avr-libc Reference Manual

1.4.5

Generated by Doxygen 1.4.6

Mon Oct 9 22:29:26 2006

Contents

1 AVR Libc

1.1 Introduction

The latest version of this document is always available from `http://savannah.nongnu.org/projects/avr-libc/`

The AVR Libc package provides a subset of the standard C library for `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 find yourself stuck on a problem which this document doesn't quite address, you may wish to post a message to the avr-gcc mailing list. Most of the developers of the AVR binutils and gcc ports in addition to the devleopers of avr-libc subscribe to the list, so you will usually be able to get your problem resolved. You can subscribe to the list at `http://lists.nongnu.org/mailman/listinfo/avr-gcc-list` . Before posting to the list, you might want to try reading the **Frequently Asked Questions**(p. 194) chapter of this document.

Note:

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 at `https://savannah.nongnu.org/bugs/?group=avr-libc` to ensure the issue won't be forgotten.

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* guarenteed to be reentrant. In particular, any functions that store local state are known to be non-reentrant, as well as functions that manipulate IO registers like the EEPROM access routines. If these functions are used within both, standard and interrupt context, undefined behaviour will result.

1.3 Supported Devices

The following is a list of AVR devices currently supported by the library. Note that actual support for some newer devices depends on the ability of the compiler/assembler to support these devices at library compile-time.

AT90S Type Devices:

- at90s1200 **[1]**(p. 5)
- at90s2313
- at90s2323
- at90s2333
- at90s2343
- at90s4414
- at90s4433
- at90s4434
- at90s8515
- at90c8534
- at90s8535

ATmega Type Devices:

- atmega8
- atmega103

- atmega128
- atmega1280
- atmega1281
- atmega16
- atmega161
- atmega162
- atmega163
- atmega164p
- atmega165
- atmega165p
- atmega168
- atmega169
- atmega169p
- atmega2560
- atmega2561
- atmega32
- atmega323
- atmega324p
- atmega325
- atmega3250
- atmega329
- atmega3290
- atmega48
- atmega406
- atmega64
- atmega640
- atmega644
- atmega644p

- atmega645
- atmega6450
- atmega649
- atmega6490
- atmega8515
- atmega8535
- atmega88

ATtiny Type Devices:

- attiny11 **[1]**(p. 5)
- attiny12 **[1]**(p. 5)
- attiny13
- attiny15 **[1]**(p. 5)
- attiny22
- attiny24
- attiny25
- attiny26
- attiny261
- attiny28 **[1]**(p. 5)
- attiny2313
- attiny44
- attiny45
- attiny461
- attiny84
- attiny85
- attiny861

Misc Devices:

- at94K **[2]**(p. 5)
- at76c711 **[3]**(p. 5)
- at43usb320
- at43usb355
- at86rf401
- at90can32
- at90can64
- at90can128
- at90pwm2
- at90pwm3
- at90usb646
- at90usb647
- at90usb1286
- at90usb1287

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**(p. 214) 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.

2 avr-libc Module Index

2.1 avr-libc Modules

Here is a list of all modules:

3 avr-libc Hierarchical Index

3.1 avr-libc Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

4 avr-libc Data Structure Index

4.1 avr-libc Data Structures

Here are the data structures with brief descriptions:

5 avr-libc Page Index

5.1 avr-libc Related Pages

Here is a list of all related documentation pages:

6 avr-libc Module Documentation

6.1 <assert.h>: Diagnostics

6.1.1 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()**(p. 99) will be called to halt the application.

Defines

- #define **assert**(expression)

6.1.2 Define Documentation

6.1.2.1 #define assert(expression)

Parameters:

expression Expression to test for.

The **assert()**(p. 9) macro tests the given expression and if it is false, the calling process is terminated. A diagnostic message is written to stderr and the function **abort()**(p. 99) is called, effectively terminating the program.

If expression is true, the **assert()**(p. 9) macro does nothing.

The **assert()**(p. 9) macro may be removed at compile time by defining NDEBUG as a macro (e.g., by using the compiler option -DNDEBUG).

6.2 <avr/boot.h>: Bootloader Support Utilities

6.2.1 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.

Todo

From email with Marek: On smaller devices (all except ATmega64/128), __SPM_REG is in the I/O space, accessible with the shorter "in" and "out" instructions - since the boot loader has a limited size, this could be an important optimization.

API Usage Example

The following code shows typical usage of the boot API.

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

void boot_program_page (uint32_t page, uint8_t *buf)
{
    uint16_t i;
    uint8_t sreg;

    // Disable interrupts.

    sreg = SREG;
    cli();

    eeprom_busy_wait ();

    boot_page_erase (page);
    boot_spm_busy_wait ();      // Wait until the memory is erased.

    for (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;
}
```

Defines

- #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(SPMEN))
- #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_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)

6.2.2 Define Documentation

6.2.2.1 #define boot_is_spm_interrupt() (__SPM_REG & (uint8_t)_BV(SPMIE))

Check if the SPM interrupt is enabled.

6.2.2.2 #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 (BLB12));
```

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.

6.2.2.3 #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()**(p. 11) except waits for eeprom and spm operations to complete before setting the lock bits.

6.2.2.4 #define boot_lock_fuse_bits_get(address)

Value:

```
(__extension__({                                           \
    uint8_t __result;                                      \
    __asm__ __volatile__                                   \
    (                                                      \
        "ldi r30, %3\n\t"                                  \
        "ldi r31, 0\n\t"                                   \
        "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),             \
          "M" (address)                                    \
        : "r0", "r30", "r31"                               \
    );                                                     \
    __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.

6.2.2.5 #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.

6.2.2.6 #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()**(p. 13) except it waits for eeprom and spm operations to complete before erasing the page.

6.2.2.7 #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.

6.2.2.8 #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()**(p. 13) except it waits for eeprom and spm operations to complete before filling the page.

6.2.2.9 #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.

6.2.2.10 #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()**(p. 14) except it waits for eeprom and spm operations to complete before writing the page.

6.2.2.11 #define boot_rww_busy() (__SPM_REG & (uint8_t)_BV(__COMMON_ASB))

Check if the RWW section is busy.

6.2.2.12 #define boot_rww_enable() __boot_rww_enable()

Enable the Read-While-Write memory section.

6.2.2.13 #define boot_rww_enable_safe()

Value:

```
do { \
    boot_spm_busy_wait();                      \
    eeprom_busy_wait();                        \
    boot_rww_enable();                         \
} while (0)
```

Same as **boot_rww_enable()**(p. 14) except waits for eeprom and spm operations to complete before enabling the RWW mameory.

6.2.2.14 #define boot_spm_busy() (__SPM_REG & (uint8_t)_BV(SPMEN))

Check if the SPM instruction is busy.

6.2.2.15 #define boot_spm_busy_wait() do{}while(boot_spm_busy())

Wait while the SPM instruction is busy.

6.2.2.16 #define boot_spm_interrupt_disable() (__SPM_REG &= (uint8_t)~_BV(SPMIE))

Disable the SPM interrupt.

6.2.2.17 #define boot_spm_interrupt_enable() (__SPM_REG |= (uint8_t)_BV(SPMIE))

Enable the SPM interrupt.

6.2.2.18 #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.

6.2.2.19 #define GET_EXTENDED_FUSE_BITS (0x0002)

address to read the extended fuse bits, using boot_lock_fuse_bits_get

6.2.2.20 #define GET_HIGH_FUSE_BITS (0x0003)

address to read the high fuse bits, using boot_lock_fuse_bits_get

6.2.2.21 #define GET_LOCK_BITS (0x0001)

address to read the lock bits, using boot_lock_fuse_bits_get

6.2.2.22 #define GET_LOW_FUSE_BITS (0x0000)

address to read the low fuse bits, using boot_lock_fuse_bits_get

6.3 <avr/eeprom.h>: EEPROM handling

6.3.1 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.

Note:

> 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()**(p. 18) before attempting any actual I/O.
> This header file declares inline functions that call the assembler subroutines directly. This prevents that the compiler generates push/pops for the call-clobbered registers. This way also a specific calling convention could be used for the eeprom routines e.g. by passing values in __tmp_reg__, eeprom addresses in X and memory addresses in Z registers. Method is optimized for code size.
> Presently supported are two locations of the EEPROM register set: 0x1F,0x20,0x21 and 0x1C,0x1D,0x1E (see **__EEPROM_REG_LOCATIONS__**(p. 17)).
> 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).

avr-libc declarations

- #define **EEMEM** __attribute__((section(".eeprom")))
- #define **eeprom_is_ready**()
- #define **eeprom_busy_wait**() do {} while (!eeprom_is_ready())
- **uint8_t eeprom_read_byte** (const **uint8_t** *addr)
- **uint16_t eeprom_read_word** (const **uint16_t** *addr)
- void **eeprom_read_block** (void *pointer_ram, const void *pointer_eeprom, size_t n)
- void **eeprom_write_byte** (**uint8_t** *addr, **uint8_t** value)
- void **eeprom_write_word** (**uint16_t** *addr, **uint16_t** value)
- void **eeprom_write_block** (const void *pointer_ram, void *pointer_eeprom, size_t n)

IAR C compatibility defines

- #define **_EEPUT**(addr, val) eeprom_write_byte ((**uint8_t** ∗)(addr), (**uint8_t**)(val))
- #define **_EEGET**(var, addr) (var) = eeprom_read_byte ((**uint8_t** ∗)(addr))

Defines

- #define **__EEPROM_REG_LOCATIONS__** 1C1D1E

6.3.2 Define Documentation

6.3.2.1 #define __EEPROM_REG_LOCATIONS__ 1C1D1E

In order to be able to work without a requiring a multilib approach for dealing with controllers having the EEPROM registers at different positions in memory space, the eeprom functions evaluate __EEPROM_REG_LOCATIONS__: It is assumed to be defined by the device io header and contains 6 uppercase hex digits encoding the addresses of EECR,EEDR and EEAR. First two letters: EECR address. Second two letters: EEDR address. Last two letters: EEAR address. The default 1C1D1E corresponds to the register location that is valid for most controllers. The value of this define symbol is used for appending it to the base name of the assembler functions.

6.3.2.2 #define _EEGET(var, addr) (var) = eeprom_read_byte ((uint8_t ∗)(addr))

Read a byte from EEPROM. Compatibility define for IAR C.

6.3.2.3 #define _EEPUT(addr, val) eeprom_write_byte ((uint8_t ∗)(addr), (uint8_t)(val))

Write a byte to EEPROM. Compatibility define for IAR C.

6.3.2.4 #define EEMEM __attribute__((section(".eeprom")))

Attribute expression causing a variable to be allocated within the .eeprom section.

6.3.2.5 #define eeprom_busy_wait() do {} while (!eeprom_is_ready())

Loops until the eeprom is no longer busy.

Returns:

Nothing.

6.3.2.6 #define eeprom_is_ready()

Returns:

1 if EEPROM is ready for a new read/write operation, 0 if not.

6.3.3 Function Documentation

6.3.3.1 void eeprom_read_block (void ∗ *pointer_ram*, const void ∗ *pointer_eeprom*, size_t *n*)

Read a block of `n` bytes from EEPROM address `pointer_eeprom` to `pointer_ram`. For constant n <= 256 bytes a library function is used. For block sizes unknown at compile time or block sizes > 256 an inline loop is expanded.

6.3.3.2 uint8_t eeprom_read_byte (const uint8_t ∗ *addr*)

Read one byte from EEPROM address `addr`.

6.3.3.3 uint16_t eeprom_read_word (const uint16_t ∗ *addr*)

Read one 16-bit word (little endian) from EEPROM address `addr`.

6.3.3.4 void eeprom_write_block (const void ∗ *pointer_ram*, void ∗ *pointer_eeprom*, size_t *n*)

Write a block of `n` bytes to EEPROM address `pointer_eeprom` from `pointer_ram`.

6.3.3.5 void eeprom_write_byte (uint8_t ∗ *addr*, uint8_t *value*)

Write a byte `value` to EEPROM address `addr`.

6.3.3.6 void eeprom_write_word (uint16_t ∗ *addr*, uint16_t *value*)

Write a word `value` to EEPROM address `addr`.

6.4 <avr/io.h>: AVR device-specific IO definitions

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

This header file includes the apropriate IO definitions for the device that has been specified by the `-mmcu=` compiler command-line switch. This is done by diverting to the appropriate file `<avr/io`*XXXX*`.h>` which should never be included directly. Some register names common to all AVR devices are defined directly within `<avr/io.h>`, but most of the details come from the respective include file.

Note that this file always includes

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

See **<avr/sfr_defs.h>: Special function registers**(p. 144) for the details.

Included are definitions of the IO register set and their respective bit values as specified in the Atmel documentation. Note that Atmel is not very consistent in its naming conventions, so even identical functions sometimes get different names on different devices.

Also included are the specific names useable for interrupt function definitions as documented **here**(p. 129).

Finally, the following macros are defined:

- **RAMEND**

  A constant describing the last on-chip RAM location.

- **XRAMEND**

  A constant describing the last possible location in RAM. This is equal to RAMEND for devices that do not allow for external RAM.

- **E2END**

  A constant describing the address of the last EEPROM cell.

- **FLASHEND**

  A constant describing the last byte address in flash ROM.

- **SPM_PAGESIZE**

  For devices with bootloader support, the flash pagesize (in bytes) to be used for the `SPM` instruction.

6.5 <avr/pgmspace.h>: Program Space String Utilities

6.5.1 Detailed Description

```
#include <avr/io.h>
#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. In order to use these functions, the target device must support either the `LPM` or `ELPM` instructions.

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 (GCC does not have full support for multiple address spaces yet).

If you are working with strings which are completely based in ram, use the standard string functions described in **<string.h>: Strings**(p. 107).

If possible, put your constant tables in the lower 64 KB and use **pgm_read_byte_near()**(p. 22) or **pgm_read_word_near()**(p. 23) instead of **pgm_read_byte_far()**(p. 22) or **pgm_read_word_far()**(p. 23) since it is more efficient that way, and you can still use the upper 64K for executable code. All functions that are suffixed with a `_P` *require* their arguments to be in the lower 64 KB 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 the macros from this header file so they are 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 KB of ROM. *All these functions will not work in that situation.*

Defines

- #define **PROGMEM** __ATTR_PROGMEM__
- #define **PSTR**(s) ((const PROGMEM char *)(s))
- #define **pgm_read_byte_near**(address_short) __LPM((**uint16_t**)(address_short))
- #define **pgm_read_word_near**(address_short) __LPM_word((**uint16_t**)(address_short))
- #define **pgm_read_dword_near**(address_short) __LPM_dword((**uint16_t**)(address_short))
- #define **pgm_read_byte_far**(address_long) __ELPM((**uint32_t**)(address_long))
- #define **pgm_read_word_far**(address_long) __ELPM_word((**uint32_t**)(address_long))
- #define **pgm_read_dword_far**(address_long) __ELPM_dword((**uint32_t**)(address_long))
- #define **pgm_read_byte**(address_short) pgm_read_byte_near(address_short)
- #define **pgm_read_word**(address_short) pgm_read_word_near(address_short)

- #define **pgm_read_dword**(address_short) pgm_read_dword_near(address_short)
- #define **PGM_P** const **prog_char** *
- #define **PGM_VOID_P** const **prog_void** *

Typedefs

- typedef void PROGMEM **prog_void**
- typedef char PROGMEM **prog_char**
- typedef unsigned char PROGMEM **prog_uchar**
- typedef **int8_t** PROGMEM **prog_int8_t**
- typedef **uint8_t** PROGMEM **prog_uint8_t**
- typedef **int16_t** PROGMEM **prog_int16_t**
- typedef **uint16_t** PROGMEM **prog_uint16_t**
- typedef **int32_t** PROGMEM **prog_int32_t**
- typedef **uint32_t** PROGMEM **prog_uint32_t**
- typedef **int64_t** PROGMEM **prog_int64_t**
- typedef **uint64_t** PROGMEM **prog_uint64_t**

Functions

- void * **memcpy_P** (void *, PGM_VOID_P, size_t)
- int **strcasecmp_P** (const char *, PGM_P) __ATTR_PURE__
- char * **strcat_P** (char *, PGM_P)
- int **strcmp_P** (const char *, PGM_P) __ATTR_PURE__
- char * **strcpy_P** (char *, PGM_P)
- size_t **strlcat_P** (char *, PGM_P, size_t)
- size_t **strlcpy_P** (char *, PGM_P, size_t)
- size_t **strlen_P** (PGM_P) __ATTR_CONST__
- int **strncasecmp_P** (const char *, PGM_P, size_t) __ATTR_PURE__
- char * **strncat_P** (char *, PGM_P, size_t)
- int **strncmp_P** (const char *, PGM_P, size_t) __ATTR_PURE__
- char * **strncpy_P** (char *, PGM_P, size_t)
- size_t **strnlen_P** (PGM_P, size_t) __ATTR_CONST__
- char * **strstr_P** (const char *, PGM_P) __ATTR_PURE__

6.5.2 Define Documentation

6.5.2.1 #define PGM_P const prog_char *

Used to declare a variable that is a pointer to a string in program space.

6.5.2.2 #define pgm_read_byte(address_short) pgm_read_byte_near(address_short)

Read a byte from the program space with a 16-bit (near) address.

Note:

The address is a byte address. The address is in the program space.

6.5.2.3 #define pgm_read_byte_far(address_long) __ELPM((uint32_t)(address_long))

Read a byte from the program space with a 32-bit (far) address.

Note:

The address is a byte address. The address is in the program space.

6.5.2.4 #define pgm_read_byte_near(address_short) __LPM((uint16_t)(address_short))

Read a byte from the program space with a 16-bit (near) address.

Note:

The address is a byte address. The address is in the program space.

6.5.2.5 #define pgm_read_dword(address_short) pgm_read_dword_near(address_short)

Read a double word from the program space with a 16-bit (near) address.

Note:

The address is a byte address. The address is in the program space.

6.5.2.6 #define pgm_read_dword_far(address_long) __ELPM_dword((uint32_t)(address_long))

Read a double word from the program space with a 32-bit (far) address.

Note:

The address is a byte address. The address is in the program space.

6.5.2.7 #define pgm_read_dword_near(address_short) __LPM_dword((uint16_t)(address_short))

Read a double word from the program space with a 16-bit (near) address.

Note:

The address is a byte address. The address is in the program space.

6.5.2.8 #define pgm_read_word(address_short) pgm_read_word_near(address_short)

Read a word from the program space with a 16-bit (near) address.

Note:

The address is a byte address. The address is in the program space.

6.5.2.9 #define pgm_read_word_far(address_long) __ELPM_word((uint32_t)(address_long))

Read a word from the program space with a 32-bit (far) address.

Note:

The address is a byte address. The address is in the program space.

6.5.2.10 #define pgm_read_word_near(address_short) __LPM_word((uint16_t)(address_short))

Read a word from the program space with a 16-bit (near) address.

Note:

The address is a byte address. The address is in the program space.

6.5.2.11 #define PGM_VOID_P const prog_void *

Used to declare a generic pointer to an object in program space.

6.5.2.12 #define PROGMEM __ATTR_PROGMEM__

Attribute to use in order to declare an object being located in flash ROM.

6.5.2.13 #define PSTR(s) ((const PROGMEM char *)(s))

Used to declare a static pointer to a string in program space.

6.5.3 Typedef Documentation

6.5.3.1 prog_char

Type of a "char" object located in flash ROM.

6.5.3.2 prog_int16_t

Type of an "int16_t" object located in flash ROM.

6.5.3.3 prog_int32_t

Type of an "int32_t" object located in flash ROM.

6.5.3.4 prog_int64_t

Type of an "int64_t" object located in flash ROM.

6.5.3.5 prog_int8_t

Type of an "int8_t" object located in flash ROM.

6.5.3.6 prog_uchar

Type of an "unsigned char" object located in flash ROM.

6.5.3.7 prog_uint16_t

Type of an "uint16_t" object located in flash ROM.

6.5.3.8 prog_uint32_t

Type of an "uint32_t" object located in flash ROM.

6.5.3.9 prog_uint64_t

Type of an "uint64_t" object located in flash ROM.

6.5.3.10 prog_uint8_t

Type of an "uint8_t" object located in flash ROM.

6.5.3.11 prog_void

Type of a "void" object located in flash ROM. Does not make much sense by itself, but can be used to declare a "void *" object in flash ROM.

6.5.4 Function Documentation

6.5.4.1 void * memcpy_P (void * *dest*, PGM_VOID_P *src*, size_t *n*)

The **memcpy_P()**(p. 25) function is similar to **memcpy()**(p. 110), except the src string resides in program space.

Returns:

The **memcpy_P()**(p. 25) function returns a pointer to dest.

6.5.4.2 int strcasecmp_P (const char * *s1*, PGM_P *s2*)

Compare two strings ignoring case.

The **strcasecmp_P()**(p. 25) function compares the two strings s1 and s2, ignoring the case of the characters.

Parameters:

s1 A pointer to a string in the devices SRAM.

s2 A pointer to a string in the devices Flash.

Returns:

The **strcasecmp_P()**(p. 25) 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.

6.5.4.3 char * strcat_P (char * *dest*, PGM_P *src*)

The **strcat_P()**(p. 25) function is similar to **strcat()**(p. 111) except that the *src* string must be located in program space (flash).

Returns:

The **strcat()**(p. 111) function returns a pointer to the resulting string *dest*.

6.5.4.4 int strcmp_P (const char * *s1*, PGM_P *s2*)

The **strcmp_P()**(p. 25) function is similar to **strcmp()**(p. 112) except that s2 is pointer to a string in program space.

Returns:

The **strcmp_P()**(p. 25) 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.

6.5.4.5 char * strcpy_P (char * *dest*, PGM_P *src*)

The **strcpy_P()**(p. 26) function is similar to **strcpy()**(p. 112) except that src is a pointer to a string in program space.

Returns:

The **strcpy_P()**(p. 26) function returns a pointer to the destination string dest.

6.5.4.6 size_t strlcat_P (char * *dst*, PGM_P, size_t *siz*)

Concatenate two strings.

The **strlcat_P()**(p. 26) function is similar to **strlcat()**(p. 112), except that the *src* string must be located in program space (flash).

Appends src to string dst of size siz (unlike **strncat()**(p. 114), 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()**(p. 26) function returns strlen(src) + MIN(siz, strlen(initial dst)). If retval >= siz, truncation occurred.

6.5.4.7 size_t strlcpy_P (char * *dst*, PGM_P, 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_P()**(p. 26) function returns strlen(src). If retval >= siz, truncation occurred.

6.5.4.8 size_t strlen_P (PGM_P *src*)

The **strlen_P()**(p. 26) function is similar to **strlen()**(p. 113), except that src is a pointer to a string in program space.

Returns:

The **strlen()**(p. 113) function returns the number of characters in src.

6.5.4.9 int strncasecmp_P (const char * *s1*, PGM_P *s2*, size_t *n*)

Compare two strings ignoring case.

The **strncasecmp_P()**(p. 27) function is similar to **strcasecmp_P()**(p. 25), except it only compares the first n characters of s1.

Parameters:

- ***s1*** A pointer to a string in the devices SRAM.
- ***s2*** A pointer to a string in the devices Flash.
- ***n*** The maximum number of bytes to compare.

Returns:

The **strcasecmp_P()**(p. 25) 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.

6.5.4.10 char * strncat_P (char * *dest*, PGM_P *src*, size_t *len*)

Concatenate two strings.

The **strncat_P()**(p. 27) function is similar to **strncat()**(p. 114), except that the *src* string must be located in program space (flash).

Returns:

The **strncat_P()**(p. 27) function returns a pointer to the resulting string dest.

6.5.4.11 int strncmp_P (const char * *s1*, PGM_P *s2*, size_t *n*)

The **strncmp_P()**(p. 27) function is similar to **strcmp_P()**(p. 25) except it only compares the first (at most) n characters of s1 and s2.

Returns:

The **strncmp_P()**(p. 27) 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.

6.5.4.12 char * strncpy_P (char * *dest*, PGM_P *src*, size_t *n*)

The **strncpy_P()**(p. 27) function is similar to **strcpy_P()**(p. 26) 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_P()**(p. 27) function returns a pointer to the destination string dest.

6.5.4.13 size_t strnlen_P (PGM_P *src*, size_t *len*)

Determine the length of a fixed-size string.

The **strnlen_P()**(p. 28) function is similar to **strnlen()**(p. 114), except that `src` is a pointer to a string in 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`.

6.5.4.14 char * strstr_P (const char * *s1*, PGM_P *s2*)

Locate a substring.

The **strstr_P()**(p. 28) function finds the first occurrence of the substring `s2` in the string `s1`. The terminating '\0' characters are not compared. The **strstr_P()**(p. 28) function is similar to **strstr()**(p. 116) except that `s2` is pointer to a string in program space.

Returns:

The **strstr_P()**(p. 28) 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`.

6.6 <avr/power.h>: Power Reduction Management

```
#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.

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 ATmega128). On those devices without a Power Reduction Register, these 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.

Power Macro	Description	Applicable for device
power_adc_enable()	Enable the Analog to Digital Converter module.	ATmega640, ATmega1280, ATmega1281, ATmega2560, ATmega2561, AT90USB646, AT90USB647, AT90USB1286, AT90USB1287, AT90PWM1, AT90PWM2, AT90PWM3, ATmega165, ATmega165P, ATmega325, ATmega3250, ATmega645, ATmega6450, ATmega169, ATmega169P, ATmega329, ATmega3290, ATmega649, ATmega6490, ATmega164P, ATmega324P, ATmega644, ATmega48, ATmega88, ATmega168, ATtiny24, ATtiny44, ATtiny84, ATtiny25, ATtiny45, ATtiny85, ATtiny261, ATtiny461, ATtiny861
power_adc_disable()	Disable the Analog to Digital Converter module.	ATmega640, ATmega1280, ATmega1281, ATmega2560, ATmega2561, AT90USB646, AT90USB647, AT90USB1286, AT90USB1287, AT90PWM1, AT90PWM2, AT90PWM3, ATmega165, ATmega165P, ATmega325, ATmega3250, ATmega645, ATmega6450, ATmega169, ATmega169P, ATmega329, ATmega3290, ATmega649, ATmega6490, ATmega164P, ATmega324P, ATmega644, ATmega48, ATmega88, ATmega168, ATtiny24, ATtiny44, ATtiny84, ATtiny25, ATtiny45, ATtiny85, ATtiny261, ATtiny461, ATtiny861
power_lcd_enable()	Enable the LCD module.	ATmega169, ATmega169P, ATmega329, ATmega3290, ATmega649, ATmega6490
power_lcd_disable().	Disable the LCD module.	ATmega169, ATmega169P, ATmega329, ATmega3290, ATmega649, ATmega6490
power_psc0_enable()	Enable the Power Stage Controller 0 module.	AT90PWM1, AT90PWM2, AT90PWM3
power_psc0_disable()	Disable the Power Stage Controller 0 module.	AT90PWM1, AT90PWM2, AT90PWM3
power_psc1_enable()	Enable the Power Stage Controller 1 module.	AT90PWM1, AT90PWM2, AT90PWM3
power_psc1_disable()	Disable the Power Stage Controller 1 module.	AT90PWM1, AT90PWM2, AT90PWM3
power_psc2_enable()	Enable the Power Stage Controller 2 module.	AT90PWM1, AT90PWM2, AT90PWM3

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. Below are two macros and an enumerated type that can be used to interface to the Clock Prescale Register.

Note:

Not all AVR devices have a Clock Prescale Register. On those devices without a Clock Prescale 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_t;
```

Clock prescaler setting enumerations.

```
 clock_prescale_set(x)
```

Set the clock prescaler register select bits, selecting a system clock division setting. They type of x is clock_div_t.

```
 clock_prescale_get()
```

Gets and returns the clock prescaler register setting. The return type is clock_div_t.

6.7 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).

6.8 <avr/sleep.h>: Power Management and Sleep Modes

6.8.1 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()**(p. 35) (it usually defaults to idle mode where the CPU is put to sleep but all peripheral clocks are still running), and then call **sleep_mode()**(p. 35). Unless it is the purpose to lock the CPU hard (until a hardware reset), interrupts need to be enabled at this point. This macro automatically takes care to enable the sleep mode in the CPU before going to sleep, and disable it again afterwards.

As this combined 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()**(p. 35), **sleep_disable()**(p. 35), and **sleep_cpu()**(p. 35). 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>

...
  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 intruction 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.

Sleep Modes

Note:

Some of these modes are not available on all devices. See the datasheet for target device for the available sleep modes.

- #define **SLEEP_MODE_IDLE** 0
- #define **SLEEP_MODE_ADC** _BV(SM0)
- #define **SLEEP_MODE_PWR_DOWN** _BV(SM1)
- #define **SLEEP_MODE_PWR_SAVE** (_BV(SM0) | _BV(SM1))
- #define **SLEEP_MODE_STANDBY** (_BV(SM1) | _BV(SM2))
- #define **SLEEP_MODE_EXT_STANDBY** (_BV(SM0) | _BV(SM1) | _BV(SM2))

Sleep Functions

- void **set_sleep_mode** (**uint8_t** mode)
- void **sleep_mode** (void)
- void **sleep_enable** (void)
- void **sleep_disable** (void)
- void **sleep_cpu** (void)

6.8.2 Define Documentation

6.8.2.1 #define SLEEP_MODE_ADC _BV(SM0)

ADC Noise Reduction Mode.

6.8.2.2 #define SLEEP_MODE_EXT_STANDBY (_BV(SM0) | _BV(SM1) | _BV(SM2))

Extended Standby Mode.

6.8.2.3 #define SLEEP_MODE_IDLE 0

Idle mode.

6.8.2.4 #define SLEEP_MODE_PWR_DOWN _BV(SM1)

Power Down Mode.

6.8.2.5 #define SLEEP_MODE_PWR_SAVE (_BV(SM0) | _BV(SM1))

Power Save Mode.

6.8.2.6 #define SLEEP_MODE_STANDBY (_BV(SM1) | _BV(SM2))

Standby Mode.

6.8.3 Function Documentation

6.8.3.1 void set_sleep_mode (uint8_t *mode*)

Select a sleep mode.

6.8.3.2 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.

6.8.3.3 void sleep_disable (void)

Clear the SE (sleep enable) bit.

6.8.3.4 void sleep_enable (void)

Set the SE (sleep enable) bit.

6.8.3.5 void sleep_mode (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()**(p. 35) function. See the data sheet for your device for more details.

6.9 <avr/version.h>: avr-libc version macros

6.9.1 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.

Defines

- #define **__AVR_LIBC_VERSION_STRING__** "1.4.5"
- #define **__AVR_LIBC_VERSION__** 10405UL
- #define **__AVR_LIBC_DATE_STRING__** "20061009"
- #define **__AVR_LIBC_DATE_** 20061009UL
- #define **__AVR_LIBC_MAJOR__** 1
- #define **__AVR_LIBC_MINOR__** 4
- #define **__AVR_LIBC_REVISION__** 5

6.9.2 Define Documentation

6.9.2.1 #define __AVR_LIBC_DATE_ 20061009UL

Numerical representation of the release date.

6.9.2.2 #define __AVR_LIBC_DATE_STRING__ "20061009"

String literal representation of the release date.

6.9.2.3 #define __AVR_LIBC_MAJOR__ 1

Library major version number.

6.9.2.4 #define __AVR_LIBC_MINOR__ 4

Library minor version number.

6.9.2.5 #define __AVR_LIBC_REVISION__ 5

Library revision number.

6.9.2.6 #define __AVR_LIBC_VERSION__ 10405UL

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 intented to provide a monotonically increasing numerical value that can easily be used in numerical checks.

6.9.2.7 #define __AVR_LIBC_VERSION_STRING__ "1.4.5"

String literal representation of the current library version.

6.10 <avr/wdt.h>: Watchdog timer handling

6.10.1 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;

void get_mcusr(void) \
  __attribute__((naked)) \
  __attribute__((section(".init3")));
```

```
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 clearing in particular the watchdog reset flag before disabling the watchdog is required, according to the datasheet.

Defines

- #define **wdt_reset**() __asm__ __volatile__ ("wdr")
- #define **wdt_disable**()
- #define **wdt_enable**(timeout) _wdt_write(timeout)
- #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

6.10.2 Define Documentation

6.10.2.1 #define wdt_disable()

Value:

```
__asm__ __volatile__ (  \
    "in __tmp_reg__, __SREG__" "\n\t" \
     "cli" "\n\t" \
    "out %0, %1" "\n\t" \
    "out %0, __zero_reg__" "\n\t" \
    "out __SREG__,__tmp_reg__" "\n\t" \
    : /* no outputs */ \
    : "I" (_SFR_IO_ADDR(_WD_CONTROL_REG)), \
    "r" ((uint8_t)(_BV(_WD_CHANGE_BIT) | _BV(WDE))) \
    : "r0" \
)
```

Disable the watchdog timer, if possible. This attempts to turn off the Enable bit in the watchdog control register. See the datasheet for details.

6.10.2.2 #define wdt_enable(timeout) _wdt_write(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.

6.10.2.3 #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.

6.10.2.4 #define WDTO_120MS 3

See `WDTO_15MS`

6.10.2.5 #define WDTO_15MS 0

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);
```

6.10.2.6 #define WDTO_1S 6

See `WDTO_15MS`

6.10.2.7 #define WDTO_250MS 4

See `WDTO_15MS`

6.10.2.8 #define WDTO_2S 7

See `WDTO_15MS`

6.10.2.9 #define WDTO_30MS 1

See `WDTO_15MS`

6.10.2.10 #define WDTO_4S 8

See `WDTO_15MS` Note: This is only available on the ATtiny2313, ATtiny24, ATtiny44, ATtiny84, ATtiny25, ATtiny45, ATtiny85, ATtiny261, ATtiny461, ATtiny861, ATmega48, ATmega88, ATmega168, ATmega164P, ATmega324P, ATmega644P, ATmega644, ATmega640, ATmega1280, ATmega1281, ATmega2560, ATmega2561, ATmega406, AT90PWM2, AT90PWM3, AT90USB646, AT90USB647, AT90USB1286, AT90USB1287.

6.10.2.11 #define WDTO_500MS 5

See `WDTO_15MS`

6.10.2.12 #define WDTO_60MS 2

`WDTO_15MS`

6.10.2.13 #define WDTO_8S 9

See `WDTO_15MS` Note: This is only available on the ATtiny2313, ATtiny24, ATtiny44, ATtiny84, ATtiny25, ATtiny45, ATtiny85, ATtiny261, ATtiny461, ATtiny861, ATmega48, ATmega88, ATmega168, ATmega164P, ATmega324P, ATmega644P, ATmega644, ATmega640, ATmega1280, ATmega1281, ATmega2560, ATmega2561, ATmega406, AT90PWM2, AT90PWM3, AT90USB646, AT90USB647, AT90USB1286, AT90USB1287.

6.11 <compat/deprecated.h>: Deprecated items

6.11.1 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/deprected.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.

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 inadvertantly disable it by enabling another intterupt.

- #define **enable_external_int**(mask) (__EICR = mask)
- #define **INTERRUPT**(signame)
- #define **__INTR_ATTRS** used
- static __inline__ void **timer_enable_int** (unsigned char ints)

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))

6.11.2 Define Documentation

6.11.2.1 #define cbi(port, bit) (port) &= ~(1 << (bit))

Deprecated

Clear `bit` in IO port `port`.

6.11.2.2 #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.

6.11.2.3 #define inb(port) (port)

Deprecated

Read a value from an IO port `port`.

6.11.2.4 #define inp(port) (port)

Deprecated

Read a value from an IO port `port`.

6.11.2.5 #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**(p. 128).

6.11.2.6 #define outb(port, val) (port) = (val)

Deprecated

Write `val` to IO port `port`.

6.11.2.7 #define outp(val, port) (port) = (val)

Deprecated

Write `val` to IO port `port`.

6.11.2.8 #define sbi(port, bit) (port) |= (1 << (bit))

Deprecated

Set `bit` in IO port `port`.

6.11.3 Function Documentation

6.11.3.1 static __inline__ void timer_enable_int (unsigned char *ints*) `[static]`

Deprecated

This function modifies the `timsk` register. The value you pass via `ints` is device specific.

6.12 <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.

6.13 <ctype.h>: Character Operations

6.13.1 Detailed Description

These functions perform various operations on characters.

```
#include <ctype.h>
```

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()**(p. 45) returns true if its argument is any value '0' though '9', inclusive.)

- int **isalnum** (int __c) __ATTR_CONST__
- int **isalpha** (int __c) __ATTR_CONST__
- int **isascii** (int __c) __ATTR_CONST__
- int **isblank** (int __c) __ATTR_CONST__
- int **iscntrl** (int __c) __ATTR_CONST__
- int **isdigit** (int __c) __ATTR_CONST__
- int **isgraph** (int __c) __ATTR_CONST__
- int **islower** (int __c) __ATTR_CONST__
- int **isprint** (int __c) __ATTR_CONST__
- int **ispunct** (int __c) __ATTR_CONST__
- int **isspace** (int __c) __ATTR_CONST__
- int **isupper** (int __c) __ATTR_CONST__
- int **isxdigit** (int __c) __ATTR_CONST__

Character convertion routines

If `c` is not an unsigned char value, or `EOF`, the behaviour of these functions is undefined.

- int **toascii** (int __c) __ATTR_CONST__
- int **tolower** (int __c) __ATTR_CONST__
- int **toupper** (int __c) __ATTR_CONST__

6.13.2 Function Documentation

6.13.2.1 int isalnum (int __c)

Checks for an alphanumeric character. It is equivalent to `(isalpha(c) || isdigit(c))`.

6.13.2.2 int isalpha (int __c)

Checks for an alphabetic character. It is equivalent to `(isupper(c) || islower(c))`.

6.13.2.3 int isascii (int __c)

Checks whether `c` is a 7-bit unsigned char value that fits into the ASCII character set.

6.13.2.4 int isblank (int __c)

Checks for a blank character, that is, a space or a tab.

6.13.2.5 int iscntrl (int __c)

Checks for a control character.

6.13.2.6 int isdigit (int __c)

Checks for a digit (0 through 9).

6.13.2.7 int isgraph (int __c)

Checks for any printable character except space.

6.13.2.8 int islower (int __c)

Checks for a lower-case character.

6.13.2.9 int isprint (int __c)

Checks for any printable character including space.

6.13.2.10 int ispunct (int __c)

Checks for any printable character which is not a space or an alphanumeric character.

6.13.2.11 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').

6.13.2.12 int isupper (int __c)

Checks for an uppercase letter.

6.13.2.13 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.

6.13.2.14 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.

6.13.2.15 int tolower (int __c)

Converts the letter `c` to lower case, if possible.

6.13.2.16 int toupper (int __c)

Converts the letter `c` to upper case, if possible.

6.14 <errno.h>: System Errors

6.14.1 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.

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.

Defines

- #define **EDOM** 33
- #define **ERANGE** 34

6.14.2 Define Documentation

6.14.2.1 #define EDOM 33

Domain error.

6.14.2.2 #define ERANGE 34

Range error.

6.15 <inttypes.h>: Integer Type conversions

6.15.1 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**(p. 77). 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()**(p. 88) or **scanf()**(p. 89) 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);
```

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 **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 **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

Far pointers for memory access >64K

- typedef **int32_t** **int_farptr_t**
- typedef **uint32_t** **uint_farptr_t**

6.15.2 Define Documentation

6.15.2.1 #define PRId16 "d"

decimal printf format for int16_t

6.15.2.2 #define PRId32 "ld"

decimal printf format for int32_t

6.15.2.3 #define PRId8 "d"

decimal printf format for int8_t

6.15.2.4 #define PRIdFAST16 "d"

decimal printf format for int_fast16_t

6.15.2.5 #define PRIdFAST32 "ld"

decimal printf format for int_fast32_t

6.15.2.6 #define PRIdFAST8 "d"

decimal printf format for int_fast8_t

6.15.2.7 #define PRIdLEAST16 "d"

decimal printf format for int_least16_t

6.15.2.8 #define PRIdLEAST32 "ld"

decimal printf format for int_least32_t

6.15.2.9 #define PRIdLEAST8 "d"

decimal printf format for int_least8_t

6.15.2.10 #define PRIdPTR PRId16

decimal printf format for intptr_t

6.15.2.11 #define PRIi16 "i"

integer printf format for int16_t

6.15.2.12 #define PRIi32 "li"

integer printf format for int32_t

6.15.2.13 #define PRIi8 "i"

integer printf format for int8_t

6.15.2.14 #define PRIiFAST16 "i"

integer printf format for int_fast16_t

6.15.2.15 #define PRIiFAST32 "li"

integer printf format for int_fast32_t

6.15.2.16 #define PRIiFAST8 "i"

integer printf format for int_fast8_t

6.15.2.17 #define PRIiLEAST16 "i"

integer printf format for int_least16_t

6.15.2.18 #define PRIiLEAST32 "li"

integer printf format for int_least32_t

6.15.2.19 #define PRIiLEAST8 "i"

integer printf format for int_least8_t

6.15.2.20 #define PRIiPTR PRIi16

integer printf format for intptr_t

6.15.2.21 #define PRIo16 "o"

octal printf format for uint16_t

6.15.2.22 #define PRIo32 "lo"

octal printf format for uint32_t

6.15.2.23 #define PRIo8 "o"

octal printf format for uint8_t

6.15.2.24 #define PRIoFAST16 "o"

octal printf format for uint_fast16_t

6.15.2.25 #define PRIoFAST32 "lo"

octal printf format for uint_fast32_t

6.15.2.26 #define PRIoFAST8 "o"

octal printf format for uint_fast8_t

6.15.2.27 #define PRIoLEAST16 "o"

octal printf format for uint_least16_t

6.15.2.28 #define PRIoLEAST32 "lo"

octal printf format for uint_least32_t

6.15.2.29 #define PRIoLEAST8 "o"

octal printf format for uint_least8_t

6.15.2.30 #define PRIoPTR PRIo16

octal printf format for uintptr_t

6.15.2.31 #define PRIu16 "u"

decimal printf format for uint16_t

6.15.2.32 #define PRIu32 "lu"

decimal printf format for uint32_t

6.15.2.33 #define PRIu8 "u"

decimal printf format for uint8_t

6.15.2.34 #define PRIuFAST16 "u"

decimal printf format for uint_fast16_t

6.15.2.35 #define PRIuFAST32 "lu"

decimal printf format for uint_fast32_t

6.15.2.36 #define PRIuFAST8 "u"

decimal printf format for uint_fast8_t

6.15.2.37 #define PRIuLEAST16 "u"

decimal printf format for uint_least16_t

6.15.2.38 #define PRIuLEAST32 "lu"

decimal printf format for uint_least32_t

6.15.2.39 #define PRIuLEAST8 "u"

decimal printf format for uint_least8_t

6.15.2.40 #define PRIuPTR PRIu16

decimal printf format for uintptr_t

6.15.2.41 #define PRIX16 "X"

uppercase hexadecimal printf format for uint16_t

6.15.2.42 #define PRIx16 "x"

hexadecimal printf format for uint16_t

6.15.2.43 #define PRIX32 "lX"

uppercase hexadecimal printf format for uint32_t

6.15.2.44 #define PRIx32 "lx"

hexadecimal printf format for uint32_t

6.15.2.45 #define PRIX8 "X"

uppercase hexadecimal printf format for uint8_t

6.15.2.46 #define PRIx8 "x"

hexadecimal printf format for uint8_t

6.15.2.47 #define PRIXFAST16 "X"

uppercase hexadecimal printf format for uint_fast16_t

6.15.2.48 #define PRIxFAST16 "x"

hexadecimal printf format for uint_fast16_t

6.15.2.49 #define PRIXFAST32 "lX"

uppercase hexadecimal printf format for uint_fast32_t

6.15.2.50 #define PRIxFAST32 "lx"

hexadecimal printf format for uint_fast32_t

6.15.2.51 #define PRIXFAST8 "X"

uppercase hexadecimal printf format for uint_fast8_t

6.15.2.52 #define PRIxFAST8 "x"

hexadecimal printf format for uint_fast8_t

6.15.2.53 #define PRIXLEAST16 "X"

uppercase hexadecimal printf format for uint_least16_t

6.15.2.54 #define PRIxLEAST16 "x"

hexadecimal printf format for uint_least16_t

6.15.2.55 #define PRIXLEAST32 "lX"

uppercase hexadecimal printf format for uint_least32_t

6.15.2.56 #define PRIxLEAST32 "lx"

hexadecimal printf format for uint_least32_t

6.15.2.57 #define PRIXLEAST8 "X"

uppercase hexadecimal printf format for uint_least8_t

6.15.2.58 #define PRIxLEAST8 "x"

hexadecimal printf format for uint_least8_t

6.15.2.59 #define PRIXPTR PRIX16

uppercase hexadecimal printf format for uintptr_t

6.15.2.60 #define PRIxPTR PRIx16

hexadecimal printf format for uintptr_t

6.15.2.61 #define SCNd16 "d"

decimal scanf format for int16_t

6.15.2.62 #define SCNd32 "ld"

decimal scanf format for int32_t

6.15.2.63 #define SCNdFAST16 "d"

decimal scanf format for int_fast16_t

6.15.2.64 #define SCNdFAST32 "ld"

decimal scanf format for int_fast32_t

6.15.2.65 #define SCNdLEAST16 "d"

decimal scanf format for int_least16_t

6.15.2.66 #define SCNdLEAST32 "ld"

decimal scanf format for int_least32_t

6.15.2.67 #define SCNdPTR SCNd16

decimal scanf format for intptr_t

6.15.2.68 #define SCNi16 "i"

generic-integer scanf format for int16_t

6.15.2.69 #define SCNi32 "li"

generic-integer scanf format for int32_t

6.15.2.70 #define SCNiFAST16 "i"

generic-integer scanf format for int_fast16_t

6.15.2.71 #define SCNiFAST32 "li"

generic-integer scanf format for int_fast32_t

6.15.2.72 #define SCNiLEAST16 "i"

generic-integer scanf format for int_least16_t

6.15.2.73 #define SCNiLEAST32 "li"

generic-integer scanf format for int_least32_t

6.15.2.74 #define SCNiPTR SCNi16

generic-integer scanf format for intptr_t

6.15.2.75 #define SCNo16 "o"

octal scanf format for uint16_t

6.15.2.76 #define SCNo32 "lo"

octal scanf format for uint32_t

6.15.2.77 #define SCNoFAST16 "o"

octal scanf format for uint_fast16_t

6.15.2.78 #define SCNoFAST32 "lo"

octal scanf format for uint_fast32_t

6.15.2.79 #define SCNoLEAST16 "o"

octal scanf format for uint_least16_t

6.15.2.80 #define SCNoLEAST32 "lo"

octal scanf format for uint_least32_t

6.15.2.81 #define SCNoPTR SCNo16

octal scanf format for uintptr_t

6.15.2.82 #define SCNu16 "u"

decimal scanf format for uint16_t

6.15.2.83 #define SCNu32 "lu"

decimal scanf format for uint32_t

6.15.2.84 #define SCNuFAST16 "u"

decimal scanf format for uint_fast16_t

6.15.2.85 #define SCNuFAST32 "lu"

decimal scanf format for uint_fast32_t

6.15.2.86 #define SCNuLEAST16 "u"

decimal scanf format for uint_least16_t

6.15.2.87 #define SCNuLEAST32 "lu"

decimal scanf format for uint_least32_t

6.15.2.88 #define SCNuPTR SCNu16

decimal scanf format for uintptr_t

6.15.2.89 #define SCNx16 "x"

hexadecimal scanf format for uint16_t

6.15.2.90 #define SCNx32 "lx"

hexadecimal scanf format for uint32_t

6.15.2.91 #define SCNxFAST16 "x"

hexadecimal scanf format for uint_fast16_t

6.15.2.92 #define SCNxFAST32 "lx"

hexadecimal scanf format for uint_fast32_t

6.15.2.93 #define SCNxLEAST16 "x"

hexadecimal scanf format for uint_least16_t

6.15.2.94 #define SCNxLEAST32 "lx"

hexadecimal scanf format for uint_least32_t

6.15.2.95 #define SCNxPTR SCNx16

hexadecimal scanf format for uintptr_t

6.15.3 Typedef Documentation

6.15.3.1 typedef int32_t int_farptr_t

signed integer type that can hold a pointer > 64 KB

6.15.3.2 typedef uint32_t uint_farptr_t

unsigned integer type that can hold a pointer > 64 KB

6.16 <math.h>: Mathematics

6.16.1 Detailed Description

```
#include <math.h>
```

This header file declares basic mathematics constants and functions.

Note:

> In order to access the functions delcared herein, it is usually also required to additionally link against the library `libm.a`. See also the related **FAQ entry**(p. 196).

Defines

- #define **M_PI** 3.14159265358979323846264
- #define **M_SQRT2** 1.4142135623730950488016887

Functions

- double **cos** (double __x) __ATTR_CONST__
- double **fabs** (double __x) __ATTR_CONST__
- double **fmod** (double __x, double __y) __ATTR_CONST__
- double **modf** (double __value, double ∗__iptr)
- double **sin** (double __x) __ATTR_CONST__
- double **sqrt** (double __x) __ATTR_CONST__
- double **tan** (double __x) __ATTR_CONST__
- double **floor** (double __x) __ATTR_CONST__
- double **ceil** (double __x) __ATTR_CONST__
- double **frexp** (double __value, int ∗__exp)
- double **ldexp** (double __x, int __exp) __ATTR_CONST__
- double **exp** (double __x) __ATTR_CONST__
- double **cosh** (double __x) __ATTR_CONST__
- double **sinh** (double __x) __ATTR_CONST__
- double **tanh** (double __x) __ATTR_CONST__
- double **acos** (double __x) __ATTR_CONST__
- double **asin** (double __x) __ATTR_CONST__
- double **atan** (double __x) __ATTR_CONST__
- double **atan2** (double __y, double __x) __ATTR_CONST__
- double **log** (double __x) __ATTR_CONST__
- double **log10** (double __x) __ATTR_CONST__
- double **pow** (double __x, double __y) __ATTR_CONST__
- int **isnan** (double __x) __ATTR_CONST__
- int **isinf** (double __x) __ATTR_CONST__
- double **square** (double __x) __ATTR_CONST__

6.16.2 Define Documentation

6.16.2.1 #define M_PI 3.14159265358979323846264

The constant `pi`.

6.16.2.2 #define M_SQRT2 1.4142135623730950488016887

The square root of 2.

6.16.3 Function Documentation

6.16.3.1 double acos (double __*x*)

The **acos()**(p. 61) function computes the principal value of the arc cosine of `x`. The returned value is in the range [0, pi] radians. A domain error occurs for arguments not in the range [-1, +1].

6.16.3.2 double asin (double __*x*)

The **asin()**(p. 61) function computes the principal value of the arc sine of `x`. The returned value is in the range [0, pi] radians. A domain error occurs for arguments not in the range [-1, +1].

6.16.3.3 double atan (double __*x*)

The **atan()**(p. 61) function computes the principal value of the arc tangent of `x`. The returned value is in the range [0, pi] radians. A domain error occurs for arguments not in the range [-1, +1].

6.16.3.4 double atan2 (double __*y*, double __*x*)

The **atan2()**(p. 61) 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 [-pi, +pi] radians. If both `x` and `y` are zero, the global variable `errno` is set to `EDOM`.

6.16.3.5 double ceil (double __*x*)

The **ceil()**(p. 61) function returns the smallest integral value greater than or equal to `x`, expressed as a floating-point number.

6.16.3.6 double cos (double __*x*)

The **cos()**(p. 61) function returns the cosine of `x`, measured in radians.

6.16.3.7 double cosh (double __*x*)

The **cosh()**(p. 61) function returns the hyperbolic cosine of `x`.

6.16.3.8 double exp (double __*x*)

The **exp()**(p. 61) function returns the exponential value of `x`.

6.16.3.9 double fabs (double __ *x*)

The **fabs()**(p. 62) function computes the absolute value of a floating-point number `x`.

6.16.3.10 double floor (double __ *x*)

The **floor()**(p. 62) function returns the largest integral value less than or equal to `x`, expressed as a floating-point number.

6.16.3.11 double fmod (double __ *x*, double __ *y*)

The function **fmod()**(p. 62) returns the floating-point remainder of `x / y`.

6.16.3.12 double frexp (double __ *value*, int * __ *exp*)

The **frexp()**(p. 62) 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 `exp`.

The **frexp()**(p. 62) function returns the value `x`, such that `x` is a double with magnitude in the interval [1/2, 1) or zero, and `value` equals `x` times 2 raised to the power `*exp`. If value is zero, both parts of the result are zero.

6.16.3.13 int isinf (double __ *x*)

The function **isinf()**(p. 62) returns 1 if the argument `x` is either positive or negative infinity, otherwise 0.

6.16.3.14 int isnan (double __ *x*)

The function **isnan()**(p. 62) returns 1 if the argument `x` represents a "not-a-number" (NaN) object, otherwise 0.

6.16.3.15 double ldexp (double __ *x*, int __ *exp*)

The **ldexp()**(p. 62) function multiplies a floating-point number by an integral power of 2.

The **ldexp()**(p. 62) function returns the value of `x` times 2 raised to the power `exp`.

If the resultant value would cause an overflow, the global variable errno is set to ERANGE, and the value NaN is returned.

6.16.3.16 double log (double __ *x*)

The **log()**(p. 62) function returns the natural logarithm of argument `x`.

If the argument is less than or equal 0, a domain error will occur.

6.16.3.17 double log10 (double __ *x*)

The **log10()**(p. 63) function returns the logarithm of argument `x` to base 10.

If the argument is less than or equal 0, a domain error will occur.

6.16.3.18 double modf (double __ *value*, double * __ *iptr*)

The **modf()**(p. 63) function breaks the argument `value` 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()**(p. 63) function returns the signed fractional part of `value`.

6.16.3.19 double pow (double __ *x*, double __ *y*)

The function **pow()**(p. 63) returns the value of `x` to the exponent `y`.

6.16.3.20 double sin (double __ *x*)

The **sin()**(p. 63) function returns the sine of `x`, measured in radians.

6.16.3.21 double sinh (double __ *x*)

The **sinh()**(p. 63) function returns the hyperbolic sine of `x`.

6.16.3.22 double sqrt (double __ *x*)

The **sqrt()**(p. 63) function returns the non-negative square root of `x`.

6.16.3.23 double square (double __ *x*)

The function **square()**(p. 63) returns `x * x`.

Note:

This function does not belong to the C standard definition.

6.16.3.24 double tan (double __ *x*)

The **tan()**(p. 63) function returns the tangent of `x`, measured in radians.

6.16.3.25 double tanh (double _ _ *x*)

The **tanh()**(p. 64) function returns the hyperbolic tangent of `x`.

6.17 <setjmp.h>: Non-local goto

6.17.1 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()**(p. 65) and **longjmp()**(p. 65) functions. **setjmp()**(p. 65) and **longjmp()**(p. 65) are useful for dealing with errors and interrupts encountered in a low-level subroutine of a program.

Note:

setjmp()(p. 65) and **longjmp()**(p. 65) make programs hard to understand and maintain. If possible, an alternative should be used.

longjmp()(p. 65) can destroy changes made to global register variables (see **How to permanently bind a variable to a register?**(p. 196)).

For a very detailed discussion of **setjmp()**(p. 65)/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() some where ...
    }
}

...

void foo (void)
{
    ... blah, blah, blah ...

    if (err)
    {
```

```
            longjmp (env, 1);
        }
    }
```

Functions

- int **setjmp** (jmp_buf __jmpb)
- void **longjmp** (jmp_buf __jmpb, int __ret) __ATTR_NORETURN__

6.17.2 Function Documentation

6.17.2.1 void longjmp (jmp_buf __*jmpb*, int __*ret*)

Non-local jump to a saved stack context.

```
#include <setjmp.h>
```

longjmp()(p. 65) restores the environment saved by the last call of **setjmp()**(p. 65) with the corresponding *__jmpb* argument. After **longjmp()**(p. 65) is completed, program execution continues as if the corresponding call of **setjmp()**(p. 65) had just returned the value *__ret*.

Note:

longjmp()(p. 65) cannot cause 0 to be returned. If **longjmp()**(p. 65) is invoked with a second argument of 0, 1 will be returned instead.

Parameters:

__jmpb Information saved by a previous call to **setjmp()**(p. 65).

__ret Value to return to the caller of **setjmp()**(p. 65).

Returns:

This function never returns.

6.17.2.2 int setjmp (jmp_buf __*jmpb*)

Save stack context for non-local goto.

```
#include <setjmp.h>
```

setjmp()(p. 65) saves the stack context/environment in *__jmpb* for later use by **longjmp()**(p. 65). The stack context will be invalidated if the function which called **setjmp()**(p. 65) returns.

Parameters:

__*jmpb* Variable of type `jmp_buf` which holds the stack information such that the environment can be restored.

Returns:

setjmp()(p. 65) returns 0 if returning directly, and non-zero when returning from **longjmp()**(p. 65) using the saved context.

6.18 <stdint.h>: Standard Integer Types

6.18.1 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.

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 **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_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_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

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** (__CONCAT(INT16_MAX, U))

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 **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)

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 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**

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 **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 **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**

6.18.2 Define Documentation

6.18.2.1 #define INT16_C(value) value

define a constant of type int16_t

6.18.2.2 #define INT16_MAX 0x7fff

largest positive value an int16_t can hold.

6.18.2.3 #define INT16_MIN (-INT16_MAX - 1)

smallest negative value an int16_t can hold.

6.18.2.4 #define INT32_C(value) __CONCAT(value, L)

define a constant of type int32_t

6.18.2.5 #define INT32_MAX 0x7fffffffL

largest positive value an int32_t can hold.

6.18.2.6 #define INT32_MIN (-INT32_MAX - 1L)

smallest negative value an int32_t can hold.

6.18.2.7 #define INT64_C(value) __CONCAT(value, LL)

define a constant of type int64_t

6.18.2.8 #define INT64_MAX 0x7fffffffffffffffLL

largest positive value an int64_t can hold.

6.18.2.9 #define INT64_MIN (-INT64_MAX - 1LL)

smallest negative value an int64_t can hold.

6.18.2.10 #define INT8_C(value) ((int8_t) value)

define a constant of type int8_t

6.18.2.11 #define INT8_MAX 0x7f

largest positive value an int8_t can hold.

6.18.2.12 #define INT8_MIN (-INT8_MAX - 1)

smallest negative value an int8_t can hold.

6.18.2.13 #define INT_FAST16_MAX INT16_MAX

largest positive value an int_fast16_t can hold.

6.18.2.14 #define INT_FAST16_MIN INT16_MIN

smallest negative value an int_fast16_t can hold.

6.18.2.15 #define INT_FAST32_MAX INT32_MAX

largest positive value an int_fast32_t can hold.

6.18.2.16 #define INT_FAST32_MIN INT32_MIN

smallest negative value an int_fast32_t can hold.

6.18.2.17 #define INT_FAST64_MAX INT64_MAX

largest positive value an int_fast64_t can hold.

6.18.2.18 #define INT_FAST64_MIN INT64_MIN

smallest negative value an int_fast64_t can hold.

6.18.2.19 #define INT_FAST8_MAX INT8_MAX

largest positive value an int_fast8_t can hold.

6.18.2.20 #define INT_FAST8_MIN INT8_MIN

smallest negative value an int_fast8_t can hold.

6.18.2.21 #define INT_LEAST16_MAX INT16_MAX

largest positive value an int_least16_t can hold.

6.18.2.22 #define INT_LEAST16_MIN INT16_MIN

smallest negative value an int_least16_t can hold.

6.18.2.23 #define INT_LEAST32_MAX INT32_MAX

largest positive value an int_least32_t can hold.

6.18.2.24 #define INT_LEAST32_MIN INT32_MIN

smallest negative value an int_least32_t can hold.

6.18.2.25 #define INT_LEAST64_MAX INT64_MAX

largest positive value an int_least64_t can hold.

6.18.2.26 #define INT_LEAST64_MIN INT64_MIN

smallest negative value an int_least64_t can hold.

6.18.2.27 #define INT_LEAST8_MAX INT8_MAX

largest positive value an int_least8_t can hold.

6.18.2.28 #define INT_LEAST8_MIN INT8_MIN

smallest negative value an int_least8_t can hold.

6.18.2.29 #define INTMAX_C(value) __CONCAT(value, LL)

define a constant of type intmax_t

6.18.2.30 #define INTMAX_MAX INT64_MAX

largest positive value an intmax_t can hold.

6.18.2.31 #define INTMAX_MIN INT64_MIN

smallest negative value an intmax_t can hold.

6.18.2.32 #define INTPTR_MAX INT16_MAX

largest positive value an intptr_t can hold.

6.18.2.33 #define INTPTR_MIN INT16_MIN

smallest negative value an intptr_t can hold.

6.18.2.34 #define PTRDIFF_MAX INT16_MAX

largest positive value a ptrdiff_t can hold.

6.18.2.35 #define PTRDIFF_MIN INT16_MIN

smallest negative value a ptrdiff_t can hold.

6.18.2.36 #define SIG_ATOMIC_MAX INT8_MAX

largest positive value a sig_atomic_t can hold.

6.18.2.37 #define SIG_ATOMIC_MIN INT8_MIN

smallest negative value a sig_atomic_t can hold.

6.18.2.38 #define SIZE_MAX (__CONCAT(INT16_MAX, U))

largest value a size_t can hold.

6.18.2.39 #define UINT16_C(value) __CONCAT(value, U)

define a constant of type uint16_t

6.18.2.40 #define UINT16_MAX (__CONCAT(INT16_MAX, U) * 2U + 1U)

largest value an uint16_t can hold.

6.18.2.41 #define UINT32_C(value) __CONCAT(value, UL)

define a constant of type uint32_t

6.18.2.42 #define UINT32_MAX (__CONCAT(INT32_MAX, U) * 2UL + 1UL)

largest value an uint32_t can hold.

6.18.2.43 #define UINT64_C(value) __CONCAT(value, ULL)

define a constant of type uint64_t

6.18.2.44 #define UINT64_MAX (__CONCAT(INT64_MAX, U) * 2ULL + 1ULL)

largest value an uint64_t can hold.

6.18.2.45 #define UINT8_C(value) ((uint8_t) __CONCAT(value, U))

define a constant of type uint8_t

6.18.2.46 #define UINT8_MAX (__CONCAT(INT8_MAX, U) * 2U + 1U)

largest value an uint8_t can hold.

6.18.2.47 #define UINT_FAST16_MAX UINT16_MAX

largest value an uint_fast16_t can hold.

6.18.2.48 #define UINT_FAST32_MAX UINT32_MAX

largest value an uint_fast32_t can hold.

6.18.2.49 #define UINT_FAST64_MAX UINT64_MAX

largest value an uint_fast64_t can hold.

6.18.2.50 #define UINT_FAST8_MAX UINT8_MAX

largest value an uint_fast8_t can hold.

6.18.2.51 #define UINT_LEAST16_MAX UINT16_MAX

largest value an uint_least16_t can hold.

6.18.2.52 #define UINT_LEAST32_MAX UINT32_MAX

largest value an uint_least32_t can hold.

6.18.2.53 #define UINT_LEAST64_MAX UINT64_MAX

largest value an uint_least64_t can hold.

6.18.2.54 #define UINT_LEAST8_MAX UINT8_MAX

largest value an uint_least8_t can hold.

6.18.2.55 #define UINTMAX_C(value) __CONCAT(value, ULL)

define a constant of type uintmax_t

6.18.2.56 #define UINTMAX_MAX UINT64_MAX

largest value an uintmax_t can hold.

6.18.2.57 #define UINTPTR_MAX UINT16_MAX

largest value an uintptr_t can hold.

6.18.3 Typedef Documentation

6.18.3.1 typedef signed int int16_t

16-bit signed type.

6.18.3.2 typedef signed long int int32_t

32-bit signed type.

6.18.3.3 typedef signed long long int int64_t

64-bit signed type.

6.18.3.4 typedef signed char int8_t

8-bit signed type.

6.18.3.5 typedef int16_t int_fast16_t

fastest signed int with at least 16 bits.

6.18.3.6 typedef int32_t int_fast32_t

fastest signed int with at least 32 bits.

6.18.3.7 typedef int64_t int_fast64_t

fastest signed int with at least 64 bits.

6.18.3.8 typedef int8_t int_fast8_t

fastest signed int with at least 8 bits.

6.18.3.9 typedef int16_t int_least16_t

signed int with at least 16 bits.

6.18.3.10 typedef int32_t int_least32_t

signed int with at least 32 bits.

6.18.3.11 typedef int64_t int_least64_t

signed int with at least 64 bits.

6.18.3.12 typedef int8_t int_least8_t

signed int with at least 8 bits.

6.18.3.13 typedef int64_t intmax_t

largest signed int available.

6.18.3.14 typedef int16_t intptr_t

Signed pointer compatible type.

6.18.3.15 typedef unsigned int uint16_t

16-bit unsigned type.

6.18.3.16 typedef unsigned long int uint32_t

32-bit unsigned type.

6.18.3.17 typedef unsigned long long int uint64_t

64-bit unsigned type.

6.18.3.18 typedef unsigned char uint8_t

8-bit unsigned type.

6.18.3.19 typedef uint16_t uint_fast16_t

fastest unsigned int with at least 16 bits.

6.18.3.20 typedef uint32_t uint_fast32_t

fastest unsigned int with at least 32 bits.

6.18.3.21 typedef uint64_t uint_fast64_t

fastest unsigned int with at least 64 bits.

6.18.3.22 typedef uint8_t uint_fast8_t

fastest unsigned int with at least 8 bits.

6.18.3.23 typedef uint16_t uint_least16_t

unsigned int with at least 16 bits.

6.18.3.24 typedef uint32_t uint_least32_t

unsigned int with at least 32 bits.

6.18.3.25 typedef uint64_t uint_least64_t

unsigned int with at least 64 bits.

6.18.3.26 typedef uint8_t uint_least8_t

unsigned int with at least 8 bits.

6.18.3.27 typedef uint64_t uintmax_t

largest unsigned int available.

6.18.3.28 typedef uint16_t uintptr_t

Unsigned pointer compatible type.

6.19 <stdio.h>: Standard IO facilities

6.19.1 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()**(p. 90) which is the heart of the printf family can be selected in different flavours using linker options. See the documentation of **vfprintf()**(p. 90) for a detailed description. The same applies to **vfscanf()**(p. 93) 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**(p. 80).) Instead, the function **fdevopen()**(p. 86) 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**(p. 80)).

As an alternative method to **fdevopen()**(p. 86), the macro **fdev_setup_stream()**(p. 84) 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()**(p. 86) that opens a stream for reading will cause the resulting stream to be aliased to `stdin`. Likewise, the first call to **fdevopen()**(p. 86) 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()**(p. 85) on such a stream will also effectively close all of its aliases (**note 3**(p. 80)).

It is possible to tie additional user data to a stream, using **fdev_set_udata()**(p. 83). The backend put and get functions can then extract this user data using **fdev_get_udata()**(p. 83), 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 two flavours: the standard name, where the format string is expected to be in SRAM, as well as a version with the suffix "_P" where the format string is expected to reside in the flash ROM. The macro `PSTR` (explained in **<avr/pgmspace.h>: Program Space String Utilities**(p. 19)) becomes very handy for declaring these format strings.

Running stdio without malloc()(p. 102) By default, **fdevopen()**(p. 86) as well as the floating-point versions of the printf and scanf family require **malloc()**(p. 102). As this is often not desired in the limited environment of a microcontroller, an alternative option is provided to run completely without **malloc()**(p. 102).

The macro **fdev_setup_stream()**(p. 84) is provided to prepare a user-supplied FILE buffer for operation with stdio. If floating-point operation is desired, a user-supplied buffer can as well be passed for the internal buffering for the floating-point numbers (and processing of %[scanf data).

Example

```
#include <stdio.h>

static int uart_putchar(char c, FILE *stream);

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()**(p. 83) rather than the function-like **fdev_setup_stream()**(p. 84), 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()**(p. 83), and then destroying the object itself. No call to **fclose()**(p. 85) should be issued for these streams. While calling **fclose()**(p. 85) itself is harmless, it will cause an undefined reference to **free()**(p. 101) and thus cause the linker to link the malloc module into the application.

Notes

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()**(p. 86) that talks to a UART interface might look like this:

```
int
uart_putchar(char c, FILE *stream)
{

  if (c == '\n')
    uart_putchar('\r');
  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()(p. 88) 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` will also save some execution time.

Defines

- #define **FILE** struct __file
- #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**()
- #define **putc**(__c, __stream) fputc(__c, __stream)
- #define **putchar**(__c) fputc(__c, stdout)
- #define **getc**(__stream) fgetc(__stream)
- #define **getchar**() fgetc(stdin)

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 *))

6.19.2 Define Documentation

6.19.2.1 #define _FDEV_EOF (-2)

Return code for an end-of-file condition during device read.

To be used in the get function of **fdevopen()**(p. 86).

6.19.2.2 #define _FDEV_ERR (-1)

Return code for an error condition during device read.

To be used in the get function of **fdevopen()**(p. 86).

6.19.2.3 #define _FDEV_SETUP_READ __SRD

fdev_setup_stream()(p. 84) with read intent

6.19.2.4 #define _FDEV_SETUP_RW (__SRD|__SWR)

fdev_setup_stream()(p. 84) with read/write intent

6.19.2.5 #define _FDEV_SETUP_WRITE __SWR

fdev_setup_stream()(p. 84) with write intent

6.19.2.6 #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.

6.19.2.7 #define fdev_close()

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.)

6.19.2.8 #define fdev_get_udata(stream) ((stream) → udata)

This macro retrieves a pointer to user defined data from a FILE stream object.

6.19.2.9 #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()**(p. 86) function.

6.19.2.10 #define FDEV_SETUP_STREAM(put, get, rwflag)

Initializer for a user-supplied stdio stream.

This macro acts similar to **fdev_setup_stream()**(p. 84), 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()**(p. 84).

6.19.2.11 #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()**(p. 86). 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()**(p. 86).

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()**(p. 84). If standard streams are to be used, these need to be assigned by the user. See also under **Running stdio without malloc()**(p. 79).

6.19.2.12 #define FILE struct __file

`FILE` is the opaque structure that is passed around between the various standard IO functions.

6.19.2.13 #define getc(__stream) fgetc(__stream)

The macro `getc` used to be a "fast" macro implementation with a functionality identical to **fgetc()**(p. 87). For space constraints, in `avr-libc`, it is just an alias for `fgetc`.

6.19.2.14 #define getchar(void) fgetc(stdin)

The macro `getchar` reads a character from `stdin`. Return values and error handling is identical to **fgetc()**(p. 87).

6.19.2.15 #define putc(__c, __stream) fputc(__c, __stream)

The macro `putc` used to be a "fast" macro implementation with a functionality identical to **fputc()**(p. 87). For space constraints, in `avr-libc`, it is just an alias for `fputc`.

6.19.2.16 #define putchar(__c) fputc(__c, stdout)

The macro `putchar` sends character `c` to `stdout`.

6.19.2.17 #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()**(p. 86) must be explicitly assigned to it without closing the previous `stderr` (since this would also close `stdout`).

6.19.2.18 #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()**(p. 86) will be assigned to `stdin`.

6.19.2.19 #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()**(p. 86) will be assigned to both, `stdin`, and `stderr`.

6.19.3 Function Documentation

6.19.3.1 void clearerr (FILE * *__stream*)

Clear the error and end-of-file flags of `stream`.

6.19.3.2 int fclose (FILE * *__stream*)

This function closes `stream`, and disallows and further IO to and from it.

When using **fdevopen()**(p. 86) to setup the stream, a call to **fclose()**(p. 85) is needed in order to free the internal resources allocated.

If the stream has been set up using **fdev_setup_stream()**(p. 84) or **FDEV_SETUP_STREAM()**(p. 83), use **fdev_close()**(p. 83) instead.

It currently always returns 0 (for success).

6.19.3.3 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()(p. 86) uses **calloc()**(p. 100) (und thus **malloc()**(p. 102)) 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()**(p. 86) 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.

6.19.3.4 int feof (FILE ∗ __*stream*)

Test the end-of-file flag of `stream`. This flag can only be cleared by a call to **clearerr()**(p. 85).

6.19.3.5 int ferror (FILE ∗ __*stream*)

Test the error flag of `stream`. This flag can only be cleared by a call to **clearerr()**(p. 85).

6.19.3.6 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.

6.19.3.7 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()**(p. 86) or **ferror()**(p. 86) must be used to distinguish between both situations.

6.19.3.8 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()**(p. 86). Otherwise, a pointer to the string will be returned.

6.19.3.9 int fprintf (FILE * *__stream*, const char * *__fmt*, ...)

The function `fprintf` performs formatted output to `stream`. See **vfprintf()**(p. 90) for details.

6.19.3.10 int fprintf_P (FILE * *__stream*, const char * *__fmt*, ...)

Variant of **fprintf()**(p. 87) that uses a `fmt` string that resides in program memory.

6.19.3.11 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.

6.19.3.12 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.

6.19.3.13 int fputs_P (const char * __*str*, FILE * __*stream*)

Variant of **fputs()**(p. 87) where `str` resides in program memory.

6.19.3.14 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 occured or end-of-file was encountered. **feof()**(p. 86) and **ferror()**(p. 86) must be used to distinguish between these two conditions.

6.19.3.15 int fscanf (FILE * __*stream*, const char * __*fmt*, ...)

The function `fscanf` performs formatted input, reading the input data from `stream`.

See **vfscanf()**(p. 93) for details.

6.19.3.16 int fscanf_P (FILE * __*stream*, const char * __*fmt*, ...)

Variant of **fscanf()**(p. 88) using a `fmt` string in program memory.

6.19.3.17 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 occured.

6.19.3.18 char* gets (char * __*str*)

Similar to **fgets()**(p. 87) 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.

6.19.3.19 int printf (const char * __*fmt*, ...)

The function `printf` performs formatted output to stream `stderr`. See **vfprintf()**(p. 90) for details.

6.19.3.20 int printf_P (const char ∗ __*fmt*, ...)

Variant of **printf()** (p. 88) that uses a `fmt` string that resides in program memory.

6.19.3.21 int puts (const char ∗ __*str*)

Write the string pointed to by `str`, and a trailing newline character, to `stdout`.

6.19.3.22 int puts_P (const char ∗ __*str*)

Variant of **puts()** (p. 89) where `str` resides in program memory.

6.19.3.23 int scanf (const char ∗ __*fmt*, ...)

The function `scanf` performs formatted input from stream `stdin`.

See **vfscanf()** (p. 93) for details.

6.19.3.24 int scanf_P (const char ∗ __*fmt*, ...)

Variant of **scanf()** (p. 89) where `fmt` resides in program memory.

6.19.3.25 int snprintf (char ∗ __*s*, size_t __*n*, const char ∗ __*fmt*, ...)

Like **sprintf()** (p. 89), 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.

6.19.3.26 int snprintf_P (char ∗ __*s*, size_t __*n*, const char ∗ __*fmt*, ...)

Variant of **snprintf()** (p. 89) that uses a `fmt` string that resides in program memory.

6.19.3.27 int sprintf (char ∗ __*s*, const char ∗ __*fmt*, ...)

Variant of **printf()** (p. 88) that sends the formatted characters to string `s`.

6.19.3.28 int sprintf_P (char ∗ __*s*, const char ∗ __*fmt*, ...)

Variant of **sprintf()** (p. 89) that uses a `fmt` string that resides in program memory.

6.19.3.29 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()**(p. 93) for details.

6.19.3.30 int sscanf_P (const char ∗ __buf, const char ∗ __fmt, ...)

Variant of **sscanf()**(p. 90) using a `fmt` string in program memory.

6.19.3.31 int ungetc (int __c, FILE ∗ __stream)

The **ungetc()**(p. 90) 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()**(p. 90) 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.

6.19.3.32 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-adjustŋ ment 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` length modifier, that specifies that the argument for the d, i, o, u, x, or X conversion is a `"long int"` rather than `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.dddeśdd"` 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()**(p. 90) can be selected using linker options. The default **vfprintf()**(p. 90) implements all the mentioned functionality except floating point conversions. A minimized version of **vfprintf()**(p. 90) is available that only implements the very basic integer and string conversion facilities, but none of the additional options that can be specified using conversion flags (these flags are parsed correctly from the format specification, but then simply ignored). This version can be requested using the following **compiler options**(p. 257):

```
-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 -lm
```

Limitations:

- The specified width and precision can be at most 127.
- For floating-point conversions, trailing digits will be lost if a number close to DBL_MAX is converted with a precision > 0.

6.19.3.33 int vfprintf_P (FILE * __*stream*, const char * __*fmt*, va_list __*ap*)

Variant of **vfprintf()**(p. 90) that uses a `fmt` string that resides in program memory.

6.19.3.34 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 character `l` indicating that the argument is a pointer to `long int` (rather than `int`, for integer type conversions), or a pointer to `double` (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 127 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, 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.

- `p` Matches a pointer value (as printed by `p` in **printf()**(p. 88)); 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 `%[` conversion. These conversions will be available in the extended version provided by the library `libscanf_flt.a`. Note that either of these conversions requires the availability of a buffer that needs to be obtained at run-time using **malloc()**(p. 102). If this buffer cannot be obtained, the operation is aborted, returning the value `EOF`. To link a program against the extended version, use the following compiler flags in the link stage:

```
-Wl,-u,vfscanf -lscanf_flt -lm
```

A third version is available for environments that are tight on space. This version is provided in the library `libscanf_min.a`, and can be requested using the following options in the link stage:

```
-Wl,-u,vfscanf -lscanf_min -lm
```

In addition to the restrictions of the standard version, this version implements no field width specification, no conversion assignment suppression flag (∗), no `n` specification, and no general format character matching at all. All characters in `fmt` that do not comprise a conversion specification will simply be ignored, including white space (that is normally used to consume *any* amount of white space in the input stream). However, the usual skip of initial white space in the formats that support it is implemented.

6.19.3.35 int vfscanf_P (FILE ∗ __*stream*, const char ∗ __*fmt*, va_list __ *ap*)

Variant of **vfscanf()**(p. 93) using a `fmt` string in program memory.

6.19.3.36 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()**(p. 90).

See **vfprintf()**(p. 90) for details.

6.19.3.37 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()**(p. 93).

See **vfscanf()**(p. 93) for details.

6.19.3.38 int vsnprintf (char ∗ __ *s*, size_t __ *n*, const char ∗ __-*fmt*, va_list *ap*)

Like **vsprintf()**(p. 96), 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.

6.19.3.39 int vsnprintf_P (char ∗ __ *s*, size_t __ *n*, const char ∗ __*fmt*, va_list *ap*)

Variant of **vsnprintf()**(p. 96) that uses a `fmt` string that resides in program memory.

6.19.3.40 int vsprintf (char ∗ __ *s*, const char ∗ __*fmt*, va_list *ap*)

Like **sprintf()**(p. 89) but takes a variable argument list for the arguments.

6.19.3.41 int vsprintf_P (char * __s, const char * __fmt, va_list ap)

Variant of **vsprintf()**(p. 96) that uses a `fmt` string that resides in program memory.

6.20 <stdlib.h>: General utilities

6.20.1 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.

Data Structures

- struct **div_t**
- struct **ldiv_t**

Non-standard (i.e. non-ISO C) functions.

- #define **RANDOM_MAX** 0x7FFFFFFF
- char * **ltoa** (long int __val, char *__s, int __radix)
- char * **utoa** (unsigned int __val, char *__s, int __radix)
- char * **ultoa** (unsigned long int __val, char *__s, int __radix)
- long **random** (void)
- void **srandom** (unsigned long __seed)
- long **random_r** (unsigned long *ctx)
- char * **itoa** (int __val, char *__s, int __radix)

Defines

- #define **RAND_MAX** 0x7FFF

Typedefs

- typedef int(* **__compar_fn_t**)(const void *, const void *)

Functions

- __inline__ void **abort** (void) __ATTR_NORETURN__
- int **abs** (int __i) __ATTR_CONST__
- long **labs** (long __i) __ATTR_CONST__
- void * **bsearch** (const void *__key, const void *__base, size_t __nmemb, size_t __size, int(*__compar)(const void *, const void *))
- **div_t div** (int __num, int __denom) __asm__("__divmodhi4") __ATTR_CONST__
- **ldiv_t ldiv** (long __num, long __denom) __asm__("__divmodsi4") __ATTR_CONST__
- void **qsort** (void *__base, size_t __nmemb, size_t __size, **__compar_fn_t** __compar)
- long **strtol** (const char *__nptr, char **__endptr, int __base)
- unsigned long **strtoul** (const char *__nptr, char **__endptr, int __base)
- __inline__ long **atol** (const char *__nptr) __ATTR_PURE__
- __inline__ int **atoi** (const char *__nptr) __ATTR_PURE__
- void **exit** (int __status) __ATTR_NORETURN__
- void * **malloc** (size_t __size) __ATTR_MALLOC__
- void **free** (void *__ptr)
- void * **calloc** (size_t __nele, size_t __size) __ATTR_MALLOC__
- void * **realloc** (void *__ptr, size_t __size) __ATTR_MALLOC__
- double **strtod** (const char *__nptr, char **__endptr)
- double **atof** (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**

6.20.2 Define Documentation

6.20.2.1 #define RAND_MAX 0x7FFF

Highest number that can be generated by **rand()**(p. 103).

6.20.2.2 #define RANDOM_MAX 0x7FFFFFFF

Highest number that can be generated by **random()**(p. 103).

6.20.3 Typedef Documentation

6.20.3.1 typedef int(∗ __compar_fn_t)(const void ∗, const void ∗)

Comparision function type for **qsort()**(p. 102), just for convenience.

6.20.4 Function Documentation

6.20.4.1 __inline__ void abort (void)

The **abort()**(p. 99) function causes abnormal program termination to occur. In the limited AVR environment, execution is effectively halted by entering an infinite loop.

6.20.4.2 int abs (int __*i*)

The **abs()**(p. 99) function computes the absolute value of the integer `i`.

Note:

The **abs()**(p. 99) and **labs()**(p. 101) functions are builtins of gcc.

6.20.4.3 double atof (const char ∗ __*nptr*)

The **atof()**(p. 99) function converts the initial portion of the string pointed to by `nptr` to double representation.

It is equivalent to calling

```
strtod(nptr, (char **)NULL);
```

6.20.4.4 int atoi (const char ∗ *string*)

Convert a string to an integer.

The **atoi()**(p. 99) function converts the initial portion of the string pointed to by `nptr` to integer representation.

It is equivalent to:

```
(int)strtol(nptr, (char **)NULL, 10);
```

except that **atoi()**(p. 99) does not detect errors.

6.20.4.5 long int atol (const char * *string*)

Convert a string to a long integer.

The **atol()**(p. 100) function converts the initial portion of the string pointed to by `stringp` to integer representation.

It is equivalent to:

```
strtol(nptr, (char **)NULL, 10);
```

except that **atol()**(p. 100) does not detect errors.

6.20.4.6 void* bsearch (const void * *__key*, const void * *__base*, size_t *__nmemb*, size_t *__size*, int(*)(const void *, const void *) *__compar*)

The **bsearch()**(p. 100) 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()**(p. 100) 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.

6.20.4.7 void* calloc (size_t *__nele*, size_t *__size*)

Allocate `nele` elements of `size` each. Identical to calling **malloc()**(p. 102) using `nele * size` as argument, except the allocated memory will be cleared to zero.

6.20.4.8 div_t div (int *__num*, int *__denom*)

The **div()**(p. 100) function computes the value `num/denom` and returns the quotient and remainder in a structure named **div_t**(p. 186) that contains two int members named `quot` and `rem`.

6.20.4.9 void exit (int *__status*)

The **exit()**(p. 100) 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.

In a C++ context, global destructors will be called before halting execution.

6.20.4.10 void free (void * __*ptr*)

The **free()**(p. 101) function causes the allocated memory referenced by `ptr` to be made available for future allocations. If `ptr` is NULL, no action occurs.

6.20.4.11 char* itoa (int __*val*, char * __*s*, int __*radix*)

Convert an integer to a string.

The function **itoa()**(p. 101) 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()**(p. 101) function returns the pointer passed as `s`.

6.20.4.12 long labs (long __*i*)

The **labs()**(p. 101) function computes the absolute value of the long integer `i`.

Note:

> The **abs()**(p. 99) and **labs()**(p. 101) functions are builtins of gcc.

6.20.4.13 ldiv_t ldiv (long __*num*, long __*denom*)

The **ldiv()**(p. 101) function computes the value `num/denom` and returns the quotient and remainder in a structure named **ldiv_t**(p. 186) that contains two long integer members named `quot` and `rem`.

6.20.4.14 char* ltoa (long int __val, char * __s, int __radix)

Convert a long integer to a string.

The function **ltoa()**(p. 102) 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()**(p. 102) function returns the pointer passed as `s`.

6.20.4.15 void* malloc (size_t __size)

The **malloc()**(p. 102) function allocates `size` bytes of memory. If **malloc()**(p. 102) fails, a NULL pointer is returned.

Note that **malloc()**(p. 102) does *not* initialize the returned memory to zero bytes.

See the chapter about **malloc() usage**(p. 230) for implementation details.

6.20.4.16 void qsort (void * __base, size_t __nmemb, size_t __size, __compar_fn_t __compar)

The **qsort()**(p. 102) function is a modified partition-exchange sort, or quicksort.

The **qsort()**(p. 102) 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.

6.20.4.17 int rand (void)

The **rand()**(p. 103) 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()**(p. 104) function sets its argument `seed` as the seed for a new sequence of pseudo-random numbers to be returned by **rand()**(p. 103). These sequences are repeatable by calling **srand()**(p. 104) with the same seed value.

If no seed value is provided, the functions are automatically seeded with a value of 1.

In compliance with the C standard, these functions operate on `int` arguments. Since the underlying algorithm already uses 32-bit calculations, this causes a loss of precision. See **random()**(p. 103) for an alternate set of functions that retains full 32-bit precision.

6.20.4.18 int rand_r (unsigned long ∗ *ctx*)

Variant of **rand()**(p. 103) that stores the context in the user-supplied variable located at `ctx` instead of a static library variable so the function becomes re-entrant.

6.20.4.19 long random (void)

The **random()**(p. 103) 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()**(p. 104) function sets its argument `seed` as the seed for a new sequence of pseudo-random numbers to be returned by **rand()**(p. 103). These sequences are repeatable by calling **srandom()**(p. 104) with the same seed value.

If no seed value is provided, the functions are automatically seeded with a value of 1.

6.20.4.20 long random_r (unsigned long ∗ *ctx*)

Variant of **random()**(p. 103) that stores the context in the user-supplied variable located at `ctx` instead of a static library variable so the function becomes re-entrant.

6.20.4.21 void∗ realloc (void ∗ *__ptr*, size_t *__size*)

The **realloc()**(p. 103) 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()**(p. 103) will behave identical to **malloc()**(p. 102).

If the new memory cannot be allocated, **realloc()**(p. 103) returns NULL, and the region at `ptr` will not be changed.

6.20.4.22 void srand (unsigned int __ *seed*)

Pseudo-random number generator seeding; see **rand()**(p. 103).

6.20.4.23 void srandom (unsigned long __ *seed*)

Pseudo-random number generator seeding; see **random()**(p. 103).

6.20.4.24 double strtod (const char * __ *nptr*, char ** __ *endptr*)

The **strtod()**(p. 104) function converts the initial portion of the string pointed to by `nptr` to double 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 **strtod()**(p. 104) 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 `HUGE_VAL` 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`.

FIXME: HUGE_VAL needs to be defined somewhere. The bit pattern is 0x7fffffff, but what number would this be?

6.20.4.25 long strtol (const char * __ *nptr*, char ** __ *endptr*, int __ *base*)

The **strtol()**(p. 104) 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()**(p. 46)) followed by a single optional `'+'` or `'-'` sign. If `base` is zero or 16, the string may then include a `"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).

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()**(p. 104) stores the address of the first invalid character in ∗`endptr`. If there were no digits at all, however, **strtol()**(p. 104) 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()**(p. 104) 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**(p. 47) and the function return value is clamped to `LONG_MIN` or `LONG_MAX`, respectively.

6.20.4.26 unsigned long strtoul (const char ∗ __ *nptr*, char ∗∗ __ *endptr*, int __ *base*)

The **strtoul()**(p. 105) 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()**(p. 46)) followed by a single optional `'+'` or `'-'` sign. If `base` is zero or 16, the string may then include a `"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).

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()**(p. 105) stores the address of the first invalid character in ∗`endptr`. If there were no digits at all, however, **strtoul()**(p. 105) 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()**(p. 105) function return 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, **str-**

toul()(p. 105) returns ULONG_MAX, and `errno` is set to **ERANGE**(p. 47). If no conversion could be performed, 0 is returned.

6.20.4.27 char∗ ultoa (unsigned long int __*val*, char ∗ __*s*, int __*radix*)

Convert an unsigned long integer to a string.

The function **ultoa()**(p. 106) 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()**(p. 106) function returns the pointer passed as `s`.

6.20.4.28 char∗ utoa (unsigned int __*val*, char ∗ __*s*, int __*radix*)

Convert an unsigned integer to a string.

The function **utoa()**(p. 106) 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()**(p. 106) function returns the pointer passed as `s`.

6.20.5 Variable Documentation

6.20.5.1 char* __malloc_heap_end

malloc()(p. 102) **tunable**(p. 232).

6.20.5.2 char* __malloc_heap_start

malloc()(p. 102) **tunable**(p. 232).

6.20.5.3 size_t __malloc_margin

malloc()(p. 102) **tunable**(p. 232).

6.21 <string.h>: Strings

6.21.1 Detailed Description

```
#include <string.h>
```

The string functions perform string operations on NULL terminated strings.

Note:

> If the strings you are working on resident in program space (flash), you will need to use the string functions described in **<avr/pgmspace.h>: Program Space String Utilities**(p. 19).

Defines

- #define **_FFS**(x)

Functions

- int **ffs** (int) __attribute__((const))
- int **ffsl** (long) __attribute__((const))
- int **ffsll** (long long) __attribute__((const))
- void * **memccpy** (void *, const void *, int, size_t)
- void * **memchr** (const void *, int, size_t) __ATTR_PURE__

- int **memcmp** (const void *, const void *, size_t) __ATTR_PURE__
- void * **memcpy** (void *, const void *, size_t)
- void * **memmove** (void *, const void *, size_t)
- void * **memset** (void *, int, size_t)
- int **strcasecmp** (const char *, const char *) __ATTR_PURE__
- char * **strcat** (char *, const char *)
- char * **strchr** (const char *, int) __ATTR_PURE__
- int **strcmp** (const char *, const char *) __ATTR_PURE__
- char * **strcpy** (char *, const char *)
- size_t **strlcat** (char *, const char *, size_t)
- size_t **strlcpy** (char *, const char *, size_t)
- size_t **strlen** (const char *) __ATTR_PURE__
- char * **strlwr** (char *)
- int **strncasecmp** (const char *, const char *, size_t) __ATTR_PURE__
- char * **strncat** (char *, const char *, size_t)
- int **strncmp** (const char *, const char *, size_t) __ATTR_PURE__
- char * **strncpy** (char *, const char *, size_t)
- size_t **strnlen** (const char *, size_t) __ATTR_PURE__
- char * **strrchr** (const char *, int) __ATTR_PURE__
- char * **strrev** (char *)
- char * **strsep** (char **, const char *)
- char * **strstr** (const char *, const char *) __ATTR_PURE__
- char * **strtok_r** (char *, const char *, char **)
- char * **strupr** (char *)

6.21.2 Define Documentation

6.21.2.1 #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()**(p. 109) 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()**(p. 108) 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.

6.21.3 Function Documentation

6.21.3.1 int ffs (int *val*) const

This function finds the first (least significant) bit set in the input value.

Returns:

The **ffs()**(p. 109) 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**(p. 108) macro instead.

6.21.3.2 int ffsl (long) const

Same as **ffs()**(p. 109), for an argument of type long.

6.21.3.3 int ffsll (long *long*) const

Same as **ffs()**(p. 109), for an argument of type long long.

6.21.3.4 void ∗ memccpy (void ∗ *dest*, const void ∗ *src*, int *val*, size_t *len*)

Copy memory area.

The **memccpy()**(p. 109) 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()**(p. 109) 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.

6.21.3.5 void ∗ memchr (const void ∗ *src*, int *val*, size_t *len*)

Scan memory for a character.

The **memchr()**(p. 109) 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()**(p. 109) function returns a pointer to the matching byte or NULL if the character does not occur in the given memory area.

6.21.3.6 int memcmp (const void * *s1*, const void * *s2*, size_t *len*)

Compare memory areas.

The **memcmp()**(p. 110) function compares the first len bytes of the memory areas s1 and s2. The comparision is performed using unsigned char operations.

Returns:

The **memcmp()**(p. 110) 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.

6.21.3.7 void * memcpy (void * *dest*, const void * *src*, size_t *len*)

Copy a memory area.

The **memcpy()**(p. 110) function copies len bytes from memory area src to memory area dest. The memory areas may not overlap. Use **memmove()**(p. 110) if the memory areas do overlap.

Returns:

The **memcpy()**(p. 110) function returns a pointer to dest.

6.21.3.8 void * memmove (void * *dest*, const void * *src*, size_t *len*)

Copy memory area.

The **memmove()**(p. 110) function copies len bytes from memory area src to memory area dest. The memory areas may overlap.

Returns:

The **memmove()**(p. 110) function returns a pointer to dest.

6.21.3.9 void ∗ memset (void ∗ *dest*, int *val*, size_t *len*)

Fill memory with a constant byte.

The **memset()**(p. 111) function fills the first len bytes of the memory area pointed to by dest with the constant byte val.

Returns:

The **memset()**(p. 111) function returns a pointer to the memory area dest.

6.21.3.10 int strcasecmp (const char ∗ *s1*, const char ∗ *s2*)

Compare two strings ignoring case.

The **strcasecmp()**(p. 111) function compares the two strings s1 and s2, ignoring the case of the characters.

Returns:

The **strcasecmp()**(p. 111) 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.

6.21.3.11 char ∗ strcat (char ∗ *dest*, const char ∗ *src*)

Concatenate two strings.

The **strcat()**(p. 111) 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()**(p. 111) function returns a pointer to the resulting string dest.

6.21.3.12 char ∗ strchr (const char ∗ *src*, int *val*)

Locate character in string.

The **strchr()**(p. 111) function returns a pointer to the first 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 **strchr()**(p. 111) function returns a pointer to the matched character or NULL if the character is not found.

6.21.3.13 int strcmp (const char ∗ *s1*, const char ∗ *s2*)

Compare two strings.

The **strcmp()**(p. 112) function compares the two strings s1 and s2.

Returns:

The **strcmp()**(p. 112) 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.

6.21.3.14 char ∗ strcpy (char ∗ *dest*, const char ∗ *src*)

Copy a string.

The **strcpy()**(p. 112) 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()**(p. 112) function returns a pointer to the destination string dest.

Note:

If the destination string of a **strcpy()**(p. 112) is not large enough (that is, if the programmer was stupid/lazy, and failed to check the size before copying) then anything might happen. Overflowing fixed length strings is a favourite cracker technique.

6.21.3.15 size_t strlcat (char ∗ *dst*, const char ∗ *src*, size_t *siz*)

Concatenate two strings.

Appends src to string dst of size siz (unlike **strncat()**(p. 114), 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. 112) function returns strlen(src) + MIN(siz, strlen(initial dst)). If retval >= siz, truncation occurred.

6.21.3.16 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 NULL terminates (unless siz == 0).

Returns:

The **strlcpy()**(p. 112) function returns strlen(src). If retval >= siz, truncation occurred.

6.21.3.17 size_t strlen (const char * *src*)

Calculate the length of a string.

The **strlen()**(p. 113) function calculates the length of the string src, not including the terminating '\0' character.

Returns:

The **strlen()**(p. 113) function returns the number of characters in src.

6.21.3.18 char * strlwr (char * *string*)

Convert a string to lower case.

The **strlwr()**(p. 113) 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()**(p. 113) function returns a pointer to the converted string.

6.21.3.19 int strncasecmp (const char * *s1*, const char * *s2*, size_t *len*)

Compare two strings ignoring case.

The **strncasecmp()**(p. 113) function is similar to **strcasecmp()**(p. 111), except it only compares the first n characters of s1.

Returns:

The **strncasecmp()**(p. 113) 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.

6.21.3.20 char * strncat (char * *dest*, const char * *src*, size_t *len*)

Concatenate two strings.

The **strncat()**(p. 114) function is similar to **strcat()**(p. 111), except that only the first n characters of src are appended to dest.

Returns:

The **strncat()**(p. 114) function returns a pointer to the resulting string dest.

6.21.3.21 int strncmp (const char * *s1*, const char * *s2*, size_t *len*)

Compare two strings.

The **strncmp()**(p. 114) function is similar to **strcmp()**(p. 112), except it only compares the first (at most) n characters of s1 and s2.

Returns:

The **strncmp()**(p. 114) 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.

6.21.3.22 char * strncpy (char * *dest*, const char * *src*, size_t *len*)

Copy a string.

The **strncpy()**(p. 114) function is similar to **strcpy()**(p. 112), 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()**(p. 114) function returns a pointer to the destination string dest.

6.21.3.23 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.

6.21.3.24 char * strrchr (const char * *src*, int *val*)

Locate character in string.

The **strrchr()**(p. 115) 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()**(p. 115) function returns a pointer to the matched character or NULL if the character is not found.

6.21.3.25 char * strrev (char * *string*)

Reverse a string.

The **strrev()**(p. 115) function reverses the order of the string.

Returns:

The **strrev()**(p. 115) function returns a pointer to the beginning of the reversed string.

6.21.3.26 char * strsep (char ** *string*, const char * *delim*)

Parse a string into tokens.

The **strsep()**(p. 115) function locates, in the string referenced by *string, 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 *string. 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 *string to '\0'.

Returns:

The **strtok_r()**(p. 116) function returns a pointer to the original value of *string. If *stringp is initially NULL, **strsep()**(p. 115) returns NULL.

6.21.3.27 char ∗ strstr (const char ∗ *s1*, const char ∗ *s2*)

Locate a substring.

The **strstr()**(p. 116) function finds the first occurrence of the substring `s2` in the string `s1`. The terminating '\0' characters are not compared.

Returns:

The **strstr()**(p. 116) 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.

6.21.3.28 char ∗ strtok_r (char ∗ *string*, const char ∗ *delim*, char ∗∗ *last*)

Parses the string s into tokens.

strtok_r parses the string s 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()**(p. 116) function returns a pointer to the next token or NULL when no more tokens are found.

6.21.3.29 char ∗ strupr (char ∗ *string*)

Convert a string to upper case.

The **strupr()**(p. 116) 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()**(p. 116) function returns a pointer to the converted string. The pointer is the same as that passed in since the operation is perform in place.

6.22 <util/crc16.h>: CRC Computations

6.22.1 Detailed Description

```
#include <util/crc16.h>
```

This header file provides a optimized inline functions for calculating cyclic redundancy checks (CRC) using common polynomials.

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 isue 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.

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
}
```

Functions

- static __inline__ **uint16_t _crc16_update** (**uint16_t** __crc, **uint8_t** __data)
- static __inline__ **uint16_t _crc_xmodem_update** (**uint16_t** __crc, **uint8_t** __data)
- static __inline__ **uint16_t _crc_ccitt_update** (**uint16_t** __crc, **uint8_t** __data)
- static __inline__ **uint8_t _crc_ibutton_update** (**uint8_t** __crc, **uint8_t** __data)

6.22.2 Function Documentation

6.22.2.1 static __inline__ uint16_t _crc16_update (uint16_t __crc, uint8_t __data) `[static]`

Optimized CRC-16 calculation.

Polynomial: x^16 + x^15 + x^2 + 1 (0xa001)

Initial value: 0xffff

This CRC is normally used in disk-drive controllers.

The following is the equivalent functionality written in C.

```
uint16_t
crc16_update(uint16_t crc, uint8_t a)
{
    int i;

    crc ^= a;
    for (i = 0; i < 8; ++i)
    {
        if (crc & 1)
            crc = (crc >> 1) ^ 0xA001;
        else
            crc = (crc >> 1);
    }

    return crc;
}
```

6.22.2.2 static __inline__ uint16_t _crc_ccitt_update (uint16_t __crc, uint8_t __data) `[static]`

Optimized CRC-CCITT calculation.

Polynomial: x^16 + x^12 + x^5 + 1 (0x8408)

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 alorgithm. 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.

```
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));
}
```

6.22.2.3 static __inline__ uint8_t _crc_ibutton_update (uint8_t __*crc*, uint8_t __*data*) `[static]`

Optimized Dallas (now Maxim) iButton 8-bit CRC calculation.

Polynomial: x^8 + x^5 + x^4 + 1 (0x8C)

Initial value: 0x0

See `http://www.maxim-ic.com/appnotes.cfm/appnote_number/27`

The following is the equivalent functionality written in C.

```
uint8_t
_crc_ibutton_update(uint8_t crc, uint8_t data)
{
    uint8_t i;

    crc = crc ^ data;
    for (i = 0; i < 8; i++)
    {
        if (crc & 0x01)
            crc = (crc >> 1) ^ 0x8C;
        else
            crc >>= 1;
    }

    return crc;
}
```

6.22.2.4 static __inline__ uint16_t _crc_xmodem_update (uint16_t __*crc*, uint8_t __*data*) `[static]`

Optimized CRC-XMODEM calculation.

Polynomial: x^16 + x^12 + x^5 + 1 (0x1021)

Initial value: 0x0

This is the CRC used by the Xmodem-CRC protocol.

The following is the equivalent functionality written in C.

```
uint16_t
crc_xmodem_update (uint16_t crc, uint8_t data)
{
    int i;

    crc = crc ^ ((uint16_t)data << 8);
    for (i=0; i<8; i++)
    {
        if (crc & 0x8000)
            crc = (crc << 1) ^ 0x1021;
        else
            crc <<= 1;
    }

    return crc;
}
```

6.23 <util/delay.h>: Busy-wait delay loops

6.23.1 Detailed Description

```
#define F_CPU 1000000UL  // 1 MHz
//#define F_CPU 14.7456E6
#include <util/delay.h>
```

Note:

As an alternative method, it is possible to pass the F_CPU macro down to the compiler from the Makefile. Obviously, in that case, no `#define` statement should be used.

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 preferrable, 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.

Two wrapper functions allow the specification of microsecond, and millisecond delays directly, using the application-supplied macro F_CPU as the CPU clock frequency (in Hertz). These functions operate on double typed arguments, however when optimization is turned on, the entire floating-point calculation will be done at compile-time.

Note:

When using **_delay_us()**(p. 121) and **_delay_ms()**(p. 121), the expressions passed as arguments to these functions shall be compile-time constants, otherwise the floating-point calculations to setup the loops will be done at run-time, thereby drastically increasing both the resulting code size, as well as the time required to setup the loops.

Functions

- void **_delay_loop_1** (**uint8_t** __count)
- void **_delay_loop_2** (**uint16_t** __count)
- void **_delay_us** (double __us)
- void **_delay_ms** (double __ms)

6.23.2 Function Documentation

6.23.2.1 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, at a CPU speed of 1 MHz, delays of up to 768 microseconds can be achieved.

6.23.2.2 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, at a CPU speed of 1 MHz, delays of up to about 262.1 milliseconds can be achieved.

6.23.2.3 void _delay_ms (double __ *ms*)

Perform a delay of `__ms` milliseconds, using **_delay_loop_2()**(p. 121).

The macro F_CPU is supposed to be defined to a constant defining the CPU clock frequency (in Hertz).

The maximal possible delay is 262.14 ms / F_CPU in MHz.

6.23.2.4 void _delay_us (double __ *us*)

Perform a delay of **__us** microseconds, using **_delay_loop_1()**(p. 121).

The macro F_CPU is supposed to be defined to a constant defining the CPU clock frequency (in Hertz).

The maximal possible delay is 768 us / F_CPU in MHz.

6.24 <util/parity.h>: Parity bit generation

6.24.1 Detailed Description

```
#include <util/parity.h>
```

This header file contains optimized assembler code to calculate the parity bit for a byte.

Defines

- #define **parity_even_bit**(val)

6.24.2 Define Documentation

6.24.2.1 #define parity_even_bit(val)

Value:

```
(__extension__({                                        \
        unsigned char __t;                              \
        __asm__ (                                       \
                "mov __tmp_reg__,%0" "\n\t"             \
                "swap %0" "\n\t"                        \
                "eor %0,__tmp_reg__" "\n\t"             \
                "mov __tmp_reg__,%0" "\n\t"             \
                "lsr %0" "\n\t"                         \
                "lsr %0" "\n\t"                         \
                "eor %0,__tmp_reg__"                    \
                : "=r" (__t)                            \
                : "0" ((unsigned char)(val))            \
                : "r0"                                  \
        );                                              \
        (((__t + 1) >> 1) & 1);                         \
 }))
```

Returns:

1 if `val` has an odd number of bits set.

6.25 <util/twi.h>: TWI bit mask definitions

6.25.1 Detailed Description

```
#include <util/twi.h>
```

This header file contains bit mask definitions for use with the AVR TWI interface.

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

6.25.2 Define Documentation

6.25.2.1 #define TW_BUS_ERROR 0x00

illegal start or stop condition

6.25.2.2 #define TW_MR_ARB_LOST 0x38

arbitration lost in SLA+R or NACK

6.25.2.3 #define TW_MR_DATA_ACK 0x50

data received, ACK returned

6.25.2.4 #define TW_MR_DATA_NACK 0x58

data received, NACK returned

6.25.2.5 #define TW_MR_SLA_ACK 0x40

SLA+R transmitted, ACK received

6.25.2.6 #define TW_MR_SLA_NACK 0x48

SLA+R transmitted, NACK received

6.25.2.7 #define TW_MT_ARB_LOST 0x38

arbitration lost in SLA+W or data

6.25.2.8 #define TW_MT_DATA_ACK 0x28

data transmitted, ACK received

6.25.2.9 #define TW_MT_DATA_NACK 0x30

data transmitted, NACK received

6.25.2.10 #define TW_MT_SLA_ACK 0x18

SLA+W transmitted, ACK received

6.25.2.11 #define TW_MT_SLA_NACK 0x20

SLA+W transmitted, NACK received

6.25.2.12 #define TW_NO_INFO 0xF8

no state information available

6.25.2.13 #define TW_READ 1

SLA+R address

6.25.2.14 #define TW_REP_START 0x10

repeated start condition transmitted

6.25.2.15 #define TW_SR_ARB_LOST_GCALL_ACK 0x78

arbitration lost in SLA+RW, general call received, ACK returned

6.25.2.16 #define TW_SR_ARB_LOST_SLA_ACK 0x68

arbitration lost in SLA+RW, SLA+W received, ACK returned

6.25.2.17 #define TW_SR_DATA_ACK 0x80

data received, ACK returned

6.25.2.18 #define TW_SR_DATA_NACK 0x88

data received, NACK returned

6.25.2.19 #define TW_SR_GCALL_ACK 0x70

general call received, ACK returned

6.25.2.20 #define TW_SR_GCALL_DATA_ACK 0x90

general call data received, ACK returned

6.25.2.21 #define TW_SR_GCALL_DATA_NACK 0x98

general call data received, NACK returned

6.25.2.22 #define TW_SR_SLA_ACK 0x60

SLA+W received, ACK returned

6.25.2.23 #define TW_SR_STOP 0xA0

stop or repeated start condition received while selected

6.25.2.24 #define TW_ST_ARB_LOST_SLA_ACK 0xB0

arbitration lost in SLA+RW, SLA+R received, ACK returned

6.25.2.25 #define TW_ST_DATA_ACK 0xB8

data transmitted, ACK received

6.25.2.26 #define TW_ST_DATA_NACK 0xC0

data transmitted, NACK received

6.25.2.27 #define TW_ST_LAST_DATA 0xC8

last data byte transmitted, ACK received

6.25.2.28 #define TW_ST_SLA_ACK 0xA8

SLA+R received, ACK returned

6.25.2.29 #define TW_START 0x08

start condition transmitted

6.25.2.30 #define TW_STATUS (TWSR & TW_STATUS_MASK)

TWSR, masked by TW_STATUS_MASK

6.25.2.31 #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.

6.25.2.32 #define TW_WRITE 0

SLA+W address

6.26 <avr/interrupt.h>: Interrupts

6.26.1 Detailed Description

Note:

> This discussion of interrupts was originally taken from Rich Neswold's document. See **Acknowledgments**(p. 187).

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. The extra code needed to do this is enabled by tagging the interrupt function with `__attribute__((signal))`.

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()(p. 143). This macro 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
}
```

Refer to the chapter explaining **assembler programming**(p. 192) for an explanation about interrupt routines written solely in assembler language.

Catch-all interrupt vector If an unexpected interrupt occurs (interrupt is enabled and 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 `__vector_default` which should be defined with **ISR()**(p. 143) as such.

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

ISR(__vector_default)
{
    // user code here
}
```

Nested interrupts The AVR hardware clears the global interrupt flag in SREG before entering an interrupt vector. 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 interrupt handler) 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 an SEI instruction right at the beginning of an interrupt handler by declaring the handler the following way:

```
void XXX_vect(void) __attribute__((interrupt));
void XXX_vect(void) {
  ...
}
```

where `XXX_vect` is the name of a valid interrupt vector for the MCU type in question, as explained below.

Chosing the vector: Interrupt vector names The interrupt is chosen by supplying one of the symbols in following table.

There are currently two different styles present for naming the vectors. One form 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.

Starting with avr-libc version 1.4.0, a second style of interrupt vector names has been added, where a short phrase for the vector description is followed by `_vect`. The short phrase matches the vector name as described in the datasheet of the respective device (and in Atmel's XML files), with spaces replaced by an underscore and other non-alphanumeric characters dropped. Using the suffix `_vect` is intented to improve portability to other C compilers available for the AVR that use a similar naming convention.

The historical naming style might become deprecated in a future release, so it is not recommended for new projects.

Note:

The **ISR()**(p. 143) macro cannot really spell-check the argument passed to them. Thus, by misspelling one of the names below in a call to **ISR()**(p. 143), 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 a **ISR()**(p. 143) function (i.e. one that after macro replacement does not start with "__vector_").

Vector name	Old vector name	Description	Applicable for device

Vector name	**Old vector name**	**Description**	**Applicable for device**
ADC_vect	SIG_ADC	ADC Conversion Complete	AT90S2333, AT90S4433, AT90S4434, AT90S8535, AT90PWM3, AT90PWM2, AT90CAN128, AT90CAN32, AT90CAN64, ATmega103, ATmega128, ATmega16, ATmega163, ATmega165, ATmega169, ATmega32, ATmega323, ATmega325, ATmega3250, ATmega329, ATmega3290, ATmega64, ATmega645, ATmega6450, ATmega649, ATmega6490, ATmega8, ATmega8535, ATmega168, ATmega48, ATmega88, ATmega640, ATmega1280, ATmega1281, ATmega324, ATmega164, ATmega644, ATtiny13, ATtiny15, ATtiny26, ATtiny24, ATtiny44, ATtiny84, ATtiny45, ATtiny25, ATtiny85, ATtiny261, ATtiny461, ATtiny861
ANALOG_COMP_0_vect	SIG_COMPARATOR0	Analog Comparator 0	AT90PWM3, AT90PWM2
ANALOG_COMP_1_vect	SIG_COMPARATOR1	Analog Comparator 1	AT90PWM3, AT90PWM2
ANALOG_COMP_2_vect	SIG_COMPARATOR2	Analog Comparator 2	AT90PWM3, AT90PWM2
ANALOG_COMP_vect	SIG_COMPARATOR	Analog Comparator	AT90CAN128, AT90CAN32, AT90CAN64, ATmega103, ATmega128, ATmega165, ATmega169, ATmega325, ATmega3250, ATmega329, ATmega3290, ATmega64, ATmega645, ATmega6450, ATmega649, ATmega6490, ATmega168, ATmega48, ATmega88, ATmega640, ATmega1280, ATmega1281, ATmega324, ATmega164, ATmega644
ANA_COMP_vect	SIG_COMPARATOR	Analog Comparator	AT90S1200, AT90S2313, AT90S2333, AT90S4414, AT90S4433, AT90S4434, AT90S8515, AT90S8535, ATmega16, ATmega161, ATmega162, ATmega163, ATmega32, ATmega323, ATmega8, ATmega8515, ATmega8535, ATtiny11, ATtiny12, ATtiny13, ATtiny15, ATtiny2313, ATtiny26, ATtiny28, ATtiny24, ATtiny44, ATtiny84, ATtiny45, ATtiny25, ATtiny85, ATtiny261, ATtiny461, ATtiny861
CANIT_vect	SIG_CAN_INTERRUPT1	CAN Transfer Complete or Error	AT90CAN128, AT90CAN32, AT90CAN64

Vector name	Old vector name	Description	Applicable for device
EEPROM_READY_vect	SIG_EEPROM_READY, SIG_EE_READY		ATtiny2313
EE_RDY_vect	SIG_EEPROM_READY	EEPROM Ready	AT90S2333, AT90S4433, AT90S4434, AT90S8535, ATmega16, ATmega161, ATmega162, ATmega163, ATmega32, ATmega323, ATmega8, ATmega8515, ATmega8535, ATtiny12, ATtiny13, ATtiny15, ATtiny26, ATtiny24, ATtiny44, ATtiny84, ATtiny45, ATtiny25, ATtiny85, ATtiny261, ATtiny461, ATtiny861
EE_READY_vect	SIG_EEPROM_READY	EEPROM Ready	AT90PWM3, AT90PWM2, AT90CAN128, AT90CAN32, AT90CAN64, ATmega103, ATmega128, ATmega165, ATmega169, ATmega325, ATmega3250, ATmega329, ATmega3290, ATmega406, ATmega64, ATmega645, ATmega6450, ATmega649, ATmega6490, ATmega168, ATmega48, ATmega88, ATmega640, ATmega1280, ATmega1281, ATmega324, ATmega164, ATmega644
INT0_vect	SIG_INTERRUPT0	External Interrupt 0	AT90S1200, AT90S2313, AT90S2323, AT90S2333, AT90S2343, AT90S4414, AT90S4433, AT90S4434, AT90S8515, AT90S8535, AT90PWM3, AT90PWM2, AT90CAN128, AT90CAN32, AT90CAN64, ATmega103, ATmega128, ATmega16, ATmega161, ATmega162, ATmega163, ATmega165, ATmega169, ATmega32, ATmega323, ATmega325, ATmega3250, ATmega329, ATmega3290, ATmega406, ATmega64, ATmega645, ATmega6450, ATmega649, ATmega6490, ATmega8, ATmega8515, ATmega8535, ATmega168, ATmega48, ATmega88, ATmega640, ATmega1280, ATmega1281, ATmega324, ATmega164, ATmega644, ATtiny11, ATtiny12, ATtiny13, ATtiny15, ATtiny22, ATtiny2313, ATtiny26, ATtiny28, ATtiny24, ATtiny44, ATtiny84, ATtiny45, ATtiny25, ATtiny85, ATtiny261, ATtiny461, ATtiny861

Vector name	**Old vector name**	**Description**	**Applicable for device**
INT1_vect	SIG_INTERRUPT1	External Interrupt Request 1	AT90S2313, AT90S2333, AT90S4414, AT90S4433, AT90S4434, AT90S8515, AT90S8535, AT90PWM3, AT90PWM2, AT90CAN128, AT90CAN32, AT90CAN64, ATmega103, ATmega128, ATmega16, ATmega161, ATmega162, ATmega163, ATmega32, ATmega323, ATmega406, ATmega64, ATmega8, ATmega8515, ATmega8535, ATmega168, ATmega48, ATmega88, ATmega640, ATmega1280, ATmega1281, ATmega324, ATmega164, ATmega644, ATtiny2313, ATtiny28, ATtiny261, ATtiny461, ATtiny861
INT2_vect	SIG_INTERRUPT2	External Interrupt Request 2	AT90PWM3, AT90PWM2, AT90CAN128, AT90CAN32, AT90CAN64, ATmega103, ATmega128, ATmega16, ATmega161, ATmega162, ATmega32, ATmega323, ATmega406, ATmega64, ATmega8515, ATmega8535, ATmega640, ATmega1280, ATmega1281, ATmega324, ATmega164, ATmega644
INT3_vect	SIG_INTERRUPT3	External Interrupt Request 3	AT90PWM3, AT90PWM2, AT90CAN128, AT90CAN32, AT90CAN64, ATmega103, ATmega128, ATmega406, ATmega64, ATmega640, ATmega1280, ATmega1281
INT4_vect	SIG_INTERRUPT4	External Interrupt Request 4	AT90CAN128, AT90CAN32, AT90CAN64, ATmega103, ATmega128, ATmega64, ATmega640, ATmega1280, ATmega1281
INT5_vect	SIG_INTERRUPT5	External Interrupt Request 5	AT90CAN128, AT90CAN32, AT90CAN64, ATmega103, ATmega128, ATmega64, ATmega640, ATmega1280, ATmega1281
INT6_vect	SIG_INTERRUPT6	External Interrupt Request 6	AT90CAN128, AT90CAN32, AT90CAN64, ATmega103, ATmega128, ATmega64, ATmega640, ATmega1280, ATmega1281
INT7_vect	SIG_INTERRUPT7	External Interrupt Request 7	AT90CAN128, AT90CAN32, AT90CAN64, ATmega103, ATmega128, ATmega64, ATmega640, ATmega1280, ATmega1281
IO_PINS_vect	SIG_PIN, SIG_PIN_CHANGE	External Interrupt Request 0	ATtiny11, ATtiny12, ATtiny15, ATtiny26
LCD_vect	SIG_LCD	LCD Start of Frame	ATmega169, ATmega329, ATmega3290, ATmega649, ATmega6490
LOWLEVEL_IO_PINS_vect	SIG_PIN	Low-level Input on Port B	ATtiny28

Vector name	Old vector name	Description	Applicable for device
OVRIT_vect	SIG_CAN_OVERFLOW1	CAN Timer Overrun	AT90CAN128, AT90CAN32, AT90CAN64
PCINT0_vect	SIG_PIN_CHANGE0	Pin Change Interrupt Request 0	ATmega162, ATmega165, ATmega169, ATmega325, ATmega3250, ATmega329, ATmega3290, ATmega406, ATmega645, ATmega6450, ATmega649, ATmega6490, ATmega168, ATmega48, ATmega88, ATmega640, ATmega1280, ATmega1281, ATmega324, ATmega164, ATmega644, ATtiny13, ATtiny24, ATtiny44, ATtiny84, ATtiny45, ATtiny25, ATtiny85
PCINT1_vect	SIG_PIN_CHANGE1	Pin Change Interrupt Request 1	ATmega162, ATmega165, ATmega169, ATmega325, ATmega3250, ATmega329, ATmega3290, ATmega406, ATmega645, ATmega6450, ATmega649, ATmega6490, ATmega168, ATmega48, ATmega88, ATmega640, ATmega1280, ATmega1281, ATmega324, ATmega164, ATmega644, ATtiny24, ATtiny44, ATtiny84
PCINT2_vect	SIG_PIN_CHANGE2	Pin Change Interrupt Request 2	ATmega3250, ATmega3290, ATmega6450, ATmega6490, ATmega168, ATmega48, ATmega88, ATmega640, ATmega1280, ATmega1281, ATmega324, ATmega164, ATmega644
PCINT3_vect	SIG_PIN_CHANGE3	Pin Change Interrupt Request 3	ATmega3250, ATmega3290, ATmega6450, ATmega6490, ATmega324, ATmega164, ATmega644
PCINT_vect	SIG_PIN_CHANGE, SIG_PCINT		ATtiny2313, ATtiny261, ATtiny461, ATtiny861
PSC0_CAPT_vect	SIG_PSC0_CAPTURE	PSC0 Capture Event	AT90PWM3, AT90PWM2
PSC0_EC_vect	SIG_PSC0_END_CYCLE	PSC0 End Cycle	AT90PWM3, AT90PWM2
PSC1_CAPT_vect	SIG_PSC1_CAPTURE	PSC1 Capture Event	AT90PWM3, AT90PWM2
PSC1_EC_vect	SIG_PSC1_END_CYCLE	PSC1 End Cycle	AT90PWM3, AT90PWM2
PSC2_CAPT_vect	SIG_PSC2_CAPTURE	PSC2 Capture Event	AT90PWM3, AT90PWM2

Vector name	**Old vector name**	**Description**	**Applicable for device**
PSC2_EC_vect	SIG_PSC2_END_CYCLE	PSC2 End Cycle	AT90PWM3, AT90PWM2
SPI_STC_vect	SIG_SPI	Serial Transfer Complete	AT90S2333, AT90S4414, AT90S4433, AT90S4434, AT90S8515, AT90S8535, AT90PWM3, AT90PWM2, AT90CAN128, AT90CAN32, AT90CAN64, ATmega103, ATmega128, ATmega16, ATmega161, ATmega162, ATmega163, ATmega165, ATmega169, ATmega32, ATmega323, ATmega325, ATmega3250, ATmega329, ATmega3290, ATmega64, ATmega645, ATmega6450, ATmega649, ATmega6490, ATmega8, ATmega8515, ATmega8535, ATmega168, ATmega48, ATmega88, ATmega640, ATmega1280, ATmega1281, ATmega324, ATmega164, ATmega644
SPM_RDY_vect	SIG_SPM_READY	Store Program Memory Ready	ATmega16, ATmega162, ATmega32, ATmega323, ATmega8, ATmega8515, ATmega8535
SPM_READY_vect	SIG_SPM_READY	Store Program Memory Read	AT90PWM3, AT90PWM2, AT90CAN128, AT90CAN32, AT90CAN64, ATmega128, ATmega165, ATmega169, ATmega325, ATmega3250, ATmega329, ATmega3290, ATmega406, ATmega64, ATmega645, ATmega6450, ATmega649, ATmega6490, ATmega168, ATmega48, ATmega88, ATmega640, ATmega1280, ATmega1281, ATmega324, ATmega164, ATmega644
TIM0_COMPA_vect	SIG_OUTPUT_COMPARE0A	Timer/Counter Compare Match A	ATtiny13, ATtiny24, ATtiny44, ATtiny84, ATtiny45, ATtiny25, ATtiny85
TIM0_COMPB_vect	SIG_OUTPUT_COMPARE0B	Timer/Counter Compare Match B	ATtiny13, ATtiny24, ATtiny44, ATtiny84, ATtiny45, ATtiny25, ATtiny85
TIM0_OVF_vect	SIG_OVERFLOW0	Timer/Counter0 Overflow	ATtiny13, ATtiny24, ATtiny44, ATtiny84, ATtiny45, ATtiny25, ATtiny85
TIM1_COMPA_vect	SIG_OUTPUT_COMPARE1A	Timer/Counter1 Compare Match A	ATtiny24, ATtiny44, ATtiny84, ATtiny45, ATtiny25, ATtiny85
TIM1_COMPB_vect	SIG_OUTPUT_COMPARE1B	Timer/Counter1 Compare Match B	ATtiny24, ATtiny44, ATtiny84, ATtiny45, ATtiny25, ATtiny85
TIM1_OVF_vect	SIG_OVERFLOW1	Timer/Counter1 Overflow	ATtiny24, ATtiny44, ATtiny84, ATtiny45, ATtiny25, ATtiny85
TIMER0_CAPT_vect	SIG_INPUT_CAPTURE0	ADC Conversion Complete	ATtiny261, ATtiny461, ATtiny861

Vector name	**Old vector name**	**Description**	**Applicable for device**
TIMER0_COMPA_vect	SIG_OUTPUT_COMPARE0A	TimerCounter0 Compare Match A	ATmega168, ATmega48, ATmega88, ATmega640, ATmega1280, ATmega1281, ATmega324, ATmega164, ATmega644, ATtiny2313, ATtiny261, ATtiny461, ATtiny861
TIMER0_COMPB_vect	SIG_OUTPUT_COMPARE0B, SIG_OUTPUT_COMPARE0_B	Timer Counter 0 Compare Match B	AT90PWM3, AT90PWM2, ATmega168, ATmega48, ATmega88, ATmega640, ATmega1280, ATmega1281, ATmega324, ATmega164, ATmega644, ATtiny2313, ATtiny261, ATtiny461, ATtiny861
TIMER0_COMP_A_vect	SIG_OUTPUT_COMPARE0A, SIG_OUTPUT_COMPARE0_A	Timer/Counter0 Compare Match A	AT90PWM3, AT90PWM2
TIMER0_COMP_vect	SIG_OUTPUT_COMPARE0	Timer/Counter0 Compare Match	AT90CAN128, AT90CAN32, AT90CAN64, ATmega103, ATmega128, ATmega16, ATmega161, ATmega162, ATmega165, ATmega169, ATmega32, ATmega323, ATmega325, ATmega3250, ATmega329, ATmega3290, ATmega64, ATmega645, ATmega6450, ATmega649, ATmega6490, ATmega8515, ATmega8535
TIMER0_OVF0_vect	SIG_OVERFLOW0	Timer/Counter0 Overflow	AT90S2313, AT90S2323, AT90S2343, ATtiny22, ATtiny26
TIMER0_OVF_vect	SIG_OVERFLOW0	Timer/Counter0 Overflow	AT90S1200, AT90S2333, AT90S4414, AT90S4433, AT90S4434, AT90S8515, AT90S8535, AT90PWM3, AT90PWM2, AT90CAN128, AT90CAN32, AT90CAN64, ATmega103, ATmega128, ATmega16, ATmega161, ATmega162, ATmega163, ATmega165, ATmega169, ATmega32, ATmega323, ATmega325, ATmega3250, ATmega329, ATmega3290, ATmega64, ATmega645, ATmega6450, ATmega649, ATmega6490, ATmega8, ATmega8515, ATmega8535, ATmega168, ATmega48, ATmega88, ATmega640, ATmega1280, ATmega1281, ATmega324, ATmega164, ATmega644, ATtiny11, ATtiny12, ATtiny15, ATtiny2313, ATtiny28, ATtiny261, ATtiny461, ATtiny861
TIMER1_CAPT1_vect	SIG_INPUT_CAPTURE1	Timer/Counter1 Capture Event	AT90S2313

Vector name	**Old vector name**	**Description**	**Applicable for device**
TIMER1_CAPT_vect	SIG_INPUT_CAPTURE1	Timer/Counter Capture Event	AT90S2333, AT90S4414, AT90S4433, AT90S4434, AT90S8515, AT90S8535, AT90PWM3, AT90PWM2, AT90CAN128, AT90CAN32, AT90CAN64, ATmega103, ATmega128, ATmega16, ATmega161, ATmega162, ATmega163, ATmega165, ATmega169, ATmega32, ATmega323, ATmega325, ATmega3250, ATmega329, ATmega3290, ATmega64, ATmega645, ATmega6450, ATmega649, ATmega6490, ATmega8, ATmega8515, ATmega8535, ATmega168, ATmega48, ATmega88, ATmega640, ATmega1280, ATmega1281, ATmega324, ATmega164, ATmega644, ATtiny2313, ATtiny24, ATtiny44, ATtiny84
TIMER1_CMPA_vect	SIG_OUTPUT_COMPARE1A	Timer/Counter1 Compare Match 1A	ATtiny26
TIMER1_CMPB_vect	SIG_OUTPUT_COMPARE1B	Timer/Counter1 Compare Match 1B	ATtiny26
TIMER1_COMP1_vect	SIG_OUTPUT_COMPARE1A	Timer/Counter1 Compare Match	AT90S2313
TIMER1_COMPA_vect	SIG_OUTPUT_COMPARE1A	Timer/Counter1 Compare Match A	AT90S4414, AT90S4434, AT90S8515, AT90S8535, AT90PWM3, AT90PWM2, AT90CAN128, AT90CAN32, AT90CAN64, ATmega103, ATmega128, ATmega16, ATmega161, ATmega162, ATmega163, ATmega165, ATmega169, ATmega32, ATmega323, ATmega325, ATmega3250, ATmega329, ATmega3290, ATmega64, ATmega645, ATmega6450, ATmega649, ATmega6490, ATmega8, ATmega8515, ATmega8535, ATmega168, ATmega48, ATmega88, ATmega640, ATmega1280, ATmega1281, ATmega324, ATmega164, ATmega644, ATtiny2313, ATtiny261, ATtiny461, ATtiny861

Vector name	Old vector name	Description	Applicable for device
TIMER1_COMPB_vect	SIG_OUTPUT_COMPARE1B	Timer/Counter1 Compare MatchB	AT90S4414, AT90S4434, AT90S8515, AT90S8535, AT90PWM3, AT90PWM2, AT90CAN128, AT90CAN32, AT90CAN64, ATmega103, ATmega128, ATmega16, ATmega161, ATmega162, ATmega163, ATmega165, ATmega169, ATmega32, ATmega323, ATmega325, ATmega3250, ATmega329, ATmega3290, ATmega64, ATmega645, ATmega6450, ATmega649, ATmega6490, ATmega8, ATmega8515, ATmega8535, ATmega168, ATmega48, ATmega88, ATmega640, ATmega1280, ATmega1281, ATmega324, ATmega164, ATmega644, ATtiny2313, ATtiny261, ATtiny461, ATtiny861
TIMER1_COMPC_vect	SIG_OUTPUT_COMPARE1C	Timer/Counter1 Compare Match C	AT90CAN128, AT90CAN32, AT90CAN64, ATmega128, ATmega64, ATmega640, ATmega1280, ATmega1281
TIMER1_COMPD_vect	SIG_OUTPUT_COMPARE0D	Timer/Counter1 Compare Match D	ATtiny261, ATtiny461, ATtiny861
TIMER1_COMP_vect	SIG_OUTPUT_COMPARE1A	Timer/Counter1 Compare Match	AT90S2333, AT90S4433, ATtiny15
TIMER1_OVF1_vect	SIG_OVERFLOW1	Timer/Counter1 Overflow	AT90S2313, ATtiny26
TIMER1_OVF_vect	SIG_OVERFLOW1	Timer/Counter1 Overflow	AT90S2333, AT90S4414, AT90S4433, AT90S4434, AT90S8515, AT90S8535, AT90PWM3, AT90PWM2, AT90CAN128, AT90CAN32, AT90CAN64, ATmega103, ATmega128, ATmega16, ATmega161, ATmega162, ATmega163, ATmega165, ATmega169, ATmega32, ATmega323, ATmega325, ATmega3250, ATmega329, ATmega3290, ATmega64, ATmega645, ATmega6450, ATmega649, ATmega6490, ATmega8, ATmega8515, ATmega8535, ATmega168, ATmega48, ATmega88, ATmega640, ATmega1280, ATmega1281, ATmega324, ATmega164, ATmega644, ATtiny15, ATtiny2313, ATtiny261, ATtiny461, ATtiny861
TIMER2_COMPA_vect	SIG_OUTPUT_COMPARE2A	Timer/Counter2 Compare Match A	ATmega168, ATmega48, ATmega88, ATmega640, ATmega1280, ATmega1281, ATmega324, ATmega164, ATmega644

Vector name	**Old vector name**	**Description**	**Applicable for device**
TIMER2_COMPB_vect	SIG_OUTPUT_COMPARE2B	Timer/Counter2 Compare Match A	ATmega168, ATmega48, ATmega88, ATmega640, ATmega1280, ATmega1281, ATmega324, ATmega164, ATmega644
TIMER2_COMP_vect	SIG_OUTPUT_COMPARE2	Timer/Counter2 Compare Match	AT90S4434, AT90S8535, AT90CAN128, AT90CAN32, AT90CAN64, ATmega103, ATmega128, ATmega16, ATmega161, ATmega162, ATmega163, ATmega165, ATmega169, ATmega32, ATmega323, ATmega325, ATmega3250, ATmega329, ATmega3290, ATmega64, ATmega645, ATmega6450, ATmega649, ATmega6490, ATmega8, ATmega8535
TIMER2_OVF_vect	SIG_OVERFLOW2	Timer/Counter2 Overflow	AT90S4434, AT90S8535, AT90CAN128, AT90CAN32, AT90CAN64, ATmega103, ATmega128, ATmega16, ATmega161, ATmega162, ATmega163, ATmega165, ATmega169, ATmega32, ATmega323, ATmega325, ATmega3250, ATmega329, ATmega3290, ATmega64, ATmega645, ATmega6450, ATmega649, ATmega6490, ATmega8, ATmega8535, ATmega168, ATmega48, ATmega88, ATmega640, ATmega1280, ATmega1281, ATmega324, ATmega164, ATmega644
TIMER3_CAPT_vect	SIG_INPUT_CAPTURE3	Timer/Counter3 Capture Event	AT90CAN128, AT90CAN32, AT90CAN64, ATmega128, ATmega162, ATmega64, ATmega640, ATmega1280, ATmega1281
TIMER3_COMPA_vect	SIG_OUTPUT_COMPARE3A	Timer/Counter3 Compare Match A	AT90CAN128, AT90CAN32, AT90CAN64, ATmega128, ATmega162, ATmega64, ATmega640, ATmega1280, ATmega1281
TIMER3_COMPB_vect	SIG_OUTPUT_COMPARE3B	Timer/Counter3 Compare Match B	AT90CAN128, AT90CAN32, AT90CAN64, ATmega128, ATmega162, ATmega64, ATmega640, ATmega1280, ATmega1281
TIMER3_COMPC_vect	SIG_OUTPUT_COMPARE3C	Timer/Counter3 Compare Match C	AT90CAN128, AT90CAN32, AT90CAN64, ATmega128, ATmega64, ATmega640, ATmega1280, ATmega1281
TIMER3_OVF_vect	SIG_OVERFLOW3	Timer/Counter3 Overflow	AT90CAN128, AT90CAN32, AT90CAN64, ATmega128, ATmega162, ATmega64, ATmega640, ATmega1280, ATmega1281
TIMER4_CAPT_vect	SIG_INPUT_CAPTURE4	Timer/Counter4 Capture Event	ATmega640, ATmega1280, ATmega1281
TIMER4_COMPA_vect	SIG_OUTPUT_COMPARE4A	Timer/Counter4 Compare Match A	ATmega640, ATmega1280, ATmega1281

Vector name	Old vector name	Description	Applicable for device
TIMER4_COMPB_vect	SIG_OUTPUT_COMPARE4B	Timer/Counter4 Compare Match B	ATmega640, ATmega1280, ATmega1281
TIMER4_COMPC_vect	SIG_OUTPUT_COMPARE4C	Timer/Counter4 Compare Match C	ATmega640, ATmega1280, ATmega1281
TIMER4_OVF_vect	SIG_OVERFLOW4	Timer/Counter4 Overflow	ATmega640, ATmega1280, ATmega1281
TIMER5_CAPT_vect	SIG_INPUT_CAPTURE5	Timer/Counter5 Capture Event	ATmega640, ATmega1280, ATmega1281
TIMER5_COMPA_vect	SIG_OUTPUT_COMPARE5A	Timer/Counter5 Compare Match A	ATmega640, ATmega1280, ATmega1281
TIMER5_COMPB_vect	SIG_OUTPUT_COMPARE5B	Timer/Counter5 Compare Match B	ATmega640, ATmega1280, ATmega1281
TIMER5_COMPC_vect	SIG_OUTPUT_COMPARE5C	Timer/Counter5 Compare Match C	ATmega640, ATmega1280, ATmega1281
TIMER5_OVF_vect	SIG_OVERFLOW5	Timer/Counter5 Overflow	ATmega640, ATmega1280, ATmega1281
TWI_vect	SIG_2WIRE_SERIAL	2-wire Serial Interface	AT90CAN128, AT90CAN32, AT90CAN64, ATmega128, ATmega16, ATmega163, ATmega32, ATmega323, ATmega406, ATmega64, ATmega8, ATmega8535, ATmega168, ATmega48, ATmega88, ATmega640, ATmega1280, ATmega1281, ATmega324, ATmega164, ATmega644
TXDONE_vect	SIG_TXDONE	Transmission Done, Bit Timer Flag 2 Interrupt	AT86RF401
TXEMPTY_vect	SIG_TXBE	Transmit Buffer Empty, Bit Itmer Flag 0 Interrupt	AT86RF401
UART0_RX_vect	SIG_UART0_RECV	UART0, Rx Complete	ATmega161
UART0_TX_vect	SIG_UART0_TRANS	UART0, Tx Complete	ATmega161
UART0_UDRE_vect	SIG_UART0_DATA	UART0 Data Register Empty	ATmega161
UART1_RX_vect	SIG_UART1_RECV	UART1, Rx Complete	ATmega161
UART1_TX_vect	SIG_UART1_TRANS	UART1, Tx Complete	ATmega161

Vector name	Old vector name	Description	Applicable for device
UART1_UDRE_vect	SIG_UART1_DATA	UART1 Data Register Empty	ATmega161
UART_RX_vect	SIG_UART_RECV	UART, Rx Complete	AT90S2313, AT90S2333, AT90S4414, AT90S4433, AT90S4434, AT90S8515, AT90S8535, ATmega103, ATmega163, ATmega8515
UART_TX_vect	SIG_UART_TRANS	UART, Tx Complete	AT90S2313, AT90S2333, AT90S4414, AT90S4433, AT90S4434, AT90S8515, AT90S8535, ATmega103, ATmega163, ATmega8515
UART_UDRE_vect	SIG_UART_DATA	UART Data Register Empty	AT90S2313, AT90S2333, AT90S4414, AT90S4433, AT90S4434, AT90S8515, AT90S8535, ATmega103, ATmega163, ATmega8515
USART0_RXC_vect	SIG_USART0_RECV	USART0, Rx Complete	ATmega162
USART0_RX_vect	SIG_UART0_RECV	USART0, Rx Complete	AT90CAN128, AT90CAN32, AT90CAN64, ATmega128, ATmega165, ATmega169, ATmega325, ATmega329, ATmega64, ATmega645, ATmega649, ATmega640, ATmega1280, ATmega1281, ATmega324, ATmega164, ATmega644
USART0_TXC_vect	SIG_USART0_TRANS	USART0, Tx Complete	ATmega162
USART0_TX_vect	SIG_UART0_TRANS	USART0, Tx Complete	AT90CAN128, AT90CAN32, AT90CAN64, ATmega128, ATmega165, ATmega169, ATmega325, ATmega3250, ATmega329, ATmega3290, ATmega64, ATmega645, ATmega6450, ATmega649, ATmega6490, ATmega640, ATmega1280, ATmega1281, ATmega324, ATmega164, ATmega644
USART0_UDRE_vect	SIG_UART0_DATA	USART0 Data Register Empty	AT90CAN128, AT90CAN32, AT90CAN64, ATmega128, ATmega162, ATmega165, ATmega169, ATmega325, ATmega329, ATmega64, ATmega645, ATmega649, ATmega640, ATmega1280, ATmega1281, ATmega324, ATmega164, ATmega644
USART1_RXC_vect	SIG_USART1_RECV	USART1, Rx Complete	ATmega162
USART1_RX_vect	SIG_UART1_RECV	USART1, Rx Complete	AT90CAN128, AT90CAN32, AT90CAN64, ATmega128, ATmega64, ATmega640, ATmega1280, ATmega1281, ATmega324, ATmega164, ATmega644
USART1_TXC_vect	SIG_USART1_TRANS	USART1, Tx Complete	ATmega162

Vector name	**Old vector name**	**Description**	**Applicable for device**
USART1_TX_vect	SIG_UART1_TRANS	USART1, Tx Complete	AT90CAN128, AT90CAN32, AT90CAN64, ATmega128, ATmega64, ATmega640, ATmega1280, ATmega1281, ATmega324, ATmega164, ATmega644
USART1_UDRE_vect	SIG_UART1_DATA	USART1, Data Register Empty	AT90CAN128, AT90CAN32, AT90CAN64, ATmega128, ATmega162, ATmega64, ATmega640, ATmega1280, ATmega1281, ATmega324, ATmega164, ATmega644
USART2_RX_vect	SIG_USART2_RECV	USART2, Rx Complete	ATmega640, ATmega1280, ATmega1281
USART2_TX_vect	SIG_USART2_TRANS	USART2, Tx Complete	ATmega640, ATmega1280, ATmega1281
USART2_UDRE_vect	SIG_USART2_DATA	USART2 Data register Empty	ATmega640, ATmega1280, ATmega1281
USART3_RX_vect	SIG_USART3_RECV	USART3, Rx Complete	ATmega640, ATmega1280, ATmega1281
USART3_TX_vect	SIG_USART3_TRANS	USART3, Tx Complete	ATmega640, ATmega1280, ATmega1281
USART3_UDRE_vect	SIG_USART3_DATA	USART3 Data register Empty	ATmega640, ATmega1280, ATmega1281
USART_RXC_vect	SIG_USART_RECV, SIG_UART_RECV	USART, Rx Complete	ATmega16, ATmega32, ATmega323, ATmega8
USART_RX_vect	SIG_USART_RECV, SIG_UART_RECV	USART, Rx Complete	AT90PWM3, AT90PWM2, ATmega3250, ATmega3290, ATmega6450, ATmega6490, ATmega8535, ATmega168, ATmega48, ATmega88, ATtiny2313
USART_TXC_vect	SIG_USART_TRANS, SIG_UART_TRANS	USART, Tx Complete	ATmega16, ATmega32, ATmega323, ATmega8
USART_TX_vect	SIG_USART_TRANS, SIG_UART_TRANS	USART, Tx Complete	AT90PWM3, AT90PWM2, ATmega8535, ATmega168, ATmega48, ATmega88, ATtiny2313

Vector name	**Old vector name**	**Description**	**Applicable for device**
USART_UDRE_vect	SIG_USART_DATA, SIG_UART_DATA	USART Data Register Empty	AT90PWM3, AT90PWM2, ATmega16, ATmega32, ATmega323, ATmega3250, ATmega3290, ATmega6450, ATmega6490, ATmega8, ATmega8535, ATmega168, ATmega48, ATmega88, ATtiny2313
USI_OVERFLOW_vect	SIG_USI_OVERFLOW	USI Overflow	ATmega165, ATmega169, ATmega325, ATmega3250, ATmega329, ATmega3290, ATmega645, ATmega6450, ATmega649, ATmega6490, ATtiny2313
USI_OVF_vect	SIG_USI_OVERFLOW	USI Overflow	ATtiny26, ATtiny24, ATtiny44, ATtiny84, ATtiny45, ATtiny25, ATtiny85, ATtiny261, ATtiny461, ATtiny861
USI_START_vect	SIG_USI_START	USI Start Condition	ATmega165, ATmega169, ATmega325, ATmega3250, ATmega329, ATmega3290, ATmega645, ATmega6450, ATmega649, ATmega6490, ATtiny2313, ATtiny24, ATtiny44, ATtiny84, ATtiny45, ATtiny25, ATtiny85, ATtiny261, ATtiny461, ATtiny861
USI_STRT_vect	SIG_USI_START	USI Start	ATtiny26
WDT_OVERFLOW_vect	SIG_WATCHDOG_TIMEOUT, SIG_WDT_OVERFLOW	Watchdog Timer Overflow	ATtiny2313
WDT_vect	SIG_WDT, SIG_WATCHDOG_TIMEOUT	Watchdog Timeout Interrupt	AT90PWM3, AT90PWM2, ATmega406, ATmega168, ATmega48, ATmega88, ATmega640, ATmega1280, ATmega1281, ATmega324, ATmega164, ATmega644, ATtiny13, ATtiny24, ATtiny44, ATtiny84, ATtiny45, ATtiny25, ATtiny85, ATtiny261, ATtiny461, ATtiny861

Macros for writing interrupt handler functions

- #define **ISR**(vector)
- #define **SIGNAL**(vector)
- #define **EMPTY_INTERRUPT**(vector)
- #define **ISR_ALIAS**(vector, target_vector)

6.26.2 Define Documentation

6.26.2.1 #define EMPTY_INTERRUPT(vector)

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

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);
```

6.26.2.2 #define ISR(vector)

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

Introduces an interrupt handler function (interrupt service routine) that runs with global interrupts initially disabled.

`vector` must be one of the interrupt vector names that are valid for the particular MCU type.

6.26.2.3 #define ISR_ALIAS(vector, target_vector)

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

Defines `vector` to point to the same interrupt vector as `target_vector`. That way, a single interrupt vector implementation can be used to serve several interrupt sources.

Do not define a function body.

Note:

This requires a recent version of AVR-GCC to work (GCC 4.2 or patched GCC 4.1.x).

Example:

```
ISR(INT0_vect)
{
    PORTB = 42;
}

ISR_ALIAS(INT1_vect, INT0_vect);
```

6.26.2.4 #define SIGNAL(vector)

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

Introduces an interrupt handler function that runs with global interrupts initially disabled.

This is the same as the ISR macro.

Deprecated

Do not use anymore in new code.

6.27 <avr/sfr_defs.h>: Special function registers

6.27.1 Detailed Description

When working with microcontrollers, many of the tasks usually consist of controlling the peripherals that are connected to the device, respectively programming the subsystems that are contained in the controller (which by itself communicate with the circuitry connected to the controller).

The AVR series of microcontrollers offers two different paradigms to perform this task. There's a separate IO address space available (as it is known from some high-level CISC CPUs) that can be addressed with specific IO instructions that are applicable to some or all of the IO address space (`in`, `out`, `sbi` etc.). The entire IO address space is also made available as *memory-mapped IO*, i. e. it can be accessed using all the MCU instructions that are applicable to normal data memory. 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 either address 0x60, or 0x100 depending on the device.)

AVR Libc supports both these paradigms. While by default, the implementation uses memory-mapped IO access, this is hidden from the programmer. So the programmer can access IO registers either with a special function like **outb()**(p. 43):

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

outb(PORTA, 0x33);
```

or they can assign a value directly to the symbolic address:

```
PORTA = 0x33;
```

The compiler's choice of which method to use when actually accessing the IO port is completely independent of the way the programmer chooses to write the code. So even if the programmer uses the memory-mapped paradigm and writes

```
PORTA |= 0x40;
```

the compiler can optimize this into the use of an `sbi` instruction (of course, provided the target address is within the allowable range for this instruction, and the right-hand side of the expression is a constant value known at compile-time).

The advantage of using the memory-mapped paradigm in C programs is that it makes the programs more portable to other C compilers for the AVR platform. Some people might also feel that this is more readable. For example, the following two statements would be equivalent:

```
outb(DDRD, inb(DDRD) & ~LCDBITS);
DDRD &= ~LCDBITS;
```

The generated code is identical for both. Without optimization, the compiler strictly generates code following the memory-mapped paradigm, while with optimization turned on, code is generated using the (faster and smaller) `in/out` MCU instructions.

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?**(p. 200).

Porting programs that use sbi/cbi

As described above, access to the AVR single bit set and clear instructions are provided via the standard C bit manipulation commands. The sbi and cbi commands are no longer directly supported. sbi (sfr,bit) can be replaced by sfr |= **_BV(bit)**(p. 146) .

ie: **sbi(PORTB, PB1)**(p. 43); is now PORTB |= **_BV(PB1)**(p. 146);

This actually is more flexible than having sbi directly, as the optimizer will use a hardware sbi if appropriate, or a read/or/write if not. You do not need to keep track of which registers sbi/cbi will operate on.

Likewise, cbi (sfr,bit) is now sfr &= ~(**_BV(bit)**(p. 146));

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))

6.27.2 Define Documentation

6.27.2.1 #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()**(p. 146).

6.27.2.2 #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.

6.27.2.3 #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.

6.27.2.4 #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.

6.27.2.5 #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.

6.28 Demo projects

6.28.1 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**(p. 151) 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**(p. 165) 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**(p. 173) intends to explain that complex topic, using a practical microcontroller peripheral setup with one RS-232 connection, and an HD44780-compatible industry-standard LCD display.

The **Example using the two-wire interface (TWI)**(p. 181) 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**(p. 148) 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.

While the simple demo is meant to run on about any AVR setup possible where a LED could be connected to the OCR1[A] output, the **large**(p. 165) and **stdio**(p. 173) demos are mainly targeted to the Atmel STK500 starter kit, and the **TWI**(p. 181) example requires a controller where some 24Cxx two-wire EEPPROM can be connected to. For the STK500 demos, the default CPU (either an AT90S8515 or an ATmega8515) should be removed from its socket, and the ATmega16 that ships with the kit should be inserted into socket SCKT3100A3. The ATmega16 offers an on-board ADC that is used in the **large**(p. 165) demo, and all AVRs with an ADC feature a different pinout than the industry-standard compatible devices.

In order to fully utilize the **large**(p. 165) demo, a female 10-pin header with cable, connecting to a 10 kOhm potentiometer will be useful.

For the **stdio**(p. 173) demo, an industry-standard HD44780-compatible LCD display of at least 16x1 characters will be needed. Among other things, the `LCD4Linux` project page describes many things around these displays, including common pinouts.

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)**

6.29 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 avaible.

6.29.1 Hardware setup

The incoming PWM pulse train is fed into PB4. It will generate a pin change interrupt there on eache 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.

6.29.2 A code walkthrough

6.29.2.1 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.

6.29.2.2 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?**(p. 196)). 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.

6.29.2.3 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 (`/* ...`

6.30 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.

6.30.1 The Project

This project will use the pulse-width modulator (`PWM`) to ramp an LED on and off every two seconds. An AT90S2313 processor will be used as the controller. The circuit for this demonstration is shown in the **schematic diagram**(p. 151). If you have a development kit, you should be able to use it, rather than build the circuit, for this project.

Note:

> Meanwhile, the AT90S2313 became obsolete. Either use its successor, the (pin-compatible) ATtiny2313 for the project, or perhaps the ATmega8 or one of its successors (ATmega48/88/168) which have become quite popular since the original demo project had been established. For all these more modern devices, it is no longer necessary to use an external crystal for clocking as they ship with the internal 1 MHz oscillator enabled, so C1, C2, and Q1 can be omitted. Normally, for this experiment, the external circuitry on /RESET (R1, C3) can be omitted as well, leaving only the AVR, the LED, the bypass capacitor C4, and perhaps R2. For the ATmega8/48/88/168, use PB1 (pin 15 at the DIP-28 package) to connect the LED to. Additionally, this demo has been ported to many different other AVRs. The location of the respective OC pin varies between different AVRs, and it is mandated by the AVR hardware.

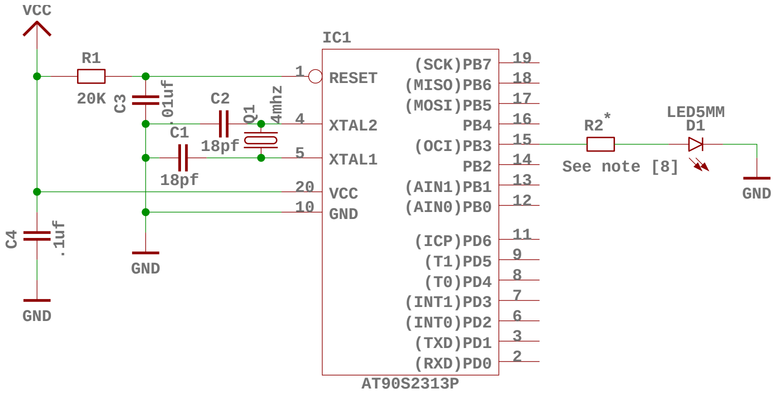

Figure 1: Schematic of circuit for demo project

The source code is given in **demo.c**(p. 153). 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]:

As the AVR microcontroller series has been developed during the past years, new features have been added over time. Even though the basic concepts of the timer/counter1 are still the same as they used to be back in early 2001 when this simple demo was written initially, the names of registers and bits have been changed slightly to reflect the new features. Also, the port and pin mapping of the output compare match 1A (or 1 for older devices) pin which is used to control the LED varies between different AVRs. The file `iocompat.h` tries to abstract between all this differences using some preprocessor `ifdef` statements, so the actual program itself can operate on a common set of symbolic names. The macros defined by that file are:

- `OCR` the name of the OCR register used to control the PWM (usually either OCR1 or OCR1A)
- `DDROC` the name of the DDR (data direction register) for the OC output
- `OC1` the pin number of the OC1[A] output within its port
- `TIMER1_TOP` the TOP value of the timer used for the PWM (1023 for 10-bit PWMs, 255 for devices that can only handle an 8-bit PWM)
- `TIMER1_PWM_INIT` the initialization bits to be set into control register 1A in order to setup 10-bit (or 8-bit) phase and frequency correct PWM mode
- `TIMER1_CLOCKSOURCE` the clock bits to set in the respective control register to start the PWM timer; usually the timer runs at full CPU clock for 10-bit PWMs, while it runs on a prescaled clock for 8-bit PWMs

Note [2]:

ISR()(p. 143) is a macro that marks the function as an interrupt routine. In this case, the function will get called when timer 1 overflows. Setting up interrupts is explained in greater detail in **<avr/interrupt.h>: Interrupts**(p. 127).

Note [3]:

The `PWM` is being used in 10-bit mode, so we need a 16-bit variable to remember the current value.

Note [4]:

This section determines the new value of the `PWM`.

Note [5]:

Here's where the newly computed value is loaded into the `PWM` 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**(p. 200).

Note [6]:

This routine gets called after a reset. It initializes the `PWM` and enables interrupts.

Note [7]:

The main loop of the program does nothing – all the work is done by the interrupt routine! The **sleep_mode()**(p. 35) puts the processor on sleep until the next interrupt, to conserve power. Of course, that probably won't be noticable as we are still driving a LED, it is merely mentioned here to demonstrate the basic principle.

Note [8]:

Early AVR devices saturate their outputs at rather low currents when sourcing current, so the LED can be connected directly, the resulting current through the LED will be about 15 mA. For modern parts (at least for the ATmega 128), however Atmel has drastically increased the IO source capability, so when operating at 5 V Vcc, R2 is needed. Its value should be about 150 Ohms. When operating the circuit at 3 V, it can still be omitted though.

6.30.2 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 OC1/OC1A 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.
 *
 * $Id: demo.c,v 1.6.2.3 2006/01/05 21:33:08 joerg_wunsch Exp $
 */

#include <inttypes.h>
#include <avr/io.h>
#include <avr/interrupt.h>
#include <avr/sleep.h>

#include "iocompat.h"           /* Note [1] */

enum { UP, DOWN };

ISR (TIMER1_OVF_vect)           /* Note [2] */
{
    static uint16_t pwm;        /* Note [3] */
    static uint8_t direction;

    switch (direction)          /* Note [4] */
    {
        case UP:
            if (++pwm == TIMER1_TOP)
                direction = DOWN;
            break;

        case DOWN:
            if (--pwm == 0)
                direction = UP;
            break;
    }

    OCR = pwm;                  /* Note [5] */
}

void
ioinit (void)                   /* Note [6] */
{
    /* Timer 1 is 10-bit PWM (8-bit PWM on some ATtinys). */
    TCCR1A = TIMER1_PWM_INIT;
    /*
     * Start timer 1.
     *
     * NB: TCCR1A and TCCR1B could actually be the same register, so
     * take care to not clobber it.
     */
    TCCR1B |= TIMER1_CLOCKSOURCE;
    /*
```

```
     * Run any device-dependent timer 1 setup hook if present.
     */
#if defined(TIMER1_SETUP_HOOK)
    TIMER1_SETUP_HOOK();
#endif

    /* Set PWM value to 0. */
    OCR = 0;

    /* Enable OC1 as output. */
    DDROC = _BV (OC1);

    /* Enable timer 1 overflow interrupt. */
    TIMSK = _BV (TOIE1);
    sei ();
}

int
main (void)
{

    ioinit ();

    /* loop forever, the interrupts are doing the rest */

    for (;;)                    /* Note [7] */
        sleep_mode();

    return (0);
}
```

6.30.3 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=atmega8 -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=atmega8 -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 8515 processor environment, which is most certainly what you didn't want.

6.30.4 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 sections (the .stab and .stabstr sections hold the debugging information and won't make it into the ROM file).

An even more useful option is `-S`. This option disassembles the binary file and intersperses the source code in the output! This method is much better, in my opinion, than using the `-S` with the compiler because this listing includes routines 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 .text         000000de  00000000  00000000  00000074  2**1
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .bss          00000003  00800060  00800060  00000152  2**0
                  ALLOC
  2 .stab         000007a4  00000000  00000000  00000154  2**2
                  CONTENTS, READONLY, DEBUGGING
  3 .stabstr      000006b8  00000000  00000000  000008f8  2**0
                  CONTENTS, READONLY, DEBUGGING
Disassembly of section .text:

00000000 <__vectors>:
   0: 12 c0       rjmp .+36      ; 0x26 <__ctors_end>
   2: 6c c0       rjmp .+216     ; 0xdc <__bad_interrupt>
   4: 6b c0       rjmp .+214     ; 0xdc <__bad_interrupt>
   6: 6a c0       rjmp .+212     ; 0xdc <__bad_interrupt>
   8: 69 c0       rjmp .+210     ; 0xdc <__bad_interrupt>
   a: 68 c0       rjmp .+208     ; 0xdc <__bad_interrupt>
```

```
   c: 67 c0        rjmp .+206     ; 0xdc <__bad_interrupt>
   e: 66 c0        rjmp .+204     ; 0xdc <__bad_interrupt>
  10: 11 c0        rjmp .+34      ; 0x34 <__vector_8>
  12: 64 c0        rjmp .+200     ; 0xdc <__bad_interrupt>
  14: 63 c0        rjmp .+198     ; 0xdc <__bad_interrupt>
  16: 62 c0        rjmp .+196     ; 0xdc <__bad_interrupt>
  18: 61 c0        rjmp .+194     ; 0xdc <__bad_interrupt>
  1a: 60 c0        rjmp .+192     ; 0xdc <__bad_interrupt>
  1c: 5f c0        rjmp .+190     ; 0xdc <__bad_interrupt>
  1e: 5e c0        rjmp .+188     ; 0xdc <__bad_interrupt>
  20: 5d c0        rjmp .+186     ; 0xdc <__bad_interrupt>
  22: 5c c0        rjmp .+184     ; 0xdc <__bad_interrupt>
  24: 5b c0        rjmp .+182     ; 0xdc <__bad_interrupt>

00000026 <__ctors_end>:
  26: 11 24        eor r1, r1
  28: 1f be        out 0x3f, r1 ; 63
  2a: cf e5        ldi r28, 0x5F ; 95
  2c: d4 e0        ldi r29, 0x04 ; 4
  2e: de bf        out 0x3e, r29 ; 62
  30: cd bf        out 0x3d, r28 ; 61
  32: 4b c0        rjmp .+150     ; 0xca <main>

00000034 <__vector_8>:

enum { UP, DOWN };

ISR (TIMER1_OVF_vect) /* Note [2] */
{
  34: 1f 92        push r1
  36: 0f 92        push r0
  38: 0f b6        in r0, 0x3f ; 63
  3a: 0f 92        push r0
  3c: 11 24        eor r1, r1
  3e: 2f 93        push r18
  40: 3f 93        push r19
  42: 8f 93        push r24
    static uint16_t pwm; /* Note [3] */
    static uint8_t direction;

    switch (direction) /* Note [4] */
  44: 80 91 60 00  lds r24, 0x0060
  48: 88 23        and r24, r24
  4a: 39 f0        breq .+14      ; 0x5a <__SREG__+0x1b>
  4c: 81 30        cpi r24, 0x01 ; 1
  4e: b9 f0        breq .+46      ; 0x7e <__SREG__+0x3f>
  50: 20 91 61 00  lds r18, 0x0061
  54: 30 91 62 00  lds r19, 0x0062
  58: 21 c0        rjmp .+66      ; 0x9c <__SREG__+0x5d>
    {
        case UP:
            if (++pwm == TIMER1_TOP)
  5a: 20 91 61 00  lds r18, 0x0061
  5e: 30 91 62 00  lds r19, 0x0062
  62: 2f 5f        subi r18, 0xFF ; 255
  64: 3f 4f        sbci r19, 0xFF ; 255
  66: 30 93 62 00  sts 0x0062, r19
```

```
  6a: 20 93 61 00  sts 0x0061, r18
  6e: 83 e0        ldi r24, 0x03 ; 3
  70: 2f 3f        cpi r18, 0xFF ; 255
  72: 38 07        cpc r19, r24
  74: 99 f4        brne .+38      ; 0x9c <__SREG__+0x5d>
                direction = DOWN;
  76: 81 e0        ldi r24, 0x01 ; 1
  78: 80 93 60 00  sts 0x0060, r24
  7c: 0f c0        rjmp .+30      ; 0x9c <__SREG__+0x5d>
            break;

        case DOWN:
            if (--pwm == 0)
  7e: 20 91 61 00  lds r18, 0x0061
  82: 30 91 62 00  lds r19, 0x0062
  86: 21 50        subi r18, 0x01 ; 1
  88: 30 40        sbci r19, 0x00 ; 0
  8a: 30 93 62 00  sts 0x0062, r19
  8e: 20 93 61 00  sts 0x0061, r18
  92: 21 15        cp r18, r1
  94: 31 05        cpc r19, r1
  96: 11 f4        brne .+4       ; 0x9c <__SREG__+0x5d>
                direction = UP;
  98: 10 92 60 00  sts 0x0060, r1
            break;
    }

    OCR = pwm; /* Note [5] */
  9c: 3b bd        out 0x2b, r19 ; 43
  9e: 2a bd        out 0x2a, r18 ; 42
  a0: 8f 91        pop r24
  a2: 3f 91        pop r19
  a4: 2f 91        pop r18
  a6: 0f 90        pop r0
  a8: 0f be        out 0x3f, r0 ; 63
  aa: 0f 90        pop r0
  ac: 1f 90        pop r1
  ae: 18 95        reti

000000b0 <ioinit>:
}

void
ioinit (void) /* Note [6] */
{
    /* Timer 1 is 10-bit PWM (8-bit PWM on some ATtinys). */
    TCCR1A = TIMER1_PWM_INIT;
  b0: 83 e8        ldi r24, 0x83 ; 131
  b2: 8f bd        out 0x2f, r24 ; 47
    /*
     * Start timer 1.
     *
     * NB: TCCR1A and TCCR1B could actually be the same register, so
     * take care to not clobber it.
     */
    TCCR1B |= TIMER1_CLOCKSOURCE;
  b4: 8e b5        in r24, 0x2e ; 46
```

```
  b6: 81 60        ori r24, 0x01 ; 1
  b8: 8e bd        out 0x2e, r24 ; 46
    /*
     * Run any device-dependent timer 1 setup hook if present.
     */
#if defined(TIMER1_SETUP_HOOK)
    TIMER1_SETUP_HOOK();
#endif

    /* Set PWM value to 0. */
    OCR = 0;
  ba: 1b bc        out 0x2b, r1 ; 43
  bc: 1a bc        out 0x2a, r1 ; 42

    /* Enable OC1 as output. */
    DDROC = _BV (OC1);
  be: 82 e0        ldi r24, 0x02 ; 2
  c0: 87 bb        out 0x17, r24 ; 23

    /* Enable timer 1 overflow interrupt. */
    TIMSK = _BV (TOIE1);
  c2: 84 e0        ldi r24, 0x04 ; 4
  c4: 89 bf        out 0x39, r24 ; 57
    sei ();
  c6: 78 94        sei
  c8: 08 95        ret

000000ca <main>:
}

int
main (void)
{

    ioinit ();
  ca: f2 df        rcall .-28      ; 0xb0 <ioinit>

    /* loop forever, the interrupts are doing the rest */

    for (;;) /* Note [7] */
        sleep_mode();
  cc: 85 b7        in r24, 0x35 ; 53
  ce: 80 68        ori r24, 0x80 ; 128
  d0: 85 bf        out 0x35, r24 ; 53
  d2: 88 95        sleep
  d4: 85 b7        in r24, 0x35 ; 53
  d6: 8f 77        andi r24, 0x7F ; 127
  d8: 85 bf        out 0x35, r24 ; 53
  da: f8 cf        rjmp .-16      ; 0xcc <main+0x2>

000000dc <__bad_interrupt>:
  dc: 91 cf        rjmp .-222     ; 0x0 <__heap_end>
```

6.30.5 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).

```
$ avr-gcc -g -mmcu=atmega8 -Wl,-Map,demo.map -o demo.elf demo.o
```

Some points of interest in the `demo.map` file are:

```
.rela.plt
 *(.rela.plt)

.text           0x00000000       0xde
 *(.vectors)
 .vectors       0x00000000       0x26 /junk/AVR/avr-libc-1.4/avr/lib/avr4/atmega8/crtm8.o
                0x00000000                __vectors
                0x00000000                __vector_default
 *(.vectors)
 *(.progmem.gcc*)
 *(.progmem*)
                0x00000026                . = ALIGN (0x2)
                0x00000026                __trampolines_start = .
 *(.trampolines)
 .trampolines   0x00000026        0x0 linker stubs
 *(.trampolines*)
                0x00000026                __trampolines_end = .
 *(.jumptables)
 *(.jumptables*)
 *(.lowtext)
 *(.lowtext*)
                0x00000026                __ctors_start = .
```

The .text segment (where program instructions are stored) starts at location 0x0.

```
 *(.fini2)
 *(.fini2)
 *(.fini1)
 *(.fini1)
 *(.fini0)
 *(.fini0)
                0x000000de                _etext = .

.data           0x00800060        0x0 load address 0x000000de
                0x00800060                PROVIDE (__data_start, .)
 *(.data)
 .data          0x00800060        0x0 demo.o
 .data          0x00800060        0x0 /junk/AVR/avr-libc-1.4/avr/lib/avr4/atmega8/crtm8.o
 *(.data*)
```

```
 *(.rodata)
 *(.rodata*)
 *(.gnu.linkonce.d*)
                0x00800060                . = ALIGN (0x2)
                0x00800060                _edata = .
                0x00800060                PROVIDE (__data_end, .)

.bss            0x00800060        0x3
                0x00800060                PROVIDE (__bss_start, .)
 *(.bss)
 .bss           0x00800060        0x3 demo.o
 .bss           0x00800063        0x0 /junk/AVR/avr-libc-1.4/avr/lib/avr4/atmega8/crtm8.o
 *(.bss*)
 *(COMMON)
                0x00800063                PROVIDE (__bss_end, .)
                0x000000de                __data_load_start = LOADADDR (.data)
                0x000000de                __data_load_end = (__data_load_start + SIZEOF (.data))

.noinit         0x00800063        0x0
                0x00800063                PROVIDE (__noinit_start, .)
 *(.noinit*)
                0x00800063                PROVIDE (__noinit_end, .)
                0x00800063                _end = .
                0x00800063                PROVIDE (__heap_start, .)

.eeprom         0x00810000        0x0
 *(.eeprom*)
                0x00810000                __eeprom_end = .
```

The last address in the .text segment is location `0x114` (denoted by `_etext`), so the instructions use up 276 bytes of FLASH.

The .data segment (where initialized static variables are stored) starts at location `0x60`, which is the first address after the register bank on an ATmega8 processor.

The next available address in the .data segment is also location `0x60`, so the application has no initialized data.

The .bss segment (where uninitialized data is stored) starts at location `0x60`.

The next available address in the .bss segment is location 0x63, so the application uses 3 bytes of uninitialized data.

The .eeprom segment (where EEPROM variables are stored) starts at location 0x0.

The next available address in the .eeprom segment is also location 0x0, so there aren't any EEPROM variables.

6.30.6 Generating Intel Hex Files

We have a binary of the application, but how do we get it into the processor? Most (if not all) programmers will not accept a GNU executable as an input file, so we need to do a little more processing. The next step is to extract portions of the binary and save the information into .hex files. 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 resulting `demo.hex` file contains:

```
:1000000012C06CC06BC06AC069C068C067C066C0FF
:1000100011C064C063C062C061C060C05FC05EC028
:100020005DC05CC05BC011241FBECFE5D4E0DEBF65
:10003000CDBF4BC01F920F920FB60F9211242F937A
:100040003F938F9380916000882339F08130B9F01D
:100050002091610030916200 21C0209161003091B7
:1000600062002F5F3F4F30936200209361008 3E076
:100070002F3F380799F481E0809360000FC02091F2
:10008000610030916200215030403093620020 9333
:1000900061002115310511F4109260003BBD2ABDAD
:1000A0008F913F912F910F900FBE0F901F90189539
:1000B00083E88FBD8EB581608EBD1BBC1ABC82E00B
:1000C00087BB84E089BF78940895F2DF85B78068A4
:0E00D00085BF889585B78F7785BFF8CF91CF14
:00000001FF
```

The `-j` option indicates that we want the information from the .text and .data segment extracted. 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
```

The resulting `demo_eeprom.hex` file contains:

which is an empty .hex file (which is expected, since we didn't define any EEPROM variables). Starting with version 2.17 of the GNU binutils, the `avr-objcopy` command that used to generate the empty EEPROM files now aborts because of the empty input section .eeprom, so these empty files are not generated. It also signals an error to the Makefile which will be caught there, and makes it print a message about the empty file not being generated.

6.30.7 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 `make`, save the following in a file called `Makefile`.

Note:

This `Makefile` can only be used as input for the GNU version of `make`.

```
PRG            = demo
OBJ            = demo.o
#MCU_TARGET     = at90s2313
#MCU_TARGET     = at90s2333
#MCU_TARGET     = at90s4414
#MCU_TARGET     = at90s4433
#MCU_TARGET     = at90s4434
#MCU_TARGET     = at90s8515
#MCU_TARGET     = at90s8535
#MCU_TARGET     = atmega128
#MCU_TARGET     = atmega1280
#MCU_TARGET     = atmega1281
#MCU_TARGET     = atmega16
#MCU_TARGET     = atmega163
#MCU_TARGET     = atmega164p
#MCU_TARGET     = atmega165
#MCU_TARGET     = atmega165p
#MCU_TARGET     = atmega168
#MCU_TARGET     = atmega169
#MCU_TARGET     = atmega169p
#MCU_TARGET     = atmega32
#MCU_TARGET     = atmega324p
#MCU_TARGET     = atmega325
#MCU_TARGET     = atmega3250
#MCU_TARGET     = atmega329
#MCU_TARGET     = atmega3290
#MCU_TARGET     = atmega48
#MCU_TARGET     = atmega64
#MCU_TARGET     = atmega640
#MCU_TARGET     = atmega644
#MCU_TARGET     = atmega644p
#MCU_TARGET     = atmega645
#MCU_TARGET     = atmega6450
#MCU_TARGET     = atmega649
#MCU_TARGET     = atmega6490
MCU_TARGET     = atmega8
#MCU_TARGET     = atmega8515
#MCU_TARGET     = atmega8535
#MCU_TARGET     = atmega88
#MCU_TARGET     = attiny2313
#MCU_TARGET     = attiny24
#MCU_TARGET     = attiny25
#MCU_TARGET     = attiny26
#MCU_TARGET     = attiny261
#MCU_TARGET     = attiny44
#MCU_TARGET     = attiny45
#MCU_TARGET     = attiny461
#MCU_TARGET     = attiny84
#MCU_TARGET     = attiny85
#MCU_TARGET     = attiny861
OPTIMIZE       = -O2

DEFS           =
LIBS           =
```

```
# You should not have to change anything below here.

CC             = avr-gcc

# Override is only needed by avr-lib build system.

override CFLAGS        = -g -Wall $(OPTIMIZE) -mmcu=$(MCU_TARGET) $(DEFS)
override LDFLAGS       = -Wl,-Map,$(PRG).map

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 iocompat.h

clean:
	rm -rf *.o $(PRG).elf *.eps *.png *.pdf *.bak
	rm -rf *.lst *.map $(EXTRA_CLEAN_FILES)

lst:  $(PRG).lst

%.lst: %.elf
	$(OBJDUMP) -h -S $< > $@

# 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 $< $@

# 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; }

# Every thing below here is used by avr-libc's build system and can be ignored
# by the casual user.

FIG2DEV                 = fig2dev
EXTRA_CLEAN_FILES       = *.hex *.bin *.srec

dox: eps png pdf

eps: $(PRG).eps
png: $(PRG).png
pdf: $(PRG).pdf

%.eps: %.fig
        $(FIG2DEV) -L eps $< $@

%.pdf: %.fig
        $(FIG2DEV) -L pdf $< $@

%.png: %.fig
        $(FIG2DEV) -L png $< $@
```

6.30.8 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`.

6.31 A more sophisticated project

This project extends the basic idea of the **simple project**(p. 151) 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.

6.31.1 Hardware setup

The demo is set up in a way so it can be run on the ATmega16 that ships with the STK500 development kit. The only external part needed is a potentiometer attached to the ADC. It is connected to a 10-pin ribbon cable for port A, both ends of the potentiometer to pins 9 (GND) and 10 (VCC), and the wiper to pin 1 (port A0). A bypass capacitor from pin 1 to pin 9 (like 47 nF) is recommendable.

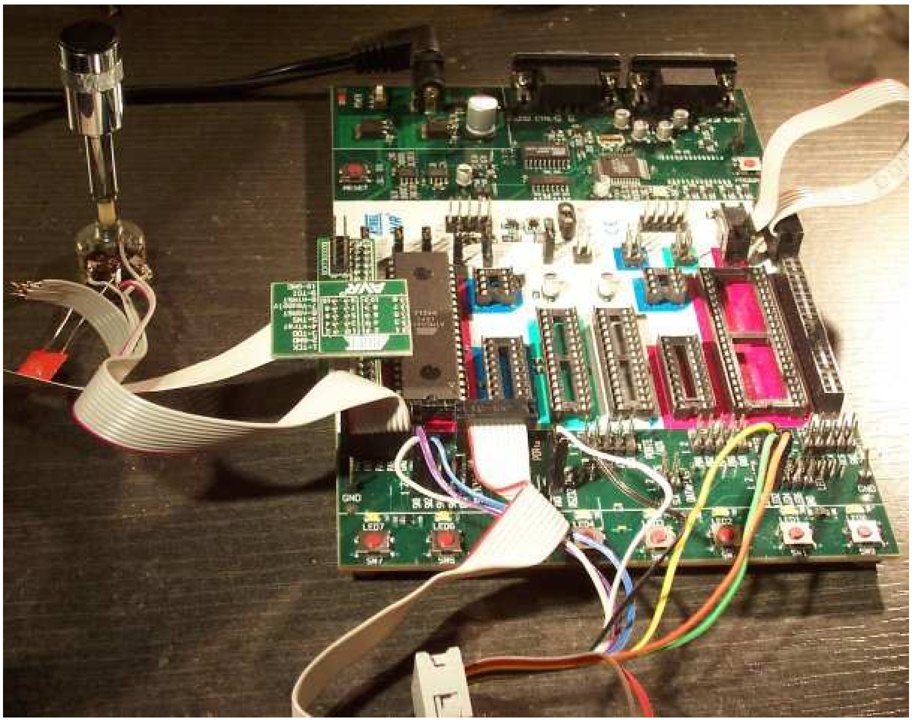

Figure 2: Setup of the STK500

The coloured patch cables are used to provide various interconnections. As there are only four of them in the STK500, there are two options to connect them for this demo. The second option for the yellow-green cable is shown in parenthesis in the table. Alternatively, the "squid" cable from the JTAG ICE kit can be used if available.

Port	**Header**	**Color**	**Function**	**Connect to**
D0	1	brown	RxD	RXD of the RS-232 header
D1	2	grey	TxD	TXD of the RS-232 header
D2	3	black	button "down"	SW0 (pin 1 switches header)
D3	4	red	button "up"	SW1 (pin 2 switches header)
D4	5	green	button "ADC"	SW2 (pin 3 switches header)
D5	6	blue	LED	LED0 (pin 1 LEDs header)
D6	7	(green)	clock out	LED1 (pin 2 LEDs header)
D7	8	white	1-second flash	LED2 (pin 3 LEDs header)
GND	9		unused	
VCC	10		unused	

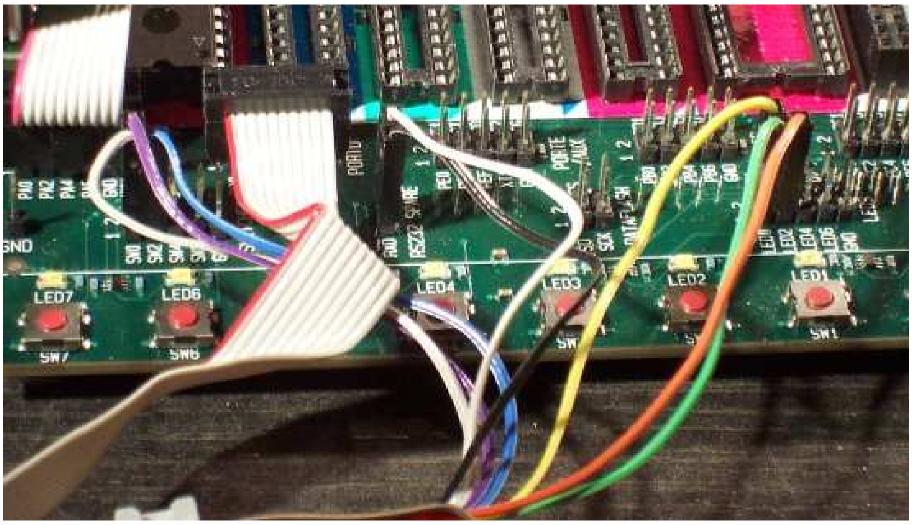

Figure 3: Wiring of the STK500

The following picture shows the alternate wiring where LED1 is connected but SW2 is not:

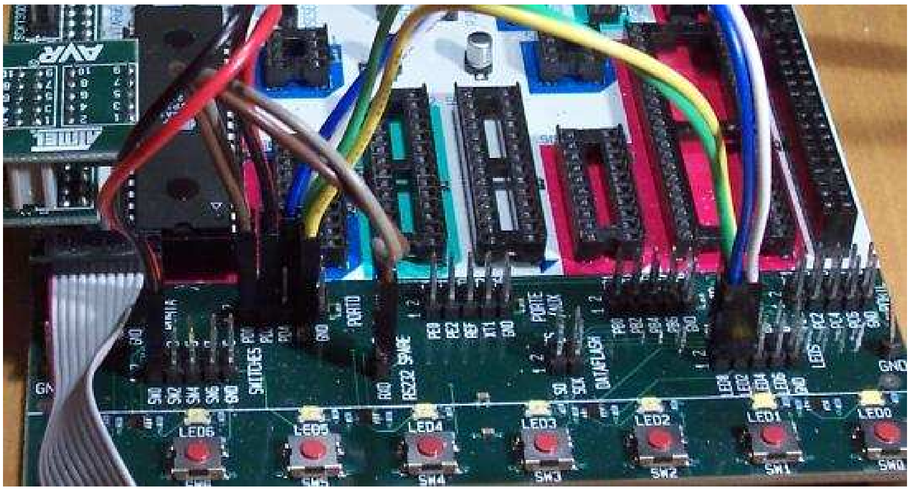

Figure 4: Wiring option #2 of the STK500

As an alternative, this demo can also be run on the popular ATmega8 controller, or its successor ATmega88 as well as the ATmega48 and ATmega168 variants of the latter. These controllers do not have a port named "A", so their ADC inputs are located on port C instead, thus the potentiometer needs to be attached to port C. Likewise, the OC1A output is not on port D pin 5 but on port B pin 1 (PB1). Thus, the above cabling scheme needs to be changed so that PB1 connects to the LED0 pin. (PD6 remains unconnected.) When using the STK500, use one of the jumper cables for this connection. All other port D pins should be connected the same way as described for the ATmega16 above.

When not using an STK500 starter kit, attach the LEDs through some resistor to Vcc (low-active LEDs), and attach pushbuttons from the respective input pins to GND. The internal pull-up resistors are enabled for the pushbutton pins, so no external resistors are needed.

The `MCU_TARGET` macro in the Makefile needs to be adjusted appropriately.

The flash ROM and RAM consumption of this demo are way below the resources of even an ATmega48. The major advantage of experimenting with the ATmega16 (in addition that it ships together with an STK500 anyway) is that it can be debugged online via JTAG.

Note that in the explanation below, all port/pin names are applicable to the ATmega16 setup.

6.31.2 Functional overview

PD6 will be toggled with each internal clock tick (approx. 10 ms). PD7 will flash 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.

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. VAREF on the STK500 should be set to the same value as VCC.

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 immmediately but only after a timeout, EEPROM wear is reduced considerably compared to immediately writing the value at each change.

6.31.3 A code walkthrough

This section explains the ideas behind individual parts of the code. The **source code**(p. 173) has been divided into numbered parts, and the following subsections explain each of these parts.

6.31.3.1 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 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 the internal calibrated 1 MHz RC oscillator that is enabled by default. 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.

6.31.3.2 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**(p. 196).

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

```
__attribute__((section(".eeprom")))
```

marks it as belonging to the **EEPROM section**(p. 240). 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**(p. 240) section in order to prevent it from being cleared upon application startup.

6.31.3.3 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).

6.31.3.4 Part 4: Auxiliary functions

The function `handle_mcucsr()` uses two `__attribute__` declarators to achieve specific goals. First, it will instruct the compiler to place the generated code into the **.init3**(p. 241) 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. 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.

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.

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()**(p. 18).

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**(p. 19). Both functions translate a newline character into a carriage return/newline sequence, so a simple `\n` can be used in the source code.

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 noticable anyway.

6.31.3.5 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**(p. 19) so no SRAM needs to be allocated for them. This is done by using the PSTR macro, and passing the string to `printstr_p()`.

6.31.4 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`.

6.32 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**(p. 77), not to replace it.

6.32.1 Hardware setup

The demo is set up in a way so it can be run on the ATmega16 that ships with the STK500 development kit. The UART port needs to be connected to the RS-232 "spare" port by a jumper cable that connects PD0 to RxD and PD1 to TxD. The RS-232 channel is set up 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 port A of the STK500 in the following way:

Port	Header	Function
A0	1	LCD D4
A1	2	LCD D5
A2	3	LCD D6
A3	4	LCD D7
A4	5	LCD R/∼W
A5	6	LCD E
A6	7	LCD RS
A7	8	unused
GND	9	GND
VCC	10	Vcc

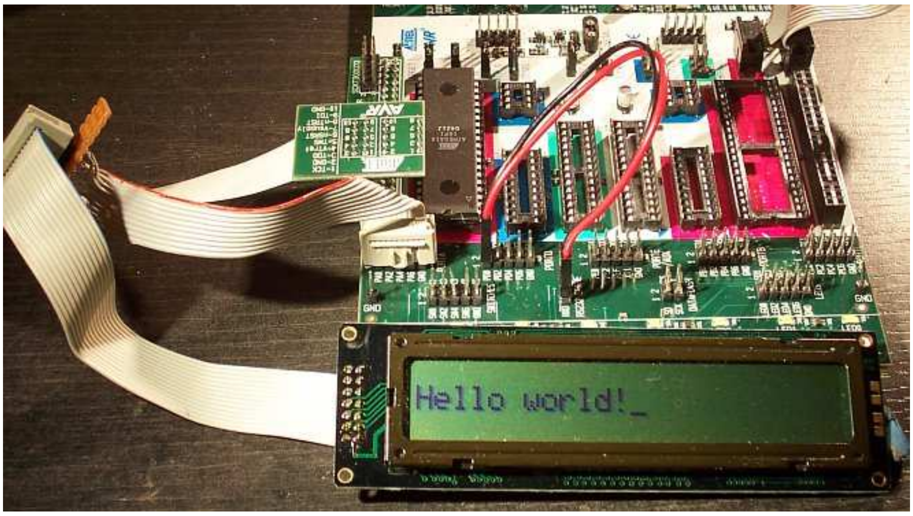

Figure 5: Wiring of the STK500

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.

Port A has been chosen as 7 pins on a single port are needed to connect the LCD, yet all other ports are already partially in use: port B has the pins for in-system programming (ISP), port C has the ports for JTAG (can be used for debugging), and port D is used for the UART connection.

6.32.2 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

6.32.3 A code walkthrough

6.32.3.1 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()**(p. 83) 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 RS-232 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()**(p. 121) function from `<util/delay.h>` which in turn needs the `F_CPU` macro in order to adjust the cycle counts. As the **_delay_ms()**(p. 121) 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 accomodate 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()**(p. 88) instead of **fprintf()**(p. 87)), 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 RS-232 connection, and performs a few simple actions based on the commands.

First, a prompt is sent out using **printf_P()**(p. 89) (which takes a **program space string**(p. 19)). The string is read into an internal buffer as one line of input, using **fgets()**(p. 87). While it would be also possible to use **gets()**(p. 88) (which implicitly reads from `stdin`), **gets()**(p. 88) 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()**(p. 87) 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()**(p. 87) 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 three "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.

Command recognition is done using **sscanf()**(p. 90) where the first format in the format string just skips over the command itself (as the assignment suppression modifier $*$ is given).

6.32.3.2 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 RS-232 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.

6.32.3.3 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.

6.32.3.4 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**(p. 177).

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).

6.32.3.5 lcd.h

This function declares the public interface of the higher-level (character IO) LCD driver.

6.32.3.6 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()**(p. 86), **FDEV_SETUP_STREAM()**(p. 83) 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()**(p. 83), and can be accessed inside `lcd_putchar()` using **fdev_get_udata()**(p. 83).

6.32.3.7 uart.h

Public interface definition for the RS-232 UART driver, much like in **lcd.h**(p. 178) except there is now also a character input function available.

As the RS-232 input is line-buffered in this example, the macro `RX_BUFSIZE` determines the size of that buffer.

6.32.3.8 uart.c

This implements an stdio-compatible RS-232 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.)

6.32.4 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`.

6.33 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 I2C by Philips, but that term is avoided in Atmel's documentation due to patenting issues.

For the original Philips documentation, see

`http://www.semiconductors.philips.com/buses/i2c/index.html`

6.33.1 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) transfered 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.)

6.33.2 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.

6.33.3 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 [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 intented 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 transfered within a single request, maintaining an internal address counter that is updated after each data byte transfered 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 transfering 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 transfering 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.

7 avr-libc Data Structure Documentation

7.1 div_t Struct Reference

7.1.1 Detailed Description

Result type for function **div()**(p. 100).

Data Fields

- int **quot**
- int **rem**

7.1.2 Field Documentation

7.1.2.1 int div_t::quot

The Quotient.

7.1.2.2 int div_t::rem

The Remainder.

The documentation for this struct was generated from the following file:

- stdlib.h

7.2 ldiv_t Struct Reference

7.2.1 Detailed Description

Result type for function **ldiv()**(p. 101).

Data Fields

- long **quot**
- long **rem**

7.2.2 Field Documentation

7.2.2.1 long ldiv_t::quot

The Quotient.

7.2.2.2 long ldiv_t::rem

The Remainder.

The documentation for this struct was generated from the following file:

- stdlib.h

8 avr-libc Page Documentation

8.1 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**(p. 151).

- Brian Dean [`bsd@bsdhome.com`] for developing **avrdude** (an alternative to **uisp**) and for contributing **documentation**(p. 250) which describes how to use it. **Avrdude** was previously called **avrprog**.

- Eric Weddington [`eric@evcohs.com`] for maintaining the **WinAVR** package and thus making the continued improvements to the Opensource 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**(p. 151).

- 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. ;-)

8.2 avr-libc and assembler programs

8.2.1 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**(p. 217) 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**(p. 241) can also be utilized.

8.2.2 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 invokation of the C preprocessor will be automatic, and will include the appropriate options to locate required include files in the filesystem.
- The invokation 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 invokation 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.

Alternatively, the language can explicitly be specified using the `-x assembler-with-cpp` option.

8.2.3 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]

                                ; Note [4]:
tmconst= 10700000 / 200000      ; 100 kHz => 200000 edges/s
fuzz=   8                       ; # clocks in ISR until TCNT0 is set

        .section .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     _SFR_IO_ADDR(TCNT0), inttmp     ; Note [8]

        in      intsav, _SFR_IO_ADDR(SREG)      ; Note [9]

        sbic    _SFR_IO_ADDR(PORTD), SQUARE
        rjmp    1f
        sbi     _SFR_IO_ADDR(PORTD), SQUARE
        rjmp    2f
1:      cbi     _SFR_IO_ADDR(PORTD), SQUARE
2:

        out     _SFR_IO_ADDR(SREG), intsav
        reti

ioinit:
        sbi     _SFR_IO_ADDR(DDRD), SQUARE

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

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

        ldi     work, 256 - tmconst
        out     _SFR_IO_ADDR(TCNT0), work

        sei

        ret

        .global __vector_default                ; Note [10]
__vector_default:
        reti

        .end
```

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 ininitalization 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**(p. 129) 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**(p. 31), 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**(p. 206) imposed by the C compiler. Also, any register modified inside the interrupt sevice needs to be saved, usually on the stack.

Note [10]

As explained in **Interrupts**(p. 127), 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.)

8.2.4 Pseudo-ops and operators

The available pseudo-ops in the assembler are described in the GNU assembler (gas) manual. The manual can be found online as part of the current binutils release under `http://sources.redhat.com/binutils/`.

As gas comes from a Unix origin, its pseudo-op 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), expressions use a C-like syntax.

Some common pseudo-ops include:

- .byte allocates single byte constants
- .ascii allocates a non-terminated string of characters
- .asciz allocates a \0-terminated string of characters (C string)
- .data switches to the .data section (initialized RAM variables)
- .text switches to the .text section (code and ROM constants)
- .set declares a symbol as a constant expression (identical to .equ)
- .global (or .globl) declares a public symbol that is visible to the linker (e. g. function entry point, global variable)
- .extern declares a symbol to be externally defined; this is effectively a comment only, as gas treats all undefined symbols it encounters as globally undefined anyway

Note that .org is available in gas as well, but is a fairly pointless pseudo-op in an assembler environment that uses relocatable object files, as it is the linker that determines the final position of some object in ROM or RAM.

Along with the architecture-independent standard operators, there are some AVR-specific operators available which are unfortunately not yet described in the official documentation. The most notable operators are:

- `lo8` Takes the least significant 8 bits of a 16-bit integer
- `hi8` Takes the most significant 8 bits of a 16-bit integer
- `pm` Takes a program-memory (ROM) address, and converts it into a RAM address. This implies a division by 2 as the AVR handles ROM addresses as 16-bit words (e.g. in an `IJMP` or `ICALL` instruction), and can also handle relocatable symbols on the right-hand side.

Example:

```
ldi r24, lo8(pm(somefunc))
ldi r25, hi8(pm(somefunc))
call something
```

This passes the address of function `somefunc` as the first parameter to function `something`.

8.3 Frequently Asked Questions

8.3.1 FAQ Index

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

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

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

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

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

```
volatile uint8_t flag;
```

Back to **FAQ Index**(p. 194).

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

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

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

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

Back to **FAQ Index**(p. 194).

8.3.4 How to permanently bind a variable to a register?

This can be done with

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

Typically, it should be possible to use r2 through r15 that way.

See **C Names Used in Assembler Code**(p. 229) for more details.

Back to **FAQ Index**(p. 194).

8.3.5 How to modify MCUCR or WDTCR early?

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

```
;; begin xram.S

#include <avr/io.h>

        .section .init1,"ax",@progbits

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

;; end xram.S
```

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

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

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

For more information on using sections, see **Memory Sections**(p. 239). There is also an example for **Using Sections in C Code**(p. 243). Note that in C code, any such function would preferrably be placed into section .init3 as the code in .init2 ensures the internal register `__zero_reg__` is already cleared.

Back to **FAQ Index**(p. 194).

8.3.6 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**(p. 18) 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()**(p. 146) macro. Of course, the implementation of this macro is just the usual bit shift (which is done by the compiler anyway, thus doesn't impose any run-time penalty), so the following applies:

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

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

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

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

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

Back to **FAQ Index**(p. 194).

8.3.7 Can I use C++ on the AVR?

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

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

- Obviously, none of the C++ related standard functions, classes, and template classes are available.

- The operators `new` and `delete` are not implemented, attempting to use them will cause the linker to complain about undefined external references. (This could perhaps be fixed.)
- Some of the supplied include files are not C++ safe, i. e. they need to be wrapped into

  ```
  extern "C" { . . . }
  ```

  (This could certainly be fixed, too.)
- Exceptions are not supported. Since exceptions are enabled by default in the C++ frontend, they explicitly need to be turned off using `-fno-exceptions` in the compiler options. Failing this, the linker will complain about an undefined external reference to `__gxx_personality_sj0`.

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

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

Back to **FAQ Index**(p. 194).

8.3.8 Shouldn't I initialize all my variables?

Global and static variables are guaranteed to be initialized to 0 by the C standard. `avr-gcc` does this by placing the appropriate code into section .init4 (see **The .initN Sections**(p. 241)). With respect to the standard, this sentence is somewhat simplified (because the standard allows for machines where the actual bit pattern used differs from all bits being 0), but for the AVR target, in general, all integer-type variables are set to 0, all pointers to a NULL pointer, and all floating-point variables to 0.0.

As long as these variables are not initialized (i. e. they don't have an equal sign and an initialization expression to the right within the definition of the variable), they go into the **.bss**(p. 240) section of the file. This section simply records the size of the variable, but otherwise doesn't consume space, neither within the object file nor within flash memory. (Of course, being a variable, it will consume space in the target's SRAM.)

In contrast, global and static variables that have an initializer go into the **.data**(p. 240) section of the file. This will cause them to consume space in the object file (in order to record the initializing value), *and* in the flash ROM of the target device. The latter is needed since the flash ROM is the only way that the compiler can tell the target device the value this variable is going to be initialized to.

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

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

Note:

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

Back to **FAQ Index**(p. 194).

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

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

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

To protect interrupt routines against other interrupt routines, it's usually best to use the **ISR()**(p. 143) macro when declaring the interrupt function, and to ensure that interrupts are still disabled when accessing those 16-bit timer registers.

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

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

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

        return val;
}
```

Back to **FAQ Index**(p. 194).

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

So you tried this:

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

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

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

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

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

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

Note:

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

Back to **FAQ Index**(p. 194).

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

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

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

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

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

Back to **FAQ Index**(p. 194).

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

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

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

Example:

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

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

Note:

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

Example:

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

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

Example:

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

```
        brlo    1b              ; jump back to top of loop

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

Back to **FAQ Index**(p. 194).

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

Consider this example code:

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

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

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

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

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

    return (0);
}
```

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

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

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

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

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

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

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

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

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

Note:

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

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

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

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

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

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

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

Back to **FAQ Index**(p. 194).

8.3.14 What registers are used by the C compiler?

- **Data types:**

  `char` is 8 bits, `int` is 16 bits, `long` is 32 bits, `long` long is 64 bits, `float` and `double` are 32 bits (this is the only supported floating point format), pointers are 16 bits (function pointers are word addresses, to allow addressing the whole 128K program memory space on the ATmega devices with > 64 KB of flash ROM). There is a `-mint8` option (see **Options for the C compiler avr-gcc**(p. 252)) to make `int` 8 bits, but that is not supported by avr-libc and violates C standards (`int` *must* be at least 16 bits). It may be removed in a future release.

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

  May be allocated by gcc for local data. You *may* use them freely in assembler subroutines. Calling C subroutines can clobber any of them - the caller is responsible for saving and restoring.

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

  May be allocated by gcc for local data. Calling C subroutines leaves them unchanged. Assembler subroutines are responsible for saving and restoring these registers, if changed. r29:r28 (Y pointer) is used as a frame pointer (points to local data on stack) if necessary.

- **Fixed registers (r0, r1):**

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

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

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

- **Function call conventions:**

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

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

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

Warning:

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

Back to **FAQ Index**(p. 194).

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

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

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

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

int main (void)
{
    char buf[32];
```

```
    strcpy_P (buf, array[1]);
    return 0;
}
```

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

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

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

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

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

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

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

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

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

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

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

`foo` is at addr 0x002e.

`bar` is at addr 0x002a.

`array` is at addr 0x0026.

Then in main we see this:

```
    memcpy_P(&p, &array[i], sizeof(PGM_P));
```

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

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

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

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

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

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

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

Back to **FAQ Index**(p. 194).

8.3.16 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**(p. 241) so the initialization of these variable will take place. Refer to **How to modify MCUCR or WDTCR early?**(p. 197) 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**(p. 243).

The explanation of **malloc()**(p. 102) contains a **discussion**(p. 232) 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()**(p. 102)). 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**(p. 194).

8.3.17 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 tradeoff. See the **detailed discussion**(p. 256) 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()**(p. 102) using the standard library **strcmp()**(p. 112), test #2 used a function that sorted the strings by their size (thus had two calls to **strlen()**(p. 113) 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**(p. 194).

8.3.18 How do I relocate code to a fixed address?

First, the code should be put into a new **named section**(p. 239). This is done with a section attribute:

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

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

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

To relocate the section to a fixed address the linker flag `-section-start` is used. This option can be passed to the linker using the **-Wl compiler option**(p. 257):

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

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

Back to **FAQ Index**(p. 194).

8.3.19 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**(p. 194).

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

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

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

Of course, this is going to waste a lot of SRAM. In **Program Space String Utilities**(p. 19), a method is described how such constant data can be moved out to flash ROM. However, a constant string located in flash ROM is no longer a valid argument to pass to a function that expects a `const char *`-type string, since the AVR processor needs the special instruction `LPM` to access these strings. Thus, separate functions are needed that take this into account. Many of the standard C library functions have equivalents available where one of the string arguments can be located in flash ROM. Private functions in the applications need to handle this, too. For example, the following can be used to implement simple debugging messages that will be sent through a UART:

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

int
uart_putchar(char c)
```

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

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

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

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

Note:

> By convention, the suffix **_P** to the function name is used as an indication that this function is going to accept a "program-space string". Note also the use of the **PSTR()**(p. 23) macro.

Back to **FAQ Index**(p. 194).

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

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

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

This may be especially important when clearing a bit:

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

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

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

Back to **FAQ Index**(p. 194).

8.3.22 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 avaialable 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()**(p. 102), which e. g. also can happen inside **printf()**(p. 88), the heap for dynamic memory is also located there. See **Using malloc()**(p. 230).)

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

Back to **FAQ Index**(p. 194).

8.3.23 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**(p. 194).

8.3.24 What is this "clock skew detected" messsage?

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**(p. 194).

8.3.25 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**(p. 194).

8.3.26 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**(p. 194).

8.3.27 Which AVR-specific assembler operators are available?

See **Pseudo-ops and operators**(p. 193).

Back to **FAQ Index**(p. 194).

8.3.28 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**(p. 194).

8.3.29 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**(p. 194).

8.4 Inline Asm

AVR-GCC

Inline Assembler Cookbook

About this Document

The GNU C compiler for Atmel AVR RISC processors offers, to embed assembly language code into C programs. This cool feature may be used for manually optimizing time critical parts of the software or to use specific processor instruction, which are not available in the C language.

Because of a lack of documentation, especially for the AVR version of the compiler, it may take some time to figure out the implementation details by studying the compiler and assembler source code. There are also a few sample programs available in the net. Hopefully this document will help to increase their number.

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 language tutorial either.

Note that this document does not cover file written completely in assembler language, refer to **avr-libc and assembler programs**(p. 188) for this.

This document describes version 3.3 of the compiler. There may be some parts, which hadn't been completely understood by the author himself and not all samples had been tested so far. Because the author is German and not familiar with the English language, there are definitely some typos and syntax errors in the text. As a programmer the author knows, that a wrong documentation sometimes might be worse than none. Anyway, he decided to offer his little knowledge to the public, in the hope to get enough response to improve this document. Feel free to contact the author via e-mail. For the latest release check `http://www.ethernut.de/`.

Herne, 17th of May 2002 Harald Kipp harald.kipp-at-egnite.de

Note:

> As of 26th of July 2002, this document has been merged into the documentation for avr-libc. The latest version is now available at `http://savannah.nongnu.org/projects/avr-libc/`.

8.4.1 GCC asm Statement

Let's start with a simple example of reading a value from port D:

```
asm("in %0, %1" : "=r" (value) : "I" (_SFR_IO_ADDR(PORTD)) );
```

Each `asm` statement is devided by colons into (up to) four parts:

1. The assembler instructions, defined as a single string constant:

   ```
   "in %0, %1"
   ```

2. A list of output operands, separated by commas. Our example uses just one:

   ```
   "=r" (value)
   ```

3. A comma separated list of input operands. Again our example uses one operand only:

   ```
   "I" (_SFR_IO_ADDR(PORTD))
   ```

4. Clobbered registers, left empty in our example.

You can write assembler instructions in much the same way as you would write assembler programs. However, registers and constants are used in a different way if they refer to expressions of your C program. The connection between registers and C operands is specified in the second and third part of the `asm` instruction, the list of input and output operands, respectively. The general form is

```
asm(code : output operand list : input operand list [: clobber list]);
```

In the code section, operands are referenced by a percent sign followed by a single digit. `0` refers to the first `1` to the second operand and so forth. From the above example:

`0` refers to `"=r" (value)` and

`1` refers to `"I" (_SFR_IO_ADDR(PORTD))`.

This may still look a little odd now, but the syntax of an operand list will be explained soon. Let us first examine the part of a compiler listing which may have been generated from our example:

```
        lds r24,value
/* #APP */
        in r24, 12
/* #NOAPP */
        sts value,r24
```

The comments have been added by the compiler to inform the assembler that the included code was not generated by the compilation of C statements, but by inline assembler statements. The compiler selected register `r24` for storage of the value read from `PORTD`. The compiler could have selected any other register, though. It may not explicitely load or store the value and it may even decide not to include your assembler code at all. All these decisions are part of the compiler's optimization strategy. For example, if you never use the variable value in the remaining part of the C program, the compiler will most likely remove your code unless you switched off optimization. To avoid this, you can add the volatile attribute to the `asm` statement:

```
asm volatile("in %0, %1" : "=r" (value) : "I" (_SFR_IO_ADDR(PORTD)));
```

The last part of the `asm` instruction, the clobber list, is mainly used to tell the compiler about modifications done by the assembler code. This part may be omitted, all other parts are required, but may be left empty. If your assembler routine won't use any input or output operand, two colons must still follow the assembler code string. A good example is a simple statement to disable interrupts:

```
asm volatile("cli"::);
```

8.4.2 Assembler Code

You can use the same assembler instruction mnemonics as you'd use with any other AVR assembler. And you can write as many assembler statements into one code string as you like and your flash memory is able to hold.

Note:

The available assembler directives vary from one assembler to another.

To make it more readable, you should put each statement on a seperate line:

```
asm volatile("nop\n\t"
             "nop\n\t"
             "nop\n\t"
             "nop\n\t"
             ::);
```

The linefeed and tab characters will make the assembler listing generated by the compiler more readable. It may look a bit odd for the first time, but that's the way the compiler creates it's own assembler code.

You may also make use of some special registers.

Symbol	Register
`__SREG__`	Status register at address 0x3F
`__SP_H__`	Stack pointer high byte at address 0x3E
`__SP_L__`	Stack pointer low byte at address 0x3D
`__tmp_reg__`	Register r0, used for temporary storage
`__zero_reg__`	Register r1, always zero

Register `r0` may be freely used by your assembler code and need not be restored at the end of your code. It's a good idea to use `__tmp_reg__` and `__zero_reg__` instead of `r0` or `r1`, just in case a new compiler version changes the register usage definitions.

8.4.3 Input and Output Operands

Each input and output operand is described by a constraint string followed by a C expression in parantheses. `AVR-GCC` 3.3 knows the following constraint characters:

Note:

The most up-to-date and detailed information on contraints for the avr can be found in the gcc manual.

The `x` register is `r27:r26`, the `y` register is `r29:r28`, and the `z` register is `r31:r30`

Constraint	Used for	Range
a	Simple upper registers	r16 to r23
b	Base pointer registers pairs	y, z
d	Upper register	r16 to r31
e	Pointer register pairs	x, y, z
G	Floating point constant	0.0
I	6-bit positive integer constant	0 to 63
J	6-bit negative integer constant	-63 to 0
K	Integer constant	2
L	Integer constant	0
l	Lower registers	r0 to r15
M	8-bit integer constant	0 to 255
N	Integer constant	-1
O	Integer constant	8, 16, 24
P	Integer constant	1
q	Stack pointer register	SPH:SPL
r	Any register	r0 to r31
t	Temporary register	r0
w	Special upper register pairs	r24, r26, r28, r30
x	Pointer register pair X	x (r27:r26)
y	Pointer register pair Y	y (r29:r28)
z	Pointer register pair Z	z (r31:r30)

These definitions seem not to fit properly to the AVR instruction set. The author's assumption is, that this part of the compiler has never been really finished in this version, but that assumption may be wrong. The selection of the proper contraint 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. For example, if you specify the constraint `"r"` and you are using this register with an `"ori"` instruction in your assembler code, 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 value. Later on we will see how to pass multibyte expression results to the assembler code.

The following table shows all AVR assembler mnemonics which require operands, and the related contraints. Because of the improper constraint definitions in version 3.3, they aren't strict enough. There is, for example, no constraint, which restricts integer constants to the range 0 to 7 for bit set and bit clear operations.

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
cbi	I,I		cbr	d,I
com	r		cp	r,r
cpc	r,r		cpi	d,M
cpse	r,r		dec	r
elpm	t,z		eor	r,r
in	r,I		inc	r
ld	r,e		ldd	r,b
ldi	d,M		lds	r,label
lpm	t,z		lsl	r
lsr	r		mov	r,r
movw	r,r		mul	r,r
neg	r		or	r,r
ori	d,M		out	I,r
pop	r		push	r
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	label,r
sub	r,r		subi	d,M
swap	r			

Constraint characters may be prepended by a single constraint modifier. Contraints without a modifier specify read-only operands. Modifiers are:

Modifier	**Specifies**
=	Write-only operand, usually used for all output operands.
+	Read-write operand (not supported by inline assembler)
&	Register should be used for output only

Output operands must be write-only and the C expression result must be an lvalue, which means that the operands must be valid on the left side of assignments. Note, that the compiler will not check if the operands are of reasonable type for the kind of operation used in the assembler instructions.

Input operands are, you guessed it, read-only. But what if you need the same operand for input and output? As stated above, read-write operands are not supported in inline assembler code. But there is another solution. For input operators it is possible to use a single digit in the constraint string. Using digit n tells the compiler to use the same register as for the n-th operand, starting with zero. Here is an example:

```
asm volatile("swap %0" : "=r" (value) : "0" (value));
```

This statement will swap the nibbles of an 8-bit variable named value. Constraint `"0"` tells the compiler, to use the same input register as for the first operand. Note however, that this doesn't automatically imply the reverse case. The compiler may choose the same registers for input and output, even if not told to do so. This is not a problem in most cases, but may be fatal if the output operator is modified by the assembler code before the input operator is used. In the situation where your code depends on different registers used for input and output operands, you must add the `&` constraint modifier to your output operand. The following example demonstrates this problem:

```
asm volatile("in %0,%1"    "\n\t"
             "out %1, %2"  "\n\t"
             : "=&r" (input)
             : "I" (_SFR_IO_ADDR(port)), "r" (output)
            );
```

In this example an input value is read from a port and then an output value is written to the same port. If the compiler would have choosen the same register for input and output, then the output value would have been destroyed on the first assembler instruction. Fortunately, this example uses the `&` constraint modifier to instruct the compiler not to select any register for the output value, which is used for any of the input operands. Back to swapping. Here is the code to swap high and low byte of a 16-bit value:

```
asm volatile("mov __tmp_reg__, %A0" "\n\t"
             "mov %A0, %B0"         "\n\t"
             "mov %B0, __tmp_reg__" "\n\t"
             : "=r" (value)
             : "0" (value)
            );
```

First you will notice the usage of register `__tmp_reg__`, which we listed among other special registers in the **Assembler Code**(p. 220) section. You can use this register without saving its contents. Completely new are those letters `A` and `B` in `%A0` and `%B0`. In fact they refer to two different 8-bit registers, both containing a part of value.

Another example to swap bytes of a 32-bit value:

```
asm volatile("mov __tmp_reg__, %A0" "\n\t"
             "mov %A0, %D0"         "\n\t"
             "mov %D0, __tmp_reg__" "\n\t"
             "mov __tmp_reg__, %B0" "\n\t"
             "mov %B0, %C0"         "\n\t"
             "mov %C0, __tmp_reg__" "\n\t"
             : "=r" (value)
             : "0" (value)
            );
```

If operands do not fit into a single register, the compiler will automatically assign enough registers to hold the entire operand. In the assembler code you use `%A0` to refer to the lowest byte of the first operand, `%A1` to the lowest byte of the second operand and so on. The next byte of the first operand will be `%B0`, the next byte `%C0` and so on.

This also implies, that it is often neccessary to cast the type of an input operand to the desired size.

A final problem may arise while using pointer register pairs. If you define an input operand

```
"e" (ptr)
```

and the compiler selects register `Z` (r30:r31), then

`%A0` refers to `r30` and

`%B0` refers to `r31`.

But both versions will fail during the assembly stage of the compiler, if you explicitely need `Z`, like in

```
ld r24,Z
```

If you write

```
ld r24, %a0
```

with a lower case `a` following the percent sign, then the compiler will create the proper assembler line.

8.4.4 Clobbers

As stated previously, the last part of the `asm` statement, the list of clobbers, may be omitted, including the colon seperator. However, if you are using registers, which had not been passed as operands, you need to inform the compiler about this. The following example will do an atomic increment. It increments an 8-bit value pointed to by a pointer variable in one go, without being interrupted by an interrupt routine or another thread in a multithreaded environment. Note, that we must use a pointer, because the incremented value needs to be stored before interrupts are enabled.

```
asm volatile(
    "cli"               "\n\t"
    "ld r24, %a0"       "\n\t"
    "inc r24"           "\n\t"
    "st %a0, r24"       "\n\t"
    "sei"               "\n\t"
    :
    : "e" (ptr)
    : "r24"
);
```

The compiler might produce the following code:

```
    cli
    ld r24, Z
    inc r24
    st Z, r24
    sei
```

One easy solution to avoid clobbering register `r24` is, to make use of the special temporary register `__tmp_reg__` defined by the compiler.

```
asm volatile(
    "cli"                       "\n\t"
    "ld __tmp_reg__, %a0"       "\n\t"
    "inc __tmp_reg__"           "\n\t"
    "st %a0, __tmp_reg__"       "\n\t"
    "sei"                       "\n\t"
    :
    : "e" (ptr)
);
```

The compiler is prepared to reload this register next time it uses it. Another problem with the above code is, that it should not be called in code sections, where interrupts are disabled and should be kept disabled, because it will enable interrupts at the end. We may store the current status, but then we need another register. Again we can solve this without clobbering a fixed, but let the compiler select it. This could be done with the help of a local C variable.

```
{
    uint8_t s;
    asm volatile(
        "in %0, __SREG__"           "\n\t"
        "cli"                       "\n\t"
        "ld __tmp_reg__, %a1"       "\n\t"
```

```
        "inc __tmp_reg__"           "\n\t"
        "st %a1, __tmp_reg__"       "\n\t"
        "out __SREG__, %0"          "\n\t"
        : "=&r" (s)
        : "e" (ptr)
    );
}
```

Now every thing seems correct, but it isn't really. The assembler code modifies the variable, that `ptr` points to. The compiler will not recognize this and may keep its value in any of the other registers. Not only does the compiler work with the wrong value, but the assembler code does too. The C program may have modified the value too, but the compiler didn't update the memory location for optimization reasons. The worst thing you can do in this case is:

```
{
    uint8_t s;
    asm volatile(
        "in %0, __SREG__"           "\n\t"
        "cli"                       "\n\t"
        "ld __tmp_reg__, %a1"       "\n\t"
        "inc __tmp_reg__"           "\n\t"
        "st %a1, __tmp_reg__"       "\n\t"
        "out __SREG__, %0"          "\n\t"
        : "=&r" (s)
        : "e" (ptr)
        : "memory"
    );
}
```

The special clobber "memory" informs the compiler that the assembler code may modify any memory location. It forces the compiler to update all variables for which the contents are currently held in a register before executing the assembler code. And of course, everything has to be reloaded again after this code.

In most situations, a much better solution would be to declare the pointer destination itself volatile:

```
volatile uint8_t *ptr;
```

This way, the compiler expects the value pointed to by `ptr` to be changed and will load it whenever used and store it whenever modified.

Situations in which you need clobbers are very rare. In most cases there will be better ways. Clobbered registers will force the compiler to store their values before and reload them after your assembler code. Avoiding clobbers gives the compiler more freedom while optimizing your code.

8.4.5 Assembler Macros

In order to reuse your assembler language parts, it is useful to define them as macros and put them into include files. AVR Libc comes with a bunch of them, which could be found in the directory `avr/include`. Using such include files may produce compiler warnings, if they are used in modules, which are compiled in strict ANSI mode. To avoid that, you can write `__asm__` instead of `asm` and `__volatile__` instead of `volatile`. These are equivalent aliases.

Another problem with reused macros arises if you are using labels. In such cases you may make use of the special pattern =, which is replaced by a unique number on each `asm` statement. The following code had been taken from `avr/include/iomacros.h`:

```
#define loop_until_bit_is_clear(port,bit)  \
        __asm__ __volatile__ (             \
        "L_%=: " "sbic %0, %1" "\n\t"      \
                 "rjmp L_%="               \
                 : /* no outputs */        \
                 : "I" (_SFR_IO_ADDR(port)),
                   "I" (bit)
        )
```

When used for the first time, `L_=` may be translated to `L_1404`, the next usage might create `L_1405` or whatever. In any case, the labels became unique too.

Another option is to use Unix-assembler style numeric labels. They are explained in **How do I trace an assembler file in avr-gdb?**(p. 202). The above example would then look like:

```
#define loop_until_bit_is_clear(port,bit)
        __asm__ __volatile__ (
        "1: " "sbic %0, %1" "\n\t"
                 "rjmp 1b"
                 : /* no outputs */
                 : "I" (_SFR_IO_ADDR(port)),
                   "I" (bit)
        )
```

8.4.6 C Stub Functions

Macro definitions will include the same assembler code whenever they are referenced. This may not be acceptable for larger routines. In this case you may define a C stub function, containing nothing other than your assembler code.

```
void delay(uint8_t ms)
{
    uint16_t cnt;
    asm volatile (
```

```
        "\n"
        "L_dl1%=:" "\n\t"
        "mov %A0, %A2" "\n\t"
        "mov %B0, %B2" "\n"
        "L_dl2%=:" "\n\t"
        "sbiw %A0, 1" "\n\t"
        "brne L_dl2%=" "\n\t"
        "dec %1" "\n\t"
        "brne L_dl1%=" "\n\t"
        : "=&w" (cnt)
        : "r" (ms), "r" (delay_count)
        );
}
```

The purpose of this function is to delay the program execution by a specified number of milliseconds using a counting loop. The global 16 bit variable delay_count must contain the CPU clock frequency in Hertz divided by 4000 and must have been set before calling this routine for the first time. As described in the **clobber**(p. 225) section, the routine uses a local variable to hold a temporary value.

Another use for a local variable is a return value. The following function returns a 16 bit value read from two successive port addresses.

```
uint16_t inw(uint8_t port)
{
    uint16_t result;
    asm volatile (
        "in %A0,%1" "\n\t"
        "in %B0,(%1) + 1"
        : "=r" (result)
        : "I" (_SFR_IO_ADDR(port))
        );
    return result;
}
```

Note:

inw() is supplied by avr-libc.

8.4.7 C Names Used in Assembler Code

By default `AVR-GCC` uses the same symbolic names of functions or variables in C and assembler code. You can specify a different name for the assembler code by using a special form of the `asm` statement:

```
unsigned long value asm("clock") = 3686400;
```

This statement instructs the compiler to use the symbol name clock rather than value. This makes sense only for external or static variables, because local variables do not have symbolic names in the assembler code. However, local variables may be held in registers.

With `AVR-GCC` you can specify the use of a specific register:

```
void Count(void)
{
    register unsigned char counter asm("r3");

    ... some code...
    asm volatile("clr r3");
    ... more code...
}
```

The assembler instruction, `"clr r3"`, will clear the variable counter. `AVR-GCC` will not completely reserve the specified register. If the optimizer recognizes that the variable will not be referenced any longer, the register may be re-used. But the compiler is not able to check wether this register usage conflicts with any predefined register. If you reserve too many registers in this way, the compiler may even run out of registers during code generation.

In order to change the name of a function, you need a prototype declaration, because the compiler will not accept the `asm` keyword in the function definition:

```
extern long Calc(void) asm ("CALCULATE");
```

Calling the function `Calc()` will create assembler instructions to call the function `CALCULATE`.

8.4.8 Links

For a more thorough discussion of inline assembly usage, see the gcc user manual. The latest version of the gcc manual is always available here: `http://gcc.gnu.org/onlinedocs/`

8.5 Using malloc()

8.5.1 Introduction

On a simple device like a microcontroller, implementing dynamic memory allocation is quite a challenge.

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 128 bytes of RAM. This needs to be shared between initialized and uninitialized variables (**sections**(p. 239) .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 function.

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.

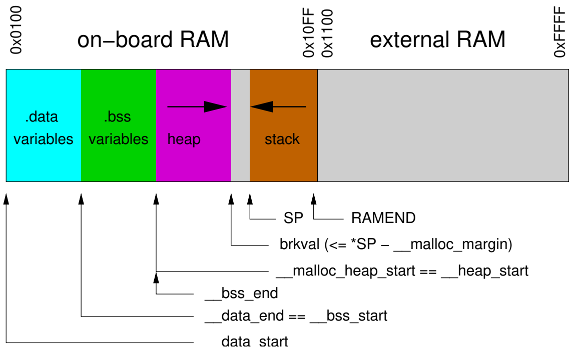

Figure 6: RAM map of a device with internal RAM

Finally, there's a challenge to make the memory allocator 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 since microcontrollers aren't only often low on space, but 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.

8.5.2 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.

8.5.3 Tunables for malloc()

There are a number of variables that can be tuned to adapt the behavior of **malloc()**(p. 102) to the expected requirements and constraints of the application. Any changes to these tunables should be made before the very first call to **malloc()**(p. 102). Note that some library functions might also use dynamic memory (notably those from the **<stdio.h>: Standard IO facilities**(p. 77)), 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()**(p. 102) 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()**(p. 102) 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,-Tdata=0x801100,--defsym=__heap_end=0x80ffff ...
```

Note:

See **explanation**(p. 241) for offset 0x800000. See the chapter about **using gcc**(p. 257) for the `-Wl` options.

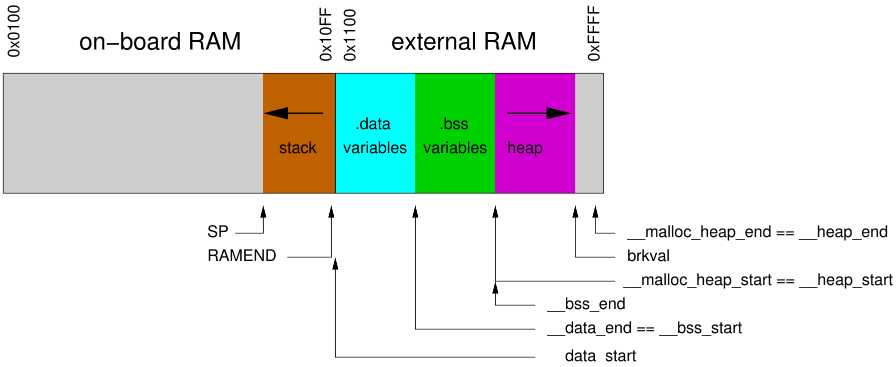

Figure 7: Internal RAM: stack only, external RAM: variables and heap

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 ...
```

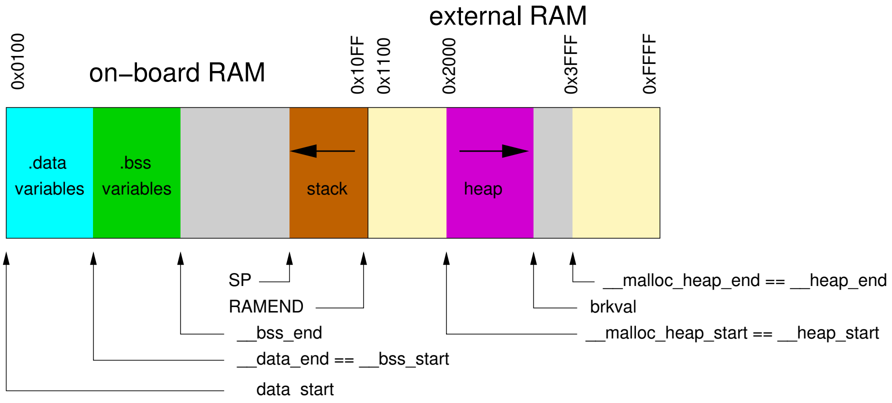

Figure 8: Internal RAM: variables and stack, external RAM: heap

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.

8.5.4 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()**(p. 101). 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()**(p. 101). 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 reqired 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()**(p. 101), 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.

A call to **realloc()**(p. 103) 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()**(p. 101) 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()**(p. 103) 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()**(p. 102)), the entire request will fail.

Otherwise, **malloc()**(p. 102) will be called with the new request size, the existing data will be copied over, and **free()**(p. 101) will be called on the old region.

8.6 Release Numbering and Methodology

8.6.1 Release Version Numbering Scheme

8.6.1.1 Stable Versions

A stable release will always have a minor number that is an even number. This implies that you should be able to upgrade to a new version of the library with the same major and minor numbers without fear that any of the APIs have changed. The only changes that should be made to a stable branch are bug fixes and under some circumstances, additional functionality (e.g. adding support for a new device).

If major version number has changed, this implies that the required versions of gcc and binutils have changed. Consult the README file in the toplevel directory of the AVR Libc source for which versions are required.

8.6.1.2 Development Versions

The major version number of a development series is always the same as the last stable release.

The minor version number of a development series is always an odd number and is 1 more than the last stable release.

The patch version number of a development series is always 0 until a new branch is cut at which point the patch number is changed to 90 to denote the branch is approaching a release and the date appended to the version to denote that it is still in development.

All versions in development in cvs will also always have the date appended as a fourth version number. The format of the date will be YYYYMMDD.

So, the development version number will look like this:

```
1.1.0.20030825
```

While a pre-release version number on a branch (destined to become either 1.2 or 2.0) will look like this:

```
1.1.90.20030828
```

8.6.2 Releasing AVR Libc

The information in this section is only relevant to AVR Libc developers and can be ignored by end users.

Note:

> In what follows, I assume you know how to use cvs and how to checkout multiple source trees in a single directory without having them clobber each other. If you don't know how to do this, you probably shouldn't be making releases or cutting branches.

8.6.2.1 Creating a cvs branch The following steps should be taken to cut a branch in cvs:

1. Check out a fresh source tree from cvs HEAD.
2. Update the NEWS file with pending release number and commit to cvs HEAD:

   Change "Changes since avr-libc-<last_release>:" to "Changes in avr-libc-<this_relelase>:".
3. Set the branch-point tag (setting <major> and <minor> accordingly):

   'cvs tag avr-libc-<major>_<minor>-branchpoint'
4. Create the branch:

   'cvs tag -b avr-libc-<major>_<minor>-branch'

5. Update the package version in configure.ac and commit configure.ac to cvs HEAD:

   Change minor number to next odd value.

6. Update the NEWS file and commit to cvs HEAD:

   Add "Changes since avr-libc-<this_release>:"

7. Check out a new tree for the branch:

   'cvs co -r avr-libc-<major>_<minor>-branch'

8. Update the package version in configure.ac and commit configure.ac to cvs branch:

   Change the patch number to 90 to denote that this now a branch leading up to a release. Be sure to leave the <date> part of the version.

9. Bring the build system up to date by running bootstrap and configure.
10. Perform a 'make distcheck' and make sure it succeeds. This will create the snapshot source tarball. This should be considered the first release candidate.
11. Upload the snapshot tarball to savannah.
12. Announce the branch and the branch tag to the avr-libc-dev list so other developers can checkout the branch.

Note:

CVS tags do not allow the use of periods ('.').

8.6.2.2 Making a release A stable release will only be done on a branch, not from the cvs HEAD.

The following steps should be taken when making a release:

1. Make sure the source tree you are working from is on the correct branch:

   'cvs update -r avr-libc-<major>_<minor>-branch'

2. Update the package version in configure.in and commit it to cvs.
3. Update the gnu tool chain version requirements in the README and commit to cvs.
4. Update the ChangeLog file to note the release and commit to cvs on the branch:

   Add "Released avr-libc-<this_release>."

5. Update the NEWS file with pending release number and commit to cvs:

   Change "Changes since avr-libc-<last_release>:" to "Changes in avr-libc-<this_relelase>:".

6. Bring the build system up to date by running bootstrap and configure.

7. Perform a 'make distcheck' and make sure it succeeds. This will create the source tarball.

8. Tag the release:

   'cvs tag avr-libc-<major>_<minor>_<patch>-release'

9. Upload the tarball to savannah.

10. Update the NEWS file, and commit to cvs:

    Add "Changes since avr-libc-<major>_<minor>_<patch>:"

11. Generate the latest documentation and upload to savannah.

12. Announce the release.

The following hypothetical diagram should help clarify version and branch relationships.

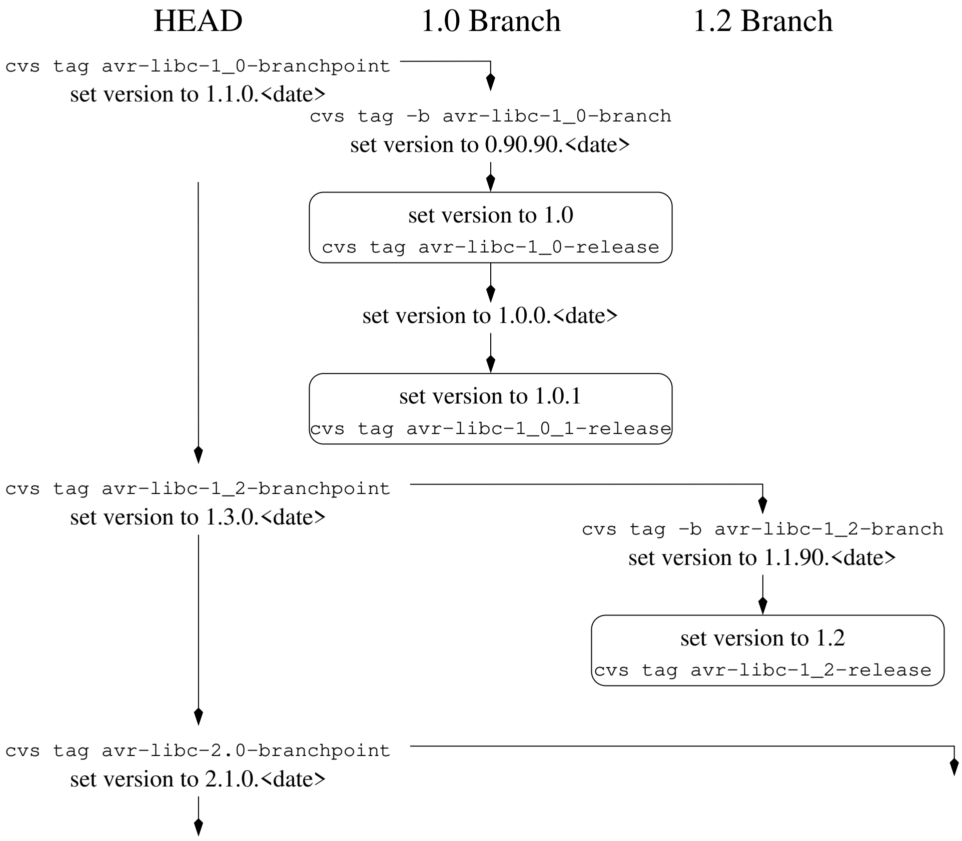

Figure 9: Release tree

8.7 Memory Sections

Remarks:

Need to list all the sections which are available to the avr.

Weak Bindings

FIXME: need to discuss the .weak directive.

The following describes the various sections available.

8.7.1 The .text Section

The .text section contains the actual machine instructions which make up your program. This section is further subdivided by the .initN and .finiN sections dicussed below.

Note:

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.

8.7.2 The .data Section

This section contains static data which was defined in your code. Things like the following would end up in .data:

```
char err_str[] = "Your program has died a horrible death!";

struct point pt = { 1, 1 };
```

It is possible to tell the linker the SRAM address of the beginning of the .data section. This is accomplished by adding `-Wl,-Tdata,`*`addr`* to the `avr-gcc` command used to the link your program. Not that `addr` must be offset by adding 0x800000 the to real SRAM address so that the linker knows that the address is in the SRAM memory space. Thus, if you want the .data section to start at 0x1100, pass 0x801100 at the address to the linker. [offset **explained**(p. 241)]

Note:

When using **malloc()**(p. 102) in the application (which could even happen inside library calls), **additional adjustments**(p. 232) are required.

8.7.3 The .bss Section

Uninitialized global or static variables end up in the .bss section.

8.7.4 The .eeprom Section

This is where eeprom variables are stored.

8.7.5 The .noinit Section

This sections is a part of the .bss section. What makes the .noinit section special is that variables which are defined as such:

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

will not be initialized to zero during startup as would normal .bss data.

Only uninitialized variables can be placed in the .noinit section. Thus, the following code will cause `avr-gcc` to issue an error:

```
int bar __attribute__ ((section (".noinit"))) = 0xaa;
```

It is possible to tell the linker explicitly where to place the .noinit section by adding `-Wl,-section-start=.noinit=0x802000` to the `avr-gcc` command line at the linking stage. For example, suppose you wish to place the .noinit section at SRAM address 0x2000:

```
$ avr-gcc ... -Wl,--section-start=.noinit=0x802000 ...
```

Note:

Because of the Harvard architecture of the AVR devices, you must manually add 0x800000 to the address you pass to the linker as the start of the section. Otherwise, the linker thinks you want to put the .noinit section into the .text section instead of .data/.bss and will complain.

Alternatively, you can write your own linker script to automate this. [FIXME: need an example or ref to dox for writing linker scripts.]

8.7.6 The .initN Sections

These sections are used to define the startup code from reset up through the start of main(). These all are subparts of the **.text section**(p. 239).

The purpose of these sections is to allow for more specific placement of code within your program.

Note:

Sometimes, it is convenient to think of the .initN and .finiN sections as functions, but in reality they are just symbolic names which tell the linker where to stick a chunk of code which is *not* a function. Notice that the examples for **asm**(p. 243) and **C**(p. 243) can not be called as functions and should not be jumped into.

The **.initN** sections are executed in order from 0 to 9.

.init0:

Weakly bound to __init(). If user defines __init(), it will be jumped into immediately after a reset.

.init1:

Unused. User definable.

.init2:
In C programs, weakly bound to initialize the stack, and to clear __zero_reg__ (r1).

.init3:
Unused. User definable.

.init4:

For devices with > 64 KB of ROM, .init4 defines the code which takes care of copying the contents of .data from the flash to SRAM. For all other devices, this code as well as the code to zero out the .bss section is loaded from libgcc.a.

.init5:
Unused. User definable.

.init6:
Unused for C programs, but used for constructors in C++ programs.

.init7:
Unused. User definable.

.init8:
Unused. User definable.

.init9:
Jumps into main().

8.7.7 The .finiN Sections

These sections are used to define the exit code executed after return from main() or a call to **exit()**(p. 100). These all are subparts of the **.text section**(p. 239).

The **.finiN** sections are executed in descending order from 9 to 0.

.finit9:
Unused. User definable. This is effectively where _exit() starts.

.fini8:
Unused. User definable.

.fini7:
Unused. User definable.

.fini6:
Unused for C programs, but used for destructors in C++ programs.

.fini5:
Unused. User definable.

.fini4:
Unused. User definable.

.fini3:
Unused. User definable.

.fini2:
Unused. User definable.

.fini1:
Unused. User definable.

.fini0:
Goes into an infinite loop after program termination and completion of any _exit() code (execution of code in the .fini9 -> .fini1 sections).

8.7.8 Using Sections in Assembler Code

Example:

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

        .section .init1,"ax",@progbits
        ldi       r0, 0xff
        out       _SFR_IO_ADDR(PORTB), r0
        out       _SFR_IO_ADDR(DDRB), r0
```

Note:
The `,"ax",@progbits` tells the assembler that the section is allocatable ("a"), executable ("x") and contains data ("@progbits"). For more detailed information on the .section directive, see the gas user manual.

8.7.9 Using Sections in C Code

Example:

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

void my_init_portb (void) __attribute__ ((naked)) \
    __attribute__ ((section (".init3")));

void
my_init_portb (void)
{
        PORTB = 0xff;
        DDRB = 0xff;
}
```

Note:

Section .init3 is used in this example, as this ensures the inernal `__zero_reg__` has already been set up. The code generated by the compiler might blindly rely on `__zero_reg__` being really 0.

8.8 Installing the GNU Tool Chain

Note:

This discussion was taken directly from Rich Neswold's document. (See **Acknowledgments**(p. 187)).

This discussion is Unix specific. [FIXME: troth/2002-08-13: we need a volunteer to add windows specific notes to these instructions.]

This chapter shows how to build and install a complete development environment for the AVR processors using the GNU toolset.

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 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.

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
```

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.

Note:

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

8.8.1 Required Tools

- **GNU Binutils**

  `http://sources.redhat.com/binutils/`

  Installation(p. 246)

- **GCC**

  `http://gcc.gnu.org/`

  Installation(p. 247)

- **AVR Libc**

  `http://savannah.gnu.org/projects/avr-libc/`

  Installation(p. 248)

8.8.2 Optional Tools

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

- **uisp**

  `http://savannah.gnu.org/projects/uisp/`

  Installation

- **avrdude**

  `http://savannah.nongnu.org/projects/avrdude/`

  Installation(p. 248)

  Usage Notes(p. 250)

- **GDB**

  `http://sources.redhat.com/gdb/`

  Installation(p. 249)

- **Simulavr**

  `http://savannah.gnu.org/projects/simulavr/`

  Installation(p. 249)

- **AVaRice**

  `http://avarice.sourceforge.net/`

  Installation(p. 249)

8.8.3 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`), dissassemble 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:

```
$ bunzip2 -c binutils-<version>.tar.bz2 | tar xf -
$ cd binutils-<version>
```

Note:

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

If you obtained a gzip compressed file (.gz), use `gunzip` instead of `bunzip2`.

It is usually a good idea to configure and build **binutils** in a subdirectory so as not to pollute the source with the compiled files. This is recommended by the **binutils** developers.

```
$ mkdir obj-avr
$ cd obj-avr
```

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.

```
$ ../configure --prefix=$PREFIX --target=avr --disable-nls
```

If you don't specify the `-prefix` option, the tools will get installed in the `/usr/local` hierarchy (i.e. the binaries will get installed in `/usr/local/bin`, the info pages get installed in `/usr/local/info`, etc.) Since these tools are changing frequently, It is preferrable to put them in a location that is easily removed.

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 `Makefiles` that are custom tailored to your platform. At this point, you can build the project.

```
$ make
```

Note:

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**(p. 244) environment variable before going to build avr-gcc.

Note:

The official version of binutils might lack support for recent AVR devices. A patch that adds more AVR types can be found at `http://www.freebsd.org/cgi/cvsweb.cgi/ports/devel/avr-binutils/files/patch-newdevice`

8.8.4 GCC for the AVR target

Warning:

You **must** install **avr-binutils**(p. 246) and make sure your **path is set**(p. 244) properly before installing avr-gcc.

The steps to build `avr-gcc` are essentially same as for **binutils**(p. 246):

```
$ bunzip2 -c gcc-<version>.tar.bz2 | tar xf -
$ cd gcc-<version>
$ mkdir obj-avr
$ cd obj-avr
$ ../configure --prefix=$PREFIX --target=avr --enable-languages=c,c++ \
    --disable-nls --disable-libssp --with-dwarf2
$ make
$ make install
```

To save your self some download time, you can alternatively download only the `gcc-core-<version>.tar.bz2` and `gcc-c++-<version>.tar.bz2` parts of the gcc. Also, if you don't need C++ support, you only need the core part and should only enable the C language support.

Note:

Early versions of these tools did not support C++.

The stdc++ libs are not included with C++ for AVR due to the size limitations of the devices.
The official version of GCC might lack support for recent AVR devices. A patch that adds more AVR types can be found at `http://www.freebsd.org/cgi/cvsweb.cgi/ports/devel/avr-gcc/files/patch-newdevices`

8.8.5 AVR Libc

Warning:
You **must** install **avr-binutils**(p. 246), **avr-gcc**(p. 247) and make sure your **path is set**(p. 244) properly before installing avr-libc.

Note:
If you have obtained the latest avr-libc from cvs, you will have to run the `bootstrap` script before using either of the build methods described below.

To build and install avr-libc:

```
$ gunzip -c avr-libc-<version>.tar.gz | tar xf -
$ cd avr-libc-<version>
$ ./configure --prefix=$PREFIX --build=`./config.guess` --host=avr
$ make
$ make install
```

8.8.6 Avrdude

Note:
It 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
```

Note:
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
```

8.8.7 GDB for the AVR target

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

```
$ bunzip2 -c gdb-<version>.tar.bz2 | tar xf -
$ cd gdb-<version>
$ mkdir obj-avr
$ cd obj-avr
$ ../configure --prefix=$PREFIX --target=avr
$ make
$ make install
```

Note:

> If you are planning on using `avr-gdb`, you will probably want to install either **simulavr**(p. 249) or **avarice**(p. 249) since avr-gdb needs one of these to run as a a remote target backend.

8.8.8 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
```

Note:

> You might want to have already installed **avr-binutils**(p. 246), **avr-gcc**(p. 247) and **avr-libc**(p. 248) if you want to have the test programs built in the simulavr source.

8.8.9 AVaRice

Note:

> These install notes are not applicable to avarice-1.5 or older. You probably don't want to use anything that old anyways 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
```

Note:

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
```

8.9 Using the avrdude program

Note:

This section was contributed by Brian Dean [`bsd@bsdhome.com`].

The avrdude program was previously called avrprog. The name was changed to avoid confusion with the avrprog program that Atmel ships with AvrStudio.

`avrdude` is a program that is used to update or read the flash and EEPROM memories of Atmel AVR microcontrollers on FreeBSD Unix. It supports the Atmel serial programming protocol using the PC's parallel port and can upload either a raw binary file or an Intel Hex format file. It can also be used in an interactive mode to individually update EEPROM cells, fuse bits, and/or lock bits (if their access is supported by the Atmel serial programming protocol.) The main flash instruction memory of the AVR can also be programmed in interactive mode, however this is not very useful because one can only turn bits off. The only way to turn flash bits on is to erase the entire memory (using `avrdude`'s `-e` option).

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

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

Once installed, `avrdude` can program processors using the contents of the .hex file specified on the command line. In this example, the file `main.hex` is burned into the flash memory:

```
# avrdude -p 2313 -e -m flash -i main.hex

avrdude: AVR device initialized and ready to accept instructions

avrdude: Device signature = 0x1e9101

avrdude: erasing chip
avrdude: done.
avrdude: reading input file "main.hex"
avrdude: input file main.hex auto detected as Intel Hex

avrdude: writing flash:
1749 0x00
avrdude: 1750 bytes of flash written
avrdude: verifying flash memory against main.hex:
avrdude: reading on-chip flash data:
1749  0x00
avrdude: verifying ...
avrdude: 1750 bytes of flash verified

avrdude done.  Thank you.
```

The `-p 2313` option lets `avrdude` know that we are operating on an AT90S2313 chip. This option specifies the device id and is matched up with the device of the same id in `avrdude`'s configuration file (`/usr/local/etc/avrdude`.conf). To list valid parts, specify the `-v` option. The `-e` option instructs `avrdude` to perform a chip-erase before programming; this is almost always necessary before programming the flash. The `-m flash` option indicates that we want to upload data into the flash memory, while `-i main.hex` specifies the name of the input file.

The EEPROM is uploaded in the same way, the only difference is that you would use `-m eeprom` instead of `-m flash`.

To use interactive mode, use the `-t` option:

```
# avrdude -p 2313 -t
avrdude: AVR device initialized and ready to accept instructions
avrdude: Device signature = 0x1e9101
avrdude>

The '?' command displays a list of valid
commands:

avrdude> ?
>>> ?
Valid commands:

  dump   : dump memory  : dump <memtype> <addr> <N-Bytes>
  read   : alias for dump
  write  : write memory : write <memtype> <addr> <b1> <b2> ... <bN>
  erase  : perform a chip erase
  sig    : display device signature bytes
  part   : display the current part information
  send   : send a raw command : send <b1> <b2> <b3> <b4>
```

```
  help   : help
  ?      : help
  quit   : quit

Use the 'part' command to display valid memory types for use with the
'dump' and 'write' commands.

avrdude>
```

8.10 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.

8.10.1 Options for the C compiler avr-gcc

8.10.1.1 Machine-specific options for the AVR The following machine-specific options are recognized by the C compiler frontend.

- -mmcu=*architecture*

Compile code for *architecture*. Currently known architectures are

avr1	Simple CPU core, only assembler support
avr2	"Classic" CPU core, up to 8 KB of ROM
avr3	"Classic" CPU core, more than 8 KB of ROM
avr4	"Enhanced" CPU core, up to 8 KB of ROM
avr5	"Enhanced" CPU core, more than 8 KB of ROM
avr6	"Enhanced" CPU core, Program Counter of 3 bytes

By default, code is generated for the avr2 architecture.

Note that when only using -mmcu=*architecture* but no -mmcu=*MCU type*, including the file <`avr/io.h`> cannot work since it cannot decide which device's definitions to select.

- -mmcu=*MCU type*

The following MCU types are currently understood by avr-gcc. The table matches them against the corresponding avr-gcc architecture name, and shows the preprocessor symbol declared by the `-mmcu` option.

Architecture	MCU name	Macro
avr1	at90s1200	__AVR_AT90S1200__
avr1	attiny11	__AVR_ATtiny11__
avr1	attiny12	__AVR_ATtiny12__
avr1	attiny15	__AVR_ATtiny15__
avr1	attiny28	__AVR_ATtiny28__
avr2	at90s2313	__AVR_AT90S2313__
avr2	at90s2323	__AVR_AT90S2323__
avr2	at90s2333	__AVR_AT90S2333__
avr2	at90s2343	__AVR_AT90S2343__
avr2	attiny22	__AVR_ATtiny22__
avr2	attiny24	__AVR_ATtiny24__
avr2	attiny25	__AVR_ATtiny25__
avr2	attiny26	__AVR_ATtiny26__
avr2	attiny261	__AVR_ATtiny261__
avr2	attiny44	__AVR_ATtiny44__
avr2	attiny45	__AVR_ATtiny45__
avr2	attiny461	__AVR_ATtiny461__
avr2	attiny84	__AVR_ATtiny84__
avr2	attiny85	__AVR_ATtiny85__
avr2	attiny861	__AVR_ATtiny861__
avr2	at90s4414	__AVR_AT90S4414__
avr2	at90s4433	__AVR_AT90S4433__
avr2	at90s4434	__AVR_AT90S4434__
avr2	at90s8515	__AVR_AT90S8515__
avr2	at90c8534	__AVR_AT90C8534__
avr2	at90s8535	__AVR_AT90S8535__
avr2	at86rf401	__AVR_AT86RF401__
avr2	attiny13	__AVR_ATtiny13__
avr2	attiny2313	__AVR_ATtiny2313__
avr3	atmega103	__AVR_ATmega103__
avr3	atmega603	__AVR_ATmega603__
avr3	at43usb320	__AVR_AT43USB320__
avr3	at43usb355	__AVR_AT43USB355__
avr3	at76c711	__AVR_AT76C711__
avr4	atmega48	__AVR_ATmega48__
avr4	atmega8	__AVR_ATmega8__
avr4	atmega8515	__AVR_ATmega8515__
avr4	atmega8535	__AVR_ATmega8535__
avr4	atmega88	__AVR_ATmega88__

Architecture	MCU name	Macro
avr4	at90pwm2	__AVR_AT90PWM2__
avr4	at90pwm3	__AVR_AT90PWM3__
avr5	at90can32	__AVR_AT90CAN32__
avr5	at90can64	__AVR_AT90CAN64__
avr5	at90can128	__AVR_AT90CAN128__
avr5	at90usb646	__AVR_AT90USB646__
avr5	at90usb647	__AVR_AT90USB647__
avr5	at90usb1286	__AVR_AT90USB1286__
avr5	at90usb1287	__AVR_AT90USB1287__
avr5	atmega128	__AVR_ATmega128__
avr5	atmega1280	__AVR_ATmega1280__
avr5	atmega1281	__AVR_ATmega1281__
avr5	atmega16	__AVR_ATmega16__
avr5	atmega161	__AVR_ATmega161__
avr5	atmega162	__AVR_ATmega162__
avr5	atmega163	__AVR_ATmega163__
avr5	atmega164p	__AVR_ATmega164P__
avr5	atmega165	__AVR_ATmega165__
avr5	atmega165p	__AVR_ATmega165P__
avr5	atmega168	__AVR_ATmega168__
avr5	atmega169	__AVR_ATmega169__
avr5	atmega169p	__AVR_ATmega169P__
avr5	atmega32	__AVR_ATmega32__
avr5	atmega323	__AVR_ATmega323__
avr5	atmega324p	__AVR_ATmega324P__
avr5	atmega325	__AVR_ATmega325__
avr5	atmega3250	__AVR_ATmega3250__
avr5	atmega329	__AVR_ATmega329__
avr5	atmega3290	__AVR_ATmega3290__
avr5	atmega406	__AVR_ATmega406__
avr5	atmega64	__AVR_ATmega64__
avr5	atmega640	__AVR_ATmega640__
avr5	atmega644	__AVR_ATmega644__
avr5	atmega644p	__AVR_ATmega644P__
avr5	atmega645	__AVR_ATmega645__
avr5	atmega6450	__AVR_ATmega6450__
avr5	atmega649	__AVR_ATmega649__
avr5	atmega6490	__AVR_ATmega6490__
avr5	at94k	__AVR_AT94K__
avr6	atmega2560	__AVR_ATmega2560__
avr6	atmega2561	__AVR_ATmega2561__

- `-morder1`

- `-morder2`

Change the order of register assignment. The default is

r24, r25, r18, r19, r20, r21, r22, r23, r30, r31, r26, r27, r28, r29, r17, r16, r15, r14, r13, r12, r11, r10, r9, r8, r7, r6, r5, r4, r3, r2, r0, r1

Order 1 uses

r18, r19, r20, r21, r22, r23, r24, r25, r30, r31, r26, r27, r28, r29, r17, r16, r15, r14, r13, r12, r11, r10, r9, r8, r7, r6, r5, r4, r3, r2, r0, r1

Order 2 uses

r25, r24, r23, r22, r21, r20, r19, r18, r30, r31, r26, r27, r28, r29, r17, r16, r15, r14, r13, r12, r11, r10, r9, r8, r7, r6, r5, r4, r3, r2, r1, r0

- `-mint8`

Assume `int` to be an 8-bit integer. Note that this is not really supported by `avr-libc`, so it should normally not be used. The default is to use 16-bit integers.

- `-mno-interrupts`

Generates code that changes the stack pointer without disabling interrupts. Normally, the state of the status register `SREG` is saved in a temporary register, interrupts are disabled while changing the stack pointer, and `SREG` is restored.

- `-mcall-prologues`

Use subroutines for function prologue/epilogue. For complex functions that use many registers (that needs to be saved/restored on function entry/exit), this saves some space at the cost of a slightly increased execution time.

- `-minit-stack=`*nnnn*

Set the initial stack pointer to *nnnn*. By default, the stack pointer is initialized to the symbol `__stack`, which is set to `RAMEND` by the run-time initialization code.

- `-mtiny-stack`

Change only the low 8 bits of the stack pointer.

- `-mno-tablejump`

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.

- `-mshort-calls`

Use `rjmp/rcall` (limited range) on >8K devices. On `avr2` and `avr4` architectures (less than 8 KB or flash memory), this is always the case. On `avr3` and `avr5` architectures, calls and jumps to targets outside the current function will by default use `jmp/call` instructions that can cover the entire address range, but that require more flash ROM and execution time.

- `-mrtl`

Dump the internal compilation result called "RTL" into comments in the generated assembler code. Used for debugging avr-gcc.

- `-msize`

Dump the address, size, and relative cost of each statement into comments in the generated assembler code. Used for debugging avr-gcc.

- `-mdeb`

Generate lots of debugging information to `stderr`.

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

- `-O`*n*

Optimization level *n*. Increasing *n* is meant to optimize more, an optimization level of 0 means no optimization at all, which is the default if no `-O` option is present. The special option `-Os` is meant to turn on all `-O2` optimizations that are not expected to increase code size.

Note that at `-O3`, gcc attempts to inline all "simple" functions. For the AVR target, this will normally constitute a large pessimization due to the code increasement. The only other optimization turned on with `-O3` is `-frename-registers`, which could rather be enabled manually instead.

A simple `-O` option is equivalent to `-O1`.

Note also that 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).

See also the **appropriate FAQ entry**(p. 202) for issues regarding debugging optimized code.

- `-Wa,`*assembler-options*
- `-Wl,`*linker-options*

Pass the listed options to the assembler, or linker, respectively.

- `-g`

Generate debugging information that can be used by avr-gdb.

- `-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 which makes some sense in a microcontroller environment where the application cannot meaningfully provide a return value to its environment (in most cases, `main()` won't even return anyway). However, this also turns off all optimizations normally done by the compiler which assume that functions known by a certain name behave as described by the standard. E. g., applying the function **strlen()**(p. 113) 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()**(p. 113) at run-time.

- `-funsigned-char`

Make any unqualfied `char` type an unsigned char. Without this option, they default to a signed char.

- `-funsigned-bitfields`

Make any unqualified bitfield type unsigned. By default, they are signed.

- `-fshort-enums`

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`

Pack all structure members together without holes.

8.10.2 Options for the assembler avr-as

8.10.2.1 Machine-specific assembler options

- `-mmcu=`*architecture*
- `-mmcu=`*MCU name*

avr-as understands the same `-mmcu=` options as **avr-gcc**(p. 252). By default, avr2 is assumed, but this can be altered by using the appropriate .arch pseudo-instruction inside the assembler source file.

- `-mall-opcodes`

Turns off opcode checking for the actual MCU type, 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 KB 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.

8.10.2.2 Examples for assembler options passed through the C compiler

Remember that assembler options can be passed from the C compiler frontend using `-Wa` (see **above**(p. 257)), so in order to include the C source code into the assembler listing in file `foo.lst`, when compiling `foo.c`, the following compiler command-line can be used:

```
$ avr-gcc -c -O foo.c -o foo.o -Wa,-ahls=foo.lst
```

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.S -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).

8.10.3 Controlling the linker avr-ld

8.10.3.1 Selected linker options While there are no machine-specific options for avr-ld, a number of the standard options might be of interest to AVR users.

- `-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 compile 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 location to look for archive libraries requested by `-l` options.

- `-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`.

- `-section-start` *sectionname=org*

Start section *sectionname* at absolute address *org*.

- `-Tbss` *org*

- `-Tdata` *org*
- `-Ttext` *org*

Start the `bss`, `data`, or `text` section at *org*, respectively.

- `-T` *scriptfile*

Use *scriptfile* as the linker script, replacing 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 avr5) with the suffix .x appended. They describe how the various **memory sections**(p. 239) will be linked together.

8.10.3.2 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 two system libraries, `libc.a`, and `libm.a`. While the standard library `libc.a` will always be searched for unresolved references when the linker is started using the C compiler frontend (i. e., there's always at least one implied `-lc` option), the mathematics library `libm.a` needs to be explicitly requested using `-lm`. See also the **entry in the FAQ**(p. 196) explaining this.

Conventionally, Makefiles use the `make` macro `LDLIBS` 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 **above**(p. 257). 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 -O -o foo.out -Wl,-Map=foo.map -Wl,--cref foo.c
```

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 -o foo.out -Wl,-Tdata,0x802000
```

See the explanation of the **data section**(p. 240) for why 0x800000 needs to be added to the actual value. Note that unless a `-minit-stack` option has been given when compiling the C source file that contains the function `main()`, 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()**(p. 102) will still be placed after all the variables in the data section, so in this situation, no stack/heap collision can occur.

8.11 Todo List

Group avr_boot(p. 9) From email with Marek: On smaller devices (all except ATmega64/128), __SPM_REG is in the I/O space, accessible with the shorter "in" and "out" instructions - since the boot loader has a limited size, this could be an important optimization.

8.12 Deprecated List

Global enable_external_int(p. 42)

Global INTERRUPT(p. 42)

Global timer_enable_int(p. 43)

Global inp(p. 42)

Global outp(p. 43)

Global inb(p. 42)

Global outb(p. 43)

Global sbi(p. 43)

Global cbi(p. 42)

Global SIGNAL(p. 144) Do not use anymore in new code.

Index