avr-libc Reference Manual

20021209cvs

Generated by Doxygen 1.2.18

Mon Dec 9 22:14:26 2002

Contents

1 AVR Libc

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://www.avr1.org/mailman/listinfo/avr-gcc-list/.` Before posting to the list, you might want to try reading the Frequently Asked Questions chapter of this document.

Note:

This document is a work in progress. As such, it may contain incorrect information. If you find a mistake, please send an email to the `avr-libc-dev@nongnu.org` describing the mistake. Also, send us an email if you find that a specific topic is missing from the document.

1.0.1 Supported Devices

The following is a list of AVR devices currently supported by the library.

AT90S Type Devices:

- at90s1200 [1]
- at90s2313
- at90s2323
- at90s2333
- at90s2343
- at90s4414
- at90s4433
- at90s4434
- at90s8515
- at90s8534
- at90s8535

ATmega Type Devices:

- atmega8
- atmega103
- atmega128
- atmega16
- atmega161
- atmega162
- atmega163
- atmega169
- atmega32
- atmega323
- atmega64 [untested]
- atmega8515 [untested]
- atmega8535 [untested]

ATtiny Type Devices:

- attiny10 [1]
- attiny11 [1]
- attiny12 [1]
- attiny15 [1]
- attiny22
- attiny26
- attiny28 [1]

Misc Devices:

- at94K [2]
- at76c711 [3]

Note:

[1] Assembly only. There is no support for these devices to be programmed in C since they do not have a ram based stack.

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. It seems to be supported by binutils and gcc, but is only partially supported by avr-libc. The missing piece seems to be `crt76711.o`.

2 avr-libc Module Index

2.1 avr-libc Modules

Here is a list of all modules:

3 avr-libc Data Structure Index

3.1 avr-libc Data Structures

Here are the data structures with brief descriptions:

4 avr-libc Page Index

4.1 avr-libc Related Pages

Here is a list of all related documentation pages:

5 avr-libc Module Documentation

5.1 EEPROM handling

5.1.1 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() before attempting any actual I/O.

avr-libc declarations

- #define eeprom_is_ready() bit_is_clear(EECR, EEWE)
- uint8_t eeprom_read_byte (uint8_t ∗addr)
- uint16_t eeprom_read_word (uint16_t ∗addr)
- void eeprom_write_byte (uint8_t ∗addr, uint8_t val)
- void eeprom_read_block (void ∗buf, void ∗addr, size_t n)

Backwards compatibility defines

- #define eeprom_rb(addr) eeprom_read_byte ((uint8_t ∗)(addr))
- #define eeprom_rw(addr) eeprom_read_word ((uint16_t ∗)(addr))
- #define eeprom_wb(addr, val) eeprom_write_byte ((uint8_t ∗)(addr), (uint8_t)(val))

IAR C compatibility defines

- #define _EEPUT(addr, val) eeprom_wb(addr, val)
- #define _EEGET(var, addr) (var) = eeprom_rb(addr)

5.1.2 Define Documentation

5.1.2.1 #define _EEGET(var, addr) (var) = eeprom_rb(addr)

Read a byte from EEPROM.

5.1.2.2 #define _EEPUT(addr, val) eeprom_wb(addr, val)

Write a byte to EEPROM.

5.1.2.3 #define eeprom_is_ready() bit_is_clear(EECR, EEWE)

Returns:

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

5.1.2.4 #define eeprom_rb(addr) eeprom_read_byte ((uint8_t *)(addr))

Deprecated:

Use eeprom_read_byte() in new programs.

5.1.2.5 #define eeprom_rw(addr) eeprom_read_word ((uint16_t *)(addr))

Deprecated:

Use eeprom_read_word() in new programs.

5.1.2.6 #define eeprom_wb(addr, val) eeprom_write_byte ((uint8_t *)(addr), (uint8_t)(val))

Deprecated:

Use eeprom_write_byte() in new programs.

5.1.3 Function Documentation

5.1.3.1 void eeprom_read_block (void * *buf*, void * *addr*, size_t *n*)

Read a block of `n` bytes from EEPROM address `addr` to `buf`.

5.1.3.2 uint8_t eeprom_read_byte (uint8_t * *addr*)

Read one byte from EEPROM address `addr`.

5.1.3.3 uint16_t eeprom_read_word (uint16_t ∗ *addr*)

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

5.1.3.4 void eeprom_write_byte (uint8_t ∗ *addr*, uint8_t *val*)

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

5.2 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.

Note that each of these files always includes

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

See Special function registers 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.

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.

5.3 Program Space String Utilities

5.3.1 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 function 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).

Note:

If you are working with strings which are completely based in ram, use the standard string functions described in Strings.

Defines

- #define PSTR(s) ({static char __c[ ] PROGMEM = (s); __c;})
- #define PGM_P const prog_char ∗
- #define PGM_VOID_P const prog_void ∗

Functions

- unsigned char __elpm_inline (unsigned long __addr) __ATTR_CONST__
- 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 strlen_P (PGM_P) __ATTR_CONST__
- int strncasecmp_P (const char ∗, PGM_P, size_t) __ATTR_PURE__
- int strncmp_P (const char ∗, PGM_P, size_t) __ATTR_PURE__
- char ∗ strncpy_P (char ∗, PGM_P, size_t)

5.3.2 Define Documentation

5.3.2.1 #define PGM_P const prog_char ∗

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

5.3.2.2 #define PGM_VOID_P const prog_void ∗

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

5.3.2.3 #define PSTR(s) ({static char __c[ ] PROGMEM = (s); __c;})

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

5.3.3 Function Documentation

5.3.3.1 unsigned char __elpm_inline (unsigned long *__addr*) `[static]`

Use this for access to >64K program memory (ATmega103, ATmega128), addr = RAMPZ:r31:r30

Note:

If possible, put your constant tables in the lower 64K and use "lpm" since it is more efficient that way, and you can still use the upper 64K for executable code.

5.3.3.2 void ∗ memcpy_P (void ∗ *dest*, PGM_VOID_P *src*, size_t *n*)

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

Returns:

The memcpy_P() function returns a pointer to dest.

5.3.3.3 int strcasecmp_P (const char ∗ *s1*, PGM_P *s2*)

Compare two strings ignoring case.

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

Parameters:

s1 A pointer to a string in the devices SRAM.

s2 A pointer to a string in the devices Flash.

Returns:

The strcasecmp_P() function returns an integer less than, equal to, or greater than zero if s1 is found, respectively, to be less than, to match, or be greater than s2.

5.3.3.4 char ∗ strcat_P (char ∗ *dest*, PGM_P *src*)

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

Returns:
The strcat() function returns a pointer to the resulting string *dest*.

5.3.3.5 int strcmp_P (const char ∗ *s1*, PGM_P *s2*)

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

Returns:
The strcmp_P() function returns an integer less than, equal to, or greater than zero if s1 is found, respectively, to be less than, to match, or be greater than s2.

5.3.3.6 char ∗ strcpy_P (char ∗ *dest*, PGM_P *src*)

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

Returns:
The strcpy_P() function returns a pointer to the destination string dest.

5.3.3.7 size_t strlen_P (PGM_P *src*)

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

Returns:
The strlen() function returns the number of characters in src.

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

Compare two strings ignoring case.

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

Parameters:
s1 A pointer to a string in the devices SRAM.
s2 A pointer to a string in the devices Flash.

n The maximum number of bytes to compare.

Returns:

The strcasecmp_P() function returns an integer less than, equal to, or greater than zero if s1 (or the first n bytes thereof) is found, respectively, to be less than, to match, or be greater than s2.

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

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

Returns:

The strncmp_P() function returns an integer less than, equal to, or greater than zero if s1 (or the first n bytes thereof) is found, respectively, to be less than, to match, or be greater than s2.

5.3.3.10 char ∗ strncpy_P (char ∗ *dest*, PGM_P *src*, size_t *n*)

The strncpy_P() function is similar to strcpy_P() except that not more than n bytes of src are copied. Thus, if there is no null byte among the first n bytes of src, the result will not be null-terminated.

In the case where the length of src is less than that of n, the remainder of dest will be padded with nulls.

Returns:

The strncpy_P() function returns a pointer to the destination string dest.

5.4 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/iom128.h>` to show how to define such macros:

```
#define PORTA _SFR_IO8(0x1b)
#define TCNT1 _SFR_IO16(0x2c)
#define PORTF _SFR_MEM8(0x61)
#define TCNT3 _SFR_MEM16(0x88)
```

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).

The old inp()/outp() macros are still supported, but not recommended to use in new code. The order of outp() arguments is confusing.

5.5 Power Management and Sleep Modes

5.5.1 Detailed Description

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

Use of the `SLEEP` instruction can allow your application to reduce it's power consumption considerably. AVR devices can be put into different sleep modes by changing the `SMn` bits of the `MCU` Control Register (`MCUCR`). Refer to the datasheet for the details relating to the device you are using.

Sleep Modes

Note:

FIXME: TRoth/2002-11-01: These modes were taken from the mega128 datasheet and might not be applicable or correct for all devices.

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

Sleep Functions

- void set_sleep_mode (uint8_t mode)
- void sleep_mode (void)

5.5.2 Define Documentation

5.5.2.1 #define SLEEP_MODE_ADC SM0

ADC Noise Reduction Mode.

5.5.2.2 #define SLEEP_MODE_EXT_STANDBY (SM0 | SM1 | SM2)

Extended Standby Mode.

5.5.2.3 #define SLEEP_MODE_IDLE 0

Idle mode.

5.5.2.4 #define SLEEP_MODE_PWR_DOWN SM1

Power Down Mode.

5.5.2.5 #define SLEEP_MODE_PWR_SAVE (SM0 | SM1)

Power Save Mode.

5.5.2.6 #define SLEEP_MODE_STANDBY (SM1 | SM2)

Standby Mode.

5.5.3 Function Documentation

5.5.3.1 void set_sleep_mode (uint8_t *mode*)

Set the bits in the `MCUCR` to select a sleep mode.

5.5.3.2 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() function. See the data sheet for your device for more details.

5.6 Character Operations

5.6.1 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() 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__

5.6.2 Function Documentation

5.6.2.1 int isalnum (int *__c*)

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

5.6.2.2 int isalpha (int *__c*)

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

5.6.2.3 int isascii (int *__c*)

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

5.6.2.4 int isblank (int *__c*)

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

5.6.2.5 int iscntrl (int *__c*)

Checks for a control character.

5.6.2.6 int isdigit (int __*c*)

Checks for a digit (0 through 9).

5.6.2.7 int isgraph (int __*c*)

Checks for any printable character except space.

5.6.2.8 int islower (int __*c*)

Checks for a lower-case character.

5.6.2.9 int isprint (int __*c*)

Checks for any printable character including space.

5.6.2.10 int ispunct (int __*c*)

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

5.6.2.11 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').

5.6.2.12 int isupper (int __*c*)

Checks for an uppercase letter.

5.6.2.13 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.

5.6.2.14 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.

5.6.2.15 int tolower (int *__c*)

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

5.6.2.16 int toupper (int *__c*)

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

5.7 System Errors (errno)

5.7.1 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

5.7.2 Define Documentation

5.7.2.1 #define EDOM 33

Domain error.

5.7.2.2 #define ERANGE 34

Range error.

5.8 Integer Types

5.8.1 Detailed Description

```
#include <inttypes.h>
```

Use [u]intN_t if you need exactly N bits.

Note:

If avr-gcc's `-mint8` option is used, no 32-bit types will be available.

5.9 Mathematics

5.9.1 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.

Defines

- #define M_PI 3.141592653589793238462643
- #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__
- double square (double __x) __ATTR_CONST__
- double inverse (double) __ATTR_CONST__

5.9.2 Define Documentation

5.9.2.1 #define M_PI 3.141592653589793238462643

The constant `pi`.

5.9.2.2 #define M_SQRT2 1.4142135623730950488016887

The square root of 2.

5.9.3 Function Documentation

5.9.3.1 double acos (double *__x*)

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

5.9.3.2 double asin (double *__x*)

The asin() 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].

5.9.3.3 double atan (double *__x*)

The atan() 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].

5.9.3.4 double atan2 (double *__y*, double *__x*)

The atan2() function computes the principal value of the arc tangent of `y / x`, using the signs of both arguments to determine the quadrant of the return value. The returned value is in the range [-pi, +pi] radians. If both `x` and `y` are zero, the global variable `errno` is set to `EDOM`.

5.9.3.5 double ceil (double *__x*)

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

5.9.3.6 double cos (double *__x*)

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

5.9.3.7 double cosh (double *__x*)

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

5.9.3.8 double exp (double *_x*)

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

5.9.3.9 double fabs (double *__x*)

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

5.9.3.10 double floor (double *__x*)

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

5.9.3.11 double fmod (double *__x*, double *__y*)

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

5.9.3.12 double frexp (double *__value*, int ∗ *__exp*)

The frexp() function breaks a floating-point number into a normalized fraction and an integral power of 2. It stores the integer in the `int` object pointed to by `exp`.

The frexp() 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.

5.9.3.13 double inverse (double)

The function inverse() returns `1 / x`.

Note:

This function does not belong to the C standard definition.

5.9.3.14 double ldexp (double *__x*, int *__exp*)

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

The ldexp() 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.

5.9.3.15 double log (double *__x*)

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

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

5.9.3.16 double log10 (double *__x*)

The log() function returns the logarithm of argument `x` to base 10.

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

5.9.3.17 double modf (double *__value*, double ∗ *__iptr*)

The modf() 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() function returns the signed fractional part of `value`.

5.9.3.18 double pow (double *__x*, double *__y*)

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

5.9.3.19 double sin (double *__x*)

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

5.9.3.20 double sinh (double *__x*)

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

5.9.3.21 double sqrt (double *__x*)

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

5.9.3.22 double square (double *__x*)

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

Note:

This function does not belong to the C standard definition.

5.9.3.23 double tan (double *__x*)

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

5.9.3.24 double tanh (double *__x*)

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

5.10 Setjmp and Longjmp

5.10.1 Detailed Description

While the C language has the dreaded `goto` statement, it can only be used to jump to a label in the same (local) function. In order to jump directly to another (non-local) function, the C library provides the setjmp() and longjmp() functions. setjmp() and longjmp() are useful for dealing with errors and interrupts encountered in a low-level subroutine of a program.

Note:

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

For a very detailed discussion of setjmp()/longjmp(), see Chapter 7 of *Advanced Programming in the UNIX Environment*, by W. Richard Stevens.

Example:

```
#include <setjmp.h>

jmp_buf env;

int main (void)
{
    if (setjmp (env))
    {
        ... handle error ...
    }

    while (1)
    {
       ... main processing loop which calls foo() 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__

5.10.2 Function Documentation

5.10.2.1 void longjmp (jmp_buf *__jmpb*, int *__ret*)

Non-local jump to a saved stack context.

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

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

Note:

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

Parameters:

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

__ret Value to return to the caller of setjmp().

Returns:

This function never returns.

5.10.2.2 int setjmp (jmp_buf *__jmpb*)

Save stack context for non-local goto.

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

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

Parameters:

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

Returns:

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

5.11 Standard IO facilities

5.11.1 Detailed Description

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

Warning:

This implementation of the standard IO facilities is new to avr-libc. It is not yet expected to remain stable, so some aspects of the API might change in a future release.

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's 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` family, although usually associated with presumably simple things like the famous "Hello, world!" program, is actually a fairly complex one which causes quite some amount of code space to be taken, and it's not fast either 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.

In order to allow programmers a code size vs. functionality tradeoff, the function vfprintf() which is the heart of the printf family can be selected in different flavours using linker options. See the documentation of vfprintf() for a detailed description.

The 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's no notion of "file" whatsoever to avr-libc, there's no function `fopen()` that could be used to associate a stream to some device. (See note 1.) Instead, function `fdevopen()` is provided to associate a stream to a device, where the device needs to provide a function to send a character, to receive a character, or both. There's 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).

For convenience, the first call to `fdevopen()` that opens a stream for reading will cause the resulting stream to be aliased to `stdin`. Likewise, the first call to `fdevopen()` that opens a stream for writing will cause the resulting stream to be aliased to both, `stdout`, and `stderr`. (Thus, if the open was done with both, read and write intent, all three standard streams will be identical.) Note that these aliases are indistinguishable from each other, thus calling `fclose()` on such a stream will effectively also close all of its aliases (note 3).

All the `printf` 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 "_P" appended where the format string is expected to reside in the flash ROM. The macro `PSTR` (explained in Program Space String Utilities) will become very handy to declare these format strings.

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 were to be provided by the application, this approach has not been taken.

Note 2:

This basically follows the Unix approach: if a device such as a terminal needs special handling, it is in the domain of the terminal device driver to provide this functionality. Thus, a simple function suitable as `put()` for `fdevopen()` that talks to a UART interface might look like this:

```
int
uart_putchar(char c)
{

  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()` 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 putc(__c, __stream) fputc(__c, __stream)
- #define putchar(__c) fputc(__c, stdout)

Functions

- FILE ∗ fdevopen (int(∗__put)(char), int(∗__get)(void), int __opts)
- int fclose (FILE ∗__stream)
- int vfprintf (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 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 fprintf (FILE ∗__stream, const char ∗__fmt,...)
- int fprintf_P (FILE ∗__stream, const char ∗__fmt,...)

5.11.2 Define Documentation

5.11.2.1 #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.

5.11.2.2 #define FILE struct __file

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

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

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

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

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

5.11.2.5 #define stderr (__iob[2])

Stream destined for error output. Unless specifically assigned, identical to `stdout`.

If `stderr` should point to another stream, the result of another `fdevopen()` must be explicitly assigned to it without closing the previous `stderr` (since this would also close `stdout`).

5.11.2.6 #define stdin (__iob[0])

Stream that will be used as an input stream by the simplified functions that don't take a `stream` argument.

The first stream opened with read intent using `fdevopen()` will be assigned to `stdin`.

5.11.2.7 #define stdout (__iob[1])

Stream that will be used as an output stream by the simplified functions that don't take a `stream` argument.

The first stream opened with write intent using `fdevopen()` will be assigned to both, `stdin`, and `stderr`.

5.11.3 Function Documentation

5.11.3.1 int fclose (FILE ∗ *__stream*)

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

It currently always returns 0 (for success).

5.11.3.2 FILE∗ fdevopen (int(∗ *__put*)(char), int(∗ *__get*)(void), int *__opts*)

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 one character to write to the device as argument , 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 no arguments, 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 `-1`.

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`.

The third parameter `opts` is currently unused, but reserved for future extensions.

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

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

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

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

5.11.3.5 int fputc (int *__c*, FILE ∗ *__stream*)

The function `fputc` sends the character `(though` given as type `int)` to `stream`. It returns the character, or `EOF` in case an error occurred.

5.11.3.6 int printf (const char ∗ *__fmt*, ...)

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

5.11.3.7 int printf_P (const char ∗ *__fmt*, ...)

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

5.11.3.8 int snprintf (char ∗ *__s*, size_t *__n*, const char ∗ *__fmt*, ...)

Like `sprintf()`, but instead of assuming `s` to be of infinite size, no more than `n` characters (including the trailing NUL character) will be converted to `s`.

Returns the number of characters that would have been written to `s` if there were enough space.

5.11.3.9 int snprintf_P (char ∗ __s, size_t __n, const char ∗ __fmt, ...)

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

5.11.3.10 int sprintf (char ∗ __s, const char ∗ __fmt, ...)

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

5.11.3.11 int sprintf_P (char ∗ __s, const char ∗ __fmt, ...)

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

5.11.3.12 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 correspond properly (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-adjust173 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 con173 versions.
- 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.ddde177dd"` 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.

In no case does a non-existent or small field width cause truncation of a numeric field; if the result of a conversion is wider than the field width, the field is expanded to contain the conversion result.

Since the full implementation of all the mentioned features becomes fairly large, three different flavours of vfprintf() can be selected using linker options. The default vfprintf() implements all the mentioned functionality except floating point conversions. A minimized version of vfprintf() 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:

```
-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.

5.12 General utilities

5.12.1 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 * itoa (int __val, char *__s, int __radix)
- 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)

Conversion functions for double arguments.

Note that these functions are not located in the default library, `libc.a`, but in the mathematical library, `libm.a`. So when linking the application, the `-lm` option needs to be specified.

- #define DTOSTR_ALWAYS_SIGN 0x01
- #define DTOSTR_PLUS_SIGN 0x02
- #define DTOSTR_UPPERCASE 0x04
- char * dtostre (double __val, char *__s, unsigned char __prec, unsigned char __flags)
- char * dtostrf (double __val, char __width, char __prec, char *__s)

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__
- double strtod (const char ∗__nptr, char ∗∗__endptr)
- 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

5.12.2 Define Documentation

5.12.2.1 #define DTOSTR_ALWAYS_SIGN 0x01

Bit value that can be passed in `flags` to dtostre().

5.12.2.2 #define DTOSTR_PLUS_SIGN 0x02

Bit value that can be passed in `flags` to dtostre().

5.12.2.3 #define DTOSTR_UPPERCASE 0x04

Bit value that can be passed in `flags` to dtostre().

5.12.2.4 #define RAND_MAX 0x7FFF

Highest number that can be generated by rand().

5.12.2.5 #define RANDOM_MAX 0x7FFFFFFF

Highest number that can be generated by random().

5.12.3 Typedef Documentation

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

Comparision function type for qsort(), just for convenience.

5.12.4 Function Documentation

5.12.4.1 __inline__ void abort (void)

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

5.12.4.2 int abs (int *__i*)

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

Note:

The abs() and labs() functions are builtins of gcc.

5.12.4.3 __inline__ int atoi (const char ∗ *__nptr*)

The atoi() 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);
```

5.12.4.4 __inline__ long atol (const char ∗ *__nptr*)

The atol() function converts the initial portion of the string pointed to by `nptr` to long integer representation.

It is equivalent to:

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

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

The bsearch() function searches an array of `nmemb` objects, the initial member of which is pointed to by `base`, for a member that matches the object pointed to by `key`. The size of each member of the array is specified by `size`.

The contents of the array should be in ascending sorted order according to the comparison function referenced by `compar`. The `compar` routine is expected to have two arguments which point to the key object and to an array member, in that order, and should return an integer less than, equal to, or greater than zero if the key object is found, respectively, to be less than, to match, or be greater than the array member.

The bsearch() function returns a pointer to a matching member of the array, or a null pointer if no match is found. If two members compare as equal, which member is matched is unspecified.

5.12.4.6 void∗ calloc (size_t *__nele*, size_t *__size*)

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

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

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

5.12.4.8 char∗ dtostre (double *__val*, char ∗ *__s*, unsigned char *__prec*, unsigned char *__flags*)

The dtostre() function converts the double value passed in `val` into an ASCII representation that will be stored under `s`. The caller is responsible for providing sufficient storage in `s`.

Conversion is done in the format `"[-]d.ddde177dd"` where there is one digit before the decimal-point character and the number of digits after it is equal to the precision `prec`; if the precision is zero, no decimal-point character appears. If `flags` has the DTOSTRE_UPPERCASE bit set, the letter ’`E`’ (rather than ’`e`’) will be used to introduce the exponent. The exponent always contains two digits; if the value is zero, the exponent is `"00"`.

If `flags` has the DTOSTRE_ALWAYS_SIGN bit set, a space character will be placed into the leading position for positive numbers.

If `flags` has the DTOSTRE_PLUS_SIGN bit set, a plus sign will be used instead of a space character in this case.

5.12.4.9 char∗ dtostrf (double *__val*, char *__width*, char *__prec*, char ∗ *__s*)

The dtostrf() function converts the double value passed in `val` into an ASCII representationthat will be stored under `s`. The caller is responsible for providing sufficient storage in `s`.

Conversion is done in the format `"[-]d.ddd"`. The minimum field width of the output string (including the '.' `and` the possible sign for negative values) is given in `width`, and `prec` determines the number of digits after the decimal sign.

5.12.4.10 void exit (int *__status*)

The exit() function terminates the application. Since there is no environment to return to, `status` is ignored, and code execution will eventually reach an infinite loop, thereby effectively halting all code processing.

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

5.12.4.11 void free (void ∗ *__ptr*)

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

5.12.4.12 char∗ itoa (int *__val*, char ∗ *__s*, int *__radix*)

The function itoa() converts the integer value from `val` into an ASCII representation that will be stored under `s`. The caller is responsible for providing sufficient storage in `s`.

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 itoa() function returns the pointer passed as `s`.

5.12.4.13 long labs (long *__i*)

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

Note:

The abs() and labs() functions are builtins of gcc.

5.12.4.14 ldiv_t ldiv (long *__num*, long *__denom*)

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

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

The function ltoa() converts the long integer value from `val` into an ASCII representation that will be stored under `s`. The caller is responsible for providing sufficient storage in `s`.

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 ltoa() function returns the pointer passed as `s`.

5.12.4.16 void* malloc (size_t *__size*)

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

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

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

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

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

The qsort() function sorts an array of `nmemb` objects, the initial member of which is pointed to by `base`. The size of each object is specified by `size`. The contents of the array base are sorted in ascending order according to a comparison function pointed to by `compar`, which requires two arguments pointing to the objects being compared.

The comparison function must return an integer less than, equal to, or greater than zero if the first argument is considered to be respectively less than, equal to, or greater than the second.

5.12.4.18 int rand (void)

The rand() function computes a sequence of pseudo-random integers in the range of 0 to `RAND_MAX` (as defined by the header file <stdlib.h>).

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

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

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()` for an alternate set of functions that retains full 32-bit precision.

5.12.4.19 int rand_r (unsigned long * *ctx*)

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

5.12.4.20 long random (void)

The random() function computes a sequence of pseudo-random integers in the range of 0 to `RANDOM_MAX` (as defined by the header file <stdlib.h>).

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

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

5.12.4.21 long random_r (unsigned long * *ctx*)

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

5.12.4.22 void srand (unsigned int *__seed*)

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

5.12.4.23 void srandom (unsigned long *__seed*)

Pseudo-random number generator seeding; see random().

5.12.4.24 double strtod (const char * *__nptr*, char ** *__endptr*)

The strtod() 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() 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?

Note:
Implemented but not tested.

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

The strtol() function converts the string in nptr to a long value. The conversion is done according to the given base, which must be between 2 and 36 inclusive, or be the special value 0.

The string may begin with an arbitrary amount of white space (as determined by isspace()) followed by a single optional '+' or '-' sign. If base is zero or 16, the string may then include a "0x" 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() stores the address of the first invalid character in *endptr. If there were no digits at all, however, strtol() stores the original value of nptr in endptr. (Thus, if *nptr is not '\0' but **endptr is '\0' on return, the entire string was valid.)

The strtol() function returns the result of the conversion, unless the value would underflow or overflow. If no conversion could be performed, 0 is returned. If an overflow or underflow occurs, errno is set to ERANGE and the function return value is clamped to LONG_MIN or LONG_MAX, respectively.

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

The strtoul() function converts the string in nptr to an unsigned long value. The conversion is done according to the given base, which must be between 2 and 36 inclusive, or be the special value 0.

The string may begin with an arbitrary amount of white space (as determined by isspace()) followed by a single optional '+' or '-' sign. If base is zero or 16, the string may then include a "0x" 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() stores the address of the first invalid character in ∗`endptr`. If there were no digits at all, however, strtoul() stores the original value of `nptr` in `endptr`. (Thus, if ∗`nptr` is not '\`0`' but ∗∗`endptr` is '\`0`' on return, the entire string was valid.)

The strtoul() function 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, strtoul() returns ULONG_MAX, and `errno` is set to ERANGE. If no conversion could be performed, 0 is returned.

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

The function ultoa() converts the unsigned long integer value from `val` into an ASCII representation that will be stored under `s`. The caller is responsible for providing sufficient storage in `s`.

Conversion is done using the `radix` as base, which may be a number between 2 (binary conversion) and up to 36. If `radix` is greater than 10, the next digit after '`9`' will be the letter '`a`'.

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

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

The function utoa() converts the unsigned integer value from `val` into an ASCII representation that will be stored under `s`. The caller is responsible for providing sufficient storage in `s`.

Conversion is done using the `radix` as base, which may be a number between 2 (binary conversion) and up to 36. If `radix` is greater than 10, the next digit after '`9`' will be the letter '`a`'.

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

5.12.5 Variable Documentation

5.12.5.1 char∗ __malloc_heap_end

`malloc()` tunable.

5.12.5.2 char∗ __malloc_heap_start

`malloc()` tunable.

5.12.5.3 size_t __malloc_margin

`malloc()` tunable.

5.13 Strings

5.13.1 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 Program Space String Utilities.

Functions

- 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)
- 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 ∗ strstr (const char ∗, const char ∗) __ATTR_PURE__
- char ∗ strupr (char ∗)

5.13.2 Function Documentation

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

Copy memory area.

The memccpy() function copies no more than len bytes from memory area src to memory area dest, stopping when the character val is found.

Returns:

The memccpy() function returns a pointer to the next character in dest after val, or NULL if val was not found in the first len characters of src.

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

Scan memory for a character.

The memchr() function scans the first len bytes of the memory area pointed to by src for the character val. The first byte to match val (interpreted as an unsigned character) stops the operation.

Returns:

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

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

Compare memory areas.

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

Returns:

The memcmp() function returns an integer less than, equal to, or greater than zero if the first len bytes of s1 is found, respectively, to be less than, to match, or be greater than the first len bytes of s2.

Note:

Be sure to store the result in a 16 bit variable since you may get incorrect results if you use an unsigned char or char due to truncation.

Warning:

This function is not -mint8 compatible, although if you only care about testing for equality, this function should be safe to use.

5.13.2.4 void ∗ memcpy (void ∗ *dest*, const void ∗ *src*, size_t *len*)

Copy a memory area.

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

Returns:

The memcpy() function returns a pointer to dest.

5.13.2.5 void ∗ memmove (void ∗ *dest*, const void ∗ *src*, size_t *len*)

Copy memory area.

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

Returns:

The memmove() function returns a pointer to dest.

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

Fill memory with a constant byte.

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

Returns:

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

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

Compare two strings ignoring case.

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

Returns:

The strcasecmp() function returns an integer less than, equal to, or greater than zero if s1 is found, respectively, to be less than, to match, or be greater than s2.

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

Concatenate two strings.

The strcat() function appends the src string to the dest string overwriting the '\0' character at the end of dest, and then adds a terminating '\0' character. The strings may not overlap, and the dest string must have enough space for the result.

Returns:

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

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

Locate character in string.

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

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

Compare two strings.

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

Returns:

The strcmp() function returns an integer less than, equal to, or greater than zero if s1 is found, respectively, to be less than, to match, or be greater than s2.

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

Copy a string.

The strcpy() function copies the string pointed to by src (including the terminating '\0' character) to the array pointed to by dest. The strings may not overlap, and the destination string dest must be large enough to receive the copy.

Returns:

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

Note:

If the destination string of a strcpy() 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.

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

Concatenate two strings.

Appends src to string dst of size siz (unlike strncat(), siz is the full size of dst, not space left). At most siz-1 characters will be copied. Always NULL terminates (unless siz <= strlen(dst)).

Returns:

The strlcat() function returns strlen(src) + MIN(siz, strlen(initial dst)). If retval >= siz, truncation occurred.

5.13.2.13 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() function returns strlen(src). If retval >= siz, truncation occurred.

5.13.2.14 size_t strlen (const char ∗ *src*)

Calculate the length of a string.

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

Returns:

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

5.13.2.15 char ∗ strlwr (char ∗ *string*)

Convert a string to lower case.

The strlwr() function will convert a string to lower case. Only the upper case alphabetic characters [A .. Z] are converted. Non-alphabetic characters will not be changed.

Returns:

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

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

Compare two strings ignoring case.

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

Returns:

The strncasecmp() 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.

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

Concatenate two strings.

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

Returns:

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

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

Compare two strings.

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

Returns:

The strncmp() function returns an integer less than, equal to, or greater than zero if s1 (or the first n bytes thereof) is found, respectively, to be less than, to match, or be greater than s2.

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

Copy a string.

The strncpy() function is similar to strcpy(), except that not more than 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() function returns a pointer to the destination string dest.

5.13.2.20 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.

5.13.2.21 char ∗ strrchr (const char ∗ *src*, int *val*)

Locate character in string.

The strrchr() function returns a pointer to the last occurrence of the character val in the string src.

Here "character" means "byte" - these functions do not work with wide or multi-byte characters.

Returns:

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

5.13.2.22 char ∗ strrev (char ∗ *string*)

Reverse a string.

The strrev() function reverses the order of the string.

Returns:

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

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

Locate a substring.

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

Returns:

The strstr() function returns a pointer to the beginning of the substring, or NULL if the substring is not found.

5.13.2.24 char ∗ strupr (char ∗ *string*)

Convert a string to upper case.

The strupr() function will convert a string to upper case. Only the lower case alphabetic characters [a .. z] are converted. Non-alphabetic characters will not be changed.

Returns:

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

5.14 Interrupts and Signals

5.14.1 Detailed Description

Note:

This discussion of interrupts and signals was taken from Rich Neswold's document. See Acknowledgments.

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__((interrupt))`.

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 one of two macros, INTERRUPT() and SIGNAL(). These macros register and mark the routine as an interrupt handler for the specified peripheral. The following is an example definition of a handler for the ADC interrupt.

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

INTERRUPT(SIG_ADC)
{
    // user code here
}
```

Refer to the chapter explaining assembler programming for an explanation about interrupt routines written solely in assembler language.

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 either SIGNAL() or INTERRUPT() as such.

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

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

The interrupt is chosen by supplying one of the symbols in following table. Note that every AVR device has a different interrupt vector table so some signals might not be available. Check the data sheet for the device you are using.

[FIXME: Fill in the blanks! Gotta read those durn data sheets ;-)]

Note:

The SIGNAL() and INTERRUPT() macros currently cannot spell-check the argument passed to them. Thus, by misspelling one of the names below in a call to SIGNAL() or INTERRUPT(), a function will be created that, while possibly being usable as an interrupt function, is not actually wired into the interrupt vector table. No warning will be given about this situation.

Signal Name	**Description**
SIG_2WIRE_SERIAL	2-wire serial interface (aka. I178C [tm])
SIG_ADC	ADC Conversion complete
SIG_COMPARATOR	Analog Comparator Interrupt
SIG_EEPROM_READY	Eeprom ready
SIG_FPGA_INTERRUPT0	
SIG_FPGA_INTERRUPT1	
SIG_FPGA_INTERRUPT2	
SIG_FPGA_INTERRUPT3	
SIG_FPGA_INTERRUPT4	
SIG_FPGA_INTERRUPT5	
SIG_FPGA_INTERRUPT6	
SIG_FPGA_INTERRUPT7	
SIG_FPGA_INTERRUPT8	
SIG_FPGA_INTERRUPT9	
SIG_FPGA_INTERRUPT10	
SIG_FPGA_INTERRUPT11	
SIG_FPGA_INTERRUPT12	
SIG_FPGA_INTERRUPT13	
SIG_FPGA_INTERRUPT14	
SIG_FPGA_INTERRUPT15	
SIG_INPUT_CAPTURE1	Input Capture1 Interrupt
SIG_INPUT_CAPTURE3	Input Capture3 Interrupt
SIG_INTERRUPT0	External Interrupt0
SIG_INTERRUPT1	External Interrupt1
SIG_INTERRUPT2	External Interrupt2

Signal Name	Description
SIG_INTERRUPT3	External Interrupt3
SIG_INTERRUPT4	External Interrupt4
SIG_INTERRUPT5	External Interrupt5
SIG_INTERRUPT6	External Interrupt6
SIG_INTERRUPT7	External Interrupt7
SIG_OUTPUT_COMPARE0	Output Compare0 Interrupt
SIG_OUTPUT_COMPARE1A	Output Compare1(A) Interrupt
SIG_OUTPUT_COMPARE1B	Output Compare1(B) Interrupt
SIG_OUTPUT_COMPARE1C	Output Compare1(C) Interrupt
SIG_OUTPUT_COMPARE2	Output Compare2 Interrupt
SIG_OUTPUT_COMPARE3A	Output Compare3(A) Interrupt
SIG_OUTPUT_COMPARE3B	Output Compare3(B) Interrupt
SIG_OUTPUT_COMPARE3C	Output Compare3(C) Interrupt
SIG_OVERFLOW0	Overflow0 Interrupt
SIG_OVERFLOW1	Overflow1 Interrupt
SIG_OVERFLOW2	Overflow2 Interrupt
SIG_OVERFLOW3	Overflow3 Interrupt
SIG_PIN	
SIG_PIN_CHANGE0	
SIG_PIN_CHANGE1	
SIG_RDMAC	
SIG_SPI	SPI Interrupt
SIG_SPM_READY	Store program memory ready
SIG_SUSPEND_RESUME	
SIG_TDMAC	
SIG_UART0	
SIG_UART0_DATA	UART(0) Data Register Empty Interrupt
SIG_UART0_RECV	UART(0) Receive Complete Interrupt
SIG_UART0_TRANS	UART(0) Transmit Complete Interrupt
SIG_UART1	
SIG_UART1_DATA	UART(1) Data Register Empty Interrupt
SIG_UART1_RECV	UART(1) Receive Complete Interrupt
SIG_UART1_TRANS	UART(1) Transmit Complete Interrupt
SIG_UART_DATA	UART Data Register Empty Interrupt
SIG_UART_RECV	UART Receive Complete Interrupt
SIG_UART_TRANS	UART Transmit Complete Interrupt
SIG_USART0_DATA	USART(0) Data Register Empty Interrupt
SIG_USART0_RECV	USART(0) Receive Complete Interrupt
SIG_USART0_TRANS	USART(0) Transmit Complete Interrupt
SIG_USART1_DATA	USART(1) Data Register Empty Interrupt
SIG_USART1_RECV	USART(1) Receive Complete Interrupt
SIG_USART1_TRANS	USART(1) Transmit Complete Interrupt
SIG_USB_HW	

Global manipulation of the interrupt flag

The global interrupt flag is maintained in the I bit of the status register (SREG).

- #define sei() __asm__ __volatile__ ("sei" ::)
- #define cli() __asm__ __volatile__ ("cli" ::)

Macros for writing interrupt handler functions

- #define SIGNAL(signame)
- #define INTERRUPT(signame)

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.

- void enable_external_int (unsigned char ints)
- void timer_enable_int (unsigned char ints)

5.14.2 Define Documentation

5.14.2.1 #define cli() __asm__ __volatile__ ("cli" ::)

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

Disables all interrupts by clearing the global interrupt mask. This function actually compiles into a single line of assembly, so there is no function call overhead.

5.14.2.2 #define INTERRUPT(signame)

Value:

```
void signame (void) __attribute__ ((interrupt));        \
void signame (void)
```

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

Introduces an interrupt handler function that runs with global interrupts initially enabled. This allows interrupt handlers to be interrupted.

5.14.2.3 #define sei() __asm__ __volatile__ ("sei" ::)

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

Enables interrupts by clearing the global interrupt mask. This function actually compiles into a single line of assembly, so there is no function call overhead.

5.14.2.4 #define SIGNAL(signame)

Value:

```
void signame (void) __attribute__ ((signal));           \
void signame (void)
```

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

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

5.14.3 Function Documentation

5.14.3.1 void enable_external_int (unsigned char *ints*)

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

This function gives access to the `gimsk` register (or `eimsk` register if using an AVR Mega device). Although this function is essentially the same as using the outb() function, it does adapt slightly to the type of device being used.

5.14.3.2 void timer_enable_int (unsigned char *ints*)

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

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

5.15 Special function registers

5.15.1 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()`:

```
#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. Whitout 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?.

Modules

- Additional notes from <avr/sfr_defs.h>

Bit manipulation

- #define _BV(bit) (1 << (bit))

IO operations

- #define inb(sfr) _SFR_BYTE(sfr)
- #define inw(sfr) _SFR_WORD(sfr)
- #define outb(sfr, val) (_SFR_BYTE(sfr) = (val))
- #define outw(sfr, val) (_SFR_WORD(sfr) = (val))

IO register bit manipulation

- #define cbi(sfr, bit) (_SFR_BYTE(sfr) &= ~_BV(bit))
- #define sbi(sfr, bit) (_SFR_BYTE(sfr) |= _BV(bit))
- #define bit_is_set(sfr, bit) (inb(sfr) & _BV(bit))
- #define bit_is_clear(sfr, bit) (~inb(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))

Deprecated Macros

- #define outp(val, sfr) outb(sfr, val)
- #define inp(sfr) inb(sfr)
- #define BV(bit) _BV(bit)

5.15.2 Define Documentation

5.15.2.1 #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().

5.15.2.2 #define bit_is_clear(sfr, bit) (~inb(sfr) & _BV(bit))

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

Test whether bit `bit` in IO register `sfr` is clear.

5.15.2.3 #define bit_is_set(sfr, bit) (inb(sfr) & _BV(bit))

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

Test whether bit `bit` in IO register `sfr` is set.

5.15.2.4 #define BV(bit) _BV(bit)

Deprecated:
For backwards compatibility only. This macro will eventually be removed.

Use _BV() in new programs.

5.15.2.5 #define cbi(sfr, bit) (_SFR_BYTE(sfr) &= ~_BV(bit))

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

Clear bit `bit` in IO register `sfr`.

5.15.2.6 #define inb(sfr) _SFR_BYTE(sfr)

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

Read a byte from IO register `sfr`.

5.15.2.7 #define inp(sfr) inb(sfr)

Deprecated:
For backwards compatibility only. This macro will eventually be removed.

Use inb() in new programs.

5.15.2.8 #define inw(sfr) _SFR_WORD(sfr)

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

Read a 16-bit word from IO register pair `sfr`.

5.15.2.9 #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.

5.15.2.10 #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.

5.15.2.11 #define outb(sfr, val) (_SFR_BYTE(sfr) = (val))

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

Write `val` to IO register `sfr`.

Note:
The order of the arguments was switched in older versions of avr-libc (versions <= 20020203).

5.15.2.12 #define outp(val, sfr) outb(sfr, val)

Deprecated:
For backwards compatibility only. This macro will eventually be removed.

Use outb() in new programs.

5.15.2.13 #define outw(sfr, val) (_SFR_WORD(sfr) = (val))

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

Write the 16-bit value `val` to IO register pair `sfr`. Care will be taken to write the lower register first. When used to update 16-bit registers where the timing is critical and the operation can be interrupted, the programmer is the responsible for disabling interrupts before accessing the register pair.

Note:
The order of the arguments was switched in older versions of avr-libc (versions <= 20020203).

5.15.2.14 #define sbi(sfr, bit) (_SFR_BYTE(sfr) |= _BV(bit))

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

Set bit `bit` in IO register `sfr`.

6 avr-libc Data Structure Documentation

6.1 div_t Struct Reference

6.1.1 Detailed Description

Result type for function div().

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

- stdlib.h

6.2 ldiv_t Struct Reference

6.2.1 Detailed Description

Result type for function ldiv().

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

- stdlib.h

7 avr-libc Page Documentation

7.1 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**.
- Theodore A. Roth [`troth@verinet.com`] for setting up avr-libc's CVS repository, bootstrapping the documentation project using doxygen, and continued maintenance of the project on `http://savannah.gnu.org/projects/avr-libc`
- 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 demo project.
- Brian Dean [`bsd@bsdhome.com`] for developing **avrprog** (an alternate to **uisp**) and for contributing documentation which describes how to use it.
- All the people who 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. ;-)

7.2 avr-libc and assembler programs

7.2.1 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 facility of the compiler.

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

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

7.2.2 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.

7.2.3 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 SIG_OVERFLOW0                   ; Note [7]
SIG_OVERFLOW0:
        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 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.

Note [8]

As explained in the section about special function registers, the actual IO port address should be obtained using the macro `_SFR_IO_ADDR`. (The AT90S1200 does not have RAM thus the memory-mapped approach to access the IO registers is not available. It would be slower than using `in` / `out` instructions anyway.)
Since the operation to reload `TCCNT0` is time-critical, it is even performed before saving `SREG`. Obviously, this requires that the instructions involved would not change any of the flag bits in `SREG`.

Note [9]

Interrupt routines must not clobber the global CPU state. Thus, it is usually necessary to save at least the state of the flag bits in `SREG`. (Note that this serves as an example here only since actually, all the following instructions would not modify `SREG` either, but that's not commonly the case.)
Also, it must be made sure that registers used inside the interrupt routine do not conflict with those used outside. In the case of a RAM-less device like the AT90S1200, this can only be done by agreeing on a set of registers to be used exclusively inside the interrupt routine; there would not be any other chance to "save" a register anywhere.
If the interrupt routine is to be linked together with C modules, care must be taken to follow the register usage guidelines imposed by the C compiler. Also, any register modified inside the interrupt sevice needs to be saved, usually on the stack.

Note [10]

As explained in Interrupts and Signals, 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.)

7.3 Frequently Asked Questions

7.3.1 FAQ Index

7.3.2 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;
...

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

the compiler will typically optimize the access to `flag` 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.

7.3.3 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.

7.3.4 How to permanently bind a variable to a register?

This can be done with

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

See C Names Used in Assembler Code for more details.

Back to FAQ Index.

7.3.5 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, including how to use them from C code, see Memory Sections.

Back to FAQ Index.

7.3.6 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 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()` call), so the definitions cannot usefully be made as byte values in the first place.

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

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

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

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

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

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

Back to FAQ Index.

7.3.7 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.

7.3.8 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). With respect to the standard, this sentence is somewhat simplified (because the standard allows for machines where the actual bit pattern used differs from all bits being 0), but for the AVR target, in general, all integer-type variables are set to 0, all pointers to a NULL pointer, and all floating-point variables to 0.0.

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

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

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

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

Back to FAQ Index.

7.3.9 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 SIGNAL() 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.

7.3.10 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" (PORTB):);
```

Note:

`avr/io.h` already provides a sbi() macro definition, which can be used in C programs.

Back to FAQ Index.

7.3.11 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.

7.3.12 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.

7.3.13 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 in the 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 the 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.

Back to FAQ Index.

7.3.14 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) 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.

7.3.15 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. This most obvious 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 want 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];
    strcpy_P (buf, array[1]);
    return 0;
}
```

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

```
0000008c <foo>:
  8c:   46 6f           ori r20, 0xF6   ; 246
  8e:   6f 00           .word   0x006f  ; ????

00000090 <bar>:
  90:   42 61           ori r20, 0x12   ; 18
  92:   72 00           .word   0x0072  ; ????

00000094 <array>:
  94:   8c 00           .word   0x008c  ; ????
  96:   90 00           .word   0x0090  ; ????
```

`foo` is at addr 0x008c.

`bar` is at addr 0x0090.

`array` is at addr 0x0094.

Then in main we see this:

```
    strcpy_P (buf, array[1]);   /* copy bar into buf
  de:   60 e9           ldi r22, 0x90   ; 144
  e0:   70 e0           ldi r23, 0x00   ; 0
  e2:   ce 01           movw    r24, r28
  e4:   01 96           adiw    r24, 0x01   ; 1
  e6:   0e 94 79 00     call    0xf2
```

The addr of bar (0x0090) is loaded into the r23:r22 pair which is the second parameter passed to strcpy_P. The r25:r24 pair is the addr of buf.

Back to FAQ Index.

7.3.16 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 so the initialization of these variable will take place. Refer to How to modify MCUCR or WDTCR early? for a description how to do this using few lines of assembler code, or to the chapter about memory sections for an example written in C.

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

Finally, if the application simply wants to use the additional RAM for private data storage kept outside the domain of the C compiler (e. g. through a `char *` variable initialized directly to a particular address), it would be sufficient to defer the initialization of the external RAM interface to the beginning of main(), so no tweaking of the `.init1` 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.

7.4 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 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@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/.`

7.4.1 GCC asm Statement

Let's start with a simple example of reading a value from port D:

```
asm("in %0, %1" : "=r" (value) : "I" (PORTD) : );
```

Each `asm` statement is devided by colons into 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" (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" (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" (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"::);
```

7.4.2 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.

7.4.3 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.

Note:

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
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" (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 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.

7.4.4 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.

7.4.5 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" ((uint8_t)(port)),  \
                   "I" ((uint8_t)(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.

7.4.6 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 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" (port)
        );
    return result;
}
```

Note:

inw() is supplied by avr-libc.

7.4.7 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`.

7.4.8 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/`

7.5 Using malloc()

7.5.1 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 `.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.

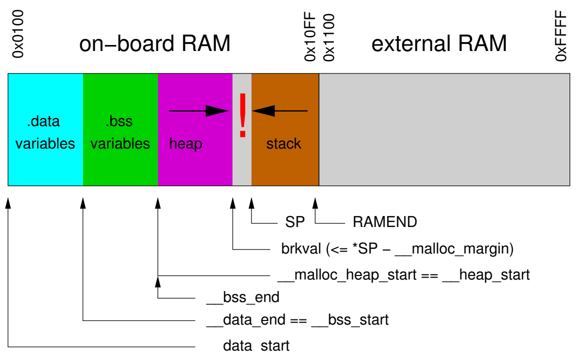

Figure 1: RAM map of a device with internal RAM

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.

7.5.2 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.

7.5.3 Tunables for malloc()

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

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

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

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

```
avr-gcc ... -Wl,-Tdata=0x801100,--defsym=__heap_end=0x80ffff ...
```

Note:

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

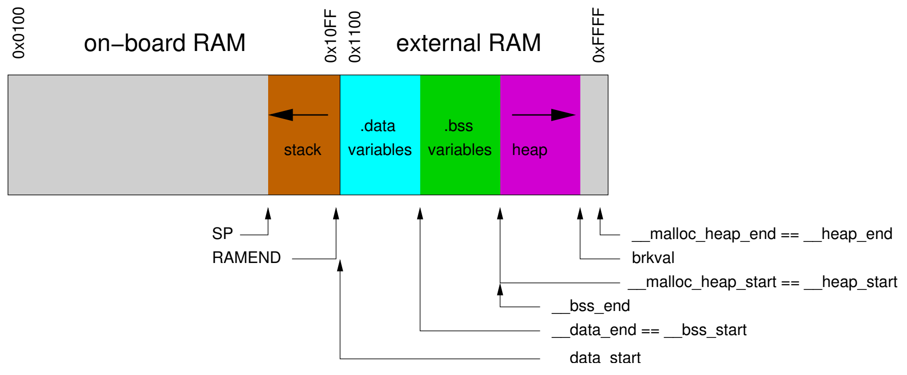

Figure 2: Internal RAM: stack only, external RAM: variables and heap

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

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

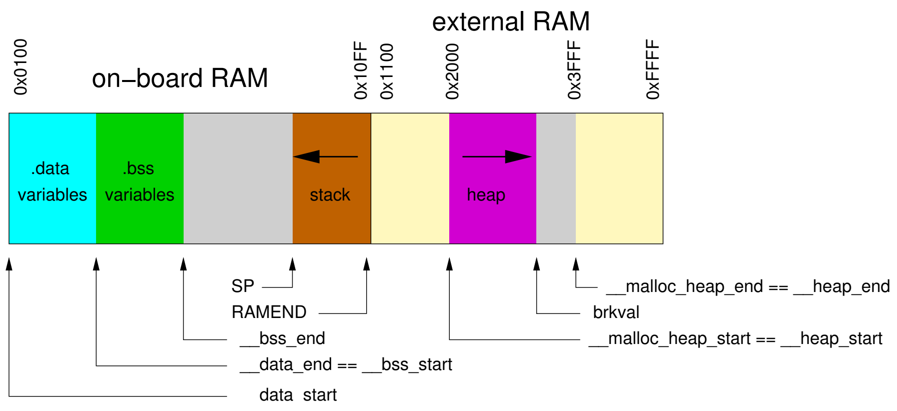

Figure 3: Internal RAM: variables and stack, external RAM: heap

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

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

7.5.4 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(). The returned address points just beyond that header. Thus, if the application accidentally writes before the returned memory region, the internal consistency of the memory allocator is compromised.

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

The freelist itself is not maintained as a separate data structure, but rather by modifying the contents of the freed memory to contain pointers chaining the pieces together. That way, no additional memory is 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(), 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.

7.6 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.

7.6.1 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.

7.6.2 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]

Note:

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

7.6.3 The .bss Section

Uninitialized global or static variables end up in the .bss section.

7.6.4 The .eeprom Section

This is where eeprom variables are stored.

7.6.5 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.]

7.6.6 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.

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 the tell the linker where to stick a chunk of code which is *not* a function. Notice that the examples for asm and C 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.

.init3:

Unused. User definable.

.init4:
Copies the .data section from flash to SRAM. Also sets up and zeros out the .bss section. In Unix-like targets, .data is normally initialized by the OS directly from the executable file. Since this is impossible in an MCU environment, `avr-gcc` instead takes care of appending the .data variables after .text in the flash ROM image. .init4 then defines the code (weakly bound) which takes care of copying the contents of .data from the flash to SRAM.

.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().

7.6.7 The .finiN Sections

These sections are used to define the exit code executed after return from main() or a call to exit(). These all are subparts of the .text section.

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).

7.6.8 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.

7.6.9 Using Sections in C Code

Example:

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

void my_init_portb (void) __attribute__ ((naked)) \
    __attribute__ ((section (".init1")));

void
my_init_portb (void)
{
        outb (PORTB, 0xff);
        outb (DDRB,  0xff);
}
```

7.7 Installing the GNU Tool Chain

Note:

This discussion was taken directly from Rich Neswold's document. (See Acknowledgments).

Note:

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
```

Note:

The versions for the packages listed below are known to work together. If you mix and match different versions, you may have problems.

7.7.1 Required Tools

- **GNU Binutils** (2.14)

  `http://sources.redhat.com/binutils/`

  Installation

- **GCC** (3.3)

  `http://gcc.gnu.org/`

  Installation

- **AVR Libc** (20020816-cvs)

  `http://savannah.gnu.org/projects/avr-libc/`

  Installation

Note:

As of 2002-08-15, the versions mentioned above are still considered experimental and must be obtained from cvs. Instructions for obtaining the latest cvs versions are available at the URLs noted above. Significant changes have been made which are not compatible with previous stable releases. These incompatilities should be noted in the documentation.

7.7.2 Optional Tools

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

- **uisp** (20020626)

  `http://savannah.gnu.org/projects/uisp/`

  Installation

- **avrprog** (2.1.0)

  `http://www.bsdhome.com/avrprog/`

  Installation

  Usage Notes

- **GDB** (5.2.1)

  `http://sources.redhat.com/gdb/`

  Installation

- **Simulavr** (0.1.0)
  `http://savannah.gnu.org/projects/simulavr/`
  Installation

- **AVaRice** (1.5)
  `http://avarice.sourceforge.net/`
  Installation

7.7.3 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.

Note:
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 `Makefile`s 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 environment variable before going to build avr-gcc.

7.7.4 GCC for the AVR target

Warning:

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

The steps to build `avr-gcc` are essentially same as for binutils:

```
$ 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
$ 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++.

Note:

The stdc++ libs are not included with C++ for AVR due to the size limitations of the devices.

7.7.5 AVR Libc

Warning:

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

Note:

If you have obtained the latest avr-libc from cvs, you will have to run the `reconf` script before using either of the build methods described below.

To build and install avr-libc:

```
$ gunzip -c avr-libc-<version>.tar.gz
$ cd avr-libc-<version>
$ ./doconf
$ ./domake
$ cd build
$ make install
```

Note:

The `doconf` script will automatically use the `$PREFIX` environment variable if you have set and exported it.

Alternatively, you could do this (shown for consistency with `binutils` and `gcc)`:

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

7.7.6 UISP

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

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

7.7.7 Avrprog

Note:

This is currently a FreeBSD only program, although adaptation to other systems should not be hard.

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

```
# cd /usr/ports/devel/avrprog
# 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.

7.7.8 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 or avarice since avr-gdb needs one of these to run as a a remote target.

7.7.9 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, avr-gcc and avr-libc if you want to have the test programs built in the simulavr source.

7.7.10 AVaRice

Note:

These install notes are specific to avarice-1.5.

You will have to edit `prog/avarice/Makefile` for avarice in order to install into a directory other than `/usr/local/avr/bin`. Edit the line which looks like this:

```
INSTALL_DIR = /usr/local/avr/bin
```

such that `INSTALL_DIR` is now set to whatever you decided on `$PREFIX/bin` to be.

```
$ gunzip -c avarice-1.5.tar.gz | tar xf -
$ cd avarice-1.5/prog/avarice
$ make
$ make install
```

7.8 Using the avrprog program

Note:

This section was contributed by Brian Dean [bsd@bsdhome.com].

`avrprog` 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 `avrprog`'s `-e` option).

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

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

Once installed, `avrprog` 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:

```
# avrprog -p 2313 -e -m flash -i main.hex

avrprog: AVR device initialized and ready to accept instructions
```

```
avrprog: Device signature = 0x1e9101

avrprog: erasing chip
avrprog: done.
avrprog: reading input file "main.hex"
avrprog: input file main.hex auto detected as Intel Hex

avrprog: writing flash:
1749 0x00
avrprog: 1750 bytes of flash written
avrprog: verifying flash memory against main.hex:
avrprog: reading on-chip flash data:
1749  0x00
avrprog: verifying ...
avrprog: 1750 bytes of flash verified

avrprog done.  Thank you.
```

The `-p 2313` option lets `avrprog` 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 `avrprog`'s configuration file (`/usr/local/etc/avrprog.conf`). To list valid parts, specify the `-v` option. The `-e` option instructs `avrprog` 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:

```
# avrprog -p 2313 -t
avrprog: AVR device initialized and ready to accept instructions
avrprog: Device signature = 0x1e9101
avrprog>

The '?' command displays a list of valid
commands:

avrprog> ?
>>> ?
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.

avrprog>
```

7.9 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.

7.9.1 Options for the C compiler avr-gcc

7.9.1.1 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

  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__

Architecture	MCU name	Macro
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	attiny26	__AVR_ATtiny26__
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__
avr3	atmega103	__AVR_ATmega103__
avr3	atmega603	__AVR_ATmega603__
avr3	at43usb320	__AVR_AT43USB320__
avr3	at43usb355	__AVR_AT43USB355__
avr3	at76c711	__AVR_AT76C711__
avr4	atmega8	__AVR_ATmega8__
avr4	atmega8515	__AVR_ATmega8515__
avr4	atmega8535	__AVR_ATmega8535__
avr5	atmega16	__AVR_ATmega16__
avr5	atmega161	__AVR_ATmega161__
avr5	atmega162	__AVR_ATmega162__
avr5	atmega163	__AVR_ATmega163__
avr5	atmega169	__AVR_ATmega169__
avr5	atmega32	__AVR_ATmega32__
avr5	atmega323	__AVR_ATmega323__
avr5	atmega64	__AVR_ATmega64__
avr5	atmega128	__AVR_ATmega128__
avr5	at94k	__AVR_AT94K__

- `-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`.

7.9.1.2 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 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).

7.9.2 Options for the assembler avr-as

7.9.2.1 Machine-specific assembler options

- `-mmcu=`*architecture*
- `-mmcu=`*MCU name*

  avr-as understands the same `-mmcu=` options as avr-gcc. 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.

7.9.2.2 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), 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).

7.9.3 Controlling the linker avr-ld

7.9.3.1 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 will be linked together.

7.9.3.2 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 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. 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 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()` will still be placed after all the variables in the data section, so in this situation, no stack/heap collision can occur.

7.10 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.

7.10.1 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. If you have a development kit, you should be able to use it, rather than build the circuit, for this project.

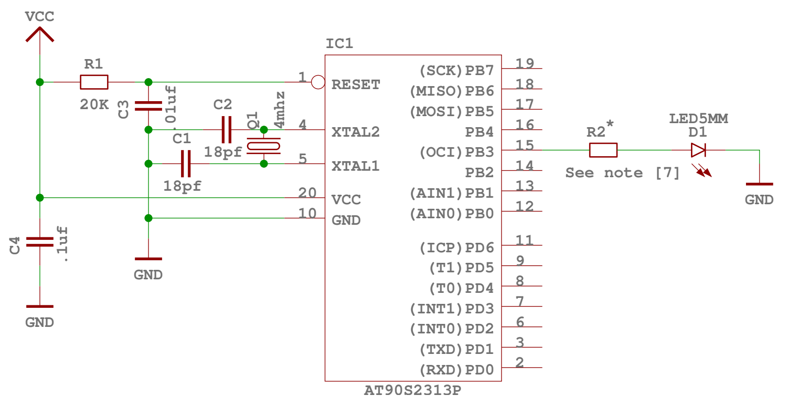

Figure 4: Schematic of circuit for demo project

The source code is given in demo.c. For the sake of this example, create a file called `demo.c` containing this source code. Some of the more important parts of the code are:

Note [1]:
The `PWM` is being used in 10-bit mode, so we need a 16-bit variable to remember the current value.

Note [2]:
SIGNAL() is a macro that marks the function as an interrupt routine. In this case, the function will get called when the timer overflows. Setting up interrupts is explained in greater detail in Interrupts and Signals.

Note [3]:
This section determines the new value of the `PWM`.

Note [4]:
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.

Note [5]:
This routine gets called after a reset. It initializes the `PWM` and enables interrupts.

Note [6]:

The main loop of the program does nothing – all the work is done by the interrupt routine! If this was a real product, we'd probably put a `SLEEP` instruction in this loop to conserve power.

Note [7]:

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.

7.10.2 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 1.1 2002/09/30 18:16:07 troth Exp $
 */

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

#if defined(__AVR_AT90S2313__)
#  define OC1 PB3
#  define OCR OCR1
#  define DDROC DDRB
#elif defined(__AVR_AT90S2333__) || defined(__AVR_AT90S4433__)
#  define OC1 PB1
#  define DDROC DDRB
#  define OCR OCR1
#elif defined(__AVR_AT90S4414__) || defined(__AVR_AT90S8515__) || \
      defined(__AVR_AT90S4434__) || defined(__AVR_AT90S8535__) || \
      defined(__AVR_ATmega163__)
#  define OC1 PD5
#  define DDROC DDRD
#  define OCR OCR1A
#else
#  error "Don't know what kind of MCU you are compiling for"
```

```
#endif

#if defined(COM11)
#  define XCOM11 COM11
#elif defined(COM1A1)
#  define XCOM11 COM1A1
#else
#  error "need either COM1A1 or COM11"
#endif

enum { UP, DOWN };

volatile uint16_t pwm; /* Note [1] */
volatile uint8_t direction;

SIGNAL (SIG_OVERFLOW1) /* Note [2] */
{
    switch (direction) /* Note [3] */
    {
        case UP:
            if (++pwm == 1023)
                direction = DOWN;
            break;

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

    OCR = pwm; /* Note [4] */
}

void
ioinit (void) /* Note [5] */
{
    /* tmr1 is 10-bit PWM */
    TCCR1A = _BV (PWM10) | _BV (PWM11) | _BV (XCOM11);

    /* tmr1 running on full MCU clock */
    TCCR1B = _BV (CS10);

    /* set PWM value to 0 */
    OCR = 0;

    /* enable OC1 and PB2 as output */
    DDROC = _BV (OC1);

    timer_enable_int (_BV (TOIE1));

    /* enable interrupts */
    sei ();
}

int
```

```
main (void)
{
    ioinit ();

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

    for (;;) /* Note [6] */
        ;

    return (0);
}
```

7.10.3 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=at90s2333 -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=at90s2333 -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.

7.10.4 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         000000ca  00000000  00000000  00000094  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .data         00000000  00800060  000000ca  0000015e  2**0
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000003  00800060  00800060  0000015e  2**0
                  ALLOC
  3 .noinit       00000000  00800063  00800063  0000015e  2**0
                  CONTENTS
  4 .eeprom       00000000  00810000  00810000  0000015e  2**0
                  CONTENTS
  5 .stab         0000066c  00000000  00000000  00000160  2**2
                  CONTENTS, READONLY, DEBUGGING
  6 .stabstr      00000630  00000000  00000000  000007cc  2**0
                  CONTENTS, READONLY, DEBUGGING
Disassembly of section .text:

00000000 <__vectors>:
   0:   0a c0           rjmp    .+20            ; 0x16
   2:   62 c0           rjmp    .+196           ; 0xc8
   4:   61 c0           rjmp    .+194           ; 0xc8
   6:   60 c0           rjmp    .+192           ; 0xc8
   8:   5f c0           rjmp    .+190           ; 0xc8
   a:   0a c0           rjmp    .+20            ; 0x20
   c:   5d c0           rjmp    .+186           ; 0xc8
   e:   5c c0           rjmp    .+184           ; 0xc8
  10:   5b c0           rjmp    .+182           ; 0xc8
  12:   5a c0           rjmp    .+180           ; 0xc8
  14:   59 c0           rjmp    .+178           ; 0xc8

00000016 <__ctors_end>:
  16:   11 24           eor     r1, r1
  18:   1f be           out     0x3f, r1        ; 63
  1a:   cf ed           ldi     r28, 0xDF       ; 223
  1c:   cd bf           out     0x3d, r28       ; 61
  1e:   4e c0           rjmp    .+156           ; 0xbc
```

```
00000020 <__vector_5>:
volatile uint16_t pwm; /* Note [1] */
volatile uint8_t direction;

SIGNAL (SIG_OVERFLOW1) /* Note [2] */
{
  20:   1f 92           push    r1
  22:   0f 92           push    r0
  24:   0f b6           in      r0, 0x3f        ; 63
  26:   0f 92           push    r0
  28:   11 24           eor     r1, r1
  2a:   2f 93           push    r18
  2c:   8f 93           push    r24
  2e:   9f 93           push    r25
    switch (direction) /* Note [3] */
  30:   80 91 60 00     lds     r24, 0x0060
  34:   99 27           eor     r25, r25
  36:   00 97           sbiw    r24, 0x00       ; 0
  38:   a1 f0           breq    .+40            ; 0x62
  3a:   01 97           sbiw    r24, 0x01       ; 1
  3c:   29 f5           brne    .+74            ; 0x88
    {
        case UP:
            if (++pwm == 1023)
                direction = DOWN;
            break;

        case DOWN:
            if (--pwm == 0)
  3e:   80 91 61 00     lds     r24, 0x0061
  42:   90 91 62 00     lds     r25, 0x0062
  46:   01 97           sbiw    r24, 0x01       ; 1
  48:   90 93 62 00     sts     0x0062, r25
  4c:   80 93 61 00     sts     0x0061, r24
  50:   80 91 61 00     lds     r24, 0x0061
  54:   90 91 62 00     lds     r25, 0x0062
  58:   89 2b           or      r24, r25
  5a:   b1 f4           brne    .+44            ; 0x88
                direction = UP;
  5c:   10 92 60 00     sts     0x0060, r1
  60:   13 c0           rjmp    .+38            ; 0x88
  62:   80 91 61 00     lds     r24, 0x0061
  66:   90 91 62 00     lds     r25, 0x0062
  6a:   01 96           adiw    r24, 0x01       ; 1
  6c:   90 93 62 00     sts     0x0062, r25
  70:   80 93 61 00     sts     0x0061, r24
  74:   80 91 61 00     lds     r24, 0x0061
  78:   90 91 62 00     lds     r25, 0x0062
  7c:   8f 5f           subi    r24, 0xFF       ; 255
  7e:   93 40           sbci    r25, 0x03       ; 3
  80:   19 f4           brne    .+6             ; 0x88
  82:   81 e0           ldi     r24, 0x01       ; 1
  84:   80 93 60 00     sts     0x0060, r24
            break;
    }
    OCR = pwm; /* Note [4] */
  88:   80 91 61 00     lds     r24, 0x0061
  8c:   90 91 62 00     lds     r25, 0x0062
  90:   9b bd           out     0x2b, r25       ; 43
  92:   8a bd           out     0x2a, r24       ; 42
}
  94:   9f 91           pop     r25
  96:   8f 91           pop     r24
  98:   2f 91           pop     r18
  9a:   0f 90           pop     r0
  9c:   0f be           out     0x3f, r0        ; 63
  9e:   0f 90           pop     r0
  a0:   1f 90           pop     r1
  a2:   18 95           reti

000000a4 <ioinit>:

void
ioinit (void) /* Note [5] */
{
    /* tmr1 is 10-bit PWM */
    TCCR1A = _BV (PWM10) | _BV (PWM11) | _BV (XCOM11);
  a4:   83 e8           ldi     r24, 0x83       ; 131
  a6:   8f bd           out     0x2f, r24       ; 47

    /* tmr1 running on full MCU clock */
    TCCR1B = _BV (CS10);
  a8:   81 e0           ldi     r24, 0x01       ; 1
  aa:   8e bd           out     0x2e, r24       ; 46

    /* set PWM value to 0 */
    OCR = 0;
  ac:   1b bc           out     0x2b, r1        ; 43
  ae:   1a bc           out     0x2a, r1        ; 42

    /* enable OC1 and PB2 as output */
    DDROC = _BV (OC1);
  b0:   88 e0           ldi     r24, 0x08       ; 8
  b2:   87 bb           out     0x17, r24       ; 23

extern inline void timer_enable_int (unsigned char ints)
{
#ifdef TIMSK
  outb(TIMSK, ints);
  b4:   80 e8           ldi     r24, 0x80       ; 128
  b6:   89 bf           out     0x39, r24       ; 57

    timer_enable_int (_BV (TOIE1));

    /* enable interrupts */
    sei ();
  b8:   78 94           sei
}
  ba:   08 95           ret
```

```
000000bc <main>:

int
main (void)
{
  bc:   cf ed           ldi     r28, 0xDF       ; 223
  be:   d0 e0           ldi     r29, 0x00       ; 0
  c0:   de bf           out     0x3e, r29       ; 62
  c2:   cd bf           out     0x3d, r28       ; 61
    ioinit ();
  c4:   ef df           rcall   .-34            ; 0xa4

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

    for (;;) /* Note [6] */
  c6:   ff cf           rjmp    .-2             ; 0xc6

000000c8 <__bad_interrupt>:
  c8:   9b cf           rjmp    .-202           ; 0x0
```

7.10.5 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=at90s2313 -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       0xca
     *(.vectors)
     .vectors       0x00000000       0x16 ../../../obj-i386-redhat-linux-gnu/crt1/crts2313.o
                    0x00000000                __vectors
                    0x00000000                __vector_default
                    0x00000016                __ctors_start = .
```

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

```
     *(.fini2)
     *(.fini1)
```

```
 *(.fini0)
                0x000000ca                _etext = .
.data           0x00800060        0x0 load address 0x000000ca
                0x00800060                PROVIDE (__data_start, .)
 *(.data)
 *(.gnu.linkonce.d*)
                0x00800060                . = ALIGN (0x2)
                0x00800060                _edata = .
                0x00800060                PROVIDE (__data_end, .)
.bss            0x00800060        0x3
                0x00800060                PROVIDE (__bss_start, .)
 *(.bss)
 *(COMMON)
 COMMON         0x00800060        0x3 demo.o
                                  0x0 (size before relaxing)
                0x00800060                direction
                0x00800061                pwm
                0x00800063                PROVIDE (__bss_end, .)
                0x000000ca                __data_load_start = LOADADDR (.data)
                0x000000ca                __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 load address 0x000000ca
 *(.eeprom*)
                0x00810000                __eeprom_end = .
```

The last address in the `.text` segment is location `0xf2` (denoted by `_etext`), so the instructions use up 242 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 a 2313 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.

7.10.6 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:

```
:100000000AC062C061C060C05FC00AC05DC05CC0A1
:100010005BC05AC059C011241FBECFEDCDBF4EC02A
:100020001F920F920FB60F9211242F938F939F93CD
:10003000809160009927009A1F0019729F58091A0
:1000400061009091620001979093620080936100 3B
:100050008091610090916200892BB1F41092600050
:1000600013C08091610090916200019690936200AC
:1000700080936100809161009091620 08F5F934056
:1000800019F481E080936000809161009091620 09A
:100090009BBD8ABD9F918F912F910F900FBE0F90A6
:1000A0001F90189583E88FBD81E08EBD1BBC1ABCE4
:1000B00088E087BB80E889BF78940895CFEDD0E0D1
:0A00C000DEBFCDBFEFDFFFCF9BCF07
: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 -O ihex demo.elf demo_eeprom.hex
```

The resulting `demo_eeprom.hex` file contains:

```
:00000001FF
```

which is an empty `.hex` file (which is expected, since we didn't define any EEPROM variables).

7.10.7 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
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)

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 $< $@

%_eeprom.srec: %.elf
        $(OBJCOPY) -j .eeprom --change-section-lma .eeprom=0 -O srec $< $@

%_eeprom.bin: %.elf
        $(OBJCOPY) -j .eeprom --change-section-lma .eeprom=0 -O binary $< $@

# 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 $< $@
```

7.11 Deprecated List

Global eeprom_rb(addr) Use eeprom_read_byte() in new programs.

Global eeprom_rw(addr) Use eeprom_read_word() in new programs.

Global eeprom_wb(addr, val) Use eeprom_write_byte() in new programs.

Global outp(val, sfr) For backwards compatibility only. This macro will eventually be removed.

Global inp(sfr) For backwards compatibility only. This macro will eventually be removed.

Global BV(bit) For backwards compatibility only. This macro will eventually be removed.

Index