLA_ACK

```
#define TW_SR_SLA_ACK 0x60
```

SLA+W received, ACK returned

20.39.2.23 TW_SR_STOP

```
#define TW_SR_STOP 0xA0
```

stop or repeated start condition received while selected

20.39.2.24 TW_ST_ARB_LOST_SLA_ACK

```
#define TW_ST_ARB_LOST_SLA_ACK 0xB0
```

arbitration lost in SLA+RW, SLA+R received, ACK returned

20.39.2.25 TW_ST_DATA_ACK

```
#define TW_ST_DATA_ACK 0xB8
```

data transmitted, ACK received

20.39.2.26 TW_ST_DATA_NACK

```
#define TW_ST_DATA_NACK 0xC0
```

data transmitted, NACK received

20.39.2.27 TW_ST_LAST_DATA

```
#define TW_ST_LAST_DATA 0xC8
```

last data byte transmitted, ACK received

20.39.2.28 TW_ST_SLA_ACK

```
#define TW_ST_SLA_ACK 0xA8
```

SLA+R received, ACK returned

20.39.2.29 TW_START

```
#define TW_START 0x08
```

start condition transmitted

20.39.2.30 TW_STATUS

```
#define TW_STATUS (TWSR & TW_STATUS_MASK)
```

TWSR, masked by TW_STATUS_MASK

20.39.2.31 TW_STATUS_MASK

```
#define TW_STATUS_MASK
```

Value:

```
(_BV(TWS7)|_BV(TWS6)|_BV(TWS5)|_BV(TWS4)|\
_BV(TWS3))
```

The lower 3 bits of TWSR are reserved on the ATmega163. The 2 LSB carry the prescaler bits on the newer ATmegas.

20.39.2.32 TW_WRITE

```
#define TW_WRITE 0
```

SLA+W address

20.40 <util/usa_dst.h>: Daylight Saving function for the USA.

Functions

- int usa_dst (const time_t *timer, int32_t *z)

20.40.1 Detailed Description

```
#include <util/usa_dst.h>
```

Daylight Saving function for the USA.

20.40.2 Function Documentation

20.40.2.1 usa_dst()

```
int usa_dst (
            const time_t * timer,
            int32_t * z )
```

To utilize this function, call

```
set_dst(usa_dst);
```

Given the time stamp and time zone parameters provided, the Daylight Saving function must return a value appropriate for the tm structures' tm_isdst element. That is:

- 0 : If Daylight Saving is not in effect.
- −1 : If it cannot be determined if Daylight Saving is in effect.
- A positive integer : Represents the number of seconds a clock is advanced for Daylight Saving. This will typically be ONE_HOUR.

Daylight Saving 'rules' are subject to frequent change. For production applications it is recommended to write your own DST function, which uses 'rules' obtained from, and modifiable by, the end user (perhaps stored in EEPROM).

20.41 <compat/deprecated.h>: Deprecated items

Allowing specific system-wide interrupts

In addition to globally enabling interrupts, each device's particular interrupt needs to be enabled separately if interrupts for this device are desired. While some devices maintain their interrupt enable bit inside the device's register set, external and timer interrupts have system-wide configuration registers.

Example:

```
// Enable timer 1 overflow interrupts.
timer_enable_int(_BV(TOIE1));

// Do some work...

// Disable all timer interrupts.
timer_enable_int(0);
```

Note

Be careful when you use these functions. If you already have a different interrupt enabled, you could inadvertently disable it by enabling another interrupt.

- static void timer_enable_int (unsigned char ints)
- #define enable_external_int(mask) (__EICR = mask)
- #define INTERRUPT(signame)

Obsolete IO macros

Back in a time when AVR-GCC and AVR-LibC could not handle IO port access in the direct assignment form as they are handled now, all IO port access had to be done through specific macros that eventually resulted in inline assembly instructions performing the desired action.

These macros became obsolete, as reading and writing IO ports can be done by simply using the IO port name in an expression, and all bit manipulation (including those on IO ports) can be done using generic C bit manipulation operators.

The macros in this group simulate the historical behaviour. While they are supposed to be applied to IO ports, the emulation actually uses standard C methods, so they could be applied to arbitrary memory locations as well.

- #define inp(port) (port)
- #define outp(val, port) (port) = (val)
- #define inb(port) (port)
- #define outb(port, val) (port) = (val)
- #define sbi(port, bit) (port) |= (1 << (bit))
- #define cbi(port, bit) (port) &= ~(1 << (bit))

20.41.1 Detailed Description

This header file contains several items that used to be available in previous versions of this library, but have eventually been deprecated over time.

```
#include <compat/deprecated.h>
```

These items are supplied within that header file for backward compatibility reasons only, so old source code that has been written for previous library versions could easily be maintained until its end-of-life. Use of any of these items in new code is strongly discouraged.

Some device headers provide deprecated vector names starting with `SIG_`, followed by a relatively verbose but arbitrarily chosen name describing the interrupt vector. This has been the only available style in AVR-LibC up to version 1.2.x. This historical naming style is not recommended for new projects, and some headers require that the macro `__AVR_LIBC_DEPRECATED_ENABLE__` is defined so that the `SIG_` names ISR names are available. For available ISR vector names and ISR_N vector numbers, see What ISR names are available for my device? in the FAQ.

20.41.2 Macro Definition Documentation

20.41.2.1 cbi

```
#define cbi(
              port,
              bit ) (port) &= ~(1 << (bit))
```

Deprecated

Clear `bit` in IO port `port`.

20.41.2.2 enable_external_int

```
#define enable_external_int(
              mask ) (__EICR = mask)
```

Deprecated

This macro gives access to the `GIMSK` register (or `EIMSK` register if using an AVR Mega device or `GICR` register for others). Although this macro is essentially the same as assigning to the register, it does adapt slightly to the type of device being used. This macro is unavailable if none of the registers listed above are defined.

20.41.2.3 inb

```
#define inb(
              port ) (port)
```

Deprecated

Read a value from an IO port `port`.

20.41.2.4 inp

```
#define inp(
              port ) (port)
```

Deprecated

Read a value from an IO port `port`.

20.41.2.5 INTERRUPT

```
#define INTERRUPT(
              signame )
```

Value:

```
void signame (void) __attribute__ ((__interrupt__,__INTR_ATTRS));    \
void signame (void)
```

Deprecated

Introduces an interrupt handler function that runs with global interrupts initially enabled. This allows interrupt handlers to be interrupted.

As this macro has been used by too many unsuspecting people in the past, it has been deprecated, and will be removed in a future version of the library. Users who want to legitimately re-enable interrupts in their interrupt handlers as quickly as possible are encouraged to explicitly declare their handlers as described above.

20.41.2.6 outb

```
#define outb(
              port,
              val ) (port) = (val)
```

Deprecated

Write `val` to IO port `port`.

20.41.2.7 outp

```
#define outp(
              val,
              port ) (port) = (val)
```

Deprecated

Write `val` to IO port `port`.

20.41.2.8 sbi

```
#define sbi(
              port,
              bit ) (port) |= (1 << (bit))
```

Deprecated

Set `bit` in IO port `port`.

20.41.3 Function Documentation

20.41.3.1 timer_enable_int()

```
static void timer_enable_int (
             unsigned char ints )  [inline], [static]
```

Deprecated

This function modifies the `timsk` register. The value you pass via `ints` is device specific.

20.42 <compat/ina90.h>: Compatibility with IAR EWB 3.x

```
#include <compat/ina90.h>
```

This is an attempt to provide some compatibility with header files that come with IAR C, to make porting applications between different compilers easier. No 100% compatibility though.

Note

For actual documentation, please see the IAR manual.

20.43 Demo projects

Modules

- Combining C and assembly source files
- A simple project
- A more sophisticated project
- Using the standard IO facilities
- Example using the two-wire interface (TWI)

20.43.1 Detailed Description

Various small demo projects are provided to illustrate several aspects of using the opensource utilities for the AVR controller series. It should be kept in mind that these demos serve mainly educational purposes, and are normally not directly suitable for use in any production environment. Usually, they have been kept as simple as sufficient to demonstrate one particular feature.

The simple project is somewhat like the "Hello world!" application for a microcontroller, about the most simple project that can be done. It is explained in good detail, to allow the reader to understand the basic concepts behind using the tools on an AVR microcontroller.

The more sophisticated demo project builds on top of that simple project, and adds some controls to it. It touches a number of AVR-LibC's basic concepts on its way.

A comprehensive example on using the standard IO facilities intends to explain that complex topic, using a practical microcontroller peripheral setup with a serial connection (usually attached to an USB bridge), and an HD44780-compatible industry-standard LCD display.

The Example using the two-wire interface (TWI) project explains the use of the two-wire hardware interface (also known as "I2C") that is present on many AVR controllers.

Finally, the Combining C and assembly source files demo shows how C and assembly language source files can collaborate within one project. While the overall project is managed by a C program part for easy maintenance, time-critical parts are written directly in manually optimized assembly language for shortest execution times possible. Naturally, this kind of project is very closely tied to the hardware design, thus it is custom-tailored to a particular controller type and peripheral setup. As an alternative to the assembly-language solution, this project also offers a C-only implementation (deploying the exact same peripheral setup) based on a more sophisticated (and thus more expensive) but pin-compatible controller.

Most of these projects have been primarily targeted to a board compatible with the popular Arduino Nano device, using an ATmega328P. The simple project can run on that board without requiring any additional hardware, other demos only make sense when attaching some additional components, like an LCD, external buttons, or a potentiometer.

20.44 Combining C and assembly source files

For time- or space-critical applications, it can often be desirable to combine C code (for easy maintenance) and assembly code (for maximal speed or minimal code size) together. This demo provides an example of how to do that.

The objective of the demo is to decode radio-controlled model PWM signals, and control an output PWM based on the current input signal's value. The incoming PWM pulses follow a standard encoding scheme where a pulse width of 920 microseconds denotes one end of the scale (represented as 0 % pulse width on output), and 2120 microseconds mark the other end (100 % output PWM). Normally, multiple channels would be encoded that way in subsequent pulses, followed by a larger gap, so the entire frame will repeat each 14 through 20 ms, but this is ignored for the purpose of the demo, so only a single input PWM channel is assumed.

The basic challenge is to use the cheapest controller available for the task, an ATtiny13 that has only a single timer channel. As this timer channel is required to run the outgoing PWM signal generation, the incoming PWM decoding had to be adjusted to the constraints set by the outgoing PWM.

As PWM generation toggles the counting direction of timer 0 between up and down after each 256 timer cycles, the current time cannot be deduced by reading TCNT0 only, but the current counting direction of the timer needs to be considered as well. This requires servicing interrupts whenever the timer hits *TOP* (255) and *BOTTOM* (0) to learn about each change of the counting direction. For PWM generation, it is usually desired to run it at the highest possible speed so filtering the PWM frequency from the modulated output signal is made easy. Thus, the PWM timer runs at full CPU speed. This causes the overflow and compare match interrupts to be triggered each 256 CPU clocks, so they must run with the minimal number of processor cycles possible in order to not impose a too high CPU load by these interrupt service routines. This is the main reason to implement the entire interrupt handling in fine-tuned assembly code rather than in C.

In order to verify parts of the algorithm, and the underlying hardware, the demo has been set up in a way so the pin-compatible but more expensive ATtiny45 (or its siblings ATtiny25 and ATtiny85) could be used as well. In that case, no separate assembly code is required, as two timer channels are available.

20.44.1 Hardware setup

The incoming PWM pulse train is fed into PB4. It will generate a pin change interrupt there on each edge of the incoming signal.

The outgoing PWM is generated through OC0B of timer channel 0 (PB1). For demonstration purposes, a LED should be connected to that pin (like, one of the LEDs of an STK500).

The controllers run on their internal calibrated RC oscillators, 1.2 MHz on the ATtiny13, and 1.0 MHz on the ATtiny45.

20.44.2 A code walkthrough

20.44.2.1 asmdemo.c

After the usual include files, two variables are defined. The first one, `pwm_incoming` is used to communicate the most recent pulse width detected by the incoming PWM decoder up to the main loop.

The second variable actually only constitutes of a single bit, `intbits.pwm_received`. This bit will be set whenever the incoming PWM decoder has updated `pwm_incoming`.

Both variables are marked *volatile* to ensure their readers will always pick up an updated value, as both variables will be set by interrupt service routines.

The function `ioinit()` initializes the microcontroller peripheral devices. In particular, it starts timer 0 to generate the outgoing PWM signal on OC0B. Setting OCR0A to 255 (which is the *TOP* value of timer 0) is used to generate a timer 0 overflow A interrupt on the ATtiny13. This interrupt is used to inform the incoming PWM decoder that the counting direction of channel 0 is just changing from up to down. Likewise, an overflow interrupt will be generated whenever the countdown reached *BOTTOM* (value 0), where the counter will again alter its counting direction to upwards. This information is needed in order to know whether the current counter value of `TCNT0` is to be evaluated from bottom or top.

Further, `ioinit()` activates the pin-change interrupt `PCINT0` on any edge of PB4. Finally, PB1 (OC0B) will be activated as an output pin, and global interrupts are being enabled.

In the ATtiny45 setup, the C code contains an ISR for `PCINT0`. At each pin-change interrupt, it will first be analyzed whether the interrupt was caused by a rising or a falling edge. In case of the rising edge, timer 1 will be started with a prescaler of 16 after clearing the current timer value. Then, at the falling edge, the current timer value will be recorded (and timer 1 stopped), the pin-change interrupt will be suspended, and the upper layer will be notified that the incoming PWM measurement data is available.

Function `main()` first initializes the hardware by calling `ioinit()`, and then waits until some incoming PWM value is available. If it is, the output PWM will be adjusted by computing the relative value of the incoming PWM. Finally, the pin-change interrupt is re-enabled, and the CPU is put to sleep.

20.44.2.2 project.h

In order for the interrupt service routines to be as fast as possible, some of the CPU registers are set aside completely for use by these routines, so the compiler would not use them for C code. This is arranged for in `project.h`.

The file is divided into one section that will be used by the assembly source code, and another one to be used by C code. The assembly part is distinguished by the preprocessing macro `__ASSEMBLER__` (which will be automatically set by the compiler front-end when preprocessing an assembly-language file), and it contains just macros that give symbolic names to a number of CPU registers. The preprocessor will then replace the symbolic names by their right-hand side definitions before calling the assembler.

In C code, the compiler needs to see variable declarations for these objects. This is done by using declarations that bind a variable permanently to a CPU register (see How to permanently bind a variable to a register?). Even in case the C code never has a need to access these variables, declaring the register binding that way causes the compiler to not use these registers in C code at all.

The `flags` variable needs to be in the range of r16 through r31 as it is the target of a *load immediate* (or `SER`) instruction that is not applicable to the entire register file.

20.44.2.3 isrs.S

This file is a preprocessed assembly source file. The C preprocessor will be run by the compiler front-end first, resolving all `#include`, `#define` etc. directives. The resulting program text will then be passed on to the assembler.

As the C preprocessor strips all C-style comments, preprocessed assembly source files can have both, C-style (`/* ... */`, `// ...`) as well as assembly-style (`; ...`) comments.

At the top, the IO register definition file `avr/io.h` and the project declaration file `project.h` are included. The remainder of the file is conditionally assembled only if the target MCU type is an ATtiny13, so it will be completely ignored for the ATtiny45 option.

Next are the two interrupt service routines for timer 0 compare A match (timer 0 hits *TOP*, as OCR0A is set to 255) and timer 0 overflow (timer 0 hits *BOTTOM*). As discussed above, these are kept as short as possible. They only save `SREG` (as the flags will be modified by the `INC` instruction), increment the `counter_hi` variable which forms the high part of the current time counter (the low part is formed by querying `TCNT0` directly), and clear or set the variable `flags`, respectively, in order to note the current counting direction. The `RETI` instruction terminates these interrupt service routines. Total cycle count is 8 CPU cycles, so together with the 4 CPU cycles needed for interrupt setup, and the 2 cycles for the RJMP from the interrupt vector to the handler, these routines will require 14 out of each 256 CPU cycles, or about 5 % of the overall CPU time.

The pin-change interrupt `PCINT0` will be handled in the final part of this file. The basic algorithm is to quickly evaluate the current system time by fetching the current timer value of `TCNT0`, and combining it with the overflow part in `counter_hi`. If the counter is currently counting down rather than up, the value fetched from `TCNT0` must be negated. Finally, if this pin-change interrupt was triggered by a rising edge, the time computed will be recorded as the start time only. Then, at the falling edge, this start time will be subtracted from the current time to compute the actual pulse width seen (left in `pwm_incoming`), and the upper layers are informed of the new value by setting bit 0 in the `intbits` flags. At the same time, this pin-change interrupt will be disabled so no new measurement can be performed until the upper layer had a chance to process the current value.

20.44.3 The source code

The source code is installed under

`$prefix/share/doc/avr-libc/examples/asmdemo/`,

where `$prefix` is a configuration option. For Unix systems, it is usually set to either `/usr` or `/usr/local`.

20.45 A simple project

At this point, you should have the GNU tools configured, built, and installed on your system. In this chapter, we present a simple example of using the GNU tools in an AVR project. After reading this chapter, you should have a better feel as to how the tools are used and how a `Makefile` can be configured.

20.45.1 The Project

This project will use pulse-width modulation (PWM) to ramp an LED on and off every two seconds. This project once used to start out with a simple homemade AVR circuit. Meanwhile, in particular the Arduino project has provided a large boost to the use of AVR microcontrollers, and Arduino compatible hardware is ubiquitous. We thus concentrate on something compatible to an Arduino Nano with an ATmega328P (or PB) MCU.

Alas, the only user LED on these Arduino devices is not attached to a GPIO pin that can be directly controlled by the timer hardware to generate the PWM signal. We thus toggle the LED in software.

The source code is given in demo.c. For the sake of this example, create a file called `demo.c` containing this source code. Some of the more important parts of the code are:

Note [1]:

It is always a good idea to avoid "magic numbers", and define macros for certain constants at the top of the project (or, in a separate header file). Note the use of the `_BV` macro: it turns a bit number into the respective bit mask.

Note [2]:

ISR() is a macro that marks the function as an interrupt routine. In this case, the function will get called when timer 1 reaches its top value. Setting up interrupts is explained in greater detail in <avr/interrupt.h>: Interrupts.
As the 16-bit timer 1 is configured to run with a reduced 10 bit resolution, the top value is configured through the "input capture" register `ICP`, thus the input capture interrupt triggers when the timer reaches its top value.

Note [3]:

Immediately at the start of the interrupt service, handle toggling the LED. As the Arduino Nano LED is attached from PB5 to GND, turn it on here. While the construct of reading the port register, ORing a bit mask into it, and writing it back might look cumbersome – the compiler will actually turn this into an `SBI` instruction by recognizing that the arguments to the operation are suitable for this optimization.

Note [4]:

The PWM is being used in 10-bit mode, so we need a 16-bit variable to remember the current value.

Note [5]:

This section determines the new value of the PWM.

Note [6]:

Here's where the newly computed value is loaded into the compare register. Since we are in an interrupt routine, it is safe to use a 16-bit assignment to the register. Outside of an interrupt, the assignment should only be performed with interrupts disabled if there's a chance that an interrupt routine could also access this register (or another register that uses `TEMP`), see the appropriate FAQ entry.

Note [7]:

This interrupt service routine gets called every time the counter reaches its compare/match value. Turn off the LED here.

Note [8]:

This routine configures all the hardware after a reset.

Note [9]:

The main loop of the program does nothing – all the work is done by the interrupt routines! The `sleep_mode()` puts the processor on sleep until the next interrupt, to conserve power. Of course, that probably won't be noticeable as we are still driving a LED, it is merely mentioned here to demonstrate the basic principle.

20.45.2 The Source Code

```
/*
 * ----------------------------------------------------------------------------
 * "THE BEER-WARE LICENSE" (Revision 42):
 * <joerg@FreeBSD.ORG> wrote this file.  As long as you retain this notice you
 * can do whatever you want with this stuff. If we meet some day, and you think
 * this stuff is worth it, you can buy me a beer in return.        Joerg Wunsch
 * ----------------------------------------------------------------------------
 *
 * Simple AVR demonstration.  Controls a LED that can be directly
 * connected from a pin to GND.  The brightness of the LED is
 * controlled with the PWM.  After each period of the PWM, the PWM
 * value is either incremented or decremented, that's all.
 *
 * Configured to run on an Arduino Nano compatible device with an
 * ATmega328P.  The board has a LED, which is connected to PB5.  Since
 * PB5 is not a hardware waveform output from a timer on the
 * ATmega328P, the LED is toggled within the timer interrupts.
 */

#include <stdint.h>
#include <avr/io.h>
#include <avr/interrupt.h>
#include <avr/sleep.h>

/* Note [1] */
#define TIMER1_TOP 1023

#define LED_PORT PORTB
#define LED_DDR  DDRB
#define LED_PIN  _BV(5)

enum { UP, DOWN };

ISR(TIMER1_CAPT_vect)           // Note [2]
{
    // turn on LED
    LED_PORT |= LED_PIN;        // Note [3]

    static uint16_t pwm;        // Note [4]
    static uint8_t direction;

    switch (direction)          // Note [5]
    {
        case UP:
            if (++pwm == TIMER1_TOP)
                direction = DOWN;
            break;

        case DOWN:
            if (--pwm == 0)
                direction = UP;
            break;
    }

    OCR1A = pwm;                // Note [6]
}

ISR(TIMER1_COMPA_vect)
{
    // turn off LED
    LED_PORT &= ~LED_PIN;       // Note [7]
}

static void
ioinit(void)                    // Note [8]
{
    /* Timer 1 in CTC mode with 10 bits, CPU/8 speed */
    ICR1 = TIMER1_TOP;
    TCCR1A = 0;
```

```
    TCCR1B = _BV(WGM13) | _BV(WGM12) | 2;

    /* Set PWM value to 0. */
    OCR1A = 1;

    /* Enable OC1 as output. */
    LED_DDR |= LED_PIN;

    /* Enable timer 1 overflow and compare A interrupt. */
    TIMSK1 = _BV(ICIE1) | _BV(OCIE1A);
    sei ();
}

int
main(void)
{
    ioinit ();

    /* Loop forever, the interrupts are doing the rest.  */

    for (;;)                    // Note [9]
        sleep_mode();

    return 0;
}
```

20.45.3 Compiling and Linking

This first thing that needs to be done is compile the source. When compiling, the compiler needs to know the processor type so the `-mmcu` option is specified. The `-Os` option will tell the compiler to optimize the code for efficient space usage (at the possible expense of code execution speed). The `-g` is used to embed debug info. The debug info is useful for disassemblies and doesn't end up in the `.hex` files, so I usually specify it. Finally, the `-c` tells the compiler to compile and stop – don't link. This demo is small enough that we could compile and link in one step. However, real-world projects will have several modules and will typically need to break up the building of the project into several compiles and one link.

```
$ avr-gcc -g -Os -mmcu=atmega328p -c demo.c
```

The compilation will create a `demo.o` file. Next we link it into a binary called `demo.elf`.

```
$ avr-gcc -g -mmcu=atmega328p -mrelax -o demo.elf demo.o
```

It is important to specify the MCU type when linking. The compiler uses the `-mmcu` option to choose start-up files and run-time libraries that get linked together. If this option isn't specified, the compiler defaults to the AT90S8515 processor environment, which is most certainly what you didn't want.

The `-mrelax` option instructs the linker to replace `CALL` instruction with faster and smaller `RCALL` instructions if possible.

20.45.4 Examining the Object File

Now we have a binary file. Can we do anything useful with it (besides put it into the processor?) The GNU Binutils suite is made up of many useful tools for manipulating object files that get generated. One tool is `avr-objdump`, which takes information from the object file and displays it in many useful ways. Typing the command by itself will cause it to list out its options.

For instance, to get a feel of the application's size, the `-h` option can be used. The output of this option shows how much space is used in each of the output sections. The `.comment`, `.note` and `.debug` sections hold additional (debug) information and won't make it into the ROM file.

An overview of the program space and consumed by the program, and of the data in static storage, can be obtained with

```
$ avr-objdump -P mem-usage demo.elf
```

Since non-zero data in static storage consumes RAM, and also needs initialization values stored in non-volatile memory, the `.data` ouput section contributes both to the text segment (called *Program* below) and also to the data segment (called *Data*). For our program, the output reads:

```
demo.elf:     file format elf32-avr
AVR Memory Usage
----------------
Device: atmega328p

Program:     320 bytes (1.0% Full)
(.text + .data + .rodata + .bootloader)

Data:          3 bytes (0.1% Full)
(.data + .bss + .noinit)
```

An even more useful option is `-S`. This option disassembles the binary file and intersperses the source code in the output! Such a listing includes routines pulled in from the libraries and the vector table contents. Also, all the "fix-ups" have been satisfied. In other words, the listing generated by this option reflects the actual code that the processor will run.

```
$ avr-objdump -h -S demo.elf > demo.lst
```

Here's the output as saved in the `demo.lst` file:

```
demo.elf:     file format elf32-avr

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .data         00000000  00800100  00000140  000001d4  2**0
                  CONTENTS, ALLOC, LOAD, DATA
  1 .text         00000140  00000000  00000000  00000094  2**1
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  2 .bss          00000003  00800100  00800100  000001d4  2**0
                  ALLOC
  3 .comment      00000012  00000000  00000000  000001d4  2**0
                  CONTENTS, READONLY
  4 .note.gnu.avr.deviceinfo 00000040  00000000  00000000  000001e8  2**2
                  CONTENTS, READONLY, OCTETS
  5 .debug_aranges 00000068  00000000  00000000  00000228  2**3
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  6 .debug_info   0000075b  00000000  00000000  00000290  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  7 .debug_abbrev 000006c1  00000000  00000000  000009eb  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  8 .debug_line   00000312  00000000  00000000  000010ac  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  9 .debug_frame  00000050  00000000  00000000  000013c0  2**2
                  CONTENTS, READONLY, DEBUGGING, OCTETS
 10 .debug_str    00000361  00000000  00000000  00001410  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
 11 .debug_line_str 000000d0  00000000  00000000  00001771  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
 12 .debug_rnglists 00000019  00000000  00000000  00001841  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS

Disassembly of section .text:

00000000 <__vectors>:
   0: 33 c0        rjmp .+102      ; 0x68 <__ctors_end>
   2: 00 00        nop
   4: 7a c0        rjmp .+244      ; 0xfa <__bad_interrupt>
   6: 00 00        nop
```

```
   8: 78 c0        rjmp .+240      ; 0xfa <__bad_interrupt>
   a: 00 00        nop
   c: 76 c0        rjmp .+236      ; 0xfa <__bad_interrupt>
   e: 00 00        nop
  10: 74 c0        rjmp .+232      ; 0xfa <__bad_interrupt>
  12: 00 00        nop
  14: 72 c0        rjmp .+228      ; 0xfa <__bad_interrupt>
  16: 00 00        nop
  18: 70 c0        rjmp .+224      ; 0xfa <__bad_interrupt>
  1a: 00 00        nop
  1c: 6e c0        rjmp .+220      ; 0xfa <__bad_interrupt>
  1e: 00 00        nop
  20: 6c c0        rjmp .+216      ; 0xfa <__bad_interrupt>
  22: 00 00        nop
  24: 6a c0        rjmp .+212      ; 0xfa <__bad_interrupt>
  26: 00 00        nop
  28: 2f c0        rjmp .+94       ; 0x88 <__vector_10>
  2a: 00 00        nop
  2c: 64 c0        rjmp .+200      ; 0xf6 <__vector_11>
  2e: 00 00        nop
  30: 64 c0        rjmp .+200      ; 0xfa <__bad_interrupt>
  32: 00 00        nop
  34: 62 c0        rjmp .+196      ; 0xfa <__bad_interrupt>
  36: 00 00        nop
  38: 60 c0        rjmp .+192      ; 0xfa <__bad_interrupt>
  3a: 00 00        nop
  3c: 5e c0        rjmp .+188      ; 0xfa <__bad_interrupt>
  3e: 00 00        nop
  40: 5c c0        rjmp .+184      ; 0xfa <__bad_interrupt>
  42: 00 00        nop
  44: 5a c0        rjmp .+180      ; 0xfa <__bad_interrupt>
  46: 00 00        nop
  48: 58 c0        rjmp .+176      ; 0xfa <__bad_interrupt>
  4a: 00 00        nop
  4c: 56 c0        rjmp .+172      ; 0xfa <__bad_interrupt>
  4e: 00 00        nop
  50: 54 c0        rjmp .+168      ; 0xfa <__bad_interrupt>
  52: 00 00        nop
  54: 52 c0        rjmp .+164      ; 0xfa <__bad_interrupt>
  56: 00 00        nop
  58: 50 c0        rjmp .+160      ; 0xfa <__bad_interrupt>
  5a: 00 00        nop
  5c: 4e c0        rjmp .+156      ; 0xfa <__bad_interrupt>
  5e: 00 00        nop
  60: 4c c0        rjmp .+152      ; 0xfa <__bad_interrupt>
  62: 00 00        nop
  64: 4a c0        rjmp .+148      ; 0xfa <__bad_interrupt>
 ...

00000068 <__ctors_end>:
  68: 11 24        eor r1, r1
  6a: 1f be        out 0x3f, r1 ; 63

0000006c <__init_sp>:
  6c: cf ef        ldi r28, 0xFF ; 255
  6e: d8 e0        ldi r29, 0x08 ; 8
  70: de bf        out 0x3e, r29 ; 62
  72: cd bf        out 0x3d, r28 ; 61

00000074 <__do_clear_bss>:
  74: 21 e0        ldi r18, 0x01 ; 1

00000076 <.Loc.1>:
  76: a0 e0        ldi r26, 0x00 ; 0

00000078 <.Loc.2>:
  78: b1 e0        ldi r27, 0x01 ; 1

0000007a <.Loc.3>:
  7a: 01 c0        rjmp .+2        ; 0x7e <.Loc.5>

0000007c <.Loc.4>:
  7c: 1d 92        st X+, r1
```

```
0000007e <.Loc.5>:
  7e: a3 30        cpi r26, 0x03 ; 3

00000080 <.Loc.6>:
  80: b2 07        cpc r27, r18

00000082 <.Loc.7>:
  82: e1 f7        brne .-8       ; 0x7c <.Loc.4>

00000084 <__call_main>:
  84: 3b d0        rcall .+118     ; 0xfc <main>
  86: 58 c0        rjmp .+176     ; 0x138 <exit>

00000088 <__vector_10>:
#define LED_PIN  _BV(5)

enum { UP, DOWN };

ISR(TIMER1_CAPT_vect)           // Note [2]
{
  88: 1f 92        push r1
  8a: 1f b6        in r1, 0x3f ; 63
  8c: 1f 92        push r1
  8e: 11 24        eor r1, r1
  90: 2f 93        push r18
  92: 8f 93        push r24
  94: 9f 93        push r25

00000096 <.Loc.1>:
    // turn on LED
    LED_PORT |= LED_PIN;        // Note [3]
  96: 2d 9a        sbi 0x05, 5 ; 5

00000098 <.Loc.3>:

    static uint16_t pwm;        // Note [4]
    static uint8_t direction;

    switch (direction)          // Note [5]
  98: 20 91 02 01  lds r18, 0x0102 ; 0x800102 <direction.1>

0000009c <.Loc.6>:
    {
        case UP:
            if (++pwm == TIMER1_TOP)
  9c: 80 91 00 01  lds r24, 0x0100 ; 0x800100 <pwm.0>
  a0: 90 91 01 01  lds r25, 0x0101 ; 0x800101 <pwm.0+0x1>

000000a4 <.Loc.7>:
    switch (direction)          // Note [5]
  a4: 21 15        cp r18, r1
  a6: 89 f0        breq .+34      ; 0xca <.L2>
  a8: 21 30        cpi r18, 0x01 ; 1
  aa: d9 f0        breq .+54      ; 0xe2 <.L3>

000000ac <.L4>:
            if (--pwm == 0)
                direction = UP;
            break;
    }

    OCR1A = pwm;                // Note [6]
  ac: 80 91 00 01  lds r24, 0x0100 ; 0x800100 <pwm.0>
  b0: 90 91 01 01  lds r25, 0x0101 ; 0x800101 <pwm.0+0x1>
  b4: 90 93 89 00  sts 0x0089, r25 ; 0x800089 <__TEXT_REGION_LENGTH__+0x7f8089>
  b8: 80 93 88 00  sts 0x0088, r24 ; 0x800088 <__TEXT_REGION_LENGTH__+0x7f8088>

000000bc <.Loc.10>:
}
  bc: 9f 91        pop r25
  be: 8f 91        pop r24
  c0: 2f 91        pop r18
```

```
  c2: 1f 90        pop r1
  c4: 1f be        out 0x3f, r1 ; 63
  c6: 1f 90        pop r1
  c8: 18 95        reti

000000ca <.L2>:
            if (++pwm == TIMER1_TOP)
  ca: 01 96        adiw r24, 0x01 ; 1

000000cc <.Loc.13>:
  cc: 90 93 01 01  sts 0x0101, r25 ; 0x800101 <pwm.0+0x1>
  d0: 80 93 00 01  sts 0x0100, r24 ; 0x800100 <pwm.0>
  d4: 8f 3f        cpi r24, 0xFF ; 255
  d6: 93 40        sbci r25, 0x03 ; 3
  d8: 49 f7        brne .-46      ; 0xac <.L4>

000000da <.Loc.14>:
                direction = DOWN;
  da: 81 e0        ldi r24, 0x01 ; 1
  dc: 80 93 02 01  sts 0x0102, r24 ; 0x800102 <direction.1>
  e0: e5 cf        rjmp .-54      ; 0xac <.L4>

000000e2 <.L3>:
            if (--pwm == 0)
  e2: 01 97        sbiw r24, 0x01 ; 1

000000e4 <.Loc.18>:
  e4: 90 93 01 01  sts 0x0101, r25 ; 0x800101 <pwm.0+0x1>
  e8: 80 93 00 01  sts 0x0100, r24 ; 0x800100 <pwm.0>
  ec: 89 2b        or r24, r25
  ee: f1 f6        brne .-68      ; 0xac <.L4>

000000f0 <.Loc.19>:
                direction = UP;
  f0: 10 92 02 01  sts 0x0102, r1 ; 0x800102 <direction.1>
  f4: db cf        rjmp .-74      ; 0xac <.L4>

000000f6 <__vector_11>:

ISR(TIMER1_COMPA_vect)
{
    // turn off LED
    LED_PORT &= ~LED_PIN;       // Note [7]
  f6: 2d 98        cbi 0x05, 5 ; 5

000000f8 <.Loc.24>:
}
  f8: 18 95        reti

000000fa <__bad_interrupt>:
  fa: 82 cf        rjmp .-252     ; 0x0 <__vectors>

000000fc <main>:

static void
ioinit(void)                    // Note [8]
{
    /* Timer 1 in CTC mode with 10 bits, CPU/8 speed */
    ICR1 = TIMER1_TOP;
  fc: 8f ef        ldi r24, 0xFF ; 255
  fe: 93 e0        ldi r25, 0x03 ; 3
 100: 90 93 87 00  sts 0x0087, r25 ; 0x800087 <__TEXT_REGION_LENGTH__+0x7f8087>
 104: 80 93 86 00  sts 0x0086, r24 ; 0x800086 <__TEXT_REGION_LENGTH__+0x7f8086>

00000108 <.Loc.30>:
    TCCR1A = 0;
 108: 10 92 80 00  sts 0x0080, r1 ; 0x800080 <__TEXT_REGION_LENGTH__+0x7f8080>

0000010c <.Loc.32>:
    TCCR1B = _BV(WGM13) | _BV(WGM12) | 2;
 10c: 8a e1        ldi r24, 0x1A ; 26
 10e: 80 93 81 00  sts 0x0081, r24 ; 0x800081 <__TEXT_REGION_LENGTH__+0x7f8081>
```

```
00000112 <.Loc.34>:

    /* Set PWM value to 0. */
    OCR1A = 1;
 112: 81 e0        ldi r24, 0x01 ; 1
 114: 90 e0        ldi r25, 0x00 ; 0
 116: 90 93 89 00  sts 0x0089, r25 ; 0x800089 <__TEXT_REGION_LENGTH__+0x7f8089>
 11a: 80 93 88 00  sts 0x0088, r24 ; 0x800088 <__TEXT_REGION_LENGTH__+0x7f8088>

0000011e <.Loc.36>:

    /* Enable OC1 as output. */
    LED_DDR |= LED_PIN;
 11e: 25 9a        sbi 0x04, 5 ; 4

00000120 <.Loc.38>:

    /* Enable timer 1 overflow and compare A interrupt. */
    TIMSK1 = _BV(ICIE1) | _BV(OCIE1A);
 120: 82 e2        ldi r24, 0x22 ; 34
 122: 80 93 6f 00  sts 0x006F, r24 ; 0x80006f <__TEXT_REGION_LENGTH__+0x7f806f>

00000126 <.Loc.40>:
    sei ();
 126: 78 94        sei

00000128 <.L9>:
    ioinit ();

    /* Loop forever, the interrupts are doing the rest.  */

    for (;;)                    // Note [9]
        sleep_mode();
 128: 83 b7        in r24, 0x33 ; 51
 12a: 81 60        ori r24, 0x01 ; 1
 12c: 83 bf        out 0x33, r24 ; 51

0000012e <.Loc.45>:
 12e: 88 95        sleep

00000130 <.Loc.48>:
 130: 83 b7        in r24, 0x33 ; 51
 132: 8e 7f        andi r24, 0xFE ; 254
 134: 83 bf        out 0x33, r24 ; 51

00000136 <.Loc.51>:
    for (;;)                    // Note [9]
 136: f8 cf        rjmp .-16      ; 0x128 <.L9>

00000138 <exit>:
 138: f8 94        cli
 13a: 00 c0        rjmp .+0       ; 0x13c <_exit>

0000013c <_exit>:
 13c: f8 94        cli

0000013e <__stop_program>:
 13e: ff cf        rjmp .-2       ; 0x13e <__stop_program>
```

20.45.5 Linker Map Files

`avr-objdump` is very useful, but sometimes it's necessary to see information about the link that can only be generated by the linker. A map file contains this information. A map file is useful for monitoring the sizes of your code and data. It also shows where modules are loaded and which modules were loaded from libraries. It is yet another view of your application. To get a map file, I usually add **`-Wl,-Map,demo.map`** to my link command. Relink the application using the following command to generate `demo.map` (a portion of which is shown below). The linker will create a map file even in the case when some memory region overflows and the linker fails with an error. So you can collect valuable information about the cause of the overflow.

```
$ avr-gcc -g -mmcu=atmega328p -Wl,-Map,demo.map -mrelax -o demo.elf demo.o
```

Some points of interest in the `demo.map` file are:

```
.rela.plt
 *(.rela.plt)

.text           0x00000000      0x140
 *(.vectors)
 .vectors       0x00000000       0x68
      /home/joerg/src/avr-libc/build/avr/devices/atmega328p/crtatmega328p.o
                0x00000000                __vectors
                0x00000000                __vector_default
 *(.vectors)
 *(.progmem.gcc*)
                0x00000068                        . = ALIGN (0x2)
                0x00000068                        __trampolines_start = .
 *(.trampolines)
 .trampolines   0x00000068        0x0 linker stubs
 *(.trampolines*)
                0x00000068                        __trampolines_end = .
 *libprintf_flt.a:*(.progmem.data)
 *libc.a:*(.progmem.data)
 *(.progmem.*)
                0x00000068                        . = ALIGN (0x2)
 *(.lowtext)
 *(.lowtext*)
                0x00000068                        __ctors_start = .
```

The `.text` segment (where program instructions are stored) starts at location 0x0 and occupies 0x140 = 320 bytes. 0x68 = 104 bytes of that are allocated to interrupt vectors in `.vectors`, which is basically defined by the MCU hardware.

```
 *(.fini2)
 *(.fini2)
 *(.fini1)
 *(.fini1)
 *(.fini0)
 .fini0         0x0000013c        0x4 /usr/local/lib/gcc/avr/14.2.0/avr5/libgcc.a(_exit.o)
 *(.fini0)
 *(.hightext)
 *(.hightext*)
 *(.progmemx.*)
                0x00000140                        . = ALIGN (0x2)
 *(.jumptables)
 *(.jumptables*)
                0x00000140                        _etext = .

.data           0x00800100        0x0 load address 0x00000140
                [!provide]                        PROVIDE (__data_start = .)
 *(.data)
 .data          0x00800100        0x0 demo.o
 .data          0x00800100        0x0
      /home/joerg/src/avr-libc/build/avr/devices/atmega328p/crtatmega328p.o
 .data          0x00800100        0x0
      /usr/local/lib/gcc/avr/14.2.0/avr5/libgcc.a(_clear_bss.o)
 .data          0x00800100        0x0
      /home/joerg/src/avr-libc/build/avr/devices/atmega328p/libatmega328p.a(init_sp.o)
 .data          0x00800100        0x0
      /home/joerg/src/avr-libc/build/avr/devices/atmega328p/libatmega328p.a(call_main.o)
 .data          0x00800100        0x0
      /home/joerg/src/avr-libc/build/avr/lib/avr5/libc.a(exit.o)
 .data          0x00800100        0x0 /usr/local/lib/gcc/avr/14.2.0/avr5/libgcc.a(_exit.o)
 *(.data*)
 *(.gnu.linkonce.d*)
 *(.rodata)
 *(.rodata*)
 *(.gnu.linkonce.r*)
                0x00800100                        . = ALIGN (0x2)
                0x00800100                        _edata = .
                [!provide]                        PROVIDE (__data_end = .)

.bss            0x00800100        0x3
```

```
                0x00800100                        PROVIDE (__bss_start = .)
 *(.bss)
 .bss           0x00800100        0x3 demo.o
 .bss           0x00800103        0x0
      /home/joerg/src/avr-libc/build/avr/devices/atmega328p/crtatmega328p.o
 .bss           0x00800103        0x0
      /usr/local/lib/gcc/avr/14.2.0/avr5/libgcc.a(_clear_bss.o)
 .bss           0x00800103        0x0
      /home/joerg/src/avr-libc/build/avr/devices/atmega328p/libatmega328p.a(init_sp.o)
 .bss           0x00800103        0x0
      /home/joerg/src/avr-libc/build/avr/devices/atmega328p/libatmega328p.a(call_main.o)
 .bss           0x00800103        0x0
      /home/joerg/src/avr-libc/build/avr/lib/avr5/libc.a(exit.o)
 .bss           0x00800103        0x0 /usr/local/lib/gcc/avr/14.2.0/avr5/libgcc.a(_exit.o)
 *(.bss*)
 *(COMMON)
                0x00800103                        PROVIDE (__bss_end = .)
                0x00000140                        __data_load_start = LOADADDR (.data)
                0x00000140                        __data_load_end = (__data_load_start +
      SIZEOF (.data))

.noinit         0x00800103        0x0
                [!provide]                        PROVIDE (__noinit_start = .)
 *(.noinit .noinit.* .gnu.linkonce.n.*)
                [!provide]                        PROVIDE (__noinit_end = .)
                0x00800103                        _end = .
                [!provide]                        PROVIDE (__heap_start = .)
                0x00000000                        __flmap_init_label = DEFINED
      (__flmap_noinit_start)?__flmap_noinit_start:0x0
                0x00000000                        __flmap = DEFINED (__flmap)?__flmap:0x0

.eeprom         0x00810000        0x0
 *(.eeprom*)
                0x00810000                        __eeprom_end = .
```

The address one byte past the last address in the `.text` segment is denoted by `_etext`.

The `.data` segment (where initialized static variables are stored) starts at location 0x100, which is the first address after the IO registers on an ATmega328P processor. The segment is mapped to 0x800000 to simulate a flat address space.

The next available address in the `.data` segment is also location 0x100, so the application has no non-zero initialized data.

The `.bss` segment (where zero-initialized data is stored) starts at location 0x100.

The next available address in the `.bss` segment is location 0x103, so the application uses a total of 3 bytes of zero-initialized data, all from the `demo.o` module. These are the `pwm` (2 bytes) and `direction` (1 byte) static variables of the `TIMER1_CAPT_vect` ISR.

The range from `__data_load_start` to `__data_load_end` holds the initialization data for the `.data` segment, which is read by the startup code to initialize `.data` before main() starts. As the two symbols have the same value, there is nothing to copy to `.data` (since there are no variables in the `.data` input sections).

The `.eeprom` segment (where EEPROM variables are stored) starts at location 0x0, mapped to 0x810000 to simulate a flat address space.

The next available address in the `.eeprom` segment is also location 0x0, so there aren't any EEPROM variables.

20.45.6 Generating Intel Hex Files

We have a binary of the application, but how do we get it into the processor?

Many programming applications like AVRDUDE now accept the final ELF file as input, so no further step is needed there.

Some programmers require a specific kind of load file, like "Intel Hex" or "Motorola SRecord" format. For them, portions of the binary need to be extracted. The GNU utility that does this is called `avr-objcopy`.

The ROM contents can be pulled from our project's binary and put into the file demo.hex using the following command:

```
$ avr-objcopy -j .text -j .data -O ihex demo.elf demo.hex
```

The `-j` option indicates that we want the information from the `.text` and `.data` segment extracted. Note that the `.data` segment must be included as it contains the initialization values for the data segment, which the startup code expects to be placed immediately behind the instruction code (the `.text` segment) in flash. The startup code then copies them over to RAM before calling `main()`.

On devices with a `.rodata` segment, `-j .rodata` must be added to the options. (The ATmega328P doesn't have a `.rodata` segment since its `.rodata` input sections are allocated to the `.data` segment. See the FAQ for an explanation.)

The resulting `demo.hex` file contains:

```
:1000000033C000007AC0000078C0000076C0000055
:1000100074C0000072C0000070C000006EC000001C
:100020006CC000006AC000002FC0000064C0000067
:1000300064C0000062C0000060C000005EC000003C
:100040005CC000005AC0000058C0000056C000004C
:1000500054C0000052C0000050C000004EC000005C
:100060004CC000004AC0000011241FBECFEFD8E0F2
:10007000DEBFCDBF21E0A0E0B1E001C01D92A33002
:10008000B207E1F73BD058C01F921FB61F92112450
:100090002F938F939F932D9A2091020180910001BD
:1000A00090910101211589F02130D9F08091000152
:1000B000909101010190938900809388009F918F9186
:1000C0002F911F901FBE1F90189501969093010 1CC
:1000D000809300018F3F934049F781E080930201B4
:1000E000E5CF01979093010180930001892BF1F6F0
:1000F00010920201DBCF2D98189582CF8FEF93E0FD
:10010000909387008093860010928000 8AE180930C
:10011000810081E090E0909389008093880025 9A87
:1001200082E280936F00789483B7816083BF889563
:1001300083B78E7F83BFF8CFF89400C0F894FFCFC9
:00000001FF
```

If we specify the EEPROM segment, we can generate a `.hex` file that can be used to program the EEPROM:

```
$ avr-objcopy -j .eeprom --change-section-lma .eeprom=0 -O ihex demo.elf demo_eeprom.hex
```

There is no `demo_eeprom.hex` file written, as that file would be empty.

20.45.7 Letting Make Build the Project

Rather than type these commands over and over, they can all be placed in a make file. To build the demo project using GNU make, save the following in a file called `Makefile`.

This also adds a "program" target, so in order to download the image to the device's flash, it can simply called as

```
$ make program
```

The actual port (serial device) to talk to when programming can be specified in the environment variable `AVRDUDE_PORT`. There is some guesswork about possible ports (OS dependant) inside the Makefile if this variable does not exist.

Note

This `Makefile` can only be used as input for the GNU version of `make`.

```
PRG            = demo
OBJ            = demo.o

MCU_TARGET     = atmega328p

OPTIMIZE = -Os

WARN = -Wall -Wextra -Werror=missing-prototypes -Werror=strict-prototypes

DEFS =
LIBS =

# AVRDUDE configuration
# override that from the environment if needed
ifndef AVRDUDE_PORT
  ifeq ($(OS),Windows_NT)
    AVRDUDE_PORT = COM3
  else
    UNAME_S := $(shell uname -s)
    ifeq ($(UNAME_S),Linux)
        AVRDUDE_PORT = /dev/ttyUSB0
    endif
    ifeq ($(UNAME_S),Darwin)
        AVRDUDE_PORT = /dev/cu.usbserial-10
    endif
    ifeq ($(UNAME_S),FreeBSD)
        AVRDUDE_PORT = /dev/cuaU0
    endif
  endif
endif
ifndef AVRDUDE_PROGRAMMER
  AVRDUDE_PROGRAMMER = arduino
endif
ifndef AVRDUDE_ADDITIONAL
  AVRDUDE_ADDITIONAL = -b 57600 # for Arduino Nano compat device
endif
ifndef AVRDUDE
  AVRDUDE = avrdude # search along $PATH
endif

# You should not have to change anything below here.

CC      = avr-gcc

CFLAGS  = -g $(WARN) $(OPTIMIZE) -mmcu=$(MCU_TARGET) $(DEFS)
LDFLAGS = -Wl,-Map,$(PRG).map -mrelax

OBJCOPY = avr-objcopy
```

```
OBJDUMP = avr-objdump

all: $(PRG).elf lst text eeprom

$(PRG).elf: $(OBJ)
    $(CC) $(CFLAGS) $(LDFLAGS) -o $@ $^ $(LIBS)

# dependency:
demo.o: demo.c

clean:
    rm -f -- *.o $(PRG).elf
    rm -f -- *.lst *.map $(EXTRA_CLEAN_FILES)

lst:  $(PRG).lst

%.lst: %.elf
    $(OBJDUMP) -h -S $< > $@

# device programming

program: $(PRG).elf
    $(AVRDUDE) -c $(AVRDUDE_PROGRAMMER) \
      -p $(MCU_TARGET) -P $(AVRDUDE_PORT) \
      $(AVRDUDE_ADDITIONAL) \
      -U $(PRG).elf

.PHONY: all clean lst program

# Rules for building the .text rom images

text: hex bin srec

hex:  $(PRG).hex
bin:  $(PRG).bin
srec: $(PRG).srec

%.hex: %.elf
    $(OBJCOPY) -j .text -j .data -O ihex $< $@

%.srec: %.elf
    $(OBJCOPY) -j .text -j .data -O srec $< $@

%.bin: %.elf
    $(OBJCOPY) -j .text -j .data -O binary $< $@

.PHONY: text hex bin srec

# Rules for building the .eeprom rom images

eeprom: ehex ebin esrec

ehex:  $(PRG)_eeprom.hex
ebin:  $(PRG)_eeprom.bin
esrec: $(PRG)_eeprom.srec

%_eeprom.hex: %.elf
    $(OBJCOPY) -j .eeprom --change-section-lma .eeprom=0 -O ihex $< $@ \
    || { echo empty $@ not generated; exit 0; }

%_eeprom.srec: %.elf
    $(OBJCOPY) -j .eeprom --change-section-lma .eeprom=0 -O srec $< $@ \
    || { echo empty $@ not generated; exit 0; }

%_eeprom.bin: %.elf
    $(OBJCOPY) -j .eeprom --change-section-lma .eeprom=0 -O binary $< $@ \
    || { echo empty $@ not generated; exit 0; }

EXTRA_CLEAN_FILES += *.hex *.bin *.srec

.PHONY: eeprom ehex ebin esrec
```

20.45.8 Reference to the source code

The source code is installed under

`$prefix/share/doc/avr-libc/examples/demo/`,

where `$prefix` is a configuration option. For Unix systems, it is usually set to either `/usr` or `/usr/local`.

20.46 A more sophisticated project

This project extends the basic idea of the simple project to control a LED with a PWM output, but adds methods to adjust the LED brightness. It employs a lot of the basic concepts of AVR-LibC to achieve that goal.

Understanding this project assumes the simple project has been understood in full, as well as being acquainted with the basic hardware concepts of an AVR microcontroller.

20.46.1 Hardware setup

The demo is again arranged around an Arduino Nano compatible board. This time, the hardware PWM of the ATmega328P is used, so a LED (with resistor) needs to be attached to Arduino D8 (= PB1 = OC1A on the ATmega328P). Further, a potentiometer needs to be connected to ADC0 (Arduino A0), with GND and +5V at the outer terminals. Finally, three push buttons are connected to Arduino D2, D3, and D4, respectively (= PD2, PD3, PD4).

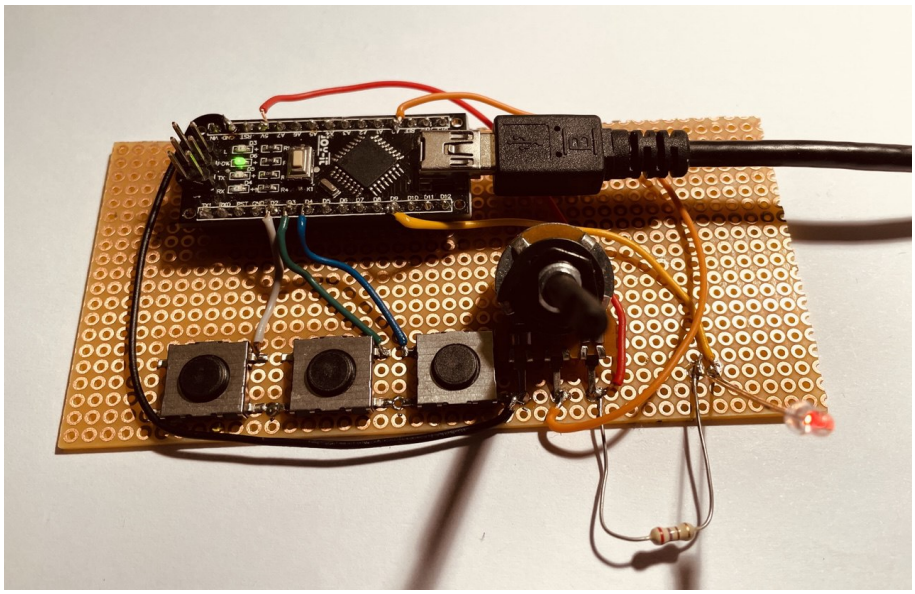

Figure 4 Setup of the Arduino Nano

Note that in the explanation below, all port/pin names are applicable to the ATmega328P setup.

20.46.2 Functional overview

PD6 (Arduino D6) will be toggled with each internal clock tick (approx. 10 ms). PD7 (Arduino D7) will flash (short low pulse) once per second.

PD0 and PD1 are configured as UART IO, and can be used to connect the demo kit to a PC (9600 Bd, 8N1 frame format). The demo application talks to the serial port, and it can be controlled from the serial port. On an Arduino Nano compatible board, this serial connection is routed to a USB-to-serial interface, so a PC can communicate through USB.

PD2 through PD4 are configured as inputs, and control the application unless control has been taken over by the serial port. Shorting PD2 to GND will decrease the current PWM value, shorting PD3 to GND will increase it.

While PD4 is shorted to GND, one ADC conversion for channel 0 (ADC input is on PA0) will be triggered each internal clock tick, and the resulting value will be used as the PWM value. So the brightness of the LED follows the analog input value on PC0. On the Arduino Nano, the ADC uses VCC internally as reference; all other setups require the reference voltage to be attached to the AREF pin.

When running in serial control mode, the function of the watchdog timer can be demonstrated by typing an `r'. This will make the demo application run in a tight loop without retriggering the watchdog so after some seconds, the watchdog will reset the MCU. This situation can be figured out on startup by reading the MCUCSR register.

The current value of the PWM is backed up in an EEPROM cell after about 3 seconds of idle time after the last change. If that EEPROM cell contains a reasonable (i. e. non-erased) value at startup, it is taken as the initial value for the PWM. This virtually preserves the last value across power cycles. By not updating the EEPROM immediately but only after a timeout, EEPROM wear is reduced considerably compared to immediately writing the value at each change.

20.46.3 A code walkthrough

This section explains the ideas behind individual parts of the code. The source code has been divided into numbered parts, and the following subsections explain each of these parts.

20.46.3.1 Part 1: Macro definitions

A number of preprocessor macros are defined to improve readability and/or portability of the application.

The first macros describe the IO pins our LEDs and pushbuttons are connected to. This provides some kind of mini-HAL (hardware abstraction layer) so should some of the connections be changed, they don't need to be changed inside the code but only on top. Note that the location of the PWM output itself is mandated by the hardware, so it cannot be easily changed. As the ATmega48/88/168/328P controllers belong to a more recent generation of AVRs, a number of register and bit names have been changed there, so they are mapped back to their ATmega8/16 equivalents to keep the actual program code portable.

The name `F_CPU` is the conventional name to describe the CPU clock frequency of the controller. This demo project just uses a 1 MHz CPU clock. On many AVR devices, this is the default startup frequency from the internal RC oscillator. On the Arduino Nano, the MCU is clocked by an external 16 MHz crystal though. Therefore, during device initialization, a 1:16 prescaler is configured at startup so the remaining code could remain the same. Note that when using the `<util/delay.h>` functions, `F_CPU` needs to be defined before including that file.

The remaining macros have their own comments in the source code. The macro `TMR1_SCALE` shows how to use the preprocessor and the compiler's constant expression computation to calculate the value of timer 1's post-scaler in a way so it only depends on `F_CPU` and the desired software clock frequency. While the formula looks a bit complicated, using a macro offers the advantage that the application will automatically scale to new target softclock or master CPU frequencies without having to manually re-calculate hardcoded constants.

20.46.3.2 Part 2: Variable definitions

The `intflags` structure demonstrates a way to allocate bit variables in memory. Each of the interrupt service routines just sets one bit within that structure, and the application's main loop then monitors the bits in order to act appropriately.

Like all variables that are used to communicate values between an interrupt service routine and the main application, it is declared volatile.

The variable `ee_pwm` is not a variable in the classical C sense that could be used as an lvalue or within an expression to obtain its value. Instead, the

```
uint16_t ee_pwm EEMEM ...
```

marks it as belonging to the EEPROM section. This section is merely used as a placeholder so the compiler can arrange for each individual variable's location in EEPROM. The compiler will also keep track of initial values assigned, and usually the Makefile is arranged to extract these initial values into a separate load file (`largedemo_eeprom.*` in this case) that can be used to initialize the EEPROM.

The actual EEPROM IO must be performed manually.

Similarly, the variable `mcucsr` is kept in the .noinit section in order to prevent it from being cleared upon application startup.

20.46.3.3 Part 3: Interrupt service routines

The ISR to handle timer 1's overflow interrupt arranges for the software clock. While timer 1 runs the PWM, it calls its overflow handler rather frequently, so the `TMR1_SCALE` value is used as a postscaler to reduce the internal software clock frequency further. If the software clock triggers, it sets the `tmr_int` bitfield, and defers all further tasks to the main loop.

The ADC ISR just fetches the value from the ADC conversion, disables the ADC interrupt again, and announces the presence of the new value in the `adc_int` bitfield. The interrupt is kept disabled while not needed, because the ADC will also be triggered by executing the SLEEP instruction in idle mode (which is the default sleep mode). Another option would be to turn off the ADC completely here, but that increases the ADC's startup time (not that it would matter much for this application).

20.46.3.4 Part 4: Auxiliary functions

20.46.3.4.1 handle_mcucsr() The function `handle_mcucsr()` uses `__attribute__` declarators to achieve specific goals:

`section(".init3")`
: First, it will instruct the compiler to place the generated code into the .init3 section of the output. Thus, it will become part of the application initialization sequence. This is done in order to fetch (and clear) the reason of the last hardware reset from `MCUCSR` as early as possible.

`naked`
: As the initialization code is not called using CALL/RET instructions but rather concatenated together, the compiler needs to be instructed to omit the entire function prologue and epilogue. This is performed by the *naked* attribute. So while syntactically, `handle_mcucsr()` is a function to the compiler, the compiler will just emit the instructions for it without setting up any stack frame, and not even a RET instruction at the end. Notice that this code is not strictly conforming to GCC requirements, which documents that `naked` functions should only contain inline asm statements. However, the code is simple enough so it doesn't require a stack frame as the project is compiled with optimizations turned on.

`used`
: Tell the compiler that handle_mcucsr() is used by the code, even though the function is never called. This keeps the compiler from optimizing out a seemingly unused function.

There is a short period of time where the next reset could already trigger before the current reason has been evaluated. This also explains why the variable `mcucsr` that mirrors the register's value needs to be placed into the .noinit section, because otherwise the default initialization (which happens after `.init3`) would blank the value again.

20.46.3.4.2 ioinit() Function `ioinit()` centralizes all hardware setup. The very last part of that function demonstrates the use of the EEPROM variable `ee_pwm` to obtain an EEPROM address that can in turn be applied as an argument to `eeprom_read_word()`.

20.46.3.4.3 UART Output The following functions handle UART character and string output. (UART input is handled by an ISR.) There are two string output functions, `printstr()` and `printstr_p()`. The latter function fetches the string from program memory. Both functions translate a newline character into a carriage return/newline sequence, so a simple `\n` can be used in the source code.

20.46.3.4.4 set_pwm() The function `set_pwm()` propagates the new PWM value to the PWM, performing range checking. When the value has been changed, the new percentage will be announced on the serial link. The current value is mirrored in the variable `pwm` so others can use it in calculations. In order to allow for a simple calculation of a percentage value without requiring floating-point mathematics, the maximal value of the PWM is restricted to 1000 rather than 1023, so a simple division by 10 can be used. Due to the nature of the human eye, the difference in LED brightness between 1000 and 1023 is not noticeable anyway.

20.46.3.5 Part 5: main()

At the start of `main()`, a variable `mode` is declared to keep the current mode of operation. An enumeration is used to improve the readability. By default, the compiler would allocate a variable of type *int* for an enumeration. The *packed* attribute declarator instructs the compiler to use the smallest possible integer type (which would be an 8-bit type here).

After some initialization actions, the application's main loop follows. In an embedded application, this is normally an infinite loop as there is nothing an application could "exit" into anyway.

At the beginning of the loop, the watchdog timer will be retriggered. If that timer is not triggered for about 2 seconds, it will issue a hardware reset. Care needs to be taken that no code path blocks longer than this, or it needs to frequently perform watchdog resets of its own. An example of such a code path would be the string IO functions: for an overly large string to print (about 2000 characters at 9600 Bd), they might block for too long.

The loop itself then acts on the interrupt indication bitfields as appropriate, and will eventually put the CPU on sleep at its end to conserve power.

The first interrupt bit that is handled is the (software) timer, at a frequency of approximately 100 Hz. The `CLOCKOUT` pin will be toggled here, so e. g. an oscilloscope can be used on that pin to measure the accuracy of our software clock. Then, the LED flasher for LED2 ("We are alive"-LED) is built. It will flash that LED for about 50 ms, and pause it for another 950 ms. Various actions depending on the operation mode follow. Finally, the 3-second backup timer is implemented that will write the PWM value back to EEPROM once it is not changing anymore.

The ADC interrupt will just adjust the PWM value only.

Finally, the UART Rx interrupt will dispatch on the last character received from the UART.

All the string literals that are used as informational messages within `main()` are placed in program memory so no SRAM needs to be allocated for them. This is done by using the PSTR macro for the string argument of `printstr_p()`.

20.46.4 The source code

The source code is installed under

`$prefix/share/doc/avr-libc/examples/largedemo/largedemo.c`,

where `$prefix` is a configuration option. For Unix systems, it is usually set to either `/usr` or `/usr/local`.

20.47 Using the standard IO facilities

This project illustrates how to use the standard IO facilities (stdio) provided by this library. It assumes a basic knowledge of how the stdio subsystem is used in standard C applications, and concentrates on the differences in this library's implementation that mainly result from the differences of the microcontroller environment, compared to a hosted environment of a standard computer.

This demo is meant to supplement the documentation, not to replace it.

20.47.1 Hardware setup

The demo is set up in a way so it can be run on an Arduino Nano compatible board. This board contains a USB-to-serial bridge that attaches to USART0 of the ATmega328P MCU, so it can easily communicate with a standard terminal program on the host PC. This channel is used as standard input (`stdin`) and standard output (`stdout`), respectively.

In order to have a different device available for a standard error channel (`stderr`), an industry-standard LCD display with an HD44780-compatible LCD controller has been chosen. This display needs to be connected to the Arduino Nano in the following way:

Table 76 Arduino Nano ↔ HD44780 LCD Wiring

Port	Header	Function
D2	D2	LCD D4
D3	D3	LCD D5
D4	D4	LCD D6
D5	D5	LCD D7
D6	D6	LCD E
D7	D7	LCD RS
B0	D8	LCD R/∼W
GND	GND	GND, V5
VCC	5V	Vcc

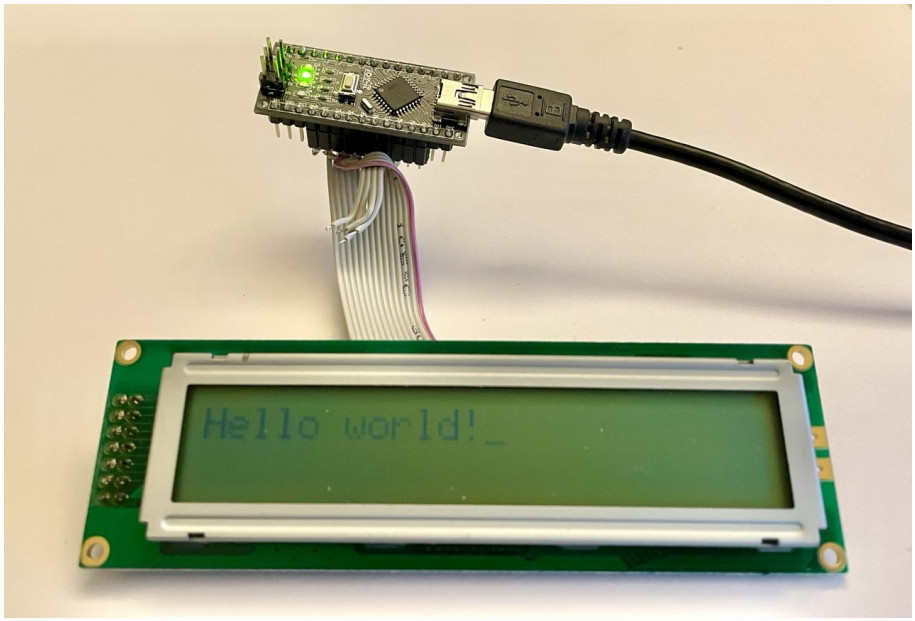

Figure 5 Wiring of the Arduino Nano

The LCD controller is used in 4-bit mode, including polling the "busy" flag so the R/∼W line from the LCD controller needs to be connected. Note that the LCD controller has yet another supply pin that is used to adjust the LCD's contrast (V5). Typically, that pin connects to a potentiometer between Vcc and GND. Often, it might work to just connect that pin to GND, while leaving it unconnected usually yields an unreadable display.

These pins have been chosen as they are all in a single row on the Arduino Nano, normally designated as "digital IO" in the Arduino Framework. The only important consideration is that the HD44780 data connections D4 through D7 must be in consecutive order on a single AVR MCU port.

20.47.2 Functional overview

The project consists of the following files:

- `stdiodemo.c` This is the main example file.
- `defines.h` Contains some global defines, like the LCD wiring
- `hd44780.c` Implementation of an HD44780 LCD display driver
- `hd44780.h` Interface declarations for the HD44780 driver
- `lcd.c` Implementation of LCD character IO on top of the HD44780 driver
- `lcd.h` Interface declarations for the LCD driver
- `uart.c` Implementation of a character IO driver for the internal UART
- `uart.h` Interface declarations for the UART driver

20.47.3 A code walkthrough

20.47.3.1 stdiodemo.c

As usual, include files go first. While conventionally, system header files (those in angular brackets < ... >) go before application-specific header files (in double quotes), `defines.h` comes as the first header file here. The main reason is that this file defines the value of `F_CPU` which needs to be known before including `<utils/delay.h>`.

The function `ioinit()` summarizes all hardware initialization tasks. As this function is declared to be module-internal only (`static`), the compiler will notice its simplicity, and with a reasonable optimization level in effect, it will inline that function. That needs to be kept in mind when debugging, because the inlining might cause the debugger to "jump around wildly" at a first glance when single-stepping.

The definitions of `uart_str` and `lcd_str` set up two stdio streams. The initialization is done using the `FDEV_SETUP_STREAM()` initializer template macro, so a static object can be constructed that can be used for IO purposes. This initializer macro takes three arguments, two function macros to connect the corresponding output and input functions, respectively, the third one describes the intent of the stream (read, write, or both). Those functions that are not required by the specified intent (like the input function for `lcd_str` which is specified to only perform output operations) can be given as `NULL`.

The stream `uart_str` corresponds to input and output operations performed over the USB-serial connection to a terminal (e.g. from/to a PC running a terminal program), while the `lcd_str` stream provides a method to display character data on the LCD text display.

The function `delay_1s()` suspends program execution for approximately one second. This is done using the `_delay_ms()` function from `<util/delay.h>` which in turn needs the `F_CPU` macro in order to adjust the cycle counts. As the `_delay_ms()` function has a limited range of allowable argument values (depending on

F_CPU), a value of 10 ms has been chosen as the base delay which would be safe for CPU frequencies of up to about 26 MHz. This function is then called 100 times to accommodate for the actual one-second delay.

In a practical application, long delays like this one were better be handled by a hardware timer, so the main CPU would be free for other tasks while waiting, or could be put on sleep.

At the beginning of main(), after initializing the peripheral devices, the default stdio streams stdin, stdout, and stderr are set up by using the existing static FILE stream objects. While this is not mandatory, the availability of stdin and stdout allows to use the shorthand functions (e.g. printf() instead of fprintf()), and stderr can mnemonically be referred to when sending out diagnostic messages.

Just for demonstration purposes, stdin and stdout are connected to a stream that will perform UART IO, while stderr is arranged to output its data to the LCD text display.

Finally, a main loop follows that accepts simple "commands" entered via the serial connection, and performs a few simple actions based on the commands.

First, a prompt is sent out using printf_P() (which takes a program space string). The string is read into an internal buffer as one line of input, using fgets(). While it would be also possible to use gets() (which implicitly reads from stdin), gets() has no control that the user's input does not overflow the input buffer provided so it should never be used at all.

If fgets() fails to read anything, the main loop is left. Of course, normally the main loop of a microcontroller application is supposed to never finish, but again, for demonstrational purposes, this explains the error handling of stdio. fgets() will return NULL in case of an input error or end-of-file condition on input. Both these conditions are in the domain of the function that is used to establish the stream, uart_putchar() in this case. In short, this function returns EOF in case of a serial line "break" condition (extended start condition) has been recognized on the serial line. Common PC terminal programs allow to assert this condition as some kind of out-of-band signalling on an RS-232 connection.

When leaving the main loop, a goodbye message is sent to standard error output (i.e. to the LCD), followed by three dots in one-second spacing, followed by a sequence that will clear the LCD. Finally, main() will be terminated, and the library will add an infinite loop, so only a CPU reset will be able to restart the application.

There are four "commands" recognized, each determined by the first letter of the line entered (converted to lower case):

- The 'q' (quit) command has the same effect of leaving the main loop.
- The 'l' (LCD) command takes its second argument, and sends it to the LCD.
- The 'u' (UART) command takes its second argument, and sends it back to the UART connection.
- The 'd' (LED) command takes an integer number as its second argument. If the number is 0, the onboard LED is turned off, with all other values, it is turned on.

Command recognition is done using sscanf() where the first format in the format string just skips over the command itself (as the assignment suppression modifier * is given).

20.47.3.2 defines.h

This file just contains a few peripheral definitions.

The F_CPU macro defines the CPU clock frequency, to be used in delay loops, as well as in the UART baud rate calculation.

The macro UART_BAUD defines the USART baud rate. Depending on the actual CPU frequency, only a limited range of baud rates can be supported.

The remaining macros customize the IO port and pins used for the HD44780 LCD driver. Each definition consists of a letter naming the port this pin is attached to, and a respective bit number. For accessing the data lines, only the first data line gets its own macro (line D4 on the HD44780, lines D0 through D3 are not used in 4-bit mode), all other data lines are expected to be in ascending order next to D4.

Finally, the LED port definitions are similar to those from the simple demo.

20.47.3.3 hd44780.h

This file describes the public interface of the low-level LCD driver that interfaces to the HD44780 LCD controller. Public functions are available to initialize the controller into 4-bit mode, to wait for the controller's busy bit to be clear, and to read or write one byte from or to the controller.

As there are two different forms of controller IO, one to send a command or receive the controller status (RS signal clear), and one to send or receive data to/from the controller's SRAM (RS asserted), macros are provided that build on the mentioned function primitives.

Finally, macros are provided for all the controller commands to allow them to be used symbolically. The HD44780 datasheet explains these basic functions of the controller in more detail.

20.47.3.4 hd44780.c

This is the implementation of the low-level HD44780 LCD controller driver.

On top, a few preprocessor glueing tricks are used to establish symbolic access to the hardware port pins the LCD controller is attached to, based on the application's definitions made in defines.h.

The `hd44780_pulse_e()` function asserts a short pulse to the controller's E (enable) pin. Since reading back the data asserted by the LCD controller needs to be performed while E is active, this function reads and returns the input data if the parameter `readback` is true. When called with a compile-time constant parameter that is false, the compiler will completely eliminate the unused readback operation, as well as the return value as part of its optimizations.

As the controller is used in 4-bit interface mode, all byte IO to/from the controller needs to be handled as two nibble IOs. The functions `hd44780_outnibble()` and `hd44780_innibble()` implement this. They do not belong to the public interface, so they are declared static.

Building upon these, the public functions `hd44780_outbyte()` and `hd44780_inbyte()` transfer one byte to/from the controller.

The function `hd44780_wait_ready()` waits for the controller to become ready, by continuously polling the controller's status (which is read by performing a byte read with the RS signal cleard), and examining the BUSY flag within the status byte. This function needs to be called before performing any controller IO.

Finally, `hd44780_init()` initializes the LCD controller into 4-bit mode, based on the initialization sequence mandated by the datasheet. As the BUSY flag cannot be examined yet at this point, this is the only part of this code where timed delays are used. While the controller can perform a power-on reset when certain constraints on the power supply rise time are met, always calling the software initialization routine at startup ensures the controller will be in a known state. This function also puts the interface into 4-bit mode (which would not be done automatically after a power-on reset).

20.47.3.5 lcd.h

This function declares the public interface of the higher-level (character IO) LCD driver.

20.47.3.6 lcd.c

The implementation of the higher-level LCD driver. This driver builds on top of the HD44780 low-level LCD controller driver, and offers a character IO interface suitable for direct use by the standard IO facilities. Where the low-level HD44780 driver deals with setting up controller SRAM addresses, writing data to the controller's SRAM, and controlling display functions like clearing the display, or moving the cursor, this high-level driver allows to just write a character to the LCD, in the assumption this will somehow show up on the display.

Control characters can be handled at this level, and used to perform specific actions on the LCD. Currently, there is only one control character that is being dealt with: a newline character (`\n`) is taken as an indication to clear the display and set the cursor into its initial position upon reception of the next character, so a "new line" of text can be displayed. Therefore, a received newline character is remembered until more characters have been sent by the application, and will only then cause the display to be cleared before continuing. This provides a convenient abstraction where full lines of text can be sent to the driver, and will remain visible at the LCD until the next line is to be displayed.

Further control characters could be implemented, e. g. using a set of escape sequences. That way, it would be possible to implement self-scrolling display lines etc.

The public function `lcd_init()` first calls the initialization entry point of the lower-level HD44780 driver, and then sets up the LCD in a way we'd like to (display cleared, non-blinking cursor enabled, SRAM addresses are increasing so characters will be written left to right).

The public function `lcd_putchar()` takes arguments that make it suitable for being passed as a `put()` function pointer to the stdio stream initialization functions and macros (`fdevopen()`, `FDEV_SETUP_STREAM()` etc.). Thus, it takes two arguments, the character to display itself, and a reference to the underlying stream object, and it is expected to return 0 upon success.

This function remembers the last unprocessed newline character seen in the function-local static variable `nl_seen`. If a newline character is encountered, it will simply set this variable to a true value, and return to the caller. As soon as the first non-newline character is to be displayed with `nl_seen` still true, the LCD controller is told to clear the display, put the cursor home, and restart at SRAM address 0. All other characters are sent to the display.

The single static function-internal variable `nl_seen` works for this purpose. If multiple LCDs should be controlled using the same set of driver functions, that would not work anymore, as a way is needed to distinguish between the various displays. This is where the second parameter can be used, the reference to the stream itself: instead of keeping the state inside a private variable of the function, it can be kept inside a private object that is attached to the stream itself. A reference to that private object can be attached to the stream (e.g. inside the function `lcd_init()` that then also needs to be passed a reference to the stream) using `fdev_set_udata()`, and can be accessed inside `lcd_putchar()` using fdev_get_udata().

20.47.3.7 uart.h

Public interface definition for the serial UART driver, much like in lcd.h except there is now also a character input function available.

As the serial input is line-buffered in this example, the macro `RX_BUFSIZE` determines the size of that buffer.

20.47.3.8 uart.c

This implements an stdio-compatible serial driver using an AVR's standard UART (or USART in asynchronous operation mode). Both, character output as well as character input operations are implemented. Character output takes care of converting the internal newline `\n` into its external representation carriage return/line feed (`\r\n`).

Character input is organized as a line-buffered operation that allows to minimally edit the current line until it is "sent" to the application when either a carriage return (`\r`) or newline (`\n`) character is received from the terminal. The line editing functions implemented are:

- \b (back space) or \177 (delete) deletes the previous character
- ^u (control-U, ASCII NAK) deletes the entire input buffer
- ^w (control-W, ASCII ETB) deletes the previous input word, delimited by white space
- ^r (control-R, ASCII DC2) sends a \r, then reprints the buffer (refresh)
- \t (tabulator) will be replaced by a single space

The function uart_init() takes care of all hardware initialization that is required to put the UART into a mode with 8 data bits, no parity, one stop bit (commonly referred to as 8N1) at the baud rate configured in defines.h. At low CPU clock frequencies, the U2X bit in the UART is set, reducing the oversampling from 16x to 8x, which allows for a 9600 Bd rate to be achieved with tolerable error using the default 1 MHz RC oscillator.

The public function uart_putchar() again has suitable arguments for direct use by the stdio stream interface. It performs the \n into \r\n translation by recursively calling itself when it sees a \n character. Just for demonstration purposes, the \a (audible bell, ASCII BEL) character is implemented by sending a string to stderr, so it will be displayed on the LCD.

The public function uart_getchar() implements the line editor. If there are characters available in the line buffer (variable rxp is not NULL), the next character will be returned from the buffer without any UART interaction.

If there are no characters inside the line buffer, the input loop will be entered. Characters will be read from the UART, and processed accordingly. If the UART signalled a framing error (FE bit set), typically caused by the terminal sending a *line break* condition (start condition held much longer than one character period), the function will return an end-of-file condition using _FDEV_EOF. If there was a data overrun condition on input (DOR bit set), an error condition will be returned as _FDEV_ERR.

Line editing characters are handled inside the loop, potentially modifying the buffer status. If characters are attempted to be entered beyond the size of the line buffer, their reception is refused, and a \a character is sent to the terminal. If a \r or \n character is seen, the variable rxp (receive pointer) is set to the beginning of the buffer, the loop is left, and the first character of the buffer will be returned to the application. (If no other characters have been entered, this will just be the newline character, and the buffer is marked as being exhausted immediately again.)

20.47.4 The source code

The source code is installed under

$prefix/share/doc/avr-libc/examples/stdiodemo/,

where $prefix is a configuration option. For Unix systems, it is usually set to either /usr or /usr/local.

20.48 Example using the two-wire interface (TWI)

Some newer devices of the ATmega series contain builtin support for interfacing the microcontroller to a two-wire bus, called TWI. This is essentially the same called I^2C by Philips, but that term is avoided in Atmel's documentation due to patenting issues.

20.48.1 Introduction into TWI

The two-wire interface consists of two signal lines named *SDA* (serial data) and *SCL* (serial clock) (plus a ground line, of course). All devices participating in the bus are connected together, using open-drain driver circuitry, so the wires must be terminated using appropriate pullup resistors. The pullups must be small enough to recharge the line capacity in short enough time compared to the desired maximal clock frequency, yet large enough so all drivers will not be overloaded. There are formulas in the datasheet that help selecting the pullups.

Devices can either act as a master to the bus (i. e., they initiate a transfer), or as a slave (they only act when being called by a master). The bus is multi-master capable, and a particular device implementation can act as either master or slave at different times. Devices are addressed using a 7-bit address (coordinated by Philips) transferred as the first byte after the so-called start condition. The LSB of that byte is R/∼W, i. e. it determines whether the request to the slave is to read or write data during the next cycles. (There is also an option to have devices using 10-bit addresses but that is not covered by this example.)

20.48.2 The TWI example project

The ATmega TWI hardware supports both, master and slave operation. This example will only demonstrate how to use an AVR microcontroller as TWI master. The implementation is kept simple in order to concentrate on the steps that are required to talk to a TWI slave, so all processing is done in polled-mode, waiting for the TWI interface to indicate that the next processing step is due (by setting the TWINT interrupt bit). If it is desired to have the entire TWI communication happen in "background", all this can be implemented in an interrupt-controlled way, where only the start condition needs to be triggered from outside the interrupt routine.

There is a variety of slave devices available that can be connected to a TWI bus. For the purpose of this example, an EEPROM device out of the industry-standard **24C*xx*** series has been chosen (where *xx* can be one of **01**, **02**, **04**, **08**, or **16**) which are available from various vendors. The choice was almost arbitrary, mainly triggered by the fact that an EEPROM device is being talked to in both directions, reading and writing the slave device, so the example will demonstrate the details of both.

Usually, there is probably not much need to add more EEPROM to an ATmega system that way: the smallest possible AVR device that offers hardware TWI support is the ATmega8 which comes with 512 bytes of EEPROM, which is equivalent to an 24C04 device. The ATmega128 already comes with twice as much EEPROM as the 24C16 would offer. One exception might be to use an externally connected EEPROM device that is removable; e. g. SDRAM PC memory comes with an integrated TWI EEPROM that carries the RAM configuration information.

20.48.3 The Source Code

The source code is installed under

`$prefix/share/doc/avr-libc/examples/twitest/twitest.c`,

where `$prefix` is a configuration option. For Unix systems, it is usually set to either `/usr` or `/usr/local`.

Note [1]

The header file `<util/twi.h>` contains some macro definitions for symbolic constants used in the TWI status register. These definitions match the names used in the Atmel datasheet except that all names have been prefixed with `TW_`.

Note [2]

The clock is used in timer calculations done by the compiler, for the UART baud rate and the TWI clock rate.

Note [3]

The address assigned for the 24Cxx EEPROM consists of 1010 in the upper four bits. The following three bits are normally available as slave sub-addresses, allowing to operate more than one device of the same type on a single bus, where the actual subaddress used for each device is configured by hardware strapping. However, since the next data packet following the device selection only allows for 8 bits that are used as an EEPROM address, devices that require more than 8 address bits (24C04 and above) "steal" subaddress bits and use them for the EEPROM cell address bits 9 to 11 as required. This example simply assumes all subaddress bits are 0 for the smaller devices, so the E0, E1, and E2 inputs of the 24Cxx must be grounded.

Note [3a]

EEPROMs of type 24C32 and above cannot be addressed anymore even with the subaddress bit trick. Thus, they require the upper address bits being sent separately on the bus. When activating the `WORD_ADDRESS_16BIT` define, the algorithm implements that auxiliary address byte transmission.

Note [4]

For slow clocks, enable the 2 x U[S]ART clock multiplier, to improve the baud rate error. This will allow a 9600 Bd communication using the standard 1 MHz calibrated RC oscillator. See also the Baud rate tables in the datasheets.

Note [5]

The datasheet explains why a minimum TWBR value of 10 should be maintained when running in master mode. Thus, for system clocks below 3.6 MHz, we cannot run the bus at the intended clock rate of 100 kHz but have to slow down accordingly.

Note [6]

This function is used by the standard output facilities that are utilized in this example for debugging and demonstration purposes.

Note [7]

In order to shorten the data to be sent over the TWI bus, the 24Cxx EEPROMs support multiple data bytes transferred within a single request, maintaining an internal address counter that is updated after each data byte transferred successfully. When reading data, one request can read the entire device memory if desired (the counter would wrap around and start back from 0 when reaching the end of the device).

Note [8]

When reading the EEPROM, a first device selection must be made with write intent (R/∼W bit set to 0 indicating a write operation) in order to transfer the EEPROM address to start reading from. This is called *master transmitter mode*. Each completion of a particular step in TWI communication is indicated by an asserted TWINT bit in TWCR. (An interrupt would be generated if allowed.) After performing any actions that are needed for the next communication step, the interrupt condition must be manually cleared by *setting* the TWINT bit. Unlike with many other interrupt sources, this would even be required when using a true interrupt routine, since as soon as TWINT is re-asserted, the next bus transaction will start.

Note [9]

Since the TWI bus is multi-master capable, there is potential for a bus contention when one master starts to access the bus. Normally, the TWI bus interface unit will detect this situation, and will not initiate a start condition while the bus is busy. However, in case two masters were starting at exactly the same time, the way bus arbitration works, there is always a chance that one master could lose arbitration of the bus during any transmit operation. A master that has lost arbitration is required by the protocol to immediately cease talking on the bus; in particular it must not initiate a stop condition in order to not corrupt the ongoing transfer from the active master. In this example, upon detecting a lost arbitration condition, the entire transfer is going to be restarted. This will cause a new start condition to be initiated, which will normally be delayed until the currently active master has released the bus.

Note [10]

Next, the device slave is going to be reselected (using a so-called repeated start condition which is meant to guarantee that the bus arbitration will remain at the current master) using the same slave address (SLA), but this time with read intent (R/∼W bit set to 1) in order to request the device slave to start transferring data from the slave to the master in the next packet.

Note [11]

If the EEPROM device is still busy writing one or more cells after a previous write request, it will simply leave its bus interface drivers at high impedance, and does not respond to a selection in any way at all. The master selecting the device will see the high level at SDA after transferring the SLA+R/W packet as a NACK to its selection request. Thus, the select process is simply started over (effectively causing a *repeated start condition*), until the device will eventually respond. This polling procedure is recommended in the 24Cxx datasheet in order to minimize the busy wait time when writing. Note that in case a device is broken and never responds to a selection (e. g. since it is no longer present at all), this will cause an infinite loop. Thus the maximal number of iterations made until the device is declared to be not responding at all, and an error is returned, will be limited to MAX_ITER.

Note [12]

This is called *master receiver mode*: the bus master still supplies the SCL clock, but the device slave drives the SDA line with the appropriate data. After 8 data bits, the master responds with an ACK bit (SDA driven low) in order to request another data transfer from the slave, or it can leave the SDA line high (NACK), indicating to the slave that it is going to stop the transfer now. Assertion of ACK is handled by setting the TWEA bit in TWCR when starting the current transfer.

Note [13]

The control word sent out in order to initiate the transfer of the next data packet is initially set up to assert the TWEA bit. During the last loop iteration, TWEA is de-asserted so the client will get informed that no further transfer is desired.

Note [14]

Except in the case of lost arbitration, all bus transactions must properly be terminated by the master initiating a stop condition.

Note [15]

Writing to the EEPROM device is simpler than reading, since only a master transmitter mode transfer is needed. Note that the first packet after the SLA+W selection is always considered to be the EEPROM address for the next operation. (This packet is exactly the same as the one above sent before starting to read the device.) In case a master transmitter mode transfer is going to send more than one data packet, all following packets will be considered data bytes to write at the indicated address. The internal address pointer will be incremented after each write operation.

Note [16]

24Cxx devices can become write-protected by strapping their ~WC pin to logic high. (Leaving it unconnected is explicitly allowed, and constitutes logic low level, i. e. no write protection.) In case of a write protected device, all data transfer attempts will be NACKed by the device. Note that some devices might not implement this.

21 Data Structure Documentation

21.1 div_t Struct Reference

```
#include <stdlib.h>
```

Data Fields

- int quot
- int rem

21.1.1 Detailed Description

Result type for function div().

21.1.2 Field Documentation

21.1.2.1 quot

```
int div_t::quot
```

The Quotient.

21.1.2.2 rem

```
int div_t::rem
```

The Remainder.

The documentation for this struct was generated from the following file:

- stdlib.h

21.2 ldiv_t Struct Reference

```
#include <stdlib.h>
```

Data Fields

- long quot
- long rem

21.2.1 Detailed Description

Result type for function ldiv().

21.2.2 Field Documentation

21.2.2.1 quot

```
long ldiv_t::quot
```

The Quotient.

21.2.2.2 rem

```
long ldiv_t::rem
```

The Remainder.

The documentation for this struct was generated from the following file:

- stdlib.h

21.3 tm Struct Reference

```
#include <time.h>
```

Data Fields

- int8_t tm_sec
- int8_t tm_min
- int8_t tm_hour
- int8_t tm_mday
- int8_t tm_wday
- int8_t tm_mon
- int16_t tm_year
- int16_t tm_yday
- int16_t tm_isdst

21.3.1 Detailed Description

The `tm` structure contains a representation of time 'broken down' into components of the Gregorian calendar.

The value of tm_isdst is zero if Daylight Saving Time is not in effect, and is negative if the information is not available.

When Daylight Saving Time is in effect, the value represents the number of seconds the clock is advanced.

See the `set_dst()` function for more information about Daylight Saving.

21.3.2 Field Documentation

21.3.2.1 tm_hour

```
int8_t tm::tm_hour
```

Hours since midnight (0 to 23)

21.3.2.2 tm_isdst

```
int16_t tm::tm_isdst
```

Daylight Saving Time flag

21.3.2.3 tm_mday

```
int8_t tm::tm_mday
```

Day of the month (1 to 31)

21.3.2.4 tm_min

```
int8_t tm::tm_min
```

Minutes after the hour (0 to 59)

21.3.2.5 tm_mon

```
int8_t tm::tm_mon
```

Months since January (0 to 11)

21.3.2.6 tm_sec

```
int8_t tm::tm_sec
```

Seconds after the minute (0 to 59)

21.3.2.7 tm_wday

```
int8_t tm::tm_wday
```

Days since Sunday (0 to 6)

21.3.2.8 tm_yday

```
int16_t tm::tm_yday
```

Days since January 1 (0 to 365)

21.3.2.9 tm_year

```
int16_t tm::tm_year
```

Years since 1900

The documentation for this struct was generated from the following file:

- time.h

21.4 week_date Struct Reference

`#include <time.h>`

Data Fields

- int year
- int week
- int day

21.4.1 Detailed Description

Structure which represents a date as a year, week number of that year, and day of week. See http://en.wikipedia.org/wiki/ISO_week_date for more information.

21.4.2 Field Documentation

21.4.2.1 day

`int week_date::day`

Day within week

21.4.2.2 week

`int week_date::week`

Week number (#1 is where first Thursday is in)

21.4.2.3 year

`int week_date::year`

Year number (Gregorian calendar)

The documentation for this struct was generated from the following file:

- time.h

22 File Documentation

22.1 stdfix-avrlibc.h File Reference

Macros

- #define FXTOA_ALL 0x1f
- #define FXTOA_ROUND 0x00
- #define FXTOA_TRUNC 0x80
- #define FXTOA_NTZ 0x40
- #define FXTOA_DOT 0x00
- #define FXTOA_COMMA 0x20

Functions

ASCII Conversions (not in ISO/IEC TR18037)

- char * hktoa (short accum x, char *buf, unsigned char mode)
- char * hrtoa (short fract x, char *buf, unsigned char mode)
- char * ktoa (accum x, char *buf, unsigned char mode)
- char * rtoa (fract x, char *buf, unsigned char mode)
- char * uhktoa (unsigned short accum x, char *buf, unsigned char mode)
- char * uhrtoa (unsigned short fract x, char *buf, unsigned char mode)
- char * uktoa (unsigned accum x, char *buf, unsigned char mode)
- char * urtoa (unsigned fract x, char *buf, unsigned char mode)

Absolute Value

- short fract abshr (short fract val)
- fract absr (fract val)
- long fract abslr (long fract val)
- long long fract absllr (long long fract val)
- short accum abshk (short accum val)
- accum absk (accum val)
- long accum abslk (long accum val)
- long long accum absllk (long long accum val)

Bit-Conversions to Integer

- signed char bitshr (short fract val)
- unsigned char bitsuhr (unsigned short fract val)
- int bitsr (fract val)
- unsigned int bitsur (unsigned fract val)
- long bitslr (long fract val)
- unsigned long bitsulr (unsigned long fract val)
- long long bitsllr (long long fract val)
- unsigned long long bitsullr (unsigned long long fract val)
- int bitshk (short accum val)
- unsigned int bitsuhk (unsigned short accum val)
- long bitsk (accum val)
- unsigned long bitsuk (unsigned accum val)
- long long bitslk (long accum val)
- unsigned long long bitsulk (unsigned long accum val)
- long long bitsllk (long long accum val)
- unsigned long long bitsullk (unsigned long long accum val)

Bit-Conversions to Fixed-Point

- short fract hrbits (signed char val)
- unsigned short fract uhrbits (unsigned char val)
- fract rbits (int val)
- unsigned fract urbits (unsigned int val)
- long fract lrbits (long val)
- unsigned long fract ulrbits (unsigned long val)
- long long fract llrbits (long long val)
- unsigned long long fract ullrbits (unsigned long long val)
- short accum hkbits (int val)
- unsigned short accum uhkbits (unsigned int val)
- accum kbits (long val)
- unsigned accum ukbits (unsigned long val)
- long accum lkbits (long long val)
- unsigned long accum ulkbits (unsigned long long val)
- long long accum llkbits (long long val)
- unsigned long long accum ullkbits (unsigned long long val)

Count Left-Shift

- int countlshr (short fract val)
- int countlsuhr (unsigned short fract val)
- int countlsr (fract val)
- int countlsur (unsigned fract val)
- int countlslr (long fract val)
- int countlsulr (unsigned long fract val)
- int countlsllr (long long fract val)
- int countlsullr (unsigned long long fract val)
- int countlshk (short accum val)
- int countlsuhk (unsigned short accum val)
- int countlsk (accum val)
- int countlsuk (unsigned accum val)
- int countlslk (long accum val)
- int countlsulk (unsigned long accum val)
- int countlsllk (long long accum val)
- int countlsullk (unsigned long long accum val)

Division

- fract rdivi (int num, int denom)
- long fract lrdivi (long int num, long int denom)
- unsigned fract urdivi (unsigned int num, unsigned int denom)
- unsigned long fract ulrdivi (unsigned long int num, unsigned long int denom)

Rounding

- short fract roundhr (short fract val, int bit)
- unsigned short fract rounduhr (unsigned short fract val, int bit)
- fract roundr (fract val, int bit)
- unsigned fract roundur (unsigned fract val, int bit)
- long fract roundlr (long fract val, int bit)
- unsigned long fract roundulr (unsigned long fract val, int bit)
- long long fract roundllr (long long fract val, int bit)
- unsigned long long fract roundullr (unsigned long long fract val, int bit)
- short accum roundhk (short accum val, int bit)
- unsigned short accum rounduhk (unsigned short accum val, int bit)
- accum roundk (accum val, int bit)
- unsigned accum rounduk (unsigned accum val, int bit)
- long accum roundlk (long accum val, int bit)
- unsigned long accum roundulk (unsigned long accum val, int bit)
- long long accum roundllk (long long accum val, int bit)
- unsigned long long accum roundullk (unsigned long long accum val, int bit)

Square Root and Transcendental Functions

- accum acosk (accum x)
- unsigned accum acosuk (unsigned accum x)
- accum asink (accum x)
- unsigned accum asinuk (unsigned accum x)
- accum atank (accum x)
- unsigned accum atanuk (unsigned accum x)
- unsigned fract atanur (unsigned fract x)
- accum exp2k (accum x)
- unsigned accum exp2uk (unsigned accum x)
- unsigned fract exp2m1ur (unsigned fract x)
- accum log2uk (unsigned accum x)
- short accum log2uhk (unsigned short accum x)
- unsigned short fract log21puhr (unsigned short fract x)
- unsigned fract log21pur (unsigned fract x)
- accum cospi2k (accum deg)
- accum sinpi2k (accum deg)
- fract sinuhk_deg (unsigned short accum deg)
- fract cosuhk_deg (unsigned short accum deg)
- unsigned fract sinpi2ur (unsigned fract x)
- short accum sqrthk (short accum radic)
- short fract sqrthr (short fract radic)
- accum sqrtk (accum radic)
- long fract sqrtlr (long fract radic)
- fract sqrtr (fract radic)
- unsigned short accum sqrtuhk (unsigned short accum radic)
- unsigned short fract sqrtuhr (unsigned short fract radic)
- unsigned accum sqrtuk (unsigned accum radic)
- unsigned long fract sqrtulr (unsigned long fract radic)
- unsigned fract sqrtur (unsigned fract radic)

Type-Generic Functions

- type absfx (type val)
- int countlsfx (type val)
- type roundfx (type val, int bit)

Functions reading from PROGMEM

- static short fract pgm_read_hr (const short fract *addr)
- static unsigned short fract pgm_read_uhr (const unsigned short fract *addr)
- static fract pgm_read_r (const fract *addr)
- static unsigned fract pgm_read_ur (const unsigned fract *addr)
- static long fract pgm_read_lr (const long fract *addr)
- static unsigned long fract pgm_read_ulr (const unsigned long fract *addr)
- static long long fract pgm_read_llr (const long long fract *addr)
- static unsigned long long fract pgm_read_ullr (const unsigned long long fract *addr)
- static short accum pgm_read_hk (const short accum *addr)
- static unsigned short accum pgm_read_uhk (const unsigned short accum *addr)
- static accum pgm_read_k (const accum *addr)
- static unsigned accum pgm_read_uk (const unsigned accum *addr)
- static long accum pgm_read_lk (const long accum *addr)
- static unsigned long accum pgm_read_ulk (const unsigned long accum *addr)
- static long long accum pgm_read_llk (const long long accum *addr)
- static unsigned long long accum pgm_read_ullk (const unsigned long long accum *addr)

Functions reading from PROGMEM_FAR

- static short fract pgm_read_hr_far (uint_farptr_t addr)
- static unsigned short fract pgm_read_uhr_far (uint_farptr_t addr)
- static fract pgm_read_r_far (uint_farptr_t addr)

- static unsigned fract pgm_read_ur_far (uint_farptr_t addr)
- static long fract pgm_read_lr_far (uint_farptr_t addr)
- static unsigned long fract pgm_read_ulr_far (uint_farptr_t addr)
- static long long fract pgm_read_llr_far (uint_farptr_t addr)
- static unsigned long long fract pgm_read_ullr_far (uint_farptr_t addr)
- static short accum pgm_read_hk_far (uint_farptr_t addr)
- static unsigned short accum pgm_read_uhk_far (uint_farptr_t addr)
- static accum pgm_read_k_far (uint_farptr_t addr)
- static unsigned accum pgm_read_uk_far (uint_farptr_t addr)
- static long accum pgm_read_lk_far (uint_farptr_t addr)
- static unsigned long accum pgm_read_ulk_far (uint_farptr_t addr)
- static long long accum pgm_read_llk_far (uint_farptr_t addr)
- static unsigned long long accum pgm_read_ullk_far (uint_farptr_t addr)

EEPROM Read Functions

- short fract eeprom_read_hr (const short fract *__p)
- unsigned short fract eeprom_read_uhr (const unsigned short fract *__p)
- fract eeprom_read_r (const fract *__p)
- unsigned fract eeprom_read_ur (const unsigned fract *__p)
- long fract eeprom_read_lr (const long fract *__p)
- unsigned long fract eeprom_read_ulr (const unsigned long fract *__p)
- long long fract eeprom_read_llr (const long long fract *__p)
- unsigned long long fract eeprom_read_ullr (const unsigned long long fract *__p)
- short accum eeprom_read_hk (const short accum *__p)
- unsigned short accum eeprom_read_uhk (const unsigned short accum *__p)
- accum eeprom_read_k (const accum *__p)
- unsigned accum eeprom_read_uk (const unsigned accum *__p)
- long accum eeprom_read_lk (const long accum *__p)
- unsigned long accum eeprom_read_ulk (const unsigned long accum *__p)
- long long accum eeprom_read_llk (const long long accum *__p)
- unsigned long long accum eeprom_read_ullk (const unsigned long long accum *__p)

EEPROM Write Functions

- void eeprom_write_hr (short fract *__p, short fract __value)
- void eeprom_write_uhr (unsigned short fract *__p, unsigned short fract __value)
- void eeprom_write_r (fract *__p, fract __value)
- void eeprom_write_ur (unsigned fract *__p, unsigned fract __value)
- void eeprom_write_lr (long fract *__p, long fract __value)
- void eeprom_write_ulr (unsigned long fract *__p, unsigned long fract __value)
- void eeprom_write_llr (long long fract *__p, long long fract __value)
- void eeprom_write_ullr (unsigned long long fract *__p, unsigned long long fract __value)
- void eeprom_write_hk (short accum *__p, short accum __value)
- void eeprom_write_uhk (unsigned short accum *__p, unsigned short accum __value)
- void eeprom_write_k (accum *__p, accum __value)
- void eeprom_write_uk (unsigned accum *__p, unsigned accum __value)
- void eeprom_write_lk (long accum *__p, long accum __value)
- void eeprom_write_ulk (unsigned long accum *__p, unsigned long accum __value)
- void eeprom_write_llk (long long accum *__p, long long accum __value)
- void eeprom_write_ullk (unsigned long long accum *__p, unsigned long long accum __value)

EEPROM Update Functions

- void eeprom_update_hr (short fract *__p, short fract __value)
- void eeprom_update_uhr (unsigned short fract *__p, unsigned short fract __value)
- void eeprom_update_r (fract *__p, fract __value)
- void eeprom_update_ur (unsigned fract *__p, unsigned fract __value)
- void eeprom_update_lr (long fract *__p, long fract __value)
- void eeprom_update_ulr (unsigned long fract *__p, unsigned long fract __value)
- void eeprom_update_llr (long long fract *__p, long long fract __value)
- void eeprom_update_ullr (unsigned long long fract *__p, unsigned long long fract __value)
- void eeprom_update_hk (short accum *__p, short accum __value)
- void eeprom_update_uhk (unsigned short accum *__p, unsigned short accum __value)
- void eeprom_update_k (accum *__p, accum __value)
- void eeprom_update_uk (unsigned accum *__p, unsigned accum __value)
- void eeprom_update_lk (long accum *__p, long accum __value)
- void eeprom_update_ulk (unsigned long accum *__p, unsigned long accum __value)
- void eeprom_update_llk (long long accum *__p, long long accum __value)
- void eeprom_update_ullk (unsigned long long accum *__p, unsigned long long accum __value)

22.2 stdfix-avrlibc.h

Go to the documentation of this file.

```
00001 /* Copyright (c) 2013 Joerg Wunsch
00002    All rights reserved.
00003
00004    Redistribution and use in source and binary forms, with or without
00005    modification, are permitted provided that the following conditions are met:
00006
00007    * Redistributions of source code must retain the above copyright
00008      notice, this list of conditions and the following disclaimer.
00009
00010    * Redistributions in binary form must reproduce the above copyright
00011      notice, this list of conditions and the following disclaimer in
00012      the documentation and/or other materials provided with the
00013      distribution.
00014
00015    * Neither the name of the copyright holders nor the names of
00016      contributors may be used to endorse or promote products derived
00017      from this software without specific prior written permission.
00018
00019   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
00020   AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
00021   IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
00022   ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
00023   LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
00024   CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
00025   SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
00026   INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
00027   CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
00028   ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
00029   POSSIBILITY OF SUCH DAMAGE. */
00030
00031 #ifndef _STDFIX_AVRLIBC_H
00032 #define _STDFIX_AVRLIBC_H
00033
00034 /*
00035  * AVR-LibC addendum file for Embedded C Fixed-Point support
00036  *
00037  * See: ISO/IEC TR 18037
00038  * http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1169.pdf
00039  */
00040
00041 #ifndef _AVRGCC_STDFIX_H  /* Defined in stdfix.h from avr-gcc */
00042 #warning please include <stdfix.h> directly rather than <stdfix-avrlibc.h>
00043 #endif /* _AVRGCC_STDFIX_H */
00044
00045 #include <bits/attribs.h>
00046
00047 /** \file */
00048 /** \anchor stdfix_h
00049     \defgroup avr_stdfix <stdfix.h>: ISO/IEC TR 18037 Fixed-Point Arithmetic
00050 \if STDFIX_AVRLIBC_H_NOT_INCLUDED
00051     \code
00052         #include <stdfix.h>
00053         #include <stdfix-avrlibc.h>
00054     \endcode
00055 \else
00056     \code
00057         #include <stdfix.h>
00058     \endcode
00059 \endif
00060
00061 As an extension, GNU C supports fixed-point types as defined in the
00062 N1169 draft of ISO/IEC DTR 18037.
00063
00064 \since <a href="https://gcc.gnu.org/gcc-4.8/changes.html#avr"
00065         >avr-gcc v4.8</a>
00066
00067 Two groups of fixed-point data types are added:
00068 - The <em>fract types</em> and the <em>accum types</em>.
00069   The data value of a \e fract type has no integral part,
```

```
00070   hence values of a \e fract type are between -1.0 and +1.0.
00071 - The value range of an \e accum type depends on the number of integral bits
00072   in the data type.
00073
00074 <table>
00075 <caption>Fixed-Point Type Layout</caption>
00076 <tr><th>Const Suffix <th>Type <th>Size <th>Q-Format <th> Epsilon
00077 <tr><td align="right"><tt>hr</tt>  <td align="right"><tt>short fract</tt> <td>1<td>s.7
      <td>7.81&middot;10<sup>-3</sup>
00078 <tr><td align="right"><tt>r</tt>   <td align="right"><tt>fract</tt>       <td>2<td>s.15
      <td>3.05&middot;10<sup>-5</sup>
00079 <tr><td align="right"><tt>lr</tt>  <td align="right"><tt>long fract</tt>  <td>4<td>s.31
      <td>4.66&middot;10<sup>-10</sup>
00080 <tr><td align="right"><tt>llr</tt> <td align="right"><tt>long long fract</tt>
      <td>8<td>s.63<td>1.08&middot;10<sup>-19</sup>
00081 <tr><td align="right"><tt>hk</tt>  <td align="right"><tt>short accum</tt> <td>2<td>s8.7
      <td>7.81&middot;10<sup>-3</sup>
00082 <tr><td align="right"><tt>k</tt>   <td align="right"><tt>accum</tt>
      <td>4<td>s16.15  <td>3.05&middot;10<sup>-5</sup>
00083 <tr><td align="right"><tt>lk</tt>  <td align="right"><tt>long accum</tt>
      <td>8<td>s32.31  <td>4.66&middot;10<sup>-10</sup>
00084 <tr><td align="right"><tt>llk</tt> <td align="right"><tt>long long accum</tt>
      <td>8<td>s16.47    <td>7.11&middot;10<sup>-15</sup>
00085 <tr><td align="right"><tt>uhr</tt> <td align="right"><tt>unsigned short fract</tt>
      <td>1<td>0.8  <td>3.91&middot;10<sup>-3</sup>
00086 <tr><td align="right"><tt>ur</tt>  <td align="right"><tt>unsigned fract</tt>
      <td>2<td>0.16 <td>1.53&middot;10<sup>-5</sup>
00087 <tr><td align="right"><tt>ulr</tt> <td align="right"><tt>unsigned long fract</tt>
      <td>4<td>0.32 <td>2.33&middot;10<sup>-10</sup>
00088 <tr><td align="right"><tt>ullr</tt><td align="right"><tt>unsigned long long
      fract</tt><td>8<td>0.64 <td>5.42&middot;10<sup>-20</sup>
00089 <tr><td align="right"><tt>uhk</tt> <td align="right"><tt>unsigned short accum</tt>
      <td>2<td>8.8  <td>3.91&middot;10<sup>-3</sup>
00090 <tr><td align="right"><tt>uk</tt>  <td align="right"><tt>unsigned accum</tt>
      <td>4<td>16.16<td>1.53&middot;10<sup>-5</sup>
00091 <tr><td align="right"><tt>ulk</tt> <td align="right"><tt>unsigned long accum</tt>
      <td>8<td>32.32<td>2.33&middot;10<sup>-10</sup>
00092 <tr><td align="right"><tt>ullk</tt><td align="right"><tt>unsigned long long accum</tt>
      <td>8<td>16.48<td>3.55&middot;10<sup>-15</sup>
00093 </table>
00094 \remark
00095   - Upper case constant suffixes are also supported.
00096   - The \c long \c long fixed-point types are avr-gcc extensions.
00097
00098 See also some \ref bench_fxlib "benchmarks".
00099 */
00100
00101
00102 /* Room for AVR-LibC specific extensions */
00103
00104 /* 7.18a.6.1 The fixed-point arithmetic operation support functions */
00105
00106 #ifdef __cplusplus
00107 extern "C" {
00108 #endif
00109
00110 /** \ingroup avr_stdfix
00111     Include all significant digits in the result of a
00112     fixed-point to decimal ASCII conversion.
00113     The result has no trailing zeros.
00114
00115     To be used in the \a mode parameter of such a conversion.
00116     For details and examples, see uktoa().
00117     \since AVR-LibC v2.3  */
00118 #define FXTOA_ALL   0x1f
00119
00120 /** \ingroup avr_stdfix
00121     A flag to select rounding to nearest in a fixed-point to
00122     decimal ASCII conversion. Rounding mode is the default,
00123     i.e. FXTOA_ROUND can be omitted.
00124
00125     To be used in the \a mode parameter of such a conversion.
00126     For details and examples, see uktoa().
```

```
00127     \since AVR-LibC v2.3  */
00128 #define FXTOA_ROUND 0x00
00129
00130 /** \ingroup avr_stdfix
00131     A flag to select truncation (rounding to zero) in a fixed-point to
00132     decimal ASCII conversion.
00133
00134     To be used in the \a mode parameter of such a conversion.
00135     For details and examples, see uktoa().
00136     \since AVR-LibC v2.3  */
00137 #define FXTOA_TRUNC 0x80
00138
00139 /** \ingroup avr_stdfix
00140     A flag to select that the result of a fixed-point to
00141     decimal ASCII conversion has no trailing zeros.
00142
00143     To be used in the \a mode parameter of such a conversion.
00144     For details and examples, see uktoa().
00145     \since AVR-LibC v2.3  */
00146 #define FXTOA_NTZ 0x40
00147
00148 /** \ingroup avr_stdfix
00149     The fixed-point to decimal ASCII conversion routines use a
00150     dot (<tt>.</tt>) for the decimal point.  This is the default, i.e.
00151     FXTOA_DOT can be omitted.
00152
00153     For details, and examples, see uktoa().
00154     \since AVR-LibC v2.3  */
00155 #define FXTOA_DOT 0x00
00156
00157 /** \ingroup avr_stdfix
00158     The fixed-point to decimal ASCII conversion routines use a
00159     comma (<tt>,</tt>) for the decimal point.
00160
00161     For details and examples, see uktoa().
00162     \since AVR-LibC v2.3  */
00163 #define FXTOA_COMMA 0x20
00164
00165
00166 #ifdef __DOXYGEN__
00167
00168 /** \name ASCII Conversions (not in ISO/IEC TR18037) */
00169
00170 /** \ingroup avr_stdfix
00171     Convert fixed-point value \p x to a decimal ASCII representation.
00172     The result is written to \p buf, and the user is responsible for
00173     providing enough memory in \p buf.  Returns \p buf.
00174
00175     For the meaning of \p mode, and for the (maximal) number of
00176     character written by this function, see uktoa().
00177     \since AVR-LibC v2.3 */
00178 char* hktoa (short accum x, char *buf, unsigned char mode);
00179
00180 /** \ingroup avr_stdfix
00181     Convert fixed-point value \p x to a decimal ASCII representation.
00182     The result is written to \p buf, and the user is responsible for
00183     providing enough memory in \p buf.  Returns \p buf.
00184
00185     For the meaning of \p mode, and for the (maximal) number of
00186     character written by this function, see uktoa().
00187     \since AVR-LibC v2.3 */
00188 char* hrtoa (short fract x, char *buf, unsigned char mode);
00189
00190 /** \ingroup avr_stdfix
00191     Convert fixed-point value \p x to a decimal ASCII representation.
00192     The result is written to \p buf, and the user is responsible for
00193     providing enough memory in \p buf.  Returns \p buf.
00194
00195     For the meaning of \p mode, and for the (maximal) number of
00196     character written by this function, see uktoa().
00197     \since AVR-LibC v2.3 */
00198 char* ktoa (accum x, char *buf, unsigned char mode);
00199
```

```
00200 /** \ingroup avr_stdfix
00201     Convert fixed-point value \p x to a decimal ASCII representation.
00202     The result is written to \p buf, and the user is responsible for
00203     providing enough memory in \p buf.  Returns \p buf.
00204
00205     For the meaning of \p mode, and for the (maximal) number of
00206     character written by this function, see uktoa().
00207     \since AVR-LibC v2.3 */
00208 char* rtoa (fract x, char *buf, unsigned char mode);
00209
00210 /** \ingroup avr_stdfix
00211     Convert fixed-point value \p x to a decimal ASCII representation.
00212     The result is written to \p buf, and the user is responsible for
00213     providing enough memory in \p buf.  Returns \p buf.
00214
00215     For the meaning of \p mode, and for the (maximal) number of
00216     character written by this function, see uktoa().
00217     \since AVR-LibC v2.3 */
00218 char* uhktoa (unsigned short accum x, char *buf, unsigned char mode);
00219
00220 /** \ingroup avr_stdfix
00221     Convert fixed-point value \p x to a decimal ASCII representation.
00222     The result is written to \p buf, and the user is responsible for
00223     providing enough memory in \p buf.  Returns \p buf.
00224
00225     For the meaning of \p mode, and for the (maximal) number of
00226     character written by this function, see uktoa().
00227     \since AVR-LibC v2.3 */
00228 char* uhrtoa (unsigned short fract x, char *buf, unsigned char mode);
00229
00230 /** \ingroup avr_stdfix
00231     Convert fixed-point value \p x to a decimal ASCII representation.
00232     The result is written to \p buf, and the user is responsible for
00233     providing enough memory in \p buf.  Returns \p buf.
00234
00235     The format of the output is controlled by the \a mode parameter.
00236     It is composed from the format flags below together with \a Digs,
00237     the number of fractional digits.
00238     The \p mode is the ORed result from \a Digs and a combination of
00239     #FXTOA_ALL, #FXTOA_ROUND or #FXTOA_TRUNC, #FXTOA_DOT or #FXTOA_COMMA,
00240     and #FXTOA_NTZ, like in:
00241     \code
00242         uktoa (x, buf, FXTOA_ROUND | FXTOA_NTZ | Digs);
00243     \endcode
00244
00245     - Supported values for \a Digs are in the range 0...30.
00246
00247 <dl>
00248   <dt>#FXTOA_ALL
00249   <dd>Include all significant digits. \a Digs will be ignored.
00250
00251   <dt>#FXTOA_ROUND
00252   <dd>Round to nearest for \a Digs fractional digits.
00253     Rounding for \a Digs &ge; 5 has no effect, i.e. for such \a Digs values
00254     the rounded result will be the same like the truncated result.
00255     Rounding mode is the default, i.e. FXTOA_ROUND can be omitted.
00256
00257   <dt>#FXTOA_TRUNC
00258   <dd>Like FXTOA_ALL, but truncate the result (round to zero)
00259       after \a Digs fractional digits.
00260
00261   <dt>#FXTOA_NTZ
00262   <dd>
00263     &bull; With FXTOA_NTZ the result has
00264       <b>n</b>o <b>t</b>railing <b>z</b>eros.
00265       When the result represents an integral value,
00266       then no decimal point and no fractional digits are present.<br>
00267     &bull; Without FXTOA_NTZ and with FXTOA_ROUND or FXTOA_TRUNC,
00268       the result has the specified number of fractional digits.<br>
00269     &bull; FXTOA_NTZ has no effect with FXTOA_ALL.<br>
00270     &bull; FXTOA_NTZ has no effect on the required buffer size.
00271   <dt>#FXTOA_DOT
00272   <dd>The decimal point is a dot (<tt>.</tt>).
```

```
00273       This is the default, i.e. FXTOA_DOT can be omitted.
00274
00275   <dt>#FXTOA_COMMA
00276   <dd>The decimal point is a comma (<tt>,</tt>).
00277 </dl>
00278
00279 <table>
00280 <caption>Examples</caption>
00281 <tr>
00282   <th>Value
00283   <th>Mode
00284   <th>Result
00285 </tr>
00286 <tr><td>1.8uk<td><tt>0</tt><td><tt>"2"</tt></tr>
00287 <tr><td>1.8uk<td><tt>1</tt><td><tt>"1.8"</tt></tr>
00288 <tr><td>1.8uk<td><tt>2</tt><td><tt>"1.80"</tt></tr>
00289 <tr><td>1.8uk<td><tt>3</tt><td><tt>"1.800"</tt></tr>
00290 <tr><td>1.8uk<td><tt>FXTOA_NTZ | 0</tt><td><tt>"2"</tt></tr>
00291 <tr><td>1.8uk<td><tt>FXTOA_NTZ | 1</tt><td><tt>"1.8"</tt></tr>
00292 <tr><td>1.8uk<td><tt>FXTOA_NTZ | 2</tt><td><tt>"1.8"</tt></tr>
00293 <tr><td>1.8uk<td><tt>FXTOA_NTZ | 3</tt><td><tt>"1.8"</tt></tr>
00294 <tr><td>1.8uk<td><tt>FXTOA_TRUNC | FXTOA_COMMA | 0</tt><td><tt>"1"</tt></tr>
00295 <tr><td>1.8uk<td><tt>FXTOA_TRUNC | FXTOA_COMMA | 1</tt><td><tt>"1,7"</tt></tr>
00296 <tr><td>1.8uk<td><tt>FXTOA_TRUNC | FXTOA_COMMA | 2</tt><td><tt>"1,79"</tt></tr>
00297 <tr><td>1.8uk<td><tt>FXTOA_TRUNC | FXTOA_COMMA | 3</tt><td><tt>"1,799"</tt></tr>
00298 <tr><td>1.8uk<td><tt>FXTOA_ALL</tt> <td><tt>"1.79998779296875"</tt></tr>
00299 </table>
00300
00301 The following table helps with providing enough memory in \a buf.
00302
00303 Notice that the required size of \a buf is independent of #FXTOA_NTZ.
00304 This is the case since with #FXTOA_NTZ the required buffer size may
00305 be larger that the size of the returned string.
00306
00307 <table>
00308   <caption>Maximum Number of Bytes written to buf</caption>
00309   <tr>
00310     <th>Type
00311     <th>Function
00312     <th>FXTOA_ALL
00313     <th>All other Modes
00314     <th colspan="2">Exact Maximal Value
00315   </tr>
00316   <tr>
00317     <td align="right"><tt>unsigned accum</tt>
00318     <td align="right">#uktoa
00319     <td align="center">23
00320     <td align="center">7 + \a Digs
00321     <td align="center">2<sup>16</sup>-2<sup>-16</sup>
00322     <td>65535.9999847412109375
00323   <tr>
00324     <td align="right"><tt>accum</tt>
00325     <td align="right">#ktoa
00326     <td align="center">23
00327     <td align="center">8 + \a Digs
00328     <td align="center">2<sup>16</sup>-2<sup>-15</sup>
00329     <td>65535.999969482421875
00330   <tr>
00331     <td align="right"><tt>unsigned short accum</tt>
00332     <td align="right">#uhktoa
00333     <td align="center">13
00334     <td align="center">5 + \a Digs
00335     <td align="center">2<sup>8</sup>-2<sup>-8</sup>
00336     <td>255.99609375
00337   <tr>
00338     <td align="right"><tt>short accum</tt>
00339     <td align="right">#hktoa
00340     <td align="center">13
00341     <td align="center">6 + \a Digs
00342     <td align="center">2<sup>8</sup>-2<sup>-7</sup>
00343     <td>255.9921875
00344   <tr>
00345     <td align="right"><tt>unsigned fract</tt>
```

```
00346     <td align="right">#urtoa
00347     <td align="center">19
00348     <td align="center">3 + \a Digs
00349     <td align="center">1-2<sup>-16</sup>
00350     <td>0.9999847412109375
00351   <tr>
00352     <td align="right"><tt>fract</tt>
00353     <td align="right">#rtoa
00354     <td align="center">19
00355     <td align="center">4 + \a Digs
00356     <td align="center">1-2<sup>-15</sup>
00357     <td>0.999969482421875
00358   <tr>
00359     <td align="right"><tt>unsigned short fract</tt>
00360     <td align="right">#uhrtoa
00361     <td align="center">11
00362     <td align="center">3 + \a Digs
00363     <td align="center">1-2<sup>-8</sup>
00364     <td>0.99609375
00365   <tr>
00366     <td align="right"><tt>short fract</tt>
00367     <td align="right">#hrtoa
00368     <td align="center">11
00369     <td align="center">4 + \a Digs
00370     <td align="center">1-2<sup>-7</sup>
00371     <td>0.9921875
00372   </tr>
00373 </table>
00374
00375     \since AVR-LibC v2.3 */
00376 char* uktoa (unsigned accum x, char *buf, unsigned char mode);
00377
00378 /** \ingroup avr_stdfix
00379     Convert fixed-point value \p x to a decimal ASCII representation.
00380     The result is written to \p buf, and the user is responsible for
00381     providing enough memory in \p buf.  Returns \p buf.
00382
00383     For the meaning of \p mode, and for the (maximal) number of
00384     character written by this function, see uktoa().
00385     \since AVR-LibC v2.3 */
00386 char* urtoa (unsigned fract x, char *buf, unsigned char mode);
00387
00388
00389 /** \name Absolute Value */
00390
00391 /** \ingroup avr_stdfix
00392     Computes the absolute value of \p val.  When the result does not
00393     fit into the range of the return type, the result is saturated.  */
00394 short fract abshr (short fract val);
00395
00396 /** \ingroup avr_stdfix
00397     Computes the absolute value of \p val.  When the result does not
00398     fit into the range of the return type, the result is saturated.  */
00399 fract absr (fract val);
00400
00401 /** \ingroup avr_stdfix
00402     Computes the absolute value of \p val.  When the result does not
00403     fit into the range of the return type, the result is saturated.  */
00404 long fract abslr (long fract val);
00405
00406 /** \ingroup avr_stdfix
00407     Computes the absolute value of \p val.  When the result does not
00408     fit into the range of the return type, the result is saturated.  */
00409 long long fract absllr (long long fract val);
00410
00411 /** \ingroup avr_stdfix
00412     Computes the absolute value of \p val.  When the result does not
00413     fit into the range of the return type, the result is saturated.  */
00414 short accum abshk (short accum val);
00415
00416 /** \ingroup avr_stdfix
00417     Computes the absolute value of \p val.  When the result does not
00418     fit into the range of the return type, the result is saturated.  */
```

```
00419 accum absk (accum val);
00420
00421 /** \ingroup avr_stdfix
00422     Computes the absolute value of \p val.  When the result does not
00423     fit into the range of the return type, the result is saturated.  */
00424 long accum abslk (long accum val);
00425
00426 /** \ingroup avr_stdfix
00427     Computes the absolute value of \p val.  When the result does not
00428     fit into the range of the return type, the result is saturated.  */
00429 long long accum absllk (long long accum val);
00430
00431
00432 /** \name Bit-Conversions to Integer */
00433
00434 /** \ingroup avr_stdfix
00435     Return an integer value of the same size and signedness,
00436     and with the same bit representation like \p val.  */
00437 signed char bitshr (short fract val);
00438
00439 /** \ingroup avr_stdfix
00440     Return an integer value of the same size and signedness,
00441     and with the same bit representation like \p val.  */
00442 unsigned char bitsuhr (unsigned short fract val);
00443
00444 /** \ingroup avr_stdfix
00445     Return an integer value of the same size and signedness,
00446     and with the same bit representation like \p val.  */
00447 int bitsr (fract val);
00448
00449 /** \ingroup avr_stdfix
00450     Return an integer value of the same size and signedness,
00451     and with the same bit representation like \p val.  */
00452 unsigned int bitsur (unsigned fract val);
00453
00454 /** \ingroup avr_stdfix
00455     Return an integer value of the same size and signedness,
00456     and with the same bit representation like \p val.  */
00457 long bitslr (long fract val);
00458
00459 /** \ingroup avr_stdfix
00460     Return an integer value of the same size and signedness,
00461     and with the same bit representation like \p val.  */
00462 unsigned long bitsulr (unsigned long fract val);
00463
00464 /** \ingroup avr_stdfix
00465     Return an integer value of the same size and signedness,
00466     and with the same bit representation like \p val.  */
00467 long long bitsllr (long long fract val);
00468
00469 /** \ingroup avr_stdfix
00470     Return an integer value of the same size and signedness,
00471     and with the same bit representation like \p val.  */
00472 unsigned long long bitsullr (unsigned long long fract val);
00473
00474 /** \ingroup avr_stdfix
00475     Return an integer value of the same size and signedness,
00476     and with the same bit representation like \p val.  */
00477 int bitshk (short accum val);
00478
00479 /** \ingroup avr_stdfix
00480     Return an integer value of the same size and signedness,
00481     and with the same bit representation like \p val.  */
00482 unsigned int bitsuhk (unsigned short accum val);
00483
00484 /** \ingroup avr_stdfix
00485     Return an integer value of the same size and signedness,
00486     and with the same bit representation like \p val.  */
00487 long bitsk (accum val);
00488
00489 /** \ingroup avr_stdfix
00490     Return an integer value of the same size and signedness,
00491     and with the same bit representation like \p val.  */
```

```
00492 unsigned long bitsuk (unsigned accum val);
00493
00494 /** \ingroup avr_stdfix
00495     Return an integer value of the same size and signedness,
00496     and with the same bit representation like \p val.  */
00497 long long bitslk (long accum val);
00498
00499 /** \ingroup avr_stdfix
00500     Return an integer value of the same size and signedness,
00501     and with the same bit representation like \p val.  */
00502 unsigned long long bitsulk (unsigned long accum val);
00503
00504 /** \ingroup avr_stdfix
00505     Return an integer value of the same size and signedness,
00506     and with the same bit representation like \p val.  */
00507 long long bitsllk (long long accum val);
00508
00509 /** \ingroup avr_stdfix
00510     Return an integer value of the same size and signedness,
00511     and with the same bit representation like \p val.  */
00512 unsigned long long bitsullk (unsigned long long accum val);
00513
00514
00515 /** \name Bit-Conversions to Fixed-Point */
00516
00517 /** \ingroup avr_stdfix
00518     Return a fixed-point value of the same size and signedness,
00519     and with the same bit representation like \p val.  */
00520 short fract hrbits (signed char val);
00521
00522 /** \ingroup avr_stdfix
00523     Return a fixed-point value of the same size and signedness,
00524     and with the same bit representation like \p val.  */
00525 unsigned short fract uhrbits (unsigned char val);
00526
00527 /** \ingroup avr_stdfix
00528     Return a fixed-point value of the same size and signedness,
00529     and with the same bit representation like \p val.  */
00530 fract rbits (int val);
00531
00532 /** \ingroup avr_stdfix
00533     Return a fixed-point value of the same size and signedness,
00534     and with the same bit representation like \p val.  */
00535 unsigned fract urbits (unsigned int val);
00536
00537 /** \ingroup avr_stdfix
00538     Return a fixed-point value of the same size and signedness,
00539     and with the same bit representation like \p val.  */
00540 long fract lrbits (long val);
00541
00542 /** \ingroup avr_stdfix
00543     Return a fixed-point value of the same size and signedness,
00544     and with the same bit representation like \p val.  */
00545 unsigned long fract ulrbits (unsigned long val);
00546
00547 /** \ingroup avr_stdfix
00548     Return a fixed-point value of the same size and signedness,
00549     and with the same bit representation like \p val.  */
00550 long long fract llrbits (long long val);
00551
00552 /** \ingroup avr_stdfix
00553     Return a fixed-point value of the same size and signedness,
00554     and with the same bit representation like \p val.  */
00555 unsigned long long fract ullrbits (unsigned long long val);
00556
00557 /** \ingroup avr_stdfix
00558     Return a fixed-point value of the same size and signedness,
00559     and with the same bit representation like \p val.  */
00560 short accum hkbits (int val);
00561
00562 /** \ingroup avr_stdfix
00563     Return a fixed-point value of the same size and signedness,
00564     and with the same bit representation like \p val.  */
```

```
00565 unsigned short accum uhkbits (unsigned int val);
00566
00567 /** \ingroup avr_stdfix
00568     Return a fixed-point value of the same size and signedness,
00569     and with the same bit representation like \p val.  */
00570 accum kbits (long val);
00571
00572 /** \ingroup avr_stdfix
00573     Return a fixed-point value of the same size and signedness,
00574     and with the same bit representation like \p val.  */
00575 unsigned accum ukbits (unsigned long val);
00576
00577 /** \ingroup avr_stdfix
00578     Return a fixed-point value of the same size and signedness,
00579     and with the same bit representation like \p val.  */
00580 long accum lkbits (long long val);
00581
00582 /** \ingroup avr_stdfix
00583     Return a fixed-point value of the same size and signedness,
00584     and with the same bit representation like \p val.  */
00585 unsigned long accum ulkbits (unsigned long long val);
00586
00587 /** \ingroup avr_stdfix
00588     Return a fixed-point value of the same size and signedness,
00589     and with the same bit representation like \p val.  */
00590 long long accum llkbits (long long val);
00591
00592 /** \ingroup avr_stdfix
00593     Return a fixed-point value of the same size and signedness,
00594     and with the same bit representation like \p val.  */
00595 unsigned long long accum ullkbits (unsigned long long val);
00596
00597 /** \name Count Left-Shift */
00598
00599 /** \ingroup avr_stdfix
00600     - If \p val is non-zero, the return value is the largest integer
00601     \c k for which the expression <tt>val &lt;&lt; k</tt> does not overflow.
00602     - If \p val is zero, an integer value is returned that is at least
00603     as large as <tt>N - 1</tt>, where <tt>N</tt> is the total number
00604     of bits of the type of the argument. */
00605 int countlshr (short fract val);
00606
00607 /** \ingroup avr_stdfix
00608     - If \p val is non-zero, the return value is the largest integer
00609     \c k for which the expression <tt>val &lt;&lt; k</tt> does not overflow.
00610     - If \p val is zero, an integer value is returned that is at least
00611     as large as <tt>N - 1</tt>, where <tt>N</tt> is the total number
00612     of bits of the type of the argument. */
00613 int countlsuhr (unsigned short fract val);
00614
00615 /** \ingroup avr_stdfix
00616     - If \p val is non-zero, the return value is the largest integer
00617     \c k for which the expression <tt>val &lt;&lt; k</tt> does not overflow.
00618     - If \p val is zero, an integer value is returned that is at least
00619     as large as <tt>N - 1</tt>, where <tt>N</tt> is the total number
00620     of bits of the type of the argument. */
00621 int countlsr (fract val);
00622
00623 /** \ingroup avr_stdfix
00624     - If \p val is non-zero, the return value is the largest integer
00625     \c k for which the expression <tt>val &lt;&lt; k</tt> does not overflow.
00626     - If \p val is zero, an integer value is returned that is at least
00627     as large as <tt>N - 1</tt>, where <tt>N</tt> is the total number
00628     of bits of the type of the argument. */
00629 int countlsur (unsigned fract val);
00630
00631 /** \ingroup avr_stdfix
00632     - If \p val is non-zero, the return value is the largest integer
00633     \c k for which the expression <tt>val &lt;&lt; k</tt> does not overflow.
00634     - If \p val is zero, an integer value is returned that is at least
00635     as large as <tt>N - 1</tt>, where <tt>N</tt> is the total number
00636     of bits of the type of the argument. */
00637 int countlslr (long fract val);
```

```
00638
00639 /** \ingroup avr_stdfix
00640     - If \p val is non-zero, the return value is the largest integer
00641     \c k for which the expression <tt>val &lt;&lt; k</tt> does not overflow.
00642     - If \p val is zero, an integer value is returned that is at least
00643     as large as <tt>N - 1</tt>, where <tt>N</tt> is the total number
00644     of bits of the type of the argument. */
00645 int countlsulr (unsigned long fract val);
00646
00647 /** \ingroup avr_stdfix
00648     - If \p val is non-zero, the return value is the largest integer
00649     \c k for which the expression <tt>val &lt;&lt; k</tt> does not overflow.
00650     - If \p val is zero, an integer value is returned that is at least
00651     as large as <tt>N - 1</tt>, where <tt>N</tt> is the total number
00652     of bits of the type of the argument. */
00653 int countlsllr (long long fract val);
00654
00655 /** \ingroup avr_stdfix
00656     - If \p val is non-zero, the return value is the largest integer
00657     \c k for which the expression <tt>val &lt;&lt; k</tt> does not overflow.
00658     - If \p val is zero, an integer value is returned that is at least
00659     as large as <tt>N - 1</tt>, where <tt>N</tt> is the total number
00660     of bits of the type of the argument. */
00661 int countlsullr (unsigned long long fract val);
00662
00663 /** \ingroup avr_stdfix
00664     - If \p val is non-zero, the return value is the largest integer
00665     \c k for which the expression <tt>val &lt;&lt; k</tt> does not overflow.
00666     - If \p val is zero, an integer value is returned that is at least
00667     as large as <tt>N - 1</tt>, where <tt>N</tt> is the total number
00668     of bits of the type of the argument. */
00669 int countlshk (short accum val);
00670
00671 /** \ingroup avr_stdfix
00672     - If \p val is non-zero, the return value is the largest integer
00673     \c k for which the expression <tt>val &lt;&lt; k</tt> does not overflow.
00674     - If \p val is zero, an integer value is returned that is at least
00675     as large as <tt>N - 1</tt>, where <tt>N</tt> is the total number
00676     of bits of the type of the argument. */
00677 int countlsuhk (unsigned short accum val);
00678
00679 /** \ingroup avr_stdfix
00680     - If \p val is non-zero, the return value is the largest integer
00681     \c k for which the expression <tt>val &lt;&lt; k</tt> does not overflow.
00682     - If \p val is zero, an integer value is returned that is at least
00683     as large as <tt>N - 1</tt>, where <tt>N</tt> is the total number
00684     of bits of the type of the argument. */
00685 int countlsk (accum val);
00686
00687 /** \ingroup avr_stdfix
00688     - If \p val is non-zero, the return value is the largest integer
00689     \c k for which the expression <tt>val &lt;&lt; k</tt> does not overflow.
00690     - If \p val is zero, an integer value is returned that is at least
00691     as large as <tt>N - 1</tt>, where <tt>N</tt> is the total number
00692     of bits of the type of the argument. */
00693 int countlsuk (unsigned accum val);
00694
00695 /** \ingroup avr_stdfix
00696     - If \p val is non-zero, the return value is the largest integer
00697     \c k for which the expression <tt>val &lt;&lt; k</tt> does not overflow.
00698     - If \p val is zero, an integer value is returned that is at least
00699     as large as <tt>N - 1</tt>, where <tt>N</tt> is the total number
00700     of bits of the type of the argument. */
00701 int countlslk (long accum val);
00702
00703 /** \ingroup avr_stdfix
00704     - If \p val is non-zero, the return value is the largest integer
00705     \c k for which the expression <tt>val &lt;&lt; k</tt> does not overflow.
00706     - If \p val is zero, an integer value is returned that is at least
00707     as large as <tt>N - 1</tt>, where <tt>N</tt> is the total number
00708     of bits of the type of the argument. */
00709 int countlsulk (unsigned long accum val);
00710
```

```
00711 /** \ingroup avr_stdfix
00712     - If \p val is non-zero, the return value is the largest integer
00713     \c k for which the expression <tt>val &lt;&lt; k</tt> does not overflow.
00714     - If \p val is zero, an integer value is returned that is at least
00715     as large as <tt>N - 1</tt>, where <tt>N</tt> is the total number
00716     of bits of the type of the argument. */
00717 int countlsllk (long long accum val);
00718
00719 /** \ingroup avr_stdfix
00720     - If \p val is non-zero, the return value is the largest integer
00721     \c k for which the expression <tt>val &lt;&lt; k</tt> does not overflow.
00722     - If \p val is zero, an integer value is returned that is at least
00723     as large as <tt>N - 1</tt>, where <tt>N</tt> is the total number
00724     of bits of the type of the argument. */
00725 int countlsullk (unsigned long long accum val);
00726
00727 #endif /* Doxygen */
00728
00729 /*
00730  * The functions below compute the result of the multiplication or division
00731  * operation on the operands with the indicated types, and return a value with
00732  * the indicated type.  The return value is rounded towards zero, and is
00733  * saturated on overflow.  If the second operand of one of the divide
00734  * functions is zero, the behavior is undefined.
00735  */
00736
00737 /** \name Division */
00738
00739 /** \ingroup avr_stdfix
00740     The rdivi() function computes the value \c num/denom and returns the
00741     result of the \c fract type.
00742     The return value is rounded towards zero, and is saturated on overflow.
00743     If \c denom is zero, the behavior is undefined.
00744     \since AVR-LibC v2.3
00745 */
00746 __ATTR_CONST__
00747 extern fract rdivi (int num, int denom);
00748
00749 /** \ingroup avr_stdfix
00750     The lrdivi() function computes the value of \c num/denom and returns the
00751     result of the long fract type.
00752     The return value is rounded towards zero, and is saturated on overflow.
00753     If \c denom is zero, the behavior is undefined.
00754     \since AVR-LibC v2.3
00755 */
00756 __ATTR_CONST__
00757 extern long fract lrdivi (long int num, long int denom);
00758
00759 /** \ingroup avr_stdfix
00760     The urdivi() function computes the value \c num/denom and returns the
00761     result of the \c unsigned \c fract type.
00762     The return value is rounded towards zero, and is saturated on overflow.
00763     If \c denom is zero, the behavior is undefined.
00764     \since AVR-LibC v2.3
00765 */
00766 __ATTR_CONST__
00767 extern unsigned fract urdivi (unsigned int num, unsigned int denom);
00768
00769 /** \ingroup avr_stdfix
00770     The ulrdivi() function computes the value of \c num/denom and returns the
00771     result of the <tt>unsigned long fract</tt> type.
00772     The return value is rounded towards zero, and is saturated on overflow.
00773     If \c denom is zero, the behavior is undefined.
00774     \since AVR-LibC v2.3
00775 */
00776 __ATTR_CONST__
00777 extern unsigned long fract
00778 ulrdivi (unsigned long int num, unsigned long int denom);
00779
00780
00781 /* AVR-LibC extensions for mathematical functions on fixed-point numbers. */
00782
00783 /* Fixed-point square roots using integer arithmetics.
```

```
00784  * See "Fast Integer Square Root" by Ross M. Fosler, Microchip DS91040 (2000).
00785  */
00786
00787 #ifdef __DOXYGEN__
00788 /** \name Rounding */
00789
00790 /** \ingroup avr_stdfix
00791     Round \p val to \p bit fractional bits.  When the result does not
00792     fit into the range of the return type, the result is saturated.  */
00793 short fract roundhr (short fract val, int bit);
00794
00795 /** \ingroup avr_stdfix
00796     Round \p val to \p bit fractional bits.  When the result does not
00797     fit into the range of the return type, the result is saturated.  */
00798 unsigned short fract rounduhr (unsigned short fract val, int bit);
00799
00800 /** \ingroup avr_stdfix
00801     Round \p val to \p bit fractional bits.  When the result does not
00802     fit into the range of the return type, the result is saturated.  */
00803 fract roundr (fract val, int bit);
00804
00805 /** \ingroup avr_stdfix
00806     Round \p val to \p bit fractional bits.  When the result does not
00807     fit into the range of the return type, the result is saturated.  */
00808 unsigned fract roundur (unsigned fract val, int bit);
00809
00810 /** \ingroup avr_stdfix
00811     Round \p val to \p bit fractional bits.  When the result does not
00812     fit into the range of the return type, the result is saturated.  */
00813 long fract roundlr (long fract val, int bit);
00814
00815 /** \ingroup avr_stdfix
00816     Round \p val to \p bit fractional bits.  When the result does not
00817     fit into the range of the return type, the result is saturated.  */
00818 unsigned long fract roundulr (unsigned long fract val, int bit);
00819
00820 /** \ingroup avr_stdfix
00821     Round \p val to \p bit fractional bits.  When the result does not
00822     fit into the range of the return type, the result is saturated.  */
00823 long long fract roundllr (long long fract val, int bit);
00824
00825 /** \ingroup avr_stdfix
00826     Round \p val to \p bit fractional bits.  When the result does not
00827     fit into the range of the return type, the result is saturated.  */
00828 unsigned long long fract roundullr (unsigned long long fract val, int bit);
00829
00830 /** \ingroup avr_stdfix
00831     Round \p val to \p bit fractional bits.  When the result does not
00832     fit into the range of the return type, the result is saturated.
00833
00834     As an extension, \p bit may be in the range
00835     <tt>-IBIT &lt; bit &lt; FBIT</tt>.
00836     For example, <tt>bit = -1</tt> rounds to an even value.  */
00837 short accum roundhk (short accum val, int bit);
00838
00839 /** \ingroup avr_stdfix
00840     Round \p val to \p bit fractional bits.  When the result does not
00841     fit into the range of the return type, the result is saturated.
00842
00843     As an extension, \p bit may be in the range
00844     <tt>-IBIT &lt; bit &lt; FBIT</tt>.
00845     For example, <tt>bit = -1</tt> rounds to an even value.  */
00846 unsigned short accum rounduhk (unsigned short accum val, int bit);
00847
00848 /** \ingroup avr_stdfix
00849     Round \p val to \p bit fractional bits.  When the result does not
00850     fit into the range of the return type, the result is saturated.
00851
00852     As an extension, \p bit may be in the range
00853     <tt>-IBIT &lt; bit &lt; FBIT</tt>.
00854     For example, <tt>bit = -1</tt> rounds to an even value.  */
00855 accum roundk (accum val, int bit);
00856
```

```
00857 /** \ingroup avr_stdfix
00858     Round \p val to \p bit fractional bits.  When the result does not
00859     fit into the range of the return type, the result is saturated.
00860
00861     As an extension, \p bit may be in the range
00862     <tt>-IBIT &lt; bit &lt; FBIT</tt>.
00863     For example, <tt>bit = -1</tt> rounds to an even value.  */
00864 unsigned accum rounduk (unsigned accum val, int bit);
00865
00866 /** \ingroup avr_stdfix
00867     Round \p val to \p bit fractional bits.  When the result does not
00868     fit into the range of the return type, the result is saturated.
00869
00870     As an extension, \p bit may be in the range
00871     <tt>-IBIT &lt; bit &lt; FBIT</tt>.
00872     For example, <tt>bit = -1</tt> rounds to an even value.  */
00873 long accum roundlk (long accum val, int bit);
00874
00875 /** \ingroup avr_stdfix
00876     Round \p val to \p bit fractional bits.  When the result does not
00877     fit into the range of the return type, the result is saturated.
00878
00879     As an extension, \p bit may be in the range
00880     <tt>-IBIT &lt; bit &lt; FBIT</tt>.
00881     For example, <tt>bit = -1</tt> rounds to an even value.  */
00882 unsigned long accum roundulk (unsigned long accum val, int bit);
00883
00884 /** \ingroup avr_stdfix
00885     Round \p val to \p bit fractional bits.  When the result does not
00886     fit into the range of the return type, the result is saturated.
00887
00888     As an extension, \p bit may be in the range
00889     <tt>-IBIT &lt; bit &lt; FBIT</tt>.
00890     For example, <tt>bit = -1</tt> rounds to an even value.  */
00891 long long accum roundllk (long long accum val, int bit);
00892
00893 /** \ingroup avr_stdfix
00894     Round \p val to \p bit fractional bits.  When the result does not
00895     fit into the range of the return type, the result is saturated.
00896
00897     As an extension, \p bit may be in the range
00898     <tt>-IBIT &lt; bit &lt; FBIT</tt>.
00899     For example, <tt>bit = -1</tt> rounds to an even value.  */
00900 unsigned long long accum roundullk (unsigned long long accum val, int bit);
00901
00902
00903 /** \name Square Root and Transcendental Functions */
00904
00905 /** \ingroup avr_stdfix
00906     Compute the arcus cosine of \p x.
00907     The returned value is in the range
00908 \htmlonly
00909 [0, &pi;].
00910 \endhtmlonly
00911 \latexonly
00912 \begin{math} [0, \pi ]\end{math} .
00913 \endlatexonly
00914 \manonly
00915 .EQ
00916 [0, pi ]
00917 .EN
00918 .
00919 \endmanonly
00920     For invalid values of \p x the returned value
00921     is -65536 = #kbits (0x80000000).
00922
00923     The absolute error is bounded by
00924 \htmlonly
00925 5.5&middot;10<sup>&minus;5</sup> &asymp; 2<sup>&minus;14.1</sup>.
00926 \endhtmlonly
00927 \latexonly
00928 \begin{math} 5.5\cdot 10^{-5} \approx  2^{-14.1}\end{math} .
00929 \endlatexonly
```

```
00930 \manonly
00931 .EQ
00932 5.5 * 10^{-5} = 2^{-14.1}
00933 .EN
00934 .
00935 \endmanonly
00936
00937     \since AVR-LibC v2.3  */
00938 accum acosk (accum x);
00939
00940 /** \ingroup avr_stdfix
00941     Compute the arcus cosine of \p x.
00942     The returned value is in the range
00943 \htmlonly
00944 [0, &pi;/2].
00945 \endhtmlonly
00946 \latexonly
00947 \begin{math} [0, \pi /2]\end{math} .
00948 \endlatexonly
00949 \manonly
00950 .EQ
00951 [0, pi/2]
00952 .EN
00953 .
00954 \endmanonly
00955     For invalid values of \p x the returned value
00956     is 32768 = #ukbits (0x80000000).
00957
00958     The absolute error is bounded by
00959 \htmlonly
00960 4.6&middot;10<sup>&minus;5</sup> &asymp; 2<sup>&minus;14.4</sup>.
00961 \endhtmlonly
00962 \latexonly
00963 \begin{math} 4.6\cdot 10^{-5} \approx  2^{-14.4}\end{math} .
00964 \endlatexonly
00965 \manonly
00966 .EQ
00967 4.6 * 10^{-5} = 2^{-14.4}
00968 .EN
00969 .
00970 \endmanonly
00971
00972     \since AVR-LibC v2.3  */
00973 unsigned accum acosuk (unsigned accum x);
00974
00975 /** \ingroup avr_stdfix
00976     Compute the arcus sine of \p x.
00977     The returned value is in the range
00978 \htmlonly
00979 [&minus;&pi;/2, &pi;/2].
00980 \endhtmlonly
00981 \latexonly
00982 \begin{math} [-\pi /2, \pi /2]\end{math} .
00983 \endlatexonly
00984 \manonly
00985 .EQ
00986 [- pi/2, pi/2]
00987 .EN
00988 .
00989 \endmanonly
00990     For invalid values of \p x the returned value
00991     is -65536 = #kbits (0x80000000).
00992
00993     The absolute error is bounded by
00994 \htmlonly
00995 5.1&middot;10<sup>&minus;5</sup> &asymp; 2<sup>&minus;14</sup>.
00996 \endhtmlonly
00997 \latexonly
00998 \begin{math} 5.1\cdot 10^{-5} \approx  2^{-14}\end{math} .
00999 \endlatexonly
01000 \manonly
01001 .EQ
01002 5.1 * 10^{-5} = 2^{-14}
```

```
01003 .EN
01004 .
01005 \endmanonly
01006
01007     \since AVR-LibC v2.3  */
01008 accum asink (accum x);
01009
01010 /** \ingroup avr_stdfix
01011     Compute the arcus sine of \p x.
01012     The returned value is in the range
01013 \htmlonly
01014 [0, &pi;/2].
01015 \endhtmlonly
01016 \latexonly
01017 \begin{math} [0, \pi /2]\end{math} .
01018 \endlatexonly
01019 \manonly
01020 .EQ
01021 [0, pi/2]
01022 .EN
01023 .
01024 \endmanonly
01025     For invalid values of \p x the returned value
01026     is 32768 = #ukbits (0x80000000).
01027
01028     The absolute error is bounded by
01029 \htmlonly
01030 4.5&middot;10<sup>&minus;5</sup> &asymp; 2<sup>&minus;14.4</sup>.
01031 \endhtmlonly
01032 \latexonly
01033 \begin{math} 4.5\cdot 10^{-5} \approx  2^{-14.4}\end{math} .
01034 \endlatexonly
01035 \manonly
01036 .EQ
01037 4.5 * 10^{-5} = 2^{-14.4}
01038 .EN
01039 .
01040 \endmanonly
01041
01042     \since AVR-LibC v2.3  */
01043 unsigned accum asinuk (unsigned accum x);
01044
01045 /** \ingroup avr_stdfix
01046     Compute the arcus tangent of \p x.
01047     The returned value is in the range
01048 \htmlonly
01049 (&minus;&pi;/2, &pi;/2&asymp;1.5708).
01050 \endhtmlonly
01051 \latexonly
01052 \begin{math} (-\pi /2, \pi /2\approx 1.5708)\end{math} .
01053 \endlatexonly
01054 \manonly
01055 .EQ
01056 (- pi/2, pi/2=1.5708)
01057 .EN
01058 .
01059 \endmanonly
01060
01061     \since AVR-LibC v2.3  */
01062 accum atank (accum x);
01063
01064 /** \ingroup avr_stdfix
01065     Compute the arcus tangent of \p x.
01066     The returned value is in the range
01067 \htmlonly
01068 [0, &pi;/2&asymp;1.5708).
01069 \endhtmlonly
01070 \latexonly
01071 \begin{math} [0, \pi /2\approx 1.5708)\end{math} .
01072 \endlatexonly
01073 \manonly
01074 .EQ
01075 [0, pi/2=1.5708)
```

```
01076 .EN
01077 .
01078 \endmanonly
01079
01080     \since AVR-LibC v2.3  */
01081 unsigned accum atanuk (unsigned accum x);
01082
01083 /** \ingroup avr_stdfix
01084     Compute the arcus tangent of \p x.
01085     The returned value is in the range
01086 \htmlonly
01087 [0, &pi;/4&asymp;0.7854].
01088 \endhtmlonly
01089 \latexonly
01090 \begin{math} [0, \pi /4\approx 0.7854]\end{math} .
01091 \endlatexonly
01092 \manonly
01093 .EQ
01094 [0, pi/4=0.7854]
01095 .EN
01096 .
01097 \endmanonly
01098
01099     The absolute error is bounded by
01100 \htmlonly
01101 2.6&middot;10<sup>&minus;5</sup> &asymp; 2<sup>&minus;15</sup>.
01102 \endhtmlonly
01103 \latexonly
01104 \begin{math} 2.6\cdot 10^{-5} \approx  2^{-15}\end{math} .
01105 \endlatexonly
01106 \manonly
01107 .EQ
01108 2.6 * 10^{-5} = 2^{-15}
01109 .EN
01110 .
01111 \endmanonly
01112     The worst case execution time (WCET) is around 210 cycles when MUL is
01113     available, and around  1000 cycles when MUL is not available
01114     (measured with avr-gcc v15).
01115     \since AVR-LibC v2.3  */
01116 unsigned fract atanur (unsigned fract x);
01117
01118 /** \ingroup avr_stdfix
01119     Compute
01120 \htmlonly
01121 2<sup>x</sup>
01122 \endhtmlonly
01123 \latexonly
01124 \begin{math} 2^{x}\end{math}
01125 \endlatexonly
01126 \manonly
01127 .EQ
01128 2^x
01129 .EN
01130 \endmanonly
01131     with saturation.
01132
01133     The WCET is at least the one of exp2m1ur().
01134     \since AVR-LibC v2.3  */
01135 accum exp2k (accum x);
01136
01137 /** \ingroup avr_stdfix
01138     Compute
01139 \htmlonly
01140 2<sup>x</sup>
01141 \endhtmlonly
01142 \latexonly
01143 \begin{math} 2^{x}\end{math}
01144 \endlatexonly
01145 \manonly
01146 .EQ
01147 2^x
01148 .EN
```

```
01149 \endmanonly
01150     with saturation.
01151
01152     The WCET is at least the one of exp2m1ur().
01153     \since AVR-LibC v2.3  */
01154 unsigned accum exp2uk (unsigned accum x);
01155
01156 /** \ingroup avr_stdfix
01157     Compute
01158 \htmlonly
01159 2<sup>x</sup> &minus; 1.
01160 \endhtmlonly
01161 \latexonly
01162 \begin{math} 2^{x} - 1\end{math} .
01163 \endlatexonly
01164 \manonly
01165 .EQ
01166 2^x - 1
01167 .EN
01168 .
01169 \endmanonly
01170     The returned value is in the range [0, 1).
01171
01172     The absolute error is bounded by
01173 \htmlonly
01174 2.2&middot;10<sup>&minus;5</sup> &asymp; 2<sup>&minus;15.4</sup>.
01175 \endhtmlonly
01176 \latexonly
01177 \begin{math} 2.2\cdot 10^{-5} \approx  2^{-15.4}\end{math} .
01178 \endlatexonly
01179 \manonly
01180 .EQ
01181 2.2 * 10^{-5} = 2^{-15.4}
01182 .EN
01183 .
01184 \endmanonly
01185     The worst case execution time (WCET) is around 200 cycles when MUL is
01186     available, and around  1000 cycles when MUL is not available
01187     (measured with avr-gcc v15).
01188     \since AVR-LibC v2.3  */
01189 unsigned fract exp2m1ur (unsigned fract x);
01190
01191 /** \ingroup avr_stdfix
01192     Returns
01193 \htmlonly
01194 log<sub>2</sub>(x),
01195 \endhtmlonly
01196 \latexonly
01197 \begin{math} \log _{2}(x)\end{math} ,
01198 \endlatexonly
01199 \manonly
01200 .EQ
01201 log_2(x)
01202 .EN
01203 ,
01204 \endmanonly
01205     the logarithm to base 2 of the value \p x.
01206     The returned value for \p x = 0 is -32768.
01207
01208     The absolute error is bounded by
01209 \htmlonly
01210 4.5&middot;10<sup>&minus;5</sup> &asymp; 2<sup>&minus;14.5</sup>.
01211 \endhtmlonly
01212 \latexonly
01213 \begin{math} 4.5\cdot 10^{-5} \approx  2^{-14.5}\end{math} .
01214 \endlatexonly
01215 \manonly
01216 .EQ
01217 4.5 * 10^{-5} = 2^{-14.5}
01218 .EN
01219 .
01220 \endmanonly
01221     The worst case execution time (WCET) is around 150 cycles more
```

```
01222     than the WCET of log21pur().
01223     \since AVR-LibC v2.3  */
01224 accum log2uk(unsigned accum x);
01225
01226 /** \ingroup avr_stdfix
01227     Return
01228 \htmlonly
01229 log<sub>2</sub>(x),
01230 \endhtmlonly
01231 \latexonly
01232 \begin{math} \log _{2}(x)\end{math} ,
01233 \endlatexonly
01234 \manonly
01235 .EQ
01236 log_2(x)
01237 .EN
01238 ,
01239 \endmanonly
01240     the logarithm to base 2 of the value \p x.
01241     The returned value for \p x = 0 is -128.
01242
01243     The absolute error is bounded by
01244 \htmlonly
01245 8&middot;10<sup>&minus;3</sup> &asymp; 2<sup>&minus;7</sup>.
01246 \endhtmlonly
01247 \latexonly
01248 \begin{math} 8\cdot 10^{-3} \approx  2^{-7}\end{math} .
01249 \endlatexonly
01250 \manonly
01251 .EQ
01252 8 * 10^{-3} = 2^{-7}
01253 .EN
01254 .
01255 \endmanonly
01256     The worst case execution time (WCET) is around 60 cycles plus
01257     the WCET of log21puhr().
01258     \since AVR-LibC v2.3  */
01259 short accum log2uhk(unsigned short accum x);
01260
01261 /** \ingroup avr_stdfix
01262     Return
01263 \htmlonly
01264 log<sub>2</sub>(1 + x),
01265 \endhtmlonly
01266 \latexonly
01267 \begin{math} \log _{2}(1 + x)\end{math} ,
01268 \endlatexonly
01269 \manonly
01270 .EQ
01271 log_2(1 + x)
01272 .EN
01273 ,
01274 \endmanonly
01275     the logarithm to base 2 of the value 1 + \p x.
01276     The result is in the range [0, 1).
01277
01278     The absolute error is bounded by
01279 \htmlonly
01280 4.3&middot;10<sup>&minus;3</sup> &asymp; 2<sup>&minus;7.5</sup>.
01281 \endhtmlonly
01282 \latexonly
01283 \begin{math} 4.3\cdot 10^{-3} \approx  2^{-7.5}\end{math} .
01284 \endlatexonly
01285 \manonly
01286 .EQ
01287 4.3 * 10^{-3} = 2^{-7.5}
01288 .EN
01289 .
01290 \endmanonly
01291     The worst case execution time (WCET) is around 25 cycles when MUL is
01292     available, and around 340 cycles when MUL is not available.
01293     \since AVR-LibC v2.3  */
01294 unsigned short fract log21puhr(unsigned short fract x);
```

```
01295
01296 /** \ingroup avr_stdfix
01297     Return
01298 \htmlonly
01299 log<sub>2</sub>(1 + x),
01300 \endhtmlonly
01301 \latexonly
01302 \begin{math} \log _{2}(1 + x)\end{math} ,
01303 \endlatexonly
01304 \manonly
01305 .EQ
01306 log_2(1 + x)
01307 .EN
01308 ,
01309 \endmanonly
01310     the logarithm to base 2 of the value 1 + \p x.
01311     The result is in the range [0, 1).
01312
01313     The absolute error is bounded by
01314 \htmlonly
01315 3&middot;10<sup>&minus;5</sup> &asymp; 2<sup>&minus;15</sup>.
01316 \endhtmlonly
01317 \latexonly
01318 \begin{math} 3\cdot 10^{-5} \approx  2^{-15}\end{math} .
01319 \endlatexonly
01320 \manonly
01321 .EQ
01322 3 * 10^{-5} = 2^{-15}
01323 .EN
01324 .
01325 \endmanonly
01326     The worst case execution time (WCET) is around 250 cycles when MUL is
01327     available, and around 1300 cycles when MUL is not available.
01328     \since AVR-LibC v2.3  */
01329 unsigned fract log21pur(unsigned fract x);
01330
01331 /** \ingroup avr_stdfix
01332     Cosine of
01333 \htmlonly
01334 x&middot;&pi;/2
01335 \endhtmlonly
01336 \latexonly
01337 \begin{math} x\cdot \pi /2\end{math}
01338 \endlatexonly
01339 \manonly
01340 .EQ
01341 x * pi/2
01342 .EN
01343 \endmanonly
01344     radians.
01345
01346     The absolute error is bounded by
01347 \htmlonly
01348 4.6&middot;10<sup>&minus;5</sup> &asymp; 2<sup>&minus;14.4</sup>.
01349 \endhtmlonly
01350 \latexonly
01351 \begin{math} 4.6\cdot 10^{-5} \approx  2^{-14.4}\end{math} .
01352 \endlatexonly
01353 \manonly
01354 .EQ
01355 4.6 * 10^{-5} = 2^{-14.4}
01356 .EN
01357 .
01358 \endmanonly
01359     The worst case execution time (WCET) is around 300 cycles when MUL is
01360     available, and around 1400 cycles when MUL is not available.
01361     \since AVR-LibC v2.3  */
01362 accum cospi2k(accum deg);
01363
01364 /** \ingroup avr_stdfix
01365     Sine of
01366 \htmlonly
01367 x&middot;&pi;/2
```

```
01368 \endhtmlonly
01369 \latexonly
01370 \begin{math} x\cdot \pi /2\end{math}
01371 \endlatexonly
01372 \manonly
01373 .EQ
01374 x * pi/2
01375 .EN
01376 \endmanonly
01377     radians.
01378
01379     The absolute error is bounded by
01380 \htmlonly
01381 4.6&middot;10<sup>&minus;5</sup> &asymp; 2<sup>&minus;14.4</sup>.
01382 \endhtmlonly
01383 \latexonly
01384 \begin{math} 4.6\cdot 10^{-5} \approx  2^{-14.4}\end{math} .
01385 \endlatexonly
01386 \manonly
01387 .EQ
01388 4.6 * 10^{-5} = 2^{-14.4}
01389 .EN
01390 .
01391 \endmanonly
01392     The worst case execution time (WCET) is around 300 cycles when MUL is
01393     available, and around 1400 cycles when MUL is not available.
01394     \since AVR-LibC v2.3  */
01395 accum sinpi2k(accum deg);
01396
01397 /** \ingroup avr_stdfix
01398     Sine of the angle \a deg where \a deg is specified in degrees, i.e.
01399     in the range
01400 \htmlonly
01401 [0&deg;, 256&deg;).
01402 \endhtmlonly
01403 \latexonly
01404 \begin{math} [0^\circ , 256^\circ )\end{math} .
01405 \endlatexonly
01406 \manonly
01407 .EQ
01408 [0 degree, 256 degree)
01409 .EN
01410 .
01411 \endmanonly
01412     The returned value is in the range (-1, +1), i.e. is never -1.
01413
01414     The absolute error is bounded by
01415 \htmlonly
01416 6.5&middot;10<sup>&minus;5</sup> &asymp; 2<sup>&minus;13.9</sup>.
01417 \endhtmlonly
01418 \latexonly
01419 \begin{math} 6.5\cdot 10^{-5} \approx  2^{-13.9}\end{math} .
01420 \endlatexonly
01421 \manonly
01422 .EQ
01423 6.5 * 10^{-5} = 2^{-13.9}
01424 .EN
01425 .
01426 \endmanonly
01427     The worst case execution time (WCET) is around 70 cycles.
01428     \since AVR-LibC v2.3  */
01429 fract sinuhk_deg(unsigned short accum deg);
01430
01431 /** \ingroup avr_stdfix
01432     Cosine of the angle \a deg where \a deg is specified in degrees, i.e.
01433     in the range
01434 \htmlonly
01435 [0&deg;, 256&deg;).
01436 \endhtmlonly
01437 \latexonly
01438 \begin{math} [0^\circ , 256^\circ )\end{math} .
01439 \endlatexonly
01440 \manonly
```

```
01441 .EQ
01442 [0 degree, 256 degree)
01443 .EN
01444 .
01445 \endmanonly
01446     The returned value is in the range (-1, +1), i.e. is never -1.
01447
01448     The absolute error is bounded by
01449 \htmlonly
01450 6.5&middot;10<sup>&minus;5</sup> &asymp; 2<sup>&minus;13.9</sup>.
01451 \endhtmlonly
01452 \latexonly
01453 \begin{math} 6.5\cdot 10^{-5} \approx  2^{-13.9}\end{math} .
01454 \endlatexonly
01455 \manonly
01456 .EQ
01457 6.5 * 10^{-5} = 2^{-13.9}
01458 .EN
01459 .
01460 \endmanonly
01461     The worst case execution time (WCET) is around 90 cycles.
01462     \since AVR-LibC v2.3  */
01463 fract cosuhk_deg(unsigned short accum deg);
01464
01465 /** \ingroup avr_stdfix
01466     Sine of
01467 \htmlonly
01468 x&middot;&pi;/2
01469 \endhtmlonly
01470 \latexonly
01471 \begin{math} x\cdot \pi /2\end{math}
01472 \endlatexonly
01473 \manonly
01474 .EQ
01475 x * pi/2
01476 .EN
01477 \endmanonly
01478     redians.
01479     The returned value is in the range [0, 1).
01480
01481     The absolute error is bounded by
01482 \htmlonly
01483 2.9&middot;10<sup>&minus;5</sup> &asymp; 2<sup>&minus;15</sup>.
01484 \endhtmlonly
01485 \latexonly
01486 \begin{math} 2.9\cdot 10^{-5} \approx  2^{-15}\end{math} .
01487 \endlatexonly
01488 \manonly
01489 .EQ
01490 2.9 * 10^{-5} = 2^{-15}
01491 .EN
01492 .
01493 \endmanonly
01494     The worst case execution time (WCET) is around 260 cycles when MUL is
01495     available, and around 1400 cycles when MUL is not available.
01496     \since AVR-LibC v2.3  */
01497 unsigned fract sinpi2ur(unsigned fract x);
01498
01499 /** \ingroup avr_stdfix
01500     Square root of the value \p radic.  Negative values are returned unchanged.
01501     The worst case execution time (WCET) is around 310 cycles.
01502     \since AVR-LibC v2.3  */
01503 short accum sqrthk(short accum radic);
01504
01505 /** \ingroup avr_stdfix
01506     Square root of the value \p radic rounded down.
01507     \since AVR-LibC v2.3  */
01508 short fract sqrthr(short fract radic);
01509
01510 /** \ingroup avr_stdfix
01511     Square root of the value \p radic.  Negative values are returned unchanged.
01512     The worst case execution time (WCET) is around 640 cycles.
01513     \since AVR-LibC v2.3  */
```

```
01514 accum sqrtk(accum radic);
01515
01516 /** \ingroup avr_stdfix
01517     Square root of the value \p radic.  Negative values are returned unchanged.
01518     The worst case execution time (WCET) is around 1080 cycles.
01519     \since AVR-LibC v2.3  */
01520 long fract sqrtlr(long fract radic);
01521
01522 /** \ingroup avr_stdfix
01523     Square root of the value \p radic.  Negative values are returned unchanged.
01524     The worst case execution time (WCET) is around 320 cycles.
01525     \since AVR-LibC v2.3  */
01526 fract sqrtr(fract radic);
01527
01528 /** \ingroup avr_stdfix
01529     Square root of the value \p radic rounded down.
01530     The worst case execution time (WCET) is around 300 cycles.
01531     \since AVR-LibC v2.3  */
01532 unsigned short accum sqrtuhk(unsigned short accum radic);
01533
01534 /** \ingroup avr_stdfix
01535     Square root of the value \p radic rounded down.
01536     The result is in the range [0, 1).
01537
01538     The absolute error is in the range
01539 \htmlonly
01540 [&minus;3.9&middot;10<sup>&minus;3</sup> &asymp; 2<sup>&minus;8</sup>, 0].
01541 \endhtmlonly
01542 \latexonly
01543 \begin{math} [-3.9\cdot 10^{-3} \approx  2^{-8}, 0]\end{math} .
01544 \endlatexonly
01545 \manonly
01546 .EQ
01547 [-3.9 * 10^{-3} = 2^{-8}, 0]
01548 .EN
01549 .
01550 \endmanonly
01551     The worst case execution time (WCET) is around 120 cycles.
01552     \since AVR-LibC v2.3  */
01553 unsigned short fract sqrtuhr(unsigned short fract radic);
01554
01555 /** \ingroup avr_stdfix
01556     Square root of the value \p radic.
01557     The worst case execution time (WCET) is around 620 cycles.
01558     \since AVR-LibC v2.3  */
01559 unsigned accum sqrtuk(unsigned accum radic);
01560
01561
01562 /** \ingroup avr_stdfix
01563     Square root of the value \p radic. The result is in the range [0, 1).
01564     The worst case execution time (WCET) is around 1060 cycles.
01565     \since AVR-LibC v2.3  */
01566 unsigned long fract sqrtulr(unsigned long fract radic);
01567
01568 /** \ingroup avr_stdfix
01569     Square root of the value \p radic.  The result is in the range [0, 1).
01570
01571     The absolute error is in the range
01572 \htmlonly
01573 [&minus;1.5&middot;10<sup>&minus;5</sup> &asymp; 2<sup>&minus;16</sup>, 0].
01574 \endhtmlonly
01575 \latexonly
01576 \begin{math} [-1.5\cdot 10^{-5} \approx  2^{-16}, 0]\end{math} .
01577 \endlatexonly
01578 \manonly
01579 .EQ
01580 [-1.5 * 10^{-5} = 2^{-16}, 0]
01581 .EN
01582 .
01583 \endmanonly
01584     The worst case execution time (WCET) is around 320 cycles.
01585     \since AVR-LibC v2.3  */
01586 unsigned fract sqrtur(unsigned fract radic);
```

```
01587
01588
01589 /** \name Type-Generic Functions */
01590
01591 /** \ingroup avr_stdfix
01592     Computes the absolute value of fixed-point value \p val.
01593     When the result does not fit into the range of the return type,
01594     the result is saturated.  */
01595 type absfx (type val);
01596
01597 /** \ingroup avr_stdfix
01598     - If \p val is non-zero, the return value is the largest integer
01599     \c k for which the expression <tt>val &lt;&lt; k</tt> does not overflow.
01600     - If \p val is zero, an integer value is returned that is at least
01601     as large as <tt>N - 1</tt>, where <tt>N</tt> is the total number
01602     of bits of the type of the argument. */
01603 int countlsfx (type val);
01604
01605 /** \ingroup avr_stdfix
01606     Round \p val to \p bit fractional bits.  When the result does not
01607     fit into the range of the return type, the result is saturated.
01608
01609     As an extension, \p bit may be in the range
01610     <tt>-IBIT &lt; bit &lt; FBIT</tt>.
01611     For example, <tt>bit = -1</tt> rounds to an even value.  */
01612 type roundfx (type val, int bit);
01613
01614 #else /* Doxygen */
01615 extern unsigned accum acosuk(unsigned accum) __ATTR_CONST__;
01616 extern unsigned accum asinuk(unsigned accum) __ATTR_CONST__;
01617 extern unsigned accum atanuk(unsigned accum) __ATTR_CONST__;
01618 extern accum acosk(accum) __ATTR_CONST__;
01619 extern accum asink(accum) __ATTR_CONST__;
01620 extern accum atank(accum) __ATTR_CONST__;
01621 extern unsigned fract atanur(unsigned fract) __ATTR_CONST__;
01622
01623 extern short fract sqrthr(short fract) __asm__("__sqrthr") __ATTR_CONST__;
01624 extern unsigned short fract sqrtuhr(unsigned short fract) __asm__("__sqrtuhr")
     __ATTR_CONST__;
01625 fract sqrtr(fract) __ATTR_CONST__;
01626 unsigned fract sqrtur(unsigned fract) __ATTR_CONST__;
01627 long fract sqrtlr(long fract) __ATTR_CONST__;
01628 unsigned long fract sqrtulr(unsigned long fract) __ATTR_CONST__;
01629 unsigned accum sqrtuk(unsigned accum) __ATTR_CONST__;
01630 accum sqrtk(accum) __ATTR_CONST__;
01631 short accum sqrthk(short accum) __ATTR_CONST__;
01632 unsigned short accum sqrtuhk(unsigned short accum) __ATTR_CONST__;
01633
01634 extern unsigned short fract log21puhr(unsigned short fract) __ATTR_CONST__;
01635 extern unsigned fract log21pur(unsigned fract) __ATTR_CONST__;
01636 extern accum log2uk(unsigned accum) __ATTR_CONST__;
01637 extern short accum log2uhk(unsigned short accum) __ATTR_CONST__;
01638
01639 extern fract sinuhk_deg(unsigned short accum) __ATTR_CONST__;
01640 extern fract cosuhk_deg(unsigned short accum) __ATTR_CONST__;
01641 extern unsigned fract sinpi2ur(unsigned fract) __ATTR_CONST__;
01642 extern accum sinpi2k(accum) __ATTR_CONST__;
01643 extern accum cospi2k(accum) __ATTR_CONST__;
01644
01645 extern accum exp2k (accum) __ATTR_CONST__;
01646 extern unsigned accum exp2uk (unsigned accum) __ATTR_CONST__;
01647 extern unsigned fract exp2m1ur (unsigned fract) __ATTR_CONST__;
01648
01649 extern char* uktoa (unsigned accum, char*, unsigned char);
01650 extern char* urtoa (unsigned fract, char*, unsigned char);
01651 extern char* ktoa (accum, char*, unsigned char);
01652 extern char* rtoa (fract, char*, unsigned char);
01653 extern char* uhktoa (unsigned short accum, char*, unsigned char);
01654 extern char* uhrtoa (unsigned short fract, char*, unsigned char);
01655 extern char* hktoa (short accum, char*, unsigned char);
01656 extern char* hrtoa (short fract, char*, unsigned char);
01657 #endif /* Doxygen */
01658
```

```
01659 #ifdef __DOXYGEN__
01660 /** \name Functions reading from PROGMEM */
01661
01662 /** \ingroup avr_stdfix
01663     Read a <tt>short fract</tt> from 16-bit address \p addr.
01664     The address is in the lower 64 KiB of program memory.
01665     \since AVR-LibC v2.3 */
01666 static inline short fract pgm_read_hr (const short fract *addr);
01667
01668 /** \ingroup avr_stdfix
01669     Read an <tt>unsigned short fract</tt> from 16-bit address \p addr.
01670     The address is in the lower 64 KiB of program memory.
01671     \since AVR-LibC v2.3 */
01672 static inline unsigned short fract pgm_read_uhr (const unsigned short fract *addr);
01673
01674 /** \ingroup avr_stdfix
01675     Read a <tt>fract</tt> from 16-bit address \p addr.
01676     The address is in the lower 64 KiB of program memory.
01677     \since AVR-LibC v2.3 */
01678 static inline fract pgm_read_r (const fract *addr);
01679
01680 /** \ingroup avr_stdfix
01681     Read an <tt>unsigned fract</tt> from 16-bit address \p addr.
01682     The address is in the lower 64 KiB of program memory.
01683     \since AVR-LibC v2.3 */
01684 static inline unsigned fract pgm_read_ur (const unsigned fract *addr);
01685
01686 /** \ingroup avr_stdfix
01687     Read a <tt>long fract</tt> from 16-bit address \p addr.
01688     The address is in the lower 64 KiB of program memory.
01689     \since AVR-LibC v2.3 */
01690 static inline long fract pgm_read_lr (const long fract *addr);
01691
01692 /** \ingroup avr_stdfix
01693     Read an <tt>unsigned long fract</tt> from 16-bit address \p addr.
01694     The address is in the lower 64 KiB of program memory.
01695     \since AVR-LibC v2.3 */
01696 static inline unsigned long fract pgm_read_ulr (const unsigned long fract *addr);
01697
01698 /** \ingroup avr_stdfix
01699     Read a <tt>long long fract</tt> from 16-bit address \p addr.
01700     The address is in the lower 64 KiB of program memory.
01701     \since AVR-LibC v2.3 */
01702 static inline long long fract pgm_read_llr (const long long fract *addr);
01703
01704 /** \ingroup avr_stdfix
01705     Read an <tt>unsigned long long fract</tt> from 16-bit address \p addr.
01706     The address is in the lower 64 KiB of program memory.
01707     \since AVR-LibC v2.3 */
01708 static inline unsigned long long fract pgm_read_ullr (const unsigned long long fract
      *addr);
01709
01710 /** \ingroup avr_stdfix
01711     Read a <tt>short accum</tt> from 16-bit address \p addr.
01712     The address is in the lower 64 KiB of program memory.
01713     \since AVR-LibC v2.3 */
01714 static inline short accum pgm_read_hk (const short accum *addr);
01715
01716 /** \ingroup avr_stdfix
01717     Read an <tt>unsigned short accum</tt> from 16-bit address \p addr.
01718     The address is in the lower 64 KiB of program memory.
01719     \since AVR-LibC v2.3 */
01720 static inline unsigned short accum pgm_read_uhk (const unsigned short accum *addr);
01721
01722 /** \ingroup avr_stdfix
01723     Read an <tt>accum</tt> from 16-bit address \p addr.
01724     The address is in the lower 64 KiB of program memory.
01725     \since AVR-LibC v2.3 */
01726 static inline accum pgm_read_k (const accum *addr);
01727
01728 /** \ingroup avr_stdfix
01729     Read an <tt>unsigned accum</tt> from 16-bit address \p addr.
01730     The address is in the lower 64 KiB of program memory.
```

```
01731     \since AVR-LibC v2.3 */
01732 static inline unsigned accum pgm_read_uk (const unsigned accum *addr);
01733
01734 /** \ingroup avr_stdfix
01735     Read a <tt>long accum</tt> from 16-bit address \p addr.
01736     The address is in the lower 64 KiB of program memory.
01737     \since AVR-LibC v2.3 */
01738 static inline long accum pgm_read_lk (const long accum *addr);
01739
01740 /** \ingroup avr_stdfix
01741     Read an <tt>unsigned long accum</tt> from 16-bit address \p addr.
01742     The address is in the lower 64 KiB of program memory.
01743     \since AVR-LibC v2.3 */
01744 static inline unsigned long accum pgm_read_ulk (const unsigned long accum *addr);
01745
01746 /** \ingroup avr_stdfix
01747     Read a <tt>long long accum</tt> from 16-bit address \p addr.
01748     The address is in the lower 64 KiB of program memory.
01749     \since AVR-LibC v2.3 */
01750 static inline long long accum pgm_read_llk (const long long accum *addr);
01751
01752 /** \ingroup avr_stdfix
01753     Read an <tt>unsigned long long accum</tt> from 16-bit address \p addr.
01754     The address is in the lower 64 KiB of program memory.
01755     \since AVR-LibC v2.3 */
01756 static inline unsigned long long accum pgm_read_ullk (const unsigned long long accum
      *addr);
01757
01758 #else /* Doxygen */
01759
01760 #include <avr/pgmspace.h>
01761
01762 _Avrlibc_Def_Pgm_1 (hr, short fract)
01763 _Avrlibc_Def_Pgm_1 (uhr, unsigned short fract)
01764 _Avrlibc_Def_Pgm_2 (r, fract)
01765 _Avrlibc_Def_Pgm_2 (ur, unsigned fract)
01766 _Avrlibc_Def_Pgm_4 (lr, long fract)
01767 _Avrlibc_Def_Pgm_4 (ulr, unsigned long fract)
01768 _Avrlibc_Def_Pgm_8 (llr, long long fract)
01769 _Avrlibc_Def_Pgm_8 (ullr, unsigned long long fract)
01770
01771 _Avrlibc_Def_Pgm_2 (hk, short accum)
01772 _Avrlibc_Def_Pgm_2 (uhk, unsigned short accum)
01773 _Avrlibc_Def_Pgm_4 (k, accum)
01774 _Avrlibc_Def_Pgm_4 (uk, unsigned accum)
01775 _Avrlibc_Def_Pgm_8 (lk, long accum)
01776 _Avrlibc_Def_Pgm_8 (ulk, unsigned long accum)
01777 _Avrlibc_Def_Pgm_8 (llk, long long accum)
01778 _Avrlibc_Def_Pgm_8 (ullk, unsigned long long accum)
01779 #endif /* Doxygen */
01780
01781 #ifdef __DOXYGEN__
01782 /** \name Functions reading from PROGMEM_FAR */
01783
01784 /** \ingroup avr_stdfix
01785     Read a <tt>short fract</tt> from far address \p addr.
01786     The address is in the program memory.
01787     \since AVR-LibC v2.3 */
01788 static inline short fract pgm_read_hr_far (uint_farptr_t addr);
01789
01790 /** \ingroup avr_stdfix
01791     Read an <tt>unsigned short fract</tt> from far address \p addr.
01792     The address is in the program memory.
01793     \since AVR-LibC v2.3 */
01794 static inline unsigned short fract pgm_read_uhr_far (uint_farptr_t addr);
01795
01796 /** \ingroup avr_stdfix
01797     Read a <tt>fract</tt> from far address \p addr.
01798     The address is in the program memory.
01799     \since AVR-LibC v2.3 */
01800 static inline fract pgm_read_r_far (uint_farptr_t addr);
01801
01802 /** \ingroup avr_stdfix
```

```
01803     Read an <tt>unsigned fract</tt> from far address \p addr.
01804     The address is in the program memory.
01805     \since AVR-LibC v2.3 */
01806 static inline unsigned fract pgm_read_ur_far (uint_farptr_t addr);
01807
01808 /** \ingroup avr_stdfix
01809     Read a <tt>long fract</tt> from far address \p addr.
01810     The address is in the program memory.
01811     \since AVR-LibC v2.3 */
01812 static inline long fract pgm_read_lr_far (uint_farptr_t addr);
01813
01814 /** \ingroup avr_stdfix
01815     Read an <tt>unsigned long fract</tt> from far address \p addr.
01816     The address is in the program memory.
01817     \since AVR-LibC v2.3 */
01818 static inline unsigned long fract pgm_read_ulr_far (uint_farptr_t addr);
01819
01820 /** \ingroup avr_stdfix
01821     Read a <tt>long long fract</tt> from far address \p addr.
01822     The address is in the program memory.
01823     \since AVR-LibC v2.3 */
01824 static inline long long fract pgm_read_llr_far (uint_farptr_t addr);
01825
01826 /** \ingroup avr_stdfix
01827     Read an <tt>unsigned long long fract</tt> from far address \p addr.
01828     The address is in the program memory.
01829     \since AVR-LibC v2.3 */
01830 static inline unsigned long long fract pgm_read_ullr_far (uint_farptr_t addr);
01831
01832 /** \ingroup avr_stdfix
01833     Read a <tt>short accum</tt> from far address \p addr.
01834     The address is in the program memory.
01835     \since AVR-LibC v2.3 */
01836 static inline short accum pgm_read_hk_far (uint_farptr_t addr);
01837
01838 /** \ingroup avr_stdfix
01839     Read an <tt>unsigned short accum</tt> from far address \p addr.
01840     The address is in the program memory.
01841     \since AVR-LibC v2.3 */
01842 static inline unsigned short accum pgm_read_uhk_far (uint_farptr_t addr);
01843
01844 /** \ingroup avr_stdfix
01845     Read an <tt>accum</tt> from far address \p addr.
01846     The address is in the program memory.
01847     \since AVR-LibC v2.3 */
01848 static inline accum pgm_read_k_far (uint_farptr_t addr);
01849
01850 /** \ingroup avr_stdfix
01851     Read an <tt>unsigned accum</tt> from far address \p addr.
01852     The address is in the program memory.
01853     \since AVR-LibC v2.3 */
01854 static inline unsigned accum pgm_read_uk_far (uint_farptr_t addr);
01855
01856 /** \ingroup avr_stdfix
01857     Read a <tt>long accum</tt> from far address \p addr.
01858     The address is in the program memory.
01859     \since AVR-LibC v2.3 */
01860 static inline long accum pgm_read_lk_far (uint_farptr_t addr);
01861
01862 /** \ingroup avr_stdfix
01863     Read an <tt>unsigned long accum</tt> from far address \p addr.
01864     The address is in the program memory.
01865     \since AVR-LibC v2.3 */
01866 static inline unsigned long accum pgm_read_ulk_far (uint_farptr_t addr);
01867
01868 /** \ingroup avr_stdfix
01869     Read a <tt>long long accum</tt> from far address \p addr.
01870     The address is in the program memory.
01871     \since AVR-LibC v2.3 */
01872 static inline long long accum pgm_read_llk_far (uint_farptr_t addr);
01873
01874 /** \ingroup avr_stdfix
01875     Read an <tt>unsigned long long accum</tt> from far address \p addr.
```

```
01876     The address is in the program memory.
01877     \since AVR-LibC v2.3 */
01878 static inline unsigned long long accum pgm_read_ullk_far (uint_farptr_t addr);
01879
01880 #else /* Doxygen */
01881
01882 _Avrlibc_Def_Pgm_Far_1 (hr, short fract)
01883 _Avrlibc_Def_Pgm_Far_1 (uhr, unsigned short fract)
01884 _Avrlibc_Def_Pgm_Far_2 (r, fract)
01885 _Avrlibc_Def_Pgm_Far_2 (ur, unsigned fract)
01886 _Avrlibc_Def_Pgm_Far_4 (lr, long fract)
01887 _Avrlibc_Def_Pgm_Far_4 (ulr, unsigned long fract)
01888 _Avrlibc_Def_Pgm_Far_8 (llr, long long fract)
01889 _Avrlibc_Def_Pgm_Far_8 (ullr, unsigned long long fract)
01890
01891 _Avrlibc_Def_Pgm_Far_2 (hk, short accum)
01892 _Avrlibc_Def_Pgm_Far_2 (uhk, unsigned short accum)
01893 _Avrlibc_Def_Pgm_Far_4 (k, accum)
01894 _Avrlibc_Def_Pgm_Far_4 (uk, unsigned accum)
01895 _Avrlibc_Def_Pgm_Far_8 (lk, long accum)
01896 _Avrlibc_Def_Pgm_Far_8 (ulk, unsigned long accum)
01897 _Avrlibc_Def_Pgm_Far_8 (llk, long long accum)
01898 _Avrlibc_Def_Pgm_Far_8 (ullk, unsigned long long accum)
01899 #endif /* Doxygen */
01900
01901 /** \name EEPROM Read Functions */
01902
01903 /** \ingroup avr_stdfix
01904     Read a <tt>short fract</tt> from EEPROM address \a __p.
01905     \since AVR-LibC v2.3 */
01906 short fract eeprom_read_hr (const short fract *__p) __asm("eeprom_read_byte")
      __ATTR_PURE__;
01907
01908 /** \ingroup avr_stdfix
01909     Read an <tt>unsigned short fract</tt> from EEPROM address \a __p.
01910     \since AVR-LibC v2.3 */
01911 unsigned short fract eeprom_read_uhr (const unsigned short fract *__p)
      __asm("eeprom_read_byte") __ATTR_PURE__;
01912
01913 /** \ingroup avr_stdfix
01914     Read a <tt>fract</tt> from EEPROM address \a __p.
01915     \since AVR-LibC v2.3 */
01916 fract eeprom_read_r (const fract *__p) __asm("eeprom_read_word") __ATTR_PURE__;
01917
01918 /** \ingroup avr_stdfix
01919     Read an <tt>unsigned fract</tt> from EEPROM address \a __p.
01920     \since AVR-LibC v2.3 */
01921 unsigned fract eeprom_read_ur (const unsigned fract *__p) __asm("eeprom_read_word")
      __ATTR_PURE__;
01922
01923 /** \ingroup avr_stdfix
01924     Read a <tt>long fract</tt> from EEPROM address \a __p.
01925     \since AVR-LibC v2.3 */
01926 long fract eeprom_read_lr (const long fract *__p) __asm("eeprom_read_dword")
      __ATTR_PURE__;
01927
01928 /** \ingroup avr_stdfix
01929     Read an <tt>unsigned long fract</tt> from EEPROM address \a __p.
01930     \since AVR-LibC v2.3 */
01931 unsigned long fract eeprom_read_ulr (const unsigned long fract *__p)
      __asm("eeprom_read_dword") __ATTR_PURE__;
01932
01933 /** \ingroup avr_stdfix
01934     Read a <tt>long long fract</tt> from EEPROM address \a __p.
01935     \since AVR-LibC v2.3 */
01936 long long fract eeprom_read_llr (const long long fract *__p) __asm("eeprom_read_qword")
      __ATTR_PURE__;
01937
01938 /** \ingroup avr_stdfix
01939     Read an <tt>unsigned long long fract</tt> from EEPROM address \a __p.
01940     \since AVR-LibC v2.3 */
01941 unsigned long long fract eeprom_read_ullr (const unsigned long long fract *__p)
      __asm("eeprom_read_qword") __ATTR_PURE__;
```

```
01942
01943 /** \ingroup avr_stdfix
01944     Read a <tt>short accum</tt> from EEPROM address \a __p.
01945     \since AVR-LibC v2.3 */
01946 short accum eeprom_read_hk (const short accum *__p) __asm("eeprom_read_word")
      __ATTR_PURE__;
01947
01948 /** \ingroup avr_stdfix
01949     Read an <tt>unsigned short accum</tt> from EEPROM address \a __p.
01950     \since AVR-LibC v2.3 */
01951 unsigned short accum eeprom_read_uhk (const unsigned short accum *__p)
      __asm("eeprom_read_word") __ATTR_PURE__;
01952
01953 /** \ingroup avr_stdfix
01954     Read an <tt>accum</tt> from EEPROM address \a __p.
01955     \since AVR-LibC v2.3 */
01956 accum eeprom_read_k (const accum *__p) __asm("eeprom_read_dword") __ATTR_PURE__;
01957
01958 /** \ingroup avr_stdfix
01959     Read an <tt>unsigned accum</tt> from EEPROM address \a __p.
01960     \since AVR-LibC v2.3 */
01961 unsigned accum eeprom_read_uk (const unsigned accum *__p) __asm("eeprom_read_dword")
      __ATTR_PURE__;
01962
01963 /** \ingroup avr_stdfix
01964     Read a <tt>long accum</tt> from EEPROM address \a __p.
01965     \since AVR-LibC v2.3 */
01966 long accum eeprom_read_lk (const long accum *__p) __asm("eeprom_read_qword")
      __ATTR_PURE__;
01967
01968 /** \ingroup avr_stdfix
01969     Read an <tt>unsigned long accum</tt> from EEPROM address \a __p.
01970     \since AVR-LibC v2.3 */
01971 unsigned long accum eeprom_read_ulk (const unsigned long accum *__p)
      __asm("eeprom_read_qword") __ATTR_PURE__;
01972
01973 /** \ingroup avr_stdfix
01974     Read a <tt>long long accum</tt> from EEPROM address \a __p.
01975     \since AVR-LibC v2.3 */
01976 long long accum eeprom_read_llk (const long long accum *__p) __asm("eeprom_read_qword")
      __ATTR_PURE__;
01977
01978 /** \ingroup avr_stdfix
01979     Read an <tt>unsigned long long accum</tt> from EEPROM address \a __p.
01980     \since AVR-LibC v2.3 */
01981 unsigned long long accum eeprom_read_ullk (const unsigned long long accum *__p)
      __asm("eeprom_read_qword") __ATTR_PURE__;
01982
01983
01984 /** \name EEPROM Write Functions */
01985
01986 /** \ingroup avr_stdfix
01987     Write a <tt>short fract</tt> to EEPROM address \a __p.
01988     \since AVR-LibC v2.3 */
01989 void eeprom_write_hr (short fract *__p, short fract __value) __asm("eeprom_write_byte");
01990
01991 /** \ingroup avr_stdfix
01992     Write an <tt>unsigned short fract</tt> to EEPROM address \a __p.
01993     \since AVR-LibC v2.3 */
01994 void eeprom_write_uhr (unsigned short fract *__p, unsigned short fract __value)
      __asm("eeprom_write_byte");
01995
01996 /** \ingroup avr_stdfix
01997     Write a <tt>fract</tt> to EEPROM address \a __p.
01998     \since AVR-LibC v2.3 */
01999 void eeprom_write_r (fract *__p, fract __value) __asm("eeprom_write_word");
02000
02001 /** \ingroup avr_stdfix
02002     Write an <tt>unsigned fract</tt> to EEPROM address \a __p.
02003     \since AVR-LibC v2.3 */
02004 void eeprom_write_ur (unsigned fract *__p, unsigned fract __value)
      __asm("eeprom_write_word");
02005
```

```
02006 /** \ingroup avr_stdfix
02007     Write a <tt>long fract</tt> to EEPROM address \a __p.
02008     \since AVR-LibC v2.3 */
02009 void eeprom_write_lr (long fract *__p, long fract __value) __asm("eeprom_write_dword");
02010
02011 /** \ingroup avr_stdfix
02012     Write an <tt>unsigned long fract</tt> to EEPROM address \a __p.
02013     \since AVR-LibC v2.3 */
02014 void eeprom_write_ulr (unsigned long fract *__p, unsigned long fract __value)
      __asm("eeprom_write_dword");
02015
02016 /** \ingroup avr_stdfix
02017     Write a <tt>long long fract</tt> to EEPROM address \a __p.
02018     \since AVR-LibC v2.3 */
02019 void eeprom_write_llr (long long fract *__p, long long fract __value)
      __asm("eeprom_write_qword");
02020
02021 /** \ingroup avr_stdfix
02022     Write an <tt>unsigned long long fract</tt> to EEPROM address \a __p.
02023     \since AVR-LibC v2.3 */
02024 void eeprom_write_ullr (unsigned long long fract *__p, unsigned long long fract __value)
      __asm("eeprom_write_qword");
02025
02026 /** \ingroup avr_stdfix
02027     Write a <tt>short accum</tt> to EEPROM address \a __p.
02028     \since AVR-LibC v2.3 */
02029 void eeprom_write_hk (short accum *__p, short accum __value) __asm("eeprom_write_word");
02030
02031 /** \ingroup avr_stdfix
02032     Write an <tt>unsigned short accum</tt> to EEPROM address \a __p.
02033     \since AVR-LibC v2.3 */
02034 void eeprom_write_uhk (unsigned short accum *__p, unsigned short accum __value)
      __asm("eeprom_write_word");
02035
02036 /** \ingroup avr_stdfix
02037     Write an <tt>accum</tt> to EEPROM address \a __p.
02038     \since AVR-LibC v2.3 */
02039 void eeprom_write_k (accum *__p, accum __value) __asm("eeprom_write_dword");
02040
02041 /** \ingroup avr_stdfix
02042     Write an <tt>unsigned accum</tt> to EEPROM address \a __p.
02043     \since AVR-LibC v2.3 */
02044 void eeprom_write_uk (unsigned accum *__p, unsigned accum __value)
      __asm("eeprom_write_dword");
02045
02046 /** \ingroup avr_stdfix
02047     Write a <tt>long accum</tt> to EEPROM address \a __p.
02048     \since AVR-LibC v2.3 */
02049 void eeprom_write_lk (long accum *__p, long accum __value) __asm("eeprom_write_qword");
02050
02051 /** \ingroup avr_stdfix
02052     Write an <tt>unsigned long accum</tt> to EEPROM address \a __p.
02053     \since AVR-LibC v2.3 */
02054 void eeprom_write_ulk (unsigned long accum *__p, unsigned long accum __value)
      __asm("eeprom_write_qword");
02055
02056 /** \ingroup avr_stdfix
02057     Write a <tt>long long accum</tt> to EEPROM address \a __p.
02058     \since AVR-LibC v2.3 */
02059 void eeprom_write_llk (long long accum *__p, long long accum __value)
      __asm("eeprom_write_qword");
02060
02061 /** \ingroup avr_stdfix
02062     Write an <tt>unsigned long long accum</tt> to EEPROM address \a __p.
02063     \since AVR-LibC v2.3 */
02064 void eeprom_write_ullk (unsigned long long accum *__p, unsigned long long accum __value)
      __asm("eeprom_write_qword");
02065
02066
02067 /** \name EEPROM Update Functions */
02068
02069 /** \ingroup avr_stdfix
02070     Update a <tt>short fract</tt> at EEPROM address \a __p.
```

```
02071     \since AVR-LibC v2.3 */
02072 void eeprom_update_hr (short fract *__p, short fract __value)
      __asm("eeprom_update_byte");
02073
02074 /** \ingroup avr_stdfix
02075     Update an <tt>unsigned short fract</tt> at EEPROM address \a __p.
02076     \since AVR-LibC v2.3 */
02077 void eeprom_update_uhr (unsigned short fract *__p, unsigned short fract __value)
      __asm("eeprom_update_byte");
02078
02079 /** \ingroup avr_stdfix
02080     Update a <tt>fract</tt> at EEPROM address \a __p.
02081     \since AVR-LibC v2.3 */
02082 void eeprom_update_r (fract *__p, fract __value) __asm("eeprom_update_word");
02083
02084 /** \ingroup avr_stdfix
02085     Update an <tt>unsigned fract</tt> at EEPROM address \a __p.
02086     \since AVR-LibC v2.3 */
02087 void eeprom_update_ur (unsigned fract *__p, unsigned fract __value)
      __asm("eeprom_update_word");
02088
02089 /** \ingroup avr_stdfix
02090     Update a <tt>long fract</tt> at EEPROM address \a __p.
02091     \since AVR-LibC v2.3 */
02092 void eeprom_update_lr (long fract *__p, long fract __value)
      __asm("eeprom_update_dword");
02093
02094 /** \ingroup avr_stdfix
02095     Update an <tt>unsigned long fract</tt> at EEPROM address \a __p.
02096     \since AVR-LibC v2.3 */
02097 void eeprom_update_ulr (unsigned long fract *__p, unsigned long fract __value)
      __asm("eeprom_update_dword");
02098
02099 /** \ingroup avr_stdfix
02100     Update a <tt>long long fract</tt> at EEPROM address \a __p.
02101     \since AVR-LibC v2.3 */
02102 void eeprom_update_llr (long long fract *__p, long long fract __value)
      __asm("eeprom_update_qword");
02103
02104 /** \ingroup avr_stdfix
02105     Update an <tt>unsigned long long fract</tt> at EEPROM address \a __p.
02106     \since AVR-LibC v2.3 */
02107 void eeprom_update_ullr (unsigned long long fract *__p, unsigned long long fract
      __value) __asm("eeprom_update_qword");
02108
02109 /** \ingroup avr_stdfix
02110     Update a <tt>short accum</tt> at EEPROM address \a __p.
02111     \since AVR-LibC v2.3 */
02112 void eeprom_update_hk (short accum *__p, short accum __value)
      __asm("eeprom_update_word");
02113
02114 /** \ingroup avr_stdfix
02115     Update an <tt>unsigned short accum</tt> at EEPROM address \a __p.
02116     \since AVR-LibC v2.3 */
02117 void eeprom_update_uhk (unsigned short accum *__p, unsigned short accum __value)
      __asm("eeprom_update_word");
02118
02119 /** \ingroup avr_stdfix
02120     Update an <tt>accum</tt> at EEPROM address \a __p.
02121     \since AVR-LibC v2.3 */
02122 void eeprom_update_k (accum *__p, accum __value) __asm("eeprom_update_dword");
02123
02124 /** \ingroup avr_stdfix
02125     Update an <tt>unsigned accum</tt> at EEPROM address \a __p.
02126     \since AVR-LibC v2.3 */
02127 void eeprom_update_uk (unsigned accum *__p, unsigned accum __value)
      __asm("eeprom_update_dword");
02128
02129 /** \ingroup avr_stdfix
02130     Update a <tt>long accum</tt> at EEPROM address \a __p.
02131     \since AVR-LibC v2.3 */
02132 void eeprom_update_lk (long accum *__p, long accum __value)
      __asm("eeprom_update_qword");
```

```
02133
02134 /** \ingroup avr_stdfix
02135     Update an <tt>unsigned long accum</tt> at EEPROM address \a __p.
02136     \since AVR-LibC v2.3 */
02137 void eeprom_update_ulk (unsigned long accum *__p, unsigned long accum __value)
      __asm("eeprom_update_qword");
02138
02139 /** \ingroup avr_stdfix
02140     Update a <tt>long long accum</tt> at EEPROM address \a __p.
02141     \since AVR-LibC v2.3 */
02142 void eeprom_update_llk (long long accum *__p, long long accum __value)
      __asm("eeprom_update_qword");
02143
02144 /** \ingroup avr_stdfix
02145     Update an <tt>unsigned long long accum</tt> at EEPROM address \a __p.
02146     \since AVR-LibC v2.3 */
02147 void eeprom_update_ullk (unsigned long long accum *__p, unsigned long long accum
      __value) __asm("eeprom_update_qword");
02148
02149 #ifdef __cplusplus
02150 }
02151 #endif
02152
02153 #endif /* _STDFIX_AVRLIBC_H */
```

22.3 stdlib.h File Reference

Data Structures

- struct div_t
- struct ldiv_t

Macros

- #define EXIT_SUCCESS 0
- #define EXIT_FAILURE 1
- #define RAND_MAX 0x7FFF

Typedefs

- typedef int(* __compar_fn_t) (const void *, const void *)

Functions

- void abort (void)
- int abs (int __i)
- long labs (long __i)
- long long llabs (long long __i)
- void * bsearch (const void *__key, const void *__base, size_t __nmemb, size_t __size, __compar_fn_t __compar)
- div_t div (int __num, int __denom) __asm__("__divmodhi4")
- ldiv_t ldiv (long __num, long __denom) __asm__("__divmodsi4")
- void qsort (void *__base, size_t __nmemb, size_t __size, __compar_fn_t __compar)
- long strtol (const char *__nptr, char **__endptr, int __base)
- long long strtoll (const char *__nptr, char **__endptr, int __base)
- unsigned long strtoul (const char *__nptr, char **__endptr, int __base)
- unsigned long long strtoull (const char *__nptr, char **__endptr, int __base)

- long atol (const char *__s)
- int atoi (const char *__s)
- void exit (int __status)
- void * malloc (size_t __size)
- void free (void *__ptr)
- void * calloc (size_t __nele, size_t __size)
- void * realloc (void *__ptr, size_t __size)
- float strtof (const char *__nptr, char **__endptr)
- double strtod (const char *__nptr, char **__endptr)
- long double strtold (const char *__nptr, char **__endptr)
- int atexit (void(*func)(void))
- float atoff (const char *__nptr)
- double atof (const char *__nptr)
- long double atofl (const char *__nptr)
- int rand (void)
- void srand (unsigned int __seed)
- int rand_r (unsigned long *__ctx)

Variables

- size_t __malloc_margin
- char * __malloc_heap_start
- char * __malloc_heap_end

Non-standard (i.e. non-ISO C) functions.

- #define RANDOM_MAX 0x7FFFFFFF
- char * itoa (int val, char *s, int radix)
- char * ltoa (long val, char *s, int radix)
- char * utoa (unsigned int val, char *s, int radix)
- char * ultoa (unsigned long val, char *s, int radix)
- char * ulltoa (unsigned long long val, char *s, int radix)
- char * ulltoa_base10 (unsigned long long val, char *s)
- char * lltoa (long long val, char *s, int radix)
- long random (void)
- void srandom (unsigned long __seed)
- long random_r (unsigned long *__ctx)
- unsigned char sqrtu16_floor (unsigned int radic)
- unsigned int sqrtu32_floor (unsigned long radic)
- unsigned long sqrtu64_floor (unsigned long long radic)

Conversion functions for double arguments.

- #define DTOSTR_ALWAYS_SIGN 0x01 /* put '+' or ' ' for positives */
- #define DTOSTR_PLUS_SIGN 0x02 /* put '+' rather than ' ' */
- #define DTOSTR_UPPERCASE 0x04 /* put 'E' rather 'e' */
- char * ftostre (float __val, char *__s, unsigned char __prec, unsigned char __flags)
- char * dtostre (double __val, char *__s, unsigned char __prec, unsigned char __flags)
- char * ldtostre (long double __val, char *__s, unsigned char __prec, unsigned char __flags)
- char * ftostrf (float __val, signed char __width, unsigned char __prec, char *__s)
- char * dtostrf (double __val, signed char __width, unsigned char __prec, char *__s)
- char * ldtostrf (long double __val, signed char __width, unsigned char __prec, char *__s)

22.4 stdlib.h

Go to the documentation of this file.

```
00001 /* Copyright (c) 2002, Marek Michalkiewicz
00002    Copyright (c) 2004,2007 Joerg Wunsch
00003    Copyright (c) 2025 Georg-Johann Lay
00004
00005    Portions of documentation Copyright (c) 1990, 1991, 1993, 1994
00006    The Regents of the University of California.
00007
00008    All rights reserved.
00009
00010    Redistribution and use in source and binary forms, with or without
00011    modification, are permitted provided that the following conditions are met:
00012
00013    * Redistributions of source code must retain the above copyright
00014      notice, this list of conditions and the following disclaimer.
00015
00016    * Redistributions in binary form must reproduce the above copyright
00017      notice, this list of conditions and the following disclaimer in
00018      the documentation and/or other materials provided with the
00019      distribution.
00020
00021    * Neither the name of the copyright holders nor the names of
00022      contributors may be used to endorse or promote products derived
00023      from this software without specific prior written permission.
00024
00025   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
00026   AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
00027   IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
00028   ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
00029   LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
00030   CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
00031   SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
00032   INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
00033   CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
00034   ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
00035   POSSIBILITY OF SUCH DAMAGE. */
00036
00037 #ifndef _STDLIB_H_
00038 #define _STDLIB_H_ 1
00039
00040 #ifndef __ASSEMBLER__
00041
00042 #ifndef __DOXYGEN__
00043 #define __need_NULL
00044 #define __need_size_t
00045 #define __need_wchar_t
00046 #include <stddef.h>
00047
00048 #ifndef __ptr_t
00049 #define __ptr_t void *
00050 #endif
00051 #endif /* !__DOXYGEN__ */
00052
00053 #ifdef __cplusplus
00054 extern "C" {
00055 #endif
00056
00057 /** \file */
00058
00059 /** \defgroup avr_stdlib <stdlib.h>: General utilities
00060     \code #include <stdlib.h> \endcode
00061
00062     This file declares some basic C macros and functions as
00063     defined by the ISO standard, plus some AVR-specific extensions.
00064
00065     For some functions, \ref bench_libc "benchmarks" are available.
00066 */
00067
00068 /** \ingroup avr_stdlib */
00069 /**@{*/
```

```
00070 /** Result type for function div(). */
00071 typedef struct
00072 {
00073     int quot;                   /**< The Quotient. */
00074     int rem;                    /**< The Remainder. */
00075 } div_t;
00076
00077 /** Result type for function ldiv(). */
00078 typedef struct
00079 {
00080     long quot;                  /**< The Quotient. */
00081     long rem;                   /**< The Remainder. */
00082 } ldiv_t;
00083
00084 /** Comparision function type for qsort() and bsearch(),
00085     just for convenience. */
00086 typedef int (*__compar_fn_t)(const void *, const void *);
00087
00088 #ifndef __DOXYGEN__
00089
00090 #include <bits/attribs.h>
00091
00092 #endif
00093
00094 /** The abort() function causes abnormal program termination to occur.
00095     This realization disables interrupts and execution is
00096     effectively halted by entering an infinite loop. Static destructors
00097     and atexit() registered functions are not executed.  */
00098 extern void abort(void) __ATTR_NORETURN__;
00099
00100
00101 /** The abs() function computes the absolute value of the integer \c i.
00102    \note The abs() and labs() functions are builtins of gcc.  */
00103 extern __ATTR_ALWAYS_INLINE__ __ATTR_GNU_INLINE__
00104 int abs (int __i)
00105 {
00106     return __builtin_abs (__i);
00107 }
00108
00109
00110 /** The labs() function computes the absolute value of the long integer \c i.
00111    \note The abs() and labs() functions are builtins of gcc.  */
00112 extern __ATTR_ALWAYS_INLINE__ __ATTR_GNU_INLINE__
00113 long labs(long __i)
00114 {
00115     return __builtin_labs (__i);
00116 }
00117
00118
00119 /** The llabs() function computes the absolute value of the
00120     64-bit integer \c i.
00121     \since AVR-LibC v2.3 */
00122 extern __ATTR_ALWAYS_INLINE__ __ATTR_GNU_INLINE__
00123 long long llabs (long long __i)
00124 {
00125     if (__builtin_constant_p (__i))
00126     {
00127         return __builtin_llabs (__i);
00128     }
00129     else
00130     {
00131         register long long __r18 __asm("18") = __i;
00132         __asm (
00133 #ifdef __AVR_ERRATA_SKIP_JMP_CALL__
00134             "tst %r0+7"       "\n\t"
00135             "brpl 1f"         "\n\t"
00136             "%~call __negdi2" "\n"
00137             "1:"
00138 #else
00139             "sbrc %r0+7,7"    "\n\t"
00140             "%~call __negdi2"
00141 #endif
00142             : "+r" (__r18));
```

```
00143         return __r18;
00144     }
00145 }
00146
00147
00148 /**
00149      The bsearch() function searches an array of \c nmemb objects, the
00150      initial member of which is pointed to by \c base, for a member
00151      that matches the object pointed to by \c key.  The size of each
00152      member of the array is specified by \c size.
00153
00154      The contents of the array should be in ascending sorted order
00155      according to the comparison function referenced by \c compar.
00156      The \c compar routine is expected to have two arguments which
00157      point to the key object and to an array member, in that order,
00158      and should return an integer less than, equal to, or greater than
00159      zero if the key object is found, respectively, to be less than,
00160      to match, or be greater than the array member.
00161
00162      The bsearch() function returns a pointer to a matching member of
00163      the array, or a null pointer if no match is found.  If two
00164      members compare as equal, which member is matched is unspecified.
00165 */
00166 extern void *bsearch(const void *__key, const void *__base, size_t __nmemb, size_t
      __size, __compar_fn_t __compar);
00167
00168 /* __divmodhi4 and __divmodsi4 from libgcc.a */
00169 /**
00170      The div() function computes the value \c num/denom and returns
00171      the quotient and remainder in a structure named \c div_t that
00172      contains two int members named \c quot and \c rem.
00173 */
00174 extern div_t div(int __num, int __denom) __asm__("__divmodhi4") __ATTR_CONST__;
00175 /**
00176      The ldiv() function computes the value \c num/denom and returns
00177      the quotient and remainder in a structure named \c ldiv_t that
00178      contains two long integer members named \c quot and \c rem.
00179 */
00180 extern ldiv_t ldiv(long __num, long __denom) __asm__("__divmodsi4") __ATTR_CONST__;
00181
00182 /**
00183      The qsort() function is a modified partition-exchange sort, or
00184      quicksort.
00185
00186      The qsort() function sorts an array of \c nmemb objects, the
00187      initial member of which is pointed to by \c base.  The size of
00188      each object is specified by \c size.  The contents of the array
00189      base are sorted in ascending order according to a comparison
00190      function pointed to by \c compar, which requires two arguments
00191      pointing to the objects being compared.
00192
00193      The comparison function must return an integer less than, equal
00194      to, or greater than zero if the first argument is considered to
00195      be respectively less than, equal to, or greater than the second.
00196 */
00197 extern void qsort(void *__base, size_t __nmemb, size_t __size, __compar_fn_t __compar);
00198
00199 /**
00200     The strtol() function converts the string in \c nptr to a long
00201     value.  The conversion is done according to the given base, which
00202     must be between 2 and 36 inclusive, or be the special value 0.
00203
00204     The string may begin with an arbitrary amount of white space (as
00205     determined by isspace()) followed by a single optional \c '+' or \c '-'
00206     sign.  If \c base is zero or 16, the string may then include a
00207     \c "0x" or \c "0X" prefix, and the number will be read in base 16;
00208     otherwise, a zero base is taken as 10 (decimal) unless the next
00209     character is \c '0', in which case it is taken as 8 (octal).
00210
00211     Similarly, prefixes \c "0b" and \c "0B" signify base 2,
00212     and \c "0o" and \c "0O" signify base 8.
00213
00214     The remainder of the string is converted to a long value in the
```

```
00215     obvious manner, stopping at the first character which is not a
00216     valid digit in the given base.  (In bases above 10, the letter \c 'A'
00217     in either upper or lower case represents 10, \c 'B' represents 11,
00218     and so forth, with \c 'Z' representing 35.)
00219
00220     If \c endptr is not NULL, strtol() stores the address of the first
00221     invalid character in \c *endptr.  If there were no digits at all,
00222     however, strtol() stores the original value of \c nptr in \c
00223     *endptr.  (Thus, if \c *nptr is not \c '\\0' but \c **endptr is \c '\\0'
00224     on return, the entire string was valid.)
00225
00226     The strtol() function returns the result of the conversion, unless
00227     the value would underflow or overflow.  If no conversion could be
00228     performed, 0 is returned.  If an overflow or underflow occurs, \c
00229     errno is set to \ref avr_errno "ERANGE" and the function return value
00230     is clamped to \c LONG_MIN or \c LONG_MAX, respectively.
00231 */
00232 extern long strtol(const char *__nptr, char **__endptr, int __base);
00233
00234 /**
00235     The strtoll() function converts the string in \c nptr to a long long
00236     value.  The conversion is done according to the given base, which
00237     must be between 2 and 36 inclusive, or be the special value 0.
00238
00239     The string may begin with an arbitrary amount of white space (as
00240     determined by isspace()) followed by a single optional \c '+' or \c '-'
00241     sign.  If \c base is zero or 16, the string may then include a
00242     \c "0x" or \c "0X" prefix, and the number will be read in base 16;
00243     otherwise, a zero base is taken as 10 (decimal) unless the next
00244     character is \c '0', in which case it is taken as 8 (octal).
00245
00246     Similarly, prefixes \c "0b" and \c "0B" signify base 2,
00247     and \c "0o" and \c "0O" signify base 8.
00248
00249     The remainder of the string is converted to a long long value in the
00250     obvious manner, stopping at the first character which is not a
00251     valid digit in the given base.  (In bases above 10, the letter \c 'A'
00252     in either upper or lower case represents 10, \c 'B' represents 11,
00253     and so forth, with \c 'Z' representing 35.)
00254
00255     If \c endptr is not NULL, strtoll() stores the address of the first
00256     invalid character in \c *endptr.  If there were no digits at all,
00257     however, strtoll() stores the original value of \c nptr in \c
00258     *endptr.  (Thus, if \c *nptr is not \c '\\0' but \c **endptr is \c '\\0'
00259     on return, the entire string was valid.)
00260
00261     The strtoll() function returns the result of the conversion, unless
00262     the value would underflow or overflow.  If no conversion could be
00263     performed, 0 is returned.  If an overflow or underflow occurs, \c
00264     errno is set to \ref avr_errno "ERANGE" and the function return value
00265     is clamped to \c LLONG_MIN or \c LLONG_MAX, respectively.
00266     \since AVR-LibC v2.3
00267 */
00268 extern long long strtoll(const char *__nptr, char **__endptr, int __base);
00269
00270 /**
00271     The strtoul() function converts the string in \c nptr to an
00272     unsigned long value.  The conversion is done according to the
00273     given base, which must be between 2 and 36 inclusive, or be the
00274     special value 0.
00275
00276     The string may begin with an arbitrary amount of white space (as
00277     determined by isspace()) followed by a single optional \c '+' or \c '-'
00278     sign.  If \c base is zero or 16, the string may then include a
00279     \c "0x" or \c "0X" prefix, and the number will be read in base 16;
00280     otherwise, a zero base is taken as 10 (decimal) unless the next
00281     character is \c '0', in which case it is taken as 8 (octal).
00282
00283     Similarly, prefixes \c "0b" and \c "0B" signify base 2,
00284     and \c "0o" and \c "0O" signify base 8.
00285
00286     The remainder of the string is converted to an unsigned long value
00287     in the obvious manner, stopping at the first character which is
```

```
00288     not a valid digit in the given base.  (In bases above 10, the
00289     letter \c 'A' in either upper or lower case represents 10, \c 'B'
00290     represents 11, and so forth, with \c 'Z' representing 35.)
00291
00292     If \c endptr is not NULL, strtoul() stores the address of the first
00293     invalid character in \c *endptr.  If there were no digits at all,
00294     however, strtoul() stores the original value of \c nptr in \c
00295     *endptr.  (Thus, if \c *nptr is not \c '\\0' but \c **endptr is \c '\\0'
00296     on return, the entire string was valid.)
00297
00298     The strtoul() function returns either the result of the conversion
00299     or, if there was a leading minus sign, the negation of the result
00300     of the conversion, unless the original (non-negated) value would
00301     overflow; in the latter case, strtoul() returns ULONG_MAX, and \c
00302     errno is set to \ref avr_errno "ERANGE".  If no conversion could
00303     be performed, 0 is returned.
00304 */
00305 extern unsigned long strtoul(const char *__nptr, char **__endptr, int __base);
00306
00307 /**
00308     The strtoull() function converts the string in \c nptr to an
00309     unsigned long long value.  The conversion is done according to the
00310     given base, which must be between 2 and 36 inclusive, or be the
00311     special value 0.
00312
00313     The string may begin with an arbitrary amount of white space (as
00314     determined by isspace()) followed by a single optional \c '+' or \c '-'
00315     sign.  If \c base is zero or 16, the string may then include a
00316     \c "0x" or \c "0X" prefix, and the number will be read in base 16;
00317     otherwise, a zero base is taken as 10 (decimal) unless the next
00318     character is \c '0', in which case it is taken as 8 (octal).
00319
00320     Similarly, prefixes \c "0b" and \c "0B" signify base 2,
00321     and \c "0o" and \c "0O" signify base 8.
00322
00323     The remainder of the string is converted to an unsigned long long value
00324     in the obvious manner, stopping at the first character which is
00325     not a valid digit in the given base.  (In bases above 10, the
00326     letter \c 'A' in either upper or lower case represents 10, \c 'B'
00327     represents 11, and so forth, with \c 'Z' representing 35.)
00328
00329     If \c endptr is not NULL, strtoull() stores the address of the first
00330     invalid character in \c *endptr.  If there were no digits at all,
00331     however, strtoull() stores the original value of \c nptr in \c
00332     *endptr.  (Thus, if \c *nptr is not \c '\\0' but \c **endptr is \c '\\0'
00333     on return, the entire string was valid.)
00334
00335     The strtoull() function returns either the result of the conversion
00336     or, if there was a leading minus sign, the negation of the result
00337     of the conversion, unless the original (non-negated) value would
00338     overflow; in the latter case, strtoull() returns ULLONG_MAX, and \c
00339     errno is set to \ref avr_errno "ERANGE".  If no conversion could
00340     be performed, 0 is returned.
00341     \since AVR-LibC v2.3
00342 */
00343 extern unsigned long long strtoull(const char *__nptr, char **__endptr, int __base);
00344
00345 /**
00346     The atol() function converts the initial portion of the string
00347     pointed to by \p s to long integer representation. In contrast to
00348
00349         \code strtol(s, (char **)NULL, 10); \endcode
00350
00351     this function does not detect overflow (\c errno is not changed and
00352     the result value is not predictable), uses smaller memory (flash and
00353     stack) and works more quickly.
00354 */
00355 extern long atol(const char *__s) __ATTR_PURE__;
00356
00357 /**
00358     The atoi() function converts the initial portion of the string
00359     pointed to by \p s to integer representation. In contrast to
00360
```

```
00361         \code (int)strtol(s, (char **)NULL, 10); \endcode
00362
00363     this function does not detect overflow (\c errno is not changed and
00364     the result value is not predictable), uses smaller memory (flash and
00365     stack) and works more quickly.
00366 */
00367 extern int atoi(const char *__s) __ATTR_PURE__;
00368
00369 /**
00370    The exit() function terminates the application.  Since there is no
00371    environment to return to, \c status is ignored, and code execution
00372    will eventually reach an infinite loop, thereby effectively halting
00373    all code processing.  Before entering the infinite loop, interrupts
00374    are globally disabled.
00375
00376    Global destructors will be called before halting
00377    execution, see the \ref sec_dot_fini ".fini" sections.
00378 */
00379 extern void exit(int __status) __ATTR_NORETURN__;
00380
00381 /**
00382    \anchor a_malloc
00383    \fn void *malloc(size_t size)
00384    The malloc() function allocates \c size bytes of memory.
00385    If malloc() fails, a NULL pointer is returned.
00386
00387    Note that malloc() does \e not initialize the returned memory to
00388    zero bytes.  For that, see calloc().
00389
00390    See the chapter about \ref malloc "malloc() usage" for implementation
00391    details.
00392 */
00393 extern void *malloc(size_t __size) __ATTR_MALLOC__;
00394
00395 /**
00396    \anchor a_free
00397    The free() function makes the memory referenced by \c ptr
00398    available for future allocations.  The memory must have been
00399    allocated by a call to \ref a_malloc "malloc()",
00400    \ref a_realloc "realloc()", calloc() or other functions like strdup()
00401    or fdevopen() that allocate dynamic memory on the heap.
00402    If \c ptr is NULL, no action occurs.
00403 */
00404 extern void free(void *__ptr);
00405
00406 /**
00407    \ref malloc_tunables "tunable" for \ref a_malloc "malloc()".
00408    Default value is 32 bytes.
00409 */
00410 extern size_t __malloc_margin;
00411
00412 /**
00413    \ref malloc_tunables "tunable" for \ref a_malloc "malloc()".
00414    Default value is \ref __heap_start "__heap_start".
00415 */
00416 extern char *__malloc_heap_start;
00417
00418 /**
00419    \ref malloc_tunables "tunable" for \ref a_malloc "malloc()".
00420    Default value is __heap_end, which is weakly defined to 0 in
00421    the startup code.
00422 */
00423 extern char *__malloc_heap_end;
00424
00425 /**
00426    Allocate \c nele elements of \c size each.  Identical to calling
00427    \ref a_malloc "malloc()" using <tt>nele * size</tt> as argument
00428    (provided the product doesn't overflow),
00429    except the allocated memory will be cleared to zero.
00430 */
00431 extern void *calloc(size_t __nele, size_t __size) __ATTR_MALLOC__;
00432
00433 /**
```

```
00434    \anchor a_realloc
00435    The realloc() function tries to change the size of the region
00436    allocated at \c ptr to the new \c size value.  It returns a
00437    pointer to the new region.  The returned pointer might be the
00438    same as the old pointer, or a pointer to a completely different
00439    region.
00440
00441    The contents of the returned region up to either the old or the new
00442    size value (whatever is less) will be identical to the contents of
00443    the old region, even in case a new region had to be allocated.
00444
00445    It is acceptable to pass \c ptr as NULL, in which case realloc()
00446    will behave identical to \ref a_malloc "malloc()".
00447
00448    If the new memory cannot be allocated, realloc() returns NULL, and
00449    the region at \c ptr will not be changed.
00450 */
00451 extern void *realloc(void *__ptr, size_t __size) __ATTR_MALLOC__;
00452
00453 extern float strtof(const char *__nptr, char **__endptr);
00454
00455 /** \ingroup avr_stdlib
00456     The strtod() function is similar to strtof(), except that the conversion
00457     result is of type \c double instead of \c float. */
00458 extern double strtod(const char *__nptr, char **__endptr);
00459
00460 /** \ingroup avr_stdlib
00461     The strtold() function is similar to strtof(), except that the conversion
00462     result is of type \c long \c double instead of \c float. */
00463 extern long double strtold(const char *__nptr, char **__endptr);
00464
00465 /**
00466    \ingroup avr_stdlib
00467    The atexit() function registers function \a func to be run as part of
00468    the \c exit() function during \ref sec_dot_fini ".fini8".
00469    atexit() calls \ref a_malloc "malloc()". */
00470 extern int atexit(void (*func)(void));
00471
00472 /** \ingroup avr_stdlib
00473     \fn float atoff (const char *nptr)
00474
00475     The atoff() function converts the initial portion of the string pointed
00476     to by \a nptr to \c float representation.
00477
00478     It is equivalent to calling
00479     \code strtof(nptr, (char**) 0); \endcode */
00480 extern float atoff(const char *__nptr);
00481
00482 /** \ingroup avr_stdlib
00483     \fn  double atof (const char *nptr)
00484
00485     The atof() function converts the initial portion of the string pointed
00486     to by \a nptr to \c double representation.
00487
00488     It is equivalent to calling
00489     \code strtod(nptr, (char**) 0); \endcode */
00490 extern double atof(const char *__nptr);
00491
00492 /** \ingroup avr_stdlib
00493     \fn long double atofl (const char *nptr)
00494
00495     The atofl() function converts the initial portion of the string pointed
00496     to by \a nptr to \c long \c double representation.
00497
00498     It is equivalent to calling
00499     \code strtold(nptr, (char**) 0); \endcode */
00500 extern long double atofl(const char *__nptr);
00501
00502 /**
00503    \ingroup avr_stdlib
00504     Successful termination for exit(); evaluates to 0.
00505 */
00506 #define EXIT_SUCCESS 0
```

```
00507
00508 /**
00509    \ingroup avr_stdlib
00510     Unsuccessful termination for exit(); evaluates to a non-zero value.
00511 */
00512 #define EXIT_FAILURE 1
00513
00514 /** Highest number that can be generated by rand(). */
00515 #define RAND_MAX 0x7FFF
00516
00517 /**
00518      The rand() function computes a sequence of pseudo-random integers in the
00519      range of 0 to #RAND_MAX (as defined by the header file <stdlib.h>).
00520
00521      The srand() function sets its argument \c seed as the seed for a new
00522      sequence of pseudo-random numbers to be returned by rand().
00523      These sequences have a period of
00524 \htmlonly
00525 2<sup>32</sup>&minus;1
00526 \endhtmlonly
00527 \latexonly
00528 \begin{math} 2^{32}-1\end{math}
00529 \endlatexonly
00530 \manonly
00531 .EQ
00532 2^{32}-1
00533 .EN
00534 \endmanonly
00535      and are repeatable by calling srand() with the same seed value.
00536
00537      If no seed value is provided, the functions are automatically seeded with
00538      a value of 1.
00539
00540      rand() achieves a score of 100% in the \c bbattery_SmallCrush tests from
00541      the <a href="https://simul.iro.umontreal.ca/testu01/tu01.html">TestU01</a>
00542      suite.
00543
00544      For the resource consumptions, see the \ref bench_libc "libc benchmarks".
00545 */
00546 extern int rand(void);
00547 /**
00548    Pseudo-random number generator seeding; see rand().
00549 */
00550 extern void srand(unsigned int __seed);
00551
00552 /**
00553    Variant of rand() that stores the context in the user-supplied
00554    variable located at \c ctx instead of a static library variable
00555    so the function becomes re-entrant.
00556 */
00557 extern int rand_r(unsigned long *__ctx);
00558 /**@}*/
00559
00560 /**@{*/
00561 /** \name Non-standard (i.e. non-ISO C) functions.
00562  \ingroup avr_stdlib
00563 */
00564 /**
00565    \brief Convert an integer to a string.
00566
00567    The function itoa() converts the integer value from \c val into an
00568    ASCII representation that will be stored under \c s.  The caller
00569    is responsible for providing sufficient storage in \c s.
00570
00571    \note The minimal size of the buffer \c s depends on the choice of
00572    radix. For example, if the radix is 2 (binary), you need to supply a buffer
00573    with a minimal length of 8 * sizeof (int) + 1 characters, i.e. one
00574    character for each bit plus one for the string terminator. Using a larger
00575    radix will require a smaller minimal buffer size.
00576
00577    \warning If the buffer is too small, you risk a buffer overflow.
00578
00579    Conversion is done using the \c radix as base, which may be a
```

```
00580    number between 2 (binary conversion) and up to 36.  If \c radix
00581    is greater than 10, the next digit after \c '9' will be the letter
00582    \c 'a'.
00583
00584     If radix is 10 and val is negative, a minus sign will be prepended.
00585
00586    The itoa() function returns the pointer passed as \c s.
00587
00588    \note Decimal conversions can be sped up by using the ktoa() function
00589    from \ref avr_stdfix "<stdfix.h>" that converts fixed-point values to
00590    decimal ASCII, like in <code>ktoa((accum) val, s, FXTOA_TRUNC)</code>
00591    that converts \c val to a decimal ASCII representation with zero
00592    fractional digits.  For example, converting 1000 using itoa() takes
00593    around 700 cycles whereas ktoa() does the job in less than 300 cycles.
00594 */
00595 #ifdef  __DOXYGEN__
00596 extern char *itoa(int val, char *s, int radix);
00597 #else
00598 extern __ATTR_ALWAYS_INLINE__ __ATTR_GNU_INLINE__
00599 char *itoa (int __val, char *__s, int __radix)
00600 {
00601     if (!__builtin_constant_p (__radix))
00602     {
00603         extern char *__itoa (int, char *, int);
00604         return __itoa (__val, __s, __radix);
00605     }
00606     else if (__radix < 2 || __radix > 36)
00607     {
00608         *__s = 0;
00609         return __s;
00610     }
00611     else
00612     {
00613         extern char *__itoa_ncheck (int, char *, unsigned char);
00614         return __itoa_ncheck (__val, __s, __radix);
00615     }
00616 }
00617 #endif
00618
00619 /**
00620  \ingroup avr_stdlib
00621
00622    \brief Convert a long integer to a string.
00623
00624    The function ltoa() converts the long integer value from \c val into an
00625    ASCII representation that will be stored under \c s.  The caller
00626    is responsible for providing sufficient storage in \c s.
00627
00628    \note The minimal size of the buffer \c s depends on the choice of
00629    radix. For example, if the radix is 2 (binary), you need to supply a buffer
00630    with a minimal length of 8 * sizeof (long int) + 1 characters, i.e. one
00631    character for each bit plus one for the string terminator. Using a larger
00632    radix will require a smaller minimal buffer size.
00633
00634    \warning If the buffer is too small, you risk a buffer overflow.
00635
00636    Conversion is done using the \c radix as base, which may be a
00637    number between 2 (binary conversion) and up to 36.  If \c radix
00638    is greater than 10, the next digit after \c '9' will be the letter
00639    \c 'a'.
00640
00641    If radix is 10 and val is negative, a minus sign will be prepended.
00642
00643    The ltoa() function returns the pointer passed as \c s.
00644 */
00645 #ifdef  __DOXYGEN__
00646 extern char *ltoa(long val, char *s, int radix);
00647 #else
00648 extern __ATTR_ALWAYS_INLINE__ __ATTR_GNU_INLINE__
00649 char *ltoa (long __val, char *__s, int __radix)
00650 {
00651     if (!__builtin_constant_p (__radix))
00652     {
```

```
00653         extern char *__ltoa (long, char *, int);
00654         return __ltoa (__val, __s, __radix);
00655     }
00656     else if (__radix < 2 || __radix > 36)
00657     {
00658         *__s = 0;
00659         return __s;
00660     }
00661     else
00662     {
00663         extern char *__ltoa_ncheck (long, char *, unsigned char);
00664         return __ltoa_ncheck (__val, __s, __radix);
00665     }
00666 }
00667 #endif
00668
00669 /**
00670  \ingroup avr_stdlib
00671
00672    \brief Convert an unsigned integer to a string.
00673
00674    The function utoa() converts the unsigned integer value from \c val into an
00675    ASCII representation that will be stored under \c s.  The caller
00676    is responsible for providing sufficient storage in \c s.
00677
00678    \note The minimal size of the buffer \c s depends on the choice of
00679    radix. For example, if the radix is 2 (binary), you need to supply a buffer
00680    with a minimal length of 8 * sizeof (unsigned int) + 1 characters, i.e. one
00681    character for each bit plus one for the string terminator. Using a larger
00682    radix will require a smaller minimal buffer size.
00683
00684    \warning If the buffer is too small, you risk a buffer overflow.
00685
00686    Conversion is done using the \c radix as base, which may be a
00687    number between 2 (binary conversion) and up to 36.  If \c radix
00688    is greater than 10, the next digit after \c '9' will be the letter
00689    \c 'a'.
00690
00691    The utoa() function returns the pointer passed as \c s.
00692
00693    \note Decimal conversions can be sped up by using the uktoa() function from
00694    \ref avr_stdfix "<stdfix.h>" that converts fixed-point values to decimal
00695    ASCII, like in <code>uktoa((unsigned accum) val, s, FXTOA_TRUNC)</code>
00696    that converts \c val to a decimal ASCII representation with zero
00697    fractional digits.  For example, converting 1000 using utoa() takes
00698    around 700 cycles whereas uktoa() does the job in less than 300 cycles.
00699 */
00700 #ifdef  __DOXYGEN__
00701 extern char *utoa(unsigned int val, char *s, int radix);
00702 #else
00703 extern __ATTR_ALWAYS_INLINE__ __ATTR_GNU_INLINE__
00704 char *utoa (unsigned int __val, char *__s, int __radix)
00705 {
00706     if (!__builtin_constant_p (__radix))
00707     {
00708         extern char *__utoa (unsigned int, char *, int);
00709         return __utoa (__val, __s, __radix);
00710     }
00711     else if (__radix < 2 || __radix > 36)
00712     {
00713         *__s = 0;
00714         return __s;
00715     }
00716     else
00717     {
00718         extern char *__utoa_ncheck (unsigned int, char *, unsigned char);
00719         return __utoa_ncheck (__val, __s, __radix);
00720     }
00721 }
00722 #endif
00723
00724 /**
00725  \ingroup avr_stdlib
```

```
00726    \brief Convert an unsigned long integer to a string.
00727
00728    The function ultoa() converts the unsigned long integer value from
00729    \c val into an ASCII representation that will be stored under \c s.
00730    The caller is responsible for providing sufficient storage in \c s.
00731
00732    \note The minimal size of the buffer \c s depends on the choice of
00733    radix. For example, if the radix is 2 (binary), you need to supply a buffer
00734    with a minimal length of 8 * sizeof (unsigned long int) + 1 characters,
00735    i.e. one character for each bit plus one for the string terminator. Using a
00736    larger radix will require a smaller minimal buffer size.
00737
00738    \warning If the buffer is too small, you risk a buffer overflow.
00739
00740    Conversion is done using the \c radix as base, which may be a
00741    number between 2 (binary conversion) and up to 36.  If \c radix
00742    is greater than 10, the next digit after \c '9' will be the letter
00743    \c 'a'.
00744
00745    The ultoa() function returns the pointer passed as \c s.
00746 */
00747 #ifdef  __DOXYGEN__
00748 extern char *ultoa(unsigned long val, char *s, int radix);
00749 #else
00750 extern __ATTR_ALWAYS_INLINE__ __ATTR_GNU_INLINE__
00751 char *ultoa (unsigned long __val, char *__s, int __radix)
00752 {
00753     if (!__builtin_constant_p (__radix))
00754     {
00755         extern char *__ultoa (unsigned long, char *, int);
00756         return __ultoa (__val, __s, __radix);
00757     }
00758     else if (__radix < 2 || __radix > 36)
00759     {
00760         *__s = 0;
00761         return __s;
00762     }
00763     else
00764     {
00765         extern char *__ultoa_ncheck (unsigned long, char *, unsigned char);
00766         return __ultoa_ncheck (__val, __s, __radix);
00767     }
00768 }
00769 #endif
00770
00771
00772 /** \ingroup avr_stdlib
00773    \brief Convert an unsigned 64-bit integer to a string.
00774
00775    The function ulltoa() writes the ASCII representation of
00776    the unsigned 64-bit integer \c val to a string starting at \c s.
00777
00778    A very rough estimation of the execution time is
00779
00780 \htmlonly
00781 <i>Cycles</i> &asymp; 950 + 23&middot;<i>N</i> +
      8.3&middot;<i>N</i><sup>2</sup>&middot;log(<i>radix</i>) &plusmn; 400
00782 \endhtmlonly
00783 \latexonly
00784 \begin{math} \mathit{Cycles} \approx  950 + 23\cdot \mathit{N} + 8.3\cdot
      \mathit{N}^{2}\cdot \log (\mathit{radix}) \pm  400\end{math}
00785 \endlatexonly
00786 \manonly
00787 .EQ
00788 Cycles = 950 + 23*N + 8.3*N^2*log(radix) +/- 400
00789 .EN
00790 \endmanonly
00791
00792    where <i>N</i> denotes the number of digits in the result,
00793    and \e log stands for the Natural Logarithm.
00794    This means a decimal conversion can take up to 9000 cycles,
00795    a hexadecimal conversions can take up to 7800 cycles,
00796    and a binary conversion can take more than 27000 cycles.
```

```
00797
00798    \param val
00799    An unsigned 64-bit integral value for which the ASCII representation
00800    is computed.
00801
00802    \param s
00803    The location to which the string representation should be stored.
00804    The caller is responsible for providing sufficient storage in \c s.
00805    The minimal size of the buffer \c s depends on the choice of the
00806    radix. For example, if the radix is 10 (decimal), the function will
00807    write at most 21 characters (including the terminating '\\0').
00808
00809    \param radix
00810    The Conversion is done using the \c radix as base, which may be a number
00811    between 2 (binary conversion) and up to 36.  If \c radix is greater than 10,
00812    the next digit after \c '9' will be the letter \c 'a'.
00813
00814    \return The ulltoa() function returns the pointer passed as \c s.
00815
00816    \since AVR-LibC v2.3
00817 */
00818 #ifdef  __DOXYGEN__
00819 extern char* ulltoa(unsigned long long val, char *s, int radix);
00820 #else
00821 extern char* ulltoa(unsigned long long, char*, int) __asm("__ulltoa");
00822 #endif /* Doxygen */
00823
00824 /** \ingroup avr_stdlib
00825    \brief Convert an unsigned 64-bit integer to a decimal string.
00826
00827    The function ulltoa_base10() writes the decimal ASCII representation of
00828    the unsigned 64-bit integer \c val to a string starting at \c s.
00829    The effect is the same like for <tt>ulltoa(val, s, 10)</tt>.
00830
00831    This function can be used for decimal ASCII conversions when
00832    ulltoa() is not fast enough.  It consumes no more than 3300 cycles
00833    (no more than 2800 cycles with \c MUL),
00834    where ulltoa() may consume up to 9000 cycles for a decimal conversion.
00835
00836    \param val
00837    An unsigned 64-bit integral value for which the decimal ASCII
00838    representation is computed.
00839
00840    \param s
00841    The location to which the string representation should be stored.
00842    The caller is responsible for providing sufficient storage in \c s.
00843    The function will write at most 21 characters (including the
00844    terminating '\\0').
00845
00846    \return The ulltoa_base10() function returns the pointer passed as \c s.
00847
00848    \since AVR-LibC v2.3
00849 */
00850 #ifdef  __DOXYGEN__
00851 extern char* ulltoa_base10(unsigned long long val, char *s);
00852 #else
00853 extern char* ulltoa_base10(unsigned long long, char*) __asm("__ulltoa_base10");
00854 #endif /* Doxygen */
00855
00856 /** \ingroup avr_stdlib
00857    \brief Convert a signed 64-bit integer to a string.
00858
00859    The function lltoa() writes the ASCII representation of
00860    the signed 64-bit integer \c val to a string starting at \c s.
00861    Except for decimal conversions with a negative \p val,
00862    the effect is the same like with \c ulltoa().
00863
00864    \param val
00865    A signed 64-bit integral value for which the ASCII representation
00866    is computed.
00867
00868    \param s
00869    The location to which the string representation should be stored.
```

```
00870    The caller is responsible for providing sufficient storage in \c s.
00871    The minimal size of the buffer \c s depends on the choice of the
00872    radix. For example, if the radix is 10 (decimal), the function will
00873    write at most 21 characters (including the terminating '\\0').
00874
00875    \param radix
00876    The Conversion is done using the \c radix as base, which may be a number
00877    between 2 (binary conversion) and up to 36.  If \c radix is greater than 10,
00878    the next digit after \c '9' will be the letter \c 'a'.
00879
00880    \return The lltoa() function returns the pointer passed as \c s.
00881
00882    \since AVR-LibC v2.3
00883 */
00884 #ifdef  __DOXYGEN__
00885 extern char* lltoa(long long val, char *s, int radix);
00886 #else
00887 extern char* lltoa(long long, char*, int) __asm("__lltoa");
00888 #endif /* Doxygen */
00889
00890 /**  \ingroup avr_stdlib
00891 Highest number that can be generated by random(). */
00892 #define RANDOM_MAX 0x7FFFFFFF
00893
00894 /**
00895  \ingroup avr_stdlib
00896      The random() function computes a sequence of pseudo-random integers in the
00897      range of 0 to #RANDOM_MAX (as defined by the header file <stdlib.h>).
00898
00899      The srandom() function sets its argument \c seed as the seed for a new
00900      sequence of pseudo-random numbers to be returned by random().
00901      These sequences have a period of
00902 \htmlonly
00903 2<sup>31</sup>&minus;2
00904 \endhtmlonly
00905 \latexonly
00906 \begin{math} 2^{31}-2\end{math}
00907 \endlatexonly
00908 \manonly
00909 .EQ
00910 2^{31}-2
00911 .EN
00912 \endmanonly
00913      and are repeatable by calling srandom() with the same seed value.
00914
00915      If no seed value is provided, the functions are automatically seeded with
00916      a value of 1.
00917
00918      For the resource consumptions, see the \ref bench_libc "libc benchmarks".
00919 */
00920 extern long random(void);
00921 /**
00922  \ingroup avr_stdlib
00923    Pseudo-random number generator seeding; see random().
00924 */
00925 extern void srandom(unsigned long __seed);
00926
00927 /**
00928  \ingroup avr_stdlib
00929    Variant of random() that stores the context in the user-supplied
00930    variable located at \c ctx instead of a static library variable
00931    so the function becomes re-entrant.
00932 */
00933 extern long random_r(unsigned long *__ctx);
00934
00935 /** \ingroup avr_stdlib
00936     \return Returns the square root of the 16-bit value \p radic,
00937     rounded down to the next integral value.
00938  */
00939 #ifdef __DOXYGEN__
00940 extern unsigned char sqrtu16_floor(unsigned int radic);
00941 #else
00942 extern unsigned char sqrtu16_floor(unsigned) __asm("__sqrthi");
```

```
00943 extern __ATTR_ALWAYS_INLINE__ __ATTR_GNU_INLINE__
00944 unsigned char sqrtu16_floor(unsigned __r)
00945 {
00946   if (__builtin_constant_p (__r))
00947     {
00948       return (unsigned char) __builtin_sqrtf ((float) __r);
00949     }
00950   else
00951     {
00952       extern unsigned char __sqrthi (unsigned);
00953       return __sqrthi (__r);
00954     }
00955 }
00956 #endif /* Doxygen */
00957
00958 /** \ingroup avr_stdlib
00959     \return Returns the square root of the 32-bit value \p radic,
00960     rounded down to the next integral value.
00961  */
00962 #ifdef __DOXYGEN__
00963 extern unsigned int sqrtu32_floor(unsigned long radic);
00964 #else
00965 extern unsigned int sqrtu32_floor(unsigned long radic) __asm("__sqrtsi");
00966 #endif /* Doxygen */
00967
00968 /** \ingroup avr_stdlib
00969     \return Returns the square root of the 64-bit value \p radic,
00970     rounded down to the next integral value.
00971  */
00972 #ifdef __DOXYGEN__
00973 extern unsigned long sqrtu64_floor(unsigned long long radic);
00974 #else
00975 __extension__ extern unsigned long sqrtu64_floor(unsigned long long radic)
      __asm("__sqrtdi");
00976 #endif /* Doxygen */
00977
00978
00979 #endif /* __ASSEMBLER */
00980 /**@}*/
00981
00982 /**@{*/
00983 /** \name Conversion functions for double arguments. */
00984 /** \ingroup avr_stdlib
00985     Bit value that can be passed in \c flags to ftostre(),
00986     dtostre() and ldtostre(). */
00987 #define DTOSTR_ALWAYS_SIGN 0x01        /* put '+' or ' ' for positives */
00988 /** \ingroup avr_stdlib
00989     Bit value that can be passed in \c flags to ftostre(),
00990     dtostre() and ldtostre(). */
00991 #define DTOSTR_PLUS_SIGN   0x02        /* put '+' rather than ' ' */
00992 /** \ingroup avr_stdlib
00993     Bit value that can be passed in \c flags to ftostre(),
00994     dtostre() and ldtostre(). */
00995 #define DTOSTR_UPPERCASE   0x04        /* put 'E' rather 'e' */
00996
00997 #ifndef __ASSEMBLER__
00998
00999 /**
01000    \ingroup avr_stdlib
01001    The ftostre() function converts the \c float value passed in \c val into
01002    an ASCII representation that will be stored under \c s.  The caller
01003    is responsible for providing sufficient storage in \c s.
01004
01005    Conversion is done in the format
01006    <tt>&quot;[-]d.ddde&plusmn;dd&quot;</tt> where there is
01007    one digit before the decimal-point character and the number of
01008    digits after it is equal to the precision \c prec; if the precision
01009    is zero, no decimal-point character appears.  If \c flags has the
01010    #DTOSTR_UPPERCASE bit set, the letter \c 'E' (rather than \c 'e' ) will be
01011    used to introduce the exponent.  The exponent always contains two
01012    digits; if the value is zero, the exponent is \c "00".
01013
01014    If \c flags has the #DTOSTR_ALWAYS_SIGN bit set, a space character
```

```
01015    will be placed into the leading position for positive numbers.
01016
01017    If \c flags has the #DTOSTR_PLUS_SIGN bit set, a plus sign will be
01018    used instead of a space character in this case.
01019
01020    The ftostre() function returns the pointer to the converted string \c s.
01021 */
01022 extern char *ftostre(float __val, char *__s, unsigned char __prec, unsigned char
     __flags);
01023 /**
01024    \ingroup avr_stdlib
01025    The dtostre() function is similar to the ftostre() function, except that
01026    it converts a \c double value instead of a \c float value.
01027
01028    dtostre() is currently only supported when \c double is a 32-bit type. */
01029 extern char *dtostre(double __val, char *__s, unsigned char __prec, unsigned char
     __flags);
01030 /**
01031    \ingroup avr_stdlib
01032    The ldtostre() function is similar to the ftostre() function, except that
01033    it converts a \c long \c double value instead of a \c float value.
01034
01035    ldtostre() is currently only supported when \c long \c double is a
01036    32-bit type. */
01037 extern char *ldtostre(long double __val, char *__s, unsigned char __prec, unsigned char
     __flags);
01038
01039 /**
01040    \ingroup avr_stdlib
01041    The ftostrf() function converts the \c float value passed in \c val into
01042    an ASCII representation that will be stored in \c s.  The caller
01043    is responsible for providing sufficient storage in \c s.
01044
01045    Conversion is done in the format \c "[-]d.ddd".  The minimum field
01046    width of the output string (including the possible \c '.' and the possible
01047    sign for negative values) is given in \c width, and \c prec determines
01048    the number of digits after the decimal sign. \c width is signed value,
01049    negative for left adjustment.
01050
01051    The ftostrf() function returns the pointer to the converted string \c s.
01052 */
01053 extern char *ftostrf(float __val, signed char __width, unsigned char __prec, char *__s);
01054 /**
01055    \ingroup avr_stdlib
01056    The dtostrf() function is similar to the ftostrf() function, except that
01057    converts a \c double value instead of a \c float value.
01058
01059    ldtostre() is currently only supported when \c double is a 32-bit type. */
01060 extern char *dtostrf(double __val, signed char __width, unsigned char __prec, char
     *__s);
01061 /**
01062    \ingroup avr_stdlib
01063    The ldtostrf() function is similar to the ftostrf() function, except that
01064    converts a \c long \c double value instead of a \c float value.
01065
01066    ldtostre() is currently only supported when \c long \c double is a
01067    32-bit type. */
01068 extern char *ldtostrf(long double __val, signed char __width, unsigned char __prec, char
     *__s);
01069
01070 /**@}*/
01071
01072 #ifndef __DOXYGEN__
01073 /* dummy declarations for libstdc++ compatibility */
01074 extern int system (const char *);
01075 extern char *getenv (const char *);
01076 #endif /* __DOXYGEN__ */
01077
01078 #ifdef __cplusplus
01079 }
01080 #endif
01081
01082 #endif /* __ASSEMBLER */
```

```
01083
01084 #endif /* _STDLIB_H_ */
```

22.5 builtins.h File Reference

Functions

- void __builtin_avr_sei (void)
- void __builtin_avr_cli (void)
- void __builtin_avr_sleep (void)
- void __builtin_avr_wdr (void)
- uint8_t __builtin_avr_swap (uint8_t __b)
- uint16_t __builtin_avr_fmul (uint8_t __a, uint8_t __b)
- int16_t __builtin_avr_fmuls (int8_t __a, int8_t __b)
- int16_t __builtin_avr_fmulsu (int8_t __a, uint8_t __b)
- float __builtin_powif (float base, int expo)
- double __builtin_powi (double base, int expo)
- long double __builtin_powil (long double base, int expo)

22.6 builtins.h

Go to the documentation of this file.

```
00001 /* Copyright (c) 2008 Anatoly Sokolov
00002    Copyright (c) 2010 Joerg Wunsch
00003    All rights reserved.
00004
00005    Redistribution and use in source and binary forms, with or without
00006    modification, are permitted provided that the following conditions are met:
00007
00008    * Redistributions of source code must retain the above copyright
00009      notice, this list of conditions and the following disclaimer.
00010
00011    * Redistributions in binary form must reproduce the above copyright
00012      notice, this list of conditions and the following disclaimer in
00013      the documentation and/or other materials provided with the
00014      distribution.
00015
00016    * Neither the name of the copyright holders nor the names of
00017      contributors may be used to endorse or promote products derived
00018      from this software without specific prior written permission.
00019
00020   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
00021   AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
00022   IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
00023   ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
00024   LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
00025   CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
00026   SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
00027   INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
00028   CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
00029   ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
00030   POSSIBILITY OF SUCH DAMAGE. */
00031
00032 /* include/avr/builtins.h.  Generated from builtins.h.in by configure. */
00033
00034 /*
00035    avr/builtins.h - Intrinsic functions built into the compiler
00036  */
00037
00038 #ifndef _AVR_BUILTINS_H_
00039 #define _AVR_BUILTINS_H_
00040
```

```
00041 #ifndef __DOXYGEN__
00042 #ifndef __HAS_DELAY_CYCLES
00043 #define __HAS_DELAY_CYCLES 1
00044 #endif
00045 #endif /* __DOXYGEN__ */
00046
00047 /* For GCC built-ins, we should not define prototypes,
00048    hence only document that stuff.  */
00049 #ifdef __DOXYGEN__
00050
00051 /** \file */
00052 /** \defgroup avr_builtins <avr/builtins.h>: avr-gcc builtins documentation
00053     \code #include <avr/builtins.h> \endcode
00054
00055     \note This file only documents some avr-gcc builtins.
00056     For functions built-in in the compiler, there should be no
00057     prototype declarations.
00058
00059     See also the
00060     <a href="https://gcc.gnu.org/onlinedocs/gcc/AVR-Built-in-Functions.html"
00061        >GCC documentation</a> for a full list of avr-gcc builtins.
00062 */
00063
00064 /**
00065     \ingroup avr_builtins
00066
00067     Enables interrupts by setting the global interrupt mask.  */
00068 extern void __builtin_avr_sei(void);
00069
00070 /**
00071     \ingroup avr_builtins
00072
00073     Disables all interrupts by clearing the global interrupt mask.  */
00074 extern void __builtin_avr_cli(void);
00075
00076 /**
00077     \ingroup avr_builtins
00078
00079     Emits a \c SLEEP instruction.  */
00080
00081 extern void __builtin_avr_sleep(void);
00082
00083 /**
00084     \ingroup avr_builtins
00085
00086     Emits a WDR (watchdog reset) instruction.  */
00087 extern void __builtin_avr_wdr(void);
00088
00089 /**
00090     \ingroup avr_builtins
00091
00092     Emits a SWAP (nibble swap) instruction on __b.  */
00093 extern uint8_t __builtin_avr_swap(uint8_t __b);
00094
00095 /**
00096     \ingroup avr_builtins
00097
00098     Emits an FMUL (fractional multiply unsigned) instruction.  */
00099 extern uint16_t __builtin_avr_fmul(uint8_t __a, uint8_t __b);
00100
00101 /**
00102     \ingroup avr_builtins
00103
00104     Emits an FMUL (fractional multiply signed) instruction.  */
00105 extern int16_t __builtin_avr_fmuls(int8_t __a, int8_t __b);
00106
00107 /**
00108     \ingroup avr_builtins
00109
00110     Emits an FMUL (fractional multiply signed with unsigned) instruction.  */
00111 extern int16_t __builtin_avr_fmulsu(int8_t __a, uint8_t __b);
00112
00113 #if __HAS_DELAY_CYCLES
```

```
00114 /**
00115     \ingroup avr_builtins
00116
00117     Emits a sequence of instructions causing the CPU to spend
00118     \c __n cycles on it.  */
00119 extern void __builtin_avr_delay_cycles(uint32_t __n);
00120 #endif
00121
00122 /** \ingroup avr_builtins
00123     Returns \a base raised to the power of \a expo.
00124     Since avr-gcc v15 this function is implemented in assembly.
00125     See also the \ref bench_libm "benchmarks". */
00126 float __builtin_powif (float base, int expo);
00127
00128 /** \ingroup avr_builtins
00129     Returns \a base raised to the power of \a expo.
00130     Since avr-gcc v15 this function is implemented in assembly. */
00131 double __builtin_powi (double base, int expo);
00132
00133 /** \ingroup avr_builtins
00134     Returns \a base raised to the power of \a expo.
00135     Since avr-gcc v15 this function is implemented in assembly.
00136     See also the \ref bench_libf7 "benchmarks". */
00137 long double __builtin_powil (long double base, int expo);
00138
00139 #endif /* DOXYGEN */
00140 #endif /* _AVR_BUILTINS_H_ */
```

22.7 version.h

```
00001 /* Copyright (c) 2005, Joerg Wunsch                                -*- c -*-
00002    All rights reserved.
00003
00004    Redistribution and use in source and binary forms, with or without
00005    modification, are permitted provided that the following conditions are met:
00006
00007    * Redistributions of source code must retain the above copyright
00008      notice, this list of conditions and the following disclaimer.
00009
00010    * Redistributions in binary form must reproduce the above copyright
00011      notice, this list of conditions and the following disclaimer in
00012      the documentation and/or other materials provided with the
00013      distribution.
00014
00015    * Neither the name of the copyright holders nor the names of
00016      contributors may be used to endorse or promote products derived
00017      from this software without specific prior written permission.
00018
00019   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
00020   AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
00021   IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
00022   ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
00023   LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
00024   CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
00025   SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
00026   INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
00027   CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
00028   ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
00029   POSSIBILITY OF SUCH DAMAGE. */
00030
00031 /* include/avr/version.h.  Generated from version.h.in by configure. */
00032
00033 /** \defgroup avr_version <avr/version.h>: AVR-LibC version macros
00034     \code #include <avr/version.h> \endcode
00035
00036     This header file defines macros that contain version numbers and
00037     strings describing the current version of AVR-LibC.
00038
00039     The version number itself basically consists of three pieces that
00040     are separated by a dot: the major number, the minor number, and
00041     the revision number.  For development versions (which use an odd
```

```
00042     minor number), the string representation additionally gets the
00043     date code (YYYYMMDD) appended.
00044
00045     This file will also be included by \c <avr/io.h>.  That way,
00046     portable tests can be implemented using \c <avr/io.h> that can be
00047     used in code that wants to remain backwards-compatible to library
00048     versions prior to the date when the library version API had been
00049     added, as referenced but undefined C preprocessor macros
00050     automatically evaluate to 0.
00051 */
00052
00053 #ifndef _AVR_VERSION_H_
00054 #define _AVR_VERSION_H_
00055
00056 /** \ingroup avr_version
00057     String literal representation of the current library version. */
00058 #define __AVR_LIBC_VERSION_STRING__ "2.3.0"
00059
00060 /** \ingroup avr_version
00061     Numerical representation of the current library version.
00062
00063     In the numerical representation, the major number is multiplied by
00064     10000, the minor number by 100, and all three parts are then
00065     added.  It is intended to provide a monotonically increasing
00066     numerical value that can easily be used in numerical checks.
00067  */
00068 #define __AVR_LIBC_VERSION__        20300UL
00069
00070 /** \ingroup avr_version
00071     String literal representation of the release date. */
00072 #define __AVR_LIBC_DATE_STRING__    "20251223"
00073
00074 /** \ingroup avr_version
00075     Numerical representation of the release date. */
00076 #define __AVR_LIBC_DATE__           20251223UL
00077
00078 /** \ingroup avr_version
00079     Library major version number. */
00080 #define __AVR_LIBC_MAJOR__          2
00081
00082 /** \ingroup avr_version
00083     Library minor version number. */
00084 #define __AVR_LIBC_MINOR__          3
00085
00086 /** \ingroup avr_version
00087     Library revision number. */
00088 #define __AVR_LIBC_REVISION__       0
00089
00090 #endif /* _AVR_VERSION_H_ */
```

22.8 delay.h File Reference

Macros

- #define F_CPU 1000000UL

Functions

- static void _delay_ms (double __ms)
- static void _delay_us (double __us)

22.9 delay.h

Go to the documentation of this file.

```
00001 /* Copyright (c) 2002, Marek Michalkiewicz
00002    Copyright (c) 2004,2005,2007 Joerg Wunsch
00003    Copyright (c) 2007  Florin-Viorel Petrov
00004    All rights reserved.
00005
00006    Redistribution and use in source and binary forms, with or without
00007    modification, are permitted provided that the following conditions are met:
00008
00009    * Redistributions of source code must retain the above copyright
00010      notice, this list of conditions and the following disclaimer.
00011
00012    * Redistributions in binary form must reproduce the above copyright
00013      notice, this list of conditions and the following disclaimer in
00014      the documentation and/or other materials provided with the
00015      distribution.
00016
00017    * Neither the name of the copyright holders nor the names of
00018      contributors may be used to endorse or promote products derived
00019      from this software without specific prior written permission.
00020
00021   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
00022   AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
00023   IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
00024   ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
00025   LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
00026   CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
00027   SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
00028   INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
00029   CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
00030   ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
00031   POSSIBILITY OF SUCH DAMAGE. */
00032
00033 /* include/util/delay.h.  Generated from delay.h.in by configure. */
00034
00035 #ifndef _UTIL_DELAY_H_
00036 #define _UTIL_DELAY_H_ 1
00037
00038 #ifndef __DOXYGEN__
00039 #  ifndef __HAS_DELAY_CYCLES
00040 #    define __HAS_DELAY_CYCLES 1
00041 #  endif
00042
00043 #endif  /* __DOXYGEN__ */
00044
00045 #include <stdint.h>
00046 #include <util/delay_basic.h>
00047 #include <bits/attribs.h>
00048
00049 /** \file */
00050 /** \defgroup util_delay <util/delay.h>: Convenience functions for busy-wait delay loops
00051     \code
00052     #define F_CPU 1000000UL  // 1 MHz
00053     //#define F_CPU 14.7456e6
00054     #include <util/delay.h>
00055     \endcode
00056
00057     The functions in this header are meant as convenience functions where
00058     actual time values can be specified as a delay time,
00059     rather than a number of cycles to wait for.
00060
00061     This requires that the clock frequency of the device is provided
00062     in the \c #F_CPU macro in units of Hertz.  The macro must be defined
00063     before including the <util/delay.h> header.  It can be defined
00064     in the source code like indicated above, or it can be defined on the
00065     command line / in a Makefile by means of <tt>-D F_CPU=...</tt>
00066
00067     The functions in this header file are wrappers around the basic
00068     busy-wait functions from <tt>\<util/delay_basic.h\></tt>, or, when
00069     supported by the compiler, then
```

```
00070     <a
      href="https://gcc.gnu.org/onlinedocs/gcc/AVR-Built-in-Functions.html#index-_005f_005fbuiltin_005favr_005
00071     >__builtin_avr_delay_cycles()</a> is used instead.
00072
00073     In any case, the delay functions provided by this header will not disable
00074     interrupts, which means hat the delay time will be longer than
00075     specified when interrupts occur while a delay function is running.
00076
00077     In order for these functions to work as intended, compiler
00078     optimizations <em>must</em> be enabled, and the delay time
00079     <em>must</em> be an expression that is a known constant at
00080     compile-time.  If these requirements are not met, the resulting
00081     delay will be much longer (and basically unpredictable), and
00082     applications that otherwise do not use floating-point calculations
00083     will experience severe code bloat by the floating-point library
00084     routines linked into the application.
00085     The idea is
00086     that compile-time constant expressions will be eliminated by
00087     compiler optimization so floating-point expressions can be used
00088     to calculate the number of delay cycles needed based on the CPU
00089     frequency passed by the macro F_CPU.
00090
00091     The functions available allow the specification of microsecond, and
00092     millisecond delays directly, using the application-supplied macro
00093     F_CPU as the CPU clock frequency in Hertz.
00094 */
00095
00096
00097 #ifndef F_CPU
00098 /* prevent compiler error by supplying a default */
00099 # warning "F_CPU not defined for <util/delay.h>"
00100 /** \ingroup util_delay
00101     \def F_CPU
00102     \brief CPU frequency in Hz
00103
00104     The macro F_CPU specifies the CPU frequency in Hertz to be considered by
00105     the delay functions.  This macro is normally supplied by the
00106     environment (e.g. from within a project header, or the project's
00107     Makefile).  The value 1 MHz here is only provided as a
00108     fallback default if no such user-provided definition could be found.
00109
00110     In terms of the delay functions, the CPU frequency can be given as
00111     a floating-point constant (e.g. 3.6864e6 for 3.6864 MHz).
00112     However, the macros in <util/setbaud.h> require it to be an
00113     integer value.
00114  */
00115 # define F_CPU 1000000UL
00116 #endif
00117
00118 #ifndef __OPTIMIZE__
00119 # warning "Compiler optimizations disabled; functions from <util/delay.h> won't work as
      designed"
00120 #endif
00121
00122 /**
00123    \ingroup util_delay
00124
00125    Perform a delay of \c __ms milliseconds.
00126
00127    The macro #F_CPU is supposed to be defined to a
00128    constant defining the CPU clock frequency in Hertz.
00129
00130 -  If \c __builtin_avr_delay_cycles() is
00131 \if GIT_BRANCH_IS_ONLINEDOCS
00132    supported
00133 \else
00134    \ref faq_toolchain_support "supported"
00135 \endif
00136    by the compiler, then
00137    the maximal possible delay is 4294967.04 / f<sub>CPU</sub> milliseconds
00138    where f<sub>CPU</sub> denotes the CPU frequency in units of 1 MHz.
00139    This is around 71 minutes / f<sub>CPU</sub>.
00140    Values greater than that are saturated to this value.
```

```
00141
00142 -  Otherwise, _delay_loop_2() is used as a fallback, and the maximal
00143    possible delay is 262.14 / f<sub>CPU</sub> milliseconds.
00144    When the user requests a delay which exceeds the maximum possible one,
00145    _delay_ms() provides a decreased resolution functionality. In this
00146    mode _delay_ms() will work with a resolution of 1/10 ms, providing
00147    delays up to 6.5535 seconds (independent from CPU frequency).  The
00148    user will not be informed about decreased resolution.
00149 .
00150
00151    Conversion of \c __ms into clock cycles may not always result in
00152    an integral value.  By default, the clock cycles are rounded up to the next
00153    integer. This ensures that the user gets at least \c __ms
00154    microseconds of delay.
00155    Alternatively, by defining the macro \c __DELAY_ROUND_DOWN__, or
00156    \c __DELAY_ROUND_CLOSEST__, before including this header file, the
00157    algorithm can be made to round down, or round to closest integer,
00158    respectively.
00159
00160    \note The implementation of _delay_ms() based on
00161    \c __builtin_avr_delay_cycles() is not backward compatible with older
00162    implementations.  In order to get a functionality backward compatible
00163    with previous versions, the macro \c __DELAY_BACKWARD_COMPATIBLE__
00164    must be defined before including this header file.
00165  */
00166 static __ATTR_ALWAYS_INLINE__ void
00167 _delay_ms(double __ms)
00168 {
00169 #if (__HAS_DELAY_CYCLES \
00170      && defined(__OPTIMIZE__) \
00171      && !defined(__DELAY_BACKWARD_COMPATIBLE__))
00172     uint32_t __ticks_dc;
00173     double __tmp = ((F_CPU) / 1e3) * __ms;
00174
00175     #if defined(__DELAY_ROUND_DOWN__)
00176         __tmp = __builtin_fabs (__tmp);
00177     #elif defined(__DELAY_ROUND_CLOSEST__)
00178         __tmp = __builtin_fabs (__tmp) + 0.5;
00179     #else
00180         /* Round up by default */
00181         __tmp = __builtin_ceil (__builtin_fabs (__tmp));
00182     #endif
00183
00184     /* Saturate.  Notice that the largest representable float
00185        in this context is 0xffffff00.   */
00186     __ticks_dc = __tmp >= 4294967040.0 ? 0xffffff00 : (uint32_t) __tmp;
00187     __builtin_avr_delay_cycles(__ticks_dc);
00188 #else
00189     uint16_t __ticks;
00190     double __tmp = ((F_CPU) / 4e3) * __ms;
00191     if (__tmp < 1.0)
00192         __ticks = 1;
00193     else if (__tmp > 65535)
00194     {
00195         //  __ticks = requested delay in 1/10 ms
00196         __ticks = (uint16_t) (__ms * 10.0);
00197         while(__ticks)
00198         {
00199             // wait 1/10 ms
00200             _delay_loop_2(((F_CPU) / 4e3) / 10);
00201             __ticks--;
00202         }
00203         return;
00204     }
00205     else
00206         __ticks = (uint16_t)__tmp;
00207     _delay_loop_2(__ticks);
00208 #endif
00209 }
00210
00211 /**
00212    \ingroup util_delay
00213
```

```
00214    Perform a delay of \c __us microseconds.
00215
00216    The macro #F_CPU is supposed to be defined to a
00217    constant defining the CPU clock frequency in Hertz.
00218
00219 -  If \c __builtin_avr_delay_cycles() is
00220 \if GIT_BRANCH_IS_ONLINEDOCS
00221    supported
00222 \else
00223    \ref faq_toolchain_support "supported"
00224 \endif
00225    by the compiler, then
00226    the maximal possible delay is 4294967040 / f<sub>CPU</sub> microseconds
00227    where f<sub>CPU</sub> denotes the CPU frequency in units of 1 MHz.
00228    This is around 71 minutes / f<sub>CPU</sub>.
00229    Values greater than that are saturated to this value.
00230
00231 -  Otherwise, _delay_loop_1() is used as a fallback, and the maximal
00232    possible delay is 768 / f<sub>CPU</sub> microseconds.
00233    If the user requests a delay greater than the maximal possible one,
00234    _delay_us() will automatically call _delay_ms() instead.  The user
00235    will not be informed about this case.
00236 .
00237
00238    Conversion of \c __us into clock cycles may not always result in an
00239    integral value.  By default, the clock cycles are rounded up to next
00240    integer. This ensures that the user gets at least \c __us
00241    microseconds of delay.
00242    Alternatively, by defining the macro \c __DELAY_ROUND_DOWN__, or
00243    \c __DELAY_ROUND_CLOSEST__, before including this header file, the
00244    algorithm can be made to round down, or round to closest integer,
00245    respectively.
00246
00247    \note The implementation of _delay_us() based on
00248    \c __builtin_avr_delay_cycles() is not backward compatible with older
00249    implementations.  In order to get a functionality backward compatible
00250    with previous versions, the macro \c __DELAY_BACKWARD_COMPATIBLE__
00251    must be defined before including this header file.
00252  */
00253 static __ATTR_ALWAYS_INLINE__ void
00254 _delay_us(double __us)
00255 {
00256 #if (__HAS_DELAY_CYCLES \
00257      && defined(__OPTIMIZE__) \
00258      && !defined(__DELAY_BACKWARD_COMPATIBLE__))
00259     uint32_t __ticks_dc;
00260     double __tmp = ((F_CPU) / 1e6) * __us;
00261
00262     #if defined(__DELAY_ROUND_DOWN__)
00263         __tmp = __builtin_fabs (__tmp);
00264     #elif defined(__DELAY_ROUND_CLOSEST__)
00265         __tmp = __builtin_fabs (__tmp) + 0.5;
00266     #else
00267         /* Round up by default */
00268         __tmp = __builtin_ceil (__builtin_fabs (__tmp));
00269     #endif
00270
00271     /* Saturate.  Notice that the largest representable float
00272        in this context is 0xffffff00.   */
00273     __ticks_dc = __tmp >= 4294967040.0 ? 0xffffff00 : (uint32_t) __tmp;
00274     __builtin_avr_delay_cycles(__ticks_dc);
00275 #else
00276     uint8_t __ticks;
00277     double __tmp = ((F_CPU) / 3e6) * __us;
00278     double __tmp2 = ((F_CPU) / 4e6) * __us;
00279     if (__tmp < 1.0)
00280         __ticks = 1;
00281     else if (__tmp2 > 65535)
00282     {
00283         _delay_ms(__us / 1000.0);
00284         return;
00285     }
00286     else if (__tmp > 255)
```

```
00287     {
00288         uint16_t __ticks=(uint16_t)__tmp2;
00289         _delay_loop_2(__ticks);
00290         return;
00291     }
00292     else
00293         __ticks = (uint8_t)__tmp;
00294     _delay_loop_1(__ticks);
00295 #endif
00296 }
00297
00298 #endif /* _UTIL_DELAY_H_ */
```

22.10 project.h

```
00001 /*
00002  * ----------------------------------------------------------------------------
00003  * "THE BEER-WARE LICENSE" (Revision 42):
00004  * Joerg Wunsch wrote this file.  As long as you retain this notice you
00005  * can do whatever you want with this stuff. If we meet some day, and you think
00006  * this stuff is worth it, you can buy me a beer in return.        Joerg Wunsch
00007  * ----------------------------------------------------------------------------
00008  *
00009  * Demo combining C and assembly source files.
00010  */
00011
00012 /*
00013  * Global register variables.
00014  */
00015 #ifdef __ASSEMBLER__
00016
00017 #  define sreg_save r2
00018 #  define flags     r16
00019 #  define counter_hi    r4
00020
00021 #else  /* !ASSEMBLER */
00022
00023 #include <stdint.h>
00024
00025 register uint8_t sreg_save asm("r2");
00026 register uint8_t flags     asm("r16");
00027 register uint8_t counter_hi asm("r4");
00028
00029 #endif /* ASSEMBLER */
```

22.11 defines.h

```
00001 /*
00002  * ----------------------------------------------------------------------------
00003  * "THE BEER-WARE LICENSE" (Revision 42):
00004  * <joerg@FreeBSD.ORG> wrote this file.  As long as you retain this notice you
00005  * can do whatever you want with this stuff. If we meet some day, and you think
00006  * this stuff is worth it, you can buy me a beer in return.        Joerg Wunsch
00007  * ----------------------------------------------------------------------------
00008  *
00009  * General stdiodemo defines
00010  */
00011
00012 /* CPU frequency */
00013 #define F_CPU 16000000UL
00014
00015 /* UART baud rate */
00016 #define UART_BAUD  9600
00017
00018 /* HD44780 LCD port connections */
00019 #define HD44780_RW B, 0 // D8 on Arduino Nano
00020 #define HD44780_RS D, 7 // D7 on Arduino Nano
00021
00022 #define HD44780_E  D, 6 // D6 on Arduino Nano
```

```
00023 /* The data bits D4-D7 have to be not only in ascending order but also
00024    consecutive. The LCD is operated in 4-bit mode, so D0-D4 remain
00025    unconnected. */
00026 #define HD44780_D4 D, 2 // D2 through D5 on Arduino Nano
00027
00028 /* Whether to read the busy flag, or fall back to
00029    worst-time delays. */
00030 #define USE_BUSY_BIT 1
00031
00032 /* Arduino Nano onboard LED */
00033 #define LED_PORT PORTB
00034 #define LED_DDR  DDRB
00035 #define LED_PIN  _BV(5)
```

22.12 hd44780.h

```
00001 /*
00002  * ----------------------------------------------------------------------------
00003  * "THE BEER-WARE LICENSE" (Revision 42):
00004  * <joerg@FreeBSD.ORG> wrote this file.  As long as you retain this notice you
00005  * can do whatever you want with this stuff. If we meet some day, and you think
00006  * this stuff is worth it, you can buy me a beer in return.        Joerg Wunsch
00007  * ----------------------------------------------------------------------------
00008  *
00009  * HD44780 LCD display driver
00010  */
00011
00012 /*
00013  * Send byte b to the LCD.  rs is the RS signal (register select), 0
00014  * selects instruction register, 1 selects the data register.
00015  */
00016 void    hd44780_outbyte(uint8_t b, uint8_t rs);
00017
00018 /*
00019  * Read one byte from the LCD controller.  rs is the RS signal, 0
00020  * selects busy flag (bit 7) and address counter, 1 selects the data
00021  * register.
00022  */
00023 uint8_t hd44780_inbyte(uint8_t rs);
00024
00025 /*
00026  * Wait for the busy flag to clear.
00027  */
00028 void    hd44780_wait_ready(bool islong);
00029
00030 /*
00031  * Initialize the LCD controller hardware.
00032  */
00033 void    hd44780_init(void);
00034
00035 /*
00036  * Prepare the LCD controller pins for powerdown.
00037  */
00038 void    hd44780_powerdown(void);
00039
00040
00041 /* Send a command to the LCD controller. */
00042 #define hd44780_outcmd(n)       hd44780_outbyte((n), 0)
00043
00044 /* Send a data byte to the LCD controller. */
00045 #define hd44780_outdata(n)      hd44780_outbyte((n), 1)
00046
00047 /* Read the address counter and busy flag from the LCD. */
00048 #define hd44780_incmd()         hd44780_inbyte(0)
00049
00050 /* Read the current data byte from the LCD. */
00051 #define hd44780_indata()        hd44780_inbyte(1)
00052
00053
00054 /* Clear LCD display command. */
00055 #define HD44780_CLR \
```

```
00056         0x01
00057
00058 /* Home cursor command. */
00059 #define HD44780_HOME \
00060         0x02
00061
00062 /*
00063  * Select the entry mode.  inc determines whether the address counter
00064  * auto-increments, shift selects an automatic display shift.
00065  */
00066 #define HD44780_ENTMODE(inc, shift) \
00067         (0x04 | ((inc)? 0x02: 0) | ((shift)? 1: 0))
00068
00069 /*
00070  * Selects disp[lay] on/off, cursor on/off, cursor blink[ing]
00071  * on/off.
00072  */
00073 #define HD44780_DISPCTL(disp, cursor, blink) \
00074         (0x08 | ((disp)? 0x04: 0) | ((cursor)? 0x02: 0) | ((blink)? 1: 0))
00075
00076 /*
00077  * With shift = 1, shift display right or left.
00078  * With shift = 0, move cursor right or left.
00079  */
00080 #define HD44780_SHIFT(shift, right) \
00081         (0x10 | ((shift)? 0x08: 0) | ((right)? 0x04: 0))
00082
00083 /*
00084  * Function set.  if8bit selects an 8-bit data path, twoline arranges
00085  * for a two-line display, font5x10 selects the 5x10 dot font (5x8
00086  * dots if clear).
00087  */
00088 #define HD44780_FNSET(if8bit, twoline, font5x10) \
00089         (0x20 | ((if8bit)? 0x10: 0) | ((twoline)? 0x08: 0) | \
00090                 ((font5x10)? 0x04: 0))
00091
00092 /*
00093  * Set the next character generator address to addr.
00094  */
00095 #define HD44780_CGADDR(addr) \
00096         (0x40 | ((addr) & 0x3f))
00097
00098 /*
00099  * Set the next display address to addr.
00100  */
00101 #define HD44780_DDADDR(addr) \
00102         (0x80 | ((addr) & 0x7f))
```

22.13 lcd.h

```
00001 /*
00002  * ----------------------------------------------------------------------------
00003  * "THE BEER-WARE LICENSE" (Revision 42):
00004  * <joerg@FreeBSD.ORG> wrote this file.  As long as you retain this notice you
00005  * can do whatever you want with this stuff. If we meet some day, and you think
00006  * this stuff is worth it, you can buy me a beer in return.        Joerg Wunsch
00007  * ----------------------------------------------------------------------------
00008  *
00009  * Stdio demo, upper layer of LCD driver.
00010  */
00011
00012 /*
00013  * Initialize LCD controller.  Performs a software reset.
00014  */
00015 void    lcd_init(void);
00016
00017 /*
00018  * Send one character to the LCD.
00019  */
00020 int     lcd_putchar(char c, FILE *stream);
```

22.14 uart.h

```
00001 /*
00002  * ----------------------------------------------------------------------------
00003  * "THE BEER-WARE LICENSE" (Revision 42):
00004  * <joerg@FreeBSD.ORG> wrote this file.  As long as you retain this notice you
00005  * can do whatever you want with this stuff. If we meet some day, and you think
00006  * this stuff is worth it, you can buy me a beer in return.        Joerg Wunsch
00007  * ----------------------------------------------------------------------------
00008  *
00009  * Stdio demo, UART declarations
00010  */
00011
00012 /*
00013  * Perform UART startup initialization.
00014  */
00015 void    uart_init(void);
00016
00017 /*
00018  * Send one character to the UART.
00019  */
00020 int     uart_putchar(char c, FILE *stream);
00021
00022 /*
00023  * Size of internal line buffer used by uart_getchar().
00024  */
00025 #define RX_BUFSIZE 80
00026
00027 /*
00028  * Receive one character from the UART.  The actual reception is
00029  * line-buffered, and one character is returned from the buffer at
00030  * each invocation.
00031  */
00032 int     uart_getchar(FILE *stream);
```

22.15 alloca.h

```
00001 /* Copyright (c) 2007, Dmitry Xmelkov
00002    All rights reserved.
00003
00004    Redistribution and use in source and binary forms, with or without
00005    modification, are permitted provided that the following conditions are met:
00006
00007    * Redistributions of source code must retain the above copyright
00008      notice, this list of conditions and the following disclaimer.
00009    * Redistributions in binary form must reproduce the above copyright
00010      notice, this list of conditions and the following disclaimer in
00011      the documentation and/or other materials provided with the
00012      distribution.
00013    * Neither the name of the copyright holders nor the names of
00014      contributors may be used to endorse or promote products derived
00015      from this software without specific prior written permission.
00016
00017   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
00018   AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
00019   IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
00020   ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
00021   LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
00022   CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
00023   SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
00024   INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
00025   CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
00026   ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
00027   POSSIBILITY OF SUCH DAMAGE. */
00028
00029 #ifndef _ALLOCA_H
00030 #define _ALLOCA_H  1
00031
00032 #include <stddef.h>
00033
00034 /** \defgroup alloca <alloca.h>: Allocate space in the stack
```

```
00035     \code #include <alloca.h> \endcode */
00036
00037 /** \ingroup alloca
00038     \brief Allocate \a __size bytes of space in the stack frame of the caller.
00039
00040     This temporary space is automatically freed when the function that
00041     called alloca() returns to its caller. AVR-LibC defines the alloca() as
00042     a macro, which is translated into the inlined \c __builtin_alloca()
00043     function. The fact that the code is inlined, means that it is impossible
00044     to take the address of this function, or to change its behaviour by
00045     linking with a different library.
00046
00047     \return alloca() returns a pointer to the beginning of the allocated
00048     space. If the allocation causes stack overflow, program behaviour is
00049     undefined.
00050
00051     \warning Avoid use alloca() inside the list of arguments of a function
00052     call.
00053  */
00054 extern void *alloca (size_t __size);
00055
00056 #define alloca(size)    __builtin_alloca (size)
00057
00058 #endif  /* alloca.h */
```

22.16 assert.h File Reference

Macros

- #define assert(expression)

22.17 assert.h

Go to the documentation of this file.

```
00001 /* Copyright (c) 2005,2007 Joerg Wunsch
00002    All rights reserved.
00003
00004    Portions of documentation Copyright (c) 1991, 1993
00005    The Regents of the University of California.
00006
00007    All rights reserved.
00008
00009    Redistribution and use in source and binary forms, with or without
00010    modification, are permitted provided that the following conditions are met:
00011
00012    * Redistributions of source code must retain the above copyright
00013      notice, this list of conditions and the following disclaimer.
00014
00015    * Redistributions in binary form must reproduce the above copyright
00016      notice, this list of conditions and the following disclaimer in
00017      the documentation and/or other materials provided with the
00018      distribution.
00019
00020    * Neither the name of the copyright holders nor the names of
00021      contributors may be used to endorse or promote products derived
00022      from this software without specific prior written permission.
00023
00024   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
00025   AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
00026   IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
00027   ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
00028   LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
00029   CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
00030   SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
00031   INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
00032   CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
```

```
00033   ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
00034   POSSIBILITY OF SUCH DAMAGE. */
00035
00036 /** \file */
00037 /** \defgroup avr_assert <assert.h>: Diagnostics
00038     \code #include <assert.h> \endcode
00039
00040     This header file defines a debugging aid.
00041
00042     As there is no standard error output stream available for many
00043     applications using this library, the generation of a printable
00044     error message is not enabled by default.  These messages will
00045     only be generated if the application defines the macro
00046
00047     \code __ASSERT_USE_STDERR \endcode
00048
00049     before including the \c <assert.h> header file.  By default,
00050     only abort() will be called to halt the application.
00051 */
00052
00053 /**@{*/
00054
00055 /*
00056  * The ability to include this file (with or without NDEBUG) is a
00057  * feature.
00058  */
00059
00060 #undef assert
00061
00062 #include <stdlib.h>
00063
00064 #if defined(__DOXYGEN__)
00065 /**
00066    \def assert
00067    \param expression Expression to test for.
00068
00069    The assert() macro tests the given expression and if it is false,
00070    the calling process is terminated by calling abort().
00071    When the macro \c __ASSERT_USE_STDERR was defined prior to including
00072    \c <assert.h>, then a diagnostic message is written to \c stderr.
00073
00074    If expression is true, the assert() macro does nothing.
00075
00076    The assert() macro may be removed at compile time by defining
00077    NDEBUG as a macro (e.g., by using the compiler option -DNDEBUG).
00078 */
00079 #  define assert(expression)
00080
00081 #else /* !DOXYGEN */
00082
00083 #  if defined(NDEBUG)
00084 #    define assert(e)   ((void)0)
00085 #  else /* !NDEBUG */
00086 #    if defined(__ASSERT_USE_STDERR)
00087 #      define assert(e) ((e) ? (void)0 : \
00088                          __assert(__func__, __FILE__, __LINE__, #e))
00089 #    else /* !__ASSERT_USE_STDERR */
00090 #      define assert(e) ((e) ? (void)0 : abort())
00091 #    endif /* __ASSERT_USE_STDERR */
00092 #  endif /* NDEBUG */
00093 #endif /* DOXYGEN */
00094
00095 #if (defined __STDC_VERSION__ && __STDC_VERSION__ >= 201112L) || \
00096     ((_GNUC_ > 4 || (_GNUC_ == 4 && _GNUC_MINOR_ >= 6)) && !defined __cplusplus)
00097 #  undef static_assert
00098 #  define static_assert _Static_assert
00099 #endif
00100
00101 #ifdef __cplusplus
00102 extern "C" {
00103 #endif
00104
00105 #if !defined(__DOXYGEN__)
```

```
00106
00107 extern void __assert(const char *__func, const char *__file,
00108              int __lineno, const char *__sexp);
00109
00110 #endif /* not __DOXYGEN__ */
00111
00112 #ifdef __cplusplus
00113 }
00114 #endif
00115
00116 /**@}*/
00117 /* EOF */
```

22.18 boot.h File Reference

Macros

- #define BOOTLOADER_SECTION __attribute__ ((__section__(".bootloader")))
- #define boot_spm_interrupt_enable() (__SPM_REG |= (uint8_t)_BV(SPMIE))
- #define boot_spm_interrupt_disable() (__SPM_REG &= (uint8_t)~_BV(SPMIE))
- #define boot_is_spm_interrupt() (__SPM_REG & (uint8_t)_BV(SPMIE))
- #define boot_rww_busy() (__SPM_REG & (uint8_t)_BV(__COMMON_ASB))
- #define boot_spm_busy() (__SPM_REG & (uint8_t)_BV(__SPM_ENABLE))
- #define boot_spm_busy_wait() do{}while(boot_spm_busy())
- #define GET_LOW_FUSE_BITS (0x0000)
- #define GET_LOCK_BITS (0x0001)
- #define GET_EXTENDED_FUSE_BITS (0x0002)
- #define GET_HIGH_FUSE_BITS (0x0003)
- #define boot_lock_fuse_bits_get(address)
- #define boot_signature_byte_get(addr)
- #define boot_page_fill(address, data) __boot_page_fill_normal(address, data)
- #define boot_page_erase(address) __boot_page_erase_normal(address)
- #define boot_page_write(address) __boot_page_write_normal(address)
- #define boot_rww_enable() __boot_rww_enable()
- #define boot_lock_bits_set(lock_bits) __boot_lock_bits_set(lock_bits)
- #define boot_page_fill_safe(address, data)
- #define boot_page_erase_safe(address)
- #define boot_page_write_safe(address)
- #define boot_rww_enable_safe()
- #define boot_lock_bits_set_safe(lock_bits)

22.19 boot.h

Go to the documentation of this file.

```
00001 /* Copyright (c) 2002,2003,2004,2005,2006,2007,2008,2009  Eric B. Weddington
00002    All rights reserved.
00003
00004    Redistribution and use in source and binary forms, with or without
00005    modification, are permitted provided that the following conditions are met:
00006
00007    * Redistributions of source code must retain the above copyright
00008      notice, this list of conditions and the following disclaimer.
00009    * Redistributions in binary form must reproduce the above copyright
00010      notice, this list of conditions and the following disclaimer in
00011      the documentation and/or other materials provided with the
00012      distribution.
00013    * Neither the name of the copyright holders nor the names of
00014      contributors may be used to endorse or promote products derived
```

```
00015      from this software without specific prior written permission.
00016
00017   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
00018   AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
00019   IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
00020   ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
00021   LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
00022   CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
00023   SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
00024   INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
00025   CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
00026   ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
00027   POSSIBILITY OF SUCH DAMAGE. */
00028
00029 #ifndef _AVR_BOOT_H_
00030 #define _AVR_BOOT_H_    1
00031
00032 /** \file */
00033 /** \defgroup avr_boot <avr/boot.h>: Bootloader Support Utilities
00034     \code
00035     #include <avr/io.h>
00036     #include <avr/boot.h>
00037     \endcode
00038
00039     The macros in this module provide a C language interface to the
00040     bootloader support functionality of certain AVR processors. These
00041     macros are designed to work with all sizes of flash memory.
00042
00043     Global interrupts are not automatically disabled for these macros. It
00044     is left up to the programmer to do this. See the code example below.
00045     Also see the processor datasheet for caveats on having global interrupts
00046     enabled during writing of the Flash.
00047
00048     \note Not all AVR processors provide bootloader support. See your
00049     processor datasheet to see if it provides bootloader support.
00050
00051     \par API Usage Example
00052     The following code shows typical usage of the boot API.
00053
00054     \code
00055     #include <stdint.h>
00056     #include <avr/interrupt.h>
00057     #include <avr/eeprom.h>
00058     #include <avr/pgmspace.h>
00059
00060     void boot_program_page (uint32_t page, uint8_t *buf)
00061     {
00062         // Disable interrupts.
00063         uint8_t sreg = SREG;
00064         cli();
00065
00066         eeprom_busy_wait ();
00067
00068         boot_page_erase (page);
00069         boot_spm_busy_wait ();      // Wait until the memory is erased.
00070
00071         for (uint16_t i = 0; i < SPM_PAGESIZE; i += 2)
00072         {
00073             // Set up little-endian word.
00074             uint16_t w = *buf++;
00075             w += (*buf++) « 8;
00076
00077             boot_page_fill (page + i, w);
00078         }
00079
00080         boot_page_write (page);     // Store buffer in flash page.
00081         boot_spm_busy_wait();       // Wait until the memory is written.
00082
00083         // Reenable RWW-section again. We need this if we want to jump back
00084         // to the application after bootloading.
00085         boot_rww_enable ();
00086
00087         // Re-enable interrupts (if they were ever enabled).
```

```
00088         SREG = sreg;
00089     }\endcode */
00090
00091 #include <avr/eeprom.h>
00092 #include <avr/io.h>
00093 #include <inttypes.h>
00094 #include <limits.h>
00095
00096 /* Check for SPM Control Register in processor. */
00097 #if defined (SPMCSR)
00098 #  define __SPM_REG    SPMCSR
00099 #else
00100 #  if defined (SPMCR)
00101 #    define __SPM_REG    SPMCR
00102 #  else
00103 #    error AVR processor does not provide bootloader support!
00104 #  endif
00105 #endif
00106
00107
00108 /* Check for SPM Enable bit. */
00109 #if defined(SPMEN)
00110 #  define __SPM_ENABLE  SPMEN
00111 #elif defined(SELFPRGEN)
00112 #  define __SPM_ENABLE  SELFPRGEN
00113 #else
00114 #  error Cannot find SPM Enable bit definition!
00115 #endif
00116
00117 /** \ingroup avr_boot
00118     \def BOOTLOADER_SECTION
00119
00120     Used to declare a function or variable to be placed into a
00121     new section called .bootloader. This section and its contents
00122     can then be relocated to any address (such as the bootloader
00123     NRWW area) at link-time. */
00124
00125 #define BOOTLOADER_SECTION    __attribute__ ((__section__(".bootloader")))
00126
00127 #ifndef __DOXYGEN__
00128 /* Create common bit definitions. */
00129 #ifdef ASB
00130 #define __COMMON_ASB    ASB
00131 #else
00132 #define __COMMON_ASB    RWWSB
00133 #endif
00134
00135 #ifdef ASRE
00136 #define __COMMON_ASRE   ASRE
00137 #else
00138 #define __COMMON_ASRE   RWWSRE
00139 #endif
00140
00141 /* Define the bit positions of the Boot Lock Bits. */
00142
00143 #define BLB12           5
00144 #define BLB11           4
00145 #define BLB02           3
00146 #define BLB01           2
00147 #endif  /* __DOXYGEN__ */
00148
00149 /** \ingroup avr_boot
00150     \def boot_spm_interrupt_enable()
00151     Enable the SPM interrupt. */
00152
00153 #define boot_spm_interrupt_enable()    (__SPM_REG |= (uint8_t)_BV(SPMIE))
00154
00155 /** \ingroup avr_boot
00156     \def boot_spm_interrupt_disable()
00157     Disable the SPM interrupt. */
00158
00159 #define boot_spm_interrupt_disable()   (__SPM_REG &= (uint8_t)~_BV(SPMIE))
00160
```

```
00161 /** \ingroup avr_boot
00162     \def boot_is_spm_interrupt()
00163     Check if the SPM interrupt is enabled. */
00164
00165 #define boot_is_spm_interrupt()       (__SPM_REG & (uint8_t)_BV(SPMIE))
00166
00167 /** \ingroup avr_boot
00168     \def boot_rww_busy()
00169     Check if the RWW section is busy. */
00170
00171 #define boot_rww_busy()          (__SPM_REG & (uint8_t)_BV(__COMMON_ASB))
00172
00173 /** \ingroup avr_boot
00174     \def boot_spm_busy()
00175     Check if the SPM instruction is busy. */
00176
00177 #define boot_spm_busy()               (__SPM_REG & (uint8_t)_BV(__SPM_ENABLE))
00178
00179 /** \ingroup avr_boot
00180     \def boot_spm_busy_wait()
00181     Wait while the SPM instruction is busy. */
00182
00183 #define boot_spm_busy_wait()          do{}while(boot_spm_busy())
00184
00185 #ifndef __DOXYGEN__
00186 #define __BOOT_PAGE_ERASE         (_BV(__SPM_ENABLE) | _BV(PGERS))
00187 #define __BOOT_PAGE_WRITE         (_BV(__SPM_ENABLE) | _BV(PGWRT))
00188 #define __BOOT_PAGE_FILL          _BV(__SPM_ENABLE)
00189 #define __BOOT_RWW_ENABLE         (_BV(__SPM_ENABLE) | _BV(__COMMON_ASRE))
00190 #if defined(BLBSET)
00191 #define __BOOT_LOCK_BITS_SET      (_BV(__SPM_ENABLE) | _BV(BLBSET))
00192 #elif defined(RFLB)  /* Some devices have RFLB defined instead of BLBSET. */
00193 #define __BOOT_LOCK_BITS_SET      (_BV(__SPM_ENABLE) | _BV(RFLB))
00194 #elif defined(RWFLB)  /* Some devices have RWFLB defined instead of BLBSET. */
00195 #define __BOOT_LOCK_BITS_SET      (_BV(__SPM_ENABLE) | _BV(RWFLB))
00196 #endif
00197
00198 #define __boot_page_fill_normal(address, data)   \
00199 (__extension__({                                 \
00200     if (_SFR_IO_REG_P(__SPM_REG))                \
00201       __asm__ __volatile__ (                     \
00202         "movw r0, %3"     "\n\t"                 \
00203         "out  %0, %1"     "\n\t"                 \
00204         "spm"             "\n\t"                 \
00205         "clr  __zero_reg__"                      \
00206         :                                        \
00207         : "i" (_SFR_IO_ADDR(__SPM_REG)),         \
00208           "r" ((uint8_t)(__BOOT_PAGE_FILL)),     \
00209           "z" ((uint16_t)(address)),             \
00210           "r" ((uint16_t)(data))                 \
00211         : "r0");                                 \
00212     else                                         \
00213       __asm__ __volatile__ (                     \
00214         "movw r0, %3"      "\n\t"                \
00215         "sts  %0, %1"      "\n\t"                \
00216         "spm"              "\n\t"                \
00217         "clr  __zero_reg__"                      \
00218         :                                        \
00219         : "i" (_SFR_MEM_ADDR(__SPM_REG)),        \
00220           "r" ((uint8_t)(__BOOT_PAGE_FILL)),     \
00221           "z" ((uint16_t)(address)),             \
00222           "r" ((uint16_t)(data))                 \
00223         : "r0");                                 \
00224 }))
00225
00226 #define __boot_page_fill_alternate(address, data)\
00227 (__extension__({                                 \
00228     __asm__ __volatile__                         \
00229     (                                            \
00230         "movw r0, %3"     "\n\t"                 \
00231         "sts  %0, %1"     "\n\t"                 \
00232         "spm"             "\n\t"                 \
00233         ".word 0xffff"    "\n\t"                 \
```

```
00234         "nop"             "\n\t"                \
00235         "clr  __zero_reg__"                     \
00236         :                                       \
00237         : "i" (_SFR_MEM_ADDR(__SPM_REG)),       \
00238           "r" ((uint8_t)(__BOOT_PAGE_FILL)),    \
00239           "z" ((uint16_t)(address)),            \
00240           "r" ((uint16_t)(data))                \
00241         : "r0"                                  \
00242     );                                          \
00243 }))
00244
00245 #define __boot_page_fill_extended(address, data) \
00246 (__extension__({                                \
00247     __asm__ __volatile__                        \
00248     (                                           \
00249         "movw r0, %4"      "\n\t"               \
00250         "movw r30, %A3"    "\n\t"               \
00251         "out  %1, %C3"     "\n\t"               \
00252         "sts  %0, %2"      "\n\t"               \
00253         "spm"              "\n\t"               \
00254         "clr __zero_reg__"                      \
00255         :                                       \
00256         : "i" (_SFR_MEM_ADDR(__SPM_REG)),       \
00257           "i" (_SFR_IO_ADDR(RAMPZ)),            \
00258           "r" ((uint8_t)(__BOOT_PAGE_FILL)),    \
00259           "r" ((uint32_t)(address)),            \
00260           "r" ((uint16_t)(data))                \
00261         : "r0", "r30", "r31"                    \
00262     );                                          \
00263 }))
00264
00265 #define __boot_page_erase_normal(address)       \
00266 (__extension__({                                \
00267     if (_SFR_IO_REG_P(__SPM_REG))               \
00268       __asm__ __volatile__ (                    \
00269         "out %0, %1"       "\n\t"               \
00270         "spm"                                   \
00271         :                                       \
00272         : "i" (_SFR_IO_ADDR(__SPM_REG)),        \
00273           "r" ((uint8_t)(__BOOT_PAGE_ERASE)),   \
00274           "z" ((uint16_t)(address)));           \
00275     else                                        \
00276       __asm__ __volatile__ (                    \
00277         "sts %0, %1"       "\n\t"               \
00278         "spm"                                   \
00279         :                                       \
00280         : "i" (_SFR_MEM_ADDR(__SPM_REG)),       \
00281           "r" ((uint8_t)(__BOOT_PAGE_ERASE)),   \
00282           "z" ((uint16_t)(address)));           \
00283 }))
00284
00285 #define __boot_page_erase_alternate(address)    \
00286 (__extension__({                                \
00287     __asm__ __volatile__                        \
00288     (                                           \
00289         "sts %0, %1"       "\n\t"               \
00290         "spm"              "\n\t"               \
00291         ".word 0xffff"     "\n\t"               \
00292         "nop"                                   \
00293         :                                       \
00294         : "i" (_SFR_MEM_ADDR(__SPM_REG)),       \
00295           "r" ((uint8_t)(__BOOT_PAGE_ERASE)),   \
00296           "z" ((uint16_t)(address))             \
00297     );                                          \
00298 }))
00299
00300 #define __boot_page_erase_extended(address)     \
00301 (__extension__({                                \
00302     __asm__ __volatile__                        \
00303     (                                           \
00304         "movw r30, %A3"    "\n\t"               \
00305         "out  %1, %C3"     "\n\t"               \
00306         "sts  %0, %2"      "\n\t"               \
```

```
00307         "spm"                                   \
00308         :                                       \
00309         : "i" (_SFR_MEM_ADDR(__SPM_REG)),       \
00310           "i" (_SFR_IO_ADDR(RAMPZ)),            \
00311           "r" ((uint8_t)(__BOOT_PAGE_ERASE)),   \
00312           "r" ((uint32_t)(address))             \
00313         : "r30", "r31"                          \
00314     );                                          \
00315 }))
00316
00317 #define __boot_page_write_normal(address)       \
00318 (__extension__({                                \
00319     if (_SFR_IO_REG_P(__SPM_REG))               \
00320       __asm__ __volatile__ (                    \
00321         "out %0, %1"        "\n\t"              \
00322         "spm"                                   \
00323         :                                       \
00324         : "i" (_SFR_IO_ADDR(__SPM_REG)),        \
00325           "r" ((uint8_t)(__BOOT_PAGE_WRITE)),   \
00326           "z" ((uint16_t)(address)));           \
00327     else                                        \
00328       __asm__ __volatile__ (                    \
00329         "sts %0, %1"        "\n\t"              \
00330         "spm"                                   \
00331         :                                       \
00332         : "i" (_SFR_MEM_ADDR(__SPM_REG)),       \
00333           "r" ((uint8_t)(__BOOT_PAGE_WRITE)),   \
00334           "z" ((uint16_t)(address)));           \
00335 }))
00336
00337 #define __boot_page_write_alternate(address)    \
00338 (__extension__({                                \
00339     __asm__ __volatile__                        \
00340     (                                           \
00341         "sts %0, %1"     "\n\t"                 \
00342         "spm"            "\n\t"                 \
00343         ".word 0xffff"   "\n\t"                 \
00344         "nop"                                   \
00345         :                                       \
00346         : "i" (_SFR_MEM_ADDR(__SPM_REG)),       \
00347           "r" ((uint8_t)(__BOOT_PAGE_WRITE)),   \
00348           "z" ((uint16_t)(address))             \
00349     );                                          \
00350 }))
00351
00352 #define __boot_page_write_extended(address)     \
00353 (__extension__({                                \
00354     __asm__ __volatile__                        \
00355     (                                           \
00356         "movw r30, %A3"    "\n\t"               \
00357         "out  %1, %C3"     "\n\t"               \
00358         "sts  %0, %2"      "\n\t"               \
00359         "spm"                                   \
00360         :                                       \
00361         : "i" (_SFR_MEM_ADDR(__SPM_REG)),       \
00362           "i" (_SFR_IO_ADDR(RAMPZ)),            \
00363           "r" ((uint8_t)(__BOOT_PAGE_WRITE)),   \
00364           "r" ((uint32_t)(address))             \
00365         : "r30", "r31"                          \
00366     );                                          \
00367 }))
00368
00369 #define __boot_rww_enable()                     \
00370 (__extension__({                                \
00371     __asm__ __volatile__                        \
00372     (                                           \
00373         "sts %0, %1"   "\n\t"                   \
00374         "spm"                                   \
00375         :                                       \
00376         : "i" (_SFR_MEM_ADDR(__SPM_REG)),       \
00377           "r" ((uint8_t)(__BOOT_RWW_ENABLE))    \
00378     );                                          \
00379 }))
```

```
00380
00381 #define __boot_rww_enable_alternate()            \
00382 (__extension__({                                 \
00383     __asm__ __volatile__                         \
00384     (                                            \
00385         "sts %0, %1"     "\n\t"                  \
00386         "spm"            "\n\t"                  \
00387         ".word 0xffff"   "\n\t"                  \
00388         "nop"                                    \
00389         :                                        \
00390         : "i" (_SFR_MEM_ADDR(__SPM_REG)),        \
00391           "r" ((uint8_t)(__BOOT_RWW_ENABLE))     \
00392     );                                           \
00393 }))
00394
00395 /* From the mega16/mega128 data sheets (maybe others):
00396
00397      Bits by SPM To set the Boot Loader Lock bits, write the desired data to
00398      R0, write "X0001001" to SPMCR and execute SPM within four clock cycles
00399      after writing SPMCR. The only accessible Lock bits are the Boot Lock bits
00400      that may prevent the Application and Boot Loader section from any
00401      software update by the MCU.
00402
00403      If bits 5..2 in R0 are cleared (zero), the corresponding Boot Lock bit
00404      will be programmed if an SPM instruction is executed within four cycles
00405      after BLBSET and SPMEN (or SELFPRGEN) are set in SPMCR. The Z-pointer is
00406      don't care during this operation, but for future compatibility it is
00407      recommended to load the Z-pointer with $0001 (same as used for reading the
00408      Lock bits). For future compatibility It is also recommended to set bits 7,
00409      6, 1, and 0 in R0 to 1 when writing the Lock bits. When programming the
00410      Lock bits the entire Flash can be read during the operation. */
00411
00412 #define __boot_lock_bits_set(lock_bits)                    \
00413 (__extension__({                                           \
00414     uint8_t value = (uint8_t)(~(lock_bits));               \
00415     __asm__ __volatile__                                   \
00416     (                                                      \
00417         "ldi r30, 1"     "\n\t"                            \
00418         "ldi r31, 0"     "\n\t"                            \
00419         "mov r0, %2"     "\n\t"                            \
00420         "sts %0, %1"     "\n\t"                            \
00421         "spm"                                              \
00422         :                                                  \
00423         : "i" (_SFR_MEM_ADDR(__SPM_REG)),                  \
00424           "r" ((uint8_t)(__BOOT_LOCK_BITS_SET)),           \
00425           "r" (value)                                      \
00426         : "r0", "r30", "r31"                               \
00427     );                                                     \
00428 }))
00429
00430 #define __boot_lock_bits_set_alternate(lock_bits)          \
00431 (__extension__({                                           \
00432     uint8_t value = (uint8_t)(~(lock_bits));               \
00433     __asm__ __volatile__                                   \
00434     (                                                      \
00435         "ldi r30, 1"     "\n\t"                            \
00436         "ldi r31, 0"     "\n\t"                            \
00437         "mov r0, %2"     "\n\t"                            \
00438         "sts %0, %1"     "\n\t"                            \
00439         "spm"            "\n\t"                            \
00440         ".word 0xffff"   "\n\t"                            \
00441         "nop"                                              \
00442         :                                                  \
00443         : "i" (_SFR_MEM_ADDR(__SPM_REG)),                  \
00444           "r" ((uint8_t)(__BOOT_LOCK_BITS_SET)),           \
00445           "r" (value)                                      \
00446         : "r0", "r30", "r31"                               \
00447     );                                                     \
00448 }))
00449 #endif  /* __DOXYGEN__ */
00450
00451 /*
00452    Reading lock and fuse bits:
```

```
00453
00454      Similarly to writing the lock bits above, set BLBSET and SPMEN (or
00455      SELFPRGEN) bits in __SPMREG, and then (within four clock cycles) issue an
00456      LPM instruction.
00457
00458      Z address:       contents:
00459      0x0000           low fuse bits
00460      0x0001           lock bits
00461      0x0002           extended fuse bits
00462      0x0003           high fuse bits
00463
00464      Sounds confusing, doesn't it?
00465
00466      Unlike the macros in pgmspace.h, no need to care for non-enhanced
00467      cores here as these old cores do not provide SPM support anyway.
00468  */
00469
00470 /** \ingroup avr_boot
00471     \def GET_LOW_FUSE_BITS
00472     address to read the low fuse bits, using boot_lock_fuse_bits_get
00473  */
00474 #define GET_LOW_FUSE_BITS           (0x0000)
00475 /** \ingroup avr_boot
00476     \def GET_LOCK_BITS
00477     address to read the lock bits, using boot_lock_fuse_bits_get
00478  */
00479 #define GET_LOCK_BITS               (0x0001)
00480 /** \ingroup avr_boot
00481     \def GET_EXTENDED_FUSE_BITS
00482     address to read the extended fuse bits, using boot_lock_fuse_bits_get
00483  */
00484 #define GET_EXTENDED_FUSE_BITS      (0x0002)
00485 /** \ingroup avr_boot
00486     \def GET_HIGH_FUSE_BITS
00487     address to read the high fuse bits, using boot_lock_fuse_bits_get
00488  */
00489 #define GET_HIGH_FUSE_BITS          (0x0003)
00490
00491 /** \ingroup avr_boot
00492     \def boot_lock_fuse_bits_get(address)
00493
00494     Read the lock or fuse bits at \c address.
00495
00496     Parameter \c address can be any of GET_LOW_FUSE_BITS,
00497     GET_LOCK_BITS, GET_EXTENDED_FUSE_BITS, or GET_HIGH_FUSE_BITS.
00498
00499     \note The lock and fuse bits returned are the physical values,
00500     i.e. a bit returned as 0 means the corresponding fuse or lock bit
00501     is programmed.
00502  */
00503 #define boot_lock_fuse_bits_get(address)                    \
00504 (__extension__({                                            \
00505     uint8_t __result;                                       \
00506     __asm__ __volatile__                                    \
00507     (                                                       \
00508         "sts %1, %2\n\t"                                    \
00509         "lpm %0, Z\n\t"                                     \
00510         : "=r" (__result)                                   \
00511         : "i" (_SFR_MEM_ADDR(__SPM_REG)),                   \
00512           "r" ((uint8_t)(__BOOT_LOCK_BITS_SET)),            \
00513           "z" ((uint16_t)(address))                         \
00514     );                                                      \
00515     __result;                                               \
00516 }))
00517
00518 #ifndef __DOXYGEN__
00519 #  if defined(SIGRD)
00520 #    define __BOOT_SIGROW_READ (_BV(__SPM_ENABLE) | _BV(SIGRD))
00521 #  elif defined(RSIG)
00522 #    define __BOOT_SIGROW_READ (_BV(__SPM_ENABLE) | _BV(RSIG))
00523 #  endif
00524 #endif
00525
```

```
00526 /** \ingroup avr_boot
00527     \def boot_signature_byte_get(address)
00528
00529     Read the Signature Row byte at \c address.  For some MCU types,
00530     this function can also retrieve the factory-stored oscillator
00531     calibration bytes.
00532
00533     Parameter \c address can be 0-0x1f as documented by the datasheet.
00534     \note The values are MCU type dependent.
00535 */
00536
00537 #define boot_signature_byte_get(addr)           \
00538   (__extension__({                              \
00539       uint8_t __result;                         \
00540       __asm__ __volatile__                      \
00541       (                                         \
00542         "sts %1, %2"   "\n\t"                   \
00543         "lpm %0, Z"                             \
00544         : "=r" (__result)                       \
00545         : "i" (_SFR_MEM_ADDR(__SPM_REG)),       \
00546           "r" ((uint8_t)(__BOOT_SIGROW_READ)),  \
00547           "z" ((uint16_t)(addr))                \
00548       );                                        \
00549       __result;                                 \
00550 }))
00551
00552 /** \ingroup avr_boot
00553     \def boot_page_fill(address, data)
00554
00555     Fill the bootloader temporary page buffer for flash
00556     address with data word.
00557
00558     \note The address is a byte address. The data is a word. The AVR
00559     writes data to the buffer a word at a time, but addresses the buffer
00560     per byte! So, increment your address by 2 between calls, and send 2
00561     data bytes in a word format! The LSB of the data is written to the lower
00562     address; the MSB of the data is written to the higher address.*/
00563
00564 /** \ingroup avr_boot
00565     \def boot_page_erase(address)
00566
00567     Erase the flash page that contains address.
00568
00569     \note address is a byte address in flash, not a word address. */
00570
00571 /** \ingroup avr_boot
00572     \def boot_page_write(address)
00573
00574     Write the bootloader temporary page buffer
00575     to flash page that contains address.
00576
00577     \note address is a byte address in flash, not a word address. */
00578
00579 /** \ingroup avr_boot
00580     \def boot_rww_enable()
00581
00582     Enable the Read-While-Write memory section. */
00583
00584 /** \ingroup avr_boot
00585     \def boot_lock_bits_set(lock_bits)
00586
00587     Set the bootloader lock bits.
00588
00589     \param lock_bits A mask of which Boot Loader Lock Bits to set.
00590
00591     \note In this context, a 'set bit' will be written to a zero value.
00592     Note also that only BLBxx bits can be programmed by this command.
00593
00594     For example, to disallow the SPM instruction from writing to the Boot
00595     Loader memory section of flash, you would use this macro as such:
00596
00597     \code
00598     boot_lock_bits_set (_BV (BLB11));
```

```
00599     \endcode
00600
00601     \note Like any lock bits, the Boot Loader Lock Bits, once set,
00602     cannot be cleared again except by a chip erase which will in turn
00603     also erase the boot loader itself. */
00604
00605 /* Normal versions of the macros use 16-bit addresses.
00606    Extended versions of the macros use 32-bit addresses.
00607    Alternate versions of the macros use 16-bit addresses and require special
00608    instruction sequences after LPM.
00609
00610    FLASHEND is defined in the ioXXXX.h file.
00611    USHRT_MAX is defined in <limits.h>. */
00612
00613 #if defined(__AVR_ATmega161__) || defined(__AVR_ATmega163__) \
00614     || defined(__AVR_ATmega323__)
00615
00616 /* Alternate: ATmega161/163/323 and 16 bit address */
00617 #define boot_page_fill(address, data) __boot_page_fill_alternate(address, data)
00618 #define boot_page_erase(address)      __boot_page_erase_alternate(address)
00619 #define boot_page_write(address)      __boot_page_write_alternate(address)
00620 #define boot_rww_enable()             __boot_rww_enable_alternate()
00621 #define boot_lock_bits_set(lock_bits) __boot_lock_bits_set_alternate(lock_bits)
00622
00623 #elif (FLASHEND > USHRT_MAX)
00624
00625 /* Extended: >16 bit address */
00626 #define boot_page_fill(address, data) __boot_page_fill_extended(address, data)
00627 #define boot_page_erase(address)      __boot_page_erase_extended(address)
00628 #define boot_page_write(address)      __boot_page_write_extended(address)
00629 #define boot_rww_enable()             __boot_rww_enable()
00630 #define boot_lock_bits_set(lock_bits) __boot_lock_bits_set(lock_bits)
00631
00632 #else
00633
00634 /* Normal: 16 bit address */
00635 #define boot_page_fill(address, data) __boot_page_fill_normal(address, data)
00636 #define boot_page_erase(address)      __boot_page_erase_normal(address)
00637 #define boot_page_write(address)      __boot_page_write_normal(address)
00638 #define boot_rww_enable()             __boot_rww_enable()
00639 #define boot_lock_bits_set(lock_bits) __boot_lock_bits_set(lock_bits)
00640
00641 #endif
00642
00643 /** \ingroup avr_boot
00644
00645     Same as boot_page_fill() except it waits for eeprom and spm operations to
00646     complete before filling the page. */
00647
00648 #define boot_page_fill_safe(address, data) \
00649 do { \
00650     boot_spm_busy_wait();                      \
00651     eeprom_busy_wait();                        \
00652     boot_page_fill(address, data);             \
00653 } while (0)
00654
00655 /** \ingroup avr_boot
00656
00657     Same as boot_page_erase() except it waits for eeprom and spm operations to
00658     complete before erasing the page. */
00659
00660 #define boot_page_erase_safe(address) \
00661 do { \
00662     boot_spm_busy_wait();                      \
00663     eeprom_busy_wait();                        \
00664     boot_page_erase (address);                 \
00665 } while (0)
00666
00667 /** \ingroup avr_boot
00668
00669     Same as boot_page_write() except it waits for eeprom and spm operations to
00670     complete before writing the page. */
00671
```

```
00672 #define boot_page_write_safe(address) \
00673 do { \
00674     boot_spm_busy_wait();                       \
00675     eeprom_busy_wait();                         \
00676     boot_page_write (address);                  \
00677 } while (0)
00678
00679 /** \ingroup avr_boot
00680
00681     Same as boot_rww_enable() except waits for eeprom and spm operations to
00682     complete before enabling the RWW mameory. */
00683
00684 #define boot_rww_enable_safe() \
00685 do { \
00686     boot_spm_busy_wait();                       \
00687     eeprom_busy_wait();                         \
00688     boot_rww_enable();                          \
00689 } while (0)
00690
00691 /** \ingroup avr_boot
00692
00693     Same as boot_lock_bits_set() except waits for eeprom and spm operations to
00694     complete before setting the lock bits. */
00695
00696 #define boot_lock_bits_set_safe(lock_bits) \
00697 do { \
00698     boot_spm_busy_wait();                       \
00699     eeprom_busy_wait();                         \
00700     boot_lock_bits_set (lock_bits);             \
00701 } while (0)
00702
00703 #endif /* _AVR_BOOT_H_ */
```

22.20 cpufunc.h File Reference

Macros

- #define _NOP() __asm__ __volatile__("nop")
- #define _MemoryBarrier() __asm__ __volatile__("" ::: "memory")

Functions

- void ccp_write_io (volatile void *__ioaddr, uint8_t __value)
- void ccp_write_spm (volatile void *__ioaddr, uint8_t __value)

22.21 cpufunc.h

Go to the documentation of this file.

```
00001 /* Copyright (c) 2010, Joerg Wunsch
00002    All rights reserved.
00003
00004    Redistribution and use in source and binary forms, with or without
00005    modification, are permitted provided that the following conditions are met:
00006
00007    * Redistributions of source code must retain the above copyright
00008      notice, this list of conditions and the following disclaimer.
00009
00010    * Redistributions in binary form must reproduce the above copyright
00011      notice, this list of conditions and the following disclaimer in
00012      the documentation and/or other materials provided with the
00013      distribution.
00014
```

```
00015     * Neither the name of the copyright holders nor the names of
00016       contributors may be used to endorse or promote products derived
00017       from this software without specific prior written permission.
00018
00019   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
00020   AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
00021   IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
00022   ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
00023   LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
00024   CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
00025   SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
00026   INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
00027   CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
00028   ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
00029   POSSIBILITY OF SUCH DAMAGE. */
00030
00031 /* avr/cpufunc.h - Special CPU functions */
00032
00033 #ifndef _AVR_CPUFUNC_H_
00034 #define _AVR_CPUFUNC_H_ 1
00035
00036 #include <stdint.h>
00037
00038 /** \file */
00039 /** \defgroup avr_cpufunc <avr/cpufunc.h>: Special AVR CPU functions
00040     \code #include <avr/cpufunc.h> \endcode
00041
00042     This header file contains macros that access special functions of
00043     the AVR CPU which do not fit into any of the other header files.
00044
00045 */
00046
00047
00048 /**
00049    \ingroup avr_cpufunc
00050    \def _NOP
00051
00052    Execute a <i>no operation</i> (NOP) CPU instruction.  This
00053    should not be used to implement delays, better use the functions
00054    from <util/delay_basic.h> or <util/delay.h> for this.  For
00055    debugging purposes, a NOP can be useful to have an instruction that
00056    is guaranteed to be not optimized away by the compiler, so it can
00057    always become a breakpoint in the debugger.
00058 */
00059 #define _NOP() __asm__ __volatile__("nop")
00060
00061
00062 /**
00063    \ingroup avr_cpufunc
00064    \def _MemoryBarrier
00065
00066    Implement a read/write <i>memory barrier</i>.  A memory
00067    barrier instructs the compiler to not cache any memory data in
00068    registers beyond the barrier.  This can sometimes be more effective
00069    than blocking certain optimizations by declaring some object with a
00070    \c volatile qualifier.
00071
00072    See \ref optim_code_reorder for things to be taken into account
00073    with respect to compiler optimizations.
00074 */
00075 #define _MemoryBarrier() __asm__ __volatile__("" ::: "memory")
00076
00077 #include <avr/io.h>
00078 #include <bits/attribs.h>
00079
00080 #ifdef __cplusplus
00081 extern "C" {
00082 #endif
00083
00084 /**
00085    \ingroup avr_cpufunc
00086
00087    Write \a __value to IO Register Protected (CCP) 8-bit IO register
```

```
00088    at \a __ioaddr. See also \c _PROTECTED_WRITE().  */
00089 extern void ccp_write_io (volatile void *__ioaddr, uint8_t __value);
00090
00091 #if __AVR_ARCH__ >= 100
00092 extern __ATTR_ALWAYS_INLINE__ __ATTR_GNU_INLINE__
00093 void ccp_write_io (volatile void *__ioaddr, uint8_t __value)
00094 {
00095   const uintptr_t __addr = (uintptr_t) __ioaddr;
00096
00097 #ifdef __AVR_TINY__
00098   if (__builtin_constant_p (__addr))
00099     __asm__ __volatile__ ("out %i0, %1" "\n\t"
00100                 "out %i2, %3"
00101                 :
00102                 : "n" (& CCP),
00103                   "d" ((uint8_t) 0xd8),
00104                   "n" (__addr),
00105                   "r" ((uint8_t) __value));
00106   else
00107     __asm__ __volatile__ ("out %i0, %1" "\n\t"
00108                 "st %a2, %3"
00109                 :
00110                 : "n" (& CCP),
00111                   "d" ((uint8_t) 0xd8),
00112                   "e" (__addr),
00113                   "r" ((uint8_t) __value));
00114 #elif defined(__AVR_XMEGA__)
00115   if (__builtin_constant_p (__addr))
00116     __asm__ __volatile__ ("out %i0, %1" "\n\t"
00117                 "sts %2, %3"
00118                 :
00119                 : "n" (& CCP),
00120                   "d" ((uint8_t) CCP_IOREG_gc),
00121                   "n" (__addr),
00122                   "r" ((uint8_t) __value));
00123   else
00124     __asm__ __volatile__ ("out %i0, %1" "\n\t"
00125                 "st %a2, %3"
00126                 :
00127                 : "n" (& CCP),
00128                   "d" ((uint8_t) CCP_IOREG_gc),
00129                   "e" (__addr),
00130                   "r" ((uint8_t) __value));
00131 #endif
00132 }
00133 #endif /* ARCH >= 100 */
00134
00135 /**
00136    \ingroup avr_cpufunc
00137
00138    Write \a __value to SPM Instruction Protected (CCP) 8-bit IO register
00139    at \a __ioaddr. See also \c _PROTECTED_WRITE_SPM().  */
00140 extern void ccp_write_spm (volatile void *__ioaddr, uint8_t __value);
00141
00142 #if defined(__AVR_XMEGA__) || defined(CCP_SPM_gc)
00143 extern __ATTR_ALWAYS_INLINE__ __ATTR_GNU_INLINE__
00144 void ccp_write_spm (volatile void *__ioaddr, uint8_t __value)
00145 {
00146   const uintptr_t __addr = (uintptr_t) __ioaddr;
00147
00148   if (__builtin_constant_p (__addr))
00149     __asm__ __volatile__ ("out %i0, %1" "\n\t"
00150 #ifdef __AVR_TINY__
00151                 "out %i2, %3"
00152 #else
00153                 "sts %2, %3"
00154 #endif
00155                 :
00156                 : "n" (& CCP),
00157                   "d" ((uint8_t) CCP_SPM_gc),
00158                   "n" (__addr),
00159                   "r" ((uint8_t) __value));
00160   else
```

```
00161     __asm__ __volatile__ ("out %i0, %1" "\n\t"
00162                 "st %a2, %3"
00163                 :
00164                 : "n" (& CCP),
00165                   "d" ((uint8_t) CCP_SPM_gc),
00166                   "e" (__addr),
00167                   "r" ((uint8_t) __value));
00168 }
00169 #endif /* AVR_XMEGA || ATtiny102/104 */
00170
00171 #ifdef __cplusplus
00172 }
00173 #endif
00174
00175 #endif /* _AVR_CPUFUNC_H_ */
```

22.22 eeprom.h File Reference

Macros

- #define EEMEM __attribute__((__section__(".eeprom")))
- #define eeprom_is_ready()
- #define eeprom_busy_wait() do {} while (!eeprom_is_ready())

IAR C Compatibility Defines

- #define _EEPUT(addr, val) eeprom_write_byte ((uint8_t *)(addr), (uint8_t)(val))
- #define __EEPUT(addr, val) eeprom_write_byte ((uint8_t *)(addr), (uint8_t)(val))
- #define _EEGET(var, addr) (var) = eeprom_read_byte ((const uint8_t *)(addr))
- #define __EEGET(var, addr) (var) = eeprom_read_byte ((const uint8_t *)(addr))

Functions

EEPROM Read Functions

- uint8_t eeprom_read_byte (const uint8_t *__p)
- char eeprom_read_char (const char *__p)
- uint8_t eeprom_read_u8 (const uint8_t *__p)
- int8_t eeprom_read_i8 (const int8_t *__p)
- uint16_t eeprom_read_word (const uint16_t *__p)
- uint16_t eeprom_read_u16 (const uint16_t *__p)
- int16_t eeprom_read_i16 (const int16_t *__p)
- uint24_t eeprom_read_u24 (const uint24_t *__p)
- int24_t eeprom_read_i24 (const int24_t *__p)
- uint32_t eeprom_read_dword (const uint32_t *__p)
- uint32_t eeprom_read_u32 (const uint32_t *__p)
- int32_t eeprom_read_i32 (const int32_t *__p)
- uint64_t eeprom_read_qword (const uint64_t *__p)
- uint64_t eeprom_read_u64 (const uint64_t *__p)
- int64_t eeprom_read_i64 (const int64_t *__p)
- float eeprom_read_float (const float *__p)
- double eeprom_read_double (const double *__p)
- long double eeprom_read_long_double (const long double *__p)
- void eeprom_read_block (void *__dst, const void *__src, size_t __n)

EEPROM Write Functions

- void eeprom_write_byte (uint8_t *__p, uint8_t __value)
- void eeprom_write_char (char *__p, char __value)

- void eeprom_write_u8 (uint8_t *__p, uint8_t __value)
- void eeprom_write_i8 (int8_t *__p, int8_t __value)
- void eeprom_write_word (uint16_t *__p, uint16_t __value)
- void eeprom_write_u16 (uint16_t *__p, uint16_t __value)
- void eeprom_write_i16 (int16_t *__p, int16_t __value)
- void eeprom_write_u24 (uint24_t *__p, uint24_t __value)
- void eeprom_write_i24 (int24_t *__p, int24_t __value)
- void eeprom_write_dword (uint32_t *__p, uint32_t __value)
- void eeprom_write_u32 (uint32_t *__p, uint32_t __value)
- void eeprom_write_i32 (int32_t *__p, int32_t __value)
- void eeprom_write_qword (uint64_t *__p, uint64_t __value)
- void eeprom_write_u64 (uint64_t *__p, uint64_t __value)
- void eeprom_write_i64 (int64_t *__p, int64_t __value)
- void eeprom_write_float (float *__p, float __value)
- void eeprom_write_double (double *__p, double __value)
- void eeprom_write_long_double (long double *__p, long double __value)
- void eeprom_write_block (const void *__src, void *__dst, size_t __n)

EEPROM Update Functions

- void eeprom_update_byte (uint8_t *__p, uint8_t __value)
- void eeprom_update_char (char *__p, char __value)
- void eeprom_update_u8 (uint8_t *__p, uint8_t __value)
- void eeprom_update_i8 (int8_t *__p, int8_t __value)
- void eeprom_update_word (uint16_t *__p, uint16_t __value)
- void eeprom_update_u16 (uint16_t *__p, uint16_t __value)
- void eeprom_update_i16 (int16_t *__p, int16_t __value)
- void eeprom_update_u24 (uint24_t *__p, uint24_t __value)
- void eeprom_update_i24 (int24_t *__p, int24_t __value)
- void eeprom_update_dword (uint32_t *__p, uint32_t __value)
- void eeprom_update_u32 (uint32_t *__p, uint32_t __value)
- void eeprom_update_i32 (int32_t *__p, int32_t __value)
- void eeprom_update_qword (uint64_t *__p, uint64_t __value)
- void eeprom_update_u64 (uint64_t *__p, uint64_t __value)
- void eeprom_update_i64 (int64_t *__p, int64_t __value)
- void eeprom_update_float (float *__p, float __value)
- void eeprom_update_double (double *__p, double __value)
- void eeprom_update_long_double (long double *__p, long double __value)
- void eeprom_update_block (const void *__src, void *__dst, size_t __n)

22.23 eeprom.h

Go to the documentation of this file.

```
00001 /* Copyright (c) 2002, 2003, 2004, 2007 Marek Michalkiewicz
00002    Copyright (c) 2005, 2006 Bjoern Haase
00003    Copyright (c) 2008 Atmel Corporation
00004    Copyright (c) 2008 Wouter van Gulik
00005    Copyright (c) 2009 Dmitry Xmelkov
00006    All rights reserved.
00007
00008    Redistribution and use in source and binary forms, with or without
00009    modification, are permitted provided that the following conditions are met:
00010
00011    * Redistributions of source code must retain the above copyright
00012      notice, this list of conditions and the following disclaimer.
00013    * Redistributions in binary form must reproduce the above copyright
00014      notice, this list of conditions and the following disclaimer in
00015      the documentation and/or other materials provided with the
00016      distribution.
00017    * Neither the name of the copyright holders nor the names of
00018      contributors may be used to endorse or promote products derived
00019      from this software without specific prior written permission.
00020
```

```
00021   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
00022   AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
00023   IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
00024   ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
00025   LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
00026   CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
00027   SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
00028   INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
00029   CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
00030   ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
00031   POSSIBILITY OF SUCH DAMAGE. */
00032
00033 #ifndef _AVR_EEPROM_H_
00034 #define _AVR_EEPROM_H_ 1
00035
00036 #include <avr/io.h>
00037
00038 #if !E2END && !defined(__DOXYGEN__) && !defined(__COMPILING_AVR_LIBC__)
00039 # warning "Device does not have EEPROM available."
00040 #else
00041
00042 #if defined (EEAR) && !defined (EEARL) && !defined (EEARH)
00043 #define EEARL EEAR
00044 #endif
00045
00046 #ifndef __ASSEMBLER__
00047
00048 #include <stddef.h> /* size_t */
00049 #include <stdint.h>
00050
00051 /** \file */
00052 /** \defgroup avr_eeprom <avr/eeprom.h>: EEPROM handling
00053     \code #include <avr/eeprom.h> \endcode
00054
00055     This header file declares the interface to some simple library
00056     routines suitable for handling the data EEPROM contained in the
00057     AVR microcontrollers.  The implementation uses a simple polled
00058     mode interface.  Applications that require interrupt-controlled
00059     EEPROM access to ensure that no time will be wasted in spinloops
00060     will have to deploy their own implementation.
00061
00062     <b>Notes:</b>
00063
00064     - Date is stores and retrieved in little endian format and with
00065     no padding bytes or alignment requirements.
00066
00067     - In addition to the write functions there is a set of update functions.
00068     This functions read each byte first and skip the burning if the
00069     old value is the same with new.  The scanning direction is from
00070     high address to low, to obtain quick return in common cases.
00071
00072     - Similar functions for fixed-point types are supplied by
00073     \ref avr_stdfix "<stdfix.h>".
00074
00075     - All of the read/write functions first make sure the EEPROM is
00076     ready to be accessed.  Since this may cause long delays if a
00077     write operation is still pending, time-critical applications
00078     should first poll the EEPROM e. g. using eeprom_is_ready() before
00079     attempting any actual I/O.  But this functions does not wait until
00080     SELFPRGEN in SPMCSR becomes zero.  Do this manually, if your
00081     software contains the Flash burning.
00082
00083     - As these functions modify IO registers, they are known to be
00084     non-reentrant.  If any of these functions are used from both,
00085     standard and interrupt context, the applications must ensure
00086     proper protection (e.g. by disabling interrupts before accessing
00087     them).
00088
00089     - All write functions force erase_and_write programming mode.
00090
00091     - For Xmega the EEPROM start address is 0, like other architectures.
00092     The reading functions add the 0x2000 value to use EEPROM mapping into
00093     data space.
```

```
00094  */
00095
00096 #ifdef __cplusplus
00097 extern "C" {
00098 #endif
00099
00100 #include <bits/attribs.h>
00101
00102 /** \ingroup avr_eeprom
00103     Attribute expression causing a variable to be allocated within the
00104     .eeprom section.    */
00105 #define EEMEM __attribute__((__section__(".eeprom")))
00106
00107 /** \def eeprom_is_ready
00108     \ingroup avr_eeprom
00109     \returns 1 if EEPROM is ready for a new read/write operation, 0 if not.
00110  */
00111 #if defined (__DOXYGEN__)
00112 # define eeprom_is_ready()
00113 #elif   defined (NVM_STATUS)
00114 # define eeprom_is_ready()  bit_is_clear (NVM_STATUS, NVM_NVMBUSY_bp)
00115 #elif   defined (NVMCTRL_STATUS)
00116 # define eeprom_is_ready()  bit_is_clear (NVMCTRL_STATUS, NVMCTRL_EEBUSY_bp)
00117 #elif   defined (DEECR)
00118 # define eeprom_is_ready()  bit_is_clear (DEECR, BSY)
00119 #elif   defined (EEPE)
00120 # define eeprom_is_ready()  bit_is_clear (EECR, EEPE)
00121 #else
00122 # define eeprom_is_ready()  bit_is_clear (EECR, EEWE)
00123 #endif
00124
00125
00126 /** \ingroup avr_eeprom
00127     Loops until the eeprom is no longer busy.
00128     \returns Nothing. */
00129 #define eeprom_busy_wait() do {} while (!eeprom_is_ready())
00130
00131 /** \name EEPROM Read Functions */
00132
00133 /** \ingroup avr_eeprom
00134     Read one byte from EEPROM address \a __p. */
00135 uint8_t eeprom_read_byte (const uint8_t *__p) __ATTR_PURE__;
00136
00137 /** \ingroup avr_eeprom
00138     Read a char from EEPROM address \a __p.
00139     \since AVR-LibC v2.3 */
00140 char eeprom_read_char (const char *__p) __asm("eeprom_read_byte") __ATTR_PURE__;
00141
00142 /** \ingroup avr_eeprom
00143     Read an unsigned 8-bit integer from EEPROM address \a __p.
00144     \since AVR-LibC v2.3 */
00145 uint8_t eeprom_read_u8 (const uint8_t *__p) __asm("eeprom_read_byte") __ATTR_PURE__;
00146
00147 /** \ingroup avr_eeprom
00148     Read a signed 8-bit integer from EEPROM address \a __p.
00149     \since AVR-LibC v2.3 */
00150 int8_t eeprom_read_i8 (const int8_t *__p) __asm("eeprom_read_byte") __ATTR_PURE__;
00151
00152 /** \ingroup avr_eeprom
00153     Read one 16-bit word from EEPROM address \a __p. */
00154 uint16_t eeprom_read_word (const uint16_t *__p) __ATTR_PURE__;
00155
00156 /** \ingroup avr_eeprom
00157     Read an unsigned 16-bit integer from EEPROM address \a __p.
00158     \since AVR-LibC v2.3 */
00159 uint16_t eeprom_read_u16 (const uint16_t *__p) __asm("eeprom_read_word") __ATTR_PURE__;
00160
00161 /** \ingroup avr_eeprom
00162     Read a signed 16-bit integer from EEPROM address \a __p.
00163     \since AVR-LibC v2.3 */
00164 int16_t eeprom_read_i16 (const int16_t *__p) __asm("eeprom_read_word") __ATTR_PURE__;
00165
00166 /** \ingroup avr_eeprom
```

```
00167     Read an unsigned 24-bit integer from EEPROM address \a __p.
00168     \since AVR-LibC v2.3 */
00169 #if defined(__DOXYGEN__) || defined(__INT24_MAX__)
00170 uint24_t eeprom_read_u24 (const uint24_t *__p) __ATTR_PURE__;
00171 #endif
00172
00173 /** \ingroup avr_eeprom
00174     Read a signed 24-bit integer from EEPROM address \a __p.
00175     \since AVR-LibC v2.3 */
00176 #if defined(__DOXYGEN__) || defined(__INT24_MAX__)
00177 int24_t eeprom_read_i24 (const int24_t *__p) __ATTR_PURE__;
00178 #endif
00179
00180 /** \ingroup avr_eeprom
00181     Read one 32-bit double word from EEPROM address \a __p. */
00182 uint32_t eeprom_read_dword (const uint32_t *__p) __ATTR_PURE__;
00183
00184 /** \ingroup avr_eeprom
00185     Read an unsigned 32-bit integer from EEPROM address \a __p.
00186     \since AVR-LibC v2.3 */
00187 uint32_t eeprom_read_u32 (const uint32_t *__p) __asm("eeprom_read_dword") __ATTR_PURE__;
00188
00189 /** \ingroup avr_eeprom
00190     Read a signed 32-bit integer from EEPROM address \a __p.
00191     \since AVR-LibC v2.3 */
00192 int32_t eeprom_read_i32 (const int32_t *__p) __asm("eeprom_read_dword") __ATTR_PURE__;
00193
00194 /** \ingroup avr_eeprom
00195     Read one 64-bit quad word from EEPROM address \a __p.
00196     \since AVR-LibC v2.2 */
00197 #if defined(__DOXYGEN__) || __SIZEOF_LONG_LONG__ == 8
00198 uint64_t eeprom_read_qword (const uint64_t *__p) __ATTR_PURE__;
00199 #endif
00200
00201 /** \ingroup avr_eeprom
00202     Read an unsigned 64-bit integer from EEPROM address \a __p.
00203     \since AVR-LibC v2.3 */
00204 #if defined(__DOXYGEN__) || __SIZEOF_LONG_LONG__ == 8
00205 uint64_t eeprom_read_u64 (const uint64_t *__p) __asm("eeprom_read_qword") __ATTR_PURE__;
00206 #endif
00207
00208 /** \ingroup avr_eeprom
00209     Read a signed 64-bit integer from EEPROM address \a __p.
00210     \since AVR-LibC v2.3 */
00211 #if defined(__DOXYGEN__) || __SIZEOF_LONG_LONG__ == 8
00212 int64_t eeprom_read_i64 (const int64_t *__p) __asm("eeprom_read_qword") __ATTR_PURE__;
00213 #endif
00214
00215 /** \ingroup avr_eeprom
00216     Read one float value from EEPROM address \a __p. */
00217 float eeprom_read_float (const float *__p) __ATTR_PURE__;
00218
00219 /** \ingroup avr_eeprom
00220     Read one double value from EEPROM address \a __p.
00221     \since AVR-LibC v2.2 */
00222 #if defined(__DOXYGEN__)
00223 double eeprom_read_double (const double *__p);
00224 #elif __SIZEOF_DOUBLE__ == 4
00225 double eeprom_read_double (const double *__p) __asm("eeprom_read_dword");
00226 #elif __SIZEOF_DOUBLE__ == 8
00227 double eeprom_read_double (const double *__p) __asm("eeprom_read_qword");
00228 #endif
00229
00230 /** \ingroup avr_eeprom
00231     Read one long double value from EEPROM address \a __p.
00232     \since AVR-LibC v2.2 */
00233 #if defined(__DOXYGEN__)
00234 long double eeprom_read_long_double (const long double *__p);
00235 #elif __SIZEOF_LONG_DOUBLE__ == 4
00236 long double eeprom_read_long_double (const long double *__p) __asm("eeprom_read_dword");
00237 #elif __SIZEOF_LONG_DOUBLE__ == 8
00238 long double eeprom_read_long_double (const long double *__p) __asm("eeprom_read_qword");
00239 #endif
```

```
00240
00241 /** \ingroup avr_eeprom
00242     Read a block of \a __n bytes from EEPROM address \a __src to SRAM
00243     \a __dst. */
00244 void eeprom_read_block (void *__dst, const void *__src, size_t __n);
00245
00246
00247 /** \name EEPROM Write Functions */
00248
00249 /** \ingroup avr_eeprom
00250     Write a byte \a __value to EEPROM address \a __p. */
00251 void eeprom_write_byte (uint8_t *__p, uint8_t __value);
00252
00253 /** \ingroup avr_eeprom
00254     Write a char to EEPROM address \a __p.
00255     \since AVR-LibC v2.3 */
00256 void eeprom_write_char (char *__p, char __value) __asm("eeprom_write_byte");
00257
00258 /** \ingroup avr_eeprom
00259     Write an unsigned 8-bit integer to EEPROM address \a __p.
00260     \since AVR-LibC v2.3 */
00261 void eeprom_write_u8 (uint8_t *__p, uint8_t __value) __asm("eeprom_write_byte");
00262
00263 /** \ingroup avr_eeprom
00264     Write a signed 8-bit integer to EEPROM address \a __p.
00265     \since AVR-LibC v2.3 */
00266 void eeprom_write_i8 (int8_t *__p, int8_t __value) __asm("eeprom_write_byte");
00267
00268 /** \ingroup avr_eeprom
00269     Write a word \a __value to EEPROM address \a __p. */
00270
00271 void eeprom_write_word (uint16_t *__p, uint16_t __value);
00272
00273 /** \ingroup avr_eeprom
00274     Write an unsigned 16-bit integer to EEPROM address \a __p.
00275     \since AVR-LibC v2.3 */
00276 void eeprom_write_u16 (uint16_t *__p, uint16_t __value) __asm("eeprom_write_word");
00277
00278 /** \ingroup avr_eeprom
00279     Write a signed 16-bit integer to EEPROM address \a __p.
00280     \since AVR-LibC v2.3 */
00281 void eeprom_write_i16 (int16_t *__p, int16_t __value) __asm("eeprom_write_word");
00282
00283 /** \ingroup avr_eeprom
00284     Write an unsigned 24-bit integer to EEPROM address \a __p.
00285     \since AVR-LibC v2.3 */
00286 #if defined(__DOXYGEN__) || defined(__INT24_MAX__)
00287 void eeprom_write_u24 (uint24_t *__p, uint24_t __value);
00288 #endif
00289
00290 /** \ingroup avr_eeprom
00291     Write a signed 24-bit integer to EEPROM address \a __p.
00292     \since AVR-LibC v2.3 */
00293 #if defined(__DOXYGEN__) || defined(__INT24_MAX__)
00294 void eeprom_write_i24 (int24_t *__p, int24_t __value);
00295 #endif
00296
00297 /** \ingroup avr_eeprom
00298     Write a 32-bit double word \a __value to EEPROM address \a __p. */
00299 void eeprom_write_dword (uint32_t *__p, uint32_t __value);
00300
00301 /** \ingroup avr_eeprom
00302     Write an unsigned 32-bit integer to EEPROM address \a __p.
00303     \since AVR-LibC v2.3 */
00304 void eeprom_write_u32 (uint32_t *__p, uint32_t __value) __asm("eeprom_write_dword");
00305
00306 /** \ingroup avr_eeprom
00307     Write a signed 32-bit integer to EEPROM address \a __p.
00308     \since AVR-LibC v2.3 */
00309 void eeprom_write_i32 (int32_t *__p, int32_t __value) __asm("eeprom_write_dword");
00310
00311 /** \ingroup avr_eeprom
00312     Write a 64-bit quad word \a __value to EEPROM address \a __p.
```

```
00313     \since AVR-LibC v2.2 */
00314 #if defined(__DOXYGEN__) || __SIZEOF_LONG_LONG__ == 8
00315 void eeprom_write_qword (uint64_t *__p, uint64_t __value);
00316 #endif
00317
00318 /** \ingroup avr_eeprom
00319     Write an unsigned 64-bit integer to EEPROM address \a __p.
00320     \since AVR-LibC v2.3 */
00321 #if defined(__DOXYGEN__) || __SIZEOF_LONG_LONG__ == 8
00322 void eeprom_write_u64 (uint64_t *__p, uint64_t __value) __asm("eeprom_write_qword");
00323 #endif
00324
00325 /** \ingroup avr_eeprom
00326     Write a signed 64-bit integer to EEPROM address \a __p.
00327     \since AVR-LibC v2.3 */
00328 #if defined(__DOXYGEN__) || __SIZEOF_LONG_LONG__ == 8
00329 void eeprom_write_i64 (int64_t *__p, int64_t __value) __asm("eeprom_write_qword");
00330 #endif
00331
00332 /** \ingroup avr_eeprom
00333     Write a float \a __value to EEPROM address \a __p. */
00334 void eeprom_write_float (float *__p, float __value);
00335
00336 /** \ingroup avr_eeprom
00337     Write a double \a __value to EEPROM address \a __p.
00338     \since AVR-LibC v2.2 */
00339 #if defined(__DOXYGEN__)
00340 void eeprom_write_double (double *__p, double __value);
00341 #elif __SIZEOF_DOUBLE__ == 4
00342 void eeprom_write_double (double *__p, double __value) __asm("eeprom_write_dword");
00343 #elif __SIZEOF_DOUBLE__ == 8
00344 void eeprom_write_double (double *__p, double __value) __asm("eeprom_write_qword");
00345 #endif
00346
00347 /** \ingroup avr_eeprom
00348     Write a long double \a __value to EEPROM address \a __p.
00349     \since AVR-LibC v2.2 */
00350 #if defined(__DOXYGEN__)
00351 void eeprom_write_long_double (long double *__p, long double __value);
00352 #elif __SIZEOF_LONG_DOUBLE__ == 4
00353 void eeprom_write_long_double (long double *__p, long double __value)
    __asm("eeprom_write_dword");
00354 #elif __SIZEOF_LONG_DOUBLE__ == 8
00355 void eeprom_write_long_double (long double *__p, long double __value)
    __asm("eeprom_write_qword");
00356 #endif
00357
00358 /** \ingroup avr_eeprom
00359     Write a block of \a __n bytes to EEPROM address \a __dst from \a __src.
00360     \note The argument order is mismatch with common functions like strcpy(). */
00361 void eeprom_write_block (const void *__src, void *__dst, size_t __n);
00362
00363
00364 /** \name EEPROM Update Functions */
00365
00366 /** \ingroup avr_eeprom
00367     Update a byte \a __value at EEPROM address \a __p. */
00368 void eeprom_update_byte (uint8_t *__p, uint8_t __value);
00369
00370 /** \ingroup avr_eeprom
00371     Update a char \a at EEPROM address \a __p.
00372     \since AVR-LibC v2.3 */
00373 void eeprom_update_char (char *__p, char __value) __asm("eeprom_update_byte");
00374
00375 /** \ingroup avr_eeprom
00376     Update an unsigned 8-bit integer \a at EEPROM address \a __p.
00377     \since AVR-LibC v2.3 */
00378 void eeprom_update_u8 (uint8_t *__p, uint8_t __value) __asm("eeprom_update_byte");
00379
00380 /** \ingroup avr_eeprom
00381     Update a signed 8-bit integer \a at EEPROM address \a __p.
00382     \since AVR-LibC v2.3 */
00383 void eeprom_update_i8 (int8_t *__p, int8_t __value) __asm("eeprom_update_byte");
```

```
00384
00385 /** \ingroup avr_eeprom
00386     Update a word \a __value at EEPROM address \a __p. */
00387 void eeprom_update_word (uint16_t *__p, uint16_t __value);
00388
00389 /** \ingroup avr_eeprom
00390     Update an unsigned 16-bit integer \a at EEPROM address \a __p.
00391     \since AVR-LibC v2.3 */
00392 void eeprom_update_u16 (uint16_t *__p, uint16_t __value) __asm("eeprom_update_word");
00393
00394 /** \ingroup avr_eeprom
00395     Update a signed 16-bit integer \a at EEPROM address \a __p.
00396     \since AVR-LibC v2.3 */
00397 void eeprom_update_i16 (int16_t *__p, int16_t __value) __asm("eeprom_update_word");
00398
00399 /** \ingroup avr_eeprom
00400     Update an unsigned 24-bit integer \a at EEPROM address \a __p.
00401     \since AVR-LibC v2.3 */
00402 #if defined(__DOXYGEN__) || defined(__INT24_MAX__)
00403 void eeprom_update_u24 (uint24_t *__p, uint24_t __value);
00404 #endif
00405
00406 /** \ingroup avr_eeprom
00407     Update a signed 24-bit integer \a at EEPROM address \a __p.
00408     \since AVR-LibC v2.3 */
00409 #if defined(__DOXYGEN__) || defined(__INT24_MAX__)
00410 void eeprom_update_i24 (int24_t *__p, int24_t __value);
00411 #endif
00412
00413 /** \ingroup avr_eeprom
00414     Update a 32-bit double word \a __value at EEPROM address \a __p. */
00415 void eeprom_update_dword (uint32_t *__p, uint32_t __value);
00416
00417 /** \ingroup avr_eeprom
00418     Update an unsigned 32-bit integer \a at EEPROM address \a __p.
00419     \since AVR-LibC v2.3 */
00420 void eeprom_update_u32 (uint32_t *__p, uint32_t __value) __asm("eeprom_update_dword");
00421
00422 /** \ingroup avr_eeprom
00423     Update a signed 32-bit integer \a at EEPROM address \a __p.
00424     \since AVR-LibC v2.3 */
00425 void eeprom_update_i32 (int32_t *__p, int32_t __value) __asm("eeprom_update_dword");
00426
00427 /** \ingroup avr_eeprom
00428     Update a 64-bit quad word \a __value at EEPROM address \a __p.
00429     \since AVR-LibC v2.2 */
00430 #if defined(__DOXYGEN__) || __SIZEOF_LONG_LONG__ == 8
00431 void eeprom_update_qword (uint64_t *__p, uint64_t __value);
00432 #endif
00433
00434 /** \ingroup avr_eeprom
00435     Update an unsigned 64-bit integer \a at EEPROM address \a __p.
00436     \since AVR-LibC v2.3 */
00437 #if defined(__DOXYGEN__) || __SIZEOF_LONG_LONG__ == 8
00438 void eeprom_update_u64 (uint64_t *__p, uint64_t __value) __asm("eeprom_update_qword");
00439 #endif
00440
00441 /** \ingroup avr_eeprom
00442     Update a signed 64-bit integer \a at EEPROM address \a __p.
00443     \since AVR-LibC v2.3 */
00444 #if defined(__DOXYGEN__) || __SIZEOF_LONG_LONG__ == 8
00445 void eeprom_update_i64 (int64_t *__p, int64_t __value) __asm("eeprom_update_qword");
00446 #endif
00447
00448 /** \ingroup avr_eeprom
00449     Update a float \a __value at EEPROM address \a __p. */
00450 void eeprom_update_float (float *__p, float __value);
00451
00452 /** \ingroup avr_eeprom
00453     Update a double \a __value at EEPROM address \a __p.
00454     \since AVR-LibC v2.2 */
00455 #if defined(__DOXYGEN__)
00456 void eeprom_update_double (double *__p, double __value);
```

```
00457 #elif __SIZEOF_DOUBLE__ == 4
00458 void eeprom_update_double (double *__p, double __value) __asm("eeprom_update_dword");
00459 #elif __SIZEOF_DOUBLE__ == 8
00460 void eeprom_update_double (double *__p, double __value) __asm("eeprom_update_qword");
00461 #endif
00462
00463 /** \ingroup avr_eeprom
00464     Update a long double \a __value at EEPROM address \a __p.
00465     \since AVR-LibC v2.2 */
00466 #if defined(__DOXYGEN__)
00467 void eeprom_update_long_double (long double *__p, long double __value);
00468 #elif __SIZEOF_LONG_DOUBLE__ == 4
00469 void eeprom_update_long_double (long double *__p, long double __value)
      __asm("eeprom_update_dword");
00470 #elif __SIZEOF_LONG_DOUBLE__ == 8
00471 void eeprom_update_long_double (long double *__p, long double __value)
      __asm("eeprom_update_qword");
00472 #endif
00473
00474 /** \ingroup avr_eeprom
00475     Update a block of \a __n bytes at EEPROM address \a __dst from \a __src.
00476     \note The argument order is mismatch with common functions like strcpy(). */
00477 void eeprom_update_block (const void *__src, void *__dst, size_t __n);
00478
00479
00480 /** \name IAR C Compatibility Defines   */
00481 /**@{*/
00482
00483 /** \ingroup avr_eeprom
00484     Write a byte to EEPROM. Compatibility define for IAR C. */
00485 #define _EEPUT(addr, val) eeprom_write_byte ((uint8_t *)(addr), (uint8_t)(val))
00486
00487 /** \ingroup avr_eeprom
00488     Write a byte to EEPROM. Compatibility define for IAR C. */
00489 #define __EEPUT(addr, val) eeprom_write_byte ((uint8_t *)(addr), (uint8_t)(val))
00490
00491 /** \ingroup avr_eeprom
00492     Read a byte from EEPROM. Compatibility define for IAR C.    */
00493 #define _EEGET(var, addr) (var) = eeprom_read_byte ((const uint8_t *)(addr))
00494
00495 /** \ingroup avr_eeprom
00496     Read a byte from EEPROM. Compatibility define for IAR C.    */
00497 #define __EEGET(var, addr) (var) = eeprom_read_byte ((const uint8_t *)(addr))
00498
00499 /**@}*/
00500
00501 #ifdef __cplusplus
00502 }
00503 #endif
00504
00505 #endif  /* !__ASSEMBLER__ */
00506 #endif  /* E2END || defined(__DOXYGEN__) || defined(__COMPILING_AVR_LIBC__) */
00507 #endif  /* !_AVR_EEPROM_H_ */
```

22.24 flash.h File Reference

Macros

Macros

- #define FSTR(str) ({ static const __flash char c[ ] = (str); &c[0]; })
- #define FXSTR(str) ({ static const __flashx char c[ ] = (str); &c[0]; })
- #define FLIT(str) ((const __flash char[ ]) { str })
- #define FXLIT(str) ((const __flashx char[ ]) { str })

Convenience macros for functions from stdio.h, that allocate the format string with FSTR

- #define vfprintf_FSTR(stream, fmt, ap) vfprintf_F(stream, FSTR(fmt), ap)

- #define printf_FSTR(fmt, ...) printf_F(FSTR(fmt), ##__VA_ARGS__)
- #define sprintf_FSTR(s, fmt, ...) sprintf_F(s, FSTR(fmt), ##__VA_ARGS__)
- #define snprintf_FSTR(s, n, fmt, ...) snprintf_F(s, n, FSTR(fmt), ##__VA_ARGS__)
- #define vsprintf_FSTR(s, fmt, ap) vsprintf_F(s, FSTR(fmt), ap)
- #define vsnprintf_FSTR(s, n, fmt, ap) vsnprintf_F(s, n, FSTR(fmt), ap);
- #define fprintf_FSTR(stream, fmt, ...) fprintf_F(stream, FSTR(fmt), ##__VA_ARGS__)
- #define fputs_FSTR(str, stream) fputs_F(FSTR(str), stream);
- #define puts_FSTR(str) puts_F(FSTR(str));
- #define vfscanf_FSTR(stream, fmt, ap) vfscanf_F(stream, FSTR(fmt), ap);
- #define fscanf_FSTR(stream, fmt, ...) fscanf_F(stream, FSTR(fmt), ##__VA_ARGS__)
- #define scanf_FSTR(fmt, ...) scanf_F(FSTR(fmt), ##__VA_ARGS__)
- #define sscanf_FSTR(buf, fmt, ...) sscanf_F(buf, FSTR(fmt), ##__VA_ARGS__)

Functions

Functions from string.h, but one argument is in address-space __flash

- const __flash void * memchr_F (const __flash void *, int, size_t)
- int memcmp_F (const void *, const __flash void *, size_t)
- void * memccpy_F (void *, const __flash void *, int val, size_t)
- void * memcpy_F (void *, const __flash void *, size_t)
- void * memmem_F (const void *, size_t, const __flash void *, size_t)
- const __flash void * memrchr_F (const __flash void *, int val, size_t len)
- static size_t strlen_F (const __flash char *src)
- char * strcat_F (char *, const __flash char *)
- const __flash char * strchr_F (const __flash char *, int val)
- const __flash char * strchrnul_F (const __flash char *, int val)
- int strcmp_F (const char *, const __flash char *)
- char * strcpy_F (char *, const __flash char *)
- char * stpcpy_F (char *, const __flash char *)
- int strcasecmp_F (const char *, const __flash char *)
- char * strcasestr_F (const char *, const __flash char *)
- size_t strcspn_F (const char *s, const __flash char *reject)
- size_t strlcat_F (char *, const __flash char *, size_t)
- size_t strlcpy_F (char *, const __flash char *, size_t)
- size_t strnlen_F (const __flash char *, size_t)
- int strncmp_F (const char *, const __flash char *, size_t)
- int strncasecmp_F (const char *, const __flash char *, size_t)
- char * strncat_F (char *, const __flash char *, size_t)
- char * strncpy_F (char *, const __flash char *, size_t)
- char * strpbrk_F (const char *, const __flash char *accept)
- const __flash char * strrchr_F (const __flash char *, int val)
- char * strsep_F (char **sp, const __flash char *delim)
- size_t strspn_F (const char *s, const __flash char *accept)
- char * strstr_F (const char *, const __flash char *)
- char * strtok_F (char *s, const __flash char *delim)
- char * strtok_rF (char *s, const __flash char *delim, char **last)

Functions from string.h, but one argument is in 24-bit address-space __flashx

- void * memcpy_FX (void *dest, const __flashx void *src, size_t len)
- int memcmp_FX (const void *s1, const __flashx void *s2, size_t)
- size_t strlen_FX (const __flashx char *src)
- size_t strnlen_FX (const __flashx char *src, size_t len)
- char * strcpy_FX (char *dest, const __flashx char *src)
- char * stpcpy_FX (char *dest, const __flashx char *src)
- char * strncpy_FX (char *dest, const __flashx char *src, size_t len)
- char * strcat_FX (char *dest, const __flashx char *src)
- size_t strlcat_FX (char *dst, const __flashx char *src, size_t siz)
- char * strncat_FX (char *dest, const __flashx char *src, size_t len)
- int strcmp_FX (const char *s1, const __flashx char *s2)
- int strncmp_FX (const char *s1, const __flashx char *s2, size_t n)

- int strcasecmp_FX (const char *s1, const __flashx char *s2)
- int strncasecmp_FX (const char *s1, const __flashx char *s2, size_t n)
- const __flashx char * strchr_FX (const __flashx char *s, int val)
- char * strstr_FX (const char *s1, const __flashx char *s2)
- size_t strlcpy_FX (char *, const __flashx char *, size_t)

Functions from stdio.h, but with a format string in address-space __flash

- int vfprintf_F (FILE *stream, const __flash char *fmt, va_list ap)
- int printf_F (const __flash char *fmt,...)
- int sprintf_F (char *s, const __flash char *fmt,...)
- int snprintf_F (char *s, size_t n, const __flash char *fmt,...)
- int vsprintf_F (char *s, const __flash char *fmt, va_list ap)
- int vsnprintf_F (char *s, size_t n, const __flash char *fmt, va_list ap)
- int fprintf_F (FILE *stream, const __flash char *fmt,...)
- int fputs_F (const __flash char *str, FILE *stream)
- int puts_F (const __flash char *str)
- int vfscanf_F (FILE *stream, const __flash char *fmt, va_list ap)
- int fscanf_F (FILE *stream, const __flash char *fmt,...)
- int scanf_F (const __flash char *fmt,...)
- int sscanf_F (const char *buf, const __flash char *fmt,...)

More efficient reading of 64-bit values from __flash and __flashx

- static uint64_t flash_read_u64 (const __flash uint64_t *addr)
- static int64_t flash_read_i64 (const __flash int64_t *addr)
- static double flash_read_double (const __flash double *addr)
- static long double flash_read_long_double (const __flash long double *addr)
- static uint64_t flashx_read_u64 (const __flashx uint64_t *addr)
- static int64_t flashx_read_i64 (const __flashx int64_t *addr)
- static double flashx_read_double (const __flashx double *addr)
- static long double flashx_read_long_double (const __flashx long double *addr)

Variables

AVR Named Address-Spaces

- __flash
- __flashx
- __memx

22.25 flash.h

Go to the documentation of this file.

```
00001 /* Copyright (c) 2002-2025 Joerg Wunsch
00002    All rights reserved.
00003
00004    Portions of documentation Copyright (c) 1990, 1991, 1993
00005    The Regents of the University of California.
00006
00007    All rights reserved.
00008
00009    Redistribution and use in source and binary forms, with or without
00010    modification, are permitted provided that the following conditions are met:
00011
00012    * Redistributions of source code must retain the above copyright
00013      notice, this list of conditions and the following disclaimer.
00014
00015    * Redistributions in binary form must reproduce the above copyright
00016      notice, this list of conditions and the following disclaimer in
```

```
00017      the documentation and/or other materials provided with the
00018      distribution.
00019
00020    * Neither the name of the copyright holders nor the names of
00021      contributors may be used to endorse or promote products derived
00022      from this software without specific prior written permission.
00023
00024  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
00025  AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
00026  IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
00027  ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
00028  LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
00029  CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
00030  SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
00031  INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
00032  CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
00033  ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
00034  POSSIBILITY OF SUCH DAMAGE. */
00035
00036 #ifndef _AVR_FLASH_H_
00037 #define _AVR_FLASH_H_
00038
00039 #if !defined(__AVR_TINY__) && !defined(__cplusplus)
00040
00041 #ifdef __DOXYGEN__
00042 /** \file */
00043 /** \defgroup avr_flash <avr/flash.h>: Utilities for named address-spaces __flash and
      __flashx
00044
00045     \since AVR-LibC v2.3
00046
00047     \code
00048     #include <avr/flash.h>
00049     \endcode
00050
00051     The functions and macros in this module provide interfaces for a program
00052     to use data stored in program space (flash memory) by means of the
00053     #__flash and #__flashx named address-spaces
00054     supported by avr-gcc.
00055
00056     <h3>Purpose</h3>
00057     The prototypes and macros provided by this header allow to write
00058     C programs that are address-space correct, i.e. they will compile
00059     without diagnostics due to <tt>-Waddr-space-convert</tt>.
00060
00061     For example, you can call #printf_P with a format string located
00062     in RAM resulting in non-functional code, and the compiler won't complain.
00063     This is different with #printf_F which will trigger a diagnostic when
00064     <tt>-Waddr-space-convert</tt> is on, and you feed in a format string that
00065     does not carry the #__flash named address-space qualifier.
00066
00067     <h3>Structure of this Header</h3>
00068
00069     This header provides:
00070
00071     - Functions from \c stdio.h, \c string.h resp. \c avr/pgmspace.h, but
00072     where one input string resides in the 16-bit address-space #__flash.
00073     For example, the #strlen_F function works the same like the "progmem"
00074     function #strlen_P but uses a prototype that describes the flash string
00075     as <tt>const __flash char*</tt>.
00076
00077     - Functions from \c string.h resp. \c avr/pgmspace.h, but where one
00078     input string resides in 24-bit address-space #__flashx.
00079     For example, the #strlen_FX function works the same like the "far"
00080     function #strlen_PF but uses a prototype that describes the flash string
00081     as <tt>const __flashx char*</tt>.
00082
00083     - Some macros for convenience.
00084
00085     <h3>Examples</h3>
00086
00087     \code
00088 #include <stdbool.h>
```

```
00089 #include <avr/flash.h>
00090
00091 // Array of string literals where the array
00092 // and also the literals reside in __flash.
00093 const __flash char* const __flash pets[] =
00094 {
00095     FLIT("gnu"), FLIT("cat"), FLIT("bat"), FLIT("rat")
00096 };
00097
00098 void test_pet (const char *pet, const __flash char *what)
00099 {
00100     const __flash char *yesno;
00101     bool is_what = ! strcmp_F (pet, what);
00102     yesno = is_what ? FSTR("yes") : FSTR("no");
00103
00104     // %S denotes a string in lower 64 KiB flash.
00105     printf_FSTR ("%s is a %S? %S!\n", pet, what, yesno);
00106 }
00107
00108 int main (void)
00109 {
00110     char pet[] = "cat";
00111     for (size_t i = 0; i < sizeof (pets) / sizeof (*pets); ++i)
00112     {
00113         pet[0] ^= 1;
00114         test_pet (pet, pets[i]);
00115     }
00116   return 0;
00117 }
00118     \endcode
00119     It will print
00120 \verbatim
00121 bat is a gnu? no!
00122 cat is a cat? yes!
00123 bat is a bat? yes!
00124 cat is a rat? no!
00125 \endverbatim
00126     provided stdout has been set up appropriately.
00127
00128     <h3>Efficiency</h3>
00129
00130     The internal handling of 64-bit values in avr-gcc is such that it splits
00131     them into single byte operations for the purpose of moving them around.
00132     This can lead to quite some overhead when reading such values from
00133     named address-spaces.  To that end, <tt>avr/flash.h</tt> provides some
00134     inline functions like #flash_read_u64 and #flashx_read_double that
00135     work similar to the #pgm_read_u64 and #pgm_read_double_far functions,
00136     but are coming with proper address-space qualification.
00137
00138     <h3>Limitations</h3>
00139
00140     - Named address-spaces are supported by avr-gcc as part of GNU-C
00141     (e.g. <tt>-std=gnu99</tt>).  They are not available in Standard C,
00142     and are not supported in C++.
00143
00144     - Address-spaces #__flash and #__memx are supported since
00145     avr-gcc v4.7 (Release 2012), whereas #__flashx is available
00146     since avr-gcc v15 (Release 2025).
00147
00148     - Named address-spaces are not supported for Reduced Tiny (core AVRrc,
00149     <tt>-mmcu=avrtiny</tt>), like ATtiny10, ATtiny102 or ATtiny40.
00150
00151     <h3>Further Reading</h3>
00152
00153     - <a
      href="https://gcc.gnu.org/onlinedocs/gcc/Named-Address-Spaces.html#AVR-Named-Address-Spaces"
00154     >avr-gcc: Named Address-Spaces</a>
00155 */
00156
00157 /** \name Macros */
00158
00159 /** \ingroup avr_flash
00160     \def FSTR(str)
```

```
00161
00162     Used to get a pointer to a static string in address-space #__flash.
00163     This macro can only be used in the context of a funcion, like in:
00164
00165     \code
00166     #include <avr/flash.h>
00167
00168     void say_hello (int x)
00169     {
00170         printf_F (FSTR ("Hello number %d\n"), x);
00171
00172         // Same effect, but more convenient.
00173         printf_FSTR ("Hello number %d\n", x);
00174     }
00175     \endcode
00176     #FSTR works similar to the #PSTR macro but returns a pointer to the 16-bit
00177     named address-space #__flash, whereas #PSTR returns a 16-bit
00178     address in the generic address-space (but isn't).
00179 */
00180 # define FSTR(str) ({ static const __flash char c[] = (str); &c[0]; })
00181
00182 /** \ingroup avr_flash
00183     Used to get a pointer to a static string in the 24-bit address-space
00184     #__flashx.
00185     This macro can only be used in the context of a funcion, like in:
00186
00187     \code
00188     #include <stdbool.h>
00189     #include <avr/flash.h>
00190
00191     bool text_contains_dog (const char *text)
00192     {
00193         return strstr_FX (text, FXSTR ("dog")) != NULL;
00194     }
00195     \endcode
00196     #FXSTR works similar to the #PSTR_FAR macro but returns a pointer to
00197     the 24-bit named address-space #__flashx, whereas #PSTR_FAR returns
00198     a 32-bit integer that represents an address in the generic address-space
00199     (but isn't).
00200 */
00201 # define FXSTR(str) ({ static const __flashx char c[] = (str); &c[0]; })
00202
00203 /** \ingroup avr_flash
00204     Turn string literal \p str into a compound literal in address-space
00205     #__flash. This macro can be used to construct arrays of string pointers:
00206     Suppose the following 2-dimensional array of animal names:
00207     \code
00208     // Each string occupies 13 bytes, hence
00209     // animals[] occupies 9 * 13 = 117 bytes.
00210     const __flash char animals[][13] =
00211     {
00212         "hippopotamus",
00213         "cat", "pig", "gnu", "bat",
00214         "dog", "cow", "fox", "rat"
00215     };
00216     \endcode
00217     A more memory-friendly way to represent the strings is an
00218     array of string \e pointers, like in:
00219     \code
00220     // Occupies 9*2 + 13 + 8*4 = 63 bytes
00221     const char* const animals2[] =
00222     {
00223         "hippopotamus",
00224         "cat", "pig", "gnu", "bat",
00225         "dog", "cow", "fox", "rat"
00226     };
00227     \endcode
00228     \c animals2[] occupies only 63 bytes (9 * 2 bytes for the string
00229     addresses plus the lengths of the very strings).  However, all
00230     objects are located in the generic address-space.  When all objects
00231     should be put in a non-generic address-space like #__flash,
00232     then the type of the literals has to be forced to be
00233     <tt>const __flash char[]</tt>.  This can be accomplished with #FLIT:
```

```
00234     \code
00235     // Occupies 9*2 + 13 + 8*4 = 63 bytes
00236     const __flash char* const __flash animals3[] =
00237     {
00238         FLIT("hippopotamus"),
00239         FLIT("cat"), FLIT("pig"), FLIT("gnu"), FLIT("bat"),
00240         FLIT("dog"), FLIT("cow"), FLIT("fox"), FLIT("rat")
00241     };
00242     \endcode
00243     Notice the two #__flash's in the declarator: The left one says that the
00244     pointed-to strings are in #__flash, whereas the right one says that
00245     the \c animals3[] array itself resides in #__flash.
00246
00247     Unfortunately, to date (avr-gcc v15), #FLIT can only be used at
00248     global scope.  Though what works in a function is using
00249     \c constexpr as introduced in C23:
00250     \code
00251     void func (int i)
00252     {
00253         static constexpr __flash char s_hip[] = "hippopotamus";
00254         static constexpr __flash char s_cat[] = "cat";
00255         // ...
00256         static const __flash char* const __flash animals4[] =
00257         {
00258             s_hip, s_cat, // ...
00259         };
00260         printf_FSTR ("Animal %d = %S\n", i, animals4[i]);
00261     }
00262     \endcode  */
00263 #define FLIT(str) ((const __flash char[]) { str })
00264
00265 /** \ingroup avr_flash
00266     Turn string literal \p str into a compound literal in address-space
00267     #__flashx. This macro can be used to construct arrays of string pointers.
00268
00269     For example code and usage, see the #FLIT macro.  */
00270 #define FXLIT(str) ((const __flashx char[]) { str })
00271
00272 /** \name Functions from string.h, but one argument is in address-space __flash */
00273
00274 /** \ingroup avr_flash
00275     \fn const __flash void * memchr_F(const __flash void *s, int val, size_t len)
00276     \brief Scan flash memory for a character.
00277
00278     The #memchr_F function scans the first \p len bytes of the flash
00279     memory area pointed to by \p s for the character \p val.  The first
00280     byte to match \p val (interpreted as an unsigned character) stops
00281     the operation.
00282
00283     \return The #memchr_F function returns a pointer to the matching
00284     byte or \c NULL if the character does not occur in the given memory
00285     area.  */
00286 extern const __flash void * memchr_F(const __flash void *, int, size_t);
00287
00288 /** \ingroup avr_flash
00289     \fn int memcmp_F(const void *s1, const __flash void *s2, size_t len)
00290     \brief Compare memory areas
00291
00292     The #memcmp_F function compares the first \p len bytes of the memory
00293     areas \p s1 and flash \p s2. The comparision is performed using unsigned
00294     char operations.
00295
00296     \returns The #memcmp_F function returns an integer less than, equal
00297     to, or greater than zero if the first \p len bytes of \p s1 is found,
00298     respectively, to be less than, to match, or be greater than the first
00299     \p len bytes of \p s2.  */
00300 extern int memcmp_F(const void *, const __flash void *, size_t);
00301
00302 /** \ingroup avr_flash
00303     \fn void *memccpy_F(void *dest, const __flash void *src, int val, size_t len)
00304
00305     This function is similar to #memccpy except that \p src is pointer
00306     to a string in address-space #__flash.  */
```

```
00307 extern void *memccpy_F(void *, const __flash void *, int val, size_t);
00308
00309 /** \ingroup avr_flash
00310     \fn void *memcpy_F(void *dest, const __flash void *src, size_t n)
00311
00312     The #memcpy_F function is similar to #memcpy, except the src string
00313     resides in address-space __flash.
00314
00315     \returns The #memcpy_F function returns a pointer to dest.  */
00316 extern void *memcpy_F(void *, const __flash void *, size_t);
00317
00318 /** \ingroup avr_flash
00319     \fn void *memmem_F(const void *s1, size_t len1, const __flash void *s2, size_t len2)
00320
00321     The #memmem_F function is similar to #memmem except that \p s2 is
00322     pointer to a string in address-space #__flash.  */
00323 extern void *memmem_F(const void *, size_t, const __flash void *, size_t);
00324
00325 /** \ingroup avr_flash
00326     \fn const __flash void *memrchr_F(const __flash void *src, int val, size_t len)
00327
00328     The #memrchr_F function is like the #memchr_F function, except
00329     that it searches backwards from the end of the \p len bytes pointed
00330     to by \p src instead of forwards from the front. (Glibc, GNU extension.)
00331
00332     \return The #memrchr_F function returns a pointer to the matching
00333     byte or \c NULL if the character does not occur in the given memory
00334     area.  */
00335 extern const __flash void * memrchr_F(const __flash void *, int val, size_t len);
00336
00337
00338 /** \ingroup avr_flash
00339     \fn size_t strlen_F(const __flash char *src)
00340
00341     The #strlen_F function is similar to #strlen, except that \p src is a
00342     pointer to a string in address-space #__flash.
00343
00344     \returns The #strlen_F function returns the number of characters in \p src.
00345
00346     \note #strlen_F is implemented as an inline function in the avr/flash.h
00347     header file, which will check if the length of the string is a constant
00348     and known at compile time. If it is not known at compile time, the macro
00349     will issue a call to a libc function which will then calculate the length
00350     of the string as usual.  */
00351 static inline size_t strlen_F(const __flash char *src);
00352
00353 /** \ingroup avr_flash
00354     \fn char *strcat_F(char *dest, const __flash char *src)
00355
00356     The #strcat_F function is similar to #strcat except that the \p src
00357     string must be located in address-space #__flash.
00358
00359     \returns The #strcat function returns a pointer to the resulting string
00360     \p dest. */
00361 extern char *strcat_F(char *, const __flash char *);
00362
00363 /** \ingroup avr_flash
00364     \fn const __flash char *strchr_F(const __flash char *s, int val)
00365     \brief Locate character in a string in address-space #__flash.
00366
00367     The #strchr_F function locates the first occurrence of \p val
00368     (converted to a char) in the string pointed to by \p s in address-space
00369     #__flash. The terminating NULL character is considered to be part of
00370     the string.
00371
00372     The #strchr_F function is similar to #strchr except that \p s is
00373     pointer to a string in address-space #__flash.
00374
00375     \returns The #strchr_F function returns a pointer to the matched
00376     character or \c NULL if the character is not found. */
00377 extern const __flash char * strchr_F(const __flash char *, int val);
00378
00379 /** \ingroup avr_flash
```

```
00380     \fn const __flash char *strchrnul_F(const __flash char *s, int c)
00381
00382     The #strchrnul_F function is like #strchr_F except that if \p c is
00383     not found in \p s, then it returns a pointer to the NULL byte at the
00384     end of \p s, rather than \c NULL. (Glibc, GNU extension.)
00385
00386     \return The #strchrnul_F function returns a pointer to the matched
00387     character, or a pointer to the NULL byte at the end of \p s (i.e.,
00388     \c s+strlen(s)) if the character is not found.  */
00389 extern const __flash char * strchrnul_F(const __flash char *, int val);
00390
00391 /** \ingroup avr_flash
00392     \fn int strcmp_F(const char *s1, const __flash char *s2)
00393
00394     The #strcmp_F function is similar to #strcmp except that \p s2 is
00395     pointer to a string in address-space #__flash.
00396
00397     \returns The #strcmp_F function returns an integer less than, equal
00398     to, or greater than zero if \p s1 is found, respectively, to be less
00399     than, to match, or be greater than \p s2. A consequence of the
00400     ordering used by #strcmp_F is that if \p s1 is an initial substring
00401     of \p s2, then \p s1 is considered to be "less than" \p s2. */
00402 extern inline int strcmp_F(const char *, const __flash char *);
00403
00404 /** \ingroup avr_flash
00405     \fn char *strcpy_F(char *dest, const __flash char *src)
00406
00407     The #strcpy_F function is similar to #strcpy except that src is a
00408     pointer to a string in address-space #__flash.
00409
00410     \returns The #strcpy_F function returns a pointer to the destination
00411     string dest. */
00412 extern inline char *strcpy_F(char *, const __flash char *);
00413
00414 /** \ingroup avr_flash
00415     \fn char *stpcpy_F(char *dest, const __flash char *src)
00416
00417     The #stpcpy_F function is similar to #stpcpy except that \p src is a
00418     pointer to a string in address-space #__flash.
00419
00420     \returns #stpcpy_F returns a pointer to the <b>end</b> of
00421     the string \p dest (that is, the address of the terminating null byte)
00422     rather than the beginning.  */
00423 extern inline char *stpcpy_F(char *, const __flash char *);
00424
00425 /** \ingroup avr_flash
00426     \fn int strcasecmp_F(const char *s1, const __flash char *s2)
00427     \brief Compare two strings ignoring case.
00428
00429     The #strcasecmp_F function compares the two strings \p s1 and \p s2,
00430     ignoring the case of the characters.
00431
00432     \param s1 A pointer to a string in SRAM.
00433     \param s2 A pointer to a string in address-space #__flash.
00434
00435     \returns The #strcasecmp_F function returns an integer less than,
00436     equal to, or greater than zero if \p s1 is found, respectively, to
00437     be less than, to match, or be greater than \p s2. A consequence of
00438     the ordering used by #strcasecmp_F is that if \p s1 is an initial
00439     substring of \p s2, then \p s1 is considered to be "less than" \p s2. */
00440 extern int strcasecmp_F(const char *, const __flash char *);
00441
00442 /** \ingroup avr_flash
00443     \fn char *strcasestr_F(const char *s1, const __flash char *s2)
00444
00445     This function is similar to #strcasestr except that \p s2 is pointer
00446     to a string in address-space #__flash.  */
00447 extern char *strcasestr_F(const char *, const __flash char *);
00448
00449 /** \ingroup avr_flash
00450     \fn size_t strcspn_F(const char *s, const __flash char *reject)
00451
00452     The #strcspn_F function calculates the length of the initial segment
```

```
00453     of \p s which consists entirely of characters not in \p reject. This
00454     function is similar to #strcspn except that \p reject is a pointer
00455     to a string in address-space #__flash.
00456
00457     \return The #strcspn_F function returns the number of characters in
00458     the initial segment of \p s which are not in the string \p reject.
00459     The terminating zero is not considered as a part of string.  */
00460 extern size_t strcspn_F(const char *s, const __flash char *reject);
00461
00462 /** \ingroup avr_flash
00463     \fn size_t strlcat_F(char *dst, const __flash char *src, size_t siz)
00464     \brief Concatenate two strings.
00465
00466     The #strlcat_F function is similar to #strlcat, except that the \p src
00467     string must be located in address-space #__flash.
00468
00469     Appends \p src to string \p dst of size \p siz (unlike #strncat,
00470     \p siz is the full size of \p dst, not space left).  At most \p siz-1
00471     characters will be copied.  Always NULL terminates (unless \p siz <=
00472     \p strlen(dst)).
00473
00474     \returns The #strlcat_F function returns strlen(src) + MIN(siz,
00475     strlen(initial dst)).  If retval >= siz, truncation occurred.  */
00476 extern size_t strlcat_F(char *, const __flash char *, size_t);
00477
00478 /** \ingroup avr_flash
00479     \fn size_t strlcpy_F(char *dst, const __flash char *src, size_t siz)
00480     \brief Copy a string from address-space #__flash to RAM.
00481
00482     Copy \p src to string \p dst of size \p siz.  At most \p siz-1
00483     characters will be copied.  Always NULL terminates (unless \p siz == 0).
00484     The #strlcpy_F function is similar to #strlcpy except that the
00485     \p src is pointer to a string in address-space #__flash.
00486
00487     \returns The #strlcpy_F function returns strlen(src). If
00488     retval >= siz, truncation occurred.  */
00489 extern size_t strlcpy_F(char *, const __flash char *, size_t);
00490
00491 /** \ingroup avr_flash
00492     \fn size_t strnlen_F(const __flash char *src, size_t len)
00493     \brief Determine the length of a fixed-size string.
00494
00495     The #strnlen_F function is similar to #strnlen, except that \c src is a
00496     pointer to a string in address-space #__flash.
00497
00498     \returns The #strnlen_F function returns strlen_F(src), if that is less than
00499     \c len, or \c len if there is no '\\0' character among the first \c len
00500     characters pointed to by \c src. */
00501 extern size_t strnlen_F(const __flash char *, size_t);
00502
00503 /** \ingroup avr_flash
00504     \fn int strncmp_F(const char *s1, const __flash char *s2, size_t n)
00505
00506     The #strncmp_F function is similar to #strcmp_F except it only compares
00507     the first (at most) n characters of s1 and s2.
00508
00509     \returns The #strncmp_F function returns an integer less than, equal to,
00510     or greater than zero if s1 (or the first n bytes thereof) is found,
00511     respectively, to be less than, to match, or be greater than s2.  */
00512 extern int strncmp_F(const char *, const __flash char *, size_t);
00513
00514 /** \ingroup avr_flash
00515     \fn int strncasecmp_F(const char *s1, const __flash char *s2, size_t n)
00516     \brief Compare two strings ignoring case.
00517
00518     The #strncasecmp_F function is similar to #strcasecmp_F, except it
00519     only compares the first \p n characters of \p s1.
00520
00521     \param s1 A pointer to a string in SRAM.
00522     \param s2 A pointer to a string in address-space #__flash.
00523     \param n The maximum number of bytes to compare.
00524
00525     \returns The #strncasecmp_F function returns an integer less than,
```

```
00526     equal to, or greater than zero if \p s1 (or the first \p n bytes
00527     thereof) is found, respectively, to be less than, to match, or be
00528     greater than \p s2. A consequence of the ordering used by
00529     #strncasecmp_F is that if \p s1 is an initial substring of \p s2,
00530     then \p s1 is considered to be "less than" \p s2.  */
00531 extern int strncasecmp_F(const char *, const __flash char *, size_t);
00532
00533 /** \ingroup avr_flash
00534     \fn char *strncat_F(char *dest, const __flash char *src, size_t len)
00535     \brief Concatenate two strings.
00536
00537     The #strncat_F function is similar to #strncat, except that the \p src
00538     string must be located in address-space #__flash.
00539
00540     \returns The #strncat_F function returns a pointer to the resulting string
00541     dest.  */
00542 extern char *strncat_F(char *, const __flash char *, size_t);
00543
00544 /** \ingroup avr_flash
00545     \fn char *strncpy_F(char *dest, const __flash char *src, size_t n)
00546
00547     The #strncpy_F function is similar to #strcpy_F except that not more
00548     than \p n bytes of src are copied.  Thus, if there is no null byte among
00549     the first \p n bytes of \p src, the result will not be null-terminated.
00550
00551     In the case where the length of \p src is less than that of \p n,
00552     the remainder of \p dest will be padded with nulls.
00553
00554     \returns The #strncpy_F function returns a pointer to the destination
00555     string dest.  */
00556 extern char *strncpy_F(char *, const __flash char *, size_t);
00557
00558 /** \ingroup avr_flash
00559     \fn char *strpbrk_F(const char *s, const __flash char *accept)
00560
00561     The #strpbrk_F function locates the first occurrence in the string
00562     \p s of any of the characters in the #__flash string \p accept. This
00563     function is similar to #strpbrk except that \p accept is a pointer
00564     to a string in address-space #__flash.
00565
00566     \return  The #strpbrk_F function returns a pointer to the character
00567     in \p s that matches one of the characters in \p accept, or \c NULL
00568     if no such character is found. The terminating zero is not considered
00569     as a part of string: If one or both args are empty, the result will
00570     \c NULL. */
00571 extern char *strpbrk_F(const char *, const __flash char * accept);
00572
00573 /** \ingroup avr_flash
00574     \fn const __flash char *strrchr_F(const __flash char *s, int val)
00575     \brief Locate character in string.
00576
00577     The #strrchr_F function returns a pointer to the last occurrence of
00578     the character \p val in the #__flash string \p s.
00579
00580     \return The #strrchr_F function returns a pointer to the matched
00581     character or \c NULL if the character is not found. */
00582 extern const __flash char * strrchr_F(const __flash char *, int val);
00583
00584 /** \ingroup avr_flash
00585     \fn char *strsep_F(char **sp, const __flash char *delim)
00586     \brief Parse a string into tokens.
00587
00588     The #strsep_F function locates, in the string referenced by \p *sp,
00589     the first occurrence of any character in the string \p delim (or the
00590     terminating '\\0' character) and replaces it with a '\\0'.  The
00591     location of the next character after the delimiter character (or \c
00592     NULL, if the end of the string was reached) is stored in \p *sp. An
00593     "empty" field, i.e. one caused by two adjacent delimiter
00594     characters, can be detected by comparing the location referenced by
00595     the pointer returned in \p *sp to '\\0'. This function is similar to
00596     #strsep except that \p delim is a pointer to a string in address-space
00597     #__flash.
00598
```

```
00599     \return The #strsep_F function returns a pointer to the original
00600     value of \p *sp. If \p *sp is initially \c NULL, #strsep_F returns
00601     \c NULL. */
00602 extern char *strsep_F(char **sp, const __flash char * delim);
00603
00604 /** \ingroup avr_flash
00605     \fn size_t strspn_F(const char *s, const __flash char *accept)
00606
00607     The #strspn_F function calculates the length of the initial segment
00608     of \p s which consists entirely of characters in \p accept. This
00609     function is similar to #strspn except that \p accept is a pointer
00610     to a string in address-space #__flash.
00611
00612     \return  The #strspn_F function returns the number of characters in
00613     the initial segment of \p s which consist only of characters from \p
00614     accept. The terminating zero is not considered as a part of string.  */
00615 extern size_t strspn_F(const char *s, const __flash char * accept);
00616
00617 /** \ingroup avr_flash
00618     \fn char *strstr_F(const char *s1, const __flash char *s2)
00619     \brief Locate a substring.
00620
00621     The #strstr_F function finds the first occurrence of the substring
00622     \p s2 in the string \p s1.  The terminating '\\0' characters are not
00623     compared. The #strstr_F function is similar to #strstr except that
00624     \p s2 is pointer to a string in address-space #__flash.
00625
00626     \returns The #strstr_F function returns a pointer to the beginning
00627     of the substring, or NULL if the substring is not found. If \p s2
00628     points to a string of zero length, the function returns \p s1. */
00629 extern char *strstr_F(const char *, const __flash char *);
00630
00631 /** \ingroup avr_flash
00632     \fn char *strtok_F(char *s, const __flash char * delim)
00633     \brief Parses the string into tokens.
00634
00635     #strtok_F parses the string \p s into tokens. The first call to
00636     #strtok_F should have \p s as its first argument. Subsequent calls
00637     should have the first argument set to NULL. If a token ends with a
00638     delimiter, this delimiting character is overwritten with a '\\0' and a
00639     pointer to the next character is saved for the next call to #strtok_F.
00640     The delimiter string \p delim may be different for each call.
00641
00642     The #strtok_F function is similar to #strtok except that \p delim
00643     is pointer to a string in address-space #__flash.
00644
00645     \returns The #strtok_F function returns a pointer to the next token or
00646     NULL when no more tokens are found.
00647
00648     \note #strtok_F is NOT reentrant. For a reentrant version of this
00649     function see #strtok_rF.
00650  */
00651 extern char *strtok_F(char *s, const __flash char * delim);
00652
00653 /** \ingroup avr_flash
00654     \fn char *strtok_rF(char *string, const __flash char *delim, char **last)
00655     \brief Parses string into tokens.
00656
00657     The #strtok_rF function parses \p string into tokens. The first call to
00658     #strtok_rF should have \p string as its first argument. Subsequent calls
00659     should have the first argument set to NULL. If a token ends with a
00660     delimiter, this delimiting character is overwritten with a '\\0' and a
00661     pointer to the next character is saved for the next call to #strtok_rF.
00662     The delimiter string \p delim may be different for each call. \p last is
00663     a user allocated <tt>char*</tt> pointer. It must be the same while
00664     parsing the same string. #strtok_rF is a reentrant version of #strtok_F.
00665
00666     The #strtok_rF function is similar to #strtok_r except that \p delim
00667     is pointer to a string in address-space #__flash.
00668
00669     \returns The #strtok_rF function returns a pointer to the next token or
00670     NULL when no more tokens are found. */
00671 extern char *strtok_rF(char *s, const __flash char * delim, char **last);
```

```
00672
00673
00674 /** \name Functions from string.h, but one argument is in 24-bit address-space __flashx
      */
00675
00676 /** \ingroup avr_flash
00677     \fn void *memcpy_FX(void *dest, const __flashx void *src, size_t n)
00678     \brief Copy a memory block from address-space __flashx to SRAM
00679
00680     The #memcpy_FX function is similar to #memcpy, except the data
00681     is copied from address-space #__flashx and is addressed using an
00682     according pointer.
00683
00684     \param dest A pointer to the destination buffer.
00685     \param src A pointer to the origin of data in #__flashx.
00686     \param n The number of bytes to be copied.
00687
00688     \returns The #memcpy_FX function returns a pointer to \e dst. */
00689 extern void *memcpy_FX(void *dest, const __flashx void *src, size_t len);
00690
00691 /** \ingroup avr_flash
00692     \fn int memcmp_FX(const void *s1, const __flashx void *s2, size_t len)
00693     \brief Compare memory areas
00694
00695     The #memcmp_FX function compares the first \p len bytes of the memory
00696     areas \p s1 and __flashx \p s2. The comparision is performed using unsigned
00697     char operations. It is an equivalent of #memcmp_F function, except
00698     that it is capable working on all Flash including the extended area
00699     above 64 KiB.
00700
00701     \returns The #memcmp_FX function returns an integer less than, equal
00702     to, or greater than zero if the first \p len bytes of \p s1 is found,
00703     respectively, to be less than, to match, or be greater than the first
00704     \p len bytes of \p s2.  */
00705 extern int memcmp_FX(const void *s1, const __flashx void *s2, size_t);
00706
00707 /** \ingroup avr_flash
00708     \fn size_t strlen_FX(const __flashx char *s)
00709     \brief Obtain the length of a string located in address-space #__flashx
00710
00711     The #strlen_FX function is similar to #strlen, except that \e s is a
00712     pointer to a string in address-space #__flashx.
00713
00714     \returns The #strlen_FX function returns the number of characters in
00715     \e s. */
00716 extern size_t strlen_FX(const __flashx char *src);
00717
00718 /** \ingroup avr_flash
00719     \fn size_t strnlen_FX(const __flashx char *s, size_t len)
00720     \brief Determine the length of a fixed-size string in address-space __flashx
00721
00722     The #strnlen_FX function is similar to #strnlen, except that \e s is a
00723     pointer to a string in address-space #__flashx.
00724
00725     \param s The address of a string in #__flashx.
00726     \param len The maximum number of length to return.
00727
00728     \returns The #strnlen_FX function returns strlen_FX(\e s), if that is less
00729     than \e len, or \e len if there is no '\\0' character among the first \e
00730     len characters pointed to by \e s.  */
00731 extern size_t strnlen_FX(const __flashx char *src, size_t len);
00732
00733 /** \ingroup avr_flash
00734     \fn char *strcpy_FX(char *dst, const __flashx char *src)
00735     \brief Duplicate a string from address-space __flashx
00736
00737     The #strcpy_FX function is similar to #strcpy except that \e src is a
00738     string located in address-space #__flashx.
00739
00740     \param dst A pointer to the destination string in SRAM.
00741     \param src A pointer to the source string in #__flashx.
00742
00743     \returns The #strcpy_FX function returns a pointer to the destination
```

```
00744     string \e dst. */
00745 extern char *strcpy_FX(char *dest, const __flashx char *src);
00746
00747 /** \ingroup avr_flash
00748     \fn char *stpcpy_FX(char *dst, const __flashx char *src)
00749     \brief Duplicate a string from address-space __flashx
00750
00751     The #stpcpy_FX function is similar to #stpcpy except that \e src
00752     is a string located in address-space #__flashx.
00753
00754     \param dst A pointer to the destination string in SRAM.
00755     \param src A pointer to the source string in #__flashx.
00756
00757     \returns The stpcpy_PF() function returns a pointer to the
00758     terminating '\\0' character of the destination string \e dst.  */
00759 extern char *stpcpy_FX(char *dest, const __flashx char *src);
00760
00761 /** \ingroup avr_flash
00762     \fn char *strncpy_FX(char *dst, const __flashx char *src, size_t n)
00763     \brief Duplicate a string from address-space __flashx until a limited length
00764
00765     The #strncpy_FX function is similar to #strcpy_FX except that not more
00766     than \e n bytes of \e src are copied.  Thus, if there is no null byte among
00767     the first \e n bytes of \e src, the result will not be null-terminated.
00768
00769     In the case where the length of \e src is less than that of \e n, the
00770     remainder of \e dst will be padded with nulls.
00771
00772     \param dst A pointer to the destination string in SRAM.
00773     \param src A far pointer to the source string in address-space #__flashx.
00774     \param n The maximum number of bytes to copy.
00775
00776     \returns The #strncpy_FX function returns a pointer to the destination
00777     string \e dst. */
00778 extern char *strncpy_FX(char *dest, const __flashx char *src, size_t len);
00779
00780 /** \ingroup avr_flash
00781     \fn char *strcat_FX(char *dst, const __flashx char *src)
00782     \brief Concatenates two strings
00783
00784     The #strcat_FX function is similar to #strcat except that the \e src
00785     string must be located in address-space #__flashx.
00786
00787     \param dst A pointer to the destination string in SRAM.
00788     \param src A pointer to the string located in #__flashx to be appended.
00789
00790     \returns The #strcat_FX function returns a pointer to the resulting
00791     string \e dst. */
00792 extern char *strcat_FX(char *dest, const __flashx char *src);
00793
00794 /** \ingroup avr_flash
00795     \fn size_t strlcat_FX(char *dst, const __flashx char *src, size_t n)
00796     \brief Concatenate two strings
00797
00798     The #strlcat_FX function is similar to #strlcat, except that the \e src
00799     string must be located in address-space #__flashx.
00800
00801     Appends src to string dst of size \e n (unlike #strncat, \e n is the
00802     full size of \e dst, not space left).  At most \e n-1 characters
00803     will be copied.  Always NULL terminates (unless \e n <= strlen(\e dst)).
00804
00805     \param dst A pointer to the destination string in SRAM.
00806     \param src A pointer to the source string in __flashx.
00807     \param n The total number of bytes allocated to the destination string.
00808
00809     \returns The #strlcat_FX function returns strlen(\e src) + MIN(\e n,
00810     strlen(initial \e dst)).  If retval >= \e n, truncation occurred. */
00811 extern size_t strlcat_FX(char *dst, const __flashx char *src, size_t siz);
00812
00813 /** \ingroup avr_flash
00814     \fn char *strncat_FX(char *dst, const __flashx char *src, size_t n)
00815     \brief Concatenate two strings
00816
```

```
00817     The #strncat_FX function is similar to #strncat, except that the \e src
00818     string must be located in address-space __flashx.
00819
00820     \param dst A pointer to the destination string in SRAM.
00821     \param src A pointer to the source string in __flashx.
00822     \param n The maximum number of bytes to append.
00823
00824     \returns The #strncat_FX function returns a pointer to the resulting
00825     string \e dst. */
00826 extern char *strncat_FX(char *dest, const __flashx char *src, size_t len);
00827
00828 /** \ingroup avr_flash
00829     \fn int strcmp_FX(const char *s1, const __flashx char *s2)
00830     \brief Compares two strings
00831
00832     The #strcmp_FX function is similar to #strcmp except that \e s2 is a
00833     pointer to a string in address-space #__flashx.
00834
00835     \param s1 A pointer to the first string in SRAM.
00836     \param s2 A pointer to the second string in #__flashx.
00837
00838     \returns The #strcmp_FX function returns an integer less than, equal to,
00839     or greater than zero if \e s1 is found, respectively, to be less than, to
00840     match, or be greater than \e s2. */
00841 extern int strcmp_FX(const char *s1, const __flashx char *s2);
00842
00843 /** \ingroup avr_flash
00844     \fn int strncmp_FX(const char *s1, const __flashx char *s2, size_t n)
00845     \brief Compare two strings with limited length
00846
00847     The #strncmp_FX function is similar to #strcmp_FX except it only
00848     compares the first (at most) \e n characters of \e s1 and \e s2.
00849
00850     \param s1 A pointer to the first string in SRAM.
00851     \param s2 A pointer to the second string in address-space #__flashx.
00852     \param n The maximum number of bytes to compare.
00853
00854     \returns The #strncmp_FX function returns an integer less than, equal
00855     to, or greater than zero if \e s1 (or the first \e n bytes thereof) is found,
00856     respectively, to be less than, to match, or be greater than \e s2. */
00857 extern int strncmp_FX(const char *s1, const __flashx char *s2, size_t n);
00858
00859 /** \ingroup avr_flash
00860     \fn int strcasecmp_FX(const char *s1, const __flashx char *s2)
00861     \brief Compare two strings ignoring case
00862
00863     The #strcasecmp_FX function compares the two strings \e s1 and \e s2,
00864     ignoring the case of the characters.
00865
00866     \param s1 A pointer to the first string in SRAM.
00867     \param s2 A pointer to the second string in address-space #__flashx.
00868
00869     \returns The #strcasecmp_FX function returns an integer less than, equal
00870     to, or greater than zero if \e s1 is found, respectively, to be less than,
00871     to match, or be greater than \e s2. */
00872 extern int strcasecmp_FX(const char *s1, const __flashx char *s2);
00873
00874 /** \ingroup avr_flash
00875     \fn int strncasecmp_FX(const char *s1, const __flashx char *s2, size_t n)
00876     \brief Compare two strings ignoring case
00877
00878     The #strncasecmp_FX function is similar to #strcasecmp_FX, except it
00879     only compares the first \e n characters of \e s1, and the string \e s2
00880     is located in address-space #__flashx.
00881
00882     \param s1 A pointer to a string in SRAM.
00883     \param s2 A pointer to a string in #__flashx.
00884     \param n The maximum number of bytes to compare.
00885
00886     \returns The #strncasecmp_FX function returns an integer less than, equal
00887     to, or greater than zero if \e s1 (or the first \e n bytes thereof) is found,
00888     respectively, to be less than, to match, or be greater than \e s2. */
00889 extern int strncasecmp_FX(const char *s1, const __flashx char *s2, size_t n);
```

```
00890
00891 /** \ingroup avr_flash
00892     \fn const __flashx char *strchr_FX(const __flashx char *s, int val)
00893     \brief Locate a character in a string located in address-space #__flashx
00894
00895     The #strchr_FX function locates the first occurrence of \p val
00896     (converted to a char) in the string pointed to by \p s in address-space
00897     #__flashx. The terminating null character is considered to be part of
00898     the string.
00899
00900     The #strchr_FX function is similar to #strchr except that \p s is
00901     a pointer to a string in address-space #__flashx that's \e not
00902     \e required to be located in the lower 64 KiB block like it is
00903     the case  for #strchr_F.
00904
00905     \returns The #strchr_FX function returns a far pointer to the matched
00906     character or \c 0 if the character is not found. */
00907 extern const __flashx char *strchr_FX(const __flashx char *s, int val);
00908
00909 /** \ingroup avr_flash
00910     \fn char *strstr_FX(const char *s1, const __flashx char *s2)
00911     \brief Locate a substring.
00912
00913     The #strstr_FX function finds the first occurrence of the substring \c s2
00914     in the string \c s1.  The terminating '\\0' characters are not
00915     compared.
00916     The #strstr_FX function is similar to #strstr except that \c s2 points
00917     to a string located in address-space #__flashx.
00918
00919     \returns The #strstr_FX function returns a pointer to the beginning of the
00920     substring, or NULL if the substring is not found.
00921     If \c s2 points to a string of zero length, the function returns \c s1. */
00922 extern char *strstr_FX(const char *s1, const __flashx char *s2);
00923
00924 /** \ingroup avr_flash
00925     \fn size_t strlcpy_FX(char *dst, const __flashx char *src, size_t len)
00926     \brief Copy a string from address-space #__flashx to RAM.
00927
00928     Copy src to string dst of length len.  At most len-1 characters will be
00929     copied. Always NULL terminates (unless len == 0).
00930
00931     \returns The #strlcpy_FX function returns strlen_FX(src).
00932     If retval >= len, truncation occurred.  */
00933 extern size_t strlcpy_FX(char *, const __flashx char *, size_t);
00934
00935 /** \name AVR Named Address-Spaces */
00936
00937 /** \ingroup avr_flash
00938     A <b>named address-space</b> for data in the lower 64 KiB of program memory
00939
00940     Pointers to #__flash are 16 bits wide.
00941
00942     Objects in #__flash are located in section
00943     \ref sec_dot_progmem ".progmem.data",
00944     which is located \e prior to the code sections by the default linker
00945     description files.  Similar to #PROGMEM, the assertion is that all
00946     objects in that section will fit in the lower 64 KiB flash segment
00947     (flash byte addresses 0x0 ... 0xffff).
00948
00949     The compiler defines the built-in macro \c __FLASH when this
00950     address-space is available.  It is supported as part of GNU-C
00951     (<tt>-std=gnu99</tt> etc.), and is not available on Reduced
00952     Tiny (core AVRrc).
00953
00954     \since <a href="https://gcc.gnu.org/gcc-4.7/changes.html#avr">avr-gcc v4.7</a>
00955     (Release 2012)
00956
00957     \see <a
      href="https://gcc.gnu.org/onlinedocs/gcc/Named-Address-Spaces.html#index-_005f_005fflash-AVR-Named-Addre
00958     >avr-gcc: <tt>__flash</tt></a>.
00959 */
00960 __flash;
00961
```

```
00962 /** \ingroup avr_flash
00963     A <b>named address-space</b> for data in program memory
00964
00965     Pointers to #__flashx are 24 bits wide.
00966
00967     Objects in #__flashx are located in section
00968     \ref sec_dot_progmemx ".progmemx.data",
00969     which is located \e after the code sections by the default linker
00970     description files.  There is no restriction on the address range
00971     occupied by objects in that section, and #__flashx supports
00972     reading across the 64 KiB segment boundaries.
00973
00974     The compiler defines the built-in macro \c __FLASHX when this
00975     address-space is available.  It is supported as part of GNU-C
00976     (<tt>-std=gnu99</tt> etc.), and is not available on Reduced
00977     Tiny (core AVRrc).
00978
00979     \since
00980     <a href="https://gcc.gnu.org/gcc-15/changes.html#avr">avr-gcc v15</a>
00981     (Release 2025)
00982
00983     \see <a
      href="https://gcc.gnu.org/onlinedocs/gcc/Named-Address-Spaces.html#index-_005f_005fflashx-AVR-Named-Addr
00984     >avr-gcc: <tt>__flashx</tt></a>.
00985 */
00986 __flashx;
00987
00988
00989 /** \ingroup avr_flash
00990     A <b>named address-space</b> for data in program memory or RAM
00991
00992     Pointers to #__memx are 24 bits wide.
00993
00994     Objects in #__memx are located in section
00995     \ref sec_dot_progmemx ".progmemx.data",
00996     which is located \e after the code sections by the default linker
00997     description files.  There is no restriction on the address range
00998     occupied by objects in that section, and #__memx supports
00999     reading across the 64 KiB flash segment boundaries.
01000
01001     #__memx pointers can also hold RAM addresses, and reading from
01002     #__memx reads from RAM or from program memory depending on the
01003     most significant bit of the address.
01004
01005     The compiler defines the built-in macro \c __MEMX when this
01006     address-space is available.  It is supported as part of GNU-C
01007     (<tt>-std=gnu99</tt> etc.), and is not available on Reduced
01008     Tiny (core AVRrc).
01009     To date, AVR-LibC does not have support for functions operating
01010     on #__memx.
01011
01012     \since
01013     <a href="https://gcc.gnu.org/gcc-4.7/changes.html#avr">avr-gcc v4.7</a>
01014     (Release 2012)
01015
01016     \see <a
      href="https://gcc.gnu.org/onlinedocs/gcc/Named-Address-Spaces.html#index-_005f_005fmemx-AVR-Named-Addres
01017     >avr-gcc: <tt>__memx</tt></a>.
01018 */
01019 __memx;
01020
01021
01022 #endif /* __DOXYGEN__ */
01023
01024 /** \name Convenience macros for functions from stdio.h, that allocate the format string
      with FSTR */
01025
01026 /** \ingroup avr_flash
01027     A convenience macro that wraps #vfprintf_F's format string with #FSTR.
01028 */
01029 #define vfprintf_FSTR(stream, fmt, ap) vfprintf_F(stream, FSTR(fmt), ap)
01030
01031 /** \ingroup avr_flash
```

```
01032     A convenience macro that wraps #printf_F's format string with #FSTR.
01033 */
01034 #define printf_FSTR(fmt, ...) printf_F(FSTR(fmt), ##__VA_ARGS__)
01035
01036 /** \ingroup avr_flash
01037     A convenience macro that wraps #sprintf_F's format string with #FSTR.
01038 */
01039 #define sprintf_FSTR(s, fmt, ...) sprintf_F(s, FSTR(fmt), ##__VA_ARGS__)
01040
01041 /** \ingroup avr_flash
01042     A convenience macro that wraps #snprintf_F's format string with #FSTR.
01043 */
01044 #define snprintf_FSTR(s, n, fmt, ...) snprintf_F(s, n, FSTR(fmt), ##__VA_ARGS__)
01045
01046 /** \ingroup avr_flash
01047     A convenience macro that wraps #vsprintf_F's format string with #FSTR.
01048 */
01049 #define vsprintf_FSTR(s, fmt, ap) vsprintf_F(s, FSTR(fmt), ap)
01050
01051 /** \ingroup avr_flash
01052     A convenience macro that wraps #vsnprintf_F's format string with #FSTR.
01053 */
01054 #define vsnprintf_FSTR(s, n, fmt, ap) vsnprintf_F(s, n, FSTR(fmt), ap);
01055
01056 /** \ingroup avr_flash
01057     A convenience macro that wraps #fprintf_F's format string with #FSTR.
01058 */
01059 #define fprintf_FSTR(stream, fmt, ...) fprintf_F(stream, FSTR(fmt), ##__VA_ARGS__)
01060
01061 /** \ingroup avr_flash
01062     A convenience macro that wraps #fputs_F's string with #FSTR.
01063 */
01064 #define fputs_FSTR(str, stream) fputs_F(FSTR(str), stream);
01065
01066 /** \ingroup avr_flash
01067     A convenience macro that wraps #puts_F's string with #FSTR.
01068 */
01069 #define puts_FSTR(str) puts_F(FSTR(str));
01070
01071 /** \ingroup avr_flash
01072     A convenience macro that wraps #vfscanf_F's format string with #FSTR.
01073 */
01074 #define vfscanf_FSTR(stream, fmt, ap) vfscanf_F(stream, FSTR(fmt), ap);
01075
01076 /** \ingroup avr_flash
01077     A convenience macro that wraps #fscanf_F's format string with #FSTR.
01078 */
01079 #define fscanf_FSTR(stream, fmt, ...) fscanf_F(stream, FSTR(fmt), ##__VA_ARGS__)
01080
01081 /** \ingroup avr_flash
01082     A convenience macro that wraps #scanf_F's format string with #FSTR.
01083 */
01084 #define scanf_FSTR(fmt, ...) scanf_F(FSTR(fmt), ##__VA_ARGS__)
01085
01086 /** \ingroup avr_flash
01087     A convenience macro that wraps #sscanf_F's format string with #FSTR.
01088 */
01089 #define sscanf_FSTR(buf, fmt, ...) sscanf_F(buf, FSTR(fmt), ##__VA_ARGS__)
01090
01091 #ifdef __DOXYGEN__
01092
01093 /** \name Functions from stdio.h, but with a format string in address-space __flash */
01094
01095 /** \ingroup avr_flash
01096     Variant of \c #vfprintf that uses a \c fmt string that resides
01097     in address-space #__flash.  See also #vfprintf_FSTR.
01098 */
01099 extern int vfprintf_F(FILE *stream, const __flash char *fmt, va_list ap);
01100
01101 /** \ingroup avr_flash
01102     Variant of \c #printf that uses a \c fmt string that resides
01103     in address-space #__flash.  See also #printf_FSTR.
01104 */
```

```
01105 extern int printf_F(const __flash char *fmt, ...);
01106
01107 /** \ingroup avr_flash
01108     Variant of \c #sprintf that uses a \c fmt string that resides
01109     in address-space #__flash.  See also #sprintf_FSTR.
01110 */
01111 extern int sprintf_F(char *s, const __flash char *fmt, ...);
01112
01113 /** \ingroup avr_flash
01114     Variant of \c #snprintf that uses a \c fmt string that resides
01115     in address-space #__flash.  See also #snprintf_FSTR.
01116 */
01117 extern int snprintf_F(char *s, size_t n, const __flash char *fmt, ...);
01118
01119 /** \ingroup avr_flash
01120     Variant of \c #vsprintf that uses a \c fmt string that resides
01121     in address-space #__flash.  See also #vsprintf_FSTR.
01122 */
01123 extern int vsprintf_F(char *s, const __flash char *fmt, va_list ap);
01124
01125 /** \ingroup avr_flash
01126     Variant of \c #vsnprintf that uses a \c fmt string that resides
01127     in address-space #__flash.  See also #vsnprintf_FSTR.
01128 */
01129 extern int vsnprintf_F(char *s, size_t n, const __flash char *fmt, va_list ap);
01130
01131 /** \ingroup avr_flash
01132     Variant of \c #fprintf that uses a \c fmt string that resides
01133     in address-space #__flash.  See also #fprintf_FSTR.
01134 */
01135 extern int fprintf_F(FILE *stream, const __flash char *fmt, ...);
01136
01137 /** \ingroup avr_flash
01138     Variant of #fputs where \c str resides in address-space #__flash.
01139     See also #fputs_FSTR.
01140 */
01141 extern int fputs_F(const __flash char *str, FILE *stream);
01142
01143 /** \ingroup avr_flash
01144     Variant of #puts where \c str resides in address-space #__flash.
01145     See also #puts_FSTR.
01146 */
01147 extern int puts_F(const __flash char *str);
01148
01149 /** \ingroup avr_flash
01150     Variant of #vfscanf using a \c fmt string in address-space #__flash.
01151     See also #vfscanf_FSTR.
01152 */
01153 extern int vfscanf_F(FILE *stream, const __flash char *fmt, va_list ap);
01154
01155 /** \ingroup avr_flash
01156     Variant of #fscanf using a \c fmt string in address-space #__flash.
01157     See also #fscanf_FSTR.
01158 */
01159 extern int fscanf_F(FILE *stream, const __flash char *fmt, ...);
01160
01161 /** \ingroup avr_flash
01162     Variant of #scanf where \c fmt resides in address-space #__flash.
01163     See also #scanf_FSTR.
01164 */
01165 extern int scanf_F(const __flash char *fmt, ...);
01166
01167 /** \ingroup avr_flash
01168     Variant of #sscanf using a \c fmt string in address-space #__flash.
01169     See also #sscanf_FSTR.
01170 */
01171 extern int sscanf_F(const char *buf, const __flash char *fmt, ...);
01172
01173
01174 /** \name More efficient reading of 64-bit values from __flash and __flashx */
01175
01176 /** \ingroup avr_flash
01177     \fn uint64_t flash_read_u64 (const __flash uint64_t *addr)
```

```
01178     Read an <tt>uint64_t</tt> from 16-bit #__flash address \p addr. */
01179 static inline uint64_t flash_read_u64 (const __flash uint64_t *addr);
01180
01181 /** \ingroup avr_flash
01182     \fn int64_t flash_read_i64 (const __flash int64_t *addr)
01183     Read an <tt>int64_t</tt> from 16-bit #__flash address \p addr. */
01184 static inline int64_t flash_read_i64 (const __flash int64_t *addr);
01185
01186 /** \ingroup avr_flash
01187     \fn double flash_read_double (const __flash double *addr)
01188     Read a <tt>double</tt> from 16-bit __flash address \p addr. */
01189 static inline double flash_read_double (const __flash double *addr);
01190
01191 /** \ingroup avr_flash
01192     \fn long double flash_read_long_double (const __flash long double *addr)
01193     Read a <tt>long double</tt> from 16-bit #__flash address \p addr. */
01194 static inline long double flash_read_long_double (const __flash long double *addr);
01195
01196
01197 /** \ingroup avr_flash
01198     \fn uint64_t flashx_read_u64 (const __flashx uint64_t *addr)
01199     Read an <tt>uint64_t</tt> from 24-bit #__flashx address \p addr. */
01200 static inline uint64_t flashx_read_u64 (const __flashx uint64_t *addr);
01201
01202 /** \ingroup avr_flash
01203     \fn int64_t flashx_read_i64 (const __flashx int64_t *addr)
01204     Read an <tt>int64_t</tt> from 24-bit #__flashx address \p addr. */
01205 static inline int64_t flashx_read_i64 (const __flashx int64_t *addr);
01206
01207 /** \ingroup avr_flash
01208     \fn double flashx_read_double (const __flashx double *addr)
01209     Read a <tt>double</tt> from 24-bit #__flashx address \p addr. */
01210 static inline double flashx_read_double (const __flashx double *addr);
01211
01212 /** \ingroup avr_flash
01213     \fn long double flashx_read_long_double (const __flashx long double *addr)
01214     Read a <tt>long double</tt> from 24-bit #__flashx address \p addr. */
01215 static inline long double flashx_read_long_double (const __flashx long double *addr);
01216
01217
01218 /* ************************************************************************/
01219
01220
01221 #else /* !__DOXYGEN__ */
01222
01223 #define __need_size_t
01224 #include <stddef.h>
01225 #include <bits/attribs.h>
01226
01227 #ifdef __FLASH
01228
01229 #define FSTR(s) (__extension__({static const __flash char __c[] = (s); &__c[0];}))
01230 #define FLIT(lit) ((const __flash char[]) { lit })
01231
01232 extern const __flash void * memchr_F(const __flash void *, int __val, size_t __len)
      __asm("memchr_P") __ATTR_CONST__;
01233 extern int memcmp_F(const void *, const __flash void *, size_t) __asm("memcmp_P")
      __ATTR_PURE__;
01234 extern void *memccpy_F(void *, const __flash void *, int __val, size_t)
      __asm("memccpy_P");
01235 extern void *memcpy_F(void *, const __flash void *, size_t) __asm("memcpy_P");
01236 extern void *memmem_F(const void *, size_t, const __flash void *, size_t)
      __asm("memmem_P") __ATTR_PURE__;
01237 extern const __flash void * memrchr_F(const __flash void *, int __val, size_t __len)
      __asm("memrchr_P") __ATTR_CONST__;
01238
01239 extern char *strcat_F(char *, const __flash char *) __asm("strcat_P");
01240
01241 extern const __flash char * strchr_F(const __flash char *, int __val) __asm("strchr_P")
      __ATTR_CONST__;
01242 extern __ATTR_ALWAYS_INLINE__ __ATTR_GNU_INLINE__
01243 const __flash char * strchr_F(const __flash char *__hay, int __val)
01244 {
```

```
01245   register const __flash char *__r24 __asm("24") = __hay;
01246   register int __r22 __asm("22") = __val;
01247   __asm ("%~call strchr_P"
01248          : "+r" (__r24) : "r" (__r22) : "0", "30", "31");
01249   return __r24;
01250 }
01251
01252 extern const __flash char * strchrnul_F(const __flash char *, int __val)
      __asm("strchrnul_P") __ATTR_CONST__;
01253 extern int strcasecmp_F(const char *, const __flash char *) __asm("strcasecmp_P")
      __ATTR_PURE__;
01254 extern char *strcasestr_F(const char *, const __flash char *) __asm("strcasestr_P")
      __ATTR_PURE__;
01255 extern size_t strcspn_F(const char *, const __flash char * __reject) __asm("strcspn_P")
      __ATTR_PURE__;
01256 extern size_t strnlen_F(const __flash char *, size_t) __asm("strnlen_P") __ATTR_CONST__;
01257 extern int strncmp_F(const char *, const __flash char *, size_t) __asm("strncmp_P")
      __ATTR_PURE__;
01258 extern int strncasecmp_F(const char *, const __flash char *, size_t)
      __asm("strncasecmp_P") __ATTR_PURE__;
01259 extern char *strncat_F(char *, const __flash char *, size_t) __asm("strncat_P");
01260 extern char *strncpy_F(char *, const __flash char *, size_t) __asm("strncpy_P");
01261 extern char *strpbrk_F(const char *__s, const __flash char * __accept)
      __asm("strpbrk_P") __ATTR_PURE__;
01262 extern const __flash char * strrchr_F(const __flash char *, int __val)
      __asm("strrchr_P") __ATTR_CONST__;
01263 extern char *strsep_F(char **__sp, const __flash char * __delim) __asm("strsep_P");
01264 extern size_t strspn_F(const char *__s, const __flash char * __accept) __asm("strspn_P")
      __ATTR_PURE__;
01265 extern char *strstr_F(const char *, const __flash char *) __asm("strstr_P")
      __ATTR_PURE__;
01266 extern char *strtok_F(char *, const __flash char * __delim) __asm("strtok_P");
01267 extern char *strtok_rF(char *, const __flash char * __delim, char **__last)
      __asm("strtok_rP");
01268
01269 /* memcpy_F is common so we model its GPR footprint.  */
01270 extern __ATTR_ALWAYS_INLINE__ __ATTR_GNU_INLINE__
01271 void* memcpy_F(void *__x, const __flash void *__z, size_t __s)
01272 {
01273   register size_t __r20 __asm("20") = __s;
01274   void *__ret = __x;
01275   __asm volatile ("%~call __memcpy_P" : "+x" (__x), "+z" (__z), "+r" (__r20)
01276                   :: "0", "memory");
01277   return __ret;
01278 }
01279
01280 /* strcmp_F is common so we model strcmp_P's GPR footprint. */
01281 extern int strcmp_F (const char*, const __flash char*) __asm ("strcmp_P");
01282 extern __ATTR_ALWAYS_INLINE__ __ATTR_GNU_INLINE__
01283 int strcmp_F (const char *__x, const __flash char *__z)
01284 {
01285   register int __ret __asm("24");
01286   __asm ("%~call __strcmp_P"
01287          : "=r" (__ret), "+x" (__x), "+z" (__z) :: "memory");
01288   return __ret;
01289 }
01290
01291
01292 /* strcpy_F is common so we model strcpy_P's GPR footprint. */
01293 extern char* strcpy_F (char *__x, const __flash char *__z) __asm("strcpy_P");
01294 extern __ATTR_ALWAYS_INLINE__ __ATTR_GNU_INLINE__
01295 char* strcpy_F (char *__x, const __flash char *__z)
01296 {
01297   char *__ret = __x;
01298   __asm volatile ("%~call __strcpy_P"
01299                   : "+x" (__x), "+z" (__z) :: "0", "memory");
01300   return __ret;
01301 }
01302
01303
01304 extern char* stpcpy_F (char *__x, const __flash char *__z) __asm("stpcpy_P");
01305 extern __ATTR_ALWAYS_INLINE__ __ATTR_GNU_INLINE__
01306 char* stpcpy_F (char *__x, const __flash char *__z)
```

```
01307 {
01308   __asm volatile ("%~call __strcpy_P"
01309                   : "+x" (__x), "+z" (__z) :: "0", "memory");
01310   return __x - 1;
01311 }
01312
01313
01314 /* strlen_F is common so we model strlen_P's GPR footprint. */
01315 extern size_t strlen_F (const __flash char *__s) __asm("strlen_P");
01316 extern __ATTR_ALWAYS_INLINE__ __ATTR_GNU_INLINE__
01317 size_t strlen_F (const __flash char *__s)
01318 {
01319 #ifdef __BUILTIN_AVR_STRLEN_FLASH
01320   if (__builtin_constant_p (__builtin_avr_strlen_flash (__s)))
01321     return __builtin_avr_strlen_flash (__s);
01322 #endif
01323   {
01324     register const __flash char *__r24 __asm("24") = __s;
01325     register size_t __res __asm("24");
01326     __asm ("%~call strlen_P" : "=r" (__res) : "r" (__r24)
01327            : "0", "30", "31");
01328     return __res;
01329   }
01330 }
01331
01332
01333 #include <stdio.h> /* FILE */
01334
01335 extern int vfprintf_F(FILE *__stream, const __flash char *__fmt, va_list __ap)
      __asm("vfprintf_P");
01336 extern int printf_F(const __flash char *__fmt, ...) __asm("printf_P");
01337 extern int sprintf_F(char *__s, const __flash char *__fmt, ...) __asm("sprintf_P");
01338 extern int snprintf_F(char *__s, size_t __n, const __flash char *__fmt, ...)
      __asm("snprintf_P");
01339 extern int vsprintf_F(char *__s, const __flash char *__fmt, va_list ap)
      __asm("vsprintf_P");
01340 extern int vsnprintf_F(char *__s, size_t __n, const __flash char *__fmt, va_list ap)
      __asm("vsnprintf_P");
01341 extern int fprintf_F(FILE *__stream, const __flash char *__fmt, ...) __asm("fprintf_P");
01342 extern int fputs_F(const __flash char *__str, FILE *__stream) __asm("fputs_P");
01343 extern int puts_F(const __flash char *__str) __asm("puts_P");
01344 extern int vfscanf_F(FILE *__stream, const __flash char *__fmt, va_list __ap)
      __asm("vfscanf_P");
01345 extern int fscanf_F(FILE *__stream, const __flash char *__fmt, ...) __asm("fscanf_P");
01346 extern int scanf_F(const __flash char *__fmt, ...) __asm("scanf_P");
01347 extern int sscanf_F(const char *__buf, const __flash char *__fmt, ...)
      __asm("sscanf_P");
01348 #endif /* Have __flash */
01349
01350 #ifdef __FLASHX
01351
01352 #define FXSTR(s) (__extension__({static const __flashx char __c[] = (s); &__c[0];}))
01353 #define FXLIT(lit) ((const __flashx char[]) { lit })
01354
01355 extern size_t strnlen_FX(const __flashx char *, size_t) __asm("strnlen_PF")
      __ATTR_CONST__;
01356 extern void *memcpy_FX(void *, const __flashx void *, size_t) __asm("memcpy_PF");
01357 extern char *strcpy_FX(char *, const __flashx char *) __asm("strcpy_PF");
01358 extern char *stpcpy_FX(char *, const __flashx char *) __asm("stpcpy_PF");
01359 extern char *strncpy_FX(char *, const __flashx char *, size_t) __asm("strncpy_PF");
01360 extern char *strcat_FX(char *, const __flashx char *) __asm("strcat_PF");
01361 extern size_t strlcat_FX(char *, const __flashx char *, size_t) __asm("strlcat_PF");
01362 extern char *strncat_FX(char *, const __flashx char *, size_t) __asm("strncat_PF");
01363 extern int strcmp_FX(const char *, const __flashx char *) __asm("strcmp_PF")
      __ATTR_PURE__;
01364 extern int strncmp_FX(const char *, const __flashx char *, size_t) __asm("strncmp_PF")
      __ATTR_PURE__;
01365 extern int strcasecmp_FX(const char *, const __flashx char *) __asm("strcasecmp_PF")
      __ATTR_PURE__;
01366 extern int strncasecmp_FX(const char *, const __flashx char *, size_t)
      __asm("strncasecmp_PF") __ATTR_PURE__;
01367 extern const __flashx char *strchr_FX(const __flashx char *, int) __asm("strchr_PF")
      __ATTR_CONST__;
```

```
01368 extern char *strstr_FX(const char *, const __flashx char *) __asm("strstr_PF");
01369 extern size_t strlcpy_FX(char *, const __flashx char *, size_t) __asm("strlcpy_PF");
01370 extern int memcmp_FX(const void *, const __flashx void *, size_t) __asm("memcmp_PF")
      __ATTR_PURE__;
01371
01372 #ifdef __BUILTIN_AVR_STRLEN_FLASHX
01373 extern size_t __strlen_FX(const __flashx char*) __asm("strlen_PF") __ATTR_CONST__;
01374
01375 static inline __ATTR_ALWAYS_INLINE__ size_t
01376 strlen_FX (const __flashx char *__s)
01377 {
01378   return __builtin_constant_p (__builtin_avr_strlen_flashx (__s))
01379     ? __builtin_avr_strlen_flashx (__s)
01380     : __strlen_FX (__s);
01381 }
01382 #else
01383 extern size_t strlen_FX(const __flashx char*) __asm("strlen_PF") __ATTR_CONST__;
01384 #endif /* Have __builtin_avr_strlen_flashx */
01385
01386 #endif /* Have __flashx */
01387
01388
01389 #ifdef __FLASH
01390
01391 #include <stdint.h>
01392 #include <bits/lpm-elpm.h>
01393 #include <bits/def-flash-read.h>
01394
01395 #if __SIZEOF_LONG_LONG__ == 8
01396 _Avrlibc_Def_F_8 (u64, uint64_t)
01397 _Avrlibc_Def_F_8 (i64, int64_t)
01398 #endif
01399
01400 #if __SIZEOF_DOUBLE__ == 8
01401 _Avrlibc_Def_F_8 (double, double)
01402 #else
01403 _Avrlibc_Def_F_4 (double, double)
01404 #endif
01405
01406 #if __SIZEOF_LONG_DOUBLE__ == 8
01407 _Avrlibc_Def_F_8 (long_double, long double)
01408 #else
01409 _Avrlibc_Def_F_4 (long_double, long double)
01410 #endif
01411
01412 #endif /* Have __flash */
01413
01414 #ifdef __FLASHX
01415
01416 #include <bits/lpm-elpm.h>
01417
01418 #if defined(__AVR_HAVE_ELPM__)
01419 #define __ELPM__8fx(r,a,T) __ELPM__8(r,a,T)
01420 #else
01421 #define __ELPM__8fx(r,a,T)                          \
01422   uintptr_t __a = (uintptr_t)(uintptr24_t) a;    \
01423   __LPM__8(r,__a)
01424 #endif
01425
01426
01427 #define _Avrlibc_Def_FX_4(Name, Typ)                    \
01428   static __ATTR_ALWAYS_INLINE__                         \
01429   Typ flashx_read_##Name (const __flashx Typ *__addr)   \
01430   {                                                     \
01431     return *__addr;                                     \
01432   }
01433
01434 #define _Avrlibc_Def_FX_8(Name, Typ)                    \
01435   static __ATTR_ALWAYS_INLINE__                         \
01436   Typ flashx_read_##Name (const __flashx Typ *__addr)   \
01437   {                                                     \
01438     Typ __res;                                          \
01439     __ELPM__8fx (__res, __addr, Typ);                   \
```

```
01440     return __res;                                         \
01441   }
01442
01443 #if __SIZEOF_LONG_LONG__ == 8
01444 _Avrlibc_Def_FX_8 (u64, uint64_t)
01445 _Avrlibc_Def_FX_8 (i64, int64_t)
01446 #endif
01447
01448 #if __SIZEOF_DOUBLE__ == 8
01449 _Avrlibc_Def_FX_8 (double, double)
01450 #else
01451 _Avrlibc_Def_FX_4 (double, double)
01452 #endif
01453
01454 #if __SIZEOF_LONG_DOUBLE__ == 8
01455 _Avrlibc_Def_FX_8 (long_double, long double)
01456 #else
01457 _Avrlibc_Def_FX_4 (long_double, long double)
01458 #endif
01459
01460 #endif /* Have __flashx */
01461
01462 #endif /* !DOXYGEN */
01463
01464 #endif /* !__AVR_TINY__ && !C++ */
01465
01466 #endif /* _AVR_FLASH_H_ */
```

22.26 fuse.h File Reference

Macros

- #define FUSEMEM __attribute__((__used__, __section__ (".fuse")))
- #define FUSES

22.27 fuse.h

Go to the documentation of this file.

```
00001 /* Copyright (c) 2007, Atmel Corporation
00002    All rights reserved.
00003
00004    Redistribution and use in source and binary forms, with or without
00005    modification, are permitted provided that the following conditions are met:
00006
00007    * Redistributions of source code must retain the above copyright
00008      notice, this list of conditions and the following disclaimer.
00009
00010    * Redistributions in binary form must reproduce the above copyright
00011      notice, this list of conditions and the following disclaimer in
00012      the documentation and/or other materials provided with the
00013      distribution.
00014
00015    * Neither the name of the copyright holders nor the names of
00016      contributors may be used to endorse or promote products derived
00017      from this software without specific prior written permission.
00018
00019   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
00020   AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
00021   IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
00022   ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
00023   LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
00024   CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
00025   SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
00026   INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
00027   CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
```

```
00028   ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
00029   POSSIBILITY OF SUCH DAMAGE. */
00030
00031 /* avr/fuse.h - Fuse API */
00032
00033 #ifndef _AVR_FUSE_H_
00034 #define _AVR_FUSE_H_ 1
00035
00036 /* This file must be explicitly included by <avr/io.h>. */
00037 #if !defined(_AVR_IO_H_)
00038 #error "You must #include <avr/io.h> and not <avr/fuse.h> by itself."
00039 #endif
00040
00041
00042 /** \file */
00043 /** \defgroup avr_fuse <avr/fuse.h>: Fuse Support
00044     \code #include <avr/io.h> \endcode
00045
00046     The <avr/fuse.h> header is included by <avr/io.h>.
00047
00048     \par Introduction
00049
00050     The Fuse API allows a user to specify the fuse settings for the specific
00051     AVR device they are compiling for. These fuse settings will be placed
00052     in a special section in the ELF output file, after linking.
00053
00054     Programming tools can take advantage of the fuse information embedded in
00055     the ELF file, by extracting this information and determining if the fuses
00056     need to be programmed before programming the Flash and EEPROM memories.
00057     This also allows a single ELF file to contain all the
00058     information needed to program an AVR.
00059
00060     To use the Fuse API, include the <avr/io.h> header file, which in turn
00061     automatically includes the individual I/O header file and the <avr/fuse.h>
00062     file. These other two files provides everything necessary to set the AVR
00063     fuses.
00064
00065     \par Fuse API
00066
00067     Each I/O header file must define the \c FUSE_MEMORY_SIZE macro which is
00068     defined to the number of fuse bytes that exist in the AVR device.
00069
00070     A new type, __fuse_t, is defined as a structure. The number of fields in
00071     this structure are determined by the number of fuse bytes in the
00072     \c FUSE_MEMORY_SIZE macro:
00073
00074     - \c If FUSE_MEMORY_SIZE == 1, there is only a single field: byte, of type
00075     #uint8_t.
00076
00077     - If \c FUSE_MEMORY_SIZE == 2, there are two fields: low, and high, of type
00078     #uint8_t.
00079
00080     - If FUSE_MEMORY_SIZE == 3, there are three fields: low, high, and extended,
00081     of type #uint8_t.
00082
00083     - If \c FUSE_MEMORY_SIZE > 3, there is a single field: byte, which is an
00084     array of #uint8_t with the size of the array being \c FUSE_MEMORY_SIZE.
00085
00086     A convenience macro, \c #FUSEMEM, is defined as a GCC attribute for a
00087     custom-named section of \c ".fuse".
00088
00089     A convenience macro, \c #FUSES, is defined that declares a variable,
00090     \c __fuse, of type \c __fuse_t with the attribute defined by \c #FUSEMEM.
00091     This variable allows the end user to easily set the fuse data.
00092
00093     \note If a device-specific I/O header file has previously defined
00094     \c FUSEMEM, then \c FUSEMEM is not redefined. If a device-specific
00095     I/O header file has previously defined \c FUSES, then
00096     \c FUSES is not redefined.
00097
00098     Each AVR device I/O header file has a set of defined macros which specify
00099     the actual fuse bits available on that device. The AVR fuses have inverted
00100     values, logical 1 for an unprogrammed (disabled) bit and logical 0 for a
```

```
00101     programmed (enabled) bit. The defined macros for each individual fuse
00102     bit represent this in their definition by a bit-wise inversion of a mask.
00103     For example, the \c FUSE_EESAVE fuse in the ATmega128 is defined as:
00104     \code
00105     #define FUSE_EESAVE  ~_BV(3)
00106     \endcode
00107     The \c #_BV macro creates a bit mask from a bit number. It is then
00108     inverted to represent logical values for a fuse memory byte.
00109     To combine the fuse bits macros together to represent a whole fuse byte,
00110     use the bitwise AND operator, like so:
00111     \code
00112     (FUSE_BOOTSZ0 & FUSE_BOOTSZ1 & FUSE_EESAVE & FUSE_SPIEN & FUSE_JTAGEN)
00113     \endcode
00114
00115     \warning Many device headers define fuse macros for <b>not inverted</b>
00116     fuse bits, like for example devices from the 0-series, 1-series and
00117     2-series.  Make sure you are using the right logic operations when
00118     using fuse values, or otherwise you can damage a device.
00119
00120     Each device I/O header file also defines macros that provide default values
00121     for each fuse byte that is available. \c LFUSE_DEFAULT is defined for a Low
00122     Fuse byte. \c HFUSE_DEFAULT is defined for a High Fuse byte.
00123     \c EFUSE_DEFAULT is defined for an Extended Fuse byte.
00124
00125     If \c FUSE_MEMORY_SIZE > 3, then the I/O header file defines macros that
00126     provide default values for each fuse byte like so:
00127     \code
00128     FUSE0_DEFAULT
00129     FUSE1_DEFAULT
00130     FUSE2_DEFAULT
00131     FUSE3_DEFAULT
00132     FUSE4_DEFAULT
00133     ...
00134     \endcode
00135
00136     \par API Usage Example
00137
00138     Putting all of this together is easy. Using C99's designated initializers:
00139
00140     \code
00141     #include <avr/io.h>
00142
00143     FUSES =
00144     {
00145         .low = LFUSE_DEFAULT,
00146         .high = FUSE_BOOTSZ0 & FUSE_BOOTSZ1 & FUSE_EESAVE & FUSE_SPIEN & FUSE_JTAGEN,
00147         .extended = EFUSE_DEFAULT
00148     };
00149     \endcode
00150
00151     Or, using the variable directly instead of the FUSES macro,
00152
00153     \code
00154     #include <avr/io.h>
00155
00156     __fuse_t __fuse FUSEMEM =
00157     {
00158         .low = LFUSE_DEFAULT,
00159         .high = FUSE_BOOTSZ0 & FUSE_BOOTSZ1 & FUSE_EESAVE & FUSE_SPIEN & FUSE_JTAGEN,
00160         .extended = EFUSE_DEFAULT
00161     };
00162     \endcode
00163
00164     If you are compiling in C++, you cannot use the designated initializers so
00165     you must do:
00166
00167     \code
00168     #include <avr/io.h>
00169
00170     FUSES =
00171     {
00172         LFUSE_DEFAULT, // .low
00173         FUSE_BOOTSZ0 & FUSE_BOOTSZ1 & FUSE_EESAVE & FUSE_SPIEN & FUSE_JTAGEN, // .high
```

```
00174         EFUSE_DEFAULT  // .extended
00175     };
00176     \endcode
00177
00178     However there are a number of caveats that you need to be aware of to
00179     use this API properly.
00180
00181     Be sure to include <avr/io.h> to get all of the definitions for the API.
00182     The FUSES macro defines a global variable to store the fuse data. This
00183     variable is assigned to its own linker section. Assign the desired fuse
00184     values immediately in the variable initialization.
00185
00186     The .fuse section in the ELF file will get its values from the initial
00187     variable assignment ONLY. This means that you can NOT assign values to
00188     this variable in functions and the new values will not be put into the
00189     ELF \c \.fuse section.
00190
00191     The global variable is declared in the \c FUSES macro has two leading
00192     underscores, which means that it is reserved for the "implementation",
00193     meaning the library, so it will not conflict with a user-named variable.
00194
00195     You must initialize ALL fields in the \c __fuse_t structure. This is because
00196     the fuse bits in all bytes default to a logical 1, meaning unprogrammed.
00197     Normal uninitialized data defaults to all logical zeros. So it is vital that
00198     all fuse bytes are initialized, even with default data. If they are not,
00199     then the fuse bits may not programmed to the desired settings.
00200
00201     Be sure to have the <tt>-mmcu=<em>device</em></tt> flag in your
00202     compile command line and
00203     your linker command line to have the correct device selected and to have
00204     the correct I/O header file included when you include <avr/io.h>.
00205
00206     You can print out the contents of the .fuse section in the ELF file by
00207     using this command line:
00208     \code
00209     avr-objdump -s -j .fuse <ELF file>
00210     \endcode
00211     The section contents shows the address on the left, then the data going from
00212     lower address to a higher address, left to right.
00213
00214 */
00215
00216 #if !defined(__ASSEMBLER__)
00217
00218 #include <stdint.h>
00219
00220 /** \ingroup avr_fuse  */
00221 #ifndef FUSEMEM
00222 #define FUSEMEM  __attribute__((__used__, __section__ (".fuse")))
00223 #endif
00224
00225 #ifdef __DOXYGEN__
00226 /** \ingroup avr_fuse
00227     A convenience macro. On Xmega devices, it is defined as
00228     \code
00229     #define FUSES NVM_FUSES_t __fuse FUSEMEM
00230     \endcode
00231     Otherwise, the definition is:
00232     \code
00233     #define FUSES __fuse_t __fuse FUSEMEM
00234     \endcode */
00235 #define FUSES
00236 #else /* Doxygen */
00237
00238 #if FUSE_MEMORY_SIZE > 3
00239
00240 typedef struct
00241 {
00242     uint8_t byte[FUSE_MEMORY_SIZE];
00243 } __fuse_t;
00244
00245
00246 #elif FUSE_MEMORY_SIZE == 3
```

```
00247
00248 typedef struct
00249 {
00250     uint8_t low;
00251     uint8_t high;
00252     uint8_t extended;
00253 } __fuse_t;
00254
00255 #elif FUSE_MEMORY_SIZE == 2
00256
00257 typedef struct
00258 {
00259     uint8_t low;
00260     uint8_t high;
00261 } __fuse_t;
00262
00263 #elif FUSE_MEMORY_SIZE == 1
00264
00265 typedef struct
00266 {
00267     uint8_t byte;
00268 } __fuse_t;
00269
00270 #endif
00271
00272 #if !defined(FUSES)
00273   #if defined(__AVR_XMEGA__)
00274     #define FUSES NVM_FUSES_t __fuse FUSEMEM
00275   #else
00276     #define FUSES __fuse_t __fuse FUSEMEM
00277   #endif
00278 #endif
00279
00280
00281 #endif /* !Doxygen */
00282 #endif /* !__ASSEMBLER__ */
00283
00284 #endif /* _AVR_FUSE_H_ */
```

22.28 interrupt.h File Reference

Macros

Global manipulation of the interrupt flag

The global interrupt flag is maintained in the I bit of the status register (SREG).

Handling interrupts frequently requires attention regarding atomic access to objects that could be altered by code running within an interrupt context, see <util/atomic.h>.

Frequently, interrupts are being disabled for periods of time in order to perform certain operations without being disturbed; see Problems with Reordering Code for things to be taken into account with respect to compiler optimizations.

- #define sei() __asm__ __volatile__ ("sei" ::: "memory")
- #define cli() __asm__ __volatile__ ("cli" ::: "memory")

Macros for writing interrupt handler functions

- #define ISR(vector, attributes)
- #define ISR_N(vector_num, attributes)
- #define SIGNAL(vector)
- #define EMPTY_INTERRUPT(vector)
- #define ISR_ALIAS(vector, target_vector)
- #define reti() __asm__ __volatile__ ("reti" ::: "memory")
- #define BADISR_vect

ISR attributes

- #define ISR_BLOCK
- #define ISR_NOBLOCK
- #define ISR_NAKED
- #define ISR_FLATTEN
- #define ISR_NOICF
- #define ISR_NOGCCISR
- #define ISR_ALIASOF(target_vector)

22.29 interrupt.h

Go to the documentation of this file.

```
00001 /* Copyright (c) 2002,2005,2007 Marek Michalkiewicz
00002    Copyright (c) 2007, Dean Camera
00003
00004    All rights reserved.
00005
00006    Redistribution and use in source and binary forms, with or without
00007    modification, are permitted provided that the following conditions are met:
00008
00009    * Redistributions of source code must retain the above copyright
00010      notice, this list of conditions and the following disclaimer.
00011
00012    * Redistributions in binary form must reproduce the above copyright
00013      notice, this list of conditions and the following disclaimer in
00014      the documentation and/or other materials provided with the
00015      distribution.
00016
00017    * Neither the name of the copyright holders nor the names of
00018      contributors may be used to endorse or promote products derived
00019      from this software without specific prior written permission.
00020
00021   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
00022   AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
00023   IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
00024   ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
00025   LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
00026   CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
00027   SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
00028   INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
00029   CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
00030   ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
00031   POSSIBILITY OF SUCH DAMAGE. */
00032
00033 #ifndef _AVR_INTERRUPT_H_
00034 #define _AVR_INTERRUPT_H_
00035
00036 #include <avr/io.h>
00037
00038 #if !defined(__DOXYGEN__) && !defined(__STRINGIFY)
00039 /* Auxiliary macro for ISR_ALIAS(). */
00040 #define __STRINGIFY(x) #x
00041 #endif /* !defined(__DOXYGEN__) */
00042
00043 /** \file */
00044 /**@{*/
00045
00046
00047 /** \name Global manipulation of the interrupt flag
00048
00049     The global interrupt flag is maintained in the I bit of the status
00050     register (SREG).
00051
00052     Handling interrupts frequently requires attention regarding atomic
00053     access to objects that could be altered by code running within an
00054     interrupt context, see <util/atomic.h>.
00055
```

```
00056     Frequently, interrupts are being disabled for periods of time in
00057     order to perform certain operations without being disturbed; see
00058     \ref optim_code_reorder for things to be taken into account with
00059     respect to compiler optimizations.
00060 */
00061
00062 /** \def sei()
00063     \ingroup avr_interrupts
00064
00065     Enables interrupts by setting the global interrupt mask. This function
00066     actually compiles into a single line of assembly, so there is no function
00067     call overhead.  However, the macro also implies a <i>memory barrier</i>
00068     which can cause additional loss of optimization.
00069
00070     In order to implement atomic access to multi-byte objects,
00071     consider using the macros from <util/atomic.h>, rather than
00072     implementing them manually with cli() and sei().
00073 */
00074 # define sei()  __asm__ __volatile__ ("sei" ::: "memory")
00075
00076 /** \def cli()
00077     \ingroup avr_interrupts
00078
00079     Disables all interrupts by clearing the global interrupt mask. This function
00080     actually compiles into a single line of assembly, so there is no function
00081     call overhead.  However, the macro also implies a <i>memory barrier</i>
00082     which can cause additional loss of optimization.
00083
00084     In order to implement atomic access to multi-byte objects,
00085     consider using the macros from <util/atomic.h>, rather than
00086     implementing them manually with cli() and sei().
00087 */
00088 # define cli()  __asm__ __volatile__ ("cli" ::: "memory")
00089
00090
00091 /** \name Macros for writing interrupt handler functions */
00092
00093
00094 #if defined(__DOXYGEN__)
00095 /** \def ISR(vector [, attributes])
00096     \ingroup avr_interrupts
00097
00098     Introduces an interrupt handler function (interrupt service
00099     routine) that runs with the \c SREG.I flag unchanged
00100     by default with no attributes specified.
00101     (On most devices this means that global interrupts are disabled
00102     upon servicing the IRQ.)
00103
00104     The \c attributes are optional and alter the behaviour and resultant
00105     generated code of the interrupt routine. Multiple attributes may
00106     be used for a single function, with a space separating each
00107     attribute.
00108
00109     Valid attributes are #ISR_BLOCK, #ISR_NOBLOCK, #ISR_NAKED,
00110     #ISR_FLATTEN, #ISR_NOICF, #ISR_NOGCCISR and ISR_ALIASOF(vect).
00111
00112     \c vector must be one of the
00113     \ref avr_mcu_signames "interrupt vector names" that are
00114     valid for the particular MCU type.
00115
00116     See also the #ISR_N macro for an alternative way to introduce an ISR.
00117 */
00118 #  define ISR(vector, [attributes])
00119 #else  /* real code */
00120
00121 #if defined (__clang__)
00122 #  define __INTR_ATTRS __used__
00123 #elif (__GNUC__ == 4 && __GNUC_MINOR__ >= 1) || (__GNUC__ > 4)
00124 #  define __INTR_ATTRS __used__, __externally_visible__
00125 #else /* GCC < 4.1 */
00126 #  define __INTR_ATTRS __used__
00127 #endif
00128
```

```
00129 #ifdef __cplusplus
00130 #  define ISR(vector, ...)            \
00131     extern "C" void vector (void) __attribute__ ((__signal__,__INTR_ATTRS)) __VA_ARGS__;
      \
00132     void vector (void)
00133 #else
00134 #  define ISR(vector, ...)            \
00135     void vector (void) __attribute__ ((__signal__,__INTR_ATTRS)) __VA_ARGS__; \
00136     void vector (void)
00137 #endif
00138
00139 #endif /* DOXYGEN */
00140
00141 #if defined(__DOXYGEN__)
00142 /** \def ISR_N(vector_num [, attributes])
00143     \ingroup avr_interrupts
00144
00145     Introduces an interrupt handler function (interrupt service
00146     routine) that runs with the \c SREG.I flag unchanged
00147     by default with no attributes specified.
00148     (On most devices this means that global interrupts are disabled
00149     upon servicing the IRQ.)
00150
00151     \c vector_num must be a positive interrupt vector number that is
00152     valid for the particular MCU type. For available vector numbers, see
00153     for example the \ref avr_mcu_signames "MCU &rarr; Vector Names" table.
00154
00155     Contrary to the #ISR macro, #ISR_N does not provide a declarator for
00156     the ISR.  #ISR_N may be specified more than once, which can be used
00157     to define aliases.  For example, the following definition provides
00158     an ISR for IRQ numbers 3 and 4 on an ATmega328:
00159 \code
00160     ISR_N (PCINT0_vect_num)
00161     ISR_N (PCINT1_vect_num)
00162     static void my_isr_handler (void)
00163     {
00164       // Code
00165     }
00166 \endcode
00167     The \c attributes are optional and alter the behaviour and resultant
00168     generated code of the interrupt routine. Multiple attributes may
00169     be used for a single function, with a space separating each
00170     attribute.
00171
00172     Valid attributes are #ISR_BLOCK, #ISR_NOBLOCK, #ISR_NAKED,
00173     #ISR_FLATTEN, #ISR_NOICF and #ISR_NOGCCISR.
00174
00175 \since AVR-LibC v2.3, <a href="https://gcc.gnu.org/gcc-15/changes.html#avr">GCC v15</a>
00176 */
00177 #  define ISR_N(vector_num, [attributes])
00178 #else  /* real code */
00179
00180 #if defined __HAVE_SIGNAL_N__
00181 /* Notice that "used" is implicit since v15, and that there is no requirement
00182    that the handler function is externally visible.  */
00183 #define ISR_N(N, ...)                 \
00184   __attribute__((__signal__(N))) __VA_ARGS__
00185 #else /* HAVE_SIGNAL_N */
00186 /* When GCC does not support "signal(n)", which is the case up to v14,
00187    then try to emit a helpful error message.  */
00188 #define __ISR_N_error2(L)                                              \
00189   __attribute__((__used__,__error__(                                   \
00190     "ISR_N not supported by this version of the compiler")))           \
00191   int AVR_LibC_show_error##L (int x)                                   \
00192   {                                                                    \
00193     __asm (".error \"ISR_N not supported by this version of the compiler\"");\
00194     return x ? 1 : x * AVR_LibC_show_error##L (x - 1);                 \
00195   }
00196 #define __ISR_N_error1(L) __ISR_N_error2(L)
00197 #define ISR_N(...) __ISR_N_error1(__LINE__)
00198 #endif /* HAVE_SIGNAL_N */
00199 #endif /* DOXYGEN ISR_N */
00200
```

```
00201 #if defined(__DOXYGEN__)
00202 /** \def SIGNAL(vector)
00203     \ingroup avr_interrupts
00204
00205     Introduces an interrupt handler function that runs with global interrupts
00206     initially disabled.
00207
00208     This is the same as the ISR macro without optional attributes.
00209     \deprecated Do not use SIGNAL() in new code. Use ISR() or ISR_N() instead.
00210 */
00211 #  define SIGNAL(vector)
00212 #else  /* real code */
00213
00214 #ifdef __cplusplus
00215 #  define SIGNAL(vector)                    \
00216     extern "C" void vector(void) __attribute__ ((__signal__, __INTR_ATTRS));    \
00217     void vector (void)
00218 #else
00219 #  define SIGNAL(vector)                    \
00220     void vector (void) __attribute__ ((__signal__, __INTR_ATTRS));      \
00221     void vector (void)
00222 #endif
00223
00224 #endif /* DOXYGEN */
00225
00226 #if defined(__DOXYGEN__)
00227 /** \def EMPTY_INTERRUPT(vector)
00228     \ingroup avr_interrupts
00229
00230     Defines an empty interrupt handler function. This will not generate
00231     any prolog or epilog code and will only return from the #ISR. Do not
00232     define a function body as this will define it for you.
00233     Example:
00234     \code EMPTY_INTERRUPT(ADC_vect);\endcode */
00235 #  define EMPTY_INTERRUPT(vector)
00236 #else  /* real code */
00237
00238 #ifdef __cplusplus
00239 #  define EMPTY_INTERRUPT(vector)               \
00240     extern "C" void vector(void) __attribute__ ((__signal__,__naked__,__INTR_ATTRS));
    \
00241     void vector (void) {  __asm__ __volatile__ ("reti" ::: "memory"); }
00242 #else
00243 #  define EMPTY_INTERRUPT(vector)               \
00244     void vector (void) __attribute__ ((__signal__,__naked__,__INTR_ATTRS));    \
00245     void vector (void) { __asm__ __volatile__ ("reti" ::: "memory"); }
00246 #endif
00247
00248 #endif /* DOXYGEN */
00249
00250 #if defined(__DOXYGEN__)
00251 /** \def ISR_ALIAS(vector, target_vector)
00252     \ingroup avr_interrupts
00253
00254     Aliases a given vector to another one in the same manner as the
00255     ISR_ALIASOF attribute for the ISR() macro.
00256
00257     \note This macro creates a trampoline function for the aliased
00258     macro.  This will result in a two cycle penalty for the aliased
00259     vector compared to the ISR the vector is aliased to, due to the
00260     JMP/RJMP opcode used.
00261
00262     \deprecated
00263     For new code, the use of ISR(..., ISR_ALIASOF(...))  or #ISR_N is
00264     recommended.  Notice that using #ISR_N does \e not impose a
00265     JMP/RJMP overhead.
00266
00267     Example:
00268     \code
00269     ISR (INT0_vect)
00270     {
00271         PORTB = 42;
00272     }
```

```
00273
00274     ISR_ALIAS (INT1_vect, INT0_vect);
00275
00276     // Alternative using ISR_N
00277
00278     ISR_N (INT0_vect_num)
00279     ISR_N (INT1_vect_num)
00280     static void my_int01_handler (void)
00281     {
00282         PORTB = 42;
00283     }
00284
00285     // or
00286
00287     ISR (INT0_vect,
00288          [attributes]
00289          ISR_N (INT1_vect_num))
00290     {
00291         PORTB = 42;
00292     }
00293     \endcode
00294 */
00295 #  define ISR_ALIAS(vector, target_vector)
00296 #else /* real code */
00297
00298 #ifdef __cplusplus
00299 #    define ISR_ALIAS(vector, tgt) extern "C" void vector (void) \
00300     __attribute__((__signal__, __naked__, __INTR_ATTRS)); \
00301     void vector (void) { __asm__ __volatile__ ("%~jmp " __STRINGIFY(tgt) ::); }
00302 #else     /* !__cplusplus */
00303 #  define ISR_ALIAS(vector, tgt) void vector (void) \
00304     __attribute__((__signal__, __naked__, __INTR_ATTRS)); \
00305     void vector (void) { __asm__ __volatile__ ("%~jmp " __STRINGIFY(tgt) ::); }
00306 #endif  /* __cplusplus */
00307
00308 #endif /* DOXYGEN */
00309
00310 /** \def reti()
00311     \ingroup avr_interrupts
00312
00313     Returns from an interrupt routine, enabling global interrupts. This should
00314     be the last command executed before leaving an #ISR defined with the
00315     #ISR_NAKED attribute.
00316
00317     This macro actually compiles into a single line of assembly, so there is
00318     no function call overhead.
00319
00320     \note According to the GCC documentation, the only code supported in
00321     naked functions is \ref inline_asm "inline assembly".
00322 */
00323 #  define reti()  __asm__ __volatile__ ("reti" ::: "memory")
00324
00325 #if defined(__DOXYGEN__)
00326 /** \def BADISR_vect
00327     \ingroup avr_interrupts
00328
00329     \code #include <avr/interrupt.h> \endcode
00330
00331     This is a vector which is aliased to \c __vector_default, the vector
00332     executed when an IRQ fires with no accompanying ISR handler. This
00333     may be used along with the ISR() macro to create a catch-all for
00334     undefined but used ISRs for debugging purposes.  It cannot be used
00335     with #ISR_N since there is no associated interrupt number.  */
00336 #  define BADISR_vect
00337 #else  /* !DOXYGEN */
00338 #  define BADISR_vect __vector_default
00339 #endif /* DOXYGEN */
00340
00341 /** \name ISR attributes */
00342
00343 #if defined(__DOXYGEN__)
00344 /** \def ISR_BLOCK
00345     \ingroup avr_interrupts
```

```
00346
00347     Identical to an ISR with no attributes specified. Global
00348     interrupts are initially disabled by the AVR hardware when
00349     entering the ISR, without the compiler modifying this state.
00350
00351     Use this attribute in the \c attributes parameter of the #ISR
00352     and #ISR_N macros.
00353 */
00354 #  define ISR_BLOCK
00355
00356 /** \def ISR_NOBLOCK
00357     \ingroup avr_interrupts
00358
00359     ISR runs with global interrupts initially enabled.  The interrupt
00360     enable flag is activated by the compiler as early as possible
00361     within the ISR to ensure minimal processing delay for nested
00362     interrupts.
00363
00364     This may be used to create nested ISRs, however care should be
00365     taken to avoid stack overflows, or to avoid infinitely entering
00366     the ISR for those cases where the AVR hardware does not clear the
00367     respective interrupt flag before entering the ISR.
00368
00369     Use this attribute in the \c attributes parameter of the #ISR and
00370     #ISR_N macros.
00371 */
00372 #  define ISR_NOBLOCK
00373
00374 /** \def ISR_NAKED
00375     \ingroup avr_interrupts
00376
00377     ISR is created with no prologue or epilogue code. The user code is
00378     responsible for preservation of the machine state including the
00379     SREG register, as well as placing a reti() at the end of the
00380     interrupt routine.
00381
00382     Use this attribute in the \c attributes parameter of the #ISR and
00383     #ISR_N macros.
00384
00385     \note According to GCC documentation, the only code supported in
00386     naked functions is \ref inline_asm "inline assembly".
00387 */
00388 #  define ISR_NAKED
00389
00390 /** \def ISR_FLATTEN
00391     \ingroup avr_interrupts
00392
00393     The compiler will try to inline all called function into the ISR.
00394
00395     Use this attribute in the \c attributes parameter of the #ISR and
00396     #ISR_N macros.
00397 */
00398 #  define ISR_FLATTEN
00399
00400 /** \def ISR_NOICF
00401     \ingroup avr_interrupts
00402
00403     Avoid identical-code-folding optimization against this ISR.
00404     This has an effect with GCC 5 and newer only.
00405
00406     Use this attribute in the \c attributes parameter of the #ISR and
00407     #ISR_N macros.
00408 */
00409 #  define ISR_NOICF
00410
00411 /** \def ISR_NOGCCISR
00412     \ingroup avr_interrupts
00413
00414     Do not generate
00415     <a href="https://sourceware.org/binutils/docs/as/AVR-Pseudo-Instructions.html">\c
      __gcc_isr pseudo instructions</a>
00416     for this ISR.
00417     This has an effect with
```

```
00418     <a href="https://gcc.gnu.org/gcc-8/changes.html#avr">GCC 8</a>
00419     and newer only.
00420
00421     Use this attribute in the \c attributes parameter of the #ISR and
00422     #ISR_N macros.
00423 */
00424 #  define ISR_NOGCCISR
00425
00426 /** \def ISR_ALIASOF(target_vector)
00427     \ingroup avr_interrupts
00428
00429     The ISR is linked to another ISR, specified by the vect parameter.
00430
00431     Use this attribute in the \c attributes parameter of the #ISR macro.
00432     Example:
00433     \code
00434     ISR (INT0_vect)
00435     {
00436         PORTB = 42;
00437     }
00438
00439     ISR (INT1_vect, ISR_ALIASOF (INT0_vect));
00440     \endcode
00441
00442     Notice that the #ISR_ALIASOF macro implements its own IRQ handler that
00443     jumps to the aliased ISR, which means there is a run-time overhead of
00444     a JMP/RJMP instruction.  For an alternative without overhead, see
00445     the #ISR_N macro.
00446 */
00447 #  define ISR_ALIASOF(target_vector)
00448 #else  /* !DOXYGEN */
00449 #  define ISR_BLOCK /* empty */
00450 /* FIXME: This won't work with older versions of avr-gcc as ISR_NOBLOCK
00451           will use 'signal' and 'interrupt' at the same time.  */
00452 #  ifdef __HAVE_SIGNAL_N__
00453 /* Use "noblock" if available.  This works rather like a flag that can be
00454    combined with "signal(n)" without imposing a specific function name,
00455    like "interrupt" would do.  */
00456 #  define ISR_NOBLOCK    __attribute__((__noblock__))
00457 #  else
00458 #  define ISR_NOBLOCK    __attribute__((__interrupt__))
00459 #  endif /* Have signal(n) and noblock */
00460
00461 #  define ISR_NAKED      __attribute__((__naked__))
00462
00463 #if (__GNUC__ == 4 && __GNUC_MINOR__ >= 6) || (__GNUC__ >= 5)
00464 #  define ISR_FLATTEN    __attribute__((__flatten__))
00465 #else
00466 #  define ISR_FLATTEN    /* empty */
00467 #endif /* has flatten (GCC 4.6+) */
00468
00469 #if defined (__has_attribute)
00470 #if __has_attribute (__no_icf__)
00471 #  define ISR_NOICF      __attribute__((__no_icf__))
00472 #else
00473 #  define ISR_NOICF      /* empty */
00474 #endif /* has no_icf */
00475
00476 #if __has_attribute (__no_gccisr__)
00477 #  define ISR_NOGCCISR      __attribute__((__no_gccisr__))
00478 #else
00479 #  define ISR_NOGCCISR      /* empty */
00480 #endif /* has no_gccisr */
00481 #endif /* has __has_attribute (GCC 5+) */
00482
00483 #  define ISR_ALIASOF(v) __attribute__((__alias__(__STRINGIFY(v))))
00484 #endif /* DOXYGEN */
00485
00486 /**@}*/
00487
00488 #endif
```

22.30 io.h File Reference

22.31 io.h

Go to the documentation of this file.

```
00001 /* Copyright (c) 2002,2003,2005,2006,2007 Marek Michalkiewicz, Joerg Wunsch
00002    Copyright (c) 2007 Eric B. Weddington
00003    All rights reserved.
00004
00005    Redistribution and use in source and binary forms, with or without
00006    modification, are permitted provided that the following conditions are met:
00007
00008    * Redistributions of source code must retain the above copyright
00009      notice, this list of conditions and the following disclaimer.
00010
00011    * Redistributions in binary form must reproduce the above copyright
00012      notice, this list of conditions and the following disclaimer in
00013      the documentation and/or other materials provided with the
00014      distribution.
00015
00016    * Neither the name of the copyright holders nor the names of
00017      contributors may be used to endorse or promote products derived
00018      from this software without specific prior written permission.
00019
00020   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
00021   AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
00022   IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
00023   ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
00024   LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
00025   CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
00026   SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
00027   INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
00028   CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
00029   ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
00030   POSSIBILITY OF SUCH DAMAGE. */
00031
00032 /** \file */
00033 /** \defgroup avr_io <avr/io.h>: AVR device-specific IO definitions
00034     \code #include <avr/io.h> \endcode
00035
00036     This header file includes the appropriate IO definitions for the
00037     device that has been specified by the <tt>-mmcu=</tt> compiler
00038     command-line option.  This is done by diverting to the appropriate
00039     file <tt>&lt;avr/io</tt><em>XXXX</em><tt>.h&gt;</tt> which should
00040     never be included directly.  Some register names common to all
00041     AVR devices are defined directly within <tt>&lt;avr/common.h&gt;</tt>,
00042     which is included in <tt>&lt;avr/io.h&gt;</tt>,
00043     but most of the details come from the respective include file.
00044
00045     Note that this header always includes the following ones:
00046     \code
00047     #include <avr/sfr_defs.h>
00048     #include <avr/portpins.h>
00049     #include <avr/common.h>
00050     #include <avr/version.h>
00051     \endcode
00052     See \ref avr_sfr for more details about that header file.
00053
00054     Included are definitions of the IO register set and their
00055     respective bit values as specified in the device manual.
00056     Note that due to inconsistencies in naming conventions,
00057     so even identical functions sometimes get different names on
00058     different devices.
00059
00060     Also included are the specific names useable for interrupt
00061     service routines as documented
00062     \ref avr_mcu_signames "here".
00063
00064     Finally, the following macros are defined:
00065
```

```
00066     <dl>
00067     <dt>RAMEND
00068     <dd>The last on-chip RAM address.
00069
00070     <dt>XRAMEND
00071     <dd>The last possible RAM location that is addressable. This is equal to
00072     RAMEND for devices that do not allow for external RAM. For devices
00073     that allow external RAM, this will be larger than RAMEND.
00074
00075     <dt>E2END
00076     <dd>The last EEPROM address.
00077
00078     <dt>FLASHEND
00079     <dd>The last byte address in the Flash program space.
00080
00081     <dt>SPM_PAGESIZE
00082     <dd>For devices with bootloader support, the flash pagesize
00083     (in bytes) to be used for the \c SPM instruction.
00084
00085     <dt>E2PAGESIZE
00086     <dd>The size of the EEPROM page.
00087     </dl>
00088 */
00089
00090 #ifndef _AVR_IO_H_
00091 #define _AVR_IO_H_
00092
00093 #include <avr/sfr_defs.h>
00094
00095 #if defined(BAUD)
00096 #if !defined (__CONFIGURING_AVR_LIBC__)
00097 #include <bits/devinfo.h>
00098 #endif
00099 #if defined (__AVR_IO_H_USES_BAUD__)
00100 #error BAUD is a reserved identifier in <avr/io.h>
00101 #endif
00102 #endif /* defined BAUD? */
00103
00104 #if defined (__AVR_AT94K__)
00105 #  include <avr/ioat94k.h>
00106 #elif defined (__AVR_AT43USB320__)
00107 #  include <avr/io43u32x.h>
00108 #elif defined (__AVR_AT43USB355__)
00109 #  include <avr/io43u35x.h>
00110 #elif defined (__AVR_AT76C711__)
00111 #  include <avr/io76c711.h>
00112 #elif defined (__AVR_AT86RF401__)
00113 #  include <avr/io86r401.h>
00114 #elif defined (__AVR_AT90PWM1__)
00115 #  include <avr/io90pwm1.h>
00116 #elif defined (__AVR_AT90PWM2__)
00117 #  include <avr/io90pwmx.h>
00118 #elif defined (__AVR_AT90PWM2B__)
00119 #  include <avr/io90pwm2b.h>
00120 #elif defined (__AVR_AT90PWM3__)
00121 #  include <avr/io90pwmx.h>
00122 #elif defined (__AVR_AT90PWM3B__)
00123 #  include <avr/io90pwm3b.h>
00124 #elif defined (__AVR_AT90PWM216__)
00125 #  include <avr/io90pwm216.h>
00126 #elif defined (__AVR_AT90PWM316__)
00127 #  include <avr/io90pwm316.h>
00128 #elif defined (__AVR_AT90PWM161__)
00129 #  include <avr/io90pwm161.h>
00130 #elif defined (__AVR_AT90PWM81__)
00131 #  include <avr/io90pwm81.h>
00132 #elif defined (__AVR_ATmega8U2__)
00133 #  include <avr/iom8u2.h>
00134 #elif defined (__AVR_ATmega16M1__)
00135 #  include <avr/iom16m1.h>
00136 #elif defined (__AVR_ATmega16U2__)
00137 #  include <avr/iom16u2.h>
00138 #elif defined (__AVR_ATmega16U4__)
```

```
00139 #  include <avr/iom16u4.h>
00140 #elif defined (__AVR_ATmega32C1__)
00141 #  include <avr/iom32c1.h>
00142 #elif defined (__AVR_ATmega32M1__)
00143 #  include <avr/iom32m1.h>
00144 #elif defined (__AVR_ATmega32U2__)
00145 #  include <avr/iom32u2.h>
00146 #elif defined (__AVR_ATmega32U4__)
00147 #  include <avr/iom32u4.h>
00148 #elif defined (__AVR_ATmega32U6__)
00149 #  include <avr/iom32u6.h>
00150 #elif defined (__AVR_ATmega64C1__)
00151 #  include <avr/iom64c1.h>
00152 #elif defined (__AVR_ATmega64M1__)
00153 #  include <avr/iom64m1.h>
00154 #elif defined (__AVR_ATmega128__)
00155 #  include <avr/iom128.h>
00156 #elif defined (__AVR_ATmega128A__)
00157 #  include <avr/iom128a.h>
00158 #elif defined (__AVR_ATmega1280__)
00159 #  include <avr/iom1280.h>
00160 #elif defined (__AVR_ATmega1281__)
00161 #  include <avr/iom1281.h>
00162 #elif defined (__AVR_ATmega1284__)
00163 #  include <avr/iom1284.h>
00164 #elif defined (__AVR_ATmega1284P__)
00165 #  include <avr/iom1284p.h>
00166 #elif defined (__AVR_ATmega128RFA1__)
00167 #  include <avr/iom128rfa1.h>
00168 #elif defined (__AVR_ATmega1284RFR2__)
00169 #  include <avr/iom1284rfr2.h>
00170 #elif defined (__AVR_ATmega128RFR2__)
00171 #  include <avr/iom128rfr2.h>
00172 #elif defined (__AVR_ATmega2564RFR2__)
00173 #  include <avr/iom2564rfr2.h>
00174 #elif defined (__AVR_ATmega256RFR2__)
00175 #  include <avr/iom256rfr2.h>
00176 #elif defined (__AVR_ATmega2560__)
00177 #  include <avr/iom2560.h>
00178 #elif defined (__AVR_ATmega2561__)
00179 #  include <avr/iom2561.h>
00180 #elif defined (__AVR_AT90CAN32__)
00181 #  include <avr/iocan32.h>
00182 #elif defined (__AVR_AT90CAN64__)
00183 #  include <avr/iocan64.h>
00184 #elif defined (__AVR_AT90CAN128__)
00185 #  include <avr/iocan128.h>
00186 #elif defined (__AVR_AT90USB82__)
00187 #  include <avr/iousb82.h>
00188 #elif defined (__AVR_AT90USB162__)
00189 #  include <avr/iousb162.h>
00190 #elif defined (__AVR_AT90USB646__)
00191 #  include <avr/iousb646.h>
00192 #elif defined (__AVR_AT90USB647__)
00193 #  include <avr/iousb647.h>
00194 #elif defined (__AVR_AT90USB1286__)
00195 #  include <avr/iousb1286.h>
00196 #elif defined (__AVR_AT90USB1287__)
00197 #  include <avr/iousb1287.h>
00198 #elif defined (__AVR_ATmega644RFR2__)
00199 #  include <avr/iom644rfr2.h>
00200 #elif defined (__AVR_ATmega64RFR2__)
00201 #  include <avr/iom64rfr2.h>
00202 #elif defined (__AVR_ATmega64__)
00203 #  include <avr/iom64.h>
00204 #elif defined (__AVR_ATmega64A__)
00205 #  include <avr/iom64a.h>
00206 #elif defined (__AVR_ATmega640__)
00207 #  include <avr/iom640.h>
00208 #elif defined (__AVR_ATmega644__)
00209 #  include <avr/iom644.h>
00210 #elif defined (__AVR_ATmega644A__)
00211 #  include <avr/iom644a.h>
```

```
00212 #elif defined (__AVR_ATmega644P__)
00213 #  include <avr/iom644p.h>
00214 #elif defined (__AVR_ATmega644PA__)
00215 #  include <avr/iom644pa.h>
00216 #elif defined (__AVR_ATmega645__) || defined (__AVR_ATmega645A__) || defined
      (__AVR_ATmega645P__)
00217 #  include <avr/iom645.h>
00218 #elif defined (__AVR_ATmega6450__) || defined (__AVR_ATmega6450A__) || defined
      (__AVR_ATmega6450P__)
00219 #  include <avr/iom6450.h>
00220 #elif defined (__AVR_ATmega649__) || defined (__AVR_ATmega649A__)
00221 #  include <avr/iom649.h>
00222 #elif defined (__AVR_ATmega6490__) || defined (__AVR_ATmega6490A__) || defined
      (__AVR_ATmega6490P__)
00223 #  include <avr/iom6490.h>
00224 #elif defined (__AVR_ATmega649P__)
00225 #  include <avr/iom649p.h>
00226 #elif defined (__AVR_ATmega64HVE__)
00227 #  include <avr/iom64hve.h>
00228 #elif defined (__AVR_ATmega64HVE2__)
00229 #  include <avr/iom64hve2.h>
00230 #elif defined (__AVR_ATmega103__)
00231 #  include <avr/iom103.h>
00232 #elif defined (__AVR_ATmega32__)
00233 #  include <avr/iom32.h>
00234 #elif defined (__AVR_ATmega32A__)
00235 #  include <avr/iom32a.h>
00236 #elif defined (__AVR_ATmega323__)
00237 #  include <avr/iom323.h>
00238 #elif defined (__AVR_ATmega324P__) || defined (__AVR_ATmega324A__)
00239 #  include <avr/iom324.h>
00240 #elif defined (__AVR_ATmega324PA__)
00241 #  include <avr/iom324pa.h>
00242 #elif defined (__AVR_ATmega324PB__)
00243 #  include <avr/iom324pb.h>
00244 #elif defined (__AVR_ATmega325__) || defined (__AVR_ATmega325A__)
00245 #  include <avr/iom325.h>
00246 #elif defined (__AVR_ATmega325P__)
00247 #  include <avr/iom325.h>
00248 #elif defined (__AVR_ATmega325PA__)
00249 #  include <avr/iom325pa.h>
00250 #elif defined (__AVR_ATmega3250__) || defined (__AVR_ATmega3250A__)
00251 #  include <avr/iom3250.h>
00252 #elif defined (__AVR_ATmega3250P__)
00253 #  include <avr/iom3250.h>
00254 #elif defined (__AVR_ATmega3250PA__)
00255 #  include <avr/iom3250pa.h>
00256 #elif defined (__AVR_ATmega328P__) || defined (__AVR_ATmega328__)
00257 #  include <avr/iom328p.h>
00258 #elif defined (__AVR_ATmega328PB__)
00259 #  include <avr/iom328pb.h>
00260 #elif defined (__AVR_ATmega329__) || defined (__AVR_ATmega329A__)
00261 #  include <avr/iom329.h>
00262 #elif defined (__AVR_ATmega329P__) || defined (__AVR_ATmega329PA__)
00263 #  include <avr/iom329.h>
00264 #elif defined (__AVR_ATmega3290__) || defined (__AVR_ATmega3290A__)
00265 #  include <avr/iom3290.h>
00266 #elif defined (__AVR_ATmega3290P__)
00267 #  include <avr/iom3290.h>
00268 #elif defined (__AVR_ATmega3290PA__)
00269 #  include <avr/iom3290pa.h>
00270 #elif defined (__AVR_ATmega32HVB__)
00271 #  include <avr/iom32hvb.h>
00272 #elif defined (__AVR_ATmega32HVBREVB__)
00273 #  include <avr/iom32hvbrevb.h>
00274 #elif defined (__AVR_ATmega406__)
00275 #  include <avr/iom406.h>
00276 #elif defined (__AVR_ATmega16__)
00277 #  include <avr/iom16.h>
00278 #elif defined (__AVR_ATmega16A__)
00279 #  include <avr/iom16a.h>
00280 #elif defined (__AVR_ATmega161__)
00281 #  include <avr/iom161.h>
```

```
00282 #elif defined (__AVR_ATmega162__)
00283 #  include <avr/iom162.h>
00284 #elif defined (__AVR_ATmega163__)
00285 #  include <avr/iom163.h>
00286 #elif defined (__AVR_ATmega164P__) || defined (__AVR_ATmega164A__)
00287 #  include <avr/iom164.h>
00288 #elif defined (__AVR_ATmega164PA__)
00289 #  include <avr/iom164pa.h>
00290 #elif defined (__AVR_ATmega165__)
00291 #  include <avr/iom165.h>
00292 #elif defined (__AVR_ATmega165A__)
00293 #  include <avr/iom165a.h>
00294 #elif defined (__AVR_ATmega165P__)
00295 #  include <avr/iom165p.h>
00296 #elif defined (__AVR_ATmega165PA__)
00297 #  include <avr/iom165pa.h>
00298 #elif defined (__AVR_ATmega168__)
00299 #  include <avr/iom168.h>
00300 #elif defined (__AVR_ATmega168A__)
00301 #  include <avr/iom168a.h>
00302 #elif defined (__AVR_ATmega168P__)
00303 #  include <avr/iom168p.h>
00304 #elif defined (__AVR_ATmega168PA__)
00305 #  include <avr/iom168pa.h>
00306 #elif defined (__AVR_ATmega168PB__)
00307 #  include <avr/iom168pb.h>
00308 #elif defined (__AVR_ATmega169__) || defined (__AVR_ATmega169A__)
00309 #  include <avr/iom169.h>
00310 #elif defined (__AVR_ATmega169P__)
00311 #  include <avr/iom169p.h>
00312 #elif defined (__AVR_ATmega169PA__)
00313 #  include <avr/iom169pa.h>
00314 #elif defined (__AVR_ATmega8HVA__)
00315 #  include <avr/iom8hva.h>
00316 #elif defined (__AVR_ATmega16HVA__)
00317 #  include <avr/iom16hva.h>
00318 #elif defined (__AVR_ATmega16HVA2__)
00319 #  include <avr/iom16hva2.h>
00320 #elif defined (__AVR_ATmega16HVB__)
00321 #  include <avr/iom16hvb.h>
00322 #elif defined (__AVR_ATmega16HVBREVB__)
00323 #  include <avr/iom16hvbrevb.h>
00324 #elif defined (__AVR_ATmega8__)
00325 #  include <avr/iom8.h>
00326 #elif defined (__AVR_ATmega8A__)
00327 #  include <avr/iom8a.h>
00328 #elif defined (__AVR_ATmega48__)
00329 #  include <avr/iom48.h>
00330 #elif defined (__AVR_ATmega48A__)
00331 #  include <avr/iom48a.h>
00332 #elif defined (__AVR_ATmega48PA__)
00333 #  include <avr/iom48pa.h>
00334 #elif defined (__AVR_ATmega48PB__)
00335 #  include <avr/iom48pb.h>
00336 #elif defined (__AVR_ATmega48P__)
00337 #  include <avr/iom48p.h>
00338 #elif defined (__AVR_ATmega88__)
00339 #  include <avr/iom88.h>
00340 #elif defined (__AVR_ATmega88A__)
00341 #  include <avr/iom88a.h>
00342 #elif defined (__AVR_ATmega88P__)
00343 #  include <avr/iom88p.h>
00344 #elif defined (__AVR_ATmega88PA__)
00345 #  include <avr/iom88pa.h>
00346 #elif defined (__AVR_ATmega88PB__)
00347 #  include <avr/iom88pb.h>
00348 #elif defined (__AVR_ATmega8515__)
00349 #  include <avr/iom8515.h>
00350 #elif defined (__AVR_ATmega8535__)
00351 #  include <avr/iom8535.h>
00352 #elif defined (__AVR_AT90S8535__)
00353 #  include <avr/io8535.h>
00354 #elif defined (__AVR_AT90C8534__)
```

```
00355 #  include <avr/io8534.h>
00356 #elif defined (__AVR_AT90S8515__)
00357 #  include <avr/io8515.h>
00358 #elif defined (__AVR_AT90S4434__)
00359 #  include <avr/io4434.h>
00360 #elif defined (__AVR_AT90S4433__)
00361 #  include <avr/io4433.h>
00362 #elif defined (__AVR_AT90S4414__)
00363 #  include <avr/io4414.h>
00364 #elif defined (__AVR_ATtiny22__)
00365 #  include <avr/iotn22.h>
00366 #elif defined (__AVR_ATtiny26__)
00367 #  include <avr/iotn26.h>
00368 #elif defined (__AVR_AT90S2343__)
00369 #  include <avr/io2343.h>
00370 #elif defined (__AVR_AT90S2333__)
00371 #  include <avr/io2333.h>
00372 #elif defined (__AVR_AT90S2323__)
00373 #  include <avr/io2323.h>
00374 #elif defined (__AVR_AT90S2313__)
00375 #  include <avr/io2313.h>
00376 #elif defined (__AVR_ATtiny4__)
00377 #  include <avr/iotn4.h>
00378 #elif defined (__AVR_ATtiny5__)
00379 #  include <avr/iotn5.h>
00380 #elif defined (__AVR_ATtiny9__)
00381 #  include <avr/iotn9.h>
00382 #elif defined (__AVR_ATtiny10__)
00383 #  include <avr/iotn10.h>
00384 #elif defined (__AVR_ATtiny102__)
00385 #  include <avr/iotn102.h>
00386 #elif defined (__AVR_ATtiny104__)
00387 #  include <avr/iotn104.h>
00388 #elif defined (__AVR_ATtiny20__)
00389 #  include <avr/iotn20.h>
00390 #elif defined (__AVR_ATtiny40__)
00391 #  include <avr/iotn40.h>
00392 #elif defined (__AVR_ATtiny2313__)
00393 #  include <avr/iotn2313.h>
00394 #elif defined (__AVR_ATtiny2313A__)
00395 #  include <avr/iotn2313a.h>
00396 #elif defined (__AVR_ATtiny13__)
00397 #  include <avr/iotn13.h>
00398 #elif defined (__AVR_ATtiny13A__)
00399 #  include <avr/iotn13a.h>
00400 #elif defined (__AVR_ATtiny25__)
00401 #  include <avr/iotn25.h>
00402 #elif defined (__AVR_ATtiny4313__)
00403 #  include <avr/iotn4313.h>
00404 #elif defined (__AVR_ATtiny45__)
00405 #  include <avr/iotn45.h>
00406 #elif defined (__AVR_ATtiny85__)
00407 #  include <avr/iotn85.h>
00408 #elif defined (__AVR_ATtiny24__)
00409 #  include <avr/iotn24.h>
00410 #elif defined (__AVR_ATtiny24A__)
00411 #  include <avr/iotn24a.h>
00412 #elif defined (__AVR_ATtiny44__)
00413 #  include <avr/iotn44.h>
00414 #elif defined (__AVR_ATtiny44A__)
00415 #  include <avr/iotn44a.h>
00416 #elif defined (__AVR_ATtiny441__)
00417 #  include <avr/iotn441.h>
00418 #elif defined (__AVR_ATtiny84__)
00419 #  include <avr/iotn84.h>
00420 #elif defined (__AVR_ATtiny84A__)
00421 #  include <avr/iotn84a.h>
00422 #elif defined (__AVR_ATtiny841__)
00423 #  include <avr/iotn841.h>
00424 #elif defined (__AVR_ATtiny261__)
00425 #  include <avr/iotn261.h>
00426 #elif defined (__AVR_ATtiny261A__)
00427 #  include <avr/iotn261a.h>
```

```
00428 #elif defined (__AVR_ATtiny461__)
00429 #  include <avr/iotn461.h>
00430 #elif defined (__AVR_ATtiny461A__)
00431 #  include <avr/iotn461a.h>
00432 #elif defined (__AVR_ATtiny861__)
00433 #  include <avr/iotn861.h>
00434 #elif defined (__AVR_ATtiny861A__)
00435 #  include <avr/iotn861a.h>
00436 #elif defined (__AVR_ATtiny43U__)
00437 #  include <avr/iotn43u.h>
00438 #elif defined (__AVR_ATtiny48__)
00439 #  include <avr/iotn48.h>
00440 #elif defined (__AVR_ATtiny88__)
00441 #  include <avr/iotn88.h>
00442 #elif defined (__AVR_ATtiny828__)
00443 #  include <avr/iotn828.h>
00444 #elif defined (__AVR_ATtiny87__)
00445 #  include <avr/iotn87.h>
00446 #elif defined (__AVR_ATtiny167__)
00447 #  include <avr/iotn167.h>
00448 #elif defined (__AVR_ATtiny1634__)
00449 #  include <avr/iotn1634.h>
00450 #elif defined (__AVR_ATtiny202__)
00451 #  include <avr/iotn202.h>
00452 #elif defined (__AVR_ATtiny204__)
00453 #  include <avr/iotn204.h>
00454 #elif defined (__AVR_ATtiny212__)
00455 #  include <avr/iotn212.h>
00456 #elif defined (__AVR_ATtiny214__)
00457 #  include <avr/iotn214.h>
00458 #elif defined (__AVR_ATtiny402__)
00459 #  include <avr/iotn402.h>
00460 #elif defined (__AVR_ATtiny404__)
00461 #  include <avr/iotn404.h>
00462 #elif defined (__AVR_ATtiny406__)
00463 #  include <avr/iotn406.h>
00464 #elif defined (__AVR_ATtiny412__)
00465 #  include <avr/iotn412.h>
00466 #elif defined (__AVR_ATtiny414__)
00467 #  include <avr/iotn414.h>
00468 #elif defined (__AVR_ATtiny416__)
00469 #  include <avr/iotn416.h>
00470 #elif defined (__AVR_ATtiny416AUTO__)
00471 #  include <avr/iotn416auto.h>
00472 #elif defined (__AVR_ATtiny417__)
00473 #  include <avr/iotn417.h>
00474 #elif defined (__AVR_ATtiny424__)
00475 #  include <avr/iotn424.h>
00476 #elif defined (__AVR_ATtiny426__)
00477 #  include <avr/iotn426.h>
00478 #elif defined (__AVR_ATtiny427__)
00479 #  include <avr/iotn427.h>
00480 #elif defined (__AVR_ATtiny804__)
00481 #  include <avr/iotn804.h>
00482 #elif defined (__AVR_ATtiny806__)
00483 #  include <avr/iotn806.h>
00484 #elif defined (__AVR_ATtiny807__)
00485 #  include <avr/iotn807.h>
00486 #elif defined (__AVR_ATtiny814__)
00487 #  include <avr/iotn814.h>
00488 #elif defined (__AVR_ATtiny816__)
00489 #  include <avr/iotn816.h>
00490 #elif defined (__AVR_ATtiny817__)
00491 #  include <avr/iotn817.h>
00492 #elif defined (__AVR_ATtiny824__)
00493 #  include <avr/iotn824.h>
00494 #elif defined (__AVR_ATtiny826__)
00495 #  include <avr/iotn826.h>
00496 #elif defined (__AVR_ATtiny827__)
00497 #  include <avr/iotn827.h>
00498 #elif defined (__AVR_ATtiny1604__)
00499 #  include <avr/iotn1604.h>
00500 #elif defined (__AVR_ATtiny1606__)
```

```
00501 #  include <avr/iotn1606.h>
00502 #elif defined (__AVR_ATtiny1607__)
00503 #  include <avr/iotn1607.h>
00504 #elif defined (__AVR_ATtiny1614__)
00505 #  include <avr/iotn1614.h>
00506 #elif defined (__AVR_ATtiny1616__)
00507 #  include <avr/iotn1616.h>
00508 #elif defined (__AVR_ATtiny1617__)
00509 #  include <avr/iotn1617.h>
00510 #elif defined (__AVR_ATtiny1624__)
00511 #  include <avr/iotn1624.h>
00512 #elif defined (__AVR_ATtiny1626__)
00513 #  include <avr/iotn1626.h>
00514 #elif defined (__AVR_ATtiny1627__)
00515 #  include <avr/iotn1627.h>
00516 #elif defined (__AVR_ATtiny3214__)
00517 #  include <avr/iotn3214.h>
00518 #elif defined (__AVR_ATtiny3216__)
00519 #  include <avr/iotn3216.h>
00520 #elif defined (__AVR_ATtiny3217__)
00521 #  include <avr/iotn3217.h>
00522 #elif defined (__AVR_ATtiny3224__)
00523 #  include <avr/iotn3224.h>
00524 #elif defined (__AVR_ATtiny3226__)
00525 #  include <avr/iotn3226.h>
00526 #elif defined (__AVR_ATtiny3227__)
00527 #  include <avr/iotn3227.h>
00528 #elif defined (__AVR_ATmega808__)
00529 #  include <avr/iom808.h>
00530 #elif defined (__AVR_ATmega809__)
00531 #  include <avr/iom809.h>
00532 #elif defined (__AVR_ATmega1608__)
00533 #  include <avr/iom1608.h>
00534 #elif defined (__AVR_ATmega1609__)
00535 #  include <avr/iom1609.h>
00536 #elif defined (__AVR_ATmega3208__)
00537 #  include <avr/iom3208.h>
00538 #elif defined (__AVR_ATmega3209__)
00539 #  include <avr/iom3209.h>
00540 #elif defined (__AVR_ATmega4808__)
00541 #  include <avr/iom4808.h>
00542 #elif defined (__AVR_ATmega4809__)
00543 #  include <avr/iom4809.h>
00544 #elif defined (__AVR_AT90SCR100__)
00545 #  include <avr/io90scr100.h>
00546 #elif defined (__AVR_ATxmega8E5__)
00547 #  include <avr/iox8e5.h>
00548 #elif defined (__AVR_ATxmega16A4__)
00549 #  include <avr/iox16a4.h>
00550 #elif defined (__AVR_ATxmega16A4U__)
00551 #  include <avr/iox16a4u.h>
00552 #elif defined (__AVR_ATxmega16C4__)
00553 #  include <avr/iox16c4.h>
00554 #elif defined (__AVR_ATxmega16D4__)
00555 #  include <avr/iox16d4.h>
00556 #elif defined (__AVR_ATxmega32A4__)
00557 #  include <avr/iox16e5.h>
00558 #elif defined (__AVR_ATxmega16E5__)
00559 #  include <avr/iox32a4.h>
00560 #elif defined (__AVR_ATxmega32A4U__)
00561 #  include <avr/iox32a4u.h>
00562 #elif defined (__AVR_ATxmega32C3__)
00563 #  include <avr/iox32c3.h>
00564 #elif defined (__AVR_ATxmega32C4__)
00565 #  include <avr/iox32c4.h>
00566 #elif defined (__AVR_ATxmega32D3__)
00567 #  include <avr/iox32d3.h>
00568 #elif defined (__AVR_ATxmega32D4__)
00569 #  include <avr/iox32d4.h>
00570 #elif defined (__AVR_ATxmega32E5__)
00571 #  include <avr/iox32e5.h>
00572 #elif defined (__AVR_ATxmega64A1__)
00573 #  include <avr/iox64a1.h>
```

```
00574 #elif defined (__AVR_ATxmega64A1U__)
00575 #  include <avr/iox64a1u.h>
00576 #elif defined (__AVR_ATxmega64A3__)
00577 #  include <avr/iox64a3.h>
00578 #elif defined (__AVR_ATxmega64A3U__)
00579 #  include <avr/iox64a3u.h>
00580 #elif defined (__AVR_ATxmega64A4U__)
00581 #  include <avr/iox64a4u.h>
00582 #elif defined (__AVR_ATxmega64B1__)
00583 #  include <avr/iox64b1.h>
00584 #elif defined (__AVR_ATxmega64B3__)
00585 #  include <avr/iox64b3.h>
00586 #elif defined (__AVR_ATxmega64C3__)
00587 #  include <avr/iox64c3.h>
00588 #elif defined (__AVR_ATxmega64D3__)
00589 #  include <avr/iox64d3.h>
00590 #elif defined (__AVR_ATxmega64D4__)
00591 #  include <avr/iox64d4.h>
00592 #elif defined (__AVR_ATxmega128A1__)
00593 #  include <avr/iox128a1.h>
00594 #elif defined (__AVR_ATxmega128A1U__)
00595 #  include <avr/iox128a1u.h>
00596 #elif defined (__AVR_ATxmega128A4U__)
00597 #  include <avr/iox128a4u.h>
00598 #elif defined (__AVR_ATxmega128A3__)
00599 #  include <avr/iox128a3.h>
00600 #elif defined (__AVR_ATxmega128A3U__)
00601 #  include <avr/iox128a3u.h>
00602 #elif defined (__AVR_ATxmega128B1__)
00603 #  include <avr/iox128b1.h>
00604 #elif defined (__AVR_ATxmega128B3__)
00605 #  include <avr/iox128b3.h>
00606 #elif defined (__AVR_ATxmega128C3__)
00607 #  include <avr/iox128c3.h>
00608 #elif defined (__AVR_ATxmega128D3__)
00609 #  include <avr/iox128d3.h>
00610 #elif defined (__AVR_ATxmega128D4__)
00611 #  include <avr/iox128d4.h>
00612 #elif defined (__AVR_ATxmega192A3__)
00613 #  include <avr/iox192a3.h>
00614 #elif defined (__AVR_ATxmega192A3U__)
00615 #  include <avr/iox192a3u.h>
00616 #elif defined (__AVR_ATxmega192C3__)
00617 #  include <avr/iox192c3.h>
00618 #elif defined (__AVR_ATxmega192D3__)
00619 #  include <avr/iox192d3.h>
00620 #elif defined (__AVR_ATxmega256A3__)
00621 #  include <avr/iox256a3.h>
00622 #elif defined (__AVR_ATxmega256A3U__)
00623 #  include <avr/iox256a3u.h>
00624 #elif defined (__AVR_ATxmega256A3B__)
00625 #  include <avr/iox256a3b.h>
00626 #elif defined (__AVR_ATxmega256A3BU__)
00627 #  include <avr/iox256a3bu.h>
00628 #elif defined (__AVR_ATxmega256C3__)
00629 #  include <avr/iox256c3.h>
00630 #elif defined (__AVR_ATxmega256D3__)
00631 #  include <avr/iox256d3.h>
00632 #elif defined (__AVR_ATxmega384C3__)
00633 #  include <avr/iox384c3.h>
00634 #elif defined (__AVR_ATxmega384D3__)
00635 #  include <avr/iox384d3.h>
00636 #elif defined (__AVR_ATA5700M322__)
00637 #  include <avr/ioa5700m322.h>
00638 #elif defined (__AVR_ATA5702M322__)
00639 #  include <avr/ioa5702m322.h>
00640 #elif defined (__AVR_ATA5787__)
00641 #  include <avr/ioa5787.h>
00642 #elif defined (__AVR_ATA5782__)
00643 #  include <avr/ioa5782.h>
00644 #elif defined (__AVR_ATA5790__)
00645 #  include <avr/ioa5790.h>
00646 #elif defined (__AVR_ATA5790N__)
```

```
00647 #  include <avr/ioa5790n.h>
00648 #elif defined (__AVR_ATA5791__)
00649 #  include <avr/ioa5791.h>
00650 #elif defined (__AVR_ATA5831__)
00651 #  include <avr/ioa5831.h>
00652 #elif defined (__AVR_ATA5835__)
00653 #  include <avr/ioa5835.h>
00654 #elif defined (__AVR_ATA5272__)
00655 #  include <avr/ioa5272.h>
00656 #elif defined (__AVR_ATA5505__)
00657 #  include <avr/ioa5505.h>
00658 #elif defined (__AVR_ATA5795__)
00659 #  include <avr/ioa5795.h>
00660 #elif defined (__AVR_ATA6285__)
00661 #  include <avr/ioa6285.h>
00662 #elif defined (__AVR_ATA6286__)
00663 #  include <avr/ioa6286.h>
00664 #elif defined (__AVR_ATA6289__)
00665 #  include <avr/ioa6289.h>
00666 #elif defined (__AVR_ATA6612C__)
00667 #  include <avr/ioa6612c.h>
00668 #elif defined (__AVR_ATA6613C__)
00669 #  include <avr/ioa6613c.h>
00670 #elif defined (__AVR_ATA6614Q__)
00671 #  include <avr/ioa6614q.h>
00672 #elif defined (__AVR_ATA6616C__)
00673 #  include <avr/ioa6616c.h>
00674 #elif defined (__AVR_ATA6617C__)
00675 #  include <avr/ioa6617c.h>
00676 #elif defined (__AVR_ATA664251__)
00677 #  include <avr/ioa664251.h>
00678 #elif defined (__AVR_ATA8210__)
00679 #  include <avr/ioa8210.h>
00680 #elif defined (__AVR_ATA8510__)
00681 #  include <avr/ioa8510.h>
00682 /* avr1: the following only supported for assembler programs */
00683 #elif defined (__AVR_ATtiny28__)
00684 #  include <avr/iotn28.h>
00685 #elif defined (__AVR_AT90S1200__)
00686 #  include <avr/io1200.h>
00687 #elif defined (__AVR_ATtiny15__)
00688 #  include <avr/iotn15.h>
00689 #elif defined (__AVR_ATtiny12__)
00690 #  include <avr/iotn12.h>
00691 #elif defined (__AVR_ATtiny11__)
00692 #  include <avr/iotn11.h>
00693 #elif defined (__AVR_M3000__)
00694 #  include <avr/iom3000.h>
00695 /* The headers for the AVR-Dx and AVR-Ex devices follow the uniform
00696    naming convention of io<mcu>.h.  Thus there is no need to list them
00697    all since they are covered by the __AVR_DEVICE_NAME__ case below.  */
00698
00699 #elif defined (__AVR_DEV_LIB_NAME__)
00700 /* This case is used for devices that are not supported by AVR-LibC but by
00701    means of an ATPACK device support pack.  For details see the avr-gcc Wiki.
00702    The macro is here due to the dreaded AVR-LibC naming for device headers
00703    like listed above, and that are impossible to predict by the compiler.  */
00704 #  define __concat__(a,b) a##b
00705 #  define __header1__(a,b) __concat__(a,b)
00706 #  define __AVR_DEVICE_HEADER__ <avr/__header1__(io,__AVR_DEV_LIB_NAME__).h>
00707 #  include __AVR_DEVICE_HEADER__
00708
00709 #elif defined (__AVR_DEVICE_NAME__)
00710 /* Since GCC v5:  __AVR_DEVICE_NAME__ is defined in the device-specs file
00711    to <mcu> qua  avr-gcc -mmcu=<mcu>  (except for cores like avr2).  */
00712 #  define __concat__(a,b) a##b
00713 #  define __header1__(a,b) __concat__(a,b)
00714 #  define __AVR_DEVICE_HEADER__ <avr/__header1__(io,__AVR_DEVICE_NAME__).h>
00715 #  include __AVR_DEVICE_HEADER__
00716
00717 #else
00718 #  if !defined(__COMPILING_AVR_LIBC__)
00719 #    warning "device type not defined"
```

```
00720 #  endif
00721 #endif
00722
00723 #include <avr/portpins.h>
00724
00725 #include <avr/common.h>
00726
00727 /* version.h may not be generated yet when configure is running.  */
00728 #if !defined (__CONFIGURING_AVR_LIBC__)
00729 #include <avr/version.h>
00730 #endif
00731
00732 #if __AVR_ARCH__ >= 100
00733 #  include <avr/xmega.h>
00734 #endif
00735
00736 /* Include fuse.h after individual IO header files. */
00737 #include <avr/fuse.h>
00738
00739 /* Include lock.h after individual IO header files. */
00740 #include <avr/lock.h>
00741
00742 #endif /* _AVR_IO_H_ */
```

22.32 lock.h File Reference

22.33 lock.h

Go to the documentation of this file.

```
00001 /* Copyright (c) 2007, Atmel Corporation
00002    All rights reserved.
00003
00004    Redistribution and use in source and binary forms, with or without
00005    modification, are permitted provided that the following conditions are met:
00006
00007    * Redistributions of source code must retain the above copyright
00008      notice, this list of conditions and the following disclaimer.
00009
00010    * Redistributions in binary form must reproduce the above copyright
00011      notice, this list of conditions and the following disclaimer in
00012      the documentation and/or other materials provided with the
00013      distribution.
00014
00015    * Neither the name of the copyright holders nor the names of
00016      contributors may be used to endorse or promote products derived
00017      from this software without specific prior written permission.
00018
00019   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
00020   AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
00021   IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
00022   ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
00023   LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
00024   CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
00025   SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
00026   INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
00027   CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
00028   ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
00029   POSSIBILITY OF SUCH DAMAGE. */
00030
00031 /* avr/lock.h - Lock Bits API */
00032
00033 #ifndef _AVR_LOCK_H_
00034 #define _AVR_LOCK_H_  1
00035
00036
00037 /** \file */
00038 /** \defgroup avr_lock <avr/lock.h>: Lockbit Support
00039
```

```
00040     \par Introduction
00041
00042     The Lockbit API allows a user to specify the lockbit settings for the
00043     specific AVR device they are compiling for. These lockbit settings will be
00044     placed in a special section in the ELF output file, after linking.
00045
00046     Programming tools can take advantage of the lockbit information embedded in
00047     the ELF file, by extracting this information and determining if the lockbits
00048     need to be programmed after programming the Flash and EEPROM memories.
00049     This also allows a single ELF file to contain all the
00050     information needed to program an AVR.
00051
00052     To use the Lockbit API, include the <avr/io.h> header file, which in turn
00053     automatically includes the individual I/O header file and the <avr/lock.h>
00054     file. These other two files provides everything necessary to set the AVR
00055     lockbits.
00056
00057     \par Lockbit API
00058
00059     Each I/O header file may define up to 3 macros that controls what kinds
00060     of lockbits are available to the user.
00061
00062     If __LOCK_BITS_EXIST is defined, then two lock bits are available to the
00063     user and 3 mode settings are defined for these two bits.
00064
00065     If __BOOT_LOCK_BITS_0_EXIST is defined, then the two BLB0 lock bits are
00066     available to the user and 4 mode settings are defined for these two bits.
00067
00068     If __BOOT_LOCK_BITS_1_EXIST is defined, then the two BLB1 lock bits are
00069     available to the user and 4 mode settings are defined for these two bits.
00070
00071     If __BOOT_LOCK_APPLICATION_TABLE_BITS_EXIST is defined then two lock bits
00072     are available to set the locking mode for the Application Table Section
00073     (which is used in the XMEGA family).
00074
00075     If __BOOT_LOCK_APPLICATION_BITS_EXIST is defined then two lock bits are
00076     available to set the locking mode for the Application Section (which is used
00077     in the XMEGA family).
00078
00079     If __BOOT_LOCK_BOOT_BITS_EXIST is defined then two lock bits are available
00080     to set the locking mode for the Boot Loader Section (which is used in the
00081     XMEGA family).
00082
00083     The AVR lockbit modes have inverted values, logical 1 for an unprogrammed
00084     (disabled) bit and logical 0 for a programmed (enabled) bit. The defined
00085     macros for each individual lock bit represent this in their definition by a
00086     bit-wise inversion of a mask. For example, the LB_MODE_3 macro is defined
00087     as:
00088     \code
00089     #define LB_MODE_3   (0xFC)
00090 `   \endcode
00091
00092     To combine the lockbit mode macros together to represent a whole byte,
00093     use the bitwise AND operator, like so:
00094     \code
00095     (LB_MODE_3 & BLB0_MODE_2)
00096     \endcode
00097
00098     <avr/lock.h> also defines a macro that provides a default lockbit value:
00099     LOCKBITS_DEFAULT which is defined to be 0xFF.
00100
00101     See the AVR device specific datasheet for more details about these
00102     lock bits and the available mode settings.
00103
00104     A convenience macro, LOCKMEM, is defined as a GCC attribute for a
00105     custom-named section of ".lock".
00106
00107     A convenience macro, LOCKBITS, is defined that declares a variable, __lock,
00108     of type unsigned char with the attribute defined by LOCKMEM. This variable
00109     allows the end user to easily set the lockbit data.
00110
00111     \note If a device-specific I/O header file has previously defined LOCKMEM,
00112     then LOCKMEM is not redefined. If a device-specific I/O header file has
```

```
00113     previously defined LOCKBITS, then LOCKBITS is not redefined. LOCKBITS is
00114     currently known to be defined in the I/O header files for the XMEGA devices.
00115
00116     \par API Usage Example
00117
00118     Putting all of this together is easy:
00119
00120     \code
00121     #include <avr/io.h>
00122
00123     LOCKBITS = (LB_MODE_1 & BLB0_MODE_3 & BLB1_MODE_4);
00124
00125     int main(void)
00126     {
00127         return 0;
00128     }
00129     \endcode
00130
00131     Or:
00132
00133     \code
00134     #include <avr/io.h>
00135
00136     unsigned char __lock __attribute__((section (".lock"))) =
00137         (LB_MODE_1 & BLB0_MODE_3 & BLB1_MODE_4);
00138
00139     int main(void)
00140     {
00141         return 0;
00142     }
00143     \endcode
00144
00145
00146
00147     However there are a number of caveats that you need to be aware of to
00148     use this API properly.
00149
00150     Be sure to include <avr/io.h> to get all of the definitions for the API.
00151     The LOCKBITS macro defines a global variable to store the lockbit data. This
00152     variable is assigned to its own linker section. Assign the desired lockbit
00153     values immediately in the variable initialization.
00154
00155     The .lock section in the ELF file will get its values from the initial
00156     variable assignment ONLY. This means that you can NOT assign values to
00157     this variable in functions and the new values will not be put into the
00158     ELF .lock section.
00159
00160     The global variable is declared in the LOCKBITS macro has two leading
00161     underscores, which means that it is reserved for the "implementation",
00162     meaning the library, so it will not conflict with a user-named variable.
00163
00164     You must initialize the lockbit variable to some meaningful value, even
00165     if it is the default value. This is because the lockbits default to a
00166     logical 1, meaning unprogrammed. Normal uninitialized data defaults to all
00167     locgial zeros. So it is vital that all lockbits are initialized, even with
00168     default data. If they are not, then the lockbits may not programmed to the
00169     desired settings and can possibly put your device into an unrecoverable
00170     state.
00171
00172     Be sure to have the -mmcu=<em>device</em> flag in your compile command line and
00173     your linker command line to have the correct device selected and to have
00174     the correct I/O header file included when you include <avr/io.h>.
00175
00176     You can print out the contents of the .lock section in the ELF file by
00177     using this command line:
00178     \code
00179     avr-objdump -s -j .lock <ELF file>
00180     \endcode
00181
00182 */
00183
00184
00185 #if !(defined(__ASSEMBLER__) || defined(__DOXYGEN__))
```

```
00186
00187 #ifndef LOCKMEM
00188 #define LOCKMEM  __attribute__((__used__, __section__ (".lock")))
00189 #endif
00190
00191 #ifndef LOCKBITS
00192 #define LOCKBITS unsigned char __lock LOCKMEM
00193 #endif
00194
00195 /* Lock Bit Modes */
00196 #if defined(__LOCK_BITS_EXIST)
00197 #define LB_MODE_1  (0xFF)
00198 #define LB_MODE_2  (0xFE)
00199 #define LB_MODE_3  (0xFC)
00200 #endif
00201
00202 #if defined(__BOOT_LOCK_BITS_0_EXIST)
00203 #define BLB0_MODE_1  (0xFF)
00204 #define BLB0_MODE_2  (0xFB)
00205 #define BLB0_MODE_3  (0xF3)
00206 #define BLB0_MODE_4  (0xF7)
00207 #endif
00208
00209 #if defined(__BOOT_LOCK_BITS_1_EXIST)
00210 #define BLB1_MODE_1  (0xFF)
00211 #define BLB1_MODE_2  (0xEF)
00212 #define BLB1_MODE_3  (0xCF)
00213 #define BLB1_MODE_4  (0xDF)
00214 #endif
00215
00216 #if defined(__BOOT_LOCK_APPLICATION_TABLE_BITS_EXIST)
00217 #define BLBAT0 ~_BV(2)
00218 #define BLBAT1 ~_BV(3)
00219 #endif
00220
00221 #if defined(__BOOT_LOCK_APPLICATION_BITS_EXIST)
00222 #define BLBA0 ~_BV(4)
00223 #define BLBA1 ~_BV(5)
00224 #endif
00225
00226 #if defined(__BOOT_LOCK_BOOT_BITS_EXIST)
00227 #define BLBB0 ~_BV(6)
00228 #define BLBB1 ~_BV(7)
00229 #endif
00230
00231 #ifndef LOCKBITS_DEFAULT
00232 #define LOCKBITS_DEFAULT (0xFF)
00233 #endif
00234
00235 #endif  /* !(__ASSEMBLER || __DOXYGEN__) */
00236
00237
00238 #endif /* _AVR_LOCK_H_ */
```

22.34 pgmspace.h File Reference

Macros

Macros

- #define PROGMEM_FAR __attribute__((__section__(".progmemx.data")))
- #define PROGMEM __attribute__((__progmem__))
- #define PSTR(str) ({ static const PROGMEM char c[ ] = (str); &c[0]; })
- #define PSTR_FAR(str) ({ static const PROGMEM_FAR char c[ ] = (str); pgm_get_far_address(c[0]); })
- #define pgm_get_far_address(var)

Macros reading from PROGMEM

- #define pgm_read_byte_near(__addr) __LPM ((uint16_t)(__addr))
- #define pgm_read_word_near(__addr) __LPM_word ((uint16_t)(__addr))
- #define pgm_read_dword_near(__addr) __LPM_dword ((uint16_t)(__addr))
- #define pgm_read_qword_near(__addr) __LPM_qword ((uint16_t)(__addr))
- #define pgm_read_float_near(addr) pgm_read_float (addr)
- #define pgm_read_ptr_near(__addr) ((void*) __LPM_word ((uint16_t)(__addr)))
- #define pgm_read_byte(__addr) pgm_read_byte_near(__addr)
- #define pgm_read_word(__addr) pgm_read_word_near(__addr)
- #define pgm_read_dword(__addr) pgm_read_dword_near(__addr)
- #define pgm_read_qword(__addr) pgm_read_qword_near(__addr)
- #define pgm_read_ptr(__addr) pgm_read_ptr_near(__addr)

Macros reading from PROGMEM_FAR

- #define pgm_read_byte_far(__addr) __ELPM (__addr)
- #define pgm_read_word_far(__addr) __ELPM_word (__addr)
- #define pgm_read_dword_far(__addr) __ELPM_dword (__addr)
- #define pgm_read_qword_far(__addr) __ELPM_qword (__addr)
- #define pgm_read_ptr_far(__addr) ((void*) __ELPM_word (__addr))

Functions

Functions reading from PROGMEM

- static char pgm_read_char (const char *addr)
- static unsigned char pgm_read_unsigned_char (const unsigned char *addr)
- static signed char pgm_read_signed_char (const signed char *addr)
- static uint8_t pgm_read_u8 (const uint8_t *addr)
- static int8_t pgm_read_i8 (const int8_t *addr)
- static short pgm_read_short (const short *addr)
- static unsigned short pgm_read_unsigned_short (const unsigned short *addr)
- static uint16_t pgm_read_u16 (const uint16_t *addr)
- static int16_t pgm_read_i16 (const int16_t *addr)
- static int pgm_read_int (const int *addr)
- static signed pgm_read_signed (const signed *addr)
- static unsigned pgm_read_unsigned (const unsigned *addr)
- static signed int pgm_read_signed_int (const signed int *addr)
- static unsigned int pgm_read_unsigned_int (const unsigned int *addr)
- static int24_t pgm_read_i24 (const int24_t *addr)
- static uint24_t pgm_read_u24 (const uint24_t *addr)
- static uint32_t pgm_read_u32 (const uint32_t *addr)
- static int32_t pgm_read_i32 (const int32_t *addr)
- static long pgm_read_long (const long *addr)
- static unsigned long pgm_read_unsigned_long (const unsigned long *addr)
- static long long pgm_read_long_long (const long long *addr)
- static unsigned long long pgm_read_unsigned_long_long (const unsigned long long *addr)
- static uint64_t pgm_read_u64 (const uint64_t *addr)
- static int64_t pgm_read_i64 (const int64_t *addr)
- static float pgm_read_float (const float *addr)
- static double pgm_read_double (const double *addr)
- static long double pgm_read_long_double (const long double *addr)

Functions reading from PROGMEM_FAR

- static char pgm_read_char_far (uint_farptr_t addr)
- static unsigned char pgm_read_unsigned_char_far (uint_farptr_t addr)
- static signed char pgm_read_signed_char_far (uint_farptr_t addr)
- static uint8_t pgm_read_u8_far (uint_farptr_t addr)
- static int8_t pgm_read_i8_far (uint_farptr_t addr)
- static short pgm_read_short_far (uint_farptr_t addr)
- static unsigned short pgm_read_unsigned_short_far (uint_farptr_t addr)

- static uint16_t pgm_read_u16_far (uint_farptr_t addr)
- static int16_t pgm_read_i16_far (uint_farptr_t addr)
- static int pgm_read_int_far (uint_farptr_t addr)
- static unsigned pgm_read_unsigned_far (uint_farptr_t addr)
- static unsigned int pgm_read_unsigned_int_far (uint_farptr_t addr)
- static signed pgm_read_signed_far (uint_farptr_t addr)
- static signed int pgm_read_signed_int_far (uint_farptr_t addr)
- static long pgm_read_long_far (uint_farptr_t addr)
- static unsigned long pgm_read_unsigned_long_far (uint_farptr_t addr)
- static int24_t pgm_read_i24_far (uint_farptr_t addr)
- static uint24_t pgm_read_u24_far (uint_farptr_t addr)
- static uint32_t pgm_read_u32_far (uint_farptr_t addr)
- static int32_t pgm_read_i32_far (uint_farptr_t addr)
- static long long pgm_read_long_long_far (uint_farptr_t addr)
- static unsigned long long pgm_read_unsigned_long_long_far (uint_farptr_t addr)
- static uint64_t pgm_read_u64_far (uint_farptr_t addr)
- static int64_t pgm_read_i64_far (uint_farptr_t addr)
- static float pgm_read_float_far (uint_farptr_t addr)
- static double pgm_read_double_far (uint_farptr_t addr)
- static long double pgm_read_long_double_far (uint_farptr_t addr)

Functions with a PROGMEM argument

Similar to the functions from <string.h>, but with one string argument located in the lower 64 KiB segment of program memory.
For similar functions with address-space __flash, see <avr/flash.h>

- static size_t strlen_P (const char *s)
- const void * memchr_P (const void *, int __val, size_t __len)
- int memcmp_P (const void *, const void *, size_t)
- void * memccpy_P (void *, const void *, int __val, size_t)
- void * memcpy_P (void *, const void *, size_t)
- void * memmem_P (const void *, size_t, const void *, size_t)
- const void * memrchr_P (const void *, int __val, size_t __len)
- char * strcat_P (char *, const char *)
- const char * strchr_P (const char *, int __val)
- const char * strchrnul_P (const char *, int __val)
- int strcmp_P (const char *, const char *)
- char * strcpy_P (char *, const char *)
- char * stpcpy_P (char *, const char *)
- int strcasecmp_P (const char *, const char *)
- char * strcasestr_P (const char *, const char *)
- size_t strcspn_P (const char *__s, const char *__reject)
- size_t strlcat_P (char *, const char *, size_t)
- size_t strlcpy_P (char *, const char *, size_t)
- size_t strnlen_P (const char *, size_t)
- int strncmp_P (const char *, const char *, size_t)
- int strncasecmp_P (const char *, const char *, size_t)
- char * strncat_P (char *, const char *, size_t)
- char * strncpy_P (char *, const char *, size_t)
- char * strpbrk_P (const char *__s, const char *__accept)
- const char * strrchr_P (const char *, int __val)
- char * strsep_P (char **__sp, const char *__delim)
- size_t strspn_P (const char *__s, const char *__accept)
- char * strstr_P (const char *, const char *)
- char * strtok_P (char *__s, const char *__delim)
- char * strtok_rP (char *__s, const char *__delim, char **__last)

Functions with a PROGMEM_FAR argument

Similar to the functions from <string.h>, but with one string argument located in program memory.
For similar functions with address-space __flashx, see <avr/flash.h>

- size_t strlen_PF (uint_farptr_t src)

- size_t strnlen_PF (uint_farptr_t src, size_t len)
- void * memcpy_PF (void *dest, uint_farptr_t src, size_t len)
- char * strcpy_PF (char *dest, uint_farptr_t src)
- char * stpcpy_PF (char *dest, uint_farptr_t src)
- char * strncpy_PF (char *dest, uint_farptr_t src, size_t len)
- char * strcat_PF (char *dest, uint_farptr_t src)
- size_t strlcat_PF (char *dst, uint_farptr_t src, size_t siz)
- char * strncat_PF (char *dest, uint_farptr_t src, size_t len)
- int strcmp_PF (const char *s1, uint_farptr_t s2)
- int strncmp_PF (const char *s1, uint_farptr_t s2, size_t n)
- int strcasecmp_PF (const char *s1, uint_farptr_t s2)
- int strncasecmp_PF (const char *s1, uint_farptr_t s2, size_t n)
- uint_farptr_t strchr_PF (uint_farptr_t, int __val)
- char * strstr_PF (const char *s1, uint_farptr_t s2)
- size_t strlcpy_PF (char *dst, uint_farptr_t src, size_t siz)
- int memcmp_PF (const void *, uint_farptr_t, size_t)

Templates

- template<typename T >
  T pgm_read< T > (const T *addr)
- template<typename T >
  T pgm_read_far< T > (uint_farptr_t addr)

22.35 pgmspace.h

Go to the documentation of this file.

```
00001 /* Copyright (c) 2002-2007  Marek Michalkiewicz
00002    Copyright (c) 2006, Carlos Lamas
00003    Copyright (c) 2009-2010, Jan Waclawek
00004    All rights reserved.
00005
00006    Redistribution and use in source and binary forms, with or without
00007    modification, are permitted provided that the following conditions are met:
00008
00009    * Redistributions of source code must retain the above copyright
00010      notice, this list of conditions and the following disclaimer.
00011    * Redistributions in binary form must reproduce the above copyright
00012      notice, this list of conditions and the following disclaimer in
00013      the documentation and/or other materials provided with the
00014      distribution.
00015    * Neither the name of the copyright holders nor the names of
00016      contributors may be used to endorse or promote products derived
00017      from this software without specific prior written permission.
00018
00019   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
00020   AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
00021   IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
00022   ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
00023   LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
00024   CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
00025   SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
00026   INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
00027   CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
00028   ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
00029   POSSIBILITY OF SUCH DAMAGE. */
00030
00031 /*
00032    pgmspace.h
00033
00034    Contributors:
00035      Created by Marek Michalkiewicz <marekm@linux.org.pl>
00036      Eric B. Weddington <eric@ecentral.com>
00037      Wolfgang Haidinger <wh@vmars.tuwien.ac.at> (pgm_read_dword())
00038      Ivanov Anton <anton@arc.com.ru> (pgm_read_float())
```

```
00039  */
00040
00041 /** \file */
00042 /** \defgroup avr_pgmspace <avr/pgmspace.h>: Program Space Utilities
00043     \code
00044     #include <avr/pgmspace.h>
00045     \endcode
00046
00047     The functions in this module provide interfaces for a program to access
00048     data stored in program space (flash memory) of the device.<br>
00049     For a different approach using named address-spaces like #__flash,
00050     see \ref avr_flash "<avr/flash.h>".
00051     For functions to read fixed-point values from program memory,
00052     see \ref avr_stdfix "<stdfix.h>".
00053
00054     \note These functions are an attempt to provide some compatibility with
00055     header files that come with IAR C, to make porting applications between
00056     different compilers easier.  This is not 100% compatibility though.
00057
00058     \note If you are working with strings which are completely based in RAM,
00059     use the standard string functions described in \ref avr_string.
00060
00061     \note If possible, put your constant tables in the lower 64 KiB and use
00062     #pgm_read_byte, #pgm_read_char or #pgm_read_u8 etc. instead of
00063     #pgm_read_byte_far since it is more efficient that
00064     way, and you can still use the upper 64 KiB for executable code.
00065     All functions that are suffixed with a \c _P \e require their
00066     arguments to be in the lower 64 KiB of the flash ROM, as they do
00067     not use \c ELPM instructions.  This is normally not a big concern as
00068     the linker setup arranges any program space constants declared
00069     using #PROGMEM to be placed right after the interrupt vectors,
00070     and in front of any executable code.  However,
00071     it can become a problem if there are too many of these constants, or
00072     for bootloaders on devices with more than 64 KiB of ROM.
00073     <em>All these functions will not work in that situation.</em>
00074
00075     \note For <b>Xmega</b> devices, make sure the NVM controller
00076     command register (\c NVM.CMD or \c NVM_CMD) is set to 0x00 (NOP)
00077     before using any of these functions.
00078 */
00079
00080 #ifndef __PGMSPACE_H_
00081 #define __PGMSPACE_H_ 1
00082
00083 #ifndef __DOXYGEN__
00084 #define __need_size_t
00085 #endif
00086 #include <inttypes.h>
00087 #include <stddef.h>
00088
00089 #ifndef __DOXYGEN__
00090
00091 #include <bits/attribs.h>
00092 #include <bits/lpm-elpm.h>
00093
00094 #define PROGMEM __ATTR_PROGMEM__
00095
00096 #endif  /* !__DOXYGEN__ */
00097
00098 /** \name Macros */
00099
00100 /**
00101    \ingroup avr_pgmspace
00102    Attribute to use in order to declare a read-only object being located in
00103    far flash ROM.  This is similar to #PROGMEM, except that it puts
00104    the static storage object in section
00105    <tt>\ref sec_dot_progmemx ".progmemx.data"</tt>.
00106    In order to access the object,
00107    the <tt>pgm_read_*_far</tt> and \c _PF functions declared in this header
00108    can be used.  In order to get its address, see pgm_get_far_address().
00109
00110    It only makes sense to put read-only objects in this section,
00111    though the compiler does not diagnose when this is not the case.
```

```
00112
00113    As an alternative available since AVR-LibC v2.3 and avr-gcc v15,
00114    you can use the 24-bit address-space #__flashx and functions from
00115    \ref avr_flash "<avr/flash.h>" that work the same like the \c _far
00116    functions.
00117
00118    \since AVR-LibC v2.2  */
00119 #define PROGMEM_FAR __attribute__((__section__(".progmemx.data")))
00120
00121 #ifdef __DOXYGEN__
00122
00123 /**
00124    \ingroup avr_pgmspace
00125    Attribute to use in order to declare a read-only object in static storage
00126    being located in the lower 64 KiB of flash ROM.
00127
00128    Objects in this section will be located in the
00129    <tt>\ref sec_dot_progmem ".progmem.data"</tt> section.
00130    In order to access PROGMEM objects:
00131    <dl>
00132    <dt>Reduced Tiny (AVRrc) devices</dt>
00133    <dd>No extra code is needed. Use vanilla C/C++ code to access.</dd>
00134    <dt>All other devices (non-AVRrc)</dt>
00135    <dd>(Inline) assembly must be used in order to read from <tt>PROGMEM</tt>
00136      objects, like for example by means of the <tt>pgm_read_xxx</tt> functions
00137      and macros as declared in this header.</dd>
00138    </dl>
00139
00140    For an alternative, see the \c #__flash named address-space which is
00141    supported since
00142    <a href="https://gcc.gnu.org/gcc-4.7/changes.html#avr">avr-gcc v4.7</a>,
00143    and \ref avr_flash "<avr/flash.h>" which exists since AVR-LibC v2.3.  */
00144 #define PROGMEM __attribute__((__progmem__))
00145
00146 /** \name Functions reading from PROGMEM */
00147
00148 /** \ingroup avr_pgmspace
00149     Read a <tt>char</tt> from 16-bit address \p addr.
00150     The address is in the lower 64 KiB of program memory.
00151     \since AVR-LibC v2.2 */
00152 static inline char pgm_read_char (const char *addr);
00153
00154 /** \ingroup avr_pgmspace
00155     Read an <tt>unsigned char</tt> from 16-bit address \p addr.
00156     The address is in the lower 64 KiB of program memory.
00157     \since AVR-LibC v2.2 */
00158 static inline unsigned char pgm_read_unsigned_char (const unsigned char *addr);
00159
00160 /** \ingroup avr_pgmspace
00161     Read a <tt>signed char</tt> from 16-bit address \p addr.
00162     The address is in the lower 64 KiB of program memory.
00163     \since AVR-LibC v2.2 */
00164 static inline signed char pgm_read_signed_char (const signed char *addr);
00165
00166 /** \ingroup avr_pgmspace
00167     Read an <tt>uint8_t</tt> from 16-bit address \p addr.
00168     The address is in the lower 64 KiB of program memory.
00169     \since AVR-LibC v2.2 */
00170 static inline uint8_t pgm_read_u8 (const uint8_t *addr);
00171
00172 /** \ingroup avr_pgmspace
00173     Read an <tt>int8_t</tt> from 16-bit address \p addr.
00174     The address is in the lower 64 KiB of program memory.
00175     \since AVR-LibC v2.2 */
00176 static inline int8_t pgm_read_i8 (const int8_t *addr);
00177
00178 /** \ingroup avr_pgmspace
00179     Read a <tt>short</tt> from 16-bit address \p addr.
00180     The address is in the lower 64 KiB of program memory.
00181     \since AVR-LibC v2.2 */
00182 static inline short pgm_read_short (const short *addr);
00183
00184 /** \ingroup avr_pgmspace
```

```
00185     Read an <tt>unsigned short</tt> from 16-bit address \p addr.
00186     The address is in the lower 64 KiB of program memory.
00187     \since AVR-LibC v2.2 */
00188 static inline unsigned short pgm_read_unsigned_short (const unsigned short *addr);
00189
00190 /** \ingroup avr_pgmspace
00191     Read an <tt>uint16_t</tt> from 16-bit address \p addr.
00192     The address is in the lower 64 KiB of program memory.
00193     \since AVR-LibC v2.2 */
00194 static inline uint16_t pgm_read_u16 (const uint16_t *addr);
00195
00196 /** \ingroup avr_pgmspace
00197     Read an <tt>int16_t</tt> from 16-bit address \p addr.
00198     The address is in the lower 64 KiB of program memory.
00199     \since AVR-LibC v2.2 */
00200 static inline int16_t pgm_read_i16 (const int16_t *addr);
00201
00202 /** \ingroup avr_pgmspace
00203     Read an <tt>int</tt> from 16-bit address \p addr.
00204     The address is in the lower 64 KiB of program memory.
00205     \since AVR-LibC v2.2 */
00206 static inline int pgm_read_int (const int *addr);
00207
00208 /** \ingroup avr_pgmspace
00209     Read a <tt>signed</tt> from 16-bit address \p addr.
00210     The address is in the lower 64 KiB of program memory.
00211     \since AVR-LibC v2.2 */
00212 static inline signed pgm_read_signed (const signed *addr);
00213
00214 /** \ingroup avr_pgmspace
00215     Read an <tt>unsigned</tt> from 16-bit address \p addr.
00216     The address is in the lower 64 KiB of program memory.
00217     \since AVR-LibC v2.2 */
00218 static inline unsigned pgm_read_unsigned (const unsigned *addr);
00219
00220 /** \ingroup avr_pgmspace
00221     Read a <tt>signed int</tt> from 16-bit address \p addr.
00222     The address is in the lower 64 KiB of program memory.
00223     \since AVR-LibC v2.2 */
00224 static inline signed int pgm_read_signed_int (const signed int *addr);
00225
00226 /** \ingroup avr_pgmspace
00227     Read an <tt>unsigned int</tt> from 16-bit address \p addr.
00228     The address is in the lower 64 KiB of program memory.
00229     \since AVR-LibC v2.2 */
00230 static inline unsigned int pgm_read_unsigned_int (const unsigned int *addr);
00231
00232 /** \ingroup avr_pgmspace
00233     Read an \c #int24_t from 16-bit address \p addr.
00234     The address is in the lower 64 KiB of program memory.
00235     \since AVR-LibC v2.2 */
00236 static inline int24_t pgm_read_i24 (const int24_t *addr);
00237
00238 /** \ingroup avr_pgmspace
00239     Read an \c #uint24_t from 16-bit address \p addr.
00240     The address is in the lower 64 KiB of program memory.
00241     \since AVR-LibC v2.2 */
00242 static inline uint24_t pgm_read_u24 (const uint24_t *addr);
00243
00244 /** \ingroup avr_pgmspace
00245     Read an <tt>uint32_t</tt> from 16-bit address \p addr.
00246     The address is in the lower 64 KiB of program memory.
00247     \since AVR-LibC v2.2 */
00248 static inline uint32_t pgm_read_u32 (const uint32_t *addr);
00249
00250 /** \ingroup avr_pgmspace
00251     Read an <tt>int32_t</tt> from 16-bit address \p addr.
00252     The address is in the lower 64 KiB of program memory.
00253     \since AVR-LibC v2.2 */
00254 static inline int32_t pgm_read_i32 (const int32_t *addr);
00255
00256 /** \ingroup avr_pgmspace
00257     Read a <tt>long</tt> from 16-bit address \p addr.
```

```
00258     The address is in the lower 64 KiB of program memory.
00259     \since AVR-LibC v2.2 */
00260 static inline long pgm_read_long (const long *addr);
00261
00262 /** \ingroup avr_pgmspace
00263     Read an <tt>unsigned long</tt> from 16-bit address \p addr.
00264     The address is in the lower 64 KiB of program memory.
00265     \since AVR-LibC v2.2 */
00266 static inline unsigned long pgm_read_unsigned_long (const unsigned long *addr);
00267
00268 /** \ingroup avr_pgmspace
00269     Read a <tt>long long</tt> from 16-bit address \p addr.
00270     The address is in the lower 64 KiB of program memory.
00271     \since AVR-LibC v2.2 */
00272 static inline long long pgm_read_long_long (const long long *addr);
00273
00274 /** \ingroup avr_pgmspace
00275     Read an <tt>unsigned long long</tt> from 16-bit address \p addr.
00276     The address is in the lower 64 KiB of program memory.
00277     \since AVR-LibC v2.2 */
00278 static inline unsigned long long pgm_read_unsigned_long_long (const unsigned long long
      *addr);
00279
00280 /** \ingroup avr_pgmspace
00281     Read an <tt>uint64_t</tt> from 16-bit address \p addr.
00282     The address is in the lower 64 KiB of program memory.
00283     \since AVR-LibC v2.2 */
00284 static inline uint64_t pgm_read_u64 (const uint64_t *addr);
00285
00286 /** \ingroup avr_pgmspace
00287     Read an <tt>int64_t</tt> from 16-bit address \p addr.
00288     The address is in the lower 64 KiB of program memory.
00289     \since AVR-LibC v2.2 */
00290 static inline int64_t pgm_read_i64 (const int64_t *addr);
00291
00292 /** \ingroup avr_pgmspace
00293     Read a <tt>float</tt> from 16-bit address \p addr.
00294     The address is in the lower 64 KiB of program memory. */
00295 static inline float pgm_read_float (const float *addr);
00296
00297 /** \ingroup avr_pgmspace
00298     Read a <tt>double</tt> from 16-bit address \p addr.
00299     The address is in the lower 64 KiB of program memory.
00300     \since AVR-LibC v2.2 */
00301 static inline double pgm_read_double (const double *addr);
00302
00303 /** \ingroup avr_pgmspace
00304     Read a <tt>long double</tt> from 16-bit address \p addr.
00305     The address is in the lower 64 KiB of program memory.
00306     \since AVR-LibC v2.2 */
00307 static inline long double pgm_read_long_double (const long double *addr);
00308
00309 #else /* !DOXYGEN */
00310
00311 #include <bits/def-pgm-read.h>
00312
00313 _Avrlibc_Def_Pgm_1 (char, char)
00314 _Avrlibc_Def_Pgm_1 (unsigned_char, unsigned char)
00315 _Avrlibc_Def_Pgm_1 (signed_char, signed char)
00316 _Avrlibc_Def_Pgm_1 (u8, uint8_t)
00317 _Avrlibc_Def_Pgm_1 (i8, int8_t)
00318 #if __SIZEOF_INT__ == 1
00319 _Avrlibc_Def_Pgm_1 (int, int)
00320 _Avrlibc_Def_Pgm_1 (signed, signed)
00321 _Avrlibc_Def_Pgm_1 (unsigned, unsigned)
00322 _Avrlibc_Def_Pgm_1 (signed_int, signed int)
00323 _Avrlibc_Def_Pgm_1 (unsigned_int, unsigned int)
00324 #endif
00325 #if __SIZEOF_SHORT__ == 1
00326 _Avrlibc_Def_Pgm_1 (short, short)
00327 _Avrlibc_Def_Pgm_1 (unsigned_short, unsigned short)
00328 #endif
00329
```

```
00330 _Avrlibc_Def_Pgm_2 (u16, uint16_t)
00331 _Avrlibc_Def_Pgm_2 (i16, int16_t)
00332 #if __SIZEOF_INT__ == 2
00333 _Avrlibc_Def_Pgm_2 (int, int)
00334 _Avrlibc_Def_Pgm_2 (signed, signed)
00335 _Avrlibc_Def_Pgm_2 (unsigned, unsigned)
00336 _Avrlibc_Def_Pgm_2 (signed_int, signed int)
00337 _Avrlibc_Def_Pgm_2 (unsigned_int, unsigned int)
00338 #endif
00339 #if __SIZEOF_SHORT__ == 2
00340 _Avrlibc_Def_Pgm_2 (short, short)
00341 _Avrlibc_Def_Pgm_2 (unsigned_short, unsigned short)
00342 #endif
00343 #if __SIZEOF_LONG__ == 2
00344 _Avrlibc_Def_Pgm_2 (long, long)
00345 _Avrlibc_Def_Pgm_2 (unsigned_long, unsigned long)
00346 #endif
00347
00348 #if defined(__INT24_MAX__)
00349 _Avrlibc_Def_Pgm_3 (i24, int24_t)
00350 _Avrlibc_Def_Pgm_3 (u24, uint24_t)
00351 #endif /* Have __int24 */
00352
00353 _Avrlibc_Def_Pgm_4 (u32, uint32_t)
00354 _Avrlibc_Def_Pgm_4 (i32, int32_t)
00355 _Avrlibc_Def_Pgm_4 (float, float)
00356 #if __SIZEOF_LONG__ == 4
00357 _Avrlibc_Def_Pgm_4 (long, long)
00358 _Avrlibc_Def_Pgm_4 (unsigned_long, unsigned long)
00359 #endif
00360 #if __SIZEOF_LONG_LONG__ == 4
00361 _Avrlibc_Def_Pgm_4 (long_long, long long)
00362 _Avrlibc_Def_Pgm_4 (unsigned_long_long, unsigned long long)
00363 #endif
00364 #if __SIZEOF_DOUBLE__ == 4
00365 _Avrlibc_Def_Pgm_4 (double, double)
00366 #endif
00367 #if __SIZEOF_LONG_DOUBLE__ == 4
00368 _Avrlibc_Def_Pgm_4 (long_double, long double)
00369 #endif
00370
00371 #if __SIZEOF_LONG_LONG__ == 8
00372 _Avrlibc_Def_Pgm_8 (u64, uint64_t)
00373 _Avrlibc_Def_Pgm_8 (i64, int64_t)
00374 _Avrlibc_Def_Pgm_8 (long_long, long long)
00375 _Avrlibc_Def_Pgm_8 (unsigned_long_long, unsigned long long)
00376 #endif
00377 #if __SIZEOF_DOUBLE__ == 8
00378 _Avrlibc_Def_Pgm_8 (double, double)
00379 #endif
00380 #if __SIZEOF_LONG_DOUBLE__ == 8
00381 _Avrlibc_Def_Pgm_8 (long_double, long double)
00382 #endif
00383
00384 #endif /* DOXYGEN */
00385
00386 #ifdef __DOXYGEN__
00387
00388 /** \name Functions reading from PROGMEM_FAR */
00389
00390 /** \ingroup avr_pgmspace
00391     Read a <tt>char</tt> from far address \p addr.
00392     The address is in the program memory.
00393     \since AVR-LibC v2.2 */
00394 static inline char pgm_read_char_far (uint_farptr_t addr);
00395
00396 /** \ingroup avr_pgmspace
00397     Read an <tt>unsigned char</tt> from far address \p addr.
00398     The address is in the program memory.
00399     \since AVR-LibC v2.2 */
00400 static inline unsigned char pgm_read_unsigned_char_far (uint_farptr_t addr);
00401
00402 /** \ingroup avr_pgmspace
```

```
00403     Read a <tt>signed char</tt> from far address \p addr.
00404     The address is in the program memory.
00405     \since AVR-LibC v2.2 */
00406 static inline signed char pgm_read_signed_char_far (uint_farptr_t addr);
00407
00408 /** \ingroup avr_pgmspace
00409     Read an <tt>uint8_t</tt> from far address \p addr.
00410     The address is in the program memory.
00411     \since AVR-LibC v2.2 */
00412 static inline uint8_t pgm_read_u8_far (uint_farptr_t addr);
00413
00414 /** \ingroup avr_pgmspace
00415     Read an <tt>int8_t</tt> from far address \p addr.
00416     The address is in the program memory.
00417     \since AVR-LibC v2.2 */
00418 static inline int8_t pgm_read_i8_far (uint_farptr_t addr);
00419
00420 /** \ingroup avr_pgmspace
00421     Read a <tt>short</tt> from far address \p addr.
00422     The address is in the program memory.
00423     \since AVR-LibC v2.2 */
00424 static inline short pgm_read_short_far (uint_farptr_t addr);
00425
00426 /** \ingroup avr_pgmspace
00427     Read an <tt>unsigned short</tt> from far address \p addr.
00428     The address is in the program memory.
00429     \since AVR-LibC v2.2 */
00430 static inline unsigned short pgm_read_unsigned_short_far (uint_farptr_t addr);
00431
00432 /** \ingroup avr_pgmspace
00433     Read an <tt>uint16_t</tt> from far address \p addr.
00434     The address is in the program memory.
00435     \since AVR-LibC v2.2 */
00436 static inline uint16_t pgm_read_u16_far (uint_farptr_t addr);
00437
00438 /** \ingroup avr_pgmspace
00439     Read an <tt>int16_t</tt> from far address \p addr.
00440     The address is in the program memory.
00441     \since AVR-LibC v2.2 */
00442 static inline int16_t pgm_read_i16_far (uint_farptr_t addr);
00443
00444 /** \ingroup avr_pgmspace
00445     Read an <tt>int</tt> from far address \p addr.
00446     The address is in the program memory.
00447     \since AVR-LibC v2.2 */
00448 static inline int pgm_read_int_far (uint_farptr_t addr);
00449
00450 /** \ingroup avr_pgmspace
00451     Read an <tt>unsigned</tt> from far address \p addr.
00452     The address is in the program memory.
00453     \since AVR-LibC v2.2 */
00454 static inline unsigned pgm_read_unsigned_far (uint_farptr_t addr);
00455
00456 /** \ingroup avr_pgmspace
00457     Read an <tt>unsigned int</tt> from far address \p addr.
00458     The address is in the program memory.
00459     \since AVR-LibC v2.2 */
00460 static inline unsigned int pgm_read_unsigned_int_far (uint_farptr_t addr);
00461
00462 /** \ingroup avr_pgmspace
00463     Read a <tt>signed</tt> from far address \p addr.
00464     The address is in the program memory.
00465     \since AVR-LibC v2.2 */
00466 static inline signed pgm_read_signed_far (uint_farptr_t addr);
00467
00468 /** \ingroup avr_pgmspace
00469     Read a <tt>signed int</tt> from far address \p addr.
00470     The address is in the program memory.
00471     \since AVR-LibC v2.2 */
00472 static inline signed int pgm_read_signed_int_far (uint_farptr_t addr);
00473
00474 /** \ingroup avr_pgmspace
00475     Read a <tt>long</tt> from far address \p addr.
```

```
00476     The address is in the program memory.
00477     \since AVR-LibC v2.2 */
00478 static inline long pgm_read_long_far (uint_farptr_t addr);
00479
00480 /** \ingroup avr_pgmspace
00481     Read an <tt>unsigned long</tt> from far address \p addr.
00482     The address is in the program memory.
00483     \since AVR-LibC v2.2 */
00484 static inline unsigned long pgm_read_unsigned_long_far (uint_farptr_t addr);
00485
00486 /** \ingroup avr_pgmspace
00487     Read an \c #int24_t from far address \p addr.
00488     The address is in the program memory.
00489     \since AVR-LibC v2.2 */
00490 static inline int24_t pgm_read_i24_far (uint_farptr_t addr);
00491
00492 /** \ingroup avr_pgmspace
00493     Read an \c #uint24_t from far address \p addr.
00494     The address is in the program memory.
00495     \since AVR-LibC v2.2 */
00496 static inline uint24_t pgm_read_u24_far (uint_farptr_t addr);
00497
00498 /** \ingroup avr_pgmspace
00499     Read an <tt>uint32_t</tt> from far address \p addr.
00500     The address is in the program memory.
00501     \since AVR-LibC v2.2 */
00502 static inline uint32_t pgm_read_u32_far (uint_farptr_t addr);
00503
00504 /** \ingroup avr_pgmspace
00505     Read an <tt>int32_t</tt> from far address \p addr.
00506     The address is in the program memory.
00507     \since AVR-LibC v2.2 */
00508 static inline int32_t pgm_read_i32_far (uint_farptr_t addr);
00509
00510 /** \ingroup avr_pgmspace
00511     Read a <tt>long long</tt> from far address \p addr.
00512     The address is in the program memory.
00513     \since AVR-LibC v2.2 */
00514 static inline long long pgm_read_long_long_far (uint_farptr_t addr);
00515
00516 /** \ingroup avr_pgmspace
00517     Read an <tt>unsigned long long</tt> from far address \p addr.
00518     The address is in the program memory.
00519     \since AVR-LibC v2.2 */
00520 static inline unsigned long long pgm_read_unsigned_long_long_far (uint_farptr_t addr);
00521
00522 /** \ingroup avr_pgmspace
00523     Read an <tt>uint64_t</tt> from far address \p addr.
00524     The address is in the program memory.
00525     \since AVR-LibC v2.2 */
00526 static inline uint64_t pgm_read_u64_far (uint_farptr_t addr);
00527
00528 /** \ingroup avr_pgmspace
00529     Read an <tt>int64_t</tt> from far address \p addr.
00530     The address is in the program memory.
00531     \since AVR-LibC v2.2 */
00532 static inline int64_t pgm_read_i64_far (uint_farptr_t addr);
00533
00534 /** \ingroup avr_pgmspace
00535     Read a <tt>float</tt> from far address \p addr.
00536     The address is in the program memory. */
00537 static inline float pgm_read_float_far (uint_farptr_t addr);
00538
00539 /** \ingroup avr_pgmspace
00540     Read a <tt>double</tt> from far address \p addr.
00541     The address is in the program memory.
00542     \since AVR-LibC v2.2 */
00543 static inline double pgm_read_double_far (uint_farptr_t addr);
00544
00545 /** \ingroup avr_pgmspace
00546     Read a <tt>long double</tt> from far address \p addr.
00547     The address is in the program memory.
00548     \since AVR-LibC v2.2 */
```

```
00549 static inline long double pgm_read_long_double_far (uint_farptr_t addr);
00550
00551 #else /* !DOXYGEN */
00552
00553 #include <bits/def-pgm-read-far.h>
00554
00555 _Avrlibc_Def_Pgm_Far_1 (char, char)
00556 _Avrlibc_Def_Pgm_Far_1 (unsigned_char, unsigned char)
00557 _Avrlibc_Def_Pgm_Far_1 (signed_char, signed char)
00558 _Avrlibc_Def_Pgm_Far_1 (u8, uint8_t)
00559 _Avrlibc_Def_Pgm_Far_1 (i8, int8_t)
00560 #if __SIZEOF_INT__ == 1
00561 _Avrlibc_Def_Pgm_Far_1 (int, int)
00562 _Avrlibc_Def_Pgm_Far_1 (unsigned, unsigned)
00563 _Avrlibc_Def_Pgm_Far_1 (unsigned_int, unsigned int)
00564 _Avrlibc_Def_Pgm_Far_1 (signed, signed)
00565 _Avrlibc_Def_Pgm_Far_1 (signed_int, signed int)
00566 #endif
00567 #if __SIZEOF_SHORT__ == 1
00568 _Avrlibc_Def_Pgm_Far_1 (short, short)
00569 _Avrlibc_Def_Pgm_Far_1 (unsigned_short, unsigned short)
00570 #endif
00571
00572 _Avrlibc_Def_Pgm_Far_2 (u16, uint16_t)
00573 _Avrlibc_Def_Pgm_Far_2 (i16, int16_t)
00574 #if __SIZEOF_INT__ == 2
00575 _Avrlibc_Def_Pgm_Far_2 (int, int)
00576 _Avrlibc_Def_Pgm_Far_2 (unsigned, unsigned)
00577 _Avrlibc_Def_Pgm_Far_2 (unsigned_int, unsigned int)
00578 _Avrlibc_Def_Pgm_Far_2 (signed, signed)
00579 _Avrlibc_Def_Pgm_Far_2 (signed_int, signed int)
00580 #endif
00581 #if __SIZEOF_SHORT__ == 2
00582 _Avrlibc_Def_Pgm_Far_2 (short, short)
00583 _Avrlibc_Def_Pgm_Far_2 (unsigned_short, unsigned short)
00584 #endif
00585 #if __SIZEOF_LONG__ == 2
00586 _Avrlibc_Def_Pgm_Far_2 (long, long)
00587 _Avrlibc_Def_Pgm_Far_2 (unsigned_long, unsigned long)
00588 #endif
00589
00590 #if defined(__INT24_MAX__)
00591 _Avrlibc_Def_Pgm_Far_3 (i24, int24_t)
00592 _Avrlibc_Def_Pgm_Far_3 (u24, uint24_t)
00593 #endif /* Have __int24 */
00594
00595 _Avrlibc_Def_Pgm_Far_4 (u32, uint32_t)
00596 _Avrlibc_Def_Pgm_Far_4 (i32, int32_t)
00597 _Avrlibc_Def_Pgm_Far_4 (float, float)
00598 #if __SIZEOF_LONG__ == 4
00599 _Avrlibc_Def_Pgm_Far_4 (long, long)
00600 _Avrlibc_Def_Pgm_Far_4 (unsigned_long, unsigned long)
00601 #endif
00602 #if __SIZEOF_LONG_LONG__ == 4
00603 _Avrlibc_Def_Pgm_Far_4 (long_long, long long)
00604 _Avrlibc_Def_Pgm_Far_4 (unsigned_long_long, unsigned long long)
00605 #endif
00606 #if __SIZEOF_DOUBLE__ == 4
00607 _Avrlibc_Def_Pgm_Far_4 (double, double)
00608 #endif
00609 #if __SIZEOF_LONG_DOUBLE__ == 4
00610 _Avrlibc_Def_Pgm_Far_4 (long_double, long double)
00611 #endif
00612
00613 #if __SIZEOF_LONG_LONG__ == 8
00614 _Avrlibc_Def_Pgm_Far_8 (u64, uint64_t)
00615 _Avrlibc_Def_Pgm_Far_8 (i64, int64_t)
00616 _Avrlibc_Def_Pgm_Far_8 (long_long, long long)
00617 _Avrlibc_Def_Pgm_Far_8 (unsigned_long_long, unsigned long long)
00618 #endif
00619 #if __SIZEOF_DOUBLE__ == 8
00620 _Avrlibc_Def_Pgm_Far_8 (double, double)
00621 #endif
```

```
00622 #if __SIZEOF_LONG_DOUBLE__ == 8
00623 _Avrlibc_Def_Pgm_Far_8 (long_double, long double)
00624 #endif
00625
00626 #endif /* DOXYGEN */
00627
00628 #ifdef __cplusplus
00629 extern "C" {
00630 #endif
00631
00632 #if defined(__DOXYGEN__)
00633 /* No documentation for the deprecated stuff.  */
00634 #elif defined(__PROG_TYPES_COMPAT__)  /* !DOXYGEN */
00635
00636 typedef void prog_void __attribute__((__progmem__,__deprecated__("prog_void type is
      deprecated.")));
00637 typedef char prog_char __attribute__((__progmem__,__deprecated__("prog_char type is
      deprecated.")));
00638 typedef unsigned char prog_uchar __attribute__((__progmem__,__deprecated__("prog_uchar
      type is deprecated.")));
00639 typedef int8_t    prog_int8_t   __attribute__((__progmem__,__deprecated__("prog_int8_t
      type is deprecated.")));
00640 typedef uint8_t   prog_uint8_t  __attribute__((__progmem__,__deprecated__("prog_uint8_t
      type is deprecated.")));
00641 typedef int16_t   prog_int16_t  __attribute__((__progmem__,__deprecated__("prog_int16_t
      type is deprecated.")));
00642 typedef uint16_t  prog_uint16_t __attribute__((__progmem__,__deprecated__("prog_uint16_t
      type is deprecated.")));
00643 typedef int32_t   prog_int32_t  __attribute__((__progmem__,__deprecated__("prog_int32_t
      type is deprecated.")));
00644 typedef uint32_t  prog_uint32_t __attribute__((__progmem__,__deprecated__("prog_uint32_t
      type is deprecated.")));
00645 #if !__USING_MINT8
00646 typedef int64_t   prog_int64_t  __attribute__((__progmem__,__deprecated__("prog_int64_t
      type is deprecated.")));
00647 typedef uint64_t  prog_uint64_t __attribute__((__progmem__,__deprecated__("prog_uint64_t
      type is deprecated.")));
00648 #endif
00649
00650 #ifndef PGM_P
00651 #define PGM_P const prog_char *
00652 #endif
00653
00654 #ifndef PGM_VOID_P
00655 #define PGM_VOID_P const prog_void *
00656 #endif
00657
00658 #else /* !defined(__DOXYGEN__), !defined(__PROG_TYPES_COMPAT__) */
00659
00660 #ifndef PGM_P
00661 #define PGM_P const char *
00662 #endif
00663
00664 #ifndef PGM_VOID_P
00665 #define PGM_VOID_P const void *
00666 #endif
00667 #endif /* defined(__DOXYGEN__), defined(__PROG_TYPES_COMPAT__) */
00668
00669 /* Although in C, we can get away with just using __c, it does not work in
00670    C++. We need to use &__c[0] to avoid the compiler puking. Dave Hylands
00671    explained it thusly,
00672
00673      Let's suppose that we use PSTR("Test"). In this case, the type returned
00674      by __c is a prog_char[5] and not a prog_char *. While these are
00675      compatible, they aren't the same thing (especially in C++). The type
00676      returned by &__c[0] is a prog_char *, which explains why it works
00677      fine. */
00678
00679 #if defined(__DOXYGEN__)
00680
00681 /** \name Macros */
00682
00683 /*
```

```
00684  * The #define below is just a dummy that serves documentation
00685  * purposes only.
00686  */
00687 /** \ingroup avr_pgmspace
00688     \def PSTR(str)
00689
00690     Used to declare a static pointer to a string in program space. */
00691 # define PSTR(str) ({ static const PROGMEM char c[] = (str); &c[0]; })
00692 #else  /* !DOXYGEN */
00693 /* The real thing. */
00694 # define PSTR(s) (__extension__({static const char __c[] PROGMEM = (s); &__c[0];}))
00695 #endif /* DOXYGEN */
00696
00697 #if defined(__DOXYGEN__)
00698 /** \ingroup avr_pgmspace
00699     \def PSTR_FAR(str)
00700
00701     Used to define a string literal in far program space, and to return its
00702     address of type #uint_farptr_t.
00703     \since AVR-LibC v2.2 */
00704 # define PSTR_FAR(str) ({ static const PROGMEM_FAR char c[] = (str);
      pgm_get_far_address(c[0]); })
00705 #else  /* !DOXYGEN */
00706 /* The real thing. */
00707 # define PSTR_FAR(s) (__extension__({static const char __c[] PROGMEM_FAR = (s);
      pgm_get_far_address(__c[0]);}))
00708 #endif /* DOXYGEN */
00709
00710 #ifndef __DOXYGEN__
00711
00712 /* These are used down the line for pgm_read_byte[_near] etc. */
00713
00714 #if defined (__AVR_TINY__)
00715 /* Attribute __progmem__ on Reduced Tiny works different than for
00716    all the other devices:  When taking the address of a symbol that's
00717    attributed as progmem, then the compiler adds an offset of 0x4000
00718    to the value of the symbol.  This means that accessing data in
00719    progmem can be performed by vanilla C/C++ code.  This requires
00720    - GCC PR71948 - Make progmem work on Reduced Tiny (GCC v7 / 2016-08)  */
00721 #define __LPM(addr)         (* (const uint8_t*)(addr))
00722 #define __LPM_word(addr)    (* (const uint16_t*)(addr))
00723 #define __LPM_dword(addr)   (* (const uint32_t*)(addr))
00724 #  if __SIZEOF_LONG_LONG__ == 8
00725 #  define __LPM_qword(addr)   (* (const uint64_t*)(addr))
00726 #  endif
00727 #else
00728 #define __LPM(addr)                             \
00729   (__extension__({                              \
00730       uint16_t __addr16 = (uint16_t) (addr);    \
00731       uint8_t __result;                         \
00732       __LPM__1 (__result, __addr16);            \
00733       __result;                                 \
00734 }))
00735
00736 #define __LPM_word(addr)                        \
00737   (__extension__({                              \
00738       uint16_t __addr16 = (uint16_t) (addr);    \
00739       uint16_t __result;                        \
00740       __LPM__2 (__result, __addr16);            \
00741       __result;                                 \
00742 }))
00743
00744 #define __LPM_dword(addr)                       \
00745   (__extension__({                              \
00746       uint16_t __addr16 = (uint16_t) (addr);    \
00747       uint32_t __result;                        \
00748       __LPM__4 (__result, __addr16);            \
00749       __result;                                 \
00750 }))
00751
00752 #if __SIZEOF_LONG_LONG__ == 8
00753 #define __LPM_qword(addr)                       \
00754   (__extension__({                              \
```

```
00755       uint16_t __addr16 = (uint16_t) (addr);     \
00756       uint64_t __result;                         \
00757       __LPM__8 (__result, __addr16);             \
00758       __result;                                  \
00759 }))
00760 #endif
00761 #endif /* AVR_TINY */
00762
00763
00764 #define __ELPM(addr)                                      \
00765   (__extension__({                                        \
00766       uint_farptr_t __addr32 = (addr);                    \
00767       uint8_t __result;                                   \
00768       __ELPM__1 (__result, __addr32, uint8_t);            \
00769       __result;                                           \
00770 }))
00771
00772 #define __ELPM_word(addr)                                 \
00773   (__extension__({                                        \
00774       uint_farptr_t __addr32 = (addr);                    \
00775       uint16_t __result;                                  \
00776       __ELPM__2 (__result, __addr32, uint16_t);           \
00777       __result;                                           \
00778 }))
00779
00780 #define __ELPM_dword(addr)                                \
00781   (__extension__({                                        \
00782       uint_farptr_t __addr32 = (addr);                    \
00783       uint32_t __result;                                  \
00784       __ELPM__4 (__result, __addr32, uint32_t);           \
00785       __result;                                           \
00786 }))
00787
00788 #if __SIZEOF_LONG_LONG__ == 8
00789 #define __ELPM_qword(addr)                                \
00790   (__extension__({                                        \
00791       uint_farptr_t __addr32 = (addr);                    \
00792       uint64_t __result;                                  \
00793       __ELPM__8 (__result, __addr32, uint64_t);           \
00794       __result;                                           \
00795 }))
00796 #endif
00797
00798 #endif  /* !__DOXYGEN__ */
00799
00800 /** \name Macros reading from PROGMEM */
00801
00802 /** \ingroup avr_pgmspace
00803     \def pgm_read_byte_near(__addr)
00804     Read a byte from the program space with a 16-bit address. */
00805 #define pgm_read_byte_near(__addr) __LPM ((uint16_t)(__addr))
00806
00807 /** \ingroup avr_pgmspace
00808     \def pgm_read_word_near(__addr)
00809     Read a word from the program space with a 16-bit address. */
00810 #define pgm_read_word_near(__addr) __LPM_word ((uint16_t)(__addr))
00811
00812 /** \ingroup avr_pgmspace
00813     \def pgm_read_dword_near(__addr)
00814     Read a double word from the program space with a 16-bit address.  */
00815 #define pgm_read_dword_near(__addr) \
00816     __LPM_dword ((uint16_t)(__addr))
00817
00818 /** \ingroup avr_pgmspace
00819     \def pgm_read_qword_near(__addr)
00820     Read a quad-word from the program space with a 16-bit address.
00821     \since AVR-LibC v2.2  */
00822 #define pgm_read_qword_near(__addr) __LPM_qword ((uint16_t)(__addr))
00823
00824 /** \ingroup avr_pgmspace
00825     \def pgm_read_float_near (const float *address)
00826     Read a \c float from the program space with a 16-bit address.*/
00827 #define pgm_read_float_near(addr)  pgm_read_float (addr)
```

```
00828
00829 /** \ingroup avr_pgmspace
00830     \def pgm_read_ptr_near(__addr)
00831     Read a pointer from the program space with a 16-bit address. */
00832 #define pgm_read_ptr_near(__addr) \
00833     ((void*) __LPM_word ((uint16_t)(__addr)))
00834
00835 /** \name Macros reading from PROGMEM_FAR */
00836
00837 /** \ingroup avr_pgmspace
00838     \def pgm_read_byte_far(__addr)
00839     Read a byte from the program space with a 32-bit (far) address. */
00840 #define pgm_read_byte_far(__addr)  __ELPM (__addr)
00841
00842 /** \ingroup avr_pgmspace
00843     \def pgm_read_word_far(__addr)
00844     Read a word from the program space with a 32-bit (far) address. */
00845 #define pgm_read_word_far(__addr)  __ELPM_word (__addr)
00846
00847 /** \ingroup avr_pgmspace
00848     \def pgm_read_dword_far(__addr)
00849     Read a double word from the program space with a 32-bit (far) address. */
00850 #define pgm_read_dword_far(__addr) __ELPM_dword (__addr)
00851
00852 /** \ingroup avr_pgmspace
00853     \def pgm_read_qword_far(__addr)
00854     Read a quad-word from the program space with a 32-bit (far) address.
00855     \since AVR-LibC v2.2 */
00856 #define pgm_read_qword_far(__addr) __ELPM_qword (__addr)
00857
00858 /** \ingroup avr_pgmspace
00859     \def pgm_read_ptr_far(__addr)
00860     Read a pointer from the program space with a 32-bit (far) address. */
00861 #define pgm_read_ptr_far(__addr) ((void*) __ELPM_word (__addr))
00862
00863 /** \name Macros reading from PROGMEM */
00864
00865 /** \ingroup avr_pgmspace
00866     \def pgm_read_byte(__addr)
00867     Read a byte from the program space with a 16-bit address. */
00868 #define pgm_read_byte(__addr)    pgm_read_byte_near(__addr)
00869
00870 /** \ingroup avr_pgmspace
00871     \def pgm_read_word(__addr)
00872     Read a word from the program space with a 16-bit address. */
00873 #define pgm_read_word(__addr)    pgm_read_word_near(__addr)
00874
00875 /** \ingroup avr_pgmspace
00876     \def pgm_read_dword(__addr)
00877     Read a double word from the program space with a 16-bit address. */
00878 #define pgm_read_dword(__addr)   pgm_read_dword_near(__addr)
00879
00880 /** \ingroup avr_pgmspace
00881     \def pgm_read_qword(__addr)
00882     Read a quad-word from the program space with a 16-bit address.
00883     \since AVR-LibC v2.2 */
00884 #define pgm_read_qword(__addr)   pgm_read_qword_near(__addr)
00885
00886 /** \ingroup avr_pgmspace
00887     \def pgm_read_ptr(__addr)
00888     Read a pointer from the program space with a 16-bit address. */
00889 #define pgm_read_ptr(__addr)     pgm_read_ptr_near(__addr)
00890
00891 /** \name Macros */
00892
00893 /** \ingroup avr_pgmspace
00894     \def pgm_get_far_address(var)
00895
00896    This macro evaluates to a ::uint_farptr_t 32-bit "far" pointer (only
00897    24 bits used) to data even beyond the 64 KiB limit for the 16-bit ordinary
00898    pointer.  It is similar to the '&' operator, with some limitations.
00899    Example:
00900    \code
```

```
00901    #include <avr/pgmspace.h>
00902
00903    // Section .progmemx.data is located after all the code sections.
00904    PROGMEM_FAR
00905    const int data[] = { 2, 3, 5, 7, 9, 11 };
00906
00907    int get_data (uint8_t idx)
00908    {
00909        uint_farptr_t pdata = pgm_get_far_address (data[0]);
00910        return pgm_read_int_far (pdata + idx * sizeof(int));
00911    }
00912    \endcode
00913
00914    Comments:
00915
00916    - The overhead is minimal and it's mainly due to the 32-bit size operation.
00917
00918    - 24 bit sizes guarantees the code compatibility for use in future devices.
00919
00920    - \p var has to be resolved at link-time as an existing symbol,
00921      i.e. a simple variable name, an array name, or an array or structure
00922      element provided the offset is known at compile-time, and \p var is
00923      located in static storage, etc.
00924 */
00925 #ifdef __DOXYGEN__
00926 #define pgm_get_far_address(var)
00927 #else
00928 #ifndef __AVR_TINY__
00929 #define pgm_get_far_address(var)                        \
00930 (__extension__({                                        \
00931     uint_farptr_t __tmp;                                \
00932                                                         \
00933     __asm__ (                                           \
00934         "ldi    %A0, lo8(%1)"           "\n\t"          \
00935         "ldi    %B0, hi8(%1)"           "\n\t"          \
00936         "ldi    %C0, hh8(%1)"           "\n\t"          \
00937         "clr    %D0"                                    \
00938         :   "=d" (__tmp)                                \
00939         :   "i"  (&(var))                               \
00940     );                                                  \
00941     __tmp;                                              \
00942 }))
00943 #else
00944 /* The working of the pgm_read_far() functions and macros is such
00945    that they decay to pgm_read() for devices without ELPM.
00946    Since GCC v7 PR71948, the compiler adds an offset of 0x4000 on
00947    Reduced Tiny when it takes the address of an object in PROGMEM,
00948    which means we have to add 0x4000 here, too.  Notice that
00949    PROGMEM_FAR is just a section attribute without __progmem__, and
00950    therefore the compiler doesn't add 0x4000.  */
00951 #define pgm_get_far_address(var)                        \
00952 (__extension__({                                        \
00953     uint_farptr_t __tmp;                                \
00954                                                         \
00955     __asm__ (                                           \
00956         "ldi    %A0, lo8(0x4000+(%1))"  "\n\t"          \
00957         "ldi    %B0, hi8(0x4000+(%1))"  "\n\t"          \
00958         "ldi    %C0, hh8(0x4000+(%1))"  "\n\t"          \
00959         "clr    %D0"                                    \
00960         :   "=d" (__tmp)                                \
00961         :   "i"  (&(var))                               \
00962     );                                                  \
00963     __tmp;                                              \
00964 }))
00965 #endif /* AVR TINY */
00966 #endif /* DOXYGEN */
00967
00968 /** \name Functions with a PROGMEM argument
00969     Similar to the functions from <string.h>, but with one string
00970     argument located in the lower 64 KiB segment of program memory.<br>
00971     For similar functions with address-space #__flash,
00972     see \ref avr_flash "<avr/flash.h>" */
00973
```

```
00974 #ifdef __DOXYGEN__
00975 /** \ingroup avr_pgmspace
00976     \fn size_t strlen_P(const char *src)
00977
00978     The strlen_P() function is similar to strlen(), except that src is a
00979     pointer to a string in program space.
00980
00981     \returns The strlen_P() function returns the number of characters in src.
00982 */
00983 static inline size_t strlen_P(const char * s);
00984 #endif
00985
00986 /** \ingroup avr_pgmspace
00987     \fn const void * memchr_P(const void *s, int val, size_t len)
00988     \brief Scan flash memory for a character.
00989
00990     The memchr_P() function scans the first \p len bytes of the flash
00991     memory area pointed to by \p s for the character \p val.  The first
00992     byte to match \p val (interpreted as an unsigned character) stops
00993     the operation. \p s is located in the lower 64 KiB of program memory.
00994
00995     \return The memchr_P() function returns a pointer to the matching
00996     byte or \c NULL if the character does not occur in the given memory
00997     area.   */
00998 extern const void * memchr_P(const void *, int __val, size_t __len) __ATTR_CONST__;
00999
01000 /** \ingroup avr_pgmspace
01001     \fn int memcmp_P(const void *s1, const void *s2, size_t len)
01002     \brief Compare memory areas
01003
01004     The memcmp_P() function compares the first \p len bytes of the memory
01005     areas \p s1 and flash \p s2. The comparision is performed using unsigned
01006     char operations. \p s2 is located in the lower 64 KiB of program memory.
01007
01008     \returns The memcmp_P() function returns an integer less than, equal
01009     to, or greater than zero if the first \p len bytes of \p s1 is found,
01010     respectively, to be less than, to match, or be greater than the first
01011     \p len bytes of \p s2.  */
01012 extern int memcmp_P(const void *, const void *, size_t) __ATTR_PURE__;
01013
01014 /** \ingroup avr_pgmspace
01015     \fn void *memccpy_P (void *dest, const void *src, int val, size_t len)
01016
01017     This function is similar to memccpy() except that \p src points
01018     to a string in the lower 64 KiB of program space.   */
01019 extern void *memccpy_P(void *, const void *, int __val, size_t);
01020
01021 /** \ingroup avr_pgmspace
01022     \fn void *memcpy_P(void *dest, const void *src, size_t n)
01023
01024     The memcpy_P() function is similar to memcpy(), except the src string
01025     resides in the lower 64 KiB of program space.
01026
01027     \returns The memcpy_P() function returns a pointer to dest.  */
01028 extern void *memcpy_P(void *, const void *, size_t);
01029
01030 /** \ingroup avr_pgmspace
01031     \fn void *memmem_P(const void *s1, size_t len1, const void *s2, size_t len2)
01032
01033     The memmem_P() function is similar to memmem() except that \p s2 is
01034     pointer to a string in the lower 64 KiB of program space.   */
01035 extern void *memmem_P(const void *, size_t, const void *, size_t) __ATTR_PURE__;
01036
01037 /** \ingroup avr_pgmspace
01038     \fn const void *memrchr_P(const void *src, int val, size_t len)
01039
01040     The memrchr_P() function is like the memchr_P() function, except
01041     that it searches backwards from the end of the \p len bytes pointed
01042     to by \p src instead of forwards from the front. (Glibc, GNU extension.)
01043     \p src is located in the lower 64 KiB of program memory.
01044
01045     \return The memrchr_P() function returns a pointer to the matching
01046     byte or \c NULL if the character does not occur in the given memory
```

```
01047     area.   */
01048 extern const void * memrchr_P(const void *, int __val, size_t __len) __ATTR_CONST__;
01049
01050 /** \ingroup avr_pgmspace
01051     \fn char *strcat_P(char *dest, const char *src)
01052
01053     The strcat_P() function is similar to strcat() except that the \e src
01054     string must be located in the lower 64 KiB of program space (flash).
01055
01056     \returns The strcat() function returns a pointer to the resulting string
01057     \e dest. */
01058 extern char *strcat_P(char *, const char *);
01059
01060 /** \ingroup avr_pgmspace
01061     \fn const char *strchr_P(const char *s, int val)
01062     \brief Locate character in program space string.
01063
01064     The strchr_P() function locates the first occurrence of \p val
01065     (converted to a char) in the string pointed to by \p s in the lower
01066     64 KiB of program space. The terminating null character is considered
01067     to be part of the string.
01068
01069     The strchr_P() function is similar to strchr() except that \p s
01070     points to a string in the lower 64 KiB of program space.
01071
01072     \returns The strchr_P() function returns a pointer to the matched
01073     character or \c NULL if the character is not found. */
01074 #ifdef __DOXYGEN__
01075 extern const char * strchr_P(const char *, int __val) __ATTR_CONST__;
01076 #else
01077 /* strchr_P is used in variants of printf, so we model its GPR footprint.  */
01078 extern __ATTR_ALWAYS_INLINE__ __ATTR_GNU_INLINE__
01079 const char * strchr_P(const char *__hay, int __val)
01080 {
01081   register const char *__r24 __asm("24") = __hay;
01082   register int __r22 __asm("22") = __val;
01083   __asm ("%~call strchr_P"
01084          : "+r" (__r24) : "r" (__r22) : "30", "31");
01085   return __r24;
01086 }
01087 #endif /* DOXYGEN */
01088
01089 /** \ingroup avr_pgmspace
01090     \fn const char *strchrnul_P(const char *s, int c)
01091
01092     The strchrnul_P() function is like strchr_P() except that if \p c is
01093     not found in \p s, then it returns a pointer to the null byte at the
01094     end of \p s, rather than \c NULL. (Glibc, GNU extension.)
01095
01096     \return The strchrnul_P() function returns a pointer to the matched
01097     character, or a pointer to the null byte at the end of \p s (i.e.,
01098     \c s+strlen(s)) if the character is not found.  */
01099 extern const char * strchrnul_P(const char *, int __val) __ATTR_CONST__;
01100
01101 /** \ingroup avr_pgmspace
01102     \fn int strcmp_P(const char *s1, const char *s2)
01103
01104     The strcmp_P() function is similar to strcmp() except that \p s2 is
01105     pointer to a string in the lower 64 KiB of program space.
01106
01107     \returns The strcmp_P() function returns an integer less than, equal
01108     to, or greater than zero if \p s1 is found, respectively, to be less
01109     than, to match, or be greater than \p s2. A consequence of the
01110     ordering used by strcmp_P() is that if \p s1 is an initial substring
01111     of \p s2, then \p s1 is considered to be "less than" \p s2. */
01112 extern int strcmp_P(const char *, const char *) __ATTR_PURE__;
01113
01114 /** \ingroup avr_pgmspace
01115     \fn char *strcpy_P(char *dest, const char *src)
01116
01117     The strcpy_P() function is similar to strcpy() except that \p src
01118     points to a string in the lower 64 KiB of program space.
01119
```

```
01120     \returns The strcpy_P() function returns a pointer to the destination
01121     string dest. */
01122 #ifdef __DOXYGEN__
01123 extern inline char *strcpy_P(char *, const char *);
01124 #endif
01125
01126 /** \ingroup avr_pgmspace
01127     \fn char *stpcpy_P(char *dest, const char *src)
01128
01129     The stpcpy_P() function is similar to stpcpy() except that \p src
01130     points to a string in the lower 64 KiB of program space.
01131
01132     \returns The stpcpy_P() function returns a pointer to the
01133     terminating '\\0' character of destination string \p dest.
01134
01135     \since AVR-LibC 2.3   */
01136 extern char *stpcpy_P(char *, const char *);
01137
01138 /** \ingroup avr_pgmspace
01139     \fn int strcasecmp_P(const char *s1, const char *s2)
01140     \brief Compare two strings ignoring case.
01141
01142     The strcasecmp_P() function compares the two strings \p s1 and \p s2,
01143     ignoring the case of the characters.
01144
01145     \param s1 A pointer to a string in the device's SRAM.
01146     \param s2 A pointer to a string in the lower 64 KiB of the device's Flash.
01147
01148     \returns The strcasecmp_P() function returns an integer less than,
01149     equal to, or greater than zero if \p s1 is found, respectively, to
01150     be less than, to match, or be greater than \p s2. A consequence of
01151     the ordering used by strcasecmp_P() is that if \p s1 is an initial
01152     substring of \p s2, then \p s1 is considered to be "less than" \p s2. */
01153 extern int strcasecmp_P(const char *, const char *) __ATTR_PURE__;
01154
01155 /** \ingroup avr_pgmspace
01156     \fn char *strcasestr_P(const char *s1, const char *s2)
01157
01158     This function is similar to strcasestr() except that \p s2 points
01159     to a string in the lower 64 KiB of program space.   */
01160 extern char *strcasestr_P(const char *, const char *) __ATTR_PURE__;
01161
01162 /** \ingroup avr_pgmspace
01163     \fn size_t strcspn_P(const char *s, const char *reject)
01164
01165     The strcspn_P() function calculates the length of the initial segment
01166     of \p s which consists entirely of characters not in \p reject. This
01167     function is similar to strcspn() except that \p reject points
01168     to a string in the lower 64 KiB of program space.
01169
01170     \return The strcspn_P() function returns the number of characters in
01171     the initial segment of \p s which are not in the string \p reject.
01172     The terminating zero is not considered as a part of string. */
01173 extern size_t strcspn_P(const char *__s, const char * __reject) __ATTR_PURE__;
01174
01175 /** \ingroup avr_pgmspace
01176     \fn size_t strlcat_P(char *dst, const char *src, size_t siz)
01177     \brief Concatenate two strings.
01178
01179     The strlcat_P() function is similar to strlcat(), except that the \p src
01180     string must be located in the lower 64 KiB of program space (flash).
01181
01182     Appends \p src to string \p dst of size \p siz (unlike strncat(),
01183     \p siz is the full size of \p dst, not space left).  At most \p siz-1
01184     characters will be copied.  Always NULL terminates (unless \p siz <=
01185     \p strlen(dst)).
01186
01187     \returns The strlcat_P() function returns strlen(src) + MIN(siz,
01188     strlen(initial dst)).  If retval >= siz, truncation occurred.   */
01189 extern size_t strlcat_P (char *, const char *, size_t );
01190
01191 /** \ingroup avr_pgmspace
01192     \fn size_t strlcpy_P(char *dst, const char *src, size_t siz)
```

```
01193     \brief Copy a string from progmem to RAM.
01194
01195     Copy \p src to string \p dst of size \p siz.  At most \p siz-1
01196     characters will be copied.  Always NULL terminates (unless \p siz == 0).
01197     The strlcpy_P() function is similar to strlcpy() except that
01198     \p src points to a string in the lower 64 KiB of program memory.
01199
01200     \returns The strlcpy_P() function returns strlen(src). If
01201     retval >= siz, truncation occurred.  */
01202 extern size_t strlcpy_P (char *, const char *, size_t );
01203
01204 /** \ingroup avr_pgmspace
01205     \fn size_t strnlen_P(const char *src, size_t len)
01206     \brief Determine the length of a fixed-size string.
01207
01208     The strnlen_P() function is similar to strnlen(), except that \c src
01209     points to a string in the lower 64 KiB of program space.
01210
01211     \returns The strnlen_P function returns strlen_P(src), if that is less than
01212     \c len, or \c len if there is no '\\0' character among the first \c len
01213     characters pointed to by \c src. */
01214 extern size_t strnlen_P(const char *, size_t) __ATTR_CONST__; /* program memory can't
      change */
01215
01216 /** \ingroup avr_pgmspace
01217     \fn int strncmp_P(const char *s1, const char *s2, size_t n)
01218
01219     The strncmp_P() function is similar to strcmp_P() except it only compares
01220     the first (at most) n characters of s1 and s2.
01221     \p s2 is located in the lower 64 KiB of program memory.
01222
01223     \returns The strncmp_P() function returns an integer less than, equal to,
01224     or greater than zero if s1 (or the first n bytes thereof) is found,
01225     respectively, to be less than, to match, or be greater than s2.  */
01226 extern int strncmp_P(const char *, const char *, size_t) __ATTR_PURE__;
01227
01228 /** \ingroup avr_pgmspace
01229     \fn int strncasecmp_P(const char *s1, const char *s2, size_t n)
01230     \brief Compare two strings ignoring case.
01231
01232     The strncasecmp_P() function is similar to strcasecmp_P(), except it
01233     only compares the first \p n characters of \p s1.
01234
01235     \param s1 A pointer to a string in the device's SRAM.
01236     \param s2 A pointer to a string in the thwer 64 KiB of the device's Flash.
01237     \param n The maximum number of bytes to compare.
01238
01239     \returns The strncasecmp_P() function returns an integer less than,
01240     equal to, or greater than zero if \p s1 (or the first \p n bytes
01241     thereof) is found, respectively, to be less than, to match, or be
01242     greater than \p s2. A consequence of the ordering used by
01243     strncasecmp_P() is that if \p s1 is an initial substring of \p s2,
01244     then \p s1 is considered to be "less than" \p s2.  */
01245 extern int strncasecmp_P(const char *, const char *, size_t) __ATTR_PURE__;
01246
01247 /** \ingroup avr_pgmspace
01248     \fn char *strncat_P(char *dest, const char *src, size_t len)
01249     \brief Concatenate two strings.
01250
01251     The strncat_P() function is similar to strncat(), except that the \p src
01252     string must be located in the lower 64 KiB of program space (flash).
01253
01254     \returns The strncat_P() function returns a pointer to the resulting string
01255     \p dest.  */
01256 extern char *strncat_P(char *, const char *, size_t);
01257
01258 /** \ingroup avr_pgmspace
01259     \fn char *strncpy_P(char *dest, const char *src, size_t n)
01260
01261     The strncpy_P() function is similar to strcpy_P() except that not more
01262     than n bytes of src are copied.  Thus, if there is no null byte among the
01263     first n bytes of src, the result will not be null-terminated.
01264     \p src is located in the lower 64 KiB of program memory.
```

```
01265
01266     In the case where the length of src is less than that of n, the remainder
01267     of dest will be padded with nulls.
01268
01269     \returns The strncpy_P() function returns a pointer to the destination
01270     string \p dest.  */
01271 extern char *strncpy_P(char *, const char *, size_t);
01272
01273 /** \ingroup avr_pgmspace
01274     \fn char *strpbrk_P(const char *s, const char *accept)
01275
01276     The strpbrk_P() function locates the first occurrence in the string
01277     \p s of any of the characters in the flash string \p accept. This
01278     function is similar to strpbrk() except that \p accept points
01279     to a string in the lower 64 KiB of program space.
01280
01281     \return  The strpbrk_P() function returns a pointer to the character
01282     in \p s that matches one of the characters in \p accept, or \c NULL
01283     if no such character is found. The terminating zero is not considered
01284     as a part of string: if one or both args are empty, the result will
01285     \c NULL. */
01286 extern char *strpbrk_P(const char *__s, const char * __accept) __ATTR_PURE__;
01287
01288 /** \ingroup avr_pgmspace
01289     \fn const char *strrchr_P(const char *s, int val)
01290     \brief Locate character in string.
01291
01292     The strrchr_P() function returns a pointer to the last occurrence of
01293     the character \p val in the string \p s.
01294     \p s is located in the lower 64 KiB of program memory.
01295
01296     \return The strrchr_P() function returns a pointer to the matched
01297     character or \c NULL if the character is not found. */
01298 extern const char * strrchr_P(const char *, int __val) __ATTR_CONST__;
01299
01300 /** \ingroup avr_pgmspace
01301     \fn char *strsep_P(char **sp, const char *delim)
01302     \brief Parse a string into tokens.
01303
01304     The strsep_P() function locates, in the string referenced by \p *sp,
01305     the first occurrence of any character in the string \p delim (or the
01306     terminating '\\0' character) and replaces it with a '\\0'.  The
01307     location of the next character after the delimiter character (or \c
01308     NULL, if the end of the string was reached) is stored in \p *sp. An
01309     "empty" field, i.e. one caused by two adjacent delimiter
01310     characters, can be detected by comparing the location referenced by
01311     the pointer returned in \p *sp to '\\0'. This function is similar to
01312     strsep() except that \p delim points to a string in the lower
01313     64 KiB of program space.
01314
01315     \return The strsep_P() function returns a pointer to the original
01316     value of \p *sp. If \p *sp is initially \c NULL, strsep_P() returns
01317     \c NULL. */
01318 extern char *strsep_P(char **__sp, const char * __delim);
01319
01320 /** \ingroup avr_pgmspace
01321     \fn size_t strspn_P(const char *s, const char *accept)
01322
01323     The strspn_P() function calculates the length of the initial segment
01324     of \p s which consists entirely of characters in \p accept. This
01325     function is similar to strspn() except that \p accept points
01326     to a string in the lower 64 KiB of program space.
01327
01328     \return  The strspn_P() function returns the number of characters in
01329     the initial segment of \p s which consist only of characters from \p
01330     accept. The terminating zero is not considered as a part of string. */
01331 extern size_t strspn_P(const char *__s, const char * __accept) __ATTR_PURE__;
01332
01333 /** \ingroup avr_pgmspace
01334     \fn char *strstr_P(const char *s1, const char *s2)
01335     \brief Locate a substring.
01336
01337     The strstr_P() function finds the first occurrence of the substring
```

```
01338     \p s2 in the string \p s1.  The terminating '\\0' characters are not
01339     compared. The strstr_P() function is similar to strstr() except that
01340     \p s2 points to a string in the lower 64 KiB of program space.
01341
01342     \returns The strstr_P() function returns a pointer to the beginning
01343     of the substring, or NULL if the substring is not found. If \p s2
01344     points to a string of zero length, the function returns \p s1. */
01345 extern char *strstr_P(const char *, const char *) __ATTR_PURE__;
01346
01347 /** \ingroup avr_pgmspace
01348     \fn char *strtok_P(char *s, const char * delim)
01349     \brief Parses the string into tokens.
01350
01351     strtok_P() parses the string \p s into tokens. The first call to
01352     strtok_P() should have \p s as its first argument. Subsequent calls
01353     should have the first argument set to NULL. If a token ends with a
01354     delimiter, this delimiting character is overwritten with a '\\0' and a
01355     pointer to the next character is saved for the next call to strtok_P().
01356     The delimiter string \p delim may be different for each call.
01357
01358     The strtok_P() function is similar to strtok() except that \p delim
01359     points to a string in the lower 64 KiB of program space.
01360
01361     \returns The strtok_P() function returns a pointer to the next token or
01362     NULL when no more tokens are found.
01363
01364     \note strtok_P() is NOT reentrant. For a reentrant version of this
01365     function see strtok_rP().
01366  */
01367 extern char *strtok_P(char *__s, const char * __delim);
01368
01369 /** \ingroup avr_pgmspace
01370     \fn char *strtok_rP (char *string, const char *delim, char **last)
01371     \brief Parses string into tokens.
01372
01373     The strtok_rP() function parses \p string into tokens. The first call to
01374     strtok_rP() should have string as its first argument. Subsequent calls
01375     should have the first argument set to NULL. If a token ends with a
01376     delimiter, this delimiting character is overwritten with a '\\0' and a
01377     pointer to the next character is saved for the next call to strtok_rP().
01378     The delimiter string \p delim may be different for each call. \p last is
01379     a user allocated char* pointer. It must be the same while parsing the
01380     same string. strtok_rP() is a reentrant version of strtok_P().
01381
01382     The strtok_rP() function is similar to strtok_r() except that \p delim
01383     points to a string in the lower 64 KiB of program space.
01384
01385     \returns The strtok_rP() function returns a pointer to the next token or
01386     NULL when no more tokens are found. */
01387 extern char *strtok_rP(char *__s, const char * __delim, char **__last);
01388
01389 /** \name Functions with a PROGMEM_FAR argument
01390     Similar to the functions from <string.h>, but with one
01391     string argument located in program memory.<br>
01392     For similar functions with address-space #__flashx,
01393     see \ref avr_flash "<avr/flash.h>" */
01394
01395 /** \ingroup avr_pgmspace
01396     \fn size_t strlen_PF(uint_farptr_t s)
01397     \brief Obtain the length of a string
01398
01399     The strlen_PF() function is similar to strlen(), except that \e s is a
01400     far pointer to a string in program space.
01401
01402     \param s A far pointer to the string in flash
01403
01404     \returns The strlen_PF() function returns the number of characters in
01405     \e s. */
01406 extern size_t strlen_PF(uint_farptr_t src) __ATTR_CONST__; /* program memory can't
      change */
01407
01408 /** \ingroup avr_pgmspace
01409     \fn size_t strnlen_PF(uint_farptr_t s, size_t len)
```

```
01410     \brief Determine the length of a fixed-size string
01411
01412     The strnlen_PF() function is similar to strnlen(), except that \e s is a
01413     far pointer to a string in program space.
01414
01415     \param s A far pointer to the string in Flash
01416     \param len The maximum number of length to return
01417
01418     \returns The strnlen_PF function returns strlen_PF(\e s), if that is less
01419     than \e len, or \e len if there is no '\\0' character among the first \e
01420     len characters pointed to by \e s. */
01421 extern size_t strnlen_PF(uint_farptr_t src, size_t len) __ATTR_CONST__; /* program
      memory can't change */
01422
01423 /** \ingroup avr_pgmspace
01424     \fn void *memcpy_PF(void *dest, uint_farptr_t src, size_t n)
01425     \brief Copy a memory block from flash to SRAM
01426
01427     The memcpy_PF() function is similar to memcpy(), except the data
01428     is copied from the program space and is addressed using a far pointer.
01429
01430     \param dest A pointer to the destination buffer
01431     \param src A far pointer to the origin of data in flash memory
01432     \param n The number of bytes to be copied
01433
01434     \returns The memcpy_PF() function returns a pointer to \e dst. */
01435 extern void *memcpy_PF(void *dest, uint_farptr_t src, size_t len);
01436
01437 /** \ingroup avr_pgmspace
01438     \fn char *strcpy_PF(char *dst, uint_farptr_t src)
01439     \brief Duplicate a string
01440
01441     The strcpy_PF() function is similar to strcpy() except that \e src is a far
01442     pointer to a string in program space.
01443
01444     \param dst A pointer to the destination string in SRAM
01445     \param src A far pointer to the source string in Flash
01446
01447     \returns The strcpy_PF() function returns a pointer to the destination
01448     string \e dst. */
01449 extern char *strcpy_PF(char *dest, uint_farptr_t src);
01450
01451 /** \ingroup avr_pgmspace
01452     \fn char *stpcpy_PF(char *dst, uint_farptr_t src)
01453     \brief Duplicate a string
01454
01455     The stpcpy_PF() function is similar to stpcpy() except that \e src
01456     is a far pointer to a string in program space.
01457
01458     \param dst A pointer to the destination string in SRAM
01459     \param src A far pointer to the source string in Flash
01460
01461     \returns The stpcpy_PF() function returns a pointer to the
01462     terminating '\\0' character of the destination string \e dst.
01463
01464     \since AVR-LibC 2.3  */
01465 extern char *stpcpy_PF(char *dest, uint_farptr_t src);
01466
01467 /** \ingroup avr_pgmspace
01468     \fn char *strncpy_PF(char *dst, uint_farptr_t src, size_t n)
01469     \brief Duplicate a string until a limited length
01470
01471     The strncpy_PF() function is similar to strcpy_PF() except that not more
01472     than \e n bytes of \e src are copied.  Thus, if there is no null byte among
01473     the first \e n bytes of \e src, the result will not be null-terminated.
01474
01475     In the case where the length of \e src is less than that of \e n, the
01476     remainder of \e dst will be padded with nulls.
01477
01478     \param dst A pointer to the destination string in SRAM
01479     \param src A far pointer to the source string in Flash
01480     \param n The maximum number of bytes to copy
01481
```

```
01482     \returns The strncpy_PF() function returns a pointer to the destination
01483     string \e dst. */
01484 extern char *strncpy_PF(char *dest, uint_farptr_t src, size_t len);
01485
01486 /** \ingroup avr_pgmspace
01487     \fn char *strcat_PF(char *dst, uint_farptr_t src)
01488     \brief Concatenates two strings
01489
01490     The strcat_PF() function is similar to strcat() except that the \e src
01491     string must be located in program space (flash) and is addressed using
01492     a far pointer
01493
01494     \param dst A pointer to the destination string in SRAM
01495     \param src A far pointer to the string to be appended in Flash
01496
01497     \returns The strcat_PF() function returns a pointer to the resulting
01498     string \e dst. */
01499 extern char *strcat_PF(char *dest, uint_farptr_t src);
01500
01501 /** \ingroup avr_pgmspace
01502     \fn size_t strlcat_PF(char *dst, uint_farptr_t src, size_t n)
01503     \brief Concatenate two strings
01504
01505     The strlcat_PF() function is similar to strlcat(), except that the \e src
01506     string must be located in program space (flash) and is addressed using
01507     a far pointer.
01508
01509     Appends src to string dst of size \e n (unlike strncat(), \e n is the
01510     full size of \e dst, not space left).  At most \e n-1 characters
01511     will be copied.  Always NULL terminates (unless \e n <= strlen(\e dst)).
01512
01513     \param dst A pointer to the destination string in SRAM
01514     \param src A far pointer to the source string in Flash
01515     \param n The total number of bytes allocated to the destination string
01516
01517     \returns The strlcat_PF() function returns strlen(\e src) + MIN(\e n,
01518     strlen(initial \e dst)).  If retval >= \e n, truncation occurred. */
01519 extern size_t strlcat_PF(char *dst, uint_farptr_t src, size_t siz);
01520
01521 /** \ingroup avr_pgmspace
01522     \fn char *strncat_PF(char *dst, uint_farptr_t src, size_t n)
01523     \brief Concatenate two strings
01524
01525     The strncat_PF() function is similar to strncat(), except that the \e src
01526     string must be located in program space (flash) and is addressed using a
01527     far pointer.
01528
01529     \param dst A pointer to the destination string in SRAM
01530     \param src A far pointer to the source string in Flash
01531     \param n The maximum number of bytes to append
01532
01533     \returns The strncat_PF() function returns a pointer to the resulting
01534     string \e dst. */
01535 extern char *strncat_PF(char *dest, uint_farptr_t src, size_t len);
01536
01537 /** \ingroup avr_pgmspace
01538     \fn int strcmp_PF(const char *s1, uint_farptr_t s2)
01539     \brief Compares two strings
01540
01541     The strcmp_PF() function is similar to strcmp() except that \e s2 is a far
01542     pointer to a string in program space.
01543
01544     \param s1 A pointer to the first string in SRAM
01545     \param s2 A far pointer to the second string in Flash
01546
01547     \returns The strcmp_PF() function returns an integer less than, equal to,
01548     or greater than zero if \e s1 is found, respectively, to be less than, to
01549     match, or be greater than \e s2. */
01550 extern int strcmp_PF(const char *s1, uint_farptr_t s2) __ATTR_PURE__;
01551
01552 /** \ingroup avr_pgmspace
01553     \fn int strncmp_PF(const char *s1, uint_farptr_t s2, size_t n)
01554     \brief Compare two strings with limited length
```

```
01555
01556     The strncmp_PF() function is similar to strcmp_PF() except it only
01557     compares the first (at most) \e n characters of \e s1 and \e s2.
01558
01559     \param s1 A pointer to the first string in SRAM
01560     \param s2 A far pointer to the second string in Flash
01561     \param n The maximum number of bytes to compare
01562
01563     \returns The strncmp_PF() function returns an integer less than, equal
01564     to, or greater than zero if \e s1 (or the first \e n bytes thereof) is found,
01565     respectively, to be less than, to match, or be greater than \e s2. */
01566 extern int strncmp_PF(const char *s1, uint_farptr_t s2, size_t n) __ATTR_PURE__;
01567
01568 /** \ingroup avr_pgmspace
01569     \fn int strcasecmp_PF(const char *s1, uint_farptr_t s2)
01570     \brief Compare two strings ignoring case
01571
01572     The strcasecmp_PF() function compares the two strings \e s1 and \e s2, ignoring
01573     the case of the characters.
01574
01575     \param s1 A pointer to the first string in SRAM
01576     \param s2 A far pointer to the second string in Flash
01577
01578     \returns The strcasecmp_PF() function returns an integer less than, equal
01579     to, or greater than zero if \e s1 is found, respectively, to be less than, to
01580     match, or be greater than \e s2. */
01581 extern int strcasecmp_PF(const char *s1, uint_farptr_t s2) __ATTR_PURE__;
01582
01583 /** \ingroup avr_pgmspace
01584     \fn int strncasecmp_PF(const char *s1, uint_farptr_t s2, size_t n)
01585     \brief Compare two strings ignoring case
01586
01587     The strncasecmp_PF() function is similar to strcasecmp_PF(), except it
01588     only compares the first \e n characters of \e s1 and the string in flash is
01589     addressed using a far pointer.
01590
01591     \param s1 A pointer to a string in SRAM
01592     \param s2 A far pointer to a string in Flash
01593     \param n The maximum number of bytes to compare
01594
01595     \returns The strncasecmp_PF() function returns an integer less than, equal
01596     to, or greater than zero if \e s1 (or the first \e n bytes thereof) is found,
01597     respectively, to be less than, to match, or be greater than \e s2. */
01598 extern int strncasecmp_PF(const char *s1, uint_farptr_t s2, size_t n) __ATTR_PURE__;
01599
01600 /** \ingroup avr_pgmspace
01601     \fn uint_farptr_t strchr_PF(uint_farptr_t s, int val)
01602     \brief Locate character in far program space string.
01603
01604     The strchr_PF() function locates the first occurrence of \p val
01605     (converted to a char) in the string pointed to by \p s in far program
01606     space. The terminating null character is considered to be part of
01607     the string.
01608
01609     The strchr_PF() function is similar to strchr() except that \p s is
01610     a far pointer to a string in program space that's \e not \e required to be
01611     located in the lower 64 KiB block like it is the case  for strchr_P().
01612
01613     \returns The strchr_PF() function returns a far pointer to the matched
01614     character or \c 0 if the character is not found. */
01615 extern uint_farptr_t strchr_PF(uint_farptr_t, int __val) __ATTR_CONST__;
01616
01617 /** \ingroup avr_pgmspace
01618     \fn char *strstr_PF(const char *s1, uint_farptr_t s2)
01619     \brief Locate a substring.
01620
01621     The strstr_PF() function finds the first occurrence of the substring \c s2
01622     in the string \c s1.  The terminating '\\0' characters are not
01623     compared.
01624     The strstr_PF() function is similar to strstr() except that \c s2 is a
01625     far pointer to a string in program space.
01626
01627     \returns The strstr_PF() function returns a pointer to the beginning of the
```

```
01628     substring, or NULL if the substring is not found.
01629     If \c s2 points to a string of zero length, the function returns \c s1. */
01630 extern char *strstr_PF(const char *s1, uint_farptr_t s2);
01631
01632 /** \ingroup avr_pgmspace
01633     \fn size_t strlcpy_PF(char *dst, uint_farptr_t src, size_t siz)
01634     \brief Copy a string from progmem to RAM.
01635
01636     Copy src to string dst of size siz.  At most siz-1 characters will be
01637     copied. Always NULL terminates (unless siz == 0).
01638
01639     \returns The strlcpy_PF() function returns strlen(src). If retval >= siz,
01640     truncation occurred.  */
01641 extern size_t strlcpy_PF(char *dst, uint_farptr_t src, size_t siz);
01642
01643 /** \ingroup avr_pgmspace
01644     \fn int memcmp_PF(const void *s1, uint_farptr_t s2, size_t len)
01645     \brief Compare memory areas
01646
01647     The memcmp_PF() function compares the first \p len bytes of the memory
01648     areas \p s1 and flash \p s2. The comparision is performed using unsigned
01649     char operations. It is an equivalent of memcmp_P() function, except
01650     that it is capable working on all FLASH including the extended area
01651     above 64kB.
01652
01653     \returns The memcmp_PF() function returns an integer less than, equal
01654     to, or greater than zero if the first \p len bytes of \p s1 is found,
01655     respectively, to be less than, to match, or be greater than the first
01656     \p len bytes of \p s2.  */
01657 extern int memcmp_PF(const void *, uint_farptr_t, size_t) __ATTR_PURE__;
01658
01659 #ifdef __DOXYGEN__
01660 #else /* !DOXYGEN */
01661
01662 #ifdef __AVR_TINY__
01663 /* GCC PR71948: AVR_TINY may use open coded C/C++ to read from progmem.  */
01664 #include <string.h>
01665
01666 #define memcpy_P(x, y, z) memcpy(x, y, z)
01667 #define strlen_P(x) strlen(x)
01668 #define strcpy_P(x, y) strcpy(x, y)
01669 #define stpcpy_P(x, y) stpcpy(x, y)
01670
01671 #else
01672
01673 /* memcpy_P is common so we model its GPR footprint.  */
01674 extern __ATTR_ALWAYS_INLINE__ __ATTR_GNU_INLINE__
01675 void* memcpy_P(void *__x, const void *__z, size_t __s)
01676 {
01677   register size_t __r20 __asm("20") = __s;
01678   void *__ret = __x;
01679   __asm volatile ("%~call __memcpy_P" : "+x" (__x), "+z" (__z), "+r" (__r20)
01680                   :: "0", "memory");
01681   return __ret;
01682 }
01683
01684 /* strlen_P is common so we model its GPR footprint.  */
01685 extern __ATTR_ALWAYS_INLINE__ __ATTR_GNU_INLINE__
01686 size_t strlen_P(const char *__s)
01687 {
01688   if (__builtin_constant_p (__builtin_strlen (__s)))
01689     {
01690       return __builtin_strlen (__s);
01691     }
01692   else
01693     {
01694       register const char *__r24 __asm("24") = __s;
01695       register size_t __res __asm("24");
01696       __asm ("%~call strlen_P" : "=r" (__res) : "r" (__r24)
01697              : "0", "30", "31");
01698       return __res;
01699     }
01700 }
```

```
01701
01702 /* strcpy_P is common so we model its GPR footprint.  */
01703 extern __ATTR_ALWAYS_INLINE__ __ATTR_GNU_INLINE__
01704 char* strcpy_P(char *__x, const char *__z)
01705 {
01706   char *__ret = __x;
01707   __asm volatile ("%~call __strcpy_P"
01708                   : "+x" (__x), "+z" (__z) :: "0", "memory");
01709   return __ret;
01710 }
01711
01712 extern __ATTR_ALWAYS_INLINE__ __ATTR_GNU_INLINE__
01713 char* stpcpy_P(char *__x, const char *__z)
01714 {
01715   __asm volatile ("%~call __strcpy_P"
01716                   : "+x" (__x), "+z" (__z) :: "0", "memory");
01717   return __x - 1;
01718 }
01719
01720 /* strcmp_P is common so we model its GPR footprint.  */
01721 extern __ATTR_ALWAYS_INLINE__ __ATTR_GNU_INLINE__
01722 int strcmp_P(const char *__x, const char *__z)
01723 {
01724   register int __ret __asm("24");
01725   __asm ("%~call __strcmp_P"
01726          : "=r" (__ret), "+x" (__x), "+z" (__z) :: "memory");
01727   return __ret;
01728 }
01729
01730 #endif /* AVR_TINY */
01731
01732 #endif /* DOXYGEN */
01733
01734 #ifdef __cplusplus
01735 } // extern "C"
01736 #endif
01737
01738 #if defined(__cplusplus) && defined(__pgm_read_template__)
01739
01740 /* Caveat: When this file is found via -isystem <path>, then some older
01741    avr-g++ versions come up with
01742
01743        error: template with C linkage
01744
01745    because the target description did not define NO_IMPLICIT_EXTERN_C.  */
01746
01747 template<typename __T, size_t>
01748 struct __pgm_read_impl
01749 {
01750   // A default implementation for T's with a size not in { 1, 2, 3, 4, 8 }.
01751   // While this works, the performance is absolute scrap because GCC does
01752   // not handle objects well that don't fit in a register (i.e. avr-gcc
01753   // has no respective machine_mode).
01754   __T operator() (const __T *__addr) const
01755   {
01756     __T __res;
01757     memcpy_P (&__res, __addr, sizeof(__T));
01758     return __res;
01759   }
01760 };
01761
01762 template<typename __T>
01763 struct __pgm_read_impl<__T, 1>
01764 {
01765   __T operator() (const __T *__addr) const
01766   {
01767     __T __res; __LPM__1 (__res, __addr); return __res;
01768   }
01769 };
01770
01771 template<typename __T>
01772 struct __pgm_read_impl<__T, 2>
01773 {
```

```
01774   __T operator() (const __T *__addr) const
01775   {
01776     __T __res; __LPM__2 (__res, __addr); return __res;
01777   }
01778 };
01779
01780 template<typename __T>
01781 struct __pgm_read_impl<__T, 3>
01782 {
01783   __T operator() (const __T *__addr) const
01784   {
01785     __T __res; __LPM__3 (__res, __addr); return __res;
01786   }
01787 };
01788
01789 template<typename __T>
01790 struct __pgm_read_impl<__T, 4>
01791 {
01792   __T operator() (const __T *__addr) const
01793   {
01794     __T __res; __LPM__4 (__res, __addr); return __res;
01795   }
01796 };
01797
01798 template<typename __T>
01799 struct __pgm_read_impl<__T, 8>
01800 {
01801   __T operator() (const __T *__addr) const
01802   {
01803     __T __res; __LPM__8 (__res, __addr); return __res;
01804   }
01805 };
01806
01807 template<typename __T>
01808 __T pgm_read (const __T *__addr)
01809 {
01810   return __pgm_read_impl<__T, sizeof(__T)>() (__addr);
01811 }
01812
01813 ////////////////////////////////////////////////////////
01814
01815 template<typename __T, size_t>
01816 struct __pgm_read_far_impl
01817 {
01818   // A default implementation for T's with a size not in { 1, 2, 3, 4, 8 }.
01819   // While this works, the performance is absolute scrap because GCC does
01820   // not handle objects well that don't fit in a register (i.e. avr-gcc
01821   // has no respective machine_mode).
01822   __T operator() (const __T *__addr) const
01823   {
01824     __T __res;
01825     memcpy_PF (&__res, __addr, sizeof(__T));
01826     return __res;
01827   }
01828 };
01829
01830 template<typename __T>
01831 struct __pgm_read_far_impl<__T, 1>
01832 {
01833   __T operator() (uint_farptr_t __addr) const
01834   {
01835     __T __res; __ELPM__1 (__res, __addr, __T); return __res;
01836   }
01837 };
01838
01839 template<typename __T>
01840 struct __pgm_read_far_impl<__T, 2>
01841 {
01842   __T operator() (uint_farptr_t __addr) const
01843   {
01844     __T __res; __ELPM__2 (__res, __addr, __T); return __res;
01845   }
01846 };
```

```
01847
01848 template<typename __T>
01849 struct __pgm_read_far_impl<__T, 3>
01850 {
01851   __T operator() (uint_farptr_t __addr) const
01852   {
01853     __T __res; __ELPM__3 (__res, __addr, __T); return __res;
01854   }
01855 };
01856
01857 template<typename __T>
01858 struct __pgm_read_far_impl<__T, 4>
01859 {
01860   __T operator() (uint_farptr_t __addr) const
01861   {
01862     __T __res; __ELPM__4 (__res, __addr, __T); return __res;
01863   }
01864 };
01865
01866 template<typename __T>
01867 struct __pgm_read_far_impl<__T, 8>
01868 {
01869   __T operator() (uint_farptr_t __addr) const
01870   {
01871     __T __res; __ELPM__8 (__res, __addr, __T); return __res;
01872   }
01873 };
01874
01875 template<typename __T>
01876 __T pgm_read_far (uint_farptr_t __addr)
01877 {
01878   return __pgm_read_far_impl<__T, sizeof(__T)>() (__addr);
01879 }
01880
01881 #endif /* C++ */
01882
01883 #ifdef __DOXYGEN__
01884
01885 /** \name Templates */
01886
01887 /** \ingroup avr_pgmspace
01888     \fn T pgm_read<T> (const T *addr)
01889
01890     Read an object of type \c T from program memory address \p addr and
01891     return it.
01892     This template is only available when macro \c __pgm_read_template__
01893     is defined.
01894     \since AVR-LibC v2.2  */
01895 template<typename T>
01896 T pgm_read<T> (const T *addr);
01897
01898 /** \ingroup avr_pgmspace
01899     \fn T pgm_read_far<T> (uint_farptr_t addr)
01900
01901     Read an object of type \c T from program memory address \p addr and
01902     return it.
01903     This template is only available when macro \c __pgm_read_template__
01904     is defined.
01905     \since AVR-LibC v2.2  */
01906 template<typename T>
01907 T pgm_read_far<T> (uint_farptr_t addr);
01908 #endif /* DOXYGEN */
01909
01910 #endif /* __PGMSPACE_H_ */
```

22.36 portpins.h

```
00001 /* Copyright (c) 2003  Theodore A. Roth
00002    All rights reserved.
00003
00004    Redistribution and use in source and binary forms, with or without
```

```
00005    modification, are permitted provided that the following conditions are met:
00006
00007    * Redistributions of source code must retain the above copyright
00008      notice, this list of conditions and the following disclaimer.
00009
00010    * Redistributions in binary form must reproduce the above copyright
00011      notice, this list of conditions and the following disclaimer in
00012      the documentation and/or other materials provided with the
00013      distribution.
00014
00015    * Neither the name of the copyright holders nor the names of
00016      contributors may be used to endorse or promote products derived
00017      from this software without specific prior written permission.
00018
00019   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
00020   AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
00021   IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
00022   ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
00023   LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
00024   CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
00025   SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
00026   INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
00027   CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
00028   ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
00029   POSSIBILITY OF SUCH DAMAGE. */
00030
00031 #ifndef _AVR_PORTPINS_H_
00032 #define _AVR_PORTPINS_H_ 1
00033
00034 /* This file should only be included from <avr/io.h>, never directly. */
00035
00036 #ifndef _AVR_IO_H_
00037 #  error "Include <avr/io.h> instead of this file."
00038 #endif
00039
00040 /* Define Generic PORTn, DDn, and PINn values. */
00041
00042 /* Port Data Register (generic) */
00043 #define    PORT7        7
00044 #define    PORT6        6
00045 #define    PORT5        5
00046 #define    PORT4        4
00047 #define    PORT3        3
00048 #define    PORT2        2
00049 #define    PORT1        1
00050 #define    PORT0        0
00051
00052 /* Port Data Direction Register (generic) */
00053 #define    DD7          7
00054 #define    DD6          6
00055 #define    DD5          5
00056 #define    DD4          4
00057 #define    DD3          3
00058 #define    DD2          2
00059 #define    DD1          1
00060 #define    DD0          0
00061
00062 /* Port Input Pins (generic) */
00063 #define    PIN7         7
00064 #define    PIN6         6
00065 #define    PIN5         5
00066 #define    PIN4         4
00067 #define    PIN3         3
00068 #define    PIN2         2
00069 #define    PIN1         1
00070 #define    PIN0         0
00071
00072 /* Define PORTxn an Pxn values for all possible port pins if not defined already by
      io.h. */
00073
00074 /* PORT A */
00075
00076 #if defined(PA0) && !defined(PORTA0)
```

```
00077 #  define PORTA0 PA0
00078 #elif defined(PORTA0) && !defined(PA0)
00079 #  define PA0 PORTA0
00080 #endif
00081 #if defined(PA1) && !defined(PORTA1)
00082 #  define PORTA1 PA1
00083 #elif defined(PORTA1) && !defined(PA1)
00084 #  define PA1 PORTA1
00085 #endif
00086 #if defined(PA2) && !defined(PORTA2)
00087 #  define PORTA2 PA2
00088 #elif defined(PORTA2) && !defined(PA2)
00089 #  define PA2 PORTA2
00090 #endif
00091 #if defined(PA3) && !defined(PORTA3)
00092 #  define PORTA3 PA3
00093 #elif defined(PORTA3) && !defined(PA3)
00094 #  define PA3 PORTA3
00095 #endif
00096 #if defined(PA4) && !defined(PORTA4)
00097 #  define PORTA4 PA4
00098 #elif defined(PORTA4) && !defined(PA4)
00099 #  define PA4 PORTA4
00100 #endif
00101 #if defined(PA5) && !defined(PORTA5)
00102 #  define PORTA5 PA5
00103 #elif defined(PORTA5) && !defined(PA5)
00104 #  define PA5 PORTA5
00105 #endif
00106 #if defined(PA6) && !defined(PORTA6)
00107 #  define PORTA6 PA6
00108 #elif defined(PORTA6) && !defined(PA6)
00109 #  define PA6 PORTA6
00110 #endif
00111 #if defined(PA7) && !defined(PORTA7)
00112 #  define PORTA7 PA7
00113 #elif defined(PORTA7) && !defined(PA7)
00114 #  define PA7 PORTA7
00115 #endif
00116
00117 /* PORT B */
00118
00119 #if defined(PB0) && !defined(PORTB0)
00120 #  define PORTB0 PB0
00121 #elif defined(PORTB0) && !defined(PB0)
00122 #  define PB0 PORTB0
00123 #endif
00124 #if defined(PB1) && !defined(PORTB1)
00125 #  define PORTB1 PB1
00126 #elif defined(PORTB1) && !defined(PB1)
00127 #  define PB1 PORTB1
00128 #endif
00129 #if defined(PB2) && !defined(PORTB2)
00130 #  define PORTB2 PB2
00131 #elif defined(PORTB2) && !defined(PB2)
00132 #  define PB2 PORTB2
00133 #endif
00134 #if defined(PB3) && !defined(PORTB3)
00135 #  define PORTB3 PB3
00136 #elif defined(PORTB3) && !defined(PB3)
00137 #  define PB3 PORTB3
00138 #endif
00139 #if defined(PB4) && !defined(PORTB4)
00140 #  define PORTB4 PB4
00141 #elif defined(PORTB4) && !defined(PB4)
00142 #  define PB4 PORTB4
00143 #endif
00144 #if defined(PB5) && !defined(PORTB5)
00145 #  define PORTB5 PB5
00146 #elif defined(PORTB5) && !defined(PB5)
00147 #  define PB5 PORTB5
00148 #endif
00149 #if defined(PB6) && !defined(PORTB6)
```

```
00150 #  define PORTB6 PB6
00151 #elif defined(PORTB6) && !defined(PB6)
00152 #  define PB6 PORTB6
00153 #endif
00154 #if defined(PB7) && !defined(PORTB7)
00155 #  define PORTB7 PB7
00156 #elif defined(PORTB7) && !defined(PB7)
00157 #  define PB7 PORTB7
00158 #endif
00159
00160 /* PORT C */
00161
00162 #if defined(PC0) && !defined(PORTC0)
00163 #  define PORTC0 PC0
00164 #elif defined(PORTC0) && !defined(PC0)
00165 #  define PC0 PORTC0
00166 #endif
00167 #if defined(PC1) && !defined(PORTC1)
00168 #  define PORTC1 PC1
00169 #elif defined(PORTC1) && !defined(PC1)
00170 #  define PC1 PORTC1
00171 #endif
00172 #if defined(PC2) && !defined(PORTC2)
00173 #  define PORTC2 PC2
00174 #elif defined(PORTC2) && !defined(PC2)
00175 #  define PC2 PORTC2
00176 #endif
00177 #if defined(PC3) && !defined(PORTC3)
00178 #  define PORTC3 PC3
00179 #elif defined(PORTC3) && !defined(PC3)
00180 #  define PC3 PORTC3
00181 #endif
00182 #if defined(PC4) && !defined(PORTC4)
00183 #  define PORTC4 PC4
00184 #elif defined(PORTC4) && !defined(PC4)
00185 #  define PC4 PORTC4
00186 #endif
00187 #if defined(PC5) && !defined(PORTC5)
00188 #  define PORTC5 PC5
00189 #elif defined(PORTC5) && !defined(PC5)
00190 #  define PC5 PORTC5
00191 #endif
00192 #if defined(PC6) && !defined(PORTC6)
00193 #  define PORTC6 PC6
00194 #elif defined(PORTC6) && !defined(PC6)
00195 #  define PC6 PORTC6
00196 #endif
00197 #if defined(PC7) && !defined(PORTC7)
00198 #  define PORTC7 PC7
00199 #elif defined(PORTC7) && !defined(PC7)
00200 #  define PC7 PORTC7
00201 #endif
00202
00203 /* PORT D */
00204
00205 #if defined(PD0) && !defined(PORTD0)
00206 #  define PORTD0 PD0
00207 #elif defined(PORTD0) && !defined(PD0)
00208 #  define PD0 PORTD0
00209 #endif
00210 #if defined(PD1) && !defined(PORTD1)
00211 #  define PORTD1 PD1
00212 #elif defined(PORTD1) && !defined(PD1)
00213 #  define PD1 PORTD1
00214 #endif
00215 #if defined(PD2) && !defined(PORTD2)
00216 #  define PORTD2 PD2
00217 #elif defined(PORTD2) && !defined(PD2)
00218 #  define PD2 PORTD2
00219 #endif
00220 #if defined(PD3) && !defined(PORTD3)
00221 #  define PORTD3 PD3
00222 #elif defined(PORTD3) && !defined(PD3)
```

```
00223 #  define PD3 PORTD3
00224 #endif
00225 #if defined(PD4) && !defined(PORTD4)
00226 #  define PORTD4 PD4
00227 #elif defined(PORTD4) && !defined(PD4)
00228 #  define PD4 PORTD4
00229 #endif
00230 #if defined(PD5) && !defined(PORTD5)
00231 #  define PORTD5 PD5
00232 #elif defined(PORTD5) && !defined(PD5)
00233 #  define PD5 PORTD5
00234 #endif
00235 #if defined(PD6) && !defined(PORTD6)
00236 #  define PORTD6 PD6
00237 #elif defined(PORTD6) && !defined(PD6)
00238 #  define PD6 PORTD6
00239 #endif
00240 #if defined(PD7) && !defined(PORTD7)
00241 #  define PORTD7 PD7
00242 #elif defined(PORTD7) && !defined(PD7)
00243 #  define PD7 PORTD7
00244 #endif
00245
00246 /* PORT E */
00247
00248 #if defined(PE0) && !defined(PORTE0)
00249 #  define PORTE0 PE0
00250 #elif defined(PORTE0) && !defined(PE0)
00251 #  define PE0 PORTE0
00252 #endif
00253 #if defined(PE1) && !defined(PORTE1)
00254 #  define PORTE1 PE1
00255 #elif defined(PORTE1) && !defined(PE1)
00256 #  define PE1 PORTE1
00257 #endif
00258 #if defined(PE2) && !defined(PORTE2)
00259 #  define PORTE2 PE2
00260 #elif defined(PORTE2) && !defined(PE2)
00261 #  define PE2 PORTE2
00262 #endif
00263 #if defined(PE3) && !defined(PORTE3)
00264 #  define PORTE3 PE3
00265 #elif defined(PORTE3) && !defined(PE3)
00266 #  define PE3 PORTE3
00267 #endif
00268 #if defined(PE4) && !defined(PORTE4)
00269 #  define PORTE4 PE4
00270 #elif defined(PORTE4) && !defined(PE4)
00271 #  define PE4 PORTE4
00272 #endif
00273 #if defined(PE5) && !defined(PORTE5)
00274 #  define PORTE5 PE5
00275 #elif defined(PORTE5) && !defined(PE5)
00276 #  define PE5 PORTE5
00277 #endif
00278 #if defined(PE6) && !defined(PORTE6)
00279 #  define PORTE6 PE6
00280 #elif defined(PORTE6) && !defined(PE6)
00281 #  define PE6 PORTE6
00282 #endif
00283 #if defined(PE7) && !defined(PORTE7)
00284 #  define PORTE7 PE7
00285 #elif defined(PORTE7) && !defined(PE7)
00286 #  define PE7 PORTE7
00287 #endif
00288
00289 /* PORT F */
00290
00291 #if defined(PF0) && !defined(PORTF0)
00292 #  define PORTF0 PF0
00293 #elif defined(PORTF0) && !defined(PF0)
00294 #  define PF0 PORTF0
00295 #endif
```

```
00296 #if defined(PF1) && !defined(PORTF1)
00297 #  define PORTF1 PF1
00298 #elif defined(PORTF1) && !defined(PF1)
00299 #  define PF1 PORTF1
00300 #endif
00301 #if defined(PF2) && !defined(PORTF2)
00302 #  define PORTF2 PF2
00303 #elif defined(PORTF2) && !defined(PF2)
00304 #  define PF2 PORTF2
00305 #endif
00306 #if defined(PF3) && !defined(PORTF3)
00307 #  define PORTF3 PF3
00308 #elif defined(PORTF3) && !defined(PF3)
00309 #  define PF3 PORTF3
00310 #endif
00311 #if defined(PF4) && !defined(PORTF4)
00312 #  define PORTF4 PF4
00313 #elif defined(PORTF4) && !defined(PF4)
00314 #  define PF4 PORTF4
00315 #endif
00316 #if defined(PF5) && !defined(PORTF5)
00317 #  define PORTF5 PF5
00318 #elif defined(PORTF5) && !defined(PF5)
00319 #  define PF5 PORTF5
00320 #endif
00321 #if defined(PF6) && !defined(PORTF6)
00322 #  define PORTF6 PF6
00323 #elif defined(PORTF6) && !defined(PF6)
00324 #  define PF6 PORTF6
00325 #endif
00326 #if defined(PF7) && !defined(PORTF7)
00327 #  define PORTF7 PF7
00328 #elif defined(PORTF7) && !defined(PF7)
00329 #  define PF7 PORTF7
00330 #endif
00331
00332 /* PORT G */
00333
00334 #if defined(PG0) && !defined(PORTG0)
00335 #  define PORTG0 PG0
00336 #elif defined(PORTG0) && !defined(PG0)
00337 #  define PG0 PORTG0
00338 #endif
00339 #if defined(PG1) && !defined(PORTG1)
00340 #  define PORTG1 PG1
00341 #elif defined(PORTG1) && !defined(PG1)
00342 #  define PG1 PORTG1
00343 #endif
00344 #if defined(PG2) && !defined(PORTG2)
00345 #  define PORTG2 PG2
00346 #elif defined(PORTG2) && !defined(PG2)
00347 #  define PG2 PORTG2
00348 #endif
00349 #if defined(PG3) && !defined(PORTG3)
00350 #  define PORTG3 PG3
00351 #elif defined(PORTG3) && !defined(PG3)
00352 #  define PG3 PORTG3
00353 #endif
00354 #if defined(PG4) && !defined(PORTG4)
00355 #  define PORTG4 PG4
00356 #elif defined(PORTG4) && !defined(PG4)
00357 #  define PG4 PORTG4
00358 #endif
00359 #if defined(PG5) && !defined(PORTG5)
00360 #  define PORTG5 PG5
00361 #elif defined(PORTG5) && !defined(PG5)
00362 #  define PG5 PORTG5
00363 #endif
00364 #if defined(PG6) && !defined(PORTG6)
00365 #  define PORTG6 PG6
00366 #elif defined(PORTG6) && !defined(PG6)
00367 #  define PG6 PORTG6
00368 #endif
```

```
00369 #if defined(PG7) && !defined(PORTG7)
00370 #  define PORTG7 PG7
00371 #elif defined(PORTG7) && !defined(PG7)
00372 #  define PG7 PORTG7
00373 #endif
00374
00375 /* PORT H */
00376
00377 #if defined(PH0) && !defined(PORTH0)
00378 #  define PORTH0 PH0
00379 #elif defined(PORTH0) && !defined(PH0)
00380 #  define PH0 PORTH0
00381 #endif
00382 #if defined(PH1) && !defined(PORTH1)
00383 #  define PORTH1 PH1
00384 #elif defined(PORTH1) && !defined(PH1)
00385 #  define PH1 PORTH1
00386 #endif
00387 #if defined(PH2) && !defined(PORTH2)
00388 #  define PORTH2 PH2
00389 #elif defined(PORTH2) && !defined(PH2)
00390 #  define PH2 PORTH2
00391 #endif
00392 #if defined(PH3) && !defined(PORTH3)
00393 #  define PORTH3 PH3
00394 #elif defined(PORTH3) && !defined(PH3)
00395 #  define PH3 PORTH3
00396 #endif
00397 #if defined(PH4) && !defined(PORTH4)
00398 #  define PORTH4 PH4
00399 #elif defined(PORTH4) && !defined(PH4)
00400 #  define PH4 PORTH4
00401 #endif
00402 #if defined(PH5) && !defined(PORTH5)
00403 #  define PORTH5 PH5
00404 #elif defined(PORTH5) && !defined(PH5)
00405 #  define PH5 PORTH5
00406 #endif
00407 #if defined(PH6) && !defined(PORTH6)
00408 #  define PORTH6 PH6
00409 #elif defined(PORTH6) && !defined(PH6)
00410 #  define PH6 PORTH6
00411 #endif
00412 #if defined(PH7) && !defined(PORTH7)
00413 #  define PORTH7 PH7
00414 #elif defined(PORTH7) && !defined(PH7)
00415 #  define PH7 PORTH7
00416 #endif
00417
00418 /* PORT J */
00419
00420 #if defined(PJ0) && !defined(PORTJ0)
00421 #  define PORTJ0 PJ0
00422 #elif defined(PORTJ0) && !defined(PJ0)
00423 #  define PJ0 PORTJ0
00424 #endif
00425 #if defined(PJ1) && !defined(PORTJ1)
00426 #  define PORTJ1 PJ1
00427 #elif defined(PORTJ1) && !defined(PJ1)
00428 #  define PJ1 PORTJ1
00429 #endif
00430 #if defined(PJ2) && !defined(PORTJ2)
00431 #  define PORTJ2 PJ2
00432 #elif defined(PORTJ2) && !defined(PJ2)
00433 #  define PJ2 PORTJ2
00434 #endif
00435 #if defined(PJ3) && !defined(PORTJ3)
00436 #  define PORTJ3 PJ3
00437 #elif defined(PORTJ3) && !defined(PJ3)
00438 #  define PJ3 PORTJ3
00439 #endif
00440 #if defined(PJ4) && !defined(PORTJ4)
00441 #  define PORTJ4 PJ4
```

```
00442 #elif defined(PORTJ4) && !defined(PJ4)
00443 #  define PJ4 PORTJ4
00444 #endif
00445 #if defined(PJ5) && !defined(PORTJ5)
00446 #  define PORTJ5 PJ5
00447 #elif defined(PORTJ5) && !defined(PJ5)
00448 #  define PJ5 PORTJ5
00449 #endif
00450 #if defined(PJ6) && !defined(PORTJ6)
00451 #  define PORTJ6 PJ6
00452 #elif defined(PORTJ6) && !defined(PJ6)
00453 #  define PJ6 PORTJ6
00454 #endif
00455 #if defined(PJ7) && !defined(PORTJ7)
00456 #  define PORTJ7 PJ7
00457 #elif defined(PORTJ7) && !defined(PJ7)
00458 #  define PJ7 PORTJ7
00459 #endif
00460
00461 /* PORT K */
00462
00463 #if defined(PK0) && !defined(PORTK0)
00464 #  define PORTK0 PK0
00465 #elif defined(PORTK0) && !defined(PK0)
00466 #  define PK0 PORTK0
00467 #endif
00468 #if defined(PK1) && !defined(PORTK1)
00469 #  define PORTK1 PK1
00470 #elif defined(PORTK1) && !defined(PK1)
00471 #  define PK1 PORTK1
00472 #endif
00473 #if defined(PK2) && !defined(PORTK2)
00474 #  define PORTK2 PK2
00475 #elif defined(PORTK2) && !defined(PK2)
00476 #  define PK2 PORTK2
00477 #endif
00478 #if defined(PK3) && !defined(PORTK3)
00479 #  define PORTK3 PK3
00480 #elif defined(PORTK3) && !defined(PK3)
00481 #  define PK3 PORTK3
00482 #endif
00483 #if defined(PK4) && !defined(PORTK4)
00484 #  define PORTK4 PK4
00485 #elif defined(PORTK4) && !defined(PK4)
00486 #  define PK4 PORTK4
00487 #endif
00488 #if defined(PK5) && !defined(PORTK5)
00489 #  define PORTK5 PK5
00490 #elif defined(PORTK5) && !defined(PK5)
00491 #  define PK5 PORTK5
00492 #endif
00493 #if defined(PK6) && !defined(PORTK6)
00494 #  define PORTK6 PK6
00495 #elif defined(PORTK6) && !defined(PK6)
00496 #  define PK6 PORTK6
00497 #endif
00498 #if defined(PK7) && !defined(PORTK7)
00499 #  define PORTK7 PK7
00500 #elif defined(PORTK7) && !defined(PK7)
00501 #  define PK7 PORTK7
00502 #endif
00503
00504 /* PORT L */
00505
00506 #if defined(PL0) && !defined(PORTL0)
00507 #  define PORTL0 PL0
00508 #elif defined(PORTL0) && !defined(PL0)
00509 #  define PL0 PORTL0
00510 #endif
00511 #if defined(PL1) && !defined(PORTL1)
00512 #  define PORTL1 PL1
00513 #elif defined(PORTL1) && !defined(PL1)
00514 #  define PL1 PORTL1
```

```
00515 #endif
00516 #if defined(PL2) && !defined(PORTL2)
00517 #  define PORTL2 PL2
00518 #elif defined(PORTL2) && !defined(PL2)
00519 #  define PL2 PORTL2
00520 #endif
00521 #if defined(PL3) && !defined(PORTL3)
00522 #  define PORTL3 PL3
00523 #elif defined(PORTL3) && !defined(PL3)
00524 #  define PL3 PORTL3
00525 #endif
00526 #if defined(PL4) && !defined(PORTL4)
00527 #  define PORTL4 PL4
00528 #elif defined(PORTL4) && !defined(PL4)
00529 #  define PL4 PORTL4
00530 #endif
00531 #if defined(PL5) && !defined(PORTL5)
00532 #  define PORTL5 PL5
00533 #elif defined(PORTL5) && !defined(PL5)
00534 #  define PL5 PORTL5
00535 #endif
00536 #if defined(PL6) && !defined(PORTL6)
00537 #  define PORTL6 PL6
00538 #elif defined(PORTL6) && !defined(PL6)
00539 #  define PL6 PORTL6
00540 #endif
00541 #if defined(PL7) && !defined(PORTL7)
00542 #  define PORTL7 PL7
00543 #elif defined(PORTL7) && !defined(PL7)
00544 #  define PL7 PORTL7
00545 #endif
00546
00547 #endif /* _AVR_PORTPINS_H_ */
```

22.37 power.h File Reference

Macros

- #define clock_prescale_get() (clock_div_t)(CLKPR & (uint8_t)((1<<CLKPS0)|(1<<CLKPS1)|(1<<CLKPS2)|(1<<CLKPS3))

Functions

- static void power_all_enable ()
- static void power_all_disable ()
- void clock_prescale_set (clock_div_t __x)

22.38 power.h

Go to the documentation of this file.

```
00001 /* Copyright (c) 2006, 2007, 2008  Eric B. Weddington
00002    Copyright (c) 2011 Frédéric Nadeau
00003    All rights reserved.
00004
00005    Redistribution and use in source and binary forms, with or without
00006    modification, are permitted provided that the following conditions are met:
00007
00008    * Redistributions of source code must retain the above copyright
00009      notice, this list of conditions and the following disclaimer.
00010    * Redistributions in binary form must reproduce the above copyright
00011      notice, this list of conditions and the following disclaimer in
00012      the documentation and/or other materials provided with the
00013      distribution.
```

```
00014    * Neither the name of the copyright holders nor the names of
00015      contributors may be used to endorse or promote products derived
00016      from this software without specific prior written permission.
00017
00018   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
00019   AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
00020   IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
00021   ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
00022   LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
00023   CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
00024   SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
00025   INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
00026   CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
00027   ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
00028   POSSIBILITY OF SUCH DAMAGE. */
00029
00030 #ifndef _AVR_POWER_H_
00031 #define _AVR_POWER_H_   1
00032
00033 #include <avr/io.h>
00034 #include <stdint.h>
00035 #include <bits/attribs.h>
00036
00037 /** \file */
00038 /** \defgroup avr_power <avr/power.h>: Power Reduction Management
00039
00040 \code #include <avr/power.h>\endcode
00041
00042 Many AVRs contain a Power Reduction Register (PRR) or Registers (PRRx) that
00043 allow you to reduce power consumption by disabling or enabling various on-board
00044 peripherals as needed. Some devices have the XTAL Divide Control Register
00045 (XDIV) which offer similar functionality as System Clock Prescale
00046 Register (CLKPR).
00047
00048 There are many macros in this header file that provide an easy interface
00049 to enable or disable on-board peripherals to reduce power. See the table below.
00050
00051 \note Not all AVR devices have a Power Reduction Register (for example
00052 the ATmega8). On those devices without a Power Reduction Register, the
00053 power reduction macros are not available..
00054
00055 \note Not all AVR devices contain the same peripherals (for example, the LCD
00056 interface), or they will be named differently (for example, USART and
00057 USART0). Please consult your device's datasheet, or the header file, to
00058 find out which macros are applicable to your device.
00059
00060 \note For device using the XTAL Divide Control Register (XDIV), when prescaler
00061 is used, Timer/Counter0 can only be used in asynchronous mode. Keep in mind
00062 that Timer/Counter0 source shall be less than ¼th of peripheral clock.
00063 Therefore, when using a typical 32.768 kHz crystal, one shall not scale
00064 the clock below 131.072 kHz.
00065
00066 \anchor avr_powermacros
00067 <small>
00068 <table>
00069   <caption>Power Macros</caption>
00070   <tr>
00071     <th>Power Macro
00072     <th>Description
00073   </tr>
00074   <tr>
00075     <td>\c power_aca_disable()</td>
00076     <td>Disable the Analog Comparator on PortA</td>
00077   </tr>
00078   <tr>
00079     <td>\c power_aca_enable()</td>
00080     <td>Enable the Analog Comparator on PortA</td>
00081   </tr>
00082   <tr>
00083     <td>\c power_adc_enable()</td>
00084     <td>Enable the Analog to Digital Converter module</td>
00085   </tr>
00086   <tr>
```

```
00087     <td>\c power_adc_disable()</td>
00088     <td>Disable the Analog to Digital Converter module</td>
00089   </tr>
00090   <tr>
00091     <td>\c power_adca_disable()</td>
00092     <td>Disable the Analog to Digital Converter module on PortA</td>
00093   </tr>
00094   <tr>
00095     <td>\c power_adca_enable()</td>
00096     <td>Enable the Analog to Digital Converter module on PortA</td>
00097   </tr>
00098   <tr>
00099     <td>\c power_evsys_disable()</td>
00100     <td>Disable the EVSYS module</td>
00101   </tr>
00102   <tr>
00103     <td>\c power_evsys_enable()</td>
00104     <td>Enable the EVSYS module</td>
00105   </tr>
00106   <tr>
00107     <td>\c power_hiresc_disable()</td>
00108     <td>Disable the HIRES module on PortC</td>
00109   </tr>
00110   <tr>
00111     <td>\c power_hiresc_enable()</td>
00112     <td>Enable the HIRES module on PortC</td>
00113   </tr>
00114   <tr>
00115     <td>\c power_lcd_enable()</td>
00116     <td>Enable the LCD module</td>
00117   </tr>
00118   <tr>
00119     <td>\c power_lcd_disable()</td>
00120     <td>Disable the LCD module</td>
00121   </tr>
00122   <tr>
00123     <td>\c power_pga_enable()</td>
00124     <td>Enable the Programmable Gain Amplifier module</td>
00125   </tr>
00126   <tr>
00127     <td>\c power_pga_disable()</td>
00128     <td>Disable the Programmable Gain Amplifier module</td>
00129   </tr>
00130   <tr>
00131     <td>\c power_pscr_enable()</td>
00132     <td>Enable the Reduced Power Stage Controller module</td>
00133   </tr>
00134   <tr>
00135     <td>\c power_pscr_disable()</td>
00136     <td>Disable the Reduced Power Stage Controller module</td>
00137   </tr>
00138   <tr>
00139     <td>\c power_psc0_enable()</td>
00140     <td>Enable the Power Stage Controller 0 module</td>
00141   </tr>
00142   <tr>
00143     <td>\c power_psc0_disable()</td>
00144     <td>Disable the Power Stage Controller 0 module</td>
00145   </tr>
00146   <tr>
00147     <td>\c power_psc1_enable()</td>
00148     <td>Enable the Power Stage Controller 1 module</td>
00149   </tr>
00150   <tr>
00151     <td>\c power_psc1_disable()</td>
00152     <td>Disable the Power Stage Controller 1 module</td>
00153   </tr>
00154   <tr>
00155     <td>\c power_psc2_enable()</td>
00156     <td>Enable the Power Stage Controller 2 module</td>
00157   </tr>
00158   <tr>
00159     <td>\c power_psc2_disable()</td>
```

```
00160     <td>Disable the Power Stage Controller 2 module</td>
00161   </tr>
00162   <tr>
00163     <td>\c power_ram0_enable()</td>
00164     <td>Enable the SRAM block 0</td>
00165   </tr>
00166   <tr>
00167     <td>\c power_ram0_disable()</td>
00168     <td>Disable the SRAM block 0</td>
00169   </tr>
00170   <tr>
00171     <td>\c power_ram1_enable()</td>
00172     <td>Enable the SRAM block 1</td>
00173   </tr>
00174   <tr>
00175     <td>\c power_ram1_disable()</td>
00176     <td>Disable the SRAM block 1</td>
00177   </tr>
00178   <tr>
00179     <td>\c power_ram2_enable()</td>
00180     <td>Enable the SRAM block 2</td>
00181   </tr>
00182   <tr>
00183     <td>\c power_ram2_disable()</td>
00184     <td>Disable the SRAM block 2</td>
00185   </tr>
00186   <tr>
00187     <td>\c power_ram3_enable()</td>
00188     <td>Enable the SRAM block 3</td>
00189   </tr>
00190   <tr>
00191     <td>\c power_ram3_disable()</td>
00192     <td>Disable the SRAM block 3</td>
00193   </tr>
00194   <tr>
00195     <td>\c power_rtc_disable()</td>
00196     <td>Disable the RTC module</td>
00197   </tr>
00198   <tr>
00199     <td>\c power_rtc_enable()</td>
00200     <td>Enable the RTC module</td>
00201   </tr>
00202   <tr>
00203     <td>\c power_spi_enable()</td>
00204     <td>Enable the Serial Peripheral Interface module</td>
00205   </tr>
00206   <tr>
00207     <td>\c power_spi_disable()</td>
00208     <td>Disable the Serial Peripheral Interface module</td>
00209   </tr>
00210   <tr>
00211     <td>\c power_spic_disable()</td>
00212     <td>Disable the SPI module on PortC</td>
00213   </tr>
00214   <tr>
00215     <td>\c power_spic_enable()</td>
00216     <td>Enable the SPI module on PortC</td>
00217   </tr>
00218   <tr>
00219     <td>\c power_spid_disable()</td>
00220     <td>Disable the SPI module on PortD</td>
00221   </tr>
00222   <tr>
00223     <td>\c power_spid_enable()</td>
00224     <td>Enable the SPI module on PortD</td>
00225   </tr>
00226   <tr>
00227     <td>\c power_tc0c_disable()</td>
00228     <td>Disable the TC0 module on PortC</td>
00229   </tr>
00230   <tr>
00231     <td>\c power_tc0c_enable()</td>
00232     <td>Enable the TC0 module on PortC</td>
```

```
00233   </tr>
00234   <tr>
00235     <td>\c power_tc0d_disable()</td>
00236     <td>Disable the TC0 module on PortD</td>
00237   </tr>
00238   <tr>
00239     <td>\c power_tc0d_enable()</td>
00240     <td>Enable the TC0 module on PortD</td>
00241   </tr>
00242   <tr>
00243     <td>\c power_tc0e_disable()</td>
00244     <td>Disable the TC0 module on PortE</td>
00245   </tr>
00246   <tr>
00247     <td>\c power_tc0e_enable()</td>
00248     <td>Enable the TC0 module on PortE</td>
00249   </tr>
00250   <tr>
00251     <td>\c power_tc0f_disable()</td>
00252     <td>Disable the TC0 module on PortF</td>
00253   </tr>
00254   <tr>
00255     <td>\c power_tc0f_enable()</td>
00256     <td>Enable the TC0 module on PortF</td>
00257   </tr>
00258   <tr>
00259     <td>\c power_tc1c_disable()</td>
00260     <td>Disable the TC1 module on PortC</td>
00261   </tr>
00262   <tr>
00263     <td>\c power_tc1c_enable()</td>
00264     <td>Enable the TC1 module on PortC</td>
00265   </tr>
00266   <tr>
00267     <td>\c power_twic_disable()</td>
00268     <td>Disable the Two Wire Interface module on PortC</td>
00269   </tr>
00270   <tr>
00271     <td>\c power_twic_enable()</td>
00272     <td>Enable the Two Wire Interface module on PortC</td>
00273   </tr>
00274   <tr>
00275     <td>\c power_twie_disable()</td>
00276     <td>Disable the Two Wire Interface module on PortE</td>
00277   </tr>
00278   <tr>
00279     <td>\c power_twie_enable()</td>
00280     <td>Enable the Two Wire Interface module on PortE</td>
00281   </tr>
00282   <tr>
00283     <td>\c power_timer0_enable()</td>
00284     <td>Enable the Timer 0 module</td>
00285   </tr>
00286   <tr>
00287     <td>\c power_timer0_disable()</td>
00288     <td>Disable the Timer 0 module</td>
00289   </tr>
00290   <tr>
00291     <td>\c power_timer1_enable()</td>
00292     <td>Enable the Timer 1 module</td>
00293   </tr>
00294   <tr>
00295     <td>\c power_timer1_disable()</td>
00296     <td>Disable the Timer 1 module</td>
00297   </tr>
00298   <tr>
00299     <td>\c power_timer2_enable()</td>
00300     <td>Enable the Timer 2 module</td>
00301   </tr>
00302   <tr>
00303     <td>\c power_timer2_disable()</td>
00304     <td>Disable the Timer 2 module</td>
00305   </tr>
```

```
00306   <tr>
00307     <td>\c power_timer3_enable()</td>
00308     <td>Enable the Timer 3 module</td>
00309   </tr>
00310   <tr>
00311     <td>\c power_timer3_disable()</td>
00312     <td>Disable the Timer 3 module</td>
00313   </tr>
00314   <tr>
00315     <td>\c power_timer4_enable()</td>
00316     <td>Enable the Timer 4 module</td>
00317   </tr>
00318   <tr>
00319     <td>\c power_timer4_disable()</td>
00320     <td>Disable the Timer 4 module</td>
00321   </tr>
00322   <tr>
00323     <td>\c power_timer5_enable()</td>
00324     <td>Enable the Timer 5 module</td>
00325   </tr>
00326   <tr>
00327     <td>\c power_timer5_disable()</td>
00328     <td>Disable the Timer 5 module</td>
00329   </tr>
00330   <tr>
00331     <td>\c power_twi_enable()</td>
00332     <td>Enable the Two Wire Interface module</td>
00333   </tr>
00334   <tr>
00335     <td>\c power_twi_disable()</td>
00336     <td>Disable the Two Wire Interface module</td>
00337   </tr>
00338   <tr>
00339     <td>\c power_usart_enable()</td>
00340     <td>Enable the USART module</td>
00341   </tr>
00342   <tr>
00343     <td>\c power_usart_disable()</td>
00344     <td>Disable the USART module</td>
00345   </tr>
00346   <tr>
00347     <td>\c power_usart0_enable()</td>
00348     <td>Enable the USART 0 module</td>
00349   </tr>
00350   <tr>
00351     <td>\c power_usart0_disable()</td>
00352     <td>Disable the USART 0 module</td>
00353   </tr>
00354   <tr>
00355     <td>\c power_usart1_enable()</td>
00356     <td>Enable the USART 1 module</td>
00357   </tr>
00358   <tr>
00359     <td>\c power_usart1_disable()</td>
00360     <td>Disable the USART 1 module</td>
00361   </tr>
00362   <tr>
00363     <td>\c power_usart2_enable()</td>
00364     <td>Enable the USART 2 module</td>
00365   </tr>
00366   <tr>
00367     <td>\c power_usart2_disable()</td>
00368     <td>Disable the USART 2 module</td>
00369   </tr>
00370   <tr>
00371     <td>\c power_usart3_enable()</td>
00372     <td>Enable the USART 3 module</td>
00373   </tr>
00374   <tr>
00375     <td>\c power_usart3_disable()</td>
00376     <td>Disable the USART 3 module</td>
00377   </tr>
00378   <tr>
```

```
00379     <td>\c power_usartc0_disable()</td>
00380     <td> Disable the USART0 module on PortC</td>
00381   </tr>
00382   <tr>
00383     <td>\c power_usartc0_enable()</td>
00384     <td> Enable the USART0 module on PortC</td>
00385   </tr>
00386   <tr>
00387     <td>\c power_usartd0_disable()</td>
00388     <td> Disable the USART0 module on PortD</td>
00389   </tr>
00390   <tr>
00391     <td>\c power_usartd0_enable()</td>
00392     <td> Enable the USART0 module on PortD</td>
00393   </tr>
00394   <tr>
00395     <td>\c power_usarte0_disable()</td>
00396     <td> Disable the USART0 module on PortE</td>
00397   </tr>
00398   <tr>
00399     <td>\c power_usarte0_enable()</td>
00400     <td> Enable the USART0 module on PortE</td>
00401   </tr>
00402   <tr>
00403     <td>\c power_usartf0_disable()</td>
00404     <td> Disable the USART0 module on PortF</td>
00405   </tr>
00406   <tr>
00407     <td>\c power_usartf0_enable()</td>
00408     <td> Enable the USART0 module on PortF</td>
00409   </tr>
00410   <tr>
00411     <td>\c power_usb_enable()</td>
00412     <td>Enable the USB module</td>
00413   </tr>
00414   <tr>
00415     <td>\c power_usb_disable()</td>
00416     <td>Disable the USB module</td>
00417   </tr>
00418   <tr>
00419     <td>\c power_usi_enable()</td>
00420     <td>Enable the Universal Serial Interface module</td>
00421   </tr>
00422   <tr>
00423     <td>\c power_usi_disable()</td>
00424     <td>Disable the Universal Serial Interface module</td>
00425   </tr>
00426   <tr>
00427     <td>\c power_vadc_enable()</td>
00428     <td>Enable the Voltage ADC module</td>
00429   </tr>
00430   <tr>
00431     <td>\c power_vadc_disable()</td>
00432     <td>Disable the Voltage ADC module</td>
00433   </tr>
00434   <tr>
00435     <td>\c power_all_enable()</td>
00436     <td>Enable all modules</td>
00437   </tr>
00438   <tr>
00439     <td>\c power_all_disable()</td>
00440     <td>Disable all modules</td>
00441   </tr>
00442 </table>
00443 </small>
00444 */
00445
00446 #if defined(__AVR_HAVE_PRR_PRADC)
00447 #define power_adc_enable()      (PRR &= (uint8_t)~(1 << PRADC))
00448 #define power_adc_disable()     (PRR |= (uint8_t)(1 << PRADC))
00449 #endif
00450
00451 #if defined(__AVR_HAVE_PRR_PRCAN)
```

```
00452 #define power_can_enable()      (PRR &= (uint8_t)~(1 « PRCAN))
00453 #define power_can_disable()     (PRR |= (uint8_t)(1 « PRCAN))
00454 #endif
00455
00456 #if defined(__AVR_HAVE_PRR_PRLCD)
00457 #define power_lcd_enable()      (PRR &= (uint8_t)~(1 « PRLCD))
00458 #define power_lcd_disable()     (PRR |= (uint8_t)(1 « PRLCD))
00459 #endif
00460
00461 #if defined(__AVR_HAVE_PRR_PRLIN)
00462 #define power_lin_enable()      (PRR &= (uint8_t)~(1 « PRLIN))
00463 #define power_lin_disable()     (PRR |= (uint8_t)(1 « PRLIN))
00464 #endif
00465
00466 #if defined(__AVR_HAVE_PRR_PRPSC)
00467 #define power_psc_enable()      (PRR &= (uint8_t)~(1 « PRPSC))
00468 #define power_psc_disable()     (PRR |= (uint8_t)(1 « PRPSC))
00469 #endif
00470
00471 #if defined(__AVR_HAVE_PRR_PRPSC0)
00472 #define power_psc0_enable()     (PRR &= (uint8_t)~(1 « PRPSC0))
00473 #define power_psc0_disable()    (PRR |= (uint8_t)(1 « PRPSC0))
00474 #endif
00475
00476 #if defined(__AVR_HAVE_PRR_PRPSC1)
00477 #define power_psc1_enable()     (PRR &= (uint8_t)~(1 « PRPSC1))
00478 #define power_psc1_disable()    (PRR |= (uint8_t)(1 « PRPSC1))
00479 #endif
00480
00481 #if defined(__AVR_HAVE_PRR_PRPSC2)
00482 #define power_psc2_enable()     (PRR &= (uint8_t)~(1 « PRPSC2))
00483 #define power_psc2_disable()    (PRR |= (uint8_t)(1 « PRPSC2))
00484 #endif
00485
00486 #if defined(__AVR_HAVE_PRR_PRPSCR)
00487 #define power_pscr_enable()     (PRR &= (uint8_t)~(1 « PRPSCR))
00488 #define power_pscr_disable()    (PRR |= (uint8_t)(1 « PRPSCR))
00489 #endif
00490
00491 #if defined(__AVR_HAVE_PRR_PRSPI)
00492 #define power_spi_enable()      (PRR &= (uint8_t)~(1 « PRSPI))
00493 #define power_spi_disable()     (PRR |= (uint8_t)(1 « PRSPI))
00494 #endif
00495
00496 #if defined(__AVR_HAVE_PRR_PRTIM0)
00497 #define power_timer0_enable()   (PRR &= (uint8_t)~(1 « PRTIM0))
00498 #define power_timer0_disable()  (PRR |= (uint8_t)(1 « PRTIM0))
00499 #endif
00500
00501 #if defined(__AVR_HAVE_PRR_PRTIM1)
00502 #define power_timer1_enable()   (PRR &= (uint8_t)~(1 « PRTIM1))
00503 #define power_timer1_disable()  (PRR |= (uint8_t)(1 « PRTIM1))
00504 #endif
00505
00506 #if defined(__AVR_HAVE_PRR_PRTIM2)
00507 #define power_timer2_enable()   (PRR &= (uint8_t)~(1 « PRTIM2))
00508 #define power_timer2_disable()  (PRR |= (uint8_t)(1 « PRTIM2))
00509 #endif
00510
00511 #if defined(__AVR_HAVE_PRR_PRTWI)
00512 #define power_twi_enable()      (PRR &= (uint8_t)~(1 « PRTWI))
00513 #define power_twi_disable()     (PRR |= (uint8_t)(1 « PRTWI))
00514 #endif
00515
00516 #if defined(__AVR_HAVE_PRR_PRUSART)
00517 #define power_usart_enable()    (PRR &= (uint8_t)~(1 « PRUSART))
00518 #define power_usart_disable()   (PRR |= (uint8_t)(1 « PRUSART))
00519 #endif
00520
00521 #if defined(__AVR_HAVE_PRR_PRUSART0)
00522 #define power_usart0_enable()   (PRR &= (uint8_t)~(1 « PRUSART0))
00523 #define power_usart0_disable()  (PRR |= (uint8_t)(1 « PRUSART0))
00524 #endif
```

```
00525
00526 #if defined(__AVR_HAVE_PRR_PRUSART1)
00527 #define power_usart1_enable()   (PRR &= (uint8_t)~(1 « PRUSART1))
00528 #define power_usart1_disable()  (PRR |= (uint8_t)(1 « PRUSART1))
00529 #endif
00530
00531 #if defined(__AVR_HAVE_PRR_PRUSI)
00532 #define power_usi_enable()      (PRR &= (uint8_t)~(1 « PRUSI))
00533 #define power_usi_disable()     (PRR |= (uint8_t)(1 « PRUSI))
00534 #endif
00535
00536 #if defined(__AVR_HAVE_PRR0_PRADC)
00537 #define power_adc_enable()      (PRR0 &= (uint8_t)~(1 « PRADC))
00538 #define power_adc_disable()     (PRR0 |= (uint8_t)(1 « PRADC))
00539 #endif
00540
00541 #if defined(__AVR_HAVE_PRR0_PRC0)
00542 #define power_clock_output_enable()     (PRR0 &= (uint8_t)~(1 « PRCO))
00543 #define power_clock_output_disable()    (PRR0 |= (uint8_t)(1 « PRCO))
00544 #endif
00545
00546 #if defined(__AVR_HAVE_PRR0_PRCRC)
00547 #define power_crc_enable()              (PRR0 &= (uint8_t)~(1 « PRCRC))
00548 #define power_crc_disable()             (PRR0 |= (uint8_t)(1 « PRCRC))
00549 #endif
00550
00551 #if defined(__AVR_HAVE_PRR0_PRCU)
00552 #define power_crypto_enable()           (PRR0 &= (uint8_t)~(1 « PRCU))
00553 #define power_crypto_disable()          (PRR0 |= (uint8_t)(1 « PRCU))
00554 #endif
00555
00556 #if defined(__AVR_HAVE_PRR0_PRDS)
00557 #define power_irdriver_enable()         (PRR0 &= (uint8_t)~(1 « PRDS))
00558 #define power_irdriver_disable()        (PRR0 |= (uint8_t)(1 « PRDS))
00559 #endif
00560
00561 #if defined(__AVR_HAVE_PRR0_PRLFR)
00562 #define power_lfreceiver_enable()       (PRR0 &= (uint8_t)~(1 « PRLFR))
00563 #define power_lfreceiver_disable()      (PRR0 |= (uint8_t)(1 « PRLFR))
00564 #endif
00565
00566 #if defined(__AVR_HAVE_PRR0_PRLFRS)
00567 #define power_lfrs_enable()             (PRR0 &= (uint8_t)~(1 « PRLFRS))
00568 #define power_lfrs_disable()            (PRR0 |= (uint8_t)(1 « PRLFRS))
00569 #endif
00570
00571 #if defined(__AVR_HAVE_PRR0_PRLIN)
00572 #define power_lin_enable()              (PRR0 &= (uint8_t)~(1 « PRLIN))
00573 #define power_lin_disable()             (PRR0 |= (uint8_t)(1 « PRLIN))
00574 #endif
00575
00576 #if defined(__AVR_HAVE_PRR0_PRPGA)
00577 #define power_pga_enable()              (PRR0 &= (uint8_t)~(1 « PRPGA))
00578 #define power_pga_disable()             (PRR0 |= (uint8_t)(1 « PRPGA))
00579 #endif
00580
00581 #if defined(__AVR_HAVE_PRR0_PRRXDC)
00582 #define power_receive_dsp_control_enable()  (PRR0 &= (uint8_t)~(1 « PRRXDC))
00583 #define power_receive_dsp_control_disable() (PRR0 |= (uint8_t)(1 « PRRXDC))
00584 #endif
00585
00586 #if defined(__AVR_HAVE_PRR0_PRSPI)
00587 #define power_spi_enable()              (PRR0 &= (uint8_t)~(1 « PRSPI))
00588 #define power_spi_disable()             (PRR0 |= (uint8_t)(1 « PRSPI))
00589 #endif
00590
00591 #if defined(__AVR_HAVE_PRR0_PRT0)
00592 #define power_timer0_enable()           (PRR0 &= (uint8_t)~(1 « PRT0))
00593 #define power_timer0_disable()          (PRR0 |= (uint8_t)(1 « PRT0))
00594 #endif
00595
00596 #if defined(__AVR_HAVE_PRR0_PRTIM0)
00597 #define power_timer0_enable()           (PRR0 &= (uint8_t)~(1 « PRTIM0))
```

```
00598 #define power_timer0_disable()          (PRR0 |= (uint8_t)(1 « PRTIM0))
00599 #endif
00600
00601 #if defined(__AVR_HAVE_PRR0_PRT1)
00602 #define power_timer1_enable()           (PRR0 &= (uint8_t)~(1 « PRT1))
00603 #define power_timer1_disable()          (PRR0 |= (uint8_t)(1 « PRT1))
00604 #endif
00605
00606 #if defined(__AVR_HAVE_PRR0_PRTIM1)
00607 #define power_timer1_enable()           (PRR0 &= (uint8_t)~(1 « PRTIM1))
00608 #define power_timer1_disable()          (PRR0 |= (uint8_t)(1 « PRTIM1))
00609 #endif
00610
00611 #if defined(__AVR_HAVE_PRR0_PRT2)
00612 #define power_timer2_enable()           (PRR0 &= (uint8_t)~(1 « PRT2))
00613 #define power_timer2_disable()          (PRR0 |= (uint8_t)(1 « PRT2))
00614 #endif
00615
00616 #if defined(__AVR_HAVE_PRR0_PRTIM2)
00617 #define power_timer2_enable()           (PRR0 &= (uint8_t)~(1 « PRTIM2))
00618 #define power_timer2_disable()          (PRR0 |= (uint8_t)(1 « PRTIM2))
00619 #endif
00620
00621 #if defined(__AVR_HAVE_PRR0_PRT3)
00622 #define power_timer3_enable()           (PRR0 &= (uint8_t)~(1 « PRT3))
00623 #define power_timer3_disable()          (PRR0 |= (uint8_t)(1 « PRT3))
00624 #endif
00625
00626 #if defined(__AVR_HAVE_PRR0_PRTM)
00627 #define power_timermodulator_enable()   (PRR0 &= (uint8_t)~(1 « PRTM))
00628 #define power_timermodulator_disable()  (PRR0 |= (uint8_t)(1 « PRTM))
00629 #endif
00630
00631 #if defined(__AVR_HAVE_PRR0_PRTWI)
00632 #define power_twi_enable()              (PRR0 &= (uint8_t)~(1 « PRTWI))
00633 #define power_twi_disable()             (PRR0 |= (uint8_t)(1 « PRTWI))
00634 #endif
00635
00636 #if defined(__AVR_HAVE_PRR0_PRTWI0)
00637 #define power_twi0_enable()             (PRR0 &= (uint8_t)~(1 « PRTWI0))
00638 #define power_twi0_disable()            (PRR0 |= (uint8_t)(1 « PRTWI0))
00639 #if !defined(__AVR_HAVE_PRR0_PRTWI)
00640 #define power_twi_enable()              power_twi0_enable()
00641 #define power_twi_disable()             power_twi0_disable()
00642 #endif
00643 #endif
00644
00645 #if defined(__AVR_HAVE_PRR0_PRTWI1)
00646 #define power_twi1_enable()             (PRR0 &= (uint8_t)~(1 « PRTWI1))
00647 #define power_twi1_disable()            (PRR0 |= (uint8_t)(1 « PRTWI1))
00648 #endif
00649
00650 #if defined(__AVR_HAVE_PRR0_PRTXDC)
00651 #define power_transmit_dsp_control_enable()   (PRR0 &= (uint8_t)~(1 « PRTXDC))
00652 #define power_transmit_dsp_control_disable()  (PRR0 |= (uint8_t)(1 « PRTXDC))
00653 #endif
00654
00655 #if defined(__AVR_HAVE_PRR0_PRUSART0)
00656 #define power_usart0_enable()           (PRR0 &= (uint8_t)~(1 « PRUSART0))
00657 #define power_usart0_disable()          (PRR0 |= (uint8_t)(1 « PRUSART0))
00658 #endif
00659
00660 #if defined(__AVR_HAVE_PRR0_PRUSART1)
00661 #define power_usart1_enable()           (PRR0 &= (uint8_t)~(1 « PRUSART1))
00662 #define power_usart1_disable()          (PRR0 |= (uint8_t)(1 « PRUSART1))
00663 #endif
00664
00665 #if defined(__AVR_HAVE_PRR0_PRVADC)
00666 #define power_vadc_enable()             (PRR0 &= (uint8_t)~(1 « PRVADC))
00667 #define power_vadc_disable()            (PRR0 |= (uint8_t)(1 « PRVADC))
00668 #endif
00669
00670 #if defined(__AVR_HAVE_PRR0_PRVM)
```

```
00671 #define power_voltage_monitor_enable()  (PRR0 &= (uint8_t)~(1 « PRVM))
00672 #define power_voltage_monitor_disable() (PRR0 |= (uint8_t)(1 « PRVM))
00673 #endif
00674
00675 #if defined(__AVR_HAVE_PRR0_PRVRM)
00676 #define power_vrm_enable()              (PRR0 &= (uint8_t)~(1 « PRVRM))
00677 #define power_vrm_disable()             (PRR0 |= (uint8_t)(1 « PRVRM))
00678 #endif
00679
00680 #if defined(__AVR_HAVE_PRR1_PRAES)
00681 #define power_aes_enable()              (PRR1 &= (uint8_t)~(1 « PRAES))
00682 #define power_aes_disable()             (PRR1 |= (uint8_t)(1 « PRAES))
00683 #endif
00684
00685 #if defined(__AVR_HAVE_PRR1_PRCI)
00686 #define power_cinterface_enable()       (PRR1 &= (uint8_t)~(1 « PRCI))
00687 #define power_cinterface_disable()      (PRR1 |= (uint8_t)(1 « PRCI))
00688 #endif
00689
00690 #if defined(__AVR_HAVE_PRR1_PRHSSPI)
00691 #define power_hsspi_enable()            (PRR1 &= (uint8_t)~(1 « PRHSSPI))
00692 #define power_hsspi_disable()           (PRR1 |= (uint8_t)(1 « PRHSSPI))
00693 #endif
00694
00695 #if defined(__AVR_HAVE_PRR1_PRKB)
00696 #define power_kb_enable()               (PRR1 &= (uint8_t)~(1 « PRKB))
00697 #define power_kb_disable()              (PRR1 |= (uint8_t)(1 « PRKB))
00698 #endif
00699
00700 #if defined(__AVR_HAVE_PRR1_PRLFPH)
00701 #define power_lfph_enable()             (PRR1 &= (uint8_t)~(1 « PRLFPH))
00702 #define power_lfph_disable()            (PRR1 |= (uint8_t)(1 « PRLFPH))
00703 #endif
00704
00705 #if defined(__AVR_HAVE_PRR1_PRLFR)
00706 #define power_lfreceiver_enable()       (PRR1 &= (uint8_t)~(1 « PRLFR))
00707 #define power_lfreceiver_disable()      (PRR1 |= (uint8_t)(1 « PRLFR))
00708 #endif
00709
00710 #if defined(__AVR_HAVE_PRR1_PRLFTP)
00711 #define power_lftp_enable()             (PRR1 &= (uint8_t)~(1 « PRLFTP))
00712 #define power_lftp_disable()            (PRR1 |= (uint8_t)(1 « PRLFTP))
00713 #endif
00714
00715 #if defined(__AVR_HAVE_PRR1_PRSCI)
00716 #define power_sci_enable()              (PRR1 &= (uint8_t)~(1 « PRSCI))
00717 #define power_sci_disable()             (PRR1 |= (uint8_t)(1 « PRSCI))
00718 #endif
00719
00720 #if defined(__AVR_HAVE_PRR1_PRSPI)
00721 #define power_spi_enable()              (PRR1 &= (uint8_t)~(1 « PRSPI))
00722 #define power_spi_disable()             (PRR1 |= (uint8_t)(1 « PRSPI))
00723 #endif
00724
00725 #if defined(__AVR_HAVE_PRR1_PRT1)
00726 #define power_timer1_enable()           (PRR1 &= (uint8_t)~(1 « PRT1))
00727 #define power_timer1_disable()          (PRR1 |= (uint8_t)(1 « PRT1))
00728 #endif
00729
00730 #if defined(__AVR_HAVE_PRR1_PRT2)
00731 #define power_timer2_enable()           (PRR1 &= (uint8_t)~(1 « PRT2))
00732 #define power_timer2_disable()          (PRR1 |= (uint8_t)(1 « PRT2))
00733 #endif
00734
00735 #if defined(__AVR_HAVE_PRR1_PRT3)
00736 #define power_timer3_enable()           (PRR1 &= (uint8_t)~(1 « PRT3))
00737 #define power_timer3_disable()          (PRR1 |= (uint8_t)(1 « PRT3))
00738 #endif
00739
00740 #if defined(__AVR_HAVE_PRR1_PRT4)
00741 #define power_timer4_enable()           (PRR1 &= (uint8_t)~(1 « PRT4))
00742 #define power_timer4_disable()          (PRR1 |= (uint8_t)(1 « PRT4))
00743 #endif
```

```
00744
00745 #if defined(__AVR_HAVE_PRR1_PRT5)
00746 #define power_timer5_enable()           (PRR1 &= (uint8_t)~(1 « PRT5))
00747 #define power_timer5_disable()          (PRR1 |= (uint8_t)(1 « PRT5))
00748 #endif
00749
00750 #if defined(__AVR_HAVE_PRR1_PRTIM3)
00751 #define power_timer3_enable()           (PRR1 &= (uint8_t)~(1 « PRTIM3))
00752 #define power_timer3_disable()          (PRR1 |= (uint8_t)(1 « PRTIM3))
00753 #endif
00754
00755 #if defined(__AVR_HAVE_PRR1_PRTIM4)
00756 #define power_timer4_enable()           (PRR1 &= (uint8_t)~(1 « PRTIM4))
00757 #define power_timer4_disable()          (PRR1 |= (uint8_t)(1 « PRTIM4))
00758 #endif
00759
00760 #if defined(__AVR_HAVE_PRR1_PRTIM5)
00761 #define power_timer5_enable()           (PRR1 &= (uint8_t)~(1 « PRTIM5))
00762 #define power_timer5_disable()          (PRR1 |= (uint8_t)(1 « PRTIM5))
00763 #endif
00764
00765 #if defined(__AVR_HAVE_PRR1_PRTRX24)
00766 #define power_transceiver_enable()      (PRR1 &= (uint8_t)~(1 « PRTRX24))
00767 #define power_transceiver_disable()     (PRR1 |= (uint8_t)(1 « PRTRX24))
00768 #endif
00769
00770 #if defined(__AVR_HAVE_PRR1_PRUSART1)
00771 #define power_usart1_enable()           (PRR1 &= (uint8_t)~(1 « PRUSART1))
00772 #define power_usart1_disable()          (PRR1 |= (uint8_t)(1 « PRUSART1))
00773 #endif
00774
00775 #if defined(__AVR_HAVE_PRR1_PRUSART2)
00776 #define power_usart2_enable()           (PRR1 &= (uint8_t)~(1 « PRUSART2))
00777 #define power_usart2_disable()          (PRR1 |= (uint8_t)(1 « PRUSART2))
00778 #endif
00779
00780 #if defined(__AVR_HAVE_PRR1_PRUSART3)
00781 #define power_usart3_enable()           (PRR1 &= (uint8_t)~(1 « PRUSART3))
00782 #define power_usart3_disable()          (PRR1 |= (uint8_t)(1 « PRUSART3))
00783 #endif
00784
00785 #if defined(__AVR_HAVE_PRR1_PRUSB)
00786 #define power_usb_enable()              (PRR1 &= (uint8_t)~(1 « PRUSB))
00787 #define power_usb_disable()             (PRR1 |= (uint8_t)(1 « PRUSB))
00788 #endif
00789
00790 #if defined(__AVR_HAVE_PRR1_PRUSBH)
00791 #define power_usbh_enable()             (PRR1 &= (uint8_t)~(1 « PRUSBH))
00792 #define power_usbh_disable()            (PRR1 |= (uint8_t)(1 « PRUSBH))
00793 #endif
00794
00795 #if defined(__AVR_HAVE_PRR1_PRSPI1)
00796 #define power_spi1_enable()             (PRR1 &= (uint8_t)~(1 « PRSPI1))
00797 #define power_spi1_disable()            (PRR1 |= (uint8_t)(1 « PRSPI1))
00798 #endif
00799
00800 #if defined(__AVR_HAVE_PRR1_PRPTC)
00801 #define power_ptc_enable()              (PRR1 &= (uint8_t)~(1 « PRPTC))
00802 #define power_ptc_disable()             (PRR1 |= (uint8_t)(1 « PRPTC))
00803 #endif
00804
00805 #if defined(__AVR_HAVE_PRR1_PRTWI1)
00806 #define power_twi1_enable()             (PRR1 &= (uint8_t)~(1 « PRTWI1))
00807 #define power_twi1_disable()            (PRR1 |= (uint8_t)(1 « PRTWI1))
00808 #endif
00809
00810 #if defined(__AVR_HAVE_PRR2_PRDF)
00811 #define power_data_fifo_enable()        (PRR2 &= (uint8_t)~(1 « PRDF))
00812 #define power_data_fifo_disable()       (PRR2 |= (uint8_t)(1 « PRDF))
00813 #endif
00814
00815 #if defined(__AVR_HAVE_PRR2_PRIDS)
00816 #define power_id_scan_enable()          (PRR2 &= (uint8_t)~(1 « PRIDS))
```

```
00817 #define power_id_scan_disable()          (PRR2 |= (uint8_t)(1 « PRIDS))
00818 #endif
00819
00820 #if defined(__AVR_HAVE_PRR2_PRRAM0)
00821 #define power_ram0_enable()              (PRR2 &= (uint8_t)~(1 « PRRAM0))
00822 #define power_ram0_disable()             (PRR2 |= (uint8_t)(1 « PRRAM0))
00823 #endif
00824
00825 #if defined(__AVR_HAVE_PRR2_PRRAM1)
00826 #define power_ram1_enable()              (PRR2 &= (uint8_t)~(1 « PRRAM1))
00827 #define power_ram1_disable()             (PRR2 |= (uint8_t)(1 « PRRAM1))
00828 #endif
00829
00830 #if defined(__AVR_HAVE_PRR2_PRRAM2)
00831 #define power_ram2_enable()              (PRR2 &= (uint8_t)~(1 « PRRAM2))
00832 #define power_ram2_disable()             (PRR2 |= (uint8_t)(1 « PRRAM2))
00833 #endif
00834
00835 #if defined(__AVR_HAVE_PRR2_PRRAM3)
00836 #define power_ram3_enable()              (PRR2 &= (uint8_t)~(1 « PRRAM3))
00837 #define power_ram3_disable()             (PRR2 |= (uint8_t)(1 « PRRAM3))
00838 #endif
00839
00840 #if defined(__AVR_HAVE_PRR2_PRRS)
00841 #define power_rssi_buffer_enable()       (PRR2 &= (uint8_t)~(1 « PRRS))
00842 #define power_rssi_buffer_disable()      (PRR2 |= (uint8_t)(1 « PRRS))
00843 #endif
00844
00845 #if defined(__AVR_HAVE_PRR2_PRSF)
00846 #define power_preamble_rssi_fifo_enable()       (PRR2 &= (uint8_t)~(1 « PRSF))
00847 #define power_preamble_rssi_fifo_disable()      (PRR2 |= (uint8_t)(1 « PRSF))
00848 #endif
00849
00850 #if defined(__AVR_HAVE_PRR2_PRSPI2)
00851 #define power_spi2_enable()              (PRR2 &= (uint8_t)~(1 « PRSPI2))
00852 #define power_spi2_disable()             (PRR2 |= (uint8_t)(1 « PRSPI2))
00853 #endif
00854
00855 #if defined(__AVR_HAVE_PRR2_PRSSM)
00856 #define power_sequencer_state_machine_enable()  (PRR2 &= (uint8_t)~(1 « PRSSM))
00857 #define power_sequencer_state_machine_disable() (PRR2 |= (uint8_t)(1 « PRSSM))
00858 #endif
00859
00860 #if defined(__AVR_HAVE_PRR2_PRTM)
00861 #define power_tx_modulator_enable()      (PRR2 &= (uint8_t)~(1 « PRTM))
00862 #define power_tx_modulator_disable()     (PRR2 |= (uint8_t)(1 « PRTM))
00863 #endif
00864
00865 #if defined(__AVR_HAVE_PRR2_PRTWI2)
00866 #define power_twi2_enable()              (PRR2 &= (uint8_t)~(1 « PRTWI2))
00867 #define power_twi2_disable()             (PRR2 |= (uint8_t)(1 « PRTWI2))
00868 #endif
00869
00870 #if defined(__AVR_HAVE_PRR2_PRXA)
00871 #define power_rx_buffer_A_enable()       (PRR2 &= (uint8_t)~(1 « PRXA))
00872 #define power_rx_buffer_A_disable()      (PRR2 |= (uint8_t)(1 « PRXA))
00873 #endif
00874
00875 #if defined(__AVR_HAVE_PRR2_PRXB)
00876 #define power_rx_buffer_B_enable()       (PRR2 &= (uint8_t)~(1 « PRXB))
00877 #define power_rx_buffer_B_disable()      (PRR2 |= (uint8_t)(1 « PRXB))
00878 #endif
00879
00880 #if defined(__AVR_HAVE_PRGEN_AES)
00881 #define power_aes_enable()               (PR_PRGEN &= (uint8_t)~(PR_AES_bm))
00882 #define power_aes_disable()              (PR_PRGEN |= (uint8_t)PR_AES_bm)
00883 #endif
00884
00885 #if defined(__AVR_HAVE_PRGEN_DMA)
00886 #define power_dma_enable()               (PR_PRGEN &= (uint8_t)~(PR_DMA_bm))
00887 #define power_dma_disable()              (PR_PRGEN |= (uint8_t)PR_DMA_bm)
00888 #endif
00889
```

```
00890 #if defined(__AVR_HAVE_PRGEN_EBI)
00891 #define power_ebi_enable()              (PR_PRGEN &= (uint8_t)~(PR_EBI_bm))
00892 #define power_ebi_disable()             (PR_PRGEN |= (uint8_t)PR_EBI_bm)
00893 #endif
00894
00895 #if defined(__AVR_HAVE_PRGEN_EDMA)
00896 #define power_edma_enable()             (PR_PRGEN &= (uint8_t)~(PR_EDMA_bm))
00897 #define power_edma_disable()            (PR_PRGEN |= (uint8_t)PR_EDMA_bm)
00898 #endif
00899
00900 #if defined(__AVR_HAVE_PRGEN_EVSYS)
00901 #define power_evsys_enable()            (PR_PRGEN &= (uint8_t)~(PR_EVSYS_bm))
00902 #define power_evsys_disable()           (PR_PRGEN |= (uint8_t)PR_EVSYS_bm)
00903 #endif
00904
00905 #if defined(__AVR_HAVE_PRGEN_LCD)
00906 #define power_lcd_enable()              (PR_PRGEN &= (uint8_t)~(PR_LCD_bm))
00907 #define power_lcd_disable()             (PR_PRGEN |= (uint8_t)PR_LCD_bm)
00908 #endif
00909
00910 #if defined(__AVR_HAVE_PRGEN_RTC)
00911 #define power_rtc_enable()              (PR_PRGEN &= (uint8_t)~(PR_RTC_bm))
00912 #define power_rtc_disable()             (PR_PRGEN |= (uint8_t)PR_RTC_bm)
00913 #endif
00914
00915 #if defined(__AVR_HAVE_PRGEN_USB)
00916 #define power_usb_enable()              (PR_PRGEN &= (uint8_t)~(PR_USB_bm))
00917 #define power_usb_disable()             (PR_PRGEN &= (uint8_t)(PR_USB_bm))
00918 #endif
00919
00920 #if defined(__AVR_HAVE_PRGEN_XCL)
00921 #define power_xcl_enable()              (PR_PRGEN &= (uint8_t)~(PR_XCL_bm))
00922 #define power_xcl_disable()             (PR_PRGEN |= (uint8_t)PR_XCL_bm)
00923 #endif
00924
00925 #if defined(__AVR_HAVE_PRPA_AC)
00926 #define power_aca_enable()      (PR_PRPA &= (uint8_t)~(PR_AC_bm))
00927 #define power_aca_disable()     (PR_PRPA |= (uint8_t)PR_AC_bm)
00928 #endif
00929
00930 #if defined(__AVR_HAVE_PRPA_ADC)
00931 #define power_adca_enable()     (PR_PRPA &= (uint8_t)~(PR_ADC_bm))
00932 #define power_adca_disable()    (PR_PRPA |= (uint8_t)PR_ADC_bm)
00933 #endif
00934
00935 #if defined(__AVR_HAVE_PRPA_DAC)
00936 #define power_daca_enable()     (PR_PRPA &= (uint8_t)~(PR_DAC_bm))
00937 #define power_daca_disable()    (PR_PRPA |= (uint8_t)PR_DAC_bm)
00938 #endif
00939
00940 #if defined(__AVR_HAVE_PRPB_AC)
00941 #define power_acb_enable()      (PR_PRPB &= (uint8_t)~(PR_AC_bm))
00942 #define power_acb_disable()     (PR_PRPB |= (uint8_t)PR_AC_bm)
00943 #endif
00944
00945 #if defined(__AVR_HAVE_PRPB_ADC)
00946 #define power_adcb_enable()     (PR_PRPB &= (uint8_t)~(PR_ADC_bm))
00947 #define power_adcb_disable()    (PR_PRPB |= (uint8_t)PR_ADC_bm)
00948 #endif
00949
00950 #if defined(__AVR_HAVE_PRPB_DAC)
00951 #define power_dacb_enable()     (PR_PRPB &= (uint8_t)~(PR_DAC_bm))
00952 #define power_dacb_disable()    (PR_PRPB |= (uint8_t)PR_DAC_bm)
00953 #endif
00954
00955 #if defined(__AVR_HAVE_PRPC_HIRES)
00956 #define power_hiresc_enable()   (PR_PRPC &= (uint8_t)~(PR_HIRES_bm))
00957 #define power_hiresc_disable()  (PR_PRPC |= (uint8_t)PR_HIRES_bm)
00958 #endif
00959
00960 #if defined(__AVR_HAVE_PRPC_SPI)
00961 #define power_spic_enable()     (PR_PRPC &= (uint8_t)~(PR_SPI_bm))
00962 #define power_spic_disable()    (PR_PRPC |= (uint8_t)PR_SPI_bm)
```

```
00963 #endif
00964
00965 #if defined(__AVR_HAVE_PRPC_TC0)
00966 #define power_tc0c_enable()      (PR_PRPC &= (uint8_t)~(PR_TC0_bm))
00967 #define power_tc0c_disable()     (PR_PRPC |= (uint8_t)PR_TC0_bm)
00968 #endif
00969
00970 #if defined(__AVR_HAVE_PRPC_TC1)
00971 #define power_tc1c_enable()      (PR_PRPC &= (uint8_t)~(PR_TC1_bm))
00972 #define power_tc1c_disable()     (PR_PRPC |= (uint8_t)PR_TC1_bm)
00973 #endif
00974
00975 #if defined(__AVR_HAVE_PRPC_TC4)
00976 #define power_tc4c_enable()      (PR_PRPC  &= (uint8_t)~(PR_TC4_bm))
00977 #define power_tc4c_disable()     (PR_PRPC  |= (uint8_t)PR_TC4_bm)
00978 #endif
00979
00980 #if defined(__AVR_HAVE_PRPC_TC5)
00981 #define power_tc5c_enable()      (PR_PRPC  &= (uint8_t)~(PR_TC5_bm))
00982 #define power_tc5c_disable()     (PR_PRPC  |= (uint8_t)PR_TC5_bm)
00983 #endif
00984
00985 #if defined(__AVR_HAVE_PRPC_TWI)
00986 #define power_twic_enable()      (PR_PRPC &= (uint8_t)~(PR_TWI_bm))
00987 #define power_twic_disable()     (PR_PRPC |= (uint8_t)PR_TWI_bm)
00988 #endif
00989
00990 #if defined(__AVR_HAVE_PRPC_USART0)
00991 #define power_usartc0_enable()   (PR_PRPC &= (uint8_t)~(PR_USART0_bm))
00992 #define power_usartc0_disable()  (PR_PRPC |= (uint8_t)PR_USART0_bm)
00993 #endif
00994
00995 #if defined(__AVR_HAVE_PRPC_USART1)
00996 #define power_usartc1_enable()   (PR_PRPC &= (uint8_t)~(PR_USART1_bm))
00997 #define power_usartc1_disable()  (PR_PRPC |= (uint8_t)PR_USART1_bm)
00998 #endif
00999
01000 #if defined(__AVR_HAVE_PRPD_HIRES)
01001 #define power_hiresd_enable()    (PR_PRPD &= (uint8_t)~(PR_HIRES_bm))
01002 #define power_hiresd_disable()   (PR_PRPD |= (uint8_t)PR_HIRES_bm)
01003 #endif
01004
01005 #if defined(__AVR_HAVE_PRPD_SPI)
01006 #define power_spid_enable()      (PR_PRPD &= (uint8_t)~(PR_SPI_bm))
01007 #define power_spid_disable()     (PR_PRPD |= (uint8_t)PR_SPI_bm)
01008 #endif
01009
01010 #if defined(__AVR_HAVE_PRPD_TC0)
01011 #define power_tc0d_enable()      (PR_PRPD &= (uint8_t)~(PR_TC0_bm))
01012 #define power_tc0d_disable()     (PR_PRPD |= (uint8_t)PR_TC0_bm)
01013 #endif
01014
01015 #if defined(__AVR_HAVE_PRPD_TC1)
01016 #define power_tc1d_enable()      (PR_PRPD &= (uint8_t)~(PR_TC1_bm))
01017 #define power_tc1d_disable()     (PR_PRPD |= (uint8_t)PR_TC1_bm)
01018 #endif
01019
01020 #if defined(__AVR_HAVE_PRPD_TC5)
01021 #define power_tc5d_enable()      (PR_PRPD  &= (uint8_t)~(PR_TC5_bm))
01022 #define power_tc5d_disable()     (PR_PRPD  |= (uint8_t)PR_TC5_bm)
01023 #endif
01024
01025 #if defined(__AVR_HAVE_PRPD_TWI)
01026 #define power_twid_enable()      (PR_PRPD &= (uint8_t)~(PR_TWI_bm))
01027 #define power_twid_disable()     (PR_PRPD |= (uint8_t)PR_TWI_bm)
01028 #endif
01029
01030 #if defined(__AVR_HAVE_PRPD_USART0)
01031 #define power_usartd0_enable()   (PR_PRPD &= (uint8_t)~(PR_USART0_bm))
01032 #define power_usartd0_disable()  (PR_PRPD |= (uint8_t)PR_USART0_bm)
01033 #endif
01034
01035 #if defined(__AVR_HAVE_PRPD_USART1)
```

```
01036 #define power_usartd1_enable()  (PR_PRPD &= (uint8_t)~(PR_USART1_bm))
01037 #define power_usartd1_disable() (PR_PRPD |= (uint8_t)PR_USART1_bm)
01038 #endif
01039
01040 #if defined(__AVR_HAVE_PRPE_HIRES)
01041 #define power_hirese_enable()   (PR_PRPE &= (uint8_t)~(PR_HIRES_bm))
01042 #define power_hirese_disable()  (PR_PRPE |= (uint8_t)PR_HIRES_bm)
01043 #endif
01044
01045 #if defined(__AVR_HAVE_PRPE_SPI)
01046 #define power_spie_enable()     (PR_PRPE &= (uint8_t)~(PR_SPI_bm))
01047 #define power_spie_disable()    (PR_PRPE |= (uint8_t)PR_SPI_bm)
01048 #endif
01049
01050 #if defined(__AVR_HAVE_PRPE_TC0)
01051 #define power_tc0e_enable()     (PR_PRPE &= (uint8_t)~(PR_TC0_bm))
01052 #define power_tc0e_disable()    (PR_PRPE |= (uint8_t)PR_TC0_bm)
01053 #endif
01054
01055 #if defined(__AVR_HAVE_PRPE_TC1)
01056 #define power_tc1e_enable()     (PR_PRPE &= (uint8_t)~(PR_TC1_bm))
01057 #define power_tc1e_disable()    (PR_PRPE |= (uint8_t)PR_TC1_bm)
01058 #endif
01059
01060 #if defined(__AVR_HAVE_PRPE_TWI)
01061 #define power_twie_enable()     (PR_PRPE &= (uint8_t)~(PR_TWI_bm))
01062 #define power_twie_disable()    (PR_PRPE |= (uint8_t)PR_TWI_bm)
01063 #endif
01064
01065 #if defined(__AVR_HAVE_PRPE_USART0)
01066 #define power_usarte0_enable()  (PR_PRPE &= (uint8_t)~(PR_USART0_bm))
01067 #define power_usarte0_disable() (PR_PRPE |= (uint8_t)PR_USART0_bm)
01068 #endif
01069
01070 #if defined(__AVR_HAVE_PRPE_USART1)
01071 #define power_usarte1_enable()  (PR_PRPE &= (uint8_t)~(PR_USART1_bm))
01072 #define power_usarte1_disable() (PR_PRPE |= (uint8_t)PR_USART1_bm)
01073 #endif
01074
01075 #if defined(__AVR_HAVE_PRPF_HIRES)
01076 #define power_hiresf_enable()   (PR_PRPF &= (uint8_t)~(PR_HIRES_bm))
01077 #define power_hiresf_disable()  (PR_PRPF |= (uint8_t)PR_HIRES_bm)
01078 #endif
01079
01080 #if defined(__AVR_HAVE_PRPF_SPI)
01081 #define power_spif_enable()     (PR_PRPF &= (uint8_t)~(PR_SPI_bm))
01082 #define power_spif_disable()    (PR_PRPF |= (uint8_t)PR_SPI_bm)
01083 #endif
01084
01085 #if defined(__AVR_HAVE_PRPF_TC0)
01086 #define power_tc0f_enable()     (PR_PRPF &= (uint8_t)~(PR_TC0_bm))
01087 #define power_tc0f_disable()    (PR_PRPF |= (uint8_t)PR_TC0_bm)
01088 #endif
01089
01090 #if defined(__AVR_HAVE_PRPF_TC1)
01091 #define power_tc1f_enable()     (PR_PRPF &= (uint8_t)~(PR_TC1_bm))
01092 #define power_tc1f_disable()    (PR_PRPF |= (uint8_t)PR_TC1_bm)
01093 #endif
01094
01095 #if defined(__AVR_HAVE_PRPF_TWI)
01096 #define power_twif_enable()     (PR_PRPF &= (uint8_t)~(PR_TWI_bm))
01097 #define power_twif_disable()    (PR_PRPF |= (uint8_t)PR_TWI_bm)
01098 #endif
01099
01100 #if defined(__AVR_HAVE_PRPF_USART0)
01101 #define power_usartf0_enable()  (PR_PRPF &= (uint8_t)~(PR_USART0_bm))
01102 #define power_usartf0_disable() (PR_PRPF |= (uint8_t)PR_USART0_bm)
01103 #endif
01104
01105 #if defined(__AVR_HAVE_PRPF_USART1)
01106 #define power_usartf1_enable()  (PR_PRPF &= (uint8_t)~(PR_USART1_bm))
01107 #define power_usartf1_disable() (PR_PRPF |= (uint8_t)PR_USART1_bm)
01108 #endif
```

```
01109
01110 #ifdef __DOXYGEN__
01111 /**
01112    \ingroup avr_power
01113    \fn void power_all_enable()
01114    Enable all modules.
01115  */
01116 static __ATTR_ALWAYS_INLINE__ void power_all_enable();
01117 #else
01118 static __ATTR_ALWAYS_INLINE__ void __power_all_enable()
01119 {
01120 #ifdef __AVR_HAVE_PRR
01121     PRR &= (uint8_t)~(__AVR_HAVE_PRR);
01122 #endif
01123
01124 #ifdef __AVR_HAVE_PRR0
01125     PRR0 &= (uint8_t)~(__AVR_HAVE_PRR0);
01126 #endif
01127
01128 #ifdef __AVR_HAVE_PRR1
01129     PRR1 &= (uint8_t)~(__AVR_HAVE_PRR1);
01130 #endif
01131
01132 #ifdef __AVR_HAVE_PRR2
01133    PRR2 &= (uint8_t)~(__AVR_HAVE_PRR2);
01134 #endif
01135
01136 #ifdef __AVR_HAVE_PRGEN
01137     PR_PRGEN &= (uint8_t)~(__AVR_HAVE_PRGEN);
01138 #endif
01139
01140 #ifdef __AVR_HAVE_PRPA
01141     PR_PRPA &= (uint8_t)~(__AVR_HAVE_PRPA);
01142 #endif
01143
01144 #ifdef __AVR_HAVE_PRPB
01145     PR_PRPB &= (uint8_t)~(__AVR_HAVE_PRPB);
01146 #endif
01147
01148 #ifdef __AVR_HAVE_PRPC
01149     PR_PRPC &= (uint8_t)~(__AVR_HAVE_PRPC);
01150 #endif
01151
01152 #ifdef __AVR_HAVE_PRPD
01153     PR_PRPD &= (uint8_t)~(__AVR_HAVE_PRPD);
01154 #endif
01155
01156 #ifdef __AVR_HAVE_PRPE
01157     PR_PRPE &= (uint8_t)~(__AVR_HAVE_PRPE);
01158 #endif
01159
01160 #ifdef __AVR_HAVE_PRPF
01161     PR_PRPF &= (uint8_t)~(__AVR_HAVE_PRPF);
01162 #endif
01163 }
01164 #endif /* __DOXYGEN__ */
01165
01166 #ifdef __DOXYGEN__
01167 /**
01168    \ingroup avr_power
01169    \fn void power_all_disable()
01170    Disable all modules.
01171  */
01172 static __ATTR_ALWAYS_INLINE__ void power_all_disable();
01173 #else
01174 static __ATTR_ALWAYS_INLINE__ void __power_all_disable()
01175 {
01176 #ifdef __AVR_HAVE_PRR
01177     PRR |= (uint8_t)(__AVR_HAVE_PRR);
01178 #endif
01179
01180 #ifdef __AVR_HAVE_PRR0
01181     PRR0 |= (uint8_t)(__AVR_HAVE_PRR0);
```

```
01182 #endif
01183
01184 #ifdef __AVR_HAVE_PRR1
01185     PRR1 |= (uint8_t)(__AVR_HAVE_PRR1);
01186 #endif
01187
01188 #ifdef __AVR_HAVE_PRR2
01189     PRR2 |= (uint8_t)(__AVR_HAVE_PRR2);
01190 #endif
01191
01192 #ifdef __AVR_HAVE_PRGEN
01193     PR_PRGEN |= (uint8_t)(__AVR_HAVE_PRGEN);
01194 #endif
01195
01196 #ifdef __AVR_HAVE_PRPA
01197     PR_PRPA |= (uint8_t)(__AVR_HAVE_PRPA);
01198 #endif
01199
01200 #ifdef __AVR_HAVE_PRPB
01201     PR_PRPB |= (uint8_t)(__AVR_HAVE_PRPB);
01202 #endif
01203
01204 #ifdef __AVR_HAVE_PRPC
01205     PR_PRPC |= (uint8_t)(__AVR_HAVE_PRPC);
01206 #endif
01207
01208 #ifdef __AVR_HAVE_PRPD
01209     PR_PRPD |= (uint8_t)(__AVR_HAVE_PRPD);
01210 #endif
01211
01212 #ifdef __AVR_HAVE_PRPE
01213     PR_PRPE |= (uint8_t)(__AVR_HAVE_PRPE);
01214 #endif
01215
01216 #ifdef __AVR_HAVE_PRPF
01217     PR_PRPF |= (uint8_t)(__AVR_HAVE_PRPF);
01218 #endif
01219 }
01220 #endif /* __DOXYGEN__ */
01221
01222 #ifndef __DOXYGEN__
01223 #ifndef power_all_enable
01224 #define power_all_enable() __power_all_enable()
01225 #endif
01226
01227 #ifndef power_all_disable
01228 #define power_all_disable() __power_all_disable()
01229 #endif
01230 #endif  /* !__DOXYGEN__ */
01231
01232
01233 #if defined(__DOXYGEN__) \
01234 || defined(__AVR_AT90CAN32__) \
01235 || defined(__AVR_AT90CAN64__) \
01236 || defined(__AVR_AT90CAN128__) \
01237 || defined(__AVR_AT90PWM1__) \
01238 || defined(__AVR_AT90PWM2__) \
01239 || defined(__AVR_AT90PWM2B__) \
01240 || defined(__AVR_AT90PWM3__) \
01241 || defined(__AVR_AT90PWM3B__) \
01242 || defined(__AVR_AT90PWM81__) \
01243 || defined(__AVR_AT90PWM161__) \
01244 || defined(__AVR_AT90PWM216__) \
01245 || defined(__AVR_AT90PWM316__) \
01246 || defined(__AVR_AT90SCR100__) \
01247 || defined(__AVR_AT90USB646__) \
01248 || defined(__AVR_AT90USB647__) \
01249 || defined(__AVR_AT90USB82__) \
01250 || defined(__AVR_AT90USB1286__) \
01251 || defined(__AVR_AT90USB1287__) \
01252 || defined(__AVR_AT90USB162__) \
01253 || defined(__AVR_ATA5505__) \
01254 || defined(__AVR_ATA5272__) \
```

```
01255 || defined(__AVR_ATmega1280__) \
01256 || defined(__AVR_ATmega1281__) \
01257 || defined(__AVR_ATmega1284__) \
01258 || defined(__AVR_ATmega128RFA1__) \
01259 || defined(__AVR_ATmega1284RFR2__) \
01260 || defined(__AVR_ATmega128RFR2__) \
01261 || defined(__AVR_ATmega1284P__) \
01262 || defined(__AVR_ATmega162__) \
01263 || defined(__AVR_ATmega164A__) \
01264 || defined(__AVR_ATmega164P__) \
01265 || defined(__AVR_ATmega164PA__) \
01266 || defined(__AVR_ATmega165__) \
01267 || defined(__AVR_ATmega165A__) \
01268 || defined(__AVR_ATmega165P__) \
01269 || defined(__AVR_ATmega165PA__) \
01270 || defined(__AVR_ATmega168__) \
01271 || defined(__AVR_ATmega168P__) \
01272 || defined(__AVR_ATmega168A__) \
01273 || defined(__AVR_ATmega168PA__) \
01274 || defined(__AVR_ATmega168PB__) \
01275 || defined(__AVR_ATmega169__) \
01276 || defined(__AVR_ATmega169A__) \
01277 || defined(__AVR_ATmega169P__) \
01278 || defined(__AVR_ATmega169PA__) \
01279 || defined(__AVR_ATmega16M1__) \
01280 || defined(__AVR_ATmega16U2__) \
01281 || defined(__AVR_ATmega16U4__) \
01282 || defined(__AVR_ATmega2560__) \
01283 || defined(__AVR_ATmega2561__) \
01284 || defined(__AVR_ATmega2564RFR2__) \
01285 || defined(__AVR_ATmega256RFR2__) \
01286 || defined(__AVR_ATmega324A__) \
01287 || defined(__AVR_ATmega324P__) \
01288 || defined(__AVR_ATmega324PA__) \
01289 || defined(__AVR_ATmega324PB__) \
01290 || defined(__AVR_ATmega325__) \
01291 || defined(__AVR_ATmega325A__) \
01292 || defined(__AVR_ATmega325PA__) \
01293 || defined(__AVR_ATmega3250__) \
01294 || defined(__AVR_ATmega3250A__) \
01295 || defined(__AVR_ATmega3250PA__) \
01296 || defined(__AVR_ATmega328__) \
01297 || defined(__AVR_ATmega328P__) \
01298 || defined(__AVR_ATmega328PB__) \
01299 || defined(__AVR_ATmega329__) \
01300 || defined(__AVR_ATmega329A__) \
01301 || defined(__AVR_ATmega329P__) \
01302 || defined(__AVR_ATmega329PA__) \
01303 || defined(__AVR_ATmega3290__) \
01304 || defined(__AVR_ATmega3290A__) \
01305 || defined(__AVR_ATmega3290P__) \
01306 || defined(__AVR_ATmega3290PA__) \
01307 || defined(__AVR_ATmega32C1__) \
01308 || defined(__AVR_ATmega32M1__) \
01309 || defined(__AVR_ATmega32U2__) \
01310 || defined(__AVR_ATmega32U4__) \
01311 || defined(__AVR_ATmega32U6__) \
01312 || defined(__AVR_ATmega48__) \
01313 || defined(__AVR_ATmega48A__) \
01314 || defined(__AVR_ATmega48PA__) \
01315 || defined(__AVR_ATmega48P__) \
01316 || defined(__AVR_ATmega640__) \
01317 || defined(__AVR_ATmega649P__) \
01318 || defined(__AVR_ATmega644__) \
01319 || defined(__AVR_ATmega644A__) \
01320 || defined(__AVR_ATmega644P__) \
01321 || defined(__AVR_ATmega644PA__) \
01322 || defined(__AVR_ATmega645__) \
01323 || defined(__AVR_ATmega645A__) \
01324 || defined(__AVR_ATmega645P__) \
01325 || defined(__AVR_ATmega6450__) \
01326 || defined(__AVR_ATmega6450A__) \
01327 || defined(__AVR_ATmega6450P__) \
```

```
01328 || defined(__AVR_ATmega649__) \
01329 || defined(__AVR_ATmega649A__) \
01330 || defined(__AVR_ATmega64M1__) \
01331 || defined(__AVR_ATmega64C1__) \
01332 || defined(__AVR_ATmega6490__) \
01333 || defined(__AVR_ATmega6490A__) \
01334 || defined(__AVR_ATmega6490P__) \
01335 || defined(__AVR_ATmega644RFR2__) \
01336 || defined(__AVR_ATmega64RFR2__) \
01337 || defined(__AVR_ATmega88__) \
01338 || defined(__AVR_ATmega88A__) \
01339 || defined(__AVR_ATmega88P__) \
01340 || defined(__AVR_ATmega88PA__) \
01341 || defined(__AVR_ATmega8U2__) \
01342 || defined(__AVR_ATmega16U2__) \
01343 || defined(__AVR_ATmega32U2__) \
01344 || defined(__AVR_ATtiny48__) \
01345 || defined(__AVR_ATtiny88__) \
01346 || defined(__AVR_ATtiny87__) \
01347 || defined(__AVR_ATtiny167__)
01348
01349
01350 /** \addtogroup avr_power
01351
01352 Some of the newer AVRs contain a System Clock Prescale Register (CLKPR) that
01353 allows you to decrease the system clock frequency and the power consumption
01354 when the need for processing power is low.
01355 On some earlier AVRs (ATmega103, ATmega64, ATmega128), similar
01356 functionality can be achieved through the XTAL Divide Control Register.
01357 Below are two macros and an enumerated type that can be used to
01358 interface to the Clock Prescale Register or
01359 XTAL Divide Control Register.
01360
01361 \note Not all AVR devices have a clock prescaler. On those devices
01362 without a Clock Prescale Register or XTAL Divide Control Register, these
01363 macros are not available.
01364
01365 \code
01366 typedef enum
01367 {
01368     clock_div_1 = 0,
01369     clock_div_2 = 1,
01370     clock_div_4 = 2,
01371     clock_div_8 = 3,
01372     clock_div_16 = 4,
01373     clock_div_32 = 5,
01374     clock_div_64 = 6,
01375     clock_div_128 = 7,
01376     clock_div_256 = 8,
01377     clock_div_1_rc = 15, // ATmega128RFA1 only
01378 } clock_div_t;
01379 \endcode
01380 Clock prescaler setting enumerations for device using
01381 System Clock Prescale Register.
01382
01383 \code
01384 typedef enum
01385 {
01386     clock_div_1 = 1,
01387     clock_div_2 = 2,
01388     clock_div_4 = 4,
01389     clock_div_8 = 8,
01390     clock_div_16 = 16,
01391     clock_div_32 = 32,
01392     clock_div_64 = 64,
01393     clock_div_128 = 128
01394 } clock_div_t;
01395 \endcode
01396 Clock prescaler setting enumerations for device using
01397 XTAL Divide Control Register.
01398
01399 */
01400 #ifndef __DOXYGEN__
```

```
01401 typedef enum
01402 {
01403     clock_div_1 = 0,
01404     clock_div_2 = 1,
01405     clock_div_4 = 2,
01406     clock_div_8 = 3,
01407     clock_div_16 = 4,
01408     clock_div_32 = 5,
01409     clock_div_64 = 6,
01410     clock_div_128 = 7,
01411     clock_div_256 = 8
01412 #if defined(__AVR_ATmega128RFA1__) \
01413 || defined(__AVR_ATmega2564RFR2__) \
01414 || defined(__AVR_ATmega1284RFR2__) \
01415 || defined(__AVR_ATmega644RFR2__) \
01416 || defined(__AVR_ATmega256RFR2__) \
01417 || defined(__AVR_ATmega128RFR2__) \
01418 || defined(__AVR_ATmega64RFR2__)
01419     , clock_div_1_rc = 15
01420 #endif
01421 } clock_div_t;
01422
01423 static __ATTR_ALWAYS_INLINE__ void clock_prescale_set(clock_div_t);
01424 #endif  /* !__DOXYGEN__ */
01425
01426 /**
01427    \ingroup avr_power
01428    \fn clock_prescale_set(clock_div_t x)
01429
01430 Set the clock prescaler register select bits, selecting a system clock
01431 division setting. This function is inlined, even if compiler
01432 optimizations are disabled.
01433
01434 The type of \c x is \c clock_div_t.
01435
01436 \note For device with XTAL Divide Control Register (XDIV), \c x can actually range
01437 from 1 to 129. Thus, one does not need to use \c clock_div_t type as argument.
01438 */
01439 void clock_prescale_set(clock_div_t __x)
01440 {
01441     uint8_t __tmp = _BV(CLKPCE);
01442     __asm__ __volatile__ (
01443         "in __tmp_reg__,__SREG__" "\n\t"
01444         "cli"                     "\n\t"
01445         "sts %1, %0"              "\n\t"
01446         "sts %1, %2"              "\n\t"
01447         "out __SREG__, __tmp_reg__"
01448         : /* no outputs */
01449         : "d" (__tmp),
01450           "M" (_SFR_MEM_ADDR(CLKPR)),
01451           "d" ((uint8_t) __x)
01452         : "r0", "memory");
01453 }
01454
01455 /** \ingroup avr_power
01456 \def clock_prescale_get()
01457 Gets and returns the clock prescaler register setting. The return type is \c
      clock_div_t.
01458
01459 \note For device with XTAL Divide Control Register (XDIV), return can actually
01460 range from 1 to 129. Care should be taken has the return value could differ from the
01461 typedef enum clock_div_t. This should only happen if clock_prescale_set was previously
01462 called with a value other than those defined by \c clock_div_t.
01463 */
01464 #define clock_prescale_get()  (clock_div_t)(CLKPR &
      (uint8_t)((1«CLKPS0)|(1«CLKPS1)|(1«CLKPS2)|(1«CLKPS3)))
01465
01466 #elif defined(__AVR_ATmega16HVB__) \
01467 || defined(__AVR_ATmega16HVBREVB__) \
01468 || defined(__AVR_ATmega32HVB__) \
01469 || defined(__AVR_ATmega32HVBREVB__)
01470
01471 typedef enum
```

```
01472 {
01473     clock_div_1 = 0,
01474     clock_div_2 = 1,
01475     clock_div_4 = 2,
01476     clock_div_8 = 3
01477 } clock_div_t;
01478
01479 static __ATTR_ALWAYS_INLINE__ void clock_prescale_set(clock_div_t);
01480
01481 void clock_prescale_set(clock_div_t __x)
01482 {
01483     uint8_t __tmp = _BV(CLKPCE);
01484     __asm__ __volatile__ (
01485         "in __tmp_reg__,__SREG__" "\n\t"
01486         "cli"                     "\n\t"
01487         "sts %1, %0"              "\n\t"
01488         "sts %1, %2"              "\n\t"
01489         "out __SREG__, __tmp_reg__"
01490         : /* no outputs */
01491         : "d" (__tmp),
01492           "M" (_SFR_MEM_ADDR(CLKPR)),
01493           "d" ((uint8_t) __x)
01494         : "r0", "memory");
01495 }
01496
01497 #define clock_prescale_get()  (clock_div_t)(CLKPR & (uint8_t)((1«CLKPS0)|(1«CLKPS1)))
01498
01499 #elif defined(__AVR_ATA5790__) \
01500 || defined (__AVR_ATA5795__)
01501
01502 typedef enum
01503 {
01504     clock_div_1 = 0,
01505     clock_div_2 = 1,
01506     clock_div_4 = 2,
01507     clock_div_8 = 3,
01508     clock_div_16 = 4,
01509     clock_div_32 = 5,
01510     clock_div_64 = 6,
01511     clock_div_128 = 7,
01512 } clock_div_t;
01513
01514 static __ATTR_ALWAYS_INLINE__ void system_clock_prescale_set(clock_div_t);
01515
01516 void system_clock_prescale_set(clock_div_t __x)
01517 {
01518     uint8_t __tmp = _BV(CLKPCE);
01519     __asm__ __volatile__ (
01520         "in __tmp_reg__,__SREG__" "\n\t"
01521         "cli"                     "\n\t"
01522         "out %1, %0"              "\n\t"
01523         "out %1, %2"              "\n\t"
01524         "out __SREG__, __tmp_reg__"
01525         : /* no outputs */
01526         : "d" (__tmp),
01527           "I" (_SFR_IO_ADDR(CLKPR)),
01528           "d" ((uint8_t) __x)
01529         : "r0", "memory");
01530 }
01531
01532 #define system_clock_prescale_get()  (clock_div_t)(CLKPR &
      (uint8_t)((1«CLKPS0)|(1«CLKPS1)|(1«CLKPS2)))
01533
01534 typedef enum
01535 {
01536     timer_clock_div_reset = 0,
01537     timer_clock_div_1 = 1,
01538     timer_clock_div_2 = 2,
01539     timer_clock_div_4 = 3,
01540     timer_clock_div_8 = 4,
01541     timer_clock_div_16 = 5,
01542     timer_clock_div_32 = 6,
01543     timer_clock_div_64 = 7
```

```
01544 } timer_clock_div_t;
01545
01546 static __ATTR_ALWAYS_INLINE__ void timer_clock_prescale_set(timer_clock_div_t);
01547
01548 void timer_clock_prescale_set(timer_clock_div_t __x)
01549 {
01550     uint8_t __t;
01551     __asm__ __volatile__ (
01552         "in __tmp_reg__,__SREG__"    "\n\t"
01553         "cli"                        "\n\t"
01554         "in %[temp],%[clkpr]"        "\n\t"
01555         "out %[clkpr],%[enable]"     "\n\t"
01556         "cbr %[temp],%[not_CLTPS]"   "\n\t"
01557         "or %[temp], %[set_value]"   "\n\t"
01558         "out %[clkpr],%[temp]"       "\n\t"
01559         "out __SREG__,__tmp_reg__"
01560         : [temp] "=d" (__t)
01561         : [clkpr] "I" (_SFR_IO_ADDR(CLKPR)),
01562           [enable] "r" ((uint8_t) _BV(CLKPCE)),
01563           [not_CLTPS] "M" ((1 « CLTPS2) | (1 « CLTPS1) | (1 « CLTPS0)),
01564           [set_value] "r" ((uint8_t) ((__x & 7) « 3))
01565         : "r0", "memory");
01566 }
01567
01568 #define timer_clock_prescale_get()  (timer_clock_div_t)(CLKPR &
      (uint8_t)((1«CLTPS0)|(1«CLTPS1)|(1«CLTPS2)))
01569
01570 #elif defined(__AVR_ATA6285__) \
01571 || defined(__AVR_ATA6286__)
01572
01573 typedef enum
01574 {
01575     clock_div_1 = 0,
01576     clock_div_2 = 1,
01577     clock_div_4 = 2,
01578     clock_div_8 = 3,
01579     clock_div_16 = 4,
01580     clock_div_32 = 5,
01581     clock_div_64 = 6,
01582     clock_div_128 = 7
01583 } clock_div_t;
01584
01585 static __ATTR_ALWAYS_INLINE__ void system_clock_prescale_set(clock_div_t);
01586
01587 void system_clock_prescale_set(clock_div_t __x)
01588 {
01589     uint8_t __t;
01590     __asm__ __volatile__ (
01591         "in __tmp_reg__,__SREG__"    "\n\t"
01592         "cli"                        "\n\t"
01593         "in %[temp],%[clpr]"         "\n\t"
01594         "out %[clpr],%[enable]"      "\n\t"
01595         "cbr %[temp],%[not_CLKPS]"   "\n\t"
01596         "or %[temp], %[set_value]"   "\n\t"
01597         "out %[clpr],%[temp]"        "\n\t"
01598         "out __SREG__,__tmp_reg__"
01599         : [temp] "=d" (__t)
01600         : [clpr] "I" (_SFR_IO_ADDR(CLKPR)),
01601           [enable] "r" ((uint8_t) (1 « CLPCE)),
01602           [not_CLKPS] "M" ((1 « CLKPS2) | (1 « CLKPS1) | (1 « CLKPS0)),
01603           [set_value] "r" ((uint8_t) (__x & 7))
01604         : "r0", "memory");
01605 }
01606
01607 #define system_clock_prescale_get()  (clock_div_t)(CLKPR &
      (uint8_t)((1«CLKPS0)|(1«CLKPS1)|(1«CLKPS2)))
01608
01609 typedef enum
01610 {
01611     timer_clock_div_reset = 0,
01612     timer_clock_div_1 = 1,
01613     timer_clock_div_2 = 2,
01614     timer_clock_div_4 = 3,
```

```
01615     timer_clock_div_8 = 4,
01616     timer_clock_div_16 = 5,
01617     timer_clock_div_32 = 6,
01618     timer_clock_div_64 = 7
01619 } timer_clock_div_t;
01620
01621 static __ATTR_ALWAYS_INLINE__ void timer_clock_prescale_set(timer_clock_div_t);
01622
01623 void timer_clock_prescale_set(timer_clock_div_t __x)
01624 {
01625     uint8_t __t;
01626     __asm__ __volatile__ (
01627         "in __tmp_reg__,__SREG__"   "\n\t"
01628         "cli"                       "\n\t"
01629         "in %[temp],%[clpr]"        "\n\t"
01630         "out %[clpr],%[enable]"     "\n\t"
01631         "cbr %[temp],%[not_CLTPS]"  "\n\t"
01632         "or %[temp], %[set_value]"  "\n\t"
01633         "out %[clpr],%[temp]"       "\n\t"
01634         "out __SREG__,__tmp_reg__"
01635         : [temp] "=d" (__t)
01636         : [clpr] "I" (_SFR_IO_ADDR(CLKPR)),
01637           [enable] "r" ((uint8_t) (1 « CLPCE)),
01638           [not_CLTPS] "M" ((1 « CLTPS2) | (1 « CLTPS1) | (1 « CLTPS0)),
01639           [set_value] "r" ((uint8_t) ((__x & 7) « 3))
01640         : "r0", "memory");
01641 }
01642
01643 #define timer_clock_prescale_get()  (timer_clock_div_t)(CLKPR &
      (uint8_t)((1«CLTPS0)|(1«CLTPS1)|(1«CLTPS2)))
01644
01645 #elif defined(__AVR_ATtiny24__) \
01646 || defined(__AVR_ATtiny24A__) \
01647 || defined(__AVR_ATtiny44__) \
01648 || defined(__AVR_ATtiny44A__) \
01649 || defined(__AVR_ATtiny84__) \
01650 || defined(__AVR_ATtiny84A__) \
01651 || defined(__AVR_ATtiny25__) \
01652 || defined(__AVR_ATtiny45__) \
01653 || defined(__AVR_ATtiny85__) \
01654 || defined(__AVR_ATtiny261A__) \
01655 || defined(__AVR_ATtiny261__) \
01656 || defined(__AVR_ATtiny461__) \
01657 || defined(__AVR_ATtiny461A__) \
01658 || defined(__AVR_ATtiny861__) \
01659 || defined(__AVR_ATtiny861A__) \
01660 || defined(__AVR_ATtiny2313__) \
01661 || defined(__AVR_ATtiny2313A__) \
01662 || defined(__AVR_ATtiny4313__) \
01663 || defined(__AVR_ATtiny13__) \
01664 || defined(__AVR_ATtiny13A__) \
01665 || defined(__AVR_ATtiny43U__) \
01666
01667 typedef enum
01668 {
01669     clock_div_1 = 0,
01670     clock_div_2 = 1,
01671     clock_div_4 = 2,
01672     clock_div_8 = 3,
01673     clock_div_16 = 4,
01674     clock_div_32 = 5,
01675     clock_div_64 = 6,
01676     clock_div_128 = 7,
01677     clock_div_256 = 8
01678 } clock_div_t;
01679
01680 static __ATTR_ALWAYS_INLINE__ void clock_prescale_set(clock_div_t);
01681
01682 void clock_prescale_set(clock_div_t __x)
01683 {
01684     __asm__ __volatile__ (
01685         "in __tmp_reg__,__SREG__" "\n\t"
01686         "cli"                     "\n\t"
```

```
01687         "out %1, %0"              "\n\t"
01688         "out %1, %2"              "\n\t"
01689         "out __SREG__, __tmp_reg__"
01690         : /* no outputs */
01691         : "d" ((uint8_t) (1 « CLKPCE)),
01692           "I" (_SFR_IO_ADDR(CLKPR)),
01693           "d" ((uint8_t) __x)
01694         : "r0", "memory");
01695 }
01696
01697
01698 #define clock_prescale_get()  (clock_div_t)(CLKPR &
      (uint8_t)((1«CLKPS0)|(1«CLKPS1)|(1«CLKPS2)|(1«CLKPS3)))
01699
01700 #elif defined(__AVR_ATtiny441__) \
01701 || defined(__AVR_ATtiny841__)
01702
01703 typedef enum
01704 {
01705     clock_div_1 = 0,
01706     clock_div_2 = 1,
01707     clock_div_4 = 2,
01708     clock_div_8 = 3,
01709     clock_div_16 = 4,
01710     clock_div_32 = 5,
01711     clock_div_64 = 6,
01712     clock_div_128 = 7,
01713     clock_div_256 = 8
01714 } clock_div_t;
01715
01716 static __ATTR_ALWAYS_INLINE__ void clock_prescale_set (clock_div_t);
01717
01718 void clock_prescale_set (clock_div_t __x)
01719 {
01720     __asm__ __volatile__ (
01721         "in __tmp_reg__,__SREG__" "\n\t"
01722         "cli"                     "\n\t"
01723         "sts %2, %3"              "\n\t"
01724         "sts %1, %0"              "\n\t"
01725         "out __SREG__, __tmp_reg__"
01726         : /* no outputs */
01727         : "r" ((uint8_t) __x),
01728           "n" (_SFR_MEM_ADDR(CLKPR)),
01729           "n" (_SFR_MEM_ADDR(CCP)),
01730           "r" ((uint8_t) 0xD8)
01731         : "r0", "memory");
01732 }
01733
01734 #define clock_prescale_get()  (clock_div_t) (CLKPR &
      (uint8_t)((1«CLKPS0)|(1«CLKPS1)|(1«CLKPS2)|(1«CLKPS3)))
01735
01736 #elif defined(__AVR_ATmega64__) \
01737 || defined(__AVR_ATmega103__) \
01738 || defined(__AVR_ATmega128__)
01739
01740 //Enum is declared for code compatibility
01741 typedef enum
01742 {
01743     clock_div_1 = 1,
01744     clock_div_2 = 2,
01745     clock_div_4 = 4,
01746     clock_div_8 = 8,
01747     clock_div_16 = 16,
01748     clock_div_32 = 32,
01749     clock_div_64 = 64,
01750     clock_div_128 = 128
01751 } clock_div_t;
01752
01753 static __ATTR_ALWAYS_INLINE__ void clock_prescale_set(clock_div_t);
01754
01755 void clock_prescale_set(clock_div_t __x)
01756 {
01757     if ((uint8_t) __x <= 0 || (uint8_t) __x > 129)
```

```
01758     {
01759         return;//Invalid value.
01760     }
01761     else
01762     {
01763         uint8_t __tmp = 0;
01764         //Algo explained:
01765         //1 - Clear XDIV in order for it to accept a new value (actually only
01766         //    XDIVEN need to be cleared, but clearing XDIV is faster than
01767         //    read-modify-write since we will rewrite XDIV later anyway)
01768         //2 - wait 8 clock cycle for stability, see datasheet errata
01769         //3 - Exit if requested prescaler is 1
01770         //4 - Calculate XDIV6..0 value = 129 - __x
01771         //5 - Set XDIVEN bit in calculated value
01772         //6 - write XDIV with calculated value
01773         //7 - wait 8 clock cycle for stability, see datasheet errata
01774         __asm__ __volatile__ (
01775             "in __tmp_reg__,__SREG__" "\n\t"
01776             "cli"                     "\n\t"
01777             "out %2, __zero_reg__"    "\n\t"
01778             "nop" "\n\t"
01779             "nop" "\n\t"
01780             "nop" "\n\t"
01781             "nop" "\n\t"
01782             "nop" "\n\t"
01783             "nop" "\n\t"
01784             "nop" "\n\t"
01785             "nop" "\n\t"
01786             "cpi %1, 0x01" "\n\t"
01787             "breq L_%="    "\n\t"
01788             "ldi %0, 0x81" "\n\t" //129
01789             "sub %0, %1"   "\n\t"
01790             "ori %0, 0x80" "\n\t" //128
01791             "out %2, %0"   "\n\t"
01792             "nop" "\n\t"
01793             "nop" "\n\t"
01794             "nop" "\n\t"
01795             "nop" "\n\t"
01796             "nop" "\n\t"
01797             "nop" "\n\t"
01798             "nop" "\n\t"
01799             "nop" "\n\t"
01800             "L_%=: " "out __SREG__, __tmp_reg__"
01801             : "=d" (__tmp)
01802             : "d" ((uint8_t) __x),
01803               "I" (_SFR_IO_ADDR(XDIV))
01804             : "r0", "memory");
01805     }
01806 }
01807
01808 static __ATTR_ALWAYS_INLINE__ clock_div_t clock_prescale_get(void);
01809
01810 clock_div_t clock_prescale_get(void)
01811 {
01812     if (bit_is_clear(XDIV, XDIVEN))
01813     {
01814         return (clock_div_t) 1;
01815     }
01816     else
01817     {
01818         return (clock_div_t) (129 - (XDIV & 0x7F));
01819     }
01820 }
01821
01822 #elif defined(__AVR_ATtiny4__) \
01823 || defined(__AVR_ATtiny5__) \
01824 || defined(__AVR_ATtiny9__) \
01825 || defined(__AVR_ATtiny10__) \
01826 || defined(__AVR_ATtiny102__) \
01827 || defined(__AVR_ATtiny104__) \
01828 || defined(__AVR_ATtiny20__) \
01829 || defined(__AVR_ATtiny40__) \
01830
```

```
01831 typedef enum
01832 {
01833     clock_div_1 = 0,
01834     clock_div_2 = 1,
01835     clock_div_4 = 2,
01836     clock_div_8 = 3,
01837     clock_div_16 = 4,
01838     clock_div_32 = 5,
01839     clock_div_64 = 6,
01840     clock_div_128 = 7,
01841     clock_div_256 = 8
01842 } clock_div_t;
01843
01844 static __ATTR_ALWAYS_INLINE__ void clock_prescale_set(clock_div_t);
01845
01846 void clock_prescale_set(clock_div_t __x)
01847 {
01848     __asm__ __volatile__ (
01849         "in __tmp_reg__,__SREG__" "\n\t"
01850         "cli"                     "\n\t"
01851         "out %1, %0"              "\n\t"
01852         "out %2, %3"              "\n\t"
01853         "out __SREG__, __tmp_reg__"
01854         : /* no outputs */
01855         : "d" ((uint8_t) 0xD8),
01856           "I" (_SFR_IO_ADDR(CCP)),
01857           "I" (_SFR_IO_ADDR(CLKPSR)),
01858           "d" ((uint8_t) __x)
01859         : "r16", "memory");
01860 }
01861
01862 #define clock_prescale_get()  (clock_div_t)(CLKPSR &
      (uint8_t)((1«CLKPS0)|(1«CLKPS1)|(1«CLKPS2)|(1«CLKPS3)))
01863
01864 #endif
01865
01866 #endif /* _AVR_POWER_H_ */
```

22.39 sfr_defs.h

```
00001 /* Copyright (c) 2002, Marek Michalkiewicz <marekm@amelek.gda.pl>
00002    All rights reserved.
00003
00004    Redistribution and use in source and binary forms, with or without
00005    modification, are permitted provided that the following conditions are met:
00006
00007    * Redistributions of source code must retain the above copyright
00008      notice, this list of conditions and the following disclaimer.
00009
00010    * Redistributions in binary form must reproduce the above copyright
00011      notice, this list of conditions and the following disclaimer in
00012      the documentation and/or other materials provided with the
00013      distribution.
00014
00015    * Neither the name of the copyright holders nor the names of
00016      contributors may be used to endorse or promote products derived
00017      from this software without specific prior written permission.
00018
00019    THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
00020    AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
00021    IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
00022    ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
00023    LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
00024    CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
00025    SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
00026    INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
00027    CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
00028    ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
00029    POSSIBILITY OF SUCH DAMAGE.  */
00030
00031 /* avr/sfr_defs.h - macros for accessing AVR special function registers */
```

```
00032
00033 #ifndef _AVR_SFR_DEFS_H_
00034 #define _AVR_SFR_DEFS_H_ 1
00035
00036 /** \defgroup avr_sfr_notes Additional notes from <avr/sfr_defs.h>
00037     \ingroup avr_sfr
00038
00039    The \c <avr/sfr_defs.h> file is included by all of the \c <avr/ioXXXX.h>
00040    files, which use macros defined here to make the special function register
00041    definitions look like C variables or simple constants, depending on the
00042    <tt>_SFR_ASM_COMPAT</tt> define.  Some examples from \c <avr/iocanxx.h> to
00043    show how to define such macros:
00044
00045 \code
00046 #define PORTA   _SFR_IO8(0x02)
00047 #define EEAR    _SFR_IO16(0x21)
00048 #define UDR0    _SFR_MEM8(0xC6)
00049 #define TCNT3   _SFR_MEM16(0x94)
00050 #define CANIDT  _SFR_MEM32(0xF0)
00051 \endcode
00052
00053    If \c _SFR_ASM_COMPAT is not defined, C programs can use names like
00054    <tt>PORTA</tt> directly in C expressions (also on the left side of
00055    assignment operators) and GCC will do the right thing (use short I/O
00056    instructions if possible).  The \c __SFR_OFFSET definition is not used in
00057    any way in this case.
00058
00059    Define \c _SFR_ASM_COMPAT as 1 to make these names work as simple constants
00060    (addresses of the I/O registers).  This is necessary when included in
00061    preprocessed assembler (*.S) source files, so it is done automatically if
00062    \c __ASSEMBLER__ is defined.  By default, all addresses are defined as if
00063    they were memory addresses (used in \c lds/sts instructions).  To use these
00064    addresses in \c in/out instructions, you must subtract 0x20 from them.
00065
00066    For more backwards compatibility, insert the following at the start of your
00067    old assembler source file:
00068
00069 \code
00070 #define __SFR_OFFSET 0
00071 \endcode
00072
00073    This automatically subtracts 0x20 from I/O space addresses, but it's a
00074    hack, so it is recommended to change your source: wrap such addresses in
00075    macros defined here, as shown below.  After this is done, the
00076    <tt>__SFR_OFFSET</tt> definition is no longer necessary and can be removed.
00077
00078    Real example - this code could be used in a boot loader that is portable
00079    between devices with \c SPMCR at different addresses.
00080
00081 \verbatim
00082 <avr/iom163.h>: #define SPMCR _SFR_IO8(0x37)
00083 <avr/iom128.h>: #define SPMCR _SFR_MEM8(0x68)
00084 \endverbatim
00085
00086 \code
00087 #if _SFR_IO_REG_P(SPMCR)
00088     out _SFR_IO_ADDR(SPMCR), r24
00089 #else
00090     sts _SFR_MEM_ADDR(SPMCR), r24
00091 #endif
00092 \endcode
00093
00094    You can use the \c in/out/cbi/sbi/sbic/sbis instructions, without the
00095    <tt>_SFR_IO_REG_P</tt> test, if you know that the register is in the I/O
00096    space (as with \c SREG, for example).  If it isn't, the assembler will
00097    complain (I/O address out of range 0...0x3f), so this should be fairly
00098    safe.
00099
00100    If you do not define \c __SFR_OFFSET (so it will be 0x20 by default), all
00101    special register addresses are defined as memory addresses (so \c SREG is
00102    0x5f), and (if code size and speed are not important, and you don't like
00103    the ugly \#if above) you can always use lds/sts to access them.  But, this
00104    will not work if <tt>__SFR_OFFSET</tt> != 0x20, so use a different macro
```

```
00105    (defined only if <tt>__SFR_OFFSET</tt> == 0x20) for safety:
00106
00107 \code
00108     sts _SFR_ADDR(SPMCR), r24
00109 \endcode
00110
00111    In C programs, all 3 combinations of \c _SFR_ASM_COMPAT and
00112    <tt>__SFR_OFFSET</tt> are supported - the \c _SFR_ADDR(SPMCR) macro can be
00113    used to get the address of the \c SPMCR register (0x57 or 0x68 depending on
00114    device). */
00115
00116 #ifdef __ASSEMBLER__
00117 #define _SFR_ASM_COMPAT 1
00118 #elif !defined(_SFR_ASM_COMPAT)
00119 #define _SFR_ASM_COMPAT 0
00120 #endif
00121
00122 #ifndef __ASSEMBLER__
00123 /* These only work in C programs.  */
00124 #include <inttypes.h>
00125
00126 #define _MMIO_BYTE(mem_addr) (*(volatile uint8_t *)(mem_addr))
00127 #define _MMIO_WORD(mem_addr) (*(volatile uint16_t *)(mem_addr))
00128 #define _MMIO_DWORD(mem_addr) (*(volatile uint32_t *)(mem_addr))
00129 #endif
00130
00131 #if _SFR_ASM_COMPAT
00132
00133 #ifndef __SFR_OFFSET
00134 /* Define as 0 before including this file for compatibility with old asm
00135    sources that don't subtract __SFR_OFFSET from symbolic I/O addresses.  */
00136 #  if __AVR_ARCH__ >= 100
00137 #    define __SFR_OFFSET 0x00
00138 #  else
00139 #    define __SFR_OFFSET 0x20
00140 #  endif
00141 #endif
00142
00143 #if (__SFR_OFFSET != 0) && (__SFR_OFFSET != 0x20)
00144 #error "__SFR_OFFSET must be 0 or 0x20"
00145 #endif
00146
00147 #define _SFR_MEM8(mem_addr) (mem_addr)
00148 #define _SFR_MEM16(mem_addr) (mem_addr)
00149 #define _SFR_MEM32(mem_addr) (mem_addr)
00150 #define _SFR_IO8(io_addr) ((io_addr) + __SFR_OFFSET)
00151 #define _SFR_IO16(io_addr) ((io_addr) + __SFR_OFFSET)
00152
00153 #define _SFR_IO_ADDR(sfr) ((sfr) - __SFR_OFFSET)
00154 #define _SFR_MEM_ADDR(sfr) (sfr)
00155 #define _SFR_IO_REG_P(sfr) ((sfr) < 0x40 + __SFR_OFFSET)
00156
00157 #if (__SFR_OFFSET == 0x20)
00158 /* No need to use ?: operator, so works in assembler too.  */
00159 #define _SFR_ADDR(sfr) _SFR_MEM_ADDR(sfr)
00160 #elif !defined(__ASSEMBLER__)
00161 #define _SFR_ADDR(sfr) (_SFR_IO_REG_P(sfr) ? (_SFR_IO_ADDR(sfr) + 0x20) :
      _SFR_MEM_ADDR(sfr))
00162 #endif
00163
00164 #else  /* !_SFR_ASM_COMPAT */
00165
00166 #ifndef __SFR_OFFSET
00167 #  if __AVR_ARCH__ >= 100
00168 #    define __SFR_OFFSET 0x00
00169 #  else
00170 #    define __SFR_OFFSET 0x20
00171 #  endif
00172 #endif
00173
00174 #define _SFR_MEM8(mem_addr) _MMIO_BYTE(mem_addr)
00175 #define _SFR_MEM16(mem_addr) _MMIO_WORD(mem_addr)
00176 #define _SFR_MEM32(mem_addr) _MMIO_DWORD(mem_addr)
```

```
00177 #define _SFR_IO8(io_addr) _MMIO_BYTE((io_addr) + __SFR_OFFSET)
00178 #define _SFR_IO16(io_addr) _MMIO_WORD((io_addr) + __SFR_OFFSET)
00179
00180 #define _SFR_MEM_ADDR(sfr) ((uint16_t) &(sfr))
00181 #define _SFR_IO_ADDR(sfr) (_SFR_MEM_ADDR(sfr) - __SFR_OFFSET)
00182 #define _SFR_IO_REG_P(sfr) (_SFR_MEM_ADDR(sfr) < 0x40 + __SFR_OFFSET)
00183
00184 #define _SFR_ADDR(sfr) _SFR_MEM_ADDR(sfr)
00185
00186 #endif /* !_SFR_ASM_COMPAT */
00187
00188 #define _SFR_BYTE(sfr) _MMIO_BYTE(_SFR_ADDR(sfr))
00189 #define _SFR_WORD(sfr) _MMIO_WORD(_SFR_ADDR(sfr))
00190 #define _SFR_DWORD(sfr) _MMIO_DWORD(_SFR_ADDR(sfr))
00191
00192 /** \name Bit manipulation */
00193
00194 /**@{*/
00195 /** \def _BV
00196     \ingroup avr_sfr
00197
00198     \code #include <avr/io.h>\endcode
00199
00200     Converts a bit number into a byte value.
00201
00202     \note The bit shift is performed by the compiler which then inserts the
00203     result into the code. Thus, there is no run-time overhead when using
00204     _BV(). */
00205
00206 #define _BV(bit) (1 << (bit))
00207
00208 /**@}*/
00209
00210 #ifndef _VECTOR
00211 #define _VECTOR(N) __vector_ ## N
00212 #endif
00213
00214 #ifndef __ASSEMBLER__
00215
00216
00217 /** \name IO register bit manipulation */
00218
00219 /**@{*/
00220
00221
00222
00223 /** \def bit_is_set
00224     \ingroup avr_sfr
00225
00226     \code #include <avr/io.h>\endcode
00227
00228     Test whether bit \c bit in IO register \c sfr is set.
00229     This will return a 0 if the bit is clear, and non-zero
00230     if the bit is set. */
00231
00232 #define bit_is_set(sfr, bit) (_SFR_BYTE(sfr) & _BV(bit))
00233
00234 /** \def bit_is_clear
00235     \ingroup avr_sfr
00236
00237     \code #include <avr/io.h>\endcode
00238
00239     Test whether bit \c bit in IO register \c sfr is clear.
00240     This will return non-zero if the bit is clear, and a 0
00241     if the bit is set. */
00242
00243 #define bit_is_clear(sfr, bit) (!(_SFR_BYTE(sfr) & _BV(bit)))
00244
00245 /** \def loop_until_bit_is_set
00246     \ingroup avr_sfr
00247
00248     \code #include <avr/io.h>\endcode
00249
```

```
00250     Wait until bit \c bit in IO register \c sfr is set. */
00251
00252 #define loop_until_bit_is_set(sfr, bit) do { } while (bit_is_clear(sfr, bit))
00253
00254 /** \def loop_until_bit_is_clear
00255     \ingroup avr_sfr
00256
00257     \code #include <avr/io.h>\endcode
00258
00259     Wait until bit \c bit in IO register \c sfr is clear. */
00260
00261 #define loop_until_bit_is_clear(sfr, bit) do { } while (bit_is_set(sfr, bit))
00262
00263 /**@}*/
00264
00265 #endif /* !__ASSEMBLER__ */
00266
00267 #endif  /* _SFR_DEFS_H_ */
```

22.40 signal.h

```
00001 /* Copyright (c) 2002,2005,2006 Marek Michalkiewicz
00002    All rights reserved.
00003
00004    Redistribution and use in source and binary forms, with or without
00005    modification, are permitted provided that the following conditions are met:
00006
00007    * Redistributions of source code must retain the above copyright
00008      notice, this list of conditions and the following disclaimer.
00009
00010    * Redistributions in binary form must reproduce the above copyright
00011      notice, this list of conditions and the following disclaimer in
00012      the documentation and/or other materials provided with the
00013      distribution.
00014
00015    * Neither the name of the copyright holders nor the names of
00016      contributors may be used to endorse or promote products derived
00017      from this software without specific prior written permission.
00018
00019   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
00020   AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
00021   IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
00022   ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
00023   LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
00024   CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
00025   SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
00026   INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
00027   CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
00028   ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
00029   POSSIBILITY OF SUCH DAMAGE. */
00030
00031 #ifndef _AVR_SIGNAL_H_
00032 #define _AVR_SIGNAL_H_
00033
00034 #warning "This header file is obsolete.  Use <avr/interrupt.h>."
00035 #include <avr/interrupt.h>
00036
00037 #endif /* _AVR_SIGNAL_H_ */
```

22.41 signature.h File Reference

22.42 signature.h

Go to the documentation of this file.

```
00001 /* Copyright (c) 2009, Atmel Corporation
00002    All rights reserved.
```

```
00003
00004    Redistribution and use in source and binary forms, with or without
00005    modification, are permitted provided that the following conditions are met:
00006
00007    * Redistributions of source code must retain the above copyright
00008      notice, this list of conditions and the following disclaimer.
00009
00010    * Redistributions in binary form must reproduce the above copyright
00011      notice, this list of conditions and the following disclaimer in
00012      the documentation and/or other materials provided with the
00013      distribution.
00014
00015    * Neither the name of the copyright holders nor the names of
00016      contributors may be used to endorse or promote products derived
00017      from this software without specific prior written permission.
00018
00019   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
00020   AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
00021   IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
00022   ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
00023   LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
00024   CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
00025   SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
00026   INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
00027   CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
00028   ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
00029   POSSIBILITY OF SUCH DAMAGE. */
00030
00031 /* avr/signature.h - Signature API */
00032
00033 #ifndef _AVR_SIGNATURE_H_
00034 #define _AVR_SIGNATURE_H_ 1
00035
00036 /** \file */
00037 /** \defgroup avr_signature <avr/signature.h>: Signature Support
00038
00039     \par Introduction
00040
00041     The <avr/signature.h> header file allows the user to automatically
00042     and easily include the device's signature data in a special section of
00043     the final linked ELF file.
00044
00045     This value can then be used by programming software to compare the on-device
00046     signature with the signature recorded in the ELF file to look for a match
00047     before programming the device.
00048
00049     \par API Usage Example
00050
00051     Usage is very simple; just include the header file:
00052
00053     \code
00054     #include <avr/signature.h>
00055     \endcode
00056
00057     This will declare a constant unsigned char array and it is initialized with
00058     the three signature bytes, MSB first, that are defined in the device I/O
00059     header file. This array is then placed in the .signature section in the
00060     resulting linked ELF file.
00061
00062     The three signature bytes that are used to initialize the array are
00063     these defined macros in the device I/O header file, from MSB to LSB:
00064     SIGNATURE_2, SIGNATURE_1, SIGNATURE_0.
00065
00066     This header file should only be included once in an application.
00067 */
00068
00069 #ifndef __ASSEMBLER__
00070
00071 #include <avr/io.h>
00072
00073 #if defined(SIGNATURE_0) && defined(SIGNATURE_1) && defined(SIGNATURE_2)
00074
00075 const unsigned char __signature[3]
```

```
00076 __attribute__((__used__, __section__(".signature"))) =
00077         { SIGNATURE_2, SIGNATURE_1, SIGNATURE_0 };
00078
00079 #endif  /* defined(SIGNATURE_0) && defined(SIGNATURE_1) && defined(SIGNATURE_2) */
00080
00081 #endif  /* __ASSEMBLER__ */
00082
00083 #endif  /* _AVR_SIGNATURE_H_ */
```

22.43 sleep.h File Reference

Functions

- void set_sleep_mode (uint8_t mode)
- void sleep_enable (void)
- void sleep_disable (void)
- void sleep_cpu (void)
- void sleep_mode (void)
- void sleep_bod_disable (void)

22.44 sleep.h

Go to the documentation of this file.

```
00001 /* Copyright (c) 2002, 2004 Theodore A. Roth
00002    Copyright (c) 2004, 2007, 2008 Eric B. Weddington
00003    Copyright (c) 2005, 2006, 2007 Joerg Wunsch
00004    All rights reserved.
00005
00006    Redistribution and use in source and binary forms, with or without
00007    modification, are permitted provided that the following conditions are met:
00008
00009    * Redistributions of source code must retain the above copyright
00010      notice, this list of conditions and the following disclaimer.
00011
00012    * Redistributions in binary form must reproduce the above copyright
00013      notice, this list of conditions and the following disclaimer in
00014      the documentation and/or other materials provided with the
00015      distribution.
00016
00017    * Neither the name of the copyright holders nor the names of
00018      contributors may be used to endorse or promote products derived
00019      from this software without specific prior written permission.
00020
00021    THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
00022    AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
00023    IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
00024    ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
00025    LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
00026    CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
00027    SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
00028    INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
00029    CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
00030    ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
00031    POSSIBILITY OF SUCH DAMAGE. */
00032
00033 #ifndef _AVR_SLEEP_H_
00034 #define _AVR_SLEEP_H_ 1
00035
00036 #include <avr/io.h>
00037 #include <stdint.h>
00038
00039
00040 /** \file */
00041 /** \defgroup avr_sleep <avr/sleep.h>: Power Management and Sleep Modes
```

```
00042     \code #include <avr/sleep.h>\endcode
00043
00044     Use of the \c SLEEP instruction can allow an application to reduce its
00045     power consumption considerably. AVR devices can be put into different
00046     sleep modes. Refer to the datasheet for the details relating to the device
00047     you are using.
00048
00049     There are several macros provided in this header file to actually
00050     put the device into sleep mode.  The simplest way is to optionally
00051     set the desired sleep mode using \c set_sleep_mode() (it usually
00052     defaults to idle mode where the CPU is put on sleep but all
00053     peripheral clocks are still running), and then call
00054     \c sleep_mode(). This macro automatically sets the sleep enable bit, goes
00055     to sleep, and clears the sleep enable bit.
00056
00057     Example:
00058     \code
00059     #include <avr/sleep.h>
00060
00061     ...
00062       set_sleep_mode(<mode>);
00063       sleep_mode();
00064     \endcode
00065
00066     Note that unless your purpose is to completely lock the CPU (until a
00067     hardware reset), interrupts need to be enabled before going to sleep.
00068
00069     As the \c sleep_mode() macro might cause race conditions in some
00070     situations, the individual steps of manipulating the sleep enable
00071     (SE) bit, and actually issuing the \c SLEEP instruction, are provided
00072     in the macros \c sleep_enable(), \c sleep_disable(), and
00073     \c sleep_cpu().  This also allows for test-and-sleep scenarios that
00074     take care of not missing the interrupt that will awake the device
00075     from sleep.
00076
00077     Example:
00078     \code
00079     #include <avr/interrupt.h>
00080     #include <avr/sleep.h>
00081
00082     ...
00083       set_sleep_mode(<mode>);
00084       cli();
00085       if (some_condition)
00086       {
00087         sleep_enable();
00088         sei();
00089         sleep_cpu();
00090         sleep_disable();
00091       }
00092       sei();
00093     \endcode
00094
00095     This sequence ensures an atomic test of \c some_condition with
00096     interrupts being disabled.  If the condition is met, sleep mode
00097     will be prepared, and the \c SLEEP instruction will be scheduled
00098     immediately after an \c SEI instruction.  As the instruction right
00099     after the \c SEI is guaranteed to be executed before an interrupt
00100     could trigger, it is sure the device will really be put to sleep.
00101
00102     Some devices have the ability to disable the Brown Out Detector (BOD) before
00103     going to sleep. This will also reduce power while sleeping. If the
00104     specific AVR device has this ability then an additional macro is defined:
00105     \c sleep_bod_disable(). This macro generates inlined assembly code
00106     that will correctly implement the timed sequence for disabling the BOD
00107     before sleeping. However, there is a limited number of cycles after the
00108     BOD has been disabled that the device can be put into sleep mode, otherwise
00109     the BOD will not truly be disabled. Recommended practice is to disable
00110     the BOD (\c sleep_bod_disable()), set the interrupts (\c sei()), and then
00111     put the device to sleep (\c sleep_cpu()), like so:
00112
00113     \code
00114     #include <avr/interrupt.h>
```

```
00115     #include <avr/sleep.h>
00116
00117     ...
00118       set_sleep_mode(<mode>);
00119       cli();
00120       if (some_condition)
00121       {
00122         sleep_enable();
00123         sleep_bod_disable();
00124         sei();
00125         sleep_cpu();
00126         sleep_disable();
00127       }
00128       sei();
00129     \endcode
00130 */
00131
00132
00133 /* Define an internal sleep control register and an internal sleep enable bit mask. */
00134 #if defined(SLEEP_CTRL)
00135
00136     /* XMEGA devices */
00137     #define _SLEEP_CONTROL_REG  SLEEP_CTRL
00138     #define _SLEEP_ENABLE_MASK  SLEEP_SEN_bm
00139     #define _SLEEP_SMODE_GROUP_MASK  SLEEP_SMODE_gm
00140
00141 #elif defined(SLPCTRL)
00142
00143     /* New xmega devices */
00144     #define _SLEEP_CONTROL_REG  SLPCTRL_CTRLA
00145     #define _SLEEP_ENABLE_MASK  SLPCTRL_SEN_bm
00146     #define _SLEEP_SMODE_GROUP_MASK  SLPCTRL_SMODE_gm
00147
00148 #elif defined(SMCR)
00149
00150     #define _SLEEP_CONTROL_REG  SMCR
00151     #define _SLEEP_ENABLE_MASK  _BV(SE)
00152
00153 #elif defined(__AVR_AT94K__)
00154
00155     #define _SLEEP_CONTROL_REG  MCUR
00156     #define _SLEEP_ENABLE_MASK  _BV(SE)
00157
00158 #elif !defined(__DOXYGEN__)
00159
00160     #define _SLEEP_CONTROL_REG  MCUCR
00161     #define _SLEEP_ENABLE_MASK  _BV(SE)
00162
00163 #endif
00164
00165 #ifdef __DOXYGEN__
00166 /** \ingroup avr_sleep
00167     Set the sleep mode control register to the specified sleep mode \a mode.
00168     The name of the sleep mode register depends on the device, see
00169     the data sheet for details.
00170     \param mode The sleep mode to be set.  The available sleep modes like
00171     \c SLEEP_MODE_PWR_SAVE, \c SLEEP_MODE_PWR_DOWN, \c SLEEP_MODE_PWR_OFF etc.
00172     depend on the device and are defined in avr/io.h.
00173 */
00174 void set_sleep_mode (uint8_t mode);
00175 #endif /* Doxygen */
00176
00177 /* Special casing these three devices - they are the
00178    only ones that need to write to more than one register. */
00179 #if defined(__AVR_ATmega161__)
00180
00181     #define set_sleep_mode(mode) \
00182     do { \
00183         MCUCR = ((MCUCR & ~_BV(SM1)) | ((mode) == SLEEP_MODE_PWR_DOWN || (mode) ==
      SLEEP_MODE_PWR_SAVE ? _BV(SM1) : 0)); \
00184         EMCUCR = ((EMCUCR & ~_BV(SM0)) | ((mode) == SLEEP_MODE_PWR_SAVE ? _BV(SM0) :
      0)); \
00185     } while(0)
```

```
00186
00187
00188 #elif defined(__AVR_ATmega162__) \
00189 || defined(__AVR_ATmega8515__)
00190
00191     #define set_sleep_mode(mode) \
00192     do { \
00193         MCUCR = ((MCUCR & ~_BV(SM1)) | ((mode) == SLEEP_MODE_IDLE ? 0 : _BV(SM1))); \
00194         MCUCSR = ((MCUCSR & ~_BV(SM2)) | ((mode) == SLEEP_MODE_STANDBY  || (mode) ==
      SLEEP_MODE_EXT_STANDBY ? _BV(SM2) : 0)); \
00195         EMCUCR = ((EMCUCR & ~_BV(SM0)) | ((mode) == SLEEP_MODE_PWR_SAVE || (mode) ==
      SLEEP_MODE_EXT_STANDBY ? _BV(SM0) : 0)); \
00196     } while(0)
00197
00198 /* For xmegas, check presence of SLEEP_SMODE<n>_bm and define set_sleep_mode
      accordingly. */
00199 #elif defined(__AVR_XMEGA__)
00200
00201 #define set_sleep_mode(mode) \
00202   do { \
00203     _SLEEP_CONTROL_REG = ((_SLEEP_CONTROL_REG & ~(_SLEEP_SMODE_GROUP_MASK)) | (mode)); \
00204   } while(0)
00205
00206 /* For everything else, check for presence of SM<n> and define set_sleep_mode
      accordingly. */
00207 #else
00208 #if defined(SM2)
00209
00210     #define set_sleep_mode(mode) \
00211     do { \
00212         _SLEEP_CONTROL_REG = ((_SLEEP_CONTROL_REG & ~(_BV(SM0) | _BV(SM1) | _BV(SM2))) |
      (mode)); \
00213     } while(0)
00214
00215 #elif defined(SM1)
00216
00217     #define set_sleep_mode(mode) \
00218     do { \
00219         _SLEEP_CONTROL_REG = ((_SLEEP_CONTROL_REG & ~(_BV(SM0) | _BV(SM1))) | (mode)); \
00220     } while(0)
00221
00222 #elif defined(SM)
00223
00224     #define set_sleep_mode(mode) \
00225     do { \
00226         _SLEEP_CONTROL_REG = ((_SLEEP_CONTROL_REG & ~_BV(SM)) | (mode)); \
00227     } while(0)
00228
00229 #else
00230
00231     #error "No SLEEP mode defined for this device."
00232
00233 #endif /* if defined(SM2) */
00234 #endif /* #if defined(__AVR_ATmega161__) */
00235
00236
00237
00238 /** \ingroup avr_sleep
00239
00240     Put the device in sleep mode. How the device is brought out of sleep mode
00241     depends on the specific mode selected with the set_sleep_mode() function.
00242     See the data sheet for your device for more details. */
00243
00244 #if defined(__DOXYGEN__)
00245
00246 /** \ingroup avr_sleep
00247
00248     Set the SE (sleep enable) bit.
00249 */
00250 extern void sleep_enable (void);
00251
00252 #else
00253
```

```
00254 #define sleep_enable()                  \
00255 do {                                    \
00256   _SLEEP_CONTROL_REG |= (uint8_t)_SLEEP_ENABLE_MASK;    \
00257 } while(0)
00258
00259 #endif
00260
00261
00262 #if defined(__DOXYGEN__)
00263
00264 /** \ingroup avr_sleep
00265
00266     Clear the SE (sleep enable) bit.
00267 */
00268 extern void sleep_disable (void);
00269
00270 #else
00271
00272 #define sleep_disable()                 \
00273 do {                                    \
00274   _SLEEP_CONTROL_REG &= (uint8_t)(~_SLEEP_ENABLE_MASK);  \
00275 } while(0)
00276
00277 #endif
00278
00279
00280 /** \ingroup avr_sleep
00281
00282     Put the device into sleep mode.  The SE bit must be set
00283     beforehand, and it is recommended to clear it afterwards.
00284 */
00285 #if defined(__DOXYGEN__)
00286
00287 extern void sleep_cpu (void);
00288
00289 #else
00290
00291 #define sleep_cpu()                                 \
00292 do {                                                \
00293   __asm__ __volatile__ ( "sleep" ::: "memory" ); \
00294 } while(0)
00295
00296 #endif
00297
00298
00299 #if defined(__DOXYGEN__)
00300
00301 /** \ingroup avr_sleep
00302
00303     Put the device into sleep mode, taking care of setting
00304     the SE bit before, and clearing it afterwards.
00305     All this command does is to run
00306     \code
00307     sleep_enable()
00308     sleep_cpu()
00309     sleep_disable()
00310     \endcode  */
00311 extern void sleep_mode (void);
00312
00313 #else
00314
00315 #define sleep_mode() \
00316 do {                 \
00317     sleep_enable();  \
00318     sleep_cpu();     \
00319     sleep_disable(); \
00320 } while (0)
00321
00322 #endif
00323
00324
00325 #if defined(__DOXYGEN__)
00326
```

```
00327 /** \ingroup avr_sleep
00328
00329     Disable BOD before going to sleep.
00330     Not available on all devices.
00331 */
00332 extern void sleep_bod_disable (void);
00333
00334 #else
00335
00336 #if defined(BODS) && defined(BODSE)
00337
00338 #ifdef BODCR
00339
00340 #define BOD_CONTROL_REG BODCR
00341
00342 #else
00343
00344 #define BOD_CONTROL_REG MCUCR
00345
00346 #endif
00347
00348 #define sleep_bod_disable() \
00349 do { \
00350   uint8_t tempreg; \
00351   __asm__ __volatile__("in %[tempreg], %[mcucr]" "\n\t" \
00352                        "ori %[tempreg], %[bods_bodse]" "\n\t" \
00353                        "out %[mcucr], %[tempreg]" "\n\t" \
00354                        "andi %[tempreg], %[not_bodse]" "\n\t" \
00355                        "out %[mcucr], %[tempreg]" \
00356                        : [tempreg] "=&d" (tempreg) \
00357                        : [mcucr] "I" _SFR_IO_ADDR(BOD_CONTROL_REG), \
00358                          [bods_bodse] "i" (_BV(BODS) | _BV(BODSE)), \
00359                          [not_bodse] "i" (~_BV(BODSE)) \
00360                        : "memory"); \
00361 } while (0)
00362
00363 #endif
00364
00365 #endif
00366
00367 #endif /* _AVR_SLEEP_H_ */
```

22.45 wdt.h File Reference

Macros

- #define wdt_reset() __asm__ __volatile__ ("wdr")
- #define wdt_enable(timeout)
- #define WDTO_8MS -1
- #define WDTO_15MS 0
- #define WDTO_30MS 1
- #define WDTO_60MS 2
- #define WDTO_120MS 3
- #define WDTO_250MS 4
- #define WDTO_500MS 5
- #define WDTO_1S 6
- #define WDTO_2S 7
- #define WDTO_4S 8
- #define WDTO_8S 9

Functions

- static void wdt_enable (const uint8_t value)
- static void wdt_disable (void)

22.46 wdt.h

Go to the documentation of this file.

```
00001 /* Copyright (c) 2002, 2004 Marek Michalkiewicz
00002    Copyright (c) 2005, 2006, 2007 Eric B. Weddington
00003    All rights reserved.
00004
00005    Redistribution and use in source and binary forms, with or without
00006    modification, are permitted provided that the following conditions are met:
00007
00008    * Redistributions of source code must retain the above copyright
00009      notice, this list of conditions and the following disclaimer.
00010
00011    * Redistributions in binary form must reproduce the above copyright
00012      notice, this list of conditions and the following disclaimer in
00013      the documentation and/or other materials provided with the
00014      distribution.
00015
00016    * Neither the name of the copyright holders nor the names of
00017      contributors may be used to endorse or promote products derived
00018      from this software without specific prior written permission.
00019
00020   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
00021   AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
00022   IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
00023   ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
00024   LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
00025   CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
00026   SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
00027   INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
00028   CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
00029   ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
00030   POSSIBILITY OF SUCH DAMAGE. */
00031
00032 /*
00033   avr/wdt.h - macros for AVR watchdog timer
00034 */
00035
00036 #ifndef _AVR_WDT_H_
00037 #define _AVR_WDT_H_
00038
00039 #include <avr/io.h>
00040 #include <stdint.h>
00041
00042 /** \file */
00043 /** \defgroup avr_watchdog <avr/wdt.h>: Watchdog timer handling
00044     \code #include <avr/wdt.h> \endcode
00045
00046     This header file declares the interface to some inline macros
00047     handling the watchdog timer present in many AVR devices.  In order
00048     to prevent the watchdog timer configuration from being
00049     accidentally altered by a crashing application, a special timed
00050     sequence is required in order to change it.  The macros within
00051     this header file handle the required sequence automatically
00052     before changing any value.  Interrupts will be disabled during
00053     the manipulation.
00054
00055     \note Depending on the fuse configuration of the particular
00056     device, further restrictions might apply, in particular it might
00057     be disallowed to turn off the watchdog timer.
00058
00059     Note that for newer devices (ATmega88 and newer, effectively any
00060     AVR that has the option to also generate interrupts), the watchdog
00061     timer remains active even after a system reset (except a power-on
00062     condition), using the fastest prescaler value (approximately 15
00063     ms).  It is therefore required to turn off the watchdog early
00064     during program startup, the datasheet recommends a sequence like
00065     the following:
00066
00067     \code
00068     #include <stdint.h>
00069     #include <avr/wdt.h>
```

```
00070
00071     uint8_t mcusr_mirror __attribute__((section (".noinit")));
00072
00073     __attribute__((used, unused, naked, section(".init3")))
00074     static void get_mcusr (void)
00075     {
00076         mcusr_mirror = MCUSR;
00077         MCUSR = 0;
00078         wdt_disable();
00079     }
00080     \endcode
00081
00082     Saving the value of MCUSR in \c mcusr_mirror is only needed if the
00083     application later wants to examine the reset source, but in particular,
00084     clearing the watchdog reset flag before disabling the
00085     watchdog is required, according to the datasheet.
00086 */
00087
00088 /**
00089    \ingroup avr_watchdog
00090    Reset the watchdog timer.  When the watchdog timer is enabled,
00091    a call to this instruction is required before the timer expires,
00092    otherwise a watchdog-initiated device reset will occur. */
00093 #define wdt_reset() __asm__ __volatile__ ("wdr")
00094
00095 #ifndef __DOXYGEN__
00096
00097 #include <bits/attribs.h>
00098
00099 #if defined(WDP3)
00100 # define _WD_PS3_MASK       _BV(WDP3)
00101 #else
00102 # define _WD_PS3_MASK       0x00
00103 #endif
00104
00105 #if defined(WDTCSR)
00106 #  define _WD_CONTROL_REG     WDTCSR
00107 #elif defined(WDTCR)
00108 #  define _WD_CONTROL_REG     WDTCR
00109 #else
00110 #  define _WD_CONTROL_REG     WDT
00111 #endif
00112
00113 #if defined(WDTOE)
00114 #define _WD_CHANGE_BIT      WDTOE
00115 #else
00116 #define _WD_CHANGE_BIT      WDCE
00117 #endif
00118
00119 #endif  /* !__DOXYGEN__ */
00120
00121 #ifdef __DOXYGEN__
00122 /**
00123    \ingroup avr_watchdog
00124    Enable the watchdog timer, configuring it for expiry after
00125    \c timeout (which is a combination of the \c WDP0 through
00126    \c WDP2 bits to write into the \c WDTCR register; For those devices
00127    that have a \c WDTCSR register, it uses the combination of the \c WDP0
00128    through \c WDP3 bits).
00129
00130    See also the symbolic constants \c WDTO_15MS et al. */
00131 #define wdt_enable(timeout)
00132 #endif  /* __DOXYGEN__ */
00133
00134
00135 #if defined(__AVR_XMEGA__)
00136
00137 #if defined (WDT_CTRLA) && !defined(RAMPD)
00138
00139 #define wdt_enable(timeout)                                             \
00140     do {                                                                \
00141         uint8_t __temp;                                                 \
00142         __asm__ __volatile__ (                                          \
```

```
00143             "wdr"                                    "\n\t"              \
00144             "out %i[ccp_reg], %[ioreg_cen_mask]"     "\n\t"              \
00145             "lds %[tmp], %[wdt_reg]"                 "\n\t"              \
00146             "sbr %[tmp], %[wdt_enable_timeout]"      "\n\t"              \
00147             "sts %[wdt_reg], %[tmp]"                 "\n\t"              \
00148             "1:lds %[tmp], %[wdt_status_reg]"        "\n\t"              \
00149             "sbrc %[tmp], %[wdt_syncbusy_bit]"       "\n\t"              \
00150             "rjmp 1b"                                                    \
00151             : [tmp]                 "=d" (__temp)                        \
00152             : [ccp_reg]             "n"  (& CCP),                        \
00153               [ioreg_cen_mask]      "r"  ((uint8_t)CCP_IOREG_gc),        \
00154               [wdt_reg]             "n"  (& WDT_CTRLA),                  \
00155               [wdt_enable_timeout]  "M"  (timeout),                      \
00156               [wdt_status_reg]      "n"  (& WDT_STATUS),                 \
00157               [wdt_syncbusy_bit]    "I"  (WDT_SYNCBUSY_bm)               \
00158             : "memory");                                                 \
00159     } while(0)
00160
00161 static __ATTR_ALWAYS_INLINE__
00162 void wdt_disable (void)
00163 {
00164     uint8_t __temp;
00165     __asm__ __volatile__ (
00166         "wdr"                                "\n\t"
00167         "out %i[ccp_reg], %[ioreg_cen_mask]" "\n\t"
00168         "lds %[tmp], %[wdt_reg]"             "\n\t"
00169         "cbr %[tmp], %[timeout_mask]"        "\n\t"
00170         "sts %[wdt_reg], %[tmp]"
00171         : [tmp]            "=d" (__temp)
00172         : [ccp_reg]        "n" (& CCP),
00173           [ioreg_cen_mask] "r" ((uint8_t)CCP_IOREG_gc),
00174           [wdt_reg]        "n" (& WDT_CTRLA),
00175           [timeout_mask]   "I" (WDT_PERIOD_gm)
00176         : "memory");
00177 }
00178
00179 #else // defined (WDT_CTRLA) && !defined(RAMPD)
00180
00181 /*
00182   wdt_enable(timeout) for xmega devices
00183   - write signature (CCP_IOREG_gc) that enables change of protected I/O
00184     registers to the CCP register
00185   - At the same time,
00186     1) set WDT change enable (WDT_CEN_bm)
00187     2) enable WDT (WDT_ENABLE_bm)
00188     3) set timeout (timeout)
00189   - Synchronization starts when ENABLE bit of WDT is set. So, wait till it
00190     finishes (SYNCBUSY of STATUS register is automatically cleared after the
00191     sync is finished).  */
00192 #define wdt_enable(timeout)                                              \
00193     do {                                                                 \
00194         uint8_t __temp;                                                  \
00195         __asm__ __volatile__ (                                           \
00196             "in __tmp_reg__, %i[rampd]"              "\n\t"              \
00197             "out %i[rampd], __zero_reg__"            "\n\t"              \
00198             "out %i[ccp_reg], %[ioreg_cen_mask]"     "\n\t"              \
00199             "sts %[wdt_reg], %[wdt_enable_timeout]" "\n\t"               \
00200             "1:lds %[tmp], %[wdt_status_reg]"        "\n\t"              \
00201             "sbrc %[tmp], %[wdt_syncbusy_bit]"       "\n\t"              \
00202             "rjmp 1b"                                "\n\t"              \
00203             "out %i[rampd], __tmp_reg__"                                 \
00204             : [tmp]                 "=r" (__temp)                        \
00205             : [rampd]               "n" (& RAMPD),                       \
00206               [ccp_reg]             "n" (& CCP),                         \
00207               [ioreg_cen_mask]      "r" ((uint8_t)CCP_IOREG_gc),         \
00208               [wdt_reg]             "n" (& WDT_CTRL),                    \
00209               [wdt_enable_timeout] "r" ((uint8_t)(WDT_CEN_bm             \
00210                                                   | WDT_ENABLE_bm        \
00211                                                   | ((timeout + 1) << 2))), \
00212               [wdt_status_reg]      "n" (& WDT_STATUS),                  \
00213               [wdt_syncbusy_bit]    "I" (WDT_SYNCBUSY_bm)                \
00214             : "memory");                                                 \
00215     } while(0)
```

```
00216
00217 static __ATTR_ALWAYS_INLINE__
00218 void wdt_disable (void)
00219 {
00220     __asm__ __volatile__ (
00221         "in __tmp_reg__, %i[rampd]"          "\n\t"
00222         "out %i[rampd], __zero_reg__"        "\n\t"
00223         "out %i[ccp_reg], %[ioreg_cen_mask]" "\n\t"
00224         "sts %[wdt_reg], %[disable_mask]"    "\n\t"
00225         "out %i[rampd], __tmp_reg__"
00226         : /* no outputs */
00227         : [rampd]             "n" (& RAMPD),
00228           [ccp_reg]           "n" (& CCP),
00229           [ioreg_cen_mask]    "r" ((uint8_t)CCP_IOREG_gc),
00230           [wdt_reg]           "n" (& WDT_CTRL),
00231           [disable_mask]      "r" ((uint8_t)((~WDT_ENABLE_bm) | WDT_CEN_bm))
00232         : "memory");
00233 }
00234
00235 #endif // defined (WDT_CTRLA) && !defined(RAMPD)
00236
00237 #elif defined(__AVR_TINY__)
00238
00239 #define wdt_enable(value)                                                \
00240     __asm__ __volatile__ (                                               \
00241         "in __tmp_reg__,__SREG__"         "\n\t"                         \
00242         "cli"                             "\n\t"                         \
00243         "wdr"                             "\n\t"                         \
00244         "out %i[CCPADDRESS],%[SIGNATURE]" "\n\t"                         \
00245         "out %i[WDTREG],%[WDVALUE]"       "\n\t"                         \
00246         "out __SREG__,__tmp_reg__"                                       \
00247         : /* no outputs */                                               \
00248         : [CCPADDRESS] "n" (& CCP),                                      \
00249           [SIGNATURE] "r" ((uint8_t)0xD8),                               \
00250           [WDTREG] "n" (& _WD_CONTROL_REG),                              \
00251           [WDVALUE] "r" ((uint8_t)((value & 0x08 ? _WD_PS3_MASK : 0x00) \
00252                                    | _BV(WDE) | (value & 0x07) ))        \
00253         : "memory")
00254
00255 static __ATTR_ALWAYS_INLINE__
00256 void wdt_disable (void)
00257 {
00258     uint8_t __temp_wd;
00259     __asm__ __volatile__ (
00260         "in __tmp_reg__,__SREG__"         "\n\t"
00261         "cli"                             "\n\t"
00262         "wdr"                             "\n\t"
00263         "out %i[CCPADDRESS],%[SIGNATURE]" "\n\t"
00264         "in  %[TEMP_WD],%i[WDTREG]"       "\n\t"
00265         "cbr %[TEMP_WD],%[WDVALUE]"       "\n\t"
00266         "out %i[WDTREG],%[TEMP_WD]"       "\n\t"
00267         "out __SREG__,__tmp_reg__"
00268         : [TEMP_WD] "=d" (__temp_wd)
00269         : [CCPADDRESS] "n" (& CCP),
00270           [SIGNATURE] "r" ((uint8_t)0xD8),
00271           [WDTREG] "n" (& _WD_CONTROL_REG),
00272           [WDVALUE] "n" (1 << WDE)
00273         : "memory");
00274 }
00275
00276 #elif defined(CCP)
00277
00278 static __ATTR_ALWAYS_INLINE__
00279 void wdt_enable (const uint8_t value)
00280 {
00281     if (!_SFR_IO_REG_P (CCP) && !_SFR_IO_REG_P (_WD_CONTROL_REG))
00282     {
00283         __asm__ __volatile__ (
00284             "in __tmp_reg__,__SREG__"         "\n\t"
00285             "cli"                             "\n\t"
00286             "wdr"                             "\n\t"
00287             "sts %[CCPADDRESS],%[SIGNATURE]" "\n\t"
00288             "sts %[WDTREG],%[WDVALUE]"       "\n\t"
```

```
00289             "out __SREG__,__tmp_reg__"
00290             : /* no outputs */
00291             : [CCPADDRESS] "n" (& CCP),
00292               [SIGNATURE] "r" ((uint8_t)0xD8),
00293               [WDTREG] "n" (& _WD_CONTROL_REG),
00294               [WDVALUE] "r" ((uint8_t)((value & 0x08 ? _WD_PS3_MASK : 0x00)
00295                                        | _BV(WDE) | (value & 0x07) ))
00296             : "memory");
00297     }
00298     else if (!_SFR_IO_REG_P (CCP) && _SFR_IO_REG_P (_WD_CONTROL_REG))
00299     {
00300         __asm__ __volatile__ (
00301             "in __tmp_reg__,__SREG__"        "\n\t"
00302             "cli"                            "\n\t"
00303             "wdr"                            "\n\t"
00304             "sts %[CCPADDRESS],%[SIGNATURE]" "\n\t"
00305             "out %i[WDTREG],%[WDVALUE]"      "\n\t"
00306             "out __SREG__,__tmp_reg__"
00307             : /* no outputs */
00308             : [CCPADDRESS] "n" (& CCP),
00309               [SIGNATURE] "r" ((uint8_t)0xD8),
00310               [WDTREG] "n" (& _WD_CONTROL_REG),
00311               [WDVALUE] "r" ((uint8_t)((value & 0x08 ? _WD_PS3_MASK : 0x00)
00312                                        | _BV(WDE) | (value & 0x07) ))
00313             : "memory");
00314     }
00315     else if (_SFR_IO_REG_P (CCP) && !_SFR_IO_REG_P (_WD_CONTROL_REG))
00316     {
00317         __asm__ __volatile__ (
00318             "in __tmp_reg__,__SREG__"         "\n\t"
00319             "cli"                             "\n\t"
00320             "wdr"                             "\n\t"
00321             "out %i[CCPADDRESS],%[SIGNATURE]" "\n\t"
00322             "sts %[WDTREG],%[WDVALUE]"        "\n\t"
00323             "out __SREG__,__tmp_reg__"
00324             : /* no outputs */
00325             : [CCPADDRESS] "n" (& CCP),
00326               [SIGNATURE] "r" ((uint8_t)0xD8),
00327               [WDTREG] "n" (& _WD_CONTROL_REG),
00328               [WDVALUE] "r" ((uint8_t)((value & 0x08 ? _WD_PS3_MASK : 0x00)
00329                                        | _BV(WDE) | (value & 0x07) ))
00330             : "memory");
00331     }
00332     else
00333     {
00334         __asm__ __volatile__ (
00335             "in __tmp_reg__,__SREG__"         "\n\t"
00336             "cli"                             "\n\t"
00337             "wdr"                             "\n\t"
00338             "out %i[CCPADDRESS],%[SIGNATURE]" "\n\t"
00339             "out %i[WDTREG],%[WDVALUE]"       "\n\t"
00340             "out __SREG__,__tmp_reg__"
00341             : /* no outputs */
00342             : [CCPADDRESS] "n" (& CCP),
00343               [SIGNATURE] "r" ((uint8_t)0xD8),
00344               [WDTREG] "n" (& _WD_CONTROL_REG),
00345               [WDVALUE] "r" ((uint8_t)((value & 0x08 ? _WD_PS3_MASK : 0x00)
00346                                        | _BV(WDE) | (value & 0x07) ))
00347             : "memory");
00348     }
00349 }
00350
00351 static __ATTR_ALWAYS_INLINE__
00352 void wdt_disable (void)
00353 {
00354     if (!_SFR_IO_REG_P (CCP) && !_SFR_IO_REG_P(_WD_CONTROL_REG))
00355     {
00356         uint8_t __temp_wd;
00357         __asm__ __volatile__ (
00358             "in __tmp_reg__,__SREG__"        "\n\t"
00359             "cli"                            "\n\t"
00360             "wdr"                            "\n\t"
00361             "sts %[CCPADDRESS],%[SIGNATURE]" "\n\t"
```

```
00362             "lds %[TEMP_WD],%[WDTREG]"       "\n\t"
00363             "cbr %[TEMP_WD],%[WDVALUE]"      "\n\t"
00364             "sts %[WDTREG],%[TEMP_WD]"       "\n\t"
00365             "out __SREG__,__tmp_reg__"
00366             : [TEMP_WD] "=d" (__temp_wd)
00367             : [CCPADDRESS] "n" (& CCP),
00368               [SIGNATURE] "r" ((uint8_t)0xD8),
00369               [WDTREG] "n" (& _WD_CONTROL_REG),
00370               [WDVALUE] "n" (1 « WDE)
00371             : "memory");
00372     }
00373     else if (!_SFR_IO_REG_P (CCP) && _SFR_IO_REG_P(_WD_CONTROL_REG))
00374     {
00375         uint8_t __temp_wd;
00376         __asm__ __volatile__ (
00377             "in __tmp_reg__,__SREG__"        "\n\t"
00378             "cli"                            "\n\t"
00379             "wdr"                            "\n\t"
00380             "sts %[CCPADDRESS],%[SIGNATURE]" "\n\t"
00381             "in %[TEMP_WD],%i[WDTREG]"       "\n\t"
00382             "cbr %[TEMP_WD],%[WDVALUE]"      "\n\t"
00383             "out %i[WDTREG],%[TEMP_WD]"      "\n\t"
00384             "out __SREG__,__tmp_reg__"
00385             : [TEMP_WD] "=d" (__temp_wd)
00386             : [CCPADDRESS] "n" (& CCP),
00387               [SIGNATURE] "r" ((uint8_t)0xD8),
00388               [WDTREG] "n" (& _WD_CONTROL_REG),
00389               [WDVALUE] "n" (1 « WDE)
00390             : "memory");
00391     }
00392     else if (_SFR_IO_REG_P (CCP) && !_SFR_IO_REG_P(_WD_CONTROL_REG))
00393     {
00394         uint8_t __temp_wd;
00395         __asm__ __volatile__ (
00396             "in __tmp_reg__,__SREG__"         "\n\t"
00397             "cli"                             "\n\t"
00398             "wdr"                             "\n\t"
00399             "out %i[CCPADDRESS],%[SIGNATURE]" "\n\t"
00400             "lds %[TEMP_WD],%[WDTREG]"        "\n\t"
00401             "cbr %[TEMP_WD],%[WDVALUE]"       "\n\t"
00402             "sts %[WDTREG],%[TEMP_WD]"        "\n\t"
00403             "out __SREG__,__tmp_reg__"
00404             : [TEMP_WD] "=d" (__temp_wd)
00405             : [CCPADDRESS] "n" (& CCP),
00406               [SIGNATURE] "r" ((uint8_t)0xD8),
00407               [WDTREG] "n" (& _WD_CONTROL_REG),
00408               [WDVALUE] "n" (1 « WDE)
00409             : "memory");
00410     }
00411     else
00412     {
00413         uint8_t __temp_wd;
00414         __asm__ __volatile__ (
00415             "in __tmp_reg__,__SREG__"         "\n\t"
00416             "cli"                             "\n\t"
00417             "wdr"                             "\n\t"
00418             "out %i[CCPADDRESS],%[SIGNATURE]" "\n\t"
00419             "in %[TEMP_WD],%i[WDTREG]"        "\n\t"
00420             "cbr %[TEMP_WD],%[WDVALUE]"       "\n\t"
00421             "out %i[WDTREG],%[TEMP_WD]"       "\n\t"
00422             "out __SREG__,__tmp_reg__"
00423             : [TEMP_WD] "=d" (__temp_wd)
00424             : [CCPADDRESS] "n" (& CCP),
00425               [SIGNATURE] "r" ((uint8_t)0xD8),
00426               [WDTREG] "n" (& _WD_CONTROL_REG),
00427               [WDVALUE] "n" (1 « WDE)
00428             : "memory");
00429     }
00430 }
00431
00432 #else
00433
00434 /** \ingroup avr_watchdog
```

```
00435     Enable the watchdog timer, configuring it for expiry after
00436     \c timeout (which is a combination of the \c WDP0 through
00437     \c WDP2 bits to write into the \c WDTCR register; For those devices
00438     that have a \c WDTCSR register, it uses the combination of the \c WDP0
00439     through \c WDP3 bits).
00440
00441     See also the symbolic constants \c WDTO_15MS et al.  */
00442 static __ATTR_ALWAYS_INLINE__
00443 void wdt_enable(const uint8_t value)
00444 {
00445     if (_SFR_IO_REG_P (_WD_CONTROL_REG))
00446     {
00447         __asm__ __volatile__ (
00448             "in __tmp_reg__,__SREG__"  "\n\t"
00449             "cli"                      "\n\t"
00450             "wdr"                      "\n\t"
00451             "out %i0, %1"              "\n\t"
00452             "out __SREG__,__tmp_reg__" "\n\t"
00453             "out %i0, %2"
00454             : /* no outputs */
00455             : "n" (& _WD_CONTROL_REG),
00456               "r" ((uint8_t)(_BV(_WD_CHANGE_BIT) | _BV(WDE))),
00457               "r" ((uint8_t) ((value & 0x08 ? _WD_PS3_MASK : 0x00)
00458                               | _BV(WDE) | (value & 0x07)) )
00459             : "memory");
00460     }
00461     else
00462     {
00463         __asm__ __volatile__ (
00464             "in __tmp_reg__,__SREG__"  "\n\t"
00465             "cli"                      "\n\t"
00466             "wdr"                      "\n\t"
00467             "sts %0, %1"               "\n\t"
00468             "out __SREG__,__tmp_reg__" "\n\t"
00469             "sts %0, %2"
00470             : /* no outputs */
00471             : "n" (& _WD_CONTROL_REG),
00472               "r" ((uint8_t)(_BV(_WD_CHANGE_BIT) | _BV(WDE))),
00473               "r" ((uint8_t) ((value & 0x08 ? _WD_PS3_MASK : 0x00)
00474                               | _BV(WDE) | (value & 0x07)) )
00475             : "memory");
00476     }
00477 }
00478
00479 /** \ingroup avr_watchdog
00480     Disable the watchdog timer.
00481 */
00482 static __ATTR_ALWAYS_INLINE__
00483 void wdt_disable (void)
00484 {
00485     if (_SFR_IO_REG_P (_WD_CONTROL_REG))
00486     {
00487         uint8_t __temp_reg;
00488         __asm__ __volatile__ (
00489             "in __tmp_reg__,__SREG__"     "\n\t"
00490             "cli"                         "\n\t"
00491             "wdr"                         "\n\t"
00492             "in  %[TEMPREG],%i[WDTREG]"   "\n\t"
00493             "ori %[TEMPREG],%[WDCE_WDE]"  "\n\t"
00494             "out %i[WDTREG],%[TEMPREG]"   "\n\t"
00495             "out %i[WDTREG],__zero_reg__" "\n\t"
00496             "out __SREG__,__tmp_reg__"
00497             : [TEMPREG] "=d" (__temp_reg)
00498             : [WDTREG]   "n" (& _WD_CONTROL_REG),
00499               [WDCE_WDE] "n"  ((uint8_t)(_BV(_WD_CHANGE_BIT) | _BV(WDE)))
00500             : "memory");
00501     }
00502     else
00503     {
00504         uint8_t __temp_reg;
00505         __asm__ __volatile__ (
00506             "in __tmp_reg__,__SREG__"     "\n\t"
00507             "cli"                         "\n\t"
```

```
00508             "wdr"                          "\n\t"
00509             "lds %[TEMPREG],%[WDTREG]"    "\n\t"
00510             "ori %[TEMPREG],%[WDCE_WDE]" "\n\t"
00511             "sts %[WDTREG],%[TEMPREG]"    "\n\t"
00512             "sts %[WDTREG],__zero_reg__" "\n\t"
00513             "out __SREG__,__tmp_reg__"
00514             : [TEMPREG] "=d" (__temp_reg)
00515             : [WDTREG]  "n"  (& _WD_CONTROL_REG),
00516               [WDCE_WDE]  "n"  ((uint8_t)(_BV(_WD_CHANGE_BIT) | _BV(WDE)))
00517             : "memory");
00518     }
00519 }
00520
00521 #endif
00522
00523
00524 /**
00525    \ingroup avr_watchdog
00526    Symbolic constants for the watchdog timeout.  Since the watchdog
00527    timer is based on a free-running RC oscillator, the times are
00528    approximate only and apply to a supply voltage of 5 V.  At lower
00529    supply voltages, the times will increase.  For older devices, the
00530    times will be as large as three times when operating at Vcc = 3 V,
00531    while the newer devices (e. g. ATmega128, ATmega8) only experience
00532    a negligible change.
00533
00534    Possible timeout values are: 15 ms, 30 ms, 60 ms, 120 ms, 250 ms,
00535    500 ms, 1 s, 2 s.  (Some devices also allow for 4 s and 8 s.)
00536    Symbolic constants are formed by the prefix
00537    \c WDTO_, followed by the time.
00538
00539    Example that would select a watchdog timer expiry of approximately
00540    500 ms:
00541    \code
00542    wdt_enable(WDTO_500MS);
00543    \endcode
00544 */
00545 #if defined(__DOXYGEN__) || defined(__AVR_XMEGA__)
00546 #define WDTO_8MS   -1
00547 #endif
00548
00549 /** \ingroup avr_watchdog
00550     A value to be passed to wdt_enable() for the specified watchdog
00551     timeout duration. */
00552 #define WDTO_15MS   0
00553
00554 /** \ingroup avr_watchdog
00555     See \c #WDTO_15MS */
00556 #define WDTO_30MS   1
00557
00558 /** \ingroup avr_watchdog
00559     See \c #WDTO_15MS */
00560 #define WDTO_60MS   2
00561
00562 /** \ingroup avr_watchdog
00563     See \c #WDTO_15MS */
00564 #define WDTO_120MS  3
00565
00566 /** \ingroup avr_watchdog
00567     See \c #WDTO_15MS */
00568 #define WDTO_250MS  4
00569
00570 /** \ingroup avr_watchdog
00571     See \c #WDTO_15MS */
00572 #define WDTO_500MS  5
00573
00574 /** \ingroup avr_watchdog
00575     See \c #WDTO_15MS */
00576 #define WDTO_1S     6
00577
00578 /** \ingroup avr_watchdog
00579     See \c #WDTO_15MS */
00580 #define WDTO_2S     7
```

```
00581
00582 #if defined(__DOXYGEN__) || defined(WDP3) || defined(__AVR_XMEGA__)
00583
00584 /** \ingroup avr_watchdog
00585     See \c WDTO_15MS
00586     Note: This is only available on the
00587     ATtiny2313,
00588     ATtiny24, ATtiny44, ATtiny84, ATtiny84A,
00589     ATtiny25, ATtiny45, ATtiny85,
00590     ATtiny261, ATtiny461, ATtiny861,
00591     ATmega48*, ATmega88*, ATmega168*, ATmega328*,
00592     ATmega164P, ATmega324P, ATmega324PB, ATmega644P, ATmega644,
00593     ATmega640, ATmega1280, ATmega1281, ATmega2560, ATmega2561,
00594     ATmega8HVA, ATmega16HVA, ATmega32HVB,
00595     ATmega406, ATmega1284P,
00596     AT90PWM1, AT90PWM2, AT90PWM2B, AT90PWM3, AT90PWM3B, AT90PWM216, AT90PWM316,
00597     AT90PWM81, AT90PWM161,
00598     AT90USB82, AT90USB162,
00599     AT90USB646, AT90USB647, AT90USB1286, AT90USB1287,
00600     ATtiny48, ATtiny88.
00601
00602     Note: This value does <em>not</em> match the bit pattern of the
00603     respective control register.  It is solely meant to be used together
00604     with wdt_enable().
00605 */
00606 #define WDTO_4S     8
00607
00608 /** \ingroup avr_watchdog
00609     See \c WDTO_15MS
00610     Note: This is only available on the
00611     ATtiny2313,
00612     ATtiny24, ATtiny44, ATtiny84, ATtiny84A,
00613     ATtiny25, ATtiny45, ATtiny85,
00614     ATtiny261, ATtiny461, ATtiny861,
00615     ATmega48*, ATmega88*, ATmega168*, ATmega328*,
00616     ATmega164P, ATmega324P, ATmega324PB, ATmega644P, ATmega644,
00617     ATmega640, ATmega1280, ATmega1281, ATmega2560, ATmega2561,
00618     ATmega8HVA, ATmega16HVA, ATmega32HVB,
00619     ATmega406, ATmega1284P,
00620     ATmega2564RFR2, ATmega256RFR2, ATmega1284RFR2, ATmega128RFR2, ATmega644RFR2,
      ATmega64RFR2
00621     AT90PWM1, AT90PWM2, AT90PWM2B, AT90PWM3, AT90PWM3B, AT90PWM216, AT90PWM316,
00622     AT90PWM81, AT90PWM161,
00623     AT90USB82, AT90USB162,
00624     AT90USB646, AT90USB647, AT90USB1286, AT90USB1287,
00625     ATtiny48, ATtiny88,
00626     ATxmega16a4u, ATxmega32a4u,
00627     ATxmega16c4, ATxmega32c4,
00628     ATxmega128c3, ATxmega192c3, ATxmega256c3.
00629
00630     Note: This value does <em>not</em> match the bit pattern of the
00631     respective control register.  It is solely meant to be used together
00632     with wdt_enable().
00633 */
00634 #define WDTO_8S     9
00635
00636 #endif  /* defined(__DOXYGEN__) || defined(WDP3) */
00637
00638 #endif /* _AVR_WDT_H_ */
```

22.47 xmega.h

```
00001 /* Copyright (c) 2012 Joerg Wunsch
00002    All rights reserved.
00003
00004    Redistribution and use in source and binary forms, with or without
00005    modification, are permitted provided that the following conditions are met:
00006
00007    * Redistributions of source code must retain the above copyright
00008      notice, this list of conditions and the following disclaimer.
00009
```

```
00010    * Redistributions in binary form must reproduce the above copyright
00011      notice, this list of conditions and the following disclaimer in
00012      the documentation and/or other materials provided with the
00013      distribution.
00014
00015    * Neither the name of the copyright holders nor the names of
00016      contributors may be used to endorse or promote products derived
00017      from this software without specific prior written permission.
00018
00019   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
00020   AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
00021   IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
00022   ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
00023   LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
00024   CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
00025   SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
00026   INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
00027   CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
00028   ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
00029   POSSIBILITY OF SUCH DAMAGE. */
00030
00031 /*
00032  * This file is included by <avr/io.h> whenever compiling for an Xmega
00033  * device.  It abstracts certain features common to the Xmega device
00034  * families.
00035  */
00036
00037 #ifndef _AVR_XMEGA_H
00038 #define _AVR_XMEGA_H
00039
00040 #ifdef __DOXYGEN__
00041 /**
00042  \ingroup avr_io
00043
00044  Write value \c value to IO register \c reg that is protected through
00045  the Xmega or AVRrc configuration change protection (CCP) mechanism.  This
00046  implements the timed sequence that is required for CCP.
00047
00048  This macro requires that the address of \c reg is a compile-time constant.
00049  When that is not the case, the \c ccp_write_io() function can be used.
00050
00051  Example to modify the CPU clock:
00052  \code
00053  #include <avr/io.h>
00054
00055  _PROTECTED_WRITE (CLK_PSCTRL, CLK_PSADIV0_bm);
00056  _PROTECTED_WRITE (CLK_CTRL, CLK_SCLKSEL0_bm);
00057  \endcode */
00058 #define _PROTECTED_WRITE(reg, value)
00059
00060 /**
00061  \ingroup avr_io
00062
00063  Write value \c value to register \c reg that is protected through
00064  the Xmega or ATtiny102/104 configuration change protection (CCP) key for self
00065  programming (SPM).  This implements the timed sequence that is
00066  required for CCP.
00067
00068  This macro requires that the address of \c reg is a compile-time constant.
00069  When that is not the case, the \c ccp_write_spm() function can be used.
00070
00071  Example to modify the CPU clock:
00072  \code
00073  #include <avr/io.h>
00074
00075  _PROTECTED_WRITE_SPM (NVMCTRL_CTRLA, NVMCTRL_CMD_PAGEERASEWRITE_gc);
00076  \endcode  */
00077 #define _PROTECTED_WRITE_SPM(reg, value)
00078
00079 #else  /* !__DOXYGEN__ */
00080
00081 #ifdef __AVR_TINY__
00082
```

```
00083 #define _PROTECTED_WRITE(reg, value)            \
00084   __asm__ __volatile__ ("out %i0, %1" "\n\t"        \
00085             "out %i2, %3"           \
00086             :               \
00087             : "n" (& CCP),          \
00088               "d" ((uint8_t) 0xd8),     \
00089               "n" (& (reg)),        \
00090               "r" ((uint8_t) (value)))
00091
00092 #elif defined (__AVR_XMEGA__)
00093
00094 #define _PROTECTED_WRITE(reg, value)            \
00095   __asm__ __volatile__ ("out %i0, %1" "\n\t"        \
00096             "sts %2, %3"            \
00097             :               \
00098             : "n" (& CCP),          \
00099               "d" ((uint8_t) CCP_IOREG_gc), \
00100               "n" (& (reg)),        \
00101               "r" ((uint8_t) (value)))
00102 #endif /* AVR_TINY || Xmega */
00103
00104 #if defined(__AVR_TINY__) && defined(CCP_SPM_gc)
00105
00106 #define _PROTECTED_WRITE_SPM(reg, value)        \
00107   __asm__ __volatile__ ("out %i0, %1" "\n\t"        \
00108             "out %i2, %3"           \
00109             :               \
00110             : "n" (& CCP),          \
00111               "d" ((uint8_t) CCP_SPM_gc),   \
00112               "n" (& (reg)),        \
00113               "r" ((uint8_t) (value)))
00114
00115 #elif defined(__AVR_XMEGA__)
00116
00117 #define _PROTECTED_WRITE_SPM(reg, value)        \
00118   __asm__ __volatile__ ("out %i0, %1" "\n\t"        \
00119             "sts %2, %3"            \
00120             :               \
00121             : "n" (& CCP),          \
00122               "d" ((uint8_t) CCP_SPM_gc),   \
00123               "n" (& (reg)),        \
00124               "r" ((uint8_t) (value)))
00125 #endif /* ATtiny102/104 || Xmega */
00126 #endif /* DOXYGEN */
00127
00128 #endif /* _AVR_XMEGA_H */
```

22.48 attribs.h

```
00001 /* Copyright (c) 2025  Georg-Johann Lay
00002    All rights reserved.
00003
00004    Redistribution and use in source and binary forms, with or without
00005    modification, are permitted provided that the following conditions are met:
00006
00007    * Redistributions of source code must retain the above copyright
00008      notice, this list of conditions and the following disclaimer.
00009    * Redistributions in binary form must reproduce the above copyright
00010      notice, this list of conditions and the following disclaimer in
00011      the documentation and/or other materials provided with the
00012      distribution.
00013    * Neither the name of the copyright holders nor the names of
00014      contributors may be used to endorse or promote products derived
00015      from this software without specific prior written permission.
00016
00017   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
00018   AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
00019   IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
00020   ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
00021   LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
00022   CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
```

```
00023   SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
00024   INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
00025   CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
00026   ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
00027   POSSIBILITY OF SUCH DAMAGE. */
00028
00029 #ifndef __BITS_ATTRIBS_H_
00030 #define __BITS_ATTRIBS_H_
00031
00032 #define __ATTR_ALWAYS_INLINE__ __inline__ __attribute__((__always_inline__))
00033
00034 #ifdef  __GNUC_STDC_INLINE__
00035 #define __ATTR_GNU_INLINE__   __attribute__((__gnu_inline__))
00036 #else
00037 #define __ATTR_GNU_INLINE__
00038 #endif
00039
00040 #define __ATTR_CONST__ __attribute__((__const__))
00041
00042 #define __ATTR_PURE__ __attribute__((__pure__))
00043
00044 #define __ATTR_MALLOC__ __attribute__((__malloc__))
00045
00046 #define __ATTR_NORETURN__ __attribute__((__noreturn__))
00047
00048 #if __GNUC__ >= 8
00049 #define __ATTR_NONSTRING__ __attribute__((__nonstring__))
00050 #else
00051 #define __ATTR_NONSTRING__
00052 #endif
00053
00054 /* AVR specific */
00055
00056 #ifdef __clang__
00057 #define __ATTR_PROGMEM__ __attribute__((__section__(".progmem.data")))
00058 #else
00059 #define __ATTR_PROGMEM__ __attribute__((__progmem__))
00060 #endif
00061
00062 #endif /* __BITS_ATTRIBS_H_ */
```

22.49 def-flash-read.h

```
00001 /* Copyright (c) 2025  Georg-Johann Lay
00002    All rights reserved.
00003
00004    Redistribution and use in source and binary forms, with or without
00005    modification, are permitted provided that the following conditions are met:
00006
00007    * Redistributions of source code must retain the above copyright
00008      notice, this list of conditions and the following disclaimer.
00009    * Redistributions in binary form must reproduce the above copyright
00010      notice, this list of conditions and the following disclaimer in
00011      the documentation and/or other materials provided with the
00012      distribution.
00013    * Neither the name of the copyright holders nor the names of
00014      contributors may be used to endorse or promote products derived
00015      from this software without specific prior written permission.
00016
00017   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
00018   AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
00019   IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
00020   ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
00021   LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
00022   CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
00023   SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
00024   INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
00025   CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
00026   ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
00027   POSSIBILITY OF SUCH DAMAGE. */
00028
```

```
00029 #ifndef __BITS_DEF_FLASH_READ_H_
00030 #define __BITS_DEF_FLASH_READ_H_
00031
00032 #define _Avrlibc_Def_F_4(Name, Typ)                  \
00033   static __ATTR_ALWAYS_INLINE__                      \
00034   Typ flash_read_##Name (const __flash Typ *__addr) \
00035   {                                                  \
00036     return *__addr;                                  \
00037   }
00038
00039 #define _Avrlibc_Def_F_8(Name, Typ)                  \
00040   static __ATTR_ALWAYS_INLINE__                      \
00041   Typ flash_read_##Name (const __flash Typ *__addr) \
00042   {                                                  \
00043     Typ __res;                                       \
00044     __LPM__8 (__res, __addr);                        \
00045     return __res;                                    \
00046   }
00047
00048 #endif /* __BITS_DEF_FLASH_READ_H_ */
```

22.50 def-pgm-read-far.h

```
00001 /* Copyright (c) 2025  Georg-Johann Lay
00002    All rights reserved.
00003
00004    Redistribution and use in source and binary forms, with or without
00005    modification, are permitted provided that the following conditions are met:
00006
00007    * Redistributions of source code must retain the above copyright
00008      notice, this list of conditions and the following disclaimer.
00009    * Redistributions in binary form must reproduce the above copyright
00010      notice, this list of conditions and the following disclaimer in
00011      the documentation and/or other materials provided with the
00012      distribution.
00013    * Neither the name of the copyright holders nor the names of
00014      contributors may be used to endorse or promote products derived
00015      from this software without specific prior written permission.
00016
00017   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
00018   AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
00019   IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
00020   ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
00021   LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
00022   CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
00023   SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
00024   INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
00025   CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
00026   ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
00027   POSSIBILITY OF SUCH DAMAGE. */
00028
00029 #ifndef __BITS_DEF_PGM_READ_FAR_H_
00030 #define __BITS_DEF_PGM_READ_FAR_H_
00031
00032 #define _Avrlibc_Def_Pgm_Far_1(Name, Typ)           \
00033   static __ATTR_ALWAYS_INLINE__                     \
00034   Typ pgm_read_##Name##_far (uint_farptr_t __addr) \
00035   {                                                 \
00036     Typ __res;                                      \
00037     __ELPM__1 (__res, __addr, Typ);                 \
00038     return __res;                                   \
00039   }
00040
00041 #define _Avrlibc_Def_Pgm_Far_2(Name, Typ)           \
00042   static __ATTR_ALWAYS_INLINE__                     \
00043   Typ pgm_read_##Name##_far (uint_farptr_t __addr) \
00044   {                                                 \
00045     Typ __res;                                      \
00046     __ELPM__2 (__res, __addr, Typ);                 \
00047     return __res;                                   \
00048   }
```

```
00049
00050 #define _Avrlibc_Def_Pgm_Far_3(Name, Typ)          \
00051   static __ATTR_ALWAYS_INLINE__                    \
00052   Typ pgm_read_##Name##_far (uint_farptr_t __addr) \
00053   {                                                \
00054     Typ __res;                                     \
00055     __ELPM__3 (__res, __addr, Typ);                \
00056     return __res;                                  \
00057   }
00058
00059 #define _Avrlibc_Def_Pgm_Far_4(Name, Typ)          \
00060   static __ATTR_ALWAYS_INLINE__                    \
00061   Typ pgm_read_##Name##_far (uint_farptr_t __addr) \
00062   {                                                \
00063     Typ __res;                                     \
00064     __ELPM__4 (__res, __addr, Typ);                \
00065     return __res;                                  \
00066   }
00067
00068 #define _Avrlibc_Def_Pgm_Far_8(Name, Typ)          \
00069   static __ATTR_ALWAYS_INLINE__                    \
00070   Typ pgm_read_##Name##_far (uint_farptr_t __addr) \
00071   {                                                \
00072     Typ __res;                                     \
00073     __ELPM__8 (__res, __addr, Typ);                \
00074     return __res;                                  \
00075   }
00076
00077 #endif /* __BITS_DEF_PGM_READ_FAR_H_ */
```

22.51 def-pgm-read.h

```
00001 /* Copyright (c) 2025  Georg-Johann Lay
00002    All rights reserved.
00003
00004    Redistribution and use in source and binary forms, with or without
00005    modification, are permitted provided that the following conditions are met:
00006
00007    * Redistributions of source code must retain the above copyright
00008      notice, this list of conditions and the following disclaimer.
00009    * Redistributions in binary form must reproduce the above copyright
00010      notice, this list of conditions and the following disclaimer in
00011      the documentation and/or other materials provided with the
00012      distribution.
00013    * Neither the name of the copyright holders nor the names of
00014      contributors may be used to endorse or promote products derived
00015      from this software without specific prior written permission.
00016
00017   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
00018   AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
00019   IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
00020   ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
00021   LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
00022   CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
00023   SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
00024   INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
00025   CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
00026   ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
00027   POSSIBILITY OF SUCH DAMAGE. */
00028
00029 #ifndef __BITS_DEF_PGM_READ_H_
00030 #define __BITS_DEF_PGM_READ_H_
00031
00032 #define _Avrlibc_Def_Pgm_1(Name, Typ)           \
00033   static __ATTR_ALWAYS_INLINE__                 \
00034   Typ pgm_read_##Name (const Typ *__addr)       \
00035   {                                             \
00036     Typ __res;                                  \
00037     __LPM__1 (__res, __addr);                   \
00038     return __res;                               \
00039   }
```

```
00040
00041 #define _Avrlibc_Def_Pgm_2(Name, Typ)           \
00042   static __ATTR_ALWAYS_INLINE__                 \
00043   Typ pgm_read_##Name (const Typ *__addr)       \
00044   {                                             \
00045     Typ __res;                                  \
00046     __LPM__2 (__res, __addr);                   \
00047     return __res;                               \
00048   }
00049
00050 #define _Avrlibc_Def_Pgm_3(Name, Typ)           \
00051   static __ATTR_ALWAYS_INLINE__                 \
00052   Typ pgm_read_##Name (const Typ *__addr)       \
00053   {                                             \
00054     Typ __res;                                  \
00055     __LPM__3 (__res, __addr);                   \
00056     return __res;                               \
00057   }
00058
00059 #define _Avrlibc_Def_Pgm_4(Name, Typ)           \
00060   static __ATTR_ALWAYS_INLINE__                 \
00061   Typ pgm_read_##Name (const Typ *__addr)       \
00062   {                                             \
00063     Typ __res;                                  \
00064     __LPM__4 (__res, __addr);                   \
00065     return __res;                               \
00066   }
00067
00068 #define _Avrlibc_Def_Pgm_8(Name, Typ)           \
00069   static __ATTR_ALWAYS_INLINE__                 \
00070   Typ pgm_read_##Name (const Typ *__addr)       \
00071   {                                             \
00072     Typ __res;                                  \
00073     __LPM__8 (__res, __addr);                   \
00074     return __res;                               \
00075   }
00076
00077 #endif /* __BITS_DEF_PGM_READ_H_ */
```

22.52 lpm-elpm.h

```
00001 /* Copyright (c) 2025  Georg-Johann Lay
00002    All rights reserved.
00003
00004    Redistribution and use in source and binary forms, with or without
00005    modification, are permitted provided that the following conditions are met:
00006
00007    * Redistributions of source code must retain the above copyright
00008      notice, this list of conditions and the following disclaimer.
00009    * Redistributions in binary form must reproduce the above copyright
00010      notice, this list of conditions and the following disclaimer in
00011      the documentation and/or other materials provided with the
00012      distribution.
00013    * Neither the name of the copyright holders nor the names of
00014      contributors may be used to endorse or promote products derived
00015      from this software without specific prior written permission.
00016
00017   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
00018   AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
00019   IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
00020   ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
00021   LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
00022   CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
00023   SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
00024   INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
00025   CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
00026   ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
00027   POSSIBILITY OF SUCH DAMAGE. */
00028
00029 #ifndef __BITS_LPM_ELPM_H_
00030 #define __BITS_LPM_ELPM_H_
```

```
00031
00032 /* This header provides low-level macros for other parts of AVR-LibC
00033    that read from program memory by means of inline assembly that
00034    uses LPM and ELPM instructions.  */
00035
00036 /* Historically, avr/pgmspace.h used asm volatile in definitions of
00037    macros like pgm_read_byte(), though that's not required since there
00038    are no hidden side effects in them.  Though volatile may avoid
00039    undesired code reordering against (volatile) I/O accesses.  */
00040
00041 #ifndef __LPM_VOLATILE
00042 #define __LPM_VOLATILE __volatile__
00043 #endif
00044
00045 #ifndef __ELPM_VOLATILE
00046 #define __ELPM_VOLATILE __volatile__
00047 #endif
00048
00049 #if defined(__AVR_TINY__)
00050 /* For Reduced Tiny devices, avr-gcc adds 0x4000 when it takes the address
00051    of a __progmem__ object.  This means we can use open coded C/C++ to read
00052    from progmem.  This assumes we have
00053    - GCC PR71948 - Make progmem work on Reduced Tiny (GCC v7 / 2016-08)  */
00054 #define __LPM__1(res, addr)  res = *addr
00055 #define __LPM__2(res, addr)  res = *addr
00056 #define __LPM__3(res, addr)  res = *addr
00057 #define __LPM__4(res, addr)  res = *addr
00058 #define __LPM__8(res, addr)  res = *addr
00059
00060 #elif defined(__AVR_HAVE_LPMX__)
00061 #define __LPM__1(res, addr)                             \
00062   __asm __LPM_VOLATILE ("lpm %0,%a1"                    \
00063                         : "=r" (res) : "z" (addr))
00064
00065 #define __LPM__2(res, addr)                               \
00066   __asm __LPM_VOLATILE ("lpm %A0,%a1+"            "\n\t"  \
00067                         "lpm %B0,%a1+"                    \
00068                         : "=r" (res), "+z" (addr))
00069
00070 #define __LPM__3(res, addr)                               \
00071   __asm __LPM_VOLATILE ("lpm %A0,%a1+"            "\n\t"  \
00072                         "lpm %B0,%a1+"            "\n\t"  \
00073                         "lpm %C0,%a1+"                    \
00074                         : "=r" (res), "+z" (addr))
00075
00076 #define __LPM__4(res, addr)                               \
00077   __asm __LPM_VOLATILE ("lpm %A0,%a1+"            "\n\t"  \
00078                         "lpm %B0,%a1+"            "\n\t"  \
00079                         "lpm %C0,%a1+"            "\n\t"  \
00080                         "lpm %D0,%a1+"                    \
00081                         : "=r" (res), "+z" (addr))
00082
00083 #define __LPM__8(res, addr)                               \
00084   __asm __LPM_VOLATILE ("lpm %r0+0,%a1+"          "\n\t"  \
00085                         "lpm %r0+1,%a1+"          "\n\t"  \
00086                         "lpm %r0+2,%a1+"          "\n\t"  \
00087                         "lpm %r0+3,%a1+"          "\n\t"  \
00088                         "lpm %r0+4,%a1+"          "\n\t"  \
00089                         "lpm %r0+5,%a1+"          "\n\t"  \
00090                         "lpm %r0+6,%a1+"          "\n\t"  \
00091                         "lpm %r0+7,%a1+"                  \
00092                         : "=r" (res), "+z" (addr))
00093 #else /* Has no LPMx and no Reduced Tiny => Has LPM.  */
00094 #define __LPM__1(res, addr)                                 \
00095   __asm __LPM_VOLATILE ("lpm $ mov %A0,r0"                  \
00096                         : "=r" (res) : "z" (addr) : "r0")
00097
00098 #define __LPM__2(res, addr)                                 \
00099   __asm __LPM_VOLATILE ("lpm $ mov %A0,r0 $ adiw %1,1"  "\n\t"  \
00100                         "lpm $ mov %B0,r0"                  \
00101                         : "=r" (res), "+z" (addr) :: "r0")
00102
00103 #define __LPM__3(res, addr)                                 \
```

```
00104   __asm __LPM_VOLATILE ("lpm $ mov %A0,r0 $ adiw %1,1"  "\n\t"  \
00105                         "lpm $ mov %B0,r0 $ adiw %1,1"  "\n\t"  \
00106                         "lpm $ mov %C0,r0"                      \
00107                         : "=r" (res), "+z" (addr) :: "r0")
00108
00109 #define __LPM__4(res, addr)                                     \
00110   __asm __LPM_VOLATILE ("lpm $ mov %A0,r0 $ adiw %1,1"  "\n\t"  \
00111                         "lpm $ mov %B0,r0 $ adiw %1,1"  "\n\t"  \
00112                         "lpm $ mov %C0,r0 $ adiw %1,1"  "\n\t"  \
00113                         "lpm $ mov %D0,r0"                      \
00114                         : "=r" (res), "+z" (addr) :: "r0")
00115
00116 #define __LPM__8(res, addr)                                       \
00117   __asm __LPM_VOLATILE ("lpm $ mov %r0+0,r0 $ adiw %1,1"  "\n\t"  \
00118                         "lpm $ mov %r0+1,r0 $ adiw %1,1"  "\n\t"  \
00119                         "lpm $ mov %r0+2,r0 $ adiw %1,1"  "\n\t"  \
00120                         "lpm $ mov %r0+3,r0 $ adiw %1,1"  "\n\t"  \
00121                         "lpm $ mov %r0+4,r0 $ adiw %1,1"  "\n\t"  \
00122                         "lpm $ mov %r0+5,r0 $ adiw %1,1"  "\n\t"  \
00123                         "lpm $ mov %r0+6,r0 $ adiw %1,1"  "\n\t"  \
00124                         "lpm $ mov %r0+7,r0"                      \
00125                         : "=r" (res), "+z" (addr) :: "r0")
00126 #endif /* LPM cases */
00127
00128
00129 #if defined(__AVR_HAVE_ELPMX__)
00130
00131 #ifdef __AVR_HAVE_RAMPD__
00132 /* For devices with EBI, reset RAMPZ to zero after.  */
00133 #define __pgm_clr_RAMPZ_ "\n\t" "out  __RAMPZ__,__zero_reg__"
00134 #else
00135 /* Devices without EBI: no need to reset RAMPZ.  */
00136 #define __pgm_clr_RAMPZ_ /* empty */
00137 #endif
00138
00139 #define __ELPM__1(res, addr, T)                         \
00140   __asm __ELPM_VOLATILE ("movw r30,%1"          "\n\t"  \
00141                          "out  __RAMPZ__,%C1"   "\n\t"  \
00142                          "elpm %A0,Z"                   \
00143                          __pgm_clr_RAMPZ_               \
00144                          : "=r" (res)                   \
00145                          : "r" (addr)                   \
00146                          : "r30", "r31")
00147
00148 #define __ELPM__2(res, addr, T)                         \
00149   __asm __ELPM_VOLATILE ("movw r30,%1"          "\n\t"  \
00150                          "out  __RAMPZ__,%C1"   "\n\t"  \
00151                          "elpm %A0,Z+"          "\n\t"  \
00152                          "elpm %B0,Z+"                  \
00153                          __pgm_clr_RAMPZ_               \
00154                          : "=r" (res)                   \
00155                          : "r" (addr)                   \
00156                          : "r30", "r31")
00157
00158 #define __ELPM__3(res, addr, T)                         \
00159   __asm __ELPM_VOLATILE ("movw r30,%1"          "\n\t"  \
00160                          "out  __RAMPZ__,%C1"   "\n\t"  \
00161                          "elpm %A0,Z+"          "\n\t"  \
00162                          "elpm %B0,Z+"          "\n\t"  \
00163                          "elpm %C0,Z+"                  \
00164                          __pgm_clr_RAMPZ_               \
00165                          : "=r" (res)                   \
00166                          : "r" (addr)                   \
00167                          : "r30", "r31")
00168
00169 #define __ELPM__4(res, addr, T)                         \
00170   __asm __ELPM_VOLATILE ("movw r30,%1"          "\n\t"  \
00171                          "out  __RAMPZ__,%C1"   "\n\t"  \
00172                          "elpm %A0,Z+"          "\n\t"  \
00173                          "elpm %B0,Z+"          "\n\t"  \
00174                          "elpm %C0,Z+"          "\n\t"  \
00175                          "elpm %D0,Z+"                  \
00176                          __pgm_clr_RAMPZ_               \
```

```
00177                           : "=r" (res)                    \
00178                           : "r" (addr)                    \
00179                           : "r30", "r31")
00180
00181 #define __ELPM__8(res, addr, T)                           \
00182   __asm __ELPM_VOLATILE ("movw r30,%1"          "\n\t"    \
00183                          "out  __RAMPZ__,%C1"   "\n\t"    \
00184                          "elpm %r0+0,Z+"        "\n\t"    \
00185                          "elpm %r0+1,Z+"        "\n\t"    \
00186                          "elpm %r0+2,Z+"        "\n\t"    \
00187                          "elpm %r0+3,Z+"        "\n\t"    \
00188                          "elpm %r0+4,Z+"        "\n\t"    \
00189                          "elpm %r0+5,Z+"        "\n\t"    \
00190                          "elpm %r0+6,Z+"        "\n\t"    \
00191                          "elpm %r0+7,Z+"                  \
00192                          __pgm_clr_RAMPZ_                 \
00193                          : "=r" (res)                     \
00194                          : "r" (addr)                     \
00195                          : "r30", "r31")
00196
00197 /* FIXME: AT43USB320 does not have RAMPZ but supports (external) program
00198    memory of 64 KiW, at least that's what the comments in io43usb32x.h are
00199    indicating.  A solution would be to put the device in a different
00200    multilib-set (see GCC PR78275), as io.h has  "#define FLASHEND 0x0FFFF".
00201    For now, just exclude AT43USB320 from code that uses RAMPZ. Also note
00202    that the manual asserts that the entire program memory can be accessed
00203    by LPM, implying only 64 KiB of program memory.  */
00204 #elif defined(__AVR_HAVE_ELPM__) \
00205       && !defined(__AVR_AT43USB320__)
00206 /* The poor devices without ELPMx: Do 24-bit addresses by hand... */
00207 #define __ELPM__1(res, addr, T)                                    \
00208   __asm __ELPM_VOLATILE ("mov r30,%A1"    "\n\t"                   \
00209                          "mov r31,%B1"    "\n\t"                   \
00210                          "out __RAMPZ__,%C1 $ elpm $ mov %A0,r0"   \
00211                          : "=r" (res)                              \
00212                          : "r" (addr)                              \
00213                          : "r30", "r31", "r0")
00214
00215 #define __ELPM__2(res, addr, T)                                    \
00216   __asm __ELPM_VOLATILE                                            \
00217   ("mov r30,%A1"    "\n\t"                                         \
00218    "mov r31,%B1"    "\n\t"                                         \
00219    "mov %B0,%C1"    "\n\t"                                         \
00220    "out __RAMPZ__,%B0 $ elpm $ mov %A0,r0 $ adiw r30,1 $ adc %B0,r1\n\t"\
00221    "out __RAMPZ__,%B0 $ elpm $ mov %B0,r0"                         \
00222    : "=r" (res)                                                    \
00223    : "r" (addr)                                                    \
00224    : "r30", "r31", "r0")
00225
00226 #define __ELPM__3(res, addr, T)                                    \
00227   __asm __ELPM_VOLATILE                                            \
00228   ("mov r30,%A1"    "\n\t"                                         \
00229    "mov r31,%B1"    "\n\t"                                         \
00230    "mov %C0,%C1"    "\n\t"                                         \
00231    "out __RAMPZ__,%C0 $ elpm $ mov %A0,r0 $ adiw r30,1 $ adc %C0,r1\n\t"\
00232    "out __RAMPZ__,%C0 $ elpm $ mov %B0,r0 $ adiw r30,1 $ adc %C0,r1\n\t"\
00233    "out __RAMPZ__,%C0 $ elpm $ mov %C0,r0"                         \
00234    : "=r" (res)                                                    \
00235    : "r" (addr)                                                    \
00236    : "r30", "r31", "r0")
00237
00238 #define __ELPM__4(res, addr, T)                                    \
00239   __asm __ELPM_VOLATILE                                            \
00240   ("mov r30,%A1"    "\n\t"                                         \
00241    "mov r31,%B1"    "\n\t"                                         \
00242    "mov %D0,%C1"    "\n\t"                                         \
00243    "out __RAMPZ__,%D0 $ elpm $ mov %A0,r0 $ adiw r30,1 $ adc %D0,r1\n\t"\
00244    "out __RAMPZ__,%D0 $ elpm $ mov %B0,r0 $ adiw r30,1 $ adc %D0,r1\n\t"\
00245    "out __RAMPZ__,%D0 $ elpm $ mov %C0,r0 $ adiw r30,1 $ adc %D0,r1\n\t"\
00246    "out __RAMPZ__,%D0 $ elpm $ mov %D0,r0"                         \
00247    : "=r" (res)                                                    \
00248    : "r" (addr)                                                    \
00249    : "r30", "r31", "r0")
```

```
00250
00251 #define __ELPM__8(res, addr, T)                                          \
00252   __asm __ELPM_VOLATILE                                                  \
00253   ("mov r30,%A1"    "\n\t"                                               \
00254    "mov r31,%B1"    "\n\t"                                               \
00255    "mov %r0+7,%C1"  "\n\t"                                               \
00256    "out __RAMPZ__,%r0+7 $ elpm $ mov %r0+0,0 $ adiw 30,1 $ adc %r0+7,1\n\t"\
00257    "out __RAMPZ__,%r0+7 $ elpm $ mov %r0+1,0 $ adiw 30,1 $ adc %r0+7,1\n\t"\
00258    "out __RAMPZ__,%r0+7 $ elpm $ mov %r0+2,0 $ adiw 30,1 $ adc %r0+7,1\n\t"\
00259    "out __RAMPZ__,%r0+7 $ elpm $ mov %r0+3,0 $ adiw 30,1 $ adc %r0+7,1\n\t"\
00260    "out __RAMPZ__,%r0+7 $ elpm $ mov %r0+4,0 $ adiw 30,1 $ adc %r0+7,1\n\t"\
00261    "out __RAMPZ__,%r0+7 $ elpm $ mov %r0+5,0 $ adiw 30,1 $ adc %r0+7,1\n\t"\
00262    "out __RAMPZ__,%r0+7 $ elpm $ mov %r0+6,0 $ adiw 30,1 $ adc %r0+7,1\n\t"\
00263    "out __RAMPZ__,%r0+7 $ elpm $ mov %r0+7,0"                            \
00264    : "=r" (res)                                                          \
00265    : "r" (addr)                                                          \
00266    : "r30", "r31", "r0")
00267 #else
00268 /* No ELPM: Fall back to __LPM__<N>.  */
00269 #define __ELPM__1(r,a,T) const T *__a = (const T*)(uint16_t) a; __LPM__1(r,__a)
00270 #define __ELPM__2(r,a,T) const T *__a = (const T*)(uint16_t) a; __LPM__2(r,__a)
00271 #define __ELPM__3(r,a,T) const T *__a = (const T*)(uint16_t) a; __LPM__3(r,__a)
00272 #define __ELPM__4(r,a,T) const T *__a = (const T*)(uint16_t) a; __LPM__4(r,__a)
00273 #define __ELPM__8(r,a,T) const T *__a = (const T*)(uint16_t) a; __LPM__8(r,__a)
00274 #endif /* ELPM cases */
00275
00276 #endif /* __BITS_LPM_ELPM_H_ */
```

22.53 deprecated.h

```
00001 /* Copyright (c) 2005,2006 Joerg Wunsch
00002    All rights reserved.
00003
00004    Redistribution and use in source and binary forms, with or without
00005    modification, are permitted provided that the following conditions are met:
00006
00007    * Redistributions of source code must retain the above copyright
00008      notice, this list of conditions and the following disclaimer.
00009
00010    * Redistributions in binary form must reproduce the above copyright
00011      notice, this list of conditions and the following disclaimer in
00012      the documentation and/or other materials provided with the
00013      distribution.
00014
00015    * Neither the name of the copyright holders nor the names of
00016      contributors may be used to endorse or promote products derived
00017      from this software without specific prior written permission.
00018
00019   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
00020   AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
00021   IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
00022   ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
00023   LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
00024   CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
00025   SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
00026   INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
00027   CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
00028   ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
00029   POSSIBILITY OF SUCH DAMAGE. */
00030
00031 #ifndef _COMPAT_DEPRECATED_H_
00032 #define _COMPAT_DEPRECATED_H_
00033
00034 /** \defgroup deprecated_items <compat/deprecated.h>: Deprecated items
00035
00036     This header file contains several items that used to be available
00037     in previous versions of this library, but have eventually been
00038     deprecated over time.
00039
00040     \code #include <compat/deprecated.h> \endcode
00041
```

```
00042     These items are supplied within that header file for backward
00043     compatibility reasons only, so old source code that has been
00044     written for previous library versions could easily be maintained
00045     until its end-of-life.  Use of any of these items in new code is
00046     strongly discouraged.
00047
00048     Some device headers provide deprecated vector names starting
00049     with \c SIG_, followed by a relatively verbose but arbitrarily
00050     chosen name describing the interrupt vector.
00051     This has been the only available style in AVR-LibC up to version 1.2.x.
00052     This historical naming style is not recommended for new projects,
00053     and some headers require that the macro \c __AVR_LIBC_DEPRECATED_ENABLE__
00054     is defined so that the \c SIG_ names ISR names are available.
00055     For available #ISR vector names and #ISR_N vector numbers, see
00056     \ref faq_isr_names in the FAQ.
00057 */
00058
00059 /** \name Allowing specific system-wide interrupts
00060
00061     In addition to globally enabling interrupts, each device's particular
00062     interrupt needs to be enabled separately if interrupts for this device are
00063     desired.  While some devices maintain their interrupt enable bit inside
00064     the device's register set, external and timer interrupts have system-wide
00065     configuration registers.
00066
00067     Example:
00068
00069     \code
00070     // Enable timer 1 overflow interrupts.
00071     timer_enable_int(_BV(TOIE1));
00072
00073     // Do some work...
00074
00075     // Disable all timer interrupts.
00076     timer_enable_int(0);
00077     \endcode
00078
00079     \note Be careful when you use these functions. If you already have a
00080     different interrupt enabled, you could inadvertently disable it by
00081     enabling another interrupt. */
00082
00083 /**@{*/
00084
00085 /** \ingroup deprecated_items
00086     \def enable_external_int(mask)
00087     \deprecated
00088
00089     This macro gives access to the \c GIMSK register (or \c EIMSK register
00090     if using an AVR Mega device or \c GICR register for others). Although this
00091     macro is essentially the same as assigning to the register, it does
00092     adapt slightly to the type of device being used. This macro is
00093     unavailable if none of the registers listed above are defined. */
00094
00095 /* Define common register definition if available. */
00096 #if defined(EIMSK)
00097 #  define __EICR  EIMSK
00098 #elif defined(GIMSK)
00099 #  define __EICR  GIMSK
00100 #elif defined(GICR)
00101 #  define __EICR  GICR
00102 #endif
00103
00104 /* If common register defined, define macro. */
00105 #if defined(__EICR) || defined(__DOXYGEN__)
00106 #define enable_external_int(mask)               (__EICR = mask)
00107 #endif
00108
00109 /** \ingroup deprecated_items
00110     \deprecated
00111
00112     This function modifies the \c timsk register.
00113     The value you pass via \c ints is device specific. */
00114
```

```
00115 static __inline__ void timer_enable_int (unsigned char ints)
00116 {
00117 #ifdef TIMSK
00118     TIMSK = ints;
00119 #endif
00120 }
00121
00122 /** \ingroup deprecated_items
00123     \def INTERRUPT(signame)
00124     \deprecated
00125
00126     Introduces an interrupt handler function that runs with global interrupts
00127     initially enabled. This allows interrupt handlers to be interrupted.
00128
00129     As this macro has been used by too many unsuspecting people in the
00130     past, it has been deprecated, and will be removed in a future
00131     version of the library.  Users who want to legitimately re-enable
00132     interrupts in their interrupt handlers as quickly as possible are
00133     encouraged to explicitly declare their handlers as described
00134     \ref attr_interrupt "above".
00135 */
00136
00137 #ifndef __DOXYGEN__
00138 #if (__GNUC__ == 4 && __GNUC_MINOR__ >= 1) || (__GNUC__ > 4)
00139 #  define __INTR_ATTRS __used__, __externally_visible__
00140 #else /* GCC < 4.1 */
00141 #  define __INTR_ATTRS __used__
00142 #endif
00143 #endif /* Doxygen */
00144
00145 #ifdef __cplusplus
00146 #define INTERRUPT(signame)                \
00147 extern "C" void signame(void);              \
00148 void signame (void) __attribute__ ((__interrupt__,__INTR_ATTRS));   \
00149 void signame (void)
00150 #else
00151 #define INTERRUPT(signame)                \
00152 void signame (void) __attribute__ ((__interrupt__,__INTR_ATTRS));   \
00153 void signame (void)
00154 #endif
00155
00156 /**@}*/
00157
00158 /**
00159    \name Obsolete IO macros
00160
00161    Back in a time when AVR-GCC and AVR-LibC could not handle IO port
00162    access in the direct assignment form as they are handled now, all
00163    IO port access had to be done through specific macros that
00164    eventually resulted in inline assembly instructions performing the
00165    desired action.
00166
00167    These macros became obsolete, as reading and writing IO ports can
00168    be done by simply using the IO port name in an expression, and all
00169    bit manipulation (including those on IO ports) can be done using
00170    generic C bit manipulation operators.
00171
00172    The macros in this group simulate the historical behaviour.  While
00173    they are supposed to be applied to IO ports, the emulation actually
00174    uses standard C methods, so they could be applied to arbitrary
00175    memory locations as well.
00176 */
00177
00178 /**@{*/
00179
00180 /**
00181    \ingroup deprecated_items
00182    \def inp(port)
00183    \deprecated
00184
00185    Read a value from an IO port \c port.
00186 */
00187 #define inp(port) (port)
```

```
00188
00189 /**
00190    \ingroup deprecated_items
00191    \def outp(val, port)
00192    \deprecated
00193
00194    Write \c val to IO port \c port.
00195 */
00196 #define outp(val, port) (port) = (val)
00197
00198 /**
00199    \ingroup deprecated_items
00200    \def inb(port)
00201    \deprecated
00202
00203    Read a value from an IO port \c port.
00204 */
00205 #define inb(port) (port)
00206
00207 /**
00208    \ingroup deprecated_items
00209    \def outb(port, val)
00210    \deprecated
00211
00212    Write \c val to IO port \c port.
00213 */
00214 #define outb(port, val) (port) = (val)
00215
00216 /**
00217    \ingroup deprecated_items
00218    \def sbi(port, bit)
00219    \deprecated
00220
00221    Set \c bit in IO port \c port.
00222 */
00223 #define sbi(port, bit) (port) |= (1 « (bit))
00224
00225 /**
00226    \ingroup deprecated_items
00227    \def cbi(port, bit)
00228    \deprecated
00229
00230    Clear \c bit in IO port \c port.
00231 */
00232 #define cbi(port, bit) (port) &= ~(1 « (bit))
00233
00234 /**@}*/
00235
00236 #endif /* _COMPAT_DEPRECATED_H_ */
```

22.54 ina90.h

```
00001 /* Copyright (c) 2002,2004 Marek Michalkiewicz
00002    All rights reserved.
00003
00004    Redistribution and use in source and binary forms, with or without
00005    modification, are permitted provided that the following conditions are met:
00006
00007    * Redistributions of source code must retain the above copyright
00008      notice, this list of conditions and the following disclaimer.
00009
00010    * Redistributions in binary form must reproduce the above copyright
00011      notice, this list of conditions and the following disclaimer in
00012      the documentation and/or other materials provided with the
00013      distribution.
00014
00015    * Neither the name of the copyright holders nor the names of
00016      contributors may be used to endorse or promote products derived
00017      from this software without specific prior written permission.
00018
00019   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
```

```
00020   AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
00021   IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
00022   ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
00023   LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
00024   CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
00025   SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
00026   INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
00027   CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
00028   ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
00029   POSSIBILITY OF SUCH DAMAGE. */
00030
00031 /*
00032    ina90.h
00033
00034    Contributors:
00035      Created by Marek Michalkiewicz <marekm@linux.org.pl>
00036  */
00037
00038 /**
00039    \defgroup compat_ina90 <compat/ina90.h>: Compatibility with IAR EWB 3.x
00040
00041    \code #include <compat/ina90.h> \endcode
00042
00043    This is an attempt to provide some compatibility with
00044    header files that come with IAR C, to make porting applications
00045    between different compilers easier.  No 100% compatibility though.
00046
00047    \note For actual documentation, please see the IAR manual.
00048  */
00049
00050 #ifndef _INA90_H_
00051 #define _INA90_H_ 1
00052
00053 #define _CLI() do { __asm__ __volatile__ ("cli"); } while (0)
00054 #define _SEI() do { __asm__ __volatile__ ("sei"); } while (0)
00055 #define _NOP() do { __asm__ __volatile__ ("nop"); } while (0)
00056 #define _WDR() do { __asm__ __volatile__ ("wdr"); } while (0)
00057 #define _SLEEP() do { __asm__ __volatile__ ("sleep"); } while (0)
00058 #define _OPC(op) do { __asm__ __volatile__ (".word %0" : : "n" (op)); } while (0)
00059
00060 /* _LPM, _ELPM */
00061 #include <avr/pgmspace.h>
00062 #define _LPM(x) do { __LPM(x); } while (0)
00063 #define _ELPM(x) do { __ELPM(x); } while (0)
00064
00065 /* _EEGET, _EEPUT */
00066 #include <avr/eeprom.h>
00067
00068 #define input(port) (port)
00069 #define output(port, val) do { (port) = (val); } while (0)
00070
00071 #define __inp_blk__(port, addr, cnt, op) do {   \
00072     unsigned char __i = (cnt);      \
00073     unsigned char *__addr = (addr);     \
00074     while (__i) {               \
00075         *(__addr op) = input(port); \
00076         __i--;              \
00077     }                   \
00078  } while (0)
00079
00080 #define input_block_inc(port, addr, cnt) __inp_blk__(port, addr, cnt, ++)
00081 #define input_block_dec(port, addr, cnt) __inp_blk__(port, addr, cnt, --)
00082
00083 #define __out_blk__(port, addr, cnt, op) do {   \
00084     unsigned char __i = (cnt);      \
00085     const unsigned char *__addr = (addr);   \
00086     while (__i) {               \
00087         output(port, *(__addr op)); \
00088         __i--;              \
00089     }                   \
00090  } while (0)
00091
00092 #define output_block_inc(port, addr, cnt) __out_blk__(port, addr, cnt, ++)
```

```
00093 #define output_block_dec(port, addr, cnt) __out_blk__(port, addr, cnt, --)
00094
00095 #endif
```

22.55 ctype.h File Reference

Functions

Character classification routines

These functions perform character classification. They return true or false status depending whether the character passed to the function falls into the function's classification (i.e. isdigit() returns true if its argument is any value '0' though '9', inclusive). If the input is not an unsigned char value, all of this function return false.

- int isalnum (int __c)
- int isalpha (int __c)
- int isascii (int __c)
- int isblank (int __c)
- int iscntrl (int __c)
- int isdigit (int __c)
- int isgraph (int __c)
- int islower (int __c)
- int isprint (int __c)
- int ispunct (int __c)
- int isspace (int __c)
- int isupper (int __c)
- int isxdigit (int __c)

Character conversion routines

This realization permits all possible values of integer argument. The toascii() function clears all highest bits. The tolower() and toupper() functions return an input argument as is, if it is not an unsigned char value.

- int toascii (int __c)
- int tolower (int __c)
- int toupper (int __c)

22.56 ctype.h

Go to the documentation of this file.

```
00001 /* Copyright (c) 2002,2007 Michael Stumpf
00002    All rights reserved.
00003
00004    Redistribution and use in source and binary forms, with or without
00005    modification, are permitted provided that the following conditions are met:
00006
00007    * Redistributions of source code must retain the above copyright
00008      notice, this list of conditions and the following disclaimer.
00009
00010    * Redistributions in binary form must reproduce the above copyright
00011      notice, this list of conditions and the following disclaimer in
00012      the documentation and/or other materials provided with the
00013      distribution.
00014
00015    * Neither the name of the copyright holders nor the names of
00016      contributors may be used to endorse or promote products derived
00017      from this software without specific prior written permission.
00018
00019   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
00020   AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
00021   IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
00022   ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
```

```
00023   LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
00024   CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
00025   SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
00026   INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
00027   CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
00028   ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
00029   POSSIBILITY OF SUCH DAMAGE. */
00030
00031 /*
00032    ctype.h - character conversion macros and ctype macros
00033
00034   Author : Michael Stumpf
00035            Michael.Stumpf@t-online.de
00036 */
00037
00038 #ifndef __CTYPE_H_
00039 #define __CTYPE_H_ 1
00040
00041 #include <bits/attribs.h>
00042
00043 #ifdef __cplusplus
00044 extern "C" {
00045 #endif
00046
00047 /** \file */
00048 /** \defgroup ctype <ctype.h>: Character Operations
00049     These functions perform various operations on characters.
00050
00051     \code #include <ctype.h>\endcode
00052
00053 */
00054
00055 /** \name Character classification routines
00056
00057     These functions perform character classification. They return true or
00058     false status depending whether the character passed to the function falls
00059     into the function's classification (i.e. isdigit() returns true if its
00060     argument is any value '0' though '9', inclusive). If the input is not
00061     an unsigned char value, all of this function return false.  */
00062
00063 /**@{*/
00064
00065 /** \ingroup ctype
00066
00067     Checks for an alphanumeric character. It is equivalent to <tt>(isalpha(c)
00068     || isdigit(c))</tt>. */
00069
00070 extern int isalnum(int __c);
00071
00072 #ifndef __DOXYGEN__
00073 extern __ATTR_ALWAYS_INLINE__ __ATTR_GNU_INLINE__
00074 int isalnum(int __c)
00075 {
00076   register int __r24 __asm("r24") = __c;
00077   __asm ("%~call isalnum" : "+r" (__r24));
00078   return __r24;
00079 }
00080 #endif
00081
00082 /** \ingroup ctype
00083
00084     Checks for an alphabetic character. It is equivalent to <tt>(isupper(c) ||
00085     islower(c))</tt>. */
00086
00087 extern int isalpha(int __c);
00088
00089 #ifndef __DOXYGEN__
00090 extern __ATTR_ALWAYS_INLINE__ __ATTR_GNU_INLINE__
00091 int isalpha(int __c)
00092 {
00093   register int __r24 __asm("r24") = __c;
00094   __asm ("%~call isalpha" : "+r" (__r24));
00095   return __r24;
```

```
00096 }
00097 #endif
00098
00099 /** \ingroup ctype
00100
00101     Checks whether \c c is a 7-bit unsigned char value that fits into the
00102     ASCII character set. */
00103
00104 extern int isascii(int __c);
00105
00106 #ifndef __DOXYGEN__
00107 extern __ATTR_ALWAYS_INLINE__ __ATTR_GNU_INLINE__
00108 int isascii(int __c)
00109 {
00110   register int __r24 __asm("r24") = __c;
00111   __asm ("%~call isascii" : "+r" (__r24));
00112   return __r24;
00113 }
00114 #endif
00115
00116 /** \ingroup ctype
00117
00118     Checks for a blank character, that is, a space or a tab. */
00119
00120 extern int isblank(int __c);
00121
00122 #ifndef __DOXYGEN__
00123 extern __ATTR_ALWAYS_INLINE__ __ATTR_GNU_INLINE__
00124 int isblank(int __c)
00125 {
00126   register int __r24 __asm("r24") = __c;
00127   __asm ("%~call isblank" : "+r" (__r24));
00128   return __r24;
00129 }
00130 #endif
00131
00132 /** \ingroup ctype
00133
00134     Checks for a control character. */
00135
00136 extern int iscntrl(int __c);
00137
00138 #ifndef __DOXYGEN__
00139 extern __ATTR_ALWAYS_INLINE__ __ATTR_GNU_INLINE__
00140 int iscntrl(int __c)
00141 {
00142   register int __r24 __asm("r24") = __c;
00143   __asm ("%~call iscntrl" : "+r" (__r24));
00144   return __r24;
00145 }
00146 #endif
00147
00148 /** \ingroup ctype
00149
00150     Checks for a digit (0 through 9). */
00151
00152 extern int isdigit(int __c);
00153
00154 #ifndef __DOXYGEN__
00155 extern __ATTR_ALWAYS_INLINE__ __ATTR_GNU_INLINE__
00156 int isdigit(int __c)
00157 {
00158   register int __r24 __asm("r24") = __c;
00159   __asm ("%~call isdigit" : "+r" (__r24));
00160   return __r24;
00161 }
00162 #endif
00163
00164 /** \ingroup ctype
00165
00166     Checks for any printable character except space. */
00167
00168 extern int isgraph(int __c);
```

```
00169
00170 #ifndef __DOXYGEN__
00171 extern __ATTR_ALWAYS_INLINE__ __ATTR_GNU_INLINE__
00172 int isgraph(int __c)
00173 {
00174   register int __r24 __asm("r24") = __c;
00175   __asm ("%~call isgraph" : "+r" (__r24));
00176   return __r24;
00177 }
00178 #endif
00179
00180 /** \ingroup ctype
00181
00182     Checks for a lower-case character. */
00183
00184 extern int islower(int __c);
00185
00186 #ifndef __DOXYGEN__
00187 extern __ATTR_ALWAYS_INLINE__ __ATTR_GNU_INLINE__
00188 int islower(int __c)
00189 {
00190   register int __r24 __asm("r24") = __c;
00191   __asm ("%~call islower" : "+r" (__r24));
00192   return __r24;
00193 }
00194 #endif
00195
00196 /** \ingroup ctype
00197
00198     Checks for any printable character including space. */
00199
00200 extern int isprint(int __c);
00201
00202 #ifndef __DOXYGEN__
00203 extern __ATTR_ALWAYS_INLINE__ __ATTR_GNU_INLINE__
00204 int isprint(int __c)
00205 {
00206   register int __r24 __asm("r24") = __c;
00207   __asm ("%~call isprint" : "+r" (__r24));
00208   return __r24;
00209 }
00210 #endif
00211
00212 /** \ingroup ctype
00213
00214     Checks for any printable character which is not a space or an alphanumeric
00215     character. */
00216
00217 extern int ispunct(int __c);
00218
00219 #ifndef __DOXYGEN__
00220 extern __ATTR_ALWAYS_INLINE__ __ATTR_GNU_INLINE__
00221 int ispunct(int __c)
00222 {
00223   register int __r24 __asm("r24") = __c;
00224   __asm ("%~call ispunct" : "+r" (__r24));
00225   return __r24;
00226 }
00227 #endif
00228
00229 /** \ingroup ctype
00230
00231     Checks for white-space characters.  For the AVR-LibC library, these are:
00232     space, form-feed ('\\f'), newline ('\\n'), carriage return ('\\r'),
00233     horizontal tab ('\\t'), and vertical tab ('\\v'). */
00234
00235 extern int isspace(int __c);
00236
00237 #ifndef __DOXYGEN__
00238 extern __ATTR_ALWAYS_INLINE__ __ATTR_GNU_INLINE__
00239 int isspace(int __c)
00240 {
00241   register int __r24 __asm("r24") = __c;
```

```
00242   __asm ("%~call isspace" : "+r" (__r24));
00243   return __r24;
00244 }
00245 #endif
00246
00247 /** \ingroup ctype
00248
00249     Checks for an uppercase letter. */
00250
00251 extern int isupper(int __c);
00252
00253 #ifndef __DOXYGEN__
00254 extern __ATTR_ALWAYS_INLINE__ __ATTR_GNU_INLINE__
00255 int isupper(int __c)
00256 {
00257   register int __r24 __asm("r24") = __c;
00258   __asm ("%~call isupper" : "+r" (__r24));
00259   return __r24;
00260 }
00261 #endif
00262
00263 /** \ingroup ctype
00264
00265     Checks for a hexadecimal digits, i.e. one of 0 1 2 3 4 5 6 7 8 9 a b c d e
00266     f A B C D E F. */
00267
00268 extern int isxdigit(int __c);
00269
00270 #ifndef __DOXYGEN__
00271 extern __ATTR_ALWAYS_INLINE__ __ATTR_GNU_INLINE__
00272 int isxdigit(int __c)
00273 {
00274   register int __r24 __asm("r24") = __c;
00275   __asm ("%~call isxdigit" : "+r" (__r24));
00276   return __r24;
00277 }
00278 #endif
00279
00280 /**@}*/
00281
00282 /** \name Character conversion routines
00283
00284     This realization permits all possible values of integer argument.
00285     The toascii() function clears all highest bits. The tolower() and
00286     toupper() functions return an input argument as is, if it is not an
00287     unsigned char value.    */
00288
00289 /**@{*/
00290
00291 /** \ingroup ctype
00292
00293     Converts \c c to a 7-bit unsigned char value that fits into the ASCII
00294     character set, by clearing the high-order bits.
00295
00296     \warning Many people will be unhappy if you use this function. This
00297     function will convert accented letters into random characters. */
00298
00299 extern int toascii(int __c);
00300
00301 #ifndef __DOXYGEN__
00302 extern __ATTR_ALWAYS_INLINE__ __ATTR_GNU_INLINE__
00303 int toascii(int __c)
00304 {
00305   register int __r24 __asm("r24") = __c;
00306   __asm ("%~call toascii" : "+r" (__r24));
00307   return __r24;
00308 }
00309 #endif
00310
00311 /** \ingroup ctype
00312
00313     Converts the letter \c c to lower case, if possible. */
00314
```

```
00315 extern int tolower(int __c);
00316
00317 #ifndef __DOXYGEN__
00318 extern __ATTR_ALWAYS_INLINE__ __ATTR_GNU_INLINE__
00319 int tolower(int __c)
00320 {
00321   register int __r24 __asm("r24") = __c;
00322   __asm ("%~call tolower" : "+r" (__r24));
00323   return __r24;
00324 }
00325 #endif
00326
00327 /** \ingroup ctype
00328
00329     Converts the letter \c c to upper case, if possible. */
00330
00331 extern int toupper(int __c);
00332
00333 #ifndef __DOXYGEN__
00334 extern __ATTR_ALWAYS_INLINE__ __ATTR_GNU_INLINE__
00335 int toupper(int __c)
00336 {
00337   register int __r24 __asm("r24") = __c;
00338   __asm ("%~call toupper" : "+r" (__r24));
00339   return __r24;
00340 }
00341 #endif
00342
00343 /**@}*/
00344
00345 #ifdef __cplusplus
00346 }
00347 #endif
00348
00349 #endif
```

22.57 errno.h File Reference

Macros

- #define EDOM 33
- #define ERANGE 34
- #define EINVAL 35

Variables

- int errno

22.58 errno.h

Go to the documentation of this file.

```
00001 /* Copyright (c) 2002,2007 Marek Michalkiewicz
00002    All rights reserved.
00003
00004    Redistribution and use in source and binary forms, with or without
00005    modification, are permitted provided that the following conditions are met:
00006
00007    * Redistributions of source code must retain the above copyright
00008      notice, this list of conditions and the following disclaimer.
00009
00010    * Redistributions in binary form must reproduce the above copyright
00011      notice, this list of conditions and the following disclaimer in
```

```
00012      the documentation and/or other materials provided with the
00013      distribution.
00014
00015    * Neither the name of the copyright holders nor the names of
00016      contributors may be used to endorse or promote products derived
00017      from this software without specific prior written permission.
00018
00019  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
00020  AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
00021  IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
00022  ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
00023  LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
00024  CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
00025  SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
00026  INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
00027  CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
00028  ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
00029  POSSIBILITY OF SUCH DAMAGE. */
00030
00031 #ifndef __ERRNO_H_
00032 #define __ERRNO_H_ 1
00033
00034 /** \file */
00035 /** \defgroup avr_errno <errno.h>: System Errors
00036
00037     \code #include <errno.h>\endcode
00038
00039     Some functions in the library set the global variable \c errno when an
00040     error occurs. The file, \c <errno.h>, provides symbolic names for various
00041     error codes.
00042  */
00043
00044 #ifdef __cplusplus
00045 extern "C" {
00046 #endif
00047
00048 /** \ingroup avr_errno
00049     \brief Error code for last error encountered by library
00050
00051     The variable \c errno holds the last error code encountered by
00052     a library function.  This variable must be cleared by the
00053     user prior to calling a library function.
00054
00055     \warning The \c errno global variable is not safe to use in a threaded or
00056     multi-task system. A race condition can occur if a task is interrupted
00057     between the call which sets \c error and when the task examines \c
00058     errno. If another task changes \c errno during this time, the result will
00059     be incorrect for the interrupted task. */
00060 #ifndef __ASSEMBLER__
00061 extern int errno;
00062 #endif
00063
00064 #ifdef __cplusplus
00065 }
00066 #endif
00067
00068 /** \ingroup avr_errno
00069     \def EDOM
00070
00071     Domain error. */
00072 #define EDOM       33
00073
00074 /** \ingroup avr_errno
00075     \def ERANGE
00076
00077     Range error. */
00078 #define ERANGE     34
00079
00080 /** \ingroup avr_errno
00081     \def EINVAL
00082
00083     Invalid value error. */
00084 #define EINVAL     35
```

```
00085
00086 #ifndef __DOXYGEN__
00087
00088 #define ENOSYS 36
00089 #define EINTR  37
00090 #define ENOERR 38
00091
00092 #define E2BIG ENOERR
00093 #define EACCES ENOERR
00094 #define EADDRINUSE ENOERR
00095 #define EADDRNOTAVAIL ENOERR
00096 #define EAFNOSUPPORT ENOERR
00097 #define EAGAIN ENOERR
00098 #define EALREADY ENOERR
00099 #define EBADF ENOERR
00100 #define EBUSY ENOERR
00101 #define ECHILD ENOERR
00102 #define ECONNABORTED ENOERR
00103 #define ECONNREFUSED ENOERR
00104 #define ECONNRESET ENOERR
00105 #define EDEADLK ENOERR
00106 #define EDESTADDRREQ ENOERR
00107 #define EEXIST ENOERR
00108 #define EFAULT ENOERR
00109 #define EFBIG ENOERR
00110 #define EHOSTUNREACH ENOERR
00111 #define EILSEQ ENOERR
00112 #define EINPROGRESS ENOERR
00113 #define EIO ENOERR
00114 #define EISCONN ENOERR
00115 #define EISDIR ENOERR
00116 #define ELOOP ENOERR
00117 #define EMFILE ENOERR
00118 #define EMLINK ENOERR
00119 #define EMSGSIZE ENOERR
00120 #define ENAMETOOLONG ENOERR
00121 #define ENETDOWN ENOERR
00122 #define ENETRESET ENOERR
00123 #define ENETUNREACH ENOERR
00124 #define ENFILE ENOERR
00125 #define ENOBUFS ENOERR
00126 #define ENODEV ENOERR
00127 #define ENOENT ENOERR
00128 #define ENOEXEC ENOERR
00129 #define ENOLCK ENOERR
00130 #define ENOMEM ENOERR
00131 #define ENOMSG ENOERR
00132 #define ENOPROTOOPT ENOERR
00133 #define ENOSPC ENOERR
00134 #define ENOTCONN ENOERR
00135 #define ENOTDIR ENOERR
00136 #define ENOTEMPTY ENOERR
00137 #define ENOTSOCK ENOERR
00138 #define ENOTTY ENOERR
00139 #define ENXIO ENOERR
00140 #define EOPNOTSUPP ENOERR
00141 #define EPERM ENOERR
00142 #define EPIPE ENOERR
00143 #define EPROTONOSUPPORT ENOERR
00144 #define EPROTOTYPE ENOERR
00145 #define EROFS ENOERR
00146 #define ESPIPE ENOERR
00147 #define ESRCH ENOERR
00148 #define ETIMEDOUT ENOERR
00149 #define EWOULDBLOCK ENOERR
00150 #define EXDEV ENOERR
00151
00152 #endif  /* !__DOXYGEN__ */
00153
00154 #endif
```

22.59 inttypes.h File Reference

Macros

macros for printf and scanf format specifiers

For C++, these are only included if __STDC_LIMIT_MACROS is defined before including <inttypes.h>.

- #define PRId8 "d"
- #define PRIdLEAST8 "d"
- #define PRIdFAST8 "d"
- #define PRIi8 "i"
- #define PRIiLEAST8 "i"
- #define PRIiFAST8 "i"
- #define PRId16 "d"
- #define PRIdLEAST16 "d"
- #define PRIdFAST16 "d"
- #define PRIi16 "i"
- #define PRIiLEAST16 "i"
- #define PRIiFAST16 "i"
- #define PRId32 "ld"
- #define PRIdLEAST32 "ld"
- #define PRIdFAST32 "ld"
- #define PRIi32 "li"
- #define PRIiLEAST32 "li"
- #define PRIiFAST32 "li"
- #define PRIdPTR PRId16
- #define PRIiPTR PRIi16
- #define PRIo8 "o"
- #define PRIoLEAST8 "o"
- #define PRIoFAST8 "o"
- #define PRIu8 "u"
- #define PRIuLEAST8 "u"
- #define PRIuFAST8 "u"
- #define PRIx8 "x"
- #define PRIxLEAST8 "x"
- #define PRIxFAST8 "x"
- #define PRIX8 "X"
- #define PRIXLEAST8 "X"
- #define PRIXFAST8 "X"
- #define PRIo16 "o"
- #define PRIoLEAST16 "o"
- #define PRIoFAST16 "o"
- #define PRIu16 "u"
- #define PRIuLEAST16 "u"
- #define PRIuFAST16 "u"
- #define PRIx16 "x"
- #define PRIxLEAST16 "x"
- #define PRIxFAST16 "x"
- #define PRIX16 "X"
- #define PRIXLEAST16 "X"
- #define PRIXFAST16 "X"
- #define PRIo32 "lo"
- #define PRIoLEAST32 "lo"
- #define PRIoFAST32 "lo"
- #define PRIu32 "lu"
- #define PRIuLEAST32 "lu"
- #define PRIuFAST32 "lu"
- #define PRIx32 "lx"
- #define PRIxLEAST32 "lx"
- #define PRIxFAST32 "lx"
- #define PRIX32 "lX"
- #define PRIXLEAST32 "lX"
- #define PRIXFAST32 "lX"

- #define PRIoPTR PRIo16
- #define PRIuPTR PRIu16
- #define PRIxPTR PRIx16
- #define PRIXPTR PRIX16
- #define SCNd8 "hhd"
- #define SCNdLEAST8 "hhd"
- #define SCNdFAST8 "hhd"
- #define SCNi8 "hhi"
- #define SCNiLEAST8 "hhi"
- #define SCNiFAST8 "hhi"
- #define SCNd16 "d"
- #define SCNdLEAST16 "d"
- #define SCNdFAST16 "d"
- #define SCNi16 "i"
- #define SCNiLEAST16 "i"
- #define SCNiFAST16 "i"
- #define SCNd32 "ld"
- #define SCNdLEAST32 "ld"
- #define SCNdFAST32 "ld"
- #define SCNi32 "li"
- #define SCNiLEAST32 "li"
- #define SCNiFAST32 "li"
- #define SCNdPTR SCNd16
- #define SCNiPTR SCNi16
- #define SCNo8 "hho"
- #define SCNoLEAST8 "hho"
- #define SCNoFAST8 "hho"
- #define SCNu8 "hhu"
- #define SCNuLEAST8 "hhu"
- #define SCNuFAST8 "hhu"
- #define SCNx8 "hhx"
- #define SCNxLEAST8 "hhx"
- #define SCNxFAST8 "hhx"
- #define SCNo16 "o"
- #define SCNoLEAST16 "o"
- #define SCNoFAST16 "o"
- #define SCNu16 "u"
- #define SCNuLEAST16 "u"
- #define SCNuFAST16 "u"
- #define SCNx16 "x"
- #define SCNxLEAST16 "x"
- #define SCNxFAST16 "x"
- #define SCNo32 "lo"
- #define SCNoLEAST32 "lo"
- #define SCNoFAST32 "lo"
- #define SCNu32 "lu"
- #define SCNuLEAST32 "lu"
- #define SCNuFAST32 "lu"
- #define SCNx32 "lx"
- #define SCNxLEAST32 "lx"
- #define SCNxFAST32 "lx"
- #define SCNoPTR SCNo16
- #define SCNuPTR SCNu16
- #define SCNxPTR SCNx16

Typedefs

Far pointers for memory access > 64K

- typedef int32_t int_farptr_t
- typedef uint32_t uint_farptr_t

22.60 inttypes.h

Go to the documentation of this file.

```
00001 /* Copyright (c) 2004,2005,2007,2012 Joerg Wunsch
00002    Copyright (c) 2005, Carlos Lamas
00003    All rights reserved.
00004
00005    Redistribution and use in source and binary forms, with or without
00006    modification, are permitted provided that the following conditions are met:
00007
00008    * Redistributions of source code must retain the above copyright
00009      notice, this list of conditions and the following disclaimer.
00010
00011    * Redistributions in binary form must reproduce the above copyright
00012      notice, this list of conditions and the following disclaimer in
00013      the documentation and/or other materials provided with the
00014      distribution.
00015
00016    * Neither the name of the copyright holders nor the names of
00017      contributors may be used to endorse or promote products derived
00018      from this software without specific prior written permission.
00019
00020   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
00021   AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
00022   IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
00023   ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
00024   LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
00025   CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
00026   SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
00027   INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
00028   CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
00029   ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
00030   POSSIBILITY OF SUCH DAMAGE. */
00031
00032 #ifndef __INTTYPES_H_
00033 #define __INTTYPES_H_
00034
00035 #include <stdint.h>
00036 #include <bits/attribs.h>
00037
00038 /** \file */
00039 /** \defgroup avr_inttypes <inttypes.h>: Integer Type conversions
00040     \code #include <inttypes.h> \endcode
00041
00042     This header file includes the exact-width integer definitions from
00043     <tt><stdint.h></tt>, and extends them with additional facilities
00044     provided by the implementation.
00045
00046     Currently, the extensions include two additional integer types
00047     that could hold a "far" pointer (i.e. a code pointer that can
00048     address more than 64 KB), as well as standard names for all printf
00049     and scanf formatting options that are supported by the \ref avr_stdio.
00050     As the library does not support the full range of conversion
00051     specifiers from ISO 9899:1999, only those conversions that are
00052     actually implemented will be listed here.
00053
00054     The idea behind these conversion macros is that, for each of the
00055     types defined by <stdint.h>, a macro will be supplied that portably
00056     allows formatting an object of that type in printf() or scanf()
00057     operations.  Example:
00058
00059     \code
00060     #include <inttypes.h>
00061
00062     uint8_t smallval;
00063     int32_t longval;
00064     ...
00065     printf("The hexadecimal value of smallval is %" PRIx8
00066            ", the decimal value of longval is %" PRId32 ".\n",
00067        smallval, longval);
00068     \endcode
00069 */
```

```
00070
00071
00072 /** \name Far pointers for memory access > 64K */
00073
00074 /**@{*/
00075 /** \ingroup avr_inttypes
00076     signed integer type that can hold a pointer > 64 KiB */
00077 typedef int32_t int_farptr_t;
00078
00079 /** \ingroup avr_inttypes
00080     unsigned integer type that can hold a pointer > 64 KiB,
00081     see also pgm_get_far_address()
00082  */
00083 typedef uint32_t uint_farptr_t;
00084 /**@}*/
00085
00086 #if !defined(__cplusplus) || defined(__STDC_LIMIT_MACROS)
00087
00088
00089 /** \name macros for printf and scanf format specifiers
00090
00091     For C++, these are only included if __STDC_LIMIT_MACROS
00092     is defined before including <inttypes.h>.
00093  */
00094
00095 /**@{*/
00096 /** \ingroup avr_inttypes
00097     decimal printf format for int8_t */
00098 #define     PRId8           "d"
00099 /** \ingroup avr_inttypes
00100     decimal printf format for int_least8_t */
00101 #define     PRIdLEAST8      "d"
00102 /** \ingroup avr_inttypes
00103     decimal printf format for int_fast8_t */
00104 #define     PRIdFAST8       "d"
00105
00106 /** \ingroup avr_inttypes
00107     integer printf format for int8_t */
00108 #define     PRIi8           "i"
00109 /** \ingroup avr_inttypes
00110     integer printf format for int_least8_t */
00111 #define     PRIiLEAST8      "i"
00112 /** \ingroup avr_inttypes
00113     integer printf format for int_fast8_t */
00114 #define     PRIiFAST8       "i"
00115
00116
00117 /** \ingroup avr_inttypes
00118     decimal printf format for int16_t */
00119 #define     PRId16          "d"
00120 /** \ingroup avr_inttypes
00121     decimal printf format for int_least16_t */
00122 #define     PRIdLEAST16     "d"
00123 /** \ingroup avr_inttypes
00124     decimal printf format for int_fast16_t */
00125 #define     PRIdFAST16      "d"
00126
00127 /** \ingroup avr_inttypes
00128     integer printf format for int16_t */
00129 #define     PRIi16          "i"
00130 /** \ingroup avr_inttypes
00131     integer printf format for int_least16_t */
00132 #define     PRIiLEAST16     "i"
00133 /** \ingroup avr_inttypes
00134     integer printf format for int_fast16_t */
00135 #define     PRIiFAST16      "i"
00136
00137
00138 /** \ingroup avr_inttypes
00139     decimal printf format for int32_t */
00140 #define     PRId32          "ld"
00141 /** \ingroup avr_inttypes
00142     decimal printf format for int_least32_t */
```

```
00143 #define     PRIdLEAST32     "ld"
00144 /** \ingroup avr_inttypes
00145     decimal printf format for int_fast32_t */
00146 #define     PRIdFAST32      "ld"
00147
00148 /** \ingroup avr_inttypes
00149     integer printf format for int32_t */
00150 #define     PRIi32          "li"
00151 /** \ingroup avr_inttypes
00152     integer printf format for int_least32_t */
00153 #define     PRIiLEAST32     "li"
00154 /** \ingroup avr_inttypes
00155     integer printf format for int_fast32_t */
00156 #define     PRIiFAST32      "li"
00157
00158
00159 #ifdef __avr_libc_does_not_implement_long_long_in_printf_or_scanf
00160
00161 #define     PRId64          "lld"
00162 #define     PRIdLEAST64     "lld"
00163 #define     PRIdFAST64      "lld"
00164
00165 #define     PRIi64          "lli"
00166 #define     PRIiLEAST64     "lli"
00167 #define     PRIiFAST64      "lli"
00168
00169
00170 #define     PRIdMAX         "lld"
00171 #define     PRIiMAX         "lli"
00172
00173 #endif
00174
00175 /** \ingroup avr_inttypes
00176     decimal printf format for intptr_t */
00177 #define     PRIdPTR         PRId16
00178 /** \ingroup avr_inttypes
00179     integer printf format for intptr_t */
00180 #define     PRIiPTR         PRIi16
00181
00182 /** \ingroup avr_inttypes
00183     octal printf format for uint8_t */
00184 #define     PRIo8           "o"
00185 /** \ingroup avr_inttypes
00186     octal printf format for uint_least8_t */
00187 #define     PRIoLEAST8      "o"
00188 /** \ingroup avr_inttypes
00189     octal printf format for uint_fast8_t */
00190 #define     PRIoFAST8       "o"
00191
00192 /** \ingroup avr_inttypes
00193     decimal printf format for uint8_t */
00194 #define     PRIu8           "u"
00195 /** \ingroup avr_inttypes
00196     decimal printf format for uint_least8_t */
00197 #define     PRIuLEAST8      "u"
00198 /** \ingroup avr_inttypes
00199     decimal printf format for uint_fast8_t */
00200 #define     PRIuFAST8       "u"
00201
00202 /** \ingroup avr_inttypes
00203     hexadecimal printf format for uint8_t */
00204 #define     PRIx8           "x"
00205 /** \ingroup avr_inttypes
00206     hexadecimal printf format for uint_least8_t */
00207 #define     PRIxLEAST8      "x"
00208 /** \ingroup avr_inttypes
00209     hexadecimal printf format for uint_fast8_t */
00210 #define     PRIxFAST8       "x"
00211
00212 /** \ingroup avr_inttypes
00213     uppercase hexadecimal printf format for uint8_t */
00214 #define     PRIX8           "X"
00215 /** \ingroup avr_inttypes
```

```
00216     uppercase hexadecimal printf format for uint_least8_t */
00217 #define     PRIXLEAST8      "X"
00218 /** \ingroup avr_inttypes
00219     uppercase hexadecimal printf format for uint_fast8_t */
00220 #define     PRIXFAST8       "X"
00221
00222
00223 /** \ingroup avr_inttypes
00224     octal printf format for uint16_t */
00225 #define     PRIo16          "o"
00226 /** \ingroup avr_inttypes
00227     octal printf format for uint_least16_t */
00228 #define     PRIoLEAST16     "o"
00229 /** \ingroup avr_inttypes
00230     octal printf format for uint_fast16_t */
00231 #define     PRIoFAST16      "o"
00232
00233 /** \ingroup avr_inttypes
00234     decimal printf format for uint16_t */
00235 #define     PRIu16          "u"
00236 /** \ingroup avr_inttypes
00237     decimal printf format for uint_least16_t */
00238 #define     PRIuLEAST16     "u"
00239 /** \ingroup avr_inttypes
00240     decimal printf format for uint_fast16_t */
00241 #define     PRIuFAST16      "u"
00242
00243 /** \ingroup avr_inttypes
00244     hexadecimal printf format for uint16_t */
00245 #define     PRIx16          "x"
00246 /** \ingroup avr_inttypes
00247     hexadecimal printf format for uint_least16_t */
00248 #define     PRIxLEAST16     "x"
00249 /** \ingroup avr_inttypes
00250     hexadecimal printf format for uint_fast16_t */
00251 #define     PRIxFAST16      "x"
00252
00253 /** \ingroup avr_inttypes
00254     uppercase hexadecimal printf format for uint16_t */
00255 #define     PRIX16          "X"
00256 /** \ingroup avr_inttypes
00257     uppercase hexadecimal printf format for uint_least16_t */
00258 #define     PRIXLEAST16     "X"
00259 /** \ingroup avr_inttypes
00260     uppercase hexadecimal printf format for uint_fast16_t */
00261 #define     PRIXFAST16      "X"
00262
00263
00264 /** \ingroup avr_inttypes
00265     octal printf format for uint32_t */
00266 #define     PRIo32          "lo"
00267 /** \ingroup avr_inttypes
00268     octal printf format for uint_least32_t */
00269 #define     PRIoLEAST32     "lo"
00270 /** \ingroup avr_inttypes
00271     octal printf format for uint_fast32_t */
00272 #define     PRIoFAST32      "lo"
00273
00274 /** \ingroup avr_inttypes
00275     decimal printf format for uint32_t */
00276 #define     PRIu32          "lu"
00277 /** \ingroup avr_inttypes
00278     decimal printf format for uint_least32_t */
00279 #define     PRIuLEAST32     "lu"
00280 /** \ingroup avr_inttypes
00281     decimal printf format for uint_fast32_t */
00282 #define     PRIuFAST32      "lu"
00283
00284 /** \ingroup avr_inttypes
00285     hexadecimal printf format for uint32_t */
00286 #define     PRIx32          "lx"
00287 /** \ingroup avr_inttypes
00288     hexadecimal printf format for uint_least32_t */
```

```
00289 #define     PRIxLEAST32     "lx"
00290 /** \ingroup avr_inttypes
00291     hexadecimal printf format for uint_fast32_t */
00292 #define     PRIxFAST32      "lx"
00293
00294 /** \ingroup avr_inttypes
00295     uppercase hexadecimal printf format for uint32_t */
00296 #define     PRIX32          "lX"
00297 /** \ingroup avr_inttypes
00298     uppercase hexadecimal printf format for uint_least32_t */
00299 #define     PRIXLEAST32     "lX"
00300 /** \ingroup avr_inttypes
00301     uppercase hexadecimal printf format for uint_fast32_t */
00302 #define     PRIXFAST32      "lX"
00303
00304
00305 #ifdef __avr_libc_does_not_implement_long_long_in_printf_or_scanf
00306
00307 #define     PRIo64          "llo"
00308 #define     PRIoLEAST64     "llo"
00309 #define     PRIoFAST64      "llo"
00310
00311 #define     PRIu64          "llu"
00312 #define     PRIuLEAST64     "llu"
00313 #define     PRIuFAST64      "llu"
00314
00315 #define     PRIx64          "llx"
00316 #define     PRIxLEAST64     "llx"
00317 #define     PRIxFAST64      "llx"
00318
00319 #define     PRIX64          "llX"
00320 #define     PRIXLEAST64     "llX"
00321 #define     PRIXFAST64      "llX"
00322
00323 #define     PRIoMAX         "llo"
00324 #define     PRIuMAX         "llu"
00325 #define     PRIxMAX         "llx"
00326 #define     PRIXMAX         "llX"
00327
00328 #endif
00329
00330 /** \ingroup avr_inttypes
00331     octal printf format for uintptr_t */
00332 #define     PRIoPTR         PRIo16
00333 /** \ingroup avr_inttypes
00334     decimal printf format for uintptr_t */
00335 #define     PRIuPTR         PRIu16
00336 /** \ingroup avr_inttypes
00337     hexadecimal printf format for uintptr_t */
00338 #define     PRIxPTR         PRIx16
00339 /** \ingroup avr_inttypes
00340     uppercase hexadecimal printf format for uintptr_t */
00341 #define     PRIXPTR         PRIX16
00342
00343
00344 /** \ingroup avr_inttypes
00345     decimal scanf format for int8_t */
00346 #define     SCNd8           "hhd"
00347 /** \ingroup avr_inttypes
00348     decimal scanf format for int_least8_t */
00349 #define     SCNdLEAST8      "hhd"
00350 /** \ingroup avr_inttypes
00351     decimal scanf format for int_fast8_t */
00352 #define     SCNdFAST8       "hhd"
00353
00354 /** \ingroup avr_inttypes
00355     generic-integer scanf format for int8_t */
00356 #define     SCNi8           "hhi"
00357 /** \ingroup avr_inttypes
00358     generic-integer scanf format for int_least8_t */
00359 #define     SCNiLEAST8      "hhi"
00360 /** \ingroup avr_inttypes
00361     generic-integer scanf format for int_fast8_t */
```

```
00362 #define     SCNiFAST8       "hhi"
00363
00364
00365 /** \ingroup avr_inttypes
00366     decimal scanf format for int16_t */
00367 #define     SCNd16          "d"
00368 /** \ingroup avr_inttypes
00369     decimal scanf format for int_least16_t */
00370 #define     SCNdLEAST16     "d"
00371 /** \ingroup avr_inttypes
00372     decimal scanf format for int_fast16_t */
00373 #define     SCNdFAST16      "d"
00374
00375 /** \ingroup avr_inttypes
00376     generic-integer scanf format for int16_t */
00377 #define     SCNi16          "i"
00378 /** \ingroup avr_inttypes
00379     generic-integer scanf format for int_least16_t */
00380 #define     SCNiLEAST16     "i"
00381 /** \ingroup avr_inttypes
00382     generic-integer scanf format for int_fast16_t */
00383 #define     SCNiFAST16      "i"
00384
00385
00386 /** \ingroup avr_inttypes
00387     decimal scanf format for int32_t */
00388 #define     SCNd32          "ld"
00389 /** \ingroup avr_inttypes
00390     decimal scanf format for int_least32_t */
00391 #define     SCNdLEAST32     "ld"
00392 /** \ingroup avr_inttypes
00393     decimal scanf format for int_fast32_t */
00394 #define     SCNdFAST32      "ld"
00395
00396 /** \ingroup avr_inttypes
00397     generic-integer scanf format for int32_t */
00398 #define     SCNi32          "li"
00399 /** \ingroup avr_inttypes
00400     generic-integer scanf format for int_least32_t */
00401 #define     SCNiLEAST32     "li"
00402 /** \ingroup avr_inttypes
00403     generic-integer scanf format for int_fast32_t */
00404 #define     SCNiFAST32      "li"
00405
00406
00407 #ifdef __avr_libc_does_not_implement_long_long_in_printf_or_scanf
00408
00409 #define     SCNd64          "lld"
00410 #define     SCNdLEAST64     "lld"
00411 #define     SCNdFAST64      "lld"
00412
00413 #define     SCNi64          "lli"
00414 #define     SCNiLEAST64     "lli"
00415 #define     SCNiFAST64      "lli"
00416
00417 #define     SCNdMAX         "lld"
00418 #define     SCNiMAX         "lli"
00419
00420 #endif
00421
00422 /** \ingroup avr_inttypes
00423     decimal scanf format for intptr_t */
00424 #define     SCNdPTR         SCNd16
00425 /** \ingroup avr_inttypes
00426     generic-integer scanf format for intptr_t */
00427 #define     SCNiPTR         SCNi16
00428
00429 /** \ingroup avr_inttypes
00430     octal scanf format for uint8_t */
00431 #define     SCNo8           "hho"
00432 /** \ingroup avr_inttypes
00433     octal scanf format for uint_least8_t */
00434 #define     SCNoLEAST8      "hho"
```

```
00435 /** \ingroup avr_inttypes
00436     octal scanf format for uint_fast8_t */
00437 #define     SCNoFAST8       "hho"
00438
00439 /** \ingroup avr_inttypes
00440     decimal scanf format for uint8_t */
00441 #define     SCNu8           "hhu"
00442 /** \ingroup avr_inttypes
00443     decimal scanf format for uint_least8_t */
00444 #define     SCNuLEAST8      "hhu"
00445 /** \ingroup avr_inttypes
00446     decimal scanf format for uint_fast8_t */
00447 #define     SCNuFAST8       "hhu"
00448
00449 /** \ingroup avr_inttypes
00450     hexadecimal scanf format for uint8_t */
00451 #define     SCNx8           "hhx"
00452 /** \ingroup avr_inttypes
00453     hexadecimal scanf format for uint_least8_t */
00454 #define     SCNxLEAST8      "hhx"
00455 /** \ingroup avr_inttypes
00456     hexadecimal scanf format for uint_fast8_t */
00457 #define     SCNxFAST8       "hhx"
00458
00459 /** \ingroup avr_inttypes
00460     octal scanf format for uint16_t */
00461 #define     SCNo16          "o"
00462 /** \ingroup avr_inttypes
00463     octal scanf format for uint_least16_t */
00464 #define     SCNoLEAST16     "o"
00465 /** \ingroup avr_inttypes
00466     octal scanf format for uint_fast16_t */
00467 #define     SCNoFAST16      "o"
00468
00469 /** \ingroup avr_inttypes
00470     decimal scanf format for uint16_t */
00471 #define     SCNu16          "u"
00472 /** \ingroup avr_inttypes
00473     decimal scanf format for uint_least16_t */
00474 #define     SCNuLEAST16     "u"
00475 /** \ingroup avr_inttypes
00476     decimal scanf format for uint_fast16_t */
00477 #define     SCNuFAST16      "u"
00478
00479 /** \ingroup avr_inttypes
00480     hexadecimal scanf format for uint16_t */
00481 #define     SCNx16          "x"
00482 /** \ingroup avr_inttypes
00483     hexadecimal scanf format for uint_least16_t */
00484 #define     SCNxLEAST16     "x"
00485 /** \ingroup avr_inttypes
00486     hexadecimal scanf format for uint_fast16_t */
00487 #define     SCNxFAST16      "x"
00488
00489
00490 /** \ingroup avr_inttypes
00491     octal scanf format for uint32_t */
00492 #define     SCNo32          "lo"
00493 /** \ingroup avr_inttypes
00494     octal scanf format for uint_least32_t */
00495 #define     SCNoLEAST32     "lo"
00496 /** \ingroup avr_inttypes
00497     octal scanf format for uint_fast32_t */
00498 #define     SCNoFAST32      "lo"
00499
00500 /** \ingroup avr_inttypes
00501     decimal scanf format for uint32_t */
00502 #define     SCNu32          "lu"
00503 /** \ingroup avr_inttypes
00504     decimal scanf format for uint_least32_t */
00505 #define     SCNuLEAST32     "lu"
00506 /** \ingroup avr_inttypes
00507     decimal scanf format for uint_fast32_t */
```

```
00508 #define     SCNuFAST32      "lu"
00509
00510 /** \ingroup avr_inttypes
00511     hexadecimal scanf format for uint32_t */
00512 #define     SCNx32          "lx"
00513 /** \ingroup avr_inttypes
00514     hexadecimal scanf format for uint_least32_t */
00515 #define     SCNxLEAST32     "lx"
00516 /** \ingroup avr_inttypes
00517     hexadecimal scanf format for uint_fast32_t */
00518 #define     SCNxFAST32      "lx"
00519
00520
00521 #ifdef __avr_libc_does_not_implement_long_long_in_printf_or_scanf
00522
00523 #define     SCNo64          "llo"
00524 #define     SCNoLEAST64     "llo"
00525 #define     SCNoFAST64      "llo"
00526
00527 #define     SCNu64          "llu"
00528 #define     SCNuLEAST64     "llu"
00529 #define     SCNuFAST64      "llu"
00530
00531 #define     SCNx64          "llx"
00532 #define     SCNxLEAST64     "llx"
00533 #define     SCNxFAST64      "llx"
00534
00535 #define     SCNoMAX         "llo"
00536 #define     SCNuMAX         "llu"
00537 #define     SCNxMAX         "llx"
00538
00539 #endif
00540
00541 /** \ingroup avr_inttypes
00542     octal scanf format for uintptr_t */
00543 #define     SCNoPTR         SCNo16
00544 /** \ingroup avr_inttypes
00545     decimal scanf format for uintptr_t */
00546 #define     SCNuPTR         SCNu16
00547 /** \ingroup avr_inttypes
00548     hexadecimal scanf format for uintptr_t */
00549 #define     SCNxPTR         SCNx16
00550
00551 /**@}*/
00552
00553
00554 #endif  /* !defined(__cplusplus) || defined(__STDC_LIMIT_MACROS) */
00555
00556
00557 #endif /* __INTTYPES_H_ */
```

22.61 math.h File Reference

Macros

- #define M_E 2.7182818284590452354
- #define M_LOG2E 1.4426950408889634074
- #define M_LOG10E 0.43429448190325182765
- #define M_LN2 0.69314718055994530942
- #define M_LN10 2.30258509299404568402
- #define M_PI 3.14159265358979323846
- #define M_PI_2 1.57079632679489661923
- #define M_PI_4 0.78539816339744830962
- #define M_1_PI 0.31830988618379067154
- #define M_2_PI 0.63661977236758134308
- #define M_2_SQRTPI 1.12837916709551257390

- #define M_SQRT2 1.41421356237309504880
- #define M_SQRT1_2 0.70710678118654752440
- #define NAN __builtin_nan("")
- #define nanf(__tag) __builtin_nanf(__tag)
- #define nan(__tag) __builtin_nan(__tag)
- #define nanl(__tag) __builtin_nanl(__tag)
- #define INFINITY __builtin_inff()
- #define HUGE_VALF __builtin_huge_valf()
- #define HUGE_VAL __builtin_huge_val()
- #define HUGE_VALL __builtin_huge_vall()

Functions

- float cosf (float x)
- double cos (double x)
- long double cosl (long double x)
- float sinf (float x)
- double sin (double x)
- long double sinl (long double x)
- void sincosf (float x, float *psin, float *pcos)
- void sincos (double x, double *psin, double *pcos)
- void sincosl (long double x, long double *psin, long double *pcos)
- float tanf (float x)
- double tan (double x)
- long double tanl (long double x)
- float fabsf (float __x)
- double fabs (double __x)
- long double fabsl (long double __x)
- float fmodf (float x, float y)
- double fmod (double x, double y)
- long double fmodl (long double x, long double y)
- float modff (float x, float *iptr)
- double modf (double x, double *iptr)
- long double modfl (long double x, long double *iptr)
- float sqrtf (float x)
- double sqrt (double x)
- long double sqrtl (long double x)
- float cbrtf (float x)
- double cbrt (double x)
- long double cbrtl (long double x)
- float hypotf (float x, float y)
- double hypot (double x, double y)
- long double hypotl (long double x, long double y)
- float floorf (float x)
- double floor (double x)
- long double floorl (long double x)
- float ceilf (float x)
- double ceil (double x)
- long double ceill (long double x)
- float frexpf (float x, int *pexp)
- double frexp (double x, int *pexp)
- long double frexpl (long double x, int *pexp)
- float ldexpf (float x, int iexp)
- double ldexp (double x, int iexp)

- long double ldexpl (long double x, int iexp)
- float expf (float x)
- double exp (double x)
- long double expl (long double x)
- float coshf (float x)
- double cosh (double x)
- long double coshl (long double x)
- float sinhf (float x)
- double sinh (double x)
- long double sinhl (long double x)
- float tanhf (float x)
- double tanh (double x)
- long double tanhl (long double x)
- float acosf (float x)
- double acos (double x)
- long double acosl (long double x)
- float asinf (float x)
- double asin (double x)
- long double asinl (long double x)
- float atanf (float x)
- double atan (double x)
- long double atanl (long double x)
- float atan2f (float y, float x)
- double atan2 (double y, double x)
- long double atan2l (long double y, long double x)
- float logf (float x)
- double log (double x)
- long double logl (long double x)
- float log10f (float x)
- double log10 (double x)
- long double log10l (long double x)
- float log2f (float x)
- double log2 (double x)
- long double log2l (long double x)
- float powf (float x, float y)
- double pow (double x, double y)
- long double powl (long double x, long double y)
- int isnanf (float x)
- int isnan (double x)
- int isnanl (long double x)
- int isinff (float x)
- int isinf (double x)
- int isinfl (long double x)
- static int isfinitef (float __x)
- static int isfinite (double __x)
- static int isfinitel (long double __x)
- static float copysignf (float __x, float __y)
- static double copysign (double __x, double __y)
- static long double copysignl (long double __x, long double __y)
- int signbitf (float x)
- int signbit (double x)
- int signbitl (long double x)
- float fdimf (float x, float y)
- double fdim (double x, double y)
- long double fdiml (long double x, long double y)

- float fmaf (float x, float y, float z)
- double fma (double x, double y, double z)
- long double fmal (long double x, long double y, long double z)
- float fmaxf (float x, float y)
- double fmax (double x, double y)
- long double fmaxl (long double x, long double y)
- float fminf (float x, float y)
- double fmin (double x, double y)
- long double fminl (long double x, long double y)
- float truncf (float x)
- double trunc (double x)
- long double truncl (long double x)
- float roundf (float x)
- double round (double x)
- long double roundl (long double x)
- long lroundf (float x)
- long lround (double x)
- long lroundl (long double x)
- long lrintf (float x)
- long lrint (double x)
- long lrintl (long double x)

Non-Standard Math Functions

- float squaref (float x)
- double square (double x)
- long double squarel (long double x)

22.62 math.h

Go to the documentation of this file.

```
00001 /* Copyright (c) 2002,2007-2009 Michael Stumpf
00002    Copyright (c) 2023-2025  Georg-Johann Lay
00003
00004    Portions of documentation Copyright (c) 1990 - 1994
00005    The Regents of the University of California.
00006
00007    All rights reserved.
00008
00009    Redistribution and use in source and binary forms, with or without
00010    modification, are permitted provided that the following conditions are met:
00011
00012    * Redistributions of source code must retain the above copyright
00013      notice, this list of conditions and the following disclaimer.
00014
00015    * Redistributions in binary form must reproduce the above copyright
00016      notice, this list of conditions and the following disclaimer in
00017      the documentation and/or other materials provided with the
00018      distribution.
00019
00020    * Neither the name of the copyright holders nor the names of
00021      contributors may be used to endorse or promote products derived
00022      from this software without specific prior written permission.
00023
00024   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
00025   AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
00026   IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
00027   ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
00028   LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
00029   CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
```

```
00030   SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
00031   INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
00032   CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
00033   ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
00034   POSSIBILITY OF SUCH DAMAGE. */
00035
00036 /*
00037    math.h - mathematical functions
00038
00039    Author : Michael Stumpf
00040             Michael.Stumpf@t-online.de
00041
00042    __ATTR_CONST__ added by marekm@linux.org.pl for functions
00043    that "do not examine any values except their arguments, and have
00044    no effects except the return value", for better optimization by gcc.
00045  */
00046
00047 #ifndef __MATH_H
00048 #define __MATH_H
00049
00050 #ifndef __DOXYGEN__
00051 /* In pursue of old GCC versions... */
00052 #ifndef __SIZEOF_FLOAT__
00053 #define __SIZEOF_FLOAT__ 4
00054 #endif
00055 #ifndef __SIZEOF_DOUBLE__
00056 #define __SIZEOF_DOUBLE__ 4
00057 #endif
00058 #ifndef __SIZEOF_LONG_DOUBLE__
00059 #define __SIZEOF_LONG_DOUBLE__ 4
00060 #endif
00061 #endif /* !Doxygen */
00062
00063 #ifdef __cplusplus
00064 extern "C" {
00065 #endif
00066
00067 /** \file */
00068 /** \defgroup avr_math <math.h>: Mathematics
00069     \code #include <math.h> \endcode
00070
00071     This header file declares basic mathematics constants and
00072     functions.
00073
00074     <b>Notes:</b>
00075
00076     - Math functions do not raise exceptions and do not change the
00077       \c errno variable. Therefore the majority of them are declared
00078       with \c const attribute, for better optimization by GCC.
00079     - 64-bit floating-point arithmetic is only available in
00080       <a href="https://gcc.gnu.org/gcc-10/changes.html#avr">GCC v10</a>
00081       and up.
00082       The size of the \c double and \c long \c double type can be selected
00083       at compile-time with options like <tt>-mdouble=64</tt> and
00084       <tt>-mlong-double=32</tt>.  Whether such options are available,
00085       and their default values, depend on how the compiler has been configured.
00086     - The implementation of 64-bit floating-point arithmetic has some
00087       shortcomings and limitations, see the
00088       <a href="https://gcc.gnu.org/wiki/avr-gcc#Libf7">avr-gcc Wiki</a>
00089       for details.
00090
00091   See also some benchmarks for \ref bench_libm "IEEE single"
00092   and \ref bench_libf7 "IEEE double".
00093 */
00094
00095
00096 /** \ingroup avr_math */
00097 /**@{*/
00098
00099 /** The constant Euler's number \a e. */
00100 #define M_E 2.7182818284590452354
00101
00102 /** The constant logarithm of Euler's number \a e to base 2. */
```

```
00103 #define M_LOG2E 1.4426950408889634074
00104
00105 /** The constant logarithm of Euler's number \a e to base 10. */
00106 #define M_LOG10E    0.43429448190325182765
00107
00108 /** The constant natural logarithm of 2.    */
00109 #define M_LN2       0.69314718055994530942
00110
00111 /** The constant natural logarithm of 10.   */
00112 #define M_LN10      2.30258509299404568402
00113
00114 /** The constant &pi;.  */
00115 #define M_PI        3.14159265358979323846
00116
00117 /** The constant &pi;/2.    */
00118 #define M_PI_2      1.57079632679489661923
00119
00120 /** The constant &pi;/4.    */
00121 #define M_PI_4      0.78539816339744830962
00122
00123 /** The constant 1/&pi;.    */
00124 #define M_1_PI      0.31830988618379067154
00125
00126 /** The constant 2/&pi;.    */
00127 #define M_2_PI      0.63661977236758134308
00128
00129 /** The constant 2/sqrt(&pi;).  */
00130 #define M_2_SQRTPI  1.12837916709551257390
00131
00132 /** The square root of 2.   */
00133 #define M_SQRT2     1.41421356237309504880
00134
00135 /** The constant \a 1/sqrt(2).  */
00136 #define M_SQRT1_2   0.70710678118654752440
00137
00138 /** The \c double representation of a constant quiet NaN. */
00139 #define NAN __builtin_nan("")
00140
00141 /** The \c float representation of a constant quiet NaN.
00142     \p __tag is a string constant like \c "" or \c "123". */
00143 #define nanf(__tag) __builtin_nanf(__tag)
00144
00145 /** The \c double representation of a constant quiet NaN.
00146     \p __tag is a string constant like \c "" or \c "123". */
00147 #define nan(__tag)  __builtin_nan(__tag)
00148
00149 /** The \c long \c double representation of a constant quiet NaN.
00150     \p __tag is a string constant like \c "" or \c "123". */
00151 #define nanl(__tag) __builtin_nanl(__tag)
00152
00153 /** \c float infinity constant. */
00154 #define INFINITY __builtin_inff()
00155
00156 /** \c float infinity constant. */
00157 #define HUGE_VALF __builtin_huge_valf()
00158
00159 /** \c double infinity constant.    */
00160 #define HUGE_VAL __builtin_huge_val()
00161
00162 /** \c long \c double infinity constant.    */
00163 #define HUGE_VALL __builtin_huge_vall()
00164
00165 #include <bits/attribs.h>
00166
00167 /** The cosf() function returns the cosine of \a x, measured in radians. */
00168 __ATTR_CONST__ extern float cosf (float x);
00169 /** The cos() function returns the cosine of \a x, measured in radians. */
00170 __ATTR_CONST__ extern double cos (double x);
00171 /** The cosl() function returns the cosine of \a x, measured in radians.
00172     \since AVR-LibC v2.2 */
00173 __ATTR_CONST__ extern long double cosl (long double x);
00174
00175 /** The sinf() function returns the sine of \a x, measured in radians. */
```

```
00176 __ATTR_CONST__ extern float sinf (float x);
00177 /** The sin() function returns the sine of \a x, measured in radians. */
00178 __ATTR_CONST__ extern double sin (double x);
00179 /** The sinl() function returns the sine of \a x, measured in radians.
00180     \since AVR-LibC v2.2 */
00181 __ATTR_CONST__ extern long double sinl (long double x);
00182
00183 /** The sincosf() function returns the sine of \a x in \c *psin, and
00184     the cosine of \a x in \c *pcos.  The angle \a x is measured in radians.
00185     A sincosf() call is a bit faster than calling sinf() and cosf()
00186     individually, but it consumes a bit more program memory.
00187     \since AVR-LibC v2.3 */
00188 void sincosf (float x, float *psin, float *pcos);
00189
00190 /** The sincos() function returns the sine of \a x in \c *psin, and
00191     the cosine of \a x in \c *pcos.  The angle \a x is measured in radians.
00192     As an example, the performance gain of the IEEE double version of
00193     sincos() compared to a sin() plus a cos() call is around 5000 cycles
00194     for \a x = 2.0.
00195     \since GCC v15.3 (IEEE double), AVR-LibC v2.3 */
00196 void sincos (double x, double *psin, double *pcos);
00197
00198 /** The sincosl() function returns the sine of \a x in \c *psin, and
00199     the cosine of \a x in \c *pcos.  The angle \a x is measured in radians.
00200     As an example, the performance gain of the IEEE double version of
00201     sincosl() compared to a sinl() plus a cosl() call is around 5000 cycles
00202     for \a x = 2.0.
00203     \since GCC v15.3 (IEEE double), AVR-LibC v2.3 */
00204 void sincosl (long double x, long double *psin, long double *pcos);
00205
00206 /** The tanf() function returns the tangent of \a x, measured in radians. */
00207 __ATTR_CONST__ extern float tanf (float x);
00208 /** The tan() function returns the tangent of \a x, measured in radians. */
00209 __ATTR_CONST__ extern double tan (double x);
00210 /** The tanl() function returns the tangent of \a x, measured in radians.
00211     \since AVR-LibC v2.2 */
00212 __ATTR_CONST__ extern long double tanl (long double x);
00213
00214 /** The fabsf() function computes the absolute value of a floating-point
00215     number \a x. */
00216 extern __ATTR_ALWAYS_INLINE__ __ATTR_GNU_INLINE__
00217 float fabsf (float __x)
00218 {
00219     return __builtin_fabsf (__x);
00220 }
00221
00222 /** The fabs() function computes the absolute value of a floating-point
00223     number \a x. */
00224 extern __ATTR_ALWAYS_INLINE__ __ATTR_GNU_INLINE__
00225 double fabs (double __x)
00226 {
00227     return __builtin_fabs (__x);
00228 }
00229
00230 /** The fabsl() function computes the absolute value of a floating-point
00231     number \a x.
00232     \since AVR-LibC v2.2 */
00233 extern __ATTR_ALWAYS_INLINE__ __ATTR_GNU_INLINE__
00234 long double fabsl (long double __x)
00235 {
00236     return __builtin_fabsl (__x);
00237 }
00238
00239 /** The function fmodf() returns the floating-point remainder of <em>x / y</em>. */
00240 __ATTR_CONST__ extern float fmodf (float x, float y);
00241 /** The function fmod() returns the floating-point remainder of <em>x / y</em>. */
00242 __ATTR_CONST__ extern double fmod (double x, double y);
00243 /** The function fmodl() returns the floating-point remainder of <em>x / y</em>.
00244     \since AVR-LibC v2.2 */
00245 __ATTR_CONST__ extern long double fmodl (long double x, long double y);
00246
00247 /** The modff() function breaks the argument \a x into integral and
00248     fractional parts, each of which has the same sign as the argument.
```

```
00249     It stores the integral part as a \c float in the object pointed to by
00250     \a iptr.
00251
00252     The modff() function returns the signed fractional part of \a x.
00253
00254     \note This implementation skips writing by zero pointer.  However,
00255     the GCC 4.3 can replace this function with inline code that does not
00256     permit to use NULL address for the avoiding of storing. */
00257 extern float modff (float x, float *iptr);
00258 /** The modf() function breaks the argument \a x into integral and
00259     fractional parts, each of which has the same sign as the argument.
00260     It stores the integral part as a \c double in the object pointed to by
00261     \a iptr.
00262
00263     The modf() function returns the signed fractional part of \a x. */
00264 extern double modf (double x, double *iptr);
00265 /** The modfl() function breaks the argument \a x into integral and
00266     fractional parts, each of which has the same sign as the argument.
00267     It stores the integral part as a \c long \c double in the object pointed to by
00268     \a iptr.
00269
00270     The modf() function returns the signed fractional part of \a x.
00271     \since AVR-LibC v2.2 */
00272 extern long double modfl (long double x, long double *iptr);
00273
00274 /** The sqrtf() function returns the non-negative square root of \a x. */
00275 __ATTR_CONST__ extern float sqrtf (float x);
00276 /** The sqrt() function returns the non-negative square root of \a x. */
00277 __ATTR_CONST__ extern double sqrt (double x);
00278 /** The sqrtl() function returns the non-negative square root of \a x.
00279     \since AVR-LibC v2.2 */
00280 __ATTR_CONST__ extern long double sqrtl (long double x);
00281
00282 /** The cbrtf() function returns the cube root of \a x. */
00283 __ATTR_CONST__ extern float cbrtf (float x);
00284 /** The cbrt() function returns the cube root of \a x. */
00285 __ATTR_CONST__ extern double cbrt (double x);
00286 /** The cbrtl() function returns the cube root of \a x.
00287     \since AVR-LibC v2.2 */
00288 __ATTR_CONST__ extern long double cbrtl (long double x);
00289
00290 /** The hypotf() function returns <em>sqrtf(x*x + y*y)</em>. This
00291     is the length of the hypotenuse of a right triangle with sides of
00292     length \a x and \a y, or the  distance of the point (\a x, \a
00293     y) from the origin. Using this function  instead of the direct
00294     formula is wise, since the error is much smaller. No underflow with
00295     small \a x and \a y. No overflow if result is in range. */
00296 __ATTR_CONST__ extern float hypotf (float x, float y);
00297 /** The hypot() function returns <em>sqrt(x*x + y*y)</em>. This
00298     is the length of the hypotenuse of a right triangle with sides of
00299     length \a x and \a y, or the  distance of the point (\a x, \a
00300     y) from the origin. Using this function  instead of the direct
00301     formula is wise, since the error is much smaller. No underflow with
00302     small \a x and \a y. No overflow if result is in range. */
00303 __ATTR_CONST__ extern double hypot (double x, double y);
00304 /** The hypotl() function returns <em>sqrtl(x*x + y*y)</em>. This
00305     is the length of the hypotenuse of a right triangle with sides of
00306     length \a x and \a y, or the  distance of the point (\a x, \a
00307     y) from the origin. Using this function  instead of the direct
00308     formula is wise, since the error is much smaller. No underflow with
00309     small \a x and \a y. No overflow if result is in range.
00310     \since AVR-LibC v2.2 */
00311 __ATTR_CONST__ extern long double hypotl (long double x, long double y);
00312
00313 /** The floorf() function returns the largest integral value less than or
00314     equal to \a x, expressed as a floating-point number. */
00315 __ATTR_CONST__ extern float floorf (float x);
00316 /** The floor() function returns the largest integral value less than or
00317     equal to \a x, expressed as a floating-point number. */
00318 __ATTR_CONST__ extern double floor (double x);
00319 /** The floorl() function returns the largest integral value less than or
00320     equal to \a x, expressed as a floating-point number.
00321     \since AVR-LibC v2.2 */
```

```
00322 __ATTR_CONST__ extern long double floorl (long double x);
00323
00324 /** The ceilf() function returns the smallest integral value greater than
00325     or equal to \a x, expressed as a floating-point number. */
00326 __ATTR_CONST__ extern float ceilf (float x);
00327 /** The ceil() function returns the smallest integral value greater than
00328     or equal to \a x, expressed as a floating-point number. */
00329 __ATTR_CONST__ extern double ceil (double x);
00330 /** The ceill() function returns the smallest integral value greater than
00331     or equal to \a x, expressed as a floating-point number.
00332     \since AVR-LibC v2.2 */
00333 __ATTR_CONST__ extern long double ceill (long double x);
00334
00335 /** The frexpf() function breaks a floating-point number into a normalized
00336     fraction and an integral power of 2.  It stores the integer in the \c
00337     int object pointed to by \a pexp.
00338
00339     If \a x is a normal float point number, the frexpf() function
00340     returns the value \c v, such that \c v has a magnitude in the
00341     interval [1/2, 1) or zero, and \a x equals \c v times 2 raised to
00342     the power \a pexp. If \a x is zero, both parts of the result are
00343     zero. If \a x is not a finite number, the frexpf() returns \a x as
00344     is and stores 0 by \a pexp.
00345
00346     \note  This implementation permits a zero pointer as a directive to
00347     skip a storing the exponent.
00348  */
00349 extern float frexpf (float x, int *pexp);
00350 /** The frexp() function breaks a floating-point number into a normalized
00351     fraction and an integral power of 2.  It stores the integer in the \c
00352     int object pointed to by \a pexp.
00353
00354     If \a x is a normal float point number, the frexp() function
00355     returns the value \c v, such that \c v has a magnitude in the
00356     interval [1/2, 1) or zero, and \a x equals \c v times 2 raised to
00357     the power \a pexp. If \a x is zero, both parts of the result are
00358     zero. If \a x is not a finite number, the frexp() returns \a x as
00359     is and stores 0 by \a pexp. */
00360 extern double frexp (double x, int *pexp);
00361 /** The frexpl() function breaks a floating-point number into a normalized
00362     fraction and an integral power of 2.  It stores the integer in the \c
00363     int object pointed to by \a pexp.
00364
00365     If \a x is a normal float point number, the frexpl() function
00366     returns the value \c v, such that \c v has a magnitude in the
00367     interval [1/2, 1) or zero, and \a x equals \c v times 2 raised to
00368     the power \a pexp. If \a x is zero, both parts of the result are
00369     zero. If \a x is not a finite number, the frexpl() returns \a x as
00370     is and stores 0 by \a pexp.
00371     \since AVR-LibC v2.2 */
00372 extern long double frexpl (long double x, int *pexp);
00373
00374 /** The ldexpf() function multiplies a floating-point number by an integral
00375     power of 2. It returns the value of \a x times 2 raised to the power
00376     \a iexp. */
00377 __ATTR_CONST__ extern float ldexpf (float x, int iexp);
00378 /** The ldexp() function multiplies a floating-point number by an integral
00379     power of 2. It returns the value of \a x times 2 raised to the power
00380     \a iexp. */
00381 __ATTR_CONST__ extern double ldexp (double x, int iexp);
00382 /** The ldexpl() function multiplies a floating-point number by an integral
00383     power of 2. It returns the value of \a x times 2 raised to the power
00384     \a iexp.
00385     \since AVR-LibC v2.2 */
00386 __ATTR_CONST__ extern long double ldexpl (long double x, int iexp);
00387
00388 /** The expf() function returns the exponential value of \a x. */
00389 __ATTR_CONST__ extern float expf (float x);
00390 /** The exp() function returns the exponential value of \a x. */
00391 __ATTR_CONST__ extern double exp (double x);
00392 /** The expl() function returns the exponential value of \a x.
00393     \since AVR-LibC v2.2 */
00394 __ATTR_CONST__ extern long double expl (long double x);
```

```
00395
00396 /** The coshf() function returns the hyperbolic cosine of \a x. */
00397 __ATTR_CONST__ extern float coshf (float x);
00398 /** The cosh() function returns the hyperbolic cosine of \a x. */
00399 __ATTR_CONST__ extern double cosh (double x);
00400 /** The coshl() function returns the hyperbolic cosine of \a x.
00401     \since AVR-LibC v2.2 */
00402 __ATTR_CONST__ extern long double coshl (long double x);
00403
00404 /** The sinhf() function returns the hyperbolic sine of \a x. */
00405 __ATTR_CONST__ extern float sinhf (float x);
00406 /** The sinh() function returns the hyperbolic sine of \a x. */
00407 __ATTR_CONST__ extern double sinh (double x);
00408 /** The sinhl() function returns the hyperbolic sine of \a x. */
00409 __ATTR_CONST__ extern long double sinhl (long double x);
00410
00411 /** The tanhf() function returns the hyperbolic tangent of \a x. */
00412 __ATTR_CONST__ extern float tanhf (float x);
00413 /** The tanh() function returns the hyperbolic tangent of \a x. */
00414 __ATTR_CONST__ extern double tanh (double x);
00415 /** The tanhl() function returns the hyperbolic tangent of \a x.
00416     \since AVR-LibC v2.2 */
00417 __ATTR_CONST__ extern long double tanhl (long double x);
00418
00419 /** The acosf() function computes the principal value of the arc cosine of
00420     \a x.  The returned value is in the range [0, &pi;] radians. A domain
00421     error occurs for arguments not in the range [&minus;1, +1].
00422     The relative error is bounded by 1.9&middot;10<sup>&minus;7</sup>. */
00423 __ATTR_CONST__ extern float acosf (float x);
00424 /** The acos() function computes the principal value of the arc cosine of
00425     \a x.  The returned value is in the range [0, &pi;] radians or NaN. */
00426 __ATTR_CONST__ extern double acos (double x);
00427 /** The acosl() function computes the principal value of the arc cosine of
00428     \a x.  The returned value is in the range [0, &pi;] radians or NaN.
00429     \since AVR-LibC v2.2 */
00430 __ATTR_CONST__ extern long double acosl (long double x);
00431
00432 /** The asinf() function computes the principal value of the arc sine of \a x.
00433     The returned value is in the range [&minus;&pi;/2, &pi;/2] radians. A
00434     domain error occurs for arguments not in the range [&minus;1, +1].
00435     The relative error is bounded by 3.3&middot;10<sup>&minus;7</sup>. */
00436 __ATTR_CONST__ extern float asinf (float x);
00437 /** The asin() function computes the principal value of the arc sine of \a x.
00438     The returned value is in the range [&minus;&pi;/2, &pi;/2] radians or NaN.*/
00439 __ATTR_CONST__ extern double asin (double x);
00440 /** The asinl() function computes the principal value of the arc sine of \a x.
00441     The returned value is in the range [&minus;&pi;/2, &pi;/2] radians or NaN.
00442     \since AVR-LibC v2.2 */
00443 __ATTR_CONST__ extern long double asinl (long double x);
00444
00445 /** The atanf() function computes the principal value of the arc tangent
00446     of \a x.  The returned value is in the range [&minus;&pi;/2, &pi;/2] radians.
00447     The relative error is bounded by 1.3&middot;10<sup>&minus;7</sup>. */
00448 __ATTR_CONST__ extern float atanf (float x);
00449 /** The atan() function computes the principal value of the arc tangent of \a x.
00450     The returned value is in the range [&minus;&pi;/2, &pi;/2] radians. */
00451 __ATTR_CONST__ extern double atan (double x);
00452 /** The atanl() function computes the principal value of the arc tangent of
00453     \a x.  The returned value is in the range [&minus;&pi;/2, &pi;/2] radians.
00454     \since AVR-LibC v2.2 */
00455 __ATTR_CONST__ extern long double atanl (long double x);
00456
00457 /** The atan2f() function computes the principal value of the arc tangent
00458     of <em>y / x</em>, using the signs of both arguments to determine
00459     the quadrant of the return value.  The returned value is in the range
00460     [&minus;&pi;, +&pi;] radians. */
00461 __ATTR_CONST__ extern float atan2f (float y, float x);
00462 /** The atan2() function computes the principal value of the arc tangent
00463     of <em>y / x</em>, using the signs of both arguments to determine
00464     the quadrant of the return value.  The returned value is in the range
00465     [&minus;&pi;, +&pi;] radians. */
00466 __ATTR_CONST__ extern double atan2 (double y, double x);
00467 /** The atan2l() function computes the principal value of the arc tangent
```

```
00468     of <em>y / x</em>, using the signs of both arguments to determine
00469     the quadrant of the return value.  The returned value is in the range
00470     [&minus;&pi;, +&pi;] radians.
00471     \since AVR-LibC v2.2 */
00472 __ATTR_CONST__ extern long double atan2l (long double y, long double x);
00473
00474 /** The logf() function returns the natural logarithm of argument \a x.
00475     The relative error is bounded by 2.3&middot;10<sup>&minus;7</sup>. */
00476 __ATTR_CONST__ extern float logf (float x);
00477 /** The log() function returns the natural logarithm of argument \a x. */
00478 __ATTR_CONST__ extern double log (double x);
00479 /** The logl() function returns the natural logarithm of argument \a x.
00480     \since AVR-LibC v2.2 */
00481 __ATTR_CONST__ extern long double logl (long double x);
00482
00483 /** The log10f() function returns the logarithm of argument \a x to base 10.
00484     The relative error is bounded by 2.8&middot;10<sup>&minus;7</sup>. */
00485 __ATTR_CONST__ extern float log10f (float x);
00486 /** The log10() function returns the logarithm of argument \a x to base 10. */
00487 __ATTR_CONST__ extern double log10 (double x);
00488 /** The log10l() function returns the logarithm of argument \a x to base 10.
00489     \since AVR-LibC v2.2 */
00490 __ATTR_CONST__ extern long double log10l (long double x);
00491
00492 /** The log2f() function returns the logarithm of argument \a x to base 2.
00493     The relative error is bounded by 1.8&middot;10<sup>&minus;7</sup>.
00494     \since AVR-LibC v2.3 */
00495 __ATTR_CONST__ extern float log2f (float x);
00496 /** The log2() function returns the logarithm of argument \a x to base 2.
00497     \since AVR-LibC v2.3 */
00498 __ATTR_CONST__ extern double log2 (double x);
00499 /** The log2l() function returns the logarithm of argument \a x to base 2.
00500     \since AVR-LibC v2.3 */
00501 __ATTR_CONST__ extern long double log2l (long double x);
00502
00503 /** The function powf() returns the value of \a x to the exponent \a y.
00504     \n Notice that for integer exponents, there is the more efficient
00505     <code>float __builtin_powif(float x, int y)</code>. */
00506 __ATTR_CONST__ extern float powf (float x, float y);
00507 /** The function pow() returns the value of \a x to the exponent \a y.
00508     \n Notice that for integer exponents, there is the more efficient
00509     <code>double __builtin_powi(double x, int y)</code>. */
00510 __ATTR_CONST__ extern double pow (double x, double y);
00511 /** The function powl() returns the value of \a x to the exponent \a y.
00512     \n Notice that for integer exponents, there is the more efficient
00513     <code>long double __builtin_powil(long double x, int y)</code>.
00514     \since AVR-LibC v2.2 */
00515 __ATTR_CONST__ extern long double powl (long double x, long double y);
00516
00517 /** The function isnanf() returns 1 if the argument \a x represents a
00518     "not-a-number" (NaN) object, otherwise 0. */
00519 __ATTR_CONST__ extern int isnanf (float x);
00520 /** The function isnan() returns 1 if the argument \a x represents a
00521     "not-a-number" (NaN) object, otherwise 0. */
00522 __ATTR_CONST__ extern int isnan (double x);
00523 /** The function isnanl() returns 1 if the argument \a x represents a
00524     "not-a-number" (NaN) object, otherwise 0.
00525     \since AVR-LibC v2.2 */
00526 __ATTR_CONST__ extern int isnanl (long double x);
00527
00528 /** The function isinff() returns 1 if the argument \a x is positive
00529     infinity, &minus;1 if \a x is negative infinity, and 0 otherwise. */
00530 __ATTR_CONST__ extern int isinff (float x);
00531 /** The function isinf() returns 1 if the argument \a x is positive
00532     infinity, &minus;1 if \a x is negative infinity, and 0 otherwise. */
00533 __ATTR_CONST__ extern int isinf (double x);
00534 /** The function isinfl() returns 1 if the argument \a x is positive
00535     infinity, &minus;1 if \a x is negative infinity, and 0 otherwise.
00536     \since AVR-LibC v2.2 */
00537 __ATTR_CONST__ extern int isinfl (long double x);
00538
00539 /** The isfinitef() function returns a nonzero value if \a __x is finite:
00540     not plus or minus infinity, and not NaN. */
```

```
00541 __ATTR_CONST__ static __ATTR_ALWAYS_INLINE__ int isfinitef (float __x)
00542 {
00543     unsigned char __exp;
00544     __asm__ (
00545         "mov    %0, %C1"     "\n\t"
00546         "lsl    %0"          "\n\t"
00547         "mov    %0, %D1"     "\n\t"
00548         "rol    %0"
00549         : "=&r" (__exp)
00550         : "r" (__x) );
00551     return __exp != 0xff;
00552 }
00553
00554 /** The isfinite() function returns a nonzero value if \a __x is finite:
00555     not plus or minus infinity, and not NaN. */
00556 #ifdef __DOXYGEN__
00557 static __ATTR_ALWAYS_INLINE__ int isfinite (double __x);
00558 #elif __SIZEOF_DOUBLE__ == __SIZEOF_FLOAT__
00559 static __ATTR_ALWAYS_INLINE__ int isfinite (double __x)
00560 {
00561     return isfinitef (__x);
00562 }
00563 #else
00564 int isfinite (double __x);
00565 #endif /* double = float */
00566
00567 /** The isfinitel() function returns a nonzero value if \a __x is finite:
00568     not plus or minus infinity, and not NaN.
00569     \since AVR-LibC v2.2 */
00570 #ifdef __DOXYGEN__
00571 static __ATTR_ALWAYS_INLINE__ int isfinitel (long double __x);
00572 #elif __SIZEOF_LONG_DOUBLE__ == __SIZEOF_FLOAT__
00573 static __ATTR_ALWAYS_INLINE__ int isfinitel (long double __x)
00574 {
00575     return isfinitef (__x);
00576 }
00577 #else
00578 int isfinitel (long double __x);
00579 #endif /* long double = float */
00580
00581 /** The copysignf() function returns \a __x but with the sign of \a __y.
00582     They work even if \a __x or \a __y are NaN or zero. */
00583 __ATTR_CONST__ static __ATTR_ALWAYS_INLINE__ float copysignf (float __x, float __y)
00584 {
00585     __asm__ (
00586         "bst    %D2, 7"  "\n\t"
00587         "bld    %D0, 7"
00588         : "=r" (__x)
00589         : "0" (__x), "r" (__y));
00590     return __x;
00591 }
00592
00593 /** The copysign() function returns \a __x but with the sign of \a __y.
00594     They work even if \a __x or \a __y are NaN or zero. */
00595 __ATTR_CONST__ static __ATTR_ALWAYS_INLINE__ double copysign (double __x, double __y)
00596 {
00597     __asm__ (
00598         "bst    %r1+%2-1, 7" "\n\t"
00599         "bld    %r0+%2-1, 7"
00600         : "+r" (__x)
00601         : "r" (__y), "n" (__SIZEOF_DOUBLE__));
00602     return __x;
00603 }
00604
00605 /** The copysignl() function returns \a __x but with the sign of \a __y.
00606     They work even if \a __x or \a __y are NaN or zero.
00607     \since AVR-LibC v2.2 */
00608 __ATTR_CONST__ static __ATTR_ALWAYS_INLINE__ long double copysignl (long double __x,
    long double __y)
00609 {
00610     __asm__ (
00611         "bst    %r1+%2-1, 7" "\n\t"
00612         "bld    %r0+%2-1, 7"
```

```
00613         : "+r" (__x)
00614         : "r" (__y), "n" (__SIZEOF_LONG_DOUBLE__));
00615     return __x;
00616 }
00617
00618 /** The signbitf() function returns a nonzero value if the value of \a x
00619     has its sign bit set.  This is not the same as `\a x < 0.0',
00620     because IEEE 754 floating point allows zero to be signed. The
00621     comparison '&minus;0.0 < 0.0' is false, but `signbit (&minus;0.0)'
00622     will return a nonzero value. */
00623 __ATTR_CONST__ extern int signbitf (float x);
00624 /** The signbit() function returns a nonzero value if the value of \a x
00625     has its sign bit set.  This is not the same as `\a x < 0.0',
00626     because IEEE 754 floating point allows zero to be signed. The
00627     comparison '&minus;0.0 < 0.0' is false, but `signbit (&minus;0.0)'
00628     will return a nonzero value. */
00629 __ATTR_CONST__ extern int signbit (double x);
00630 /** The signbitl() function returns a nonzero value if the value of \a x
00631     has its sign bit set.  This is not the same as `\a x < 0.0',
00632     because IEEE 754 floating point allows zero to be signed. The
00633     comparison '&minus;0.0 < 0.0' is false, but `signbit (&minus;0.0)'
00634     will return a nonzero value.
00635     \since AVR-LibC v2.2 */
00636 __ATTR_CONST__ extern int signbitl (long double x);
00637
00638 /** The fdimf() function returns <em>max(x &minus; y, 0)</em>. If \a x or
00639     \a y or both are NaN, NaN is returned. */
00640 __ATTR_CONST__ extern float fdimf (float x, float y);
00641 /** The fdim() function returns <em>max(x &minus; y, 0)</em>. If \a x or
00642     \a y or both are NaN, NaN is returned. */
00643 __ATTR_CONST__ extern double fdim (double x, double y);
00644 /** The fdiml() function returns <em>max(x &minus; y, 0)</em>. If \a x or
00645     \a y or both are NaN, NaN is returned.
00646     \since AVR-LibC v2.2, GCC v15.2 */
00647 __ATTR_CONST__ extern long double fdiml (long double x, long double y);
00648
00649 /** The fmaf() function performs floating-point multiply-add. This is the
00650     operation <em>(x * y) + z</em>, but the intermediate result is
00651     not rounded to the destination type.  This can sometimes improve the
00652     precision of a calculation. */
00653 __ATTR_CONST__ extern float fmaf (float x, float y, float z);
00654 /** The fma() function performs floating-point multiply-add. This is the
00655     operation <em>(x * y) + z</em>, but the intermediate result is
00656     not rounded to the destination type.  This can sometimes improve the
00657     precision of a calculation. */
00658 __ATTR_CONST__ extern double fma (double x, double y, double z);
00659 /** The fmal() function performs floating-point multiply-add. This is the
00660     operation <em>(x * y) + z</em>, but the intermediate result is
00661     not rounded to the destination type.  This can sometimes improve the
00662     precision of a calculation.
00663     \since AVR-LibC v2.2 */
00664 __ATTR_CONST__ extern long double fmal (long double x, long double y, long double z);
00665
00666 /** The fmaxf() function returns the greater of the two values \a x and
00667     \a y. If an argument is NaN, the other argument is returned. If
00668     both arguments are NaN, NaN is returned. */
00669 __ATTR_CONST__ extern float fmaxf (float x, float y);
00670 /** The fmax() function returns the greater of the two values \a x and
00671     \a y. If an argument is NaN, the other argument is returned. If
00672     both arguments are NaN, NaN is returned. */
00673 __ATTR_CONST__ extern double fmax (double x, double y);
00674 /** The fmaxl() function returns the greater of the two values \a x and
00675     \a y. If an argument is NaN, the other argument is returned. If
00676     both arguments are NaN, NaN is returned.
00677     \since AVR-LibC v2.2 */
00678 __ATTR_CONST__ extern long double fmaxl (long double x, long double y);
00679
00680 /** The fminf() function returns the lesser of the two values \a x and
00681     \a y. If an argument is NaN, the other argument is returned. If
00682     both arguments are NaN, NaN is returned. */
00683 __ATTR_CONST__ extern float fminf (float x, float y);
00684 /** The fmin() function returns the lesser of the two values \a x and
00685     \a y. If an argument is NaN, the other argument is returned. If
```

```
00686     both arguments are NaN, NaN is returned. */
00687 __ATTR_CONST__ extern double fmin (double x, double y);
00688 /** The fminl() function returns the lesser of the two values \a x and
00689     \a y. If an argument is NaN, the other argument is returned. If
00690     both arguments are NaN, NaN is returned.
00691     \since AVR-LibC v2.2 */
00692 __ATTR_CONST__ extern long double fminl (long double x, long double y);
00693
00694 /** The truncf() function rounds \a x to the nearest integer not larger
00695     in absolute value. */
00696 __ATTR_CONST__ extern float truncf (float x);
00697 /** The trunc() function rounds \a x to the nearest integer not larger
00698     in absolute value. */
00699 __ATTR_CONST__ extern double trunc (double x);
00700 /** The truncl() function rounds \a x to the nearest integer not larger
00701     in absolute value.
00702     \since AVR-LibC v2.2 */
00703 __ATTR_CONST__ extern long double truncl (long double x);
00704
00705 /** The roundf() function rounds \a x to the nearest integer, but rounds
00706     halfway cases away from zero (instead of to the nearest even integer).
00707     Overflow is impossible.
00708
00709     \return The rounded value. If \a x is an integral or infinite, \a
00710     x itself is returned. If \a x is \c NaN, then \c NaN is returned. */
00711 __ATTR_CONST__ extern float roundf (float x);
00712 /** The round() function rounds \a x to the nearest integer, but rounds
00713     halfway cases away from zero (instead of to the nearest even integer).
00714     Overflow is impossible.
00715
00716     \return The rounded value. If \a x is an integral or infinite, \a
00717     x itself is returned. If \a x is \c NaN, then \c NaN is returned. */
00718 __ATTR_CONST__ extern double round (double x);
00719 /** The roundl() function rounds \a x to the nearest integer, but rounds
00720     halfway cases away from zero (instead of to the nearest even integer).
00721     Overflow is impossible.
00722
00723     \return The rounded value. If \a x is an integral or infinite, \a
00724     x itself is returned. If \a x is \c NaN, then \c NaN is returned.
00725     \since AVR-LibC v2.2 */
00726 __ATTR_CONST__ extern long double roundl (long double x);
00727
00728 /** The lroundf() function rounds \a x to the nearest integer, but rounds
00729     halfway cases away from zero (instead of to the nearest even integer).
00730     This function is similar to round() function, but it differs in type of
00731     return value and in that an overflow is possible.
00732
00733     \return The rounded long integer value. If \a x is not a finite number
00734     or an overflow was, this realization returns the \c LONG_MIN value
00735     (0x80000000). */
00736 __ATTR_CONST__ extern long lroundf (float x);
00737 /** The lround() function rounds \a x to the nearest integer, but rounds
00738     halfway cases away from zero (instead of to the nearest even integer).
00739     This function is similar to round() function, but it differs in type of
00740     return value and in that an overflow is possible.
00741
00742     \return The rounded long integer value. If \a x is not a finite number
00743     or an overflow was, this realization returns the \c LONG_MIN value
00744     (0x80000000). */
00745 __ATTR_CONST__ extern long lround (double x);
00746 /** The lroundl() function rounds \a x to the nearest integer, but rounds
00747     halfway cases away from zero (instead of to the nearest even integer).
00748     This function is similar to round() function, but it differs in type of
00749     return value and in that an overflow is possible.
00750
00751     \return The rounded long integer value. If \a x is not a finite number
00752     or an overflow was, this realization returns the \c LONG_MIN value
00753     (0x80000000).
00754     \since AVR-LibC v2.2 */
00755 __ATTR_CONST__ extern long lroundl (long double x);
00756
00757 /** The lrintf() function rounds \a x to the nearest integer, rounding the
00758     halfway cases to the even integer direction. (That is both 1.5 and 2.5
```

```
00759     values are rounded to 2). This function is similar to rintf() function,
00760     but it differs in type of return value and in that an overflow is
00761     possible.
00762
00763     \return The rounded long integer value. If \a x is not a finite
00764     number or an overflow was, this realization returns the \c LONG_MIN
00765     value (0x80000000). */
00766 __ATTR_CONST__ extern long lrintf (float x);
00767 /** The lrint() function rounds \a x to the nearest integer, rounding the
00768     halfway cases to the even integer direction. (That is both 1.5 and 2.5
00769     values are rounded to 2). This function is similar to rint() function,
00770     but it differs in type of return value and in that an overflow is
00771     possible.
00772
00773     \return The rounded long integer value. If \a x is not a finite
00774     number or an overflow was, this realization returns the \c LONG_MIN
00775     value (0x80000000). */
00776 __ATTR_CONST__ extern long lrint (double x);
00777 /** The lrintl() function rounds \a x to the nearest integer, rounding the
00778     halfway cases to the even integer direction. (That is both 1.5 and 2.5
00779     values are rounded to 2). This function is similar to rintl() function,
00780     but it differs in type of return value and in that an overflow is
00781     possible.
00782
00783     \return The rounded long integer value. If \a x is not a finite
00784     number or an overflow was, this realization returns the \c LONG_MIN
00785     value (0x80000000).
00786     \since AVR-LibC v2.2 */
00787 __ATTR_CONST__ extern long lrintl (long double x);
00788
00789 /**@}*/
00790
00791 /**@{*/
00792 /**
00793    \name Non-Standard Math Functions
00794 */
00795
00796 /** \ingroup avr_math
00797     The function squaref() returns <em>x * x</em>.
00798     \note This function does not belong to the C standard definition. */
00799 __ATTR_CONST__ extern float squaref (float x);
00800
00801 /** The function square() returns <em>x * x</em>.
00802     \note This function does not belong to the C standard definition. */
00803
00804 #if defined(__DOXYGEN__) || __SIZEOF_DOUBLE__ == __SIZEOF_FLOAT__
00805 __ATTR_CONST__ extern double square (double x);
00806 #elif defined(__WITH_LIBF7_MATH__)
00807 __ATTR_CONST__ extern double square (double x) __asm("__f7_square");
00808 #endif
00809
00810 /** The function squarel() returns <em>x * x</em>.
00811     \note This function does not belong to the C standard definition.
00812     \since AVR-LibC v2.2 */
00813 #if defined(__DOXYGEN__) || __SIZEOF_LONG_DOUBLE__ == __SIZEOF_FLOAT__
00814 __ATTR_CONST__ extern long double squarel (long double x);
00815 #elif defined(__WITH_LIBF7_MATH__)
00816 __ATTR_CONST__ extern long double squarel (long double x) __asm("__f7_square");
00817 #endif
00818
00819 /**@}*/
00820
00821 #ifdef __cplusplus
00822 }
00823 #endif
00824
00825 #endif /* !__MATH_H */
```

22.63 setjmp.h File Reference

Functions

- int setjmp (jmp_buf __jmpb)
- void longjmp (jmp_buf __jmpb, int __ret)

22.64 setjmp.h

Go to the documentation of this file.

```
00001 /* Copyright (c) 2002,2007 Marek Michalkiewicz
00002    All rights reserved.
00003
00004    Redistribution and use in source and binary forms, with or without
00005    modification, are permitted provided that the following conditions are met:
00006
00007    * Redistributions of source code must retain the above copyright
00008      notice, this list of conditions and the following disclaimer.
00009
00010    * Redistributions in binary form must reproduce the above copyright
00011      notice, this list of conditions and the following disclaimer in
00012      the documentation and/or other materials provided with the
00013      distribution.
00014
00015    * Neither the name of the copyright holders nor the names of
00016      contributors may be used to endorse or promote products derived
00017      from this software without specific prior written permission.
00018
00019   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
00020   AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
00021   IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
00022   ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
00023   LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
00024   CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
00025   SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
00026   INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
00027   CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
00028   ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
00029   POSSIBILITY OF SUCH DAMAGE. */
00030
00031 #ifndef __SETJMP_H_
00032 #define __SETJMP_H_ 1
00033
00034 #ifdef __cplusplus
00035 extern "C" {
00036 #endif
00037
00038 /*
00039    jmp_buf:
00040     offset  size    description
00041      0  16/2    call-saved registers (r2-r17)
00042                 (AVR_TINY arch has only 2 call saved registers (r18,r19))
00043     16/2     2  frame pointer (r29:r28)
00044     18/4     2  stack pointer (SPH:SPL)
00045     20/6     1  status register (SREG)
00046     21/7     2/3    return address (PC) (2 bytes used for <=128Kw flash)
00047     23/24/9 = total size (AVR_TINY arch always has 2 bytes PC)
00048  */
00049
00050 #if !defined(__DOXYGEN__)
00051
00052 #if defined(__AVR_TINY__)
00053 # define _JBLEN  9
00054 #elif   defined(__AVR_3_BYTE_PC__) && __AVR_3_BYTE_PC__
00055 # define _JBLEN  24
00056 #else
00057 # define _JBLEN  23
```

```
00058 #endif
00059 typedef struct _jmp_buf { unsigned char _jb[_JBLEN]; } jmp_buf[1];
00060
00061 #endif /* not __DOXYGEN__ */
00062
00063 /** \file */
00064 /** \defgroup setjmp <setjmp.h>: Non-local goto
00065
00066     While the C language has the dreaded \c goto statement, it can only be
00067     used to jump to a label in the same (local) function.  In order to jump
00068     directly to another (non-local) function, the C library provides the
00069     #setjmp and #longjmp functions.  setjmp and longjmp are useful for
00070     dealing with errors and interrupts encountered in a low-level subroutine
00071     of a program.
00072
00073     \note #setjmp and #longjmp make programs hard to understand and maintain.
00074     If possible, an alternative should be used.
00075
00076     \note longjmp can destroy changes made to global register
00077     variables (see \ref faq_regbind).
00078
00079     For a very detailed discussion of setjmp/longjmp, see Chapter 7 of
00080     <em>Advanced Programming in the UNIX Environment</em>, by W. Richard
00081     Stevens.
00082
00083     Example:
00084
00085     \code
00086     #include <setjmp.h>
00087
00088     jmp_buf env;
00089
00090     int main (void)
00091     {
00092         if (setjmp (env))
00093         {
00094             // Handle error ...
00095         }
00096
00097         while (1)
00098         {
00099            // Main processing loop which calls foo() somewhere ...
00100         }
00101     }
00102
00103     void foo (void)
00104     {
00105         // blah, blah, blah ...
00106
00107         if (err)
00108         {
00109             longjmp (env, 1);
00110         }
00111     }
00112     \endcode */
00113
00114 #ifndef __DOXYGEN__
00115
00116 #include <bits/attribs.h>
00117
00118 #endif /* ! DOXYGEN */
00119
00120 /** \ingroup setjmp
00121     \brief Save stack context for non-local goto.
00122
00123     \code #include <setjmp.h>\endcode
00124
00125     setjmp() saves the stack context/environment in \e __jmpb for later use by
00126     longjmp().  The stack context will be invalidated if the function which
00127     called setjmp() returns.
00128
00129     \param __jmpb Variable of type \c jmp_buf which holds the stack
00130     information such that the environment can be restored.
```

```
00131
00132     \returns Returns 0 if returning directly, and
00133     non-zero when returning from longjmp() using the saved context. */
00134
00135 extern int setjmp(jmp_buf __jmpb);
00136
00137 /** \ingroup setjmp
00138     \brief Non-local jump to a saved stack context.
00139
00140     \code #include <setjmp.h>\endcode
00141
00142     longjmp() restores the environment saved by the last call of setjmp() with
00143     the corresponding \e __jmpb argument.  After longjmp() is completed,
00144     program execution continues as if the corresponding call of setjmp() had
00145     just returned the value \e __ret.
00146
00147     \note longjmp() cannot cause 0 to be returned.  If longjmp() is invoked
00148     with a second argument of 0, 1 will be returned instead.
00149
00150     \param __jmpb Information saved by a previous call to setjmp().
00151     \param __ret  Value to return to the caller of setjmp().
00152
00153     This function never returns. */
00154
00155 extern void longjmp(jmp_buf __jmpb, int __ret) __ATTR_NORETURN__;
00156
00157 #ifdef __cplusplus
00158 }
00159 #endif
00160
00161 #endif  /* !__SETJMP_H_ */
```

22.65 stdint.h File Reference

Macros

Limits of specified-width integer types

C++ implementations should define these macros only when __STDC_LIMIT_MACROS is defined before <stdint.h> is included.

- #define INT8_MAX 0x7f
- #define INT8_MIN (-INT8_MAX - 1)
- #define UINT8_MAX (__CONCAT(INT8_MAX, U) * 2U + 1U)
- #define INT16_MAX 0x7fff
- #define INT16_MIN (-INT16_MAX - 1)
- #define UINT16_MAX (__CONCAT(INT16_MAX, U) * 2U + 1U)
- #define INT24_MAX __INT24_MAX__
- #define INT24_MIN __INT24_MIN__
- #define UINT24_MAX __UINT24_MAX__
- #define INT32_MAX 0x7fffffffL
- #define INT32_MIN (-INT32_MAX - 1L)
- #define UINT32_MAX (__CONCAT(INT32_MAX, U) * 2UL + 1UL)
- #define INT64_MAX 0x7fffffffffffffffLL
- #define INT64_MIN (-INT64_MAX - 1LL)
- #define UINT64_MAX (__CONCAT(INT64_MAX, U) * 2ULL + 1ULL)

Limits of minimum-width integer types

- #define INT_LEAST8_MAX INT8_MAX
- #define INT_LEAST8_MIN INT8_MIN
- #define UINT_LEAST8_MAX UINT8_MAX
- #define INT_LEAST16_MAX INT16_MAX
- #define INT_LEAST16_MIN INT16_MIN
- #define UINT_LEAST16_MAX UINT16_MAX

- #define INT_LEAST24_MAX INT24_MAX
- #define INT_LEAST24_MIN INT24_MIN
- #define UINT_LEAST24_MAX UINT24_MAX
- #define INT_LEAST32_MAX INT32_MAX
- #define INT_LEAST32_MIN INT32_MIN
- #define UINT_LEAST32_MAX UINT32_MAX
- #define INT_LEAST64_MAX INT64_MAX
- #define INT_LEAST64_MIN INT64_MIN
- #define UINT_LEAST64_MAX UINT64_MAX

Limits of fastest minimum-width integer types

- #define INT_FAST8_MAX INT8_MAX
- #define INT_FAST8_MIN INT8_MIN
- #define UINT_FAST8_MAX UINT8_MAX
- #define INT_FAST16_MAX INT16_MAX
- #define INT_FAST16_MIN INT16_MIN
- #define UINT_FAST16_MAX UINT16_MAX
- #define INT_FAST24_MAX INT24_MAX
- #define INT_FAST24_MIN INT24_MIN
- #define UINT_FAST24_MAX UINT24_MAX
- #define INT_FAST32_MAX INT32_MAX
- #define INT_FAST32_MIN INT32_MIN
- #define UINT_FAST32_MAX UINT32_MAX
- #define INT_FAST64_MAX INT64_MAX
- #define INT_FAST64_MIN INT64_MIN
- #define UINT_FAST64_MAX UINT64_MAX

Limits of integer types capable of holding object pointers

- #define INTPTR_MAX INT16_MAX
- #define INTPTR_MIN INT16_MIN
- #define UINTPTR_MAX UINT16_MAX
- #define INTPTR24_MAX INT24_MAX
- #define INTPTR24_MIN INT24_MIN
- #define UINTPTR24_MAX UINT24_MAX

Limits of greatest-width integer types

- #define INTMAX_MAX INT64_MAX
- #define INTMAX_MIN INT64_MIN
- #define UINTMAX_MAX UINT64_MAX

Limits of other integer types

C++ implementations should define these macros only when __STDC_LIMIT_MACROS is defined before <stdint.h> is included.

- #define PTRDIFF_MAX INT16_MAX
- #define PTRDIFF_MIN INT16_MIN
- #define SIG_ATOMIC_MAX INT8_MAX
- #define SIG_ATOMIC_MIN INT8_MIN
- #define SIZE_MAX UINT16_MAX

Macros for integer constants

C++ implementations should define these macros only when __STDC_CONSTANT_MACROS is defined before <stdint.h> is included.

These definitions are valid for integer constants without suffix and for macros defined as integer constant without suffix.

- #define INT8_C(value) ((int8_t) value)
- #define UINT8_C(value) ((uint8_t) __CONCAT(value, U))
- #define INT16_C(value) value
- #define UINT16_C(value) __CONCAT(value, U)
- #define INT24_C(value) ((int24_t) __CONCAT(value, L))
- #define UINT24_C(value) ((uint24_t) __CONCAT(value, UL))
- #define INT32_C(value) __CONCAT(value, L)
- #define UINT32_C(value) __CONCAT(value, UL)
- #define INT64_C(value) __CONCAT(value, LL)
- #define UINT64_C(value) __CONCAT(value, ULL)
- #define INTMAX_C(value) __CONCAT(value, LL)
- #define UINTMAX_C(value) __CONCAT(value, ULL)

Typedefs

Exact-width integer types

Integer types having exactly the specified width.

- typedef signed char int8_t
- typedef unsigned char uint8_t
- typedef signed int int16_t
- typedef unsigned int uint16_t
- typedef __int24 int24_t
- typedef __uint24 uint24_t
- typedef signed long int int32_t
- typedef unsigned long int uint32_t
- typedef signed long long int int64_t
- typedef unsigned long long int uint64_t

Integer types capable of holding object pointers

These allow you to declare variables of the same size as a pointer.

- typedef int16_t intptr_t
- typedef uint16_t uintptr_t
- typedef int_least24_t intptr24_t
- typedef uint_least24_t uintptr24_t

Minimum-width integer types

Integer types having at least the specified width.

- typedef int8_t int_least8_t
- typedef uint8_t uint_least8_t
- typedef int16_t int_least16_t
- typedef uint16_t uint_least16_t
- typedef int24_t int_least24_t
- typedef uint24_t uint_least24_t
- typedef int32_t int_least32_t
- typedef uint32_t uint_least32_t
- typedef int64_t int_least64_t
- typedef uint64_t uint_least64_t

Fastest minimum-width integer types

Integer types being usually fastest having at least the specified width.

- typedef int8_t int_fast8_t
- typedef uint8_t uint_fast8_t
- typedef int16_t int_fast16_t
- typedef uint16_t uint_fast16_t
- typedef int24_t int_fast24_t

- typedef uint24_t uint_fast24_t
- typedef int32_t int_fast32_t
- typedef uint32_t uint_fast32_t
- typedef int64_t int_fast64_t
- typedef uint64_t uint_fast64_t

Greatest-width integer types

Types designating integer data capable of representing any value of any integer type in the corresponding signed or unsigned category.

- typedef int64_t intmax_t
- typedef uint64_t uintmax_t

22.66 stdint.h

Go to the documentation of this file.

```
00001 /* Copyright (c) 2002,2004,2005 Marek Michalkiewicz
00002    Copyright (c) 2005, Carlos Lamas
00003    Copyright (c) 2005,2007 Joerg Wunsch
00004    Copyright (c) 2013 Embecosm
00005    All rights reserved.
00006
00007    Redistribution and use in source and binary forms, with or without
00008    modification, are permitted provided that the following conditions are met:
00009
00010    * Redistributions of source code must retain the above copyright
00011      notice, this list of conditions and the following disclaimer.
00012
00013    * Redistributions in binary form must reproduce the above copyright
00014      notice, this list of conditions and the following disclaimer in
00015      the documentation and/or other materials provided with the
00016      distribution.
00017
00018    * Neither the name of the copyright holders nor the names of
00019      contributors may be used to endorse or promote products derived
00020      from this software without specific prior written permission.
00021
00022   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
00023   AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
00024   IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
00025   ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
00026   LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
00027   CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
00028   SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
00029   INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
00030   CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
00031   ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
00032   POSSIBILITY OF SUCH DAMAGE. */
00033
00034 /*
00035  * ISO/IEC 9899:1999  7.18 Integer types <stdint.h>
00036  */
00037
00038 #ifndef __STDINT_H_
00039 #define __STDINT_H_
00040
00041 /** \file */
00042 /** \defgroup avr_stdint <stdint.h>: Standard Integer Types
00043     \code #include <stdint.h> \endcode
00044
00045     Use [u]intN_t if you need exactly N bits.
00046
00047     Since these typedefs are mandated by the C99 standard, they are preferred
00048     over rolling your own typedefs.
00049
00050 \anchor int24
```

```
00051     \note The 24-bit types and constants are supported since AVR-LibC v2.3.
00052     In older versions, they can still be used by means of the compiler
00053     built-in types and macros \c __int24, \c __uint24, \c __INT24_MIN__,
00054     \c __INT24_MAX__ and \c __UINT24_MAX__ supported since
00055     <a href="https://gcc.gnu.org/gcc-4.7/changes.html#avr">GCC v4.7</a>.
00056 */
00057
00058
00059 #ifndef __DOXYGEN__
00060 /*
00061  * __USING_MINT8 is defined to 1 if the -mint8 option is in effect.
00062  */
00063 #if __INT_MAX__ == 127
00064 # define __USING_MINT8 1
00065 #else
00066 # define __USING_MINT8 0
00067 #endif
00068
00069 #endif /* !__DOXYGEN__ */
00070
00071 /* Integer types */
00072
00073 #if defined(__DOXYGEN__)
00074
00075 /* doxygen gets confused by the __attribute__ stuff */
00076
00077 /** \name Exact-width integer types
00078     Integer types having exactly the specified width. */
00079
00080 /**@{*/
00081
00082 /** \ingroup avr_stdint
00083     8-Bit signed integer type. */
00084 typedef signed char int8_t;
00085
00086 /** \ingroup avr_stdint
00087     8-Bit unsigned integer type. */
00088 typedef unsigned char uint8_t;
00089
00090 /** \ingroup avr_stdint
00091     16-Bit signed integer type. */
00092 typedef signed int int16_t;
00093
00094 /** \ingroup avr_stdint
00095     16-Bit unsigned integer type. */
00096 typedef unsigned int uint16_t;
00097
00098 /** \ingroup avr_stdint
00099     24-Bit signed integer type.
00100     \since AVR-LibC v2.3, \ref int24 */
00101 typedef __int24 int24_t;
00102
00103 /** \ingroup avr_stdint
00104     24-Bit unsigned integer type.
00105     \since AVR-LibC v2.3, \ref int24 */
00106 typedef __uint24 uint24_t;
00107
00108 /** \ingroup avr_stdint
00109     32-Bit signed integer type. */
00110 typedef signed long int int32_t;
00111
00112 /** \ingroup avr_stdint
00113     32-Bit unsigned integer type. */
00114 typedef unsigned long int uint32_t;
00115
00116 /** \ingroup avr_stdint
00117     64-Bit signed integer type.
00118     \note This type is not available when the compiler
00119     option \c \-mint8 is in effect. */
00120 typedef signed long long int int64_t;
00121
00122 /** \ingroup avr_stdint
00123     64-Bit unsigned integer type.
```

```
00124     \note This type is not available when the compiler
00125     option \c \-mint8 is in effect. */
00126 typedef unsigned long long int uint64_t;
00127
00128 /**@}*/
00129
00130 #else /* !defined(__DOXYGEN__) */
00131
00132 /* Actual implementation goes here.  Define to what built-ins
00133    like __UINT32_TYPE__ would resolve to.  */
00134
00135 typedef signed char int8_t;
00136 typedef unsigned char uint8_t;
00137
00138 #if !__USING_MINT8
00139
00140 typedef int int16_t;
00141 typedef unsigned int uint16_t;
00142 typedef long int int32_t;
00143 typedef long unsigned int uint32_t;
00144 typedef long long int int64_t;
00145 typedef long long unsigned int uint64_t;
00146
00147 #else /* __USING_MINT8 */
00148
00149 typedef long int int16_t;
00150 typedef long unsigned int uint16_t;
00151 typedef long long int int32_t;
00152 typedef long long unsigned int uint32_t;
00153
00154 #endif /* __USING_MINT8 */
00155
00156 #ifdef __INT24_MAX__
00157 __extension__ typedef __int24 int24_t;
00158 __extension__ typedef __uint24 uint24_t;
00159 typedef int24_t  int_least24_t;
00160 typedef uint24_t uint_least24_t;
00161 typedef int24_t  int_fast24_t;
00162 typedef uint24_t uint_fast24_t;
00163 #else
00164 typedef int32_t  int_least24_t;
00165 typedef uint32_t uint_least24_t;
00166 typedef int32_t  int_fast24_t;
00167 typedef uint32_t uint_fast24_t;
00168 #endif /* Have int24 */
00169
00170 #endif /* defined(__DOXYGEN__) */
00171
00172 /** \name Integer types capable of holding object pointers
00173     These allow you to declare variables of the same size as a pointer. */
00174
00175 /**@{*/
00176
00177 /** \ingroup avr_stdint
00178     Signed pointer compatible type. */
00179 typedef int16_t intptr_t;
00180
00181 /** \ingroup avr_stdint
00182     Unsigned pointer compatible type. */
00183 typedef uint16_t uintptr_t;
00184
00185 /** \ingroup avr_stdint
00186     Signed 24-bit pointer compatible type.
00187     \since AVR-LibC v2.3, \ref int24  */
00188 typedef int_least24_t intptr24_t;
00189
00190 /** \ingroup avr_stdint
00191     Unsigned 24-bit pointer compatible type.
00192     \since AVR-LibC v2.3, \ref int24  */
00193 typedef uint_least24_t uintptr24_t;
00194
00195 /**@}*/
00196
```

```
00197 /** \name Minimum-width integer types
00198    Integer types having at least the specified width. */
00199
00200 /**@{*/
00201
00202 /** \ingroup avr_stdint
00203     Signed integer type with at least 8 bits. */
00204 typedef int8_t   int_least8_t;
00205
00206 /** \ingroup avr_stdint
00207     Unsigned integer type with at least 8 bits. */
00208 typedef uint8_t  uint_least8_t;
00209
00210 /** \ingroup avr_stdint
00211     Signed integer type with at least 16 bits. */
00212 typedef int16_t  int_least16_t;
00213
00214 /** \ingroup avr_stdint
00215     Unsigned integer type with at least 16 bits. */
00216 typedef uint16_t uint_least16_t;
00217
00218 #ifdef __DOXYGEN__
00219 /** \ingroup avr_stdint
00220     Signed integer type with at least 24 bits.
00221     \since AVR-LibC v2.3, \ref int24  */
00222 typedef int24_t  int_least24_t;
00223
00224 /** \ingroup avr_stdint
00225     Unsigned integer type with at least 24 bits.
00226     \since AVR-LibC v2.3, \ref int24  */
00227 typedef uint24_t uint_least24_t;
00228 #endif /* Doxygen */
00229
00230 /** \ingroup avr_stdint
00231     Signed integer type with at least 32 bits. */
00232 typedef int32_t  int_least32_t;
00233
00234 /** \ingroup avr_stdint
00235     Unsigned integer type with at least 32 bits. */
00236 typedef uint32_t uint_least32_t;
00237
00238 #if !__USING_MINT8 || defined(__DOXYGEN__)
00239 /** \ingroup avr_stdint
00240     Signed integer type with at least 64 bits.
00241     \note This type is not available when the compiler
00242     option \c \-mint8 is in effect. */
00243 typedef int64_t  int_least64_t;
00244
00245 /** \ingroup avr_stdint
00246     Unsigned integer type with at least 64 bits.
00247     \note This type is not available when the compiler
00248     option \c \-mint8 is in effect. */
00249 typedef uint64_t uint_least64_t;
00250
00251 #endif
00252
00253 /**@}*/
00254
00255
00256 /** \name Fastest minimum-width integer types
00257    Integer types being usually fastest having at least the specified width. */
00258
00259 /**@{*/
00260
00261 /** \ingroup avr_stdint
00262     Fastest signed integer type with at least 8 bits. */
00263 typedef int8_t int_fast8_t;
00264
00265 /** \ingroup avr_stdint
00266     Fastest unsigned integer type with at least 8 bits. */
00267 typedef uint8_t uint_fast8_t;
00268
00269 /** \ingroup avr_stdint
```

```
00270     Fastest signed integer type with at least 16 bits. */
00271 typedef int16_t int_fast16_t;
00272
00273 /** \ingroup avr_stdint
00274     Fastest unsigned integer type with at least 16 bits. */
00275 typedef uint16_t uint_fast16_t;
00276
00277 #ifdef __DOXYGEN__
00278 /** \ingroup avr_stdint
00279     Fastest signed integer type with at least 24 bits.
00280     \since AVR-LibC v2.3, \ref int24  */
00281 typedef int24_t  int_fast24_t;
00282
00283 /** \ingroup avr_stdint
00284     Fastest unsigned integer type with at least 24 bits.
00285     \since AVR-LibC v2.3, \ref int24  */
00286 typedef uint24_t uint_fast24_t;
00287 #endif /* Doxygen */
00288
00289 /** \ingroup avr_stdint
00290     Fastest signed integer type with at least 32 bits. */
00291 typedef int32_t int_fast32_t;
00292
00293 /** \ingroup avr_stdint
00294     Fastest unsigned integer type with at least 32 bits. */
00295 typedef uint32_t uint_fast32_t;
00296
00297 #if !__USING_MINT8 || defined(__DOXYGEN__)
00298 /** \ingroup avr_stdint
00299     Fastest signed integer type with at least 64 bits.
00300     \note This type is not available when the compiler
00301     option -mint8 is in effect. */
00302 typedef int64_t int_fast64_t;
00303
00304 /** \ingroup avr_stdint
00305     Fastest unsigned integer type with at least 64 bits.
00306     \note This type is not available when the compiler
00307     option -mint8 is in effect. */
00308 typedef uint64_t uint_fast64_t;
00309 #endif
00310
00311 /**@}*/
00312
00313
00314 /** \name Greatest-width integer types
00315    Types designating integer data capable of representing any value of
00316    any integer type in the corresponding signed or unsigned category. */
00317
00318 /**@{*/
00319
00320 #if __USING_MINT8
00321
00322 typedef int32_t intmax_t;
00323 typedef uint32_t uintmax_t;
00324
00325 #else  /* !__USING_MINT8 */
00326
00327 /** \ingroup avr_stdint
00328     Largest signed integer available. */
00329 typedef int64_t intmax_t;
00330
00331 /** \ingroup avr_stdint
00332     Largest unsigned integer available. */
00333 typedef uint64_t uintmax_t;
00334
00335 #endif /* __USING_MINT8 */
00336
00337 /**@}*/
00338
00339 #ifndef __DOXYGEN__
00340 /* Helping macro */
00341 #ifndef __CONCAT
00342 #define __CONCATenate(left, right) left ## right
```

```
00343 #define __CONCAT(left, right) __CONCATenate(left, right)
00344 #endif
00345
00346 #endif /* !__DOXYGEN__ */
00347
00348 #if !defined(__cplusplus) || defined(__STDC_LIMIT_MACROS)
00349
00350 /** \name Limits of specified-width integer types
00351    C++ implementations should define these macros only when
00352    __STDC_LIMIT_MACROS is defined before <stdint.h> is included. */
00353
00354 /**@{*/
00355
00356 /** \ingroup avr_stdint
00357     Largest positive value an \c #int8_t can hold. */
00358 #define INT8_MAX 0x7f
00359
00360 /** \ingroup avr_stdint
00361     Smallest negative value an \c #int8_t can hold. */
00362 #define INT8_MIN (-INT8_MAX - 1)
00363
00364 #if __USING_MINT8
00365
00366 #define UINT8_MAX (__CONCAT(INT8_MAX, U) * 2U + 1U)
00367
00368 #define INT16_MAX 0x7fffL
00369 #define INT16_MIN (-INT16_MAX - 1L)
00370 #define UINT16_MAX (__CONCAT(INT16_MAX, U) * 2UL + 1UL)
00371
00372 #define INT32_MAX 0x7fffffffLL
00373 #define INT32_MIN (-INT32_MAX - 1LL)
00374 #define UINT32_MAX (__CONCAT(INT32_MAX, U) * 2ULL + 1ULL)
00375
00376 #else /* !__USING_MINT8 */
00377
00378 /** \ingroup avr_stdint
00379     Largest value an \c #uint8_t can hold. */
00380 #define UINT8_MAX (__CONCAT(INT8_MAX, U) * 2U + 1U)
00381
00382 /** \ingroup avr_stdint
00383     Largest positive value an \c #int16_t can hold. */
00384 #define INT16_MAX 0x7fff
00385
00386 /** \ingroup avr_stdint
00387     Smallest negative value an \c #int16_t can hold. */
00388 #define INT16_MIN (-INT16_MAX - 1)
00389
00390 /** \ingroup avr_stdint
00391     Largest value an \c #uint16_t can hold. */
00392 #define UINT16_MAX (__CONCAT(INT16_MAX, U) * 2U + 1U)
00393
00394 /** \ingroup avr_stdint
00395     Largest positive value an \c #int24_t can hold.
00396     \since AVR-LibC v2.3, \ref int24 */
00397 #define INT24_MAX __INT24_MAX__
00398
00399 /** \ingroup avr_stdint
00400     Smallest negative value an \c #int24_t can hold.
00401     \since AVR-LibC v2.3, \ref int24 */
00402 #define INT24_MIN __INT24_MIN__
00403
00404 /** \ingroup avr_stdint
00405     Largest value an \c #uint24_t can hold.
00406     \since AVR-LibC v2.3, \ref int24 */
00407 #define UINT24_MAX __UINT24_MAX__
00408
00409 /** \ingroup avr_stdint
00410     Largest positive value an \c #int32_t can hold. */
00411 #define INT32_MAX 0x7fffffffL
00412
00413 /** \ingroup avr_stdint
00414     Smallest negative value an \c #int32_t can hold. */
00415 #define INT32_MIN (-INT32_MAX - 1L)
```

```
00416
00417 /** \ingroup avr_stdint
00418     Largest value an \c #uint32_t can hold. */
00419 #define UINT32_MAX (__CONCAT(INT32_MAX, U) * 2UL + 1UL)
00420
00421 #endif /* __USING_MINT8 */
00422
00423 /** \ingroup avr_stdint
00424     Largest positive value an \c #int64_t can hold. */
00425 #define INT64_MAX 0x7fffffffffffffffLL
00426
00427 /** \ingroup avr_stdint
00428     Smallest negative value an \c #int64_t can hold. */
00429 #define INT64_MIN (-INT64_MAX - 1LL)
00430
00431 /** \ingroup avr_stdint
00432     Largest value an \c #uint64_t can hold. */
00433 #define UINT64_MAX (__CONCAT(INT64_MAX, U) * 2ULL + 1ULL)
00434
00435 /**@}*/
00436
00437 /** \name Limits of minimum-width integer types */
00438 /**@{*/
00439
00440 /** \ingroup avr_stdint
00441     Largest positive value an \c #int_least8_t can hold. */
00442 #define INT_LEAST8_MAX INT8_MAX
00443
00444 /** \ingroup avr_stdint
00445     Smallest negative value an \c #int_least8_t can hold. */
00446 #define INT_LEAST8_MIN INT8_MIN
00447
00448 /** \ingroup avr_stdint
00449     Largest value an \c #uint_least8_t can hold. */
00450 #define UINT_LEAST8_MAX UINT8_MAX
00451
00452 /** \ingroup avr_stdint
00453     Largest positive value an \c #int_least16_t can hold. */
00454 #define INT_LEAST16_MAX INT16_MAX
00455
00456 /** \ingroup avr_stdint
00457     Smallest negative value an \c #int_least16_t can hold. */
00458 #define INT_LEAST16_MIN INT16_MIN
00459
00460 /** \ingroup avr_stdint
00461     Largest value an \c #uint_least16_t can hold. */
00462 #define UINT_LEAST16_MAX UINT16_MAX
00463
00464 /** \ingroup avr_stdint
00465     Largest positive value an \c #int_least24_t can hold.
00466     \since AVR-LibC v2.3, \ref int24 */
00467 #define INT_LEAST24_MAX INT24_MAX
00468
00469 /** \ingroup avr_stdint
00470     Smallest negative value an \c #int_least24_t can hold.
00471     \since AVR-LibC v2.3, \ref int24 */
00472 #define INT_LEAST24_MIN INT24_MIN
00473
00474 /** \ingroup avr_stdint
00475     Largest value an \c #uint_least24_t can hold.
00476     \since AVR-LibC v2.3, \ref int24 */
00477 #define UINT_LEAST24_MAX UINT24_MAX
00478
00479 /** \ingroup avr_stdint
00480     Largest positive value an \c #int_least32_t can hold. */
00481 #define INT_LEAST32_MAX INT32_MAX
00482
00483 /** \ingroup avr_stdint
00484     Smallest negative value an \c #int_least32_t can hold. */
00485 #define INT_LEAST32_MIN INT32_MIN
00486
00487 /** \ingroup avr_stdint
00488     Largest value an \c #uint_least32_t can hold. */
```

```
00489 #define UINT_LEAST32_MAX UINT32_MAX
00490
00491 /** \ingroup avr_stdint
00492     Largest positive value an \c #int_least64_t can hold. */
00493 #define INT_LEAST64_MAX INT64_MAX
00494
00495 /** \ingroup avr_stdint
00496     Smallest negative value an \c #int_least64_t can hold. */
00497 #define INT_LEAST64_MIN INT64_MIN
00498
00499 /** \ingroup avr_stdint
00500     Largest value an \c #uint_least64_t can hold. */
00501 #define UINT_LEAST64_MAX UINT64_MAX
00502
00503 /**@}*/
00504
00505 /** \name Limits of fastest minimum-width integer types */
00506
00507 /**@{*/
00508
00509 /** \ingroup avr_stdint
00510     Largest positive value an \c #int_fast8_t can hold. */
00511 #define INT_FAST8_MAX INT8_MAX
00512
00513 /** \ingroup avr_stdint
00514     Smallest negative value an \c #int_fast8_t can hold. */
00515 #define INT_FAST8_MIN INT8_MIN
00516
00517 /** \ingroup avr_stdint
00518     Largest value an \c #uint_fast8_t can hold. */
00519 #define UINT_FAST8_MAX UINT8_MAX
00520
00521 /** \ingroup avr_stdint
00522     Largest positive value an \c #int_fast16_t can hold. */
00523 #define INT_FAST16_MAX INT16_MAX
00524
00525 /** \ingroup avr_stdint
00526     Smallest negative value an \c #int_fast16_t can hold. */
00527 #define INT_FAST16_MIN INT16_MIN
00528
00529 /** \ingroup avr_stdint
00530     Largest value an \c #uint_fast16_t can hold. */
00531 #define UINT_FAST16_MAX UINT16_MAX
00532
00533 /** \ingroup avr_stdint
00534     Largest positive value an \c #int_fast24_t can hold.
00535     \since AVR-LibC v2.3, \ref int24 */
00536 #define INT_FAST24_MAX INT24_MAX
00537
00538 /** \ingroup avr_stdint
00539     Smallest negative value an \c #int_fast24_t can hold.
00540     \since AVR-LibC v2.3, \ref int24 */
00541 #define INT_FAST24_MIN INT24_MIN
00542
00543 /** \ingroup avr_stdint
00544     Largest value an \c #uint_fast24_t can hold.
00545     \since AVR-LibC v2.3, \ref int24 */
00546 #define UINT_FAST24_MAX UINT24_MAX
00547
00548 /** \ingroup avr_stdint
00549     Largest positive value an \c #int_fast32_t can hold. */
00550 #define INT_FAST32_MAX INT32_MAX
00551
00552 /** \ingroup avr_stdint
00553     Smallest negative value an \c #int_fast32_t can hold. */
00554 #define INT_FAST32_MIN INT32_MIN
00555
00556 /** \ingroup avr_stdint
00557     Largest value an \c #uint_fast32_t can hold. */
00558 #define UINT_FAST32_MAX UINT32_MAX
00559
00560 /** \ingroup avr_stdint
00561     Largest positive value an \c #int_fast64_t can hold. */
```

```
00562 #define INT_FAST64_MAX INT64_MAX
00563
00564 /** \ingroup avr_stdint
00565     Smallest negative value an \c #int_fast64_t can hold. */
00566 #define INT_FAST64_MIN INT64_MIN
00567
00568 /** \ingroup avr_stdint
00569     Largest value an \c #uint_fast64_t can hold. */
00570 #define UINT_FAST64_MAX UINT64_MAX
00571
00572 /**@}*/
00573
00574 /** \name Limits of integer types capable of holding object pointers */
00575
00576 /**@{*/
00577
00578 /** \ingroup avr_stdint
00579     Largest positive value an \c #intptr_t can hold. */
00580 #define INTPTR_MAX INT16_MAX
00581
00582 /** \ingroup avr_stdint
00583     Smallest negative value an \c #intptr_t can hold. */
00584 #define INTPTR_MIN INT16_MIN
00585
00586 /** \ingroup avr_stdint
00587     Largest value an \c #uintptr_t can hold. */
00588 #define UINTPTR_MAX UINT16_MAX
00589
00590 /** \ingroup avr_stdint
00591     Largest positive value an \c #intptr24_t can hold.
00592     \since AVR-LibC v2.3, \ref int24 */
00593 #define INTPTR24_MAX INT24_MAX
00594
00595 /** \ingroup avr_stdint
00596     Smallest negative value an \c #intptr24_t can hold.
00597     \since AVR-LibC v2.3, \ref int24 */
00598 #define INTPTR24_MIN INT24_MIN
00599
00600 /** \ingroup avr_stdint
00601     Largest value an \c #uintptr24_t can hold.
00602     \since AVR-LibC v2.3, \ref int24 */
00603 #define UINTPTR24_MAX UINT24_MAX
00604
00605 /**@}*/
00606
00607 /** \name Limits of greatest-width integer types */
00608
00609 /**@{*/
00610
00611 /** \ingroup avr_stdint
00612     Largest positive value an \c #intmax_t can hold. */
00613 #define INTMAX_MAX INT64_MAX
00614
00615 /** \ingroup avr_stdint
00616     Smallest negative value an \c #intmax_t can hold. */
00617 #define INTMAX_MIN INT64_MIN
00618
00619 /** \ingroup avr_stdint
00620     Largest value an \c #uintmax_t can hold. */
00621 #define UINTMAX_MAX UINT64_MAX
00622
00623 /**@}*/
00624
00625 /** \name Limits of other integer types
00626     C++ implementations should define these macros only when
00627     __STDC_LIMIT_MACROS is defined before <stdint.h> is included. */
00628
00629 /**@{*/
00630
00631 /** \ingroup avr_stdint
00632     Largest positive value a \c ptrdiff_t can hold. */
00633 #define PTRDIFF_MAX INT16_MAX
00634
```

```
00635 /** \ingroup avr_stdint
00636     Smallest negative value a \c ptrdiff_t can hold. */
00637 #define PTRDIFF_MIN INT16_MIN
00638
00639
00640 /* Limits of sig_atomic_t */
00641 /* signal.h is currently not implemented (not avr/signal.h) */
00642
00643 /** \ingroup avr_stdint
00644     Largest positive value a \c sig_atomic_t can hold. */
00645 #define SIG_ATOMIC_MAX INT8_MAX
00646
00647 /** \ingroup avr_stdint
00648     Smallest negative value a \c sig_atomic_t can hold. */
00649 #define SIG_ATOMIC_MIN INT8_MIN
00650
00651
00652 /** \ingroup avr_stdint
00653     Largest value a \c size_t can hold. */
00654 #define SIZE_MAX UINT16_MAX
00655
00656
00657 #ifndef __DOXYGEN__
00658 /* Limits of wchar_t */
00659 /* wchar.h is currently not implemented */
00660 #ifndef WCHAR_MAX
00661 #define WCHAR_MAX __WCHAR_MAX__
00662 #define WCHAR_MIN __WCHAR_MIN__
00663 #endif
00664
00665 /* Limits of wint_t */
00666 /* wchar.h is currently not implemented */
00667 #ifndef WINT_MAX
00668 #define WINT_MAX __WINT_MAX__
00669 #define WINT_MIN __WINT_MIN__
00670 #endif
00671 #endif /* !__DOXYGEN__ */
00672
00673
00674 #endif /* !defined(__cplusplus) || defined(__STDC_LIMIT_MACROS) */
00675
00676 #if (!defined __cplusplus || __cplusplus >= 201103L \
00677      || defined __STDC_CONSTANT_MACROS)
00678
00679 /** \name Macros for integer constants
00680     C++ implementations should define these macros only when
00681     __STDC_CONSTANT_MACROS is defined before <stdint.h> is included.
00682
00683     These definitions are valid for integer constants without suffix and
00684     for macros defined as integer constant without suffix. */
00685
00686 /* The GNU C preprocessor defines special macros in the implementation
00687    namespace to allow a definition that works in #if expressions.  */
00688 #ifdef __INT8_C
00689 #define INT8_C(c) __INT8_C(c)
00690 #define INT16_C(c) __INT16_C(c)
00691 #define INT32_C(c) __INT32_C(c)
00692 #define INT64_C(c) __INT64_C(c)
00693 #define UINT8_C(c) __UINT8_C(c)
00694 #define UINT16_C(c) __UINT16_C(c)
00695 #define UINT32_C(c) __UINT32_C(c)
00696 #define UINT64_C(c) __UINT64_C(c)
00697 #define INTMAX_C(c) __INTMAX_C(c)
00698 #define UINTMAX_C(c) __UINTMAX_C(c)
00699
00700 #if __USING_MINT8
00701 #define INT24_C(value) ((int24_t) __CONCAT(value, LL))
00702 #define UINT24_C(value) ((uint24_t) __CONCAT(value, ULL))
00703 #else
00704 #define INT24_C(value) ((int24_t) __CONCAT(value, L))
00705 #define UINT24_C(value) ((uint24_t) __CONCAT(value, UL))
00706 #endif
00707
```

```
00708 #else
00709 /** \ingroup avr_stdint
00710     Define a constant of type \c #int8_t */
00711
00712 #define INT8_C(value) ((int8_t) value)
00713
00714 /** \ingroup avr_stdint
00715     Define a constant of type \c #uint8_t */
00716
00717 #define UINT8_C(value) ((uint8_t) __CONCAT(value, U))
00718
00719 #if __USING_MINT8
00720
00721 #define INT16_C(value) __CONCAT(value, L)
00722 #define UINT16_C(value) __CONCAT(value, UL)
00723
00724 #define INT32_C(value) ((int32_t) __CONCAT(value, LL))
00725 #define UINT32_C(value) ((uint32_t) __CONCAT(value, ULL))
00726
00727 #else /* !__USING_MINT8 */
00728
00729 /** \ingroup avr_stdint
00730     Define a constant of type \c #int16_t */
00731 #define INT16_C(value) value
00732
00733 /** \ingroup avr_stdint
00734     Define a constant of type \c #uint16_t */
00735 #define UINT16_C(value) __CONCAT(value, U)
00736
00737 /** \ingroup avr_stdint
00738     Define a constant of type \c #int24_t
00739     \since AVR-LibC v2.3, \ref int24 */
00740 #define INT24_C(value) ((int24_t) __CONCAT(value, L))
00741
00742 /** \ingroup avr_stdint
00743     Define a constant of type \c #uint24_t
00744     \since AVR-LibC v2.3, \ref int24 */
00745 #define UINT24_C(value) ((uint24_t) __CONCAT(value, UL))
00746
00747 /** \ingroup avr_stdint
00748     Define a constant of type \c #int32_t */
00749 #define INT32_C(value) __CONCAT(value, L)
00750
00751 /** \ingroup avr_stdint
00752     Define a constant of type \c #uint32_t */
00753 #define UINT32_C(value) __CONCAT(value, UL)
00754
00755 #endif /* __USING_MINT8 */
00756
00757 /** \ingroup avr_stdint
00758     Define a constant of type \c #int64_t */
00759 #define INT64_C(value) __CONCAT(value, LL)
00760
00761 /** \ingroup avr_stdint
00762     Define a constant of type \c #uint64_t */
00763 #define UINT64_C(value) __CONCAT(value, ULL)
00764
00765 /** \ingroup avr_stdint
00766     Define a constant of type \c #intmax_t */
00767 #define INTMAX_C(value) __CONCAT(value, LL)
00768
00769 /** \ingroup avr_stdint
00770     Define a constant of type \c #uintmax_t */
00771 #define UINTMAX_C(value) __CONCAT(value, ULL)
00772
00773 #endif /* !__INT8_C */
00774
00775 /**@}*/
00776
00777 #endif /* (!defined __cplusplus || __cplusplus >= 201103L \
00778            || defined __STDC_CONSTANT_MACROS) */
00779
00780
```

```
00781 #endif /* _STDINT_H_ */
```

22.67 stdio.h File Reference

Macros

- #define stdin (__iob[0])
- #define stdout (__iob[1])
- #define stderr (__iob[2])
- #define EOF (-1)
- #define fdev_set_udata(stream, u) do { (stream)->udata = u; } while(0)
- #define fdev_get_udata(stream) ((stream)->udata)
- #define fdev_setup_stream(stream, put, get, rwflag)
- #define _FDEV_SETUP_READ __SRD
- #define _FDEV_SETUP_WRITE __SWR
- #define _FDEV_SETUP_RW (__SRD|__SWR)
- #define _FDEV_ERR (-1)
- #define _FDEV_EOF (-2)
- #define FDEV_SETUP_STREAM(put, get, rwflag)
- #define fdev_close() ((void)0)
- #define putc(__c, __stream) fputc(__c, __stream)
- #define putchar(__c) fputc(__c, stdout)
- #define getc(__stream) fgetc(__stream)
- #define getchar() fgetc(stdin)

Typedefs

- typedef struct __file FILE

Functions

- int fclose (FILE *__stream)
- int vfprintf (FILE *__stream, const char *__fmt, va_list __ap)
- int vfprintf_P (FILE *__stream, const char *__fmt, va_list __ap)
- int fputc (int __c, FILE *__stream)
- int printf (const char *__fmt,...)
- int printf_P (const char *__fmt,...)
- int vprintf (const char *__fmt, va_list __ap)
- int sprintf (char *__s, const char *__fmt,...)
- int sprintf_P (char *__s, const char *__fmt,...)
- int snprintf (char *__s, size_t __n, const char *__fmt,...)
- int snprintf_P (char *__s, size_t __n, const char *__fmt,...)
- int vsprintf (char *__s, const char *__fmt, va_list __ap)
- int vsprintf_P (char *__s, const char *__fmt, va_list __ap)
- int vsnprintf (char *__s, size_t __n, const char *__fmt, va_list __ap)
- int vsnprintf_P (char *__s, size_t __n, const char *__fmt, va_list __ap)
- int fprintf (FILE *__stream, const char *__fmt,...)
- int fprintf_P (FILE *__stream, const char *__fmt,...)
- int fputs (const char *__str, FILE *__stream)
- int fputs_P (const char *__str, FILE *__stream)
- int puts (const char *__str)

- int puts_P (const char *__str)
- size_t fwrite (const void *__ptr, size_t __size, size_t __nmemb, FILE *__stream)
- int fgetc (FILE *__stream)
- int ungetc (int __c, FILE *__stream)
- char * fgets (char *__str, int __size, FILE *__stream)
- char * gets (char *__str)
- size_t fread (void *__ptr, size_t __size, size_t __nmemb, FILE *__stream)
- void clearerr (FILE *__stream)
- int feof (FILE *__stream)
- int ferror (FILE *__stream)
- int vfscanf (FILE *__stream, const char *__fmt, va_list __ap)
- int vfscanf_P (FILE *__stream, const char *__fmt, va_list __ap)
- int fscanf (FILE *__stream, const char *__fmt,...)
- int fscanf_P (FILE *__stream, const char *__fmt,...)
- int scanf (const char *__fmt,...)
- int scanf_P (const char *__fmt,...)
- int vscanf (const char *__fmt, va_list __ap)
- int sscanf (const char *__buf, const char *__fmt,...)
- int sscanf_P (const char *__buf, const char *__fmt,...)
- int fflush (FILE *stream)

22.68 stdio.h

Go to the documentation of this file.

```
00001 /* Copyright (c) 2002, 2005, 2007 Joerg Wunsch
00002    All rights reserved.
00003
00004    Portions of documentation Copyright (c) 1990, 1991, 1993
00005    The Regents of the University of California.
00006
00007    All rights reserved.
00008
00009    Redistribution and use in source and binary forms, with or without
00010    modification, are permitted provided that the following conditions are met:
00011
00012    * Redistributions of source code must retain the above copyright
00013      notice, this list of conditions and the following disclaimer.
00014
00015    * Redistributions in binary form must reproduce the above copyright
00016      notice, this list of conditions and the following disclaimer in
00017      the documentation and/or other materials provided with the
00018      distribution.
00019
00020    * Neither the name of the copyright holders nor the names of
00021      contributors may be used to endorse or promote products derived
00022      from this software without specific prior written permission.
00023
00024   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
00025   AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
00026   IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
00027   ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
00028   LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
00029   CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
00030   SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
00031   INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
00032   CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
00033   ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
00034   POSSIBILITY OF SUCH DAMAGE. */
00035
00036 #ifndef _STDIO_H_
00037 #define _STDIO_H_ 1
00038
00039 #ifndef __ASSEMBLER__
```

```
00040
00041 #include <inttypes.h>
00042 #include <stdarg.h>
00043
00044 #ifndef __DOXYGEN__
00045 #define __need_NULL
00046 #define __need_size_t
00047 #include <stddef.h>
00048 #endif /* !__DOXYGEN__ */
00049
00050 /** \file */
00051 /** \defgroup avr_stdio <stdio.h>: Standard IO facilities
00052     \code #include <stdio.h> \endcode
00053
00054     <h3>Introduction to the Standard IO facilities</h3>
00055
00056     This file declares the standard IO facilities that are implemented
00057     in AVR-LibC.  Due to the nature of the underlying hardware,
00058     only a limited subset of standard IO is implemented.  There is no
00059     actual file implementation available, so only device IO can be
00060     performed.  Since there's no operating system, the application
00061     needs to provide enough details about their devices in order to
00062     make them usable by the standard IO facilities.
00063
00064     Due to space constraints, some functionality has not been
00065     implemented at all (like some of the \c printf conversions that
00066     have been left out).  Nevertheless, potential users of this
00067     implementation should be warned: the \c printf and \c scanf families of functions,
      although
00068     usually associated with presumably simple things like the
00069     famous "Hello, world!" program, are actually fairly complex
00070     which causes their inclusion to eat up a fair amount of code space.
00071     Also, they are not fast due to the nature of interpreting the
00072     format string at run-time.  Whenever possible, resorting to the
00073     (sometimes non-standard) predetermined conversion facilities that are
00074     offered by AVR-LibC will usually cost much less in terms of speed
00075     and code size.
00076
00077     <h3>Tunable options for code size vs. feature set</h3>
00078
00079     In order to allow programmers a code size vs. functionality tradeoff,
00080     the function vfprintf() which is the heart of the printf family can be
00081     selected in different flavours using linker options.  See the
00082     documentation of vfprintf() for a detailed description.  The same
00083     applies to vfscanf() and the \c scanf family of functions.
00084
00085     <h3>Outline of the chosen API</h3>
00086
00087     The standard streams \c stdin, \c stdout, and \c stderr are
00088     provided, but contrary to the C standard, since AVR-LibC has no
00089     knowledge about applicable devices, these streams are not already
00090     pre-initialized at application startup.  Also, since there is no
00091     notion of "file" whatsoever to AVR-LibC, there is no function
00092     \c fopen() that could be used to associate a stream to some device.
00093     (See \ref stdio_note1 "note 1".)  Instead, the function \c fdevopen()
00094     is provided to associate a stream to a device, where the device
00095     needs to provide a function to send a character, to receive a
00096     character, or both.  There is no differentiation between "text" and
00097     "binary" streams inside AVR-LibC.  Character \c \\n is sent
00098     literally down to the device's \c put() function.  If the device
00099     requires a carriage return (\c \\r) character to be sent before
00100     the linefeed, its \c put() routine must implement this (see
00101     \ref stdio_note2 "note 2").
00102
00103     As an alternative method to fdevopen(), the macro
00104     fdev_setup_stream() might be used to setup a user-supplied FILE
00105     structure.
00106
00107     It should be noted that the automatic conversion of a newline
00108     character into a carriage return - newline sequence breaks binary
00109     transfers.  If binary transfers are desired, no automatic
00110     conversion should be performed, but instead any string that aims
00111     to issue a CR-LF sequence must use <tt>"\r\n"</tt> explicitly.
```

```
00112
00113     For convenience, the first call to \c fdevopen() that opens a
00114     stream for reading will cause the resulting stream to be aliased
00115     to \c stdin.  Likewise, the first call to \c fdevopen() that opens
00116     a stream for writing will cause the resulting stream to be aliased
00117     to both, \c stdout, and \c stderr.  Thus, if the open was done
00118     with both, read and write intent, all three standard streams will
00119     be identical.  Note that these aliases are indistinguishable from
00120     each other, thus calling \c fclose() on such a stream will also
00121     effectively close all of its aliases (\ref stdio_note3 "note 3").
00122
00123     It is possible to tie additional user data to a stream, using
00124     fdev_set_udata().  The backend put and get functions can then
00125     extract this user data using fdev_get_udata(), and act
00126     appropriately.  For example, a single put function could be used
00127     to talk to two different UARTs that way, or the put and get
00128     functions could keep internal state between calls there.
00129
00130     <h3>Format strings in flash ROM</h3>
00131
00132     All the \c printf and \c scanf family functions come in three flavours:
00133     the standard name, where the format string is expected to be in
00134     SRAM, as well as two versions with the suffix "_P" and "_F" where the
00135     format string is expected to reside in the flash ROM.  The macros
00136     #PSTR (explained in \ref avr_pgmspace) and #FSTR (explained in
00137     \ref avr_flash "<avr/flash.h>") become very handy for declaring
00138     these format strings.
00139
00140     \anchor stdio_without_malloc
00141     <h3>Running stdio without malloc()</h3>
00142
00143     By default, fdevopen() requires \ref a_malloc "malloc()".  As this is often
00144     not desired in the limited environment of a microcontroller, an
00145     alternative option is provided to run completely without malloc().
00146
00147     The macro fdev_setup_stream() is provided to prepare a
00148     user-supplied FILE buffer for operation with stdio.
00149
00150     <h4>Example</h4>
00151
00152     \code
00153     #include <stdio.h>
00154
00155     static FILE mystdout = FDEV_SETUP_STREAM (uart_putchar, NULL,
00156                                               _FDEV_SETUP_WRITE);
00157     static int
00158     uart_putchar (char c, FILE *stream)
00159     {
00160         if (c == '\n')
00161             uart_putchar ('\r', stream);
00162         loop_until_bit_is_set (UCSRA, UDRE);
00163         UDR = c;
00164         return 0;
00165     }
00166
00167     int main (void)
00168     {
00169         init_uart();
00170         stdout = &mystdout;
00171         printf ("Hello, world!\n");
00172         return 0;
00173     }
00174     \endcode
00175
00176     This example uses the initializer form FDEV_SETUP_STREAM() rather
00177     than the function-like fdev_setup_stream(), so all data
00178     initialization happens during C start-up.
00179
00180     If streams initialized that way are no longer needed, they can be
00181     destroyed by first calling the macro fdev_close(), and then
00182     destroying the object itself.  No call to fclose() should be
00183     issued for these streams.  While calling fclose() itself is
00184     harmless, it will cause an undefined reference to \ref a_free "free()"
```

```
00185     and thus cause the linker to pull in the \ref a_malloc "malloc"
00186     module into the application.
00187
00188 <dl>
00189 <dt>\anchor stdio_note1 Note 1:</dt>
00190 <dd>
00191     It might have been possible to implement a device abstraction that
00192     is compatible with \c fopen() but since this would have required
00193     to parse a string, and to take all the information needed either
00194     out of this string, or out of an additional table that would need to be
00195     provided by the application, this approach was not taken.
00196 </dd>
00197 <dt>\anchor stdio_note2 Note 2:</dt>
00198 <dd>
00199     This basically follows the Unix approach: if a device such as a
00200     terminal needs special handling, it is in the domain of the
00201     terminal device driver to provide this functionality.  Thus, a
00202     simple function suitable as \c put() for \c fdevopen() that talks
00203     to a UART interface might look like this:
00204
00205     \code
00206     int uart_putchar (char c, FILE *stream)
00207     {
00208         if (c == '\n')
00209             uart_putchar ('\r', stream);
00210         loop_until_bit_is_set (UCSRA, UDRE);
00211         UDR = c;
00212         return 0;
00213     }
00214     \endcode
00215 </dd>
00216 <dt>\anchor stdio_note3 Note 3:</dt>
00217 <dd>
00218     This implementation has been chosen because the cost of maintaining
00219     an alias is considerably smaller than the cost of maintaining full
00220     copies of each stream.  Yet, providing an implementation that offers
00221     the complete set of standard streams was deemed to be useful.  Not
00222     only that writing \c printf() instead of <tt>fprintf(mystream, ...)</tt>
00223     saves typing work, but since avr-gcc needs to resort to pass all
00224     arguments of variadic functions on the stack (as opposed to passing
00225     them in registers for functions that take a fixed number of
00226     parameters), the ability to pass one parameter less by implying
00227     \c stdin or stdout will also save some execution time.
00228 </dd>
00229 </dl>
00230 */
00231
00232 #if !defined(__DOXYGEN__)
00233
00234 /*
00235  * This is an internal structure of the library that is subject to be
00236  * changed without warnings at any time.  Please do *never* reference
00237  * elements of it beyond by using the official interfaces provided.
00238  */
00239 struct __file {
00240     char    *buf;       /* buffer pointer */
00241     unsigned char unget;    /* ungetc() buffer */
00242     uint8_t  flags;      /* flags, see below */
00243 #define __SRD   0x0001      /* OK to read */
00244 #define __SWR   0x0002      /* OK to write */
00245 #define __SSTR  0x0004      /* this is an sprintf/snprintf string */
00246 #define __SPGM  0x0008      /* fmt string is in progmem */
00247 #define __SERR  0x0010      /* found error */
00248 #define __SEOF  0x0020      /* found EOF */
00249 #define __SUNGET 0x040      /* ungetc() happened */
00250 #define __SMALLOC 0x80      /* handle is malloc()ed */
00251 #if 0
00252 /* possible future extensions, will require uint16_t flags */
00253 #define __SRW   0x0100      /* open for reading & writing */
00254 #define __SLBF  0x0200      /* line buffered */
00255 #define __SNBF  0x0400      /* unbuffered */
00256 #define __SMBF  0x0800      /* buf is from malloc */
00257 #endif
```

```
00258     int size;       /* size of buffer */
00259     int len;        /* characters read or written so far */
00260     int (*put)(char, struct __file *);  /* function to write one char to device */
00261     int (*get)(struct __file *);    /* function to read one char from device */
00262     void    *udata;     /* User defined and accessible data. */
00263 };
00264
00265 #endif /* not __DOXYGEN__ */
00266
00267 /**@{*/
00268 /**
00269    \c FILE is the opaque structure that is passed around between the
00270    various standard IO functions.
00271 */
00272 typedef struct __file FILE;
00273
00274 /**
00275    Stream that will be used as an input stream by the simplified
00276    functions that don't take a \c stream argument.
00277
00278    The first stream opened with read intent using \c fdevopen()
00279    will be assigned to \c stdin.
00280 */
00281 #define stdin (__iob[0])
00282
00283 /**
00284    Stream that will be used as an output stream by the simplified
00285    functions that don't take a \c stream argument.
00286
00287    The first stream opened with write intent using \c fdevopen()
00288    will be assigned to both, \c stdin, and \c stderr.
00289 */
00290 #define stdout (__iob[1])
00291
00292 /**
00293    Stream destined for error output.  Unless specifically assigned,
00294    identical to \c stdout.
00295
00296    If \c stderr should point to another stream, the result of
00297    another \c fdevopen() must be explicitly assigned to it without
00298    closing the previous \c stderr (since this would also close
00299    \c stdout).
00300 */
00301 #define stderr (__iob[2])
00302
00303 /**
00304    \c EOF declares the value that is returned by various standard IO
00305    functions in case of an error.  Since the AVR platform (currently)
00306    doesn't contain an abstraction for actual files, its origin as
00307    "end of file" is somewhat meaningless here.
00308 */
00309 #define EOF (-1)
00310
00311 /** This macro inserts a pointer to user defined data into a FILE
00312     stream object.
00313
00314     The user data can be useful for tracking state in the put and get
00315     functions supplied to the fdevopen() function. */
00316 #define fdev_set_udata(stream, u) do { (stream)->udata = u; } while(0)
00317
00318 /** This macro retrieves a pointer to user defined data from a FILE
00319     stream object. */
00320 #define fdev_get_udata(stream) ((stream)->udata)
00321
00322 #if defined(__DOXYGEN__)
00323 /**
00324    \brief Setup a user-supplied buffer as an stdio stream
00325
00326    This macro takes a user-supplied buffer \c stream, and sets it up
00327    as a stream that is valid for stdio operations, similar to one that
00328    has been obtained dynamically from fdevopen(). The buffer to setup
00329    must be of type #FILE.
00330
```

```
00331    The arguments \c put and \c get are identical to those that need to
00332    be passed to fdevopen().
00333
00334    The \c rwflag argument can take one of the values #_FDEV_SETUP_READ,
00335    #_FDEV_SETUP_WRITE, or #_FDEV_SETUP_RW, for read, write, or read/write
00336    intent, respectively.
00337
00338    \note No assignments to the standard streams will be performed by
00339    fdev_setup_stream().  If standard streams are to be used, these
00340    need to be assigned by the user.  See also under
00341    \ref stdio_without_malloc "Running stdio without malloc()".
00342  */
00343 #define fdev_setup_stream(stream, put, get, rwflag)
00344 #else  /* !DOXYGEN */
00345 #define fdev_setup_stream(stream, p, g, f) \
00346     do {                                   \
00347         (stream)->put = p;                 \
00348         (stream)->get = g;                 \
00349         (stream)->flags = f;               \
00350         (stream)->udata = 0;               \
00351     } while(0)
00352 #endif /* DOXYGEN */
00353
00354 #define _FDEV_SETUP_READ  __SRD /**< fdev_setup_stream() with read intent */
00355 #define _FDEV_SETUP_WRITE __SWR /**< fdev_setup_stream() with write intent */
00356 #define _FDEV_SETUP_RW    (__SRD|__SWR) /**< fdev_setup_stream() with read/write intent
     */
00357
00358 /**
00359    Return code for an error condition during device read.
00360
00361    To be used in the get function of fdevopen().  */
00362 #define _FDEV_ERR (-1)
00363
00364 /**
00365    Return code for an end-of-file condition during device read.
00366
00367    To be used in the get function of fdevopen().  */
00368 #define _FDEV_EOF (-2)
00369
00370 #if defined(__DOXYGEN__)
00371 /**
00372    \brief Initializer for a user-supplied stdio stream
00373
00374    This macro acts similar to fdev_setup_stream(), but it is to be
00375    used as the initializer of a variable of type FILE.
00376
00377    The remaining arguments are to be used as explained in
00378    fdev_setup_stream().
00379  */
00380 #define FDEV_SETUP_STREAM(put, get, rwflag)
00381 #else  /* !DOXYGEN */
00382 /* In order to work with C++, we have to mention the fields in the order
00383    as they appear in struct __file.  Also, designated initializers are
00384    only supported since C++20.  */
00385 #define FDEV_SETUP_STREAM(PU, GE, FL) \
00386     {                                 \
00387         (char*) 0  /* buf */,         \
00388         0u         /* unget */,       \
00389         FL         /* flags */,       \
00390         0          /* size */,        \
00391         0          /* len */,         \
00392         PU         /* put */,         \
00393         GE         /* get */,         \
00394         (void*) 0  /* udata */        \
00395     }
00396 #endif /* DOXYGEN */
00397
00398 #ifdef __cplusplus
00399 extern "C" {
00400 #endif
00401
00402 #if !defined(__DOXYGEN__)
```

```
00403 /*
00404  * Doxygen documentation can be found in fdevopen.c.
00405  */
00406
00407 extern struct __file *__iob[];
00408
00409 #if defined(__STDIO_FDEVOPEN_COMPAT_12)
00410 /*
00411  * Declare prototype for the discontinued version of fdevopen() that
00412  * has been in use up to AVR-LibC 1.2.x.  The new implementation has
00413  * some backwards compatibility with the old version.
00414  */
00415 extern FILE *fdevopen(int (*__put)(char), int (*__get)(void),
00416                       int __opts __attribute__((__unused__)));
00417 #else  /* !defined(__STDIO_FDEVOPEN_COMPAT_12) */
00418 /* New prototype for AVR-LibC 1.4 and above. */
00419 extern FILE *fdevopen(int (*__put)(char, FILE*), int (*__get)(FILE*));
00420 #endif /* defined(__STDIO_FDEVOPEN_COMPAT_12) */
00421
00422 #endif /* not __DOXYGEN__ */
00423
00424 /**
00425    This function closes \c stream, and disallows and further
00426    IO to and from it.
00427
00428    When using fdevopen() to setup the stream, a call to fclose() is
00429    needed in order to free the internal resources allocated.
00430
00431    If the stream has been set up using fdev_setup_stream() or
00432    FDEV_SETUP_STREAM(), use fdev_close() instead.
00433
00434    It currently always returns 0 (for success).
00435 */
00436 extern int fclose(FILE *__stream);
00437
00438 /**
00439    This macro frees up any library resources that might be associated
00440    with \c stream.  It should be called if \c stream is no longer
00441    needed, right before the application is going to destroy the
00442    \c stream object itself.
00443
00444    (Currently, this macro evaluates to nothing, but this might change
00445    in future versions of the library.)
00446 */
00447 #define fdev_close() ((void)0)
00448
00449
00450 /**
00451    \c vfprintf is the central facility of the \c printf family of
00452    functions.  It outputs values to \c stream under control of a
00453    format string passed in \c fmt.  The actual values to print are
00454    passed as a variable argument list \c ap.
00455
00456    \c vfprintf returns the number of characters written to \c stream,
00457    or \c EOF in case of an error.  Currently, this will only happen
00458    if \c stream has not been opened with write intent.
00459
00460    The format string is composed of zero or more directives: ordinary
00461    characters (not \c %), which are copied unchanged to the output
00462    stream; and conversion specifications, each of which results in
00463    fetching zero or more subsequent arguments.  Each conversion
00464    specification is introduced by the \c % character.  The arguments must
00465    properly correspond (after type promotion) with the conversion
00466    specifier.  After the \c %, the following appear in sequence:
00467
00468    - Zero or more of the following flags:
00469       <ul>
00470       <li> \c # The value should be converted to an "alternate form".  For
00471            c, d, i, s, and u conversions, this option has no effect.
00472            For o conversions, the precision of the number is
00473            increased to force the first character of the output
00474            string to a zero (except if a zero value is printed with
00475            an explicit precision of zero).  For x and X conversions,
```

```
00476            a non-zero result has the string `0x' (or `0X' for X
00477            conversions) prepended to it.</li>
00478       <li> \c 0 (zero) Zero padding.  For all conversions, the converted
00479            value is padded on the left with zeros rather than blanks.
00480            If a precision is given with a numeric conversion (d, i,
00481            o, u, i, x, and X), the 0 flag is ignored.</li>
00482       <li> \c - A negative field width flag; the converted value is to be
00483            left adjusted on the field boundary.  The converted value
00484            is padded on the right with blanks, rather than on the
00485            left with blanks or zeros.  A - overrides a 0 if both are
00486            given.</li>
00487       <li> ' ' (space) A blank should be left before a positive number
00488            produced by a signed conversion (d, or i).</li>
00489       <li> \c + A sign must always be placed before a number produced by a
00490            signed conversion.  A + overrides a space if both are
00491            used.</li>
00492       </ul>
00493
00494    -   An optional decimal digit string specifying a minimum field width.
00495        If the converted value has fewer characters than the field width, it
00496        will be padded with spaces on the left (or right, if the left-adjustment
00497        flag has been given) to fill out the field width.
00498    -   An optional precision, in the form of a period . followed by an
00499        optional digit string.  If the digit string is omitted, the
00500        precision is taken as zero.  This gives the minimum number of
00501        digits to appear for d, i, o, u, x, and X conversions, or the
00502        maximum number of characters to be printed from a string for \c s
00503        conversions.
00504    -   An optional \c l or \c h length modifier, that specifies that the
00505        argument for the d, i, o, u, x, or X conversion is a \c "long int"
00506        rather than \c int. The \c h is ignored, as \c "short int" is
00507        equivalent to \c int.
00508    -   A character that specifies the type of conversion to be applied.
00509
00510    The conversion specifiers and their meanings are:
00511
00512    - \c diouxX The int (or appropriate variant) argument is converted
00513            to signed decimal (d and i), unsigned octal (o), unsigned
00514            decimal (u), or unsigned hexadecimal (x and X) notation.
00515            The letters "abcdef" are used for x conversions; the
00516            letters "ABCDEF" are used for X conversions.  The
00517            precision, if any, gives the minimum number of digits that
00518            must appear; if the converted value requires fewer digits,
00519            it is padded on the left with zeros.
00520    - \c p  The <tt>void*</tt> argument is taken as an unsigned integer,
00521            and converted similarly as a <tt>%\#x</tt> command would do.
00522    - \c c  The \c int argument is converted to an \c "unsigned char", and the
00523            resulting character is written.
00524    - \c s  The <tt>char*</tt> argument is expected to be a pointer to an array
00525            of character type (pointer to a string).  Characters from
00526            the array are written up to (but not including) a
00527            terminating NUL character; if a precision is specified, no
00528            more than the number specified are written.  If a precision
00529            is given, no null character need be present; if the
00530            precision is not specified, or is greater than the size of
00531            the array, the array must contain a terminating NUL
00532            character.
00533    - \c %  A \c % is written.  No argument is converted.  The complete
00534            conversion specification is "%%".
00535    - \c eE The double argument is rounded and converted in the format
00536            \c "[-]d.ddde±dd" where there is one digit before the
00537            decimal-point character and the number of digits after it
00538            is equal to the precision; if the precision is missing, it
00539            is taken as 6; if the precision is zero, no decimal-point
00540            character appears.  An \e E conversion uses the letter \c 'E'
00541            (rather than \c 'e') to introduce the exponent.  The exponent
00542            always contains two digits; if the value is zero,
00543            the exponent is 00.
00544    - \c fF The double argument is rounded and converted to decimal notation
00545            in the format \c "[-]ddd.ddd", where the number of digits after the
00546            decimal-point character is equal to the precision specification.
00547            If the precision is missing, it is taken as 6; if the precision
00548            is explicitly zero, no decimal-point character appears.  If a
```

```
00549           decimal point appears, at least one digit appears before it.
00550     - \c gG The double argument is converted in style \c f or \c e (or
00551           \c F or \c E for \c G conversions).  The precision
00552           specifies the number of significant digits.  If the
00553           precision is missing, 6 digits are given; if the precision
00554           is zero, it is treated as 1.  Style \c e is used if the
00555           exponent from its conversion is less than -4 or greater
00556           than or equal to the precision.  Trailing zeros are removed
00557           from the fractional part of the result; a decimal point
00558           appears only if it is followed by at least one digit.
00559     - \c S  Similar to the \c s format, except the pointer is expected to
00560           point to a program-memory (ROM) string instead of a RAM string.
00561
00562     In no case does a non-existent or small field width cause truncation of a
00563     numeric field; if the result of a conversion is wider than the field
00564     width, the field is expanded to contain the conversion result.
00565
00566     Since the full implementation of all the mentioned features becomes
00567     fairly large, three different flavours of vfprintf() can be
00568     selected using linker options:
00569     - The default vfprintf() implements all the mentioned functionality
00570       except floating point conversions.
00571     - A minimized version of vfprintf() that only implements
00572       the very basic integer and string conversion facilities, but only
00573       the \c # additional option can be specified using conversion
00574       flags (these flags are parsed correctly from the format
00575       specification, but then are simply ignored).
00576     - A version with floating point-support, but with the following twists:
00577       - The argument will be converted to IEEE single for output.
00578       - When \c long \c double is a 64-bit type and \c double is a 32-bit
00579         type, then the former will only be printed as a single '?'.
00580         Rationale is to avoid 64-bit floating point arithmetic in printf
00581         when the used selected <tt>-mdouble=32</tt>.  In order to print
00582         IEEE double, it can be converted to IEEE single by hand.
00583
00584     The respective version can
00585     be requested using the following \ref gcc_minusW "link options":
00586
00587     <dl>
00588     <dt>Classic approach</dt>
00589     <dd>
00590     - The minimal version can be requested with the following options:
00591     \code
00592     -Wl,-u,vfprintf -lprintf_min
00593     \endcode
00594
00595     - If the full functionality including the floating point conversions
00596     is required, the following options should be used:
00597     \code
00598     -Wl,-u,vfprintf -lprintf_flt
00599     \endcode
00600
00601     This approach will always link the selected printf code,
00602     even when the application doesn't use printf.
00603     </dd>
00604     <dt>Since AVR-LibC v2.3</dt>
00605     <dd>
00606     - The minimal version can be requested with the following options:
00607     \code
00608     -Wl,--defsym,vfprintf=vfprintf_min
00609     \endcode
00610     - The full functionality including the floating point conversions
00611     can be requested with:
00612     \code
00613     -Wl,--defsym,vfprintf=vfprintf_flt
00614     \endcode
00615
00616     The difference to the "classic" approach is that when no printf
00617     is used in the application and <tt>-Wl,\--gc-sections</tt> is added to
00618     the linker options, then the printf code will not be pulled in.  See
00619     <a href="https://github.com/avrdudes/avr-libc/issues/654">issue \#654</a>
00620     for an example.
00621     </dd>
```

```
00622    </dl>
00623
00624    <b>Limitations:</b>
00625    - The specified width and precision can be at most 255.
00626
00627    <b>Notes:</b>
00628    - For floating-point conversions, if you link the default or minimized
00629      version of vfprintf(), the symbol <tt>?</tt> will be output.
00630      The same applies for the float version with <tt>-mdouble=32</tt>
00631      and when an IEEE double arguments is passed.
00632      For default version the width field and the "pad to left" (symbol
00633      \em minus) option will work in this case.
00634    - The \c hh length modifier is ignored (\c char argument is
00635      promouted to \c int). More exactly, this realization does not check
00636      the number of \c h symbols.
00637    - But the \c ll length modifier will to abort the output, as this
00638      realization does not operate \c long \c long arguments.
00639    - The variable width or precision field (an asterisk \c * symbol)
00640      is not realized and will to abort the output.
00641  */
00642 extern int vfprintf(FILE *__stream, const char *__fmt, va_list __ap);
00643
00644 /**
00645    Variant of \c vfprintf() that uses a \c fmt string that resides
00646    in program memory.  See also #vfprintf_F.
00647 */
00648 extern int vfprintf_P(FILE *__stream, const char *__fmt, va_list __ap);
00649
00650 /**
00651    The function \c fputc sends the character \c c (though given as type
00652    \c int) to \c stream.  It returns the character, or \c EOF in case
00653    an error occurred.
00654 */
00655 extern int fputc(int __c, FILE *__stream);
00656
00657 #if !defined(__DOXYGEN__)
00658
00659 /* putc() function implementation, required by standard */
00660 extern int putc(int __c, FILE *__stream);
00661
00662 /* putchar() function implementation, required by standard */
00663 extern int putchar(int __c);
00664
00665 #endif /* not __DOXYGEN__ */
00666
00667 /**
00668    The macro \c putc used to be a "fast" macro implementation with a
00669    functionality identical to fputc().  For space constraints, in
00670    AVR-LibC, it is just an alias for \c fputc.
00671 */
00672 #define putc(__c, __stream) fputc(__c, __stream)
00673
00674 /**
00675    The macro \c putchar sends character \c c to \c stdout.
00676 */
00677 #define putchar(__c) fputc(__c, stdout)
00678
00679 /**
00680    The function \c printf performs formatted output to stream
00681    \c stdout.  See \c vfprintf() for details.
00682 */
00683 extern int printf(const char *__fmt, ...);
00684
00685 /**
00686    Variant of \c printf() that uses a \c fmt string that resides
00687    in program memory.  See also #printf_F.
00688 */
00689 extern int printf_P(const char *__fmt, ...);
00690
00691 /**
00692    The function \c vprintf performs formatted output to stream
00693    \c stdout, taking a variable argument list as in vfprintf().
00694
```

```
00695    See vfprintf() for details.
00696 */
00697 extern int vprintf(const char *__fmt, va_list __ap);
00698
00699 /**
00700    Variant of \c printf() that sends the formatted characters
00701    to string \c s.
00702 */
00703 extern int sprintf(char *__s, const char *__fmt, ...);
00704
00705 /**
00706    Variant of \c sprintf() that uses a \c fmt string that resides
00707    in program memory.  See also #sprintf_F.
00708 */
00709 extern int sprintf_P(char *__s, const char *__fmt, ...);
00710
00711 /**
00712    Like \c sprintf(), but instead of assuming \c s to be of infinite
00713    size, no more than \c n characters (including the trailing NUL
00714    character) will be converted to \c s.
00715
00716    Returns the number of characters that would have been written to
00717    \c s if there were enough space.
00718 */
00719 extern int snprintf(char *__s, size_t __n, const char *__fmt, ...);
00720
00721 /**
00722    Variant of \c snprintf() that uses a \c fmt string that resides
00723    in program memory.  See also #snprintf_F.
00724 */
00725 extern int snprintf_P(char *__s, size_t __n, const char *__fmt, ...);
00726
00727 /**
00728    Like \c sprintf() but takes a variable argument list for the
00729    arguments.
00730 */
00731 extern int vsprintf(char *__s, const char *__fmt, va_list __ap);
00732
00733 /**
00734    Variant of \c vsprintf() that uses a \c fmt string that resides
00735    in program memory.  See also #vsprintf_F.
00736 */
00737 extern int vsprintf_P(char *__s, const char *__fmt, va_list __ap);
00738
00739 /**
00740    Like \c vsprintf(), but instead of assuming \c s to be of infinite
00741    size, no more than \c n characters (including the trailing NUL
00742    character) will be converted to \c s.
00743
00744    Returns the number of characters that would have been written to
00745    \c s if there were enough space.
00746 */
00747 extern int vsnprintf(char *__s, size_t __n, const char *__fmt, va_list __ap);
00748
00749 /**
00750    Variant of \c vsnprintf() that uses a \c fmt string that resides
00751    in program memory.  See also #vsnprintf_F.
00752 */
00753 extern int vsnprintf_P(char *__s, size_t __n, const char *__fmt, va_list __ap);
00754 /**
00755    The function \c fprintf performs formatted output to \c stream.
00756    See \c vfprintf() for details.
00757 */
00758 extern int fprintf(FILE *__stream, const char *__fmt, ...);
00759
00760 /**
00761    Variant of \c fprintf() that uses a \c fmt string that resides
00762    in program memory.  See also #fprintf_F.
00763 */
00764 extern int fprintf_P(FILE *__stream, const char *__fmt, ...);
00765
00766 /**
00767    Write the string pointed to by \c str to stream \c stream.
```

```
00768
00769    Returns 0 on success and EOF on error.
00770 */
00771 extern int fputs(const char *__str, FILE *__stream);
00772
00773 /**
00774    Variant of fputs() where \c str resides in program memory.
00775    See also #fputs_F.
00776 */
00777 extern int fputs_P(const char *__str, FILE *__stream);
00778
00779 /**
00780    Write the string pointed to by \c str, and a trailing newline
00781    character, to \c stdout.
00782 */
00783 extern int puts(const char *__str);
00784
00785 /**
00786    Variant of puts() where \c str resides in program memory.
00787    See also #puts_F.
00788 */
00789 extern int puts_P(const char *__str);
00790
00791 /**
00792    Write \c nmemb objects, \c size bytes each, to \c stream.
00793    The first byte of the first object is referenced by \c ptr.
00794
00795    Returns the number of objects successfully written, i. e.
00796    \c nmemb unless an output error occurred.
00797  */
00798 extern size_t fwrite(const void *__ptr, size_t __size, size_t __nmemb, FILE *__stream);
00799
00800 /**
00801    The function \c fgetc reads a character from \c stream.  It returns
00802    the character, or \c EOF in case end-of-file was encountered or an
00803    error occurred.  The routines feof() or ferror() must be used to
00804    distinguish between both situations.
00805 */
00806 extern int fgetc(FILE *__stream);
00807
00808 #if !defined(__DOXYGEN__)
00809
00810 /* getc() function implementation, required by standard */
00811 extern int getc(FILE *__stream);
00812
00813 /* getchar() function implementation, required by standard */
00814 extern int getchar(void);
00815
00816 #endif /* not __DOXYGEN__ */
00817
00818 /**
00819    The macro \c getc used to be a "fast" macro implementation with a
00820    functionality identical to fgetc().  For space constraints, in
00821    AVR-LibC, it is just an alias for \c fgetc.
00822 */
00823 #define getc(__stream) fgetc(__stream)
00824
00825 /**
00826    The macro \c getchar reads a character from \c stdin.  Return
00827    values and error handling is identical to fgetc().
00828 */
00829 #define getchar() fgetc(stdin)
00830
00831 /**
00832    The ungetc() function pushes the character \c c (converted to an
00833    unsigned char) back onto the input stream pointed to by \c stream.
00834    The pushed-back character will be returned by a subsequent read on
00835    the stream.
00836
00837    Currently, only a single character can be pushed back onto the
00838    stream.
00839
00840    The ungetc() function returns the character pushed back after the
```

```
00841    conversion, or \c EOF if the operation fails.  If the value of the
00842    argument \c c character equals \c EOF, the operation will fail and
00843    the stream will remain unchanged.
00844 */
00845 extern int ungetc(int __c, FILE *__stream);
00846
00847 /**
00848    Read at most <tt>size - 1</tt> bytes from \c stream, until a
00849    newline character was encountered, and store the characters in the
00850    buffer pointed to by \c str.  Unless an error was encountered while
00851    reading, the string will then be terminated with a \c NUL
00852    character.
00853
00854    If an error was encountered, the function returns NULL and sets the
00855    error flag of \c stream, which can be tested using ferror().
00856    Otherwise, a pointer to the string will be returned.  */
00857 extern char *fgets(char *__str, int __size, FILE *__stream);
00858
00859 /**
00860    Similar to fgets() except that it will operate on stream \c stdin,
00861    and the trailing newline (if any) will not be stored in the string.
00862    It is the caller's responsibility to provide enough storage to hold
00863    the characters read.  */
00864 extern char *gets(char *__str);
00865
00866 /**
00867    Read \c nmemb objects, \c size bytes each, from \c stream,
00868    to the buffer pointed to by \c ptr.
00869
00870    Returns the number of objects successfully read, i. e.
00871    \c nmemb unless an input error occurred or end-of-file was
00872    encountered.  feof() and ferror() must be used to distinguish
00873    between these two conditions.
00874  */
00875 extern size_t fread(void *__ptr, size_t __size, size_t __nmemb, FILE *__stream);
00876
00877 /**
00878    Clear the error and end-of-file flags of \c stream.
00879  */
00880 extern void clearerr(FILE *__stream);
00881
00882
00883 /**
00884    Test the end-of-file flag of \c stream.  This flag can only be cleared
00885    by a call to clearerr().
00886  */
00887 extern int feof(FILE *__stream);
00888
00889
00890 /**
00891    Test the error flag of \c stream.  This flag can only be cleared
00892    by a call to clearerr().
00893  */
00894 extern int ferror(FILE *__stream);
00895
00896
00897 extern int vfscanf(FILE *__stream, const char *__fmt, va_list __ap);
00898
00899 /**
00900    Variant of vfscanf() using a \c fmt string in program memory.
00901    See also #vfscanf_F.
00902  */
00903 extern int vfscanf_P(FILE *__stream, const char *__fmt, va_list __ap);
00904
00905 /**
00906    The function \c fscanf performs formatted input, reading the
00907    input data from \c stream.
00908
00909    See vfscanf() for details.
00910  */
00911 extern int fscanf(FILE *__stream, const char *__fmt, ...);
00912
00913 /**
```

```
00914    Variant of fscanf() using a \c fmt string in program memory.
00915    See also #fscanf_F.
00916  */
00917 extern int fscanf_P(FILE *__stream, const char *__fmt, ...);
00918
00919 /**
00920    The function \c scanf performs formatted input from stream \c stdin.
00921
00922    See vfscanf() for details.
00923  */
00924 extern int scanf(const char *__fmt, ...);
00925
00926 /**
00927    Variant of scanf() where \c fmt resides in program memory.
00928    See also #scanf_F.
00929  */
00930 extern int scanf_P(const char *__fmt, ...);
00931
00932 /**
00933    The function \c vscanf performs formatted input from stream
00934    \c stdin, taking a variable argument list as in vfscanf().
00935
00936    See vfscanf() for details.
00937 */
00938 extern int vscanf(const char *__fmt, va_list __ap);
00939
00940 /**
00941    The function \c sscanf performs formatted input, reading the
00942    input data from the buffer pointed to by \c buf.
00943
00944    See vfscanf() for details.
00945  */
00946 extern int sscanf(const char *__buf, const char *__fmt, ...);
00947
00948 /**
00949    Variant of sscanf() using a \c fmt string in program memory.
00950    See also #sscanf_F.
00951  */
00952 extern int sscanf_P(const char *__buf, const char *__fmt, ...);
00953
00954 /**
00955    Flush \c stream.
00956
00957    This is a null operation provided for source-code compatibility
00958    only, as the standard IO implementation currently does not perform
00959    any buffering.
00960  */
00961 extern int fflush(FILE *stream);
00962
00963 #ifndef __DOXYGEN__
00964 /* only mentioned for libstdc++ support, not implemented in library */
00965 #define BUFSIZ 1024
00966 #define _IONBF 0
00967 __extension__ typedef long long fpos_t;
00968 extern int fgetpos(FILE *stream, fpos_t *pos);
00969 extern FILE *fopen(const char *path, const char *mode);
00970 extern FILE *freopen(const char *path, const char *mode, FILE *stream);
00971 extern FILE *fdopen(int, const char *);
00972 extern int fseek(FILE *stream, long offset, int whence);
00973 extern int fsetpos(FILE *stream, fpos_t *pos);
00974 extern long ftell(FILE *stream);
00975 extern int fileno(FILE *);
00976 extern void perror(const char *s);
00977 extern int remove(const char *pathname);
00978 extern int rename(const char *oldpath, const char *newpath);
00979 extern void rewind(FILE *stream);
00980 extern void setbuf(FILE *stream, char *buf);
00981 extern int setvbuf(FILE *stream, char *buf, int mode, size_t size);
00982 extern FILE *tmpfile(void);
00983 extern char *tmpnam (char *s);
00984 #endif /* !__DOXYGEN__ */
00985
00986 #ifdef __cplusplus
```

```
00987 }
00988 #endif
00989
00990 /**@}*/
00991
00992 #ifndef __DOXYGEN__
00993 /*
00994  * The following constants are currently not used by AVR-LibC's
00995  * stdio subsystem.  They are defined here since the gcc build
00996  * environment expects them to be here.
00997  */
00998 #define SEEK_SET 0
00999 #define SEEK_CUR 1
01000 #define SEEK_END 2
01001
01002 #endif
01003
01004 #endif /* __ASSEMBLER */
01005
01006 #endif /* _STDIO_H_ */
```

22.69 string.h File Reference

Macros

- #define _FFS(x)

Functions

- int ffs (int __val)
- int ffsl (long __val)
- int ffsll (long long __val)
- void * memccpy (void *, const void *, int, size_t)
- void * memchr (const void *, int, size_t)
- int memcmp (const void *, const void *, size_t)
- void * memcpy (void *, const void *, size_t)
- void * memmem (const void *, size_t, const void *, size_t)
- void * memmove (void *, const void *, size_t)
- void * memrchr (const void *, int, size_t)
- void * memset (void *, int, size_t)
- char * strcat (char *, const char *)
- char * strchr (const char *, int)
- char * strchrnul (const char *, int)
- int strcmp (const char *, const char *)
- char * strcpy (char *, const char *)
- char * stpcpy (char *, const char *)
- int strcasecmp (const char *, const char *)
- char * strcasestr (const char *, const char *)
- size_t strcspn (const char *__s, const char *__reject)
- char * strdup (const char *s1)
- char * strndup (const char *s, size_t n)
- size_t strlcat (char *, const char *, size_t)
- size_t strlcpy (char *, const char *, size_t)
- size_t strlen (const char *)
- char * strlwr (char *)
- char * strncat (char *, const char *, size_t)
- int strncmp (const char *, const char *, size_t)

- char * strncpy (char *, const char *, size_t)
- int strncasecmp (const char *, const char *, size_t)
- size_t strnlen (const char *, size_t)
- char * strpbrk (const char *__s, const char *__accept)
- char * strrchr (const char *, int)
- char * strrev (char *)
- char * strsep (char **, const char *)
- size_t strspn (const char *__s, const char *__accept)
- char * strstr (const char *, const char *)
- char * strtok (char *, const char *)
- char * strtok_r (char *, const char *, char **)
- char * strupr (char *)

22.70 string.h

Go to the documentation of this file.

```
00001 /* Copyright (c) 2002,2007 Marek Michalkiewicz
00002    All rights reserved.
00003
00004    Redistribution and use in source and binary forms, with or without
00005    modification, are permitted provided that the following conditions are met:
00006
00007    * Redistributions of source code must retain the above copyright
00008      notice, this list of conditions and the following disclaimer.
00009
00010    * Redistributions in binary form must reproduce the above copyright
00011      notice, this list of conditions and the following disclaimer in
00012      the documentation and/or other materials provided with the
00013      distribution.
00014
00015    * Neither the name of the copyright holders nor the names of
00016      contributors may be used to endorse or promote products derived
00017      from this software without specific prior written permission.
00018
00019   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
00020   AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
00021   IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
00022   ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
00023   LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
00024   CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
00025   SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
00026   INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
00027   CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
00028   ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
00029   POSSIBILITY OF SUCH DAMAGE. */
00030
00031 /*
00032    string.h
00033
00034    Contributors:
00035      Created by Marek Michalkiewicz <marekm@linux.org.pl>
00036  */
00037
00038 #ifndef _STRING_H_
00039 #define _STRING_H_ 1
00040
00041 #ifndef __DOXYGEN__
00042 #define __need_NULL
00043 #define __need_size_t
00044 #include <stddef.h>
00045 #include <bits/attribs.h>
00046
00047 #endif  /* !__DOXYGEN__ */
00048
00049 #ifdef __cplusplus
00050 extern "C" {
```

```
00051 #endif
00052
00053 /** \file */
00054 /** \defgroup avr_string <string.h>: Strings
00055     \code #include <string.h> \endcode
00056
00057     The string functions perform string operations on \c NULL terminated
00058     strings.
00059
00060     \note
00061     If the strings you are working on reside in program memory (flash),
00062     then you will need to use the string functions provided by one
00063     of the following headers:
00064     - \ref avr_pgmspace
00065     - \ref avr_flash
00066     \note
00067     When the strings are defined with #PROGMEM or #PROGMEM_FAR,
00068     then use the first header.   When they are defined with #__flash
00069     or #__flashx, then use the second one.
00070 */
00071
00072
00073 /** \ingroup avr_string
00074
00075     This macro finds the first (least significant) bit set in the
00076     input value.
00077
00078     This macro is very similar to the function ffs() except that
00079     it evaluates its argument at compile-time, so it should only
00080     be applied to compile-time constant expressions where it will
00081     reduce to a constant itself.
00082     Application of this macro to expressions that are not constant
00083     at compile-time is not recommended, and might result in a huge
00084     amount of code generated.
00085
00086     \returns The _FFS() macro returns the position of the first
00087     (least significant) bit set in the word val, or 0 if no bits are set.
00088     The least significant bit is position 1.  Only 16 bits of argument
00089     are evaluated.
00090 */
00091 #if defined(__DOXYGEN__)
00092 #define _FFS(x)
00093 #else  /* !DOXYGEN */
00094 #define _FFS(x) \
00095     (1               \
00096      + (((x) & 1) == 0)      \
00097      + (((x) & 3) == 0)      \
00098      + (((x) & 7) == 0)      \
00099      + (((x) & 017) == 0)        \
00100      + (((x) & 037) == 0)        \
00101      + (((x) & 077) == 0)        \
00102      + (((x) & 0177) == 0)       \
00103      + (((x) & 0377) == 0)       \
00104      + (((x) & 0777) == 0)       \
00105      + (((x) & 01777) == 0)      \
00106      + (((x) & 03777) == 0)      \
00107      + (((x) & 07777) == 0)      \
00108      + (((x) & 017777) == 0)     \
00109      + (((x) & 037777) == 0)     \
00110      + (((x) & 077777) == 0)     \
00111      - (((x) & 0177777) == 0) * 16)
00112 #endif /* DOXYGEN */
00113
00114 /** \ingroup avr_string
00115     \fn int ffs(int val);
00116
00117     \brief This function finds the first (least significant) bit set in the input value.
00118
00119     \returns The ffs() function returns the position of the first
00120     (least significant) bit set in the word \p val, or 0 if no bits are set.
00121     The least significant bit is position 1.
00122
00123     \note For expressions that are constant at compile time, consider
```

```
00124     using the \ref _FFS macro instead.
00125 */
00126 extern int ffs(int __val) __ATTR_CONST__;
00127
00128 /** \ingroup avr_string
00129     \fn int ffsl(long val);
00130
00131     \brief Same as ffs(), for an argument of type long. */
00132 extern int ffsl(long __val) __ATTR_CONST__;
00133
00134 /** \ingroup avr_string
00135     \fn int ffsll(long long val);
00136
00137     \brief Same as ffs(), for an argument of type <tt>long long</tt>. */
00138 __extension__ extern int ffsll(long long __val) __ATTR_CONST__;
00139
00140 /** \ingroup avr_string
00141     \fn void *memccpy(void *dest, const void *src, int val, size_t len)
00142     \brief Copy memory area.
00143
00144     The memccpy() function copies no more than \p len bytes from memory
00145     area \p src to memory area \p dest, stopping when the character \p val
00146     is found.
00147
00148     \returns The memccpy() function returns a pointer to the next character
00149     in \p dest after \p val, or \c NULL if \p val was not found in the first
00150     \p len characters of \p src. */
00151 extern void *memccpy(void *, const void *, int, size_t);
00152
00153 /** \ingroup avr_string
00154     \fn void *memchr(const void *src, int val, size_t len)
00155     \brief Scan memory for a character.
00156
00157     The memchr() function scans the first len bytes of the memory area pointed
00158     to by src for the character val.  The first byte to match val (interpreted
00159     as an unsigned character) stops the operation.
00160
00161     \returns The memchr() function returns a pointer to the matching byte or
00162     NULL if the character does not occur in the given memory area.  */
00163 extern void *memchr(const void *, int, size_t) __ATTR_PURE__;
00164
00165 /** \ingroup avr_string
00166     \fn int memcmp(const void *s1, const void *s2, size_t len)
00167     \brief Compare memory areas
00168
00169     The memcmp() function compares the first len bytes of the memory areas s1
00170     and s2. The comparision is performed using unsigned char operations.
00171
00172     \returns The memcmp() function returns an integer less than, equal to, or
00173     greater than zero if the first len bytes of s1 is found, respectively, to be
00174     less than, to match, or be greater than the first len bytes of s2.
00175
00176     \note Be sure to store the result in a 16 bit variable since you may get
00177     incorrect results if you use an unsigned char or char due to truncation.
00178
00179     \warning This function is not -mint8 compatible, although if you only care
00180     about testing for equality, this function should be safe to use. */
00181 extern int memcmp(const void *, const void *, size_t) __ATTR_PURE__;
00182
00183 /** \ingroup avr_string
00184     \fn void *memcpy(void *dest, const void *src, size_t len)
00185     \brief Copy a memory area.
00186
00187     The memcpy() function copies len bytes from memory area src to memory area
00188     dest.  The memory areas may not overlap.  Use memmove() if the memory
00189     areas do overlap.
00190
00191     \returns The memcpy() function returns a pointer to dest.  */
00192 extern void *memcpy(void *, const void *, size_t);
00193
00194 /** \ingroup avr_string
00195     \fn void *memmem(const void *s1, size_t len1, const void *s2, size_t len2)
00196
```

```
00197     The memmem() function finds the start of the first occurrence of the
00198     substring \p s2 of length \p len2 in the memory area \p s1 of length
00199     \p len1.
00200
00201     \return The memmem() function returns a pointer to the beginning of
00202     the substring, or \c NULL if the substring is not found. If \p len2
00203     is zero, the function returns \p s1. */
00204 extern void *memmem(const void *, size_t, const void *, size_t) __ATTR_PURE__;
00205
00206 /** \ingroup avr_string
00207     \fn void *memmove(void *dest, const void *src, size_t len)
00208     \brief Copy memory area.
00209
00210     The memmove() function copies len bytes from memory area src to memory area
00211     dest.  The memory areas may overlap.
00212
00213     \returns The memmove() function returns a pointer to dest.  */
00214 extern void *memmove(void *, const void *, size_t);
00215
00216 /** \ingroup avr_string
00217     \fn void *memrchr(const void *src, int val, size_t len)
00218
00219     The memrchr() function is like the memchr() function, except that it
00220     searches backwards from the end of the \p len bytes pointed to by \p
00221     src instead of forwards from the front. (Glibc, GNU extension.)
00222
00223     \return The memrchr() function returns a pointer to the matching
00224     byte or \c NULL if the character does not occur in the given memory
00225     area. */
00226 extern void *memrchr(const void *, int, size_t) __ATTR_PURE__;
00227
00228 /** \ingroup avr_string
00229     \fn void *memset(void *dest, int val, size_t len)
00230     \brief Fill memory with a constant byte.
00231
00232     The memset() function fills the first len bytes of the memory area pointed
00233     to by dest with the constant byte val.
00234
00235     \returns The memset() function returns a pointer to the memory area dest. */
00236 extern void *memset(void *, int, size_t);
00237
00238 /** \ingroup avr_string
00239     \fn char *strcat(char *dest, const char *src)
00240     \brief Concatenate two strings.
00241
00242     The strcat() function appends the src string to the dest string
00243     overwriting the '\\0' character at the end of dest, and then adds a
00244     terminating '\\0' character.  The strings may not overlap, and the dest
00245     string must have enough space for the result.
00246
00247     \returns The strcat() function returns a pointer to the resulting string
00248     dest.  */
00249 extern char *strcat(char *, const char *);
00250
00251 /** \ingroup avr_string
00252     \fn char *strchr(const char *src, int val)
00253     \brief Locate character in string.
00254
00255     \returns The strchr() function returns a pointer to the first occurrence of
00256     the character \p val in the string \p src, or \c NULL if the character
00257     is not found.
00258     <br><br>
00259     Here "character" means "byte" -- these functions do not work with
00260     wide or multi-byte characters.  */
00261 #ifdef __DOXYGEN__
00262 extern char *strchr(const char *, int) __ATTR_PURE__;
00263 #else
00264 extern __ATTR_ALWAYS_INLINE__ __ATTR_GNU_INLINE__
00265 char * strchr(const char *__hay, int __val)
00266 {
00267   register char *__r24 __asm("24");
00268   register int __r22 __asm("22") = __val;
00269   __asm ("%~call strchr"
```

```
00270           : "=r" (__r24) : "0" (__hay), "r" (__r22) : "30", "31");
00271   return __r24;
00272 }
00273 #endif /* DOXYGEN */
00274
00275 /** \ingroup avr_string
00276     \fn char *strchrnul(const char *s, int c)
00277
00278     The strchrnul() function is like strchr() except that if \p c is not
00279     found in \p s, then it returns a pointer to the null byte at the end
00280     of \p s, rather than \c NULL. (Glibc, GNU extension.)
00281
00282     \return The strchrnul() function returns a pointer to the matched
00283     character, or a pointer to the null byte at the end of \p s (i.e.,
00284     \c s+strlen(s)) if the character is not found.  */
00285 extern char *strchrnul(const char *, int) __ATTR_PURE__;
00286
00287 /** \ingroup avr_string
00288     \fn int strcmp(const char *s1, const char *s2)
00289     \brief Compare two strings.
00290
00291     The strcmp() function compares the two strings \p s1 and \p s2.
00292
00293     \returns The strcmp() function returns an integer less than, equal
00294     to, or greater than zero if \p s1 is found, respectively, to be less
00295     than, to match, or be greater than \p s2. A consequence of the
00296     ordering used by strcmp() is that if \p s1 is an initial substring
00297     of \p s2, then \p s1 is considered to be "less than" \p s2. */
00298 extern int strcmp(const char *, const char *) __ATTR_PURE__;
00299
00300 /** \ingroup avr_string
00301     \fn char *strcpy(char *dest, const char *src)
00302     \brief Copy a string.
00303
00304     The strcpy() function copies the string pointed to by src (including the
00305     terminating '\\0' character) to the array pointed to by dest.  The strings
00306     may not overlap, and the destination string dest must be large enough to
00307     receive the copy.
00308
00309     \returns The strcpy() function returns a pointer to the destination
00310     string dest.
00311
00312     \see strncpy(), stpcpy(), strcpy_P(), strcpy_F().  */
00313 extern char *strcpy(char *, const char *);
00314
00315 /** \ingroup avr_string
00316     \fn char *stpcpy(char *dest, const char *src)
00317     \brief Copy a string.
00318
00319     The stpcpy() function copies the string pointed to by \p src
00320     (including the terminating '\\0' character) to the array pointed
00321     to by \p dest.
00322     The strings may not overlap, and the destination string \p dest must
00323     be large enough to receive the copy.
00324
00325     \returns The stpcpy() function returns a pointer to the <b>end</b> of
00326     the string \p dest (that is, the address of the terminating null byte)
00327     rather than the beginning.
00328
00329     \since AVR-LibC v2.3  */
00330 extern char *stpcpy(char *, const char *);
00331
00332 /** \ingroup avr_string
00333     \fn int strcasecmp(const char *s1, const char *s2)
00334     \brief Compare two strings ignoring case.
00335
00336     The strcasecmp() function compares the two strings \p s1 and \p s2,
00337     ignoring the case of the characters.
00338
00339     \returns The strcasecmp() function returns an integer less than,
00340     equal to, or greater than zero if \p s1 is found, respectively, to
00341     be less than, to match, or be greater than \p s2. A consequence of
00342     the ordering used by strcasecmp() is that if \p s1 is an initial
```

```
00343     substring of \p s2, then \p s1 is considered to be "less than"
00344     \p s2. */
00345 extern int strcasecmp(const char *, const char *) __ATTR_PURE__;
00346
00347 /** \ingroup avr_string
00348     \fn char *strcasestr(const char *s1, const char *s2)
00349
00350     The strcasestr() function finds the first occurrence of the
00351     substring \p s2 in the string \p s1. This is like strstr(), except
00352     that it ignores case of alphabetic symbols in searching for the
00353     substring. (Glibc, GNU extension.)
00354
00355     \return The strcasestr() function returns a pointer to the beginning
00356     of the substring, or \c NULL if the substring is not found. If \p s2
00357     points to a string of zero length, the function returns \p s1. */
00358 extern char *strcasestr(const char *, const char *) __ATTR_PURE__;
00359
00360 /** \ingroup avr_string
00361     \fn size_t strcspn(const char *s, const char *reject)
00362
00363     The strcspn() function calculates the length of the initial segment
00364     of \p s which consists entirely of characters not in \p reject.
00365
00366     \return The strcspn() function returns the number of characters in
00367     the initial segment of \p s which are not in the string \p reject.
00368     The terminating zero is not considered as a part of string. */
00369 extern size_t strcspn(const char *__s, const char *__reject) __ATTR_PURE__;
00370
00371 /** \ingroup avr_string
00372     \fn char *strdup(const char *s1)
00373     \brief Duplicate a string.
00374
00375     The strdup() function allocates memory and copies into it the string
00376     addressed by \p s1, including the terminating null character.
00377
00378     \warning The strdup() function calls \ref a_malloc "malloc()" to allocate
00379     the memory
00380     for the duplicated string! The user is responsible for freeing the
00381     memory by calling free().
00382
00383     \returns The strdup() function returns a pointer to the resulting string
00384     dest. If malloc() cannot allocate enough storage for the string, strdup()
00385     will return \c NULL.
00386
00387     \warning Be sure to check the return value of the strdup() function to
00388     make sure that the function has succeeded in allocating the memory!
00389 */
00390 extern char *strdup(const char *s1);
00391
00392 /** \ingroup avr_string
00393     \fn char *strndup(const char *s, size_t len)
00394     \brief Duplicate a string.
00395
00396     The strndup() function is similar to strdup(), but copies at most
00397     \p len bytes. If \p s is longer than \p len, only \p len bytes are copied,
00398     and a terminating null byte (<tt>'\0'</tt>) is added.
00399
00400     Memory for the new string is obtained with malloc(), and can be freed
00401     with free().
00402
00403     \returns The strndup() function returns the location of the newly malloc'ed
00404     memory, or \c NULL if the allocation failed.
00405 */
00406 extern char *strndup(const char *s, size_t n);
00407
00408 /** \ingroup avr_string
00409     \fn size_t strlcat(char *dst, const char *src, size_t siz)
00410     \brief Concatenate two strings.
00411
00412     Appends \p src to string \p dst of size \p siz (unlike strncat(),
00413     \p siz is the full size of \p dst, not space left).  At most \p siz-1
00414     characters will be copied.  Always \p '\\0' terminated (unless \p siz <=
00415     \p strlen(dst)).
```

```
00416
00417     \returns The strlcat() function returns strlen(src) + MIN(siz,
00418     strlen(initial dst)).  If retval >= siz, truncation occurred.  */
00419 extern size_t strlcat(char *, const char *, size_t);
00420
00421 /** \ingroup avr_string
00422     \fn size_t strlcpy(char *dst, const char *src, size_t siz)
00423     \brief Copy a string.
00424
00425     Copy \p src to string \p dst of size \p siz.  At most \p siz-1
00426     characters will be copied.  Always '\\0' terminated (unless \p siz == 0).
00427
00428     \returns The strlcpy() function returns strlen(src). If retval >= siz,
00429     truncation occurred.  */
00430 extern size_t strlcpy(char *, const char *, size_t);
00431
00432 /** \ingroup avr_string
00433     \fn size_t strlen(const char *src)
00434     \brief Calculate the length of a string.
00435
00436     The strlen() function calculates the length of the string \p src, not
00437     including the terminating '\\0' character.
00438
00439     \returns The strlen() function returns the number of characters in
00440     \p src.  */
00441 extern size_t strlen(const char *) __ATTR_PURE__;
00442
00443 /** \ingroup avr_string
00444     \fn char *strlwr(char *s)
00445     \brief Convert a string to lower case.
00446
00447     The strlwr() function will convert a string to lower case. Only the upper
00448     case alphabetic characters [A .. Z] are converted.  Non-alphabetic
00449     characters will not be changed.
00450
00451     \returns The strlwr() function returns a pointer to the converted
00452     string.  Conversion is perfomed in-place.  */
00453 extern char *strlwr(char *);
00454
00455 /** \ingroup avr_string
00456     \fn char *strncat(char *dest, const char *src, size_t len)
00457     \brief Concatenate two strings.
00458
00459     The strncat() function is similar to strcat(), except that only the first
00460     \p len characters of \p src are appended to \p dest.
00461
00462     \returns The strncat() function returns a pointer to the resulting string
00463     \p dest.  */
00464 extern char *strncat(char *, const char *, size_t);
00465
00466 /** \ingroup avr_string
00467     \fn int strncmp(const char *s1, const char *s2, size_t len)
00468     \brief Compare two strings.
00469
00470     The strncmp() function is similar to strcmp(), except it only compares the
00471     first (at most) \p len characters of \p s1 and \p s2.
00472
00473     \returns The strncmp() function returns an integer less than, equal to, or
00474     greater than zero if \p s1 (or the first \p len bytes thereof) is found,
00475     respectively, to be less than, to match, or be greater than \p s2.  */
00476 extern int strncmp(const char *, const char *, size_t) __ATTR_PURE__;
00477
00478 /** \ingroup avr_string
00479     \fn char *strncpy(char *dest, const char *src, size_t len)
00480     \brief Copy a string.
00481
00482     The strncpy() function is similar to strcpy(), except that not more than
00483     \p len bytes of \p src are copied. Thus, if there is no null byte among
00484     the first \p len bytes of \p src, the result will not be null-terminated.
00485
00486     In the case where the length of \p src is less than that of \p len,
00487     the remainder of \p dest will be padded with nulls (<tt>'\0'</tt>).
00488
```

```
00489     \returns The strncpy() function returns a pointer to the destination
00490     string \p dest.  */
00491 extern char *strncpy(char *, const char *, size_t);
00492
00493 /** \ingroup avr_string
00494     \fn int strncasecmp(const char *s1, const char *s2, size_t len)
00495     \brief Compare two strings ignoring case.
00496
00497     The strncasecmp() function is similar to strcasecmp(), except it
00498     only compares the first \p len characters of \p s1.
00499
00500     \returns The strncasecmp() function returns an integer less than,
00501     equal to, or greater than zero if \p s1 (or the first \p len bytes
00502     thereof) is found, respectively, to be less than, to match, or be
00503     greater than \p s2. A consequence of the ordering used by
00504     strncasecmp() is that if \p s1 is an initial substring of \p s2,
00505     then \p s1 is considered to be "less than" \p s2.  */
00506 extern int strncasecmp(const char *, const char *, size_t) __ATTR_PURE__;
00507
00508 /** \ingroup avr_string
00509     \fn size_t strnlen(const char *src, size_t len)
00510     \brief Determine the length of a fixed-size string.
00511
00512     The strnlen() function returns the number of characters in the string
00513     pointed to by \p src, not including the terminating '\\0' character, but at
00514     most \p len. In doing this, strnlen() looks only at the first \p len
00515     characters at \p src and never beyond \p src + \p len.
00516
00517     \returns The strnlen function returns strlen(src), if that is less than
00518     \p len, or \p len if there is no '\\0' character among the first \p len
00519     characters pointed to by \p src. */
00520 extern size_t strnlen(const char *, size_t) __ATTR_PURE__;
00521
00522 /** \ingroup avr_string
00523     \fn char *strpbrk(const char *s, const char *accept)
00524
00525     The strpbrk() function locates the first occurrence in the string
00526     \p s of any of the characters in the string \p accept.
00527
00528     \return  The strpbrk() function returns a pointer to the character
00529     in \p s that matches one of the characters in \p accept, or \c NULL
00530     if no such character is found. The terminating zero is not
00531     considered as a part of string: if one or both args are empty, the
00532     result will be \c NULL. */
00533 extern char *strpbrk(const char *__s, const char *__accept) __ATTR_PURE__;
00534
00535 /** \ingroup avr_string
00536     \fn char *strrchr(const char *src, int val)
00537     \brief Locate character in string.
00538
00539     The strrchr() function returns a pointer to the last occurrence of the
00540     character val in the string src.
00541
00542     Here "character" means "byte" -- these functions do not work with wide or
00543     multi-byte characters.
00544
00545     \returns The strrchr() function returns a pointer to the matched character
00546     or \c NULL if the character is not found. */
00547 extern char *strrchr(const char *, int) __ATTR_PURE__;
00548
00549 /** \ingroup avr_string
00550     \fn char *strrev(char *s)
00551     \brief Reverse a string.
00552
00553     The strrev() function reverses the order of the string.
00554
00555     \returns The strrev() function returns a pointer to the beginning of the
00556     reversed string.  */
00557 extern char *strrev(char *);
00558
00559 /** \ingroup avr_string
00560     \fn char *strsep(char **sp, const char *delim)
00561     \brief Parse a string into tokens.
```

```
00562
00563     The strsep() function locates, in the string referenced by \p *sp,
00564     the first occurrence of any character in the string \p delim (or the
00565     terminating '\\0' character) and replaces it with a '\\0'.  The
00566     location of the next character after the delimiter character (or \c
00567     NULL, if the end of the string was reached) is stored in \p *sp. An
00568     "empty" field, i.e. one caused by two adjacent delimiter
00569     characters, can be detected by comparing the location referenced by
00570     the pointer returned in \p *sp to '\\0'.
00571
00572     \return The strsep() function returns a pointer to the original
00573     value of \p *sp. If \p *sp is initially \c NULL, strsep() returns
00574     \c NULL. */
00575 extern char *strsep(char **, const char *);
00576
00577 /** \ingroup avr_string
00578     \fn size_t strspn(const char *s, const char *accept)
00579
00580     The strspn() function calculates the length of the initial segment
00581     of \p s which consists entirely of characters in \p accept.
00582
00583     \return  The strspn() function returns the number of characters in
00584     the initial segment of \p s which consist only of characters from \p
00585     accept. The terminating zero is not considered as a part of string. */
00586 extern size_t strspn(const char *__s, const char *__accept) __ATTR_PURE__;
00587
00588 /** \ingroup avr_string
00589     \fn char *strstr(const char *s1, const char *s2)
00590     \brief Locate a substring.
00591
00592     The strstr() function finds the first occurrence of the substring \p
00593     s2 in the string \p s1.  The terminating '\\0' characters are not
00594     compared.
00595
00596     \returns The strstr() function returns a pointer to the beginning of
00597     the substring, or \c NULL if the substring is not found. If \p s2
00598     points to a string of zero length, the function returns \p s1. */
00599 extern char *strstr(const char *, const char *) __ATTR_PURE__;
00600
00601 /** \ingroup avr_string
00602     \fn char *strtok(char *s, const char *delim)
00603     \brief Parses the string s into tokens.
00604
00605     strtok parses the string s into tokens. The first call to strtok
00606     should have s as its first argument. Subsequent calls should have
00607     the first argument set to \c NULL. If a token ends with a delimiter, this
00608     delimiting character is overwritten with a '\\0' and a pointer to the next
00609     character is saved for the next call to strtok. The delimiter string
00610     delim may be different for each call.
00611
00612     \returns The strtok() function returns a pointer to the next token or
00613     \c NULL when no more tokens are found.
00614
00615     \note strtok() is NOT reentrant. For a reentrant version of this function
00616     see \c strtok_r().
00617 */
00618 extern char *strtok(char *, const char *);
00619
00620 /** \ingroup avr_string
00621     \fn char *strtok_r(char *string, const char *delim, char **last)
00622     \brief Parses string into tokens.
00623
00624     strtok_r parses string into tokens. The first call to strtok_r
00625     should have string as its first argument. Subsequent calls should have
00626     the first argument set to \c NULL. If a token ends with a delimiter, this
00627     delimiting character is overwritten with a '\\0' and a pointer to the next
00628     character is saved for the next call to strtok_r. The delimiter string
00629     \p delim may be different for each call. \p last is a user allocated char*
00630     pointer. It must be the same while parsing the same string. strtok_r is
00631     a reentrant version of strtok().
00632
00633     \returns The strtok_r() function returns a pointer to the next token or
00634     \c NULL when no more tokens are found. */
```

```
00635 extern char *strtok_r(char *, const char *, char **);
00636
00637 /** \ingroup avr_string
00638     \fn char *strupr(char *s)
00639     \brief Convert a string to upper case.
00640
00641     The strupr() function will convert a string to upper case. Only the lower
00642     case alphabetic characters [a .. z] are converted.  Non-alphabetic
00643     characters will not be changed.
00644
00645     \returns The strupr() function returns a pointer to the converted
00646     string.  The pointer is the same as that passed in since the operation is
00647     perform in place. */
00648 extern char *strupr(char *);
00649
00650 #ifndef __DOXYGEN__
00651 /* libstdc++ compatibility, dummy declarations */
00652 extern int strcoll(const char *s1, const char *s2);
00653 extern char *strerror(int errnum);
00654 extern size_t strxfrm(char *dest, const char *src, size_t n);
00655
00656 /* strlen is common so we model its GPR footprint.  */
00657 extern __ATTR_ALWAYS_INLINE__ __ATTR_GNU_INLINE__
00658 size_t strlen(const char *__s)
00659 {
00660   if (__builtin_constant_p (__builtin_strlen (__s)))
00661     {
00662       return __builtin_strlen (__s);
00663     }
00664   else
00665     {
00666       register const char *__r24 __asm("24") = __s;
00667       register size_t __res __asm("24");
00668       __asm ("%~call %x2" : "=r" (__res) : "r" (__r24), "i" (strlen)
00669              : "30", "31", "memory");
00670       return __res;
00671     }
00672 }
00673
00674 /* strcpy is common so we model its GPR footprint.  */
00675 extern __ATTR_ALWAYS_INLINE__ __ATTR_GNU_INLINE__
00676 char* strcpy(char *__x, const char *__z)
00677 {
00678   char *__ret = __x;
00679   __asm volatile ("%~call __strcpy"
00680                   : "+x" (__x), "+z" (__z) :: "memory");
00681   return __ret;
00682 }
00683
00684 extern __ATTR_ALWAYS_INLINE__ __ATTR_GNU_INLINE__
00685 char* stpcpy(char *__x, const char *__z)
00686 {
00687   __asm volatile ("%~call __strcpy"
00688                   : "+x" (__x), "+z" (__z) :: "memory");
00689   return __x - 1;
00690 }
00691
00692 /* strcmp is common so we model its GPR footprint.  */
00693 extern __ATTR_ALWAYS_INLINE__ __ATTR_GNU_INLINE__
00694 int strcmp(const char *__x, const char *__z)
00695 {
00696   register int __ret __asm("24");
00697   __asm ("%~call __strcmp"
00698          : "=r" (__ret), "+x" (__x), "+z" (__z) :: "memory");
00699   return __ret;
00700 }
00701
00702 #endif  /* !__DOXYGEN__ */
00703
00704 #ifdef __cplusplus
00705 }
00706 #endif
00707
```

```
00708 #endif /* _STRING_H_ */
```

22.71 time.h File Reference

Data Structures

- struct tm
- struct week_date

Macros

- #define ONE_HOUR 3600
- #define ONE_DEGREE 3600
- #define ONE_DAY 86400
- #define UNIX_OFFSET 946684800
- #define NTP_OFFSET 3155673600

Typedefs

- typedef uint32_t time_t

Enumerations

- enum _WEEK_DAYS_ {
  SUNDAY , **MONDAY** , **TUESDAY** , **WEDNESDAY** ,
  THURSDAY , **FRIDAY** , **SATURDAY** }
- enum _MONTHS_ {
  JANUARY , **FEBRUARY** , **MARCH** , **APRIL** ,
  MAY , **JUNE** , **JULY** , **AUGUST** ,
  SEPTEMBER , **OCTOBER** , **NOVEMBER** , **DECEMBER** }

Functions

- time_t time (time_t *timer)
- int32_t difftime (time_t time1, time_t time0)
- time_t mktime (struct tm *timeptr)
- time_t mk_gmtime (const struct tm *timeptr)
- struct tm * gmtime (const time_t *timer)
- void gmtime_r (const time_t *timer, struct tm *timeptr)
- struct tm * localtime (const time_t *timer)
- void localtime_r (const time_t *timer, struct tm *timeptr)
- char * asctime (const struct tm *timeptr)
- void asctime_r (const struct tm *timeptr, char *buf)
- char * ctime (const time_t *timer)
- void ctime_r (const time_t *timer, char *buf)
- char * isotime (const struct tm *tmptr)
- void isotime_r (const struct tm *, char *)
- size_t strftime (char *s, size_t maxsize, const char *format, const struct tm *timeptr)
- void set_dst (int(*)(const time_t *, int32_t *))
- void set_zone (int32_t)

- void set_system_time (time_t timestamp)
- void system_tick (void)
- uint8_t is_leap_year (int16_t year)
- uint8_t month_length (int16_t year, uint8_t month)
- uint8_t week_of_year (const struct tm *timeptr, uint8_t start)
- uint8_t week_of_month (const struct tm *timeptr, uint8_t start)
- struct week_date * iso_week_date (int year, int yday)
- void iso_week_date_r (int year, int yday, struct week_date *)
- uint32_t fatfs_time (const struct tm *timeptr)
- void set_position (int32_t latitude, int32_t longitude)
- int16_t equation_of_time (const time_t *timer)
- int32_t daylight_seconds (const time_t *timer)
- time_t solar_noon (const time_t *timer)
- time_t sun_rise (const time_t *timer)
- time_t sun_set (const time_t *timer)
- float solar_declinationf (const time_t *timer)
- double solar_declination (const time_t *timer)
- long double solar_declinationl (const time_t *timer)
- int8_t moon_phase (const time_t *timer)
- uint32_t gm_sidereal (const time_t *timer)
- uint32_t lm_sidereal (const time_t *timer)

22.72 time.h

Go to the documentation of this file.

```
00001 /*
00002  * Copyright (C) 2012 Michael Duane Rice All rights reserved.
00003  *
00004  * Redistribution and use in source and binary forms, with or without
00005  * modification, are permitted provided that the following conditions are
00006  * met:
00007  *
00008  * Redistributions of source code must retain the above copyright notice, this
00009  * list of conditions and the following disclaimer. Redistributions in binary
00010  * form must reproduce the above copyright notice, this list of conditions
00011  * and the following disclaimer in the documentation and/or other materials
00012  * provided with the distribution. Neither the name of the copyright holders
00013  * nor the names of contributors may be used to endorse or promote products
00014  * derived from this software without specific prior written permission.
00015  *
00016  * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
00017  * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
00018  * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
00019  * ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
00020  * LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
00021  * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
00022  * SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
00023  * INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
00024  * CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
00025  * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
00026  * POSSIBILITY OF SUCH DAMAGE. */
00027
00028 #ifndef TIME_H
00029 #define TIME_H
00030
00031 #ifdef __cplusplus
00032 extern  "C" {
00033 #endif
00034
00035 #include <stdint.h>
00036 #include <stdlib.h>
00037
```

```
00038 /** \file */
00039
00040 /** \defgroup avr_time <time.h>: Time
00041     \code #include <time.h> \endcode
00042     <h3>Introduction to the Time functions</h3>
00043     This file declares the time functions implemented in AVR-LibC.
00044
00045     The implementation aspires to conform with ISO/IEC 9899 (C90).
00046     However, due to limitations of the target processor and the nature of
00047     its development environment, a practical implementation must of
00048     necessity deviate from the standard.
00049
00050     Section 7.23.2.1 \c clock()<br>
00051     The type \c clock_t, the macro \c CLOCKS_PER_SEC, and the function
00052     \c clock() are not implemented. We consider these items belong to
00053     operating system code, or to application code when no operating
00054     system is present.
00055
00056     Section 7.23.2.3 \c mktime()<br>
00057     The standard specifies that \c mktime() should return <tt>(time_t) - 1</tt>
00058     if the time cannot be represented.
00059     This implementation always returns a 'best effort' representation, which
00060     means that there are corner cases where non-representable times are not
00061     mapped to <tt>-1</tt>.  The purpose is to avoid 64-bit arithmetic.
00062
00063     Section 7.23.2.4 \c time()<br>
00064     The standard specifies that \c time() should return <tt>(time_t) - 1</tt>
00065     if the time is not available.  Since the application must initialize
00066     the time system, this functionality is not implemented.
00067
00068     Section 7.23.2.2, \c difftime()<br>
00069     Due to the lack of a 64-bit double, the function difftime() returns a
00070     \c long integer. In most cases this change will be invisible to the user,
00071     handled automatically by the compiler.
00072
00073     Section 7.23.1.4 \c struct \c #tm<br>
00074     Per the standard, struct <tt>tm->tm_isdst</tt> is greater than zero when
00075     Daylight Saving time is in effect.
00076     This implementation further specifies that, when positive, the value
00077     of \c tm_isdst represents
00078     the amount time is advanced during Daylight Saving time.
00079
00080     Section 7.23.3.5 \c strftime()<br>
00081     Only the 'C' locale is supported, therefore the modifiers 'E' and 'O' are
00082     ignored. The 'Z' conversion is also ignored, due to the lack of time
00083     zone name.
00084
00085     In addition to the above departures from the standard, there are some
00086     behaviors which are different from what is often expected,
00087     though allowed under the standard.
00088
00089     There is no 'platform standard' method to obtain the current time,
00090     time zone, or daylight savings 'rules' in the AVR environment.
00091     Therefore the application must initialize the time system with this
00092     information. The functions \c set_zone(), \c set_dst(), and
00093     \c set_system_time() are provided for initialization. Once initialized,
00094     system time is maintained by calling the function \c system_tick()
00095     at one second intervals.
00096
00097     Though not specified in the standard, it is often expected that \c time_t
00098     is a signed integer representing an offset in seconds from
00099     Midnight Jan 1 1970, i.e. 'Unix time'. This implementation uses an
00100     unsigned 32-bit integer offset from Midnight Jan 1 2000.
00101     The use of this 'epoch' helps to simplify the conversion functions,
00102     while the 32-bit value allows time to be properly represented until
00103     Tue Feb 7 06:28:15 2136 UTC. The macros \c #UNIX_OFFSET and \c #NTP_OFFSET
00104     are defined to assist in converting to and from Unix and NTP time stamps.
00105
00106     Unlike desktop counterparts, it is impractical to implement or maintain
00107     the 'zoneinfo' database.
00108     Therefore no attempt is made to account for time zone, daylight saving,
00109     or leap seconds in past dates.  All calculations are made according to
00110     the currently configured time zone and daylight saving 'rule'.
```

```
00111
00112     In addition to C standard functions, re-entrant versions of \c ctime(),
00113     \c asctime(), \c gmtime() and \c localtime() are provided which,
00114     in addition to being re-entrant, have the property of claiming
00115     less permanent storage in RAM. An additional time conversion, \c isotime()
00116     and its re-entrant version, uses far less storage than either \c ctime()
00117     or \c asctime().
00118
00119     Along with the usual smattering of utility functions, such as
00120     \c is_leap_year(), this library includes
00121     a set of functions related the sun and moon, as well as sidereal
00122     time functions.
00123 */
00124
00125 /** \ingroup avr_time */
00126 /**@{*/
00127
00128 /**
00129     \c time_t represents seconds elapsed from Midnight, Jan 1 2000 UTC
00130     (the Y2K 'epoch').  Its range allows this implementation to represent
00131     time up to Tue Feb 7 06:28:15 2136 UTC. */
00132 typedef uint32_t time_t;
00133
00134 /**
00135     The \c time function returns the systems current time stamp.
00136     If timer is not a null pointer, the return value is also assigned
00137     to the object it points to.  */
00138 time_t time(time_t *timer);
00139
00140 /**
00141     The difftime function returns the difference between two binary time stamps,
00142     time1 - time0.
00143 */
00144 int32_t difftime(time_t time1, time_t time0);
00145
00146
00147 /**
00148     The \c tm structure contains a representation of time 'broken down' into
00149     components of the Gregorian calendar.
00150
00151     The value of tm_isdst is zero if Daylight Saving Time is not in effect,
00152     and is negative if the information is not available.
00153
00154     When Daylight Saving Time is in effect, the value represents the number of
00155     seconds the clock is advanced.
00156
00157     See the \c set_dst() function for more information about Daylight Saving.
00158 */
00159 struct tm
00160 {
00161     int8_t  tm_sec; /**< Seconds after the minute (0 to 59) */
00162     int8_t  tm_min; /**< Minutes after the hour (0 to 59) */
00163     int8_t  tm_hour; /**< Hours since midnight (0 to 23) */
00164     int8_t  tm_mday; /**< Day of the month (1 to 31) */
00165     int8_t  tm_wday; /**< Days since Sunday (0 to 6) */
00166     int8_t  tm_mon; /**< Months since January (0 to 11) */
00167     int16_t tm_year; /**< Years since 1900 */
00168     int16_t tm_yday; /**< Days since January 1 (0 to 365) */
00169     int16_t tm_isdst; /**< Daylight Saving Time flag */
00170 };
00171
00172 #ifndef __DOXYGEN__
00173 /* We have to provide \c clock_t / \c CLOCKS_PER_SEC so that libstdc++-v3 can
00174    be built.  We define \c CLOCKS_PER_SEC via a symbol \c _CLOCKS_PER_SEC_
00175    so that the user can provide the value on the link line, which should
00176    result in little or no run-time overhead compared with a constant.  */
00177 typedef uint32_t clock_t;
00178 extern char *_CLOCKS_PER_SEC_;
00179 #define CLOCKS_PER_SEC ((clock_t) _CLOCKS_PER_SEC_)
00180 extern clock_t clock(void);
00181 #endif  /* !__DOXYGEN__ */
00182
00183 /**
```

```
00184     This function 'compiles' the elements of a broken-down time structure,
00185     returning a binary time stamp.
00186     The elements of timeptr are interpreted as representing Local Time.
00187
00188     The original values of the \c tm_wday and tm_yday elements of the structure
00189     are ignored, and the original values of the other elements are not
00190     restricted to the ranges stated for struct tm.
00191
00192     The element \c tm_isdst is used for input and output. If set to 0 or a
00193     positive value on input, this requests calculation for Daylight Savings
00194     Time being off or on, respectively. If set to a negative value on input,
00195     it requests calculation to return whether Daylight Savings Time is in
00196     effect or not according to the other values.
00197
00198     On successful completion, the values of all elements of timeptr are set
00199     to the appropriate range.  */
00200 time_t mktime(struct tm * timeptr);
00201
00202 /**
00203     This function 'compiles' the elements of a broken-down time structure,
00204     returning a binary time stamp.  The elements of \c timeptr are interpreted
00205     as representing UTC.
00206
00207     The original values of the tm_wday and tm_yday elements of the structure
00208     are ignored, and the original values of the other elements are not
00209     restricted to the ranges stated for struct \c #tm.
00210
00211     Unlike \c mktime(), this function \e does \e not modify the elements of
00212     \c timeptr. */
00213 time_t mk_gmtime(const struct tm * timeptr);
00214
00215 /**
00216     The gmtime function converts the time stamp pointed to by timer into
00217     broken-down time, expressed as UTC.  */
00218 struct tm *gmtime(const time_t * timer);
00219
00220 /**
00221     Re entrant version of \c gmtime().  */
00222 void gmtime_r(const time_t * timer, struct tm * timeptr);
00223
00224 /**
00225     The localtime function converts the time stamp pointed to by timer
00226     into broken-down time, expressed as Local time.  */
00227 struct tm *localtime(const time_t * timer);
00228
00229 /**
00230     Re entrant version of \c localtime().  */
00231 void localtime_r(const time_t * timer, struct tm * timeptr);
00232
00233 /**
00234     The asctime function converts the broken-down time of timeptr,
00235     into an ASCII string in the form
00236     \code Sun Mar 23 01:03:52 2013 \endcode  */
00237 char *asctime(const struct tm * timeptr);
00238
00239 /**
00240     Re entrant version of \c asctime().
00241 */
00242 void asctime_r(const struct tm * timeptr, char *buf);
00243
00244 /**
00245     The ctime function is equivalent to <tt>asctime(localtime(timer))</tt>.  */
00246 char *ctime(const time_t * timer);
00247
00248 /**
00249     Re entrant version of \c ctime(). */
00250 void ctime_r(const time_t * timer, char *buf);
00251
00252 /**
00253 The isotime function constructs an ASCII string in the form
00254     \code 2013-03-23 01:03:52 \endcode */
00255 char           *isotime(const struct tm * tmptr);
00256
```

```
00257 /**
00258     Re entrant version of isotime()
00259 */
00260 void isotime_r(const struct tm *, char *);
00261
00262 /**
00263     A complete description of \c strftime() is beyond the pale of this document.
00264     Refer to ISO/IEC document 9899 for details.
00265
00266     All conversions are made using the C Locale, ignoring the E or O modifiers.
00267     Due to the lack of a time zone 'name', the 'Z' conversion is also ignored.
00268 */
00269 size_t strftime(char *s, size_t maxsize, const char *format, const struct tm * timeptr);
00270
00271 /**
00272     Specify the Daylight Saving function.
00273
00274     The Daylight Saving function should examine its parameters to determine
00275     whether Daylight Saving is in effect, and return a value appropriate
00276     for \c tm_isdst.
00277
00278     Working examples for the USA and the EU are available:
00279
00280         \code #include <util/eu_dst.h>\endcode
00281         for the European Union, and
00282         \code #include <util/usa_dst.h>\endcode
00283         for the United States.
00284
00285     If a Daylight Saving function is not specified, the system will
00286     ignore Daylight Saving. */
00287 void set_dst(int (*) (const time_t *, int32_t *));
00288
00289 /**
00290     Set the 'time zone'. The parameter is given in seconds East of the
00291     Prime Meridian.  Example for New York City:
00292     \code set_zone(-5 * ONE_HOUR);\endcode
00293
00294     If the time zone is not set, the time system will operate in UTC only. */
00295 void set_zone(int32_t);
00296
00297 /**
00298     Initialize the system time. Examples are...
00299
00300     From a Clock / Calendar type RTC:
00301     \code
00302     struct tm rtc_time;
00303
00304     read_rtc(&rtc_time);
00305     rtc_time.tm_isdst = 0;
00306     set_system_time( mktime(&rtc_time) );
00307     \endcode
00308
00309     From a Network Time Protocol time stamp:
00310     \code
00311     set_system_time(ntp_timestamp - NTP_OFFSET);
00312     \endcode
00313
00314     From a UNIX time stamp:
00315      \code
00316     set_system_time(unix_timestamp - UNIX_OFFSET);
00317     \endcode  */
00318 void set_system_time(time_t timestamp);
00319
00320 /**
00321     Maintain the system time by calling this function at a rate of 1 Hertz.
00322
00323     It is anticipated that this function will typically be called from within
00324     an Interrupt Service Routine, (though that is not required). It therefore
00325     includes code which makes it simple to use from within a naked ISR,
00326     avoiding the cost of saving and restoring all the CPU registers.
00327
00328     Such an ISR may resemble the following example:
00329     \code
```

```
00330         ISR(RTC_OVF_vect, ISR_NAKED)
00331         {
00332             system_tick();
00333             reti();
00334         }
00335     \endcode  */
00336 void system_tick(void);
00337 
00338 /**
00339     Enumerated labels for the days of the week.
00340 */
00341 enum _WEEK_DAYS_
00342 {
00343     SUNDAY,
00344     MONDAY,
00345     TUESDAY,
00346     WEDNESDAY,
00347     THURSDAY,
00348     FRIDAY,
00349     SATURDAY
00350 };
00351 
00352 /**
00353     Enumerated labels for the months.
00354 */
00355 enum _MONTHS_
00356 {
00357     JANUARY,
00358     FEBRUARY,
00359     MARCH,
00360     APRIL,
00361     MAY,
00362     JUNE,
00363     JULY,
00364     AUGUST,
00365     SEPTEMBER,
00366     OCTOBER,
00367     NOVEMBER,
00368     DECEMBER
00369 };
00370 
00371 /**
00372     Return 1 if year is a leap year, zero if it is not.
00373 */
00374 uint8_t         is_leap_year(int16_t year);
00375 
00376 /**
00377     Return the length of month, given the year and month, where month is in
00378     the range 1 to 12.  */
00379 uint8_t         month_length(int16_t year, uint8_t month);
00380 
00381 /**
00382     Return the calendar week of year, where week 1 is considered to begin
00383     on the day of week specified by \p start. The returned value may range
00384     from zero to 52.  */
00385 uint8_t         week_of_year(const struct tm * timeptr, uint8_t start);
00386 
00387 /**
00388     Return the calendar week of month, where the first week is considered
00389     to begin on the day of week specified by \p start.
00390     The returned value may range from zero to 5.  */
00391 uint8_t         week_of_month(const struct tm * timeptr, uint8_t start);
00392 
00393 /**
00394     Structure which represents a date as a year, week number of that year,
00395     and day of week.
00396     See http://en.wikipedia.org/wiki/ISO_week_date for more information.
00397 */
00398 struct week_date {
00399     int year; /**< Year number (Gregorian calendar) */
00400     int week; /**< Week number (#1 is where first Thursday is in) */
00401     int day; /**< Day within week */
00402 };
```

```
00403
00404 /**
00405     Return a week_date structure with the ISO_8601 week based date
00406     corresponding to the given year and day of year.
00407     See http://en.wikipedia.org/wiki/ISO_week_date for more information.  */
00408 struct week_date * iso_week_date (int year, int yday);
00409
00410 /**
00411     Re-entrant version of \c iso_week_date().
00412 */
00413 void iso_week_date_r (int year, int yday, struct week_date *);
00414
00415 /**
00416     Convert a Y2K time stamp into a FAT file system time stamp.  */
00417 uint32_t        fatfs_time(const struct tm * timeptr);
00418
00419 /** One hour, expressed in seconds */
00420 #define ONE_HOUR 3600
00421
00422 /** Angular degree, expressed in arc seconds */
00423 #define ONE_DEGREE 3600
00424
00425 /** One day, expressed in seconds */
00426 #define ONE_DAY 86400
00427
00428 /** Difference between the Y2K and the UNIX epochs, in seconds. To convert a Y2K
00429     timestamp to UNIX:
00430     \code
00431     long unix;
00432     time_t y2k;
00433
00434     y2k = time(NULL);
00435     unix = y2k + UNIX_OFFSET;
00436     \endcode
00437 */
00438 #define UNIX_OFFSET 946684800
00439
00440 /** Difference between the Y2K and the NTP epochs, in seconds. To convert a Y2K
00441     timestamp to NTP:
00442     \code
00443     unsigned long ntp;
00444     time_t y2k;
00445
00446     y2k = time(NULL);
00447     ntp = y2k + NTP_OFFSET;
00448     \endcode
00449 */
00450 #define NTP_OFFSET 3155673600
00451
00452 /*
00453  * ======================================================================
00454  *                              Ephemera
00455  */
00456
00457 /**
00458     Set the geographic coordinates of the 'observer', for use with several
00459     of the following functions. Parameters are passed as seconds of
00460     North Latitude, and seconds of East Longitude.
00461
00462     For New York City:
00463     \code set_position (40.7142 * ONE_DEGREE, -74.0064 * ONE_DEGREE); \endcode
00464 */
00465 void set_position(int32_t latitude, int32_t longitude);
00466
00467 /**
00468     Computes the difference between apparent solar time and mean solar time.
00469     The returned value is in seconds.
00470 */
00471 int16_t equation_of_time(const time_t * timer);
00472
00473 /**
00474     Computes the amount of time the sun is above the horizon, at the
00475     location of the observer.
```

```
00476
00477     Note: At observer locations inside a polar circle, this value can be
00478     zero during the winter, and can exceed \c ONE_DAY during the summer.
00479
00480     The returned value is in seconds.  */
00481 int32_t daylight_seconds(const time_t * timer);
00482
00483 /**
00484     Computes the time of solar noon, at the location of the observer.
00485 */
00486 time_t solar_noon(const time_t * timer);
00487
00488 /**
00489     Return the time of sunrise, at the location of the observer.
00490     See the note about \c daylight_seconds().  */
00491 time_t sun_rise(const time_t * timer);
00492
00493 /**
00494     Return the time of sunset, at the location of the observer.
00495     See the note about \c daylight_seconds().  */
00496 time_t sun_set(const time_t * timer);
00497
00498 /**
00499     Returns the declination of the sun in radians.  */
00500 float solar_declinationf(const time_t * timer);
00501
00502 /**
00503     Returns the declination of the sun in radians.
00504
00505     This implementation is only available when \c double is a 32-bit type.  */
00506 double solar_declination(const time_t * timer);
00507
00508 /**
00509     Returns the declination of the sun in radians.
00510
00511     This implementation is only available when <tt>long double</tt> is
00512     a 32-bit type.  */
00513 long double solar_declinationl(const time_t * timer);
00514
00515 /**
00516     Returns an approximation to the phase of the moon.
00517     The sign of the returned value indicates a waning or waxing phase.
00518     The magnitude of the returned value indicates the percentage illumination.
00519 */
00520 int8_t moon_phase(const time_t * timer);
00521
00522 /**
00523     Returns Greenwich Mean Sidereal Time, as seconds into the sidereal day.
00524     The returned value will range from 0 through 86399 seconds.
00525 */
00526 uint32_t gm_sidereal(const time_t * timer);
00527
00528 /**
00529     Returns Local Mean Sidereal Time, as seconds into the sidereal day.
00530     The returned value will range from 0 through 86399 seconds.
00531 */
00532 uint32_t lm_sidereal(const time_t * timer);
00533
00534 /**@}*/
00535 #ifdef __cplusplus
00536 }
00537 #endif
00538
00539 #endif /* TIME_H  */
```

22.73 atomic.h File Reference

Macros

- #define ATOMIC_BLOCK(type)

- #define NONATOMIC_BLOCK(type)
- #define ATOMIC_RESTORESTATE
- #define ATOMIC_FORCEON
- #define NONATOMIC_RESTORESTATE
- #define NONATOMIC_FORCEOFF

22.74 atomic.h

Go to the documentation of this file.

```
00001 /* Copyright (c) 2007 Dean Camera
00002    All rights reserved.
00003
00004    Redistribution and use in source and binary forms, with or without
00005    modification, are permitted provided that the following conditions are met:
00006
00007    * Redistributions of source code must retain the above copyright
00008      notice, this list of conditions and the following disclaimer.
00009
00010    * Redistributions in binary form must reproduce the above copyright
00011      notice, this list of conditions and the following disclaimer in
00012      the documentation and/or other materials provided with the
00013      distribution.
00014
00015    * Neither the name of the copyright holders nor the names of
00016      contributors may be used to endorse or promote products derived
00017      from this software without specific prior written permission.
00018
00019   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
00020   AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
00021   IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
00022   ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
00023   LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
00024   CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
00025   SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
00026   INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
00027   CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
00028   ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
00029   POSSIBILITY OF SUCH DAMAGE. */
00030
00031 #ifndef _UTIL_ATOMIC_H_
00032 #define _UTIL_ATOMIC_H_ 1
00033
00034 #include <avr/io.h>
00035 #include <avr/interrupt.h>
00036 #include <bits/attribs.h>
00037
00038 #if !defined(__DOXYGEN__)
00039 /* Internal helper functions. */
00040 static __ATTR_ALWAYS_INLINE__ uint8_t __iSeiRetVal(void)
00041 {
00042     sei();
00043     return 1;
00044 }
00045
00046 static __ATTR_ALWAYS_INLINE__ uint8_t __iCliRetVal(void)
00047 {
00048     cli();
00049     return 1;
00050 }
00051
00052 static __ATTR_ALWAYS_INLINE__ void __iSeiParam(const uint8_t *__s)
00053 {
00054     sei();
00055     (void)__s;
00056 }
00057
00058 static __ATTR_ALWAYS_INLINE__ void __iCliParam(const uint8_t *__s)
00059 {
```

```
00060     cli();
00061     (void)__s;
00062 }
00063
00064 static __ATTR_ALWAYS_INLINE__ void __iRestore(const  uint8_t *__s)
00065 {
00066     SREG = *__s;
00067     __asm__ __volatile__ ("" ::: "memory");
00068 }
00069 #endif  /* !__DOXYGEN__ */
00070
00071 /** \file */
00072 /** \defgroup util_atomic <util/atomic.h> Atomically and Non-Atomically Executed Code
      Blocks
00073
00074     \code
00075     #include <util/atomic.h>
00076     \endcode
00077
00078     \note The macros in this header file require the ISO/IEC 9899:1999
00079     ("ISO C99") feature of for loop variables that are declared inside
00080     the for loop itself.  For that reason, this header file can only
00081     be used if the standard level of the compiler (option --std=) is
00082     set to either \c c99, \c gnu99 or higher.
00083
00084     The macros in this header file deal with code blocks that are
00085     guaranteed to be executed Atomically or Non-Atomically.  The term
00086     "Atomic" in this context refers to the inability of the respective
00087     code to be interrupted.
00088
00089     These macros operate via automatic manipulation of the Global
00090     Interrupt Status (I) bit of the SREG register. Exit paths from
00091     both block types are all managed automatically without the need
00092     for special considerations, i.e. the interrupt status will be
00093     restored to the same value it had when entering the respective
00094     block (unless ATOMIC_FORCEON or NONATOMIC_FORCEOFF are used).
00095     \warning The features in this header are implemented by means of
00096     a for loop.  This means that commands like \c break and \c continue
00097     that are located in an atomic block refer to the atomic for loop,
00098     not to a loop construct that hosts the atomic block.
00099
00100     A typical example that requires atomic access is a 16 (or more)
00101     bit variable that is shared between the main execution path and an
00102     ISR.  While declaring such a variable as volatile ensures that the
00103     compiler will not optimize accesses to it away, it does not
00104     guarantee atomic access to it.  Assuming the following example:
00105
00106     \code
00107 #include <stdint.h>
00108 #include <avr/interrupt.h>
00109 #include <avr/io.h>
00110
00111 volatile uint16_t ctr;
00112
00113 ISR(TIMER1_OVF_vect)
00114 {
00115   ctr--;
00116 }
00117
00118 ...
00119 int
00120 main(void)
00121 {
00122    ...
00123    ctr = 0x200;
00124    start_timer();
00125    while (ctr != 0)
00126      // wait
00127        ;
00128    ...
00129 }
00130     \endcode
00131
```

```
00132     There is a chance where the main context will exit its wait loop
00133     when the variable \c ctr just reached the value 0xFF.  This happens
00134     because the compiler cannot natively access a 16-bit variable
00135     atomically in an 8-bit CPU.  So the variable is for example at
00136     0x100, the compiler then tests the low byte for 0, which succeeds.
00137     It then proceeds to test the high byte, but that moment the ISR
00138     triggers, and the main context is interrupted.  The ISR will
00139     decrement the variable from 0x100 to 0xFF, and the main context
00140     proceeds.  It now tests the high byte of the variable which is
00141     (now) also 0, so it concludes the variable has reached 0, and
00142     terminates the loop.
00143
00144     Using the macros from this header file, the above code can be
00145     rewritten like:
00146
00147     \code
00148 #include <stdint.h>
00149 #include <avr/interrupt.h>
00150 #include <avr/io.h>
00151 #include <util/atomic.h>
00152
00153 volatile uint16_t ctr;
00154
00155 ISR(TIMER1_OVF_vect)
00156 {
00157   ctr--;
00158 }
00159
00160 ...
00161 int
00162 main(void)
00163 {
00164    ...
00165    ctr = 0x200;
00166    start_timer();
00167    sei();
00168    uint16_t ctr_copy;
00169    do
00170    {
00171      ATOMIC_BLOCK(ATOMIC_FORCEON)
00172      {
00173        ctr_copy = ctr;
00174      }
00175    }
00176    while (ctr_copy != 0);
00177    ...
00178 }
00179     \endcode
00180
00181     This will install the appropriate interrupt protection before
00182     accessing variable \c ctr, so it is guaranteed to be consistently
00183     tested.  If the global interrupt state were uncertain before
00184     entering the #ATOMIC_BLOCK, it should be executed with the
00185     parameter #ATOMIC_RESTORESTATE rather than #ATOMIC_FORCEON.
00186
00187     See \ref optim_code_reorder for things to be taken into account
00188     with respect to compiler optimizations.
00189 */
00190
00191 /** \def ATOMIC_BLOCK(type)
00192     \ingroup util_atomic
00193
00194     Creates a block of code that is guaranteed to be executed
00195     atomically. Upon entering the block the Global Interrupt Status
00196     flag in SREG is disabled, and re-enabled upon exiting the block
00197     from any exit path.
00198
00199     Two possible macro parameters are permitted, #ATOMIC_RESTORESTATE
00200     and #ATOMIC_FORCEON.
00201 */
00202 #if defined(__DOXYGEN__)
00203 #define ATOMIC_BLOCK(type)
00204 #else
```

```
00205 #define ATOMIC_BLOCK(type) for ( type, __ToDo = __iCliRetVal(); \
00206                                  __ToDo ; __ToDo = 0 )
00207 #endif  /* __DOXYGEN__ */
00208
00209 /** \def NONATOMIC_BLOCK(type)
00210     \ingroup util_atomic
00211
00212     Creates a block of code that is executed non-atomically. Upon
00213     entering the block the Global Interrupt Status flag in SREG is
00214     enabled, and disabled upon exiting the block from any exit
00215     path. This is useful when nested inside ATOMIC_BLOCK sections,
00216     allowing for non-atomic execution of small blocks of code while
00217     maintaining the atomic access of the other sections of the parent
00218     ATOMIC_BLOCK.
00219
00220     Two possible macro parameters are permitted,
00221     #NONATOMIC_RESTORESTATE and #NONATOMIC_FORCEOFF.
00222 */
00223 #if defined(__DOXYGEN__)
00224 #define NONATOMIC_BLOCK(type)
00225 #else
00226 #define NONATOMIC_BLOCK(type) for ( type, __ToDo = __iSeiRetVal(); \
00227                                     __ToDo ;  __ToDo = 0 )
00228 #endif  /* __DOXYGEN__ */
00229
00230 /** \def ATOMIC_RESTORESTATE
00231     \ingroup util_atomic
00232
00233     This is a possible parameter for #ATOMIC_BLOCK. When used, it will
00234     cause the ATOMIC_BLOCK to restore the previous state of the SREG
00235     register, saved before the Global Interrupt Status flag bit was
00236     disabled. The net effect of this is to make the ATOMIC_BLOCK's
00237     contents guaranteed atomic, without changing the state of the
00238     Global Interrupt Status flag when execution of the block
00239     completes.
00240 */
00241 #if defined(__DOXYGEN__)
00242 #define ATOMIC_RESTORESTATE
00243 #else
00244 #define ATOMIC_RESTORESTATE uint8_t sreg_save \
00245     __attribute__((__cleanup__(__iRestore))) = SREG
00246 #endif  /* __DOXYGEN__ */
00247
00248 /** \def ATOMIC_FORCEON
00249     \ingroup util_atomic
00250
00251     This is a possible parameter for #ATOMIC_BLOCK. When used, it will
00252     cause the ATOMIC_BLOCK to force the state of the SREG register on
00253     exit, enabling the Global Interrupt Status flag bit. This saves a
00254     small amount of flash space, a register, and one or more processor
00255     cycles, since the previous value of the SREG register does not need
00256     to be saved at the start of the block.
00257
00258     Care should be taken that ATOMIC_FORCEON is only used when it is
00259     known that interrupts are enabled before the block's execution or
00260     when the side effects of enabling global interrupts at the block's
00261     completion are known and understood.
00262 */
00263 #if defined(__DOXYGEN__)
00264 #define ATOMIC_FORCEON
00265 #else
00266 #define ATOMIC_FORCEON uint8_t sreg_save \
00267     __attribute__((__cleanup__(__iSeiParam))) = 0
00268 #endif  /* __DOXYGEN__ */
00269
00270 /** \def NONATOMIC_RESTORESTATE
00271     \ingroup util_atomic
00272
00273     This is a possible parameter for #NONATOMIC_BLOCK. When used, it
00274     will cause the NONATOMIC_BLOCK to restore the previous state of
00275     the SREG register, saved before the Global Interrupt Status flag
00276     bit was enabled. The net effect of this is to make the
00277     NONATOMIC_BLOCK's contents guaranteed non-atomic, without changing
```

```
00278     the state of the Global Interrupt Status flag when execution of
00279     the block completes.
00280 */
00281 #if defined(__DOXYGEN__)
00282 #define NONATOMIC_RESTORESTATE
00283 #else
00284 #define NONATOMIC_RESTORESTATE uint8_t sreg_save \
00285     __attribute__((__cleanup__(__iRestore))) = SREG
00286 #endif  /* __DOXYGEN__ */
00287
00288 /** \def NONATOMIC_FORCEOFF
00289     \ingroup util_atomic
00290
00291     This is a possible parameter for #NONATOMIC_BLOCK. When used, it
00292     will cause the NONATOMIC_BLOCK to force the state of the SREG
00293     register on exit, disabling the Global Interrupt Status flag
00294     bit.  This saves a small amount of flash space, a register, and one
00295     or more processor cycles, since the previous value of the SREG
00296     register does not need to be saved at the start of the block.
00297
00298     Care should be taken that NONATOMIC_FORCEOFF is only used when it
00299     is known that interrupts are disabled before the block's execution
00300     or when the side effects of disabling global interrupts at the
00301     block's completion are known and understood.
00302 */
00303 #if defined(__DOXYGEN__)
00304 #define NONATOMIC_FORCEOFF
00305 #else
00306 #define NONATOMIC_FORCEOFF uint8_t sreg_save \
00307     __attribute__((__cleanup__(__iCliParam))) = 0
00308 #endif  /* __DOXYGEN__ */
00309
00310 #endif
```

22.75 crc16.h File Reference

Functions

- static uint16_t _crc16_update (uint16_t __crc, uint8_t __data)
- static uint16_t _crc_xmodem_update (uint16_t __crc, uint8_t __data)
- static uint16_t _crc_ccitt_update (uint16_t __crc, uint8_t __data)
- static uint8_t _crc_ibutton_update (uint8_t __crc, uint8_t __data)
- static uint8_t _crc8_ccitt_update (uint8_t __crc, uint8_t __data)

22.76 crc16.h

Go to the documentation of this file.

```
00001 /* Copyright (c) 2002, 2003, 2004  Marek Michalkiewicz
00002    Copyright (c) 2005, 2007 Joerg Wunsch
00003    Copyright (c) 2013 Dave Hylands
00004    Copyright (c) 2013 Frederic Nadeau
00005    All rights reserved.
00006
00007    Redistribution and use in source and binary forms, with or without
00008    modification, are permitted provided that the following conditions are met:
00009
00010    * Redistributions of source code must retain the above copyright
00011      notice, this list of conditions and the following disclaimer.
00012
00013    * Redistributions in binary form must reproduce the above copyright
00014      notice, this list of conditions and the following disclaimer in
00015      the documentation and/or other materials provided with the
00016      distribution.
00017
```

```
00018    * Neither the name of the copyright holders nor the names of
00019      contributors may be used to endorse or promote products derived
00020      from this software without specific prior written permission.
00021
00022   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
00023   AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
00024   IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
00025   ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
00026   LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
00027   CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
00028   SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
00029   INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
00030   CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
00031   ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
00032   POSSIBILITY OF SUCH DAMAGE. */
00033
00034 #ifndef _UTIL_CRC16_H_
00035 #define _UTIL_CRC16_H_
00036
00037 #include <stdint.h>
00038 #include <bits/attribs.h>
00039
00040 /** \file */
00041 /** \defgroup util_crc <util/crc16.h>: CRC Computations
00042     \code#include <util/crc16.h>\endcode
00043
00044     This header file provides optimized inline functions for calculating
00045     cyclic redundancy checks (CRC) using common polynomials.
00046
00047     A typical application would look like:
00048
00049     \code
00050     // Dallas iButton test vector.
00051     uint8_t serno[] = { 0x02, 0x1c, 0xb8, 0x01, 0, 0, 0, 0xa2 };
00052
00053     int
00054     checkcrc (void)
00055     {
00056         uint8_t crc = 0, i;
00057
00058         for (i = 0; i < sizeof serno / sizeof serno[0]; i++)
00059             crc = _crc_ibutton_update (crc, serno[i]);
00060
00061         return crc; // must be 0
00062     }
00063     \endcode
00064
00065     \par References:
00066     See the Dallas Semiconductor app note 27 for 8051 assembler example and
00067     general CRC optimization suggestions. The table on the last page of the
00068     app note is the key to understanding these implementations.
00069     \par
00070     Jack Crenshaw's "Implementing CRCs" article in the January 1992 issue of \e
00071     Embedded \e Systems \e Programming. This may be difficult to find, but it
00072     explains CRC's in very clear and concise terms. Well worth the effort to
00073     obtain a copy.
00074
00075 The hexadecimal values shown beneath the polynomials may be in
00076 little-endian or big-endian notation.  The leading term is implicit.
00077 For details, see the respective implementation and the references.
00078 */
00079
00080 /** \ingroup util_crc
00081     Optimized CRC-16 calculation.
00082
00083     Polynomial: x<sup>16</sup> + x<sup>15</sup> + x<sup>2</sup> + 1 (0xa001,
      big-endian)<br>
00084     Initial value: \c 0xffff
00085
00086     This CRC is normally used in disk-drive controllers.
00087
00088     The following is the equivalent functionality written in C.
00089
```

```
00090     \code
00091     static inline uint16_t
00092     _crc16_update (uint16_t crc, uint8_t a)
00093     {
00094         crc ^= a;
00095         for (int i = 0; i < 8; ++i)
00096         {
00097             if (crc & 1)
00098                 crc = (crc » 1) ^ 0xA001;
00099             else
00100                 crc = crc » 1;
00101         }
00102
00103         return crc;
00104     }
00105     \endcode */
00106
00107 static __ATTR_ALWAYS_INLINE__ uint16_t
00108 _crc16_update(uint16_t __crc, uint8_t __data)
00109 {
00110     uint8_t __tmp;
00111     uint16_t __ret;
00112
00113     __asm__ __volatile__ (
00114         "eor %A0,%2" "\n\t"
00115         "mov %1,%A0" "\n\t"
00116         "swap %1" "\n\t"
00117         "eor %1,%A0" "\n\t"
00118         "mov __tmp_reg__,%1" "\n\t"
00119         "lsr %1" "\n\t"
00120         "lsr %1" "\n\t"
00121         "eor %1,__tmp_reg__" "\n\t"
00122         "mov __tmp_reg__,%1" "\n\t"
00123         "lsr %1" "\n\t"
00124         "eor %1,__tmp_reg__" "\n\t"
00125         "andi %1,0x07" "\n\t"
00126         "mov __tmp_reg__,%A0" "\n\t"
00127         "mov %A0,%B0" "\n\t"
00128         "lsr %1" "\n\t"
00129         "ror __tmp_reg__" "\n\t"
00130         "ror %1" "\n\t"
00131         "mov %B0,__tmp_reg__" "\n\t"
00132         "eor %A0,%1" "\n\t"
00133         "lsr __tmp_reg__" "\n\t"
00134         "ror %1" "\n\t"
00135         "eor %B0,__tmp_reg__" "\n\t"
00136         "eor %A0,%1"
00137         : "=r" (__ret), "=d" (__tmp)
00138         : "r" (__data), "0" (__crc)
00139         : "r0"
00140     );
00141     return __ret;
00142 }
00143
00144 /** \ingroup util_crc
00145     Optimized CRC-XMODEM calculation.
00146
00147     Polynomial: x<sup>16</sup> + x<sup>12</sup> + x<sup>5</sup> + 1 (0x1021,
      little-endian)<br>
00148     Initial value: \c 0x0
00149
00150     This is the CRC used by the Xmodem-CRC protocol.
00151
00152     The following is the equivalent functionality written in C.
00153
00154     \code
00155     static inline uint16_t
00156     _crc_xmodem_update (uint16_t crc, uint8_t data)
00157     {
00158         crc = crc ^ ((uint16_t)data « 8);
00159         for (int i = 0; i < 8; i++)
00160         {
00161             if (crc & 0x8000)
```

```
00162                 crc = (crc « 1) ^ 0x1021;
00163             else
00164                 crc «= 1;
00165         }
00166 
00167         return crc;
00168     }
00169     \endcode */
00170 
00171 static __ATTR_ALWAYS_INLINE__ uint16_t
00172 _crc_xmodem_update (uint16_t __crc, uint8_t __data)
00173 {
00174     uint16_t __ret;             /* %B0:%A0 (alias for __crc) */
00175     uint8_t __tmp1;             /* %1 */
00176     uint8_t __tmp2;             /* %2 */
00177                                 /* %3  __data */
00178 
00179     __asm__ __volatile__ (
00180         "eor    %B0,%3"          "\n\t"
00181         "mov    %1,%A0"          "\n\t"
00182         "mov    %2,%B0"          "\n\t"
00183 
00184         "mov    %A0,%B0"         "\n\t"
00185         "swap   %B0"             "\n\t"
00186         "eor    %A0,%B0"         "\n\t"
00187 
00188         "andi   %A0,0xf0"        "\n\t"
00189         "andi   %B0,0x0f"        "\n\t"
00190 
00191         "eor    %1,%A0"          "\n\t"
00192         "eor    %2,%B0"          "\n\t"
00193 
00194         "lsl    %A0"             "\n\t"
00195         "rol    %B0"             "\n\t"
00196 
00197         "eor    %B0,%1"          "\n\t"
00198         "eor    %A0,%2"
00199         : "=d" (__ret), "=r" (__tmp1), "=r" (__tmp2)
00200         : "r" (__data), "0" (__crc)
00201     );
00202     return __ret;
00203 }
00204 
00205 /** \ingroup util_crc
00206     Optimized CRC-CCITT calculation.
00207 
00208     Polynomial: x<sup>16</sup> + x<sup>12</sup> + x<sup>5</sup> + 1 (0x8408,
    big-endian)<br>
00209     Initial value: \c 0xffff
00210 
00211     This is the CRC used by PPP and IrDA.
00212 
00213     See RFC1171 (PPP protocol) and IrDA IrLAP 1.1
00214 
00215     \note Although the CCITT polynomial is the same as that used by the Xmodem
00216     protocol, they are quite different. The difference is in how the bits are
00217     shifted through the algorithm. Xmodem shifts the MSB of the CRC and the
00218     input first, while CCITT shifts the LSB of the CRC and the input first.
00219 
00220     The following is the equivalent functionality written in C.
00221 
00222     \code
00223     static inline uint16_t
00224     _crc_ccitt_update (uint16_t crc, uint8_t data)
00225     {
00226         data ^= lo8 (crc);
00227         data ^= data « 4;
00228 
00229         return ((((uint16_t)data « 8) | hi8 (crc)) ^ (uint8_t)(data » 4)
00230                 ^ ((uint16_t)data « 3));
00231     }
00232     \endcode */
00233 
```

```
00234 static __ATTR_ALWAYS_INLINE__ uint16_t
00235 _crc_ccitt_update (uint16_t __crc, uint8_t __data)
00236 {
00237     uint16_t __ret;
00238
00239     __asm__ __volatile__ (
00240         "eor    %A0,%1"          "\n\t"
00241
00242         "mov    __tmp_reg__,%A0" "\n\t"
00243         "swap   %A0"             "\n\t"
00244         "andi   %A0,0xf0"        "\n\t"
00245         "eor    %A0,__tmp_reg__" "\n\t"
00246
00247         "mov    __tmp_reg__,%B0" "\n\t"
00248
00249         "mov    %B0,%A0"         "\n\t"
00250
00251         "swap   %A0"             "\n\t"
00252         "andi   %A0,0x0f"        "\n\t"
00253         "eor    __tmp_reg__,%A0" "\n\t"
00254
00255         "lsr    %A0"             "\n\t"
00256         "eor    %B0,%A0"         "\n\t"
00257
00258         "eor    %A0,%B0"         "\n\t"
00259         "lsl    %A0"             "\n\t"
00260         "lsl    %A0"             "\n\t"
00261         "lsl    %A0"             "\n\t"
00262         "eor    %A0,__tmp_reg__"
00263
00264         : "=d" (__ret)
00265         : "r" (__data), "0" (__crc)
00266         : "r0"
00267     );
00268     return __ret;
00269 }
00270
00271 /** \ingroup util_crc
00272     Optimized Dallas (now Maxim) iButton 8-bit CRC calculation.
00273
00274     Polynomial: x<sup>8</sup> + x<sup>5</sup> + x<sup>4</sup> + 1 (0x8C, big-endian)<br>
00275     Initial value: \c 0x0
00276
00277     See http://www.maxim-ic.com/appnotes.cfm/appnote_number/27
00278
00279     The following is the equivalent functionality written in C.
00280
00281     \code
00282     static inline uint8_t
00283     _crc_ibutton_update (uint8_t crc, uint8_t data)
00284     {
00285         crc = crc ^ data;
00286         for (uint8_t i = 0; i < 8; i++)
00287         {
00288             if (crc & 0x01)
00289                 crc = (crc » 1) ^ 0x8C;
00290             else
00291                 crc »= 1;
00292         }
00293
00294         return crc;
00295     }
00296     \endcode
00297 */
00298
00299 static __ATTR_ALWAYS_INLINE__ uint8_t
00300 _crc_ibutton_update (uint8_t __crc, uint8_t __data)
00301 {
00302     uint8_t __i, __pattern;
00303     __asm__ __volatile__ (
00304         "eor    %0, %4"     "\n\t"
00305         "ldi    %1, 8"      "\n\t"
00306         "ldi    %2, 0x8C"   "\n"
```

```
00307         "1: lsr %0"         "\n\t"
00308         "brcc   2f"         "\n\t"
00309         "eor    %0, %2"     "\n"
00310         "2: dec %1"         "\n\t"
00311         "brne   1b"
00312         : "=r" (__crc), "=d" (__i), "=d" (__pattern)
00313         : "0" (__crc), "r" (__data));
00314     return __crc;
00315 }
00316
00317 /** \ingroup util_crc
00318     Optimized CRC-8-CCITT calculation.
00319
00320     Polynomial: x<sup>8</sup> + x<sup>2</sup> + x + 1 (0xE0, big-endian)<br>
00321
00322     For use with simple CRC-8<br>
00323     Initial value: 0x0
00324
00325     For use with CRC-8-ROHC<br>
00326     Initial value: 0xff<br>
00327     Reference: http://tools.ietf.org/html/rfc3095#section-5.9.1
00328
00329     For use with CRC-8-ATM/ITU<br>
00330     Initial value: 0xff<br>
00331     Final XOR value: 0x55<br>
00332     Reference: http://www.itu.int/rec/T-REC-I.432.1-199902-I/en
00333
00334     The C equivalent has been originally written by Dave Hylands.
00335     Assembly code is based on \c _crc_ibutton_update optimization.
00336
00337     The following is the equivalent functionality written in C.
00338
00339     \code
00340     static inline uint8_t
00341     _crc8_ccitt_update (uint8_t inCrc, uint8_t inData)
00342     {
00343         uint8_t data = inCrc ^ inData;
00344
00345         for (int i = 0; i < 8; i++)
00346         {
00347             if ((data & 0x80) != 0)
00348             {
00349                 data «= 1;
00350                 data ^= 0x07;
00351             }
00352             else
00353             {
00354                 data «= 1;
00355             }
00356         }
00357         return data;
00358     }
00359     \endcode
00360 */
00361
00362 static __ATTR_ALWAYS_INLINE__ uint8_t
00363 _crc8_ccitt_update(uint8_t __crc, uint8_t __data)
00364 {
00365     uint8_t __i, __pattern;
00366     __asm__ __volatile__ (
00367         "eor    %0, %4"   "\n\t"
00368         "ldi    %1, 8"    "\n\t"
00369         "ldi    %2, 0x07" "\n"
00370         "1: lsl    %0"    "\n\t"
00371         "brcc   2f"       "\n\t"
00372         "eor    %0, %2"   "\n"
00373         "2: dec    %1"    "\n\t"
00374         "brne   1b"
00375         : "=r" (__crc), "=d" (__i), "=d" (__pattern)
00376         : "0" (__crc), "r" (__data));
00377     return __crc;
00378 }
00379
```

```
00380 #endif /* _UTIL_CRC16_H_ */
```

22.77 delay_basic.h File Reference

Functions

- void _delay_loop_1 (uint8_t __count)
- void _delay_loop_2 (uint16_t __count)

22.78 delay_basic.h

Go to the documentation of this file.

```
00001 /* Copyright (c) 2002, Marek Michalkiewicz
00002    Copyright (c) 2007 Joerg Wunsch
00003    All rights reserved.
00004
00005    Redistribution and use in source and binary forms, with or without
00006    modification, are permitted provided that the following conditions are met:
00007
00008    * Redistributions of source code must retain the above copyright
00009      notice, this list of conditions and the following disclaimer.
00010
00011    * Redistributions in binary form must reproduce the above copyright
00012      notice, this list of conditions and the following disclaimer in
00013      the documentation and/or other materials provided with the
00014      distribution.
00015
00016    * Neither the name of the copyright holders nor the names of
00017      contributors may be used to endorse or promote products derived
00018      from this software without specific prior written permission.
00019
00020   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
00021   AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
00022   IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
00023   ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
00024   LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
00025   CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
00026   SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
00027   INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
00028   CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
00029   ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
00030   POSSIBILITY OF SUCH DAMAGE. */
00031
00032 #ifndef _UTIL_DELAY_BASIC_H_
00033 #define _UTIL_DELAY_BASIC_H_ 1
00034
00035 #include <inttypes.h>
00036 #include <bits/attribs.h>
00037
00038 #if !defined(__DOXYGEN__)
00039 static __ATTR_ALWAYS_INLINE__ void _delay_loop_1(uint8_t __count);
00040 static __ATTR_ALWAYS_INLINE__ void _delay_loop_2(uint16_t __count);
00041 #endif
00042
00043 /** \file */
00044 /** \defgroup util_delay_basic <util/delay_basic.h>: Basic busy-wait delay loops
00045     \code
00046     #include <util/delay_basic.h>
00047     \endcode
00048
00049     The functions in this header file implement simple delay loops
00050     that perform a busy-waiting.  They are typically used to
00051     facilitate short delays in the program execution.  They are
00052     implemented as count-down loops with a well-known CPU cycle
00053     count per loop iteration.  As such, no other processing can
```

```
00054     occur simultaneously.  It should be kept in mind that the
00055     functions described here do not disable interrupts.
00056
00057     In general, for long delays, the use of hardware timers is
00058     much preferable, as they free the CPU, and allow for
00059     concurrent processing of other events while the timer is
00060     running.  However, in particular for very short delays, the
00061     overhead of setting up a hardware timer is too much compared
00062     to the overall delay time.
00063
00064     Two inline functions are provided for the actual delay algorithms.
00065 */
00066
00067 /** \ingroup util_delay_basic
00068
00069     Delay loop using an 8-bit counter \c __count, so up to 256
00070     iterations are possible.  (The value 256 would have to be passed
00071     as 0.)  The loop executes three CPU cycles per iteration, not
00072     including the overhead the compiler needs to setup the counter
00073     register.
00074
00075     Thus, delays of up to 768 / f<sub>CPU</sub> microseconds can be achieved,
00076     where f<sub>CPU</sub> denotes the CPU speed in units of 1 MHz.
00077
00078     As an alternative, consider <a
00079
      href="https://gcc.gnu.org/onlinedocs/gcc/AVR-Built-in-Functions.html#index-_005f_005fbuiltin_005favr_005
00080     ><tt>__builtin_avr_delay_cycles</tt></a> which allows to specify the cycle
00081     count and can implement delays of up to 71.5 / f<sub>CPU</sub> seconds.
00082 */
00083 void
00084 _delay_loop_1(uint8_t __count)
00085 {
00086     __asm__ volatile (
00087         "1: dec %0" "\n\t"
00088         "brne 1b"
00089         : "=r" (__count)
00090         : "0" (__count)
00091     );
00092 }
00093
00094 /** \ingroup util_delay_basic
00095
00096     Delay loop using a 16-bit counter \c __count, so up to 65536
00097     iterations are possible.  (The value 65536 would have to be
00098     passed as 0.)  The loop executes four CPU cycles per iteration,
00099     not including the overhead the compiler requires to setup the
00100     counter register pair.
00101
00102     Thus, delays of up to 262.1 / f<sub>CPU</sub> milliseconds can be achieved,
00103     where f<sub>CPU</sub> denotes the CPU speed in units of 1 MHz.
00104
00105     As an alternative, consider <a
00106
      href="https://gcc.gnu.org/onlinedocs/gcc/AVR-Built-in-Functions.html#index-_005f_005fbuiltin_005favr_005
00107     ><tt>__builtin_avr_delay_cycles</tt></a> which allows to specify the cycle
00108     count and can implement delays of up to 71.5 / f<sub>CPU</sub> seconds.
00109  */
00110 void
00111 _delay_loop_2(uint16_t __count)
00112 {
00113 #if defined (__AVR_TINY__)
00114     __asm__ volatile (
00115         "1: subi %A0,1" "\n\t"
00116         "   sbci %B0,0" "\n\t"
00117         "brne 1b"
00118         : "+d" (__count)
00119     );
00120 #else
00121     __asm__ volatile (
00122         "1: sbiw %0,1" "\n\t"
00123         "brne 1b"
00124         : "+w" (__count)
```

```
00125     );
00126 #endif /* AVR_TINY */
00127 }
00128
00129 #endif /* _UTIL_DELAY_BASIC_H_ */
```

22.79 eu_dst.h File Reference

Functions

- int eu_dst (const time_t *timer, int32_t *z)

22.80 eu_dst.h

Go to the documentation of this file.

```
00001 /*
00002  * Copyright (c) 2012 Michael Duane Rice All rights reserved.
00003  *
00004  * Redistribution and use in source and binary forms, with or without
00005  * modification, are permitted provided that the following conditions are
00006  * met:
00007  *
00008  * Redistributions of source code must retain the above copyright notice, this
00009  * list of conditions and the following disclaimer. Redistributions in binary
00010  * form must reproduce the above copyright notice, this list of conditions
00011  * and the following disclaimer in the documentation and/or other materials
00012  * provided with the distribution. Neither the name of the copyright holders
00013  * nor the names of contributors may be used to endorse or promote products
00014  * derived from this software without specific prior written permission.
00015  *
00016  * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
00017  * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
00018  * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
00019  * ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
00020  * LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
00021  * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
00022  * SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
00023  * INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
00024  * CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
00025  * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
00026  * POSSIBILITY OF SUCH DAMAGE. */
00027
00028 #ifndef EU_DST_H
00029 #define EU_DST_H
00030
00031 #ifdef __cplusplus
00032 extern "C" {
00033 #endif
00034
00035 #include <time.h>
00036 #include <stdint.h>
00037
00038 /** \file */
00039 /** \defgroup eu_dst <util/eu_dst.h>: Daylight Saving function for the European Union.
00040
00041     \code #include <util/eu_dst.h> \endcode
00042     Daylight Saving Time for the European Union */
00043
00044 /** \ingroup eu_dst
00045     \fn int eu_dst (const time_t *timer, int32_t *z)
00046     To utilize this function, call \code set_dst(eu_dst); \endcode
00047
00048     Given the time stamp and time zone parameters provided, the Daylight
00049     Saving function must return a value appropriate for the tm structures'
00050     tm_isdst element. That is:
00051
```

```
00052     - \c 0 : If Daylight Saving is not in effect.
00053
00054     - \c -1 : If it cannot be determined if Daylight Saving is in effect.
00055
00056     - A positive integer: Represents the number of seconds a clock is advanced
00057     for Daylight Saving.  This will typically be ONE_HOUR.
00058
00059     Daylight Saving 'rules' are subject to frequent change.  For production
00060     applications it is recommended to write your own DST function, which
00061     uses 'rules' obtained from, and modifiable by, the end user (perhaps
00062     stored in EEPROM).
00063 */
00064 int eu_dst (const time_t *timer, int32_t *z);
00065
00066 #ifdef __cplusplus
00067 }
00068 #endif
00069
00070 #endif
```

22.81 parity.h File Reference

Functions

- static uint8_t parity_even_bit (uint8_t __val)

22.82 parity.h

Go to the documentation of this file.

```
00001 /* Copyright (c) 2002, Marek Michalkiewicz
00002    Copyright (c) 2004,2005,2007 Joerg Wunsch
00003    All rights reserved.
00004
00005    Redistribution and use in source and binary forms, with or without
00006    modification, are permitted provided that the following conditions are met:
00007
00008    * Redistributions of source code must retain the above copyright
00009      notice, this list of conditions and the following disclaimer.
00010
00011    * Redistributions in binary form must reproduce the above copyright
00012      notice, this list of conditions and the following disclaimer in
00013      the documentation and/or other materials provided with the
00014      distribution.
00015
00016    * Neither the name of the copyright holders nor the names of
00017      contributors may be used to endorse or promote products derived
00018      from this software without specific prior written permission.
00019
00020   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
00021   AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
00022   IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
00023   ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
00024   LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
00025   CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
00026   SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
00027   INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
00028   CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
00029   ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
00030   POSSIBILITY OF SUCH DAMAGE. */
00031
00032 #ifndef _UTIL_PARITY_H_
00033 #define _UTIL_PARITY_H_
00034
00035 #include <stdint.h>
00036 #include <bits/attribs.h>
00037
```

```
00038 /** \file */
00039 /** \defgroup util_parity <util/parity.h>: Parity bit generation
00040     \code #include <util/parity.h> \endcode
00041
00042     This header file contains optimized assembler code to calculate
00043     the parity bit for a byte.
00044 */
00045 /** \fn uint8_t parity_even_bit (uint8_t val);
00046     \ingroup util_parity
00047     \returns 1 if \c val has an odd number of bits set, and 0 otherwise. */
00048
00049 static __ATTR_ALWAYS_INLINE__
00050 uint8_t parity_even_bit (uint8_t __val)
00051 {
00052   if (__builtin_constant_p (__builtin_parity (__val)))
00053     return (uint8_t) __builtin_parity (__val);
00054
00055   __asm (/* parity is in [0..7] */
00056          "mov  __tmp_reg__, %0"  "\n\t"
00057          "swap __tmp_reg__"      "\n\t"
00058          "eor  %0, __tmp_reg__"  "\n\t"
00059          /* parity is in [0..3] */
00060          "subi %0, -4"           "\n\t"
00061          "andi %0, -5"           "\n\t"
00062          "subi %0, -6"           "\n\t"
00063          /* parity is in [0,3] */
00064          "sbrc %0, 3"            "\n\t"
00065          "inc  %0"
00066          /* parity is in [0] */
00067          : "+d" (__val));
00068
00069   return __val & 1;
00070 }
00071
00072 #endif /* _UTIL_PARITY_H_ */
```

22.83 ram-usage.h File Reference

Functions

- uint16_t _get_ram_unused (void)

Variables

Symbols

- __heap_start
- __ram_color_end
- __ram_color_value

22.84 ram-usage.h

Go to the documentation of this file.

```
00001 /* Copyright (c) 2025  Georg-Johann Lay
00002    All rights reserved.
00003
00004    Redistribution and use in source and binary forms, with or without
00005    modification, are permitted provided that the following conditions are met:
00006
00007    * Redistributions of source code must retain the above copyright
00008      notice, this list of conditions and the following disclaimer.
00009
```

```
00010     * Redistributions in binary form must reproduce the above copyright
00011       notice, this list of conditions and the following disclaimer in
00012       the documentation and/or other materials provided with the
00013       distribution.
00014
00015     * Neither the name of the copyright holders nor the names of
00016       contributors may be used to endorse or promote products derived
00017       from this software without specific prior written permission.
00018
00019   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
00020   AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
00021   IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
00022   ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
00023   LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
00024   CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
00025   SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
00026   INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
00027   CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
00028   ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
00029   POSSIBILITY OF SUCH DAMAGE. */
00030
00031 #ifndef _UTIL_RAM_USAGE_H_
00032 #define _UTIL_RAM_USAGE_H_
00033
00034 #include <stdint.h>
00035 #include <bits/attribs.h>
00036
00037 #ifdef __cplusplus
00038 extern "C" {
00039 #endif
00040
00041 /** \file */
00042 /** \defgroup util_ram_usage <util/ram-usage.h>: Determine dynamic RAM usage
00043     \code
00044     #include <util/ram-usage.h>
00045     \endcode
00046
00047     This header provides a single function, _get_ram_unused(), that can
00048     be used during development to get a rough estimate of how much RAM might
00049     be used by an application.  This works as follows:
00050
00051     -# The startup code paints the RAM location with a specific value.  This
00052     happens in \ref sec_dot_init ".init3", and only when _get_ram_unused()
00053     is actually used.  The coloring extends from #__heap_start (which is
00054     the RAM location right after static storage) up to and
00055     including \c RAMEND as defined in <avr/io.h>.
00056
00057     -# The application calls _get_ram_unused() and determines how much
00058     of the colored bytes are still intact.  This can be used to calculate
00059     a <b>lower bound</b> of how much RAM is used by the application.
00060
00061     \since AVR-LibC v2.3
00062 */
00063
00064 /** \ingroup util_ram_usage
00065
00066     Determines how much of the initial RAM coloring is still intact.
00067     It can be used to get a <b>lower bound</b> of how much RAM might be
00068     used by an application.
00069
00070     The function is written in such a way that it has a small register foot
00071     print, so that it is not a big issue to call it in an #ISR.
00072     Though that's not required for the intended purpose:
00073     _get_ram_unused() can simply be called after some time has elapsed
00074     and enough ISRs and other functions have been invoked.
00075
00076     \return The value returned by _get_ram_unused() is an <em>upper bound</em>
00077     for how much bytes of RAM are still
00078     <em>unused at the time of invocation</em>.<br>
00079     The return value will never increase with time (except for very rare
00080     occasions where #__ram_color_value is written to the top of the stack).
00081
00082     \par Limitations
```

```
00083     - The start of the coloring is hard coded as #__heap_start, which
00084     is the beginning of the RAM area after static storage.
00085     - The algorithm assumes that the stack is located <em>after
00086     static storage</em> and grows downwards towards #__heap_start.
00087     - The current implementation is <b>not compatible with malloc</b> et al.
00088     (alloca() is no problem though, since it allocates on the stack.)
00089 */
00090 #ifdef __DOXYGEN__
00091 extern inline uint16_t _get_ram_unused (void);
00092 #else
00093 extern __ATTR_ALWAYS_INLINE__ __ATTR_GNU_INLINE__
00094 uint16_t _get_ram_unused (void)
00095 {
00096   register uint16_t __r24 __asm("r24");
00097   __asm ("%~call _get_ram_unused"
00098          : "=r" (__r24) :: "r30", "r31");
00099   return __r24;
00100 }
00101 #endif /* !DOXYGEN */
00102
00103 #ifdef __DOXYGEN__
00104
00105 /** \name Symbols */
00106
00107 /** \ingroup util_ram_usage
00108     A symbol defined in the default linker script.
00109     It points one byte past static storage (\ref sec_dot_data ".data",
00110     \ref sec_dot_bss ".bss", \ref sec_dot_noinit ".noinit"). */
00111 extern __heap_start;
00112
00113 /** \ingroup util_ram_usage
00114     A weak symbol that defaults to  <tt>RAMEND + 1</tt>.
00115     It points one byte past the last location that is colored by
00116     the startup code.
00117     It can be adjusted by say<br>
00118 \code
00119     avr-gcc ... -Wl,-defsym,__ram_color_end=<value>
00120 \endcode
00121     in the link command, or by means of a global inline assembly statement
00122     like:
00123 \code
00124     __asm (".global __ram_color_end\n"
00125            "__ram_color_end = <value>");
00126 \endcode */
00127 __ram_color_end;
00128
00129 /** \ingroup util_ram_usage
00130     A weak symbol that defaults to 0xaa.
00131     It represents the "color" that's being used by the startup code to
00132     paint the RAM, and the value that _get_ram_unused() will check against.
00133     It can be adjusted by say
00134 \code
00135     avr-gcc ... -Wl,-defsym,__ram_color_value=<value>
00136 \endcode
00137     in the link command. */
00138 __ram_color_value;
00139
00140 #endif /* DOXYGEN */
00141
00142 #ifdef __cplusplus
00143 } // extern "C"
00144 #endif
00145
00146 #endif /* _UTIL_RAM_USAGE_H_ */
```

22.85 setbaud.h File Reference

Macros

- #define BAUD_TOL 2

- #define UBRR_VALUE
- #define UBRRL_VALUE
- #define UBRRH_VALUE
- #define USE_2X 0

22.86 setbaud.h

Go to the documentation of this file.

```
00001 /* Copyright (c) 2007  Cliff Lawson
00002    Copyright (c) 2007  Carlos Lamas
00003    Copyright (c) 2025  Ian Gregg
00004    All rights reserved.
00005
00006    Redistribution and use in source and binary forms, with or without
00007    modification, are permitted provided that the following conditions are met:
00008
00009    * Redistributions of source code must retain the above copyright
00010      notice, this list of conditions and the following disclaimer.
00011
00012    * Redistributions in binary form must reproduce the above copyright
00013      notice, this list of conditions and the following disclaimer in
00014      the documentation and/or other materials provided with the
00015      distribution.
00016
00017    * Neither the name of the copyright holders nor the names of
00018      contributors may be used to endorse or promote products derived
00019      from this software without specific prior written permission.
00020
00021   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
00022   AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
00023   IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
00024   ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
00025   LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
00026   CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
00027   SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
00028   INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
00029   CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
00030   ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
00031   POSSIBILITY OF SUCH DAMAGE. */
00032
00033 /**
00034    \file
00035 */
00036
00037 /**
00038    \defgroup util_setbaud <util/setbaud.h>: Helper macros for baud rate calculations
00039    \code
00040    #define F_CPU 11059200
00041    #define BAUD 38400
00042    #include <util/setbaud.h>
00043    \endcode
00044
00045    This header file requires that on entry values are already defined
00046    for F_CPU and BAUD.  In addition, the macro BAUD_TOL will define
00047    the baud rate tolerance (in percent) that is acceptable during
00048    the calculations.  The value of BAUD_TOL will default to 2 %.
00049
00050    This header file defines macros suitable to setup the UART baud
00051    rate prescaler registers of an AVR.  All calculations are done
00052    using the C preprocessor.  Including this header file causes no
00053    other side effects so it is possible to include this file more than
00054    once (supposedly, with different values for the BAUD parameter),
00055    possibly even within the same function.
00056
00057    Assuming that the requested BAUD is valid for the given F_CPU then
00058    the macro UBRR_VALUE is set to the required prescaler value.  Two
00059    additional macros are provided for the low and high bytes of the
00060    prescaler, respectively: UBRRL_VALUE is set to the lower byte of
```

```
00061    the UBRR_VALUE and UBRRH_VALUE is set to the upper byte.  An
00062    additional macro USE_2X will be defined.  Its value is set to 1 if
00063    the desired BAUD rate within the given tolerance could only be
00064    achieved by setting the U2X bit in the UART configuration.  It will
00065    be defined to 0 if U2X is not needed.
00066
00067    Example usage:
00068
00069    \code
00070    #include <avr/io.h>
00071
00072    #define F_CPU 4000000
00073
00074    static void
00075    uart_9600(void)
00076    {
00077    #define BAUD 9600
00078    #include <util/setbaud.h>
00079        UBRRH = UBRRH_VALUE;
00080        UBRRL = UBRRL_VALUE;
00081    #if USE_2X
00082        UCSRA |= (1 « U2X);
00083    #else
00084        UCSRA &= ~(1 « U2X);
00085    #endif
00086    }
00087
00088    static void
00089    uart_38400(void)
00090    {
00091    #undef BAUD  // avoid compiler warning
00092    #define BAUD 38400
00093    #include <util/setbaud.h>
00094        UBRRH = UBRRH_VALUE;
00095        UBRRL = UBRRL_VALUE;
00096    #if USE_2X
00097        UCSRA |= (1 « U2X);
00098    #else
00099        UCSRA &= ~(1 « U2X);
00100    #endif
00101    }
00102    \endcode
00103
00104    In this example, two functions are defined to setup the UART
00105    to run at 9600 Bd, and 38400 Bd, respectively.  Using a CPU
00106    clock of 4 MHz, 9600 Bd can be achieved with an acceptable
00107    tolerance without setting U2X (prescaler 25), while 38400 Bd
00108    require U2X to be set (prescaler 12).
00109 */
00110
00111 #ifndef F_CPU
00112 #  error "setbaud.h requires F_CPU to be defined"
00113 #endif
00114
00115 #ifndef BAUD
00116 #  error "setbaud.h requires BAUD to be defined"
00117 #endif
00118
00119 #if !(F_CPU)
00120 #  error "F_CPU must be a constant value"
00121 #endif
00122
00123 #if !(BAUD)
00124 #  error "BAUD must be a constant value"
00125 #endif
00126
00127 #if defined(__DOXYGEN__)
00128 /**
00129    \def BAUD_TOL
00130    \ingroup util_setbaud
00131
00132    Input and output macro for <util/setbaud.h>
00133
```

```
00134    Define the acceptable baud rate tolerance in percent.  If not set
00135    on entry, it will be set to its default value of 2.
00136 */
00137 #define BAUD_TOL 2
00138
00139 /**
00140    \def UBRR_VALUE
00141    \ingroup util_setbaud
00142
00143    Output macro from <util/setbaud.h>
00144
00145    Contains the calculated baud rate prescaler value for the UBRR
00146    register.
00147 */
00148 #define UBRR_VALUE
00149
00150 /**
00151    \def UBRRL_VALUE
00152    \ingroup util_setbaud
00153
00154    Output macro from <util/setbaud.h>
00155
00156    Contains the lower byte of the calculated prescaler value
00157    (UBRR_VALUE).
00158 */
00159 #define UBRRL_VALUE
00160
00161 /**
00162    \def UBRRH_VALUE
00163    \ingroup util_setbaud
00164
00165    Output macro from <util/setbaud.h>
00166
00167    Contains the upper byte of the calculated prescaler value
00168    (UBRR_VALUE).
00169 */
00170 #define UBRRH_VALUE
00171
00172 /**
00173    \def USE_2X
00174    \ingroup util_setbaud
00175
00176    Output macro from <util/setbaud.h>
00177
00178    Contains the value 1 if the desired baud rate tolerance could only
00179    be achieved by setting the U2X bit in the UART configuration.
00180    Contains 0 otherwise.
00181 */
00182 #define USE_2X 0
00183
00184 #else /* !__DOXYGEN__ */
00185
00186 #undef USE_2X
00187
00188 /* Baud rate tolerance is 2 % unless previously defined */
00189 #ifndef BAUD_TOL
00190 #  define BAUD_TOL 2
00191 #endif
00192
00193 #ifdef __ASSEMBLER__
00194 #define UBRR_VALUE ((((F_CPU) - 16 * (BAUD)) * 100 / (16 * (BAUD)) + 50) / 100)
00195 #else
00196 #define UBRR_VALUE ((((F_CPU) - 16UL * (BAUD)) * 100UL / (16UL * (BAUD)) + 50UL) /
      100UL)
00197 #endif
00198
00199 #if 100 * (F_CPU) > \
00200   (16 * ((UBRR_VALUE) + 1) * (100 + (BAUD_TOL)) * (BAUD))
00201 #  define USE_2X 1
00202 #elif 100 * (F_CPU) < \
00203   (16 * ((UBRR_VALUE) + 1) * (100 - (BAUD_TOL)) * (BAUD))
00204 #  define USE_2X 1
00205 #else
```

```
00206 #  define USE_2X 0
00207 #endif
00208
00209 #if USE_2X
00210 /* U2X required, recalculate */
00211 #undef UBRR_VALUE
00212
00213 #ifdef __ASSEMBLER__
00214 #define UBRR_VALUE ((((F_CPU) - 8 * (BAUD)) * 100 / (8 * (BAUD)) + 50) / 100)
00215 #else
00216 #define UBRR_VALUE ((((F_CPU) - 8UL * (BAUD)) * 100UL / (8UL * (BAUD)) + 50UL) / 100UL)
00217 #endif
00218
00219 #if 100 * (F_CPU) > \
00220   (8 * ((UBRR_VALUE) + 1)) * (100 + (BAUD_TOL)) * (BAUD))
00221 #  warning "Baud rate achieved is higher than allowed"
00222 #endif
00223
00224 #if 100 * (F_CPU) < \
00225   (8 * ((UBRR_VALUE) + 1)) * (100 - (BAUD_TOL)) * (BAUD))
00226 #  warning "Baud rate achieved is lower than allowed"
00227 #endif
00228
00229 #endif /* USE_U2X */
00230
00231 #ifdef UBRR_VALUE
00232    /* Check for overflow */
00233 #  if UBRR_VALUE >= (1 « 12)
00234 #    warning "UBRR value overflow"
00235 #  endif
00236
00237 #  define UBRRL_VALUE (UBRR_VALUE & 0xff)
00238 #  define UBRRH_VALUE (UBRR_VALUE » 8)
00239 #endif
00240
00241 #endif /* __DOXYGEN__ */
00242 /* end of util/setbaud.h */
```

22.87 compat/twi.h

```
00001 /* Copyright (c) 2005 Joerg Wunsch
00002    All rights reserved.
00003
00004    Redistribution and use in source and binary forms, with or without
00005    modification, are permitted provided that the following conditions are met:
00006
00007    * Redistributions of source code must retain the above copyright
00008      notice, this list of conditions and the following disclaimer.
00009
00010    * Redistributions in binary form must reproduce the above copyright
00011      notice, this list of conditions and the following disclaimer in
00012      the documentation and/or other materials provided with the
00013      distribution.
00014
00015    * Neither the name of the copyright holders nor the names of
00016      contributors may be used to endorse or promote products derived
00017      from this software without specific prior written permission.
00018
00019   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
00020   AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
00021   IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
00022   ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
00023   LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
00024   CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
00025   SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
00026   INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
00027   CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
00028   ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
00029   POSSIBILITY OF SUCH DAMAGE. */
00030
00031 #ifndef _COMPAT_TWI_H_
```

```
00032 #define _COMPAT_TWI_H_
00033
00034 #include <util/twi.h>
00035
00036 #endif /* _COMPAT_TWI_H_ */
```

22.88 twi.h File Reference

Macros

TWSR values

Mnemonics:
TW_MT_xxx - master transmitter
TW_MR_xxx - master receiver
TW_ST_xxx - slave transmitter
TW_SR_xxx - slave receiver

- #define TW_START 0x08
- #define TW_REP_START 0x10
- #define TW_MT_SLA_ACK 0x18
- #define TW_MT_SLA_NACK 0x20
- #define TW_MT_DATA_ACK 0x28
- #define TW_MT_DATA_NACK 0x30
- #define TW_MT_ARB_LOST 0x38
- #define TW_MR_ARB_LOST 0x38
- #define TW_MR_SLA_ACK 0x40
- #define TW_MR_SLA_NACK 0x48
- #define TW_MR_DATA_ACK 0x50
- #define TW_MR_DATA_NACK 0x58
- #define TW_ST_SLA_ACK 0xA8
- #define TW_ST_ARB_LOST_SLA_ACK 0xB0
- #define TW_ST_DATA_ACK 0xB8
- #define TW_ST_DATA_NACK 0xC0
- #define TW_ST_LAST_DATA 0xC8
- #define TW_SR_SLA_ACK 0x60
- #define TW_SR_ARB_LOST_SLA_ACK 0x68
- #define TW_SR_GCALL_ACK 0x70
- #define TW_SR_ARB_LOST_GCALL_ACK 0x78
- #define TW_SR_DATA_ACK 0x80
- #define TW_SR_DATA_NACK 0x88
- #define TW_SR_GCALL_DATA_ACK 0x90
- #define TW_SR_GCALL_DATA_NACK 0x98
- #define TW_SR_STOP 0xA0
- #define TW_NO_INFO 0xF8
- #define TW_BUS_ERROR 0x00
- #define TW_STATUS_MASK
- #define TW_STATUS (TWSR & TW_STATUS_MASK)

R/∼W bit in SLA+R/W address field.

- #define TW_READ 1
- #define TW_WRITE 0

22.89 util/twi.h

Go to the documentation of this file.

```
00001 /* Copyright (c) 2002, Marek Michalkiewicz
00002    Copyright (c) 2005, 2007 Joerg Wunsch
00003    All rights reserved.
00004
00005    Redistribution and use in source and binary forms, with or without
00006    modification, are permitted provided that the following conditions are met:
00007
00008    * Redistributions of source code must retain the above copyright
00009      notice, this list of conditions and the following disclaimer.
00010
00011    * Redistributions in binary form must reproduce the above copyright
00012      notice, this list of conditions and the following disclaimer in
00013      the documentation and/or other materials provided with the
00014      distribution.
00015
00016    * Neither the name of the copyright holders nor the names of
00017      contributors may be used to endorse or promote products derived
00018      from this software without specific prior written permission.
00019
00020   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
00021   AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
00022   IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
00023   ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
00024   LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
00025   CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
00026   SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
00027   INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
00028   CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
00029   ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
00030   POSSIBILITY OF SUCH DAMAGE. */
00031
00032 #ifndef _UTIL_TWI_H_
00033 #define _UTIL_TWI_H_ 1
00034
00035 #include <avr/io.h>
00036
00037 /** \file */
00038 /** \defgroup util_twi <util/twi.h>: TWI bit mask definitions
00039     \code #include <util/twi.h> \endcode
00040
00041     This header file contains bit mask definitions for use with
00042     the AVR TWI interface.
00043 */
00044 /** \name TWSR values
00045
00046   Mnemonics:
00047   <br>TW_MT_xxx - master transmitter
00048   <br>TW_MR_xxx - master receiver
00049   <br>TW_ST_xxx - slave transmitter
00050   <br>TW_SR_xxx - slave receiver
00051   */
00052
00053 /**@{*/
00054 /* Master */
00055 /** \ingroup util_twi
00056     \def TW_START
00057     start condition transmitted */
00058 #define TW_START         0x08
00059
00060 /** \ingroup util_twi
00061     \def TW_REP_START
00062     repeated start condition transmitted */
00063 #define TW_REP_START         0x10
00064
00065 /* Master Transmitter */
00066 /** \ingroup util_twi
00067     \def TW_MT_SLA_ACK
00068     SLA+W transmitted, ACK received */
00069 #define TW_MT_SLA_ACK        0x18
```

```
00070
00071 /** \ingroup util_twi
00072     \def TW_MT_SLA_NACK
00073     SLA+W transmitted, NACK received */
00074 #define TW_MT_SLA_NACK      0x20
00075
00076 /** \ingroup util_twi
00077     \def TW_MT_DATA_ACK
00078     data transmitted, ACK received */
00079 #define TW_MT_DATA_ACK      0x28
00080
00081 /** \ingroup util_twi
00082     \def TW_MT_DATA_NACK
00083     data transmitted, NACK received */
00084 #define TW_MT_DATA_NACK     0x30
00085
00086 /** \ingroup util_twi
00087     \def TW_MT_ARB_LOST
00088     arbitration lost in SLA+W or data */
00089 #define TW_MT_ARB_LOST      0x38
00090
00091 /* Master Receiver */
00092 /** \ingroup util_twi
00093     \def TW_MR_ARB_LOST
00094     arbitration lost in SLA+R or NACK */
00095 #define TW_MR_ARB_LOST      0x38
00096
00097 /** \ingroup util_twi
00098     \def TW_MR_SLA_ACK
00099     SLA+R transmitted, ACK received */
00100 #define TW_MR_SLA_ACK       0x40
00101
00102 /** \ingroup util_twi
00103     \def TW_MR_SLA_NACK
00104     SLA+R transmitted, NACK received */
00105 #define TW_MR_SLA_NACK      0x48
00106
00107 /** \ingroup util_twi
00108     \def TW_MR_DATA_ACK
00109     data received, ACK returned */
00110 #define TW_MR_DATA_ACK      0x50
00111
00112 /** \ingroup util_twi
00113     \def TW_MR_DATA_NACK
00114     data received, NACK returned */
00115 #define TW_MR_DATA_NACK     0x58
00116
00117 /* Slave Transmitter */
00118 /** \ingroup util_twi
00119     \def TW_ST_SLA_ACK
00120     SLA+R received, ACK returned */
00121 #define TW_ST_SLA_ACK       0xA8
00122
00123 /** \ingroup util_twi
00124     \def TW_ST_ARB_LOST_SLA_ACK
00125     arbitration lost in SLA+RW, SLA+R received, ACK returned */
00126 #define TW_ST_ARB_LOST_SLA_ACK  0xB0
00127
00128 /** \ingroup util_twi
00129     \def TW_ST_DATA_ACK
00130     data transmitted, ACK received */
00131 #define TW_ST_DATA_ACK      0xB8
00132
00133 /** \ingroup util_twi
00134     \def TW_ST_DATA_NACK
00135     data transmitted, NACK received */
00136 #define TW_ST_DATA_NACK     0xC0
00137
00138 /** \ingroup util_twi
00139     \def TW_ST_LAST_DATA
00140     last data byte transmitted, ACK received */
00141 #define TW_ST_LAST_DATA     0xC8
00142
```

```
00143 /* Slave Receiver */
00144 /** \ingroup util_twi
00145     \def TW_SR_SLA_ACK
00146     SLA+W received, ACK returned */
00147 #define TW_SR_SLA_ACK       0x60
00148
00149 /** \ingroup util_twi
00150     \def TW_SR_ARB_LOST_SLA_ACK
00151     arbitration lost in SLA+RW, SLA+W received, ACK returned */
00152 #define TW_SR_ARB_LOST_SLA_ACK  0x68
00153
00154 /** \ingroup util_twi
00155     \def TW_SR_GCALL_ACK
00156     general call received, ACK returned */
00157 #define TW_SR_GCALL_ACK     0x70
00158
00159 /** \ingroup util_twi
00160     \def TW_SR_ARB_LOST_GCALL_ACK
00161     arbitration lost in SLA+RW, general call received, ACK returned */
00162 #define TW_SR_ARB_LOST_GCALL_ACK 0x78
00163
00164 /** \ingroup util_twi
00165     \def TW_SR_DATA_ACK
00166     data received, ACK returned */
00167 #define TW_SR_DATA_ACK      0x80
00168
00169 /** \ingroup util_twi
00170     \def TW_SR_DATA_NACK
00171     data received, NACK returned */
00172 #define TW_SR_DATA_NACK     0x88
00173
00174 /** \ingroup util_twi
00175     \def TW_SR_GCALL_DATA_ACK
00176     general call data received, ACK returned */
00177 #define TW_SR_GCALL_DATA_ACK    0x90
00178
00179 /** \ingroup util_twi
00180     \def TW_SR_GCALL_DATA_NACK
00181     general call data received, NACK returned */
00182 #define TW_SR_GCALL_DATA_NACK   0x98
00183
00184 /** \ingroup util_twi
00185     \def TW_SR_STOP
00186     stop or repeated start condition received while selected */
00187 #define TW_SR_STOP      0xA0
00188
00189 /* Misc */
00190 /** \ingroup util_twi
00191     \def TW_NO_INFO
00192     no state information available */
00193 #define TW_NO_INFO      0xF8
00194
00195 /** \ingroup util_twi
00196     \def TW_BUS_ERROR
00197     illegal start or stop condition */
00198 #define TW_BUS_ERROR        0x00
00199
00200
00201 /**
00202  * \ingroup util_twi
00203  * \def TW_STATUS_MASK
00204  * The lower 3 bits of TWSR are reserved on the ATmega163.
00205  * The 2 LSB carry the prescaler bits on the newer ATmegas.
00206  */
00207 #define TW_STATUS_MASK      (_BV(TWS7)|_BV(TWS6)|_BV(TWS5)|_BV(TWS4)|\
00208                  _BV(TWS3))
00209 /**
00210  * \ingroup util_twi
00211  * \def TW_STATUS
00212  *
00213  * TWSR, masked by TW_STATUS_MASK
00214  */
00215 #define TW_STATUS       (TWSR & TW_STATUS_MASK)
```

```
00216 /**@}*/
00217
00218 /**
00219  * \name R/~W bit in SLA+R/W address field.
00220  */
00221
00222 /**@{*/
00223 /** \ingroup util_twi
00224     \def TW_READ
00225     SLA+R address */
00226 #define TW_READ     1
00227
00228 /** \ingroup util_twi
00229     \def TW_WRITE
00230     SLA+W address */
00231 #define TW_WRITE    0
00232 /**@}*/
00233
00234 #endif  /* _UTIL_TWI_H_ */
```

22.90 usa_dst.h File Reference

Functions

- int usa_dst (const time_t *timer, int32_t *z)

22.91 usa_dst.h

Go to the documentation of this file.

```
00001 /*
00002  * Copyright (c) 2012 Michael Duane Rice All rights reserved.
00003  *
00004  * Redistribution and use in source and binary forms, with or without
00005  * modification, are permitted provided that the following conditions are
00006  * met:
00007  *
00008  * Redistributions of source code must retain the above copyright notice, this
00009  * list of conditions and the following disclaimer. Redistributions in binary
00010  * form must reproduce the above copyright notice, this list of conditions
00011  * and the following disclaimer in the documentation and/or other materials
00012  * provided with the distribution. Neither the name of the copyright holders
00013  * nor the names of contributors may be used to endorse or promote products
00014  * derived from this software without specific prior written permission.
00015  *
00016  * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
00017  * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
00018  * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
00019  * ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
00020  * LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
00021  * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
00022  * SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
00023  * INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
00024  * CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
00025  * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
00026  * POSSIBILITY OF SUCH DAMAGE. */
00027
00028 #ifndef USA_DST_H
00029 #define USA_DST_H
00030
00031 #ifdef __cplusplus
00032 extern "C" {
00033 #endif
00034
00035 #include <time.h>
00036 #include <stdint.h>
00037
```

```
00038 /** \file */
00039 /** \defgroup usa_dst <util/usa_dst.h>: Daylight Saving function for the USA.
00040     \code #include <util/usa_dst.h> \endcode
00041     Daylight Saving function for the USA. */
00042
00043 /** \ingroup usa_dst
00044     \fn int usa_dst (const time_t *timer, int32_t *z)
00045     To utilize this function, call
00046     \code set_dst(usa_dst); \endcode
00047
00048     Given the time stamp and time zone parameters provided, the Daylight
00049     Saving function must return a value appropriate for the tm structures'
00050     tm_isdst element. That is:
00051
00052     - \c 0 : If Daylight Saving is not in effect.
00053
00054     - \c -1 : If it cannot be determined if Daylight Saving is in effect.
00055
00056     - A positive integer : Represents the number of seconds a clock is
00057     advanced for Daylight Saving.  This will typically be ONE_HOUR.
00058
00059     Daylight Saving 'rules' are subject to frequent change. For production
00060     applications it is recommended to write your own DST function, which
00061     uses 'rules' obtained from, and modifiable by, the end user
00062     (perhaps stored in EEPROM).
00063 */
00064 int usa_dst (const time_t *timer, int32_t *z);
00065
00066 #ifdef __cplusplus
00067 }
00068 #endif
00069
00070 #endif
```

22.92 eedef.h

```
00001 /* Copyright (c) 2009  Dmitry Xmelkov
00002    All rights reserved.
00003
00004    Redistribution and use in source and binary forms, with or without
00005    modification, are permitted provided that the following conditions are met:
00006
00007    * Redistributions of source code must retain the above copyright
00008      notice, this list of conditions and the following disclaimer.
00009    * Redistributions in binary form must reproduce the above copyright
00010      notice, this list of conditions and the following disclaimer in
00011      the documentation and/or other materials provided with the
00012      distribution.
00013    * Neither the name of the copyright holders nor the names of
00014      contributors may be used to endorse or promote products derived
00015      from this software without specific prior written permission.
00016
00017    THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
00018    AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
00019    IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
00020    ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
00021    LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
00022    CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
00023    SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
00024    INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
00025    CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
00026    ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
00027    POSSIBILITY OF SUCH DAMAGE.  */
00028
00029 #ifndef _EEDEF_H_
00030 #define _EEDEF_H_  1
00031
00032 #ifndef __DOXYGEN__
00033
00034 #include <bits/devinfo.h>
00035
```

```
00036 /* EEPROM address arg for a set of byte/word/dword functions and for
00037    the internal eeprom_read_blraw().    */
00038 #define addr_lo r24
00039 #define addr_hi r25
00040
00041 /* Number of bytes arg for all block read/write functions, include
00042    internal.    */
00043 #define n_lo    r20
00044 #define n_hi    r21
00045
00046 #if __AVR_XMEGA__
00047
00048 # define NVM_BASE   NVM_ADDR0
00049
00050 #if defined(NVMCTRL_CTRLA)
00051 #  undef NVM_BASE
00052 #  define NVM_BASE       NVMCTRL_CTRLA
00053
00054 #  define NVM_ADDR0      NVMCTRL_ADDR0
00055 #  define NVM_ADDR1      NVMCTRL_ADDR1
00056 #  define NVM_DATA0      NVMCTRL_DATA0
00057 #  define NVM_DATA1      NVMCTRL_DATA1
00058 #  define NVM_NVMBUSY_bp NVMCTRL_EEBUSY_bp
00059 #  define NVM_STATUS     NVMCTRL_STATUS
00060 #  define NVM_CTRLA      NVMCTRL_CTRLA
00061 #  define NVM_CTRLB      NVMCTRL_CTRLB
00062 #  ifndef CCP_SPM_gc
00063 #    define CCP_SPM_gc   (0x9D)
00064 #  endif
00065 #  ifndef NVMCTRL_CMD_PAGEERASEWRITE_gc
00066 #    if NVMCTRL_CMD_gm == 0x7F
00067 #      if defined (__AVR_Ex__) || defined (__AVR_Lx__)
00068           /* AVR-Ex family
00069            * value of NVMCTRL_CMD_enum.NVMCTRL_CMD_EEPERW_gc */
00070 #        define NVMCTRL_CMD_PAGEERASEWRITE_gc (0x15<<0)
00071 #      elif defined (__AVR_Dx__) || defined (__AVR_Sx__)
00072           /* AVR-Dx family
00073            * value of NVMCTRL_CMD_enum.NVMCTRL_CMD_EEERWR_gc */
00074 #        define NVMCTRL_CMD_PAGEERASEWRITE_gc (0x13<<0)
00075 #      else
00076           /* To support a new device, define NVMCTRL CMD_PAGEERASEWRITE_gc
00077            * with the value of "Erase and Write EEPROM Page" comand code
00078            * for - Persistent Memory Controller (NVMCTRL).  */
00079 #        error "Not supported devices"
00080 #      endif
00081 #    else
00082         /* the rest of the AVR devices with NVMCTRL_CTRLA (0x07)
00083          * value of NVMCTRL_CMD_enum.NVMCTRL_CMD_PAGEERASEWRITE_gc */
00084 #      define NVMCTRL_CMD_PAGEERASEWRITE_gc 3
00085 #    endif
00086 #  endif /* NVMCTRL_CMD_PAGEERASEWRITE_gc */
00087 #endif /* defined(NVMCTRL_CTRLA) */
00088 #else
00089
00090 # if    !defined (EECR) && defined (DEECR)  /* AT86RF401    */
00091 #  define EECR  DEECR
00092 #  define EEARL DEEAR
00093 #  define EEDR  DEEDR
00094 # endif
00095
00096 # if    !defined (EERE) && defined (EER)    /* AT86RF401    */
00097 #  define EERE  EER
00098 # endif
00099
00100 # if    !defined (EEWE) && defined (EEPE)   /* A part of Mega and Tiny */
00101 #  define EEWE  EEPE
00102 # endif
00103 # if    !defined (EEWE) && defined (EEL)    /* AT86RF401    */
00104 #  define EEWE  EEL
00105 # endif
00106
00107 # if    !defined (EEMWE) && defined (EEMPE) /* A part of Mega and Tiny */
00108 #  define EEMWE EEMPE
```

```
00109 # endif
00110 # if    !defined (EEMWE) && defined (EEU)   /* AT86RF401    */
00111 #  define EEMWE EEU
00112 # endif
00113
00114 # if    !_SFR_IO_REG_P (EECR)   \
00115      || !_SFR_IO_REG_P (EEDR)   \
00116      || !_SFR_IO_REG_P (EEARL)  \
00117      || (defined (EEARH) && !_SFR_IO_REG_P (EEARH))
00118 #  error
00119 # endif
00120
00121 #endif  /* !__AVR_XMEGA__ */
00122 #endif  /* !__DOXYGEN__ */
00123 #endif  /* !_EEDEF_H_ */
```

22.93 sqrtdef.h

```
00001 #ifndef SQRTDEF_H
00002 #define SQRTDEF_H
00003
00004 #include "asmdef.h"
00005
00006 ;; The following utility macros define basic multi-register
00007 ;; operations for the integer square root algorithm from
00008 ;; an unsigned integer.  They all accept `size` parameter
00009 ;; which is the number of *bytes* in the *result*.  Some
00010 ;; of them also have `flag`, which is used when the first
00011 ;; instruction is different from the rest of the instructions
00012 ;; (e.g. does not use the C-flag).
00013
00014 ;; Set the initial value val to the register number rnum.
00015 ;; Also, use SUB instruction to set C=0, since this is a
00016 ;; precondition at the start.
00017 .macro initval _rnum _size _val
00018 .if \_size > 1
00019     sub \_rnum, \_rnum
00020     initval (\_rnum + 1), (\_size - 1), \_val
00021 .else
00022     ldi \_rnum, \_val
00023 .endif
00024 .endm
00025
00026 ;; Test next binary digit in developing square root.
00027 ;; If flag is set, the first instruction does not
00028 ;; use the carry flag.
00029 .macro testdigit _rarg _rres _size _flag
00030 .if \_flag
00031     cp  \_rarg + \_size, \_rres
00032 .else
00033     cpc \_rarg + \_size, \_rres
00034 .endif
00035 .if \_size > 1
00036     testdigit (\_rarg + 2), (\_rres + 1), (\_size - 1), 0
00037 .endif
00038 .endm
00039
00040 ;; Update Y_m (see notations below), if next binary
00041 ;; digit is set.  If flag is set, the first instruction
00042 ;; does not use carry.
00043 .macro updatearg _rarg _rres _size _flag
00044 .if \_flag
00045     sub  \_rarg + \_size, \_rres
00046 .else
00047     sbc \_rarg + \_size, \_rres
00048 .endif
00049 .if \_size > 1
00050     updatearg (\_rarg + 2), (\_rres + 1), (\_size - 1), 0
00051 .endif
00052 .endm
00053
```

```
00054 ;; Set binary digit in the result if X_m < Y_m is true.
00055 .macro setdigit _rres _mask _size
00056     or \_rres, \_mask
00057 .if \_size > 1
00058     setdigit (\_rres + 1), (\_mask + 1), (\_size - 1)
00059 .endif
00060 .endm
00061
00062 ;; Shift the rotation mask right. If flag is set, the first
00063 ;; instruction does not use C-flag.
00064 .macro shiftmask _mask _size _flag
00065 .if \_flag
00066     lsr \_mask + \_size - 1
00067 .else
00068     ror \_mask + \_size - 1
00069 .endif
00070 .if \_size > 1
00071     shiftmask \_mask, (\_size - 1), 0
00072 .endif
00073 .endm
00074
00075 ;; Set the next test binary digit in the result that will be
00076 ;; tested in the next iteration of the main loop.
00077 .macro shiftdigit _rres _mask _size
00078     eor \_rres + \_size - 1, \_mask + \_size - 1
00079 .if \_size > 1
00080     shiftdigit \_rres, \_mask, (\_size - 1)
00081 .endif
00082 .endm
00083
00084 ;; Argument is rolled left to keep relevant bits in the higher
00085 ;; (size / 2) bytes.  During the last iteration, this also rolls
00086 ;; carry bit into bit 0 of the LSB of the `rarg` registers.
00087 .macro shiftarg _rarg _size
00088     rol \_rarg
00089 .if \_size > 1
00090     shiftarg (\_rarg + 1), (\_size - 1)
00091 .endif
00092 .endm
00093
00094 ;; Last bit requires special treatment. The following macro saves
00095 ;; the last bit into C-flag, which is added to the result at the
00096 ;; very end.
00097 .macro setlast _rarg _rres _size
00098     cpc \_rres, \_rarg + \_size
00099 .if \_size > 1
00100     setlast (\_rarg + 2), (\_rres + 1), (\_size - 1)
00101 .endif
00102 .endm
00103
00104 ; Generic macro that generates an integer square root algorithm, where the
00105 ; result has 'size' bytes.  The argument provided in the registers starting
00106 ; with 'rarg' number and has twice the number of bytes than does
00107 ; the result.
00108 ;
00109 ; Based on the sqrt32_floor algorithm:
00110 ;
      https://www.mikrocontroller.net/articles/AVR_Arithmetik#avr-gcc-Implementierung_(32_Bit)
00111 ;
00112 ; The algorithm is built around the following inequality to test if
00113 ; c_{m-1} binary digit in the result is 0 or 1:
00114 ;         (P_m + 2^{m-1})^2 < N^2,
00115 ; where N^2 is the input, and P_m is the current developing square root that
00116 ; has m binary digits P_m = (c_n, c_{n-1}, ... c_m), where n is the number of
00117 ; bits in the result.
00118 ;
00119 ; This inequality can be rewritten as
00120 ;         X_m < Y_m,
00121 ; where
00122 ;         X_m = 2^m P_m + 2^{2m-2},
00123 ;         Y_m = N^2 - P_m^2,
00124 ; The recurrence relations to update X_m and Y_m are:
00125 ;         2X_m = X_{m + 1} + 2^{2m} + 2^{2m-1}, (n-bits) stored in rres,
```

```
00126 ;         Y_m = Y_{m + 1} - X_{m+1}           , (n-bits) stored in arg MSB,
00127 ; and the last two terms in 2X_m are kept in the rotation mask.
00128 ;
00129 ; For example, for 8-bit square root, starting values are:
00130 ; m = 8, P_8 = 0, X_8 = 2^14 = 0x4000, and Y_8 = N^2.
00131 ;
00132 ; Parameters:
00133 ;     rarg: Register number that contains LSB of the argument.
00134 ;     rres: Register number that contains LSB of the result.
00135 ;     mask: Register number that contains LSB of the mask.
00136 ;
00137 ; This macro neither saves any call-saved registers, nor sets it any registers
00138 ; in compliance with the AVR ABI.  These should be done separately.
00139 .macro sqrtengine _rarg _rres _mask _size
00140     ;; Set initial values and carry flag, C = 0.
00141     initval    \_mask, \_size, 0xc0      ; Rotation mask
00142 .if \_size <= 2 || \_rres % 2 != 0 || \_mask % 2 != 0
00143     initval    \_rres, \_size, 0x40      ; Expanding square root
00144 .else
00145     X_movw     \_rres, \_mask            ; Copy 0x0000 from _mask
00146     initval    \_rres+2, \_size-2, 0x40  ; Expanding square root
00147 .endif
00148 .if \_size == 1
00149     clc
00150 .endif
00151     ;; Precondition: C = 0.
00152 .Lnext_bit:
00153     brcs 1f                              ; if C = 1, X_m < Y_m already
00154     testdigit  \_rarg, \_rres, \_size, 1
00155     brcs 2f                              ; if X_m < Y_m, update X_m and Y_m
00156 1:
00157     updatearg  \_rarg, \_rres, \_size, 1 ; Update Y_m <- Y_m - X_m
00158     setdigit   \_rres, \_mask, \_size    ; Set next test digit in the result.
00159 2:
00160     shiftmask  \_mask, \_size, 1         ; Shift mask, C = 1 --> end of loop.
00161     shiftdigit \_rres, \_mask, \_size
00162     shiftarg   \_rarg, (2 * \_size)      ; Shift right, and save C into bit 0.
00163     sbrs       \_rarg, 0                 ; Exit if bit 0 is set from the mask
00164     rjmp .Lnext_bit                      ; through the C-flag.
00165
00166     brcs 3f                              ; C = 1 means last bit is 1.
00167     lsl        \_rarg + \_size - 1       ; Restore last bit in arg from C
00168     setlast    \_rarg, \_rres, \_size    ; after this, C is the last bit.
00169 3:
00170     adc        \_rres, __zero_reg__      ; Add last bit from C to the result.
00171 .endm
00172
00173 #endif /* SQRTDEF_H */
```

22.94 fdevopen.c File Reference

Functions

- FILE * fdevopen (int(*put)(char, FILE *), int(*get)(FILE *))

22.95 stdio_private.h

```
00001 /* Copyright (c) 2002,2005, Joerg Wunsch
00002    All rights reserved.
00003
00004    Redistribution and use in source and binary forms, with or without
00005    modification, are permitted provided that the following conditions are met:
00006
00007    * Redistributions of source code must retain the above copyright
00008      notice, this list of conditions and the following disclaimer.
00009    * Redistributions in binary form must reproduce the above copyright
00010      notice, this list of conditions and the following disclaimer in
```

```
00011      the documentation and/or other materials provided with the
00012      distribution.
00013    * Neither the name of the copyright holders nor the names of
00014      contributors may be used to endorse or promote products derived
00015      from this software without specific prior written permission.
00016
00017   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
00018   AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
00019   IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
00020   ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
00021   LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
00022   CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
00023   SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
00024   INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
00025   CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
00026   ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
00027   POSSIBILITY OF SUCH DAMAGE. */
00028
00029 #include <stdint.h>
00030 #include <stdio.h>
00031
00032 /* values for PRINTF_LEVEL */
00033 #define PRINTF_MIN 1
00034 #define PRINTF_STD 2
00035 #define PRINTF_FLT 3
00036
00037 /* values for SCANF_LEVEL */
00038 #define SCANF_MIN 1
00039 #define SCANF_STD 2
00040 #define SCANF_FLT 3
```

22.96 xtoa_fast.h

```
00001 /* Copyright (c) 2005, Dmitry Xmelkov
00002    All rights reserved.
00003
00004    Redistribution and use in source and binary forms, with or without
00005    modification, are permitted provided that the following conditions are met:
00006
00007    * Redistributions of source code must retain the above copyright
00008      notice, this list of conditions and the following disclaimer.
00009    * Redistributions in binary form must reproduce the above copyright
00010      notice, this list of conditions and the following disclaimer in
00011      the documentation and/or other materials provided with the
00012      distribution.
00013    * Neither the name of the copyright holders nor the names of
00014      contributors may be used to endorse or promote products derived
00015      from this software without specific prior written permission.
00016
00017   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
00018   AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
00019   IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
00020   ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
00021   LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
00022   CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
00023   SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
00024   INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
00025   CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
00026   ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
00027   POSSIBILITY OF SUCH DAMAGE. */
00028
00029 #ifndef _XTOA_FAST_H_
00030 #define _XTOA_FAST_H_
00031
00032 #ifndef __ASSEMBLER__
00033
00034 #include <stddef.h> /* for 'size_t' */
00035
00036 char * itoa_fast (int val, char *s, int base);
00037 char * utoa_fast (unsigned val, char *s, int base);
00038 char * ltoa_fast (long val, char *s, int base);
```

```
00039 char * ultoa_fast (unsigned long val, char *s, int base);
00040
00041 char * itoa_width (int val, char *s, int base, size_t width);
00042 char * utoa_width (unsigned val, char *s, int base, size_t width);
00043 char * ltoa_width (long val, char *s, int base, size_t width);
00044 char * ultoa_width (unsigned long val, char *s, int base, size_t width);
00045
00046 /* Internal function for use from 'printf'. */
00047 char * __ultoa_invert (unsigned long val, char *s, int base);
00048
00049 #endif  /* ifndef __ASSEMBLER__ */
00050
00051 /* Next flags are to use with 'base'. Unused fields are reserved.   */
00052 #define XTOA_PREFIX 0x0100  /* put prefix for octal or hex  */
00053 #define XTOA_UPPER  0x0200  /* use upper case letters   */
00054
00055 #endif  /* _XTOA_FAST_H_ */
```

22.97 ftoa_conv.h

```
00001 /* Copyright (c) 2005, Dmitry Xmelkov
00002    All rights reserved.
00003
00004    Redistribution and use in source and binary forms, with or without
00005    modification, are permitted provided that the following conditions are met:
00006
00007    * Redistributions of source code must retain the above copyright
00008      notice, this list of conditions and the following disclaimer.
00009    * Redistributions in binary form must reproduce the above copyright
00010      notice, this list of conditions and the following disclaimer in
00011      the documentation and/or other materials provided with the
00012      distribution.
00013    * Neither the name of the copyright holders nor the names of
00014      contributors may be used to endorse or promote products derived
00015      from this software without specific prior written permission.
00016
00017   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
00018   AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
00019   IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
00020   ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
00021   LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
00022   CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
00023   SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
00024   INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
00025   CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
00026   ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
00027   POSSIBILITY OF SUCH DAMAGE. */
00028
00029 #ifndef _FTOA_CONV_H
00030 #define _FTOA_CONV_H
00031
00032 #include <stdio.h>
00033
00034 int ftoa_prf (float val, char *s, unsigned char width, unsigned char prec,
00035               unsigned char flags);
00036
00037 #define DTOA_SPACE  0x01    /* put space for positives  */
00038 #define DTOA_PLUS   0x02    /* put '+' for positives    */
00039 #define DTOA_UPPER  0x04    /* use uppercase letters    */
00040 #define DTOA_ZFILL  0x08    /* fill zeroes          */
00041 #define DTOA_LEFT   0x10    /* adjust to left       */
00042 #define DTOA_NOFILL 0x20    /* do not fill to width     */
00043 #define DTOA_EXP    0x40    /* d2stream: 'e(E)' format  */
00044 #define DTOA_FIX    0x80    /* d2stream: 'f(F)' format  */
00045
00046 #define DTOA_EWIDTH     (-1)    /* Width too small  */
00047 #define DTOA_NONFINITE  (-2)    /* Value is NaN or Inf  */
00048
00049 #endif  /* _FTOA_CONV_H */
```

22.98 stdlib_private.h

```
00001 /* Copyright (c) 2004, Joerg Wunsch
00002    All rights reserved.
00003
00004    Redistribution and use in source and binary forms, with or without
00005    modification, are permitted provided that the following conditions are met:
00006
00007    * Redistributions of source code must retain the above copyright
00008      notice, this list of conditions and the following disclaimer.
00009    * Redistributions in binary form must reproduce the above copyright
00010      notice, this list of conditions and the following disclaimer in
00011      the documentation and/or other materials provided with the
00012      distribution.
00013    * Neither the name of the copyright holders nor the names of
00014      contributors may be used to endorse or promote products derived
00015      from this software without specific prior written permission.
00016
00017   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
00018   AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
00019   IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
00020   ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
00021   LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
00022   CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
00023   SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
00024   INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
00025   CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
00026   ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
00027   POSSIBILITY OF SUCH DAMAGE. */
00028
00029 #include <inttypes.h>
00030 #include <stdlib.h>
00031 #include <avr/io.h>
00032
00033 #if !defined(__DOXYGEN__)
00034
00035 struct __freelist {
00036     size_t sz;
00037     struct __freelist *nx;
00038 };
00039
00040 #endif
00041
00042 extern char *__brkval;      /* first location not yet allocated */
00043 extern struct __freelist *__flp; /* freelist pointer (head of freelist) */
00044 extern size_t __malloc_margin;   /* user-changeable before the first malloc() */
00045 extern char *__malloc_heap_start;
00046 extern char *__malloc_heap_end;
00047
00048 extern uint32_t __seed;
00049
00050 #ifndef __AVR__
00051
00052 /*
00053  * When compiling malloc.c/realloc.c natively on a host machine, it will
00054  * include a main() that performs a regression test.  This is meant as
00055  * a debugging aid, where a normal source-level debugger will help to
00056  * verify that the various allocator structures have the desired
00057  * appearance at each stage.
00058  *
00059  * When cross-compiling with avr-gcc, it will compile into just the
00060  * library functions malloc() and free().
00061  */
00062 #define MALLOC_TEST
00063
00064 #endif /* !__AVR__ */
00065
00066 #ifdef MALLOC_TEST
00067
00068 extern void *mymalloc(size_t);
00069 extern void myfree(void *);
00070 extern void *myrealloc(void *, size_t);
00071
```

```
00072 #define malloc mymalloc
00073 #define free myfree
00074 #define realloc myrealloc
00075
00076 #define __heap_start mymem[0]
00077 #define __heap_end mymem[256]
00078 extern char mymem[];
00079 #define STACK_POINTER() (mymem + 256)
00080
00081 #else /* !MALLOC_TEST */
00082
00083 extern char __heap_start;
00084 extern char __heap_end;
00085
00086 #define STACK_POINTER() ((char *) SP)
00087
00088 #endif /* MALLOC_TEST */
```

22.99 strto32.h

```
00001 /* Copyright (c) 2025  Georg-Johann Lay
00002    All rights reserved.
00003
00004    Redistribution and use in source and binary forms, with or without
00005    modification, are permitted provided that the following conditions are met:
00006
00007    * Redistributions of source code must retain the above copyright
00008      notice, this list of conditions and the following disclaimer.
00009    * Redistributions in binary form must reproduce the above copyright
00010      notice, this list of conditions and the following disclaimer in
00011      the documentation and/or other materials provided with the
00012      distribution.
00013    * Neither the name of the copyright holders nor the names of
00014      contributors may be used to endorse or promote products derived
00015      from this software without specific prior written permission.
00016
00017    THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
00018    AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
00019    IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
00020    ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
00021    LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
00022    CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
00023    SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
00024    INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
00025    CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
00026    ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
00027    POSSIBILITY OF SUCH DAMAGE.  */
00028
00029 ;;; Return the result in RET[]
00030 #define RET0    r22
00031 #define RET1    r23
00032 #define RET2    r24
00033 #define RET3    r25
00034
00035 ;;; During madd, the following regs contain porcelaine: R22, R23, R26...R29.
00036
00037 ;;; Expand the result in A[]
00038 #define A3      r19
00039 #define A2      r18
00040 #define A1      r17
00041 #define A0      r16
00042
00043 ;;; Used in __strto32.madd
00044
00045 #define Tmp     r20
00046
00047 #define B3      r15
00048 #define B2      r14
00049 #define B1      r25
00050 #define B0      r24
```

22.100 strto64.h

```
00001 /* Copyright (c) 2025  Georg-Johann Lay
00002    All rights reserved.
00003
00004    Redistribution and use in source and binary forms, with or without
00005    modification, are permitted provided that the following conditions are met:
00006
00007    * Redistributions of source code must retain the above copyright
00008      notice, this list of conditions and the following disclaimer.
00009    * Redistributions in binary form must reproduce the above copyright
00010      notice, this list of conditions and the following disclaimer in
00011      the documentation and/or other materials provided with the
00012      distribution.
00013    * Neither the name of the copyright holders nor the names of
00014      contributors may be used to endorse or promote products derived
00015      from this software without specific prior written permission.
00016
00017    THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
00018    AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
00019    IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
00020    ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
00021    LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
00022    CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
00023    SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
00024    INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
00025    CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
00026    ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
00027    POSSIBILITY OF SUCH DAMAGE.  */
00028
00029 ;;; Return the result in RET[]
00030 #define RET0    r18
00031 #define RET1    r19
00032 #define RET2    r20
00033 #define RET3    r21
00034 #define RET4    r22
00035 #define RET5    r23
00036 #define RET6    r24
00037 #define RET7    r25
00038
00039 ;;; During madd, the following regs contain porcelaine: R22, R23, R26...R29.
00040
00041 ;;; Expand the result in A[]
00042 #define A7      r17
00043 #define A6      r16
00044 #define A5      r15
00045 #define A4      r14
00046 #define A3      r13
00047 #define A2      r12
00048 #define A1      r19
00049 #define A0      r18
00050
00051 ;;; Used in __strto64.madd
00052
00053 #define Tmp     r20
00054
00055 #define B7      r25
00056 #define B6      r24
00057 #define B5      r11
00058 #define B4      r10
00059 #define B3      r9
00060 #define B2      r8
00061 #define B1      r1
00062 #define B0      r0
```

22.101 strtoxx.h

```
00001 /* Copyright (c) 2025  Georg-Johann Lay
00002    All rights reserved.
00003
00004    Redistribution and use in source and binary forms, with or without
```

```
00005    modification, are permitted provided that the following conditions are met:
00006
00007    * Redistributions of source code must retain the above copyright
00008      notice, this list of conditions and the following disclaimer.
00009    * Redistributions in binary form must reproduce the above copyright
00010      notice, this list of conditions and the following disclaimer in
00011      the documentation and/or other materials provided with the
00012      distribution.
00013    * Neither the name of the copyright holders nor the names of
00014      contributors may be used to endorse or promote products derived
00015      from this software without specific prior written permission.
00016
00017    THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
00018    AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
00019    IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
00020    ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
00021    LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
00022    CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
00023    SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
00024    INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
00025    CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
00026    ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
00027    POSSIBILITY OF SUCH DAMAGE.  */
00028
00029 #define Nptr    28
00030 #define ENDptr  ZL
00031 #define ENDptr2 r22
00032 #define Radix   r27
00033 #define Flags   r26
00034 #define Prefix  r25
00035 #define PFmadd  r30
00036 #define Digit   r21
00037
00038 ;;; Bits positions for Flags
00039 #define B_SIGN      0   /* Must be bit 0 */
00040 #define B_PREFIXED  1   /* Has 0x or similar prefix */
00041 #define B_NO_DIGITS 2   /* No digits yet, i.e. *endptr == nptr */
```

22.102 ephemera_common.h

```
00001 /*
00002  * Copyright (C) 2012 Michael Duane Rice All rights reserved.
00003  *
00004  * Redistribution and use in source and binary forms, with or without
00005  * modification, are permitted provided that the following conditions are
00006  * met:
00007  *
00008  * Redistributions of source code must retain the above copyright notice, this
00009  * list of conditions and the following disclaimer. Redistributions in binary
00010  * form must reproduce the above copyright notice, this list of conditions
00011  * and the following disclaimer in the documentation and/or other materials
00012  * provided with the distribution. Neither the name of the copyright holders
00013  * nor the names of contributors may be used to endorse or promote products
00014  * derived from this software without specific prior written permission.
00015  *
00016  * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
00017  * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
00018  * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
00019  * ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
00020  * LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
00021  * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
00022  * SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
00023  * INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
00024  * CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
00025  * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
00026  * POSSIBILITY OF SUCH DAMAGE. */
00027
00028 #ifndef EPHEMERA_PRIVATE_H
00029 #define EPHEMERA_PRIVATE_H
00030
00031 #define TROP_YEAR 31556925
```

```
00032 #define ANOM_YEAR 31558433
00033 #define INCLINATION 0.409105176667471    /* Earths axial tilt at the epoch */
00034 #define PERIHELION 31316400    /* perihelion of 1999, 03 jan 13:00 UTC */
00035 #define SOLSTICE 836160        /* winter solstice of 1999, 22 Dec 07:44 UTC */
00036 #define TWO_PI 6.283185307179586
00037 #define TROP_CYCLE 5022440.6025
00038 #define ANOM_CYCLE 5022680.6082
00039 #define DELTA_V 0.03342044    /* 2x orbital eccentricity */
00040
00041 #endif
```

22.103 time-private.h

```
00001 /* Copyright (c) 2025  Georg-Johann Lay
00002    All rights reserved.
00003
00004    Redistribution and use in source and binary forms, with or without
00005    modification, are permitted provided that the following conditions are met:
00006
00007    * Redistributions of source code must retain the above copyright
00008      notice, this list of conditions and the following disclaimer.
00009    * Redistributions in binary form must reproduce the above copyright
00010      notice, this list of conditions and the following disclaimer in
00011      the documentation and/or other materials provided with the
00012      distribution.
00013    * Neither the name of the copyright holders nor the names of
00014      contributors may be used to endorse or promote products derived
00015      from this software without specific prior written permission.
00016
00017    THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
00018    AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
00019    IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
00020    ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
00021    LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
00022    CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
00023    SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
00024    INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
00025    CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
00026    ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
00027    POSSIBILITY OF SUCH DAMAGE.  */
00028
00029 /* Function used locally in libc/time/ */
00030
00031 #include <stdint.h>
00032 #include <time.h>
00033
00034 extern long __latitude;
00035 extern long __longitude;
00036 extern long __utc_offset;
00037 extern char *__asc_store;
00038 extern struct tm __tm_store;
00039 extern int (*__dst_ptr) (const time_t*, int32_t*);
00040
00041 extern volatile time_t __system_time;
00042
00043 extern char* __print_10 (uint8_t, char*, char);
00044 extern void __print_x3210 (uint16_t, char*);
```

Index