A GNU Development Environment for the AVR Microcontroller

Rich Neswold

rneswold@earthlink.net

A GNU Development Environment for the AVR Microcontroller

by Rich Neswold

Published $Date$

Copyright © 1999, 2000, 2001, 2002 by Richard M. Neswold, Jr.

This document attempts to cover the details of the GNU Tools that are specific to the AVR family of processors.

Acknowledgements

This document tries to tie together the labors of a large group of people. Without these individuals' efforts, we wouldn't have a terrific, *free* set of tools to develop AVR projects. We all owe thanks to:

- The GCC Team, which produced a very capable set of development tools for an amazing number of platforms and processors.
- Denis Chertykov <`denisc@overta.ru`> for making the AVR-specific changes to the GNU tools.
- Denis Chertykov and Marek Michalkiewicz <`marekm@linux.org.pl`> for developing the standard libraries and startup code for **AVR-GCC**.
- Uros Platise for developing the AVR programmer tool, **uisp**.
- Joerg Wunsch <`joerg@FreeBSD.ORG`> for adding all the AVR development tools to the FreeBSD (http://www.freebsd.org) ports tree and for providing the demo project in Chapter 2.
- Brian Dean <`bsd@bsdhome.com`> for developing **avrprog** (an alternate to **uisp**) and for contributing Section 1.4.1 which describes how to use it.

Table of Contents

1. Installing the GNU Tools 1

1.1. GNU Binutils 1

1.2. AVR-GCC 2

1.2.1. Downloading the Source 2

1.2.2. Building the Project 2

1.2.3. Installing the Tools 2

1.3. AVR-LIBC 3

1.3.1. Downloading the Source 3

1.3.2. Building the Libraries 3

1.3.3. Installing the Libraries and Header Files 3

1.4. Programming Your Processor 3

1.4.1. Using **avrprog** 3

1.4.2. Using **uisp** 5

2. Using the GNU Tools 6

2.1. The Project 6

2.2. Compiling and Linking 8

2.3. “Map” Files 13

2.4. Generating `.hex` Files 15

2.5. Letting **Make** Build the Project 15

3. Application Start-up 17

4. Standard C Library 19

4.1. Function Reference 19

4.1.1. `abort` 19

4.1.2. `abs` 19

4.1.3. `atoi` 19

4.1.4. `atol` 20

4.1.5. `bsearch` 20

4.1.6. `cos` 20

4.1.7. `cosh` 21

4.1.8. `div` 21

4.1.9. `exit` 21

4.1.10. `fabs` 21

4.1.11. `isalnum`, `isalpha`, `isascii`, `isblank`, `iscntrl`, `isdigit`, `isgraph`, `islower`, `isprint`, `ispunct`, `isspace`, `isupper`, `isxdigit` 22

4.1.12. `labs` 22

4.1.13. `ldiv` 23

4.1.14. `longjmp` 23

4.1.15. `qsort` 23

4.1.16. `setjmp` 24

4.1.17. `sin` 24

4.1.18. `sinh` 24

4.1.19. `sqrt` 25

4.1.20. `tan` 25

4.1.21. `toascii` 25

4.1.22. `tolower`, `toupper` 25

5. Memory APIs 27

5.1. Program Memory 27

5.2. Function Reference 27

5.2.1. `__ATTR_CONST__`, `__ATTR_PROGMEM__`, `__ATTR_PURE__` 28

5.2.2. `__elpm_inline` 28

5.2.3. `__lpm_inline` 28

5.2.4. `memcpy_P` 29

5.2.5. `PRG_RDB` 29

5.2.6. `PSTR` 29

5.2.7. `strcat_P` 29

5.2.8. `strcmp_P` 30

5.2.9. `strcpy_P` 30

5.2.10. `strcasecmp_P` 30

5.2.11. `strlen_P` 31

5.2.12. `strncasecmp_P` 31

5.2.13. `strncmp_P` 31

5.2.14. `strncpy_P` 31

5.3. EEPROM 32

5.4. Function Reference 32

5.4.1. `eeprom_is_ready` 32

5.4.2. `eeprom_rb` 33

5.4.3. `eeprom_read_block` 33

5.4.4. `eeprom_rw` 33

5.4.5. `eeprom_wb` 33

6. Interrupt API 35

6.1. Function Reference 36

6.1.1. `cli` 36

6.1.2. `enable_external_int` 36

6.1.3. `INTERRUPT` 36

6.1.4. `sei` 37

6.1.5. `SIGNAL` 37

6.1.6. `timer_enable_int` 38

7. I/O API 39

7.1. I/O Port APIs 39

7.2. Function Reference 39

7.2.1. `BV` 39

7.2.2. `bit_is_clear` 39

7.2.3. `bit_is_set` 39

7.2.4. `cbi` 40

7.2.5. `inp` 40

7.2.6. `__inw` 40

7.2.7. `__inw_atomic` 41

7.2.8. `loop_until_bit_is_clear` 41

7.2.9. `loop_until_bit_is_set` 41

7.2.10. `outp` 42

7.2.11. `__outw` 42

7.2.12. `__outw_atomic` 42
7.2.13. `parity_even_bit` 42
7.2.14. `sbi` 43
7.3. Watchdog API 43
7.4. Function Reference 43
7.4.1. `wdt_disable` 43
7.4.2. `wdt_enable` 44
7.4.3. `wdt_reset` 44
A. AVR-GCC Configuration 45
A.1. `avr-as` Options 45
A.2. `avr-gcc` Options 45

List of Tables

1-1. Tarball Locations 1
5-1. Primitive types in program memory 27
6-1. Signal names 35
A-1. `avr-as` Options 45
A-2. `avr-gcc` Options 45

List of Figures

2-1. Schematic of demo project 6
3-1. Hex file for empty `main()` 18

List of Examples

2-1. Demo source code 6
2-2. Disassembly of Demo Application 9
2-3. Portion of demo map file 14
2-4. `Makefile` for Demo Project 16
3-1. Code that runs immediately after RESET 17
3-2. Configuring the watchdog during reset 18
5-1. Proper use of EEPROM variables 32
6-1. Setting up an interrupt handler 37
6-2. Setting up a signal handler 37

Chapter 1. Installing the GNU Tools

This chapter shows how to build and install a complete development environment for the AVR processors using the GNU toolset.

I created an area for the AVR tools under `/usr/local` to keep this stuff separate from the base system. As `root`, I `chown`'ed `/usr/local/avr` under my normal account. This way, I don't have to be root to install the tools. All the instructions assume the tools will be installed in this location. If you want to place them in a different locations you need to specify the new location using the `--prefix` option.

Table 1-1. Tarball Locations

Tool	Version	Location
GNU Binutils	2.11	`binutils-2.11.tar.bz2` (http://mirrors.rcn.net/pub/sourceware/binutils/releases/binutils-2.11.tar.bz2)
AVR-GCC	3.0.1	`gcc-core-3.0.1.tar.gz` (ftp://gatekeeper.dec.com/pub/GNU/gcc/gcc-3.0.1/gcc-core-3.0.1.tar.gz)
AVR libc	20020203	`avr-libc-20020203.tar.gz` (http://www.amelek.gda.pl/avr/libc/avr-libc-20020203.tar.gz)
AVR Programmer	1.0b	`uisp-1.0b.src.tar.gz` ()

1.1. GNU Binutils

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.

The binutils source archive, used in preparing this document, is version 2.11. Its location is given in Table 1-1. Download the file and extract[1] its contents.

```
% bunzip2 -c binutils-2.11.tar.bz2 | tar xf -
% cd binutils-2.11
```

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 --target=avr \
    --prefix=/usr/local/avr
```

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.[2]

```
% make
```

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
```

Once this completes, you will have a set of utilities for the AVR processor. The executables are located in the `bin` directory located in the base directory you specified in the `--prefix` option. You'll have to add that directory to your search path in order to run them conveniently.

1.2. AVR-GCC

1.2.1. Downloading the Source

> **Warning:**This section is being rewritten. Ignore it for now.

The gcc source archive, used in preparing this document, is version 2.95.2. You also need to apply AVR-specific patches.[3] The three files can be downloaded using the URLs in Table 1-1. Create a directory in which to build the tools and put the downloaded files in it. You are now ready to build the utilities.

1.2.2. Building the Project

The first step is to pull the source from the archive and apply the patches to the code.

```
% tar zxf gcc-core-2.95.2.tar.gz
% cd gcc-2.95.2
% gunzip -dc ../gcc-core-2.95.2-avr-patch-1.1.gz | patch -p1
```

The next step is to configure and build the compiler. This is done by supplying arguments to the **configure** script that enable the AVR-specific options and then making the project.

```
% configure --target=avr \
    --prefix=/usr/local/avr \
    --disable-nls \
    --enable-languages=c
% make
```

I specify the same installation directory as the binutils. Also, since there is little C++ support (in the case of standard libraries), I only build the C compiler.

1.2.3. Installing the Tools

If the compiler was built cleanly, you're ready to install it. To install:

```
% make install
```

1.3. AVR-LIBC

1.3.1. Downloading the Source

The AVR standard library archive used in this document is version 20020203. The archive can be obtained using the URL in Table 1-1. Download this file and place it in your working directory.

1.3.2. Building the Libraries

Before we can build the libraries, we need to unarchive the tarball.

```
% tar zxf avr-libc-20020203.tar.gz
% cd avr-libc-20020203
```

Now simply build the project.

> **Note:** At this point, the user can configure some library options, like setting whether the watchdog can be initialized through the linker hack.

```
% cd src
% make prefix=/usr/local/avr
```

1.3.3. Installing the Libraries and Header Files

Once the libraries have been built, you need to install them with the rest of the tools.

```
% make prefix=/usr/local/avr install
```

1.4. Programming Your Processor

There are several applications that will burn your `.hex` files into an AVR processor. This section covers these tools.

1.4.1. Using avrprog

(This section was contributed by Brian Dean <`bsd@bsdhome.com`>.)

avrprog is a program that is used to update or read the flash and EEPROM memories of Atmel AVR microcontrollers on FreeBSD Unix. It supports the Atmel serial programming protocol using the PC's parallel port and can upload either a raw binary file or an Intel Hex format file. It can also be used in an interactive mode to individually update EEPROM cells, fuse bits, and/or lock bits (if their access is supported by the Atmel serial programming protocol.) The main flash instruction memory of the AVR can also be programmed in interactive mode, however this is not very useful because one can only turn bits off. The only way to turn flash bits on is to erase the entire memory (using AVRPROG's `-e` option).

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

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

Once installed, **avrprog** can program processors using the contents of the `.hex` file specified on the command line. In this example, the file `main.hex` is burned into the flash memory:

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

avrprog: AVR device initialized and ready to accept instructions

avrprog: Device signature = 0x1e9101

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

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

avrprog done.  Thank you.
```

The "`-p 2313`" option lets **avrprog** know that we are operating on an AT90S2313 chip. This option specifies the device id and is matched up with the device of the same id in AVRPROG's configuration file (`/usr/local/etc/avrprog.conf`). To list valid parts, specify the "`-v`" option. The "`-e`" option instructs **avrprog** to perform a chip-erase before programming; this is almost always necessary before programming the flash. The "`-m flash`" option indicates that we want to upload data into the flash memory, while "`-i main.hex`" specifies the name of the input file.

The EEPROM is uploaded in the same way, the only difference is that you would use "`-m eeprom`" instead of "`-m flash`".

To use interactive mode, use the "`-t`" option:

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

```
avrprog>
```

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

```
avrprog> ?
>>> ?
Valid commands:

  dump   : dump memory  : dump <memtype> <addr> <N-Bytes>
  read   : alias for dump
  write  : write memory : write <memtype> <addr> <b1> <b2> ... <bN>
  erase  : perform a chip erase
  sig    : display device signature bytes
  part   : display the current part information
  send   : send a raw command : send <b1> <b2> <b3> <b4>
  help   : help
  ?      : help
  quit   : quit

Use the ’part’ command to display valid memory types for use with the
’dump’ and ’write’ commands.

avrprog>
```

1.4.2. Using uisp

(This section hasn’t been written yet. Sorry!)

Notes

1. This file has been archived with **bzip2**, so `bunzip2` needs to be already installed on your system.
2. BSD users should note that the project’s `Makefile` uses GNU `make` syntax. This means FreeBSD users need to build the tools by using `gmake`.
3. Again, the AVR patches have been committed to the GNU project, so future releases will have AVR support built-in.

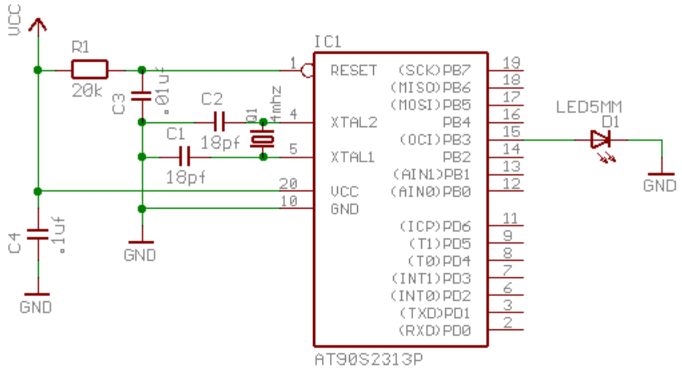
Chapter 2. Using the GNU Tools

At this point, you should have the GNU tools configured, built, and installed on your system. In this chapter, we present a simple example of using the GNU tools in an AVR project. After reading this chapter, you should have a better feel as to how the tools are used and how a `Makefile` can be configured.

2.1. The Project

This project will use the pulse-width modulator (PWM) to ramp an LED on and off every two seconds. An AT90S2313 processor will be used as the controller. The circuit for this demonstration is shown in Figure 2-1. If you have a development kit, you should be able to use it, rather than build the circuit, for this project.

Figure 2-1. Schematic of demo project



VCC
R1
20k
C3
.01uf
C2
18pf
C1
18pf
Q1
4mhz
IC1
1
4
5
20
10
RESET
XTAL2
XTAL1
VCC
GND
(SCK)PB7
(MISO)PB6
(MOSI)PB5
PB4
(OCI)PB3
PB2
(AIN1)PB1
(AIN0)PB0
(ICP)PD6
(T1)PD5
(T0)PD4
(INT1)PD3
(INT0)PD2
(TXD)PD1
(RXD)PD0
19
18
17
16
15
14
13
12
11
9
8
7
6
3
2
AT90S2313P
LED5MM
D1
GND
C4
.1uf
GND
GND

The source code is given in Example 2-1. For the sake of this example, create a file called `demo.c` containing this source code. Some of the more important parts are:

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

❷ `SIGNAL`() is a macro that marks the function as an interrupt routine. In this case, the function will get called when the timer overflows. Setting up interrupts is explained in greater detail in Chapter 6.

❸ This section determines the new value of the PWM.

❹ Here's where the newly computed value is loaded into the PWM regsiter. Since we are in an interrupt routine, it is safe to use `outw`(). Outside of an interrupt, `outw_atomic`() should be used.

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

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

Example 2-1. Demo source code

```
/*
 * ----------------------------------------------------------------------------
 * "THE BEER-WARE LICENSE" (Revision 42):
```

```
 * <joerg@FreeBSD.ORG> wrote this file.  As long as you retain this notice you
 * can do whatever you want with this stuff. If we meet some day, and you think
 * this stuff is worth it, you can buy me a beer in return.        Joerg Wunsch
 * ----------------------------------------------------------------------------
 *
 * Simple AVR demonstration.  Controls a LED that can be directly
 * connected from OC1/OC1A to GND.  The brightness of the LED is
 * controlled with the PWM.  After each period of the PWM, the PWM
 * value is either incremented or decremented, that’s all.
 *
 * $Id: simple-demo.c,v 1.3 2002/02/11 19:51:51 j Exp $
 */

#include <io.h>
#include <interrupt.h>
#include <sig-avr.h>

#if defined(__AVR_AT90S2313__)
# define OC1 PB3
# define OCR OCR1L
# define DDROC DDRB
#elif defined(__AVR_AT90S2333__) || defined(__AVR_AT90S4433__)
# define OC1 PB1
# define DDROC DDRB
# define OCR OCR1L
#elif defined(__AVR_AT90S4414__) || defined(__AVR_AT90S8515__) || \
 defined(__AVR_AT90S4434__) || defined(__AVR_AT90S8535__) || \
 defined(__AVR_ATmega163__)
# define OC1 PD5
# define DDROC DDRD
# define OCR OCR1AL
#else
# error "Don’t know what kind of MCU you are compiling for"
#endif

volatile uint16_t pwm;❶
volatile enum
{
  UP, DOWN
}
direction;


SIGNAL(SIG_OVERFLOW1)❷
{

  switch (direction)❸
    {
    case UP:
      if (++pwm == 1023)
 direction = DOWN;
      break;
    case DOWN:
      if (--pwm == 0)
 direction = UP;
      break;
    }

  __outw(pwm, OCR);❹
}

void
ioinit(void)❺
{

#if defined(COM11)
  outp(BV(PWM10)|BV(PWM11)|BV(COM11), TCCR1A); /* tmr1 is 10-bit PWM */
#elif defined(COM1A1)
  outp(BV(PWM10)|BV(PWM11)|BV(COM1A1), TCCR1A); /* tmr1 is 10-bit PWM */
#else
# error "need either COM1A1 or COM11"
#endif
  outp(BV(CS10), TCCR1B); /* tmr1 running on full MCU clock */

  __outw(0, OCR); /* set PWM value to 0 */

  outp(BV(OC1), DDROC); /* enable OC1 and PB2 as output */

  timer_enable_int(BV(TOIE1));
  sei(); /* enable interrupts */

}

void
main(void)
{

  ioinit();

  for (;;)❻
    /* wait forever, the interrupts are doing the rest */;
}
```

2.2. Compiling and Linking

This first thing that needs to be done is compile the source. When compiling, the compiler needs to know the processor type so the `-mmcu` option is specified. The `-Os` option will tell the compiler to optimize the code for efficient space usage (at the possible expense of code execution speed.) The `-g` is used to embed debug info. The debug info is useful for disassemblies and doesn’t end up in the `.hex` files, so I usually specify it. Finally, the `-c` tells the compiler to compile and stop -- don’t link. This demo is small enough that we could compile and link in one step.

However, real-world projects will have several modules and will typically need to break up the building of the project into several compiles and one link.

```
% avr-gcc -g -Os -mmcu=at90s2313 -c demo.c
```

The compilation will create a `demo.o` file. Next we link it into a binary called `demo.out`.

```
% avr-gcc -g -mmcu=at90s2313 -o demo.out demo.o
```

It is important to specify the MCU type when linking. The compiler uses the `-mmcu` option to choose start-up files and run-time libraries that get linked together. If this option isn't specified, the compiler defaults to the 8515 processor environment, which is most certainly what you didn't want.

Now we have a binary file. Can we do anything useful with it (besides put it into the processor?) The GNU Binutils suite is made up of many useful tools for manipulating object files that get generated. One tool is `avr-objdump`, which takes information from the object file and displays it in many useful ways. Typing the command by itself will cause it to list out its options.

For instance, to get a feel of the application's size, the `-h` option can be used:

```
% avr-objdump -h demo.out

demo.out:     file format elf32-avr

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         000000ec  00000000  00000000  00000094  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .data         00000000  00800060  000000ec  00000180  2**0
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000004  00800060  00800060  00000180  2**0
                  ALLOC
  3 .eeprom       00000000  00810000  00810000  00000180  2**0
                  CONTENTS
  4 .stab         00000690  00000000  00000000  00000180  2**2
                  CONTENTS, READONLY, DEBUGGING
  5 .stabstr      00000637  00000000  00000000  00000810  2**0
                  CONTENTS, READONLY, DEBUGGING
```

The output of this command shows how much space is used in each of the sections (the .stab and .stabstr sections hold the debugging information and won't make it into the ROM file.)

An even more useful option is `-S`. This option disassembles the binary file and intersperses the source code in the output! This method is much better, in my opinion, than using the `-S` with the compiler because this listing includes routines from the libraries and the vector table contents. Also, all the "fix-ups" have been satisfied. In other words, the listing generated by this option reflects the actual code that the processor will run.

```
% avr-objdump -S demo.out
```

This command generates the output shown in Example 2-2.

Example 2-2. Disassembly of Demo Application

```
demo.out:     file format elf32-avr

Disassembly of section .text:

00000000 <.__start_of_init__>:
   0: 0a c0        rjmp .+20      ; 0x16
   2: 21 c0        rjmp .+66      ; 0x46
   4: 20 c0        rjmp .+64      ; 0x46
   6: 1f c0        rjmp .+62      ; 0x46
   8: 1e c0        rjmp .+60      ; 0x46
   a: 1f c0        rjmp .+62      ; 0x4a
   c: 1c c0        rjmp .+56      ; 0x46
   e: 1b c0        rjmp .+54      ; 0x46
  10: 1a c0        rjmp .+52      ; 0x46
  12: 19 c0        rjmp .+50      ; 0x46
  14: 18 c0        rjmp .+48      ; 0x46

00000016 <_real_init_>:
  16: 11 24        eor r1, r1
  18: 1f be        out 0x3f, r1 ; 63
  1a: 20 e0        ldi r18, 0x00 ; 0
  1c: a8 95        wdr
  1e: 21 bd        out 0x21, r18 ; 33
  20: 20 e0        ldi r18, 0x00 ; 0
  22: 25 bf        out 0x35, r18 ; 53
  24: ec ee        ldi r30, 0xEC ; 236
  26: f0 e0        ldi r31, 0x00 ; 0
  28: a0 e6        ldi r26, 0x60 ; 96
  2a: b0 e0        ldi r27, 0x00 ; 0
  2c: 03 c0        rjmp .+6       ; 0x34

0000002e <.copy_data_loop>:
  2e: c8 95        lpm
  30: 31 96        adiw r30, 0x01 ; 1
  32: 0d 92        st X+, r0

00000034 <.copy_data_start>:
  34: a0 36        cpi r26, 0x60 ; 96
  36: d9 f7        brne .-10      ; 0x2e
  38: a0 e6        ldi r26, 0x60 ; 96
  3a: b0 e0        ldi r27, 0x00 ; 0
  3c: 01 c0        rjmp .+2       ; 0x40

0000003e <.zero_bss_loop>:
  3e: 1d 92        st X+, r1

00000040 <.zero_bss_start>:
  40: a4 36        cpi r26, 0x64 ; 100
  42: e9 f7        brne .-6       ; 0x3e
  44: 4c c0        rjmp .+152     ; 0xde
```

```
00000046 <_comparator_>:
  46: 00 c0           rjmp .+0       ; 0x48

00000048 <_unexpected_>:
  48: 18 95           reti

0000004a <_overflow1_>:
static uint16_t pwm;
enum {UP, DOWN} direction;

SIGNAL(SIG_OVERFLOW1)
{
  4a: 1f 92           push r1
  4c: 0f 92           push r0
  4e: 0f b6           in r0, 0x3f ; 63
  50: 0f 92           push r0
  52: 11 24           eor r1, r1
  54: 2f 93           push r18
  56: 8f 93           push r24
  58: 9f 93           push r25
    switch (direction) {
  5a: 80 91 62 00  lds r24, 0x0062
  5e: 90 91 63 00  lds r25, 0x0063
  62: 00 97           sbiw r24, 0x00 ; 0
  64: 71 f0           breq .+28      ; 0x82
  66: 01 97           sbiw r24, 0x01 ; 1
  68: f1 f4           brne .+60      ; 0xa6
     case UP:
 if (++pwm == 1023)
     direction = DOWN;
 break;

     case DOWN:
 if (--pwm == 0)
  6a: 80 91 60 00  lds r24, 0x0060
  6e: 90 91 61 00  lds r25, 0x0061
  72: 01 97           sbiw r24, 0x01 ; 1
  74: 90 93 61 00  sts 0x0061, r25
  78: 80 93 60 00  sts 0x0060, r24
  7c: 00 97           sbiw r24, 0x00 ; 0
  7e: 99 f4           brne .+38      ; 0xa6
     direction = UP;
  80: 0e c0           rjmp .+28      ; 0x9e
  82: 80 91 60 00  lds r24, 0x0060
  86: 90 91 61 00  lds r25, 0x0061
  8a: 01 96           adiw r24, 0x01 ; 1
  8c: 90 93 61 00  sts 0x0061, r25
  90: 80 93 60 00  sts 0x0060, r24
  94: 8f 5f           subi r24, 0xFF ; 255
  96: 93 40           sbci r25, 0x03 ; 3
  98: 31 f4           brne .+12      ; 0xa6
  9a: 81 e0           ldi r24, 0x01 ; 1
  9c: 90 e0           ldi r25, 0x00 ; 0
  9e: 90 93 63 00  sts 0x0063, r25
  a2: 80 93 62 00  sts 0x0062, r24
 break;
    }
    __outw(pwm, OCR1L);
  a6: 80 91 60 00  lds r24, 0x0060
  aa: 90 91 61 00  lds r25, 0x0061
  ae: 9b bd        out 0x2b, r25 ; 43
  b0: 8a bd        out 0x2a, r24 ; 42
  b2: 9f 91        pop r25
  b4: 8f 91        pop r24
  b6: 2f 91        pop r18
  b8: 0f 90        pop r0
  ba: 0f be        out 0x3f, r0 ; 63
  bc: 0f 90        pop r0
  be: 1f 90        pop r1
  c0: 18 95        reti

000000c2 <ioinit>:
}

void
ioinit(void)
{
#if defined(COM11)
    outp(BV(PWM10)|BV(PWM11)|BV(COM11), TCCR1A); /* tmr1 is 10-bit PWM */
#elif defined(COM1A1)
    outp(BV(PWM10)|BV(PWM11)|BV(COM1A1), TCCR1A); /* tmr1 is 10-bit PWM */
  c2: 83 e8        ldi r24, 0x83 ; 131
  c4: 8f bd        out 0x2f, r24 ; 47
#else
# error "need either COM1A1 or COM11"
#endif
    outp(BV(CS10), TCCR1B);       /* tmr1 running on full MCU clock */
  c6: 81 e0        ldi r24, 0x01 ; 1
  c8: 8e bd        out 0x2e, r24 ; 46

    __outw(0, OCR1L);             /* set PWM value to 0 */
  ca: 80 e0        ldi r24, 0x00 ; 0
  cc: 90 e0        ldi r25, 0x00 ; 0
  ce: 9b bd        out 0x2b, r25 ; 43
  d0: 8a bd        out 0x2a, r24 ; 42

    outp(BV(OC1), DDRB);          /* enable OC1 and PB2 as output */
  d2: 88 e0        ldi r24, 0x08 ; 8
  d4: 87 bb        out 0x17, r24 ; 23
#endif
}

extern inline void timer_enable_int (unsigned char ints)
{
  d6: 80 e8        ldi r24, 0x80 ; 128
#ifdef TIMSK
```

```
  outp (ints, TIMSK);
  d8: 89 bf        out 0x39, r24 ; 57

    timer_enable_int(BV(TOIE1));
    sei();                        /* enable interrupts */
  da: 78 94        sei
}
  dc: 08 95        ret

000000de <main>:

int
main(void)
{
  de: cf ed        ldi r28, 0xDF ; 223
  e0: d0 e0        ldi r29, 0x00 ; 0
  e2: de bf        out 0x3e, r29 ; 62
  e4: cd bf        out 0x3d, r28 ; 61
    ioinit();
  e6: ed df        rcall .-38      ; 0xc2

    for (;;)
  e8: ff cf        rjmp .-2       ; 0xe8

000000ea <__stop_progIi__>:
  ea: ff cf        rjmp .-2       ; 0xea
```

2.3. “Map” Files

`avr-objdump` is very useful, but sometimes it’s necessary to see information about the link that can only be generated by the linker. A map file contains this information. A map file is useful for monitoring the sizes of your code and data. It also shows where modules are loaded and which modules were loaded from libraries. It is yet another view of your application. To get a map file, I usually add `-Wl,-Map,demo.map` to my link command. Relink the application using the following command to generate `demo.map` (a portion of which is shown in Example 2-3.

```
% avr-gcc -g -mmcu=at90s2313 -Wl,-Map,demo.map -o demo.out demo.o
```

Some points of interest in the map file are:

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

❷ The next available address in the .text segment is location 0xec, so the instructions use up 234 bytes of FLASH.

❸ The .data segment (where initialized static variables are stored) starts at location 0x60, which is the first address after the register bank on a 2313 processor.

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

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

❻ The next available address in the .bss segment is location 0x64, so the application uses 4 bytes of uninitialized data.

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

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

Example 2-3. Portion of demo map file

```
.text           0x00000000       0xec❶
 *(.init)
 .init          0x00000000       0x16 /usr/local/avr/lib/crts2313.o
 *(.progmem.gcc*)
 *(.progmem*)
                0x00000016                .=ALIGN(0x2)
 *(.text)
 .text          0x00000016       0x34 /usr/local/avr/lib/crts2313.o
                0x00000046                _interrupt1_
                0x00000046                _uart_trans_
                0x00000046                _overflow0_
                0x00000016                _init_
                0x00000046                _uart_data_
                0x00000046                _uart_recv_
                0x00000048                _unexpected_
                0x00000046                _comparator_
                0x00000046                _interrupt0_
                0x00000046                _output_compare1a_
                0x00000046                _input_capture1_
                0x00000016                _real_init_
 .text          0x0000004a       0xa2 demo.o
                0x0000004a                _overflow1_
                0x000000c2                ioinit
                0x000000de                main
                0x000000ec                .=ALIGN(0x2)
 *(.text.*)
                0x000000ec                .=ALIGN(0x2)
 *(.fini)
                0x000000ec                _etext=.❷

.data           0x00800060        0x0 load address 0x000000ec❸
                0x00800060                PROVIDE (__data_start, .)
 *(.data)
 *(.gnu.linkonce.d*)
                0x00800060                .=ALIGN(0x2)
                0x00800060                _edata=.❹

.bss            0x00800060        0x4❺
                0x00800060                PROVIDE (__bss_start, .)
 *(.bss)
 .bss           0x00800060        0x2 demo.o
 *(COMMON)
 COMMON         0x00800062        0x2 demo.o
                                  0x0 (size before relaxing)
                0x00800062                direction
                0x00800064                PROVIDE (__bss_end, .)
                0x00800064                _end=.❻

.eeprom         0x00810000        0x0 load address 0x000000ec❼
 *(.eeprom*)
                0x00810000                __eeprom_end=.❽
```

2.4. Generating `.hex` Files

We have a binary of the application, but how do we get it into the processor? Most (if not all) programmers will not accept a GNU executable as an input file, so we need to do a little more processing. The next step is to extract portions of the binary and save the information into “hex” files. The GNU utility that does this is called `avr-objcopy`.

The ROM contents can be pulled from our project’s binary and put into the file `rom.hex` using the following command:

```
% avr-objcopy -j .text -O ihex demo.out rom.hex
```

The resulting `.hex` file contains:

```
:100000000AC021C020C01FC01EC01FC01CC01BC012
:100010001AC019C018C011241FBE20E0A89521BD28
:1000200020E025BFE4EFF0E0A0E6B0E003C0C89513
:1000300031960D92A437D9F7A4E7B0E001C01D9224
:10004000A837E9F750C000C018951F920F920FB65D
:100050000F9211242F938F939F938091760090910C
:1000600077000097 91F0019711F580917400 9091BD
```

The `-j` option indicates that we want the information from the .text segment extracted. If we specify the EEPROM segment, we can generate a `.hex` file that can be used to program the EEPROM:

```
% avr-objcopy -j .eeprom -O ihex demo.out eeprom.hex
```

The resulting `.hex` file contains:

```
:00000001FF
```

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

2.5. Letting Make Build the Project

Rather than type these commands over and over, they can all be placed in a **make** file. To build the demo project using **make**, save Example 2-4 in a file called `Makefile`.

Example 2-4. `Makefile` for Demo Project

```
CC=avr-gcc
OBJCOPY=avr-objcopy

CFLAGS=-g -mmcu=at90s2313

rom.hex : demo.out
 $(OBJCOPY) -j .text -O ihex demo.out rom.hex

demo.out : demo.o
 $(CC) $(CFLAGS) -o demo.out -Wl,-Map,demo.map demo.o

demo.o : demo.c
 $(CC) $(CFLAGS) -Os -c demo.c
```

Chapter 3. Application Start-up

The standard library includes a start-up module that prepares the environment for running applications written in C. Several versions of the start-up script are available because each processor has different set-up requirements. The compiler, `avr-gcc`, selects the appropriate module based upon the processor specified by command line options (see Appendix).

For the AVR processors, the start-up module is responsible for the following tasks:

- Providing a default vector table.
- Providing default interrupt handlers.
- Initializing the globally-reserved registers.
- Initializing the watchdog.
- Initializing the mcucr register.
- Initializing the data segment.
- Zeroing out the .bss segment.
- Jumping to `main()`. (A jump is used, rather than a call, to save space on the stack -- `main()` is not expected to return.)

The start-up module contains a default interrupt vector table. The contents of the table are filled with predefined function names which can be overridden by the programmer. This is discussed completely in Chapter 6. The first entry in the table, however, is the reset vector, which is set to jump to location `_init_`. `_init_` is defined to be a "weak" symbol, which means that if the application doesn't define it, the linker will use the value from the library (or module). The start-up module defines `_init_` to be the same location as `_real_init_`.

If you want to add some custom code that gets executed immediately after a reset, name your routine `_init_`. To avoid wasting program memory, you should define the function using the `naked` attribute. This tells the compiler not to generate any prologue or epilogue code in the function. It also prevents a ret instruction from being added, which allows us to end the function with a rjmp instruction. An example of how to do this is shown in Example 3-1.

Example 3-1. Code that runs immediately after RESET.

```
void _real_init_(void);
void _init_(void) __attribute__((naked));

void _init_(void)
{
    /* This must be the last line of the function. */

    asm ( "rjmp _real_init_" );
}
```

Once execution begins at `_real_init_`, the system sets up the watchdog and the mcucr registers. The module uses a linker trick to allow you to modify the value without recompiling. The module takes the *address* of the variables __init_wdctr__ and __init_mcucr__, rather than the contents. By using the `--defsym` option to the linker, you set the address of the symbols, which are used as the load values for the registers. These two variables are defined as "weak" symbols, so the module will provide default values if you don't override them.

Example 3-2 shows the use of `avr-gcc` to link together some object files into an executable, `prog.out`. The executable will set the watchdog control register to 0x03.

Example 3-2. Configuring the watchdog during reset

```
% avr-gcc -o prog.out -Wl,--defsym,__init_wdctr__=3 file1.o file2.o file3.o
```

Next, global variables that have initial values are loaded from program memory. The compiler creates two identically laid out sections. One will be placed in static RAM and is used during program execution. The other is placed in program ROM and contains the initial values. The start-up code copies the ROM image into the static RAM so that `main()` (and everything called from `main()`) see a properly initialized data segment.

The uninitialized data section, .bss, is then zeroed out. This section contains all non-auto variables that weren't given an initial value.

Lastly, the module jumps to `main()` and the application starts running. The function `main()` is recognized by the compiler as being special, so some prolog and epilog code is placed in this function. When entering, the stack is initialized to point to the end of static RAM. The end of the function always contains an infinite loop, so if you try to exit `main()`, your application will hang.

It should be noted that the start-up modules add quite a bit of bulk to an application. If you are using a smaller part, the bloat caused by the start-up module may be unacceptible. In those cases, your application would be better served by writing it entirely in assembly language. As an example, Figure 3-1 contains the hex file, generated by an empty `main()`, targetted for the AT90S2313 processor. The processor has only 1Kwords of ROM space and the start-up code eats up nearly 5% of it!

Figure 3-1. Hex file for empty `main()`

```
:150000000FC027C026C025C024C023C022C021C020C01FC01E03
:15001500C0CFEDD0E0CDBFDEBFFFCF11241FBE20E0A89521BD86
:15002A0020E025BFE4E5F0E0A0E6B0E003C0C89531960D92A008
:15003F0036D9F7A0E6B0E001C01D92A036E9F7E3CF1895FECF3E
:00000001FF
```

Chapter 4. Standard C Library

A subset of the C standard library is supported. This chapter covers the functions that have been supported. Since AVR processors have several memory spaces, developers must be careful when passing parameters to functions that expect pointers. The C library understands only one type of pointer, so passing addresses to data in the EEPROM or FLASH will fail. Routines that understand these other memory spaces are addressed in Section 5.1 and Section 5.3.

It should also be noted that if your application uses the floating point libraries, you'll need to link with them. Add `-lm` to your link command.

4.1. Function Reference

4.1.1. `abort`

```
#include <stdlib.h>

void abort(void);
```

description. Stops your application. It is currently implemented as an infinite loop.

4.1.2. `abs`

```
#include <stdlib.h>

int abs(int x);
```

description. Computes and returns the absolute value of *x*

see also. `fabs()`, `labs()`

4.1.3. `atoi`

```
#include <stdlib.h>

int atoi(char const* str);
```

description. Returns an integer represented by *`str`*. This function ignores leading whitespace. It stops converting when it sees the first nonsensical characters (i.e. "123test" will return 123). If no leading portion of the string represents an integer, 0 is returned.)

4.1.4. `atol`

```
#include <stdlib.h>

long atol(char const* str);
```

description. Returns a long integer represented by *`str`*. This function ignores leading whitespace. It stops converting when it sees the first nonsensical characters (i.e. "123test" will return 123). If no leading portion of the string represents a long integer, 0 is returned.)

4.1.5. `bsearch`

```
#include <stdlib.h>

void* bsearch(void const* key, void const* data, size_t total, size_t size,
__compar_fn_t f);
```

description. Performs a binary search to find an element of an array that matches a key. The base address of the array is given by the *`data`* argument. The size of each element if the array is specifies by the *`size`*. The total number of elements in the array is specified by *`total`*. The data with which to search is indicated with *`key`*.

The function makes decisions by calling the user-supplied function *`f`*. It is assumed that the elements of the array are sorted in a fashion meaningful to *`key`*. The prototype for this function is:

```
int (*__compar_fn_t)(void const*, void const*)
```

The first argument passed to the compare function will be *`key`*. The second argument will be a pointer to an element in the array. If *`key`* matches the element, the function should return 0. If *`key`* is greater than the element, the function should return 1. Otherwise it should return -1.

4.1.6. `cos`

```
#include <math.h>

double cos(double x);
```

description. Computes the cosine of its argument. The argument *`x`* should be in radians.

4.1.7. `cosh`

```
#include <math.h>

double cosh(double x);
```

description. Computes the hyperbolic cosine of its argument.

4.1.8. `div`

```
#include <stdlib.h>

div_t div(int num, int denom);
```

description. Computes *`num`* / *`denom`*. The quotient and remainder are stored and returned in a *`div_t`* structure:

```
typedef struct {
    int quot;
    int rem;
} div_t;
```

see also. `ldiv()`

4.1.9. `exit`

```
#include <stdlib.h>

int exit(void);
```

description. Stops your application. It is currently implemented as an infinite loop.

4.1.10. `fabs`

```
#include <math.h>

double fabs(double x);
```

description. Returns the absolute value of `x`.

see also. `abs()`, `labs()`

4.1.11. `isalnum`, `isalpha`, `isascii`, `isblank`, `iscntrl`, `isdigit`, `isgraph`, `islower`, `isprint`, `ispunct`, `isspace`, `isupper`, `isxdigit`

```
#include <ctype.h>

int isalnum(int ch);
int isalpha(int ch);
int isascii(int ch);
int isblank(int ch);
int iscntrl(int ch);
int isdigit(int ch);
int isgraph(int ch);
int islower(int ch);
int isprint(int ch);
int ispunct(int ch);
int isspace(int ch);
int isupper(int ch);
int isxdigit(int ch);
```

description. These function perform character classification. They return true or false status depending whether the character passed to the function falls into the function's classification (i.e. `isdigit()` returns true if its argument is any value '0' though '9', inclusive.)

It should be noted that if any of these functions is used, they all get pulled in to your hex file (they're in the same source file). The comments in the source indicate that 182 bytes will get consumed.

4.1.12. `labs`

```
#include <stdlib.h>

long labs(long x);
```

description. Computes and returns the absolute value of *`x`*

see also. `abs()`, `fabs()`

4.1.13. `ldiv`

```
#include <stdlib.h>

ldiv_t ldiv(long num, long denom);
```

description. Computes *`num`*/*`denom`*. The quotient and remainder are stored and returned in a *`ldiv_t`* structure:

```
    typedef struct {
        long quot;
        long rem;
    } ldiv_t;
```

see also. `div()`

4.1.14. `longjmp`

```
#include <setjmp.h>

int longjmp(jmp_buf buf, int ret);
```

description. This function restores the application environment to what it was when `setjmp()` was used to initialize *`buf`*. This function does not return. Instead, execution continues after the corresponding `setjmp()`. The only difference is that the function `setjmp()` returns the value *`ret`* rather than 0.

see also. `setjmp()`

4.1.15. `qsort`

```
#include <stdlib.h>

void qsort(void* data, size_t total, size_t size, __compar_fn_t f);
```

description. Sorts the elements on an array using the "quick sort" algorithm. The base address of the array is given by the *`data`* argument. The size of each element if the array is specifies by the *`size`*. The total number of elements in the array is specified by *`total`*.

The sorting decision is made by passing two elements of the array to the user-supplied function *`f`*. The prototype for this function is:

```
int (*__compar_fn_t)(void const*, void const*)
```

If the first element matches the second, the function should return 0. If the first is greater than the second, the function should return 1. Otherwise it should return -1.

4.1.16. `setjmp`

```
#include <setjmp.h>

int setjmp(jmp_buf buf);
```

description. This function saves its environment into the chunk of memory defined by *`buf`*. This function returns 0. The buffer is used by a later call to `longjmp()`.

On the AVR processors, *`jmp_buf`* structures need 24 bytes of storage, so make sure you have enough stack space or RAM available.

see also. `longjmp()`

4.1.17. `sin`

```
#include <math.h>

double sin(double x);
```

description. Computes the sine of its argument. The argument *`x`* should be in radians.

4.1.18. `sinh`

```
#include <math.h>

double sinh(double x);
```

description. Computes the hyperbolic sine of its argument.

4.1.19. `sqrt`

```
#include <math.h>

double sqrt(double x);
```

description. Computes the square root of its argument. The argument `x` should be positive. If it is negative, errno will be set to EDOM.

4.1.20. `tan`

```
#include <math.h>

double tan(double x);
```

description. Computes the tangent of its argument. The argument `x` should be in radians.

4.1.21. `toascii`

```
#include <ctype.h>

int toascii(int ch);
```

description. Converts the argument to a value that fits in the range of ASCII values (0 - 0x7f). It does this masking off the upper bits. This function is defined in the same file as the `isalpha()`, etc. set of functions, so if you use this function in your code, you will pull in all the others. If you don't need those other functions, you can save a lot of program space by simply ANDing your value with 0x7f.

4.1.22. `tolower`, `toupper`

```
#include <ctype.h>

int tolower(int ch);
int toupper(int ch);
```

description. Converts the argument to its upper-case or lower-case representation, depending upon which function you call. This function does not recognise international characters. These functions are defined in the same file as the `isalpha()`, etc. set of functions, so if you use either function in your code, you will pull in all the others.

Chapter 5. Memory APIs

The AVR family of processors do not use a single address space to map data and code. Since the registers are 8 bits wide, and the registers are used to write to RAM, the static RAM was made 8 bits wide. The program memory, on the other hand, is 16 bits wide. This allows the instructions to represent more operations in a single memory access. In addition, the EEPROM resides in yet another bank of memory.

AVR-GCC places code in the flash ROM and places data in the SRAM, which would be expected. If your program needs to access the EEPROM or place data in the ROM, however, things are a little less intuitive. This chapter shows what support has been provide for these situations.

5.1. Program Memory

Placing data in ROM is very useful to embedded applications: the data is always available and doesn't have to be generated at startup. Even more importantly, the data cannot get corrupted by an errant application, which reduces the number of considerations when debugging.

Since the ROM resides in a different address space, we need a way to tell the compiler to place variables there. We also need a way to access the data (i.e. the compiler has to use the lpm instruction.)

The first detail is provided by the `__attribute__` keyword. By tagging a variable with `__attribute__((progmem))`, you can force it to reside in the ROM. *Variables with this attribute cannot be accessed like variables not using the attribute*. You need to use the macros described in this section to access the data in ROM. There are a number of data types already defined for the primitive types. These are shown in Table 5-1.

Table 5-1. Primitive types in program memory

Type Name	Definition
prog_void	void __attribute__((progmem))
prog_char	char __attribute__((progmem))
prog_uchar	unsigned char __attribute__((progmem))
prog_int	int __attribute__((progmem))
prog_long	long __attribute__((progmem))
prog_long_long	long long __attribute__((progmem))
PGM_P	prog_char const*
PGM_VOID_P	prog_void const*

The second step, accessing the data, is done using the macros in this section. These macros are found in `pgmspace.h`.

5.2. Function Reference

5.2.1. `__ATTR_CONST__`, `__ATTR_PROGMEM__`, `__ATTR_PURE__`

```
#include <pgmspace.h>

__ATTR_CONST__, __ATTR_PROGMEM__, __ATTR_PURE__
```

description. These macros are used to notify the compiler that it is to handle a function or variable specially.

If a function is marked with the `__ATTR_CONST__` macro, the compiler will assume the function produces no side effects and produces an identical result when rpesented with identical inputs. (i.e. the function takes the parameters and produces a result, but doesn't change any memory locations.) If a function marked with this attribute is in a loop and its parameters don't change, the compiler can call it once and use the return value in the loop.

The `__ATTR_PROGMEM__` macro is used in variable definitions. If a variable has this attribute, it is allocated in program memory. Since program memory can't be changed when the processor is running, a variable with this attribute is always defined with an initial value.

The `__ATTR_PURE__` macro, when used on a function, tells the compiler not to generate any prologue or epilogue code (the function's ret instruction is even suppressed!)

5.2.2. `__elpm_inline`

```
#include <pgmspace.h>

uint8_t __elpm_inline(uint32_t addr);
```

description. This macro gets converted into in-line assembly instructions to pull a byte from program ROM. The elpm instruction is used, so this macro can only be used with AVR devices that support it. The argument is the 32-bit address of the cell. The maximum address depends upon the device being used.

5.2.3. `__lpm_inline`

```
#include <pgmspace.h>

uint8_t __lpm_inline(uint16_t addr);
```

description. This function gets converted into in-line assembly instructions to pull a byte from program ROM. The argument is the 16-bit address of the cell. The maximum address depends upon the device being used.

Only one byte is returned by this function. When pulling wider values from the program memory, the `memcpy_P()` and `strcpy_P()` functions should be used.

see also. `memcpy_P()`, `strcpy_P()`

5.2.4. `memcpy_P`

```
#include <pgmspace.h>

void* memcpy_P(void* dst, PGM_VOID_P src, size_t n);
```

description. This is a special version of the `memcpy` function that copies data from program memory to RAM.

5.2.5. `PRG_RDB`

```
#include <pgmspace.h>

uint8_t PRG_RDB(uint16_t addr);
```

description. This macro simply invokes the `__lpm_inline()` function.

5.2.6. `PSTR`

```
#include <pgmspace.h>

PSTR(s);
```

description. This macro takes a literal string as an argument. It places the string into the program address space and returns its address. The string can be accessed using the macros and functions in this section.

5.2.7. `strcat_P`

```
#include <pgmspace.h>

char* strcat_P(char* s1, PGM_P s2);
```

description. This function operates similarly to the `strcat()` function. Its second argument, however, refers to a string in program memory.

5.2.8. `strcmp_P`

```
#include <pgmspace.h>

int strcmp_P(char const* s1, PGM_P s2);
```

description. This function operates similarly to the `strcmp()` function. Its second argument, however, refers to a string in program memory. Make sure you don’t get the arguments reversed.

5.2.9. `strcpy_P`

```
#include <pgmspace.h>

char* strcpy_P(char* s1, PGM_P s2);
```

description. This function operates similarly to the `strcpy()` function. Its second argument, however, refers to a string in program memory.

5.2.10. `strcasecmp_P`

```
#include <pgmspace.h>

int strcasecmp_P(char const* s1, PGM_P s2);
```

description. This function operates similarly to the `strcasecmp()` function. Its second argument, however, refers to a string in program memory.

5.2.11. `strlen_P`

```
#include <pgmspace.h>

size_t strlen_P(PGM_P s);
```

description. This function operates similarly to the `strlen()` function. Its argument, however, refers to a string in program memory.

5.2.12. `strncasecmp_P`

```
#include <pgmspace.h>

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

description. This function operates similarly to the `strncasecmp()` function. Its second argument, however, refers to a string in program memory.

5.2.13. `strncmp_P`

```
#include <pgmspace.h>

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

description. This function operates similarly to the `strncmp()` function. Its second argument, however, refers to a string in program memory. Make sure you don’t get the arguments reversed.

5.2.14. `strncpy_P`

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

```
char* strncpy_P(char* s1, PGM_P s2, size_t n);
```

description. This function operates similarly to the `strncpy()` function. Its second argument, however, refers to a string in program memory.

5.3. EEPROM

All AVR processors contain a bank of nonvolatile memory. Unfortunately, this memory doesn't reside in the same address space as the static RAM; the architecture requires that the EEPROM cells be accessed through I/O registers. The EEPROM API provides a high-level interface to the hardware, which makes using the nonvolatile memory much easier. To gain access to these functions, include the file `eeprom.h`.

The routines take an argument representing the address of the cell. Rather than using hard-coded numbers or defined symbols, it would be nice to use actual variables. AVR-GCC allows this by using the `__attribute__` keyword. Example 5-1 shows a function that returns a checksum value from the EEPROM. The example allocates space in the .eeprom section to hold the variable, but doesn't specify the actual address. By taking this approach, the linker will properly fix-up the address references.

Example 5-1. Proper use of EEPROM variables

```
static uint8_t checksum __attribute__((section (".eeprom"))) = 0;

uint8_t getChecksum(void)
{
    return eeprom_rb(&checksum);
}
```

The amount of nonvolatile memory varies from device to device. The linker "knows" the limits of the sections, so by letting the compiler and linker reserve the space for variables, you can get diagnostic messages if you exceed the size of the bank. This can also come in handy if you need to switch device types in a project.

5.4. Function Reference

5.4.1. `eeprom_is_ready`

```
#include <eeprom.h>

int eeprom_is_ready(void);
```

description. This function indicates when the eeprom is able to be accessed. When an EEPROM location is written to, the entire EEPROM become unavailable for up to 4 milliseconds. Unlike some other microcontrollers, the AVR processors use hardware timers to program EEPROM cells. A status bit is provided to give an application the state of the EEPROM. This function allows an application to poll the status to find out when the memory is accessible.

5.4.2. `eeprom_rb`

```
#include <eeprom.h>

uint8_t eeprom_rb(uint16_t addr);
```

description. Reads a single byte from the EEPROM. The parameter *`addr`* specifies the location to read. The maximum address that can be specified depends upon the device. A macro has been defined to provide compatibility with the IAR compiler. Using the macro `_EEGET(addr)` will actually call this function.

5.4.3. `eeprom_read_block`

```
#include <eeprom.h>

void eeprom_read_block(void* buf, uint16_t addr, size_t n);
```

description. Reads a block of EEPROM memory. The starting address of the EEPROM block is specified in the *`addr`* parameter. The maximum address depends upon the device. The number of bytes to transfer is indicated by the *`n`* parameter. The data is transferred to an SRAM buffer, the starting address of which is passed in the *`buf`* argument.

5.4.4. `eeprom_rw`

```
#include <eeprom.h>

uint16_t eeprom_rw(uint16_t addr);
```

description. Reads a 16-bit value from the EEPROM. The data is assumed to be in little endian format. The parameter *`addr`* specifies the location to read. The maximum address that can be specified depends upon the device.

5.4.5. `eeprom_wb`

```
#include <eeprom.h>

void eeprom_wb(uint16_t addr, uint8_t val);
```

description. Writes a value, *`val`*, to the EEPROM. The value is written to address *`addr`*. To be compatible with the IAR compiler, a macro has been defined. `_EEPUT(addr, val)` will expand to a call to `eeprom_wb()`.

Chapter 6. Interrupt API

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 correspondi ng 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()`. The interrupt is chosen by supplying one of the symbols in Table 6-1. These macros register and mark the routine as an interrupt handler for the specified peripheral. See the entries for `INTERRUPT()` and `SIGNAL()` for examples of their use.

Unused interrupt vectors point to a routine called `_unexpected_`. The default version of this function simply consists of a reti instruction. You can define your own handler, if you want to handle unexpected interrupts differently.

Table 6-1. Signal names

Name	Description
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_COMPARE2	Output Compare2 Interrupt
SIG_OVERFLOW2	Overflow2 Interrupt
SIG_INPUT_CAPTURE1	Input Capture1 Interrupt
SIG_OUTPUT_COMPARE1A	Output Compare1(A) Interrupt
SIG_OUTPUT_COMPARE1B	Output Compare1(B) Interrupt
SIG_OVERFLOW1	Overflow1 Interrupt
SIG_OUTPUT_COMPARE0	Output Compare0 Interrupt
SIG_OVERFLOW0	Overflow0 Interrupt
SIG_SPI	SPI Interrupt
SIG_UART_RECV	UART(0) Receive Complete Interrupt
SIG_UART1_RECV	UART(1) Receive Complete Interrupt
SIG_UART_DATA	UART(0) Data Register Empty Interrupt

Name	Description
SIG_UART1_DATA	UART(1) Data Register Empty Interrupt
SIG_UART_TRANS	UART(0) Transmit Complete Interrupt
SIG_UART1_TRANS	UART(1) Transmit Complete Interrupt
SIG_ADC	ADC Conversion complete
SIG_EEPROM	Eeprom ready
SIG_COMPARATOR	Analog Comparator Interrupt

6.1. Function Reference

6.1.1. `cli`

```
#include <interrupt.h>

void cli(void);
```

description. 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.

6.1.2. `enable_external_int`

```
#include <interrupt.h>

void enable_external_int(uint8_t ints);
```

description. 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 `outp()` function, it does adapt slightly to the type of device being used.

6.1.3. `INTERRUPT`

```
#include <sig-avr.h>

INTERRUPT(signame);
```

description. This macro creates the prototype and opening of a function that is to be used as an interrupt. *`signame`* should be one of the symbols found in Table 6-1. The routine will be executed with interrupts enabled. If you want interrupts disabled, use the `SIGNAL()` macro instead. Example 6-1 sets up an empty routine which gets called when the ADC has completed a conversion.

see also. `SIGNAL()`

Example 6-1. Setting up an interrupt handler

```
/* This function will get attached to the SIG_ADC interrupt vector. */

INTERRUPT(SIG_ADC)
{
}
```

6.1.4. `sei`

```
#include <interrupt.h>

void sei(void);
```

description. 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.

6.1.5. `SIGNAL`

```
#include <sig-avr.h>

SIGNAL(signame);
```

description. This macro creates the prototype and opening of a function that is to be used as an interrupt. The argument *`signame`* should be one of the symbols found in Table 6-1. The routine will be executed with interrupts disabled. If you want interrupts enabled, use the `INTERRUPT()` macro instead.

Example 6-2 sets up an empty routine which gets called when the ADC has completed a conversion.

see also. `INTERRUPT()`

Example 6-2. Setting up a signal handler

```
/* This function will get attached to the SIG_ADC interrupt vector. */

SIGNAL(SIG_ADC)
```

```
    {
    }
```

6.1.6. `timer_enable_int`

```
#include <interrupt.h>

void timer_enable_int(uint8_t ints);
```

description. This function modifies the timsk register.

Chapter 7. I/O API

7.1. I/O Port APIs

This section describes the functions and macros that make it easier to access the I/O registers. Most of these routines actually get replaced with in-line assembly, so there is little to no performance penalty to use them. These routines are defined in `io.h`. This header file also defines the registers and bit definitions for the correct AVR device.

> **Note to self...**
>
> Include a few paragraphs that mention the various symbols that have been defined. Also, mention the bit definitions and how they are typically used.

7.2. Function Reference

7.2.1. `BV`

```
#include <io.h>

BV(pos);
```

description. This macro converts a bit definition into a bit mask. It is intended to be used with the bit definitions in the `io.h` header file. For instance, to build a mask of both the wdtoe and wde watchdog bits, you would use "BV(WDTOE) | BV(WDE)".

7.2.2. `bit_is_clear`

```
#include <io.h>

uint8_t bit_is_clear(uint8_t port, uint8_t bit);
```

description. Returns 1 if the specified *`bit`* in *`port`* is clear. *`bit`* can be 0 to 7. This function uses the sbic instruction to test the bit, so *`port`* needs to be a valid address for that instruction.

7.2.3. `bit_is_set`

```
#include <io.h>

uint8_t bit_is_set(uint8_t port, uint8_t bit);
```

description. Returns 1 if the specified *`bit`* in *`port`* is set. *`bit`* can be 0 to 7. This function uses the sbis instruction to test the bit, so *`port`* needs to be a valid address for that instruction.

7.2.4. `cbi`

```
#include <io.h>

void cbi(uint8_t port, uint8_t bit);
```

description. Clears the specified bit, *`bit`*, in the I/O register specified by *`port`*. *`bit`* is a value from 0 to 7 and should be specified as one of the defined symbols in the I/O header files. If *`port`* specifies an actual I/O register, this macro reduces to a single in-line assembly instruction. If it isn't an I/O register, it attempts to generate the most efficient code to complete the operation.

see also. `sbi()`

7.2.5. `inp`

```
#include <io.h>

uint8_t inp(uint8_t port);
```

description. Reads the 8-bit value from the I/O port specified by *`port`*. If *`port`* is a constant value, this macro assumes the value refers to a valid address and tries to use the in instruction. A variable argument results in an access using direct addressing.

7.2.6. `__inw`

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

uint16_t **__inw**(uint8_t port);

description. Reads a 16-bit value from I/O registers. This routine was created for accessing the 16-bit registers (ADC, ICR1, OCR1, TCNT1) because they need to be read in the proper order. This macro should only be used if interrupts are disabled since it only generates the two lines of assembly that reads the register.

7.2.7. `__inw_atomic`

#include <io.h>

uint16_t **__inw_atomic**(uint8_t port);

description. Atomically reads a 16-bit value from I/O registers. The generated code disables interrupts during the access and properly restores the interrupt state when through. This routine was created for accessing the 16-bit registers (ADC, ICR1, OCR1, TCNT1) because they need to be read in the proper order. This macro can safely be used in interrupt and non-interrupt routines because it preserves the interrupt enable flag (although you may not want to pay for the extra lines of assembly in an interrupt routine.)

7.2.8. `loop_until_bit_is_clear`

#include <io.h>

void **loop_until_bit_is_clear**(uint8_t port, uint8_t bit);

description. This macro generates a very tight polling loop that waits for a bit to become cleared. It uses the sbic instruction to perform the test, so the value of *`port`* is restricted to valid I/O register addresses for that instruction. *`bit`* is a value from 0 to 7.

7.2.9. `loop_until_bit_is_set`

#include <io.h>

void **loop_until_bit_is_set**(uint8_t port, uint8_t bit);

description. This macro generates a very tight polling loop that waits for a bit to become set. It uses the cbic instruction to perform the test, so the value of *`port`* is restricted to valid I/O register addresses for that instruction. *`bit`* is a value from 0 to 7.

7.2.10. `outp`

```
#include <io.h>

void outp(uint8_t val, uint8_t port);
```

description. Writes the 8-bit value *`val`* to *`port`*. If *`port`* is a constant value, this macro assumes the value refers to a valid address and tries to use the out instruction. A variable argument results in an access using direct addressing.

7.2.11. `__outw`

```
#include <io.h>

void __outw(uint16_t val, uint8_t port);
```

description. Writes to a 16-bit I/O register. This routine was created for manipulating the 16-bit registers (ADC, ICR1, OCR1, TCNT1) because they need to be written in the proper order. This macro should only be used if interrupts are disabled since it only generates the two lines of assembly that modify the register.

7.2.12. `__outw_atomic`

```
#include <io.h>

void __outw_atomic(uint16_t val, uint8_t port);
```

description. Atomically writes to a 16-bit I/O register. The generated code disables interrupts during the access and properly restores the interrupt state when through. This routine was created for accessing the 16-bit registers (ADC, ICR1, OCR1, TCNT1) because they need to be written in the proper order. This macro can safely be used in interrupt and non-interrupt routines because it preserves the interrupt enable flag (although you may not want to pay for the extra lines of assembly in an interrupt routine.)

7.2.13. `parity_even_bit`

```
#include <io.h>

void parity_even_bit(uint8_t val);
```

description. Returns 1 if *`val`* has even parity. All eight bits are used in the calculation.

7.2.14. `sbi`

```
#include <io.h>

void sbi(uint8_t port, uint8_t bit);
```

description. Sets the specified bit, *`bit`*, in the I/O register specified by *`port`*. *`bit`* is a value from 0 to 7 and should be specified as one of the defined symbols in the I/O header files. If *`port`* specifies an actual I/O register, this macro reduces to a single in-line assembly instruction. If it isn't an I/O register, it attempts to generate the most efficient code to complete the operation.

see also. `cbi()`

7.3. Watchdog API

The functions in this section manipulate the watchdog hardware. These macros are defined in `wdt.h`.

The startup code is able to initialize the watchdog hardware. By default, the control register, wdctr, is zeroed out. If you want it to be set to another value, you need to specify it on the linker command line. The symbol used is `__init_wdtcr__`. For instance, to set wdctr to 0x1f, you would have a command line like this:

```
% avr-ld --defsym __init_wdctr__=0x1f ...
```

7.4. Function Reference

7.4.1. `wdt_disable`

```
#include <wdt.h>
```

```
void wdt_disable(void);
```

description. Disables the watchdog timer. This function actually generates six inline assembly instructions.

7.4.2. `wdt_enable`

```
#include <wdt.h>

void wdt_enable(uint8_t timeout);
```

description. Enables the watchdog timer. The passed value, *`timeout`*, is loaded in the watchdog control register. This function generates five inline assembly instructions.

This function should probably never be used, since the startup code can already start up the watchdog timer (and it does it using less instructions.) The only reason it would be used is if `wdt_disable` is used.

7.4.3. `wdt_reset`

```
#include <wdt.h>

void wdt_reset(void);
```

description. Resets the watchdog timer. This function generates a single wdr instruction. Your application must guarantee that this function is called sooner than the timeout rate of the watchdog. Otherwise the processor will reset.

Appendix A. AVR-GCC Configuration

This appendix describes the AVR-specific changes to the GNU toolset. See the GNU documentation for options that are common to all processor targets.

A.1. `avr-as` Options

Table A-1. `avr-as` Options

Option	Description
`-mmcu=name`	Tells `avr-as` which AVR processor is the target. *`name`* can be `at90s1200`, `at90s2313`, `at90s2323`, `at90s2333`, `attiny22`, `at90s2343`, `at90s4433`, `at90s4414`, `at90s4434`, `at90s8515`, `at90s8535`, `atmega603`, `atmega103`, or `atmega161`.

A.2. `avr-gcc` Options

Table A-2. `avr-gcc` Options

Option	Description
`-mava`	Tells `avr-gcc` to use `ava` as the assembler and linker.
`-mcall-prologues`	Use subroutines for function prologue/epilogue.
`-minit-stack=symbol`	Sets the initial stack address.
`-mint8`	Assume int to be an eight bit integer.
`-mmcu=name`	Specify the device (*`at90s23xx`*, *`attiny22`*, *`at90s44xx`*, *`at90s85xx`*, *`atmega603`*, *`atmega103`*). The default is *`at90s85xx`*.
`-mno-interrupts`	Don’t disable interrupts when updating the upper eight bits of the stack pointer.
`-msize`	Outputs instruction sizes to the asm listing.