avr-libc Reference Manual

20020910-cvs

Generated by Doxygen 1.2.17

Tue Sep 10 09:24:22 2002

Contents

1 AVR Libc

The latest version of this document is always available from `http://www.freesoftware.fsf.org/avr-libc/.`

The AVR Libc package provides a subset of the standard C library for Atmel AVR 8-bit RISC microcontrollers..........

There's a lot of work to be done on this. This file will produce the index.html (for html output) or the first chapter (in LaTeX output).

1.1 Supported Devices

AT90S Type Devices:

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

ATmega Type Devices:

- atmega8
- atmega103
- atmega128
- atmega161
- atmega162
- atmega163
- atmega169
- atmega323

ATtiny Type Devices:

- attiny10 [1]
- attiny11 [1]
- attiny12 [1]

- attiny15 [1]
- attiny22
- attiny28 [1]

Misc Devices:

- at94K
- at76c711

[FIXME: troth/2002-09-02: How do the at94 and at76 devices fit into the grand scheme of all things AVR?]

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.

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)
- unsigned char eeprom_rb (unsigned int addr)
- unsigned int eeprom_rw (unsigned int addr)
- void eeprom_wb (unsigned int addr, unsigned char val)
- void eeprom_read_block (void ∗buf, unsigned int addr, size_t n)

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)

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

5.1.3 Function Documentation

5.1.3.1 unsigned char eeprom_rb (unsigned int *addr*)

read one byte from EEPROM address `addr`

5.1.3.2 void eeprom_read_block (void ∗ *buf*, unsigned int *addr*, size_t *n*)

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

5.1.3.3 unsigned int eeprom_rw (unsigned int *addr*)

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

5.1.3.4 void eeprom_wb (unsigned int *addr*, unsigned char *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/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).

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 Integer Types

5.5.1 Detailed Description

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

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

Note:
These should probably not be used if avr-gcc's `-mint8` option is used.

Typedefs

- typedef signed char **int8_t**
- typedef unsigned char **uint8_t**
- typedef int **int16_t**
- typedef unsigned int **uint16_t**
- typedef long **int32_t**
- typedef unsigned long **uint32_t**
- typedef long long **int64_t**
- typedef unsigned long long **uint64_t**
- typedef int16_t **intptr_t**
- typedef uint16_t **uintptr_t**

5.6 Setjmp and Longjmp

5.6.1 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__

5.6.2 Function Documentation

5.6.2.1 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.

5.6.2.2 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.

5.7 General utilities

5.7.1 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

Conversion functions for double arguments.

- #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)

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

- 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)

Defines

- #define **RAND_MAX** 0x7FFFFFFF

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)
- double strtod (const char *s, char **endptr)

Variables

- size_t __malloc_margin
- char * __malloc_heap_start
- char * __malloc_heap_end

5.7.2 Define Documentation

5.7.2.1 #define DTOSTR_ALWAYS_SIGN 0x01

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

5.7.2.2 #define DTOSTR_PLUS_SIGN 0x02

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

5.7.2.3 #define DTOSTR_UPPERCASE 0x04

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

5.7.3 Typedef Documentation

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

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

5.7.4 Function Documentation

5.7.4.1 __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.

5.7.4.2 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.

5.7.4.3 __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);
```

5.7.4.4 __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);
```

5.7.4.5 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.

5.7.4.6 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`.

5.7.4.7 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 into in the style [-]d.dddedd 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.

5.7.4.8 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 into in the style [-]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.

5.7.4.9 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.

5.7.4.10 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.

5.7.4.11 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`.

5.7.4.12 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.

5.7.4.13 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`.

5.7.4.14 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`.

5.7.4.15 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.

5.7.4.16 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 sort 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.

5.7.4.17 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 con sists 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?

5.7.4.18 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.

5.7.4.19 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, 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 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.

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

5.7.4.21 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`.

5.7.5 Variable Documentation

5.7.5.1 char∗ __malloc_heap_end

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, and `__heap_end` is set to 0 which makes malloc() assume the heap is below the stack. Any changes need to be made before the very first call to malloc().

In case of a device with external SRAM where the heap is going to be allocated in external RAM, it's good practice to already define those symbols from the linker command line.

5.7.5.2 char∗ __malloc_heap_start

See `__malloc_heap_end`.

5.7.5.3 size_t __malloc_margin

When extending the data segment in malloc(), the allocator 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'll risk to collide with the data segment.

The default is set to 32. `__malloc_margin` should be changed before the very first call to malloc() within the application.

All this is only relevant in situations where the heap is allocated below the stack. For devices with external memory, the heap can be located in external memory while the stack is usually located in internal SRAM, so no special guard area is needed between both.

5.8 Strings

5.8.1 Detailed Description

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

The string functions perform string operations on NULL terminated strings.

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 *)

5.8.2 Function Documentation

5.8.2.1 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.

5.8.2.2 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.

5.8.2.3 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.

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.

5.8.2.4 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.

5.8.2.5 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.

5.8.2.6 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.

5.8.2.7 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.

5.8.2.8 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.

5.8.2.9 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.

5.8.2.10 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.

5.8.2.11 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.

5.8.2.12 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.

5.8.2.13 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.

5.8.2.14 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.

5.8.2.15 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.

5.8.2.16 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.

5.8.2.17 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.

5.8.2.18 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.

5.8.2.19 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.

5.8.2.20 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.

5.8.2.21 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.

5.8.2.22 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.

5.8.2.23 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.

5.8.2.24 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.

5.9 Interrupts and Signals

5.9.1 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
}
```

[FIXME: should there be a discussion of writing an interrupt handler in asm?]

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. IC [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	

Signal Name	Description
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
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

Signal Name	Description
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

- #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

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

5.9.2 Define Documentation

5.9.2.1 #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.

5.9.2.2 #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.

5.9.2.3 #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.

5.9.2.4 #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.

5.9.3 Function Documentation

5.9.3.1 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.

5.9.3.2 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.

5.10 Special function registers

5.10.1 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)

5.10.2 Define Documentation

5.10.2.1 #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().

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

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

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

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

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

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

5.10.2.4 #define BV(bit) _BV(bit)

Deprecated:

For backwards compatibility only. This macro will evetually be removed.
Use _BV() in new programs.

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

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

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

5.10.2.6 #define inb(sfr) _SFR_BYTE(sfr)

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

Read a byte from IO register `sfr`.

5.10.2.7 #define inp(sfr) inb(sfr)

Deprecated:

For backwards compatibility only. This macro will evetually be removed.
Use inb() in new programs.

5.10.2.8 #define inw(sfr) _SFR_WORD(sfr)

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

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

5.10.2.9 #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.

5.10.2.10 #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.

5.10.2.11 #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).

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

Deprecated:

For backwards compatibility only. This macro will evetually be removed.
Use outb() in new programs.

5.10.2.13 #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).

5.10.2.14 #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().

Data Fields

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

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().

Data Fields

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

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 in [FIXME: hasn't been merged yet, put ref here].
- Brian Dean [`bsd@bsdhome.com`] for developing **avrprog** (an alternate to **uisp**) and for contributing [FIXME: need to merge section on **avrprog**] which describes how to use it.
- All the people you 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 Frequently Asked Questions

7.2.1 FAQ Index

7.2.2 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.

7.2.3 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.

7.2.4 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.

7.2.5 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.

7.2.6 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.

7.2.7 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.

7.2.8 Shouldn't I better 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 .init-N Sections. With respect to the standard, this sentence is somewhat simplified (because the standard would allow for machines where the actual bit pattern used differs from all bits 0), but for the AVR target, in effect 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 RAM.)

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 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 how 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, initializers should only be written if they are non-zero.

Back to FAQ Index.

7.2.9 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.

7.2.10 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.

7.2.11 When single-stepping through my program in avr-gdb, the PC "jumps around"

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 in reordering 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.

Other side effects of optimzation are 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 function 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.

Back to FAQ Index.

7.2.12 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`.

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 greatly confuses the debugger.

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 has 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 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.

Copyright (C) 2001-2002 by egnite Software GmbH

Permission is granted to copy and distribute verbatim copies of this manual provided that the copyright notice and this permission notice are preserved on all copies. Permission is granted to copy and distribute modified versions of this manual provided that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

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://www.freesoftware.fsf.org/avr-libc/ .

7.3.1 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"::);
```

7.3.2 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.

7.3.3 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.

7.3.4 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.

7.3.5 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.

7.3.6 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.

7.3.7 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`.

7.3.8 Links

A GNU Development Environment for the AVR Microcontroller covers the details of the GNU Tools that are specific to the AVR family of processors. By Rich Neswold. `http://www.enteract.com/~rneswold/avr/`

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/`

7.4 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.

7.4.1 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.

7.4.2 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]

7.4.3 The .bss Section

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

7.4.4 The .eeprom Section

This is where eeprom variables are stored.

7.4.5 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.]

7.4.6 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().

7.4.7 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).

7.4.8 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.

7.4.9 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);
}
```

7.5 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.

7.5.1 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.

7.5.2 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
- **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

7.5.3 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.

7.5.4 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.

7.5.5 AVR Libc

Warning:

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

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
```

Note:

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

7.5.6 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
```

7.5.7 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.

7.5.8 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.

7.5.9 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.

7.5.10 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
```

Index