GNU Development Environment for the AVR Microcontroller

Rich Neswold
<rneswold@enteract.com>

June 23, 2000

Contents

1. Introduction

1.1. To-do

hings to add to this chapter:

- A *brief* introduction to the AVR processors.
- A *brief* introduction to the GNU compiler tools.
- The programmers that made this tool set possible.
- The fact that all the APIs mentioned in this document have been carefully written and generate tight assembly code.
- Mention the AVR mailing list and how to subscribe.

This document is based upon version 2.95.2 of the GNU tools. It also refers to version `20000514` of `avr-libc`.

2. Installing the GNU Tools

This chapter shows how to build and install a complete development environment for the AVR processors using the GNU toolset.[1]

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.

2.1. GNU Binutils

Before the compiler can be built, various utlities need to be installed. Since the compiler converts C only to assembly language, an assembler and linker (and librarian, etc.) need to be built and installed for the AVR processors. The GNU binutils can provide this support.

2.1.1. Downloading the Source

The binutils source archive, used in preparing this document, is version 2.9.5.0.13. You also need to apply AVR-specific patches.[2] The two files can be downloaded using the URLs in Table 2.1. Create a directory in which to build the tools and put the dpwnloaded files in it. You are now ready to build the utilities.

2.1.2. Building the Project

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

```
$ bunzip2 -c binutils-2.9.5.0.13.tar.bz2 | tar xf -
$ cd binutils-2.9.5.0.13
$ gunzip -dc ../binutils-2.9.5.0.13-avr-patch-1.1.gz | patch -p1
```

[1]These steps worked on my system, which is running FreeBSD 4.0. If there are problems with any of these instructions on your system, please let me know so I can resolve any problems.

[2]The AVR patches have been committed to the GNU project, so future releases will have AVR support built-in.

Tool	Location
GNU binutils	ftp://home.overta.ru/users/denisc/binutils-2.9.5.0.13.tar.bz2 http://medo.fov.uni-mb.si/mapp/uTools/avr-gcc/binutils/-binutils-2.9.5.0.13-avr-patch-1.1.gz
AVR-GCC	ftp://ftp.freesoftware.com/pub/gnu/gcc/gcc-core-2.95.2.tar.gz http://www1.itnet.pl/amelektr/avr/gcc/gcc-core-2.95.2-avr-patch-1.1.gz
AVR libc	http://www1.itnet.pl/amelektr/avr/libc/avr-libc-20000514.tar.gz http://www.enteract.com/~rneswold/avr/avr-libc-20000514-diff.gz
AVR Programmer	

Table 2.1.: “Tarball” Locations

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 and then making the project.[3]

```
$ configure --target=avr \
    --prefix=/usr/local/avr \
    --disable-nls
$ make
```

On my system, I disabled international support by using the `--disable-nls` option. I did this mainly because I was having problems linking the tools (the linker couldn’t find `libintl.a`). Whether this is an incompatibility problem with FreeBSD, or if it’s a bug in the makefile, I don’t know.

2.1.3. Installing the Tools

If the tools compiled cleanly, you’re ready 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 `/usr/local/avr/bin`. You’ll have to add that directory to your search path in order to run them. The tools have an `avr-` prefix so the assembler is `avr-as,` the linker is `avr-ld`, etc.

[3]BSD users should note that the project’s `makefile` uses GNU make syntax. This means FreeBSD users need to make the tools with `gmake`. On the other hand, once things stabilize, I plan on adding the AVR tools to the FreeBSD ports. So this chapter will be irrelevant to FreeBSD users...

2.1.4. An Alternative

The AVR-GCC compiler has support for using the AVA assembler/linker tools. If you would prefer to use this tool, the compiler can be configured to use it instead. I haven't built or used this tool, so this section will remain "unfinished".[4]

2.2. AVR-GCC

2.2.1. Downloading the Source

The gcc source archive, used in preparing this document, is version 2.95.2. You also need to apply AVR-specific patches.[5] The three files can be downloaded using the URLs in Table 2.1. Create a directory in which to build the tools and put the downloaded files in it. You are now ready to build the utilities.

2.2.2. 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.

[4]If someone wants to provide information on how to use this tool, I'd appreciate it.
[5]Again, the AVR patches have been committed to the GNU project, so future releases will have AVR support built-in.

2.2.3. Installing the Tools

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

```
$ make install
```

2.3. AVR-LIB

2.3.1. Downloading the Source

The AVR standard library archive used in this document is version 20000514. Unfortunately, it uses features of the preprocessor that are only available in later versions versions of the tools, so a series of patches need to be applied[6]. The archive and patches can be obtained using the URLs in Table 2.1. Download these two files and place them in your working directory.

2.3.2. Building the Libraries

Before we can build the libraries, we need to unarchive them and apply patches.

```
$ tar zxf avr-libc-20000514.tar.gz
$ cd avr-libc-20000514
$ gunzip -dc ../avr-libc-20000514-diff.gz | patch -p1
```

Now simply build the project.

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

2.3.3. 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
```

[6]As the project reaches a more stable release, I'll update these instructions. For now, these are the steps I take.

3. Using the GNU Tools

In this chapter, we create a simple example of using the GNU tools in an AVR project. This project will use the PWM to ramp an LED on and off every two seconds. An AT90S2313 processor will be used for the purposes of this example.

3.1. To-do

This is a list of things that need to be added to this chapter.

- Define a demo project.
  - Show the source code.
  - Describe the source.
  - Show the schematic.
- Build a Makefile.
  - Describe the Makefile.
- Build the project.
  - Show examples of generated files.
- Show how to download to programmer.

4. 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 A).

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. The reset vector 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 right out of a reset, name your routine `_init_`. Just make sure you jump to `_real_init_` at the end of your custom code. An example of how to do this is shown in Algorithm 1.

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

Algorithm 1 An example of adding boot-code.

```
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_" );
}
```

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.

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, and so some prolog and epilog code is placed in this function. When entering the function, the stack is initialized to point to the end of static RAM.[1] 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 4.1 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!

[1]I don't really like this approach because it doesn't provide much error checking. I would prefer to see something like:

```
static char stack[20] __attribute__((stack));
```

which would add 20 bytes to a stack section. This section would get combined with the data and bss sections. If the total size exceeded the static RAM, you'd know during the linking phase that you've run out of stack space. Maybe this could get incorporated in a newer version of the tools.

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

Figure 4.1.: Hex file for empty `main()`.

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.[1] These are shown in Table 5.1.

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

[1] I believe that a variable defined with `progmem` ought to have the `const` qualifier automatically added. The compiler currently doesn't do this. Time to submit a bug report...

Type Name	Definition
prog_void	void __attribute__((progmem))
prog_char	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*

Table 5.1.: Primitive types in program memory

Function Reference

__elpm_inline

syntax

```
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.

__lpm_inline

syntax

```
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()`

memcpy_P

syntax

```
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.

PRG_RDB

syntax

```
uint8_t PRG_RDB(uint16_t addr);
```

description

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

PSTR

syntax

```
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.

strcmp_P

syntax

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

description

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

strcpy_P

syntax

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

description

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

strlen_P

syntax

```
size_t strlen_P(PGM_P);
```

description

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

strncmp_P

syntax

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

description

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

strncpy_P

syntax

```
char* strncpy_P(char*, PGM_P, size_t);
```

description

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

5.2. 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. Algorithm 2 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.

Algorithm 2 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.

Function Reference

eeprom_is_ready

syntax

```
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.

eeprom_rb

syntax

```
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. Calling `_EEGET(addr)` will actually call this function.

eeprom_read_block

syntax

```
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.

eeprom_rw

syntax

```
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.

eeprom_wb

syntax

```
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()`.

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 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()`. 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.

The functions and macros are defined in `interrupt.h` and the signal symbols are defined in `sig-avr.h`.

Function Reference

cli

syntax

```
void cli(void);
```

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
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

Table 6.1.: Signal names.

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.

enable_external_int

syntax

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

INTERRUPT

syntax

```
INTERRUPT(signame)
```

description

This macro creates the prototype and opening of a function that is to be used as an interrupt (note that there is no semicolon!) The argument *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. Algorithm 3 sets up an empty routine which gets called when the ADC has completed a conversion.

Algorithm 3 Setting up an interrupt handler

```
INTERRUPT(SIG_ADC)
{
}
```

see also

`SIGNAL()`

sei

syntax

```
void sei(void);
```

description

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

SIGNAL

syntax

```
SIGNAL(signame)
```

description

This macro creates the prototype and opening of a function that is to be used as an interrupt (note that there is no semicolon!). 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. Algorithm 4 sets up an empty routine which gets called when the ADC has completed a conversion.

Algorithm 4 Setting up a signal handler

```
SIGNAL(SIG_ADC)
{
}
```

see also

`INTERRUPT()`

timer_enable_int

syntax

```
void timer_enable_int(uint8_t ints);
```

description

This function modifies the TIMSK register.

7. I/O APIs

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.

Function Reference

BV

syntax

```
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)`".

bit_is_clear

syntax

```
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.

bit_is_set

syntax

```
    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.

cbi

syntax

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

description

Clears the specified *bit* in *port*. *bit* is a value from 0 to 7 and should be specified as one of the defined symbols. 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()`

inp

syntax

```
    uint8_t inp(uint8_t port);
```

description

Reads the 8-bit value from *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.

__inw

syntax

```
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.

__inw_atomic

syntax

```
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.)

loop_until_bit_is_clear

syntax

```
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 port addresses for that instruction. *bit* is a value from 0 to 7.

loop_until_bit_is_set

syntax

```
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 `sbis` instruction to perform the test, so the value of *port* is restricted to valid port addresses for that instruction. *bit* is a value from 0 to 7.

outp

syntax

```
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.

__outw

syntax

```
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.

__outw_atomic

syntax

```
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.)

sbi

syntax

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

description

Sets the specified *bit* in *port*. *bit* is a value from 0 to 7 and should be specified as one of the defined symbols. 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()`

7.2. 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 ...
```

Function Reference

wdt_disable

syntax

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

timeout	Period
0	16K cycles
1	32K cycles
2	64K cycles
3	128K cycles
4	256K cycles
5	512K cycles
6	1024K cycles
7	2048K cycles

Table 7.1.: Watchdog timeout values.

description

This macro shuts down the watchdog hardware.

wdt_enable

syntax

```
void wdt_enable(uint8_t timeout);
```

description

This turns on the watchdog system. The parameter *timeout* indicates the expiration time of the watchdog. The valid settings for *timeout* are found in Table 7.1.

wdt_reset

syntax

```
void wdt_reset(void);
```

description

This macro generates in-line code to reset the watchdog timer.

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. Assembler Options

These added command line options are specific to the AVR processors.

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. Compiler Options

These added command line options are specific to the AVR processors.

Option	Description
`-mava`	Tells `avr-gcc` to use `ava` as the assembler and linker.
`-mcall-prologues`	Use subroutines for function prologue/epilogue.
`-minclude-target`	Add line "`#include "target.inc"`" to asm listing.
`-minit-stack=`	Sets initial stack address.
`-mint8`	Assume int to be an eight bit integer.
`-mmcu=`	Specify the device (`at90s23xx`, `attiny22`, `at90s44xx`, `at90s85xx`, `atmega603`, `atmega103`). The default is `at90s85xx`.
`-mno-interrupts`	Don't output interrupt compatible code.
`-msize`	Outputs instruction sizes to the asm listing.

A.3. Compiler-defined Symbols

The compiler defines symbols that the source code can use to adjust its compilation.

Symbol	Description
AVR, __AVR, __AVR__	Can be used to indicate source is being compiled for AVR processors
AVR_ATtiny22,	Defined when using `-mmcu=attiny22`.
AVR_AT90S8515	Defined when using `-mmcu=at90s85xx`.
AVR_ATmega603	Defined when using `-mmcu=at90mega603`.
AVR_ATmega103	Defined when using `-mmcu=at90mega103`.

A.4. Register Usage

Index